Intel® C++ Compiler Classic Developer Guide and Reference

Disclaimer and Legal Information

Contents

Notices and Disclaimers

Intel®C++ Compiler Classic Developer Guide and Reference

This guide contains information for version 2021.7 of the compiler.

Intel® C++ Compiler Classic (icc) is deprecated and will be removed in a oneAPI release in the second half of 2023. Intel recommends that customers transition now to using the LLVM-based Intel® oneAPI DPC++/C++ Compiler (icx) for continued Windows* and Linux* support, new language support, new language features, and optimizations. Note that starting with release 2021.7, macOS* support is limited to Mac* computers with Intel® Processors.

This document contains information about the Intel® C++ Compiler Classic (`icc` for Linux* and `icl` for Windows*) compiler and runtime environment. The Intel® C++ Compiler Classic can be found in the Intel® oneAPI HPC Toolkit, Intel® oneAPI IoT Toolkit, or as a standalone compiler. More information and specifications about Intel C/C++ compilers can be found on the Intel® oneAPI DPC++/C++ Compiler product page and in the Release Notes.

The following are some important sections of the compiler developer guide:

Compiler Setup	Compiler Setup explains how to invoke the compiler on the command line or from within an IDE.
OpenMP* Support	The compiler supports many OpenMP* features, including most of OpenMP* Version TR4: Version 5.0.
Compiler Options	Compiler Options provides information about options you can use to affect optimization, code generation, and more.
Intrinsics	Intrinsics let you generate more readable code, simplify instruction scheduling, reduce debugging, access the instructions that cannot be generated using the standard constructs of the C and C++ languages, and more.
Pragmas	Pragmas provide the compiler with the instructions for specific tasks, such as splitting large loops into smaller ones, enabling or disabling optimization for code, or offloading computation to the target.
Context Sensitive/F1 Help	To use the Context Sensitive/F1 Help feature, visit the Download Documentation: Intel® Compiler (Current and Previous) page and follow the instructions provided there.
Download Previous Versions of the Developer Guide and Reference	Visit the Download Documentation: Intel® Compiler (Current and Previous) page to download PDF or FAR HTML versions of previous compiler documentation.

NOTE For the best search experience, use a Google Chrome* or Internet Explorer* browser to view your downloaded copy of the Developer Guide and Reference.
If you use Mozilla Firefox*, you may encounter an issue where the **Search** tab does not work. As a workaround, you can use the **Contents** and **Index** tabs or a third-party search tool to find your content.

Part I

Introducing the Intel® C++ Compiler Classic

Using the Intel® C++ Compiler Classic, you can compile and generate applications that can run on Intel® 64 architecture. You can also create programs for the IA-32 architecture on Windows* and Linux*.

Intel® 64 architecture applications can run on the following:

- Windows operating systems for Intel® 64 architecture-based systems.
- Linux operating systems for Intel® 64 architecture-based systems.
- macOS operating systems for Intel® 64 architecture-based systems.

IA-32 architecture applications can run on the following:

- Supported Windows operating systems
- Supported Linux operating systems

Unless specified otherwise, assume the information in this document applies to all supported architectures and all operating systems.

You can use the compiler in the command-line or in a supported Integrated Development Environment (IDE):

- Microsoft Visual Studio* (Windows only)
- Eclipse*/CDT (Linux only)
- Xcode* (macOS only)

See the Release Notes for complete information on supported architectures, operating systems, and IDEs for this release.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Feature Requirements

To use these tools and features, you need licensed versions of the tools and you must have an appropriately supported version of the product edition. For more information, check the product release notes.

NOTE Some features may require additional product installation.

The following table shows components (tools) and where to find additional information on them.

Component	More Information
Intel® C++ Compiler Classic Intel® Advisor Intel® Inspector Intel® Trace Analyzer and Collector	More information on tools and features can be found on the Intel® Developer Zone and the Software Development Tools pages.

Component	More Information
Intel® VTune™ Profiler	

The following table lists dependent features and their corresponding required products. For certain compiler options, the compilation may fail if the option is specified but the required product is not installed. In this case, remove the option from the command line and recompile.

Feature Requirements

Feature	Requirement
Intel® oneAPI Threading Building Blocks (oneTBB)	The `-tbb`, `-qtbb`, and `/Qtbb` options require a oneTBB install.
Intel® oneAPI Math Kernel Library (oneMKL)	The `-qmkl`, `-mkl`, and `/Qmkl` options require a oneMKL install.
Intel® oneAPI Data Analytics Library (oneDAL)	The `-daal`, `-qdaal`, and `/Qdaal` options require a oneDAL install.
Intel® Integrated Performance Primitives (Intel® IPP)	The `-ipp`, `-qipp`, and `/Qipp` options require an Intel® IPP install.
Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography)	Use `crypto` to link to the Intel® IPP Cryptography library.
Thread Checking	Intel® Inspector
Trace Analyzing and Collecting	Intel® Trace Analyzer and Collector Compiler options related to this feature may require a set-up script. For further information, see the product documentation.

Refer to the Release Notes for detailed information about system requirements, late changes to the products, supported architectures, operating systems, and Integrated Development Environments (IDEs).

Get Help and Support

Windows*

Documentation is available from within the version of Microsoft Visual Studio*. You must install the documentation on your local system. To use the feature, visit the Download Documentation: Intel® Compiler (Current and Previous) page and follow the instructions provided there. From the **Help** menu, choose **Intel Compilers and Libraries** to view the installed user and reference documentation.

Linux* and macOS

On Linux and macOS, the documentation has limited integration in the Eclipse*/CDT and Xcode*. In both cases, the integrated documentation only provides details about where to find the product documentation on your local system.

Intel® Software Documentation

You can find product documentation for many released products at: https://software.intel.com/content/www/us/en/develop/documentation.html

Product Website and Support

To find product information, register your product, or contact Intel, visit: https://software.intel.com/content/www/us/en/develop/support.html

At this site, you will find comprehensive product information, including:

- Links to Get Started, Documentation, Individual Support, and Registration
- Links to information such as white papers, articles, and user forums
- Links to product information
- Links to news and events

Online Service Center

Each purchase of an Intel® Software Development Product includes a year of support services, which includes priority customer support at our Online Service Center. For more information about the Online Service Center visit: https://supporttickets.intel.com/servicecenter

> **NOTE** To access support, you must register your product at the Intel® Registration Center: https://registrationcenter.intel.com/en/products/

Release Notes

For detailed information on system requirements, late changes to the products, supported architectures, operating systems, and Integrated Development Environments (IDE) see the Release Notes for the product.

Forums

You can find helpful information in the Intel Software user forums. You can also submit questions to the forums. To see the list of the available forums, go to https://community.intel.com/t5/Software-Development-Tools/ct-p/software-dev-tools

Related Information

Recommended Additional Reading

You are strongly encouraged to read the following books for in-depth understanding of threading. Each book discusses general concepts of parallel programming by explaining a particular programming technology:

- For information on Intel® Threading Building Blocks (Intel® TBB): Reinders, James. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007
- For information on OpenMP* technology: Chapman, Barbara, Gabriele Jost, Ruud van der Pas, and David J. Kuck (foreword). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, October 2007
- For information on Microsoft Win32* Threading (for Windows* users): Akhter, Shameem, and Jason Roberts. *Multi-Core Programming: Increasing Performance through Software Multithreading*, Intel Press, April 2006

Intel does not endorse these books or recommend them over other books on the same subjects.

Additional Product Information

For additional technical product information including white papers, forums, and documentation, visit https://software.intel.com/content/www/us/en/develop/tools.html

Additional Language Information

- For information about the C++ standards, visit the C++ website: http://www.isocpp.org/
- For information about the C standards, visit the C website: http://www.open-std.org/jtc1/sc22/wg14/
- For information about the OpenMP* standards, visit the OpenMP website: http://www.openmp.org/

Notational Conventions

Information in this documentation applies to all supported operating systems and architectures unless otherwise specified. This documentation uses the following conventions:

Notational Conventions

THIS TYPE	Indicates language keywords.
this type	Indicates command-line or option arguments.
`This type`	Indicates a code example.
`This type`	Indicates what you type as input.
This type	Indicates menu names, menu items, button names, dialog window names, and other user-interface items.
File > Open	Menu names and menu items joined by a greater than (>) sign to indicate a sequence of actions. For example, **Click File > Open** indicates that in the **File** menu, you would click **Open** to perform this action.
{value | value}	Indicates a choice of items or values. You can usually only choose one of the values in the braces.
[*item*]	Indicates items that are optional.
item [, *item*]...	Indicates that the item preceding the ellipsis (...) can be repeated.
Intel® C++	This term refers to the name of the common compiler language supported by the Intel® C++ Compiler Classic.
compiler or the compiler	These terms are used when information is not limited to only one specific compiler, or when it is not necessary to indicate a specific compiler.
Windows or Windows operating system	These terms refer to all supported Microsoft Windows operating systems.
Linux or Linux operating system	These terms refer to all supported Linux operating systems.
macOS or macOS operating system	These terms refer to all supported macOS operating systems.

*	An asterisk at the end of a word or name indicates it is a third-party product trademark.
compiler option	This term refers to Linux, macOS, or Windows options, which are used by the compiler to compile applications.

Additional Conventions Used for Compiler Options

compiler option name shortcuts	The following conventions are used as shortcuts when referencing compiler option names in descriptions: • No initial – or / This shortcut is used for option names that are the same for Linux and Windows except for the initial character. For example, `Fa` denotes: • Linux and macOS: `-Fa` • Windows: `/Fa` • [Q]option-name This shortcut is used for option names that only differ because the Windows form starts with a Q. For example, [Q]ipo denotes: • Linux and macOS: `-ipo` • Windows: `/Qipo` • [q or Q]option-name This shortcut is used for option names that only differ because the Linux form starts with a q and the Windows form starts with a Q. For example, [q or Q]opt-report denotes: • Linux and macOS: `-qopt-report` • Windows: `/Qopt-report` More dissimilar compiler option names are shown in full.
`/option` or `-option`	A slash before an option name indicates the option is available on Windows. A dash before an option name indicates the option is available on Linux and macOS systems. For example: • Linux and macOS: `-help` • Windows: `/help`

	NOTE If an option is available on all supported operating systems, no slash or dash appears in the general description of the option. The slash and dash will only appear where the option syntax is described.
`/option:argument` or `-option=argument`	Indicates that an option requires an argument (parameter). For example, you must specify an argument for the following options: • Linux and macOS: `-mtune=`*processor* • Windows: `/tune:`*processor*
`/option:`*`keyword`* or `-option=`*`keyword`*	Indicates that an option requires one of the *`keyword`* values.
`/option`[:*`keyword`*] or `-option`[=*`keyword`*]	Indicates that the option can be used alone or with an optional keyword.
`option`[*n*] or `option`[:*n*] or `option`[=*n*]	Indicates that the option can be used alone or with an optional value. For example, in `-unroll`[=*n*], the *n* can be omitted or a valid value can be specified for *n*.
`option`[-]	Indicates that a trailing hyphen disables the option. For example, `/Qglobal_hoist-` disables the Windows option `/Qglobal_hoist`.
`[no]option` or `[no-]option`	Indicates that `no` or `no-` preceding an option disables the option. For example: In the Linux and macOS option `-[no-]global_hoist`, `-global_hoist` enables the option, while `-no-global_hoist` disables it. In the Windows option `/[no]traceback`, `/traceback` enables the option, while `/notraceback` disables it. In some options, the `no` appears later in the option name. For example, `-fno-common` disables the `-fcommon` option.

Part II

Compiler Setup

You can use the Intel® C++ Compiler Classic from the command line, Eclipse, Microsoft Visual Studio, or Xcode.

These IDEs are described in further detail in their corresponding sections.

See Also

Use the Command Line
Use Eclipse
Use Microsoft Visual Studio
Use Xcode

Use the Command Line

This section provides information about the Command Line Interface (CLI).

Specify the Location of Compiler Components

Before you invoke the compiler, you may need to set certain environment variables that define the location of compiler-related components. The compiler includes environment configuration scripts to configure your build and development environment variables:

- On Linux, and macOS, the file is a shell script called `setvars.sh`.
- On Windows, the file is a batch file called `setvars.bat`.

> **NOTE** If you are using older Intel® Parallel Studio XE or Intel® System Studio bits, you need to use `compilervars` instead of `setvars`.

Linux and macOS

Set the environment variables before using the compiler by sourcing the shell script `setvars.sh`. Depending on the shell, you can use the `source` command or a `.` (dot) to source the shell script, according to the following rules for a `.sh` script:

Using source:

```
source /<install-dir>/setvars.sh <arg1> <arg2> … <argn>
```

Example:

```
source /opt/intel/oneapi/setvars.sh intel64
```

Using . (dot):

```
. /<install-dir>/setvars.sh <arg1> <arg2> … <argn>
```

Example:

```
. /opt/intel/oneapi/setvars.sh intel64
```

Use `source /<install-dir>/setvars.sh --help` for more `setvars` usage information.

The compiler environment script file accepts an optional target architecture argument `<arg>`:

- `intel64`: Generate code and use libraries for Intel® 64 architecture-based targets.
- `ia32`: Generate code and use libraries for IA-32 architecture-based targets.

If you want the `setvars.sh` script to run automatically in all of your terminal sessions, add the source `setvars.sh` command to your startup file. For example, inside your `.bash_profile` entry for Intel® 64 architecture targets.

If the proper environment variables are not set, errors similar to the following may appear when attempting to execute a compiled program:

```
./a.out: error while loading shared libraries:
libimf.so: cannot open shared object file: No such file or directory
```

Windows

Under normal circumstances, you do not need to run the `setvars.bat` batch file. The terminal shortcuts in the Windows **Start** menu, **Intel oneAPI command prompt for <target architecture> for Visual Studio <year>**, set these variables automatically.

For additional information, see Use the Command Line on Windows.

You need to run the `setvars` batch file if a command line is opened without using one of the provided **Command Prompt** menu items in the **Start** menu, or if you want to use the compiler from a script of your own.

The `setvars` batch file inserts DLL directories used by the compiler and libraries at the beginning of the existing `Path`. Because these directories appear first, they are searched before any directories that were part of the original `Path` provided by Windows (and other applications). This is especially important if the original `Path` includes directories with files that have the same names as those added by the compiler and libraries.

The `setvars` batch file takes multiple optional arguments; the following two arguments are recognized for compiler and library initialization:

```
<install-dir>\setvars.bat [<arg1>] [<arg2>]
```

Where *<arg1>* is optional and can be one of the following:

- `intel64`: Generate code and use libraries for Intel® 64 architecture (host and target).
- `ia32`: Generate code and use libraries for IA-32 architecture (host and target).

The *<arg2>* is optional. If specified, it is one of the following:

- `vs2022`: Microsoft Visual Studio 2022
- `vs2019`: Microsoft Visual Studio 2019
- `vs2017`: Microsoft Visual Studio 2017.

NOTE Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release and will be removed in a future release.

If *<arg1>* is not specified, the script uses the `intel64` argument by default. If *<arg2>* is not specified, the script uses the highest installed version of Microsoft Visual Studio detected during the installation procedure.

See Also

oneAPI Development Environment Setup

Configure Your CPU or GPU System

Invoke the Compiler

Requirements Before Using the Command Line

You may need to set certain environment variables before using the command line. For more information, see Specify the Location of Compiler Components.

Different Compilers and Drivers

The table below provides the different compiler front-end and driver information.

Compiler	Notes	Linux Driver	Windows Driver
Intel® C++ Compiler Classic	A C++ compiler that supports an OpenMP but not OpenMP offload.	`icc` for C `icpc` for C++	`icl`

Use the Compiler from the Command Line

Use the compiler with the OS/language specific invocations below.

Linux

Invoke the compiler using `icc`/`icpc` to compile C/C++ source files.

- When you invoke the compiler with `icc` the compiler builds C source files using C libraries and C include files. If you use `icc` with a C++ source file, it is compiled as a C++ file. Use `icc` to link C object files.
- When you invoke the compiler with `icpc` the compiler builds C++ source files using C++ libraries and C++ include files. If you use `icpc` with a C source file, it is compiled as a C++ file. Use `icpc` to link C++ object files.

The `icc`/`icpc` command:

- Compiles and links the input source file(s).
- Produces one executable file, `a.out`, in the current directory.

macOS:

Invoke the compiler using `icc` or `icpc` to compile C/C++ source files.

- When you invoke the compiler with `icc`, the compiler builds C source files using C libraries and C include files. If you use `icc` with a C++ source file, it is compiled as a C++ file. Use `icc` to link C object files.
- When you invoke the compiler with `icpc` the compiler builds C++ source files using C++ libraries and C++ include files (libc++ library is used by default). If you use `icpc` with a C source file, it is compiled as a C++ file. Use `icpc` to link C++ object files.

The `icc`/`icpc` command:

- Compiles and links the input source file(s).
- Produces one executable file, `a.out`, in the current directory.

Windows

You can invoke the compiler on the command line using `icl`. This command:

- Compiles and links the input source file(s).
- Produces object file(s) and assigns the names of the respective source file(s), but with a `.obj` extension.
- Produces one executable file and assigns it the name of the first input file on the command line, but with a `.exe` extension.
- Places all the files in the current directory.

When compilation occurs with the compiler, many tools may be called to complete the task that may reproduce diagnostics unique to the given tool. For instance, the linker may return a message if it cannot resolve a global reference.

Command Line Syntax

The syntax to invoke the compiler is:

Linux and macOS

```
icc [option] file1 [file2...]
```

Windows

```
icl [option] file1 [file2...]
```

Argument	Description
`option`	Indicates one or more command line options. On Linux and macOS systems, the compiler recognizes one or more letters preceded by a hyphen (`-`). On Windows, options are preceded by a hyphen (`-`) or slash (`/`). This includes linker options. Options are not required when invoking the compiler. The default behavior of the compiler implies that some options are ON by default when invoking compiler.
`file1, file2...`	Indicates one or more files to be processed by the compiler.
`/link` (Windows)	All options following `/link` are passed to the linker. Compiler options must precede `link` if they are not to be passed to the linker.

Other Methods for Using the Command Line to Invoke the Compiler

- **Using makefiles from the Command Line:** Use makefiles to specify a number of files with various paths and to save this information for multiple compilations. For more information on using makefiles, see Use Makefiles to Compile Your Application.
- **Using a Batch File from the Command Line:** Create and use a `.bat` file to execute the compiler with a desired set of options instead of retyping the command each time you need to recompile.

See Also

Specify the Location of Compiler Components
Understand File Extensions
Use Eclipse
Use Microsoft Visual Studio
Use Xcode
Use Makefiles to Compile Your Application
watch compiler option

Use the Command Line on Windows

The compiler provides a shortcut to access the command line with the appropriate environment variables already set.

To invoke the compiler from the command line:

1. Open the Windows **Start** menu.
2. Scroll down the list of apps (programs) in the **Start** menu and find the **Intel oneAPI 2021** folder.
3. Left click on the folder name and select your component. The command prompts shown are dependent on the versions of Microsoft Visual Studio you have installed on your machine.
4. Right click on the command prompt icon to pin it to your taskbar. This step is optional.
5. The command line opens.

You can use any command recognized by the Windows command prompt, plus some additional commands.

Because the command line runs within the context of Windows, you can easily switch between the command line and other applications for Windows or have multiple instances of the command line open simultaneously.

When you are finished working in a command line, use the **exit** command to close and end the session.

File Extensions

Input File Extensions

The Intel® C++ Compiler Classic recognizes input files with the extensions listed in the following table:

File Name	Interpretation	Action
`file.c`	C source file	Passed to compiler

`file.C`
`file.CC`
`file.cc`
`file.cpp`
`file.cxx`
	C++ source file	Passed to compiler

`file.lib` (Windows) `file.a` `file.so` (Linux and macOS) `file.dylib` (macOS)	Library file	Passed to linker
`file.i`	Preprocessed file	Passed to compiler
`file.obj` (Windows) `file.o` (Linux and macOS)	Object file	Passed to linker
`file.asm` (Windows)	Assembly file	Passed to assembler

File Name	Interpretation	Action
`file.s` (Linux and macOS) `file.S` (Linux and macOS)		

Output File Extensions

The Intel® C++ Compiler Classic produces output files with the extensions listed in the following table:

File Name	Description
`file.i`	Preprocessed file: Produced with the `-E` option.
`file.o` (Linux and macOS) `file.obj` (Windows)	Object file: Produced with the `-c` (Linux, macOS, and Windows) object. The `/Fo` (Windows) or `-o` (Linux) option allows you to rename the output object file.
`file.s` (Linux and macOS) `file.asm` (Windows)	Assembly language file: Produced with the `-S` option. The `/Fa` (Windows) or `-s` (Linux) option allows you to rename the output assembly file.
`a.out` (Linux and macOS) `file.exe` (Windows)	Executable file: Produced by the default compilation. The `/Fe` (Windows) or `-o` (Linux) option allows you to rename the output executable file.

See Also

Invoke the Compiler
Specify Compiler Files

Use Makefiles for Compilation

This topic describes the use of makefiles to compile your application. You can use makefiles to specify a number of files with various paths, and to save this information for multiple compilations.

Use Makefiles to Store Information for Compilation on Linux or macOS

To run `make` from the command line using the compiler, make sure that `/usr/bin` and `/usr/local/bin` are in your *PATH* environment variable.

If you use the C shell, you can edit your `.cshrc` file and add the following:

```
setenv PATH /usr/bin:/usr/local/bin:$PATH
```

To use the compiler, your makefile must include the setting `CC=icc`. Use the same setting on the command line to instruct the makefile to use the compiler. If your makefile is written for GCC, you need to change the command line options that are not recognized by the compiler. Run `make`, using the following syntax:

```
make -f yourmakefile
```

Where `-f` is the `make` command option to specify a particular makefile name.

Use Makefiles to Store Information for Compilation on Windows

To use a makefile to compile your source files, use the `nmake` command with the following syntax:

```
nmake /f [makefile_name.mak] CPP=[compiler_name] [LINK32=[linker_name]
```

Example:

```
nmake /f your_project.mak CPP=icl LINK32=link
```

Argument	Description
`/f`	The `nmake` option to specify a makefile.
`your_project.mak`	The makefile used to generate object and executable files.
`CPP`	The preprocessor/compiler that generates object and executable files. (The name of this macro may be different for your makefile.)
`LINK32`	The linker that is used.

The `nmake` command creates object files (`.obj`) and executable files () from the information specified in the `your_project.mak` makefile.

See Also

Modify Your makefile (Linux and macOS)
Modify Your makefile (Windows)

Use Compiler Options

A compiler option is a case-sensitive, command line expression used to change the compiler's default operation. Compiler options are not required to compile your program, but they can control different aspects of your application, such as:

- Code generation
- Optimization
- Output file (type, name, location)
- Linking properties
- Size of the executable
- Speed of the executable

Linux and macOS

When you specify compiler options on the command line, the following syntax applies:

```
[invocation] [option] [@response_file] file1 [file2...]
```

The *invocation* is `icc`.

The *option* represents zero or more compiler options and the *file* is any of the following:

- C or C++ source file (`.C`, `.c`, `.cc`, `.cpp`, `.cxx`, `.c++`, `.i`, `.ii`)
- Assembly file (`.s`, `.S`)
- Object file (`.o`)
- Static library (`.a`)

When compiling C language sources, invoke the compiler with `icc`. When compiling C++ language sources or a combination of C and C++, invoke the compiler with `icpc`.

Windows

When you specify compiler options on the command line, the following syntax applies:

```
[invocation] [option] [@response_file] file1 [file2 ...] [/link linker_option]
```

The *invocation* is `icl`.

The *option* represents zero or more compiler options, the *linker_option* represents zero or more linker options, and the *file* is any of the following:

- C or C++ source file (`.c`, `.cc`, `.ccp`, `.cxx`, `.i`)
- Assembly file (`.asm`)
- Object (`.obj`)
- Static library (`.lib`)

The optional *response_file* is a text file that lists the compiler options you want to include during compilation. See Use Response Files for additional information.

Default Operation

The compiler invokes many options by default. In this example, the compiler includes the option `O2` (and other default options) in the compilation. Using C++ as an example:

Linux and macOS

```
icpx main.c
```

Windows

```
icx main.c
```

Each time you invoke the compiler, options listed in the corresponding configuration file override any competing default options. For example, if your configuration file includes the `O3` option, the compiler uses `O3` rather than the default `O2` option. Use the configuration file to list the options for the compiler to use for every compilation. See Using Configuration Files.

Options specified in the command line environment variable override any competing default options and options listed in the configuration file.

Finally, options used on the command line override any competing options that may be specified elsewhere (default options, options in the configuration file, and options specified in the command line environment variable). If you specify the option `O1` this option setting takes precedence over competing option defaults and competing options in the configuration files, in addition to the competing options in the command line environment variable.

Certain `#pragma` statements in your source code can override competing options specified on the command line. If a function in your code is preceded by `#pragma optimize("", off)`, then optimization for that function is turned off. The override is valid even when the `O2` optimization is on by default, the `O3` is listed in the configuration file, and the `O1` is specified on the command line for the rest of the program.

Use Competing Options

The compiler reads command line options from left to right. If your compilation includes competing options, then the compiler uses the one furthest to the right. Using C++ as an example:

Linux and macOS

```
icpc -xSSSE3 main.c file1.c -xSSE4.2 file2.c
```

Windows

```
icl /QxSSSE3 main.c file1.c /QxSSE4.2 file2.c
```

The compiler sees `[Q]xSSSE3` and `[Q]xSSE4.2` as two forms of the same option, where only one form can be used. Since `[Q]xSSE4.2` is last (furthest to the right), it will be used.

All options specified on the command line are used to compile each file. The compiler does not compile individual files with specific options. For example:

Linux and macOS

```
icc -O3 main.c file1.c -mp1 file2.c
```

Windows

```
icl /O3 main.c file1.c /Qprec file2.c
```

It may seem that `main.c` and `file1.c` are compiled with the option `O3`, and `file2.c` is compiled with the `-mp1` (Linux and macOS) or `/Qprec` (Windows) option. This is not correct; all files are compiled with both options.

A rare exception to this rule is the `-x` *type* option on Linux and macOS. Using C++ as an example:

Linux and macOS

```
icpc -x c file1 -x c++ file2 -x assembler file3
```

The *type* argument identifies each file type for the compiler.

Use Options with Arguments

Compiler options can be as simple as a single letter, such as the option `E`. Many options accept or require arguments. The `O` option, for example, accepts a single-value argument that the compiler uses to determine the degree of optimization. Other options require at least one argument and can accept multiple arguments. For most options that accept arguments, the compiler warns you if your option and argument are not recognized. If you specify `O9`, the compiler issues a warning, then ignores the unrecognized option `O9`, and proceeds with the compilation.

The `O` option does not require an argument, but there are other options that must include an argument. The `I` option requires an argument that identifies the directory to add to the include file search path. If you use this option without an argument, the compiler will not finish its compilation.

Other Forms of Options

You can toggle some options on or off by using the negation convention. For example, the `[Q]ipo` option (and many others) includes negation forms, `-no-ipo` (Linux and macOS) and `/Qipo-` (Windows), to change the state of the option.

Option Categories

When you invoke the Intel® C++ Compiler Classic and specify a compiler option, you have a wide range of choices to influence the compiler's default operation. Intel® C++ Compiler Classic options typically correspond to one or more of the following categories:

- Advanced Optimization
- Code Generation
- Compatibility
- Compiler Diagnostics
- Component Control
- Data
- Floating Point
- Inlining
- Interprocedural Optimizations (IPO)
- Language

- Linking/Linker
- Miscellaneous
- OpenMP and Parallel Processing
- Optimization
- Optimization Report
- Output
- Preprocessor

To see the included options in each category, invoke the compiler from the command line with the `help` category option. For example:

Linux and macOS

```
icc -help codegen
```

Windows

```
icl /help codegen
```

The help option prints to `stdout` with the names and syntax of the options found in the Code Generation category.

See Also

qopt-report, Qopt-report
Use Configuration Files

Specify Compiler Files

Specify Include Files

The compiler searches the default system areas for include files and items specified by the `I` compiler option. The compiler searches directories for include files in the following order:

1. Directories specified by the `I` option.
2. Directories specified in the environment variables.
3. Default include directories.

Use the `-nostdinc` (Linux) or `X` (Windows) option to remove the default directories from the include file search path.

For example, to direct the compiler to search the path `/alt/include` instead of the default path, use the following:

Linux and macOS

```
icpc -nostdinc -I/alt/include prog1.cpp
```

Windows

```
icl /X /I\alt\include prog1.cpp
```

Specify Assembly Files

You can use the `-S` and `-o` options (Linux and macOS) or `/Fa` option (Windows) to specify an alternate name for an assembly file. The compiler generates an assembly file named `myasm.s` (Linux and macOS) or `myasm.asm` (Windows):

Linux and macOS

```
icpc -S -o myasm.s x.cpp
```

Windows

```
icl /Famyasm x.cpp
```

Specify Object Files

You can use the `-c` and `-o` options (Linux and macOS) or `/Fo` option (Windows) to specify an alternate name for an object file. In this example, the compiler generates an object file name `myobj.o` (Linux and macOS) or `myobj.obj` (Windows):

Linux and macOS

```
icpc -c -o myobj.o x.cpp
```

Windows

```
icl /Fomyobj x.cpp
```

See Also

`-c` compiler option
`/Fa` compiler option
`/Fo` compiler option
I compiler option
`-o` compiler option
`-S` compiler option
X compiler option
Supported Environment Variables

Convert Projects to Use a Selected Compiler

You can use the command-line interface `ICProjConvert<version>.exe` to transform your Intel® C++ projects into Microsoft Visual C++* projects, or vice versa. The syntax is:

```
ICProjConvert<version>.exe <sln_file | prj_files> </VC[:"VCtoolset name"] | /IC[:"ICtoolset
name"]> [/q] [/nologo] [/msvc] [/s] [/f]
```

Where:

Parameter	Description
`version`	The `ICProjConvert` version number. Values are: 191 or 192.
`sln_file`	A path to the solution file, which should be modified to use a specified project system.
`prj_files`	A space separated list of project files (or wildcard), which should be modified to use specified project system.
`/VC`	Convert to use the Microsoft Visual C++ project system.
VCtoolset name	The possible values are `v141` (Microsoft Visual Studio* 2017), `v142` (Microsoft Visual Studio 2019), or `v143` (Microsoft Visual Studio 2022). **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.
`/IC`	Convert to use the Intel® C++ project system.

Parameter	Description
ICtoolset name	Such as `Intel C++ Compiler 2021.1` Depending on the integration version, the supported name values may be different.
`/q`	Starts quiet mode, all information messages (except errors) are hidden.
`/nologo`	Suppresses the startup banner.
`/msvc`	Sets the compiler to Microsoft Visual C++.
`/s`	Searches the project files through all subdirectories.
`/f`	Forces an update to the project even if it has an unsupported type or unsupported properties.
`/?` or `/h`	Shows help.

Example

Issue the command `ICProjConvert<version>.exe` *.icproj `/s /VC` to convert all Intel® C++ project files in the current directory and its subdirectories to use Microsoft Visual C++.

> **NOTE** If you uninstall the Intel® C++ Compiler Classic, `ICProjConvert<version>.exe` remains in the folder `Program Files (x86)\Common Files\Intel\shared files\ia32\Bin` and you can use it to transform Intel® C++ projects back into Microsoft Visual C++.

*Use Eclipse**

The Intel® C++ Compiler for Linux* provides integrations for the compiler to Eclipse* and C/C++ Development Tooling* (CDT) that let you develop, build, and debug your Intel® C++ projects in an integrated development environment (IDE).

Eclipse is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. It is an extensible, open source integrated development environment (IDE). CDT is a complete C/C++ IDE for the Eclipse platform, which allows you to develop, build, and run projects in a visual, interactive environment. CDT is layered on Eclipse and provides a C/C++ development environment perspective.

Add the Compiler to Eclipse*

This step is needed only if you are manually installing the Intel® C++ Compiler plug-in for Eclipse*.

To add the Intel® C++ Compiler product extension to your Eclipse configuration:

1. Start Eclipse.
2. Select **Help** > **Install New Software**.
3. Next to the **Work with** field, click the **Add** button. The **Add Repository** dialog box opens.
4. Click the **Archive** button and browse to the `<install_dir>/compiler/<version>/linux/ide_support` directory. Select the `.zip` file that starts with `com.intel.compiler`, then click **OK**.
5. Select **Intel® Software Development Tools** > **Intel® C++ Compiler Integration**, then click **OK**.
6. Follow the installation instructions.
7. When asked if you want to restart Eclipse, select **Yes**.

When Eclipse restarts, you can create and work with CDT projects that use the Intel® C++ Compiler.

Multi-Version Compiler Support

You can select different versions of the Intel® C++ Compiler for compiling projects with the Eclipse* Integrated Development Environment (IDE). For a list of the currently supported compiler versions by platform, refer to the Release Notes.

If multiple versions of the compiler are installed on the system, Eclipse uses the latest version by default. To select the version of the compiler to build your project:

1. Right click the project and open **Properties**.
2. In the properties dialog box, select **C/C++ Build** > **Settings**.
3. Select the **Intel® C++ Compiler Classic** for a C++ project tab.
4. Select the row with the desired compiler version.
5. Click **Use Selected**. Alternatively, click **Use Latest** to select the latest version of compiler.
6. Click **Apply**.

The corresponding compiler environment is configured automatically for your project.

Use **Settings** and **Tool Chain Editor** to select tools to be used within the toolchain, or set distinct project properties, like compiler options, to be used with different versions of the compiler.

For any project, you can set the compiler environment by specifying it within Eclipse; this overrides any other environment specifications for the compiler.

Use Cheat Sheets

The Intel® C++ Compiler Classic integration includes several Eclipse* cheat sheets that can guide you through various compilation and debugging tasks.

To view a list of available cheat sheets and select one:

1. Select **Help** > **Cheat Sheets**.
   The **Cheat Sheet Selection** dialog box opens, displaying a list of available cheat sheets.
2. Select a cheat sheet. Cheat sheets located outside of the Eclipse* integration can be entered in the **Select a cheat sheet from a file** or **Enter the URL of a cheat sheet**.
   Intel cheat sheets are located under **Intel(R) C++ Compiler**. A description of the cheat sheet appears in the lower pane.
3. To open a cheat sheet, click **OK**.

The **Cheat Sheets** view opens in the Eclipse window.

Create a Simple Eclipse Project

The sections below show you how to create a simple project using Eclipse.

Create a New Eclipse Project

To create an Eclipse project:

1. Select **File** > **New** > **Project...** The **New Project** wizard opens.
2. Expand the **C/C++ Project** tab and select the appropriate project type. Click **Next** to continue.
3. For **Project name**, enter hello_world. Deselect the **Use default location** to specify a directory for the new project.
4. In the **Project Type** list, expand the **Executable** project type and select **Hello World C++ Project** for C++.
5. In the **Toolchains** list, select **Intel C++ Compiler Classic**. Click **Next**.

NOTE

- If you need to see the toolchains for the compilers that are not locally installed, uncheck **Show project types and toolchains only if they are supported on the platform**. You are only able to view and configure these toolchains if the proper compilers are installed.
- If you have multiple versions of the compiler installed, they appear in the project's properties under **C/C++ Build** > **Settings** on the **Intel® C++ Compiler Classic** tab.

6. The **Basic Settings** page allows specifying template information, including **Author** and **Copyright notice**, which appear as a comment at the top of the generated source file. After entering desired fields, click **Next**.
7. The **Select Configurations** page allows specifying deployment platforms and configurations. By default, a **Debug** and **Release** configuration is created for the selected toolchain. Select no (**Deselect all**), multiple, or all (**Select all**) configurations. To edit project properties, click the **Advanced settings** button. Click **Finish** to create the `hello_world` project. Configurations can be created after the project is created by selecting **Project** > **Properties**.
8. If the view is not the **C/C++ Development Perspective** (default), an **Open Associated Perspective** dialog box opens. In the **C/C++ Perspective**, click **Yes** to proceed.

An entry for your `hello_world` project appears in the **Project Explorer** view.

Add a C Source File

To add a source file to the `hello_world` project:

1. Select the `hello_world` project in the **Project Explorer** view.
2. Select **File** > **New** > **Source File**. The **New Source File** dialog box opens. The dialog box automatically populates the source folder for the source file to be created. You can change this by entering a new location or selecting **Browse**.
3. Enter `new_source_file.c` in the **Source File** field.
4. Select a **Template** from the drop-down list or **Configure** a new template.
5. Click **Finish** to add the file to the `hello_world` project.
6. In the **Editor** view, add your code for `new_source_file.c`.
7. When your code is complete, **Save** your file.

Set Options for a Project or File

You can specify compiler, linker, and archiver options at the project and source file level. Follow these steps to set options for a project or file:

1. Right-click a project or source file in the **Project Explorer**.
2. Select **Properties**. The property pages dialog box opens.
3. Select **C/C++ Build** > **Settings**.
4. Select the **Tool Settings** tab and click an option category for **Intel C Compiler**, **Intel C++ Compiler**, or **Intel C++ Linker** for a C++ project.
5. Set the options to apply to the project or file.

NOTE

- Some properties use check boxes, drop-down boxes, or dialog boxes to specify compiler options. For a description on options properties, hover over the option to display a tooltip. After setting the desired options in command line syntax, select **Apply**.
- To specify an option that is not available from the **Properties** dialog, use **C/C++ Build Settings** > **Settings** > **<Compiler>** > **Command Line**. Enter the command line options in the **Additional Options** field using command-line syntax and select **Apply**.
- You can specify option settings for one or more configurations by using the **Configuration** drop-down menu.

6. Click **Apply and Close**.

The compiler applies the selected options, using the selected configurations, when building. To restore default settings to all properties for the selected configuration, click the **Restore Defaults** button on any property page.

Exclude Source Files from a Build

To exclude a source file from a build:

1. Right-click a file or folder in the **Project Explorer**.
2. Select **Resource Configurations** > **Exclude from build**. The **Exclude from build** dialog box opens.
3. Select one or more build configurations to exclude the file or folder from.
4. Click **OK**.

The compiler excludes that file or folder when it builds using the selected project configuration.

Build a Project

To build your project:

1. Select the `hello_world` project in the **Project Explorer** view.
2. Select **Project** > **Build Project**.

See the **Build** results in the **Console** view.

For a C/C++ project, use:

```
**** Build of configuration Debug for project hello_world ****
make all
Building file: ../src/hello_world.cpp
Invoking: Intel(R) C++ Compiler Classic
icpc -g -O0 -MMD -MP -MF"src/hello_world.d" -MT"src/hello_world.d" -c -o "src/hello_world.o"
"../src/hello_world.cpp"
Finished building: ../src/hello_world.cpp

Building target: hello_world
Invoking: Intel C++ Linker
icpc -O0 -o "hello_world"  ./src/hello_world.o
Finished building target: hello_world

Build Finished. 0 errors, 0 warnings.
```

Detailed descriptions of errors, warnings, and other output can be viewed by selecting the **Problems** tab.

Run a Project

After building a project, you can run your project by following these steps:

1. Select the `hello_world` project in the **Project Explorer** view.

2. Select **Run As** > **Local C/C++ Application**.

When the executable runs, the output appears in the **Console** view.

Error Parser

The Error Parser (selected by default) lets you track compile-time errors in Eclipse. To confirm that the Error Parser is active:

1. Select the `hello_world` project in the **Project Explorer** view.
2. Select **Project** > **Properties**.
3. In the **Properties** dialog box, select **C/C++ Build** > **Settings**.
4. Click the **Error Parsers** tab. Make sure that **Intel C++ Error Parser** is checked, and **CDT Visual C Error Parser** or **Microsoft Visual C Error Parser** are not checked.
5. Click **OK** to update your choices, if you have changed any settings.

Use the Error Parser

The Error Parser automatically detects and manages the diagnostics generated by the Intel® C++ Compiler Classic.

If an error occurring in the `hello_world.c` program is compiled, for example, `#include <xstdio.h>`, the error is reported in the **Problems** view next to an error marker.

You can double-click each error in the **Problems** view to highlight the source line, which causes the error in the **Editor** view.

Correct the error, then rebuild your project.

Makefiles

This section provides information about makefile project types and exporting makefiles.

Project Types and Makefiles

When you create a new project in Eclipse*, there are **Executable**, **Shared Library**, **Static Library**, or **Makefile** project types available for your selection.

- Select **Makefile Project** if the project already includes a makefile.
- Use **Executable**, **Shared Library**, or **Static Library Project** to build a makefile using options assigned from property pages specific to the Intel® C++ Compiler Classic.

Export Makefiles

Eclipse can build a makefile that includes Intel® C++ Compiler Classic options for created **Executables**, **Shared Libraries**, or **Static Library** Projects. When the project is complete, export the makefile and project source files to another directory, and then build the project from the command line using `make`.

To export the makefile:

1. Select the project in the Eclipse **Project Explorer** view.
2. Select **File** > **Export** to launch the **Export Wizard**. The **Export** dialog box opens, showing the **Select** screen.
3. Select **General** > **File system**, then click **Next**. The **File System** screen opens.
4. Check both the **hello_world** and **Release** directories in the left-hand pane. Ensure all project sources are checked in the right-hand pane.

NOTE Some files in the right-hand pane may be deselected, such as the `hello_world.o` object file and the `hello_world` executable. **Create directory structure for files** in the **Options** section must be selected to successfully create the export directory. This process applies to project files in the `hello_world` directory.

5. Use the **Browse** button to target the export to an existing directory. Eclipse can create a directory for full paths entered in the **To directory** text box. For example, if the `/code/makefile` is specified as the export directory, Eclipse creates two new subdirectories:
   - `/code/makefile/hello_world`
   - `/code/makefile/hello_world/Release`
6. Click **Finish** to complete the export.

Run Make

In a terminal window, change to the `/cpp/hello_world/Release` directory, then run `make` by typing: `make clean all`.

You should see the following output:

```
rm -rf  ./new_source_file.o  ./new_source_file.d  hello_world

Building file: ../new_source_file.c
Invoking: Intel(R) C Compiler Classic
icc -O2 -MMD -MP -MF"new_source_file.d" -MT"new_source_file.d" -c -o "new_source_file.o" "../
new_source_file.c"
Finished building: ../new_source_file.c

Building target: hello_world
Invoking: Intel C++ Linker
icc -o "hello_world"  ./new_source_file.o
Finished building target: hello_world
```

This process generates the `hello_world` executable in the same directory.

Use Intel Libraries with Eclipse*

You can use the compiler with the following Intel Libraries, which that may be included as a part of the product:

- Intel® oneAPI Data Analytics Library (oneDAL)
- Intel® Integrated Performance Primitives (Intel® IPP)
- Intel® oneAPI Math Kernel Library (oneMKL)
- Intel® oneAPI Threading Building Blocks (oneTBB)

To access these libraries in Eclipse*, use the property pages:

1. Select your project.
2. Open the property pages from **Project** > **Properties** and select **C/C++ Build** > **Settings**.
3. Select **Intel C/C++ Compiler** > **Performance Library Build Components**

The **Use Intel® oneAPI Data Analytics Library** (oneDAL) property allows enabling the library and bringing in the associated headers.

- **None**: Disable Use of oneDAL.
- **Use threaded Intel® oneDAL**: Link using the threaded version of the library.
- **Use non-threaded Intel® oneDAL**: Link using the non-threaded version of the library.

The **Use Intel® Integrated Performance Primitives Libraries** property provides the following options in a drop-down menu:

- **None**: Disable use of Intel® IPP.
- **Use main libraries set**: Link using the main libraries set.

- **Use non-pic version of libraries**: Link using the version of the libraries that do not have position-independent code.
- **Use main libraries and cryptography library**: Link using main or cryptography libraries.

The **Use Intel® oneAPI Math Kernel Library** property provides the following options in a drop-down menu:

- **None**: Disables the use of the oneMKL.
- **Use threaded Intel® oneMKL library**: Link using the threaded version of the library.
- **Use non-threaded Intel® oneMKL library**: Link using the non-threaded version of the library.
- **Use Intel® oneMKL Cluster and sequential Intel® oneMKL libraries**: Link using the oneMKL Cluster Edition libraries and the sequential oneMKL libraries.

> **NOTE** The option value **Use Intel® oneMKL Cluster and sequential Intel® oneMKL libraries** is only available for Intel C Compiler or Intel C++ Compiler.

The **Use Intel® oneAPI Threading Building Blocks** on the **Property** page allows enabling the library and bringing in the associated headers.

For more information, see the oneDAL, Intel® IPP, oneMKL, and oneTBB documentation.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

*Using Microsoft Visual Studio**

You can use the Intel® C++ Compiler within the Microsoft Visual Studio* integrated development environment (IDE) to develop C++ applications, including static library (`.LIB`), dynamic link library (`.DLL`), and main executable (`.EXE`) applications. This environment makes it easy to create, debug, and execute programs. You can build your source code into several types of programs and libraries, using the IDE or from the command line.

The IDE offers these major advantages:

- Makes application development quicker and easier by providing a visual development environment.
- Provides integration with the native Microsoft Visual Studio debugger.
- Makes other IDE tools available.

Create a New Project

Create a New Project

When you create a project, Microsoft Visual Studio automatically creates a corresponding solution to contain it. To create a new Intel® C++ project using Microsoft Visual Studio:

> **NOTE** Exact steps may vary depending on the version of Microsoft Visual Studio in use.

1. Select **File** > **New** > **Project**.
2. In the left pane, expand **Visual C++** and select **Windows Desktop**.
3. In the right pane, select **Windows Console Application**.
4. Accept or specify a project name in the **Name** field. For this example, use hello32 as the project name.
5. Accept or specify the Location for the project directory. Click **OK**.

The `hello32` project assumes focus in the **Solution Explorer** view. The default Microsoft Visual Studio solution is also named `hello32`.

Use the Intel® C++ Compiler Classic

To use the compiler with Microsoft Visual C++* (MSVC):

1. Create a MSVC project, or open an existing project.
2. In **Solution Explorer**, select the project(s) to build with Intel® C++ Compiler Classic.
3. Open **Project** > **Properties**.
4. In the left pane, expand the **Configuration Properties** category and select the **General** property page.
5. In the right pane, change the Platform Toolset to **<compiler selection>**. Alternatively, you can change the toolset by selecting **Project** > **Intel Compiler** > **Use Intel C++ Compiler**. This sets whichever version of the compiler that you specify as the toolset for all supported platforms and configurations.

   NOTE For C/C++, there are two toolsets: Select **Intel C++ Compiler <major version>** (example 2021) to invoke `icx`, or select **Intel C++ Compiler <major.minor>** (example 19.2) to invoke `icl`.

6. To add options, go to **Project** > **Properties** > **C/C++** > **Command Line** and add new options to the **Additional Options** field. Alternatively, you can select options from Intel specific properties. Refer to complete list of options in the Compiler Options section in this documentation.
7. Rebuild, using either **Build** > **Project only** > **Rebuild** for a single project, or **Build** > **Rebuild Solution** for a solution.

Switch Back to the MSVC Compiler

If your project is using the Intel® C++ Compiler Classic, you can switch back to MSVC:

1. Select your project.
2. Right-click and select **Intel Compiler** > **Use Visual C++** from the context menu.

Verify Use of the Intel® C++ Compiler Classic

To verify the use of the Intel® C++ Compiler Classic:

1. Go to **Project** > **Properties** > **C/C++** > **General**.
2. Set **Suppress Startup Banner** to **No**. Click **OK**.
3. Rebuild your application.
4. Look at the **Output** window.

You should see a message similar to the following when using the Intel® C++ Compiler Classic:

```
Intel(R) C++ Intel(R) XX Compiler Classic for applications running on XXXX, Version XX.X.X
```

Unsupported MSVC Project Types

The following project types are not supported:

- Class Library
- CLR Console Application
- CLR Empty Project
- Windows* Forms Application
- Windows Forms Control Library

Tips for Use

- Create a separate configuration for building with Intel® C++ Compiler Classic:
  - After you have created your project and specified it as an Intel project, create a new configuration (for example, `rel_intelc` based on **Release** configuration or `debug_intelc` based on **Debug** configuration).
  - Add any special optimization options offered by Intel® C++ Compiler Classic only to this new configuration (for example, `rel_intelc` or `debug_intelc`) through the project property page.
- Build with Intel® C++ Compiler Classic.

Select the Compiler Version

If you have multiple versions of the installed, you can select which version you want from the **Compiler Selection** dialog box:

1. Select a project, then go to **Tools** > **Options** > **Intel Compilers and Libraries** > **Compilers**.
2. Use the **Selected Compiler** drop-down menu to select the appropriate version of the compiler.
3. Click **OK**.

See Also

Use Intel® C++ dialog box

Specify a Base Platform Toolset

By default, when your project uses the Intel® C++ Compiler Classic, the Base Platform Toolset property is set to use that compiler with the build environment. This environment includes paths, libraries, included files, etc., of the toolset specific to the version of Microsoft Visual Studio* you are using.

You can set the general project level property **Base Platform Toolset** to use one of the non-Intel installed platform toolsets as the base.

For example, if you are using Microsoft Visual Studio 2019, and you selected the Intel® C++ Compiler Classic in the Platform Toolset property, then the Base Platform Toolset uses the Microsoft Visual Studio 2019 environment (**v142**). If you want to use other environments from Microsoft Visual Studio along with the Intel® C++ Compiler Classic, set the **Base Platform Toolset** property to:

- **v141** for Microsoft Visual Studio 2017
- **v142** for Microsoft Visual Studio 2019
- **v143** for Microsoft Visual Studio 2022

> **NOTE**
> Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

This property displays all installed toolsets, not including Intel toolsets.

To set the Base Platform Toolset:

- Using property pages:
  1. Select the project and open **Project** > **Properties**.
  2. In the left pane, select **Configuration Properties** > **General**.
  3. In the right pane, find **Intel Specific** > **Base Platform Toolset**.
  4. Select a value from the pop-up menu.
- Using the `msbuild.exe` command line tool, use the `/p:PlatformToolset` and `/p:BasePlatformToolset` options.

Example: When the Platform Toolset property is already set to use the Intel® C++ Compiler Classic, to build a project using the Microsoft Visual Studio 2019 environment use the following command:

```
Msbuild.exe myproject.vcxproj /p:BasePlatformToolset=v142
```

Example: To set the Platform Toolset property to use the Intel® C++ Compiler Classic and build a project using the Microsoft Visual Studio 2019 environment use the following command:

```
Msbuild.exe myproject.vcxproj /p:PlatformToolset="Intel C++ Compiler 19.2" /
p:BasePlatformToolset=v141
```

For possible values for the `/p:BasePlatformToolset` property, see the properties described above.

The next time you build your project with the Intel® C++ Compiler Classic, the option you selected is used to specify the build environment.

Use Property Pages

The Intel® C++ Compiler Classic includes support for Property Pages to manage both Intel-specific and general compiler options.

To set compiler options in Microsoft Visual Studio*:

1. Right-click on a project or source file in the **Solution Explorer** view.
2. Select **Properties** from the pop-up menu.
3. In the **Property Pages** dialog box, expand the **C/C++** section to view the categories of compiler options.
4. Click **OK** to complete your selection.

The option you selected is used in the compilation the next time you build your project.

Use Intel® Libraries with Microsoft Visual Studio*

You can use the compiler with the following Intel® Libraries, which may be included as a part of the product:

- Intel® oneAPI Data Analytics Library (oneDAL)

  NOTE This library is only available in the x64 configuration.

- Intel® Integrated Performance Primitives (Intel® IPP)
- Intel® oneAPI Threading Building Blocks (oneTBB)
- Intel® oneAPI Math Kernel Library (oneMKL)

Use the property pages to specify Intel Libraries to use with the selected project configuration. The functionality supports Intel® C++ and Microsoft Visual C++* project types.

To specify Intel Libraries, select **Project** > **Properties**. In **Configuration Properties**, select **Intel Libraries for oneAPI**, then do the following:

1. To use **oneDAL** change the **Use oneDAL** settings as follows:

   NOTE This library is only available in the x64 configuration.

   - **No**: Disable Use of oneDAL.
   - **Default Linking Method**: Use parallel dynamic oneDAL libraries.
   - **Multi-threaded Static Library**: Use parallel static oneDAL libraries.
   - **Single-threaded Static Library**: Use sequential static oneDAL libraries.
   - **Multi-threaded DLL**: Use parallel dynamic oneDAL libraries.
   - **Single-threaded DLL**: Use sequential dynamic oneDAL libraries.
2. To use **Intel® Integrated Performance Primitives**, change the **Use Intel® IPP** settings as follows:

- **No**: Disable use of Intel® IPP libraries.
- **Default Linking Method**: Use dynamic Intel® IPP libraries.
- **Static Library**: Use static Intel® IPP libraries.
- **Dynamic Library**: Use dynamic Intel® IPP libraries.

3. To use **oneTBB** in your project, change the **Use oneTBB** settings as follows:

   - **No**: Disable use of oneTBB libraries.
   - **Use oneTBB**: Set to **Yes** to use oneTBB in the application.
   - **Instrument for use with Analysis Tools**: Set to **Yes** to analyze your release mode application (not required for debug mode).

4. To use **oneMKL** in your project, change the **Use oneMKL** property settings as follows:

   - **No**: Disable use of oneMKL libraries.
   - **Parallel**: Use parallel oneMKL libraries.
   - **Sequential**: Use sequential oneMKL libraries.
   - **Cluster**: Use cluster libraries.

Additional settings for use with the Microsoft Visual C++* Platform Toolset are available on the **Intel Libraries for oneAPI** category, found at **Tools** > **Options**.

NOTE The **Use <library>** properties in Microsoft Visual Studio mimic the behavior of the `/Qmkl`, `/Qdaal`, `/Qipp` and `/Qtbb` compiler options. The include and library paths to the performance library, which are installed with the selected compiler, are set up with these properties.

Change the Selected Intel® Performance Libraries

If you have installed multiple versions of the Intel® Performance Libraries, you can specify which version to use with the Microsoft Visual C++* compiler. To do this:

1. Select **Tools** > **Options**.
2. In the left pane, select **Intel Compilers and Libraries** > **Performance Libraries**.
3. Select the desired library version from the drop-down boxes in the right pane.

For more information, see the Intel® oneAPI Data Analytics Library (oneDAL), Intel® Integrated Performance Primitives (Intel® IPP), Intel® oneAPI Threading Building Blocks (oneTBB), and Intel® oneAPI Math Kernel Library (oneMKL) documentation.

Include MPI Support

To specify the type of MPI support you want:

1. Open the project's property pages and select **Configuration Properties** > **Intel Libraries for oneAPI**.
2. Set the property **Use oneMKL** to **Cluster**.
3. Set the property **Use MPI Library** to one of the following values:

   - **Intel® MPI Library**
   - **MS-MPI**

4. Build the project.

The next time you build your project with the Microsoft Visual C++* compiler, it will include support for the version of MPI that you specified.

Use Guided Auto Parallelism in Microsoft Visual Studio*

The Guided Auto Parallelism (GAP) feature helps you locate portions of your serial code that can be parallelized. When you enable analysis using GAP, the compiler guides you to places in your code where you can increase efficiency through automatic parallelization and vectorization.

NOTE GAP is not supported for `ifx`.

Run Analysis on a Project

You can start analysis from the Microsoft Visual Studio* IDE in several ways:

- From the **Tools** menu: Select **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis...**

  Starting analysis in this way results in a one-time run for the current project. The default values are taken from **Tools** > **Options** unless you have chosen to override them in the dialog box.
- From the Diagnostics property page: Use the **Guided Auto Parallelism Analysis** property.

  Specifically, choose **Project** > **Properties** > **C/C++** > **Diagnostics** and enable analysis using the **Guided Auto Parallelism Analysis** property. Enabling analysis in the property page allows you to run an analysis as part of a normal project **Build** request in Microsoft Visual Studio*. In this mode, GAP-related settings in **Tools** > **Options** are ignored, in favor of other GAP-related settings available in the property page.
- From the context menu: Right-click and select **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis...**.

  This is equivalent to using the **Guided Auto Parallelism** > **Run Analysis** option on the **Tools** menu.

To receive advice for auto parallelization, be sure that certain property page settings are correct. Select **Project** > **Properties** > **C/C++** > **Optimization [Intel C++]** and set **Parallelization** to **Yes** to enable auto-parallelization optimization. You may also need to set the **Optimization** level at option `O2` or higher. To do this, use the **Optimization** property page.

GAP Scenarios

To illustrate how the various GAP settings work together, consider the following scenarios:

Scenario	Result
The GAP analysis setting in the property pages is set to Enabled.	Analysis always occurs for the project, whenever a regular project build occurs. Other analysis settings specified in the property pages are used. Analysis setting in **Tools** > **Options** are ignored.
The Gap analysis setting in the property pages is set to Disabled, and GAP is run from the **Tools** menu.	Analysis occurs for this one run. The default values for this analysis are taken from **Tools** > **Options** and can be overridden in the dialog box. Options specified in the property pages are also used, but will be overridden by any specified analysis option.
The GAP analysis setting in the property pages is set to Disabled, and GAP options are set in **Tools** > **Options**.	No analysis occurs, unless analysis is explicitly run from the **Tools** menu.

Run Analysis on a File or within a File

Right-click on **Guided Auto Parallelism** context menu item to run analysis on the following:

- **Single file:** Select a file and right-click.
- **Function (routine):**Right-click within the function scope.
- **Range of lines:** Select one or more lines for analysis and right-click.

See Also

Options: Guided Auto Parallelism dialog box

Guided Auto Parallelism

Using Guided Auto Parallelism

Use Code Coverage in Microsoft Visual Studio*

The code coverage tool provides the ability to determine how much application code is executed when a specific workload is applied to the application. The tool analyzes static profile information generated by the compiler, as well as dynamic profile information generated by running an instrumented form of the application binaries on the workload. The tool can generate a report in HTML-format and export data in both text- and XML-formatted files. The reports can be further customized to show color-coded, annotated source-code listings that distinguish between used and unused code.

> **NOTE** Code coverage is not supported for `ifx`.

To start code coverage:

1. Select **Tools** > **Intel Compiler** > **Code Coverage...**
2. Specify settings for the various phases.
3. Click **Run**.

The **Output** window shows the results of the coverage and a general summary of information from the code coverage.

See Also

Code Coverage dialog box

Code Coverage Settings dialog box

Code Coverage Tool

Use Profile Guided Optimization in Microsoft Visual Studio*

Profile Guided Optimization (PGO) improves application performance by reorganizing code layouts to reduce instruction-cache problems, shrinking code size, and reducing branch misprediction. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

> **NOTE** PGO is not supported for `ifx`.

To start PGO:

1. Choose **Tools** > **Intel Compiler** > **Profile Guided Optimization...**
2. Specify settings for the various phases.
3. Click **Run**.

The **Output** windows show the results of the optimization with a link to the composite log.

See Also

Profile Guided Optimization dialog box

Options: Profile Guided Optimization dialog box

Profile Guided Optimization

Optimization Reports

Enable in Microsoft Visual Studio*

Optimization reports can help you address vectorization and optimization issues.

When you build a solution or project, the compiler generates optimization diagnostics. You can view the optimization reports in the following windows:

- The **Compiler Optimization Report** window, either grouped by loops or in a flat format.
- The **Compiler Inline Report** window.
- The optimization annotations, which are integrated within the source editor.

To enable viewing for the optimization reports:

1. In your project's property pages, select **Configuration Properties** > **C/C++** > **Diagnostics [Intel C ++]**.
2. Set a non-default value for any of the following options:
   - **Optimization Diagnostics Level**
   - **Optimization Diagnostics Phase**
   - **Optimization Diagnostics Routine**
3. Build your project to generate an optimization report.

When the compiler generates optimization diagnostics, the **Compiler Optimization Report** and the **Compiler Inline Report** windows open. The optimization report annotations appear in the source editor.

> **NOTE** You can specify how you want the optimization reporting to appear with the **Optimization Reports** dialog box. Access this dialog box by selecting **Tools** > **Options** > **Intel Compilers and Libraries** > **Optimization Reports**.

View Reports

When the compiler generates optimization diagnostics, the **Compiler Optimization Report** and the **Compiler Inline Report** windows open, and optimization report annotations appear in the editor.

The **Compiler Optimization Report** window displays diagnostics for the following phases of the optimization report:

- PGO
- LNO
- PAR
- VEC

- Offload (Linux* only)
- OpenMP*
- CG

Information appears in this window grouped by loops, or in a flat format. To switch the presentation format, click the gear button on the toolbar of the window, and uncheck **Group by loops**.

In addition to sorting information by clicking column headers and resizing columns, you can use the windows described in the following table:

Do This	To Do This
Double-click a diagnostic.	Jump to the corresponding position in the editor.
Click a link in the **Inlined into** column.	Jump to the call site of the function where the loop is inlined.
Expand or collapse a diagnostic in **Group by loops** view.	View detailed information for the diagnostic.
Click on a column header.	Sort the information according to that column.
Click the filter button.	Select a scope by which to filter the diagnostics that appear in the window. The title bar of the **Compiler Optimization Report** window shows the applied filter. Labels on optimization phase filter buttons show how many diagnostics of each phase are in the current scope.
Click a **Compiler Optimization Report** window toolbar button corresponding to an optimization report phase.	Turn filtering diagnostics on or off for an optimization phase. Labels on optimization phase filter buttons show the total number of diagnostics for each phase. By default all phases turned on.
Enter text in the search box in the **Compiler Optimization Report** window toolbar.	Filter diagnostics using the text pattern. Diagnostics are filtered when you stop typing. Pressing Enter saves this pattern in the search history. To disable filtering, clear the search box. To use a pattern from the search history, click on the down arrow next to the search box.

The **Compiler Inline Report** window displays diagnostics for the IPO phase of the optimization report.

Information appears in this window in a tree. Each entry in the tree has corresponding information in the right-hand pane under the **Properties** tab and the **Inlining options** tab.

You can use the window as described in the following table:

Do This	To Do This
Double-click a diagnostic in the tree, or click on the source position link under the **Properties** tab.	Jump to the corresponding position in the editor.
Click **Just My Code**.	Only display functions from your code, filter all records from files that don't belong to the current solution file tree.
Right-click on a function body in the editor and select **Intel Compiler** > **Show Inline report for *function name***.	View detailed information for the specified function.

Do This	To Do This
Right-click on a function body in the editor and select **Intel Compiler** > **Show where *function name* in inlined**.	Show where the specified function is inlined.
Enter text in the search box in the **Compiler Inline Report** window toolbar.	Filter diagnostics using the text pattern. Diagnostics are filtered when you stop typing. Pressing Enter saves this pattern in the search history. To disable filtering, clear the search box. To use a pattern from the search history, click on the down arrow next to the search box.

The **Viewing Optimization Notes** window in the editor provides context for the diagnostics that the compiler generates:

- In Caller Site
- In Callee Site
- In Caller and Callee Site

You can use optimization notes as described in the following table:

Do This	To Do This

Right-click an optimization note	• Expand or collapse the current optimization note, or all of them.
	• Open the **Optimization Reports** dialog box to adjust settings for optimization report viewing. You can view optimization notes in one of the following locations:
	  • **Caller Site**
	  • **Callee Site**
	  • **Caller Site and Callee Site**

Double-click an optimization note heading.	Expand or collapse the current optimization note.
Double-click a diagnostic detail.	Jump to the corresponding position in the editor.
Click a hyperlink in the optimization note.	Show where the specified function is inlined.
Click the help **(?)** icon.	Get detailed help for the selected diagnostic. The default browser opens and, if you are connected to the internet, displays the help topic for this diagnostic.
Hover the mouse over a collapsed optimization note.	View a detailed tool tip about that optimization note.

See Also

Options: Optimization Reports dialog box
qopt-report-phase, Qopt-report-phase

Dialog Box Help

This section provides information about access to dialog boxes and information about compilers, libraries, and converter dialog boxes.

Options: Compilers dialog box

To access the **Compilers** page:

1. Open **Tools** > **Options**.
2. In the left pane, select **Intel Compilers and Libraries** > **C++** > **Compilers**.

Compiler Selection for C++ Classic

Target Platform: Select your target platform.

Platform Toolset/Selected Compiler: Select your compiler for your platform toolset. The left column lists the platform toolset names. The right column lists combo boxes, which are used to select a compiler. The default value for all combo boxes in current table is **<Latest>**.

> **NOTE** The left column contains Intel® C++ Compiler Classic and Intel® oneAPI DPC++/C++ Compiler options. The **Intel C++ Compiler <major.minor>** (example 19.2) selects the Intel® C++ Compiler Classic (`icc`). The **Intel C++ Compiler <major>** (example 2021) selects the Intel® oneAPI DPC++/C++ Compiler (`icx`).

Default Options: Sets the default options for a selected compiler. You may specify this setting for each selected compiler.

Environment: Sets custom environment variables. You may specify this setting for each selected compiler.

> **NOTE** The Environment selection is only available for `icx`.

Compiler Information: Shows the detail description of the selected compiler.

Reset All: Sets all contents back to the default value on the dialog.

See Also

Use Intel® C++ Compiler Classic dialog box

To access the **Use Intel C++ Compiler Classic** dialog box, select one or more files in the Solution Explorer, right-click and select **Use Intel C++ Compiler Classic for selected files(s)...**

Use this dialog box to change the compiler for one or more selected files to the Intel® C++ Compiler Classic.

Select the configuration(s) to update: Select the desired configuration. Choose from **Active configuration** or **All configurations**. If you select **All configurations**, the compiler is changed in all configurations for the currently selected file(s).

Select the Platform Toolset: Select the desired toolset, if multiple platform toolsets are installed.

See Also

Use the Intel® C++ Compiler Classic

Options: Intel Libraries for oneAPI dialog box

Use the **Intel Libraries for oneAPI** dialog box to specify standalone library versions to use with the Microsoft Visual C++* compiler.

To access the **Intel Libraries for oneAPI** dialog box:

1. Open **Tools** > **Options**.
2. Select **Intel Compilers and Libraries** > **Intel Libraries for oneAPI**.

In the dialog box, you can select your desired library version from the drop-down box with one of the following values:

- **oneDAL**
- **Intel IPP**
- **oneTBB**
- **oneMKL**
- **Reset All**: Use the latest libraries (default)

> **NOTE** To enable or disable the Intel Libraries for oneAPI, use the property pages located in the **Configuration Properties** category.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

Use Intel® Libraries for oneAPI

Options: Converter dialog box

To access the **Converter** page, click **Tools** > **Options** and then select **Intel Compilers and Libraries** > **C++** > **Converter**.

Use the **Converter** page to specify which platform toolset to use when upgrading an Intel® C++ solution (`.icproj`) from an older version of Microsoft Visual Studio* to a C++ project supported by Microsoft Visual Studio 2017 or later (`.vcxproj`). Once a solution is upgraded, the `.icproj` file is not used and can be deleted.

> **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

Win32: Select the desired compiler version to be used when converting projects based on IA-32 architecture.

X64: Select the desired compiler version to be used when converting projects based on x64 architecture.

Reset All: Click this button to use the default platform toolsets.

Options: Optimization Reports dialog box

To access the **Optimization Reports** page, click **Tools** > **Options** and then select **Intel Compilers and Libraries** > **Optimization Reports**. Use this page to specify how you want optimization reporting to appear.

This page, in conjunction with the **Diagnostics** property page for your project or solution, defines settings for optimization report viewing in Visual Studio*.

General

Always Show Compiler Inline Report: Specify if the Compiler Inline Report appears after building or rebuilding your solution or project when inline diagnostics are present.

Always Show Compiler Optimization Report: Specify if Compiler Optimization Report appears after building or rebuilding your solution or project when optimization diagnostics are present. This option has higher priority than **Always Show Compiler Inline Report**. If both options are set to True, then this window has focus by default.

Show Optimization Notes in Text Editor Margin: Specify if optimization notes appear in the editor as source code annotations.

Optimization Notes in Text Editor

Collapse by Default: Specify if optimization notes appear expanded or collapsed by default.

Show Optimization Notes: Specify if source code annotations appear in the editor.

Site: Specify where optimization notes appear in the editor. Select from one of the following options:

- **Caller Site**
- **Callee Site**
- **Caller and Callee Sites**

See Also

Optimization Reports: Enabling in Visual Studio*

Options: Guided Auto Parallelism dialog box

Use the **Guided Auto Parallelism** page to specify settings for Guided Auto Parallelism (GAP) analysis.

> **NOTE** The **Guided Auto Parallelism** dialog box is only available for `ifort`.

To access the **Guided Auto Parallelism** page click **Tools** > **Options** and then select: **Intel Compilers and Libraries** > **C++** > **Guided Auto Parallelism**.

These settings are used when running analysis using **Tools** > **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis on project...**

Guided Auto Parallelism Options

Level of Analysis: Specify the desired level of analysis. Choose **Simple**, **Moderate**, **Maximum**, or **Extreme**.

Suppress compiler warnings: Check this box to suppress compiler warnings. Selection adds option `W0` to the compiler command line.

Suppress Remark IDs: Specify one or more remark IDs to suppress. Use a comma to separate IDs.

Send remarks to a file: Check this box to send GAP remarks to a specified text file.

Remarks file: Specify the filename to send GAP remarks to.

Show all GAP configuration and informational dialogs: Check this box to display additional dialog boxes when you run an analysis.

Reset All: Click this button to restore the previously selected settings.

See Also

Using Guided Auto Parallelism in Microsoft Visual Studio*

Guided Auto Parallelism

Using Guided Auto Parallelism

Configure Analysis dialog box

Use the **Configure Analysis** dialog box to specify settings for Guided Auto Parallelism (GAP) analysis and run the analysis.

To access this dialog box, click **Tools** > **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis...**

Configure Analysis Options

Level of Analysis: Specify the desired level of analysis. Choose **Simple**, **Moderate**, **Maximum**, or **Extreme**.

Suppress Compiler Warnings: Check this box to suppress compiler warnings. This adds the option `W0` to the compiler command line.

Suppress remark IDs: Specify one or more remark IDs to suppress. Use a comma to separate IDs.

Send remarks to a file: Check this box to send GAP remarks to a specified text file.

Remarks file: Specify the filename where GAP remarks will be sent.

Save these settings as the default (in Tools > Options for Guided Auto Parallelism): Check this box to save the specified settings as the default settings.

Show this dialog next time (in Tools > Intel Compiler): Check this box to display this dialog box next time.

When you are done specifying settings, click **Run Analysis**.

See Also

Auto-Parallelization

Using Guided Auto Parallelism

Using Guided Auto Parallelism in Microsoft Visual Studio*

Options: Guided Auto Parallelism dialog box

Options: Profile Guided Optimization (PGO) dialog box

Use the **Profile Guided Optimization** page to specify settings for PGO. To access the **Profile Guided Optimization** page, click **Tools** > **Options** and then select **Intel Compilers and Libraries** > **Profile Guided Optimization**.

Profile Guided Optimization (PGO) Options

Show PGO Dialog: Specify whether to display the Profile Guided Optimization dialog box when you begin PGO.

See Also

Using Profile Guided Optimization in Microsoft* Visual Studio*

Profile Guided Optimization dialog box

Profile-Guided Optimizations

Profile Guided Optimization dialog box

This topic has information on the following dialog boxes:

- **Profile Guided Optimization (PGO)** dialog box
- **Application Invocations** dialog box
- **Edit Command** dialog box
- **Command** dialog box

Profile Guided Optimization dialog box

To access the **Profile Guided Optimization** dialog box, choose **Tools** > **Intel Compiler** > **Profile Guided Optimization**.

Use the **Profile Guided Optimization** dialog box to set the options for profile guided optimization.

Phase 1 - Instrument: This phase produces an instrumented object file for the profile guided optimization. The command line compiler option for each optimization instrument you choose appears in **Compiler Options**.

- **Enable Function Ordering in the optimized application:** Select this checkbox to enable ordering of static and extern routines using profile information. This optimization specifies the order in which the linker should link the functions of your application. This optimization can improve your application performance by improving code locality and by reducing paging.
- **Enable Static Data Layout in the optimized application:** Select this checkbox to enable ordering of static global data items based on profiling information. This optimization specifies the order in which the linker should link global data of your program. This optimization can improve application performance by improving the locality of static global data, reduce paging of large data sets, and improve data cache use.
- **Instrument with guards for threaded application:** Select this checkbox to produce an instrumented object file that includes the collection of PGO data on applications that use a high level of parallelism.

Selecting an option produces a static profile information file (`.spi`), but also increases the time needed to do a parallel build.

Deselect the checkbox to skip this phase to save time running profile guided optimization. When you skip this phase, you use the existing profile information when running profile guided optimization. For example, you may want to skip this phase when you change the code to fix a bug and the fix doesn't affect the architecture of the project.

Phase 2 - Run Instrumented Application(s): This phase runs the instrumented application produced in the previous phase as well as other applications in the **Applications Invocations** dialog box. To add a new application or edit an existing application in the list, click **Applications Invocations**.

Deselect the checkbox to skip this phase to save time running profile guided optimization. When you skip this phase, you do not run the applications in the list when running profile guided optimization. For example, you might want to skip this phase when you change the code to fix a bug and the fix doesn't affect the architecture of the project.

Phase 3 - Optimize with Profile Data: This phase performs the profile guided optimization.

Deselect the checkbox to skip this phase.

Profile Directory: The directory that contains the profile. Click **Edit** to edit the profile directory or the **Browse** button to browse for the profile directory.

Show this dialog next time: Deselect this checkbox to run profile guided optimization without displaying this dialog box. The profile guided optimization will use these settings.

Save Settings: Click to save your settings.

Run: Click to start the profile guided optimization.

Cancel: Click to close this dialog box without starting the profile guided optimization.

Application Invocations dialog box

To access the **Application Invocations** dialog box, click **Application Invocations...** in the **Profile Guided Optimization** dialog box. Use the **Profile Guided Optimization** dialog box to configure the application options for your application as well add additional applications when you run profile guided optimization.

The list of applications comes from the debug settings of the **Startup Project**.

Merge Environment: Select this checkbox to merge the application environment with the environment defined by the operating system.

To add, edit, or remove an application, click one of the buttons.

Add: Click to add a new application in the **Add Command** dialog box.

Duplicate: Click after selecting an application to copy its settings so that you can use a different setting.

Edit: Click after selecting an application to change its settings in the **Edit Command** dialog box.

Delete: Click to remove the selected application from the list.

OK: Click to save the settings and close this dialog box.

Cancel: Click to discard the settings and close this dialog box.

Add Command dialog box

To access the **Add Command** dialog box, click **Add** in the **Application Invocations** dialog box. Use the **Add Command** dialog box to add a new application in the **Application Invocations** dialog box.

Command: Add a new or edit an existing application. Click **Edit** to open the **Command** dialog box with a list of macros. Click **Browse** to navigate to another directory that contains the application.

Command Arguments: Enter the arguments required by the application.

Working Directory: Enter a new or edit the working directory for the application. Click **Edit** to open the **Working Directory** dialog box with a list of macros. Click **Browse** to navigate to working directory of the application.

Environment: Enter the environment variable required by this application.

Merge Environment: Select this checkbox to merge the application environment with the environment defined by the operating system.

Load from Debugging Settings: Click to load the debug settings for this application.

OK: Click to save the settings and close this dialog box.

Cancel: Click to discard the settings and close this dialog box.

> **NOTE** The **Edit Command** and **Add Command** dialog boxes are similar. To use the **Edit Command** dialog box, substitute **Edit** for **Add** in the selections above.

Command dialog box

To access the **Command** dialog box, click **Edit** in the **Edit Command** dialog box, or **Add** in the **Add Command** dialog box. Use the **Command** dialog box to specify or change the macro used in the application to run as part of the profile guided optimization.

Select a macro from the list and then click one of the buttons.

Macro: Click to show or close the list of available macros.

Insert: Click to use the selected macro.

OK: Click to save the settings and close this dialog box.

Cancel: Click to discard the settings and close this dialog box.

See Also

Profile-Guided Optimization

Using Profile Guided Optimization in Microsoft Visual Studio*

Options: Profile Guided Optimization

Using Function Order Lists, Function Grouping, Function Ordering, and Data Ordering Optimizations

Options: Code Coverage dialog box

To access the **Code Coverage** page, click **Tools** > **Options** and then select **Intel Compilers and Libraries** > **Code Coverage**.

Use this page to specify settings for code coverage. These settings are used when you run an analysis.

Codecov Options

Use the available options to:

- Select colors to be used to show covered and uncovered code.
- Enable or disable the progress meter.
- Set the email address and name of the web page owner.

General

Show Code Coverage Dialog: Specify whether to display the Code Coverage dialog box when you begin code coverage.

Profmerge Options

Suppress Startup Banner: Specify whether version information is displayed.

Verbose: Specify whether additional informational and warning messages should be displayed.

See Also

Using Code Coverage in Microsoft Visual Studio*

Code Coverage dialog box

Code Coverage Tool

Code Coverage dialog box

To access the **Code Coverage** dialog box, select **Tools** > **Intel Compiler** > **Code Coverage...**.

Use the **Code Coverage** dialog box to set the code coverage feature.

Phase 1 - Instrument: Select this checkbox to compile your code into an instrumented application.

Select the **Instrument with guards for threaded applications** checkbox to produce an instrumented object file that includes the collection of PGO data on applications that use a high level of parallelism.

The compiler option used is shown in **Compiler Options**.

Deselect the **Phase 1 - Instrument** checkbox to skip this phase.

Phase 2 - Run Instrumented Application(s): Select this checkbox to run your instrumented application as well as other applications.

You can specify the options to run with the applications by choosing the **Application Invocations...** button to access the **Applications Invocations** dialog box.

Deselect the **Phase 2 - Run Instrumented Application(s)** checkbox to skip this phase.

Phase 3 - Generate Report: Select this checkbox to generate a report with the results of running the instrumented application.

Choose the **Settings...** button to access the Code Coverage Settings dialog box to configure the settings.

Profile Directory: Where the profile is stored.

Browse: Button to browse for the profile directory.

Show this dialog next time: Choose this button to access the dialog box when you run profile guided optimization.

Save Settings: Choose this button to save your settings.

Run: Choose this button to start the profile guided optimization.

Cancel: Choose this button to close this dialog box without starting the profile guided optimization.

See Also

Using Code Coverage in Microsoft Visual Studio*
Code Coverage Settings dialog box
code coverage Tool

Code Coverage Settings dialog box

To access the **Code Coverage Settings** dialog box, choose the **Settings** button in the **Code Coverage** dialog box. Use the **Code Coverage Settings** dialog box to specify settings for the generated report.

Codecov Options

Additional Options: Any command that you enter in the edit box will be passed through to the tool: Codecov.exe.

Ignore Object Unwind Handlers: Set to **True** to ignore the object unwind handlers.

Show Execution Counts: Set to **True** to show the dynamic execution counts in the report.

Treat Partially-covered Code As Fully-covered: Set to **True** to treat partially covered code as fully covered code.

Profmerge Options

Additional Options: Any command that you enter in the edit box will be passed through to the tool: Profmerge.exe.

Dump Profile Information: Set to **True** to include profile information in the report.

Exclude Functions: Enter the functions that will be excluded from the profile. The functions must be separated by a comma (","). A period (".") can be used as a wildcard character in function names.

See Also

Code Coverage dialog box

Using Code Coverage in the Microsoft Visual Studio* IDE
code coverage Tool

Using Xcode* (macOS)

Create an Xcode* Project

To create a new Xcode* project:

1. Launch the Xcode application.
2. Select **File** > **New** > **Project...**

   The **Choose a template for your new project** window opens.
3. In the left pane, select **macOS** > **Application**.
4. Select a template, for example: **Command Line Tool**, and click **Next**.
5. Name your project, for example: Hello World, then enter a string for the **Organization Name** and **Organization Identifier** and select a language. Click **Next**.
6. Specify a directory for your project, and optionally select **Create local git repository for this project** to place your project under version control.
7. Click **Create**.

Xcode creates the named project directory, with an `.xcodeproj` extension. Your new project directory contains a `main.cpp` source file and other project files.

Each Xcode project has its own **Project Editor** window that displays project source files, targets, and executables.

See Also

Select the Intel® Compiler

To select the Intel® C++ Compiler Classic:

1. Select the target you want to change and click **Build Rules**.
2. Add a new rule by clicking **Editor** > **Add Build Rule** or pressing the **+** button.
3. Under **Process**, choose **C source files** or **C++ source files** , depending on your source files.
4. Under **Using**, select one of the options for the **ICC Intel® C++ Compiler**:

   - ***Major_Version***, such as **2021.1**: The most recently installed compiler, even if it is not the latest release.
   - **Latest Release**: The latest released compiler available on your system. This is useful when you have multiple installations of the Intel compiler and want to use the version most recently released by Intel.
   - ***Major_Version.n.nnn***: A specific package, such as 2021.1.0.000. This is useful when you have multiple packages of one major version installed.

NOTE If the Intel® Compiler does not show up in the drop-down list, it may mean that the compiler does not support your version of Xcode. To enable the Intel® Compiler in Xcode specify the Xcode path during installation and restart the program. The installer checks for the supported Xcode version and warns you in the case of an unsupported version.

NOTE
If you select a tool that does not support the source file type, that source file type is processed by a later rule that specifies that type. For example, even though Objective-C/C++ sources are derived from C sources, they are built by the Intel® C++ Compiler Classic.

See Also

Setting Compiler Options

Build the Target

A single project can contain multiple targets. The active target determines how your project is built. This topic describes how to build the target using the Xcode* IDE and documents the build steps using the `xcodebuild` command line utility.

NOTE Starting with the 19.0 release of the Intel® C++ Compiler Classic, macOS 32-bit applications are no longer supported. If you want to compile 32-bit applications, you must use an earlier version of the compiler and you must use Xcode 9.4 or earlier.

Build Using the Xcode IDE

1. Select the target in the project editor under **Targets**.
2. Select **Product** > **Build**.
3. To view the results of your build, click the **Log Navigator** button.

You can change the compilation order of the files in an Xcode target. To re-order the files listed under a target's **Compile Sources**, click a source file and drag it before, or after other compilations.

Build From the Command Line Using the xcodebuild Utility

You can use the `xcodebuild` utility to build a target. This utility uses the Xcode project settings to build target projects from the command line. If you have previously configured your Xcode project to build with the Intel compiler, `xcodebuild` invokes it from the command line.

To build from the command line:

1. Check that the Xcode project is configured to use the Intel® C++ Compiler Classic.
2. Launch a terminal window from the finder by selecting **Applications** > **Utilities** > **Terminal** .
3. Change directories to the directory containing the Xcode project file (`.xcodeproj`).
4. If you have multiple versions of Xcode, use the **xcode-select** utility to verify the current Xcode version.
5. Issue an `xcodebuild` command. For example:

```
xcodebuild -project HelloWorld.xcodeproj -target HelloWorld -configuration Debug
```

6. Run the program built in the example from the previous step by entering the following:

```
./build/Debug/HelloWorld
```

For more information, refer to the `xcodebuild` man page.

Set the Executable's Architecture

Before building a 64-bit executable from within Xcode, you may need to edit the executable's target architecture. To change the **Architectures** setting:

1. Click the target you want to change in the project editor under **Targets** and select the **Build Settings** tab.

2. Under **Architectures**, select the desired architecture.

NOTE The Intel® C++ Compiler Classic generates code solely for Intel® architectures.

Set Compiler Options

To use the Xcode* environment to set compiler options, including options specific to Intel® architecture:

1. Select a target.
2. Under the **Build Settings** tab, click **All**.
3. Scroll down to the list of ICC Intel® C++ Compiler categories.
4. To set an Intel® C++ Compiler Classic option in the Optimization category, scroll down to display **Optimization**.
5. Select a **Setting**, such as **Enable Interprocedural Optimization for Single File Compilation**, and set the option's state. If the **Quick Help** inspector is visible, information about the selected option appears under **Quick Help**.

Tip
Apple* has deprecated the **libstdc++** library, make sure you are using the **libc++** standard library.

The next time you build your project, the selected options are used in the compilation.

NOTE To view the settings that have changed from the established defaults, select the **Levels** button under **Build Settings**.

Run the Executable

Once you have built your Xcode* project, click the **Run** button. The output from the executable is displayed. This button runs the configuration currently associated with the button. Use the **Scheme Editor** to change the configuration associated with the button.

Tip To open the **Scheme Editor**, select **Product** > **Scheme...** > **Edit Scheme...**

Use Dynamic Libraries

Using the Dynamic Libraries does not assume that the Apple* System Integrity Protection feature purges environment variables, such as `DYLD_LIBRARY_PATH`, when launching the protected process. Refer to the https://developer.apple.com/library/archive/documentation/Security/Conceptual/System_Integrity_Protection_Guide/Introduction/Introduction.html for more information. Xcode must take this into account and set the proper environment variables in the Xcode environment.

You can build your Xcode project with the `-shared-intel` compiler option to link with the Intel dynamic libraries. Build your project with the `-qopenmp` or `-parallel` option to link in `libiomp5.dylib`. If you do this, you need to set Xcode build option `Runpath Search path` to an appropriate folder with the compiler and performance libraries, or specify the `DYLD_LIBRARY_PATH` environment variable in the Xcode environment.

To add the environment variable:

1. Open the **Scheme Editor** and select the **Run** action.
2. On the **Arguments** tab, under **Environment Variables**, click the **+** button.
3. Add `DYLD_LIBRARY_PATH`. Set the value to the full path to the Intel compiler's `/lib` directory.

NOTE If you build your project with the `-shared-intel`, `-qopenmp`, or `-parallel` compiler option without setting the `DYLD_LIBRARY_PATH` environment variable, a *library not found* error message results at runtime. Depending on your application, the error message may refer to a library other than the one noted in this example:

```
dyld: Library not loaded: libiomp5.dylib
Referenced
from: /Users/test/hello_world
Reason: image not found
```

Due to the Apple System Integrity Protection you may need to set the `DYLD_LIBRARY_PATH` explicitly in the launch string, or configure the `Runpath Search path` build option.

See Also

shared-intel option
openmp, Qopenmp option
parallel, Qparallel option

Use Intel Libraries with Xcode*

You can use the compiler with the following Intel Libraries, which may be included as a part of the product:

- Intel® oneAPI Data Analytics Library (oneDAL)
- Intel® Integrated Performance Primitives (Intel® IPP)
- Intel® oneAPI Threading Building Blocks (oneTBB)
- Intel® oneAPI Math Kernel Library (oneMKL)

To access these libraries in Xcode, select the target and go to **Build Settings** > **Intel® C++ Compiler Performance Library Build Components**.

To use **oneDAL** change the **Use Intel® oneAPI Data Analytics Library** settings as follows:

- **None**: Disables the use of the oneDAL.
- **Use threaded Intel® oneAPI Data Analytics Library**: Links using the threaded version of the library.
- **Use non-threaded Intel® oneAPI Data Analytics Library**: Links using the non-threaded version of the library.

The **Use Intel® oneAPI Threading Building Blocks Library** property enables the library and brings in the associated headers.

The **Use Intel Integrated Performance Primitives Libraries** property provides the following options in a drop-down menu:

- **None**: Disables the use of the Intel® IPP.
- **Use main libraries set**: Uses all the libraries, except the cryptography libraries.
- **Use main libraries and cryptography library**: Uses the cryptography and main libraries.

NOTE The cryptography libraries are subject to export laws.

The **Use Intel® oneAPI Math Kernel Library** property provides the following options in a drop-down menu:

- **None**: Disables the use of oneMKL.
- **Use threaded Intel® oneAPI Math Kernel Library**: Links using the threaded version of the library.
- **Use non-threaded Intel® oneAPI Math Kernel Library**: Links using the non-threaded version of the library.

For more information, see the oneDAL, Intel® Integrated Performance Primitives, oneTBB, and oneMKL documentation.

Part III

Compiler Reference

C/C++ Calling Conventions

There are a number of calling conventions that set the rules on how arguments are passed to a function and how the values are returned from the function.

Calling Conventions on Windows*

The following table summarizes the supported calling conventions on Windows*:

Calling Convention	Compiler Option	Description
__cdecl	`/Gd`	This is the default calling convention for C/C++ programs. It can be specified on a function with variable arguments.
__stdcall	`/Gz`	Standard calling convention used for Win32 API functions.
__fastcall	`/Gr`	Fast calling convention that specifies that arguments are passed in registers rather than on the stack.
__regcall	`/Qregcall` specifies that *__regcall* is the default calling convention for functions in the compilation, unless another calling convention is specified on a declaration.	Intel® C++ Compiler calling convention that specifies that as many arguments as possible are passed in registers; likewise, *__regcall* uses registers whenever possible to return values. This calling convention is ignored if specified on a function with variable arguments. For more information about the Intel-compatible vector functions ABI, see the article *Vector Function Application Binary Interface* at https://software.intel.com/content/www/us/en/develop/download/vector-simd-function-abi.html.

Calling Convention	Compiler Option	Description
		For more information about the GCC vector functions ABI, see the item Libmvec - vector math library document in the GLIBC wiki at sourceware.org.
__thiscall	**none**	Default calling convention used by C++ member functions that do not use variable arguments.
__vectorcall	`/Gv`	Calling convention that specifies that a function passing vector type arguments should utilize vector registers.

Calling Conventions on Linux* and macOS

The following table summarizes the supported calling conventions on Linux* and macOS:

Calling Convention	Compiler Option	Description
__attribute((cdecl))	**none**	Default calling convention for C/C ++ programs. Can be specified on a function with variable arguments.
__attribute((stdcall))	**none**	Calling convention that specifies the arguments are passed on the stack. Cannot be specified on a function with variable arguments.
__attribute((regparm (number)))	**none**	On systems based on IA-32 architecture, the regparm attribute causes the compiler to pass up to *number* arguments in registers `EAX`, `EDX`, and `ECX` instead of on the stack. Functions that take a variable number of arguments will continue to pass all of their arguments on the stack.
__attribute__((regcall))	`-regcall` specifies that *__regcall* is the default calling convention for functions in the compilation, unless another calling convention is specified on a declaration.	Intel® C++ Compiler calling convention that specifies that as many arguments as possible are passed in registers; likewise, *__regcall* uses registers whenever possible to return values. This calling convention is ignored if specified on a function with variable arguments.
__attribute__((vectorcall))	**none**	Calling convention that specifies that a function passing vector type arguments should utilize vector registers.

The __regcall Calling Convention

The *__regcall* calling convention is unique to the Intel® C++ Compiler and requires some additional explanation.

To use *__regcall*, place the keyword before a function declaration. For example:

Example

```
__regcall int foo (int i, int j);

// Linux* and macOS
__attribute__((regcall)) foo (int I, int j);
```

Available __regcall Registers

All registers in a *__regcall* function can be used for parameter passing/returning a value, except those that are reserved by the compiler. The following table lists the registers that are available in each register class depending on the default ABI for the compilation. The registers are used in the order shown in the table.

Register Class/ Architecture	IA-32 for Linux*	IA-32 for Windows*	Intel® 64 for Linux*	Intel® 64 for Windows*
GPR	EAX, ECX, EDX, EDI, ESI	ECX, EDX, EDI, ESI	RAX, RCX, RDX, RDI, RSI, R8, R9, R10, R11, R12, R14, R15	RAX, RCX, RDX, RDI, RSI, R8, R9, R11, R12, R14, R15
FP	ST0	ST0	ST0	ST0
MMX	None	None	None	None
XMM	XMM0 - XMM7	XMM0 - XMM7	XMM0 - XMM15	XMM0 - XMM15
YMM	YMM0 - YMM7	YMM0 - YMM7	YMM0 - YMM15	YMM0 - YMM15
ZMM	ZMM0 - ZMM7	ZMM0 - ZMM7	ZMM0 - YMM15	ZMM0 - YMM15

__regcall Data Type Classification

Parameters and return values for *__regcall* are classified by data type and are passed in the registers of the classes shown in the following table.

NOTE All types assigned to `XMM`, `YMM`, or `ZMM` in a non-SSE target are passed in the stack.

Type (for both unsigned and signed types)	IA-32	Intel® 64
bool, char, int, enum, _Decimal32, long, pointer	GPR	GPR
short, __mmask{8,16,32,64}	GPR	GPR
long long, __int64	See Structured Data Type Classification Rules	GPR
_Decimal64	XMM	GPR

Type (for both unsigned and signed types)	IA-32	Intel® 64
long double	`FP`	`FP`
float, double, float128, _Decimal128	`XMM`	`XMM`
__m128, __m128i, __m128d	`XMM`	`XMM`
__m256, __m256i, __m256d	`YMM`	`YMM`
__m512, __m512i, __m512d	`ZMM`	`ZMM`
complex type, struct, union	See Structured Data Type Classification Rules	See Structured Data Type Classification Rules

NOTE For the purpose of structured types, the classification of `GPR` class is used.

NOTE On systems based on IA-32 architecture, these 64-bit integer types (long long, __int64) get classified to the `GPR` class and are passed in two registers, as if they were implemented as a structure of two 32-bit integer fields.

Types that are smaller in size than registers than registers of their associated class are passed in the lower part of those registers; for example, float is passed in the lower four bytes of an `XMM` register.

__regcall Structured Data Type Classification Rules

Structures/unions and complex types are classified similarly to what is described in the x86_64 ABI, with the following exceptions:

- There is no limitation on the overall size of a structure.
- The register classes for basic types are given in Data Type Classifications.
- For systems based on the IA-32 architecture, classification is performed on four-bytes. For systems based on other architectures, classification is performed on eight-bytes.

__regcall Placement in Registers or on the Stack

After the classification described in Data Type Classifications and Structured Data Type Classification Rules, *__regcall* parameters and return values are either put into registers specified in Available Registers or placed in memory, according to the following:

- Each chunk (eight bytes on systems based on Intel® 64 architecture or four-bytes on systems based on IA-32 architecture) of a value of Data Type is assigned a register class. If enough registers from Available Registers are available, the whole value is passed in registers, otherwise the value is passed using the stack.
- If the classification were to use one or more register classes, then the registers of these classes from the table in Available Registers are used, in the order given there.
- If no more registers are available in one of the required register classes, then the whole value is put on the stack.

__regcall Registers that Preserve Their Values

The following registers preserve their values across a *__regcall* call, as long as they were not used for passing a parameter or returning a value:

Register Class/ABI	IA-32	Intel® 64 for Linux*	Intel® 64 for Windows*
GPR	ESI, EDI, EBX, EBP, ESP	R12 - R15, RBX, RBP, RSP	R12 - R15, RBX, RBP, RSP
FP	None	None	None
MMX	None	None	None
XMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15
YMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15
ZMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15

All other registers do not preserve their values across this call.

See Also
Structured Data Type Classification Rules
Data Type Classifications
Available Registers

Compiler Options

This compiler supports many compiler options you can use in your applications.

In this section, we provide the following:

- An alphabetical list of compiler options that includes their short descriptions
- Lists of deprecated and removed options
- General rules for compiler options and the conventions we use when referring to options
- Details about what appears in the compiler option descriptions
- A description of each compiler option. The descriptions appear under the option's functional category. Within each category, the options are listed in alphabetical order.

For details about new functionality, such as new compiler options, see the Release Notes for the product.

Alphabetical Option List

The following table lists all the current compiler options in alphabetical order.

A, QA	Specifies an identifier for an assertion.
alias-const, Qalias-const	Determines whether the compiler assumes a parameter of type pointer-to-const does not alias with a parameter of type pointer-to-non-const.
align	Determines whether variables and arrays are naturally aligned.
ansi	Enables language compatibility with the gcc option ansi.
ansi-alias, Qansi-alias	Enables or disables the use of ANSI aliasing rules in optimizations.
ansi-alias-check, Qansi-alias-check	Enables or disables the ansi-alias checker.

arch	Tells the compiler which features it may target, including which instruction sets it may generate.
auto-ilp32, Qauto-ilp32	Instructs the compiler to analyze the program to determine if there are 64-bit pointers that can be safely shrunk into 32-bit pointers and if there are 64-bit longs (on Linux* systems) that can be safely shrunk into 32-bit longs.
auto-p32	Instructs the compiler to analyze the program to determine if there are 64-bit pointers that can be safely shrunk to 32-bit pointers.
ax, Qax	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit.
B	Specifies a directory that can be used to find include files, libraries, and executables.
Bdynamic	Enables dynamic linking of libraries at run time.
bigobj	Increases the number of sections that an object file can contain.
Bstatic	Enables static linking of a user's library.
Bsymbolic	Binds references to all global symbols in a program to the definitions within a user's shared library.
Bsymbolic-functions	Binds references to all global function symbols in a program to the definitions within a user's shared library.
C	Places comments in preprocessed source output.
c	Prevents linking.
check	Checks for certain conditions at run time.
check-pointers, Qcheck-pointers	Determines whether the compiler checks bounds for memory access through pointers.
check-pointers-dangling, Qcheck-pointers-dangling	Determines whether the compiler checks for dangling pointer references.
check-pointers-mpx, Qcheck-pointers-mpx	Determines whether the compiler checks bounds for memory access through pointers on processors that support Intel® Memory Protection Extensions (Intel® MPX).
check-pointers-narrowing, Qcheck-pointers-narrowing	Determines whether the compiler enables or disables the narrowing of pointers to structure fields.
check-pointers-undimensioned, Qcheck-pointers-undimensioned	Determines whether the compiler checks bounds for memory access through arrays that are declared without dimensions.
clang-name	Specifies the name of the Clang compiler that should be used to set up the environment for C compilations.
clangxx-name	Specifies the name of the Clang++ compiler that should be used to set up the environment for C++ compilations.
complex-limited-range, Qcomplex-limited-range	Determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.
cxxlib	Determines whether the compiler links using the C++ run-time libraries and header files provided by gcc.
D	Defines a macro name that can be associated with an optional value.

dD, QdD	Same as option -dM, but outputs #define directives in preprocessed source.
debug (Linux* and macOS*)	Enables or disables generation of debugging information.
debug (Windows*)	Enables or disables generation of debugging information.
device-math-lib	Enables or disables certain device libraries. This is a deprecated option that may be removed in a future release.
diag, Qdiag	Controls the display of diagnostic information during compilation.
diag-dump, Qdiag-dump	Tells the compiler to print all enabled diagnostic messages.
diag-enable=power, Qdiag-enable:power	Controls whether diagnostics are enabled for possibly inefficient code that may affect power consumption on IA-32 and Intel® 64 architectures.
diag-error-limit, Qdiag-error-limit	Specifies the maximum number of errors allowed before compilation stops.
diag-file, Qdiag-file	Causes the results of diagnostic analysis to be output to a file.
diag-file-append, Qdiag-file-append	Causes the results of diagnostic analysis to be appended to a file.
diag-id-numbers, Qdiag-id-numbers	Determines whether the compiler displays diagnostic messages by using their ID number values.
diag-once, Qdiag-once	Tells the compiler to issue one or more diagnostic messages only once.
dM, QdM	Tells the compiler to output macro definitions in effect after preprocessing.
dN, QdN	Same as option -dD, but output #define directives contain only macro names.
dryrun	Specifies that driver tool commands should be shown but not executed.
dumpmachine	Displays the target machine and operating system configuration.
dumpversion	Displays the version number of the compiler.
dynamiclib	Invokes the libtool command to generate dynamic libraries.
dynamic-linker	Specifies a dynamic linker other than the default.
E	Causes the preprocessor to send output to stdout.
early-template-check	Lets you semantically check template function template prototypes before instantiation.
EH	Specifies the model of exception handling to be performed.
EP	Causes the preprocessor to send output to stdout, omitting #line directives.
F (macOS*)	Adds a framework directory to the head of an include file search path.
F (Windows*)	Specifies the stack reserve amount for the program.
Fa	Specifies that an assembly listing file should be generated.
FA	Specifies the contents of an assembly listing file.
fabi-version	Instructs the compiler to select a specific ABI implementation.

falias, Oa	Determines whether aliasing is assumed in a program.
falign-functions, Qfnalign	Tells the compiler to align functions on an optimal byte boundary.
falign-loops, Qalign-loops	Aligns loops to a power-of-two byte boundary.
falign-stack	Tells the compiler the stack alignment to use on entry to routines. This is a deprecated option that may be removed in a future release.
fargument-alias, Qalias-args	Determines whether function arguments can alias each other.
fargument-noalias-global	Tells the compiler that function arguments cannot alias each other and cannot alias global storage.
fasm-blocks	Enables the use of blocks and entire functions of assembly code within a C or C++ file.
fast	Maximizes speed across the entire program.
fast-transcendentals, Qfast-transcendentals	Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.
fasynchronous-unwind-tables	Determines whether unwind information is precise at an instruction boundary or at a call boundary.
fblocks	Determines whether Apple* blocks are enabled or disabled.
fbuiltin, Oi	Enables or disables inline expansion of intrinsic functions.
FC	Displays the full path of source files passed to the compiler in diagnostics.
fcf-protection, Qcf-protection	Enables Control-flow Enforcement Technology (CET) protection, which defends your program from certain attacks that exploit vulnerabilities. This option offers preliminary support for CET.
fcode-asm	Produces an assembly listing with machine code annotations.
fcommon	Determines whether the compiler treats common symbols as global definitions.
FD	Generates file dependencies related to the Microsoft* C/C++ compiler.
Fd	Lets you specify a name for a program database (PDB) file created by the compiler.
fdata-sections, Gw	Places each data item in its own COMDAT section.
fdefer-pop	Determines whether the compiler always pops the arguments to each function call as soon as that function returns.
Fe	Specifies the name for a built program or dynamic-link library.
feliminate-unused-debug-types, Qeliminate-unused-debug-types	Controls the debug information emitted for types declared in a compilation unit.
femit-class-debug-always	Controls the format and size of debug information generated by the compiler for C++ classes.
fexceptions	Enables exception handling table generation.
fextend-arguments, Qextend-arguments	Controls how scalar integer arguments are extended in calls to unprototyped and varargs functions.

ffat-lto-objects	Determines whether a fat link-time optimization (LTO) object, containing both intermediate language and object code, is generated during an interprocedural optimization compilation (-c –ipo).
ffnalias, Ow	Determines whether aliasing is assumed within functions.
ffreestanding , Qfreestanding	Ensures that compilation takes place in a freestanding environment.
ffriend-injection	Causes the compiler to inject friend functions into the enclosing namespace.
ffunction-sections, Gy	Places each function in its own COMDAT section.
fgnu89-inline	Tells the compiler to use C89 semantics for inline functions when in C99 mode.
fimf-absolute-error, Qimf-absolute-error	Defines the maximum allowable absolute error for math library function results.
fimf-accuracy-bits, Qimf-accuracy-bits	Defines the relative error for math library function results, including division and square root.
fimf-arch-consistency, Qimf-arch-consistency	Ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture.
fimf-domain-exclusion, Qimf-domain-exclusion	Indicates the input arguments domain on which math functions must provide correct results.
fimf-force-dynamic-target, Qimf-force-dynamic-target	Instructs the compiler to use run-time dispatch in calls to math functions.
fimf-max-error, Qimf-max-error	Defines the maximum allowable relative error for math library function results, including division and square root.
fimf-precision, Qimf-precision	Lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.
fimf-use-svml, Qimf-use-svml	Instructs the compiler to use the Short Vector Math Library (SVML) rather than the Intel® C++ Compiler Classic Math Library (LIBM) to implement math library functions.
finline	Tells the compiler to inline functions declared with __inline and perform C ++ inlining .
finline-functions	Enables function inlining for single file compilation.
finline-limit	Lets you specify the maximum size of a function to be inlined.
finstrument-functions, Qinstrument-functions	Determines whether function entry and exit points are instrumented.
FI	Tells the preprocessor to include a specified file name as the header file.
fixed	Causes the linker to create a program that can be loaded only at its preferred base address.
fjump-tables	Determines whether jump tables are generated for switch statements.
fkeep-static-consts , Qkeep-static-consts	Tells the compiler to preserve allocation of variables that are not referenced in the source.
Fm	Tells the linker to generate a link map file. This is a deprecated option that may be removed in a future release.

fma, Qfma	Determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor.
fmath-errno	Tells the compiler that errno can be reliably tested after calls to standard math library functions.
fmerge-constants	Determines whether the compiler and linker attempt to merge identical constants (string constants and floating-point constants) across compilation units.
fmerge-debug-strings	Causes the compiler to pool strings used in debugging information.
fminshared	Specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.
fmpc-privatize	Enables or disables privatization of all static data for the MultiProcessor Computing environment (MPC) unified parallel runtime.
fms-dialect	Enables support for a language dialect that is compatible with Microsoft Windows*, while maintaining link compatibility with GCC*.
fmudflap	Tells the compiler to instrument risky pointer operations to prevent buffer overflows and invalid heap use. This is a deprecated option that may be removed in a future release.
fno-gnu-keywords	Tells the compiler to not recognize typeof as a keyword.
fno-implicit-inline-templates	Tells the compiler to not emit code for implicit instantiations of inline templates.
fno-implicit-templates	Tells the compiler to not emit code for non-inline templates that are instantiated implicitly.
fnon-call-exceptions	Allows trapping instructions to throw C++ exceptions.
fnon-lvalue-assign	Determines whether casts and conditional expressions can be used as lvalues.
fno-operator-names	Disables support for the operator names specified in the standard.
fno-rtti	Disables support for run-time type information (RTTI).
fnsplit, Qfnsplit	Enables function splitting.
Fo	Specifies the name for an object file.
fomit-frame-pointer , Oy	Determines whether EBP is used as a general-purpose register in optimizations.
fopenmp-device-lib	Enables or disables certain device libraries for an OpenMP* target.
foptimize-sibling-calls	Determines whether the compiler optimizes tail recursive calls.
Fp	Lets you specify an alternate path or file name for precompiled header files.
fpack-struct	Specifies that structure members should be packed together.
fpascal-strings	Tells the compiler to allow for Pascal-style string literals.
fpermissive	Tells the compiler to allow for non-conformant code.
fpic	Determines whether the compiler generates position-independent code.
fpie	Tells the compiler to generate position-independent code. The generated code can only be linked into executables.

fp-model, fp	Controls the semantics of floating-point calculations.
fp-port, Qfp-port	Rounds floating-point results after floating-point operations.
fprotect-parens, Qprotect-parens	Determines whether the optimizer honors parentheses when floating-point expressions are evaluated.
fp-speculation, Qfp-speculation	Tells the compiler the mode in which to speculate on floating-point operations.
fp-stack-check, Qfp-stack-check	Tells the compiler to generate extra code after every function call to ensure that the floating-point stack is in the expected state.
fp-trap, Qfp-trap	Sets the floating-point trapping mode for the main routine.
fp-trap-all, Qfp-trap-all	Sets the floating-point trapping mode for all routines.
FR	Invokes the Microsoft* C/C++ compiler and tells it to produce a BSCMAKE .sbr file with complete symbolic information.
freg-struct-return	Tells the compiler to return struct and union values in registers when possible.
fshort-enums	Tells the compiler to allocate as many bytes as needed for enumerated types.
fsource-asm	Produces an assembly listing with source code annotations.
fstack-protector	Enables or disables stack overflow security checks for certain (or all) routines.
fstack-security-check	Determines whether the compiler generates code that detects some buffer overruns.
fsyntax-only	Tells the compiler to check only for correct syntax.
ftemplate-depth, Qtemplate-depth	Control the depth in which recursive templates are expanded.
ftls-model	Changes the thread local storage (TLS) model.
ftrapuv , Qtrapuv	Initializes stack local variables to an unusual value to aid error detection.
ftz, Qftz	Flushes denormal results to zero.
funroll-all-loops	Unroll all loops even if the number of iterations is uncertain when the loop is entered.
funsigned-bitfields	Determines whether the default bitfield type is changed to unsigned.
funsigned-char	Change default char type to unsigned.
fuse-ld	Tells the compiler to use a different linker instead of the default linker (ld).
fverbose-asm	Produces an assembly listing with compiler comments, including options and version information.
fvisibility	Specifies the default visibility for global symbols or the visibility for symbols in a file.
fvisibility-inlines-hidden	Causes inline member functions (those defined in the class declaration) to be marked hidden.
fzero-initialized-in-bss, Qzero-initialized-in-bss	Determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.

g	Tells the compiler to generate a level of debugging information in the object file.
GA	Enables faster access to certain thread-local storage (TLS) variables.
gcc, gcc-sys	Determines whether certain GNU macros are defined or undefined.
gcc-include-dir	Controls whether the gcc-specific include directory is put into the system include path.
gcc-name	Lets you specify the name of the GCC compiler that should be used to set up the environment for C compilations.
Gd	Makes __cdecl the default calling convention.
gdwarf	Lets you specify a DWARF Version format when generating debug information.
Ge	Enables stack-checking for all functions. This is a deprecated option that may be removed in a future release.
GF	Enables read-only string-pooling optimization.
Gh	Calls a function to aid custom user profiling.
GH	Calls a function to aid custom user profiling.
global-hoist, Qglobal-hoist	Enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source.
Gm	Enables a minimal rebuild.
gnu-prefix	Lets you specify a prefix that will be added to the names of gnu utilities called from the compiler.
GR	Enables or disables C++ Run Time Type Information (RTTI).
Gr	Makes __fastcall the default calling convention.
grecord-gcc-switches	Causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.
GS	Determines whether the compiler generates code that detects some buffer overruns.
Gs	Lets you control the threshold at which the stack checking routine is called or not called.
gsplit-dwarf	Creates a separate object file containing DWARF debug information.
GT	Enables fiber-safe thread-local storage of data.
guard	Enables the control flow protection mechanism.
guide, Qguide	Lets you set a level of guidance for auto-vectorization, auto parallelism, and data transformation.
guide-data-trans, Qguide-data-trans	Lets you set a level of guidance for data transformation.
guide-file, Qguide-file	Causes the results of guided auto parallelism to be output to a file.
guide-file-append, Qguide-file-append	Causes the results of guided auto parallelism to be appended to a file.

guide-opts, Qguide-opts	Tells the compiler to analyze certain code and generate recommendations that may improve optimizations.
guide-par, Qguide-par	Lets you set a level of guidance for auto parallelism.
guide-vec, Qguide-vec	Lets you set a level of guidance for auto-vectorization.
Gv	Tells the compiler to use the vector calling convention (__vectorcall) when passing vector type arguments.
gxx-name	Lets you specify the name of the g++ compiler that should be used to set up the environment for C++ compilations.
GZ	Initializes all local variables. This is a deprecated option that may be removed in a future release.
Gz	Makes __stdcall the default calling convention.
H, QH	Tells the compiler to display the include file order and continue compilation.
H (Windows*)	Causes the compiler to limit the length of external symbol names. This is a deprecated option. There is no replacement option.
help	Displays all supported compiler options or supported compiler options within a specified category of options.
help-pragma, Qhelp-pragma	Displays all supported pragmas.
homeparams	Tells the compiler to store parameters passed in registers to the stack.
hotpatch	Tells the compiler to prepare a routine for hotpatching.
I	Specifies an additional directory to search for include files.
I-	Splits the include path.
icc, Qicl	Determines whether certain Intel®-specific compiler macros are defined or undefined.
idirafter	Adds a directory to the second include file search path.
imacros	Allows a header to be specified that is included in front of the other headers in the translation unit.
inline-calloc, Qinline-calloc	Tells the compiler to inline calls to calloc() as calls to malloc() and memset().
inline-factor, Qinline-factor	Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.
inline-forceinline, Qinline-forceinline	Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.
inline-level, Ob	Specifies the level of inline function expansion.
inline-max-per-compile, Qinline-max-per-compile	Specifies the maximum number of times inlining may be applied to an entire compilation unit.
inline-max-per-routine, Qinline-max-per-routine	Specifies the maximum number of times the inliner may inline into a particular routine.
inline-max-size, Qinline-max-size	Specifies the lower limit for the size of what the inliner considers to be a large routine.

inline-max-total-size, Qinline-max-total-size	Specifies how much larger a routine can normally grow when inline expansion is performed.
inline-min-caller-growth, Qinline-min-caller-growth	Lets you specify a function size n for which functions of size <= n do not contribute to the estimated growth of the caller when inlined.
inline-min-size, Qinline-min-size	Specifies the upper limit for the size of what the inliner considers to be a small routine.
intel-extensions, Qintel-extensions	Enables or disables all Intel® C and Intel® C++ language extensions.
intel-freestanding	Lets you compile in the absence of a gcc environment.
intel-freestanding-target-os	Lets you specify the target operating system for compilation.
ip, Qip	Determines whether additional interprocedural optimizations for single-file compilation are enabled.
ip-no-inlining, Qip-no-inlining	Disables full and partial inlining enabled by interprocedural optimization options.
ip-no-pinlining, Qip-no-pinlining	Disables partial inlining enabled by interprocedural optimization options.
ipo, Qipo	Enables interprocedural optimization between files.
ipo-c, Qipo-c	Tells the compiler to optimize across multiple files and generate a single object file.
ipo-jobs, Qipo-jobs	Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).
ipo-S, Qipo-S	Tells the compiler to optimize across multiple files and generate a single assembly file.
ipo-separate, Qipo-separate	Tells the compiler to generate one object file for every source file.
ipp-link, Qipp-link	Controls whether the compiler links to static or dynamic threaded Intel® Integrated Performance Primitives (Intel® IPP) run-time libraries.
iprefix	Lets you indicate the prefix for referencing directories that contain header files.
iquote	Adds a directory to the front of the include file search path for files included with quotes but not brackets.
isystem	Specifies a directory to add to the start of the system include path.
iwithprefix	Appends a directory to the prefix passed in by -iprefix and puts it on the include search path at the end of the include directories.
iwithprefixbefore	Similar to -iwithprefix except the include directory is placed in the same place as -I command-line include directories.
J	Sets the default character type to unsigned.
Kc++, TP	Tells the compiler to process all source or unrecognized file types as C++ source files. This is a deprecated option that may be removed in a future release.
l	Tells the linker to search for a specified library when linking.
L	Tells the linker to search for libraries in a specified directory before searching the standard directories.

LD	Specifies that a program should be linked as a dynamic-link (DLL) library.
link	Passes user-specified options directly to the linker at compile time.
m	Tells the compiler which features it may target, including which instruction sets it may generate.
M, QM	Tells the compiler to generate makefile dependency lines for each source file.
m32, m64 , Qm32, Qm64	Tells the compiler to generate code for a specific architecture.
m80387	Specifies whether the compiler can use x87 instructions.
malign-double	Determines whether double, long double, and long long types are naturally aligned. This option is equivalent to specifying option align.
malign-mac68k	Aligns structure fields on 2-byte boundaries (m68k compatible).
malign-natural	Aligns larger types on natural size-based boundaries (overrides ABI).
malign-power	Aligns based on ABI-specified alignment rules.
map-opts, Qmap-opts	Maps one or more compiler options to their equivalent on a different operating system.
march	Tells the compiler to generate code for processors that support certain features.
masm	Tells the compiler to generate the assembler output file using a selected dialect.
mauto-arch, Qauto-arch	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit.
mbranches-within-32B-boundaries, Qbranches-within-32B-boundaries	Tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.
mcmodel	Tells the compiler to use a specific memory model to generate code and store data.
mconditional-branch, Qconditional-branch	Lets you identify and fix code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction.
MD	Tells the linker to search for unresolved references in a multithreaded, dynamic-link run-time library.
MD, QMD	Preprocess and compile, generating output file containing dependency information ending with extension .d.
mdynamic-no-pic	Generates code that is not position-independent but has position-independent external references.
MF, QMF	Tells the compiler to generate makefile dependency information in a file.
MG, QMG	Tells the compiler to generate makefile dependency lines for each source file.
minstruction, Qinstruction	Determines whether MOVBE instructions are generated for certain Intel® processors.
mlong-double	Lets you override the default configuration of the long double data type.

MM, QMM	Tells the compiler to generate makefile dependency lines for each source file.
MMD, QMMD	Tells the compiler to generate an output file containing dependency information.
momit-leaf-frame-pointer	Determines whether the frame pointer is omitted or kept in leaf functions.
MP	Tells the compiler to add a phony target for each dependency.
mp1, Qprec	Improves floating-point precision and consistency.
MP-force	Disables the default heuristics used when compiler option /MP is specified. This lets you control the number of processes spawned.
MQ	Changes the default target rule for dependency generation.
mregparm	Lets you control the number registers used to pass integer arguments.
mregparm-version	Determines which version of the Application Binary Interface (ABI) is used for the regparm parameter passing convention.
mstringop-inline-threshold, Qstringop-inline-threshold	Tells the compiler to not inline calls to buffer manipulation functions such as memcpy and memset when the number of bytes the functions handle are known at compile time and greater than the specified value.
mstringop-strategy, Qstringop-strategy	Lets you override the internal decision heuristic for the particular algorithm used when implementing buffer manipulation functions such as memcpy and memset.
MT	Tells the linker to search for unresolved references in a multithreaded, static run-time library.
MT, QMT	Changes the default target rule for dependency generation.
mtune, tune	Performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike -march).
multibyte-chars, Qmultibyte-chars	Determines whether multi-byte characters are supported.
multiple-processes , MP	Creates multiple processes that can be used to compile large numbers of source files at the same time.
noBool	Disables the bool keyword.
no-bss-init, Qnobss-init	Tells the compiler to place in the DATA section any uninitialized variables and explicitly zero-initialized variables. This is a deprecated option that may be removed in a future release.
nodefaultlibs	Prevents the compiler from using standard libraries when linking.
no-intel-lib, Qno-intel-lib	Disables linking to specified Intel® libraries, or to all Intel® libraries.
no-libgcc	Prevents the linking of certain gcc-specific libraries.
nolib-inline	Disables inline expansion of standard library or intrinsic functions.
nologo	Tells the compiler to not display compiler version information.
nostartfiles	Prevents the compiler from using standard startup files when linking.
nostdinc++	Do not search for header files in the standard directories for C++, but search the other standard directories.

nostdlib	Prevents the compiler from using standard libraries and startup files when linking.
O	Specifies the code optimization for applications.
o	Specifies the name for an output file.
Od	Disables all optimizations.
Ofast	Sets certain aggressive options to improve the speed of your application.
Os	Enables optimizations that do not increase code size; it produces smaller code size than O2.
Ot	Enables all speed optimizations.
Ox	Enables maximum optimizations.
p	Compiles and links for function profiling with gprof(1).
P	Tells the compiler to stop the compilation process and write the results to a file.
par-affinity, Qpar-affinity	Specifies thread affinity.
parallel, Qparallel	Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.
parallel-source-info, Qparallel-source-info	Enables or disables source location emission when OpenMP* or auto-parallelism code is generated.
par-loops, Qpar-loops	Lets you select between old or new implementations of parallel loop support.
par-num-threads, Qpar-num-threads	Specifies the number of threads to use in a parallel region.
par-runtime-control, Qpar-runtime-control	Generates code to perform run-time checks for loops that have symbolic loop bounds.
par-schedule, Qpar-schedule	Lets you specify a scheduling algorithm for loop iterations.
par-threshold, Qpar-threshold	Sets a threshold for the auto-parallelization of loops.
pc, Qpc	Enables control of floating-point significand precision.
pch	Tells the compiler to use appropriate precompiled header files.
pch-create	Tells the compiler to create a precompiled header file.
pch-dir	Tells the compiler the location for precompiled header files.
pch-use	Tells the compiler to use a precompiled header file.
pdbfile	Lets you specify the name for a program database (PDB) file created by the linker.
pie	Determines whether the compiler generates position-independent code that will be linked into an executable.
pragma-optimization-level	Specifies which interpretation of the optimization_level pragma should be used if no prefix is specified.
prec-div, Qprec-div	Improves precision of floating-point divides.
prec-sqrt, Qprec-sqrt	Improves precision of square root implementations.

print-multi-lib	Prints information about where system libraries should be found.
print-sysroot	Prints the target sysroot directory that is used during compilation.
prof-data-order, Qprof-data-order	Enables or disables data ordering if profiling information is enabled.
prof-dir, Qprof-dir	Specifies a directory for profiling information output files.
prof-file, Qprof-file	Specifies an alternate file name for the profiling summary files.
prof-func-groups	Enables or disables function grouping if profiling information is enabled.
prof-func-order, Qprof-func-order	Enables or disables function ordering if profiling information is enabled.
prof-gen, Qprof-gen	Produces an instrumented object file that can be used in profile guided optimization.
prof-gen-sampling	Tells the compiler to generate debug discriminators in debug output. This aids in developing more precise sampled profiling output. This is a deprecated option that may be removed in a future release.
prof-hotness-threshold, Qprof-hotness-threshold	Lets you set the hotness threshold for function grouping and function ordering.
prof-src-dir, Qprof-src-dir	Determines whether directory information of the source file under compilation is considered when looking up profile data records.
prof-src-root, Qprof-src-root	Lets you use relative directory paths when looking up profile data and specifies a directory as the base.
prof-src-root-cwd, Qprof-src-root-cwd	Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.
prof-use, Qprof-use	Enables the use of profiling information during optimization.
prof-use-sampling	Lets you use data files produced by hardware profiling to produce an optimized executable. This is a deprecated option that may be removed in a future release.
prof-value-profiling, Qprof-value-profiling	Controls which values are value profiled.
pthread	Tells the compiler to use pthreads library for multithreading support.
Qcov-dir	Specifies a directory for profiling information output files that can be used with the codecov or tselect tool.
Qcov-file	Specifies an alternate file name for the profiling summary files that can be used with the codecov or tselect tool.
Qcov-gen	Produces an instrumented object file that can be used with the codecov or tselect tool.
Qcxx-features	Enables standard C++ features without disabling Microsoft* features.
qdaal, Qdaal	Tells the compiler to link to certain libraries in the Intel® oneAPI Data Analytics Library (oneDAL).
Qgcc-dialect	Enables support for a limited gcc-compatible dialect on Windows*.
Qinline-dllimport	Determines whether dllimport functions are inlined.

Qinstall	Specifies the root directory where the compiler installation was performed.
qipp, Qipp	Tells the compiler to link to some or all of the Intel® Integrated Performance Primitives (Intel® IPP) libraries.
Qlocation	Specifies the directory for supporting tools.
Qlong-double	Changes the default size of the long double data type.
qmkl, Qmkl	Tells the compiler to link to certain libraries in the Intel® oneAPI Math Kernel Library (oneMKL). On Windows systems, you must specify this option at compile time.
Qms	Tells the compiler to emulate Microsoft* compatibility bugs.
qopenmp, Qopenmp	Enables the parallelizer to generate multi-threaded code based on OpenMP* directives.
qopenmp-lib, Qopenmp-lib	Lets you specify an OpenMP* run-time library to use for linking.
qopenmp-link	Controls whether the compiler links to static or dynamic OpenMP* run-time libraries.
qopenmp-simd, Qopenmp-simd	Enables or disables OpenMP* SIMD compilation.
qopenmp-stubs, Qopenmp-stubs	Enables compilation of OpenMP* programs in sequential mode.
qopenmp-threadprivate, Qopenmp-threadprivate	Lets you specify an OpenMP* threadprivate implementation.
qopt-args-in-regs, Qopt-args-in-regs	Determines whether calls to routines are optimized by passing parameters in registers instead of on the stack. This option is deprecated and will be removed in a future release.
qopt-assume-safe-padding, Qopt-assume-safe-padding	Determines whether the compiler assumes that variables and dynamically allocated memory are padded past the end of the object.
qopt-block-factor, Qopt-block-factor	Lets you specify a loop blocking factor.
qopt-calloc	Tells the compiler to substitute a call to _intel_fast_calloc() for a call to calloc().
qopt-class-analysis, Qopt-class-analysis	Determines whether C++ class hierarchy information is used to analyze and resolve C++ virtual function calls at compile time.
qopt-dynamic-align, Qopt-dynamic-align	Enables or disables dynamic data alignment optimizations.
Qoption	Passes options to a specified tool.
qopt-jump-tables, Qopt-jump-tables	Enables or disables generation of jump tables for switch statements.
qopt-malloc-options	Lets you specify an alternate algorithm for malloc().
qopt-matmul, Qopt-matmul	Enables or disables a compiler-generated Matrix Multiply (matmul) library call.
qopt-mem-layout-trans, Qopt-mem-layout-trans	Controls the level of memory layout transformations performed by the compiler.

qopt-multiple-gather-scatter-by-shuffles, Qopt-multiple-gather-scatter-by-shuffles	Enables or disables the optimization for multiple adjacent gather/scatter type vector memory references.
qopt-multi-version-aggressive, Qopt-multi-version-aggressive	Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.
qopt-prefetch, Qopt-prefetch	Enables or disables prefetch insertion optimization.
qopt-prefetch-distance, Qopt-prefetch-distance	Specifies the prefetch distance to be used for compiler-generated prefetches inside loops.
qopt-prefetch-issue-excl-hint, Qopt-prefetch-issue-excl-hint	Supports the prefetchW instruction in Intel® microarchitecture code name Broadwell and later.
qopt-ra-region-strategy, Qopt-ra-region-strategy	Selects the method that the register allocator uses to partition each routine into regions.
qopt-report, Qopt-report	Tells the compiler to generate an optimization report.
qopt-report-annotate, Qopt-report-annotate	Enables the annotated source listing feature and specifies its format.
qopt-report-annotate-position, Qopt-report-annotate-position	Enables the annotated source listing feature and specifies the site where optimization messages appear in the annotated source in inlined cases of loop optimizations.
qopt-report-embed, Qopt-report-embed	Determines whether special loop information annotations will be embedded in the object file and/or the assembly file when it is generated.
qopt-report-file, Qopt-report-file	Specifies that the output for the optimization report goes to a file, stderr, or stdout.
qopt-report-filter, Qopt-report-filter	Tells the compiler to find the indicated parts of your application, and generate optimization reports for those parts of your application.
qopt-report-format, Qopt-report-format	Specifies the format for an optimization report.
qopt-report-help, Qopt-report-help	Displays the optimizer phases available for report generation and a short description of what is reported at each level.
qopt-report-names, Qopt-report-names	Specifies whether mangled or unmangled names should appear in the optimization report.
qopt-report-per-object, Qopt-report-per-object	Tells the compiler that optimization report information should be generated in a separate file for each object.
qopt-report-phase, Qopt-report-phase	Specifies one or more optimizer phases for which optimization reports are generated.
qopt-report-routine, Qopt-report-routine	Tells the compiler to generate an optimization report for each of the routines whose names contain the specified substring.
qopt-streaming-stores, Qopt-streaming-stores	Enables generation of streaming stores for optimization.
qopt-subscript-in-range, Qopt-subscript-in-range	Determines whether the compiler assumes that there are no "large" integers being used or being computed inside loops.
qopt-zmm-usage, Qopt-zmm-usage	Defines a level of zmm registers usage.

qoverride-limits, Qoverride-limits	Lets you override certain internal compiler limits that are intended to prevent excessive memory usage or compile times for very large, complex compilation units.
Qpar-adjust-stack	Tells the compiler to generate code to adjust the stack size for a fiber-based main thread.
Qpatchable-addresses	Tells the compiler to generate code such that references to statically assigned addresses can be patched.
Qpchi	Enable precompiled header coexistence to reduce build time.
Qsafeseh	Registers exception handlers for safe exception handling.
Qsfalign	Specifies stack alignment for functions. This is a deprecated option that may be removed in a future release..
qsimd-honor-fp-model, Qsimd-honor-fp-model	Tells the compiler to obey the selected floating-point model when vectorizing SIMD loops.
qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction	Tells the compiler to serialize floating-point reduction when vectorizing SIMD loops.
qtbb, Qtbb	Tells the compiler to link to the Intel® oneAPI Threading Building Blocks (oneTBB) libraries.
Quse-msasm-symbols	Tells the compiler to use a dollar sign ("$") when producing symbol names.
Qvc	Specifies compatibility with Microsoft Visual C++* (MSVC) or Microsoft Visual Studio*.
Qvla	Determines whether variable length arrays are enabled.
rcd, Qrcd	Enables fast float-to-integer conversions. This is a deprecated option that may be removed in a future release.
regcall, Qregcall	Tells the compiler that the __regcall calling convention should be used for functions that do not directly specify a calling convention.
restrict, Qrestrict	Determines whether pointer disambiguation is enabled with the restrict qualifier.
RTC	Enables checking for certain run-time conditions.
S	Causes the compiler to compile to an assembly file only and not link.
save-temps , Qsave-temps	Tells the compiler to save intermediate files created during compilation.
scalar-rep, Qscalar-rep	Enables or disables the scalar replacement optimization done by the compiler as part of loop transformations.
shared	Tells the compiler to produce a dynamic shared object instead of an executable.
shared-intel	Causes Intel-provided libraries to be linked in dynamically.
shared-libgcc	Links the GNU libgcc library dynamically.
showIncludes	Tells the compiler to display a list of the include files.
simd, Qsimd	Enables or disables compiler interpretation of simd pragmas.
simd-function-pointers, Qsimd-function-pointers	Enables or disables pointers to simd-enabled functions.

sox	Tells the compiler to save the compilation options and version number in the executable file. It also lets you choose whether to include lists of certain functions .
static	Prevents linking with shared libraries.
static-intel	Causes Intel-provided libraries to be linked in statically.
static-libgcc	Links the GNU libgcc library statically.
staticlib	Invokes the libtool command to generate static libraries.
static-libstdc++	Links the GNU libstdc++ library statically.
std, Qstd	Tells the compiler to conform to a specific language standard.
stdlib	Lets you select the C++ library to be used for linking.
strict-ansi	Tells the compiler to implement strict ANSI conformance dialect.
sysroot	Specifies the root directory where headers and libraries are located.
T	Tells the linker to read link commands from a file.
Tc	Tells the compiler to process a file as a C source file.
TC	Tells the compiler to process all source or unrecognized file types as C source files.
Tp	Tells the compiler to process a file as a C++ source file.
traceback	Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.
U	Undefines any definition currently in effect for the specified macro .
u (Linux*)	Tells the compiler the specified symbol is undefined.
u (Windows*)	Disables all predefined macros and assertions.
undef	Disables all predefined macros .
unroll , Qunroll	Tells the compiler the maximum number of times to unroll loops.
unroll-aggressive, Qunroll-aggressive	Determines whether the compiler uses more aggressive unrolling for certain loops.
use-asm, Quse-asm	Tells the compiler to produce objects through the assembler. This is a deprecated option that may be removed in a future release.
use-intel-optimized-headers, Quse-intel-optimized-headers	Determines whether the performance headers directory is added to the include path search list.
use-msasm	Enables the use of blocks and entire functions of assembly code within a C or C++ file.
V	Displays the compiler version information.
V (Windows*)	Places the text string specified into the object file being generated by the compiler.
v	Specifies that driver tool commands should be displayed and executed.
vd	Enables or suppresses hidden vtordisp members in C++ objects.

vec, Qvec	Enables or disables vectorization.
vecabi, Qvecabi	Determines which vector function application binary interface (ABI) the compiler uses to create or call vector functions.
vec-guard-write, Qvec-guard-write	Tells the compiler to perform a conditional check in a vectorized loop.
vec-threshold, Qvec-threshold	Sets a threshold for the vectorization of loops.
version	Tells the compiler to display GCC-style version information.
vmb	Selects the smallest representation that the compiler uses for pointers to members.
vmg	Selects the general representation that the compiler uses for pointers to members.
vmm	Enables pointers to class members with single or multiple inheritance.
vms	Enables pointers to members of single-inheritance classes.
vmv	Enables pointers to members of any inheritance type.
w	Disables all warning messages.
w, W	Specifies the level of diagnostic messages to be generated by the compiler.
Wa	Passes options to the assembler for processing.
Wabi	Determines whether a warning is issued if generated code is not C++ ABI compliant.
Wall	Enables warning and error diagnostics.
watch	Tells the compiler to display certain information to the console output window.
Wbrief	Tells the compiler to display a shorter form of diagnostic output.
Wcheck	Tells the compiler to perform compile-time code checking for certain code.
Wcheck-unicode-security	Determines whether the compiler performs source code checking for Unicode vulnerabilities.
Wcomment	Determines whether a warning is issued when /* appears in the middle of a /* */ comment.
Wcontext-limit, Qcontext-limit	Set the maximum number of template instantiation contexts shown in diagnostic.
wd, Qwd	Disables a soft diagnostic. This is a deprecated option that may be removed in a future release.
Wdeprecated	Determines whether warnings are issued for deprecated C++ headers.
we, Qwe	Changes a soft diagnostic to an error. This is a deprecated option that may be removed in a future release.
Weffc++, Qeffc++	Enables warnings based on certain C++ programming guidelines.
Werror, WX	Changes all warnings to errors.

Werror-all	Causes all warnings and currently enabled remarks to be reported as errors.
Wextra-tokens	Determines whether warnings are issued about extra tokens at the end of preprocessor directives.
Wformat	Determines whether argument checking is enabled for calls to printf, scanf, and so forth.
Wformat-security	Determines whether the compiler issues a warning when the use of format functions may cause security problems.
Wic-pointer	Determines whether warnings are issued for conversions between pointers to distinct scalar types with the same representation.
Winline	Warns when a function that is declared as inline is not inlined.
Wl	Passes options to the linker for processing.
WL	Tells the compiler to display a shorter form of diagnostic output.
Wmain	Determines whether a warning is issued if the return type of main is not expected.
Wmissing-declarations	Determines whether warnings are issued for global functions and variables without prior declaration.
Wmissing-prototypes	Determines whether warnings are issued for missing prototypes.
wn, Qwn	Controls the number of errors displayed before compilation stops. This is a deprecated option that may be removed in a future release.
Wnon-virtual-dtor	Tells the compiler to issue a warning when a class appears to be polymorphic, yet it declares a non-virtual one.
wo, Qwo	Tells the compiler to issue one or more diagnostic messages only once. This is a deprecated option that may be removed in a future release.
Wp	Passes options to the preprocessor.
Wp64	Tells the compiler to display diagnostics for 64-bit porting.
Wpch-messages	Determines whether the compiler shows precompiled header (PCH) informational messages.
Wpointer-arith	Determines whether warnings are issued for questionable pointer arithmetic.
Wport	Tells the compiler to issue portability diagnostics.
wr, Qwr	Changes a soft diagnostic to an remark. This is a deprecated option that may be removed in a future release.
Wremarks	Tells the compiler to display remarks and comments.
Wreorder	Tells the compiler to issue a warning when the order of member initializers does not match the order in which they must be executed.
Wreturn-type	Determines whether warnings are issued when a function is declared without a return type, when the definition of a function returning void contains a return statement with an expression, or when the closing brace of a function returning non-void is reached.
Wshadow	Determines whether a warning is issued when a variable declaration hides a previous declaration.

Wsign-compare	Determines whether warnings are issued when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.
Wstrict-aliasing	Determines whether warnings are issued for code that might violate the optimizer's strict aliasing rules.
Wstrict-prototypes	Determines whether warnings are issued for functions declared or defined without specified argument types.
Wtrigraphs	Determines whether warnings are issued if any trigraphs are encountered that might change the meaning of the program.
Wuninitialized	Determines whether a warning is issued if a variable is used before being initialized.
Wunknown-pragmas	Determines whether a warning is issued if an unknown #pragma directive is used.
Wunused-function	Determines whether a warning is issued if a declared function is not used.
Wunused-variable	Determines whether a warning is issued if a local or non-constant static variable is unused after being declared.
ww, Qww	Changes a soft diagnostic to an warning. This is a deprecated option that may be removed in a future release.
Wwrite-strings	Issues a diagnostic message if const char * is converted to (non-const) char *.
X	Removes standard directories from the include file search path.
x (type option)	All source files found subsequent to -x type will be recognized as a particular type.
x, Qx	Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.
xHost, QxHost	Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.
Xlinker	Passes a linker option directly to the linker.
Y-	Tells the compiler to ignore all other precompiled header files.
Yc	Tells the compiler to create a precompiled header file.
Yd	Tells the compiler to add complete debugging information in all object files created from a precompiled header file when option /Zi or /Z7 is specified. This is a deprecated option that may be removed in a future release.
Yu	Tells the compiler to use a precompiled header file.
Za	Disables Microsoft* Visual C++* compiler language extensions.
Zc	Lets you specify ANSI C standard conformance for certain language features.
Ze	Enables Microsoft* Visual C++* compiler language extensions. This is a deprecated option that may be removed in a future release.
Zg	Tells the compiler to generate function prototypes. This is a deprecated option that may be removed in a future release.

Zi, Z7 , ZI	Tells the compiler to generate full debugging information in either an object (.obj) file or a project database (PDB) file.
Zl	Causes library names to be omitted from the object file.
Zo	Enables or disables generation of enhanced debugging information for optimized code.
Zp	Specifies alignment for structures on byte boundaries.
Zs	Tells the compiler to check only for correct syntax.

General Rules for Compiler Options

This section describes general rules for compiler options and it contains information about how we refer to compiler option names in descriptions.

General Rules for Compiler Options

Compiler options may be case sensitive, and may have different meanings depending on their case. For example, option `c` prevents linking, but option `C` places comments in preprocessed source output.

Options specified on the command line apply to all files named on the command line.

Options can take arguments in the form of file names, strings, letters, or numbers. If a string includes spaces, the string must be enclosed in quotation marks.

Compiler options can appear in any order.

Unless you specify certain options, the command line will both compile and link the files you specify.

You can abbreviate some option names, entering as many characters as are needed to uniquely identify the option.

Certain options accept one or more keyword arguments following the option name. For example, architecture option `x` option accepts several keywords.

To specify multiple keywords, you typically specify the option multiple times.

To disable an option, specify the negative form of the option if one exists.

If there are enabling and disabling versions of an option on the command line, the last one on the command line takes precedence.

Compiler options remain in effect for the whole compilation unless overridden by a compiler #pragma.

Linux and macOS

You cannot combine options with a single dash. For example, this form is incorrect: `-Ec`; this form is correct: `-E -c`

Windows

You cannot combine options with a single slash. For example: This form is incorrect: `/Ec`; this form is correct: `/E /c`

All compiler options must precede `/link` options, if any, on the command line.

Compiler options remain in effect for the whole compilation unless overridden by a compiler #pragma.

You can disable one or more optimization options by specifying option `/Od` last on the command line.

NOTE
The /Od option is part of a mutually-exclusive group of options that includes /Od, /O1, /O2, /O3, and /Ox. The last of any of these options specified on the command line will override the previous options from this group.

How We Refer to Compiler Option Names in Descriptions

Within documentation, compiler option names that are very different are spelled out in full.

However, many compiler option names are very similar except for initial characters. For these options, we use the following shortcuts when referencing their names in descriptions:

- No initial – or /

  This shortcut is used for option names that are the same for Linux and Windows except for the initial character.

  For example, Fa denotes:

  - Linux and macOS: -Fa
  - Windows: /Fa
- [Q]option-name

  This shortcut is used for option names that only differ because the Windows form starts with a Q.

  For example, [Q]ipo denotes:

  - Linux and macOS: -ipo
  - Windows: /Qipo
- [q or Q]option-name

  This shortcut is used for option names that only differ because the Linux form starts with a q and the Windows form starts with a Q.

  For example, [q or Q]opt-report denotes:

  - Linux and macOS: -qopt-report
  - Windows: /Qopt-report

What Appears in the Compiler Option Descriptions

This section contains details about what appears in the option descriptions.

Following sections include individual descriptions of all the current compiler options. The option descriptions are arranged by functional category. Within each category, the option names are listed in alphabetical order.

Each option description contains the following information:

- The primary name for the option and a short description of the option.
- Architecture Restrictions

  This section only appears if there is a known architecture restriction for the option.

  Restrictions can appear for any of the following architectures:

  - IA-32 architecture
  - Intel® 64 architecture

  Certain operating systems are not available on all the above architectures. For the latest information, check your Release Notes.
- Syntax

This section shows the syntax on Linux* and macOS systems and the syntax on Windows* systems. If the option is not valid on a particular operating system, it will specify "None".

- Arguments

  This section shows any arguments (parameters) that are related to the option. If the option has no arguments, it will specify "None".
- Default

  This section shows the default setting for the option.
- Description

  This section shows the full description of the option. It may also include further information on any applicable arguments.
- IDE Equivalent

  This section shows information related to the Intel® Integrated Development Environment (Intel® IDE) Property Pages on Linux*, macOS, and Windows* systems. It shows on which Property Page the option appears, and under what category it's listed. The Windows* IDE is Microsoft* Visual Studio* .NET; the Linux* IDE is Eclipse*; the macOS IDE is Xcode*. If the option has no IDE equivalent, it will specify "None".
- Alternate Options

  This section lists any options that are synonyms for the described option. If there are no alternate option names, it will show "None".

  Some alternate option names are deprecated and may be removed in future releases.

  Many options have an older spelling where underscores ("_") instead of hyphens ("-") connect the main option names. The older spelling is a valid alternate option name.

Some option descriptions may also have the following:

- Example (or Examples)

  This section shows one or more examples that demonstrate the option.
- See Also

  This section shows where you can get further information on the option or it shows related options.

Optimization Options

This section contains descriptions for compiler options that pertain to optimization.

falias, Oa

Determines whether aliasing is assumed in a program.

Syntax

Linux OS and macOS:

`-falias`

`-fno-alias`

Windows OS:

`/Oa`

`/Oa-`

Arguments

None

Default

`-falias` or `/Oa-`	On Linux* and macOS, aliasing is assumed in the program. On Windows*, aliasing is not assumed in a program.

Description

This option determines whether aliasing is assumed in a program.

If you specify `-fno-alias` or `/Oa`, aliasing is not assumed in a program.

If you specify `-falias` or `/Oa-`, aliasing is assumed in a program. However, this may affect performance.

IDE Equivalent

Visual Studio: None

Eclipse: **Data > Assume No Aliasing in Program**

Xcode: **Data > Assume No Aliasing in Program**

Alternate Options

None

See Also

`ffnalias` compiler option

fast

Maximizes speed across the entire program.

Syntax

Linux OS:

`-fast`

macOS:

`-fast`

Windows OS:

`/fast`

Arguments

None

Default

OFF	The optimizations that maximize speed are not enabled.

Description

This option maximizes speed across the entire program.

It sets the following options:

- On macOS systems: `-ipo`, `-mdynamic-no-pic`,`-O3`, `-no-prec-div`,`-fp-model fast=2`, and `-xHost`
- On Windows* systems: `/O3`, `/Qipo`, `/Qprec-div-`, `/fp:fast=2`, and `/QxHost`
- On Linux* systems: `-ipo`, `-O3`, `-no-prec-div`,`-static`, `-fp-model fast=2`, and `-xHost`

When option `fast` is specified, you can override the `[Q]xHost` option setting by specifying a different processor-specific `[Q]x` option on the command line. However, the last option specified on the command line takes precedence.

For example:

- On Linux* systems, if you specify option `-fast -xSSE3`, option `-xSSE3` takes effect. However, if you specify `-xSSE3 -fast`, option `-xHost` takes effect.
- On Windows* systems, if you specify option `/fast /QxSSE3`, option`/QxSSE3` takes effect. However, if you specify `/QxSSE3 /fast`, option `/QxHost` takes effect.

For implications on non-Intel processors, refer to the `[Q]xHost` documentation.

NOTE
Option `fast` sets some aggressive optimizations that may not be appropriate for all applications. The resulting executable may not run on processor types different from the one on which you compile. You should make sure that you understand the individual optimization options that are enabled by option `fast`.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

fp-model, fp compiler option

xHost, QxHost
compiler option

x, Qx
compiler option

fbuiltin, Oi

Enables or disables inline expansion of intrinsic functions.

Syntax

Linux OS:

`-fbuiltin[-name]`

`-fno-builtin[-name]`

macOS:

`-fbuiltin[-name]`

`-fno-builtin[-name]`

Windows OS:

`/Oi[-]`

`/Qno-builtin-name`

Arguments

name	Is a list of one or more intrinsic functions. If there is more than one intrinsic function, they must be separated by commas.

Default

ON	Inline expansion of intrinsic functions is enabled.

Description

This option enables or disables inline expansion of one or more intrinsic functions.

If `-fno-builtin-`*name* or `/Qno-builtin-`*name* is specified, inline expansion is disabled for the named functions. If *name* is not specified, `-fno-builtin` or `/Oi-` disables inline expansion for all intrinsic functions.

For a list of built-in functions affected by `-fbuiltin`, search for "built-in functions" in the appropriate gcc* documentation.

For a list of built-in functions affected by `/Oi`, search for "/Oi" in the appropriate Microsoft* Visual C/C++* documentation.

IDE Equivalent

Visual Studio: **Optimization > Enable Intrinsic Functions** (/Oi)

Eclipse: None

Xcode: None

Alternate Options

None

fdefer-pop

Determines whether the compiler always pops the arguments to each function call as soon as that function returns.

Syntax

Linux OS and macOS:

`-fdefer-pop`

`-fno-defer-pop`

Windows OS:

`None`

Arguments

None

Default

`-fdefer-pop`	The compiler uses default optimizations that may result in deferred clearance of the stack arguments.

Description

This option determines whether the compiler always pops the arguments to each function call as soon as that function returns.

If you want the compiler to always pop the arguments to each function call as soon as that function returns, specify `-fno-defer-pop`.

For processors that must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

IDE Equivalent

None

Alternate Options

None

ffnalias, Ow

Determines whether aliasing is assumed within functions.

Syntax

Linux OS and macOS:

`-ffnalias`

`-fno-fnalias`

Windows OS:

`/Ow`

`/Ow-`

Arguments

None

Default

`-ffnalias` or `/Ow`	Aliasing is assumed within functions.

Description

This option determines whether aliasing is assumed within functions.

If you specify `-fno-fnalias` or `/Ow-`, aliasing is not assumed within functions, but it is assumed across calls.

If you specify `-ffnalias` or `/Ow`, aliasing is assumed within functions.

IDE Equivalent

None

Alternate Options

None

See Also

falias compiler option

foptimize-sibling-calls

Determines whether the compiler optimizes tail recursive calls.

Syntax

Linux OS:

`-foptimize-sibling-calls`

`-fno-optimize-sibling-calls`

macOS:

`-foptimize-sibling-calls`

`-fno-optimize-sibling-calls`

Windows OS:

None

Arguments

None

Default

`-foptimize-sibling-calls`	The compiler optimizes tail recursive calls.

Description

This option determines whether the compiler optimizes tail recursive calls. It enables conversion of tail recursion into loops.

If you do not want to optimize tail recursive calls, specify `-fno-optimize-sibling-calls`.

Tail recursion is a special form of recursion that doesn't use stack space. In tail recursion, a recursive call is converted to a GOTO statement that returns to the beginning of the function. In this case, the return value of the recursive call is only used to be returned. It is not used in another expression. The recursive function is converted into a loop, which prevents modification of the stack space used.

IDE Equivalent

None

Alternate Options

None

fprotect-parens, Qprotect-parens

Determines whether the optimizer honors parentheses when floating-point expressions are evaluated.

Syntax

Linux OS and macOS:

`-fprotect-parens`

`-fno-protect-parens`

Windows OS:

`/Qprotect-parens`

`/Qprotect-parens-`

Arguments

None

Default

`-fno-protect-parens` or `/Qprotect-parens-`	Parentheses are ignored when determining the order of expression evaluation.

Description

This option determines whether the optimizer honors parentheses when determining the order of floating-point expression evaluation.

When option `-fprotect-parens` (Linux* and macOS) or `/Qprotect-parens` (Windows*) is specified, the optimizer will maintain the order of evaluation imposed by parentheses in the code.

When option `-fno-protect-parens` (Linux* and macOS) or `/Qprotect-parens-` (Windows*) is specified, the optimizer may reorder floating-point expressions without regard for parentheses if it produces faster executing code.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following expression:

```
A+(B+C)
```

By default, the parentheses are ignored and the compiler is free to re-order the floating-point operations based on the optimization level, the setting of option `-fp-model` (Linux* and macOS) or `/fp` (Windows*), etc. to produce faster code. Code that is sensitive to the order of operations may produce different results (such as with some floating-point computations).

However, if `-fprotect-parens` (Linux* and macOS) or `/Qprotect-parens` (Windows*) is specified, parentheses around floating-point expressions (including complex floating-point and decimal floating-point) are honored and the expression will be interpreted following the normal precedence rules, that is, B+C will be computed first and then added to A.

This may produce slower code than when parentheses are ignored. If floating-point sensitivity is a specific concern, you should use option `-fp-model precise` (Linux* and macOS) or `/fp:precise` (Windows*) to ensure precision because it controls all optimizations that may affect precision.

See Also

`fp-model, fp` compiler option

GF

Enables read-only string-pooling optimization.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/GF`

Arguments

None

Default

OFF	Read/write string-pooling optimization is enabled.

Description

This option enables read only string-pooling optimization.

IDE Equivalent

Visual Studio: **Code Generation > Enable String Pooling**

Eclipse: None

Xcode: None

Alternate Options

None

nolib-inline

Disables inline expansion of standard library or intrinsic functions.

Syntax

Linux OS:

`-nolib-inline`

macOS:

`-nolib-inline`

Windows OS:

None

Arguments

None

Default

OFF	The compiler inlines many standard library and intrinsic functions.

Description

This option disables inline expansion of standard library or intrinsic functions. It prevents the unexpected results that can arise from inline expansion of these functions.

IDE Equivalent

Visual Studio: None

Eclipse: **Optimization > Disable Intrinsic Inline Expansion**

Xcode: **Optimization > Disable Intrinsic Inline Expansion**

Alternate Options

None

O

Specifies the code optimization for applications.

Syntax

Linux OS:

`-O[`*n*`]`

macOS:

`-O[`*n*`]`

Windows OS:

`/O[`*n*`]`

Arguments

n	Is the optimization level. Possible values are 1, 2, or 3. On Linux* and macOS systems, you can also specify 0.

Default

O2	Optimizes for code speed. This default may change depending on which other compiler options are specified. For details, see below.

Description

This option specifies the code optimization for applications.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td><code>O</code> (Linux* and macOS)</td><td>This is the same as specifying <code>O2</code>.</td></tr>
<tr><td><code>O0</code> (Linux and macOS)</td><td>Disables all optimizations.

This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.</td></tr>
<tr><td><code>O1</code></td><td>Enables optimizations for speed and disables some optimizations that increase code size and affect speed.
To limit code size, this option:

Enables global optimization; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling.
Disables inlining of some intrinsics.

This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.

The <code>O1</code> option may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.</td></tr>
<tr><td><code>O2</code></td><td>Enables optimizations for speed. This is the generally recommended optimization level.
Vectorization is enabled at <code>O2</code> and higher levels.

On systems using IA-32 architecture: Some basic loop optimizations such as Distribution, Predicate Opt, Interchange, multi-versioning, and scalar replacements are performed.

This option also enables:

Inlining of intrinsics
Intra-file interprocedural optimization, which includes:

inlining
constant propagation
forward substitution
routine attribute propagation
variable address-taken analysis
dead static function elimination
removal of unreferenced variables

The following capabilities for performance gain:

constant propagation
copy propagation
dead-code elimination
global register allocation
global instruction scheduling and control speculation
loop unrolling
optimized code selection
partial redundancy elimination
strength reduction/induction variable simplification
variable renaming

</td></tr>
</table>

Option	Description
	• exception handling optimizations • tail recursions • peephole optimizations • structure assignment lowering and optimizations • dead store elimination This option may set other options, especially options that optimize for code speed. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. On Linux systems, the `-debug inline-debug-info` option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`). Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.
`O3`	Performs `O2` optimizations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. When `O3` is used with options `-ax` or `-x` (Linux) or with options `/Qax` or `/Qx` (Windows), the compiler performs more aggressive data dependency analysis than for `O2`, which may result in longer compilation times. The `O3` optimizations may not cause higher performance unless loop and memory access transformations take place. The optimizations may slow down code in some cases compared to `O2` optimizations. The `O3` option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

The last `O` option specified on the command line takes precedence over any others.

IDE Equivalent

Visual Studio: **Optimization > Optimization**

Eclipse: **General > Optimization Level**

Xcode: **General > Optimization Level**

Alternate Options

`O0`	Linux: None Windows: `/Od`

See Also

`Od` compiler option

Od

Disables all optimizations.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/Od

Arguments

None

Default

OFF	The compiler performs default optimizations.

Description

This option disables all optimizations. It can be used for selective optimizations, such as a combination of /Od and /Ob1 (disables all optimizations, but enables inlining).

On IA-32 architecture, this option sets the /Oy- option.

IDE Equivalent

Visual Studio: **Optimization > Optimization**

Eclipse: None

Xcode: None

Alternate Options

Linux and macOS: -O0

Windows: None

See Also

O compiler option (see O0)

Ofast

Sets certain aggressive options to improve the speed of your application.

Syntax

Linux OS:

-Ofast

macOS:

-Ofast

Windows OS:

`None`

Arguments

None

Default

OFF	The aggressive optimizations that improve speed are not enabled.

Description

This option improves the speed of your application.

It sets compiler options `-O3`, `-no-prec-div`, and -fp-model fast=2.

On Linux* systems, this option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

See Also

`O` compiler option

`prec-div, Qprec-div` compiler option

`fast` compiler option

`fp-model, fp` compiler option

Os

Enables optimizations that do not increase code size; it produces smaller code size than O2.

Syntax

Linux OS:

`-Os`

macOS:

`-Os`

Windows OS:

`/Os`

Arguments

None

Default

OFF	Optimizations are made for code speed. However, if `O1` is specified, `Os` is the default.

Description

This option enables optimizations that do not increase code size; it produces smaller code size than `O2`. It disables some optimizations that increase code size for a small speed benefit.

This option tells the compiler to favor transformations that reduce code size over transformations that produce maximum performance.

IDE Equivalent

Visual Studio: **Optimization > Favor Size or Speed**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`O` compiler option
`Ot` compiler option

Ot

Enables all speed optimizations.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Ot`

Arguments

None

Default

`/Ot`	Optimizations are made for code speed. If `Od` is specified, all optimizations are disabled. If `O1` is specified, `Os` is the default.

Description

This option enables all speed optimizations.

IDE Equivalent

Visual Studio: **Optimization > Favor Size or Speed**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`O` compiler option
`Os` compiler option

Ox

Enables maximum optimizations.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Ox`

Arguments

None

Default

OFF	The compiler does not enable optimizations.

Description

The compiler enables maximum optimizations by combining the following options:

- /Ob2
- /Oy
- /Ot
- /Oi

IDE Equivalent

Visual Studio: **Optimization > Optimization**

Eclipse: None

Xcode: None

Alternate Options

None

Code Generation Options

This section contains descriptions for compiler options that pertain to code generation.

arch

Tells the compiler which features it may target, including which instruction sets it may generate.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/arch:*code*

Arguments

code — Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

ALDERLAKE AMBERLAKE BROADWELL CANNONLAKE CASCADELAKE COFFEELAKE COOPERLAKE GOLDMONT GOLDMONT-PLUS HASWELL ICELAKE-CLIENT (or ICELAKE) ICELAKE-SERVER IVYBRIDGE KABYLAKE KNL KNM ROCKETLAKE SANDYBRIDGE SAPPHIRERAPIDS SILVERMONT SKYLAKE SKYLAKE-AVX512 TIGERLAKE TREMONT WHISKEYLAKE	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name. Keyword ICELAKE is deprecated and may be removed in a future release.
CORE-AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.

`CORE-AVX-I`	May generate Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`AVX2`	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`AVX`	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`SSE4.2`	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`SSE4.1`	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`SSSE3`	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`SSE3`	May generate Intel® SSE3, SSE2, and SSE instructions.
`SSE2`	May generate Intel® SSE2 and SSE instructions.
`SSE`	This option has been deprecated; it is now the same as specifying `IA32`.
`IA32`	Generates x86/x87 generic code that is compatible with IA-32 architecture. Disables any default extended instruction settings, and any previously set extended instruction settings. It also disables all feature-specific optimizations and instructions. This value is only available on IA-32 architecture.

Default

SSE2	The compiler may generate Intel® SSE2 and SSE instructions.

Description

This option tells the compiler which features it may target, including which instruction sets it may generate.

Code generated with these options should execute on any compatible, non-Intel processor with support for the corresponding instruction set.

Options `/arch` and `/Qx` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

If you specify both the `/Qax` and `/arch` options, the compiler will not generate Intel-specific instructions.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Enable Enhanced Instruction Set**

Eclipse

Eclipse: None

Xcode

Xcode: None

Alternate Options

None

See Also

`x, Qx` compiler option

`xHost, QxHost` compiler option

`ax, Qax` compiler option

`arch` compiler option

`march` compiler option

`m` compiler option

`m32, m64` compiler option

ax, Qax

Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit.

Syntax

Linux OS:

`-ax`*code*

macOS:

`-ax`*code*

Windows OS:

`/Qax`*code*

Arguments

code Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. The following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`ALDERLAKE`
`AMBERLAKE`
`BROADWELL`
`CANNONLAKE`
`CASCADELAKE`
`COFFEELAKE`
`COOPERLAKE`
`GOLDMONT`
`GOLDMONT-PLUS`
`HASWELL`
`ICELAKE-CLIENT` (or `ICELAKE`)
	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name.
	
	Keywords `KNL` and `SILVERMONT` are only available on Windows* and Linux* systems.
	
	Keyword `ICELAKE` is deprecated and may be removed in a future release.

`ICELAKE-SERVER`
`IVYBRIDGE`
`KABYLAKE`
`KNL`
`KNM`
`ROCKETLAKE`
`SANDYBRIDGE`
`SAPPHIRERAPIDS`
`SILVERMONT`
`SKYLAKE`
`SKYLAKE-AVX512`
`TIGERLAKE`
`TREMONT`
`WHISKEYLAKE`	

`COMMON-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), as well as the instructions enabled with CORE-AVX2.
`CORE-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), Intel® AVX-512 Doubleword and Quadword Instructions (DQI), Intel® AVX-512 Byte and Word Instructions (BWI) and Intel® AVX-512 Vector Length extensions, as well as the instructions enabled with CORE-AVX2.
`CORE-AVX2`	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`CORE-AVX-I`	May generate Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`AVX`	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`SSE4.2`	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel processors.
`SSE4.1`	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`SSSE3`	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions for Intel® processors. For macOS systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
`SSE3`	May generate Intel® SSE3, SSE2, and SSE instructions for Intel® processors. This value is not available on macOS systems.

`SSE2`	May generate Intel® SSE2 and SSE instructions for Intel® processors. This value is not available on macOS systems.

You can specify more than one *code* value. When specifying more than one *code* value, each value must be separated with a comma. See the Examples section below.

Default

OFF	No auto-dispatch code is generated. Feature-specific code is generated and is controlled by the setting of the following compiler options: • Linux*: `-m` and `-x` • Windows*: `/arch` and `/Qx` • macOS: `-x`

Description

This option tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit. It also generates a baseline code path. The Intel feature-specific auto-dispatch path is usually more optimized than the baseline path. Other options, such as `O3`, control how much optimization is performed on the baseline path.

The baseline code path is determined by the architecture specified by options `-m` or `-x` (Linux* and macOS) or options `/arch` or `/Qx` (Windows*). While there are defaults for the `[Q]x` option that depend on the operating system being used, you can specify an architecture and optimization level for the baseline code that is higher or lower than the default. The specified architecture becomes the effective minimum architecture for the baseline code path.

If you specify both the `[Q]ax` and `[Q]x` options, the baseline code will only execute on Intel® processors compatible with the setting specified for the `[Q]x`.

If you specify both the `-ax` and `-m` options (Linux and macOS) or the `/Qax` and `/arch` options (Windows), the baseline code will execute on non-Intel processors compatible with the setting specified for the `-m` or `/arch` option.

If you specify both the `-ax` and `-march` options (Linux and macOS), or the `/Qax` and `/arch` options (Windows), the compiler will not generate Intel-specific instructions. This is because specifying `-march` disables `-ax` and specifying `/arch` disables `/Qax`.

The `[Q]ax` option tells the compiler to find opportunities to generate separate versions of functions that take advantage of features of the specified instruction features.

If the compiler finds such an opportunity, it first checks whether generating a feature-specific version of a function is likely to result in a performance gain. If this is the case, the compiler generates both a feature-specific version of a function and a baseline version of the function. At run time, one of the versions is chosen to execute, depending on the Intel® processor in use. In this way, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older processors and non-Intel processors. A non-Intel processor always executes the baseline code path.

You can use more than one of the feature values by combining them. For example, you can specify `-axSSE4.1,SSSE3` (Linux and macOS) or `/QaxSSE4.1,SSSE3` (Windows). You cannot combine the old style, deprecated options and the new options. For example, you cannot specify `-axSSE4.1,T` (Linux and macOS) or `/QaxSSE4.1,T` (Windows).

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Product and Performance Information
Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Add Processor-Optimized Code Path**

Eclipse

Eclipse: **Code Generation > Add Processor-Optimized Code Path**

Xcode

Xcode: **Code Generation > Add Processor-Optimized Code Path**

Alternate Options

None

Examples

The following shows an example of how to specify this option:

```
icpc -axSKYLAKE file.cpp        ! Linux* and macOSsystems
icl /QaxSKYLAKE file.cpp        ! Windows* systems
```

The following shows an example of how to specify more than one *code* value:

```
icpc -axSKYLAKE,BROADWELL file.cpp      ! Linux* and macOSsystems
icl /QaxBROADWELL,SKYLAKE file.cpp      ! Windows* systems
```

Note that the comma-separated list must have no spaces between the names.

See Also

`x, Qx` compiler option
`xHost, QxHost` compiler option
`march` compiler option
`arch` compiler option
`m` compiler option

EH

Specifies the model of exception handling to be performed.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/EH*type*

/EH*type*-

Arguments

type	Specifies the exception handling model. Possible values are:	
	`a`	Specifies the asynchronous C++ exception handling model.
	`s`	Specifies the synchronous C++ exception handling model.
	`c`	Tells the compiler to assume that extern "C" functions do not throw exceptions.
	`r`	Tells the compiler to always generate runtime termination checks for all noexcept functions. IT forces runtime termination checks in all functions that have a noexcept attribute.

If you specify `c`, you must also specify `a` or `s`.

Default

OFF	Some exception handling is performed by default.

Description

This option specifies the model of exception handling to be performed.

If you specify the negative form of the option, it disables the exception handling performed by *type* or the last *type* if there are two. For example, if you specify `/EHsc-`, it is interpreted as `/EHs`.

For more details about option `/EH`, see the Microsoft documentation.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Enable C++ Exceptions**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

`/EHsc`	Linux and macOS: None Windows: `/GX`

See Also

`Qsafeseh` compiler option

fasynchronous-unwind-tables

Determines whether unwind information is precise at an instruction boundary or at a call boundary.

Syntax

Linux OS:

`-fasynchronous-unwind-tables`

`-fno-asynchronous-unwind-tables`

macOS:

`-fasynchronous-unwind-tables`

`-fno-asynchronous-unwind-tables`

Windows OS:

`None`

Arguments

None

Default

Intel® 64 architecture: `-fasynchronous-unwind-tables`	The unwind table generated is precise at an instruction boundary, enabling accurate unwinding at any instruction.
IA-32 architecture (Linux* only): `-fno-asynchronous-unwind-tables`	The unwind table generated is precise at call boundaries only.

Description

This option determines whether unwind information is precise at an instruction boundary or at a call boundary. The compiler generates an unwind table in DWARF2 or DWARF3 format, depending on which format is supported on your system.

If `-fno-asynchronous-unwind-tables` is specified, the unwind table is precise at call boundaries only. In this case, the compiler will avoid creating unwind tables for routines such as the following:

- A C++ routine that does not declare objects with destructors and does not contain calls to routines that might throw an exception.
- A C/C++ or Fortran routine compiled without `-fexceptions`, and on Intel® 64 architecture, without `-traceback`.
- A C/C++ or Fortran routine compiled with `-fexceptions` that does not contain calls to routines that might throw an exception.

IDE Equivalent

None

Alternate Options

None

See Also

fexceptions compiler option

fcf-protection, Qcf-protection

Enables Intel® Control-Flow Enforcement Technology (Intel® CET) protection, which defends your program from certain attacks that exploit vulnerabilities. This option offers preliminary support for Intel® CET.

Syntax

Linux OS:

`-fcf-protection[=keyword]`

macOS:

None

Windows OS:

`/Qcf-protection[:keyword]`

Arguments

keyword	Specifies the level of protection the compiler should perform. Possible values are:	
	`shadow_stack`	Enables shadow stack protection.
	`branch_tracking`	Enables endbranch (EB) generation.
	`full`	Enables both shadow stack protection and EB generation. This is the same as specifying this compiler option with no *keyword*.
	`none`	Disables Intel® CET protection.

Default

`-fcf-protection=none` or `/Qcf-protection:none`	No Control-flow Enforcement protection is performed.

Description

This option enables Intel® CET protection, which defends your program from certain attacks that exploit vulnerabilities.

Intel® CET protections are enforced on processors that support Intel® CET. They are ignored on processors that do not support Intel® CET, so they are safe to use in programs that might run on a variety of processors.

Specifying `shadow_stack` helps to protect your program from return-oriented programming (ROP). Return-oriented programming (ROP) is a technique to exploit computer security defenses such as non-executable memory and code signing by gaining control of the call stack to modify program control flow and then execute certain machine instruction sequences.

Specifying `branch_tracking` helps to protect your program from call/jump-oriented programming (COP/JOP). Jump-oriented programming (JOP) is a variant of ROP that uses indirect jumps and calls to emulate return instructions. Call-oriented programming (COP) is a variant of ROP that employs indirect calls.

To get both protections, specify this compiler option with no *keyword*, or specify `-fcf-protection=full` (Linux*) or `/Qcf-protection:full` (Windows*).

IDE Equivalent

None

Alternate Options

Linux and macOS: `-qcf-protection`

Windows: None

fdata-sections, Gw

Places each data item in its own COMDAT section.

Syntax

Linux OS:

`-fdata-sections`

macOS:

`-fdata-sections`

Windows OS:

`/Gw`

Arguments

None

Default

OFF	The compiler does not separate functions into COMDATs.

Description

This option places each data item in its own COMDAT section.

When using this compiler option, you can add the linker option `-Wl,--gc-sections` (LInux) or `/link /OPT:REF` (Windows), which will remove all unused code.

> **NOTE**
> When you put each data item in its own section, it enables the linker to reorder the sections for other possible optimization.

Alternate Options

None

See Also

`ffunction-sections, Gy` compiler option

fexceptions

Enables exception handling table generation.

Syntax

Linux OS:

`-fexceptions`

`-fno-exceptions`

macOS:

`-fexceptions`

`-fno-exceptions`

Windows OS:

None

Arguments

None

Default

-fexceptions	Exception handling table generation is enabled. Default for C++.
-fno-exceptions	Exception handling table generation is disabled. Default for C.

Description

This option enables exception handling table generation. The `-fno-exceptions` option disables exception handling table generation, resulting in smaller code. When this option is used, any use of exception handling constructs (such as try blocks and throw statements) will produce an error. Exception specifications are parsed but ignored. It also undefines the preprocessor symbol __EXCEPTIONS.

IDE Equivalent

None

Alternate Options

None

ffunction-sections, Gy

Places each function in its own COMDAT section.

Syntax

Linux OS:

`-ffunction-sections`

macOS:

`-ffunction-sections`

Windows OS:

`/Gy`

Arguments

None

Default

OFF	The compiler does not separate functions into COMDATs.

Description

This option places each function in its own COMDAT section.

When using this compiler option, you can add the linker option `-Wl,--gc-sections` (LInux) or `/link /OPT:REF` (Windows), which will remove all unused code.

> **NOTE**
> When you put each function in its own section, it enables the linker to reorder the sections for other possible optimization.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Enable Function-Level Linking**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

`fdata-sections, Gw` compiler option

fomit-frame-pointer, Oy

Determines whether EBP is used as a general-purpose register in optimizations.

Architecture Restrictions

Option `/Oy[-]` is only available on IA-32 architecture

Syntax

Linux OS:

`-fomit-frame-pointer`

`-fno-omit-frame-pointer`

macOS:

`-fomit-frame-pointer`

`-fno-omit-frame-pointer`

Windows OS:

`/Oy`

/Oy-

Arguments

None

Default

-fomit-frame-pointer or /Oy	EBP is used as a general-purpose register in optimizations. However, on Linux* and macOS systems, the default is -fno-omit-frame-pointer if option -O0 or -g is specified. On Windows* systems, the default is /Oy- if option /Od is specified.

Description

These options determine whether EBP is used as a general-purpose register in optimizations. Option -fomit-frame-pointer and option /Oy allows this use. Option -fno-omit-frame-pointer and option /Oy- disallows it.

Some debuggers expect EBP to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The -fno-omit-frame-pointer and the /Oy- option directs the compiler to generate code that maintains and uses EBP as a stack frame pointer for all functions so that a debugger can still produce a stack backtrace without doing the following:

- For -fno-omit-frame-pointer: turning off optimizations with -O0
- For /Oy-: turning off /O1, /O2, or /O3 optimizations

The -fno-omit-frame-pointer option is set when you specify option -O0 or the -g option. The -fomit-frame-pointer option is set when you specify option -O1, -O2, or -O3.

The /Oy option is set when you specify the /O1, /O2, or /O3 option. Option /Oy- is set when you specify the /Od option.

Using the -fno-omit-frame-pointer or /Oy- option reduces the number of available general-purpose registers by 1, and can result in slightly less efficient code.

NOTE
For Linux* systems:

There is currently an issue with GCC 3.2 exception handling. Therefore, the compiler ignores this option when GCC 3.2 is installed for C++ and exception handling is turned on (the default).

IDE Equivalent

Visual Studio: **Optimization > Omit Frame Pointers**

Eclipse: **Optimization > Provide Frame Pointer**

Xcode: **Optimization > Provide Frame Pointer**

Alternate Options

Linux and macOS: -fp (this is a deprecated option)

Windows: None

See Also

momit-leaf-frame-pointer compiler option

Gd

Makes __cdecl the default calling convention.

Architecture Restrictions

Not available on IA-32 architecture.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Gd`

Arguments

None

Default

ON	The default calling convention is `__cdecl`.

Description

This option makes `__cdecl` the default calling convention.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

C C++ Calling Conventions

Gr

Makes __fastcall the default calling convention.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Gr`

Arguments

None

Default

OFF	The default calling convention is `__cdecl`

Description

This option makes `__fastcall` the default calling convention.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

C C++ Calling Conventions

GR

Enables or disables C++ Run Time Type Information (RTTI).

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/GR`

/GR-

Arguments

None

Default

/GR	C++ Run Time Type Information (RTTI) is enabled.

Description

This option enables or disables C++ Run Time Type Information (RTTI).

To disable C++ Run Time Type Information (RTTI), specify option /GR-.

IDE Equivalent

Windows

Visual Studio: **Language > Enable Run-Time Type Information**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

guard

Enables the control flow protection mechanism.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/guard:*keyword*

Arguments

keyword Specifies the the control flow protection mechanism. Possible values are:

cf[-] Tells the compiler to analyze control flow of valid targets for indirect calls and to insert code to verify the targets at runtime.

To explicitly disable this option, specify /guard:cf-.

Default

`OFF`	The control flow protection mechanism is disabled.

Description

This option enables the control flow protection mechanism. It tells the compiler to analyze control flow of valid targets for indirect calls and inserts a call to a checking routine before each indirect call to verify the target of the given indirect call.

The `/guard:cf` option must be passed to both the compiler and linker.

Code compiled using `/guard:cf` can be linked to libraries and object files that are not compiled using the option.

This option has been added for Microsoft compatibility. It uses the Microsoft implementation.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Control Flow Guard**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

Gv

Tells the compiler to use the vector calling convention (__vectorcall) when passing vector type arguments.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Gv`

Arguments

None

Default

OFF	The default calling convention is `__cdecl`.

Description

This option tells the compiler to use the vector calling convention (__vectorcall) when passing vector type arguments.

It causes each function in the module to compile as __vectorcall unless the function is declared with a conflicting attribute, or the name of the function is main.

This option has been added for Microsoft compatibility.

For more details about the __vectorcall calling convention, see the Microsoft documentation.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

C C++ Calling Conventions

Gz

Makes __stdcall the default calling convention.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Gz`

Arguments

None

Default

OFF	The default calling convention is `__cdecl`.

Description

This option makes `__stdcall` the default calling convention.

IDE Equivalent

Windows

Visual Studio: **Advanced > Calling Convention**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

C C++ Calling Conventions

hotpatch

Tells the compiler to prepare a routine for hotpatching.

Syntax

Linux OS:

`-hotpatch[=n]`

macOS:

`None`

Windows OS:

`/hotpatch[:n]`

Arguments

n	An integer specifying the number of bytes the compiler should add before the function entry point.

Default

OFF	The compiler does not prepare routines for hotpatching.

Description

This option tells the compiler to prepare a routine for hotpatching. The compiler inserts nop padding around function entry points so that the resulting image is hot patchable.

Specifically, the compiler adds nop bytes after each function entry point and enough nop bytes before the function entry point to fit a direct jump instruction on the target architecture.

If *n* is specified, it overrides the default number of bytes that the compiler adds before the function entry point.

IDE Equivalent

None

Alternate Options

None

m

Tells the compiler which features it may target, including which instruction sets it may generate.

Syntax

Linux OS and macOS:

`-m`*`code`*

Windows OS:

`None`

Arguments

code	Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`avx`	May generate Intel® Advanced Vector Extensions (Intel® AVX), SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`sse4.2`	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`sse4.1`	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`ssse3`	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`sse3`	May generate Intel® SSE3, SSE2, and SSE instructions.
`sse2`	May generate Intel® SSE2 and SSE instructions. This value is only available on Linux systems.
`sse`	This option has been deprecated; it is now the same as specifying `ia32`.
`ia32`	Generates x86/x87 generic code that is compatible with IA-32 architecture. Disables any default extended instruction settings, and any previously set extended instruction settings. It also disables all feature-specific

optimizations and instructions. This value is only available on Linux* systems using IA-32 architecture.

This compiler option also supports many of the `-m` option settings available with gcc. For more information on gcc `-m` settings, see the gcc documentation.

Default

Linux* systems: `-msse2` macOS systems: `-mssse3`	For more information on the default values, see Arguments above.

Description

This option tells the compiler which features it may target, including which instruction sets it may generate.

Code generated with these options should execute on any compatible, non-Intel processor with support for the corresponding instruction set.

Options `-x` and `-m` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

Linux* systems: For compatibility with gcc, the compiler allows the following options but they have no effect. You will get a warning error, but the instructions associated with the name will not be generated. You should use the suggested replacement options.

gcc Compatibility Option (Linux*)	Suggested Replacement Option
`-mfma`	`-march=core-avx2`
`-mbmi`, `-mavx2`, `-mlzcnt`	`-march=core-avx2`
`-mmovbe`	`-march=atom -minstruction=movbe`
`-mcrc32`, `-maes`, `-mpclmul`, `-mpopcnt`	`-march=corei7`
`-mvzeroupper`	`-march=corei7-avx`
`-mfsgsbase`, `-mrdrnd`, `-mf16c`	`-march=core-avx-i`

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/arch`

See Also

`x, Qx` compiler option

`xHost, QxHost` compiler option

`ax, Qax` compiler option

`arch` compiler option
`march` compiler option
`m32, m64` compiler option

m32, m64, Qm32, Qm64

Tells the compiler to generate code for a specific architecture.

Syntax

Linux OS:

`-m32`

`-m64`

macOS:

`-m64`

Windows OS:

`/Qm32`

`/Qm64`

Arguments

None

Default

OFF	The compiler's behavior depends on the host system.

Description

These options tell the compiler to generate code for a specific architecture.

Option	Description
`-m32` or `/Qm32`	Tells the compiler to generate code for IA-32 architecture.
`-m64` or `/Qm64`	Tells the compiler to generate code for Intel® 64 architecture.

The `-m64` option is the same as macOS option `-arch x86_64`. This option is not related to the Intel®C++ Compiler option `arch`.

On Linux* systems, these options are provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

m80387

Specifies whether the compiler can use x87 instructions.

Syntax

Linux OS:

`-m80387`

`-mno-80387`

macOS:

`-m80387`

`-mno-80387`

Windows OS:

`None`

Arguments

None

Default

`-m80387`	The compiler may use x87 instructions.

Description

This option specifies whether the compiler can use x87 instructions.

If you specify option `-mno-80387`, it prevents the compiler from using x87 instructions. If the compiler is forced to generate x87 instructions, it issues an error message.

IDE Equivalent

None

Alternate Options

-m[no-]x87

march

Tells the compiler to generate code for processors that support certain features.

Syntax

Linux OS:

`-march=`*`processor`*

macOS:

`-march=`*`processor`*

Windows OS:

`None`

Arguments

processor	Tells the compiler the code it can generate. Possible values are:

`alderlake`
`amberlake`
`broadwell`
`cannonlake`
`cascadelake`
`coffeelake`
`cooperlake`
`goldmont`
`goldmont-plus`
`haswell`
`icelake-client` (or `icelake`)
`icelake-server`
`ivybridge`
`kabylake`
`knl`
`knm`
`rocketlake`
`sandybridge`
`sapphirerapids`
`silvermont`
`skylake`
`skylake-avx512`
`tigerlake`
`tremont`
`whiskeylake`
	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name.
	Keywords `knl` and `silvermont` are only available on Linux* systems.
	Keyword `icelake` is deprecated and may be removed in a future release.

`core-avx2`	Generates code for processors that support Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`core-avx-i`	Generates code for processors that support Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7-avx`	Generates code for processors that support Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7`	Generates code for processors that support Intel® SSE4 Efficient Accelerated String and Text Processing instructions. May also generate code for Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSE3, SSE2, SSE, and SSSE3 instructions.

`atom`	Generates code for processors that support MOVBE instructions, depending on the setting of option `-minstruction` (Linux* and macOS) or `/Qinstruction` (Windows*). May also generate code for SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`core2`	Generates code for the Intel® Core™2 processor family.
`pentium4m`	Generates for Intel® Pentium® 4 processors with MMX technology.

`pentium-m`
`pentium4`
`pentium3`
`pentium`
	Generates code for Intel® Pentium® processors. Value `pentium3` is only available on Linux* systems.

Default

`pentium4`	If no architecture option is specified, value `pentium4` is used by the compiler to generate code.

Description

This option tells the compiler to generate code for processors that support certain features.

If you specify both the `-ax` and `-march` options, the compiler will not generate Intel-specific instructions.

Specifying `-march=pentium4` sets `-mtune=pentium4`.

For compatibility, a number of historical *processor* values are also supported, but the generated code will not differ from the default.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

`-march=pentium3`	Linux: `-xSSE` macOS: None Windows: None
`-march=pentium4` `-march=pentium-m`	Linux: `-xSSE2` macOS: None Windows: None
`-march=core2`	Linux: `-xSSSE3` macOS: None

Windows: None

See Also

`xHost, QxHost` compiler option

`x, Qx` compiler option

`ax, Qax` compiler option

`arch` compiler option

`minstruction, Qinstruction` compiler option

`m` compiler option

masm

Tells the compiler to generate the assembler output file using a selected dialect.

Syntax

Linux OS:

`-masm=`*`dialect`*

macOS:

None

Windows OS:

None

Arguments

dialect	Is the dialect to use for the assembler output file. Possible values are:	
	`att`	Tells the compiler to generate the assembler output file using AT&T* syntax.
	`intel`	Tells the compiler to generate the assembler output file using Intel syntax.

Default

`-masm=att`	The compiler generates the assembler output file using AT&T* syntax.

Description

This option tells the compiler to generate the assembler output file using a selected dialect.

IDE Equivalent

None

Alternate Options

None

mauto-arch, Qauto-arch

Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit.

Syntax

Linux OS and macOS:

`-mauto-arch=value`

Windows OS:

`/Qauto-arch:value`

Arguments

value	Is any setting you can specify for option `[Q]ax`.

Default

OFF	No additional execution path is generated.

Description

This option tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit. It also generates a baseline code path.

This option cannot be used together with any options that may require Intel-specific optimizations (such as `[Q]x` or `[Q]ax`).

IDE Equivalent

None

Alternate Options

None

See Also

`ax, Qax` compiler option

mbranches-within-32B-boundaries, Qbranches-within-32B-boundaries

Tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.

Syntax

Linux OS:

`-mbranches-within-32B-boundaries`

`-mno-branches-within-32B-boundaries`

macOS:

`-mbranches-within-32B-boundaries`

`-mno-branches-within-32B-boundaries`

Windows OS:

`/Qbranches-within-32B-boundaries`

`/Qbranches-within-32B-boundaries-`

Arguments

None

Default

`-mno-branches-within-32B-boundaries` or `/Qbranches-within-32B-boundaries-`	Branches and fused branches are not aligned on 32-byte boundaries.

Description

This option tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.

> **NOTE**
> When you use this option, it may affect binary utilities usage experience, such as debugability.

IDE Equivalent

None

Alternate Options

None

mconditional-branch, Qconditional-branch

Lets you identify and fix code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction.

Syntax

Linux OS:

`-mconditional-branch=`*`keyword`*

macOS:

`-mconditional-branch=`*`keyword`*

Windows OS:

`/Qconditional-branch:`*`keyword`*

Arguments

`keyword` Indicates to the compiler what action to take. Possible values are:

`keep`	Tells the compiler to not attempt any vulnerable code detection or fixing. This is equivalent to not specifying the `-mconditional-branch` option.

`pattern-report`	Tells the compiler to perform a search of vulnerable code patterns in the compilation and report all occurrences to stderr.
`pattern-fix`	Tells the compiler to perform a search of vulnerable code patterns in the compilation and generate code to ensure that the identified data accesses are not executed speculatively. It will also report any fixed patterns to stderr. This setting does not guarantee total mitigation, it only fixes cases where all components of the vulnerability can be seen or determined by the compiler. The pattern detection will be more complete if advanced optimization options are specified or are in effect, such as option `O3` and option `-ipo` (or `/Qipo`).
`all-fix`	Tells the compiler to fix all of the vulnerable code so that it is either not executed speculatively, or there is no observable side-channel created from their speculative execution. Since it is a complete mitigation against Spectre variant 1 attacks, this setting will have the most run-time performance cost. In contrast to the `pattern-fix` setting, the compiler will not attempt to identify the exact conditional branches that may have led to the mis-speculated execution.
`all-fix-lfence`	This is the same as specifying setting `all-fix`.
`all-fix-cmov`	Tells the compiler to treat any path where speculative execution of a memory load creates vulnerability (if mispredicted). The compiler automatically adds mitigation code along any vulnerable paths found, but it uses a different method then the one used for `all-fix` (or `all-fix-lfence`). This method uses CMOVcc instruction execution, which constrains speculative execution. Thus, it is used for keeping track of the predicate value, which is updated on each conditional branch. To prevent Spectre v.1 attack, each memory load that is potentially vulnerable is bitwise ORed with the predicate to mask out the loaded value if the code is on a mispredicted path. This is analogous to the Clang compiler's option to do Speculative Load Hardening. This setting is only supported on Intel® 64 architecture-based systems.

Default

`-mconditional-branch=keep` and `/Qconditional-branch:keep`	The compiler does not attempt any vulnerable code detection or fixing.

Description

This option lets you identify code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction. Depending on the setting you choose, vulnerabilities may be detected and code may be generated to attempt to mitigate the security risk.

IDE Equivalent

Visual Studio: **Code Generation [Intel C++] > Spectre Variant 1 Mitigation**

Eclipse: None

Xcode: None

Alternate Options

None

minstruction, Qinstruction

Determines whether MOVBE instructions are generated for certain Intel processors.

Syntax

Linux OS and macOS:

`-minstruction=[no]movbe`

Windows OS:

`/Qinstruction:[no]movbe`

Arguments

None

Default

`-minstruction=nomovbe` or `/Qinstruction:nomovbe`	The compiler does not generate MOVBE instructions for Intel Atom® processors.

Description

This option determines whether MOVBE instructions are generated for Intel Atom® processors. To use this option, you must also specify `[Q]xATOM_SSSE3` or `[Q]xATOM_SSE4.2`.

If `-minstruction=movbe` or `/Qinstruction:movbe` is specified, the following occurs:

- MOVBE instructions are generated that are specific to the Intel Atom® processor.
- Generated executables can only be run on Intel Atom® processors or processors that support Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) or Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) and MOVBE.

If `-minstruction=nomovbe` or `/Qinstruction:nomovbe` is specified, the following occurs:

- The compiler optimizes code for the Intel Atom® processor, but it does not generate MOVBE instructions.
- Generated executables can be run on non-Intel Atom® processors that support Intel® SSE3 or Intel® SSE4.2.

IDE Equivalent

None

Alternate Options

None

See Also

x, Qx compiler option

momit-leaf-frame-pointer

Determines whether the frame pointer is omitted or kept in leaf functions.

Syntax

Linux OS:

`-momit-leaf-frame-pointer`

`-mno-omit-leaf-frame-pointer`

macOS:

`-momit-leaf-frame-pointer`

`-mno-omit-leaf-frame-pointer`

Windows OS:

None

Arguments

None

Default

Varies	If you specify option `-fomit-frame-pointer` (or it is set by default), the default is `-momit-leaf-frame-pointer`. If you specify option `-fno-omit-frame-pointer`, the default is `-mno-omit-leaf-frame-pointer`.

Description

This option determines whether the frame pointer is omitted or kept in leaf functions. It is related to option `-f[no-]omit-frame-pointer` and the setting for that option has an effect on this option.

Consider the following option combinations:

Option Combination	Result

`-fomit-frame-pointer -momit-leaf-frame-pointer`
or
`-fomit-frame-pointer -mno-omit-leaf-frame-pointer`
	Both combinations are the same as specifying `-fomit-frame-pointer`. Frame pointers are omitted for all routines.

`-fno-omit-frame-pointer -momit-leaf-frame-pointer`	In this case, the frame pointer is omitted for leaf routines, but other routines will keep the frame pointer.

Option Combination	Result
	This is the intended effect of option `-momit-leaf-frame-pointer`.
`-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer`	In this case, `-mno-omit-leaf-frame-pointer` is ignored since `-fno-omit-frame-pointer` retains frame pointers in all routines . This combination is the same as specifying `-fno-omit-frame-pointer`.

This option is provided for compatibility with gcc.

IDE Equivalent

Visual Studio: None

Eclipse: **Optimization > Omit frame pointer for leaf routines**

Xcode: **Optimization > Provide Frame Pointer For Leaf Routines**

Alternate Options

None

See Also

fomit-frame-pointer, Oy compiler option

mregparm

Lets you control the number registers used to pass integer arguments.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-mregparm=`*n*

macOS:

`None`

Windows OS:

`None`

Arguments

n	Specifies the number of registers to use when passing integer arguments. You can specify at most 3 registers. If you specify a nonzero value for *n*, you must build all modules, including startup modules, and all libraries, including system libraries, with the same value.

Default

OFF	The compiler does not use registers to pass arguments.

Description

Control the number registers used to pass integer arguments. This option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

See Also

`mregparm-version` compiler option

mregparm-version

Determines which version of the Application Binary Interface (ABI) is used for the regparm parameter passing convention.

Syntax

Linux OS:

`-mregparm-version=n`

macOS:

macOS:

None

Windows OS:

None

Arguments

n	Specifies the ABI implementation to use. Possible values are:	
	0	Tells the compiler to use the most recent ABI implementation.
	1	Tells the compiler to use the ABI implementation that is compatible with gcc 3.4.6 and icc 15.0.

Default

0	The compiler uses the most recent ABI implementation.

Description

This option determines which version of the Application Binary Interface (ABI) is used for the regparm parameter passing convention. This option allows compatibility with previous versions of gcc and icc.

IDE Equivalent

None

Alternate Options

None

See Also

`mregparm` compiler option

mstringop-inline-threshold, Qstringop-inline-threshold

Tells the compiler to not inline calls to buffer manipulation functions such as memcpy and memset when the number of bytes the functions handle are known at compile time and greater than the specified value.

Syntax

Linux OS:

`-mstringop-inline-threshold=`*`val`*

macOS:

`-mstringop-inline-threshold=`*`val`*

Windows OS:

`/Qstringop-inline-threshold:`*`val`*

Arguments

val	Is a positive 32-bit integer. If the size is greater than *val*, the compiler will never inline it.

Default

OFF	The compiler uses its own heuristics to determine the default.

Description

This option tells the compiler to not inline calls to buffer manipulation functions such as memcpy and memset when the number of bytes the functions handle are known at compile time and greater than the specified *val*.

IDE Equivalent

None

Alternate Options

None

See Also

mstringop-strategy, Qstringop-strategy compiler option

mstringop-strategy, Qstringop-strategy

Lets you override the internal decision heuristic for the particular algorithm used when implementing buffer manipulation functions such as memcpy and memset.

Syntax

Linux OS and macOS:

`-mstringop-strategy=`*`alg`*

Windows OS:

`/Qstringop-strategy:`*`alg`*

Arguments

`alg`	Specifies the algorithm to use. Possible values are:	
	`const_size_loop`	Tells the compiler to expand the string operations into an inline loop when the size is known at compile time and it is not greater than threshold value. Otherwise, the compiler uses its own heuristics to decide how to implement the string operation.
	`libcall`	Tells the compiler to use a library call when implementing string operations.
	`rep`	Tells the compiler to use its own heuristics to decide what form of `rep movs	stos` to use when inlining string operations.

Default

varies	If optimization option `Os` is specified, the default is `rep`. Otherwise, the default is `const_size_loop`.

Description

This option lets you override the internal decision heuristic for the particular algorithm used when implementing buffer manipulation functions such as memcpy and memset.

This option may have no effect on compiler-generated string functions, for example, a call to memcpy generated by the compiler to implement an array copy or structure copy.

IDE Equivalent

None

Alternate Options

None

See Also

mstringop-inline-threshold, Qstringop-inline-threshold compiler option
Os compiler option

mtune, tune

Performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike -march).

Syntax

Linux OS:

`-mtune=`*`processor`*

macOS:

`-mtune=`*`processor`*

Windows OS:

`/tune:`*`processor`*

Arguments

processor — Is the processor for which the compiler should perform optimizations. Possible values are:

`generic`	Optimizes code for the compiler's default behavior.

`alderlake`
`amberlake`
`broadwell`
`cannonlake`
`cascadelake`
`coffeelake`
`cooperlake`
`goldmont`
`goldmont-plus`
`haswell`
`icelake-client` (or `icelake`)
`icelake-server`
`ivybridge`
`kabylake`
`knl`
`knm`
`rocketlake`
`sandybridge`
`sapphirerapids`
`silvermont`
`skylake`
`skylake-avx512`
`tigerlake`
`tremont`
`whiskeylake`
	Optimizes code for processors that support the specified Intel® processor or microarchitecture code name.
	Keywords `knl` and `silvermont` are only available on Windows* and Linux* systems.
	Keyword `icelake` is deprecated and may be removed in a future release.

`core-avx2`	Optimizes code for processors that support Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`core-avx-i`	Optimizes code for processors that support Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7-avx`	Optimizes code for processors that support Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7`	Optimizes code for processors that support Intel® SSE4 Efficient Accelerated String and Text Processing instructions. May also generate code for Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSE3, SSE2, SSE, and SSSE3 instructions.
`atom`	Optimizes code for processors that support MOVBE instructions, depending on the setting of option `-minstruction` (Linux* and macOS) or `/Qinstruction` (Windows*). May also generate code for SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`core2`	Optimizes for the Intel® Core™2 processor family, including support for MMX™, Intel® SSE, SSE2, SSE3 and SSSE3 instruction sets.
`pentium-mmx`	Optimizes for Intel® Pentium® with MMX technology.
`pentiumpro`	Optimizes for Intel® Pentium® Pro, Intel Pentium II, and Intel Pentium III processors.
`pentium4m`	Optimizes for Intel® Pentium® 4 processors with MMX technology.

`pentium-m`
`pentium4`
`pentium3`
`pentium`
	Optimizes code for Intel® Pentium® processors. Value `pentium3` is only available on Linux* systems.

Default

generic	Code is generated for the compiler's default behavior.

Description

This option performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike `-march`).

The resulting executable is backwards compatible and generated code is optimized for specific processors. For example, code generated with `-mtune=core2` or `/tune:core2` will run correctly on 4th Generation Intel® Core™ processors, but it might not run as fast as if it had been generated using `-mtune=haswell` or `/tune:haswell`. Code generated with `-mtune=haswell` (`/tune:haswell`) or `-mtune=core-avx2` (`/tune:core-avx2`) will also run correctly on Intel® Core™2 processors, but it might not run as fast as if it had been generated using `-mtune=core2` or `/tune:core2`. This is in contrast to code generated with `-march=core-avx2` or `/arch:core-avx2`, which will not run correctly on older processors such as Intel® Core™2 processors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Windows

Visual Studio: **Code Generation [Intel C++] >Intel Processor Microarchitecture-Specific Optimization**

Linux

Eclipse: **Code Generation > Intel Processor Microarchitecture-Specific Optimization**

OS X

Xcode: **Code Generation > Intel Processor Microarchitecture-Specific Optimization**

Alternate Options

`-mtune`	Linux: `-mcpu` (this is a deprecated option) macOS: None Windows: None

See Also

`march` compiler option

Qcxx-features

Enables standard C++ features without disabling Microsoft features.*

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qcxx-features`

Arguments

None

Default

OFF	The compiler enables standard C++ features.

Description

This option enables standard C++ features without disabling Microsoft* features within the bounds of what is provided in the Microsoft headers and libraries.

This option has the same effect as specifying `/EHsc /GR`.

IDE Equivalent

None

Alternate Options

None

Qpatchable-addresses

Tells the compiler to generate code such that references to statically assigned addresses can be patched.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qpatchable-addresses`

Arguments

None

Default

OFF	The compiler does not generate patchable addresses.

Description

This option tells the compiler to generate code such that references to statically assigned addresses can be patched with arbitrary 64-bit addresses.

Normally, the Windows* system compiler that runs on Intel® 64 architecture uses 32-bit relative addressing to reference statically allocated code and data. That assumes the code or data is within 2GB of the access point, an assumption that is enforced by the Windows object format.

However, in some patching systems, it is useful to have the ability to replace a global address with some other arbitrary 64-bit address, one that might not be within 2GB of the access point.

This option causes the compiler to avoid 32-bit relative addressing in favor of 64-bit direct addressing so that the addresses can be patched in place without additional code modifications. This option causes code size to increase, and since 32-bit relative addressing is usually more efficient than 64-bit direct addressing, you may see a performance impact.

IDE Equivalent

None

Alternate Options

None

Qsafeseh

Registers exception handlers for safe exception handling.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qsafeseh[-]`

Arguments

None

Default

ON	Exception handlers are enabled for safe exception handling.

Description

Registers exception handlers for safe exception handling. It also marks objects as "compatible with the Registered Exception Handling feature" whether they contain handlers or not. This is important because the Windows linker will only generate the "special registered EH table" if ALL objects that it is building into an image are marked as compatible. If any objects are not marked as compatible, the EH table is not generated.

Digital signatures certify security and are required for components that are shipped with Windows, such as device drivers.

IDE Equivalent

None

Alternate Options

None

See Also

`/EH` compiler option

regcall, Qregcall

Tells the compiler that the __regcall calling convention should be used for functions that do not directly specify a calling convention.

Syntax

Linux OS:

`-regcall`

macOS:

`-regcall`

Windows OS:

`/Qregcall`

Arguments

None

Default

OFF	The __regcall calling convention will only be used if a function explicitly specifies it.

Description

This option tells the compiler that the __regcall calling convention should be used for functions that do not directly specify a calling convention. This calling convention ensures that as many values as possible are passed or returned in registers.

It ensures that __regcall is the default calling convention for functions in the compilation, unless another calling convention is specified in a declaration.

This calling convention is ignored if it is specified for a function with variable arguments.

Note that all __regcall functions must have prototypes.

IDE Equivalent

None

Alternate Options

None

See Also

C/C++ Calling Conventions

x, Qx

Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.

Syntax

Linux OS:

`-x`*code*

macOS:

`-x`*code*

Windows OS:

`/Qx`*code*

Arguments

code	Specifies a feature set that the compiler can target, including which instruction sets and optimizations it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (Intel® SSSE). Possible values are:

`ALDERLAKE`
`AMBERLAKE`
`BROADWELL`
`CANNONLAKE`
`CASCADELAKE`
`COFFEELAKE`
`COOPERLAKE`
`GOLDMONT`
`GOLDMONT-PLUS`
`HASWELL`
`ICELAKE-CLIENT` (or `ICELAKE`)
`ICELAKE-SERVER`
`IVYBRIDGE`
`KABYLAKE`
`KNL`
`KNM`
`ROCKETLAKE`
`SANDYBRIDGE`
`SAPPHIRERAPIDS`
`SILVERMONT`
`SKYLAKE`
`SKYLAKE-AVX512`
`TIGERLAKE`
`TREMONT`
`WHISKEYLAKE`
	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name. Optimizes for the specified Intel® processor or microarchitecture code name.
	
	Keywords `KNL` and `SILVERMONT` are only available on Windows* and Linux* systems.
	
	Keyword `ICELAKE` is deprecated and may be removed in a future release.

`COMMON-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
`CORE-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection Instructions (CDI), Intel® AVX-512

	Doubleword and Quadword Instructions (DQI), Intel® AVX-512 Byte and Word Instructions (BWI) and Intel® AVX-512 Vector Length extensions, as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
CORE-AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel® processors that support Intel® AVX2 instructions.
CORE-AVX-I	May generate Float-16 conversion instructions and the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel® processors that support Float-16 conversion instructions and the RDRND instruction.
AVX	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel processors that support Intel® AVX instructions.
SSE4.2	May generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions, Intel® SSE4 Vectorizing Compiler and Media Accelerator, and Intel® SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE4.2 instructions.
SSE4.1	May generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel® processors. May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel processors that support Intel® SSE4.1 instructions.
ATOM_SSE4.2	May generate MOVBE instructions for Intel® processors, depending on the setting of option `-minstruction` (Linux* and macOS) or `/Qinstruction` (Windows*). May also generate Intel® SSE4.2, SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for Intel Atom® processors that support Intel® SSE4.2 and MOVBE instructions. This keyword is only available on Windows* and Linux* systems.
ATOM_SSSE3	May generate MOVBE instructions for Intel® processors, depending on the setting of option `-minstruction` (Linux* and macOS) or `/Qinstruction` (Windows*). May also generate SSSE3, Intel® SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for Intel Atom® processors that support Intel® SSE3 and MOVBE instructions. This keyword is only available on Windows* and Linux* systems.

`SSSE3`	May generate SSSE3 and Intel® SSE3, SSE2, and SSE instructions for Intel® processors. Optimizes for Intel processors that support SSSE3 instructions. For macOS systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
`SSE3`	May generate Intel® SSE3, SSE2, and SSE instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE3 instructions. This value is not available on macOS systems.
`SSE2`	May generate Intel® SSE2 and SSE instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE2 instructions. This value is not available on macOS systems.

You can also specify a `Host`. For more information, see option [Q]xHost.

Default

Windows* systems: None Linux* systems: None macOS systems: `SSSE3`	On Windows systems, if neither `/Qx` nor `/arch` is specified, the default is `/arch:SSE2`. On Linux systems, if neither `-x` nor `-m` is specified, the default is `-msse2`.

Description

This option tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate. It also enables optimizations in addition to Intel feature-specific optimizations.

The specialized code generated by this option may only run on a subset of Intel® processors.

The resulting executables created from these option *code* values can only be run on Intel® processors that support the indicated instruction set.

The binaries produced by these *code* values will run on Intel® processors that support the specified features.

Do not use *code* values to create binaries that will execute on a processor that is not compatible with the targeted processor. The resulting program may fail with an illegal instruction exception or display other unexpected behavior.

Compiling the function main() with any of the *code* values produces binaries that display a fatal run-time error if they are executed on unsupported processors, including all non-Intel processors.

Compiler options `m` and `arch` produce binaries that should run on processors not made by Intel that implement the same capabilities as the corresponding Intel® processors.

The `-x` and `/Qx` options enable additional optimizations not enabled with options `-m` or `/arch` (nor with options `-ax` and `/Qax`).

On Windows systems, options `/Qx` and /arch are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning. Similarly, on Linux and macOS systems, options `-x` and `-m` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

NOTE
All settings except SSE2 do a CPU check. However, if you specify option -O0 (Linux* and macOS) or option /Od (Windows*), no CPU check is performed.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Intel Processor-Specific Optimization**

Eclipse

Eclipse: **Code Generation > Intel Processor-Specific Optimization**

Xcode

Xcode: **Code Generation > Intel Processor-Specific Optimization**

Alternate Options

None

See Also

`xHost, QxHost` compiler option
`ax, Qax` compiler option
`arch` compiler option
`march` compiler option
`minstruction, Qinstruction` compiler option
`m` compiler option

xHost, QxHost

Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.

Syntax

Linux OS and macOS:

`-xHost`

Windows OS:

`/QxHost`

Arguments

None

Default

Windows* systems: None Linux* systems: None macOS systems: `-xSSSE3`	On Windows systems, if neither `/Qx` nor `/arch` is specified, the default is `/arch:SSE2`. On Linux systems, if neither `-x` nor `-m` is specified, the default is `-msse2`.

Description

This option tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.

The instructions generated by this compiler option differ depending on the compilation host processor.

The following table describes the effects of specifying the `[Q]xHost` option and it tells whether the resulting executable will run on processors different from the host processor.

Descriptions in the table refer to Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® Advanced Vector Extensions (Intel® AVX), Intel® Streaming SIMD Extensions (Intel® SSE), and Supplemental Streaming SIMD Extensions (SSSE).

Instruction Set of Host Processor	Effects When the -xHost or /QxHost Compiler Option is Specified
Intel® AVX2	When compiling on Intel® processors: Corresponds to option `[Q]xCORE-AVX2`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® AVX2 instructions. When compiling on non-Intel processors: Corresponds to option `-march=core-avx2` (Linux* and macOS) or `/arch:CORE-AVX2` (Windows*). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® AVX2 instructions.. You may see a run-time error if the run-time processor does not support Intel® AVX2 instructions.
Intel® AVX	When compiling on Intel® processors: Corresponds to option `[Q]xAVX`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® AVX instructions. When compiling on non-Intel processors: Corresponds to option `-mavx` (Linux and macOS) or `/arch:AVX` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® AVX instructions. You may see a run-time error if the run-time processor does not support Intel® AVX instructions.
Intel® SSE4.2	When compiling on Intel® processors: Corresponds to option `[Q]xSSE4.2`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE4.2 instructions. When compiling on non-Intel processors:

Instruction Set of Host Processor	Effects When the -xHost or /QxHost Compiler Option is Specified
	Corresponds to option `-msse4.2` (Linux and macOS) or `/arch:SSE4.2` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE4.2 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE4.2 instructions.
Intel® SSE4.1	When compiling on Intel® processors: Corresponds to option `[Q]xSSE4.1`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE4.1 instructions. When compiling on non-Intel processors: Corresponds to option `-msse4.1` (Linux and macOS) or `/arch:SSE4.1` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE4.1 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE4.1 instructions.
SSSE3	When compiling on Intel® processors: Corresponds to option `[Q]xSSSE3`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support SSSE3 instructions. When compiling on non-Intel processors: Corresponds to option `-mssse3` (Linux and macOS) or `/arch:SSSE3` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least SSSE3 instructions. You may see a run-time error if the run-time processor does not support SSSE3 instructions.
Intel® SSE3	When compiling on Intel® processors: Corresponds to option `[Q]xSSE3`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE3 instructions. When compiling on non-Intel processors: Corresponds to option `-msse3` (Linux and macOS) or `/arch:SSE3` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE3 instructions. You may see a warning run-time error if the run-time processor does not support Intel® SSE3 instructions.
Intel® SSE2	When compiling on Intel® processors or non-Intel processors: Corresponds to option `-msse2` (Linux and macOS) or `/arch:SSE2` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE2 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE2 instructions.

For more information on other settings for option `[Q]x`, see that option description.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Product and Performance Information
Notice revision #20201201

IDE Equivalent

Visual Studio: **Code Generation > Intel Processor-Specific Optimization**

IDE Equivalent

Visual Studio: **Code Generation > Intel Processor-Specific Optimization**

Eclipse: **Code Generation > Intel Processor-Specific Optimization**

Xcode: **Code Generation > Intel Processor-Specific Optimization**

Alternate Options

None

See Also

`x, Qx` compiler option
`ax, Qax` compiler option
`m` compiler option
`arch` compiler option

Interprocedural Optimization Options

This section contains descriptions for compiler options that pertain to interprocedural optimization.

ffat-lto-objects

Determines whether a fat link-time optimization (LTO) object, containing both intermediate language and object code, is generated during an interprocedural optimization compilation (-c –ipo).

Syntax

Linux OS:

`-ffat-lto-objects`

`-fno-fat-lto-objects`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-ffat-lto-objects`	When `-c -ipo` is specified, the compiler generates a fat link-time optimization (LTO) object that has both a true object and a discardable intermediate language section.

Description

This option determines whether a fat link time optimization (LTO) object, containing both intermediate language and object code, is generated during an interprocedural optimization compilation (`-c -ipo`).

During an interprocedural optimization compilation (`-c -ipo`), the following occurs:

- If you specify `-ffat-lto-objects`, the compiler generates a fat link-time optimization (LTO) object that has both a true object and a discardable intermediate language section. This enables both link-time optimization (LTO) linking and normal linking.
- If you specify `-fno-fat-lto-objects`, the compiler generates a fat link-time optimization (LTO) object that only has a discardable intermediate language section; no true object is generated. This option may improve compilation time.

Note that these files will be inserted into archives in the form in which they were created.

This option is provided for compatibility with gcc. For more information about this option, see the gcc documentation.

> **NOTE**
> Intel's intermediate language representation is not compatible with gcc's intermediate language representation.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

ip, Qip

Determines whether additional interprocedural optimizations for single-file compilation are enabled.

Syntax

Linux OS:

`-ip`

`-no-ip`

macOS:

`-ip`

`-no-ip`

Windows OS:

/Qip

/Qip-

Arguments

None

Default

OFF	Some limited interprocedural optimizations occur, including inline function expansion for calls to functions defined within the current source file. These optimizations are a subset of full intra-file interprocedural optimizations. Note that this setting is not the same as -no-ip (Linux* and macOS) or /Qip- (Windows*).

Description

This option determines whether additional interprocedural optimizations for single-file compilation are enabled.

The [Q]ip option enables additional interprocedural optimizations for single-file compilation.

Options -no-ip (Linux and macOS) and /Qip- (Windows) may not disable inlining. To ensure that inlining of user-defined functions is disabled, specify -inline-level=0or -fno-inline (Linux and macOS), or specify /Ob0 (Windows). To ensure that inliningof compiler intrinsic functions is disabled, specify -fno-builtin (Linux and macOS) or /Oi- (Windows).

IDE Equivalent

Windows

Visual Studio: **Optimization > Interprocedural Optimization**

Linux

Eclipse: **Optimization > Enable Interprocedural Optimizations for Single File Compilation**

OS X

Xcode: **Optimization > Enable Interprocedural Optimization for Single File Compilation**

Alternate Options

None

See Also

finline-functions compiler option

ip-no-inlining, Qip-no-inlining

Disables full and partial inlining enabled by interprocedural optimization options.

Syntax

Linux OS:

-ip-no-inlining

macOS:

`-ip-no-inlining`

Windows OS:

`/Qip-no-inlining`

Arguments

None

Default

OFF	Inlining enabled by interprocedural optimization options is performed.

Description

This option disables full and partial inlining enabled by the following interprocedural optimization options:

- On Linux* and macOS systems: `-ip` or `-ipo`
- On Windows* systems: `/Qip`, `/Qipo`, or `/Ob2`

It has no effect on other interprocedural optimizations.

On Windows systems, this option also has no effect on user-directed inlining specified by option `/Ob1`.

IDE Equivalent

None

Alternate Options

None

ip-no-pinlining, Qip-no-pinlining

Disables partial inlining enabled by interprocedural optimization options.

Syntax

Linux OS:

`-ip-no-pinlining`

macOS:

`-ip-no-pinlining`

Windows OS:

`/Qip-no-pinlining`

Arguments

None

Default

OFF	Inlining enabled by interprocedural optimization options is performed.

Description

This option disables partial inlining enabled by the following interprocedural optimization options:

- On Linux* and macOS systems: `-ip` or `-ipo`
- On Windows* systems: `/Qip` or `/Qipo`

It has no effect on other interprocedural optimizations.

IDE Equivalent

None

Alternate Options

None

ipo, Qipo

Enables interprocedural optimization between files.

Syntax

Linux OS:

`-ipo[n]`

`-no-ipo`

macOS:

`-ipo[n]`

`-no-ipo`

Windows OS:

`/Qipo[n]`

`/Qipo-`

Arguments

n	Is an optional integer that specifies the number of object files the compiler should create. The integer must be greater than or equal to 0.

Default

`-no-ipo` or `/Qipo-`	Multifile interprocedural optimization is not enabled.

Description

This option enables interprocedural optimization between files. This is also called multifile interprocedural optimization (multifile IPO) or Whole Program Optimization (WPO).

When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

You cannot specify the names for the files that are created.

If *n* is 0, the compiler decides whether to create one or more object files based on an estimate of the size of the application. It generates one object file for small applications, and two or more object files for large applications.

If *n* is greater than 0, the compiler generates *n* object files, unless *n* exceeds the number of source files (*m*), in which case the compiler generates only *m* object files.

If you do not specify *n*, the default is 0.

NOTE
When you specify option `[Q]ipo` with option `[q or Q]opt-report`, IPO information is generated in the compiler optimization report at link time. After linking, you will see a report named ipo_out.optrpt in the folder where you linked all of the object files.

IDE Equivalent

Windows

Visual Studio: **Optimization > Interprocedural Optimization**

Linux

Eclipse: **Optimization > Enable Whole Program Optimization**

OS X

Xcode: None

Alternate Options

None

ipo-c, Qipo-c

Tells the compiler to optimize across multiple files and generate a single object file.

Syntax

Linux OS:

`-ipo-c`

macOS:

`-ipo-c`

Windows OS:

`/Qipo-c`

Arguments

None

Default

OFF	The compiler does not generate a multifile object file.

Description

This option tells the compiler to optimize across multiple files and generate a single object file (named ipo_out.o on Linux* and macOS systems; ipo_out.obj on Windows* systems).

It performs the same optimizations as the `[Q]ipo` option, but compilation stops before the final link stage, leaving an optimized object file that can be used in further link steps.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

ipo-jobs, Qipo-jobs

Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).

Syntax

Linux OS:

`-ipo-jobsn`

macOS:

`-ipo-jobsn`

Windows OS:

`/Qipo-jobs:n`

Arguments

n	Is the number of commands (jobs) to run simultaneously. The number must be greater than or equal to 1.

Default

`-ipo-jobs1` or `/Qipo-jobs:1`	One command (job) is executed in an interprocedural optimization parallel build.

Description

This option specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO). It should only be used if the link-time compilation is generating more than one object. In this case, each object is generated by a separate compilation, which can be done in parallel.

This option can be affected by the following compiler options:

- `[Q]ipo` when applications are large enough that the compiler decides to generate multiple object files.
- `[Q]ipon` when *n* is greater than 1.
- `[Q]ipo-separate`

> **Caution**
> Be careful when using this option. On a multi-processor system with lots of memory, it can speed application build time. However, if *n* is greater than the number of processors, or if there is not enough memory to avoid thrashing, this option can increase application build time.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

`ipo-separate, Qipo-separate` compiler option

ipo-S, Qipo-S

Tells the compiler to optimize across multiple files and generate a single assembly file.

Syntax

Linux OS:

`-ipo-S`

macOS:

`-ipo-S`

Windows OS:

`/Qipo-S`

Arguments

None

Default

OFF	The compiler does not generate a multifile assembly file.

Description

This option tells the compiler to optimize across multiple files and generate a single assembly file (named ipo_out.s on Linux* and macOS systems; ipo_out.asm on Windows* systems).

It performs the same optimizations as the `[Q]ipo` option, but compilation stops before the final link stage, leaving an optimized assembly file that can be used in further link steps.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

ipo-separate, Qipo-separate

Tells the compiler to generate one object file for every source file.

Syntax

Linux OS:

`-ipo-separate`

macOS:

`None`

Windows OS:

`/Qipo-separate`

Arguments

None

Default

OFF	The compiler decides whether to create one or more object files.

Description

This option tells the compiler to generate one object file for every source file. It overrides any `[Q]ipo` option specification.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

Advanced Optimization Options

This section contains descriptions for compiler options that pertain to advanced optimization.

alias-const, Qalias-const

Determines whether the compiler assumes a parameter of type pointer-to-const does not alias with a parameter of type pointer-to-non-const.

Syntax

Linux OS:

`-alias-const`

`-no-alias-const`

macOS:

`-alias-const`

`-no-alias-const`

Windows OS:

`/Qalias-const`

`/Qalias-const-`

Arguments

None

Default

`-no-alias-const` or `/Qalias-const-`	The compiler uses standard C/C++ rules for the interpretation of const.

Description

This option determines whether the compiler assumes a parameter of type pointer-to-const does not alias with a parameter of type pointer-to-non-const. It implies an additional attribute for const.

This functionality complies with the input/output buffer rule, which assumes that input and output buffer arguments do not overlap. This option allows the compiler to do some additional optimizations with those parameters.

In C99, you can also get the same result if you additionally declare your pointer parameters with the restrict keyword.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Assume Restrict Semantics for Const**

OS X

Xcode: **Data > Assume Restrict Semantics for Const**

Alternate Options

None

ansi-alias, Qansi-alias

Enables or disables the use of ANSI aliasing rules in optimizations.

Syntax

Linux OS:

`-ansi-alias`

`-no-ansi-alias`

macOS:

`-ansi-alias`

`-no-ansi-alias`

Windows OS:

`/Qansi-alias`

`/Qansi-alias-`

Arguments

None

Default

Windows* systems: `/Qansi-alias-`	ANSI aliasing rules are disabled in optimizations.
Linux* and macOS systems: `-ansi-alias`	ANSI aliasing rules are enabled in optimizations.

Description

This option tells the compiler to assume that the program adheres to ISO C Standard aliasability rules.

If your program adheres to the ANSI aliasability rules, this option allows the compiler to optimize more aggressively. If your program does not adhere to these rules, this option may cause the compiler to generate incorrect code.

If you are compiling on a Linux* or an macOS system and your program does not adhere to the ANSI aliasability rules, you can specify `-no-ansi-alias` to ensure program correctness.

When you specify the `[Q]ansi-alias` option, the ansi-alias checker is enabled by default. To disable the ansi-alias checker, you must specify `-no-ansi-alias-check` (Linux* and macOS) or `/Qansi-alias-check-` (Windows*).

IDE Equivalent

Windows

Visual Studio: **Language > Enable Use of ANSI Aliasing Rules in Optimizations**

Linux

Eclipse: **Language > Enable Use of ANSI Aliasing Rules in Optimizations**

OS X

Xcode: **Language > Enable ANSI Aliasing**

Alternate Options

Linux and macOS: `-fstrict-aliasing`

Windows: None

See Also

ansi-alias-check, Qansi-alias-check
compiler option

ansi-alias-check, Qansi-alias-check

Enables or disables the ansi-alias checker.

Syntax

Linux OS:

`-ansi-alias-check`

`-no-ansi-alias-check`

macOS:

`-ansi-alias-check`

`-no-ansi-alias-check`

Windows OS:

`/Qansi-alias-check`

`/Qansi-alias-check-`

Arguments

None

Default

`-no-ansi-alias-check` or `/Qansi-alias-check-`	The ansi-alias checker is disabled unless option `-ansi-alias-check` or `/Qansi-alias-check` has been specified.

Description

This option enables or disables the ansi-alias checker. The ansi-alias checker checks the source code for potential violations of ANSI aliasing rules and disables unsafe optimizations related to the code for those statements that are identified as potential violations.

You can use option `-Wstrict-aliasing` to identify potential violations.

If the `[Q]ansi-alias` option has been specified, the ansi-alias checker is enabled by default. You can use the negative form of the option (see Syntax above) to disable the checker.

IDE Equivalent

None

Alternate Options

None

See Also

ansi-alias, Qansi-alias
  compiler option

Wstrict-aliasing
  compiler option

complex-limited-range, Qcomplex-limited-range

Determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

Syntax

Linux OS:

`-complex-limited-range`

`-no-complex-limited-range`

macOS:

`-complex-limited-range`

`-no-complex-limited-range`

Windows OS:

`/Qcomplex-limited-range`

`/Qcomplex-limited-range-`

Arguments

None

Default

`-no-complex-limited-range` or `/Qcomplex-limited-range-`	Basic algebraic expansions of some arithmetic operations involving data of type COMPLEX are disabled.

Description

This option determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

When the option is enabled, this can cause performance improvements in programs that use a lot of COMPLEX arithmetic. However, values at the extremes of the exponent range may not compute correctly.

IDE Equivalent

Windows

Visual Studio: **Floating Point > Limit COMPLEX Range**

Linux

Eclipse: **Floating Point > Limit COMPLEX Range**

OS X

Xcode: **Floating Point > Limit COMPLEX Range**

Alternate Options

None

fargument-alias, Qalias-args

Determines whether function arguments can alias each other.

Syntax

Linux OS and macOS:

`-fargument-alias`

`-fargument-noalias`

Windows OS:

`/Qalias-args`

`/Qalias-args-`

Arguments

None

Default

`-fargument-alias` or `/Qalias-args`	Function arguments can alias each other and can alias global storage.

Description

This option determines whether function arguments can alias each other. If you specify `-fargument-noalias` or `/Qalias-args-`, function arguments cannot alias each other, but they can alias global storage.

On Linux and macOS systems, you can also disable aliasing for global storage, by specifying option `-fargument-noalias-global`.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Enable Argument Aliasing**

OS X

Xcode: **Data > Enable Argument Aliasing**

See Also

`fargument-noalias-global` compiler option

fargument-noalias-global

Tells the compiler that function arguments cannot alias each other and cannot alias global storage.

Syntax

Linux OS and macOS:

`-fargument-noalias-global`

Windows OS:

None

Arguments

None

Default

OFF	Function arguments can alias each other and can alias global storage.

Description

This option tells the compiler that function arguments cannot alias each other and they cannot alias global storage.

If you only want to prevent function arguments from being able to alias each other, specify option `-fargument-noalias`.

IDE Equivalent

None

Alternate Options

None

See Also

fargument-alias, Qalias-args
compiler option

ffreestanding, Qfreestanding

Ensures that compilation takes place in a freestanding environment.

Syntax

Linux OS:

`-ffreestanding`

macOS:

`-ffreestanding`

Windows OS:

`/Qfreestanding`

Arguments

None

Default

OFF	Standard libraries are used during compilation.

Description

This option ensures that compilation takes place in a freestanding environment. The compiler assumes that the standard library may not exist and program startup may not necessarily be at main. This environment meets the definition of a freestanding environment as described in the C and C++ standard.

An example of an application requiring such an environment is an OS kernel.

NOTE
When you specify this option, the compiler will not assume the presence of compiler-specific libraries. It will only generate calls that appear in the source code.

IDE Equivalent

None

Alternate Options

None

fjump-tables

Determines whether jump tables are generated for switch statements.

Syntax

Linux OS:

`-fjump-tables`

`-fno-jump-tables`

macOS:

`-fjump-tables`

`-fno-jump-tables`

Windows OS:

`None`

Arguments

None

Default

`-fjump-tables`	The compiler may use jump tables for switch statements.

Description

This option determines whether jump tables are generated for switch statements.

Option `-fno-jump-tables` prevents the compiler from generating jump tables for switch statements. This action is performed unconditionally and independent of any generated code performance consideration.

Option `-fno-jump-tables` also prevents the compiler from creating switch statements internally as a result of optimizations.

Use `-fno-jump-tables` with `-fpic` when compiling objects that will be loaded in a way where the jump table relocation cannot be resolved.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic` compiler option

ftls-model

Changes the thread local storage (TLS) model.

Syntax

Linux OS:

`-ftls-model=`*model*

macOS:

`-ftls-model=`*model*

Windows OS:

None

Arguments

model	Determines the TLS model used by the compiler. Possible values are:	
	`global-dynamic`	Generates a generic TLS code. The code can be used everywhere and the code can access variables defined anywhere else. This setting causes the largest size code to be generated and uses the most run time to produce.
	`local-dynamic`	Generates an optimized TLS code. To use this setting, the thread-local variables must be defined in the same object in which they are referenced.
	`initial-exec`	Generates a restrictive, optimized TLS code. To use this setting, the thread-local variables accessed must be defined in one of the modules available to the program.
	`local-exec`	Generates the most restrictive TLS code. To use this setting, the thread-local variables must be defined in the executable.

Default

OFF	The compiler uses default heuristics when determining the thread-local storage model.

Description

This option changes the thread local storage (TLS) model. Thread-local storage is a mechanism by which variables are allocated in a way that causes one instance of the variable per extant thread.

For more information on the thread-storage localization models, see the appropriate GCC* documentation.

For more information on the thread-storage localization models, see the appropriate Clang documentation.

IDE Equivalent

None

Alternate Options

None

funroll-all-loops

Unroll all loops even if the number of iterations is uncertain when the loop is entered.

Syntax

Linux OS and macOS:

`-funroll-all-loops`

Windows OS:

None

Arguments

None

Default

OFF	Do not unroll all loops.

Description

Unroll all loops, even if the number of iterations is uncertain when the loop is entered. There may a performance impact with this option.

IDE Equivalent

None

Alternate Options

None

guide, Qguide

Lets you set a level of guidance for auto-vectorization, auto parallelism, and data transformation.

Syntax

Linux OS:

`-guide[=n]`

macOS:

`-guide[=n]`

Windows OS:

`/Qguide[:n]`

Arguments

n	Is an optional value specifying the level of guidance to be provided.

The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for parallelism, vectorization, and data transformation.

Description

This option lets you set a level of guidance (advice) for auto-vectorization, auto parallelism, and data transformation. It causes the compiler to generate messages suggesting ways to improve these optimizations.

When this option is specified, the compiler does not produce any objects or executables.

You must also specify the `[Q]parallel` option to receive auto parallelism guidance.

You can set levels of guidance for the individual guide optimizations by specifying one of the following options:

`[Q]guide-data-trans`	Provides guidance for data transformation.
`[Q]guide-par`	Provides guidance for auto parallelism.
`[Q]guide-vec`	Provides guidance for auto-vectorization.

If you specify the `[Q]guide` option and also specify one of the options setting a level of guidance for an individual guide optimization, the value set for the individual guide optimization will override the setting specified in `[Q]guide`.

If you do not specify `[Q]guide`, but specify one of the options setting a level of guidance for an individual guide optimization, option `[Q]guide` is enabled with the greatest value passed among any of the three individual guide optimizations specified.

In debug mode, this option has no effect unless option `O2` (or higher) is explicitly specified in the same command line.

NOTE
The compiler speculatively performs optimizations as part of guide analysis. As a result, when you use guided auto-parallelism options with options that produce vectorization or auto-parallelizer reports (such as option `[q or Q]opt-report`), the compiler generates "LOOP WAS VECTORIZED" or similar messages as if the compilation was performed with the recommended changes.

When compilation is performed with the `[Q]guide` option, you should use extra caution when interpreting vectorizer diagnostics and auto-parallelizer diagnostics.

NOTE
You can specify `[Q]diag-disable` to prevent the compiler from issuing one or more diagnostic messages.

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Guided Auto Parallelism Analysis**

Eclipse

Eclipse: **Compilation Diagnostics > Enable Guided Auto Parallelism Analysis**

Xcode

Xcode: **Diagnostics > Enable Guided Auto Parallelism Analysis**

Alternate Options

None

See Also

`guide-data-trans, Qguide-data-trans` compiler option
`guide-par, Qguide-par` compiler option
`guide-vec, Qguide-vec` compiler option
`guide-file, Qguide-file` compiler option
`guide-file-append, Qguide-file-append` compiler option
`guide-opts, Qguide-opts` compiler option
`diag, Qdiag` compiler option
`qopt-report, Qopt-report` compiler option

guide-data-trans, Qguide-data-trans

Lets you set a level of guidance for data transformation.

Syntax

Linux OS:

`-guide-data-trans[=`*n*`]`

macOS:

`-guide-data-trans[=`*n*`]`

Windows OS:

`/Qguide-data-trans[:`*n*`]`

Arguments

n	Is an optional value specifying the level of guidance to be provided. The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for data transformation.

Description

This option lets you set a level of guidance for data transformation. It causes the compiler to generate messages suggesting ways to improve that optimization.

IDE Equivalent

None

Alternate Options

None

See Also

`guide, Qguide` compiler option

`guide-par, Qguide-par` compiler option

`guide-vec, Qguide-vec` compiler option

`guide-file, Qguide-file` compiler option

guide-file, Qguide-file

Causes the results of guided auto parallelism to be output to a file.

Syntax

Linux OS:

`-guide-file[=`*`filename`*`]`

macOS:

`-guide-file[=`*`filename`*`]`

Windows OS:

`/Qguide-file[:`*`filename`*`]`

Arguments

filename	Is the name of the file for output. It can include a path.

Default

OFF	Messages that are generated by guided auto parallelism are output to stderr.

Description

This option causes the results of guided auto parallelism to be output to a file.

This option is ignored unless you also specify one or more of the following options:

- `[Q]guide`
- `[Q]guide-vec`
- `[Q]guide-data-trans`
- `[Q]guide-par`

If you do not specify a path, the file is placed in the current working directory.

If there is already a file named *filename*, it will be overwritten.

You can include a file extension in *filename*. For example, if *file.txt* is specified, the name of the output file is *file.txt*. If you do not provide a file extension, the name of the file is *filename.guide*.

If you do not specify *filename*, the name of the file is *name-of-the-first-source-file.guide*. This is also the name of the file if the name specified for *filename* conflicts with a source file name provided in the command line.

NOTE
If you specify the `[Q]guide-file` option and you also specify option `[Q]guide-file-append`, the last option specified on the command line takes precedence.

IDE Equivalent

Windows

Visual Studio: **Diagnostics > Emit Guided Auto Parallelism Diagnostics to File**

Diagnostics > Guided Auto Parallelism Diagnostics File

Linux

Eclipse: **Compilation Diagnostics > Emit Guided Auto Parallelism diagnostics to File**

Compilation Diagnostics > Guided Auto Parallelism Report File

OS X

Xcode: **Diagnostics > Emit Guided Auto Parallelism diagnostics to File**

Diagnostics > Guided Auto Parallelism Report File

Alternate Options

None

Example

The following example shows how to cause guided auto parallelism messages to be output to a file named *my_guided_autopar.guide*:

```
-guide-file=my_guided_autopar       ! Linux and macOS systems
/Qguide-file:my_guided_autopar      ! Windows systems
```

See Also

`guide, Qguide` compiler option
`guide-file-append, Qguide-file-append` compiler option

guide-file-append, Qguide-file-append

Causes the results of guided auto parallelism to be appended to a file.

Syntax

Linux OS:

`-guide-file-append[=`*`filename`*`]`

macOS:

`-guide-file-append[=`*`filename`*`]`

Windows OS:

`/Qguide-file-append[:`*`filename`*`]`

Arguments

filename	Is the name of the file to be appended to. It can include a path.

Default

OFF	Messages that are generated by guided auto parallelism are output to stderr.

Description

This option causes the results of guided auto parallelism to be appended to a file.

This option is ignored unless you also specify one or more of the following options:

- `[Q]guide`
- `[Q]guide-vec`
- `[Q]guide-data-trans`
- `[Q]guide-par`

If you do not specify a path, the compiler looks for *filename* in the current working directory.

If *filename* is not found, then a new file with that name is created in the current working directory.

If you do not specify a file extension, the name of the file is *filename.guide*.

If the name specified for *filename* conflicts with a source file name provided in the command line, the name of the file is *name-of-the-first-source-file.guide*.

> **NOTE**
> If you specify the `[Q]guide-file-append` option and you also specify option `[Q]guide-file`, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to cause guided auto parallelism messages to be appended to a file named *my_messages.txt*:

```
-guide-file-append=my_messages.txt      ! Linux and macOS systems
/Qguide-file-append:my_messages.txt     ! Windows systems
```

See Also

`guide, Qguide` compiler option

`guide-file, Qguide-file` compiler option

guide-opts, Qguide-opts

Tells the compiler to analyze certain code and generate recommendations that may improve optimizations.

Syntax

Linux OS:

`-guide-opts=`*`string`*

macOS:

`-guide-opts=`*`string`*

Windows OS:

`/Qguide-opts:`*`string`*

Arguments

string

Is the text denoting the code to analyze. The string must appear within quotes. It can take one or more of the following forms:

filename
filename, *routine*
filename, *range* [, *range*]...
filename, *routine*, *range* [, *range*]...

If you specify more than one of the above forms in a string, a semicolon must appear between each form. If you specify more than one *range* in a string, a comma must appear between each *range*. Optional blanks can follow each parameter in the forms above and they can also follow each form in a string.

filename

Specifies the name of a file to be analyzed. It can include a path.

If you do not specify a path, the compiler looks for filename in the current working directory.

routine

Specifies the name of a routine to be analyzed. You can include an identifying parameter.

The name, including any parameter, must be enclosed in single quotes.

The compiler tries to uniquely identify the routine that corresponds to the specified routine name. It may select multiple routines to analyze, especially if the following is true:

- More than one routine has the specified routine name, so the routine cannot be uniquely identified.
- No parameter information has been specified to narrow the number of routines selected as matches.

range

Specifies a range of line numbers to analyze in the file or routine specified. The *range* must be specified in integers in the form:

first_line_number-last_line_number

The hyphen between the line numbers is required.

Default

OFF	You do not receive guidance on how to improve optimizations. However, if you specify the `[Q]guide` option, the compiler analyzes and generates recommendations for all the code in an application

Description

This option tells the compiler to analyze certain code and generate recommendations that may improve optimizations.

This option is ignored unless you also specify one or more of the following options:

- `[Q]guide`
- `[Q]guide-vec`
- `[Q]guide-data-trans`
- `[Q]guide-par`

When the `[Q]guide-opts` option is specified, a message is output that includes which parts of the input files are being analyzed. If a routine is selected to be analyzed, the complete routine name will appear in the generated message.

When inlining is involved, you should specify callee line numbers. Generated messages also use callee line numbers.

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Guided Auto Parallelism Code Selection Options**

Eclipse

Eclipse: **Compilation Diagnostics > Guided Auto Parallelism Code Selection**

Xcode

Xcode: **Diagnostics > Guided Auto Parallelism Code Selection**

Alternate Options

None

Example

Consider the following:

```
Linux*: -guide-opts="v.c, 1-10; v2.c, 1-40, 50-90, 100-200; v5.c, 300-400; x.c, 'funca(int)',
22-44, 55-77, 88-99; y.c, 'funcb'"
```

```
Windows*: /Qguide-opts:"v.c, 1-10; v2.c, 1-40, 50-90, 100-200; v5.c, 300-400; x.c, 'funca(int)',
22-44, 55-77, 88-99; y.c, 'funcb'"
```

The above command causes the following to be analyzed:

file v.c, line numbers 1 to 10
file v2.c, line numbers 1 to 40, 50 to 90, and 100 to 200
file v5.c, line numbers 300 to 400

file x.c, routine funca with parameter (int), line numbers 22 to 44, 55 to 77, and 88 to 99
file y.c, routine funcb

See Also

`guide, Qguide` compiler option

`guide-data-trans, Qguide-data-trans` compiler option

`guide-par, Qguide-par` compiler option

`guide-vec, Qguide-vec` compiler option

`guide-file, Qguide-file` compiler option

guide-par, Qguide-par

Lets you set a level of guidance for auto parallelism.

Syntax

Linux OS:

`-guide-par[=n]`

macOS:

`-guide-par[=n]`

Windows OS:

`/Qguide-par[:n]`

Arguments

n	Is an optional value specifying the level of guidance to be provided. The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for parallelism.

Description

This option lets you set a level of guidance for auto parallelism. It causes the compiler to generate messages suggesting ways to improve that optimization.

You must also specify the `[Q]parallel` option to receive auto parallelism guidance.

IDE Equivalent

None

Alternate Options

None

See Also

`guide, Qguide` compiler option

`guide-data-trans, Qguide-data-trans` compiler option

`guide-vec, Qguide-vec` compiler option

`guide-file, Qguide-file` compiler option

guide-vec, Qguide-vec

Lets you set a level of guidance for auto-vectorization.

Syntax

Linux OS:

`-guide-vec[=n]`

macOS:

`-guide-vec[=n]`

Windows OS:

`/Qguide-vec[:n]`

Arguments

n	Is an optional value specifying the level of guidance to be provided. The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for vectorization.

Description

This option lets you set a level of guidance for auto-vectorization. It causes the compiler to generate messages suggesting ways to improve that optimization.

IDE Equivalent

None

Alternate Options

None

See Also

`guide, Qguide` compiler option
`guide-data-trans, Qguide-data-trans` compiler option
`guide-par, Qguide-par` compiler option
`guide-file, Qguide-file` compiler option

ipp-link, Qipp-link

Controls whether the compiler links to static or dynamic threaded Intel® Integrated Performance Primitives (Intel® IPP) run-time libraries.

Syntax

Linux OS:

`-ipp-link[=lib]`

macOS:

`-ipp-link[=lib]`

Windows OS:

`/Qipp-link[:lib]`

Arguments

lib	Specifies the Intel® IPP library to use. Possible values are:	
	`static`	Tells the compiler to link to the set of static run-time libraries.
	`dynamic`	Tells the compiler to link to the set of dynamic threaded run-time libraries.

Default

`dynamic`	The compiler links to the Intel® IPP set of dynamic run-time libraries. However, if Linux* option `-static` is specified, the compiler links to the set of static run-time libraries.

Description

This option controls whether the compiler links to static or dynamic threaded Intel® Integrated Performance Primitives (Intel® IPP) run-time libraries.

To use this option, you must also specify the `[Q]ipp` option.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`ipp, Qipp` compiler option

qdaal, Qdaal

Tells the compiler to link to certain libraries in the Intel® oneAPI Data Analytics Library (oneDAL).

Syntax

Linux OS:

`-qdaal[=lib]`

macOS:

`-qdaal[=lib]`

Windows OS:

`/Qdaal[:lib]`

Arguments

lib	Indicates which oneDAL library files should be linked. Possible values are:
	`parallel` — Tells the compiler to link using the threaded oneDAL libraries. This is the default if the option is specified with no *lib*.
	`sequential` — Tells the compiler to link using the non-threaded oneDAL libraries.

Default

OFF	The compiler does not link to the oneDAL.

Description

This option tells the compiler to link to certain libraries in the Intel® oneAPI Data Analytics Library (oneDAL).

On Linux* and macOS systems, the associated oneDAL headers are included when you specify this option.

> **NOTE**
> On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.
>
> On Linux* and macOS systems, the driver must add the library names explicitly to the link command. You must use option `-qdaal` to perform the link to pull in the dependent libraries.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Performance Library Build Components -> Use Intel® oneAPI Data Analytics Library**

Xcode

Xcode: **Performance Library Build Components -> Use Intel® oneAPI Data Analytics Library**

Alternate Options

Linux: `-daal` (this is a deprecated option)

See Also

Using Intel® Performance Libraries

qipp, Qipp

Tells the compiler to link to some or all of the Intel® Integrated Performance Primitives (Intel® IPP) libraries.

Syntax

Linux OS:

`-qipp[=`*`lib`*`]`

macOS:

`-qipp[=`*`lib`*`]`

Windows OS:

`/Qipp[:`*`lib`*`]`

Arguments

lib	Indicates the Intel® IPP libraries that the compiler should link to. Possible values are:	
	`common`	Tells the compiler to link using the main libraries set. This is the default if the option is specified with no *lib*.
	`crypto`	Tells the compiler to link using the Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography) libraries.
	`nonpic` (Linux* only)	Tells the compiler to link using the version of the libraries that do not have position-independent code.
	`nonpic_crypto` (Linux only)	Tells the compiler to link using the Intel® IPP Cryptography libraries. It uses the version of the libraries that do not have position-independent code.

Default

OFF	The compiler does not link to the Intel® IPP libraries.

Description

The option tells the compiler to link to some or all of the Intel® IPP libraries and include the Intel® IPP headers.

The `[Q]ipp-link` option controls whether the compiler links to static, dynamic threaded, or static threaded Intel® IPP runtime libraries.

> **NOTE**
> On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.
>
> On Linux* and macOS systems, the driver must add the library names explicitly to the link command. You must use option `qipp` to perform the link to pull in the dependent libraries.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Performance Library Build Components > Use Intel(R) Integrated Performance Primitives Libraries**

Xcode

Xcode: **Performance Library Build Components > Use Intel® Integrated Performance Primitives Libraries**

Alternate Options

Linux: `-qipp` (this is a deprecated option)

See Also

`ipp-link, Qipp-link` compiler option

qmkl, Qmkl

Tells the compiler to link to certain libraries in the Intel® oneAPI Math Kernel Library (oneMKL). On Windows systems, you must specify this option at compile time.

Syntax

Linux OS:

-qmkl[=*lib*]

macOS:

`-qmkl[=lib]`

Windows OS:

`/Qmkl[:lib]`

Arguments

lib Indicates which oneMKL library files should be linked. Possible values are:

`parallel`	Tells the compiler to link using the threaded libraries in oneMKL. This is the default if the option is specified with no *lib*.
`sequential`	Tells the compiler to link using the sequential libraries in oneMKL.
`cluster`	Tells the compiler to link using the cluster-specific libraries and the sequential libraries in oneMKL. Cluster-specific libraries are not available for macOS.

Default

OFF	The compiler does not link to the oneMKL library.

Description

This option tells the compiler to link to certain libraries in the Intel® oneAPI Math Kernel Library (oneMKL).

On Linux* and macOS systems, dynamic linking is the default when you specify `-mkl`. To link with oneMKL statically, you must specify:

```
-qmkl -static-intel
```

On Windows* systems, static linking is the default when you specify `/Qmkl`. To link with oneMKL dynamically, you must specify:

```
/Qmkl /MD
```

For more information about using oneMKL libraries, see the article titled: Intel® oneAPI Math Kernel Library Link Line Advisor.

NOTE
On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. On Linux* and macOS systems, the driver must add the library names explicitly to the link command.

NOTE
If you specify option `[q or Q]mkl`, or `-qmkl=parallel` or `/Qmkl:parallel`, and you also specify option `[Q]tbb`, the compiler links to the standard threaded version of oneMKL.

However, if you specify `[q or Q]mkl`, or `-qmkl=parallel` or `/Qmkl:parallel`, and you also specify option `[Q]tbb` and option `[q or Q]openmp`, the compiler links to the OpenMP* threaded version of oneMKL.

IDE Equivalent

Visual Studio: None

Eclipse: **Performance Library Build Components > Use Intel® oneAPI Math Kernel Library**

Xcode: **Performance Library Build Components > Use Intel® oneAPI Math Kernel Library**

Alternate Options

Linux and macOS: `-mkl` (this is a deprecated option)

See Also

`static-intel` compiler option
`MD` compiler option

qopt-args-in-regs, Qopt-args-in-regs

Determines whether calls to routines are optimized by passing parameters in registers instead of on the stack. This is a deprecated option that may be removed in a future release.

Architecture Restrictions

Only available on IA-32 architecture. IA-32 support has been deprecated, and will be removed in a future release.

Syntax

Linux OS:

`-qopt-args-in-regs[=`*`keyword`*`]`

macOS:

`None`

Windows OS:

`/Qopt-args-in-regs[:`*`keyword`*`]`

Arguments

keyword Specifies whether the optimization should be performed and under what conditions. Possible values are:

`none`	The optimization is not performed. No parameters are passed in registers. They are put on the stack.
`seen`	Causes parameters to be passed in registers when they are passed to routines whose definition can be seen in the same compilation unit.
`all`	Causes parameters to be passed in registers, whether they are passed to routines whose definition can be seen in the same compilation unit, or not. This value is only available on Linux* systems.

Default

`-qopt-args-in-regs=seen` or `/Qopt-args-in-regs:seen`	Parameters are passed in registers when they are passed to routines whose definition is seen in the same compilation unit.

Description

This option determines whether calls to routines are optimized by passing parameters in registers instead of on the stack. It also indicates the conditions when the optimization will be performed.

This is a deprecated option that may be removed in a future release. There is no replacement option.

This option can improve performance for Application Binary Interfaces (ABIs) that require parameters to be passed in memory and compiled without interprocedural optimization (IPO).

Note that on Linux* systems, if `all` is specified, a small overhead may be paid when calling "unseen" routines that have not been compiled with the same option. This is because the call will need to go through a "thunk" to ensure that parameters are placed back on the stack where the callee expects them.

IDE Equivalent

None

Alternate Options

None

qopt-assume-safe-padding, Qopt-assume-safe-padding

Determines whether the compiler assumes that variables and dynamically allocated memory are padded past the end of the object.

Architecture Restrictions

Only available on all architectures that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions

Syntax

Linux OS:

`-qopt-assume-safe-padding`

`-qno-opt-assume-safe-padding`

macOS:

`-qopt-assume-safe-padding`

`-qno-opt-assume-safe-padding`

Windows OS:

`/Qopt-assume-safe-padding`

`/Qopt-assume-safe-padding-`

Arguments

None

Default

`-qno-opt-assume-safe-padding` or `/Qopt-assume-safe-padding-`	The compiler will not assume that variables and dynamically allocated memory are padded past the end of the object. It will adhere to the sizes specified in your program.

Description

This option determines whether the compiler assumes that variables and dynamically allocated memory are padded past the end of the object.

When you specify option `[q or Q]opt-assume-safe-padding`, the compiler assumes that variables and dynamically allocated memory are padded. This means that code can access up to 64 bytes beyond what is specified in your program.

The compiler does not add any padding for static and automatic objects when this option is used, but it assumes that code can access up to 64 bytes beyond the end of the object, wherever the object appears in the program. To satisfy this assumption, you must increase the size of static and automatic objects in your program when you use this option.

This option may improve performance of memory operations.

IDE Equivalent

None

Alternate Options

None

qopt-block-factor, Qopt-block-factor

Lets you specify a loop blocking factor.

Syntax

Linux OS:

`-qopt-block-factor=`*n*

macOS:

`-qopt-block-factor=`*n*

Windows OS:

`/Qopt-block-factor:`*n*

Arguments

n	Is the blocking factor. It must be an integer. The compiler may ignore the blocking factor if the value is 0 or 1.

Default

OFF	The compiler uses default heuristics for loop blocking.

Description

This option lets you specify a loop blocking factor.

IDE Equivalent

Windows

Visual Studio: **Diagnostics > Optimization Diagnostic File**

Diagnostics > Emit Optimization Diagnostics to File

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

qopt-calloc

Tells the compiler to substitute a call to _intel_fast_calloc() for a call to calloc().

Syntax

Linux OS:

`-qopt-calloc`

`-qno-opt-calloc`

macOS:

None

Windows OS:

None

Arguments

None

Default

`-qno-opt-calloc`	The compiler does not substitute a call to _intel_fast_calloc() for a call to calloc().

Description

This option tells the compiler to substitute a call to_intel_fast_calloc() for a call to calloc().

This option may increase the performance of long-running programs that use calloc() frequently. It is recommended for these programs over combinations of options `-inline-calloc` and `-qopt-malloc-options=3` because this option causes less memory fragmentation.

> **NOTE**
> Many routines in the LIBIRC library are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

IDE Equivalent

None

Alternate Options

None

qopt-class-analysis, Qopt-class-analysis

Determines whether C++ class hierarchy information is used to analyze and resolve C++ virtual function calls at compile time.

Syntax

Linux OS and macOS:

`-qopt-class-analysis`

`-qno-opt-class-analysis`

Windows OS:

`/Qopt-class-analysis`

`/Qopt-class-analysis-`

Arguments

None

Default

`-qno-opt-class-analysis` or `/Qopt-class-analysis-`	C++ class hierarchy information is not used to analyze and resolve C++ virtual function calls at compile time.

Description

This option determines whether C++ class hierarchy information is used to analyze and resolve C++ virtual function calls at compile time. The option is turned on by default with the `-ipo` compiler option, enabling improved C++ optimization. If a C++ application contains non-standard C++ constructs, such as pointer down-casting, it may result in different behaviors.

IDE Equivalent

None

Alternate Options

None

qopt-dynamic-align, Qopt-dynamic-align

Enables or disables dynamic data alignment optimizations.

Syntax

Linux OS:

`-qopt-dynamic-align`

`-qno-opt-dynamic-align`

macOS:

`-qopt-dynamic-align`

`-qno-opt-dynamic-align`

Windows OS:

`/Qopt-dynamic-align`

`/Qopt-dynamic-align-`

Arguments

None

Default

`-qopt-dynamic-align` or `/Qopt-dynamic-align`	The compiler may generate code dynamically dependent on alignment. It may do optimizations based on data location for the best performance. The result of execution on some algorithms may depend on data layout.

Description

This option enables or disables dynamic data alignment optimizations.

If you specify `-qno-opt-dynamic-align` or `/Qopt-dynamic-align-`, the compiler generates no code dynamically dependent on alignment. It will not do any optimizations based on data location and results will depend on the data values themselves.

When you specify `[q or Q]qopt-dynamic-align`, the compiler may implement conditional optimizations based on dynamic alignment of the input data. These dynamic alignment optimizations may result in different bitwise results for aligned and unaligned data with the same values.

Dynamic alignment optimizations can improve the performance of some vectorized code, especially for long trip count loops, but there is an associated cost of increased code size and compile time. Disabling such optimizations can improve the performance of some other vectorized code. It may also improve bitwise reproducibility of results, factoring out data location from possible sources of discrepancy.

IDE Equivalent

None

Alternate Options

None

qopt-jump-tables, Qopt-jump-tables

Enables or disables generation of jump tables for switch statements.

Syntax

Linux OS:

`-qopt-jump-tables=`*`keyword`*

`-qno-opt-jump-tables`

macOS:

`-qopt-jump-tables=`*`keyword`*

`-qno-opt-jump-tables`

Windows OS:

`/Qopt-jump-tables:`*`keyword`*

`/Qopt-jump-tables-`

Arguments

keyword	Is the instruction for generating jump tables. Possible values are:	
	`never`	Tells the compiler to never generate jump tables. All switch statements are implemented as chains of if-then-elses. This is the same as specifying `-qno-opt-jump-tables` (Linux* and macOS) or `/Qopt-jump-tables-` (Windows*).
	`default`	The compiler uses default heuristics to determine when to generate jump tables.
	`large`	Tells the compiler to generate jump tables up to a certain pre-defined size (64K entries).
	`n`	Must be an integer. Tells the compiler to generate jump tables up to *n* entries in size.

Default

`-qopt-jump-tables=default` or `/Qopt-jump-tables:default`	The compiler uses default heuristics to determine when to generate jump tables for switch statements.

Description

This option enables or disables generation of jump tables for switch statements. When the option is enabled, it may improve performance for programs with large switch statements.

IDE Equivalent

None

Alternate Options

None

qopt-malloc-options

Lets you specify an alternate algorithm for malloc().

Syntax

Linux OS:

`-qopt-malloc-options=`*`n`*

macOS:

`-qopt-malloc-options=`*`n`*

Windows OS:

`None`

Arguments

n	Specifies the algorithm to use for malloc(). Possible values are:

`0`	Tells the compiler to use the default algorithm for malloc(). This is the default.
`1`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=2 and M_TRIM_THRESHOLD=0x10000000.
`2`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=2 and M_TRIM_THRESHOLD=0x40000000.
`3`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=0 and M_TRIM_THRESHOLD=-1.
`4`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=0, M_TRIM_THRESHOLD=-1, M_TOP_PAD=4096.

Default

`-qopt-malloc-options=0`	The compiler uses the default algorithm when malloc() is called. No call is made to mallopt().

Description

This option lets you specify an alternate algorithm for malloc().

If you specify a non-zero value for *n*, it causes alternate configuration parameters to be set for how malloc() allocates and frees memory. It tells the compiler to insert calls to mallopt() to adjust these parameters to malloc() for dynamic memory allocation. This may improve speed.

IDE Equivalent

None

Alternate Options

None

See Also

malloc(3) man page

mallopt function (defined in malloc.h)

qopt-matmul, Qopt-matmul

Enables or disables a compiler-generated Matrix Multiply (matmul) library call.

Syntax

Linux OS:

`-qopt-matmul`

`-qno-opt-matmul`

macOS:

`None`

Windows OS:

`/Qopt-matmul`

`/Qopt-matmul-`

Arguments

None

Default

`-qno-opt-matmul` or `/Qopt-matmul-`	The matmul library call optimization does not occur unless this option is enabled or certain other compiler options are specified (see below).

Description

This option enables or disables a compiler-generated Matrix Multiply (MATMUL) library call.

The `[q or Q]opt-matmul` option tells the compiler to identify matrix multiplication loop nests (if any) and replace them with a matmul library call for improved performance. The resulting executable may improve performance on Intel® microprocessors.

> **NOTE**
> This option is dependent upon the OpenMP* library. If your product does not support OpenMP, this option will have no effect.

This option has no effect unless option `O2` or higher is set.

> **NOTE**
> Many routines in the MATMUL library are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

IDE Equivalent

Visual Studio

Visual Studio: **Optimization > Enable Matrix Multiply Library Call**

Eclipse

Eclipse: **Optimization > Optimize Matrix Multiplication**

Xcode

Xcode: None

Alternate Options

None

See Also

- `O` compiler option

qopt-mem-layout-trans, Qopt-mem-layout-trans

Controls the level of memory layout transformations performed by the compiler.

Syntax

Linux OS:

`-qopt-mem-layout-trans[=n]`

`-qno-opt-mem-layout-trans`

macOS:

`-qopt-mem-layout-trans[=n]`

`-qno-opt-mem-layout-trans`

Windows OS:

`/Qopt-mem-layout-trans[:n]`

`/Qopt-mem-layout-trans-`

Arguments

n Is the level of memory layout transformations. Possible values are:

0	Disables memory layout transformations. This is the same as specifying `-qno-opt-mem-layout-trans` (Linux* or macOS) or `/Qopt-mem-layout-trans-` (Windows*).
1	Enables basic memory layout transformations.
2	Enables more memory layout transformations. This is the same as specifying `[q or Q]opt-mem-layout-trans` with no argument.
3	Enables more memory layout transformations like copy-in/copy-out of structures for a region of code. You should only use this setting if your system has more than 4GB of physical memory per core.
4	Enables more aggressive memory layout transformations. You should only use this setting if your system has more than 4GB of physical memory per core.

Default

`-qopt-mem-layout-trans=2` or `/Qopt-mem-layout-trans:2`	The compiler performs moderate memory layout transformations.

Description

This option controls the level of memory layout transformations performed by the compiler. This option can improve cache reuse and cache locality.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.   Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

qopt-multi-version-aggressive, Qopt-multi-version-aggressive

Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.

Syntax

Linux OS:

`-qopt-multi-version-aggressive`

`-qno-opt-multi-version-aggressive`

macOS:

`-qopt-multi-version-aggressive`

`-qno-opt-multi-version-aggressive`

Windows OS:

`/Qopt-multi-version-aggressive`

`/Qopt-multi-version-aggressive-`

Arguments

None

Default

`-qno-opt-multi-version-aggressive` or `/Qopt-multi-version-aggressive-`	The compiler uses default heuristics when checking for pointer aliasing and scalar replacement.

Description

This option tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement. This option may improve performance.

The performance can be affected by certain options, such as `/arch` or `/Qx` (Windows*) or `-m` or `-x` (Linux* and macOS).

IDE Equivalent

None

Alternate Options

None

qopt-multiple-gather-scatter-by-shuffles, Qopt-multiple-gather-scatter-by-shuffles

Enables or disables the optimization for multiple adjacent gather/scatter type vector memory references.

Syntax

Linux OS:

`-qopt-multiple-gather-scatter-by-shuffles`

`-qno-opt-multiple-gather-scatter-by-shuffles`

macOS:

`-qopt-multiple-gather-scatter-by-shuffles`

`-qno-opt-multiple-gather-scatter-by-shuffles`

Windows OS:

`/Qopt-multiple-gather-scatter-by-shuffles`

`/Qopt-multiple-gather-scatter-by-shuffles-`

Arguments

None

Default

varies	When this option is not specified, the compiler uses default heuristics for optimization.

Description

This option controls the optimization for multiple adjacent gather/scatter type vector memory references. This optimization hint is useful for performance tuning. It tries to generate more optimal software sequences using shuffles.

If you specify this option, the compiler will apply the optimization heuristics. If you specify `-qno-opt-multiple-gather-scatter-by-shuffles` or `/Qopt-multiple-gather-scatter-by-shuffles-`, the compiler will not apply the optimization.

> **NOTE**
> Optimization is affected by optimization compiler options, such as `[Q]x`, `-march` (Linux* or macOS), and `/arch` (Windows*).

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

pragma vector
`x, Qx` compiler option
`march` compiler option
`arch` compiler option

qopt-prefetch, Qopt-prefetch

Enables or disables prefetch insertion optimization.

Syntax

Linux OS:

`-qopt-prefetch[=n]`

`-qno-opt-prefetch`

macOS:

`-qopt-prefetch[=n]`

`-qno-opt-prefetch`

Windows OS:

`/Qopt-prefetch[:n]`

`/Qopt-prefetch-`

Arguments

n	Is the level of software prefetching optimization desired. Possible values are:	
	`0`	Disables software prefetching. This is the same as specifying `-qno-opt-prefetch` (Linux* and macOS) or `/Qopt-prefetch-` (Windows*).
	`1` to `5`	Enables different levels of software prefetching. If you do not specify a value for *n*, the default is `2`. Use lower values to reduce the amount of prefetching.

Default

`-qno-opt-prefetch` or `/Qopt-prefetch-`	Prefetch insertion optimization is disabled.

Description

This option enables or disables prefetch insertion optimization. The goal of prefetching is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache.

This option enables prefetching when higher optimization levels are specified.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Optimization > Enable Prefetch Insertion**

Xcode

Xcode: **Optimization > Enable Prefetch Insertion**

Alternate Options

None

See Also

qopt-prefetch-distance, Qopt-prefetch-distance compiler option

qopt-prefetch-distance, Qopt-prefetch-distance

Specifies the prefetch distance to be used for compiler-generated prefetches inside loops.

Syntax

Linux OS:

`-qopt-prefetch-distance=`*n1*`[, `*n2*`]`

macOS:

`None`

Windows OS:

`/Qopt-prefetch-distance:`*n1*`[, `*n2*`]`

Arguments

n1, n2	Is the prefetch distance in terms of the number of (possibly-vectorized) iterations. Possible values are non-negative numbers >=0. *n2* is optional. *n1* = 0 turns off all compiler issued prefetches from memory to L2. *n2* = 0 turns off all compiler issued prefetches from L2 to L1. If *n2* is specified and *n1* > 0, *n1* should be >= *n2*.

Default

OFF	The compiler uses default heuristics to determine the prefetch distance.

Description

This option specifies the prefetch distance to be used for compiler-generated prefetches inside loops. The unit (*n1* and optionally *n2*) is the number of iterations. If the loop is vectorized by the compiler, the unit is the number of vectorized iterations.

The value of *n1* will be used as the distance for prefetches from memory to L2 (for example, the vprefetch1 instruction). If *n2* is specified, it will be used as the distance for prefetches from L2 to L1 (for example, the vprefetch0 instruction).

This option is ignored if option `-qopt-prefetch=0` (Linux*) or `/Qopt-prefetch:0` (Windows*) is specified.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following Linux* examples:

```
-qopt-prefetch-distance=64,32
```

The above causes the compiler to use a distance of 64 iterations for memory to L2 prefetches, and a distance of 32 iterations for L2 to L1 prefetches.

```
-qopt-prefetch-distance=24
```

The above causes the compiler to use a distance of 24 iterations for memory to L2 prefetches. The distance for L2 to L1 prefetches will be determined by the compiler.

```
-qopt-prefetch-distance=0,4
```

The above turns off all memory to L2 prefetches inserted by the compiler inside loops. The compiler will use a distance of 4 iterations for L2 to L1 prefetches.

```
-qopt-prefetch-distance=16,0
```

The above causes the compiler to use a distance of 16 iterations for memory to L2 prefetches. No L2 to L1 loop prefetches are issued by the compiler.

See Also

qopt-prefetch, Qopt-prefetch compiler option
prefetch pragma

qopt-prefetch-issue-excl-hint, Qopt-prefetch-issue-excl-hint

Supports the prefetchW instruction in Intel® microarchitecture code name Broadwell and later.

Syntax

Linux OS:

`-qopt-prefetch-issue-excl-hint`

macOS:

None

Windows OS:

`/Qopt-prefetch-issue-excl-hint`

Arguments

None

Default

OFF	The compiler does not support the PREFETCHW instruction for this microarchitecture.

Description

This option supports the PREFETCHW instruction in Intel® microarchitecture code name Broadwell and later.

When you specify this option, you must also specify option `[q or Q]opt-prefetch`.

The prefetch instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates any other cached copy in anticipation of the line being written to in the future.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-prefetch/Qopt-prefetch` compiler option

qopt-ra-region-strategy, Qopt-ra-region-strategy

Selects the method that the register allocator uses to partition each routine into regions.

Syntax

Linux OS:

`-qopt-ra-region-strategy[=`*keyword*`]`

macOS:

`-qopt-ra-region-strategy[=`*keyword*`]`

Windows OS:

`/Qopt-ra-region-strategy[:`*keyword*`]`

Arguments

keyword Is the method used for partitioning. Possible values are:

`routine`	Creates a single region for each routine.
`block`	Partitions each routine into one region per basic block.
`trace`	Partitions each routine into one region per trace.
`loop`	Partitions each routine into one region per loop.
`default`	The compiler determines which method is used for partitioning.

Default

`-qopt-ra-region-strategy=default` or `/Qopt-ra-region-strategy:default`	The compiler determines which method is used for partitioning. This is also the default if `keyword` is not specified.

Description

This option selects the method that the register allocator uses to partition each routine into regions.

When setting `default` is in effect, the compiler attempts to optimize the tradeoff between compile-time performance and generated code performance.

This option is only relevant when optimizations are enabled (option `O1` or higher).

IDE Equivalent

None

Alternate Options

None

See Also

- O compiler option

qopt-streaming-stores, Qopt-streaming-stores

Enables generation of streaming stores for optimization.

Syntax

Linux OS and macOS:

`-qopt-streaming-stores=`*`keyword`*

`-qno-opt-streaming-stores`

Windows OS:

`/Qopt-streaming-stores:`*`keyword`*

`/Qopt-streaming-stores-`

Arguments

keyword Specifies whether streaming stores are generated. Possible values are:

`always`	Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound. When this option setting is specified, it is your responsibility to also insert any fences as required to ensure correct memory ordering within a thread or across threads. One typical way to do this is to insert a _mm_sfence() intrinsic call just after the

	loops (such as the initialization loop) where the compiler may insert streaming store instructions.
`never`	Disables generation of streaming stores for optimization. Normal stores are performed. This is the same as specifying `-qno-opt-streaming-stores` (Linux*) or `/Qopt-streaming-stores-` (Windows*).
`auto`	Lets the compiler decide which instructions to use.

Default

`-qopt-streaming-stores=auto` or `/Qopt-streaming-stores:auto`	The compiler decides whether to use streaming stores or normal stores.

Description

This option enables generation of streaming stores for optimization. This method stores data with instructions that use a non-temporal buffer, which minimizes memory hierarchy pollution.

This option may be useful for applications that can benefit from streaming stores.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows a way to insert fences when specifying `-qopt-streaming-stores=always` or `/Qopt-streaming-stores:always`:

```
void simple1(double * restrict a, double * restrict b, double * restrict c, double *d, int n)
{
    int i, j;

#pragma omp parallel for
      for (j=0; j<n; j++) {
        a[j] = 1.0;
        b[j] = 2.0;
        c[j] = 0.0;
        }

      _mm_sfence(); // OR _mm_mfence();

#pragma omp parallel for
    for (i=0; i<n; i++)
        a[i] = a[i] + c[i]*b[i];
}
```

See Also

ax, Qax compiler option

x, Qx compiler option

qopt-subscript-in-range, Qopt-subscript-in-range

Determines whether the compiler assumes that there are no "large" integers being used or being computed inside loops.

Syntax

Linux OS:

`-qopt-subscript-in-range`

`-qno-opt-subscript-in-range`

macOS:

`-qopt-subscript-in-range`

`-qno-opt-subscript-in-range`

Windows OS:

`/Qopt-subscript-in-range`

`/Qopt-subscript-in-range-`

Arguments

None

Default

`-qno-opt-subscript-in-range` or `/Qopt-subscript-in-range-`	The compiler assumes there are "large" integers being used or being computed within loops.

Description

This option determines whether the compiler assumes that there are no "large" integers being used or being computed inside loops.

If you specify `[q or Q]opt-subscript-in-range`, the compiler assumes that there are no "large" integers being used or being computed inside loops. A "large" integer is typically > 2^{31}.

This feature can enable more loop transformations.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how these options can be useful. Variable m is declared as type long (64-bits) and all other variables inside the subscript are declared as type int (32-bits):

```
A[ i  + j  + ( n  + k) * m ]
```

qopt-zmm-usage, Qopt-zmm-usage

Defines a level of zmm registers usage.

Syntax

Linux OS:

`-qopt-zmm-usage=keyword`

macOS:

`-qopt-zmm-usage=keyword`

Windows OS:

`/Qopt-zmm-usage:keyword`

Arguments

keyword	Specifies the level of zmm registers usage. Possible values are:	
	`low`	Tells the compiler that the compiled program is unlikely to benefit from zmm registers usage. It specifies that the compiler should avoid using zmm registers unless it can prove the gain from their usage.
	`high`	Tells the compiler to generate zmm code without restrictions.

Default

varies	The default is low when you specify [Q]xCORE-AVX512. The default is high when you specify [Q]xCOMMON-AVX512.

Description

This option may provide better code optimization for Intel® processors that are on the Intel® microarchitecture formerly code-named Skylake.

This option defines a level of zmm registers usage. The `low` setting causes the compiler to generate code with zmm registers very carefully, only when the gain from their usage is proven. The `high` setting causes the compiler to use much less restrictive heuristics for zmm code generation.

It is not always easy to predict whether the `high` or the `low` setting will yield better performance. Programs that enjoy high performance gains from the use of xmm or ymm registers may expect performance improvement by moving to use zmm registers. However, some programs that use zmm registers may not gain as much or may even lose performance. We recommend that you try both option values to measure the performance of your programs.

This option is ignored if you do not specify an option that enables Intel® AVX-512, such as `[Q]xCORE-AVX512` or option `[Q]xCOMMON-AVX512`.

This option has no effect on loops that use pragma omp simd simdlen(n) or on functions that are generated by vector specifications specific to CORE-AVX512.

IDE Equivalent

None

Alternate Options

None

See Also

`x, Qx` compiler option

For more information about simd loops specification and vector function specification, see pragmas omp simd and omp declare simd in the OpenMP* TR4: Version 5.0 specification.

qoverride-limits, Qoverride-limits

Lets you override certain internal compiler limits that are intended to prevent excessive memory usage or compile times for very large, complex compilation units.

Syntax

Linux OS:

`-qoverride-limits`

macOS:

`-qoverride-limits`

Windows OS:

`/Qoverride-limits`

Arguments

None

Default

OFF	Certain internal compiler limits are not overridden. These limits are determined by default heuristics.

Description

This option provides a way to override certain internal compiler limits that are intended to prevent excessive memory usage or compile times for very large, complex compilation units.

If this option is not used and your program exceeds one of these internal compiler limits, some optimizations will be skipped to reduce the memory footprint and compile time. If you chose to create an optimization report by specifying `[q or Q]opt-report`, you may see a related diagnostic remark as part of the report.

Specifying this option may substantially increase compile time and/or memory usage.

NOTE
If you use this option, it is your responsibility to ensure that sufficient memory is available. If there is not sufficient available memory, the compilation may fail.

This option should only be used where there is a specific need; it is not recommended for inexperienced users.

IDE Equivalent

None

Alternate Options

None

qtbb, Qtbb

Tells the compiler to link to the Intel® oneAPI Threading Building Blocks (oneTBB) libraries.

Syntax

Linux OS:

`-qtbb`

macOS:

`-qtbb`

Windows OS:

`/Qtbb`

Arguments

None

Default

OFF	The compiler does not link to the oneTBB libraries.

Description

This option tells the compiler to link to the Intel® oneAPI Threading Building Blocks (oneTBB) libraries and include the oneTBB headers.

NOTE
On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. You do not need to specify a separate link command.

On Linux* and macOS systems, the driver must add the library names explicitly to the link command. You must use option `-qtbb` to perform the link to pull in the dependent libraries.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Performance Library Build Components > Use Intel® oneAPI Threading Building Blocks**

Xcode

Xcode: **Performance Library Build Components > Use Intel® oneAPI Threading Building Blocks**

Alternate Options

Linux: `-tbb` (this is a deprecated option)

Qvla

Determines whether variable length arrays are enabled.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qvla`

`/Qvla-`

Arguments

None

Default

`/Qvla-`	Variable length arrays are disabled.

Description

This option determines whether variable length arrays (a C99 feature) are enabled.

To enable variable length arrays, you must specify `/Qvla`.

IDE Equivalent

None

Alternate Options

None

scalar-rep, Qscalar-rep

Enables or disables the scalar replacement optimization done by the compiler as part of loop transformations.

Syntax

Linux OS:

`-scalar-rep`

`-no-scalar-rep`

macOS:

`-scalar-rep`

`-no-scalar-rep`

Windows OS:

`/Qscalar-rep`

`/Qscalar-rep-`

Arguments

None

Default

`-scalar-rep` or `/Qscalar-rep`	Scalar replacement is performed during loop transformation at optimization levels of O2 and above.

Description

This option enables or disables the scalar replacement optimization done by the compiler as part of loop transformations. This option takes effect only if you specify an optimization level of `O2` or higher.

IDE Equivalent

None

Alternate Options

None

See Also

O compiler option

simd, Qsimd

Enables or disables compiler interpretation of simd pragmas.

Syntax

Linux OS:

`-simd`

`-no-simd`

macOS:

`-simd`

`-no-simd`

Windows OS:

`/Qsimd`

`/Qsimd-`

Arguments

None

Default

`-simd`	SIMD pragmas are enabled.

or `/Qsimd`

Description

This option enables or disables compiler interpretation of simd pragmas.

To disable interpretation of simd pragmas, specify `-no-simd` (Linux* and macOS) or `/Qsimd-` (Windows*). Note that the compiler may still vectorize loops based on its own heuristics (leading to generation of SIMD instructions) even when `-no-simd` (or `/Qsimd-`) is specified.

To disable all compiler vectorization, use the "`-no-vec -no-simd`" (Linux* and macOS) or "`/Qvec- /Qsimd-`" (Windows*) compiler options. The option `-no-vec` (and `/Qvec-`) disables all auto-vectorization, including vectorization of array notation statements. The option `-no-simd` (and `/Qsimd-`) disables vectorization of loops that have simd pragmas.

> **NOTE**
> If you specify option `-mia32` (Linux*) or option `/arch:IA32` (Windows*), simd pragmas are disabled by default and vector instructions cannot be used. Therefore, you cannot explicitly enable SIMD pragmas by specifying option `[Q]simd`.

IDE Equivalent

None

Alternate Options

None

See Also

`vec, Qvec` compiler option
Function Annotations and the SIMD Directive for Vectorization
simd pragma

simd-function-pointers, Qsimd-function-pointers

Enables or disables pointers to simd-enabled functions.

Syntax

Linux OS and macOS:

`-simd-function-pointers`

`-no-simd-function-pointers`

Windows OS:

`/Qsimd-function-pointers`

`/Qsimd-function-pointers-`

Arguments

None

Default

`-no-simd-function-pointers` or `/Qsimd-function-pointers-`	Pointers to simd-enabled functions are disabled. Vector specifications can only be placed in function declarations and definitions.

Description

This option enables or disables pointers to simd-enabled functions.

When option `[Q]simd-function-pointers` is specified, it defines simd-enabled (vector) function pointers by placing vector specifications with all usual clauses in function pointer declarations. The vector specifications must be indicated in an attribute vector declaration or in pragma `omp declare simd`.

These pointers can enable indirect calls to appropriate vector versions of the function from a simd loop or another simd-enabled function.

IDE Equivalent

None

Alternate Options

None

unroll, Qunroll

Tells the compiler the maximum number of times to unroll loops.

Syntax

Linux OS:

`-unroll[=n]`

macOS:

`-unroll[=n]`

Windows OS:

`/Qunroll[:n]`

Arguments

n	Is the maximum number of times a loop can be unrolled. To disable loop enrolling, specify 0.

Default

`-unroll` or `/Qunroll`	The compiler uses default heuristics when unrolling loops.

Description

This option tells the compiler the maximum number of times to unroll loops.

If you do not specify *n*, the optimizer determines how many times loops can be unrolled.

IDE Equivalent

Windows

Visual Studio: **Optimization > Loop Unrolling**

Linux

Eclipse: **Optimization > Loop Unroll Count**

OS X

Xcode: **Optimization > Loop Unrolling**

Alternate Options

Linux and macOS: `-funroll-loops`

Windows: None

unroll-aggressive, Qunroll-aggressive

Determines whether the compiler uses more aggressive unrolling for certain loops.

Syntax

Linux OS:

`-unroll-aggressive`

`-no-unroll-aggressive`

macOS:

`-unroll-aggressive`

`-no-unroll-aggressive`

Windows OS:

`/Qunroll-aggressive`

`/Qunroll-aggressive-`

Arguments

None

Default

`-no-unroll-aggressive` or `/Qunroll-aggressive-`	The compiler uses default heuristics when unrolling loops.

Description

This option determines whether the compiler uses more aggressive unrolling for certain loops. The positive form of the option may improve performance.

This option enables aggressive, complete unrolling for loops with small constant trip counts.

IDE Equivalent

None

Alternate Options

None

use-intel-optimized-headers, Quse-intel-optimized-headers

Determines whether the performance headers directory is added to the include path search list.

Syntax

Linux OS:

`-use-intel-optimized-headers`

macOS:

`-use-intel-optimized-headers`

Windows OS:

`/Quse-intel-optimized-headers`

Arguments

None

Default

`-no-use-intel-optimized-headers` or `/Quse-intel-optimized-headers-`	The performance headers directory is not added to the include path search list.

Description

This option determines whether the performance headers directory is added to the include path search list.

The performance headers directory is added if you specify `[Q]use-intel-optimized-headers`. Appropriate libraries are also linked in, as needed, for proper functionality.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Windows

Visual Studio: **Optimization > Use Intel Optimized Headers**

Linux

Eclipse: **Preprocessor > Use Intel Optimized Headers**

OS X

Xcode: **Optimization > Use Intel Optimized Headers**

Alternate Options

None

See Also

Intel's valarray Implementation

vec, Qvec

Enables or disables vectorization.

Syntax

Linux OS:

`-vec`

`-no-vec`

macOS:

`-vec`

`-no-vec`

Windows OS:

`/Qvec`

`/Qvec-`

Arguments

None

Default

`-vec` or `/Qvec`	Vectorization is enabled if option `O2` or higher is in effect.

Description

This option enables or disables vectorization.

To disable vectorization, specify `-no-vec` (Linux* and macOS) or `/Qvec-` (Windows*).

To disable interpretation of SIMD pragmas, specify `-no-simd` (Linux* and macOS) or `/Qsimd-` (Windows*).

To disable all compiler vectorization, use the "`-no-vec -no-simd`" (Linux* and macOS) or "`/Qvec- /Qsimd-`" (Windows*) compiler options. The option `-no-vec` (and `/Qvec-`) disables all auto-vectorization, including vectorization of array notation statements. The option `-no-simd` (and `/Qsimd-`) disables vectorization of loops that have SIMD pragmas.

> **NOTE**
> Using this option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

IDE Equivalent

None

Alternate Options

None

See Also

`simd, Qsimd` compiler option
`ax, Qax` compiler option
`x, Qx` compiler option
`vec-guard-write, Qvec-guard-write` compiler option
`vec-threshold, Qvec-threshold` compiler option

vec-guard-write, Qvec-guard-write

Tells the compiler to perform a conditional check in a vectorized loop.

Syntax

Linux OS:

`-vec-guard-write`

`-no-vec-guard-write`

macOS:

`-vec-guard-write`

`-no-vec-guard-write`

Windows OS:

`/Qvec-guard-write`

`/Qvec-guard-write-`

Arguments

None

Default

`-vec-guard-write` or `/Qvec-guard-write`	The compiler performs a conditional check in a vectorized loop.

Description

This option tells the compiler to perform a conditional check in a vectorized loop. This checking avoids unnecessary stores and may improve performance.

IDE Equivalent

None

Alternate Options

None

vec-threshold, Qvec-threshold

Sets a threshold for the vectorization of loops.

Syntax

Linux OS:

`-vec-threshold[n]`

macOS:

`-vec-threshold[n]`

Windows OS:

`/Qvec-threshold[[:]n]`

Arguments

n	Is an integer whose value is the threshold for the vectorization of loops. Possible values are 0 through 100. If *n* is 0, loops get vectorized always, regardless of computation work volume. If *n* is 100, loops get vectorized when performance gains are predicted based on the compiler analysis data. Loops get vectorized only if profitable vector-level parallel execution is almost certain. The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to vectorize only if there is a 50% probability of the code speeding up if executed in vector form.

Default

`-vec-threshold100` or `/Qvec-threshold100`	Loops get vectorized only if profitable vector-level parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the vectorization of loops based on the probability of profitable execution of the vectorized loop in parallel.

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

IDE Equivalent

Windows

Visual Studio: **Optimization > Threshold For Vectorization**

Linux

Eclipse: **Optimization > Enable Maximum Vector-level Parallelism**

OS X

Xcode: **Optimization > Enable Maximum Vector-level Parallelism**

Alternate Options

None

vecabi, Qvecabi

Determines which vector function application binary interface (ABI) the compiler uses to create or call vector functions.

Syntax

Linux OS and macOS:

`-vecabi=`*`keyword`*

Windows OS:

`/Qvecabi:`*`keyword`*

Arguments

keyword	Specifies which vector function ABI to use. Possible values are:	
	`compat`	Tells the compiler to use the compatibility vector function ABI. This ABI includes Intel®-specific features.
	`cmdtarget`	Tells the compiler to generate an extended set of vector functions. The option is very similar to setting `compat`. However, for `compat`, only one vector function is created, while for `cmdtarget`, several vector functions are created for each vector specification. Vector variants are created for targets specified by compiler options `[Q]x` and/or `[Q]ax`. No change is made to the source code.
	`gcc`	Tells the compiler to use the gcc vector function ABI. Use this setting only in cases when you want to link with modules compiled by gcc. This setting is not available on Windows* systems.
	`legacy`	Tells the compiler to use the legacy vector function ABI. Use this setting if you need to keep the generated vector function binary backward compatible with the vectorized binary generated by older versions of the Intel® compilers (V13.1 or older).

Default

compat	The compiler uses the compatibility vector function ABI.

Description

This option determines which vector function application binary interface (ABI) the compiler uses to create or call vector functions.

NOTE
To avoid possible link-time and run-time errors, use identical `[Q]vecabi` settings when compiling all files in an application that define or use vector functions, including libraries. If setting `cmdtarget` is specified, options `[Q]x` and/or `[Q]ax` must have identical values.

Be careful using setting `cmdtarget` with libraries or program modules/routines with vector function definitions that cannot be recompiled. In such cases, setting `cmdtarget` may cause link errors.

On Linux* systems, since the default is `compat`, you must specify `legacy` if you need to keep the generated vector function binary backward compatible with the vectorized binary generated by the previous version of Intel® compilers.

When `cmdtarget` is specified, the additional vector function versions are created by copying each vector specification and changing target processor in the copy. The number of vector functions is determined by the settings specified in options `[Q]x` and/or `[Q]ax`.

For example, suppose we have the following function declaration:

```
__declspec (vector(processor(core_2_duo_sse4_1))) int foo(int a);
```

and the following options are specified: `-axAVX,CORE-AVX2`

The following table shows the different results for the above declaration and option specifications when setting `compat` or setting `cmdtarget` is used:

compat	cmdtarget
One vector version is created for Intel® SSE4.1 (by vector function specification).	Four vector versions are created for the following targets: • Intel® SSE2 (default because no `-x` option is used) • Intel® SSE4.1 (by vector function specification) • Intel® AVX (by the first `-ax` option value) • Intel® AVX2 (by the second `-ax` option value)

For more information about the Intel®-compatible vector functions ABI, see the downloadable PDF titled Vector Function Application Binary Interface.

For more information about the GCC vector functions ABI, see the item Libmvec - vector math library document in the GLIBC wiki at sourceware.org.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

Profile Guided Optimization Options

This section contains descriptions for compiler options that pertain to profile-guided optimization.

finstrument-functions, Qinstrument-functions

Determines whether function entry and exit points are instrumented.

Syntax

Linux OS:

`-finstrument-functions`

`-fno-instrument-functions`

macOS:

`-finstrument-functions`

`-fno-instrument-functions`

Windows OS:

`/Qinstrument-functions`

`/Qinstrument-functions-`

Arguments

None

Default

`-fno-instrument-functions` or `/Qinstrument-functions-`	Function entry and exit points are not instrumented.

Description

This option determines whether function entry and exit points are instrumented. It may increase execution time.

The following profiling functions are called with the address of the current function and the address of where the function was called (its "call site"):

- This function is called upon function entry:

```
void __cyg_profile_func_enter (void *this_fn,

void *call_site);
```

- This function is called upon function exit:

```
void __cyg_profile_func_exit (void *this_fn,

void *call_site);
```

These functions can be used to gather more information, such as profiling information or timing information. Note that it is the user's responsibility to provide these profiling functions.

If you specify `-finstrument-functions` (Linux* and macOS) or `/Qinstrument-functions` (Windows*), function inlining is disabled. If you specify `-fno-instrument-functions` or `/Qinstrument-functions-`, inlining is not disabled.

On Linux and macOS systems, you can use the following attribute to stop an individual function from being instrumented:

```
__attribute__((__no_instrument_function__))
```

It also stops inlining from being disabled for that individual function.

This option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

fnsplit, Qfnsplit

Enables function splitting.

Syntax

Linux OS:

`-fnsplit[=n]`

`-no-fnsplit`

macOS:

None

Windows OS:

`/Qfnsplit[:n]`

`/Qfnsplit-`

Arguments

n	Is an optional positive integer indicating the threshold number. The blocks can be placed into a different code segment if they are only reachable via a conditional branch whose taken probability is less than the specified *n*. Branch taken probability is heuristically calculated by the compiler and can be observed in assembly listings. The range for *n* is 0 <= n <= 100.

Default

OFF	Function splitting is not enabled. However, function grouping is still enabled.

Description

This option enables function splitting. If you specify `[Q]fnsplit` with no *n*, you must also specify option `[Q]prof-use`, or the option will have no effect and no function splitting will occur.

If you specify *n*, function splitting is enabled and you do not need to specify option `[Q]prof-use`.

To disable function splitting when you use option `[Q]prof-use`, specify `/Qfnsplit-` (Windows*) or `-no-fnsplit` (Linux*).

NOTE
Function splitting is generally not supported when exception handling is turned on for C/C++ routines in the stack of called routines. See also `-fexceptions` (Linux*) and the C/C++ option `/EH` (Windows*).

IDE Equivalent

Windows

Visual Studio: **Code Generation > Disable Function Splitting**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

Linux: `-freorder-blocks-and-partition` (a gcc option)

Windows: None

Gh

Calls a function to aid custom user profiling.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Gh`

Arguments

None

Default

OFF	The compiler uses the default libraries.

Description

This option calls the `__penter` function to aid custom user profiling. The prototype for `__penter` is not included in any of the standard libraries or Intel-provided libraries. You do not need to provide a prototype unless you plan to explicitly call `__penter`.

IDE Equivalent

None

Alternate Options

None

See Also

`GH` compiler option

GH

Calls a function to aid custom user profiling.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/GH`

Arguments

None

Default

OFF	The compiler uses the default libraries.

Description

This option calls the `__pexit` function to aid custom user profiling. The prototype for `__pexit` is not included in any of the standard libraries or Intel-provided libraries. You do not need to provide a prototype unless you plan to explicitly call `__pexit`.

IDE Equivalent

None

Alternate Options

None

See Also

`Gh` compiler option

p

Compiles and links for function profiling with gprof(1).

Syntax

Linux OS:

`-p`

macOS:

`-p`

Windows OS:

`None`

Arguments

None

Default

OFF	Files are compiled and linked without profiling.

Description

This option compiles and links for function profiling with gprof(1).

When you specify this option, inlining is disabled. However, you can override this by specifying pragma forceinline, declspec forceinline (Windows*), attribute always_inline (Linux* and macOS), or a compiler option such as `[Q]inline-forceinline`.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-qp` (this is a deprecated option)

Windows: None

prof-data-order, Qprof-data-order

Enables or disables data ordering if profiling information is enabled.

Syntax

Linux OS:

`-prof-data-order`

`-no-prof-data-order`

macOS:

None

Windows OS:

`/Qprof-data-order`

`/Qprof-data-order-`

Arguments

None

Default

`-no-prof-data-order` or `/Qprof-data-order-`	Data ordering is disabled.

Description

This option enables or disables data ordering if profiling information is enabled. It controls the use of profiling information to order static program data items.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify option `[Q]prof-gen setting globdata`.
- For feedback compilation, you must specify the `[Q]prof-use` option. You must not use multi-file optimization by specifying options such as `[Q]ipo` or `[Q]ipo-c`.

IDE Equivalent

None

Alternate Options

None

See Also

prof-gen, Qprof-gen compiler option
prof-use, Qprof-use compiler option
prof-func-order, Qprof-func-order compiler option

prof-dir, Qprof-dir

Specifies a directory for profiling information output files.

Syntax

Linux OS:

`-prof-dir dir`

macOS:

`-prof-dir dir`

Windows OS:

`/Qprof-dir:dir`

Arguments

dir	Is the name of the directory. You can specify a relative pathname or an absolute pathname.

Default

OFF	Profiling output files are placed in the directory where the program is compiled.

Description

This option specifies a directory for profiling information output files (*.dyn and *.dpi). The specified directory must already exist.

You should specify this option using the same directory name for both instrumentation and feedback compilations. If you move the .dyn files, you need to specify the new path.

Option `/Qprof-dir` is equivalent to option `/Qcov-dir`. If you specify both options, the last option specified on the command line takes precedence.

IDE Equivalent

Windows

Visual Studio: **General > Profile Directory**

Linux

Eclipse: **Optimization > Profile Directory**

OS X

Xcode: None

Alternate Options

None

prof-file, Qprof-file

Specifies an alternate file name for the profiling summary files.

Syntax

Linux OS:

`-prof-file` *`filename`*

macOS:

`-prof-file` *`filename`*

Windows OS:

`/Qprof-file:`*`filename`*

Arguments

filename	Is the name of the profiling summary file.

Default

OFF	The profiling summary files have the file name pgopti.*

Description

This option specifies an alternate file name for the profiling summary files. The *filename* is used as the base name for files created by different profiling passes.

If you add this option to profmerge, the .dpi file will be named *filename*.dpi instead of pgopti.dpi.

If you specify this option with option `[Q]prof-use`, the .dpi file will be named *filename*.dpi instead of pgopti.dpi.

Option `/Qprof-file` is equivalent to option `/Qcov-file`. If you specify both options, the last option specified on the command line takes precedence.

> **NOTE**
> When you use option `[Q]prof-file`, you can only specify a file name. If you want to specify a path (relative or absolute) for *filename*, you must also use option `[Q]prof-dir`.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-gen, Qprof-gen` compiler option

`prof-use, Qprof-use` compiler option

`prof-dir, Qprof-dir` compiler option

prof-func-groups

Enables or disables function grouping if profiling information is enabled.

Syntax

Linux OS:

`-prof-func-groups`

`-no-prof-func-groups`

macOS:

None

Windows OS:

None

Arguments

None

Default

`-no-prof-func-groups`	Function grouping is disabled.

Description

This option enables or disables function grouping if profiling information is enabled.

A "function grouping" is a profiling optimization in which entire routines are placed either in the cold code section or the hot code section.

If profiling information is enabled by option `-prof-use`, option `-prof-func-groups` is set and function grouping is enabled. However, if you explicitly enable `-prof-func-order`, function ordering is performed instead of function grouping.

If you want to disable function grouping when profiling information is enabled, specify `-no-prof-func-groups`.

To set the hotness threshold for function grouping, use option `-prof-hotness-threshold`.

IDE Equivalent

None

See Also

`prof-use, Qprof-use` compiler option

`prof-func-order, Qprof-func-order`
compiler option

`prof-hotness-threshold, Qprof-hotness-threshold`
compiler option

prof-func-order, Qprof-func-order

Enables or disables function ordering if profiling information is enabled.

Syntax

Linux OS:

`-prof-func-order`

`-no-prof-func-order`

macOS:

None

Windows OS:

`/Qprof-func-order`

`/Qprof-func-order-`

Arguments

None

Default

`-no-prof-func-order` or `/Qprof-func-order-`	Function ordering is disabled.

Description

This option enables or disables function ordering if profiling information is enabled.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify option `[Q]prof-gen setting srcpos`.
- For feedback compilation, you must specify `[Q]prof-use`. You must not use multi-file optimization by specifying options such as `[Q]ipo` or `[Q]ipo-c`.

If you enable profiling information by specifying option `[Q]prof-use`, option `[Q]prof-func-groups` is set and function grouping is enabled. However, if you explicitly enable the `[Q]prof-func-order` option, function ordering is performed instead of function grouping.

On Linux* systems, this option is only available for Linux linker 2.15.94.0.1, or later.

To set the hotness threshold for function grouping and function ordering, use option `[Q]prof-hotness-threshold`.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to use this option on a Windows system:

```
icl /Qprof-gen:globdata file1.c file2.c /Fe instrumented.exe
    ./instrumented.exe
icl /Qprof-use /Qprof-func-order file1.c file2.c /Fe feedback.exe
```

The following example shows how to use this option on a Linux system:

```
icc -prof-gen:globdata file1.c file2.c -o instrumented
    ./instrumented.exe
icc -prof-use -prof-func-order file1.c file2.c -o feedback
```

See Also

`prof-hotness-threshold, Qprof-hotness-threshold` compiler option

`prof-gen, Qprof-gen` compiler option

`prof-use, Qprof-use` compiler option

`prof-data-order, Qprof-data-order` compiler option

`prof-func-groups` compiler option

prof-gen, Qprof-gen

Produces an instrumented object file that can be used in profile guided optimization.

Syntax

Linux OS:

`-prof-gen[=`*keyword*`[, `*keyword*`],...]`

`-no-prof-gen`

macOS:

`-prof-gen[=`*keyword*`[, `*keyword*`],...]`

`-no-prof-gen`

Windows OS:

`/Qprof-gen[:`*keyword*`[,`*keyword*`],...]`

`/Qprof-gen-`

Arguments

keyword — Specifies details for the instrumented file. Possible values are:

`default`	Produces an instrumented object file. This is the same as specifying the `[Q]prof-gen` option with no keyword.
`srcpos`	Produces an instrumented object file that includes extra source position information.
`globdata`	Produces an instrumented object file that includes information for global data layout.

[no]threadsafe	Produces an instrumented object file that includes the collection of PGO data on applications that use a high level of parallelism. If [Q]prof-gen is specified with no keyword, the default is nothreadsafe.

Default

-no-prof-gen or /Qprof-gen-	Profile generation is disabled.

Description

This option produces an instrumented object file that can be used in profile guided optimization. It gets the execution count of each basic block.

You can specify more than one setting for [Q]prof-gen. For example, you can specify the following:

```
-prof-gen=srcpos -prof-gen=threadsafe (Linux* and macOS)
-prof-gen=srcpos, threadsafe (this is equivalent to the above)
```

```
/Qprof-gen:srcpos /Qprof-gen:threadsafe (Windows*)
/Qprof-gen:srcpos, threadsafe (this is equivalent to the above)
```

If you specify keyword srcpos or globdata, a static profile information file (.spi) is created. These settings may increase the time needed to do a parallel build using -prof-gen, because of contention writing the .spi file.

These options are used in phase 1 of the Profile Guided Optimizer (PGO) to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution.

When the [Q]prof-gen option is used to produce an instrumented binary file for profile generation, some optimizations are disabled. Those optimizations are not disabled for any subsequent profile-guided compilation with option [Q]prof-use that makes use of the generated profiles.

IDE Equivalent

Windows

Visual Studio: **General > Profile Guided Optimization**

General > Code Coverage Build Options

Linux

Eclipse: **Optimization > Profile Guided Optimization**

OS X

Xcode: None

Alternate Options

None

See Also

prof-use, Qprof-use compiler option
Profile an Application with Instrumentation

prof-gen-sampling

Tells the compiler to generate debug discriminators in debug output. This aids in developing more precise sampled profiling output. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-prof-gen-sampling`

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	The compiler does not generate debug discriminators in the debug output.

Description

This option tells the compiler to generate debug discriminators in debug output. Debug discriminators are used to distinguish code from different basic blocks that have the same source position information. This aids in developing more precise sampled hardware profiling output.

This is a deprecated option that may be removed in a future release. There is no replacement option.

To build an executable suitable for generating hardware profiled sampled output, compile with the following options:

```
-prof-gen-sampling -g
```

To use the data files produced by hardware profiling, compile with option `-prof-use-sampling`.

IDE Equivalent

None

Alternate Options

None

See Also

prof-use-sampling compiler option
g compiler option
Profile an Application with Instrumentation

prof-hotness-threshold, Qprof-hotness-threshold

Lets you set the hotness threshold for function grouping and function ordering.

Syntax

Linux OS:

`-prof-hotness-threshold=n`

macOS:

None

Windows OS:

`/Qprof-hotness-threshold:n`

Arguments

n	Is the hotness threshold. *n* is a percentage having a value between 0 and 100 inclusive. If you specify 0, there will be no hotness threshold setting in effect for function grouping and function ordering.

Default

OFF	The compiler's default hotness threshold setting of 10 percent is in effect for function grouping and function ordering.

Description

This option lets you set the hotness threshold for function grouping and function ordering.

The "hotness threshold" is the percentage of functions in the application that should be placed in the application's hot region. The hot region is the most frequently executed part of the application. By grouping these functions together into one hot region, they have a greater probability of remaining resident in the instruction cache. This can enhance the application's performance.

For this option to take effect, you must specify option `[Q]prof-use` and one of the following:

- On Linux systems: `-prof-func-groups` or `-prof-func-order`
- On Windows systems: `/Qprof-func-order`

IDE Equivalent

None

Alternate Options

None

See Also

prof-use, Qprof-use compiler option
prof-func-groups compiler option
prof-func-order, Qprof-func-order compiler option

prof-src-dir, Qprof-src-dir

Determines whether directory information of the source file under compilation is considered when looking up profile data records.

Syntax

Linux OS:

`-prof-src-dir`

`-no-prof-src-dir`

macOS:

`-prof-src-dir`

`-no-prof-src-dir`

Windows OS:

`/Qprof-src-dir`

`/Qprof-src-dir-`

Arguments

None

Default

[Q]prof-src-dir	Directory information is used when looking up profile data records in the .dpi file.

Description

This option determines whether directory information of the source file under compilation is considered when looking up profile data records in the .dpi file. To use this option, you must also specify the `[Q]prof-use` option.

If the option is enabled, directory information is considered when looking up the profile data records within the .dpi file. You can specify directory information by using one of the following options:

- Linux and macOS: `-prof-src-root` or `-prof-src-root-cwd`
- Windows: `/Qprof-src-root` or `/Qprof-src-root-cwd`

If the option is disabled, directory information is ignored and only the name of the file is used to find the profile data record.

Note that option `[Q]prof-src-dir` controls how the names of the user's source files get represented within the .dyn or .dpi files. Option `[Q]prof-dir` specifies the location of the .dyn or the .dpi files.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-use, Qprof-use` compiler option

`prof-src-root, Qprof-src-root` compiler option

`prof-src-root-cwd, Qprof-src-root-cwd` compiler option

prof-src-root, Qprof-src-root

Lets you use relative directory paths when looking up profile data and specifies a directory as the base.

Syntax

Linux OS:

`-prof-src-root=dir`

macOS:

`-prof-src-root=dir`

Windows OS:

`/Qprof-src-root:dir`

Arguments

dir	Is the base for the relative paths.

Default

OFF	The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It lets you specify a directory as the base. The paths are relative to a base directory specified during the `[Q]prof-gen` compilation phase.

This option is available during the following phases of compilation:

- Linux* and macOS systems: `-prof-gen` and `-prof-use` phases
- Windows* systems: `/Qprof-gen` and `/Qprof-use` phases

When this option is specified during the `[Q]prof-gen` phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the `[Q]prof-use` phase, it specifies a root directory that replaces the root directory specified at the `[Q]prof-gen` phase for forming the lookup keys.

To be effective, this option or option `[Q]prof-src-root-cwd` must be specified during the `[Q]prof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

IDE Equivalent

None

Alternate Options

None

Example

Consider the initial [Q]prof-gen compilation of the source file c:\user1\feature_foo\myproject\common\glob.c shown below:

```
Windows*: icl /Qprof-gen /Qprof-src-root=c:\user1\feature_foo\myproject -c common\glob.c
```

```
Linux* and macOS: icc -prof-gen -prof-src-root=c:\user1\feature_foo\myproject -c common\glob.c
```

For the [Q]prof-use phase, the file glob.c could be moved into the directory c:\user2\feature_bar\myproject\common\glob.c and profile information would be found from the .dpi when using the following:

```
Windows*: icl /Qprof-use /Qprof-src-root=c:\user2\feature_bar\myproject -c common\glob.c
```

```
Linux* and macOS: icc -prof-use -prof-src-root=c:\user2\feature_bar\myproject -c common\glob.c
```

If you do not use option [Q]prof-src-root during the [Q]prof-gen phase, by default, the [Q]prof-use compilation can only find the profile data if the file is compiled in the c:\user1\feature_foo\my_project\common directory.

See Also

prof-gen, Qprof-gen compiler option
prof-use, Qprof-use compiler option
prof-src-dir, Qprof-src-dir compiler option
prof-src-root-cwd, Qprof-src-root-cwd compiler option

prof-src-root-cwd, Qprof-src-root-cwd

Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.

Syntax

Linux OS:

-prof-src-root-cwd

macOS:

-prof-src-root-cwd

Windows OS:

/Qprof-src-root-cwd

Arguments

None

Default

OFF	The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It specifies the current working directory as the base. To use this option, you must also specify option [Q]prof-use.

This option is available during the following phases of compilation:

- Linux* and macOS systems: -prof-gen and -prof-use phases

- Windows* systems: `/Qprof-gen` and `/Qprof-use` phases

When this option is specified during the `[Q]prof-gen` phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the `[Q]prof-use` phase, it specifies a root directory that replaces the root directory specified at the `[Q]prof-gen` phase for forming the lookup keys.

To be effective, this option or option `[Q]prof-src-root` must be specified during the `[Q]prof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-gen, Qprof-gen` compiler option
`prof-use, Qprof-use` compiler option
`prof-src-dir, Qprof-src-dir` compiler option
`prof-src-root, Qprof-src-root` compiler option

prof-use, Qprof-use

Enables the use of profiling information during optimization.

Syntax

Linux OS:

`-prof-use[=`*keyword*`]`

`-no-prof-use`

macOS:

`-prof-use[=`*keyword*`]`

`-no-prof-use`

Windows OS:

`/Qprof-use[:`*keyword*`]`

`/Qprof-use-`

Arguments

keyword	Specifies additional instructions. Possible values are:	
	`weighted`	Tells the profmerge utility to apply a weighting to the .dyn file values when creating the .dpi file to normalize the data counts when the training runs have differentexecution durations. This argument only has an effect when the compiler invokes the profmerge utility

	to create the .dpi file. This argument does not have an effect if the .dpi file was previously created without weighting.
`[no]merge`	Enables or disables automatic invocation of the profmerge utility. The default is `merge`. Note that you cannot specify both `weighted` and `nomerge`. If you try to specify both values, a warning will be displayed and `nomerge` takes precedence.
`default`	Enables the use of profiling information during optimization. The profmerge utility is invoked by default. This value is the same as specifying `[Q]prof-use` with no argument.

Default

`-no-prof-use` or `/Qprof-use-`	Profiling information is not used during optimization.

Description

This option enables the use of profiling information (including function splitting and function grouping) during optimization. It enables option /Qfnsplit (Windows*) and -fnsplit (Linux* and macOS) .

This option instructs the compiler to produce a profile-optimized executable and it merges available profiling output files into a pgopti.dpi file.

Note that there is no way to turn off function grouping if you enable it using this option.

To set the hotness threshold for function grouping and function ordering, use option `[Q]prof-hotness-threshold`.

IDE Equivalent

Windows

Visual Studio: **General > Profile Guided Optimization**

Linux

Eclipse: **Optimization > Profile Guided Optimization**

OS X

Xcode: None

Alternate Options

None

See Also

`prof-hotness-threshold, Qprof-hotness-threshold` compiler option

`prof-gen, Qprof-gen` compiler option

Profile an Application with Instrumentation

prof-use-sampling

Lets you use data files produced by hardware profiling to produce an optimized executable. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-prof-use-sampling=`*`list`*

macOS:

None

Windows OS:

None

Arguments

list	Is a list of one or more data files. If you specify more than one data file, they must be separated by colons.

Default

OFF	Data files produced by hardware profiling will not be used to produce an optimized executable.

Description

This option lets you use data files produced by hardware profiling to produce an optimized executable.

This is a deprecated option that may be removed in a future release. There is no replacement option.

The data files are named and produced by using Intel® VTune™.

The executable should have been produced using the following options:

```
-prof-gen-sampling -g
```

IDE Equivalent

None

Alternate Options

None

See Also

prof-gen-sampling compiler option
Profile an Application with Instrumentation

prof-value-profiling, Qprof-value-profiling

Controls which values are value profiled.

Syntax

Linux OS:

`-prof-value-profiling[=keyword]`

macOS:

`-prof-value-profiling[=keyword]`

Windows OS:

`/Qprof-value-profiling[:keyword]`

Arguments

keyword	Controls which type of value profiling is performed. Possible values are:	
	`none`	Prevents all types of value profiling.
	`nodivide`	Prevents value profiling of non-compile time constants used in division or remainder operations.
	`noindcall`	Prevents value profiling of function addresses at indirect call sites.
	`all`	Enables all types of value profiling.

You can specify more than one keyword, but they must be separated by commas.

Default

`all`	All value profile types are enabled and value profiling is performed.

Description

This option controls which features are value profiled.

If this option is specified with option `[Q]prof-gen`, it turns off instrumentation of operations of the specified type. This also prevents feedback of values for the operations.

If this option is specified with option `[Q]prof-use`, it turns off feedback of values collected of the specified type.

If you specify level 2 or higher for option `[q or Q]opt-report`, the value profiling specialization information will be reported within the PGO optimization report.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-gen, Qprof-gen` compiler option
`prof-use, Qprof-use` compiler option
`qopt-report, Qopt-report` compiler option

Qcov-dir

Specifies a directory for profiling information output files that can be used with the codecov or tselect tool.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qcov-dir:`*`dir`*

Arguments

dir	Is the name of the directory.

Default

OFF	Profiling output files are placed in the directory where the program is compiled.

Description

This option specifies a directory for profiling information output files (*.dyn and *.dpi) that can be used with the code-coverage tool (codecov) or the test prioritization tool (tselect). The specified directory must already exist.

You should specify this option using the same directory name for both instrumentation and feedback compilations. If you move the .dyn files, you need to specify the new path.

Option `/Qcov-dir` is equivalent to option `/Qprof-dir`. If you specify both options, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

See Also

Qcov-gen compiler option
Qcov-file compiler option

Qcov-file

Specifies an alternate file name for the profiling summary files that can be used with the codecov or tselect tool.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qcov-file:`*`filename`*

Arguments

filename	Is the name of the profiling summary file.

Default

OFF	The profiling summary files have the file name pgopti.*.

Description

This option specifies an alternate file name for the profiling summary files. The file name can be used with the code-coverage tool (codecov) or the test prioritization tool (tselect).

The *filename* is used as the base name for the set of files created by different profiling passes.

If you specify this option with option `/Qcov-gen`, the .spi and .spl files will be named *filename*.spi and *filename*.spl instead of pgopti.spi and pgopti.spl.

Option `/Qcov-file` is equivalent to option `/Qprof-file`. If you specify both options, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

See Also

Qcov-gen compiler option
Qcov-dir compiler option

Qcov-gen

Produces an instrumented object file that can be used with the codecov or tselect tool.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qcov-gen`

`/Qcov-gen-`

Arguments

None

Default

`/Qcov-gen-`	The instrumented object file is not produced.

Description

This option produces an instrumented object file that can be used with the code-coverage tool (codecov) or the test prioritization tool (tselect). The instrumented code is included in the object file in preparation for instrumented execution.

This option also creates a static profile information file (.spi) that can be used with the codecov or tselect tool.

Option `/Qcov-gen` should be used to minimize the instrumentation overhead if you are interested in using the instrumentation only for code coverage. You should use `/Qprof-gen:srcpos` if you intend to use the collected data for code coverage and profile feedback.

IDE Equivalent

Windows

Visual Studio: **General > Code Coverage Build Options**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

Qcov-dir compiler option
Qcov-file compiler option

Optimization Report Options

This section contains descriptions for compiler options that pertain to optimization reports.

qopt-report, Qopt-report

Tells the compiler to generate an optimization report.

Syntax

Linux OS:

`-qopt-report[=n]`

macOS:

`-qopt-report[=n]`

Windows OS:

`/Qopt-report[:n]`

Arguments

n	(Optional) Indicates the level of detail in the report. You can specify values 0 through 5. If you specify zero, no report is generated. For levels *n*=1 through *n*=5, each level includes all the information of the previous level, as well as potentially some additional information. Level 5 produces the greatest level of detail. If you do not specify *n*, the default is level 2, which produces a medium level of detail.

Default

OFF	No optimization report is generated.

Description

This option tells the compiler to generate a collection of optimization report files, one per object; this is the same output produced by option `[q or Q]opt-report-per-object`.

If you prefer another form of output, you can specify option `[q or Q]opt-report-file`.

If you specify a level (*n*) higher than 5, a warning will be displayed and you will get a level 5 report.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* or macOS) or `/Qopt-report-phase:all` (Windows*).

For a description of the information that each *n* level provides, see the Example section in option `[q or Q]opt-report-phase`.

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Optimization Diagnostic Level**

Eclipse

Eclipse: **Compilation Diagnostics > Optimization Diagnostic Level**

Xcode

Xcode: **Diagnostics > Optimization Diagnostic Level**

Alternate Options

None

Example

If you only want reports about certain diagnostics, you can use this option with option `[q or Q]opt-report-phase`. The phase you specify determines which diagnostics you will receive.

For example, the following examples show how to get reports about certain specific diagnostics.

To get this specific report	Specify
Auto-parallelizer diagnostics	Linux* or macOS: `-qopt-report -qopt-report-phase=par` Windows*: `/Qopt-report /Qopt-report-phase:par`
OpenMP parallelizer diagnostics	Linux* or macOS: `-qopt-report -qopt-report-phase=openmp` Windows*: `/Qopt-report /Qopt-report-phase:openmp`
Vectorizer diagnostics	Linux* or macOS: `-qopt-report -qopt-report-phase=vec` Windows*: `/Qopt-report /Qopt-report-phase:vec`

See Also

`qopt-report-file, Qopt-report-file` compiler option

`qopt-report-per-object, Qopt-report-per-object` compiler option

`qopt-report-phase, Qopt-report-phase` compiler option

qopt-report-annotate, Qopt-report-annotate

Enables the annotated source listing feature and specifies its format.

Syntax

Linux OS:

`-qopt-report-annotate[=`*`keyword`*`]`

macOS:

`-qopt-report-annotate[=`*`keyword`*`]`

Windows OS:

`/Qopt-report-annotate[:`*`keyword`*`]`

Arguments

`keyword` Specifies the format for the annotated source listing. You can specify one of the following:

`text`	Indicates that the listing should be in text format. This is the default if you do not specify *keyword*.
`html`	Indicates that the listing should be in html format.

Default

OFF	No annotated source listing is generated

Description

This option enables the annotated source listing feature and specifies its format. The feature annotates source files with compiler optimization reports.

By default, one annotated source file is output per object. The annotated file is written to the same directory where the object files are generated. If the object file is a temporary file and an executable is generated, annotated files are placed in the directory where the executable is placed. You cannot generate annotated files to a directory of your choosing.

However, you can output annotated listings to stdout, stderr, or to a file if you also specify option `[q or Q]opt-report-file`.

By default, this option sets option `[q or Q]opt-report` with default level 2.

The following shows the file extension and listing details for the two possible *keyword*s.

Format	Listing Details
`text`	The annotated source listing has an .annot extension. It includes line numbers and compiler diagnostics placed after correspondent lines. IPO footnotes are inserted at the end of annotated file.
`html`	The annotated source listing has an .annot.html extension. It includes line numbers and compiler diagnostics placed after correspondent lines (as the text format does). It also provides hyperlinks in compiler messages and quick navigation with the routine list. IPO footnotes are displayed as tooltips.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-file, Qopt-report-file` compiler option

`qopt-report-annotate-position, Qopt-report-annotate-position` compiler option

qopt-report-annotate-position, Qopt-report-annotate-position

Enables the annotated source listing feature and specifies the site where optimization messages appear in the annotated source in inlined cases of loop optimizations.

Syntax

Linux OS:

`-qopt-report-annotate-position=`*`keyword`*

macOS:

`-qopt-report-annotate-position=`*`keyword`*

Windows OS:

`/Qopt-report-annotate-position:`*`keyword`*

Arguments

`keyword`	Specifies the site where optimization messages appear in the annotated source. You can specify one of the following:

`caller`	Indicates that the messages should appear in the caller site.
`callee`	Indicates that the messages should appear in the callee site.
`both`	Indicates that the messages should appear in both the caller and the callee sites.

Default

OFF	No annotated source listing is generated

Description

This option enables the annotated source listing feature and specifies the site where optimization messages appear in the annotated source in inlined cases of loop optimizations.

This option enables option `[q or Q]opt-report-annotate` if it is not explicitly specified.

If annotated source listing is enabled and this option is not passed to compiler, loop optimizations are placed in caller position by default.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-annotate, Qopt-report-annotate` compiler option

qopt-report-embed, Qopt-report-embed

Determines whether special loop information annotations will be embedded in the object file and/or the assembly file when it is generated.

Syntax

Linux OS:

`-qopt-report-embed`

`-qno-opt-report-embed`

macOS:

`-qopt-report-embed`

`-qno-opt-report-embed`

Windows OS:

`/Qopt-report-embed`

`/Qopt-report-embed-`

Arguments

None

Default

OFF	When an assembly file is being generated, special loop information annotations will not be embedded in the assembly file. However, if option `-g` (Linux* and macOS) or `/Zi` (Windows*) is specified, special loop information annotations will be embedded in the assembly file unless option `-qno-opt-report-embed` (Linux and macOS) or `/Qopt-report-embed-` (Windows) is specified.

Description

This option determines whether special loop information annotations will be embedded in the object file and/or the assembly file when it is generated. Specify the positive form of the option to include the annotations in the assembly file.

If an object file (or executable) is being generated, the annotations will be embedded in the object file (or executable).

If you use this option, you do not have to specify option `[q or Q]opt-report`.

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-file, Qopt-report-file

Specifies that the output for the optimization report goes to a file, stderr, or stdout.

Syntax

Linux OS:

`-qopt-report-file=`*`keyword`*

macOS:

`-qopt-report-file=`*`keyword`*

Windows OS:

`/Qopt-report-file:`*`keyword`*

Arguments

keyword	Specifies the output for the report. You can specify one of the following:	
	`filename`	Specifies the name of the file where the output should go.
	`stderr`	Indicates that the output should go to stderr.
	`stdout`	Indicates that the output should go to stdout.

Default

OFF	No optimization report is generated.

Description

This option specifies that the output for the optimization report goes to a file, stderr, or stdout.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Optimization Diagnostic File**

Diagnostics > Emit Optimization Diagnostic to File

Eclipse

Eclipse: **Compilation Diagnostics > Emit Optimization Diagnostics to File**

Compilation Diagnostics > Optimization Diagnostics File

Xcode

Xcode: None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-filter, Qopt-report-filter

Tells the compiler to find the indicated parts of your application, and generate optimization reports for those parts of your application.

Syntax

Linux OS:

`-qopt-report-filter=`*`string`*

macOS:

`-qopt-report-filter=`*`string`*

Windows OS:

`/Qopt-report-filter:`*`string`*

Arguments

string Is the information to search for. The *string* must appear within quotes. It can take one or more of the following forms:

filename
filename, *routine*
filename, *range* [, *range*]...

filename, *routine*, *range* [, *range*]...

If you specify more than one of the above forms in a string, a semicolon must appear between each form. If you specify more than one *range* in a string, a comma must appear between each *range*. Optional blanks can follow each parameter in the forms above and they can also follow each form in a string.

filename	Specifies the name of a file to be found. It can include a path. If you do not specify a path, the compiler looks for the filename in the current working directory.
routine	Specifies the name of a routine to be found. You can include an identifying parameter. The name, including any parameter, must be enclosed in single quotes. The compiler tries to uniquely identify the routine that corresponds to the specified routine name. It may select multiple routines to analyze, especially if more than one routine has the specified routine name, so the routine cannot be uniquely identified.
range	Specifies a range of line numbers to be found in the file or routine specified. The *range* must be specified in integers in the form: *first_line_number-last_line_number* The hyphen between the line numbers is required.

Default

OFF	No optimization report is generated.

Description

This option tells the compiler to find the indicated parts of your application, and generate optimization reports for those parts of your application. Optimization reports will only be generated for the routines that contain the specified *string*.

On Linux* and macOS, if you specify both `-qopt-report-routine=string1` and `-qopt-report-filter=string2`, it is treated as `-qopt-report-filter=string1;string2`. On Windows*, if you specify both `/Qopt-report-routine:string1` and `/Qopt-report-filter:string2`, it is treated as`/Qopt-report-filter:string1;string2`.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

qopt-report, Qopt-report compiler option

qopt-report-format, Qopt-report-format

Specifies the format for an optimization report.

Syntax

Linux OS:

`-qopt-report-format=keyword`

macOS:

`-qopt-report-format=keyword`

Windows OS:

`/Qopt-report-format:keyword`

Arguments

keyword Specifies the format for the report. You can specify one of the following:

`text`	Indicates that the report should be in text format.
`vs`	Indicates that the report should be in Visual Studio* (IDE) format. The Visual Studio IDE uses the information to visualize the optimization report in the context of your program source code.

Default

OFF	No optimization report is generated.

Description

This option specifies the format for an optimization report. If you use this option, you must specify either `text` or `vs`.

If you do not specify this option and another option causes an optimization report to be generated, the default format is `text`.

If the `[q or Q]opt-report-file` option is also specified, it will affect where the output goes:

- If `filename` is specified, output goes to the specified file.
- If `stdout` is specified, output goes to stdout.
- If `stderr` is specified, output goes to stderr.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-file, Qopt-report-file` compiler option

qopt-report-help, Qopt-report-help

Displays the optimizer phases available for report generation and a short description of what is reported at each level.

Syntax

Linux OS:

`-qopt-report-help`

Linux OS and macOS:

`-qopt-report-help`

Windows OS:

`/Qopt-report-help`

Arguments

None

Default

OFF	No optimization report is generated.

Description

This option displays the optimizer phases available for report generation using `[q or Q]opt-report-phase`, and a short description of what is reported at each level. No compilation is performed.

To indicate where output should go, you can specify one of the following options:

- `[q or Q]opt-report-file`
- `[q or Q]opt-report-per-object`

If you use this option, you do not have to specify option `[q or Q]opt-report`.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-phase, Qopt-report-phase` compiler option

`qopt-report-file, Qopt-report-file` compiler option

`qopt-report-per-object, Qopt-report-per-object` compiler option

qopt-report-names, Qopt-report-names

Specifies whether mangled or unmangled names should appear in the optimization report.

Syntax

Linux OS:

`-qopt-report-names=`*`keyword`*

macOS:

`-qopt-report-names=`*`keyword`*

Windows OS:

`/Qopt-report-names:`*`keyword`*

Arguments

keyword	Specifies the form for the names. You can specify one of the following:	
	`mangled`	Indicates that the optimization report should contain mangled names.
	`unmangled`	Indicates that the optimization report should contain unmangled names.

Default

OFF	No optimization report is generated.

Description

This option specifies whether mangled or unmangled names should appear in the optimization report. If you use this option, you must specify either `mangled` or `unmangled`.

If this option is not specified, unmangled names are used by default.

If you specify `mangled`, encoding (also known as decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the assembly listing.

If you specify `unmangled`, no encoding (or decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the source listing.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-per-object, Qopt-report-per-object

Tells the compiler that optimization report information should be generated in a separate file for each object.

Syntax

Linux OS:

`-qopt-report-per-object`

macOS:

`-qopt-report-per-object`

Windows OS:

`/Qopt-report-per-object`

Arguments

None

Default

OFF	No optimization report is generated.

Description

This option tells the compiler that optimization report information should be generated in a separate file for each object.

If you specify this option for a single-file compilation, a file with a .optrpt extension is produced for every object file or assembly file that is generated by the compiler. For a multifile Interprocedural Optimization (IPO) compilation, one file is produced for each of the N true objects generated in the compilation. If only one true object file is generated, the optimization report file generated is called ipo_out.optrpt. If multiple true object files are generated (N>1), the names used are ipo_out1.optrpt, ipo_out2.optrpt, … ipo_outN.optrpt.

The .optrpt files are written to the target directory of the compilation process. If an object or assembly file is explicitly generated, the corresponding .optrpt file is written to the same directory where the object file is generated. If the object file is just a temporary file and an executable is generated, the corresponding .optrpt files are placed in the directory in which the executable is placed.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* or macOS) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-phase, Qopt-report-phase

Specifies one or more optimizer phases for which optimization reports are generated.

Syntax

Linux OS:

`-qopt-report-phase[=list]`

macOS:

`-qopt-report-phase[=list]`

Windows OS:

`/Qopt-report-phase[:list]`

Arguments

list	(Optional) Specifies one or more phases to generate reports for. If you specify more than one phase, they must be separated with commas. The values you can specify are:

`cg`	The phase for code generation
`ipo`	The phase for Interprocedural Optimization
`loop`	The phase for loop nest optimization
`openmp`	The phase for OpenMP
`par`	The phase for auto-parallelization
`pgo`	The phase for Profile Guided Optimization
`tcollect`	The phase for trace collection
`vec`	The phase for vectorization
`all`	All optimizer phases. This is the default if you do not specify *list*.

Default

OFF	No optimization report is generated.

Description

This option specifies one or more optimizer phases for which optimization reports are generated.

For certain phases, you also need to specify other options:

- If you specify phase `cg`, you must also specify option `O1`, `O2` (default), or `O3`.
- If you specify phase `ipo`, you must also specify option `[Q]ipo`.
- If you specify phase `loop`, you must also specify option `O2` (default) or `O3`.
- If you specify phase `openmp`, you must also specify option `[q or Q]openmp`.
- If you specify phase `par`, you must also specify option `[Q]parallel`.
- If you specify phase `pgo`, you must also specify option `[Q]prof-use`.
- If you specify phase `tcollect`, you must also specify option `[Q]tcollect`.

- If you specify phase `vec`, you must also specify option `O2` (default) or `O3`. If you are interested in explicit vectorization by OpenMP* SIMD, you must also specify option `[q or Q]openmp`.

To find all phase possibilities, specify option `[q or Q]opt-report-help`.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

However, if you want to get more details for each phase, specify option `[q or Q]opt-report=`*n* along with this option and indicate the level of detail you want by specifying an appropriate value for *n*. (See also the Example section below.)

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* or macOS) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Optimization Diagnostic Phase**

Eclipse

Eclipse: **Compilation Diagnostics > Optimization Diagnostic Phase**

Xcode

Xcode: **Diagnostics > Optimization Diagnostic Phase**

Alternate Options

None

Example

The following shows examples of the details you may receive when you specify one of the optimizer phases and a particular level (*n*) for option `[q or Q]opt-report`. Note that details may change in future releases.

Optimizer phase	**The level specified in option `[q or Q]opt-report`**	**Description**
`cg`	1	Generates a list of which intrinsics were lowered and which memcall optimizations were performed.
`ipo`	1	For each compiled routine, generates a list of the routines that were inlined into the routine, called directly by the routine, and whose calls were deleted.
	2	Generates level 1 details, values for important inlining command line options, and a list of the routines that were discovered to be dead and eliminated.
	3	Generates level 2 details, whole program information, the sizes of inlined routines, and the reasons routines were not inlined.

Optimizer phase	The level specified in option `[q or Q]opt-report`	Description
	4	Generates level 3 details, detailed footnotes on the reasons why routines are not inlined, and what action the user can take to get them inlined.
`loop`	1	Reports high-level details about which optimizations have been performed on the loop nests (along with the line number). Most of the loop optimizations (like fusion, unroll, unroll & jam, collapsing, rerolling etc) only support this level of detail.
	2	Generates level 1 details, and provides more detail on the metrics and types of references (like prefetch distance, indirect prefetches etc) used in optimizations. Only a few optimizations (like prefetching, loop classification framework etc) support these extra details.
`openmp`	1	Reports loops, regions, sections, and tasks successfully parallelized.
	2	Generates level 1 details, and messages indicating successful handling of master constructs, single constructs, critical constructs, ordered constructs, atomic pragmas, and so forth.
`par`	1	Reports which loops were parallelized.
	2	Generates level 1 details, and reports which loops were not parallelized along with a short reason.
	3	Generates level 2 details, and prints the memory locations that are categorized as private, shared, reduction, etc..
	4	For this phase, this is the same as specifying level 3.

Optimizer phase	The level specified in option `[q or Q]opt-report`	Description
	5	Generates level 4 details, and dependency edges that inhibit parallelization.
`pgo`	1	During profile feedback, generates report status of feedback (such as, profile used, no profile available, or unable to use profile) for each routine compiled.
	2	Generates level 1 details, and reports which value profile specializations took place for indirect calls and arithmetic operations.
	3	Generates level 2 details, and reports which indirect calls had profile data, but did not meet the internal threshold limits for the percentage or execution count.
`tcollect`	1	Generates a list of routines and whether each was selected for trace collection.
`vec`	1	Reports which loops were vectorized.
	2	Generates level 1 details and reports which loops were not vectorized along with short reason.
	3	Generates level 2 details, and vectorizer loop summary information.
	4	Generates level 3 details, and greater detail about vectorized and non-vectorized loops.
	5	Generates level 4 details, and details about any proven or assumed data dependences.

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-help, Qopt-report-help` compiler option

qopt-report-routine, Qopt-report-routine

Tells the compiler to generate an optimization report for each of the routines whose names contain the specified substring.

Syntax

Linux OS:

`-qopt-report-routine=`*`substring`*

macOS:

`-qopt-report-routine=`*`substring`*

Windows OS:

`/Qopt-report-routine:`*`substring`*

Arguments

substring	Is the text (string) to look for.

Default

OFF	No optimization report is generated.

Description

This option tells the compiler to generate an optimization report for each of the routines whose names contain the specified substring.

You can also specify a sequence of substrings separated by commas. If you do this, the compiler will generate an optimization report for each of the routines whose name contains one or more of these substrings.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Optimization Diagnostic Routine**

Eclipse

Eclipse: **Compilation Diagnostics > Optimization Diagnostic Routine**

Xcode

Xcode: None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

OpenMP* Options and Parallel Processing Options

This section contains descriptions for compiler options that pertain to offload compilation, OpenMP*, or parallel processing.

device-math-lib

Enables or disables certain device libraries. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-device-math-lib=`*`library`*

`-no-device-math-lib=`*`library`*

macOS:

None

Windows OS:

`/device-math-lib:`*`library`*

`/no-device-math-lib:`*`library`*

Arguments

library	Possible values are: `fp32` — Links the fp32 device math library. `fp64` — Links the fp64 device math library. To link more than one library, include a comma between the library names. For example, if you want to link both the fp32 and fp64 device libraries, specify: fp32, fp64

Default

fp32, fp64	Both the fp32 and fp64 device libraries are linked.

Description

This option enables or disables certain device libraries.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

None

Alternate Options

None

See Also

fopenmp-device-lib compiler option

fmpc-privatize

Enables or disables privatization of all static data for the MultiProcessor Computing environment (MPC) unified parallel runtime.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`-fmpc-privatize`

`-fno-mpc-privatize`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-fno-mpc-privatize`	The privatization of all static data for the MPC unified parallel runtime is disabled.

Description

This option enables or disables privatization of all static data for the MultiProcessor Computing environment (MPC) unified parallel runtime.

Option `-fmpc-privatize` causes calls to extended thread-local-storage (TLS) resolution, run-time routines that are not supported on standard Linux* distributions.

This option requires installation of another product. For more information, see Feature Requirements.

IDE Equivalent

None

Alternate Options

None

fopenmp-device-lib

Enables or disables certain device libraries for an OpenMP target.*

Syntax

Linux OS:

`-fopenmp-device-lib=`*`library`*`[,`*`library`*`,...]`

`-fno-openmp-device-lib=`*`library`*`[,`*`library`*`,...]`

macOS:

`None`

Windows OS:

`-fopenmp-device-lib=`*`library`*`[,`*`library`*`,...]`

`-fopenmp-device-lib=`*`library`*`[,`*`library`*`,...]`

Arguments

`library`	Possible values are:	
	`libm-fp32`	Enables linking to the fp32 device math library.
	`libm-fp64`	Enables linking to the fp64 device math library.
	`libc`	Enables linking to the C library.
	`all`	Enables linking to libraries libm-fp32, libm-fp-64, and libc.
	To link more than one library, include a comma between the library names. For example, if you want to link both the libm-fp32 device library and the C library, specify: libm-fp32,libc. Do not add spaces between library names. Note that if you specify "all", it supersedes any additional value you may specify.	

Default

OFF	Disables linking to device libraries for this target.

Description

This option enables or disables certain device libraries for an OpenMP* target.

If you specify `fno-openmp-device-lib=library`, linking to the specified library is disabled for the OpenMP* target.

Alternate Options

None

par-affinity, Qpar-affinity

Specifies thread affinity.

Syntax

Linux OS:

`-par-affinity=[`*`modifier`*`,...]`*`type`*`[,`*`permute`*`][,`*`offset`*`]`

macOS:

`None`

Windows OS:

`/Qpar-affinity:[`*`modifier`*`,...]`*`type`*`[,`*`permute`*`][,`*`offset`*`]`

Arguments

modifier	Is one of the following values: `granularity={fine|thread|core|tile}`, `[no]respect`, `[no]verbose`, `[no]warnings`, `proclist=proc_list`. The default is `granularity=core, respect, and noverbose`. For information on value `proclist`, see Thread Affinity Interface.
type	Indicates the thread affinity. This argument is required and must be one of the following values: `compact`, `disabled`, `explicit`, `none`, `scatter`, `logical`, `physical`. The default is `none`. Values `logical` and `physical` are deprecated. Use `compact` and `scatter`, respectively, with no *permute* value.
permute	Is a positive integer. You cannot use this argument with *type* setting `explicit`, `none`, or `disabled`. The default is 0.
offset	Is a positive integer. You cannot use this argument with *type* setting `explicit`, `none`, or `disabled`. The default is 0.

Default

OFF	The thread affinity is determined by the run-time environment.

Description

This option specifies thread affinity, which binds threads to physical processing units. It has the same effect as environment variable KMP_AFFINITY.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- You have specified option `[Q]parallel` or option `[q or Q]openmp` (or both).
- You are compiling the main program.

> **NOTE**
> This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

None

Alternate Options

None

See Also

`parallel, Qparallel` compiler option

`qopt-report, Qopt-report` compiler option

par-loops, Qpar-loops

Lets you select between old or new implementations of parallel loop support.

Syntax

Linux OS and macOS:

`-par-loops=`*`keyword`*

Windows OS:

`/Qpar-loops:`*`keyword`*

Arguments

keyword	Specifies which implementation to use. Possible values are:	
	`new`	Enables the new implementation of parallel-loop support. As a result, parallel C++ range-based loops and collapsing complex loop stacks will not result in compilation errors. This is the default.
	`old`	Enables the old implementation of parallel-loop support. This is the same implementation that was supported in 18.0 and earlier releases.
	`default`	This is the same as specifying `new`.

Default

`-par-loops=new` or `/Qpar-loops:new`	The compiler uses the new implementation of parallel-loop support. Note that this setting may not yet be as stable as setting "old" since the implementation is new.

Description

This option lets you select between old or new implementations of parallel loop support.

The new implementation handles parallel C++ range-based loops, and also collapsing of OpenMP* parallel loops with complicated bounds expressions, for which the previous implementation reported errors.

If your code has a parallel loop that is not handled by the previous implementation, we recommend that you enable use of the new implementation.

IDE Equivalent

None

Alternate Options

None

par-num-threads, Qpar-num-threads

Specifies the number of threads to use in a parallel region.

Syntax

Linux OS:

`-par-num-threads=`*`n`*

macOS:

`-par-num-threads=`*`n`*

Windows OS:

`/Qpar-num-threads:`*`n`*

Arguments

n	Is the number of threads to use. It must be a positive integer.

Default

OFF	The number of threads to use is determined by the run-time environment.

Description

This option specifies the number of threads to use in a parallel region. It has the same effect as environment variable OMP_NUM_THREADS.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- You have specified option `[Q]parallel` or option `[q or Q]openmp` (or both).
- You are compiling the main program.

IDE Equivalent

None

Alternate Options

None

See Also

`parallel, Qparallel` compiler option
`qopt-report, Qopt-report` compiler option

par-runtime-control, Qpar-runtime-control

Generates code to perform run-time checks for loops that have symbolic loop bounds.

Syntax

Linux OS:

`-par-runtime-control[`*`n`*`]`

`-no-par-runtime-control`

macOS:

`-par-runtime-control[`*`n`*`]`

`-no-par-runtime-control`

Windows OS:

`/Qpar-runtime-control[n]`

`/Qpar-runtime-control-`

Arguments

n	Is a value denoting what kind of runtime checking to perform. Possible values are:	
	`0`	Performs no runtime check based on auto-parallelization. This is the same as specifying `-no-par-runtime-control` (Linux* and macOS) or `/Qpar-runtime-control-` (Windows*).
	`1`	Generates runtime check code under conservative mode. This is the default if you do not specify *n*.
	`2`	Generates runtime check code under heuristic mode.
	`3`	Generates runtime check code under aggressive mode.

Default

`-no-par-runtime-control` or `/Qpar-runtime-control-`	The compiler uses default heuristics when checking loops.

Description

This option generates code to perform run-time checks for loops that have symbolic loop bounds.

If the granularity of a loop is greater than the parallelization threshold, the loop will be executed in parallel.

If you do not specify this option, the compiler may not parallelize loops with symbolic loop bounds if the compile-time granularity estimation of a loop can not ensure it is beneficial to parallelize the loop.

NOTE
This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

None

Alternate Options

None

par-schedule, Qpar-schedule

Lets you specify a scheduling algorithm for loop iterations.

Syntax

Linux OS:

`-par-schedule-keyword[=n]`

macOS:

`-par-schedule-keyword[=n]`

Windows OS:

`/Qpar-schedule-keyword[[:]n]`

Arguments

keyword Specifies the scheduling algorithm or tuning method. Possible values are:

`auto`	Lets the compiler or run-time system determine the scheduling algorithm.
`static`	Divides iterations into contiguous pieces.
`static-balanced`	Divides iterations into even-sized chunks.
`static-steal`	Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
`dynamic`	Gets a set of iterations dynamically.
`guided`	Specifies a minimum number of iterations.
`guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution.
`runtime`	Defers the scheduling decision until run time.

n Is the size of the chunk or the number of iterations for each chunk. This setting can only be specified for static, dynamic, and guided. For more information, see the descriptions of each keyword below.

Default

static-balanced Iterations are divided into even-sized chunks and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

Description

This option lets you specify a scheduling algorithm for loop iterations. It specifies how iterations are to be divided among the threads of the team.

This option is only useful when specified with option `[Q]parallel`.

This option affects performance tuning and can provide better performance during auto-parallelization. It does nothing if it is used with option `[q or Q]openmp`.

Option	Description
`[Q]par-schedule-auto`	Lets the compiler or run-time system determine the scheduling algorithm. Any possible mapping may occur for iterations to threads in the team.

Option	Description
`[Q]par-schedule-static`	Divides iterations into contiguous pieces (chunks) of size *n*. The chunks are assigned to threads in the team in a round-robin fashion in the order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations. If no *n* is specified, the iteration space is divided into chunks that are approximately equal in size, and each thread is assigned at most one chunk.
`[Q]par-schedule-static-balanced`	Divides iterations into even-sized chunks. The chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.
`[Q]par-schedule-static-steal`	Divides iterations into even-sized chunks, but when a thread completes its chunk, it can steal parts of chunks assigned to neighboring threads. Each thread keeps track of L and U, which represent the lower and upper bounds of its chunks respectively. Iterations are executed starting from the lower bound, and simultaneously, L is updated to represent the new lower bound.
`[Q]par-schedule-dynamic`	Can be used to get a set of iterations dynamically. Assigns iterations to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations. Each chunk contains *n* iterations, except for the last chunk to be assigned, which may have fewer iterations. If no *n* is specified, the default is 1.
`[Q]par-schedule-guided`	Can be used to specify a minimum number of iterations. Assigns iterations to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunk of size 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For an *n* with value *k* (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than *k* iterations (except for the last chunk to be assigned, which may have fewer than *k* iterations). If no *n* is specified, the default is 1.

Option	Description
`[Q]par-schedule-guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution. The method depends on run-time implementation. Loop bounds are calculated with faster synchronization and chunks are dynamically dispatched at run time by threads in the team.
`[Q]par-schedule-runtime`	Defers the scheduling decision until run time. The scheduling algorithm and chunk size are then taken from the setting of environment variable OMP_SCHEDULE.

NOTE
This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

None

Alternate Options

None

par-threshold, Qpar-threshold

Sets a threshold for the auto-parallelization of loops.

Syntax

Linux OS:

`-par-threshold[n]`

macOS:

`-par-threshold[n]`

Windows OS:

`/Qpar-threshold[[:]n]`

Arguments

n	Is an integer whose value is the threshold for the auto-parallelization of loops. Possible values are 0 through 100. If *n* is 0, loops get auto-parallelized always, regardless of computation work volume. If *n* is 100, loops get auto-parallelized when performance gains are predicted based on the compiler analysis data. Loops get auto-parallelized only if profitable parallel execution is almost certain.

The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to parallelize only if there is a 50% probability of the code speeding up if executed in parallel.

Default

`-par-threshold100` or `/Qpar-threshold100`	Loops get auto-parallelized only if profitable parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. To use this option, you must also specify option `[Q]parallel`.

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

NOTE
This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Optimization > Auto-Parallelization Threshold**

OS X

Xcode: **Optimization > Auto-Parallelization Threshold**

Alternate Options

None

parallel, Qparallel

Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

Syntax

Linux OS:

`-parallel`

macOS:

`-parallel`

Windows OS:

`/Qparallel(or /Qpar)`

Arguments

None

Default

OFF	Multithreaded code is not generated for loops that can be safely executed in parallel.

Description

This option tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

To use this option, you must also specify option `O2` or `O3`.

This option sets option `[q or Q]opt-matmul` if option `O3` is also specified.

NOTE
On macOS systems, when you enable automatic parallelization, you must also set the DYLD_LIBRARY_PATH environment variable within Xcode* or an error will be displayed.

NOTE
Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as `/arch` or `/Qx` (Windows*) or `-m` or `-x` (Linux* and macOS).

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Optimization > Parallelization**

Eclipse

Eclipse: **Optimization > Parallelization**

Xcode

Xcode: **Optimization > Parallelization**

Alternate Options

None

See Also

qopt-report, Qopt-report compiler option
par-affinity, Qpar-affinity compiler option
par-num-threads, Qpar-num-threads compiler option
par-runtime-control, Qpar-runtime-control compiler option
par-schedule, Qpar-schedule compiler option
qopt-matmul, Qopt-matmul compiler option

parallel-source-info, Qparallel-source-info

Enables or disables source location emission when OpenMP or auto-parallelism code is generated.*

Syntax

Linux OS:

`-parallel-source-info[=n]`

`-no-parallel-source-info`

macOS:

`-parallel-source-info[=n]`

`-no-parallel-source-info`

Windows OS:

`/Qparallel-source-info`

`/Qparallel-source-info-[:n]`

Arguments

n	Is the level of source location emission. Possible values are:	
	0	Disables the emission of source location information when OpenMP* code or auto-parallelism code is generated. This is the same as specifying `-no-parallel-source-info` (Linux* and macOS) or `/Qparallel-source-info-` (Windows*).
	1	Tells the compiler to emit routine name and line information. This is the same as specifying `[Q]parallel-source-info` with no *n*.
	2	Tells the compiler to emit path, file, routine name, and line information.

Default

`-parallel-source-info=1` or `/Qparallel-source-info:1`	When OpenMP* code or auto-parallelism code is generated, the routine name and line information is emitted.

Description

This option enables or disables source location emission when OpenMP code or auto-parallelism code is generated. It also lets you set the level of emission.

IDE Equivalent

None

Alternate Options

None

qopenmp, Qopenmp

Enables the parallelizer to generate multi-threaded code based on OpenMP directives.*

Syntax

Linux OS:

`-qopenmp`

`-qno-openmp`

macOS:

`-qopenmp`

`-qno-openmp`

Windows OS:

`/Qopenmp`

`/Qopenmp-`

Arguments

None

Default

`-qno-openmp` or `/Qopenmp-`	No OpenMP* multi-threaded code is generated by the compiler.

Description

This option enables the parallelizer to generate multi-threaded code based on OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.

This option works with any optimization level. Specifying no optimization (`-O0` on Linux* or `/Od` on Windows*) helps to debug OpenMP applications.

NOTE
On macOS systems, when you enable OpenMP* API, you must also set the DYLD_LIBRARY_PATH environment variable within Xcode* or an error will be displayed.

NOTE
Options that use OpenMP* API are available for both Intel® microprocessors and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors versus non-Intel microprocessors include: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, thread affinity, and binding.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio

Visual Studio: **Language > OpenMP* Support**

Eclipse

Eclipse: **Language > Process OpenMP Directives**

Xcode

Xcode: **Language > Process OpenMP Directives**

Alternate Options

Linux and macOS: `-fopenmp`

Windows: `/openmp`

See Also

`qopenmp-stubs, Qopenmp-stubs` compiler option

qopenmp-lib, Qopenmp-lib

Lets you specify an OpenMP run-time library to use for linking.*

Syntax

Linux OS:

`-qopenmp-lib=`*type*

macOS:

`-qopenmp-lib=`*type*

Windows OS:

`/Qopenmp-lib:`*type*

Arguments

type	Specifies the type of library to use; it implies compatibility levels. Currently, the only possible value is:	
	`compat`	Tells the compiler to use the compatibility OpenMP* run-time library (libiomp). This setting provides compatibility with object files created using Microsoft* and GNU* compilers.

Default

`-qopenmp-lib=compat` or `/Qopenmp-lib:compat`	The compiler uses the compatibility OpenMP* run-time library (libiomp).

Description

This option lets you specify an OpenMP* run-time library to use for linking.

The compatibility OpenMP run-time libraries are compatible with object files created using the Microsoft* OpenMP run-time library (vcomp) or the GNU OpenMP run-time library (libgomp).

To use the compatibility OpenMP run-time library, compile and link your application using the compat setting for option `[q or Q]openmp-lib`. To use this option, you must also specify one of the following compiler options:

- Linux* systems: `-qopenmp` or `-qopenmp-stubs`
- Windows* systems: `/Qopenmp` or `/Qopenmp-stubs`

On Windows* systems, the compatibility OpenMP* run-time library lets you combine OpenMP* object files compiled with the Microsoft* C/C++ compiler with OpenMP* object files compiled with the Intel® C, Intel® C++, or Intel® Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

On Linux* systems, the compatibility Intel OpenMP* run-time library lets you combine OpenMP* object files compiled with the GNU* gcc or gfortran compilers with similar OpenMP* object files compiled with the Intel® C, Intel® C++, or Intel® Fortran Compiler. The linking phase results in a single, coherent copy of the run-time library.

NOTE The compatibility OpenMP run-time library is not compatible with object files created using versions of the Intel compilers earlier than 10.0.

NOTE On Windows* systems, this option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker. On Linux* and macOS systems, this option is processed by the `icc/icpc` command that initiates linking, adding library names explicitly to the link command.

IDE Equivalent

None

Alternate Options

None

See Also

`qopenmp, Qopenmp` compiler option

`qopenmp-stubs, Qopenmp-stubs` compiler option

qopenmp-link

Controls whether the compiler links to static or dynamic OpenMP run-time libraries.*

Syntax

Linux OS:

`-qopenmp-link=`*`library`*

macOS:

`-qopenmp-link=`*`library`*

Windows OS:

None

Arguments

library	Specifies the OpenMP library to use. Possible values are:	
	`static`	Tells the compiler to link to static OpenMP run-time libraries. Note that static OpenMP libraries are deprecated.
	`dynamic`	Tells the compiler to link to dynamic OpenMP run-time libraries.

Default

`-qopenmp-link=dynamic`	The compiler links to dynamic OpenMP* run-time libraries. However, if Linux* option `-static` is specified, the compiler links to static OpenMP run-time libraries.

Description

This option controls whether the compiler links to static or dynamic OpenMP* run-time libraries.

To link to the static OpenMP run-time library (RTL) and create a purely static executable, you must specify `-qopenmp-link=static`. However, we strongly recommend you use the default setting, `-qopenmp-link=dynamic`.

NOTE
Compiler options `-static-intel` and `-shared-intel` (Linux* and macOS) have no effect on which OpenMP run-time library is linked.

NOTE
On Linux* systems, `-qopenmp-link=dynamic` cannot be used in conjunction with option `-static`. If you try to specify both options together, an error will be displayed.

NOTE
On Linux systems, the OpenMP runtime library depends on using libpthread and libc (libgcc when compiled with gcc). Libpthread and libc (libgcc) must both be static or both be dynamic. If both libpthread and libc (libgcc) are static, then the static version of the OpenMP runtime should be used. If both libpthread and libc (libgcc) are dynamic, then either the static or dynamic version of the OpenMP runtime may be used.

IDE Equivalent

None

Alternate Options

None

qopenmp-simd, Qopenmp-simd

Enables or disables OpenMP SIMD compilation.*

Syntax

Linux OS:

`-qopenmp-simd`

`-qno-openmp-simd`

macOS:

`-qopenmp-simd`

`-qno-openmp-simd`

Windows OS:

`/Qopenmp-simd`

`/Qopenmp-simd-`

Arguments

None

Default

`-qopenmp-simd` or `/Qopenmp-simd`	OpenMP* SIMD compilation is enabled if option `O2` or higher is in effect. OpenMP* SIMD compilation is always disabled at optimization levels of `O1` or lower. When option `O2` or higher is in effect, OpenMP SIMD compilation can only be disabled by specifying option `-qno-openmp-simd` or `/Qopenmp-simd-`. It is not disabled by specifying option `-qno-openmp` or `/Qopenmp-`.

Description

This option enables or disables OpenMP* SIMD compilation.

You can use this option if you want to enable or disable the SIMD support with no impact on other OpenMP features. In this case, no OpenMP runtime library is needed to link and the compiler does not need to generate OpenMP runtime initialization code.

If you specify this option with the `[q or Q]openmp` option, it can impact other OpenMP features.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following:

```
-qno-openmp -qopenmp-simd    ! Linux or macOS
/Qopenmp- /Qopenmp-simd      ! Windows
```

The above is equivalent to specifying only `[q or Q]openmp-simd`. In this case, only SIMD support is provided, the OpenMP* library is not linked, and only the !$OMP directives related to SIMD are processed.

Consider the following:

```
-qopenmp -qopenmp-simd       ! Linux or macOS
/Qopenmp /Qopenmp-simd       ! Windows
```

In this case, SIMD support is provided, the OpenMP library is linked, and OpenMP runtime initialization code is generated. Note that when you specify `[q or Q]openmp`, it implies `[q or Q]openmp-simd`.

See Also

`qopenmp, Qopenmp` compiler option
`O` compiler option

qopenmp-stubs, Qopenmp-stubs

Enables compilation of OpenMP programs in sequential mode.*

Syntax

Linux OS:

`-qopenmp-stubs`

macOS:

`-qopenmp-stubs`

Windows OS:

`/Qopenmp-stubs`

Arguments

None

Default

OFF	The library of OpenMP* function stubs is not linked.

Description

This option enables compilation of OpenMP* programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.

IDE Equivalent

Windows

Visual Studio: **Language > OpenMP Support**

Linux

Eclipse: **Language > Process OpenMP Directives**

OS X

Xcode: **Language > Process OpenMP Directives**

Alternate Options

None

See Also

`qopenmp, Qopenmp` compiler option

qopenmp-threadprivate, Qopenmp-threadprivate

Lets you specify an OpenMP threadprivate implementation.*

Syntax

Linux OS:

`-qopenmp-threadprivate=`*`type`*

macOS:

`None`

Windows OS:

`/Qopenmp-threadprivate:`*`type`*

Arguments

type	Specifies the type of threadprivate implementation. Possible values are:	
	`legacy`	Tells the compiler to use the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel® compiler. This setting does not provide compatibility with the implementation used by other compilers.
	`compat`	Tells the compiler to use the compatibility OpenMP* threadprivate implementation based on applying the __declspec(thread) attribute to each threadprivate variable. The limitations of the attribute on a given

	platform also apply to the threadprivate implementation. This setting provides compatibility with the implementation provided by the Microsoft* and GNU* compilers.

Default

`-qopenmp-threadprivate=legacy` or `/Qopenmp-threadprivate:legacy`	The compiler uses the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel compiler.

Description

This option lets you specify an OpenMP* threadprivate implementation.

The threadprivate implementation of the legacy OpenMP run-time library may not be compatible with object files created using OpenMP run-time libraries supported in other compilers.

To use this option, you must also specify one of the following compiler options:

- Linux* systems: `-qopenmp` or `-qopenmp-stubs`
- Windows* systems: `/Qopenmp` or `/Qopenmp-stubs`

The value specified for this option is independent of the value used for the `[q or Q]openmp-lib` option.

> **NOTE**
> On macOS systems, legacy is the only type of threadprivate supported. Option `-qopenmp-threadprivate` is not recognized by the compiler.

IDE Equivalent

None

Alternate Options

None

Qpar-adjust-stack

Tells the compiler to generate code to adjust the stack size for a fiber-based main thread.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qpar-adjust-stack:`*`n`*

Arguments

n	Is the stack size (in bytes) for the fiber-based main thread. It must be a number equal to or greater than zero.

Default

`/Qpar-adjust-stack:0`	No adjustment is made to the main thread stack size.

Description

This option tells the compiler to generate code to adjust the stack size for a fiber-based main thread. This can reduce the stack size of threads.

For this option to be effective, you must also specify option `/Qparallel`.

IDE Equivalent

None

Alternate Options

None

See Also

`parallel, Qparallel` compiler option

Floating-Point Options

This section contains descriptions for compiler options that pertain to floating-point calculations.

fast-transcendentals, Qfast-transcendentals

Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.

Syntax

Linux OS:

`-fast-transcendentals`

`-no-fast-transcendentals`

macOS:

`-fast-transcendentals`

`-no-fast-transcendentals`

Windows OS:

`/Qfast-transcendentals`

`/Qfast-transcendentals-`

Arguments

None

Default

depends on the setting of `-fp-model` (Linux* and macOS) or `/fp` (Windows*)	If you do not specify option `-[no-]fast-transcendentals` or option /Qfast-transcendentals[-]: • The default is ON if option `-fp-model fast` or `/fp:fast` is specified or is in effect.

- The default is OFF if a value-safe setting is specified for `-fp-model` or `/fp` (such as "precise", "source", etc.).

Description

This option enables the compiler to replace calls to transcendental functions with implementations that may be faster but less precise.

It allows the compiler to perform certain optimizations on transcendental functions, such as replacing individual calls to sine in a loop with a single call to a less precise vectorized sine library routine. These optimizations can cause numerical differences that would not otherwise exist if you are also compiling with a value-safe option such as `-fp-model precise` (Linux* and macOS*) or `/fp:precise` (Windows).

For example, you may get different results if you specify option `O0` versus option `O2`, or you may get different results from calling the same function with the same input at different points in your program. If these kinds of numerical differences are problematic, consider using option `-fimf-use-svml` (Linux* and macOS*) or `/Qimf-use-svml` (Windows) as an alternative. When used with a value-safe option such as `-fp-model precise` or `/fp:precise`, option `-fimf-use-svml` or `/Qimf-use-svml` provides many of the positive performance benefits of `[Q]fast-transcendentals` without negatively affecting numeric consistency. For more details, see the description of option `-fimf-use-svml` and `/Qimf-use-svml`.

This option does not affect explicit Short Vector Math Library (SVML) intrinsics. It only affects scalar calls to the standard math library routines.

You cannot use option `-fast-transcendentals` with option `-fp-model strict` and you cannot use option `/Qfast-transcendentals` with option `/fp:strict`.

This option determines the setting for the maximum allowable relative error for math library function results (max-error) if none of the following options are specified:

- `-fimf-accuracy-bits` (Linux* and macOS) or `/Qimf-accuracy-bits` (Windows*)
- `-fimf-max-error` (Linux and macOS) or `/Qimf-max-error` (Windows)
- `-fimf-precision` (Linux and macOS) or `/Qimf-precision` (Windows)

This option enables extra optimization that only applies to Intel® processors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option

`fimf-use-svml, Qimf-use-svml` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fimf-precision, Qimf-precision` compiler option

fimf-absolute-error, Qimf-absolute-error

Defines the maximum allowable absolute error for math library function results.

Syntax

Linux OS:

`-fimf-absolute-error=`*value*`[:`*funclist*`]`

macOS:

`-fimf-absolute-error=`*value*`[:`*funclist*`]`

Windows OS:

`/Qimf-absolute-error:`*value*`[:`*funclist*`]`

Arguments

value	Is a positive, floating-point number. Errors in math library function results may exceed the maximum relative error (max-error) setting if the absolute-error is less than or equal to *value*. The format for the number is [digits] [.digits] [{ e | E }[sign]digits]
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-absolute-error=0.00001:sin,sinf` (or `/Qimf-absolute-error:0.00001:sin,sinf`) to specify the maximum allowable absolute error for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-absolute-error=0.00001:/` or `/Qimf-absolute-error: 0.00001:/`.

Default

Zero ("0")	An absolute-error setting of 0 means that the function is bound by the relative error setting. This is the default behavior.

Description

This option defines the maximum allowable absolute error for math library function results.

This option can improve run-time performance, but it may decrease the accuracy of results.

This option only affects functions that have zero as a possible return value, such as log, sin, asin, etc.

The relative error requirements for a particular function are determined by options that set the maximum relative error (max-error) and precision. The return value from a function must have a relative error less than the max-error value, or an absolute error less than the absolute-error value.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-absolute-error=0.00001:sin` or `/Qimf-absolute-error:0.00001:sin`, or `-fimf-absolute-error=0.00001:sqrtf` or `/Qimf-absolute-error:0.00001:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-accuracy-bits, Qimf-accuracy-bits

Defines the relative error for math library function results, including division and square root.

Syntax

Linux OS:

`-fimf-accuracy-bits=`*`bits`*`[:`*`funclist`*`]`

macOS:

`-fimf-accuracy-bits=`*`bits`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-accuracy-bits:`*`bits`*`[:`*`funclist`*`]`

Arguments

bits	Is a positive, floating-point number indicating the number of correct bits the compiler should use. The format for the number is [digits] [.digits] [{ e | E }[sign]digits].
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-accuracy-bits=23:sin,sinf` (or `/Qimf-accuracy-bits:23:sin,sinf`) to specify the relative error for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-accuracy-bits=10.0:/f` or `/Qimf-accuracy-bits:10.0:/f`.

Default

-fimf-precision=medium or /Qimf-precision:medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option defines the relative error, measured by the number of correct bits, for math library function results.

The following formula is used to convert bits into ulps: ulps = $2^{p-1-bits}$, where p is the number of the target format mantissa bits (24, 53, and 64 for single, double, and long double, respectively).

This option can affect run-time performance and the accuracy of results.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in the following:

- `-fimf-accuracy-bits=23:sinf,cosf,logf` or `/Qimf-accuracy-bits:23:sinf,cosf,logf`
- `-fimf-accuracy-bits=52:sqrt,/,trunc` or `/Qimf-accuracy-bits:52:sqrt,/,trunc`
- `-fimf-accuracy-bits=10:powf` or `/Qimf-accuracy-bits:10:powf`

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

- `-fimf-precision` (Linux* and macOS) or `/Qimf-precision` (Windows*)
- `-fimf-max-error` (Linux* and macOS) or `/Qimf-max-error` (Windows*)
- `-fimf-accuracy-bits` (Linux and macOS) or `/Qimf-accuracy-bits` (Windows)

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- [Q]fast-transcendentals
- [Q]prec-div
- [Q]prec-sqrt
- -fp-model (Linux and macOS) or /fp (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

fimf-absolute-error, Qimf-absolute-error compiler option
fimf-arch-consistency, Qimf-arch-consistency compiler option
fimf-domain-exclusion, Qimf-domain-exclusion compiler option
fimf-max-error, Qimf-max-error compiler option
fimf-precision, Qimf-precision compiler option
fimf-use-svml_Qimf-use-svml compiler option

fimf-arch-consistency, Qimf-arch-consistency

Ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture.

Syntax

Linux OS:

-fimf-arch-consistency=*value*[:*funclist*]

macOS:

-fimf-arch-consistency=*value*[:*funclist*]

Windows OS:

/Qimf-arch-consistency:*value*[:*funclist*]

Arguments

value	Is one of the logical values "true" or "false".

funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-arch-consistency=true:sin,sinf` (or `/Qimf-arch-consistency:true:sin,sinf`) to specify consistent results for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-arch-consistency=true:/` or `/Qimf-arch-consistency:true:/`.

Default

false	Implementations of some math library functions may produce slightly different results on implementations of the same architecture.

Description

This option ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture (for example, across different microarchitectural implementations of IA-32 architecture). Consistency is only guaranteed for a single binary. Consistency is not guaranteed across different architectures. For example, consistency is not guaranteed across IA-32 architecture and Intel® 64 architecture.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-arch-consistency=true:sin` or `/Qimf-arch-consistency:true:sin`, or `-fimf-arch-consistency=false:sqrtf` or `/Qimf-arch-consistency:false:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

The `-fimf-arch-consistency` (Linux* and macOS) and `/Qimf-arch-consistency` (Windows*) option may decrease run-time performance, but the option will provide bit-wise consistent results on all Intel® processors and compatible, non-Intel processors, regardless of micro-architecture. This option may not provide bit-wise consistent results between different architectures.

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Product and Performance Information
Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-domain-exclusion, Qimf-domain-exclusion

Indicates the input arguments domain on which math functions must provide correct results.

Syntax

Linux OS:

`-fimf-domain-exclusion=`*`classlist`*`[:`*`funclist`*`]`

macOS:

`-fimf-domain-exclusion=`*`classlist`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-domain-exclusion:`*`classlist`*`[:`*`funclist`*`]`

Arguments

classlist Is one of the following:

- One or more of the following floating-point value classes you can exclude from the function domain without affecting the correctness of your program. The supported class names are:

`extremes`	This class is for values which do not lie within the usual domain of arguments for a given function.
`nans`	This means "x=Nan".
`infinities`	This means "x=infinities".
`denormals`	This means "x=denormal".
`zeros`	This means "x=0".

  Each *classlist* element corresponds to a power of two. The exclusion attribute is the logical or of the associated powers of two (that is, a bitmask).

The following shows the current mapping from *classlist* mnemonics to numerical values:

extremes	1
nans	2
infinities	4
denormals	8
zeros	16
none	0
all	31
common	15
other combinations	bitwise OR of the used values

You must specify the integer value that corresponds to the class that you want to exclude.

Note that on excluded values, unexpected results may occur.

- One of the following short-hand tokens:

`none`	This means that none of the supported classes are excluded from the domain. To indicate this token, specify 0, as in `-fimf-domain-exclusion=0` (or `/Qimf-domain-exclusion:0`).
`all`	This means that all of the supported classes are excluded from the domain. To indicate this token, specify 31, as in `-fimf-domain-exclusion=31` (or `/Qimf-domain-exclusion:31`).
`common`	This is the same as specifying extremes,nans,infinities,denormals. To indicate this token, specify 15 (1 + 2+ 4 + 8), as in `-fimf-domain-exclusion=15` (or `/Qimf-domain-exclusion:15`)

funclist

Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-domain-exclusion=4:sin,sinf` (or `/Qimf-domain-exclusion:4:sin,sinf`) to specify infinities for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example, you can specify:

`-fimf-domain-exclusion=4` or `/Qimf-domain-exclusion:4`

`-fimf-domain-exclusion=5:/,powf` or `/Qimf-domain-exclusion:5:/,powf`

`-fimf-domain-exclusion=23:log,logf,/,sin,cosf`
or `/Qimf-domain-exclusion:23:log,logf,/,sin,cosf`

If you don't specify argument *funclist*, the domain restrictions apply to all math library functions.

Default

Zero ("0")	The compiler uses default heuristics when calling math library functions.

Description

This option indicates the input arguments domain on which math functions must provide correct results. It specifies that your program will function correctly if the functions specified in *funclist* do not produce standard conforming results on the number classes.

This option can affect run-time performance and the accuracy of results. As more classes are excluded, faster code sequences can be used.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-domain-exclusion=denormals:sin` or `/Qimf-domain-exclusion:denormals:sin`, or `-fimf-domain-exclusion=extremes:sqrtf` or `/Qimf-domain-exclusion:extremes:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

Example

Consider the following single-precision sequence for function exp2f:

Operation:	y = exp2f(x)
Accuracy:	1.014 ulp
Instructions:	4 (2 without fix-up)

The following shows the 2-instruction sequence without the fix-up:

```
vcvtfxpntps2dq  zmm1 {k1}, zmm0, 0x50      // zmm1 <-- rndToInt(2^24 * x)
vexp223ps       zmm1 {k1}, zmm1            // zmm1 <-- exp2(x)
```

However, the above 2-instruction sequence will not correctly process NaNs. To process Nans correctly, the following fix-up must be included following the above instruction sequence:

```
vpxord          zmm2, zmm2, zmm2             // zmm2 <-- 0
vfixupnanps     zmm1 {k1}, zmm0, zmm2 {aaaa} // zmm1 <-- QNaN(x) if x is NaN <F>
```

If the `vfixupnanps` instruction is not included, the sequence correctly processes any arguments except NaN values. For example, the following options generate the 2-instruction sequence:

```
-fimf-domain-exclusion=2:exp2f      <- NaN's are excluded (2 corresponds to NaNs)
-fimf-domain-exclusion=6:exp2f      <- NaN's and infinities are excluded (4 corresponds to
infinities; 2 + 4 = 6)
-fimf-domain-exclusion=7:exp2f      <- NaN's, infinities, and extremes are excluded (1
corresponds to extremes; 2 + 4 + 1 = 7)
-fimf-domain-exclusion=15:exp2f     <- NaN's, infinities, extremes, and denormals are excluded
(8 corresponds to denormals;  2 + 4 + 1 + 8=15)
```

If the `vfixupnanps` instruction is included, the sequence correctly processes any arguments including NaN values. For example, the following options generate the 4-instruction sequence:

```
-fimf-domain-exclusion=1:exp2f      <- only extremes are excluded (1 corresponds to extremes)
-fimf-domain-exclusion=4:exp2f      <- only infinities are excluded (4 corresponds to infinities)
-fimf-domain-exclusion=8:exp2f      <- only denormals are excluded (8 corresponds to denormals)
-fimf-domain-exclusion=13:exp2f     <- only extremes, infinities and denormals are excluded (1 +
4 + 8 = 13)
```

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-force-dynamic-target, Qimf-force-dynamic-target

Instructs the compiler to use run-time dispatch in calls to math functions.

Syntax

Linux OS:

`-fimf-force-dynamic-target[=`*funclist*`]`

macOS:

`-fimf-force-dynamic-target[=`*funclist*`]`

Windows OS:

`/Qimf-force-dynamic-target[:`*funclist*`]`

Arguments

funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-dynamic-target=sin,sinf` (or `/Qimf-dynamic-target:sin,sinf`) to specify run-time dispatch for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example, you can specify `-fimf-dynamic-target=/` or `/Qimf-dynamic-target:/`.

Default

OFF	Run-time dispatch is not forced in math libraries calls. The compiler can choose to call a CPU-specific version of a math function if one is available.

Description

This option instructs the compiler to use run-time dispatch in calls to math functions. When this option set to ON, it lets you force run-time dispatch in math libraries calls.

By default, when this option is set to OFF, the compiler often optimizes math library calls using the target CPU architecture-specific information available at compile time through the `[Q]x` and `arch` compiler options.

If you want to target multiple CPU families with a single application or you prefer to choose a target CPU at run time, you can force run-time dispatch in math libraries by using this option.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`x, Qx` compiler option

`arch` compiler option

`mtune, tune` compiler option

fimf-max-error, Qimf-max-error

Defines the maximum allowable relative error for math library function results, including division and square root.

Syntax

Linux OS:

`-fimf-max-error=`*`ulps`*`[:`*`funclist`*`]`

macOS:

`-fimf-max-error=`*`ulps`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-max-error:`*`ulps`*`[:`*`funclist`*`]`

Arguments

ulps	Is a positive, floating-point number indicating the maximum allowable relative error the compiler should use. The format for the number is [digits] [.digits] [{ e | E }[sign]digits].
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-max-error=4.0:sin,sinf` (or `/Qimf-max-error=4.0:sin,sinf`) to specify the maximum allowable relative error for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-max-error=4.0:/` or `/Qimf-max-error:4.0:/`.

Default

-fimf-precision=medium or /Qimf-precision:medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option defines the maximum allowable relative error, measured in ulps, for math library function results.

This option can affect run-time performance and the accuracy of results.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-max-error=4.0:sin` or `/Qimf-max-error:4.0:sin`, or `-fimf-max-error=4.0:sqrtf` or `/Qimf-max-error:4.0:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

- `-fimf-precision` (Linux* and macOS) or `/Qimf-precision` (Windows*)
- `-fimf-max-error` (Linux* and macOS) or `/Qimf-max-error` (Windows*)
- `-fimf-accuracy-bits` (Linux and macOS) or `/Qimf-accuracy-bits` (Windows)

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `[Q]fast-transcendentals`
- `[Q]prec-div`
- `[Q]prec-sqrt`
- `-fp-model` (Linux and macOS) or `/fp` (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-arch-consistency, Qimf-arch-consistency` compiler option

`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option

`fimf-precision, Qimf-precision` compiler option

`fimf-use-svml Qimf-use-svml` compiler option

fimf-precision, Qimf-precision

Lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.

Syntax

Linux OS:

`-fimf-precision[=`*value*`[:`*funclist*`]]`

macOS:

`-fimf-precision[=`*value*`[:`*funclist*`]]`

Windows OS:

`/Qimf-precision[:`*value*`[:`*funclist*`]]`

Arguments

value	Is one of the following values denoting the desired accuracy:	
	`high`	This is equivalent to max-error = 1.0.
	`medium`	This is equivalent to max-error = 4; this is the default setting if the option is specified and *value* is omitted.
	`low`	This is equivalent to accuracy-bits = 11 for single-precision functions; accuracy-bits = 26 for double-precision functions.
	In the above explanations, max-error means option `-fimf-max-error` (Linux* and macOS) or `/Qimf-max-error` (Windows*); accuracy-bits means option `-fimf-accuracy-bits` (Linux* and macOS) or `/Qimf-accuracy-bits` (Windows*).	
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-precision=high:sin,sinf` (or `/Qimf-precision:high:sin,sinf`) to specify high precision for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-precision=low:/` or `/Qimf-precision:low:/` and `-fimf-precision=low:/f` or `/Qimf-precision:low:/f`.	

Default

medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.

This option can be used to improve run-time performance if reduced accuracy is sufficient for the application, or it can be used to increase the accuracy of math library functions selected by the compiler.

In general, using a lower precision can improve run-time performance and using a higher precision may reduce run-time performance.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-precision=low:sin` or `/Qimf-precision:low:sin`, or `-fimf-precision=high:sqrtf` or `/Qimf-precision:high:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

- `-fimf-precision` (Linux* and macOS) or `/Qimf-precision` (Windows*)
- `-fimf-max-error` (Linux* and macOS) or `/Qimf-max-error` (Windows*)
- `-fimf-accuracy-bits` (Linux and macOS) or `/Qimf-accuracy-bits` (Windows)

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `[Q]fast-transcendentals`
- `[Q]prec-div`
- `[Q]prec-sqrt`
- `-fp-model` (Linux and macOS) or `/fp` (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-arch-consistency, Qimf-arch-consistency` compiler option

`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fast-transcendentals, Qfast-transcendentals` compiler option
`prec-div, Qprec-div` compiler option
`prec-sqrt, Qprec-sqrt` compiler option
`fp-model, fp` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-use-svml, Qimf-use-svml

Instructs the compiler to use the Short Vector Math Library (SVML) rather than the Intel® C++ Compiler Classic Math Library (LIBM) to implement math library functions.

Syntax

Linux OS:

`-fimf-use-svml=`*`value`*`[:`*`funclist`*`]`

macOS:

`-fimf-use-svml=`*`value`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-use-svml:`*`value`*`[:`*`funclist`*`]`

Arguments

funclist	Is an optional list of one or more math library functions to which the attribute should be applied. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-use-svmlt=true:sin,sinf` (or `/Qimf-use-svml:true:sin,sinf`) to specify that both the single-precision and double-precision sine functions should use SVML.

Default

false	Math library functions are implemented using the Intel® C++ Compiler Classic Math Library, though other compiler options such as `-fast-transcendentals` or `/Qfast-transcendentals` may give the compiler the flexibility to implement math library functions with either LIBM or SVML.

Description

This option instructs the compiler to implement math library functions using the Short Vector Math Library (SVML). When you specify `-fimf-use-svml=true` or `/Qimf-use-svml:true`, the specific SVML variant chosen is influenced by other compiler options such as `-fimf-precision` (Linux* and macOS) or `/Qimf-precision` (Windows*) and `-fp-model` (Linux and macOS) or `/fp` (Windows). This option has no effect on math library functions that are implemented in LIBM but not in SVML.

In value-safe settings of option `-fp-model` (Linux and macOS) or option `/fp` (Windows) such as precise, this option causes a slight decrease in the accuracy of math library functions, because even the high accuracy SVML functions are slightly less accurate than the corresponding functions in LIBM. Additionally, the SVML functions might not accurately raise floating-point exceptions, do not maintain errno, and are designed to work correctly only in round-to-nearest-even rounding mode.

The benefit of using `-fimf-use-svml=true` or `/Qimf-use-svml:true` with value-safe settings of `-fp-model` (Linux and macOS) or /fp (Windows) is that it can significantly improve performance by enabling the compiler to efficiently vectorize loops containing calls to math library functions.

If you need to use SVML for a specific math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sqrtf, as in `-fimf-use-svml=true:sin` or `/Qimf-use-svml:true:sin`, or `-fimf-use-svml =false:sqrtf` or `/Qimf-use-svml:false:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
If you specify option `-mia32` (Linux*) or option `/arch:IA32` (Windows*), vector instructions cannot be used. Therefore, you cannot use Linux* option `-mia32` with option `-fimf-use-svml=true`, and you cannot use Windows* option `/arch:IA32` with option `/Qimf-use-svml:true`.

NOTE
Since SVML functions may raise unexpected floating-point exceptions, be cautious about using features that enable trapping on floating-point exceptions. For example, be cautious about specifying option `-fimf-use-svml=true` with option `-fp-trap`, or option `/Qimf-use-svml:true` with option `/Qfp-trap`. For some inputs to some math library functions, such option combinations may cause your program to trap unexpectedly.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option
`m` compiler option
`arch` compiler option
`fp-trap, Qfp-trap` compiler option

fma, Qfma

Determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor.

Syntax

Linux OS:

`-fma`

`-no-fma`

macOS:

`-fma`

`-no-fma`

Windows OS:

`/Qfma`

`/Qfma-`

Arguments

None

Default

`-fma` or `/Qfma`	If the instructions exist on the target processor, the compiler generates fused multiply-add (FMA) instructions. However, if you specify `-fp-model strict` (Linux* and macOS) or `/fp:strict` (Windows*), but do not explicitly specify `-fma` or `/Qfma`, the default is `-no-fma` or `/Qfma-`.

Description

This option determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor. When the `[Q]fma` option is specified, the compiler may generate FMA instructions for combining multiply and add operations. When the negative form of the `[Q]fma` option is specified, the compiler must generate separate multiply and add instructions with intermediate rounding.

This option has no effect unless setting CORE-AVX2 or higher is specified for option [Q]x,`-march` (Linux and macOS), or `/arch` (Windows).

IDE Equivalent

None

See Also

`fp-model, fp` compiler option
`x, Qx` compiler option
`ax, Qax` compiler option
`march` compiler option
`arch` compiler option

fp-model, fp

Controls the semantics of floating-point calculations.

Syntax

Linux OS:

`-fp-model=`*`keyword`*

macOS:

`-fp-model=`*`keyword`*

Windows OS:

`/fp:`*`keyword`*

Arguments

keyword Specifies the semantics to be used. Possible values are:

`precise`	Disables optimizations that are not value-safe on floating-point data.
`fast[=1|2]`	Enables more aggressive optimizations on floating-point data.
`consistent`	The compiler uses default heuristics to determine results for different optimization levels or between different processors of the same architecture.
`strict`	Enables precise and except, disables contractions, and enables pragma `stdc fenv_access`.
`source`	Rounds intermediate results to source-defined precision.
`double`	Rounds intermediate results to 53-bit (double) precision.
`extended`	Rounds intermediate results to 64-bit (extended) precision.
`[no-]except` (Linux* and macOS) or `except[-]` (Windows*)	Determines whether strict floating-point exception semantics are honored.

Default

`-fp-model=fast=1` or `/fp:fast=1`	The compiler uses more aggressive optimizations on floating-point calculations.

Description

This option controls the semantics of floating-point calculations.

The *keywords* can be considered in groups:

- Group A: `precise`, `fast`, `strict`
- Group B: `source`, `double`, `extended`
- Group C: `except` (or negative forms `-no-except` or `/except-`)
- Group D: `consistent`

You can specify more than one *keyword*. However, the following rules apply:

- You cannot specify `fast` and `except` together in the same compilation. You can specify any other combination of group A, group B, and group C.
  Since `fast` is the default, you must not specify `except` without a group A or group B *keyword*.
- You should specify only one *keyword* from group A and only one *keyword* from group B. If you try to specify more than one *keyword* from either group A or group B, the last (rightmost) one takes effect.
- If you specify `except` more than once, the last (rightmost) one takes effect.
- If you specify `consistent` and any other keyword from another group, the last (rightmost) one may not fully override the heuristics set by `consistent`.

The floating-point (FP) environment is a collection of registers that control the behavior of FP machine instructions and indicate the current FP status. The floating-point environment may include rounding-mode controls, exception masks, flush-to-zero controls, exception status flags, and other floating-point related features.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td><code>-fp-model=precise</code> or <code>/fp:precise</code></td><td>
Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations, which is required for strict ANSI conformance.
These semantics ensure the reproducibility of floating-point computations for serial code, including code vectorized or auto-parallelized by the compiler, but they may slow performance. They do not ensure value safety or run-to-run reproducibility of other parallel code.
Run-to-run reproducibility for floating-point reductions in OpenMP* code may be obtained for a fixed number of threads through the KMP_DETERMINISTIC_REDUCTION environment variable. For more information about this environment variable, see topic "Supported Environment Variables".
The compiler assumes the default floating-point environment; you are not allowed to modify it.
Intermediate results are computed with the precision shown in the following table, unless it is overridden by a keyword from Group B:
<table>
<tr><td></td><td>Windows</td><td>Linux</td><td>macOS</td></tr>
<tr><td>IA-32 architecture</td><td>Double</td><td>Extended</td><td>Not applicable</td></tr>
<tr><td>Intel® 64 architecture</td><td>Source</td><td>Source</td><td>Source</td></tr>
</table>
Floating-point exception semantics are disabled by default. To enable these semantics, you must also specify <code>-fp-model=except</code> or <code>/fp:except</code>.
</td></tr>
</table>

Option	Description
`-fp-model=fast[=1|2]` or `/fp:fast[=1|2]`	Tells the compiler to use more aggressive optimizations when implementing floating-point calculations. These optimizations increase speed, but may affect the accuracy or reproducibility of floating-point computations. Specifying fast is the same as specifying `fast=1`. `fast=2` may produce faster and less accurate results. Floating-point exception semantics are disabled by default and they cannot be enabled because you cannot specify fast and except together in the same compilation. To enable exception semantics, you must explicitly specify another keyword (see other keyword descriptions for details). To enable exception semantics, you must explicitly specify another keyword (see other keyword descriptions for details).
`-fp-model=consistent` or `/fp:consistent`	The compiler uses default heuristics to generate code that will determine results for different optimization levels or between different processors of the same architecture . For more information, see the article titled: Consistency of Floating-Point Results using the Intel® Compiler.
`-fp-model=source` or `/fp:source`	This option causes intermediate results to be rounded to the precision defined in the source code. It also implies keyword `precise` unless it is overridden by a keyword from Group A. Intermediate expressions use the precision of the operand with higher precision, if any. `long double` — 64-bit precision — 80-bit data type — 15-bit exponent `double` — 53-bit precision — 64-bit data type — 11-bit exponent; on Windows systems using IA-32 architecture, the exponent may be 15-bit if an x87 register

Option	Description
	is used to hold the value.
	`float` 24-bit precision; 32-bit data type; 8-bit exponent
	The compiler assumes the default floating-point environment; you are not allowed to modify it.
`-fp-model=double` or `/fp:double`	This option causes intermediate results to be rounded as follows: 53-bit (double) precision 64-bit data type 11-bit exponent; on Windows systems using IA-32 architecture, the exponent may be 15-bit if an x87 register is used to hold the value. This option also implies keyword `precise` unless it is overridden by a keyword from Group A. The compiler assumes the default floating-point environment; you are not allowed to modify it.
`-fp-model=extended` or `/fp:extended`	This option causes intermediate results to be rounded as follows: 64-bit (extended) precision 80-bit data type 15-bit exponent This option also implies keyword `precise` unless it is overridden by a keyword from Group A. The compiler assumes the default floating-point environment; you are not allowed to modify it.
`-fp-model=except` or `/fp:except`	Tells the compiler to follow strict floating-point exception semantics.

The `-fp-model` and `/fp` options determine the setting for the maximum allowable relative error for math library function results (max-error) if none of the following options are specified (the following options are only available for ifort):

- `-fimf-accuracy-bits` (Linux* and macOS) or `/Qimf-accuracy-bits` (Windows*)
- `-fimf-max-error` (Linux and macOS) or `/Qimf-max-error` (Windows)
- `-fimf-precision` (Linux and macOS) or `/Qimf-precision` (Windows)
- `[Q]fast-transcendentals`

Option `-fp-model=fast` (and `/fp:fast`) sets option `-fimf-precision=medium` (`/Qimf-precision:medium`) and option `-fp-model=precise` (and `/fp:precise`) implies `-fimf-precision=high` (and `/Qimf-precision:high`). Option `-fp-model=fast=2` (and `/fp:fast2`) sets option `-fimf-precision=medium` (and `/Qimf-precision:medium`) and option `-fimf-domain-exclusion=15` (and `/Qimf-domain-exclusion=15`).

NOTE
In Microsoft* Visual Studio, when you create a Microsoft* Visual C++ project, option `/fp:precise` is set by default. It sets the floating-point model to improve consistency for floating-point operations by disabling certain optimizations that may reduce performance. To set the option back to the general default `/fp:fast`, change the IDE project property for Floating Point Model to Fast.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

Visual Studio: **Code Generation>Floating Point Model**

Code Generation>Enable Floating Point Exceptions

Code Generation> Floating Point Expression Evaluation

Eclipse: **Floating Point > Floating Point Model**

Xcode: **Floating Point > Floating Point Model**

Floating Point > Reliable Floating Point Exceptions Model

Alternate Options

None

See Also

`O` compiler option (specifically O0)
`Od` compiler option
`mp1, Qprec` compiler option
`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fast-transcendentals, Qfast-transcendentals` compiler option
Supported Environment Variables
The article titled: Consistency of Floating-Point Results using the Intel® Compiler

fp-port, Qfp-port

Rounds floating-point results after floating-point operations.

Syntax

Linux OS:

`-fp-port`

`-no-fp-port`

macOS:

`-fp-port`

`-no-fp-port`

Windows OS:

`/Qfp-port`

`/Qfp-port-`

Arguments

None

Default

`-no-fp-port` or `/Qfp-port-`	The default rounding behavior depends on the compiler's code generation decisions and the precision parameters of the operating system.

Description

This option rounds floating-point results after floating-point operations.

This option is designed to be used with the `-mia32` (Linux*) or `/arch:IA32` (Windows*) option on a 32-bit compiler. Under those conditions, the compiler implements floating-point calculations using the x87 instruction set, which uses an internal precision that may be higher than the precision specified in the program.

By default, the compiler may keep results of floating-point operations in this higher internal precision. Rounding to program precision occurs at unspecified points. This provides better performance, but the floating-point results are less deterministic. The `[Q]fp-port` option rounds floating-point results to user-specified precision at assignments and type conversions. This has some impact on speed.

When compiling for newer architectures, the compiler implements floating-point calculations with different instructions, such as Intel® SSE and SSE2. These Intel® Streaming SIMD Extensions round directly to single precision or double precision at every instruction. In these cases, option `[Q]fp-port` has no effect.

IDE Equivalent

Windows

Visual Studio: **Optimization > Floating-point Precision Improvements**

Linux

Eclipse: **Floating Point > Round Floating-Point Results**

OS X

Xcode: **Floating Point > Round Floating-Point Results**

Alternate Options

None

See Also

Floating-point Operations

fp-speculation, Qfp-speculation

Tells the compiler the mode in which to speculate on floating-point operations.

Syntax

Linux OS:

`-fp-speculation=`*mode*

macOS:

`-fp-speculation=`*mode*

Windows OS:

`/Qfp-speculation:`*mode*

Arguments

mode	Is the mode for floating-point operations. Possible values are:	
	`fast`	Tells the compiler to speculate on floating-point operations.
	`safe`	Tells the compiler to disable speculation if there is a possibility that the speculation may cause a floating-point exception.
	`strict`	Tells the compiler to disable speculation on floating-point operations.
	`off`	This is the same as specifying strict. This feature is only available for ifort.

Default

`-fp-speculation=fast` or `/Qfp-speculation:fast`	The compiler speculates on floating-point operations. This is also the behavior when optimizations are enabled. However, if you specify no optimizations (`-O0` on Linux*; `/Od` on Windows*), the default is `-fp-speculation=safe` (Linux*) or `/Qfp-speculation:safe` (Windows*).

Description

This option tells the compiler the mode in which to speculate on floating-point operations.

Disabling speculation may prevent the vectorization of some loops containing conditionals.

IDE Equivalent

Visual Studio: **Optimization > Floating-Point Speculation**

Eclipse: **Floating Point > Floating-Point Speculation**

Xcode: **Floating Point > Floating-Point Speculation**

Alternate Options

None

fp-stack-check, Qfp-stack-check

Tells the compiler to generate extra code after every function call to ensure that the floating-point stack is in the expected state.

Syntax

Linux OS:

`-fp-stack-check`

macOS:

`-fp-stack-check`

Windows OS:

`/Qfp-stack-check`

Arguments

None

Default

OFF	There is no checking to ensure that the floating-point (FP) stack is in the expected state.

Description

This option tells the compiler to generate extra code after every function call to ensure that the floating-point (FP) stack is in the expected state.

By default, there is no checking. So when the FP stack overflows, a NaN value is put into FP calculations and the program's results differ. Unfortunately, the overflow point can be far away from the point of the actual bug. This option places code that causes an access violation exception immediately after an incorrect call occurs, thus making it easier to locate these issues.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Floating Point > Check Floating-point Stack**

OS X

Xcode: **Floating Point > Check Floating-point Stack**

Alternate Options

None

fp-trap, Qfp-trap

Sets the floating-point trapping mode for the main routine.

Syntax

Linux OS:

`-fp-trap=`*mode*`[,`*mode*`,...]`

macOS:

`-fp-trap=`*mode*`[,`*mode*`,...]`

Windows OS:

`/Qfp-trap:`*mode*`[,`*mode*`,...]`

Arguments

mode — Is the floating-point trapping mode. If you specify more than one mode value, the list is processed sequentially from left to right. Possible values are:

`[no]divzero`	Enables or disables the IEEE trap for division by zero.
`[no]inexact`	Enables or disables the IEEE trap for inexact result.
`[no]invalid`	Enables or disables the IEEE trap for invalid operation.
`[no]overflow`	Enables or disables the IEEE trap for overflow.
`[no]underflow`	Enables or disables the IEEE trap for underflow.
`[no]denormal`	Enables or disables the trap for denormal.
`all`	Enables all of the above traps.
`none`	Disables all of the above traps.
`common`	Sets the most commonly used IEEE traps: division by zero, invalid operation, and overflow.

Default

`-fp-trap=none` or `/Qfp-trap:none`	No traps are enabled when a program starts.

Description

This option sets the floating-point trapping mode for the main routine. It does not set a handler for floating-point exceptions.

The `[no]` form of a `mode` value is only used to modify the meaning of `mode` values `all` and `common`, and can only be used with one of those values. The `[no]` form of the option by itself does not explicitly cause a particular trap to be disabled.

Use `mode` value `inexact` with caution. This results in the trap being enabled whenever a floating-point value cannot be represented exactly, which can cause unexpected results.

If `mode` value `underflow` is specified, the compiler ignores the FTZ (flush-to-zero) bit state of Intel® Streaming SIMD Extensions (Intel® SSE) floating-point units.

When a DAZ (denormals are zero) bit is set in an Intel® SSE floating-point unit control word, a denormal operand exception is never generated.

To set the floating-point trapping mode for all routines, specify the `[Q]fp-trap-all` option.

NOTE
The negative form of the `[Q]ftz` option can be used to set or reset the FTZ and the DAZ hardware flags.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Unmask Floating Point Exceptions**

Configuration Properties->C/C++ > Unmask Floating Point Exceptions

Linux

Eclipse: **Floating Point > Initial Exception Mask**

OS X

Xcode: **Floating Point > Set Initial Exception Mask**

Alternate Options

None

See Also

`ftz, Qftz` compiler option
`fp-trap-all, Qfp-trap-all` compiler option

fp-trap-all, Qfp-trap-all

Sets the floating-point trapping mode for all routines.

Syntax

Linux OS:

`-fp-trap-all=`*mode*`[,`*mode*`,...]`

macOS:

`-fp-trap-all=`*mode*`[,`*mode*`,...]`

Windows OS:

`/Qfp-trap-all:`*mode*`[,`*mode*`,...]`

Arguments

mode	Is the floating-point trapping mode. If you specify more than one mode value, the list is processed sequentially from left to right. Possible values are:

`[no]divzero`	Enables or disables the IEEE trap for division by zero.
`[no]inexact`	Enables or disables the IEEE trap for inexact result.
`[no]invalid`	Enables or disables the IEEE trap for invalid operation.
`[no]overflow`	Enables or disables the IEEE trap for overflow.
`[no]underflow`	Enables or disables the IEEE trap for underflow.
`[no]denormal`	Enables or disables the trap for denormal.
`all`	Enables all of the above traps.
`none`	Disables all of the above traps.
`common`	Sets the most commonly used IEEE traps: division by zero, invalid operation, and overflow.

Default

`-fp-trap-all=none` No traps are enabled for all routines.
or
`/Qfp-trap-all:none`

Description

This option sets the floating-point trapping mode for the main routine. It does not set a handler for floating-point exceptions.

The `[no]` form of a `mode` value is only used to modify the meaning of `mode` values `all` and `common`, and can only be used with one of those values. The `[no]` form of the option by itself does not explicitly cause a particular trap to be disabled.

Use `mode` value `inexact` with caution. This results in the trap being enabled whenever a floating-point value cannot be represented exactly, which can cause unexpected results.

If `mode` value `underflow` is specified, the compiler ignores the FTZ (flush-to-zero) bit state of Intel® Streaming SIMD Extensions (Intel® SSE) floating-point units.

When a DAZ (denormals are zero) bit is set in an Intel® SSE floating-point unit control word, a denormal operand exception is never generated.

To set the floating-point trapping mode for the main routine only, specify the `[Q]fp-trap` option.

NOTE
The negative form of the `[Q]ftz` option can be used to set or reset the FTZ and the DAZ hardware flags.

IDE Equivalent

None

Alternate Options

None

See Also

`ftz, Qftz` compiler option
`fp-trap, Qfp-trap` compiler option

ftz, Qftz

Flushes denormal results to zero.

Syntax

Linux OS:

`-ftz`

`-no-ftz`

macOS:

`-ftz`

`-no-ftz`

Windows OS:

`/Qftz`

`/Qftz-`

Arguments

None

Default

`-ftz` or `/Qftz`	Denormal results are flushed to zero. Every optimization option `O` level, except `O0`, sets `[Q]ftz`.

Description

This option flushes denormal results to zero when the application is in the gradual underflow mode. It may improve performance if the denormal values are not critical to your application's behavior.

The `[Q]ftz` option has no effect during compile-time optimization.

The `[Q]ftz` option sets or resets the FTZ and the DAZ hardware flags. If FTZ is ON, denormal results from floating-point calculations will be set to the value zero. If FTZ is OFF, denormal results remain as is. If DAZ is ON, denormal values used as input to floating-point instructions will be treated as zero. If DAZ is OFF, denormal instruction inputs remain as is. Systems using Intel® 64 architecture have both FTZ and DAZ. FTZ and DAZ are not supported on all IA-32 architectures.

When the `[Q]ftz` option is used in combination with an SSE-enabling option on systems using IA-32 architecture (for example, the `[Q]xSSE2` option), the compiler will insert code in the main routine to set FTZ and DAZ. When `[Q]ftz` is used without such an option, the compiler will insert code to conditionally set FTZ/DAZ based on a run-time processor check.

If you specify option `-no-ftz` (Linux and macOS) or option `/Qftz-` (Windows), it prevents the compiler from inserting any code that might set FTZ or DAZ.

Option `[Q]ftz` only has an effect when the main program is being compiled. It sets the FTZ/DAZ mode for the process. The initial thread and any threads subsequently created by that process will operate in FTZ/DAZ mode.

If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ/DAZ mode off by specifying `-no-ftz` or `/Qftz-` in the command line while still benefiting from the `O3` optimizations.

NOTE
Option `[Q]ftz` is a performance option. Setting this option does not guarantee that all denormals in a program are flushed to zero. The option only causes denormals generated at run time to be flushed to zero.

IDE Equivalent

Windows

Visual Studio: **Optimization > Flush Denormal Results to Zero**

Linux

Eclipse: **Floating-Point > Flush Denormal Results to Zero**

OS X

Xcode: **Floating-Point > Flush Denormal Results to Zero**

Alternate Options

None

See Also

`x, Qx` compiler option
Setting the FTZ and DAZ Flags

Ge

Enables stack-checking for all functions. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Ge`

Arguments

None

Default

OFF	Stack-checking for all functions is disabled.

Description

This option enables stack-checking for all functions.

This is a deprecated option that may be removed in a future release. The replacement option is `/Gs0`.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/Gs0`

mp1, Qprec

Improves floating-point precision and consistency.

Syntax

Linux OS:

`-mp1`

macOS:

`-mp1`

Windows OS:

`/Qprec`

Arguments

None

Default

OFF	The compiler provides good accuracy and run-time performance at the expense of less consistent floating-point results.

Description

This option improves floating-point consistency. It ensures the out-of-range check of operands of transcendental functions and improves the accuracy of floating-point compares.

This option prevents the compiler from performing optimizations that change NaN comparison semantics and causes all values to be truncated to declared precision before they are used in comparisons. It also causes the compiler to use library routines that give better precision results compared to the X87 transcendental instructions.

This option disables fewer optimizations and has less impact on performance than option `-fp-model precise` (Linux* and macOS) or option `/fp:precise` (Windows*).

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: None

Xcode

Xcode: **FloatingPoint > Improve Floating-Point Consistency**

Alternate Options

None

See Also

pc, Qpc

Enables control of floating-point significand precision.

Syntax

Linux OS:

`-pc`*n*

macOS:

`-pc`*n*

Windows OS:

`/Qpc`*n*

Arguments

n	Is the floating-point significand precision. Possible values are:	
	`32`	Rounds the significand to 24 bits (single precision).
	`64`	Rounds the significand to 53 bits (double precision).
	`80`	Rounds the significand to 64 bits (extended precision).

Default

`-pc80` or `/Qpc64`	On Linux* and macOS systems, the floating-point significand is rounded to 64 bits. On Windows* systems, the floating-point significand is rounded to 53 bits.

Description

This option enables control of floating-point significand precision.

Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with this option.

Note that a change of the default precision control or rounding mode, for example, by using the `[Q]pc32` option or by user intervention, may affect the results returned by some of the mathematical functions.

IDE Equivalent

None

Alternate Options

None

prec-div, Qprec-div

Improves precision of floating-point divides.

Syntax

Linux OS:

`-prec-div`

`-no-prec-div`

macOS:

`-prec-div`

`-no-prec-div`

Windows OS:

`/Qprec-div`

`/Qprec-div-`

Arguments

None

Default

OFF	Default heuristics are used. The default is not as accurate as full IEEE division, but it is slightly more accurate than would be obtained when `/Qprec-div-` or `-no-prec-div` is specified. If you need full IEEE precision for division, you should specify `[Q]prec-div`.

Description

This option improves precision of floating-point divides. It has a slight impact on speed.

At default optimization levels, the compiler may change floating-point division computations into multiplication by the reciprocal of the denominator. For example, A/B is computed as A * (1/B) to improve the speed of the computation.

However, sometimes the value produced by this transformation is not as accurate as full IEEE division. When it is important to have fully precise IEEE division, use this option to disable the floating-point division-to-multiplication optimization. The result is more accurate, with some loss of performance.

If you specify `-no-prec-div` (Linux* and macOS) or `/Qprec-div-` (Windows*), it enables optimizations that give slightly less precise results than full IEEE division.

Option `[Q]prec-div` is implied by option `-fp-model precise` (Linux* and macOS) and option `/fp:precise` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option

prec-sqrt, Qprec-sqrt

Improves precision of square root implementations.

Syntax

Linux OS:

`-prec-sqrt`

`-no-prec-sqrt`

macOS:

`-prec-sqrt`

`-no-prec-sqrt`

Windows OS:

`/Qprec-sqrt`

`/Qprec-sqrt-`

Arguments

None

Default

`-no-prec-sqrt` or `/Qprec-sqrt-`	The compiler uses a faster but less precise implementation of square root. However, the default is `-prec-sqrt` or `/Qprec-sqrt` if any of the following options are specified: `/Od`, `/fp:precise`, or `/Qprec` on Windows* systems; `-O0` or `-mp1` on Linux* and macOS systems.

Description

This option improves precision of square root implementations. It has a slight impact on speed.

This option inhibits any optimizations that can adversely affect the precision of a square root computation. The result is fully precise square root implementations, with some loss of performance.

IDE Equivalent

None

Alternate Options

None

qsimd-honor-fp-model, Qsimd-honor-fp-model

Tells the compiler to obey the selected floating-point model when vectorizing SIMD loops.

Syntax

Linux OS:

`-qsimd-honor-fp-model`

`-qno-simd-honor-fp-model`

macOS:

`-qsimd-honor-fp-model`

`-qno-simd-honor-fp-model`

Windows OS:

`/Qsimd-honor-fp-model`

`/Qsimd-honor-fp-model-`

Arguments

None

Default

`-qno-simd-honor-fp-model` or `/Qsimd-honor-fp-model-`	The compiler performs vectorization of SIMD loops even if it breaks the floating-point model setting.

Description

The OpenMP* SIMD specification and the setting of compiler option `-fp-model` (Linux* and macOS) or `/fp` (Windows*) can contradict in requirements. When contradiction occurs, the default behavior of the compiler is to follow the OpenMP* specification and therefore vectorize the loop.

This option lets you override this default behavior - it causes the compiler to follow the `-fp-model` (or `/fp`) specification. This means that the compiler will serialize the loop.

> **NOTE**
> This option does not affect automatic vectorization of loops. By default, the compiler uses `-fp-model` (Linux* and macOS) or `/fp` (Windows*) settings for this.

IDE Equivalent

None

Alternate Options

None

See Also

`qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction` compiler option

`fp-model, fp` compiler option

simd pragma

qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction

Tells the compiler to serialize floating-point reduction when vectorizing SIMD loops.

Syntax

Linux OS:

`-qsimd-serialize-fp-reduction`

`-qno-simd-serialize-fp-reduction`

macOS:

`-qsimd-serialize-fp-reduction`

`-qno-simd-serialize-fp-reduction`

Windows OS:

`/Qsimd-serialize-fp-reduction`

`/Qsimd-serialize-fp-reduction-`

Arguments

None

Default

`-qno-simd-serialize-fp-reduction` or `/Qsimd-serialize-fp-reduction-`	The compiler does not attempt to serialize floating-point reduction in SIMD loops.

Description

The OpenMP* SIMD reduction specification and the setting of compiler option `-fp-model` (Linux* and macOS) or `/fp` (Windows*) can contradict in requirements. When contradiction occurs, the default behavior of the compiler is to follow OpenMP* specification and therefore vectorize the loop, including floating-point reduction.

This option lets you override this default behavior - it causes the compiler to follow the `-fp-model` (or `/fp`) specification. This means that the compiler will serialize the floating-point reduction while vectorizing the rest of the loop.

NOTE
When `[q or Q]simd-honor-fp-model` is specified and OpenMP* SIMD reduction specification is the only thing causing serialization of the entire loop, addition of option `[q or Q]simd-serialize-fp-reduction` will result in vectorization of the entire loop except for reduction calculation, which will be serialized.

NOTE
This option does not affect automatic vectorization of loops. By default, the compiler uses `-fp-model` (Linux* and macOS) or `/fp` (Windows*) settings for this.

IDE Equivalent

None

Alternate Options

None

See Also

`qsimd-honor-fp-model, Qsimd-honor-fp-model` compiler option

`fp-model, fp` compiler option

simd pragma

rcd, Qrcd

Enables fast float-to-integer conversions. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-rcd`

macOS:

`-rcd`

Windows OS:

`/Qrcd`

Arguments

None

Default

OFF	Floating-point values are truncated when a conversion to an integer is involved.

Description

This option enables fast float-to-integer conversions. It can improve the performance of code that requires floating-point-to-integer conversions.

This is a deprecated option that may be removed in a future release. There is no replacement option.

The system default floating-point rounding mode is round-to-nearest. However, the C language requires floating-point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating-point-to-integer conversion and change it back afterwards.

This option disables the change to truncation of the rounding mode for all floating-point calculations, including floating point-to-integer conversions. This option can improve performance, but floating-point conversions to integer will not conform to C semantics.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/QIfist` (this is a deprecated option)

Inlining Options

This section contains descriptions for compiler options that pertain to inlining.

fgnu89-inline

Tells the compiler to use C89 semantics for inline functions when in C99 mode.

Syntax

Linux OS:

`-fgnu89-inline`

macOS:

`-fgnu89-inline`

Windows OS:

None

Arguments

None

Default

OFF

Description

This option tells the compiler to use C89 semantics for inline functions when in C99 mode.

IDE Equivalent

None

Alternate Options

None

finline

Tells the compiler to inline functions declared with __inline and perform C++ inlining.

Syntax

Linux OS:

`-finline`

`-fno-inline`

macOS:

`-finline`

`-fno-inline`

Windows OS:

None

Arguments

None

Default

`-fno-inline`	The compiler does not inline functions declared with `__inline`.

Description

This option tells the compiler to inline functions declared with `__inline` and perform C++ inlining.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-inline-level`

Windows: `/Ob`

finline-functions

Enables function inlining for single file compilation.

Syntax

Linux OS:

`-finline-functions`

`-fno-inline-functions`

macOS:

`-finline-functions`

`-fno-inline-functions`

Windows OS:

None

Arguments

None

Default

`-finline-functions`	Interprocedural optimizations occur. However, if you specify `-O0`, the default is OFF.

Description

This option enables function inlining for single file compilation.

It enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

The compiler applies a heuristic to perform the function expansion. To specify the size of the function to be expanded, use the `-finline-limit` option.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-inline-level=2`

Windows: `/Ob2`

See Also

`ip, Qip` compiler option

`finline-limit` compiler option

finline-limit

Lets you specify the maximum size of a function to be inlined.

Syntax

Linux OS:

`-finline-limit=`*n*

macOS:

`-finline-limit=`*n*

Windows OS:

`None`

Arguments

n	Must be an integer greater than or equal to zero. It is the maximum number of lines the function can have to be considered for inlining.

Default

OFF	The compiler uses default heuristics when inlining functions.

Description

This option lets you specify the maximum size of a function to be inlined. The compiler inlines smaller functions, but this option lets you inline large functions. For example, to indicate a large function, you could specify 100 or 1000 for *n*.

Note that parts of functions cannot be inlined, only whole functions.

This option is a modification of the `-finline-functions` option, whose behavior occurs by default.

IDE Equivalent

None

Alternate Options

None

See Also

`finline-functions` compiler option

inline-calloc, Qinline-calloc

Tells the compiler to inline calls to calloc() as calls to malloc() and memset().

Architectures

All

Syntax

Linux OS and macOS:

`-inline-calloc`

`-no-inline-calloc`

Windows OS:

`/Qinline-calloc`

`/Qinline-calloc-`

Arguments

None

Default

`-no-inline-calloc` or `/Qinline-calloc-`	The compiler inlines calls to `calloc()` as calls to `calloc()`.

Description

This option tells the compiler to inline calls to `calloc()` as calls to `malloc()` and `memset()`. This enables additional memset() optimizations. For example, it can enable inlining as a sequence of store operations when the size is a compile time constant.

NOTE
Many routines in the supplied libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

IDE Equivalent

None

Alternate Options

None

inline-factor, Qinline-factor

Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.

Syntax

Linux OS:

`-inline-factor=`*n*

`-no-inline-factor`

macOS:

`-inline-factor=`*n*

`-no-inline-factor`

Windows OS:

`/Qinline-factor:`*n*

`/Qinline-factor-`

Arguments

n	Is a positive integer specifying the percentage value. The default value is 100 (a factor of 1).

Default

`-inline-factor=100` or `/Qinline-factor:100`	The compiler uses a percentage multiplier of 100.

Description

This option specifies the percentage multiplier that should be applied to all inlining options that define upper limits:

- `[Q]inline-max-size`
- `[Q]inline-max-total-size`
- `[Q]inline-max-per-routine`
- `[Q]inline-max-per-compile`

The `[Q]inline-factor` option takes the default value for each of the above options and multiplies it by *n* divided by 100. For example, if 200 is specified, all inlining options that define upper limits are multiplied by a factor of 2. This option is useful if you do not want to individually increase each option limit.

If you specify `-no-inline-factor` (Linux* and macOS) or `/Qinline-factor-` (Windows*), the following occurs:

- Every function is considered to be a small or medium function; there are no large functions.
- There is no limit to the size a routine may grow when inline expansion is performed.
- There is no limit to the number of times some routine may be inlined into a particular routine.
- There is no limit to the number of times inlining can be applied to a compilation unit.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase default limits, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-max-size, Qinline-max-size` compiler option
`inline-max-total-size, Qinline-max-total-size` compiler option
`inline-max-per-routine, Qinline-max-per-routine` compiler option
`inline-max-per-compile, Qinline-max-per-compile` compiler option
`qopt-report, Qopt-report` compiler option

inline-forceinline, Qinline-forceinline

Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.

Syntax

Linux OS:

`-inline-forceinline`

macOS:

`-inline-forceinline`

Windows OS:

`/Qinline-forceinline`

Default

OFF	The compiler uses default heuristics for inline routine expansion.

Description

This option instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.

Without this option, the compiler treats functions declared with the inline keyword as merely being recommended for inlining. When this option is used, it is as if they were declared with the keyword __forceinline keyword.

NOTE
Because C++ member functions whose definitions are included in the class declaration are considered inline functions by default, using this option will also make these member functions "forceinline" functions.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

> **Caution**
> When you use this option to change the meaning of inline to "forceinline", the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

inline-level, Ob

Specifies the level of inline function expansion.

Syntax

Linux OS:

`-inline-level=n`

macOS:

`-inline-level=n`

Windows OS:

`/Obn`

Arguments

n	Is the inline function expansion level. Possible values are 0, 1, and 2.

Default

`-inline-level=2` or `/Ob2`	This is the default if option `O2` is specified or is in effect by default. On Windows* systems, this is also the default if option `O3` is specified.
`-inline-level=0` or `/Ob0`	This is the default if option `-O0` (Linux* and macOS) or `/Od` (Windows*) is specified.

Description

This option specifies the level of inline function expansion. Inlining procedures can greatly improve the run-time performance of certain programs.

Option	Description
`-inline-level=0` or `/Ob0`	Disables inlining of user-defined functions. Note that statement functions are always inlined.

Option	Description
`-inline-level=1` or `/Ob1`	Enables inlining when an inline keyword or an inline attribute is specified. Also enables inlining according to the C++ language.
`-inline-level=2` or `/Ob2`	Enables inlining of any function at the compiler's discretion.

IDE Equivalent

Windows

Visual Studio: **Optimization > Inline Function Expansion**

Linux

Eclipse: **Optimization > Inline Function Expansion**

OS X

Xcode: **Optimization > Inline Function Expansion**

Alternate Options

None

inline-max-per-compile, Qinline-max-per-compile

Specifies the maximum number of times inlining may be applied to an entire compilation unit.

Syntax

Linux OS:

`-inline-max-per-compile=`*n*

`-no-inline-max-per-compile`

macOS:

`-inline-max-per-compile=`*n*

`-no-inline-max-per-compile`

Windows OS:

`/Qinline-max-per-compile=`*n*

`/Qinline-max-per-compile-`

Arguments

n	Is a positive integer that specifies the number of times inlining may be applied.

Default

`-no-inline-max-per-compile` or `/Qinline-max-per-compile-`	The compiler uses default heuristics for inline routine expansion.

Description

This option the maximum number of times inlining may be applied to an entire compilation unit. It limits the number of times that inlining can be applied.

For compilations using Interprocedural Optimizations (IPO), the entire compilation is a compilation unit. For other compilations, a compilation unit is a file.

If you specify `-no-inline-max-per-compile` (Linux* and macOS) or `/Qinline-max-per-compile-` (Windows*), there is no limit to the number of times inlining may be applied to a compilation unit.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-factor, Qinline-factor` compiler option

`qopt-report, Qopt-report` compiler option

inline-max-per-routine, Qinline-max-per-routine

Specifies the maximum number of times the inliner may inline into a particular routine.

Syntax

Linux OS:

`-inline-max-per-routine=`*n*

`-no-inline-max-per-routine`

macOS:

`-inline-max-per-routine=`*n*

`-no-inline-max-per-routine`

Windows OS:

`/Qinline-max-per-routine=`*n*

`/Qinline-max-per-routine-`

Arguments

n	Is a positive integer that specifies the maximum number of times the inliner may inline into a particular routine.

Default

`-no-inline-max-per-routine` or `/Qinline-max-per-routine-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the maximum number of times the inliner may inline into a particular routine. It limits the number of times that inlining can be applied to any routine.

If you specify `-no-inline-max-per-routine` (Linux* and macOS) or `/Qinline-max-per-routine-` (Windows*), there is no limit to the number of times some routine may be inlined into a particular routine.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-factor, Qinline-factor` compiler option

`qopt-report, Qopt-report` compiler option

inline-max-size, Qinline-max-size

Specifies the lower limit for the size of what the inliner considers to be a large routine.

Syntax

Linux OS:

`-inline-max-size=`*n*

`-no-inline-max-size`

macOS:

`-inline-max-size=`*n*

`-no-inline-max-size`

Windows OS:

`/Qinline-max-size=`*n*

`/Qinline-max-size-`

Arguments

n	Is a positive integer that specifies the minimum size of what the inliner considers to be a large routine.

Default

`-inline-max-size` or `/Qinline-max-size`	The compiler sets the maximum size (*n*) dynamically, based on the platform.

Description

This option specifies the lower limit for the size of what the inliner considers to be a large routine (a function). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be medium and large-size routines.

The inliner prefers to inline small routines. It has a preference against inlining large routines. So, any large routine is highly unlikely to be inlined.

If you specify `-no-inline-max-size` (Linux* and macOS) or `/Qinline-max-size-` (Windows*), there are no large routines. Every routine is either a small or medium routine.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-min-size, Qinline-min-size` compiler option

`inline-factor, Qinline-factor` compiler option

`qopt-report, Qopt-report` compiler option

inline-max-total-size, Qinline-max-total-size

Specifies how much larger a routine can normally grow when inline expansion is performed.

Syntax

Linux OS:

`-inline-max-total-size=`*n*

`-no-inline-max-total-size`

macOS:

`-inline-max-total-size=`*n*

`-no-inline-max-total-size`

Windows OS:

`/Qinline-max-total-size=n`

`/Qinline-max-total-size-`

Arguments

n	Is a positive integer that specifies the permitted increase in the routine's size when inline expansion is performed.

Default

`-no-inline-max-total-size` or `/Qinline-max-total-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies how much larger a routine can normally grow when inline expansion is performed. It limits the potential size of the routine. For example, if 2000 is specified for *n*, the size of any routine will normally not increase by more than 2000.

If you specify `-no-inline-max-total-size` (Linux* and macOS) or `/Qinline-max-total-size-` (Windows*), there is no limit to the size a routine may grow when inline expansion is performed.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-factor, Qinline-factor` compiler option

`qopt-report, Qopt-report` compiler option

inline-min-caller-growth, Qinline-min-caller-growth

Lets you specify a function size n for which functions of size <= n do not contribute to the estimated growth of the caller when inlined.

Syntax

Linux OS:

`-inline-min-caller-growth=n`

macOS:

`-inline-min-caller-growth=`*n*

Windows OS:

`/Qinline-min-caller-growth=`*n*

Arguments

n	Is a non-negative integer. When *n* > 0, functions with a size of *n* are treated as if they are size 0.

Default

`-inline-min-caller-growth=0` or `/Qinline-min-caller-growth=0`	The compiler treats functions as if they have size zero.

Description

This option lets you specify a function size *n* for which functions of size <= *n* do not contribute to the estimated growth of the caller when inlined. It allows you to inline functions that the compiler would otherwise consider too large to inline.

> **NOTE**
> We recommend that you choose a value of *n* <= 10; otherwise, compile time and code size may greatly increase.

IDE Equivalent

None

Alternate Options

None

inline-min-size, Qinline-min-size

Specifies the upper limit for the size of what the inliner considers to be a small routine.

Syntax

Linux OS:

`-inline-min-size=`*n*

`-no-inline-min-size`

macOS:

`-inline-min-size=`*n*

`-no-inline-min-size`

Windows OS:

`/Qinline-min-size=`*n*

`/Qinline-min-size-`

Arguments

n	Is a positive integer that specifies the maximum size of what the inliner considers to be a small routine.

Default

`-no-inline-min-size` or `/Qinline-min-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the upper limit for the size of what the inliner considers to be a small routine (a function). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be small and medium-size routines.

The inliner has a preference to inline small routines. So, when a routine is smaller than or equal to the specified size, it is very likely to be inlined.

If you specify `-no-inline-min-size` (Linux* and macOS) or `/Qinline-min-size-` (Windows*), there is no limit to the size of small routines. Every routine is a small routine; there are no medium or large routines.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

> **Caution**
> When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-max-size, Qinline-max-size` compiler option
`qopt-report, Qopt-report` compiler option

Qinline-dllimport

Determines whether dllimport functions are inlined.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

`/Qinline-dllimport`

`/Qinline-dllimport-`

Arguments

None

Default

`/Qinline-dllimport`	The dllimport functions are inlined.

Description

This option determines whether dllimport functions are inlined. To disable dllimport functions from being inlined, specify `/Qinline-dllimport-`.

IDE Equivalent

None

Alternate Options

None

Output, Debug, and Precompiled Header Options

This section contains descriptions for compiler options that pertain to output, debugging, or precompiled headers (PCH).

c

Prevents linking.

Syntax

Linux OS:

`-c`

macOS:

`-c`

Windows OS:

`/c`

Arguments

None

Default

OFF	Linking is performed.

Description

This option prevents linking. Compilation stops after the object file is generated.

The compiler generates an object file for each C or C++ source file or preprocessed source file. It also takes an assembler file and invokes the assembler to generate an object file.

IDE Equivalent

None

Alternate Options

None

debug (Linux* and macOS)

Enables or disables generation of debugging information.

Syntax

Linux OS:

`-debug [keyword]`

macOS:

`-debug [keyword]`

Windows OS:

`None`

Arguments

keyword Is the type of debugging information to be generated. Possible values are:

`none`	Disables generation of debugging information.
`full` or `all`	Generates complete debugging information.
`minimal`	Generates line number information for debugging.
`[no]emit_column`	Determines whether the compiler generates column number information for debugging.
`[no]expr-source-pos`	Determines whether the compiler generates source position information at the expression level of granularity.
`[no]inline-debug-info`	Determines whether the compiler generates enhanced debug information for inlined code.
`[no]pubnames`	Determines whether the compiler generates a DWARF debug_pubnames section.
`[no]semantic-stepping`	Determines whether the compiler generates enhanced debug information useful for breakpoints and stepping.
`[no]variable-locations`	Determines whether the compiler generates enhanced debug information useful in finding scalar local variables.
`extended`	Generates complete debugging information and also sets keyword values `semantic-stepping` and `variable-locations`.
`[no]parallel` (Linux only)	Determines whether the compiler generates parallel debug code instrumentations useful for thread data sharing and reentrant call detection.

For information on the non-default settings for these keywords, see the Description section.

Default

varies	Normally, the default is `-debug none` and no debugging information is generated. However, on Linux*, the `-debug inline-debug-info` option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`).

Description

This option enables or disables generation of debugging information.

By default, enabling debugging, will disable optimization. To enable both debugging and optimization use the `-debug` option together with one of the optimization level options (`-O3`, `-O2` or `-O3`).

Keywords `semantic-stepping`, `inline-debug-info`, `variable-locations`, and `extended` can be used in combination with each other. If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	**Description**
`-debug none`	Disables generation of debugging information.
`-debug full` or `-debug all`	Generates complete debugging information. It is the same as specifying `-debug` with no keyword.
`-debug minimal`	Generates line number information for debugging.
`-debug emit_column`	Generates column number information for debugging.
`-debug expr-source-pos`	Generates source position information at the statement level of granularity.
`-debug inline-debug-info`	Generates enhanced debug information for inlined code. On inlined functions, symbols are (by default) associated with the caller. This option causes symbols for inlined functions to be associated with the source of the called function.
`-debug pubnames`	The compiler generates a DWARF debug_pubnames section. This provides a means to list the names of global objects and functions in a compilation unit.
`-debug semantic-stepping`	Generates enhanced debug information useful for breakpoints and stepping. It tells the debugger to stop only at machine instructions that achieve the final effect of a source statement. For example, in the case of an assignment statement, this might be a store instruction that assigns a value to a program variable; for a function call, it might be the machine instruction that executes the call. Other instructions generated for those source statements are not displayed during stepping. This option has no impact unless optimizations have also been enabled.
`-debug variable-locations`	Generates enhanced debug information useful in finding scalar local variables. It uses a feature of the Dwarf object module known as "location lists".

Option	Description
	This feature allows the run-time locations of local scalar variables to be specified more accurately; that is, whether, at a given position in the code, a variable value is found in memory or a machine register.
`-debug extended`	Sets keyword values `semantic-stepping` and `variable-locations`. It also tells the compiler to include column numbers in the line information. Generates complete debugging information and also sets keyword values `semantic-stepping` and `variable-locations`. This is a more powerful setting than `-debug full` or `-debug all`.
`-debug parallel`	Generates parallel debug code instrumentations needed for the thread data sharing and reentrant call detection. For shared data and reentrancy detection, option `-qopenmp` must be set.

On Linux* systems, debuggers read debug information from executable images. As a result, information is written to object files and then added to the executable by the linker.

On macOS systems, debuggers read debug information from object files. As a result, the executables don't contain any debug information. Therefore, if you want to be able to debug on these systems, you must retain the object files.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Advanced Debugging > Enable Parallel Debug Checks** (`-debug parallel`)

Debug > Enable Expanded Line Number Information (`-debug expr-source-pos`)

OS X

Xcode: None

Alternate Options

For `-debug full`, `-debug all`, or `-debug`	Linux and macOS: `-g` Windows: `/debug:full`, `/debug:all`, or `/debug`
For `-debug variable-locations`	Linux and macOS: `-fvar-tracking` Windows: None
For `-debug semantic-stepping`	Linux and macOS: `-fvar-tracking-assignments` Windows: None

See Also

`debug` (Windows*) compiler option

`qopenmp, Qopenmp` compiler option

debug (Windows*)

Enables or disables generation of debugging information.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/debug[:*keyword*]

Arguments

keyword	Is the type of debugging information to be generated. Possible values are:	
	none	Disables generation of debugging information.
	full or all	Generates complete debugging information.
	minimal	Generates line number information for debugging.
	partial	Deprecated. Generates global symbol table information needed for linking.
	[no]expr-source-pos	Determines whether the compiler generates source position information at the expression level of granularity.
	[no]inline-debug-info	Determines whether the compiler generates enhanced debug information for inlined code.

For information on the non-default settings for these keywords, see the Description section.

Default

/debug:none	This is the default on the command line and for a release configuration in the IDE.
/debug:all	This is the default for a debug configuration in the IDE.

Description

This option enables or disables generation of debugging information. It is passed to the linker.

By default, enabling debugging, will disable optimization. To enable both debugging and optimization use the /debug option together with one of the optimization level options (/O3, /O2 or /O3).

If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	Description
/debug:none	Disables generation of debugging information.

Option	Description
`/debug:full` or `/debug:all`	Generates complete debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking. It is the same as specifying `/debug` with no keyword.
`/debug:minimal`	Generates line number information for debugging.
`/debug:partial`	Generates global symbol table information needed for linking, but not local symbol table information needed for debugging. This option is deprecated and is not available in the IDE.
`/debug:expr-source-pos`	Generates source position information at the statement level of granularity.
`/debug:inline-debug-info`	Generates enhanced debug information for inlined code. On inlined functions, symbols are (by default) associated with the caller. This option causes symbols for inlined functions to be associated with the source of the called function.

IDE Equivalent

Windows

Visual Studio: **Debugging > Enable Expanded Line Number Information** (`/debug:expr-source-pos`)

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

For `/debug:all` or `/debug`	Linux and macOS: None Windows: `/Zi`

See Also

`debug` (Linux* and macOS) compiler option

Fa

Specifies that an assembly listing file should be generated.

Syntax

Linux OS:

`-Fa[`*`filename`*`|`*`dir`*`]`

macOS:

`-Fa[`*`filename`*`|`*`dir`*`]`

Windows OS:

`/Fa[`*`filename`*`|`*`dir`*`]`

Arguments

filename	Is the name of the assembly listing file.
dir	Is the directory where the file should be placed. It can include *filename*.

Default

OFF	No assembly listing file is produced.

Description

This option specifies that an assembly listing file should be generated (optionally named *filename*).

IDE Equivalent

Windows

Visual Studio: **Output Files > ASM List Location**

Linux

Eclipse: **Output > Generate Assembler Source and Binary Files**

OS X

Xcode: **Output Files > Filename for Generated Assembler Listing, Output > Generate Assembler Listing**

Alternate Options

Linux and macOS: `-S`

Windows: `/S`

FA

Specifies the contents of an assembly listing file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/FA[`*`specifier`*`]`

Arguments

specifier	Denotes the contents of the assembly listing file. Possible values are `c`, `s`, or `cs`.

Default

OFF	No source or machine code annotations appear in the assembly listing file, if one is produced.

Description

These options specify what information, in addition to the assembly code, should be generated in the assembly listing file.

To use this option, you must also specify option `/Fa`, which causes an assembly listing to be generated.

Option	Description
`/FA`	Produces an assembly listing without source or machine code annotations.
`/FAc`	Produces an assembly listing with machine code annotations.
`/FAs`	Produces an assembly listing with source code annotations. Note that if you use alternate option `-fsource-asm`, you must also specify the `-S` option.
`/FAcs`	Produces an assembly listing with source and machine code annotations.

IDE Equivalent

Windows

Visual Studio: **Output Files > Assembler Output**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

`/FAc`	Linux and macOS: `-fcode-asm` Windows: None
`/FAs`	Linux and macOS: `-fsource-asm` Windows: None

fasm-blocks

Enables the use of blocks and entire functions of assembly code within a C or C++ file.

Syntax

Linux OS:

`-fasm-blocks`

macOS:

`-fasm-blocks`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler allows a GNU*-style inline assembly format.

Description

This option enables the use of blocks and entire functions of assembly code within a C or C++ file.

It allows a Microsoft* MASM-style inline assembly block not a GNU*-style inline assembly block.

On macOS systems, this option is provided for compatibility with the Apple* GNU compiler.

IDE Equivalent

None

Alternate Options

`-use-msasm`

FC

Displays the full path of source files passed to the compiler in diagnostics.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/FC`

Arguments

None

Default

OFF	The compiler does not display the full path of source files passed to the compiler in diagnostics.

Description

Displays the full path of source files passed to the compiler in diagnostics. This option is supported with Microsoft Visual Studio .NET 2003* or newer.

IDE Equivalent

Windows

Visual Studio: **Advanced > Use Full Paths**

Alternate Options

None

fcode-asm

Produces an assembly listing with machine code annotations.

Syntax

Linux OS:

`-fcode-asm`

macOS:

`-fcode-asm`

Windows OS:

`None`

Arguments

None

Default

OFF	No machine code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with machine code annotations.

The assembly listing file shows the hex machine instructions at the beginning of each line of assembly code. The file cannot be assembled; the file name is the name of the source file with an extension of `.cod`.

To use this option, you must also specify option `-S`, which causes an assembly listing to be generated.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/FAc`

See Also

S compiler option

Fd

Lets you specify a name for a program database (PDB) file created by the compiler.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Fd[:`*`filename`*`]`

Arguments

filename	Is the name for the PDB file. It can include a path. If you do not specify a file extension, the extension .pdb is used.

Default

OFF	No PDB file is created unless you specify option `/Zi`. If you specify option `/Zi` and `/Fd`, the default filename is vc*x*0.pdb, where *x* represents the version of Visual C++, for example vc100.pdb.

Description

This option lets you specify a name for a program database (PDB) file that is created by the compiler.

A program database (PDB) file holds debugging and project state information that allows incremental linking of a Debug configuration of your program. A PDB file is created when you build with option `/Zi`. Option `/Fd` has no effect unless you specify option `/Zi`.

IDE Equivalent

Windows

Visual Studio: **Output Files > Program Database File Name**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

Zi, Z7, ZI compiler option
pdbfile compiler option

FD

Generates file dependencies related to the Microsoft C/C++ compiler.*

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/FD`

Arguments

None

Default

OFF	The compiler does not generate Microsoft C/C++-related file dependencies.

Description

This option generates file dependencies related to the Microsoft* C/C++ compiler. It invokes the Microsoft C/C++ compiler and passes the option to it.

IDE Equivalent

None

Alternate Options

None

Fe

Specifies the name for a built program or dynamic-link library.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Fe[[:]`*`filename`*`|`*`dir`*`]`

Arguments

filename	Is the name for the built program or dynamic-link library.
dir	Is the directory where the built program or dynamic-link library should be placed. It can include *file*.

Default

OFF	The name of the file is the name of the first source file on the command line with file extension `.exe`, so `file.f` becomes `file.exe`.

Description

This option specifies the name for a built program (`.EXE`) or a dynamic-link library (`.DLL`).

You can use this option to specify an alternate name for an executable file. This is especially useful when compiling and linking a set of input files. You can use the option to give the resulting file a name other than that of the first input file (source or object) on the command line.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-o`

Windows: None

Example

In the following example, the command produces an executable file named outfile.exe as a result of compiling and linking three files: one object file and two C++ source files.

```
prompt> icl /Feoutfile.exe file1.obj file2.cpp file3.cpp
```

By default, this command produces an executable file named file1.exe.

See Also

- compiler option

feliminate-unused-debug-types, Qeliminate-unused-debug-types

Controls the debug information emitted for types declared in a compilation unit.

Syntax

Linux OS:

`-feliminate-unused-debug-types`

`-fno-eliminate-unused-debug-types`

macOS:

`-feliminate-unused-debug-types`

`-fno-eliminate-unused-debug-types`

Windows OS:

`/Qeliminate-unused-debug-types`

`/Qeliminate-unused-debug-types-`

Arguments

None

Default

`-feliminate-unused-debug-types` or `/Qeliminate-unused-debug-types`	The compiler emits debug information only for types that are actually used by a variable/parameter/etc..

Description

This option controls the debug information emitted for types declared in a compilation unit.

If you specify `-fno-eliminate-unused-debug-types` (Linux and macOS) or `/Qeliminate-unused-debug-types-`, it will cause the compiler to emit debug information for all types present in the sources. This option may cause a large increase in the size of the debug information.

IDE Equivalent

None

Alternate Options

None

femit-class-debug-always

Controls the format and size of debug information generated by the compiler for C++ classes.

Syntax

Linux OS:

`-femit-class-debug-always`

`-fno-emit-class-debug-always`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-fno-emit-class-debug-always`	Causes the compiler to reduce the amount of debug information generated for C++ classes.

Description

When emission of debug information is enabled, this option will control the format and size of debug information generated by the compiler for C++ classes. It tells the compiler to generate full debug information, or it tells the compiler to reduce the amount of debug information it generates.

When you specify the -femit-class-debug-always option, the compiler emits debug information for a C++ class into each object file where the class is used. This option is useful for tools that are not able to resolve incomplete type descriptions. Note that this option may cause a large increase in the size of the debug information generated.

When you specify the -fno-emit-class-debug-always option, the compiler does not emit full debug information for every instance of C++ class use. In general, this reduces the size of the debugging information generated for C++ applications without impacting debugging ability when used with debuggers that have corresponding support, such as gdb.

IDE Equivalent

None

Alternate Options

None

fmerge-constants

Determines whether the compiler and linker attempt to merge identical constants (string constants and floating-point constants) across compilation units.

Syntax

Linux OS:

`-fmerge-constants`

`-fno-merge-constants`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-fmerge-constants`	The compiler and linker attempt to merge identical constants across compilation units if the compiler and linker supports it.

Description

This option determines whether the compiler and linker attempt to merge identical constants (string constants and floating-point constants) across compilation units.

If you do not want the compiler and linker to attempt to merge identical constants across compilation units. specify `-fno-merge-constants`.

IDE Equivalent

None

Alternate Options

None

fmerge-debug-strings

Causes the compiler to pool strings used in debugging information.

Syntax

Linux OS:

`-fmerge-debug-strings`

`-fno-merge-debug-strings`

macOS:

None

Windows OS:

None

Arguments

None

Default

`-fmerge-debug-strings`	The compiler will pool strings used in debugging information.

Description

This option causes the compiler to pool strings used in debugging information. The linker will automatically retain this pooling.

This option can reduce the size of debug information, but it may produce slightly slower compile and link times.

This option is only turned on by default if you are using gcc 4.3 or later, where this setting is also the default, since the generated debug tables require binutils version 2.17 or later to work reliably.

If you do not want the compiler to pool strings used in debugging information, specify option `-fno-merge-debug-strings`.

IDE Equivalent

None

Alternate Options

None

Fo

Specifies the name for an object file.

Syntax

Linux OS:

See option `o`.

macOS:

See option `o`.

Windows OS:

`/Fo[[:]`*`filename`*`|`*`dir`*`]`

Arguments

filename	Is the name for the object file.
dir	Is the directory where the object file should be placed. It can include *filename*.

Default

OFF	An object file has the same name as the name of the first source file and a file extension of .obj.

Description

This option specifies the name for an object file.

IDE Equivalent

Windows

Visual Studio: **Output Files > Object File Name**

Alternate Options

None

See Also

- compiler option

Fp

Lets you specify an alternate path or file name for precompiled header files.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Fp{filename|dir}`

Arguments

filename	Is the name for the precompiled header file.
dir	Is the directory where the precompiled header file should be placed. It can include *filename*.

Default

OFF	The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option lets you specify an alternate path or file name for precompiled header files.

IDE Equivalent

Windows

Visual Studio: **Precompiled Headers > Precompiled Header Output File**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

FR

Invokes the Microsoft C/C++ compiler and tells it to produce a BSCMAKE .sbr file with complete symbolic information.*

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/FR[filename|dir]`

Arguments

filename	Is the name for the BSCMAKE `.sbr` file.
dir	Is the directory where the file should be placed. It can include *filename*.

Default

OFF	The compiler does not invoke the Microsoft* C/C++ compiler to produce a .sbr file.

Description

This option invokes the Microsoft* C/C++ compiler and tells it to produce a BSCMAKE .sbr file with complete symbolic information.

You can provide a name for the file. If you do not specify a file name, the .sbr file gets the same base name as the source file.

A synonym for option `/FR` is option `/Fr`. Option `/Fr` is a deprecated option.

IDE Equivalent

Windows

Visual Studio: **Browse Information > Browse Information File**

Browse Information > Enable Browse Information

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

fsource-asm

Produces an assembly listing with source code annotations.

Syntax

Linux OS:

`-fsource-asm`

macOS:

`-fsource-asm`

Windows OS:

`None`

Arguments

None

Default

OFF	No source code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with source code annotations. The assembly listing file shows the source code as interspersed comments.

To use this option, you must also specify option `-S`, which causes an assembly listing to be generated.

IDE Equivalent

None

Alternate Options

None

See Also

`S` compiler option

ftrapuv, Qtrapuv

Initializes stack local variables to an unusual value to aid error detection.

Syntax

Linux OS:

`-ftrapuv`

macOS:

`-ftrapuv`

Windows OS:

`/Qtrapuv`

Arguments

None

Default

OFF	The compiler does not initialize local variables.

Description

This option initializes stack local variables to an unusual value to aid error detection. Normally, these local variables should be initialized in the application. It also unmasks the floating-point invalid exception.

The option sets any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors.

This option sets option `-g` (Linux* and macOS) and `/Zi` or `/Z7` (Windows*), which changes the default optimization level from `O2` to `-O0` (Linux and macOS) or `/Od` (Windows). You can override this effect by explicitly specifying an `O` option setting.

If option `O2` and option `-ftrapuv` (Linux and macOS) or `/Qtrapuv` (Windows) are used together, you should specify option `-fp-speculation safe` (Linux and macOS) or `/Qfp-speculation:safe` (Windows) to prevent exceptions resulting from speculated floating-point operations from being trapped.

For more details on using options `-ftrapuv` and `/Qtrapuv` with compiler option `O`, see the article titled Don't optimize when using -ftrapuv for uninitialized variable detection.

Another way to detect uninitialized local scalar variables is by specifying keyword `uninit` for option `check`.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Run-Time > Initialize Stack Variables to an Unusual Value**

OS X

Xcode: **Code Generation > Initialize Stack Variables to an Unusual Value**

Alternate Options

None

See Also

g compiler option

Zi, Z7, ZI compiler option
`O` compiler option
`check` compiler option (see setting uninit)

fverbose-asm

Produces an assembly listing with compiler comments, including options and version information.

Syntax

Linux OS:

`-fverbose-asm`

`-fno-verbose-asm`

macOS:

`-fverbose-asm`

`-fno-verbose-asm`

Windows OS:

`None`

Arguments

None

Default

`-fno-verbose-asm`	No source code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with compiler comments, including options and version information.

To use this option, you must also specify `-S`, which sets `-fverbose-asm`.

If you do not want this default when you specify `-S`, specify `-fno-verbose-asm`.

IDE Equivalent

None

Alternate Options

None

See Also

`S` compiler option

g

Tells the compiler to generate a level of debugging information in the object file.

Syntax

Linux OS:

`-g[n]`

macOS:

`-g[n]`

Windows OS:

`See option Zi, Z7, ZI.`

Arguments

n	Is the level of debugging information to be generated. Possible values are:	
	0	Disables generation of symbolic debug information.
	1	Produces minimal debug information for performing stack traces.
	2	Produces complete debug information. This is the same as specifying `-g` with no *n*.
	3	Produces extra information that may be useful for some tools.

Default

-g or -g2	The compiler produces complete debug information.

Description

Option `-g` tells the compiler to generate symbolic debugging information in the object file, which increases the size of the object file.

The compiler does not support the generation of debugging information in assemblable files. If you specify this option, the resulting object file will contain debugging information, but the assemblable file will not.

This option turns off option `-O2` and makes option `-O0` the default unless option `-O2` (or higher) is explicitly specified in the same command line.

Specifying the `-g` or `-O0` option sets the `-fno-omit-frame-pointer` option. On Linux*, the -debug inline-debug-info option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`).

Specifying the `-g` or `-O0` option sets the `-fno-omit-frame-pointer` option.

NOTE
When option `-g` is specified, debugging information is generated in the DWARF Version 3 format. Older versions of some analysis tools may require applications to be built with the `-gdwarf-2` option to ensure correct operation.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **General > Include Debug Information**

Xcode

Xcode: **General > Generate Debug Information**

Alternate Options

Linux: None

Windows: /Zi, /Z7, /ZI

See Also

gdwarf compiler option
Zi, Z7, ZI compiler option
debug (Linux* and macOS) compiler option

gdwarf

Lets you specify a DWARF Version format when generating debug information.

Syntax

Linux OS:

`-gdwarf-n`

macOS:

`-gdwarf-n`

Windows OS:

None

Arguments

n	Is a value denoting the DWARF Version format to use. Possible values are:	
	`2`	Generates debug information using the DWARF Version 2 format.
	`3`	Generates debug information using the DWARF Version 3 format.
	`4`	Generates debug information using the DWARF Version 4 format. This setting is only available on Linux*.

Default

OFF	No debug information is generated. However, if compiler option `-g` is specified, debugging information is generated in the DWARF Version 3 format.

Description

This option lets you specify a DWARF Version format when generating debug information.

Note that older versions of some analysis tools may require applications to be built with the -gdwarf-2 option to ensure correct operation.

IDE Equivalent

None

Alternate Options

None

See Also

`g` compiler option

Gm

Enables a minimal rebuild.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Gm`

Arguments

None

Default

OFF	Minimal rebuilds are disabled.

Description

This option enables a minimal rebuild.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Enable Minimal Rebuild**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

grecord-gcc-switches

Causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.

Syntax

Linux OS:

`-grecord-gcc-switches`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The command line options that were used to invoke the compiler are not appended to the DW_AT_producer attribute in DWARF debugging information.

Description

This option causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.

The options are concatenated with whitespace separating them from each other and from the compiler version.

IDE Equivalent

None

Alternate Options

None

gsplit-dwarf

Creates a separate object file containing DWARF debug information.

Syntax

Linux OS:

`-gsplit-dwarf`

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	No separate object file containing DWARF debug information is created.

Description

This option creates a separate object file containing DWARF debug information. It causes debug information to be split between the generated object (.o) file and the new DWARF object (.dwo) file.

The DWARF object file is not used by the linker, so this reduces the amount of debug information the linker must process and it results in a smaller executable file.

For this option to perform correctly, you must use binutils-2.24 or later. To debug the resulting executable, you must use gdb-7.6.1 or later.

> **NOTE**
> If you use the split executable with a tool that does not support the split DWARF format, it will behave as though the DWARF debug information is absent.

IDE Equivalent

None

Alternate Options

None

map-opts, Qmap-opts

Maps one or more compiler options to their equivalent on a different operating system.

Syntax

Linux OS:

`-map-opts`

macOS:

None

Windows OS:

`/Qmap-opts`

Arguments

None

Default

OFF	No platform mappings are performed.

Description

This option maps one or more compiler options to their equivalent on a different operating system. The result is output to `stdout`.

On Windows systems, the options you provide are presumed to be Windows options, so the options that are output to `stdout` will be Linux equivalents.

On Linux systems, the options you provide are presumed to be Linux options, so the options that are output to `stdout` will be Windows equivalents.

The tool can be invoked from the compiler command line or it can be used directly.

No compilation is performed when the option mapping tool is used.

This option is useful if you have both compilers and want to convert scripts or makefiles.

> **NOTE**
> Compiler options are mapped to their equivalent on the architecture you are using. For example, if you are using a processor with Intel® 64 architecture, you will only see equivalent options that are available on processors with Intel® 64 architecture.

IDE Equivalent

None

Alternate Options

None

Example

The following command line invokes the option mapping tool, which maps the Linux options to Windows-based options, and then outputs the results to `stdout`:

```
icc -map-opts -xP -O2
```

The following command line invokes the option mapping tool, which maps the Windows options to Linux-based options, and then outputs the results to `stdout`:

```
icl /Qmap-opts /QxP /O2
```

See Also

Compiler Option Mapping Tool

o

Specifies the name for an output file.

Syntax

Linux OS:

`-o filename`

macOS:

`-o filename`

Windows OS:

See option Fo.

Arguments

filename	Is the name for the output file. The space before *filename* is optional.

Default

OFF	The compiler uses the default file name for an output file.

Description

This option specifies the name for an output file as follows:

- If `-c` is specified, it specifies the name of the generated object file.
- If `-S` is specified, it specifies the name of the generated assembly listing file.
- If `-P` is specified, it specifies the name of the generated preprocessor file.

Otherwise, it specifies the name of the executable file.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/Fe`

See Also

`Fo` compiler option
`Fe` compiler option

pch

Tells the compiler to use appropriate precompiled header files.

Syntax

Linux OS and macOS:

`-pch`

Windows OS:

None

Arguments

None

Default

OFF The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option tells the compiler to use appropriate precompiled header (PCH) files. If none are available, they are created as `sourcefile.pchi`. This option is supported for multiple source files.

The `-pch` option will use PCH files created from other sources if the headers files are the same. For example, if you compile `source1.cpp` using `-pch`, then `source1.pchi` is created. If you then compile `source2.cpp` using `-pch`, the compiler will use `source1.pchi` if it detects the same headers.

Caution
Depending on how you organize the header files listed in your sources, this option may increase compile times.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Precompiled Headers > Automatic Processing for Precompiled Headers**

OS X

Xcode: None

Alternate Options

None

Example

Consider the following command line:

```
icpc -pch source1.cpp source2.cpp
```

It produces the following output when `.pchi` files exist:

```
"source1.cpp": using precompiled header file"source1.pchi"
"source2.cpp": using precompiled header file "source2.pchi"
```

It produces the following output when `.pchi` files *do not* exist:

```
"source1.cpp": creating precompiled header file "source1.pchi"
"source2.cpp": creating precompiled header file "source2.pchi"
```

See Also

`-pch-create` compiler option

`-pch-dir` compiler option

`-pch-use` compiler option

pch-create

Tells the compiler to create a precompiled header file.

Syntax

Linux OS and macOS:

`-pch-create` *filename*

Windows OS:

None

Arguments

filename	Is the name for the precompiled header file. A space must appear before the file name. It can include a path.

Default

OFF	The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option tells the compiler to create a precompiled header (PCH) file. It is supported only for single source file compilations.

Note that the .pchi extension is not automatically appended to the file name.

This option cannot be used in the same compilation as the `-pch-use` option.

On Windows* systems, option `-pch-create` is equivalent to the `/Yc` option.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/Yc`

Example

Consider the following command line:

```
icpc -pch-create /pch/foo.pchi foo.cpp
```

This creates the precompiled header file "/pch/foo.pchi".

See Also

`pch-use` compiler option

pch-dir

Tells the compiler the location for precompiled header files.

Syntax

Linux OS and macOS:

`-pch-dir` *dir*

Windows OS:

None

Arguments

dir	Is the path for precompiled header files. The path must exist.

Default

OFF	The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option tells the compiler the location for precompiled header files. It denotes where to find precompiled header files, and where new PCH files should be placed.

This option can be used with the `-pch`, `-pch-create`, and `-pch-use` options.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Precompiled Headers > Precompiled Headers' File Directory**

OS X

Xcode: **Precompiled Headers > Prefix Header**

Alternate Options

None

Example

Consider the following command line:

```
icpc -pch -pch-dir /pch source32.cpp
```

It produces the following output:

```
"source32.cpp": creating precompiled header file /pch/source32.pchi
```

See Also

`pch` compiler option

`pch-create` compiler option

`pch-use` compiler option

pch-use

Tells the compiler to use a precompiled header file.

Syntax

Linux OS and macOS:

`-pch-use` *`filename`*

Windows OS:

None

Arguments

filename	Is the name of the precompiled header file to use. A space must appear before the file name. It can include a path.

Default

OFF	The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option tells the compiler to use a precompiled header (PCH) file.

It is supported for multiple source files when all source files use the same .pchi file.

This option cannot be used in the same compilation as the `-pch-create` option.

To learn how to optimize compile times using the PCH options, see "Using Precompiled Header Files" in the User's Guide.

On Windows* systems, option `-pch-use` is equivalent to the `/Yu` option.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/Yu`

Example

Consider the following command line:

```
icpc -pch-use /pch/source32.pchi source.cpp
```

It produces the following output:

```
"source.cpp": using precompiled header file /pch/source32.pchi
```

See Also

`-pch-create` compiler option

pdbfile

Lets you specify the name for a program database (PDB) file created by the linker.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/pdbfile[:`*`filename`*`]`

Arguments

filename	Is the name for the PDB file. It can include a path. If you do not specify a file extension, the extension .pdb is used.

Default

OFF	No PDB file is created unless you specify option `/Zi`. If you specify option `/Zi` the default filename is *executablename*.pdb.

Description

This option lets you specify the name for a program database (PDB) file created by the linker. This option does not affect where the compiler outputs debug information.

To use this option, you must also specify option `/debug:full` or `/Zi`.

If *filename* is not specified, the default file name used is the name of your file with an extension of .pdb.

IDE Equivalent

None

Alternate Options

None

See Also

Zi, Z7, ZI compiler option
`debug` compiler option
`Fd` compiler option

print-multi-lib

Prints information about where system libraries should be found.

Syntax

Linux OS:

`-print-multi-lib`

macOS:

`-print-multi-lib`

Windows OS:

`None`

Arguments

None

Default

OFF	No information is printed unless the option is specified.

Description

This option prints information about where system libraries should be found, but no compilation occurs. On Linux* systems, it is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

Qpchi

Enable precompiled header coexistence to reduce build time.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qpchi`

`/Qpchi-`

Arguments

None

Default

ON	The compiler enables precompiled header coexistence.

Description

This option enables precompiled header (PCH) files generated by the Intel® C++ compiler and those generated by the Microsoft Visual C++* compiler to coexist, which reduces build time.

If build time is not an issue and you do not want an additional set of PCH files on your system, specify `/Qpchi-`.

IDE Equivalent

None

Alternate Options

None

Quse-msasm-symbols

Tells the compiler to use a dollar sign ("$") when producing symbol names.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Quse-msasm-symbols`

Arguments

None

Default

OFF	The compiler uses a period (".") when producing symbol names

Description

This option tells the compiler to use a dollar sign ("$") when producing symbol names.

Use this option if you require symbols in your .asm files to contain characters that are accepted by the MS assembler.

IDE Equivalent

None

Alternate Options

None

RTC

Enables checking for certain run-time conditions.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/RTC`*`option`*

Arguments

option	Specifies the condition to check. Possible values are 1, s, u, or c.

Default

OFF	No checking is performed for these run-time conditions.

Description

This option enables checking for certain run-time conditions. Using the /RTC option sets __MSVC_RUNTIME_CHECKS = 1.

Option	Description
/RTC1	This is the same as specifying /RTCsu.
/RTCs	Enables run-time checks of the stack frame.
/RTCu	Enables run-time checks for unintialized variables.
/RTCc	Enables checks for converting to smaller types.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Basic Runtime Checks / Smaller Type Check**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

S

Causes the compiler to compile to an assembly file only and not link.

Syntax

Linux OS:

-S

macOS:

-S

Windows OS:

/S

Arguments

None

Default

OFF	Normal compilation and linking occur.

Description

This option causes the compiler to compile to an assembly file only and not link.

On Linux* and macOS systems, the assembly file name has a .s suffix. On Windows* systems, the assembly file name has an .asm suffix.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Output Files > Generate Assembler Source File**

OS X

Xcode: **Output Files > Generate Assembler Source File**

Alternate Options

Linux and macOS: None

Windows: `/Fa`

See Also

Fa compiler option

use-asm, Quse-asm

Tells the compiler to produce objects through the assembler. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-use-asm`

`-no-use-asm`

macOS:

`-use-asm`

`-no-use-asm`

Windows OS:

`/Quse-asm`

`/Quse-asm-`

Arguments

None

Default

OFF	The compiler produces objects directly.

Description

This option tells the compiler to produce objects through the assembler.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

None

Alternate Options

None

use-msasm

Enables the use of blocks and entire functions of assembly code within a C or C++ file.

Syntax

Linux OS:

`-use-msasm`

macOS:

`-use-msasm`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler allows a GNU*-style inline assembly format.

Description

This option enables the use of blocks and entire functions of assembly code within a C or C++ file.

It allows a Microsoft* MASM-style inline assembly block not a GNU*-style inline assembly block.

IDE Equivalent

None

Alternate Options

`-fasm-blocks`

V (Windows*)

Places the text string specified into the object file being generated by the compiler.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/V`*`string`*

Arguments

string	Is the text string to go into the object file.

Default

OFF	No text string is placed in the object file.

Description

Places the text string specified into the object file (.obj) being generated by the compiler.

This option places the text string specified into the object file (.obj) being generated by the compiler. The string also gets propagated into the executable file.

For example, this option is useful if you want to place the version number or copyright information into the object and executable.

If the string contains a space or tab, the string must be enclosed by double quotation marks ("). A backslash (\) must precede any double quotation marks contained within the string.

IDE Equivalent

None

Alternate Options

None

Y-

Tells the compiler to ignore all other precompiled header files.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Y-`

Arguments

None

Default

OFF	The compiler recognizes precompiled header files when certain compiler options are specified.

Description

This option tells the compiler to ignore all other precompiled header files.

IDE Equivalent

None

Alternate Options

None

See Also

`Yc` compiler option
`Yu` compiler option

Yc

Tells the compiler to create a precompiled header file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Yc[`*`filename`*`]`

Arguments

filename	Is the name of a C/C++ header file, which is included in the source file using an #include preprocessor directive.

Default

OFF	The compiler does not create or use precompiled headers unless you tell it to do so.

Description

This option tells the compiler to create a precompiled header (PCH) file. It is supported only for single source file compilations.

When *filename* is specified, the compiler creates a precompiled header file from the headers in the C/C++ program up to and including the C/C++ header specified.

If you do not specify *filename*, the compiler compiles all code up to the end of the source file, or to the point in the source file where a hdrstop occurs. The default name for the resulting file is the name of the source file with extension .pchi.

This option cannot be used in the same compilation as the `/Yu` option.

On Linux* and macOS, option `/Yc` is equivalent to the `-pch-create` option.

IDE Equivalent

Windows

Visual Studio: **Precompiled Headers > Precompiled Header File**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

Linux and macOS: `-pch-create`

Windows: None

Example

If option /Fp is used, it names the PCH file. For example, consider the following command lines:

```
icl /c /Ycheader.h /Fpprecomp foo.cpp
icl /c /Yc /Fpprecomp foo.cpp
```

In both cases, the name of the PCH file is "precomp.pchi".

If the header file name is specified, the file name is based on the header file name. For example:

```
icl /c /Ycheader.h foo.cpp
```

In this case, the name of the PCH file is "header.pchi".

If no header file name is specified, the file name is based on the source file name. For example:

```
icl /c /Yc foo.cpp
```

In this case, the name of the PCH file is "foo.pchi".

See Also

`Yu` compiler option

`Fp` compiler option

Yd

Tells the compiler to add complete debugging information in all object files created from a precompiled header file when option /Zi or /Z7 is specified. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Yd`

Arguments

None

Default

OFF	If `/Zi` or `/Z7` is specified when you are compiling with a precompiled header file using `/Yc` or `/Yu`, only one .obj file contains the common debugging information.

Description

This option tells the compiler that complete debugging information should be added to all object files created from a precompiled header (PCH) file when option `/Zi` or `/Z7` is specified. It affects precompiled header files that were created by specifying the `/Yc` option.

This is a deprecated option that may be removed in a future release. There is no replacement option.

Option `/Yd` has no effect unless option `/Zi` or `/Z7` is specified.

When option `/Zi` or `/Z7` is specified and option `/Yd` is omitted, the compiler stores common debugging information in only the first object (.obj) file created from the PCH file. This information is not inserted into any .obj files subsequently created from the PCH file, only cross-references to the information are inserted.

IDE Equivalent

None

Alternate Options

None

Yu

Tells the compiler to use a precompiled header file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Yu[`*`filename`*`]`

Arguments

filename	Is the name of a C/C++ header file, which is included in the source file using an #include preprocessor directive.

Default

OFF	The compiler does not use precompiled header files unless it is told to do so.

Description

This option tells the compiler to use a precompiled header (PCH) file.

It is supported for multiple source files when all source files use the same .pchi file.

The compiler treats all code occurring before the header file as precompiled. It skips to just beyond the #include directive associated with the header file, uses the code contained in the PCH file, and then compiles all code after *filename*.

If you do not specify *filename*, the compiler will use a PCH with a name based on the source file name. If you specify option `/Fp`, it will use the PCH specified by that option.

When this option is specified, the compiler ignores all text, including declarations preceding the #include statement of the specified file.

This option cannot be used in the same compilation as the `/Yc` option.

On Linux* and macOS systems, option `/Yu` is equivalent to the `-pch-use` option.

IDE Equivalent

Windows

Visual Studio: **Precompiled Headers > Precompiled Header**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

Linux and macOS: `-pch-use`

Windows: None

Example

Consider the following command line:

```
icl /c /Yuheader.h bar.cpp
```

In this case, the name of the PCH file used is "header.pchi".

In the following command line, no filename is specified:

```
icl /Yu bar.cpp
```

In this case, the name of the PCH file used is "bar.pchi".

In the following command line, no filename is specified, but option `/Fp` is specified:

```
icl /Yu /Fpprecomp bar.cpp
```

In this case, the name of the PCH file used is "precomp.pchi".

See Also

`Yc` compiler option

Zi, Z7, ZI

Tells the compiler to generate full debugging information in either an object (.obj) file or a project database (PDB) file.

Syntax

Linux OS:

`See option g.`

macOS:

`See option g.`

Windows OS:

`/Zi`

`/Z7`

/ZI

Arguments

None

Default

OFF	No debugging information is produced.

Description

Option /Z7 tells the compiler to generate symbolic debugging information in the object (.obj) file for use with the debugger. No .pdb file is produced by the compiler.

Option /ZI is a synonym for option /Zi.

The /Zi option tells the compiler to generate symbolic debugging information in a program database (PDB) file for use with the debugger. Type information is placed in the .pdb file, and not in the .obj file, resulting in smaller object files in comparison to option /Z7.

When option /Zi is specified, two PDB files are created:

- The compiler creates the program database project.pdb. If you compile a file without a project, the compiler creates a database named vc*x*0.pdb, where *x* represents the major version of Visual C++, for example vc140.pdb.

  This file stores all debugging information for the individual object files and resides in the same directory as the project makefile. If you want to change this name, use option /Fd.
- The linker creates the program database executablename.pdb.

  This file stores all debug information for the .exe file and resides in the debug subdirectory. It contains full debug information, including function prototypes, not just the type information found in vc*x*0.pdb.

Both PDB files allow incremental updates. The linker also embeds the path to the .pdb file in the .exe or .dll file that it creates.

The compiler does not support the generation of debugging information in assemblable files. If you specify these options, the resulting object file will contain debugging information, but the assemblable file will not.

These options turn off option /O2 and make option /Od the default unless option /O2 (or higher) is explicitly specified in the same command line.

For more information about the /Z7, /Zi, and /ZI options, see the Microsoft documentation.

IDE Equivalent

Visual Studio

Visual Studio: **General > Generate Debug Information**

Eclipse

Eclipse: None

Xcode

Xcode: None

Alternate Options

Linux: -g

Windows: None

See Also

Fd compiler option

g compiler option

debug (Windows*) compiler option

Zo

Enables or disables generation of enhanced debugging information for optimized code.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/Zo

/Zo-

Arguments

None

Default

OFF	The compiler does not generate enhanced debugging information for optimized code.

Description

This option enables or disables the generation of additional debugging information for local variables and inlined routines when code optimizations are enabled. It should be used with option /Zi or /Z7 to allow improved debugging of optimized code.

Option /Zo enables generation of this enhanced debugging information. Option /Zo- disables this functionality.

For more information on code optimization, see option /O.

IDE Equivalent

None

Alternate Options

None

See Also

Zi, Z7, ZI compiler option

debug (Windows*) compiler option

O compiler option

Preprocessor Options

This section contains descriptions for compiler options that pertain to preprocessing.

A, QA

Specifies an identifier for an assertion.

Syntax

Linux OS and macOS:

`-Aname[(value)]`

Windows OS:

`/QAname[(value)]`

Arguments

`name`	Is the identifier for the assertion.
`value`	Is an optional value for the assertion. If a value is specified, it must be within quotes, including the parentheses delimiting it.

Default

OFF	Assertions have no identifiers or symbol names.

Description

This option specifies an identifier (symbol name) for an assertion. It is equivalent to an #assert preprocessing directive.

Note that this option is not the positive form of the C++ `/QA-` option.

On Linux* systems, because GCC has deprecated assertions, this option has no effect.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: None

OS X

Xcode: **Preprocessor > Undefine All Preprocessor Definitions**

Alternate Options

None

Example

To make an assertion for the identifier fruit with the associated values orange and banana use the following command.

On Windows* systems:

```
icl /QA"fruit(orange,banana)" prog1.cpp
```

On Linux* systems:

```
icpc -A"fruit(orange,banana)" prog1.cpp
```

On macOS systems:

```
icl++ -A"fruit(orange,banana)" prog1.cpp
```

```
icpc -A"fruit(orange,banana)" prog1.cpp
```

B

Specifies a directory that can be used to find include files, libraries, and executables.

Syntax

Linux OS:

-B*dir*

macOS:

-B*dir*

Windows OS:

None

Arguments

dir	Is the directory to be used. If necessary, the compiler adds a directory separator character at the end of *dir*.

Default

OFF	The compiler looks for files in the directories specified in your PATH environment variable.

Description

This option specifies a directory that can be used to find include files, libraries, and executables.

The compiler uses *dir* as a prefix.

For include files, the *dir* is converted to -I/*dir*/include. This command is added to the front of the includes passed to the preprocessor.

For libraries, the *dir* is converted to -L/*dir*. This command is added to the front of the standard -L inclusions before system libraries are added.

For executables, if *dir* contains the name of a tool, such as ld or as, the compiler will use it instead of those found in the default directories.

The compiler looks for include files in *dir* /include while library files are looked for in *dir*.

On Linux* systems, another way to get the behavior of this option is to use the environment variable GCC_EXEC_PREFIX.

IDE Equivalent

None

Alternate Options

None

C

Places comments in preprocessed source output.

Syntax

Linux OS:

`-C`

macOS:

`-C`

Windows OS:

`/C`

Arguments

None

Default

OFF	No comments are placed in preprocessed source output.

Description

This option places (or preserves) comments in preprocessed source output.

Comments following preprocessing directives, however, are not preserved.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Keep Comments**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

Example

The following commands cause the compiler to preserve comments in the prog1.i preprocessed file.

On Windows* systems:

```
icl /C /P prog1.cpp prog2.cpp
```

On Linux* systems:

```
icpc -C -P prog1.cpp prog2.cpp
```

On macOS systems:

```
icpc -C -P prog1.cpp prog2.cpp
```

D

Defines a macro name that can be associated with an optional value.

Syntax

Linux OS:

`-Dname[=value]`

macOS:

`-Dname[=value]`

Windows OS:

`/Dname[=value]`

Arguments

name	Is the name of the macro.
value	Is an optional integer or an optional character string delimited by double quotes; for example, `D`*name=string*.

Default

OFF	Only default symbols or macros are defined.

Description

Defines a macro name that can be associated with an optional value. This option is equivalent to a #define preprocessor directive.

If a *value* is not specified, *name* is defined as "1".

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Preprocessor Definitions**

Linux

Eclipse: **Preprocessor > Preprocessor Definitions**

OS X

Xcode: **Preprocessor > Preprocessor Definitions**

Alternate Options

None

Example

To define a macro called SIZE with the value 100, enter the following command:

On Windows* systems:

```
icl /DSIZE=100 prog1.cpp
```

On Linux* systems:

```
icpc -DSIZE=100 prog1.cpp
```

On macOS systems:

```
icpc -DSIZE=100 prog1.cpp
```

If you define a macro, but do not assign a value, the compiler defaults to 1 for the value of the macro.

See Also

Additional Predefined Macros

dD, QdD

Same as option -dM, but outputs #define directives in preprocessed source.

Syntax

Linux OS:

`-dD`

macOS:

`-dD`

Windows OS:

`/QdD`

Arguments

None

Default

OFF	The compiler does not output #define directives.

Description

Same as `-dM`, but outputs `#define` directives in preprocessed source. To use this option, you must also specify the `E` option.

IDE Equivalent

None

Alternate Options

None

dM, QdM

Tells the compiler to output macro definitions in effect after preprocessing.

Syntax

Linux OS:

`-dM`

macOS:

`-dM`

Windows OS:

`/QdM`

Arguments

None

Default

OFF	The compiler does not output macro definitions after preprocessing.

Description

This option tells the compiler to output macro definitions in effect after preprocessing. To use this option, you must also specify option `E`.

IDE Equivalent

None

Alternate Options

None

See Also

`E` compiler option

dN, QdN

Same as option -dD, but output #define directives contain only macro names.

Syntax

Linux OS and macOS:

`-dN`

Windows OS:

`/QdN`

Arguments

None

Default

OFF	The compiler does not output #define directives.

Description

Same as `-dD`, but output `#define` directives contain only macro names. To use this option, you must also specify option `E`.

IDE Equivalent

None

Alternate Options

None

E

Causes the preprocessor to send output to stdout.

Syntax

Linux OS:

`-E`

macOS:

`-E`

Windows OS:

`/E`

Arguments

None

Default

OFF	Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`. Compilation stops when the files have been preprocessed.

When you specify this option, the compiler's preprocessor expands your source module and writes the result to `stdout`. The preprocessed source contains #line directives, which the compiler uses to determine the source file and line number.

IDE Equivalent

None

Alternate Options

None

Example

To preprocess two source files and write them to `stdout`, enter the following command:

On Windows* systems:

```
icl /E prog1.cpp prog2.cpp
```

On Linux* systems:

```
icpc -E prog1.cpp prog2.cpp
```

On macOS systems:

```
icl -E prog1.cpp prog2.cpp
```

```
icpc -E prog1.cpp prog2.cpp
```

EP

Causes the preprocessor to send output to stdout, omitting #line directives.

Syntax

Linux OS:

`-EP`

macOS:

`-EP`

Windows OS:

`/EP`

Arguments

None

Default

OFF	Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`, omitting #line directives.

If you also specify option `P` or Linux* option `F`, the preprocessor will write the results (without #line directives) to a file instead of `stdout`.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Preprocess Suppress Line Numbers**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

Example

To preprocess to `stdout` omitting #line directives, enter the following command:

On Windows* systems:

```
icl /EP prog1.cpp prog2.cpp
```

On Linux* and macOS systems:

```
icpc -EP prog1.cpp prog2.cpp
```

FI

Tells the preprocessor to include a specified file name as the header file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/FI`*`filename`*

Arguments

filename	Is the file name to be included as the header file.

Default

OFF	The compiler uses default header files.

Description

This option tells the preprocessor to include a specified file name as the header file.

The file specified with `/FI` is included in the compilation before the first line of the primary source file.

IDE Equivalent

Windows

Visual Studio: **Advanced > Forced Include File**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

gcc, gcc-sys

Determines whether certain GNU macros are defined or undefined.

Syntax

Linux OS:

`-gcc`

`-no-gcc`

-gcc-sys

macOS:

-gcc

-no-gcc

-gcc-sys

Windows OS:

None

Arguments

None

Default

-gcc	The compiler defines the GNU macros `__GNUC__`, `__GNUC_MINOR__`, and `__GNUC_PATCHLEVEL__`

Description

This option determines whether the GNU macros `__GNUC__`, `__GNUC_MINOR__`, and `__GNUC_PATCHLEVEL__` are defined or undefined.

Option	Description
-gcc	Defines GNU macros.
-no-gcc	Undefines GNU macros.
-gcc-sys	Defines GNU macros only during compilation of system headers.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Preprocessor > gcc Predefined Macro Enablement**

OS X

Xcode: **Preprocessor > Predefine gcc Macros**

Alternate Options

None

gcc-include-dir

Controls whether the gcc-specific include directory is put into the system include path.

Syntax

Linux OS:

-gcc-include-dir

`-no-gcc-include-dir`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-gcc-include-dir`	The gcc-specific include directory is put into the system include path.

Description

This option controls whether the gcc-specific include directory is put into the system include path.

If you specify `-no-gcc-include-dir`, the gcc-specific include directory will not be put into the system include path.

IDE Equivalent

None

Alternate Options

None

H, QH

Tells the compiler to display the include file order and continue compilation.

Syntax

Linux OS:

`-H`

macOS:

`-H`

Windows OS:

`/QH`

Arguments

None

Default

OFF	Compilation occurs as usual.

Description

This option tells the compiler to display the include file order and continue compilation.

IDE Equivalent

None

Alternate Options

None

I

Specifies an additional directory to search for include files.

Syntax

Linux OS:

`-Idir`

macOS:

`-Idir`

Windows OS:

`/Idir`

Arguments

dir	Is the additional directory for the search.

Default

OFF	The default directory is searched for include files.

Description

This option specifies an additional directory to search for include files. To specify multiple directories on the command line, repeat the include option for each directory.

IDE Equivalent

Windows

Visual Studio: **General > Additional Include Directories**

Linux

Eclipse: **Preprocessor > Additional Include Directories**

OS X

Xcode: **Preprocessor > Additional Include Directories**

Alternate Options

None

I-

Splits the include path.

Syntax

Linux OS:

`-I-`

macOS:

`-I-`

Windows OS:

`/I-`

Arguments

None

Default

OFF	The default directory is searched for include files.

Description

This option splits the include path. It prevents the use of the current directory as the first search directory for '#include "file"'.

If you specify directories using the `I` option *before* you specify option `I-`, the directories are searched only for the case of '#include "file"'; they are not searched for '#include <file>'.

If you specify directories using the `I` option *after* you specify option `I-`, these directories are searched for all '#include' directives.

This option has no effect on option `nostdinc++`, which searches the standard system directories for header files.

This option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

See Also

`I` compiler option
`nostdinc++` compiler option

icc, Qicl

Determines whether certain Intel®-specific compiler macros are defined or undefined.

Syntax

Linux OS:

`-icc`

`-no-icc`

macOS:

`-icc`

`-no-icc`

Windows OS:

`/Qicl`

`/Qicl-`

Arguments

None

Default

`-icc` or `/Qicl`	The __INTEL_COMPILER macros are set to represent the current version of the compiler.

Description

This option determines whether certain Intel®-specific compiler macros are defined or undefined.

If you specify option `-no-icc` or `/Qicl-` , the compiler undefines the __INTEL_COMPILER macros. These macros are defined by default or by specifying `-icc` or `/Qicl`.

IDE Equivalent

None

Alternate Options

None

idirafter

Adds a directory to the second include file search path.

Syntax

Linux OS:

`-idirafter`*`dir`*

macOS:

`-idirafter`*`dir`*

Windows OS:

`None`

Arguments

dir	Is the name of the directory to add.

Default

OFF	Include file search paths include certain default directories.

Description

This option adds a directory to the second include file search path (after `-I`).

IDE Equivalent

None

Alternate Options

None

imacros

Allows a header to be specified that is included in front of the other headers in the translation unit.

Syntax

Linux OS:

`-imacros filename`

macOS:

`-imacros filename`

Windows OS:

`None`

Arguments

filename	Name of header file.

Default

OFF

Description

Allows a header to be specified that is included in front of the other headers in the translation unit.

IDE Equivalent

None

Alternate Options

None

iprefix

Lets you indicate the prefix for referencing directories that contain header files.

Syntax

Linux OS:

`-iprefix prefix`

macOS:

`-iprefix` *`prefix`*

Windows OS:

`None`

Arguments

prefix	Is the prefix to use.

Default

OFF	No prefix is included.

Description

Options for indicating the prefix for referencing directories containing header files. Use *prefix* with option `-iwithprefix` as a prefix.

IDE Equivalent

None

Alternate Options

None

iquote

Adds a directory to the front of the include file search path for files included with quotes but not brackets.

Syntax

Linux OS:

`-iquote` *`dir`*

macOS:

`-iquote` *`dir`*

Windows OS:

`None`

Arguments

dir	Is the name of the directory to add.

Default

OFF	The compiler does not add a directory to the front of the include file search path.

Description

Add directory to the front of the include file search path for files included with quotes but not brackets.

IDE Equivalent

None

Alternate Options

None

isystem

Specifies a directory to add to the start of the system include path.

Syntax

Linux OS:

`-isystemdir`

macOS:

`-isystemdir`

Windows OS:

`None`

Arguments

dir	Is the directory to add to the system include path.

Default

OFF	The default system include path is used.

Description

This option specifies a directory to add to the system include path. The compiler searches the specified directory for include files after it searches all directories specified by the `-I` compiler option but before it searches the standard system directories.

On Linux* systems, this option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

iwithprefix

Appends a directory to the prefix passed in by -iprefix and puts it on the include search path at the end of the include directories.

Syntax

Linux OS:

`-iwithprefixdir`

macOS:

`-iwithprefixdir`

Windows OS:

`None`

Arguments

dir	Is the include directory.

Default

OFF

Description

This option appends a directory to the prefix passed in by `-iprefix` and puts it on the include search path at the end of the include directories.

IDE Equivalent

None

Alternate Options

None

iwithprefixbefore

Similar to -iwithprefix except the include directory is placed in the same place as -I command-line include directories.

Syntax

Linux OS:

`-iwithprefixbefore`*`dir`*

macOS:

`-iwithprefixbefore`*`dir`*

Windows OS:

`None`

Arguments

dir	Is the include directory.

Default

OFF

Description

Similar to `-iwithprefix` except the include directory is placed in the same place as `-I` command-line include directories.

IDE Equivalent

None

Alternate Options

None

Kc++, TP

Tells the compiler to process all source or unrecognized file types as C++ source files. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-Kc++`

macOS:

`-Kc++`

Windows OS:

`/TP`

Arguments

None

Default

OFF	The compiler uses default rules for determining whether a file is a C++ source file.

Description

This option tells the compiler to process all source or unrecognized file types as C++ source files.

This is a deprecated option that may be removed in a future release. The replacement option for `Kc++` is `-x c++`; the replacement option for `/TP` is `/Tp<file>`.

IDE Equivalent

Windows

Visual Studio: **Advanced > Compile As**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

Linux and macOS: `-x c++`

Windows: `/Tp`

M, QM

Tells the compiler to generate makefile dependency lines for each source file.

Syntax

Linux OS:

`-M`

macOS:

`-M`

Windows OS:

`/QM`

Arguments

None

Default

OFF	The compiler does not generate makefile dependency lines for each source file.

Description

This option tells the compiler to generate makefile dependency lines for each source file, based on the #include lines found in the source file.

IDE Equivalent

None

Alternate Options

None

MD, QMD

Preprocess and compile, generating output file containing dependency information ending with extension .d.

Syntax

Linux OS:

`-MD`

macOS:

`-MD`

Windows OS:

`/QMD`

Arguments

None

Default

OFF	The compiler does not generate dependency information.

Description

Preprocess and compile, generating output file containing dependency information ending with extension .d.

IDE Equivalent

None

Alternate Options

None

MF, QMF

Tells the compiler to generate makefile dependency information in a file.

Syntax

Linux OS:

`-MF`*`filename`*

macOS:

`-MF`*`filename`*

Windows OS:

`/QMF`*`filename`*

Arguments

filename	Is the name of the file where the makefile dependency information should be placed.

Default

OFF	The compiler does not generate makefile dependency information in files.

Description

This option tells the compiler to generate makefile dependency information in a file. To use this option, you must also specify `/QM` or `/QMM`.

IDE Equivalent

None

Alternate Options

None

See Also

`QM` compiler option
`QMM` compiler option

MG, QMG

Tells the compiler to generate makefile dependency lines for each source file.

Syntax

Linux OS:

`-MG`

macOS:

`-MG`

Windows OS:

`/QMG`

Arguments

None

Default

OFF	The compiler does not generate makefile dependency information in files.

Description

This option tells the compiler to generate makefile dependency lines for each source file. It is similar to `/QM`, but it treats missing header files as generated files.

IDE Equivalent

None

Alternate Options

None

See Also

`QM` compiler option

MM, QMM

Tells the compiler to generate makefile dependency lines for each source file.

Syntax

Linux OS:

`-MM`

macOS:

`-MM`

Windows OS:

`/QMM`

Arguments

None

Default

OFF	The compiler does not generate makefile dependency information in files.

Description

This option tells the compiler to generate makefile dependency lines for each source file. It is similar to `/QM`, but it does not include system header files.

IDE Equivalent

None

Alternate Options

None

See Also

`QM` compiler option

MMD, QMMD

Tells the compiler to generate an output file containing dependency information.

Syntax

Linux OS:

`-MMD`

macOS:

`-MMD`

Windows OS:

`/QMMD`

Arguments

None

Default

OFF	The compiler does not generate an output file containing dependency information.

Description

This option tells the compiler to preprocess and compile a file, then generate an output file (with extension `.d`) containing dependency information.

It is similar to `/QMD`, but it does not include system header files.

IDE Equivalent

None

Alternate Options

None

MP

Tells the compiler to add a phony target for each dependency.

Syntax

Linux OS:

`-MP`

macOS:

`-MP`

Windows OS:

`None (see below)`

Arguments

None

Default

OFF	The compiler does not generate dependency information unless it is told to do so.

Description

This option tells the compiler to add a phony target for each dependency.

Note that this option is not related to Windows* option /MP.

IDE Equivalent

None

Alternate Options

None

MQ

Changes the default target rule for dependency generation.

Syntax

Linux OS:

`-MQ`*`target`*

macOS:

`-MQ`*`target`*

Windows OS:

`None`

Arguments

target	Is the target rule to use.

Default

OFF	The default target rule applies to dependency generation.

Description

This option changes the default target rule for dependency generation. It is similar to `-MT`, but quotes special Make characters.

IDE Equivalent

None

Alternate Options

None

MT, QMT

Changes the default target rule for dependency generation.

Syntax

Linux OS:

`-MT`*`target`*

macOS:

`-MT`*`target`*

Windows OS:

`/QMT`*`target`*

Arguments

target	Is the target rule to use.

Default

OFF	The default target rule applies to dependency generation.

Description

This option changes the default target rule for dependency generation.

IDE Equivalent

None

Alternate Options

None

nostdinc++

Do not search for header files in the standard directories for C++, but search the other standard directories.

Syntax

Linux OS:

`-nostdinc++`

macOS:

`-nostdinc++`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Do not search for header files in the standard directories for C++, but search the other standard directories.

IDE Equivalent

None

Alternate Options

None

P

Tells the compiler to stop the compilation process and write the results to a file.

Syntax

Linux OS:

`-P`

macOS:

`-P`

Windows OS:

`/P`

Arguments

None

Default

OFF	Normal compilation is performed.

Description

This option tells the compiler to stop the compilation process after C or C++ source files have been preprocessed and write the results to files named according to the compiler's default file-naming conventions.

On Linux systems, this option causes the preprocessor to expand your source module and direct the output to a `.i` file instead of `stdout`. Unlike the `-E` option, the output from `-P` on Linux does not include #line number directives. By default, the preprocessor creates the name of the output file using the prefix of the source file name with a `.i` extension. You can change this by using the `-o` option.

IDE Equivalent

Visual Studio: **Preprocessor > Generate Preprocessed File**

Eclipse: None

Xcode: None

Alternate Options

Linux and macOS: `-F`

Windows: None

pragma-optimization-level

Specifies which interpretation of the optimization_level pragma should be used if no prefix is specified.

Syntax

Linux OS:

`-pragma-optimization-level=`*`interpretation`*

macOS:

`-pragma-optimization-level=`*`interpretation`*

Windows OS:

None

Arguments

interpretation	Compiler-specific interpretation of `optimization_level` pragma. Possible values are:	
	`Intel`	Specify the Intel interpretation.
	`GCC`	Specify the GCC interpretation.

Default

`-pragma-optimization-level=Intel`	Use the Intel interpretation of the `optimization_level` pragma.

Description

Specifies which interpretation of the `optimization_level` pragma should be used if no prefix is specified.

IDE Equivalent

None

Alternate Options

None

u (Windows*)

Disables all predefined macros and assertions.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/u`

Arguments

None

Default

OFF	Defined preprocessor values are in effect until they are undefined.

Description

This option disables all predefined macros and assertions.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Undefine All Preprocessor Definitions**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

U

Undefines any definition currently in effect for the specified macro.

Syntax

Linux OS:

`-U`*`name`*

macOS:

`-U`*`name`*

Windows OS:

`/Uname`

Arguments

name	Is the name of the macro to be undefined.

Default

OFF	Macro definitions are in effect until they are undefined.

Description

This option undefines any definition currently in effect for the specified macro. It is equivalent to an #undef preprocessing directive.

On Windows systems, use the `/u` option to undefine all previously defined preprocessor values.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Undefine Preprocessor Definitions**

Linux

Eclipse: **Preprocessor > Undefine Preprocessor Definitions**

OS X

Xcode: **Preprocessor > Undefine Preprocessor Definitions**

Alternate Options

None

Example

To undefine a macro, enter the following command:

On Windows* systems:

```
icl /Uia64 prog1.cpp
```

On Linux* systems:

```
icpc -Uia64 prog1.cpp
```

On macOS systems:

```
icpc -Uia64 prog1.cpp
```

If you attempt to undefine an ANSI C macro, the compiler will emit an error:

```
invalid macro undefinition: <name of macro>
```

See Also

undef

Disables all predefined macros.

Syntax

Linux OS:

`-undef`

macOS:

`-undef`

Windows OS:

`None`

Arguments

None

Default

OFF	Defined macros are in effect until they are undefined.

Description

This option disables all predefined macros.

IDE Equivalent

None

Alternate Options

None

X

Removes standard directories from the include file search path.

Syntax

Linux OS:

`-X`

macOS:

`-X`

Windows OS:

`/X`

Arguments

None

Default

OFF	Standard directories are in the include file search path.

Description

This option removes standard directories from the include file search path. It prevents the compiler from searching the default path specified by the INCLUDE environment variable.

On Linux* and macOS systems, specifying `-X` (or `-noinclude`) prevents the compiler from searching in `/usr/include` for files specified in an INCLUDE statement.

You can use this option with the `I` option to prevent the compiler from searching the default path for include files and direct it to use an alternate path.

IDE Equivalent

Windows

Visual Studio: **Preprocessor > Ignore Standard Include Path**

Linux

Eclipse: **Preprocessor > Ignore Standard Include Path**

OS X

Xcode: **Preprocessor > Ignore Standard Include Path**

Alternate Options

Linux and macOS: `-nostdinc`

Windows: None

See Also

`I` compiler option

Component Control Options

This section contains descriptions for compiler options that pertain to component control.

Qinstall

Specifies the root directory where the compiler installation was performed.

Syntax

Linux OS:

`-Qinstall`*`dir`*

macOS:

`-Qinstall`*`dir`*

Windows OS:

`None`

Arguments

dir	Is the root directory where the installation was performed.

Default

OFF	The default root directory for compiler installation is searched for the compiler.

Description

This option specifies the root directory where the compiler installation was performed. It is useful if you want to use a different compiler or if you did not use a shell script to set your environment variables.

IDE Equivalent

None

Alternate Options

None

Qlocation

Specifies the directory for supporting tools.

Syntax

Linux OS:

`-Qlocation,`*`string`*`,`*`dir`*

macOS:

`-Qlocation,`*`string`*`,`*`dir`*

Windows OS:

`/Qlocation,`*`string`*`,`*`dir`*

Arguments

string	Is the name of the tool.
dir	Is the directory (path) where the tool is located.

Default

OFF	The compiler looks for tools in a default area.

Description

This option specifies the directory for supporting tools.

string can be any of the following:

- c - Indicates the Intel® C++ Compiler Classic.
- cpp (or fpp) - Indicates the Intel® C++ preprocessor.
- cxxinc - Indicates C++ header files.
- cinc - Indicates C header files.
- asm - Indicates the assembler.
- link - Indicates the linker.
- prof - Indicates the profiler.
- On Windows* systems, the following is also available:
  - masm - Indicates the Microsoft assembler.
- On Linux* and macOS systems, the following are also available:
  - as - Indicates the assembler.
  - gas - Indicates the GNU assembler. This setting is for Linux* only.
  - ld - Indicates the loader.

- gld - Indicates the GNU loader. This setting is for Linux* only.
- lib - Indicates an additional library.
- crt - Indicates the crt%.o files linked into executables to contain the place to start execution.

On Windows and macOS systems, you can also specify a tool command name.

The following shows an example on macOS systems:

```
-Qlocation,ld,/usr/bin              ! This tells the driver to use /usr/bin/ld for the loader
-Qlocation,ld,/usr/bin/gld          ! This tells the driver to use /usr/bin/gld as the loader
```

The following shows an example on Windows* systems:

```
/Qlocation,link,"c:\Program Files\tools\"            ! This tells the driver to use c:\Program
Files\tools\link.exe for the loader
/Qlocation,link,"c:\Program Files\tools\my_link.exe" ! This tells the driver to use c:\Program
Files\tools\my_link.exe as the loader
```

IDE Equivalent

None

Alternate Options

None

See Also

`Qoption` compiler option

Qoption

Passes options to a specified tool.

Syntax

Linux OS:

`-Qoption,`*`string`*`,`*`options`*

macOS:

`-Qoption,`*`string`*`,`*`options`*

Windows OS:

`/Qoption,`*`string`*`,`*`options`*

Arguments

string	Is the name of the tool.
options	Are one or more comma-separated, valid options for the designated tool. Note that certain tools may require that options appear within quotation marks (" ").

Default

OFF	No options are passed to tools.

Description

This option passes options to a specified tool.

If an argument contains a space or tab character, you must enclose the entire argument in quotation marks (" "). You must separate multiple arguments with commas.

string can be any of the following:

- cpp - Indicates the preprocessor for the compiler.
- c - Indicates the Intel® C++ Compiler Classic.
- asm - Indicates the assembler.
- link - Indicates the linker.
- prof - Indicates the profiler.
- On Windows* systems, the following is also available:
  - masm - Indicates the Microsoft assembler.
- On Linux* and macOS systems, the following are also available:
  - as - Indicates the assembler.
  - gas - Indicates the GNU assembler.
  - ld - Indicates the loader.
  - gld - Indicates the GNU loader.
  - lib - Indicates an additional library.
  - crt - Indicates the crt%.o files linked into executables to contain the place to start execution.

IDE Equivalent

None

Alternate Options

None

See Also

`Qlocation` compiler option

Language Options

This section contains descriptions for compiler options that pertain to language compatibility, conformity, etc.

ansi

Enables language compatibility with the gcc option ansi.

Syntax

Linux OS:

`-ansi`

macOS:

`-ansi`

Windows OS:

None

Arguments

None

Default

OFF	GNU C++ is more strongly supported than ANSI C.

Description

This option enables language compatibility with the gcc option `-ansi` and provides the same level of ANSI standard conformance as that option.

This option sets option `fmath-errno`.

If you want strict ANSI conformance, use the -strict-ansi option.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Language > ANSI Conformance**

OS X

Xcode: **Language > C Language Dialect and Language > C++ Language Dialect**

Alternate Options

None

check

Checks for certain conditions at run time.

Syntax

Linux OS and macOS:

`-check=`*`keyword`*`[, `*`keyword`*`...]`

Windows OS:

`/check:`*`keyword`*`[, `*`keyword`*`...]`

Arguments

keyword

Specifies the conditions to check. Possible values are:

`[no]conversions`	Determines whether checking occurs for converting to smaller types. Keyword `conversions` enables this checking.
`[no]stack`	Determines whether checking occurs on the stack frame. Keyword `stack` enables this checking. If `stack` is specified, the stack is checked for buffer overruns and buffer underruns. This option also enforces local variables initialization and stack pointer verification.

`[no]uninit`	Determines whether checking occurs for uninitialized variables. Keyword `uninit` enables this checking. If a variable is read before it is written, a run-time error routine will be called. Run-time checking of undefined variables is only implemented on local, scalar variables. It is not implemented on dynamically allocated variables, extern variables or static variables. It is not implemented on structs, classes, unions or arrays.

Default

`noconversions`
`nostack`
`nouninit`
	No checking is performed for the above run-time conditions.

Description

This option checks for certain conditions at run time.

On Windows* systems, this option disables any default or specified optimizations and applies the `/Od` level of optimization. If you specified optimizations, the compiler emits warning diagnostics for the disabled optimizations.

On Linux* and macOS systems, this option may disable some optimizations.

> **NOTE**
> This option requires library support. Depending on the platform, the required library is either in your operating system run-time environment or in your compiler package.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Runtime > Check Stack Frame** (-check=stack)

Runtime > Check Type Conversions (-check=conversions)

Runtime > Check Uninitialized Variables (-check=uninit)

Xcode

Xcode: **Runtime > Check Stack Frame** (-check=stack)

Runtime > Check Type Conversions (-check=conversions)

Runtime > Check Uninitialized Variables (-check=uninit)

Alternate Options

`check:conversions` Linux and macOS: None

	Windows: `/RTCc`
`check:stack`	Linux and macOS: None Windows: `/RTCs`
`check:uninit`	Linux and macOS: None Windows: `/RTCu`

early-template-check

Lets you semantically check template function template prototypes before instantiation.

Syntax

Linux OS and macOS:

`-early-template-check`

`-no-early-template-check`

Windows OS:

`None`

Arguments

None

Default

`-no-early-template-check`	The prototype instantiation of function templates and function members of class templates is deferred.

Description

Lets you semantically check template function template prototypes before instantiation. On Linux* platforms, gcc 3.4 (or newer) compatibilty modes must be in effect. For all macOS platforms, gcc 4.0 (or newer) is required.

IDE Equivalent

None

Alternate Options

None

fblocks

Determines whether Apple blocks are enabled or disabled.*

Syntax

Linux OS:

`None`

macOS:

`-fblocks`

`-fno-blocks`

Windows OS:

`None`

Arguments

None

Default

`-fblocks`	Apple* blocks are enabled.

Description

This option determines whether Apple* blocks (block variable declarations) are enabled or disabled.

If you want to disable Apple* blocks, specify -fno-blocks.

To use this feature, macOS 10.6 or greater is required.

IDE Equivalent

None

Alternate Options

None

ffriend-injection

Causes the compiler to inject friend functions into the enclosing namespace.

Syntax

Linux OS and macOS:

`-ffriend-injection`

`-fno-friend-injection`

Windows OS:

`None`

Arguments

None

Default

`-fno-friend-injection`	The compiler does not inject friend functions into the enclosing namespace. A friend function that is not declared in an enclosing scope can only be found using argument-dependent lookup.

Description

This option causes the compiler to inject friend functions into the enclosing namespace, so they are visible outside the scope of the class in which they are declared.

On Linux systems, in gcc versions 4.1 or later, this is not the default behavior. This option allows compatibility with gcc 4.0 or earlier.

IDE Equivalent

None

Alternate Options

None

fno-gnu-keywords

Tells the compiler to not recognize typeof as a keyword.

Syntax

Linux OS:

`-fno-gnu-keywords`

macOS:

`-fno-gnu-keywords`

Windows OS:

`None`

Arguments

None

Default

OFF	Keyword `typeof` is recognized.

Description

Tells the compiler to not recognize `typeof` as a keyword.

IDE Equivalent

None

Alternate Options

None

fno-implicit-inline-templates

Tells the compiler to not emit code for implicit instantiations of inline templates.

Syntax

Linux OS:

`-fno-implicit-inline-templates`

macOS:

`-fno-implicit-inline-templates`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler handles inlines so that compilations, with and without optimization, will need the same set of explicit instantiations.

Description

This option tells the compiler to not emit code for implicit instantiations of inline templates.

IDE Equivalent

None

Alternate Options

None

fno-implicit-templates

Tells the compiler to not emit code for non-inline templates that are instantiated implicitly.

Syntax

Linux OS:

`-fno-implicit-templates`

macOS:

`-fno-implicit-templates`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler handles inlines so that compilations, with and without optimization, will need the same set of explicit instantiations.

Description

This option tells the compiler to not emit code for non-inline templates that are instantiated implicitly, but to only emit code for explicit instantiations.

IDE Equivalent

None

Alternate Options

None

fno-operator-names

Disables support for the operator names specified in the standard.

Syntax

Linux OS:

`-fno-operator-names`

macOS:

`-fno-operator-names`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Disables support for the operator names specified in the standard.

IDE Equivalent

None

Alternate Options

None

fno-rtti

Disables support for run-time type information (RTTI).

Syntax

Linux OS:

`-fno-rtti`

macOS:

`-fno-rtti`

Windows OS:

`None`

Arguments

None

Default

OFF	Support for run-time type information (RTTI) is enabled.

Description

This option disables support for run-time type information (RTTI).

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: None

OS X

Xcode: **Language > Enable C++ Runtime Types**

Alternate Options

None

fnon-lvalue-assign

Determines whether casts and conditional expressions can be used as lvalues.

Syntax

Linux OS and macOS:

`-fnon-lvalue-assign`

`-fno-non-lvalue-assign`

Windows OS:

`None`

Arguments

None

Default

`-fnon-lvalue-assign`	The compiler allows casts and conditional expressions to be used as `lvalues`.

Description

This option determines whether casts and conditional expressions can be used as `lvalues`.

IDE Equivalent

None

Alternate Options

None

fpermissive

Tells the compiler to allow for non-conformant code.

Syntax

Linux OS:

`-fpermissive`

macOS:

`-fpermissive`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

Tells the compiler to allow for non-conformant code.

IDE Equivalent

None

Alternate Options

None

fshort-enums

Tells the compiler to allocate as many bytes as needed for enumerated types.

Syntax

Linux OS:

`-fshort-enums`

macOS:

`-fshort-enums`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler allocates a default number of bytes for enumerated types.

Description

This option tells the compiler to allocate as many bytes as needed for enumerated types.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Associate as Many Bytes as Needed for Enumerated Types**

OS X

Xcode: **Data > Allocate enumerated types**

Alternate Options

None

fsyntax-only

Tells the compiler to check only for correct syntax.

Syntax

Linux OS:

`-fsyntax-only`

macOS:

`-fsyntax-only`

Windows OS:

`None`

Arguments

None

Default

OFF	Normal compilation is performed.

Description

This option tells the compiler to check only for correct syntax. No object file is produced.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/Zs`

ftemplate-depth, Qtemplate-depth

Control the depth in which recursive templates are expanded.

Syntax

Linux OS:

`-ftemplate-depth=n`

macOS:

`-ftemplate-depth=n`

Windows OS:

`/Qtemplate-depth:n`

Arguments

n	The number of recursive templates that are expanded.

Default

OFF	The compiler uses default heuristics for the depth of expansion.

Description

Control the depth in which recursive templates are expanded. On Linux*, this option is supported only by invoking the compiler with `icpc`.

IDE Equivalent

None

Alternate Options

None

funsigned-bitfields

Determines whether the default bitfield type is changed to unsigned.

Syntax

Linux OS:

`-funsigned-bitfields`

`-fno-unsigned-bitfields`

macOS:

`-funsigned-bitfields`

`-fno-unsigned-bitfields`

Windows OS:

`None`

Arguments

None

Default

`-fno-unsigned-bitfields`	The default bitfield type is signed.

Description

This option determines whether the default bitfield type is changed to `unsigned`.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Change Default Bitfield Type to unsigned**

OS X

Xcode: **Data > Unsigned bitfield Type**

Alternate Options

None

funsigned-char

Change default char type to unsigned.

Syntax

Linux OS:

`-funsigned-char`

macOS:

`-funsigned-char`

Windows OS:

`None`

Arguments

None

Default

OFF	Do not change default char type to unsigned.

Description

Change default char type to unsigned.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Change default char type to unsigned**

OS X

Xcode: **Data > Unsigned char Type**

Alternate Options

None

GZ

Initializes all local variables. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/GZ`

Arguments

None

Default

OFF	The compiler does not initialize local variables.

Description

This option initializes all local variables to a non-zero value. To use this option, you must also specify option `/Od`.

This is a deprecated option that may be removed in a future release. The replacement option is `/RTC1`.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/RTC1`

H (Windows*)

Causes the compiler to limit the length of external symbol names. This is a deprecated option. There is no replacement option.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Hn`

Arguments

n	Is the maximum number of characters for external symbol names.

Default

OFF	The compiler follows default rules for the length of external symbol names.

Description

This option causes the compiler to limit the length of external symbol names to a maximum of *n* characters.

IDE Equivalent

None

Alternate Options

None

help-pragma, Qhelp-pragma

Displays all supported pragmas.

Syntax

Linux OS:

`-help-pragma`

macOS:

`-help-pragma`

Windows OS:

`/Qhelp-pragma`

Arguments

None

Default

OFF	No list is displayed unless this compiler option is specified.

Description

This option displays all supported pragmas and shows their syntaxes.

IDE Equivalent

None

Alternate Options

None

intel-extensions, Qintel-extensions

Enables or disables all Intel® C and Intel® C++ language extensions.

Syntax

Linux OS and macOS:

`-intel-extensions`

`-no-intel-extensions`

Windows OS:

`/Qintel-extensions`

`/Qintel-extensions-`

Arguments

None

Default

OFF	The Intel® C and Intel® C++ language extensions are enabled.

Description

This option enables or disables all Intel® C and Intel® C++ language extensions.

If you specify the negative form of the option (see above), it disables all Intel® C and Intel® C++ language extensions.

Note that certain settings for the `[Q]std` compiler option can enable or disable decimal floating-point support:

- The following `[Q]std` settings enable decimal floating-point support: c89, gnu89 (Linux* only), gnu99 (Linux* only)
- The following `[Q]std` setting disables decimal floating-point support: c99

IDE Equivalent

Visual Studio

Visual Studio: **Language > Disable All Intel Language Extensions**

Eclipse

Eclipse: **Language > Disable All Intel Language Extensions**

Xcode

Xcode: **Language > Enable Intel C/C++ language extensions**

Alternate Options

None

See Also

`std, Qstd` compiler option

J

Sets the default character type to unsigned.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/J`

Arguments

None

Default

OFF	The default character type is `signed`

Description

This option sets the default character type to `unsigned`. This option has no effect on character values that are explicitly declared `signed`. This option sets `_CHAR_UNSIGNED` = 1.

IDE Equivalent

Windows

Visual Studio: **Language > Default Char Unsigned**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

restrict, Qrestrict

Determines whether pointer disambiguation is enabled with the restrict qualifier.

Syntax

Linux OS:

`-restrict`

`-no-restrict`

macOS:

`-restrict`

`-no-restrict`

Windows OS:

`/Qrestrict`

`/Qrestrict-`

Arguments

None

Default

`-no-restrict` or `/Qrestrict-`	Pointers are not qualified with the restrict keyword.

Description

This option determines whether pointer disambiguation is enabled with the restrict qualifier. Option `-restrict` and `/Qrestrict` enable the recognition of the restrict keyword as defined by the ANSI standard.

By qualifying a pointer with the restrict keyword, you assert that an object accessed by the pointer is only accessed by that pointer in the given scope. You should use the restrict keyword only when this is true. When the assertion is true, the restrict option will have no effect on program correctness, but may allow better optimization.

IDE Equivalent

Windows

Visual Studio: **Language > Recognize The Restrict Keyword**

Linux

Eclipse: **Language > Recognize The Restrict Keyword**

OS X

Xcode: **Language > Recognize RESTRICT keyword**

Alternate Options

None

See Also

`std, Qstd` compiler option

std, Qstd

Tells the compiler to conform to a specific language standard.

Syntax

Linux OS:

`-std=`*`val`*

macOS:

`-std=`*`val`*

Windows OS:

`/Qstd:`*`val`*

`/std:`*`val`* `(For Microsoft* compatibility)`

Arguments

val Specifies the specific language standard to conform to.

The following values apply to Linux* `-std` and Windows* `/Qstd`:

`c++2b`	Enables support for the Working Draft for ISO C++ 2023 DIS standard.
`c++20`	Enables support for the 2020 ISO C++ DIS standard.
`c++17`	Enables support for the 2017 ISO C++ standard with amendments.
`c++14`	Enables support for the 2014 ISO C++ standard with amendments.
`c++11`	Enables support for the 2011 ISO C++ standard with amendments.
`c++98` and `c++03`	Enables support for the 1998 ISO C++ standard with amendments.
`c2x`	Enables support for the Working Draft for ISO C2x standard.
`c18` and `c17`	Enables support for the 2017 ISO C standard. Support for c17 can also be enabled by value iso9899:2017. Support for c18 can also be enabled by value iso9899:2018.
`c11`	Enables support for the 2011 ISO C standard. Support for this standard can also be enabled by value iso9899:2011.
`c99`	Enables support for the 1999 ISO C standard. Support for this standard can also be enabled by value iso9899:1999.

`c90` and `c89`	Enables support for the 1990 ISO C standard. Support for this standard can also be enabled by value iso9899:1990.

The following values apply only to Linux `-std`:

`gnu++2b`	Enables support for the Working Draft for ISO C++ 2023 DIS standard plus GNU extensions.
`gnu++20`	Enables support for the 2020 ISO C++ DIS standard plus GNU extensions.
`gnu++17`	Enables support for the 2017 ISO C++ standard with amendments plus GNU extensions.
`gnu++14`	Enables support for the 2014 ISO C++ standard with amendments plus GNU extensions.
`gnu++11`	Enables support for the 2011 ISO C++ standard with amendments plus GNU extensions.
`gnu++98` and `gnu++03`	Enables support for the 1998 ISO C++ standard with amendments plus GNU extensions.
`gnu2x`	Enables support for the Working Draft for ISO C2x standard plus GNU extensions.
`gnu18` and `gnu17`	Enables support for the 2017 ISO C standard plus GNU extensions.
`gnu11`	Enables support for the 2011 ISO C standard plus GNU extensions.
`gnu99`	Enables support for the 1999 ISO C standard plus GNU extensions.
`gnu90` and `gnu89`	Enables support for the 1990 ISO C standard plus GNU extensions.

For possible values for Microsoft*-compatible Windows* /std, see the Microsoft* documentation.

Default

Default for Windows option `/Qstd`: OFF	The compiler does not conform to a specific language standard.
Default for Windows option `/std`: `c++14`	Currently, the compiler conforms to the 2014 ISO C++ standard. For the latest information, see the Microsoft* documentation.
Default for Linux option `-std` on icc (Classic C compiler): `c99` or `c11`	If using GCC5.x or higher, the compiler conforms to c99, which is the 1999 ISO C standard. If using GCC5.0 or lower, the compiler conforms to c11, which is the 2011 ISO C standard.
Default for Linux option `-std` on icpc (Classic C++ compiler): varies	If using GCC5.x or lower, the compiler conforms to c++98, which is the 1998 ISO C++ standard. If using GCC6.0 to GCC11.0, the compiler conforms to c++14, which is the 2014 ISO C++ standard. If using GCC11.1 or higher, the compiler conforms to c++17, which is the 2017 ISO C++ standard.

Description

This option tells the compiler to conform to a specific language standard.

IDE Equivalent

Visual Studio

Visual Studio: **Language > C/C++ Language Support**

Eclipse

Eclipse: **Language > ANSI Conformance**

Xcode

Xcode: **Language > C Language Dialect and C++ Language Dialect**

Alternate Options

None

strict-ansi

Tells the compiler to implement strict ANSI conformance dialect.

Syntax

Linux OS:

`-strict-ansi`

macOS:

`-strict-ansi`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler conforms to default standards.

Description

This option tells the compiler to implement strict ANSI conformance dialect. On Linux* systems, if you need to be compatible with gcc, use the `-ansi` option.

This option sets option `fmath-errno`, which tells the compiler to assume that the program tests errno after calls to math library functions. This restricts optimization because it causes the compiler to treat most math functions as having side effects.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Language > ANSI Conformance**

OS X

Xcode: **Language > C Language Dialect and Language > C++ Language Dialect**

Alternate Options

None

vd

Enables or suppresses hidden vtordisp members in C++ objects.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/vdn`

Arguments

n	Possible values are:	
	`0`	Suppresses the creation of the hidden `vtordisp` members in C++ objects.
	`1`	Enables the creation of hidden `vtordisp` members in C++ objects when they are necessary.
	`2`	Enables the hidden `vtordisp` members for all virtual base classes with virtual functions. This setting is recommended in the following cases: • When the only virtual function in your virtual base class is a destructor • When you want to ensure correct performance of the dynamic_cast operator on a partially-constructed object

Default

`/vd1`	The compiler enables the creation of hidden `vtordisp` members in C++ objects when they are necessary.

Description

This option enables or suppresses hidden `vtordisp` members in C++ objects.

This is a compatibility option for the Microsoft Visual C++* option `/vdn`. For full details about this compiler option, see the Microsoft* documentation.

IDE Equivalent

None

Alternate Options

None

vmb

Selects the smallest representation that the compiler uses for pointers to members.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/vmb`

Arguments

None

Default

OFF	The compiler uses default rules to represent pointers to members.

Description

This option selects the smallest representation that the compiler uses for pointers to members. Use this option if you define each class before you declare a pointer to a member of the class.

IDE Equivalent

None

Alternate Options

None

vmg

Selects the general representation that the compiler uses for pointers to members.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/vmg`

Arguments

None

Default

OFF	The compiler uses default rules to represent pointers to members.

Description

This option selects the general representation that the compiler uses for pointers to members. Use this option if you declare a pointer to a member before you define the corresponding class.

IDE Equivalent

None

Alternate Options

None

vmm

Enables pointers to class members with single or multiple inheritance.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/vmm`

Arguments

None

Default

OFF	The compiler uses default rules to represent pointers to members.

Description

This option enables pointers to class members with single or multiple inheritance. To use this option, you must also specify option `/vmg`.

IDE Equivalent

None

Alternate Options

None

vms

Enables pointers to members of single-inheritance classes.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/vms`

Arguments

None

Default

OFF	The compiler uses default rules to represent pointers to members.

Description

This option enables pointers to members of single-inheritance classes. To use this option, you must also specify option `/vmg`.

IDE Equivalent

None

Alternate Options

None

x (type option)

All source files found subsequent to -x type will be recognized as a particular type.

Syntax

Linux OS:

`-x` *`type`*

macOS:

`-x` *`type`*

Windows OS:

`None`

Arguments

type	is the type of source file. Possible values are:

`c++`	C++ source file
`c++-header`	C++ header file
`c++-cpp-output`	C++ pre-processed file
`c`	C source file
`c-header`	C header file
`cpp-output`	C pre-processed file
`assembler`	Assembly file
`assembler-with-cpp`	Assembly file that needs to be preprocessed
`none`	Disable recognition, and revert to file extension

Default

`none`	Disable recognition and revert to file extension.

Description

All source files found subsequent to `-x`*type* will be recognized as a particular type.

IDE Equivalent

None

Alternate Options

None

Example

Suppose you want to compile the following C and C++ source files whose extensions are not recognized by the compiler:

File Name	Language
file1.c99	C
file2.cplusplus	C++

We will also include these files whose extensions are recognized:

File Name	Language
file3.c	C
file4.cpp	C++

The command-line invocation using the `-x` option follows:

```
icpc -x c file1.c99 -x c++ file2.cplusplus -x none file3.c file4.cpp
```

Za

Disables Microsoft Visual C++* compiler language extensions.*

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Za`

Arguments

None

Default

OFF	The compiler provides support for extended ANSI C.

Description

This option disables Microsoft* Visual C++* compiler language extensions.

IDE Equivalent

Windows

Visual Studio: **Language > Disable Language Extensions**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

`Ze` compiler option
`Zc` compiler option

Zc

Lets you specify ANSI C standard conformance for certain language features.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Zc:`*`arg1`*`[,`*`arg2`*`]`

Arguments

arg	Is the language feature for which you want standard conformance. The settings are compatible with Microsoft* settings for option /Zc. For a list of supported settings, see the table in the Description section of this topic.

Default

varies	See the table in the Description section of this topic.

Description

This option lets you specify ANSI C standard conformance for certain language features.

If you do not want the default behavior for one or more of the settings, you must specify the negative form of the setting. For example, if you do not want the `forScope` or `wchar_t` default behavior, you should specify `/Zc:forScope-,wchar_t-`.

The following table shows the supported Microsoft settings for option `/Zc`.

`/Zc` setting name	**Description**
auto[-]	Enforces compliance to the new standard meaning for auto (default). Disabled by /Zc:auto-.
forScope[-]	Enforces standard compliance in for-loop scope (default). Disabled by /Zc:forScope-.
inline[-]	Controls inline expansion. Disabled by /Zc:inline- (default).
rvalueCast[-]	Enforces Standard C++ explicit type conversion rules. Disabled by /Zc:rvalueCast- (default).
strictStrings[-]	Enforces const qualification for string literals. Disabled by /Zc:strictStrings- (default).
threadSafeInit[-]	Enables thread-safe initialization of local statics (default). Disabled by /Zc:threadSafeInit-.
throwingNew[-]	Enables link with the operator new implementation. Disabled by /Zc:throwingNew- (default).
trigraphs[-]	Enables trigraph character sequences. Disabled by /Zc:trigraphs- (default).
wchar_t[-]	Specifies that wchar_t is a native data type (default). Disabled by /Zc:wchar_t-.

IDE Equivalent

Windows

Visual Studio: **Language > Treat wchar_t as Built-in Type / Force Conformance In For Loop Scope**

Language > Enforce type conversion rules (rvalueCast)

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

Ze

Enables Microsoft Visual C++* compiler language extensions. This is a deprecated option that may be removed in a future release.*

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Ze`

Arguments

None

Default

ON	The compiler provides support for extended ANSI C.

Description

This option enables Microsoft* Visual C++* compiler language extensions.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

None

Alternate Options

None

See Also

`Zc` compiler option
`Za` compiler option

Zg

Tells the compiler to generate function prototypes. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Zg`

Arguments

None

Default

OFF	The compiler does not create function prototypes.

Description

This option tells the compiler to generate function prototypes.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

None

Alternate Options

None

Zp

Specifies alignment for structures on byte boundaries.

Syntax

Linux OS:

`-Zp[n]`

macOS:

`-Zp[n]`

Windows OS:

`/Zp[n]`

Arguments

n	Is the byte size boundary. Possible values are 1, 2, 4, 8, or 16.

Default

`Zp16`	Structures are aligned on either size boundary 16 or the boundary that will naturally align them.

Description

This option specifies alignment for structures on byte boundaries.

If you do not specify *n*, you get `Zp16`.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Struct Member Alignment**

Linux

Eclipse: **Data > Structure Member Alignment**

OS X

Xcode: **Data > Structure Member Alignment**

Alternate Options

None

Zs

Tells the compiler to check only for correct syntax.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Zs`

Arguments

None

Default

OFF	Normal compilation is performed.

Description

This option tells the compiler to check only for correct syntax.

IDE Equivalent

None

Alternate Options

Linux: `-syntax`, `-fsyntax-only`

Windows: None

Data Options

This section contains descriptions for compiler options that pertain to the treatment of data.

align

Determines whether variables and arrays are naturally aligned.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-align`

`-noalign`

macOS:

`-align`

`-noalign`

Windows OS:

`None`

Arguments

None

Default

`-noalign`	Variables and arrays are aligned according to the gcc model, which means they are aligned to 4-byte boundaries.

Description

This option determines whether variables and arrays are naturally aligned. Option `-align` forces the following natural alignment:

Type	Alignment
`double`	8 bytes
`long long`	8 bytes
`long double`	16 bytes

If you are not interacting with system libraries or other libraries that are compiled without `-align`, this option can improve performance by reducing misaligned accesses.

This option can also be specified as `-m[no-]align-double`. The options are equivalent.

Caution
If you are interacting with compatible libraries, this option can improve performance by reducing misaligned accesses. However, if you are interacting with noncompatible libraries or libraries that are compiled without option `-align`, your application may not perform as expected.

IDE Equivalent

None

Alternate Options

None

auto-ilp32, Qauto-ilp32

Instructs the compiler to analyze the program to determine if there are 64-bit pointers that can be safely shrunk into 32-bit pointers and if there are 64-bit longs (on Linux systems) that can be safely shrunk into 32-bit longs.*

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS and macOS:

`-auto-ilp32`

Windows OS:

`/Qauto-ilp32`

Arguments

None

Default

OFF	The optimization is not attempted.

Description

This option instructs the compiler to analyze the program to determine if there are 64-bit pointers that can be safely shrunk into 32-bit pointers and if there are 64-bit longs (on Linux* systems) that can be safely shrunk into 32-bit longs.

On macOS systems, you must also specify option `-no-pie` for the optimization to occur.

For this option to be effective, the compiler must be able to optimize using the `[Q]ipo` option and must be able to analyze all library calls or external calls the program makes. This option has no effect on Linux* systems unless you specify setting SSE3 or higher for option `-x`.

This option requires that the size of the program executable never exceeds 2^{32} bytes and all data values can be represented within 32 bits. If the program can run correctly in a 32-bit system, these requirements are implicitly satisfied. If the program violates these size restrictions, unpredictable behavior may occur.

IDE Equivalent

None

Alternate Options

None

See Also

auto-p32

compiler option
pie
compiler option
ipo, Qipo
compiler option
parallel, Qparallel
compiler option
x, Qx
compiler option

auto-p32

Instructs the compiler to analyze the program to determine if there are 64-bit pointers that can be safely shrunk to 32-bit pointers.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS and macOS:

`-auto-p32`

Windows OS:

None

Arguments

None

Default

OFF	The optimization is not performed.

Description

This option instructs the compiler to analyze and transform the program so that 64-bit pointers are shrunk to 32-bit pointers, wherever it is legal and safe to do so.

On macOS systems, you must also specify option `-no-pie` for the optimization to occur.

For this option to be effective, the compiler must be able to optimize using the `-ipo` option and it must be able to analyze all library calls or external calls the program makes. This option has no effect unless you specify setting SSE3 or higher for option `-x`.

The application cannot exceed a 32-bit address space; otherwise, unpredictable results can occur.

IDE Equivalent

None

Alternate Options

None

See Also

auto-ilp32, Qauto-ilp32
compiler option

pie
 compiler option
ipo, Qipo
 compiler option
x, Qx
 compiler option

check-pointers, Qcheck-pointers

Determines whether the compiler checks bounds for memory access through pointers.

Syntax

Linux OS:

`-check-pointers=`*keyword*

macOS:

None

Windows OS:

`/Qcheck-pointers:`*keyword*

Arguments

keyword	Specifies what type of bounds checking occurs. Possible values are:	
	`none`	Disables bounds checking. This is the default.
	`rw`	Checks bounds for reads and writes through pointers.
	`write`	Checks bounds for only writes through pointers.

Default

`-check-pointers=none` or `/Qcheck-pointers:none`	No bounds checking occurs for memory access through pointers.

Description

This option determines whether the compiler checks bounds for memory access through pointers. It enables checking of all indirect accesses through pointers, and all array accesses.

The compiler may optimize these checks away when it can determine that an access is safe.

When `rw` or `write` is specified, the `[Q]check-pointers-undimensioned` option is set and dimensioned and undimensioned arrays are checked.

If you do not want undimensioned arrays checked, you must specify option the negative form of the option (see Syntax above).

This pointer checker feature requires installation of another product. For more information, see Feature Requirements.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Check Pointers**

Linux

Eclipse: **Code Generation > Check Pointers**

OS X

Xcode: None

Alternate Options

None

See Also

check-pointers-undimensioned, Qcheck-pointers-undimensioned
compiler option

check-pointers-dangling, Qcheck-pointers-dangling
compiler option

check-pointers-dangling, Qcheck-pointers-dangling

Determines whether the compiler checks for dangling pointer references.

Syntax

Linux OS:

`-check-pointers-dangling=`*`keyword`*

macOS:

`None`

Windows OS:

`/Qcheck-pointers-dangling:`*`keyword`*

Arguments

keyword	Specifies what type of dangling pointer checking occurs. Possible values are:	
	`none`	Disables checking for dangling pointer references. This is the default.
	`heap`	Checks for dangling pointer references on the heap.
	`stack`	Checks for dangling pointer references on the stack.
	`all`	Checks for dangling pointer references on the heap and the stack.

Default

`-check-pointers-dangling=none` or `/Qcheck-pointers-dangling:none`	No checking occurs for dangling pointer references.

Description

This option determines whether the compiler checks for dangling pointer references.

To use this option, you must also specify the `[Q]check-pointers` option.

This pointer checker feature requires installation of another product. For more information, see Feature Requirements.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Check Dangling Pointers**

Linux

Eclipse: **Code Generation > Check Dangling Pointers**

OS X

Xcode: None

Alternate Options

None

See Also

check-pointers, Qcheck-pointers
compiler option

check-pointers-mpx, Qcheck-pointers-mpx

Determines whether the compiler checks bounds for memory access through pointers on processors that support Intel® Memory Protection Extensions (Intel® MPX).

Syntax

Linux OS:

`-check-pointers-mpx=`*`keyword`*

macOS:

`None`

Windows OS:

`/Qcheck-pointers-mpx:`*`keyword`*

Arguments

keyword	Specifies what type of bounds checking occurs. Possible values are:

`none`	Disables bounds checking. This is the default.
`rw`	Checks bounds for reads and writes through pointers.
`write`	Checks bounds for only writes through pointers.

Default

`-check-pointers-mpx=none` or `/Qcheck-pointers-mpx:none`	No bounds checking occurs for memory access through pointers on processors that support Intel® MPX.

Description

This option determines whether the compiler checks bounds for memory access through pointers on processors that support Intel® MPX. It enables checking of all indirect accesses through pointers, and all array accesses.

The compiler may optimize these checks away when it can determine that an access is safe.

If you specify option `[Q]check-pointers` along with option `[Q]check-pointers-mpx`, option `[Q]check-pointers-mpx` takes precedence.

If you specify `[Q]check-pointers-mpx`, you cannot specify option `[Q]check-pointers-dangling`.

NOTE
This feature requires supporting hardware, OS, and library support. Intel® MPX bounds exceptions are hardware exceptions that are handled by the OS and run-time library, similar to the way that a null pointer exception is handled. Pointer Checker detailed reports and report control functions are not enabled with Intel® MPX, because these require overriding the OS exception handling.

For more details, see the document titled: Intel® Memory Protection Extensions Enabling Guide.

This pointer checker feature requires installation of another product. For more information, see Feature Requirements.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Check Pointers**

Eclipse

Eclipse: **Code Generation > Check Pointers**

Xcode

Xcode: None

Alternate Options

None

See Also

check-pointers, Qcheck-pointers

compiler option

check-pointers-undimensioned, Qcheck-pointers-undimensioned
compiler option

check-pointers-narrowing, Qcheck-pointers-narrowing

Determines whether the compiler enables or disables the narrowing of pointers to structure fields.

Syntax

Linux OS:

`-check-pointers-narrowing`

`-no-check-pointers-narrowing`

macOS:

None

Windows OS:

`/Qcheck-pointers-narrowing`

`/Qcheck-pointers-narrowing-`

Arguments

None

Default

`-check-pointers-narrowing` or `/Qcheck-pointers-narrowing`	The compiler enables the narrowing of pointers to structure fields.

Description

This option determines whether the compiler enables or disables the narrowing of pointers to structure fields. Narrowing restricts a field pointer so that it can only legally point to that field.

To use this option, you must also specify the `[Q]check-pointers` option.

Disabling this feature can improve Pointer Checker compatibility with non-ANSI compliant code.

To disable the narrowing of pointers to structure fields, specify the negative form of the option (see Syntax above).

This pointer checker feature requires installation of another product. For more information, see Feature Requirements.

IDE Equivalent

None

Alternate Options

None

See Also

check-pointers, Qcheck-pointers
compiler option

check-pointers-undimensioned, Qcheck-pointers-undimensioned

Determines whether the compiler checks bounds for memory access through arrays that are declared without dimensions.

Syntax

Linux OS:

`-check-pointers-undimensioned`

`-no-check-pointers-undimensioned`

macOS:

`None`

Windows OS:

`/Qcheck-pointers-undimensioned`

`/Qcheck-pointers-undimensioned-`

Arguments

None

Default

`-check-pointers-undimensioned` or `/Qcheck-pointers-undimensioned`	Bounds checking occurs for memory access through arrays that are declared without dimensions. This checking occurs for both dimensioned and undimensioned arrays.

Description

This option determines whether the compiler checks bounds for memory access through arrays that are declared without dimensions.

To use this option, you must also specify the `[Q]check-pointers` option.

This pointer checker feature requires installation of another product. For more information, see Feature Requirements.

The default setting, `[Q]check-pointers-undimensioned`, can cause link time errors for multiple definitions for non-standard code and it can cause linker warnings for undefined symbols when linking library code that has not been compiled with pointer checking enabled. In both of these cases, the symbols will contain the string cp_array_end.

To prevent these issues, disable the checking of undimensioned arrays, by specifying the negative form of the option (see Syntax above).

Note that even if you specify the negative form of the option, dimensioned arrays are always checked.

IDE Equivalent

Windows

Visual Studio: **Code Generation > Turn off Checking for Undimensioned Arrays**

Linux

Eclipse: **Code Generation > Turn off Checking for Undimensioned Arrays**

OS X

Xcode: None

Alternate Options

None

See Also

check-pointers, Qcheck-pointers
compiler option

falign-functions, Qfnalign

Tells the compiler to align functions on an optimal byte boundary.

Syntax

Linux OS:

`-falign-functions[=n]`

`-fno-align-functions`

macOS:

`-falign-functions[=n]`

`-fno-align-functions`

Windows OS:

`/Qfnalign[:n]`

`/Qfnalign-`

Arguments

n	Is an optional positive integer initialization expression indicating the number of bytes for the minimum alignment boundary. It tells the compiler to align functions on a power-of-2 byte boundary. If you do not specify *n*, the compiler aligns the start of functions on 16-byte boundaries. The *n* must be a positive integer less than or equal to 4096. If you specify a value that is not a power of 2, *n* will be rounded up to the nearest power of 2. For example, if 23 is specified for *n*, functions will be aligned on 32 byte boundaries.

Default

`-fno-align-functions` or `/Qfnalign-`	The compiler aligns functions on 2-byte boundaries. This is the same as specifying `-falign-functions=2` (Linux* and macOS) or `/Qfnalign:2` (Windows*).

Description

This option tells the compiler to align functions on an optimal byte boundary. If you do not specify *n*, the compiler aligns the start of functions on 16-byte boundaries.

IDE Equivalent

None

Alternate Options

None

falign-loops, Qalign-loops

Aligns loops to a power-of-two byte boundary.

Syntax

Linux OS:

`-falign-loops[=n]`

`-fno-align-loops`

macOS:

`-falign-loops[=n]`

`-fno-align-loops`

Windows OS:

`/Qalign-loops[:n]`

`/Qalign-loops-`

Arguments

n	Is the optional number of bytes for the minimum alignment boundary. It must be a power of 2 between 1 and 4096, such as 1, 2, 4, 8, 16, 32, 64, 128, and so on. If you specify 1 for *n*, no alignment is performed; this is the same as specifying the negative form of the option. If you do not specify *n*, the default alignment is 16 bytes.

Default

`-fno-align-loops` or `/Qalign-loops-`	No special loop alignment is performed.

Description

This option aligns loops to a power-of-two boundary. This alignment may improve performance.

It can be affected by the pragma code_align and attribute code_align.

If code is compiled with the `-falign-loops=m` (Linux* and macOS) or `/Qalign-loops:m` (Windows*) option and a code_align:n pragma precedes a loop, the loop is aligned on a max (m, n) byte boundary. If a function is modified by a code_align:k pragma and a code_align:n pragma precedes a loop, then both the function and the loop are aligned on a max (k, n) byte boundary.

IDE Equivalent

None

Alternate Options

None

See Also

falign-functions, Qfnalign compiler option

falign-stack

Tells the compiler the stack alignment to use on entry to routines. This is a deprecated option that may be removed in a future release.

Architecture Restrictions

Only available on IA-32 architecture. IA-32 support is deprecated and will be removed in a future release.

Syntax

Linux OS:

`-falign-stack=`*`mode`*

macOS:

None

Windows OS:

None

Arguments

mode	Is the method to use for stack alignment. Possible values are:	
	`assume-4-byte`	Tells the compiler to assume the stack is aligned on 4-byte boundaries. The compiler can dynamically adjust the stack to 16-byte alignment if needed.
	`maintain-16-byte`	Tells the compiler to not assume any specific stack alignment, but attempt to maintain alignment in case the stack is already aligned. The compiler can dynamically align the stack if needed. This setting is compatible with gcc.
	`assume-16-byte`	Tells the compiler to assume the stack is aligned on 16-byte boundaries and to continue to maintain 16-byte alignment. This setting is compatible with gcc.

Default

`-falign-stack=assume-16-byte`	The compiler assumes the stack is aligned on 16-byte boundaries and continues to maintain 16-byte alignment.

Description

This option tells the compiler the stack alignment to use on entry to routines.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

None

Alternate Options

None

fcommon

Determines whether the compiler treats common symbols as global definitions.

Syntax

Linux OS:

`-fcommon`

`-fno-common`

macOS:

`-fcommon`

`-fno-common`

Windows OS:

None

Arguments

None

Default

`-fcommon` The compiler does not treat common symbols as global definitions.

Description

This option determines whether the compiler treats common symbols as global definitions and to allocate memory for each symbol at compile time.

Option `-fno-common` tells the compiler to treat common symbols as global definitions. When using this option, you can only have a common variable declared in one module; otherwise, a link time error will occur for multiple defined symbols.

Normally, a file-scope declaration with no initializer and without the `extern` or `static` keyword "int i;" is represented as a common symbol. Such a symbol is treated as an external reference. However, if no other compilation unit has a global definition for the name, the linker allocates memory for it.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Allow gprel Addressing of Common Data Variables**

OS X

Xcode: **Data > Allow gprel Addressing of Common Data Variables**

Alternate Options

None

fextend-arguments, Qextend-arguments

Controls how scalar integer arguments are extended in calls to unprototyped and varargs functions.

Syntax

Linux OS and macOS:

`-fextend-arguments=`*n*

Windows OS:

`/Qextend-arguments:`*n*

Arguments

n	Specifies the extension for the integer parameters. Possible values are:	
	`32`	Causes unprototyped integer parameters to be extended to 32 bits.
	`64`	Causes unprototyped integer parameters to be extended to 64 bits. This value is only available on Intel® 64 architecture.

Default

`-fextend-arguments=32` or `/Qextend-arguments:32`	Unprototyped integer parameters are extended to 32 bits.

Description

This option controls how scalar integer arguments are extended in calls to unprototyped and varargs functions.

IDE Equivalent

None

Alternate Options

None

fkeep-static-consts, Qkeep-static-consts

Tells the compiler to preserve allocation of variables that are not referenced in the source.

Syntax

Linux OS:

`-fkeep-static-consts`

`-fno-keep-static-consts`

macOS:

`-fkeep-static-consts`

`-fno-keep-static-consts`

Windows OS:

`/Qkeep-static-consts`

`/Qkeep-static-consts-`

Arguments

None

Default

`-fno-keep-static-consts` or `/Qkeep-static-consts-`	If a variable is never referenced in a routine, the variable is discarded unless optimizations are disabled by option `-O0` (Linux* and macOS) or `/Od` (Windows*).

Description

This option tells the compiler to preserve allocation of variables that are not referenced in the source.

The negated form can be useful when optimizations are enabled to reduce the memory usage of static data.

IDE Equivalent

None

Alternate Options

None

fmath-errno

Tells the compiler that errno can be reliably tested after calls to standard math library functions.

Syntax

Linux OS:

`-fmath-errno`

`-fno-math-errno`

macOS:

`-fmath-errno`

`-fno-math-errno`

Windows OS:

`None`

Arguments

None

Default

`-fno-math-errno`	The compiler assumes that the program does not test `errno` after calls to standard math library functions.

Description

This option tells the compiler to assume that the program tests `errno` after calls to math library functions. This restricts optimization because it causes the compiler to treat most math functions as having side effects.

Option `-fno-math-errno` tells the compiler to assume that the program does not test `errno` after calls to math library functions. This frequently allows the compiler to generate faster code. Floating-point code that relies on IEEE exceptions instead of `errno` to detect errors can safely use this option to improve performance.

IDE Equivalent

None

Alternate Options

None

fminshared

Specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.

Syntax

Linux OS:

`-fminshared`

macOS:

`-fminshared`

Windows OS:

None

Arguments

None

Default

OFF	Source files are compiled together to form a single object file.

Description

This option specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.

This option allows the compiler to optimize references to defined symbols without special visibility settings. To ensure that external and common symbol references are optimized, you need to specify visibility hidden or protected by using the `-fvisibility`, `-fvisibility-hidden`, or `-fvisibility-protected` option.

Also, the compiler does not need to generate position-independent code for the main program. It can use absolute addressing, which may reduce the size of the global offset table (GOT) and may reduce memory traffic.

IDE Equivalent

None

Alternate Options

None

See Also

`fvisibility` compiler option

fmudflap

Tells the compiler to instrument risky pointer operations to prevent buffer overflows and invalid heap use. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-fmudflap`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not instrument risky pointer operations.

Description

Tells the compiler to instrument risky pointer operations to prevent buffer overflows and invalid heap use. It requires gcc 4.0 or newer.

This is a deprecated option that may be removed in a future release. There is no replacement option. You can consider using the Pointer Checker options (such as option `check-pointers`).

When using option `fmudflap`, you must specify linker option `-lmudflap` in the link command line to resolve references to the `libmudflap` library.

IDE Equivalent

None

Alternate Options

None

fpack-struct

Specifies that structure members should be packed together.

Syntax

Linux OS:

`-fpack-struct`

macOS:

`-fpack-struct`

Windows OS:

None

Arguments

None

Default

OFF

Description

Specifies that structure members should be packed together.

> **NOTE**
> Using this option may result in code that is not usable with standard (system) c and C++ libraries.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-Zp1`

Windows: None

fpascal-strings

Tells the compiler to allow for Pascal-style string literals.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-fpascal-strings`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not allow for Pascal-style string literals.

Description

Tells the compiler to allow for Pascal-style string literals.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: None

OS X

Xcode: **Data > Recognize Pascal Strings**

Alternate Options

None

fpic

Determines whether the compiler generates position-independent code.

Syntax

Linux OS:

`-fpic`

`-fno-pic`

macOS:

`-fpic`

`-fno-pic`

Windows OS:

`None`

Arguments

None

Default

`-fno-pic`	The compiler does not generate position-independent code.

Description

This option determines whether the compiler generates position-independent code.

Option `-fpic` specifies full symbol preemption. Global symbol definitions as well as global symbol references get default (that is, preemptable) visibility unless explicitly specified otherwise.

Option `-fpic` must be used when building shared objects.

This option can also be specified as `-fPIC`.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Code Generation > Generate Position Independent Code**

OS X

Xcode: None

Alternate Options

None

fpie

Tells the compiler to generate position-independent code. The generated code can only be linked into executables.

Syntax

Linux OS:

`-fpie`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not generate position-independent code for an executable-only object.

Description

This option tells the compiler to generate position-independent code. It is similar to `-fpic`, but code generated by `-fpie` can only be linked into an executable.

Because the object is linked into an executable, this option causes better optimization of some symbol references.

To ensure that run-time libraries are set up properly for the executable, you should also specify option `-pie` to the compiler driver on the link command line.

Option `-fpie` can also be specified as `-fPIE`.

IDE Equivalent

None

Alternate Options

None

See Also

fpic compiler option
pie compiler option

freg-struct-return

Tells the compiler to return struct and union values in registers when possible.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-freg-struct-return`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

This option tells the compiler to return `struct` and `union` values in registers when possible.

IDE Equivalent

None

Alternate Options

None

fstack-protector

Enables or disables stack overflow security checks for certain (or all) routines.

Syntax

Linux OS:

`-fstack-protector[-`*`keyword`*`]`

`-fno-stack-protector[-`*`keyword`*`]`

macOS:

`-fstack-protector[-`*`keyword`*`]`

`-fno-stack-protector[-`*`keyword`*`]`

Windows OS:

None

Arguments

keyword	Possible values are:	
	`strong`	When option `-fstack-protector-strong` is specified, it enables stack overflow security checks for routines with any type of buffer.
	`all`	When option `-fstack-protector-all` is specified, it enables stack overflow security checks for every routine.

If no `-keyword` is specified, option `-fstack-protector` enables stack overflow security checks for routines with a string buffer.

Default

`-fno-stack-protector`, `-fno-stack-protector-strong`	No stack overflow security checks are enabled for the relevant routines.
`-fno-stack-protector-all`	No stack overflow security checks are enabled for any routines.

Description

This option enables or disables stack overflow security checks for certain (or all) routines. A stack overflow occurs when a program stores more data in a variable on the execution stack than is allocated to the variable. Writing past the end of a string buffer or using an index for an array that is larger than the array bound could cause a stack overflow and security violations.

The `-fstack-protector` options are provided for compatibility with gcc. They use the gcc/glibc implementation when possible. If the gcc/glibc implementation is not available, they use the Intel implementation.

For an Intel-specific version of this feature, see option `-fstack-security-check`.

IDE Equivalent

None

Alternate Options

None

See Also

`fstack-security-check` compiler option
`GS` compiler option

fstack-security-check

Determines whether the compiler generates code that detects some buffer overruns.

Syntax

Linux OS:

`-fstack-security-check`

`-fno-stack-security-check`

macOS:

`-fstack-security-check`

`-fno-stack-security-check`

Windows OS:

None

Arguments

None

Default

`-fno-stack-security-check`	The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite the return address. This is a common technique for exploiting code that does not enforce buffer size restrictions.

This option always uses an Intel implementation.

For a gcc-compliant version of this feature, see option `fstack-protector`.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/GS`

See Also

`fstack-protector` compiler option
`GS` compiler option

fvisibility

Specifies the default visibility for global symbols or the visibility for symbols in a file.

Syntax

Linux OS:

`-fvisibility=`*keyword*

`-fvisibility-`*keyword*`=`*filename*

macOS:

`-fvisibility=`*keyword*

`-fvisibility-`*keyword*`=`*filename*

Windows OS:

None

Arguments

keyword	Specifies the visibility setting. Possible values are:	
	`default`	Sets visibility to default.
	`extern`	Sets visibility to extern.
	`hidden`	Sets visibility to hidden.
	`internal`	Sets visibility to internal.
	`protected`	Sets visibility to protected. This value is not available on macOS systems.
filename	Is the pathname of a file containing the list of symbols whose visibility you want to set. The symbols must be separated by whitespace (spaces, tabs, or newlines).	

Default

-fvisibility=default	The compiler sets visibility of symbols to default.

Description

This option specifies the default visibility for global symbols (syntax `-fvisibility=`*keyword*) or the visibility for symbols in a file (syntax `-fvisibility-`*keyword*`=`*filename*).

Visibility specified by `-fvisibility-`*keyword*`=`*filename* overrides visibility specified by `-fvisibility=`*keyword* for symbols specified in a file.

Option	Description
`-fvisibility=default` `-fvisibility-default=`*filename*	Sets visibility of symbols to default. This means other components can reference the symbol, and the symbol definition can be overridden (preempted) by a definition of the same name in another component.

Option	Description
`-fvisibility=extern` `-fvisibility-extern=`*filename*	Sets visibility of symbols to extern. This means the symbol is treated as though it is defined in another component. It also means that the symbol can be overridden by a definition of the same name in another component.
`-fvisibility=hidden` `-fvisibility-hidden=`*filename*	Sets visibility of symbols to hidden. This means that other components cannot directly reference the symbol. However, its address may be passed to other components indirectly.
`-fvisibility=internal` `-fvisibility-internal=`*filename*	Sets visibility of symbols to internal. This means that the symbol cannot be referenced outside its defining component, either directly or indirectly. The affected functions can never be called from another module, including through function pointers.
`-fvisibility=protected` `-fvisibility-protected=`*filename*	Sets visibility of symbols to protected. This means other components can reference the symbol, but it cannot be overridden by a definition of the same name in another component. This value is not available on macOS systems.

If an `-fvisibility` option is specified more than once on the command line, the last specification takes precedence over any others.

If a symbol appears in more than one visibility *filename*, the setting with the least visibility takes precedence.

The following shows the precedence of the visibility settings (from greatest to least visibility):

- `extern`
- `default`
- `protected`
- `hidden`
- `internal`

Note that `extern` visibility only applies to functions. If a variable symbol is specified as `extern`, it is assumed to be `default`.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Default Symbol Visibility**

OS X

Xcode: **Data > Default Symbol Visibility**

Alternate Options

None

Example

A file named `prot.txt` contains symbols a, b, c, d, and e. Consider the following:

```
-fvisibility-protected=prot.txt
```

This option sets `protected` visibility for all the symbols in the file. It has the same effect as specifying `fvisibility=protected` in the declaration for each of the symbols.

fvisibility-inlines-hidden

Causes inline member functions (those defined in the class declaration) to be marked hidden.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-fvisibility-inlines-hidden`

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	The compiler does not cause inline member functions to be marked hidden.

Description

Causes inline member functions (those defined in the class declaration) to be marked hidden. This option is particularly useful for templates.

IDE Equivalent

None

Alternate Options

None

fzero-initialized-in-bss, Qzero-initialized-in-bss

Determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.

Syntax

Linux OS:

`-fzero-initialized-in-bss`

`-fno-zero-initialized-in-bss`

macOS:

`-fzero-initialized-in-bss`

`-fno-zero-initialized-in-bss`

Windows OS:

`/Qzero-initialized-in-bss`

`/Qzero-initialized-in-bss-`

Arguments

None

Default

`-fno-zero-initialized-in-bss` or `/Qzero-initialized-in-bss-`	Variables explicitly initialized with zeros are placed in the BSS section. This can save space in the resulting code.

Description

This option determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.

If option `-fno-zero-initialized-in-bss` (Linux* and macOS) or `/Qzero-initialized-in-bss-` (Windows*) is specified, the compiler places in the DATA section any variables that are initialized to zero.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Disable Placement of Zero-Initialized Variables in .bss - place in .data instead**

OS X

Xcode: **Data > Place Zero-Initialized Variables in .bss**

Alternate Options

None

GA

Enables faster access to certain thread-local storage (TLS) variables.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/GA`

Arguments

None

Default

OFF	Default access to TLS variables is in effect.

Description

This option enables faster access to certain thread-local storage (TLS) variables. When you compile your main executable (`.EXE`) program with this option, it allows faster access to TLS variables declared with the `__declspec(thread)` specification.

Note that if you use this option to compile `.DLL`s, you may get program errors.

IDE Equivalent

Windows

Visual Studio: **Optimization > Optimize for Windows Applications**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

Gs

Lets you control the threshold at which the stack checking routine is called or not called.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Gs[`*n*`]`

Arguments

n	Is the number of bytes that local variables and compiler temporaries can occupy before stack checking is activated. This is called the *threshold*.

Default

`/Gs`	Stack checking occurs for routines that require more than 4KB (4096 bytes) of stack space. This is also the default if you do not specify *n*.

Description

This option lets you control the threshold at which the stack checking routine is called or not called. If a routine's local stack allocation exceeds the threshold (*n*), the compiler inserts a __chkstk() call into the prologue of the routine.

IDE Equivalent

None

Alternate Options

None

GS

Determines whether the compiler generates code that detects some buffer overruns.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/GS[:`*`keyword`*`]`

`/GS-`

Arguments

`keyword` Specifies the level of stack protection heuristics used by the compiler. Possible values are:

`off`	Tells the compiler to ignore buffer overruns. This is the same as specifying `/GS-`.
`partial`	Tells the compiler to provide a stack protection level that is compatible with Microsoft* Visual Studio 2008.
`strong`	Tells the compiler to provide full stack security level checking. This setting is compatible with more recent Microsoft* Visual Studio stack protection heuristics. This is the same as specifying `/GS` with no keyword.

Default

`/GS-`	The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite a function's return address, exception handler address, or certain types of parameters.

This option has been added for Microsoft compatibility.

Following Visual Studio 2008, the Microsoft implementation of option `/GS` became more extensive (for example, more routines are protected). The performance of some programs may be impacted by the newer heuristics. In such cases, you may see better performance if you specify `/GS:partial`.

For more details about option `/GS`, see the Microsoft documentation.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Security Check**

Eclipse

Eclipse: None

Xcode

Xcode: None

Alternate Options

Linux and macOS: `-fstack-security-check`

Windows: None

See Also

`fstack-security-check` compiler option

`fstack-protector` compiler option

GT

Enables fiber-safe thread-local storage of data.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/GT`

Arguments

None

Default

OFF	There is no fiber-safe thread-local storage.

Description

This option enables fiber-safe thread-local storage (TLS) of data.

IDE Equivalent

Windows

Visual Studio: **Optimization > Enable Fiber-safe Optimizations**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

homeparams

Tells the compiler to store parameters passed in registers to the stack.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/homeparams`

Arguments

None

Default

OFF	Register parameters are not written to the stack.

Description

This option tells the compiler to store parameters passed in registers to the stack.

IDE Equivalent

None

Alternate Options

None

malign-double

Determines whether double, long double, and long long types are naturally aligned. This option is equivalent to specifying option align.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-malign-double`

`-mno-align-double`

macOS:

None

Windows OS:

None

Arguments

None

Default

`-mno-align-double`	Types are aligned according to the gcc model, which means they are aligned to 4-byte boundaries.

Description

For details, see the align option.

IDE Equivalent

None

Alternate Options

None

malign-mac68k

Aligns structure fields on 2-byte boundaries (m68k compatible).

Syntax

Linux OS:

None

macOS:

`-malign-mac68k`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not align structure fields on 2-byte boundaries.

Description

This option aligns structure fields on 2-byte boundaries (m68k compatible).

IDE Equivalent

None

Alternate Options

None

malign-natural

Aligns larger types on natural size-based boundaries (overrides ABI).

Syntax

Linux OS:

`None`

macOS:

`-malign-natural`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not align larger types on natural size-based boundaries.

Description

This option aligns larger types on natural size-based boundaries (overrides ABI).

IDE Equivalent

None

Alternate Options

None

malign-power

Aligns based on ABI-specified alignment rules.

Syntax

Linux OS:

`None`

macOS:

`-malign-power`

Windows OS:

`None`

Arguments

None

Default

ON	The compiler aligns based on ABI-specified alignment rules.

Description

Aligns based on ABI-specified alignment rules.

IDE Equivalent

None

Alternate Options

None

mcmodel

Tells the compiler to use a specific memory model to generate code and store data.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`-mcmodel=`*`mem_model`*

macOS:

`None`

Windows OS:

`None`

Arguments

mem_model	Is the memory model to use. Possible values are:

`small`	Tells the compiler to restrict code and data to the first 2GB of address space. All accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.
`medium`	Tells the compiler to restrict code to the first 2GB; it places no memory restriction on data. Accesses of code can be done with IP-relative addressing, but accesses of data must be done with absolute addressing.
`large`	Places no memory restriction on code or data. All accesses of code and data must be done with absolute addressing.

Default

`-mcmodel=small`	On systems using Intel® 64 architecture, the compiler restricts code and data to the first 2GB of address space. Instruction Pointer (IP)-relative addressing can be used to access code and data.

Description

This option tells the compiler to use a specific memory model to generate code and store data. It can affect code size and performance. If your program has global and static data with a total size smaller than 2GB, `-mcmodel=small` is sufficient. Global and static data larger than 2GB requires`-mcmodel=medium` or `-mcmodel=large`. Allocation of memory larger than 2GB can be done with any setting of `-mcmodel`.

IP-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. IP-relative addressing is somewhat faster. So, the `small` memory model has the least impact on performance.

NOTE
When you specify option `-mcmodel=medium` or `-mcmodel=large`, it sets option `-shared-intel`. This ensures that the correct dynamic versions of the Intel run-time libraries are used.

If you specify option `-static-intel` while `-mcmodel=medium` or `-mcmodel=large` is set, an error will be displayed.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to compile using `-mcmodel`:

```
icl -shared-intel -mcmodel=medium -o prog prog.c
```

See Also

`shared-intel` compiler option
`fpic` compiler option

mdynamic-no-pic

Generates code that is not position-independent but has position-independent external references.

Syntax

Linux OS:

`None`

macOS:

`-mdynamic-no-pic`

Windows OS:

`None`

Arguments

None

Default

OFF	All references are generated as position independent.

Description

This option generates code that is not position-independent but has position-independent external references.

The generated code is suitable for building executables, but it is not suitable for building shared libraries.

This option may reduce code size and produce more efficient code. It overrides the `-fpic` compiler option.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic` compiler option

mlong-double

Lets you override the default configuration of the long double data type.

Syntax

Linux OS:

`-mlong-double-`*n*

macOS:

`None`

Windows OS:

`None`

Arguments

n	Specifies the size of the long double data type. Possible values are:	
	`64`	Specifies that the size of the long double data type is 64 bits.
	`80`	Specifies that the size of the long double data type is 80 bits. This is the default.
	`128`	Specifies that the size of the long double data type is 128 bits.

Default

`-mlong-double-80` Specifies that the size of the long double data type is 80 bits.

Description

This option lets you override the default configuration of the long double data type.

When you specify `-mlong-double-64`, the size of the long double data type is 8 bytes and the macro __LONG_DOUBLE_64__ is defined.

When you specify `-mlong-double-80`, the size of the long double data type is 12 bytes on IA-32 architecture and 16 bytes on Intel® 64 architecture.

This option has no effect on floating-point significand precision. That must be specified by using the `-pc64` or `-pc80` option.

Note that this option has no effect when you pass arguments. When you pass arguments, the 64-bit long double data type is treated as the double data type and it is always 64-bit.

Remember to include the math.h and complex.h header files when you use this option.

The following restrictions apply to this option:

- __bultin_* functions using the long double type should not be used in the non-default mode with Intel compiler libraries.
- long double functions from the 'std' namespace should not be called from C++ sources when the non-default mode is set.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/Qlong-double`

See Also

pc, Qpc
  compiler option

no-bss-init, Qnobss-init

Tells the compiler to place in the DATA section any uninitialized variables and explicitly zero-initialized variables. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`-no-bss-init`

macOS:

`-no-bss-init`

Windows OS:

`/Qnobss-init`

Arguments

None

Default

OFF	Uninitialized variables and explicitly zero-initialized variables are placed in the BSS section.

Description

This option tells the compiler to place in the DATA section any uninitialized variables and explicitly zero-initialized variables.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Data > Disable Placement of Zero-initialized and Uninitialized Variables in .bss - place in .data instead**

OS X

Xcode: **Data > Allocate Zero-initialized Variables to .data**

Alternate Options

None

noBool

Disables the bool keyword.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/noBool`

Arguments

None

Default

OFF	The `bool` keyword is enabled.

Description

This option disables the `bool` keyword.

IDE Equivalent

None

Alternate Options

None

Qlong-double

Changes the default size of the long double data type.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qlong-double`

Arguments

None

Default

OFF	The default size of the long double data type is 64 bits.

Description

This option changes the default size of the long double data type to 80 bits.

However, the alignment requirement of the data type is 16 bytes, and its size must be a multiple of its alignment, so the size of a long double on Windows* is also 16 bytes. Only the lower 10 bytes (80 bits) of the 16 byte space will have valid data stored in it.

NOTE
Using the `Qlong-double` command-line option on Windows* platforms requires that any source code using `double extended precision` floating-point types (FP80) be carefully segregated from source code that was not written in a way that considers or supports their use. When this option is used, source code that makes assumptions or has requirements on the size or layout of an FP80 value may experience a variety of failures at compile time, link time, or run time.

The Microsoft* C Standard Library and Microsoft* C++ Standard Template Library do *not* support FP80 datatypes. In all circumstances where you want to use this option, please check with your library vendor to determine whether they support FP80 datatype formats.

For example, the Microsoft* compiler and Microsoft*-provided library routines (such as `printf` or `long double` math functions) do *not* provide support for 80-bit floating-point values and should not be called from code compiled with the `Qlong-double` command-line option.

Starting with the Microsoft Visual Studio 2019 version 16.10 release, you may get compilation errors when using options /std:c++latest together with /Qlong-double in programs that directly or indirectly include the <complex> header, <xutility> header, or the <cmath> header. To see an example of this, see the Example section below.

IDE Equivalent

None

Alternate Options

None

Example

In the Note above, we mention an issue with using the options /std:c++latest together with /Qlong-double in programs that directly or indirectly include the <complex>, <xutility>, or the <cmath> headers. The following shows an example of this issue:

```
#include <iostream>
#include <complex>

int main()
{long double ld2 = 1256789.98765432106L;int iNan = isnan(ld2);std::cout << "Hello World!\n"; }

ksh-3.2$ icl -c -EHsc -GR    -std:c++latest /Qlong-double /MD  test1.cpp
Intel(R) C++ Intel(R) 64 Compiler Classic for applications running on Intel(R) 64, Version xxx
Build xxxx
Copyright (C) 1985-2021 Intel Corporation.  All rights reserved.

test1.cpp
c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/14.29.30130/include/
xutility(5918): error: no instance of function template "std::_Bit_cast" matches the argument
list
            argument types are: (const long double)
      const auto _Bits = _Bit_cast<_Uint_type>(_Xx);
                         ^
```

```
c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/14.29.30130/include/
xutility(67): note: this candidate was rejected because at least one template argument could not
be deduced
  _NODISCARD _CONSTEXPR_BIT_CAST _To _Bit_cast(const _From& _Val) noexcept {
                                     ^

          detected during:
            instantiation of "auto std::_Float_abs_bits(const _Ty &) [with _Ty=long double,
<unnamed>=0]" at line 5967
            instantiation of "bool std::_Is_finite(_Ty) [with _Ty=long double, <unnamed>=0]" at
line 1307 of "c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/14.29.30130/
include/cmath"
            instantiation of "_Ty std::_Common_lerp(_Ty, _Ty, _Ty) noexcept [with _Ty=long
double]" at line 1392 of "c:/Program files/Microsoft Visual Studio/2022/Preview/VC/Tools/MSVC/
14.29.30130/include/cmath"

compilation aborted for test1.cpp (code 2)
```

Qsfalign

Specifies stack alignment for functions. This is a deprecated option that may be removed in a future release.

Architecture Restrictions

Only available on IA-32 architecture. IA-32 support is deprecated and will be removed in a future release.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qsfalign[`*n*`]`

Arguments

n	Is the byte size of aligned variables. Possible values are:	
	`8`	Specifies that alignment should occur for functions with 8-byte aligned variables. At this setting the compiler aligns the stack to 16 bytes if there is any 16-byte or 8-byte data on the stack. For 8-byte data, the compiler only aligns the stack if the alignment will produce a performance advantage.
	`16`	Specifies that alignment should occur for functions with 16-byte aligned variables. At this setting, the compiler only aligns the stack for 16-byte data. No attempt is made to align for 8-byte data.

Default

`/Qsfalign8`	Alignment occurs for functions with 8-byte aligned variables.

Description

This option specifies stack alignment for functions. It lets you disable the normal optimization that aligns a stack for 8-byte data.

This is a deprecated option that may be removed in a future release. There is no replacement option.

If you do not specify *n*, stack alignment occurs for all functions. If you specify `/Qsfalign-`, no stack alignment occurs for any function.

IDE Equivalent

None

Alternate Options

None

Compiler Diagnostic Options

This section contains descriptions for compiler options that pertain to compiler diagnostics.

diag, Qdiag

Controls the display of diagnostic information during compilation.

Syntax

Linux OS:

`-diag-`*`type`*`=`*`diag-list`*

macOS:

`-diag-`*`type`*`=`*`diag-list`*

Windows OS:

`/Qdiag-`*`type`*`:`*`diag-list`*

Arguments

type	Is an action to perform on diagnostics. Possible values are:	
	`enable`	Enables a diagnostic message or a group of messages. If you specify `-diag-enable=all` (Linux* and macOS) or `/Qdiag-enable:all` (Windows*), all diagnostic messages shown in *diag-list* are enabled.
	`disable`	Disables a diagnostic message or a group of messages. If you specify `-diag-disable=all` (Linux* and macOS) or `/Qdiag-disable:all` (Windows*), all diagnostic messages shown in *diag-list* are disabled.

`error`	Tells the compiler to change diagnostics to errors.
`warning`	Tells the compiler to change diagnostics to warnings.
`remark`	Tells the compiler to change diagnostics to remarks (comments).

diag-list

Is a diagnostic group or ID value. Possible values are:

`driver`	Specifies diagnostic messages issued by the compiler driver.
`port-linux`	Specifies diagnostic messages for language features that may cause errors when porting to Linux* systems. This diagnostic group is only available on Windows* systems.
`port-win`	Specifies diagnostic messages for GNU extensions that may cause errors when porting to Windows. This diagnostic group is only available on Linux and macOS systems.
`thread`	Specifies diagnostic messages that help in thread-enabling a program.
`vec`	Specifies diagnostic messages issued by the vectorizer.
`par`	Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer).
`openmp`	Specifies diagnostic messages issued by the OpenMP* parallelizer.
`warn`	Specifies diagnostic messages that have a "warning" severity level.
`error`	Specifies diagnostic messages that have an "error" severity level.
`remark`	Specifies diagnostic messages that are remarks or comments.
`cpu-dispatch`	Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default.
`id[,id,...]`	Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each `id`.

`tag[,tag,...]`	Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each `tag`.

The diagnostic messages generated can be affected by certain options, such as `[Q]x`, `/arch` (Windows) or `-m` (Linux and macOS).

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option controls the display of diagnostic information during compilation. Diagnostic messages are output to stderr unless the `[Q]diag-file` option is specified.

To control the diagnostic information reported by the vectorizer, use options `[q or Q]opt-report` and `[q or Q]opt-report-phase`, phase `vec`.

To control the diagnostic information reported by the auto-parallelizer, use options `[q or Q]opt-report` and `[q or Q]opt-report-phase`, phase `par`.

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Disable Specific Diagnostics** (`/Qdiag-disable:id`)

Advanced > Disable Specific Warnings (`/Qdiag-disable`)

Eclipse

Eclipse: **Compilation Diagnostics > Disable Specific Diagnostics**

Xcode

Xcode: **Diagnostics > Disable Specific Diagnostics**

Alternate Options

enable vec	Linux and macOS: `-qopt-report`;
	`-qopt-report -qopt-report-phase=vec`
	
	Windows: `/Qopt-report`;
	`/Qopt-report /Qopt-report-phase:vec`

disable vec	Linux and macOS: `-qopt-report=0 -qopt-report-phase=vec` Windows: `/Qopt-report:0 /Qopt-report-phase:vec`

enable par	Linux and macOS: `-qopt-report`;
	`-qopt-report -qopt-report-phase=par`
	
	Windows: `/Qopt-report`;
	`/Qopt-report /Qopt-report-phase:par`

disable par	Linux and macOS: `-qopt-report=0 -qopt-report-phase=par` Windows: `/Qopt-report:0 /Qopt-report-phase:par`

Example

The following example shows how to enable diagnostic IDs 117, 230 and 450:

```
-diag-enable=117,230,450     ! Linux and macOSsystems
/Qdiag-enable:117,230,450    ! Windows systems
```

The following example shows how to change vectorizer diagnostic messages to warnings:

```
-diag-enable=vec -diag-warning=vec      ! Linux and macOSsystems
/Qdiag-enable:vec /Qdiag-warning:vec    ! Windows systems
```

Note that you need to enable the vectorizer diagnostics before you can change them to warnings.

The following example shows how to disable all auto-parallelizer diagnostic messages:

```
-diag-disable=par     ! Linux and macOSsystems
/Qdiag-disable:par    ! Windows systems
```

The following example shows how to change all diagnostic warnings and remarks to errors:

```
-diag-error=warn,remark    ! Linux and macOSsystems
/Qdiag-error:warn,remark   ! Windows systems
```

The following example shows how to get a list of *only* vectorization diagnostics:

```
-diag-dump -diag-disable=all -diag-enable=vec        ! Linux and macOSsystems
/Qdiag-dump /Qdiag-disable:all /Qdiag-enable:vec     ! Windows systems
```

See Also

`diag-dump, Qdiag-dump` compiler option

`diag-id-numbers, Qdiag-id-numbers` compiler option

`diag-file, Qdiag-file` compiler option

`qopt-report, Qopt-report` compiler option

`x, Qx` compiler option

diag-dump, Qdiag-dump

Tells the compiler to print all enabled diagnostic messages.

Syntax

Linux OS:

`-diag-dump`

macOS:

`-diag-dump`

Windows OS:

`/Qdiag-dump`

Arguments

None

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to print all enabled diagnostic messages. The diagnostic messages are output to `stdout`.

This option prints the enabled diagnostics from all possible diagnostics that the compiler can issue, including any default diagnostics.

If *diag-list* is specified for the `[Q]diag-enable` option, the print out will include the *diag-list* diagnostics.

IDE Equivalent

None

Alternate Options

None

Example

The following example adds vectorizer diagnostic messages to the printout of default diagnostics:

```
-diag-enable vec -diag-dump          ! Linux and macOS systems
/Qdiag-enable:vec /Qdiag-dump        ! Windows systems
```

See Also

`diag, Qdiag` compiler option

diag-enable=power, Qdiag-enable:power

Controls whether diagnostics are enabled for possibly inefficient code that may affect power consumption on IA-32 and Intel® 64 architectures.

Syntax

Linux OS and macOS:

`-diag-enable=power`

`-diag-disable=power`

Windows OS:

`/Qdiag-enable:power`

`/Qdiag-disable:power`

Arguments

None

Default

`-diag-disable=power` or `/Qdiag-disable:power`	Power consumption diagnostics are disabled.

Description

This option controls whether diagnostics are enabled for possibly inefficient code that may affect power consumption on IA-32 and Intel® 64 architectures.

If you specify option `-diag-enable=power` (Linux* and macOS) or `/Qdiag-enable:power` (Windows*), the compiler will detect various API calls with argument values in ranges known to be inefficient for power consumption. The diagnostic issued will point out the problem argument; for example, "power inefficient use of 'Sleep' with argument in range [0;10]".

IDE Equivalent

None

Alternate Options

None

diag-error-limit, Qdiag-error-limit

Specifies the maximum number of errors allowed before compilation stops.

Syntax

Linux OS:

`-diag-error-limit=`*n*

`-no-diag-error-limit`

macOS:

`-diag-error-limit=`*n*

`-no-diag-error-limit`

Windows OS:

`/Qdiag-error-limit:`*n*

`/Qdiag-error-limit-`

Arguments

n	Is the maximum number of error-level or fatal-level compiler errors allowed.

Default

30	A maximum of 30 error-level and fatal-level messages are allowed.

Description

This option specifies the maximum number of errors allowed before compilation stops. It indicates the maximum number of error-level or fatal-level compiler errors allowed for a file specified on the command line.

If you specify the negative form of the `[Q]diag-error-limit` option on the command line, there is no limit on the number of errors that are allowed.

If the maximum number of errors is reached, a warning message is issued and the next file (if any) on the command line is compiled.

IDE Equivalent

Visual Studio

Visual Studio: **Diagnostics > Error Limit**

Eclipse

Eclipse: **Compilation Diagnostics > Set Error Limit**

Xcode

Xcode: **Diagnostics > Error Limit**

Alternate Options

Linux and macOS: `-wn` (this is a deprecated option)

Windows: `/Qwn` (this is a deprecated option)

diag-file, Qdiag-file

Causes the results of diagnostic analysis to be output to a file.

Syntax

Linux OS:

`-diag-file[=`*`filename`*`]`

macOS:

None

Windows OS:

`/Qdiag-file[:`*`filename`*`]`

Arguments

filename	Is the name of the file for output.

Default

OFF	Diagnostic messages are output to stderr.

Description

This option causes the results of diagnostic analysis to be output to a file. The file is placed in the current working directory.

You can include a file extension in *filename*. For example, if *file.txt* is specified, the name of the output file is *file.txt*. If you do not provide a file extension, the name of the file is *filename.diag*.

If *filename* is not specified, the name of the file is *name-of-the-first-source-file.diag*. This is also the name of the file if the name specified for file conflicts with a source file name provided in the command line.

NOTE
If you specify the `[Q]diag-file` option and you also specify the `[Q]diag-file-append` option, the last option specified on the command line takes precedence.

IDE Equivalent

Windows

Visual Studio: **Diagnostics > Diagnostics File**

Diagnostics > Emit Diagnostics to File

Linux

Eclipse: **Compilation Diagnostics > Diagnostics File**

OS X

Xcode: **Diagnostics > Diagnostics File, Diagnostics > Emit Diagnostics to File**

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be output to a file named *my_diagnostics.diag*:

```
-diag-file=my_diagnostics       ! Linux systems
/Qdiag-file:my_diagnostics      ! Windows systems
```

See Also

`diag-file-append, Qdiag-file-append` compiler option

diag-file-append, Qdiag-file-append

Causes the results of diagnostic analysis to be appended to a file.

Syntax

Linux OS:

`-diag-file-append[=`*`filename`*`]`

macOS:

`None`

Windows OS:

`/Qdiag-file-append[:`*`filename`*`]`

Arguments

filename	Is the name of the file to be appended to. It can include a path.

Default

OFF	Diagnostic messages are output to `stderr`.

Description

This option causes the results of diagnostic analysis to be appended to a file. If you do not specify a path, the driver will look for *filename* in the current working directory.

If *filename* is not found, then a new file with that name is created in the current working directory. If the name specified for file conflicts with a source file name provided in the command line, the name of the file is *name-of-the-first-source-file.diag*.

> **NOTE**
> If you specify the `[Q]diag-file-append` option and you also specify the `[Q]diag-file` option, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be appended to a file named *my_diagnostics.txt*:

```
-diag-file-append=my_diagnostics.txt       ! Linux systems
/Qdiag-file-append:my_diagnostics.txt      ! Windows systems
```

See Also

`diag-file, Qdiag-file` compiler option

diag-id-numbers, Qdiag-id-numbers

Determines whether the compiler displays diagnostic messages by using their ID number values.

Syntax

Linux OS:

`-diag-id-numbers`

`-no-diag-id-numbers`

macOS:

`-diag-id-numbers`

`-no-diag-id-numbers`

Windows OS:

`/Qdiag-id-numbers`

`/Qdiag-id-numbers-`

Arguments

None

Default

`-diag-id-numbers` or `/Qdiag-id-numbers`	The compiler displays diagnostic messages by using their ID number values.

Description

This option determines whether the compiler displays diagnostic messages by using their ID number values. If you specify the negative form of the `[Q]diag-id-numbers` option, mnemonic names are output for driver diagnostics only.

IDE Equivalent

None

Alternate Options

None

See Also

`diag, Qdiag` compiler option

diag-once, Qdiag-once

Tells the compiler to issue one or more diagnostic messages only once.

Syntax

Linux OS:

`-diag-once`*`id`*`[,`*`id`*`,...]`

macOS:

`-diag-once`*`id`*`[,`*`id`*`,...]`

Windows OS:

`/Qdiag-once:`*`id`*`[,`*`id`*`,...]`

Arguments

id	Is the ID number of the diagnostic message. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each *id*.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to issue one or more diagnostic messages only once.

IDE Equivalent

None

Alternate Options

Linux: `-wo` (this is a deprecated option)

Windows: `/Qwo` (this is a deprecated option)

fnon-call-exceptions

Allows trapping instructions to throw C++ exceptions.

Syntax

Linux OS and macOS:

`-fnon-call-exceptions`

`-fno-non-call-exceptions`

Windows OS:

`None`

Arguments

None

Default

`-fno-non-call-exceptions` C++ exceptions are not thrown from trapping instructions.

Description

This option allows trapping instructions to throw C++ exceptions. It allows hardware signals generated by trapping instructions to be converted into C++ exceptions and caught using the standard C++ exception handling mechanism. Examples of such signals are SIGFPE (floating-point exception) and SIGSEGV (segmentation violation).

You must write a signal handler that catches the signal and throws a C++ exception. After that, any occurrence of that signal within a C++ try block can be caught by a C++ catch handler of the same type as the C++ exception thrown within the signal handler.

Only signals generated by trapping instructions (that is, memory access instructions and floating-point instructions) can be caught. Signals that can occur at any time, such as SIGALRM, cannot be caught in this manner.

IDE Equivalent

None

Alternate Options

None

traceback

Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

Syntax

Linux OS:

`-traceback`

`-notraceback`

macOS:

`-traceback`

`-notraceback`

Windows OS:

`/traceback`

`/notraceback`

Arguments

None

Default

`notraceback`	No extra information is generated in the object file to produce traceback information.

Description

This option tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time. This is intended for use with C code that is to be linked into a Fortran program.

When the severe error occurs, source file, routine name, and line number correlation information is displayed along with call stack hexadecimal addresses (program counter trace).

Note that when a severe error occurs, advanced users can also locate the cause of the error using a map file and the hexadecimal addresses of the stack displayed when the error occurs.

This option increases the size of the executable program, but has no impact on run-time execution speeds.

It functions independently of the debug option.

On Windows* systems, `traceback` sets the `/Oy-` option, which forces the compiler to use EBP as the stack frame pointer.

On Windows* systems, the linker places the traceback information in the executable image, in a section named ".trace". To see which sections are in an image, use the command:

```
link -dump -summary your_app_name.exe
```

To see more detailed information, use the command:

```
link -dump -headers your_app_name.exe
```

On Linux* systems, to display the section headers in the image (including the header for the .trace section, if any), use the command:

```
objdump -h your_app_name.exe
```

On macOS systems, to display the section headers in the image, use the command:

```
otool -l your_app_name.exe
```

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Runtime > Generate Traceback Information**

OS X

Xcode: **Runtime > Generate Traceback Information**

Alternate Options

None

w

Disables all warning messages.

Syntax

Linux OS:

`-w`

macOS:

`-w`

Windows OS:

`/w`

Arguments

None

Default

OFF	Default warning messages are enabled.

Description

This option disables all warning messages.

IDE Equivalent

Windows

Visual Studio: **General > Warning Level**

Linux

Eclipse: **General > Warning Level**

OS X

Xcode: **General > Warning Level**

Alternate Options

Linux and macOS: `-w0`

Windows: `/W0`

w, W

Specifies the level of diagnostic messages to be generated by the compiler.

Syntax

Linux OS:

`-wn`

macOS:

`-wn`

Windows OS:

`/Wn`

Arguments

n	Is the level of diagnostic messages to be generated. Possible values are:	
	`0`	Enables diagnostics for errors. Disables diagnostics for warnings.
	`1`	Enables diagnostics for warnings and errors.
	`2`	Enables diagnostics for warnings and errors. On Linux* and macOS systems, additional warnings are enabled. On Windows* systems, this setting is equivalent to level 1 (*n* = 1).
	`3`	Enables diagnostics for remarks, warnings, and errors. Additional warnings are also enabled above level 2 (*n* = 2). This level is recommended for production purposes.
	`4`	Enables diagnostics for all level 3 (*n* = 3) warnings plus informational warnings and remarks, which in most cases can be safely ignored. This value is only available on Windows* systems.
	`5`	Enables diagnostics for all remarks, warnings, and errors. This setting produces the most diagnostic messages. This value is only available on Windows* systems.

Default

n=1	The compiler displays diagnostics for warnings and errors.

Description

This option specifies the level of diagnostic messages to be generated by the compiler.

On Windows systems, option /W4 is equivalent to option /Wall.

The `-wn`, `/Wn`, and `Wall` options can override each other. The last option specified on the command line takes precedence.

IDE Equivalent

Windows

Visual Studio: **General > Warning Level**

Linux

Eclipse: **General > Warning Level**

OS X

Xcode: **General > Warning Level**

Alternate Options

None

See Also

Wall compiler option

Wabi

Determines whether a warning is issued if generated code is not C++ ABI compliant.

Syntax

Linux OS:

`-Wabi`

`-Wno-abi`

macOS:

`-Wabi`

`-Wno-abi`

Windows OS:

None

Arguments

None

Default

`-Wno-abi`	No warning is issued when generated code is not C++ ABI compliant.

Description

This option determines whether a warning is issued if generated code is not C++ ABI compliant.

IDE Equivalent

None

Alternate Options

None

Wall

Enables warning and error diagnostics.

Syntax

Linux OS:

`-Wall`

macOS:

`-Wall`

Windows OS:

`/Wall`

Arguments

None

Default

OFF	Only default warning diagnostics are enabled.

Description

This option enables many warning and error diagnostics.

On Windows* systems, this option is equivalent to the `/W4` option. It enables diagnostics for all level 3 warnings plus informational warnings and remarks.

However, on Linux* and macOS systems, this option is similar to gcc option `-Wall`. It displays all errors and some of the warnings that are typically reported by gcc option `-Wall`. If you want to display all warnings, specify the `-w2` or `-w3` option. If you want to display remarks and comments, specify the `-Wremarks` option.

The `Wall`, `-w`*n*, and `/W`*n* options can override each other. The last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

See Also

diag, Qdiag compiler option
Wremarks compiler option
w, W compiler option

Wbrief

Tells the compiler to display a shorter form of diagnostic output.

Syntax

Linux OS and macOS:

`-Wbrief`

Windows OS:

`/WL`

Arguments

None

Default

OFF	The compiler displays its normal diagnostic output.

Description

This option tells the compiler to display a shorter form of diagnostic output. In this form, the original source line is not displayed and the error message text is not wrapped when too long to fit on a single line.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/WL`

Wcheck

Tells the compiler to perform compile-time code checking for certain code.

Syntax

Linux OS and macOS:

`-Wcheck`

Windows OS:

`/Wcheck`

Arguments

None

Default

OFF	No compile-time code checking is performed.

Description

This option tells the compiler to perform compile-time code checking for certain code. It specifies to check for code that exhibits non-portable behavior, represents a possible unintended code sequence, or possibly affects operation of the program because of a quiet change in the ANSI C Standard.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Compilation Diagnostics > Allow Usage Messages**

OS X

Xcode: **Diagnostics > Allow Usage Messages**

Alternate Options

None

Wcheck-unicode-security

Determines whether the compiler performs source code checking for Unicode vulnerabilities.

Syntax

Linux OS:

`-Wcheck-unicode-security`

`-Wno-check-unicode-security`

macOS:

`-Wcheck-unicode-security`

`-Wno-check-unicode-security`

Windows OS:

`/Wcheck-unicode-security`

`/Wno-check-unicode-security`

Arguments

None

Default

`Wno-check-unicode-security`	The compiler does not perform source code checking for Unicode vulnerabilities.

Description

This option determines whether the compiler performs source code checking for Unicode vulnerabilities.

Option `Wcheck-unicode-security` enables Unicode checking. The compiler will detect and warn about Unicode constructs that can be exploited by using bi-directional formatting codes, zero-width characters in strings, and use of zero-width characters and homoglyphs in identifiers.

Option `Wno-check-unicode-security` disables Unicode checking.

IDE Equivalent

Windows

Visual Studio: **C/C++ > Diagnostics [Intel C++] > Check Unicode Security**

Linux

Eclipse: **Intel C++ Compiler Classic > Compilation Diagnostics > Check Unicode Security**

Alternate Options

None

Wcomment

Determines whether a warning is issued when / appears in the middle of a /* */ comment.*

Syntax

Linux OS:

`-Wcomment`

`-Wno-comment`

macOS:

`-Wcomment`

`-Wno-comment`

Windows OS:

None

Arguments

None

Default

`-Wno-comment`	No warning is issued when /* appears in the middle of a /* */ comment.

Description

This option determines whether a warning is issued when /* appears in the middle of a /* */ comment.

IDE Equivalent

None

Alternate Options

None

Wcontext-limit, Qcontext-limit

Set the maximum number of template instantiation contexts shown in diagnostic.

Syntax

Linux OS and macOS:

`-Wcontext-limit=`*n*

Windows OS:

`/Qcontext-limit:`*n*

Arguments

n	Number of template instantiation contexts.

Default

OFF

Description

Set maximum number of template instantiation contexts shown in diagnostic.

IDE Equivalent

None

Alternate Options

None

wd, Qwd

Disables a soft diagnostic. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS and macOS:

`-wdn[,n]...`

Windows OS:

`/Qwdn[,n]...`

Arguments

n	Is the number of the diagnostic to disable.

Default

OFF	The compiler returns soft diagnostics as usual.

Description

This option disables the soft diagnostic that corresponds to the specified number.

This is a deprecated option that may be removed in a future release. The replacement option is `[Q]diag-disable`.

If you specify more than one *n*, each *n* must be separated by a comma.

IDE Equivalent

Windows

Visual Studio: **Advanced > Disable Specific Warnings**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

Linux and macOS: `-diag-disable`

Windows: `/Qdiag-disable`

Wdeprecated

Determines whether warnings are issued for deprecated C++ headers.

Syntax

Linux OS:

`-Wdeprecated`

`-Wno-deprecated`

macOS:

`-Wdeprecated`

`-Wno-deprecated`

Windows OS:

`None`

Arguments

None

Default

`-Wdeprecated`	The compiler issues warnings for deprecated C++ headers.

Description

This option determines whether warnings are issued for deprecated C++ headers. It has no effect in C compilation mode.

Option `-Wdeprecated` enables these warnings by defining the __DEPRECATED macro for preprocessor.

To disable warnings for deprecated C++ headers, specify `-Wno-deprecated`.

IDE Equivalent

None

Alternate Options

None

we, Qwe

Changes a soft diagnostic to an error. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS and macOS:

`-we`*`Ln[,Ln,...]`*

Windows OS:

`/QweLn[,Ln,...]`

Arguments

Ln	Is the number of the diagnostic to be changed.

Default

OFF	The compiler returns soft diagnostics as usual.

Description

This option overrides the severity of the soft diagnostic that corresponds to the specified number and changes it to an error.

This is a deprecated option that may be removed in a future release. The replacement option is `[Q]diag-error`.

Soft diagnostics are diagnostic messages that don't prevent the production of an object file; for example, warnings and remarks.

If you specify more than one *Ln*, each *Ln* must be separated by a comma.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-diag-error`

Windows: `/Qdiag-error`

Weffc++, Qeffc++

Enables warnings based on certain C++ programming guidelines.

Syntax

Linux OS:

`-Weffc++`

macOS:

`-Weffc++`

Windows OS:

`/Qeffc++`

Arguments

None

Default

OFF	Diagnostics are not enabled.

Description

This option enables warnings based on certain programming guidelines developed by Scott Meyers in his books on effective C++ programming. With this option, the compiler emits warnings for these guidelines:

- Use `const` and `inline` rather than `#define`. Note that you will only get this in user code, not system header code.
- Use `<iostream>` rather than `<stdio.h>`.
- Use `new` and `delete` rather than `malloc` and `free`.
- Use C++ style comments in preference to C style comments. C comments in system headers are not diagnosed.
- Use `delete` on pointer members in destructors. The compiler diagnoses any pointer that does not have a `delete`.
- Make sure you have a user copy constructor and assignment operator in classes containing pointers.
- Use initialization rather than assignment to members in constructors.
- Make sure the initialization list ordering matches the declartion list ordering in constructors.
- Make sure base classes have virtual destructors.
- Make sure `operator=` returns `*this`.
- Make sure prefix forms of increment and decrement return a `const` object.
- Never overload operators `&&`, `||`, and `,`.

> **NOTE**
> The warnings generated by this compiler option are based on the following books from Scott Meyers:
>
> - Effective C++ Second Edition - 50 Specific Ways to Improve Your Programs and Designs
> - More Effective C++ - 35 New Ways to Improve Your Programs and Designs

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Compilation Diagnostics > Enable Warnings for Style Guideline Violations**

OS X

Xcode: **Diagnostics > Report Effective C++ Violations**

Alternate Options

None

Werror, WX

Changes all warnings to errors.

Syntax

Linux OS:

`-Werror`

macOS:

`-Werror`

Windows OS:

`/WX`

Arguments

None

Default

OFF	The compiler returns diagnostics as usual.

Description

This option changes all warnings to errors.

IDE Equivalent

Windows

Visual Studio: **General > Treat Warnings As Errors**

Linux

Eclipse: **Compilation Diagnostics > Treat Warnings As Errors**

OS X

Xcode: **Diagnostics > Treat Warnings As Errors**

Alternate Options

Linux and macOS: `-diag-error warn`

Windows: `/Qdiag-error:warn`

Werror-all

Causes all warnings and currently enabled remarks to be reported as errors.

Syntax

Linux OS:

`-Werror-all`

macOS:

`-Werror-all`

Windows OS:

`/Werror-all`

Arguments

None

Default

OFF	The compiler returns diagnostics as usual.

Description

This option causes all warnings and currently enabled remarks to be reported as errors.

To enable display of remarks, specify option `-Wremarks`.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-diag-error warn, remark`

Windows: `/Qdiag-error:warn, remark`

See Also

diag, Qdiag
compiler option

Wremarks
compiler option

Wextra-tokens

Determines whether warnings are issued about extra tokens at the end of preprocessor directives.

Syntax

Linux OS:

`-Wextra-tokens`

`-Wno-extra-tokens`

macOS:

`-Wextra-tokens`

`-Wno-extra-tokens`

Windows OS:

None

Arguments

None

Default

`-Wno-extra-tokens`	The compiler does not warn about extra tokens at the end of preprocessor directives.

Description

This option determines whether warnings are issued about extra tokens at the end of preprocessor directives.

IDE Equivalent

None

Alternate Options

None

Wformat

Determines whether argument checking is enabled for calls to printf, scanf, and so forth.

Syntax

Linux OS:

`-Wformat`

`-Wno-format`

macOS:

`-Wformat`

`-Wno-format`

Windows OS:

None

Arguments

None

Default

`-Wno-format`	Argument checking is not enabled for calls to `printf`, `scanf`, and so forth.

Description

This option determines whether argument checking is enabled for calls to `printf`, `scanf`, and so forth.

IDE Equivalent

None

Alternate Options

None

Wformat-security

Determines whether the compiler issues a warning when the use of format functions may cause security problems.

Syntax

Linux OS:

`-Wformat-security`

`-Wno-format-security`

macOS:

`-Wformat-security`

`-Wno-format-security`

Windows OS:

`None`

Arguments

None

Default

`-Wno-format-security`	No warning is issued when the use of format functions may cause security problems.

Description

This option determines whether the compiler issues a warning when the use of format functions may cause security problems.

When `-Wformat-security` is specified, it warns about uses of format functions where the format string is not a string literal and there are no format arguments.

IDE Equivalent

None

Alternate Options

None

Wic-pointer

Determines whether warnings are issued for conversions between pointers to distinct scalar types with the same representation.

Syntax

Linux OS and macOS:

`-Wic-pointer`

`-Wno-ic-pointer`

Windows OS:

`None`

Arguments

None

Default

`-Wic-pointer`	The compiler issues warnings for conversions between pointers to distinct scalar types with the same representation.

Description

This option determines whether warnings are issued for conversions between pointers to distinct scalar types with the same representation.

For example, consider the following:

```
void f(int *p) { long *q = p; }
```

In this case, by default, the compiler issues a warning because of the conversion from pointer to `int` to pointer to `long`.

However, if you specify `-Wno-ic-pointer`, and `long` and `int` values have the same representation on the target platform, the warning will not be issued.

IDE Equivalent

None

Alternate Options

None

Winline

Warns when a function that is declared as inline is not inlined.

Syntax

Linux OS:

`-Winline`

macOS:

`-Winline`

Windows OS:

None

Arguments

None

Default

OFF	No warning is produced when a function that is declared as inline is not inlined.

Description

This option warns when a function that is declared as inline is not inlined.

To see diagnostic messages, including a message about why a particular function was not inlined, you should generate an optimization report by specifying option `-qopt-report=5`.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

WL

Tells the compiler to display a shorter form of diagnostic output.

Syntax

Linux OS and macOS:

`See Wbrief.`

Windows OS:

`/WL`

Arguments

None

Default

OFF	The compiler displays its normal diagnostic output.

Description

This option tells the compiler to display a shorter form of diagnostic output. In this form, the original source line is not displayed and the error message text is not wrapped when too long to fit on a single line.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-Wbrief`

Windows: None

Wmain

Determines whether a warning is issued if the return type of main is not expected.

Syntax

Linux OS:

`-Wmain`

`-Wno-main`

macOS:

`-Wmain`

`-Wno-main`

Windows OS:

`None`

Arguments

None

Default

`-Wno-main`	No warning is issued if the return type of `main` is not expected.

Description

This option determines whether a warning is issued if the return type of `main` is not expected.

IDE Equivalent

None

Alternate Options

None

Wmissing-declarations

Determines whether warnings are issued for global functions and variables without prior declaration.

Syntax

Linux OS:

`-Wmissing-declarations`

`-Wno-missing-declarations`

macOS:

`-Wmissing-declarations`

`-Wno-missing-declarations`

Windows OS:

`None`

Arguments

None

Default

`-Wno-missing-declarations`	No warnings are issued for global functions and variables without prior declaration.

Description

This option determines whether warnings are issued for global functions and variables without prior declaration.

IDE Equivalent

None

Alternate Options

None

Wmissing-prototypes

Determines whether warnings are issued for missing prototypes.

Syntax

Linux OS:

`-Wmissing-prototypes`

`-Wno-missing-prototypes`

macOS:

`-Wmissing-prototypes`

`-Wno-missing-prototypes`

Windows OS:

`None`

Arguments

None

Default

`-Wno-missing-prototypes`	No warnings are issued for missing prototypes.

Description

Determines whether warnings are issued for missing prototypes.

If `-Wmissing-prototypes` is specified, it tells the compiler to detect global functions that are defined without a previous prototype declaration.

IDE Equivalent

None

Alternate Options

None

wn, Qwn

Controls the number of errors displayed before compilation stops. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS and macOS:

`-wn`*n*

Windows OS:

`/Qwn`*n*

Arguments

n	Is the number of errors to display.

Default

100	The compiler displays a maximum of 100 errors before aborting compilation.

Description

This option controls the number of errors displayed before compilation stops.

This is a deprecated option that may be removed in a future release. The replacement option is `[Q]diag-error-limit`.

IDE Equivalent

Windows

Visual Studio: **Diagnostics > Error Limit**

Linux

Eclipse: **Compilation Diagnostics > Set Error Limit**

OS X

Xcode: **Diagnostics > Error Limit**

Alternate Options

Linux and macOS: `-diag-error-limit`

Windows: `/Qdiag-error-limit`

Wnon-virtual-dtor

Tells the compiler to issue a warning when a class appears to be polymorphic, yet it declares a non-virtual one.

Syntax

Linux OS and macOS:

`-Wnon-virtual-dtor`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not issue a warning.

Description

Tells the compiler to issue a warning when a class appears to be polymorphic, yet it declares a non-virtual one. This option is supported in C++ only.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: None

OS X

Xcode: **Diagnostics > Report Non-Virtual Destructor**

Alternate Options

None

wo, Qwo

Tells the compiler to issue one or more diagnostic messages only once. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS and macOS:

`-wo`*`Ln[,Ln,...]`*

Windows OS:

`/Qwo`*`Ln[,Ln,...]`*

Arguments

Ln	Is the number of the diagnostic.

Default

OFF

Description

Specifies the ID number of one or more messages. If you specify more than one *Ln*, each *Ln* must be separated by a comma.

This is a deprecated option that may be removed in a future release. The replacement option is `[Q]diag-once id`.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-diag-once id`

Windows: `/Qdiag-once:id`

Wp64

Tells the compiler to display diagnostics for 64-bit porting.

Syntax

Linux OS and macOS:

`-Wp64`

Windows OS:

`/Wp64`

Arguments

None

Default

OFF	The compiler does not display diagnostics for 64-bit porting.

Description

This option tells the compiler to display diagnostics for 64-bit porting.

IDE Equivalent

Windows

Visual Studio: **General > Detect 64-bit Portability Issues**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

Wpch-messages

Determines whether the compiler shows precompiled header (PCH) informational messages.

Syntax

Linux OS and macOS:

`-Wpch-messages`

`-Wno-pch-messages`

Windows OS:

`/Wpch-messages`

`-Wpch-messages-`

Arguments

None

Default

`Wpch-messages`	The compiler shows precompiled header (PCH) informational messages.

Description

This option determines whether the compiler shows precompiled header (PCH) informational messages. By default, these messages are displayed.

To suppress the display of the PCH informational messages, specify `-Wno-pch-messages` (Linux* and macOS) or `/Wpch-messages-` (Windows*).

IDE Equivalent

Windows

Visual Studio: **Precompiled Headers [Intel C++] > Disable Precompiled Header Messages**

Linux

Eclipse: **Precompiled Headers > Disable Precompiled Header Messages**

OS X

Xcode: **Precompiled Headers > Disable Precompiled Header Messages**

Alternate Options

None

Wpointer-arith

Determines whether warnings are issued for questionable pointer arithmetic.

Syntax

Linux OS:

`-Wpointer-arith`

`-Wno-pointer-arith`

macOS:

`-Wpointer-arith`

`-Wno-pointer-arith`

Windows OS:

`None`

Arguments

None

Default

`-Wno-pointer-arith`	No warnings are issued for questionable pointer arithmetic.

Description

Determines whether warnings are issued for questionable pointer arithmetic.

IDE Equivalent

None

Alternate Options

None

Wport

Tells the compiler to issue portability diagnostics.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Wport`

Arguments

None

Default

OFF	The compiler issues default diagnostics.

Description

This option tells the compiler to issue portability diagnostics.

IDE Equivalent

None

Alternate Options

None

wr, Qwr

Changes a soft diagnostic to an remark. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS and macOS:

`-wr`*`Ln[,Ln,...]`*

Windows OS:

`/QwrLn[,Ln,...]`

Arguments

Ln	Is the number of the diagnostic to be changed.

Default

OFF	The compiler returns soft diagnostics as usual.

Description

This option overrides the severity of the soft diagnostic that corresponds to the specified number and changes it to a remark.

This is a deprecated option that may be removed in a future release. The replacement option is `[Q]diag-remark`.

Soft diagnostics are diagnostic messages that don't prevent the production of an object file; for example, warnings and remarks.

If you specify more than one *Ln*, each *Ln* must be separated by a comma.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-diag-remark`

Windows: `/Qdiag-remark`

Wremarks

Tells the compiler to display remarks and comments.

Syntax

Linux OS and macOS:

`-Wremarks`

Windows OS:

`None`

Arguments

None

Default

OFF	Default warning messages are enabled.

Description

This option tells the compiler to display remarks and comments.

If you want to display warnings and errors, specify the `-Wall`, `-wn`, or `/Wn` option.

IDE Equivalent

None

Alternate Options

None

See Also

diag, Qdiag compiler option
Wall compiler option
w, W compiler option

Wreorder

Tells the compiler to issue a warning when the order of member initializers does not match the order in which they must be executed.

Syntax

Linux OS:

`-Wreorder`

macOS:

`-Wreorder`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not issue a warning.

Description

This option tells the compiler to issue a warning when the order of member initializers does not match the order in which they must be executed. This option is supported for C++ only.

IDE Equivalent

None

Alternate Options

None

Wreturn-type

Determines whether warnings are issued when a function is declared without a return type, when the definition of a function returning void contains a return statement with an expression, or when the closing brace of a function returning non-void is reached.

Syntax

Linux OS:

`-Wreturn-type`

`-Wno-return-type`

macOS:

`-Wreturn-type`

`-Wno-return-type`

Windows OS:

`None`

Arguments

None

Default

ON for one condition	A warning is issued when the closing brace of a function returning non-void is reached.

Description

This option determines whether warnings are issued for the following:

- When a function is declared without a return type
- When the definition of a function returning void contains a return statement with an expression
- When the closing brace of a function returning non-void is reached

Specify `-Wno-return-type` if you do not want to see warnings about the above diagnostics.

IDE Equivalent

None

Alternate Options

None

Wshadow

Determines whether a warning is issued when a variable declaration hides a previous declaration.

Syntax

Linux OS:

`-Wshadow`

`-Wno-shadow`

macOS:

`-Wshadow`

`-Wno-shadow`

Windows OS:

`None`

Arguments

None

Default

`-Wno-shadow`	No warning is issued when a variable declaration hides a previous declaration.

Description

This option determines whether a warning is issued when a variable declaration hides a previous declaration. Same as `-ww1599`.

IDE Equivalent

None

Alternate Options

None

Wsign-compare

Determines whether warnings are issued when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

Syntax

Linux OS:

`-Wsign-compare`

`-Wno-sign-compare`

macOS:

`-Wsign-compare`

`-Wno-sign-compare`

Windows OS:

`None`

Arguments

None

Default

`-Wno-sign-compare`	The compiler does not issue these warnings

Description

This option determines whether warnings are issued when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

On Linux* systems, this option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

Wstrict-aliasing

Determines whether warnings are issued for code that might violate the optimizer's strict aliasing rules.

Syntax

Linux OS:

`-Wstrict-aliasing`

`-Wno-strict-aliasing`

macOS:

`-Wstrict-aliasing`

`-Wno-strict-aliasing`

Windows OS:

None

Arguments

None

Default

`-Wno-strict-aliasing`	No warnings are issued for code that might violate the optimizer's strict aliasing rules.

Description

This option determines whether warnings are issued for code that might violate the optimizer's strict aliasing rules. These warnings will only be issued if you also specify option `-ansi-alias` or option `-fstrict-aliasing`.

IDE Equivalent

None

Alternate Options

None

See Also

ansi-alias, Qansi-alias
compiler option

Wstrict-prototypes

Determines whether warnings are issued for functions declared or defined without specified argument types.

Syntax

Linux OS:

`-Wstrict-prototypes`

`-Wno-strict-prototypes`

macOS:

`-Wstrict-prototypes`

`-Wno-strict-prototypes`

Windows OS:

None

Arguments

None

Default

`-Wno-strict-prototypes`	No warnings are issued for functions declared or defined without specified argument types.

Description

This option determines whether warnings are issued for functions declared or defined without specified argument types.

IDE Equivalent

None

Alternate Options

None

Wtrigraphs

Determines whether warnings are issued if any trigraphs are encountered that might change the meaning of the program.

Syntax

Linux OS:

`-Wtrigraphs`

`-Wno-trigraphs`

macOS:

`-Wtrigraphs`

`-Wno-trigraphs`

Windows OS:

`None`

Arguments

None

Default

`-Wno-trigraphs`	No warnings are issued if any trigraphs are encountered that might change the meaning of the program.

Description

This option determines whether warnings are issued if any trigraphs are encountered that might change the meaning of the program.

IDE Equivalent

None

Alternate Options

None

Wuninitialized

Determines whether a warning is issued if a variable is used before being initialized.

Syntax

Linux OS:

`-Wuninitialized`

`-Wno-uninitialized`

macOS:

`-Wuninitialized`

`-Wno-uninitialized`

Windows OS:

`None`

Arguments

None

Default

`-Wno-uninitialized`	No warning is issued if a variable is used before being initialized.

Description

This option determines whether a warning is issued if a variable is used before being initialized. Equivalent to `-ww592` and `-wd592`.

IDE Equivalent

None

Alternate Options

`-ww592` and `-wd592`

Wunknown-pragmas

Determines whether a warning is issued if an unknown #pragma directive is used.

Syntax

Linux OS:

`-Wunknown-pragmas`

`-Wno-unknown-pragmas`

macOS:

`-Wunknown-pragmas`

`-Wno-unknown-pragmas`

Windows OS:

`None`

Arguments

None

Default

`-Wunknown-pragmas`	A warning is issued if an unknown `#pragma` directive is used.

Description

This option determines whether a warning is issued if an unknown `#pragma` directive is used.

IDE Equivalent

None

Alternate Options

None

Wunused-function

Determines whether a warning is issued if a declared function is not used.

Syntax

Linux OS:

`-Wunused-function`

`-Wno-unused-function`

macOS:

`-Wunused-function`

`-Wno-unused-function`

Windows OS:

`None`

Arguments

None

Default

`-Wno-unused-function`	No warning is issued if a declared function is not used.

Description

This option determines whether a warning is issued if a declared function is not used.

IDE Equivalent

None

Alternate Options

None

Wunused-variable

Determines whether a warning is issued if a local or non-constant static variable is unused after being declared.

Syntax

Linux OS:

`-Wunused-variable`

`-Wno-unused-variable`

macOS:

`-Wunused-variable`

`-Wno-unused-variable`

Windows OS:

`None`

Arguments

None

Default

`-Wno-unused-variable`	No warning is issued if a local or non-constant static variable is unused after being declared.

Description

This option determines whether a warning is issued if a local or non-constant static variable is unused after being declared.

IDE Equivalent

None

Alternate Options

None

ww, Qww

Changes a soft diagnostic to an warning. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS and macOS:

`-wwLn[,Ln,...]`

Windows OS:

`/QwwLn[,Ln,...]`

Arguments

Ln	Is the number of the diagnostic to be changed.

Default

OFF	The compiler returns soft diagnostics as usual.

Description

This option overrides the severity of the soft diagnostic that corresponds to the specified number and changes it to an warning.

This is a deprecated option that may be removed in a future release. The replacement option is `[Q]diag-warning`.

Soft diagnostics are diagnostic messages that don't prevent the production of an object file; for example, warnings and remarks.

If you specify more than one *Ln*, each *Ln* must be separated by a comma.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-diag-warning`

Windows: `/Qdiag-warning`

Wwrite-strings

*Issues a diagnostic message if const char * is converted to (non-const) char *.*

Syntax

Linux OS:

`-Wwrite-strings`

macOS:

`-Wwrite-strings`

Windows OS:

`None`

Arguments

None

Default

OFF	No diagnostic message is issued if const char * is converted to (non-const) char*.

Description

This option issues a diagnostic message if const char* is converted to (non-const) char *.

IDE Equivalent

None

Alternate Options

None

Compatibility Options

This section contains descriptions for compiler options that pertain to language compatibility.

clang-name

Specifies the name of the Clang compiler that should be used to set up the environment for C compilations.

Syntax

Linux OS and macOS:

`-clang-name=`*name*

Windows OS:

`None`

Arguments

name	Is the name of the Clang compiler to use. It can include the path where the Clang compiler is located.

Default

OFF	The compiler uses the PATH setting to find the Clang compiler and resolve environment settings.

Description

This option specifies the name of the Clang compiler that should be used to set up the environment for C compilations. If you do not specify a path, the compiler will search the PATH settings for the compiler name you provide.

This option is helpful when you are referencing a non-standard Clang installation.

The C++ equivalent to option `-clang-name` is `-clangxx-name`.

NOTE
This option applies to the Intel compiler running in a CLANG environment. It does not apply to the Intel CLANG-based compiler.

IDE Equivalent

None

Alternate Options

None

Example

If the following option is specified, the compiler looks for the Clang compiler named `foobar` in the PATH setting:

```
-clang-name=foobar
```

If the following option is specified, the compiler looks for the Clang compiler named `foobar` in the path specified:

```
-clang-name=/a/b/foobar
```

See Also

`clangxx-name` compiler option

clangxx-name

Specifies the name of the Clang++ compiler that should be used to set up the environment for C++ compilations.

Syntax

Linux OS and macOS:

`-clangxx-name=`*`name`*

Windows OS:

`None`

Arguments

name	Is the name of the Clang++ compiler to use. It can include the path where the Clang++ compiler is located.

Default

OFF	The compiler uses the PATH setting to find the Clang++ compiler and resolve environment settings.

Description

This option specifies the name of the Clang++ compiler that should be used to set up the environment for C ++ compilations. If you do not specify a path, the compiler will search the PATH settings for the compiler name you provide.

The C equivalent to option `-clangxx-name` is `-clang-name`.

> **NOTE**
> This option applies to the Intel compiler running in a CLANG environment. It does not apply to the Intel CLANG-based compiler.

IDE Equivalent

None

Alternate Options

None

Example

If the following option is specified, the compiler looks for the Clang++ compiler named `foobar` in the PATH setting:

```
-clangxx-name=foobar
```

If the following option is specified, the compiler looks for the Clang++ compiler named `foobar` in the path specified:

```
-clangxx-name=/a/b/foobar
```

See Also

`clang-name` compiler option

fabi-version

Instructs the compiler to select a specific ABI implementation.

Syntax

Linux OS:

`-fabi-version=`*n*

macOS:

`-fabi-version=`*n*

Windows OS:

None

Arguments

n	Is the ABI implementation. Possible values are:	
	0	Requests the latest ABI implementation.
	1	Requests the ABI implementation used in gcc 3.2 and gcc 3.3.
	2	Requests the ABI implementation used in gcc 3.4 and higher.

Default

Varies	The compiler uses the ABI implementation that corresponds to the installed version of gcc.

Description

This option tells the compiler to select a specific ABI implementation. This option is compatible with gcc option `-fabi-version`. If you have multiple versions of gcc installed, the compiler may change the value of *n* depending on which gcc is detected in your path.

NOTE
gcc 3.2 and 3.3 are not fully ABI-compliant, but gcc 3.4 is highly ABI-compliant.

Caution
Do not mix different values for `-fabi-version` in one link.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Preprocessor > gcc Compatibility Options**

OS X

Xcode: None

Alternate Options

None

fms-dialect

Enables support for a language dialect that is compatible with Microsoft Windows, while maintaining link compatibility with GCC*.*

Syntax

Linux OS:

`-fms-dialect[=ver]`

macOS:

`None`

Windows OS:

`None`

Arguments

ver	Indicates that the language dialect should be compatible with a certain version of Microsoft Visual Studio*. Possible values are: `14.2` — Specifies the dialect should be compatible with Microsoft Visual Studio 2019. `14.1` — Specifies the dialect should be compatible with Microsoft Visual Studio 2017. **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

Default

OFF	The compiler does not support a language dialect that is compatible with Microsoft Windows.

Description

This option enables support for a limited language dialect that is compatible with Microsoft Windows, while maintaining link compatibility with GCC. It allows portability of code written on Windows that uses Microsoft extensions or language features. The code will be compiled with syntax and semantics similar to that used by the Microsoft Windows compiler, while continuing to produce object files that are link-compatible with the object files and libraries produced by the GCC compiler and/or by the Intel Compiler without this option.

The `-fms-dialect` option is intended to be used as an aid in porting code written on Windows. It is not intended to enable an all-encompassing capability for porting all such code written on Windows seamlessly. For example, even with this option enabled, there remains a need to support GCC-compatible syntax and semantics for some language constructs in order to generate object files that are link-time compatible with those produced by the GCC compiler and/or by the Intel compiler without this option.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/Qgcc-dialect`

See Also

`Qgcc-dialect` compiler option

gcc-name

Lets you specify the name of the GCC compiler that should be used to set up the environment for C compilations.

Syntax

Linux OS:

`-gcc-name=name`

macOS:

None

Windows OS:

None

Arguments

name	Is the name of the GCC compiler to use. It can include the path where the GCC compiler is located.

Default

OFF	The compiler uses the PATH setting to find the GCC compiler and resolve environment settings.

Description

This option lets you specify the name of the GCC compiler that should be used to set up the environment for C compilations. If you do not specify a path, the compiler will search the PATH settings for the compiler name you provide.

This option is helpful when you are referencing a non-standard GCC installation, or you have multiple GCC installations on your system. The compiler will match GCC version values to the GCC compiler you specify.

The C++ equivalent to option `-gcc-name` is `-gxx-name`.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Preprocessor > Nonstandard gcc Installation**

Xcode

Xcode: None

Alternate Options

None

Example

If the following option is specified, the compiler looks for the GCC compiler named `foobar` in the PATH setting:

```
-gcc-name=foobar
```

If the following option is specified, the compiler looks for the GCC compiler named `foobar` in the path specified:

```
-gcc-name=/a/b/foobar
```

See Also

`gxx-name` compiler option

gnu-prefix

Lets you specify a prefix that will be added to the names of gnu utilities called from the compiler.

Syntax

Linux OS:

`-gnu-prefix=`*`prefix`*

macOS:

None

Windows OS:

None

Arguments

prefix	Is a string that prepends the name of gnu tools called from the compiler. The value depends on the gnu toolchain used for a particular operating system. For example, for Wind River* Linux 6.x, the *prefix* value will be `x86_64-wrs-linux-`. You must append a hyphen to *prefix* only if the toolchain prefix ends with a hyphen. You can specify a short name or a pathname: • short name: `-gnu-prefix=`*prefix* In this case, the compiler calls *prefix<gnu_utility>* instead of *<gnu_utility>*. The utility with this name should be in the PATH environment variable. • pathname: `-gnu-prefix=/directory_name/`*prefix* In this case, the compiler calls `/directory_name/`*prefix<gnu_utility>*. The utility with this name will be invoked by its full pathname.

Default

OFF	The compiler calls gnu utilities by their short names, and looks for them in the path specified by the PATH environment variable.

Description

This option lets you specify a prefix that will be added to the names of gnu utilities called from the compiler. This option is available for Linux*-targeted compilers but the host may be either Windows* or Linux*.

If you specify option `-gnu-prefix` with option `-gcc-name` (or `-gxx-name`), the following occurs:

- If a name specified in `-gcc-name` (or `-gxx-name`) contains a full path to a binary then option `-gnu-prefix` has no effect on the specified name; other binutils will have the prefix.
- Otherwise, option `-gnu-prefix` is applied to the name specified in `-gcc-name` (or `-gxx-name`).

The above approach provides flexibility to specify an alternative gcc name outside of the default toolchain. At the same time, if a short name is provided in option `-gcc-name`, it is assumed to be a part of the default toolchain and a prefix will be added.

Instead of using option `-gnu-prefix`, you can create symlinks for the short names of gnu utilities in the toolchain and add them to the PATH. For example, `ld--> i686-wrs-linux-gnu-ld`.

NOTE
Even though this option is not supported for a Windows-to-Windows native compiler, it is supported for a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

Example

Consider that you are setting up the compiler to produce an application for a Wind River* Linux 6.

Assume that your gnu cross toolchain for the target operating system is located in the following directory:

```
/WRL/60/x86_64-linux/usr/bin/x86_64-wrs-linux
```

and gnu utilities in the toolchain have *prefix*`x86_64-wrs-linux-`.

Assume your sysroot for the target operating system is located in the following directory:

```
/WRL/60/qemux86-64
```

To compile your application for Wind River* Linux 6, you must enter the following commands:

```
export PATH=/WRL/60/x86_64-linux/usr/bin/x86_64-wrs-linux:PATH
icc --sysroot/WRL/60/qemux86-64 -gnu-prefix=x86_64-wrs-linux- app.c
```

The following examples show what happens when you specify both `-gcc-name` and `-gnu-prefix`.

Example 1:

```
Command line: -gcc-name=foobar -gnu-prefix=em64t-
Actual gcc name used in the compiler: em64t-foobar
ld name used in the icc: em64t-ld
```

Example 2:

```
Command line: -gcc-name=/a/b/foobar -gnu-prefix=em64t-
Actual gcc name used in the compiler: /a/b/foobar
ld name used in the icc: em64t-ld
```

See Also

`gcc-name` compiler option

`gxx-name` compiler option

`sysroot` compiler option

gxx-name

Lets you specify the name of the g++ compiler that should be used to set up the environment for C++ compilations.

Syntax

Linux OS:

`-gxx-name=`*`name`*

macOS:

None

Windows OS:

None

Arguments

name	Is the name of the g++ compiler to use. It can include the path where the g++ compiler is located.

Default

OFF	The compiler uses the PATH setting to find the g++ compiler and resolve environment settings.

Description

This option lets you specify the name of the g++ compiler that should be used to set up the environment for C++ compilations. If you do not specify a path, the compiler will search the PATH settings for the compiler name you provide.

This option is helpful if you have multiple gcc++ installations on your system. The compiler will match gcc++ version values to the gcc++ compiler you specify.

The C equivalent to option `-gxx-name` is `-gcc-name`.

> **NOTE** When compiling a C++ file with `icpc`, g++ is used to get the environment.

IDE Equivalent

None

Alternate Options

None

Example

If the following option is specified, the compiler looks for the g++ compiler named `foobar` in the PATH setting:

```
-gxx-name=foobar
```

If the following option is specified, the compiler looks for the g++ compiler named `foobar` in the path specified:

```
-gxx-name=/a/b/foobar
```

See Also

`gcc-name` compiler option

Qgcc-dialect

Enables support for a limited gcc-compatible dialect on Windows.*

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qgcc-dialect:`*`ver`*

Arguments

ver	Indicates the version of the gcc compiler that the limited language dialect should be compatible with. It must be a three-digit number with a value of 440 or higher. The number will be normalized to reflect the gcc compiler version numbering scheme. For example, if you specify 450, it indicates gcc version 4.5.0.

Default

OFF	The compiler does not support a language dialect that is compatible with the gcc compiler.

Description

This option enables support for a limited gcc-compatible dialect on Windows*. It allows portability of code written for the gcc compiler.

This option enables a limited gnu-compatible compiler dialect on Windows. The code will be compiled with syntax and semantics similar to that used by gcc, while continuing to produce object files that are link-compatible with the object files and libraries on Windows (that is, object files and libraries produced by the Microsoft compiler and/or by the Intel compiler without this option).

The `/Qgcc-dialect` option is intended to be used as an aid in porting code written for the gcc compiler. It is not intended to enable an all-encompassing capability for porting all such code written for the gcc compiler seamlessly. For example, even with this option enabled, there remains a need to support Windows-compatible syntax and semantics for some language constructs in order to generate object files that are link-time compatible with those produced by the Windows compiler and/or by the Intel compiler without this option.

IDE Equivalent

None

Alternate Options

Linux and macOS: `-fms-dialect`

Windows: None

See Also

`fms-dialect`
compiler option

Qms

Tells the compiler to emulate Microsoft compatibility bugs.*

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qms`*n*

Arguments

n	Possible values are:	
	`0`	Instructs the compiler to disable some Microsoft* compatibility bugs. It tells the compiler to emulate the fewest number of Microsoft compatibility bugs.
	`1`	Instructs the compiler to enable most Microsoft compatibility bugs. It tells the compiler to emulate more Microsoft compatibility bugs than `/Qms0`.
	`2`	Instructs the compiler to generate code that is Microsoft compatible. The compiler emulates the largest number of Microsoft compatibility bugs.

Default

`/Qms1`	The compiler emulates most Microsoft* compatibility bugs.

Description

This option tells the compiler to emulate Microsoft* compatibility bugs.

Caution
When using `/Qms0`, your program may not compile if it depends on Microsoft headers with compatibility bugs that are disabled with this option. Use `/Qms1` if your compilation fails.

IDE Equivalent

None

Alternate Options

None

Qvc

Specifies compatibility with Microsoft Visual C++ (MSVC) or Microsoft Visual Studio*.*

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qvc14.2`

`/Qvc14.1`

Arguments

None

Default

varies	When the compiler is installed, it detects which version of Microsoft Visual Studio is on your system. `Qvc` defaults to the form of the option that is compatible with that version. When multiple versions of Microsoft Visual Studio are installed, the compiler installation lets you select which version you want to use. In this case, `Qvc` defaults to the version you choose.

Description

This option specifies compatibility with MSVC or Microsoft Visual Studio.

Option	Description
`/Qvc14.2`	Specifies compatibility with Microsoft Visual Studio 2019.
`/Qvc14.1`	Specifies compatibility with Microsoft Visual Studio 2017.

NOTE
Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

IDE Equivalent

None

Alternate Options

None

stdlib

Lets you select the C++ library to be used for linking.

Syntax

Linux OS:

None

macOS:

-stdlib[=*keyword*]

Windows OS:

None

Arguments

keyword	Is the function information to include. Possible values are:	
	libc++	Links using the libc++ library.
	libstdc++	Links using the GNU libstdc++ library.

Default

-stdlib=libc++	The compiler links using the libc++ library.

Description

This option lets you select the C++ library to be used for linking. This option is processed by the command that initiates linking, adding library names explicitly to the link command.

Currently, if you do not specify this option, the libc++ headers and library are used.

> **NOTE**
> The IDE provides another possible setting for option -stdlib, which lets you choose the compiler default rather than a specific library.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: None

Xcode

Xcode: **Language > C++ standard library > libstdc++**

Language > C++ standard library > libc++

Language > C++ standard library > compiler-default

Alternate Options

None

vmv

Enables pointers to members of any inheritance type.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/vmv`

Arguments

None

Default

OFF	The compiler uses default rules to represent pointers to members.

Description

This option enables pointers to members of any inheritance type. To use this option, you must also specify option `/vmg`.

IDE Equivalent

None

Alternate Options

None

Linking or Linker Options

This section contains descriptions for compiler options that pertain to linking or to the linker.

Bdynamic

Enables dynamic linking of libraries at run time.

Syntax

Linux OS:

`-Bdynamic`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	Limited dynamic linking occurs.

Description

This option enables dynamic linking of libraries at run time. Smaller executables are created than with static linking.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option `-Bdynamic` are linked dynamically until the end of the command line or until a `-Bstatic` option is encountered. The `-Bstatic` option enables static linking of libraries.

IDE Equivalent

None

Alternate Options

None

See Also

`Bstatic` compiler option

Bstatic

Enables static linking of a user's library.

Syntax

Linux OS:

`-Bstatic`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	Default static linking occurs.

Description

This option enables static linking of a user's library.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option `-Bstatic` are linked statically until the end of the command line or until a `-Bdynamic` option is encountered. The `-Bdynamic` option enables dynamic linking of libraries.

IDE Equivalent

None

Alternate Options

None

See Also

`Bdynamic` compiler option

Bsymbolic

Binds references to all global symbols in a program to the definitions within a user's shared library.

Syntax

Linux OS:

`-Bsymbolic`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	When a program is linked to a shared library, it can override the definition within the shared library.

Description

This option binds references to all global symbols in a program to the definitions within a user's shared library.

This option is only meaningful on Executable Linkage Format (ELF) platforms that support shared libraries.

Caution
This option can have unintended side-effects of disabling symbol preemption in the shared library.

IDE Equivalent

None

Alternate Options

None

See Also

`Bsymbolic-functions` compiler option

Bsymbolic-functions

Binds references to all global function symbols in a program to the definitions within a user's shared library.

Syntax

Linux OS:

`-Bsymbolic-functions`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	When a program is linked to a shared library, it can override the definition within the shared library.

Description

This option binds references to all global function symbols in a program to the definitions within a user's shared library.

This option is only meaningful on Executable Linkage Format (ELF) platforms that support shared libraries.

> **Caution**
> This option can have unintended side-effects of disabling symbol preemption in the shared library.

IDE Equivalent

None

Alternate Options

None

See Also

`Bsymbolic` compiler option

cxxlib

Determines whether the compiler links using the C++ run-time libraries and header files provided by gcc.

Syntax

Linux OS:

`-cxxlib[=dir]`

`-cxxlib-nostd`

`-no-cxxlib`

macOS:

None

Windows OS:

None

Arguments

dir	Is an optional top-level location for the gcc binaries and libraries.

Default

C++: `-cxxlib` C: `-no-cxxlib`	For C++, the compiler uses the run-time libraries and headers provided by gcc. For C, the compiler uses the default run-time libraries and headers and does not link to any additional C++ run-time libraries and headers. However, if you specify compiler option `-std=gnu++98`, the default is `-cxxlib`.

Description

This option determines whether the compiler links using the C++ run-time libraries and header files provided by gcc.

If you specify *dir* for `cxxlib`, the compiler uses `dir/bin/gcc` to setup the environment.

Option `-cxxlib=`*dir* can be used with option `-gcc-name=`*name* to specify the location `dir/bin/name`.

Option `-cxxlib-nostd` prevents the compiler from linking with the standard C++ library.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Preprocessor > gcc Compatibility Options**

Xcode

Xcode: None

Alternate Options

None

See Also

`gcc-name` compiler option

dynamic-linker

Specifies a dynamic linker other than the default.

Syntax

Linux OS:

`-dynamic-linker file`

macOS:

`None`

Windows OS:

`None`

Arguments

file	Is the name of the dynamic linker to be used.

Default

OFF	The default dynamic linker is used.

Description

This option lets you specify a dynamic linker other than the default.

IDE Equivalent

None

Alternate Options

None

dynamiclib

Invokes the libtool command to generate dynamic libraries.

Syntax

Linux OS:

`None`

macOS:

`-dynamiclib`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option invokes the `libtool` command to generate dynamic libraries.

When passed this option, the compiler uses the `libtool` command to produce a dynamic library instead of an executable when linking.

To build static libraries, you should specify option `-staticlib` or `libtool -static <objects>`.

IDE Equivalent

None

Alternate Options

None

See Also

staticlib compiler option

F (Windows*)

Specifies the stack reserve amount for the program.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Fn`

Arguments

n	Is the stack reserve amount. It can be specified as a decimal integer or as a hexadecimal constant by using a C-style convention (for example, /F0x1000).

Default

OFF	The stack size default is chosen by the operating system.

Description

This option specifies the stack reserve amount for the program. The amount (*n*) is passed to the linker.

Note that the linker property pages have their own option to do this.

IDE Equivalent

None

Alternate Options

None

F (macOS)

Adds a framework directory to the head of an include file search path.

Syntax

Linux OS:

`None`

macOS:

`-Fdir`

Windows OS:

`None`

Arguments

dir	Is the name for the framework directory.

Default

OFF	The compiler does not add a framework directory to the head of an include file search path.

Description

This option adds a framework directory to the head of an include file search path.

IDE Equivalent

None

Alternate Options

None

fixed

Causes the linker to create a program that can be loaded only at its preferred base address.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/fixed`

Arguments

None

Default

OFF	The compiler uses default methods to load programs.

Description

This option is passed to the linker, causing it to create a program that can be loaded only at its preferred base address.

IDE Equivalent

None

Alternate Options

None

Fm

Tells the linker to generate a link map file. This is a deprecated option that may be removed in a future release.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Fm[filename|dir]`

Arguments

filename	Is the name for the link map file.
dir	Is the directory where the link map file should be placed. It can include *file*.

Default

OFF	No link map is generated.

Description

This option tells the linker to generate a link map.

This is a deprecated option that may be removed in a future release. There is no replacement option.

IDE Equivalent

None

Alternate Options

None

fuse-ld

Tells the compiler to use a different linker instead of the default linker (ld).

Syntax

Linux OS:

`-fuse-ld=`*`keyword`*

macOS:

`-fuse-ld=`*`keyword`*

Windows OS:

None

Arguments

keyword	Possible values are:	
	`bfd`	Tells the compiler to use the bfd linker.
	`gold`	Tells the compiler to use the gold linker.

Default

ld	The compiler uses the ld linker by default.

Description

This option tells the compiler to use a different linker instead of default linker (ld).

This option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

l

Tells the linker to search for a specified library when linking.

Syntax

Linux OS:

`-l`*`string`*

macOS:

`-l`*`string`*

Windows OS:

`None`

Arguments

string	Specifies the library (lib*string*) that the linker should search.

Default

OFF	The linker searches for standard libraries in standard directories.

Description

This option tells the linker to search for a specified library when linking.

When resolving references, the linker normally searches for libraries in several standard directories, in directories specified by the `L` option, then in the library specified by the `l` option.

The linker searches and processes libraries and object files in the order they are specified. So, you should specify this option following the last object file it applies to.

IDE Equivalent

None

Alternate Options

None

See Also

`L` compiler option

L

Tells the linker to search for libraries in a specified directory before searching the standard directories.

Syntax

Linux OS:

`-L`*`dir`*

macOS:

`-L`*`dir`*

Windows OS:

`None`

Arguments

dir	Is the name of the directory to search for libraries.

Default

OFF	The linker searches the standard directories for libraries.

Description

This option tells the linker to search for libraries in a specified directory before searching for them in the standard directories.

IDE Equivalent

None

Alternate Options

None

See Also

`l` compiler option

LD

Specifies that a program should be linked as a dynamic-link (DLL) library.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/LD`

`/LDd`

Arguments

None

Default

OFF	The program is not linked as a dynamic-link (DLL) library.

Description

This option specifies that a program should be linked as a dynamic-link (DLL) library instead of an executable (.exe) file. You can also specify `/LDd`, where *d* indicates a debug version.

IDE Equivalent

None

Alternate Options

None

link

Passes user-specified options directly to the linker at compile time.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/link`

Arguments

None

Default

OFF	No user-specified options are passed directly to the linker.

Description

This option passes user-specified options directly to the linker at compile time.

All options that appear following `/link` are passed directly to the linker.

IDE Equivalent

None

Alternate Options

None

See Also

`Xlinker` compiler option

MD

Tells the linker to search for unresolved references in a multithreaded, dynamic-link run-time library.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/MD`

`/MDd`

Arguments

None

Default

OFF	The linker searches for unresolved references in a multi-threaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a multithreaded, dynamic-link (DLL) run-time library. You can also specify `/MDd`, where `d` indicates a debug version.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Runtime Library**

Eclipse

Eclipse: None

Xcode

Xcode: None

Alternate Options

None

MT

Tells the linker to search for unresolved references in a multithreaded, static run-time library.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/MT`

`/MTd`

Arguments

None

Default

`/MT`	The linker searches for unresolved references in a multithreaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a multithreaded, static run-time library. You can also specify `/MTd`, where `d` indicates a debug version.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

IDE Equivalent

Visual Studio

Visual Studio: **Code Generation > Runtime Library**

Eclipse

Eclipse: None

Xcode

Xcode: None

Alternate Options

None

See Also

`Qvc` compiler option

no-libgcc

Prevents the linking of certain gcc-specific libraries.

Syntax

Linux OS:

`-no-libgcc`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF

Description

This option prevents the linking of certain gcc-specific libraries.

This option is not recommended for general use.

IDE Equivalent

None

Alternate Options

None

nodefaultlibs

Prevents the compiler from using standard libraries when linking.

Syntax

Linux OS:

`-nodefaultlibs`

macOS:

`-nodefaultlibs`

Windows OS:

None

Arguments

None

Default

OFF	The standard libraries are linked.

Description

This option prevents the compiler from using standard libraries when linking.On Linux* systems, it is provided for GNU compatibility.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Libraries > Use no system libraries**

OS X

Xcode: None

Alternate Options

None

See Also

nostdlib compiler option

no-intel-lib, Qno-intel-lib

Disables linking to specified Intel® libraries, or to all Intel® libraries.

Syntax

Linux OS:

`-no-intel-lib[=`*`library`*`]`

macOS:

`None`

Windows OS:

`/Qno-intel-lib[:library]`

Arguments

library	Indicates which Intel® library should *not* be linked. Possible values are:	
	`libirc`	Disables linking to the Intel® C/C++ library.
	`libimf`	Disables linking to the Intel® C++ Compiler Classic Math library. This value is only available for Linux*.
	`libsvml`	Disables linking to the Intel® Short Vector Math library.
	`libirng`	Disables linking to the Random Number Generator library.
	`libipgo`	Disables linking to the Profile-Guided Optimization library.

If you specify more than one *library*, they must be separated by commas.

If *library* is omitted, the compiler will not link to any of the Intel® libraries shown above.

Default

OFF	If this option is not specified, the compiler uses default heuristics for linking to libraries.

Description

This option disables linking to specified Intel® libraries, or to all Intel® libraries.

IDE Equivalent

None

Alternate Options

None

nostartfiles

Prevents the compiler from using standard startup files when linking.

Syntax

Linux OS:

`-nostartfiles`

macOS:

`-nostartfiles`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler uses standard startup files when linking.

Description

This option prevents the compiler from using standard startup files when linking.

IDE Equivalent

None

Alternate Options

None

See Also

`nostdlib` compiler option

nostdlib

Prevents the compiler from using standard libraries and startup files when linking.

Syntax

Linux OS:

`-nostdlib`

macOS:

`-nostdlib`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler uses standard startup files and standard libraries when linking.

Description

This option prevents the compiler from using standard libraries and startup files when linking. On Linux* systems, it is provided for GNU compatibility.

This option is not related to option `-stdlib`.

IDE Equivalent

None

Alternate Options

None

See Also

`nodefaultlibs` compiler option

`nostartfiles` compiler option

pie

Determines whether the compiler generates position-independent code that will be linked into an executable.

Syntax

Linux OS:

`-pie`

`-no-pie`

macOS:

`-pie`

`-no-pie`

Windows OS:

`None`

Arguments

None

Default

varies	On Linux* and on macOS versions less than 10.7, the default is -no-pie. On macOS 10.7 or greater, the default is -pie.

Description

This option determines whether the compiler generates position-independent code that will be linked into an executable. To enable generation of position-independent code that will be linked into an executable, specify `-pie`.

To disable generation of position-independent code that will be linked into an executable, specify `-no-pie`.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic` compiler option

pthread

Tells the compiler to use pthreads library for multithreading support.

Syntax

Linux OS:

`-pthread`

macOS:

`-pthread`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not use pthreads library for multithreading support.

Description

Tells the compiler to use pthreads library for multithreading support.

IDE Equivalent

None

Alternate Options

None

shared

Tells the compiler to produce a dynamic shared object instead of an executable.

Syntax

Linux OS:

`-shared`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option tells the compiler to produce a dynamic shared object (DSO) instead of an executable. This includes linking in all libraries dynamically and passing `-shared` to the linker.

You must specify option `fpic` for the compilation of each object file you want to include in the shared library.

IDE Equivalent

None

Alternate Options

None

See Also

`dynamiclib` compiler option
`fpic` compiler option
`Xlinker` compiler option

shared-intel

Causes Intel-provided libraries to be linked in dynamically.

Syntax

Linux OS:

`-shared-intel`

macOS:

`-shared-intel`

Windows OS:

None

Arguments

None

Default

OFF	Intel® libraries are linked in statically, with the exception of Intel's OpenMP* runtime support library, which is linked in dynamically unless you specify option `-qopenmp-link=static`.

Description

This option causes Intel-provided libraries to be linked in dynamically. It is the opposite of `-static-intel`.

This option is processed by the icc or icpc command that initiates linking, adding library names explicitly to the link command.

If you specify option `-mcmodel=medium` or `-mcmodel=large`, it sets option `-shared-intel`.

> **NOTE**
> On macOS systems, when you set "Intel Runtime Libraries" to "Dynamic", you must also set the DYLD_LIBRARY_PATH environment variable within Xcode* or an error will be displayed.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: None

Xcode

Xcode: **Runtime > Intel Runtime Libraries**

Alternate Options

None

See Also

`static-intel` compiler option
`qopenmp-link` compiler option

shared-libgcc

Links the GNU libgcc library dynamically.

Syntax

Linux OS:

`-shared-libgcc`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-shared-libgcc`	The compiler links the `libgcc` library dynamically.

Description

This option links the GNU `libgcc` library dynamically. It is the opposite of option `static-libgcc`.

This option is processed by the icc or icpc command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior of the `static` option, which causes all libraries to be linked statically.

IDE Equivalent

None

Alternate Options

None

See Also

static-libgcc compiler option

static

Prevents linking with shared libraries.

Syntax

Linux OS:

`-static`

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	The compiler links with shared libraries except as otherwise specified by `-static-intel` or its default.

Description

This option prevents linking with shared libraries. It causes the executable to link all libraries statically.

> **NOTE**
> This option does not cause static linking of libraries for which no static version is available, such as the OpenMP run-time libraries on Windows*. These libraries can only be linked dynamically.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: **Libraries > Link with static libraries**

Xcode

Xcode: None

Alternate Options

None

See Also

`static-intel` compiler option

static-intel

Causes Intel-provided libraries to be linked in statically.

Syntax

Linux OS:

`-static-intel`

macOS:

`-static-intel`

Windows OS:

`None`

Arguments

None

Default

ON	Intel® libraries are linked in statically, with the exception of Intel's OpenMP* runtime support library, which is linked in dynamically unless you specify option `-qopenmp-link=static`.

Description

This option causes Intel-provided libraries to be linked in statically with certain exceptions (see the Default above). It is the opposite of `-shared-intel`.

This option is processed by the icc or icpc command that initiates linking, adding library names explicitly to the link command.

If you specify option `-static-intel` while option `-mcmodel=medium` or `-mcmodel=large` is set, an error will be displayed.

If you specify option `-static-intel` and any of the Intel-provided libraries have no static version, a diagnostic will be displayed.

IDE Equivalent

Visual Studio

Visual Studio: None

Eclipse

Eclipse: None

Xcode

Xcode: **Runtime > Intel Runtime Libraries**

Alternate Options

None

See Also

`shared-intel` compiler option

`qopenmp-link` compiler option

static-libgcc

Links the GNU libgcc library statically.

Syntax

Linux OS:

`-static-libgcc`

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	The compiler links the GNU `libgcc` library dynamically.

Description

This option links the GNU `libgcc` library statically. It is the opposite of option `-shared-libgcc`.

This option is processed by the icc or icpc command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior, which causes the library to be linked dynamically.

> **NOTE**
> If you want to use traceback, you must also link to the static version of the `libgcc` library. This library enables printing of backtrace information.

IDE Equivalent

None

Alternate Options

None

See Also

`shared-libgcc` compiler option
`static-libstdc++` compiler option

static-libstdc++

Links the GNU libstdc++ library statically.

Syntax

Linux OS:

`-static-libstdc++`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler links the GNU `libstdc++` library dynamically.

Description

This option links the GNU `libstdc++` library statically.

This option is processed by the icc or icpc command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior, which causes the library to be linked dynamically.

IDE Equivalent

None

Alternate Options

None

See Also

`static-libgcc` compiler option

staticlib

Invokes the libtool command to generate static libraries.

Syntax

Linux OS:

`None`

macOS:

`-staticlib`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option invokes the `libtool` command to generate static libraries. This option is processed by the command that initiates linking, adding library names explicitly to the link command.

When passed this option, the compiler uses the `libtool` command to produce a static library instead of an executable when linking.

To build dynamic libraries, you should specify option `-dynamiclib` or `libtool -dynamic <objects>`.

IDE Equivalent

None

Alternate Options

None

See Also

dynamiclib compiler option

T

Tells the linker to read link commands from a file.

Syntax

Linux OS:

`-T`*`filename`*

macOS:

None

Windows OS:

None

Arguments

filename	Is the name of the file.

Default

OFF	The linker does not read link commands from a file.

Description

This option tells the linker to read link commands from a file.

IDE Equivalent

None

Alternate Options

None

u (Linux*)

Tells the compiler the specified symbol is undefined.

Syntax

Linux OS:

`-u` *`symbol`*

macOS:

`-u` *`symbol`*

Windows OS:

`None`

Arguments

None

Default

OFF	Standard rules are in effect for variables.

Description

This option tells the compiler the specified *symbol* is undefined.

IDE Equivalent

None

Alternate Options

None

v

Specifies that driver tool commands should be displayed and executed.

Syntax

Linux OS:

`-v [`*`filename`*`]`

macOS:

`-v [`*`filename`*`]`

Windows OS:

`None`

Arguments

filename	Is the name of a source file to be compiled. A space must appear before the file name.

Default

OFF	No tool commands are shown.

Description

This option specifies that driver tool commands should be displayed and executed.

If you use this option without specifying a source file name, the compiler displays only the version of the compiler.

IDE Equivalent

None

Alternate Options

None

See Also

`dryrun` compiler option

Wa

Passes options to the assembler for processing.

Syntax

Linux OS:

`-Wa,`*`option1`*`[,`*`option2`*`,...]`

macOS:

`-Wa,`*`option1`*`[,`*`option2`*`,...]`

Windows OS:

`None`

Arguments

option	Is an assembler option. This option is not processed by the driver and is directly passed to the assembler.

Default

OFF	No options are passed to the assembler.

Description

This option passes one or more options to the assembler for processing. If the assembler is not invoked, these options are ignored.

IDE Equivalent

None

Alternate Options

None

Wl

Passes options to the linker for processing.

Syntax

Linux OS:

`-Wl,`*`option1`*`[,`*`option2`*`,...]`

macOS:

`-Wl,`*`option1`*`[,`*`option2`*`,...]`

Windows OS:

`None`

Arguments

option	Is a linker option. This option is not processed by the driver and is directly passed to the linker.

Default

OFF	No options are passed to the linker.

Description

This option passes one or more options to the linker for processing. If the linker is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption,link,`*options*.

IDE Equivalent

None

Alternate Options

None

See Also

`Qoption` compiler option

Wp

Passes options to the preprocessor.

Syntax

Linux OS:

`-Wp,`*`option1`*`[,`*`option2`*`,...]`

macOS:

`-Wp,`*`option1`*`[,`*`option2`*`,...]`

Windows OS:

`None`

Arguments

option	Is a preprocessor option. This option is not processed by the driver and is directly passed to the preprocessor.

Default

OFF	No options are passed to the preprocessor.

Description

This option passes one or more options to the preprocessor. If the preprocessor is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption,`*cpp, options*.

IDE Equivalent

None

Alternate Options

None

See Also

`Qoption` compiler option

Xlinker

Passes a linker option directly to the linker.

Syntax

Linux OS:

`-Xlinker` *option*

macOS:

`-Xlinker` *option*

Windows OS:

`None`

Arguments

option	Is a linker option.

Default

OFF	No options are passed directly to the linker.

Description

This option passes a linker option directly to the linker. If `-Xlinker -shared` is specified, only `-shared` is passed to the linker and no special work is done to ensure proper linkage for generating a shared object. `-Xlinker` just takes whatever arguments are supplied and passes them directly to the linker.

If you want to pass compound options to the linker, for example "`-L $HOME/lib`", you must use the following method:

```
-Xlinker -L -Xlinker $HOME/lib
```

IDE Equivalent

Visual Studio: None

Eclipse: **Linker > Miscellaneous > Other Options**

Xcode: None

Alternate Options

None

See Also

`shared` compiler option

`link` compiler option

Zl

Causes library names to be omitted from the object file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Zl`

Arguments

None

Default

OFF	Default or specified library names are included in the object file.

Description

This option causes library names to be omitted from the object file.

IDE Equivalent

Windows

Visual Studio: **Advanced > Omit Default Library Names**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

Miscellaneous Options

This section contains descriptions for compiler options that do not pertain to a specific category.

bigobj

Increases the number of sections that an object file can contain.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/bigobj`

Arguments

None

Default

OFF	An object file can hold up to 65,536 (2**16) addressable sections.

Description

This option increases the number of sections that an object file can contain. It increases the address capacity to 4,294,967,296(2**32).

This option may be helpful for .obj files that can hold more sections, such as machine generated code or code that makes heavy use of template libraries.

IDE Equivalent

None

Alternate Options

None

dryrun

Specifies that driver tool commands should be shown but not executed.

Syntax

Linux OS:

`-dryrun`

macOS:

`-dryrun`

Windows OS:

`None`

Arguments

None

Default

OFF	No tool commands are shown, but they are executed.

Description

This option specifies that driver tool commands should be shown but not executed.

IDE Equivalent

None

Alternate Options

None

See Also

`v` compiler option

dumpmachine

Displays the target machine and operating system configuration.

Syntax

Linux OS:

`-dumpmachine`

macOS:

`-dumpmachine`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not display target machine or operating system information.

Description

This option displays the target machine and operating system configuration. No compilation is performed.

IDE Equivalent

None

Alternate Options

None

See Also

`dumpversion` compiler option

dumpversion

Displays the version number of the compiler.

Syntax

Linux OS:

`-dumpversion`

macOS:

`-dumpversion`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not display the compiler version number.

Description

This option displays the version number of the compiler. It does not compile your source files.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following command:

```
icc -dumpversion
```

If the above is specified when using version 19.1 of the compiler, the compiler displays "19.1".

See Also

`dumpmachine` compiler option

global-hoist, Qglobal-hoist

Enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source.

Syntax

Linux OS:

`-global-hoist`

`-no-global-hoist`

macOS:

`-global-hoist`

`-no-global-hoist`

Windows OS:

`/Qglobal-hoist`

`/Qglobal-hoist-`

Arguments

None

Default

`-global-hoist` or `/Qglobal-hoist`	Certain optimizations are enabled that can move memory loads.

Description

This option enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source. In most cases, these optimizations are safe and can improve performance.

The negative form of the option is useful for some applications, such as those that use shared or dynamically mapped memory, which can fail if a load is moved too early in the execution stream (for example, before the memory is mapped).

IDE Equivalent

None

Alternate Options

None

help

Displays all supported compiler options or supported compiler options within a specified category of options.

Syntax

Linux OS:

`-help[`*`category`*`]`

macOS:

`-help[`*`category`*`]`

Windows OS:

`/help[category]`

Arguments

category

Is a category or class of options to display. Possible values are:

`advanced`	Displays advanced optimization options that allow fine tuning of compilation or allow control over advanced features of the compiler.
`codegen`	Displays Code Generation options.
`compatibility`	Displays options affecting language compatibility.
`component`	Displays options for component control.
`data`	Displays options related to interpretation of data in programs or the storage of data.
`deprecated`	Displays options that have been deprecated.
`diagnostics`	Displays options that affect diagnostic messages displayed by the compiler.
`float`	Displays options that affect floating-point operations.
`help`	Displays all the available help categories.
`inline`	Displays options that affect inlining.
`ipo`	Displays Interprocedural Optimization (IPO) options
`language`	Displays options affecting the behavior of the compiler language features.
`link`	Displays linking or linker options.
`misc`	Displays miscellaneous options that do not fit within other categories.
`openmp`	Displays OpenMP and parallel processing options.
`opt`	Displays options that help you optimize code.
`output`	Displays options that provide control over compiler output.
`pgo`	Displays Profile Guided Optimization (PGO) options.
`preproc`	Displays options that affect preprocessing operations.
`reports`	Displays options for optimization reports.

Default

OFF	No list is displayed unless this compiler option is specified.

Description

This option displays all supported compiler options or supported compiler options within a specified category of options. If you specify *category*, it will display all available (supported) compiler options in the specified category.

On Linux* systems, this option can also be specified as `--help`.

IDE Equivalent

None

Alternate Options

Linux and macOS: None

Windows: `/?`

intel-freestanding

Lets you compile in the absence of a gcc environment.

Syntax

Linux OS:

`-intel-freestanding[=`*ver*`]`

macOS:

`None`

Windows OS:

`None`

Arguments

ver	Is a three-digit number that is used to determine the gcc version that the compiler should be compatible with for compilation. It also sets the corresponding GNUC macros. The number will be normalized to reflect the gcc compiler version numbering scheme. For example, if you specify 493, it indicates the compiler should be compatible with gcc version 4.9.3.

Default

OFF	The compiler uses default heuristics when choosing the gcc environment.

Description

This option lets you compile in the absence of a gcc environment. It disables any external compiler calls (such as calls to gcc) that the compiler driver normally performs by default.

This option also removes any default search locations for header and library files. So, for successful compilation and linking, you must provide these search locations.

This option does not affect ld, as, or cpp. They will be used for compilation as needed.

NOTE
This option does not imply option `-nostdinc -nostdlib`. If you want to assure a clean environment for compilation (including removal of Intel-specific header locations and libs), you should specify `-nostdinc` and/or `-nostdlib`.

NOTE
This option is supported for any Linux-target compiler, including a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

See Also

intel-freestanding-target-os compiler option
nostdlib compiler option
nostdinc compiler option, which is an alternate option for option X

intel-freestanding-target-os

Lets you specify the target operating system for compilation.

Syntax

Linux OS:

`-intel-freestanding-target-os=`*os*

macOS:

None

Windows OS:

None

Arguments

os	Is the target operating system for the Linux compiler. Currently, the only possible value is `linux`.

Default

OFF	The installed gcc determines the target operating system.

Description

This option lets you specify the target operating system for compilation. It sets option `-intel-freestanding`.

NOTE
This option is supported for any Linux-target compiler, including a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

See Also

intel-freestanding compiler option

MP-force

Disables the default heuristics used when compiler option /MP is specified. This lets you control the number of processes spawned.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/MP-force`

Arguments

None

Default

OFF	Default heuristics are used when option `/MP` is specified.

Description

This option disables the default heuristics used when compiler option `/MP:`*n* is specified. You must specify it when you specify option `/MP:`*n*.

Option `/MP:`*n* sets the maximum number of processes that can be used to compile large numbers of source files at the same time. However, default heuristics may cause the number of processes to be less than specified.

Option `/MP-force` ensures that *n* will be the maximum number of processes spawned regardless of other heuristics which may limit the number of processes.

IDE Equivalent

None

Alternate Options

None

See Also

`multiple-processes, MP` compiler option

multibyte-chars, Qmultibyte-chars

Determines whether multi-byte characters are supported.

Syntax

Linux OS:

`-multibyte-chars`

`-no-multibyte-chars`

macOS:

`-multibyte-chars`

`-no-multibyte-chars`

Windows OS:

`/Qmultibyte-chars`

`/Qmultibyte-chars-`

Arguments

None

Default

`-multibyte-chars` or `/Qmultibyte-chars`	Multi-byte characters are supported.

Description

This option determines whether multi-byte characters are supported.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **Language > Support Multibyte Characters in Source**

OS X

Xcode: **Language > Support Multibyte Characters in Source**

Alternate Options

None

multiple-processes, MP

Creates multiple processes that can be used to compile large numbers of source files at the same time.

Syntax

Linux OS:

`-multiple-processes[=`*n*`]`

macOS:

`-multiple-processes[=`*n*`]`

Windows OS:

`/MP[:`*n*`]`

Arguments

n	Is the maximum number of processes that the compiler should create.

Default

OFF	A single process is used to compile source files.

Description

This option creates multiple processes that can be used to compile large numbers of source files at the same time. It can improve performance by reducing the time it takes to compile source files on the command line.

This option causes the compiler to create one or more copies of itself, each in a separate process. These copies simultaneously compile the source files.

If *n* is not specified for this option, the default value is as follows:

- On Windows* systems, the value is based on the setting of the NUMBER_OF_PROCESSORS environment variable.
- On Linux* and macOS systems, the value is 2.

This option applies to compilations, but not to linking or link-time code generation.

To override default heuristics, specify option `/MP-force`. It ensures that *n* will be the maximum number of processes created regardless of other heuristics that may limit the number of processes.

IDE Equivalent

Windows

Visual Studio: **General > Multi-processor Compilation**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

`MP-force` compiler option

nologo

Tells the compiler to not display compiler version information.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/nologo`

Arguments

None

Default

OFF

Description

Tells the compiler to not display compiler version information.

IDE Equivalent

Windows

Visual Studio: **General > Suppress Startup Banner**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

print-sysroot

Prints the target sysroot directory that is used during compilation.

Syntax

Linux OS:

`-print-sysroot`

macOS:

`None`

Windows OS:

None

Arguments

None

Default

OFF	Nothing is printed.

Description

This option prints the target sysroot directory that is used during compilation.

This is the target sysroot directory that is specified in an environment file or in option `--sysroot`. This option is only effective if a target sysroot has been specified.

This option is provided for compatibility with gcc.

> **NOTE**
> Even though this option is not supported for a Windows-to-Windows native compiler, it is supported for a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

See Also

`sysroot` compiler option

save-temps, Qsave-temps

Tells the compiler to save intermediate files created during compilation.

Syntax

Linux OS:

`-save-temps`

`-no-save-temps`

macOS:

`-save-temps`

`-no-save-temps`

Windows OS:

`/Qsave-temps`

`/Qsave-temps-`

Arguments

None

Default

Linux* and macOS systems: `-no-save-temps` Windows* systems: .obj files are saved	On Linux and macOS systems, the compiler deletes intermediate files after compilation is completed. On Windows systems, the compiler saves only intermediate object files after compilation is completed.

Description

This option tells the compiler to save intermediate files created during compilation. The names of the files saved are based on the name of the source file; the files are saved in the current working directory.

If option `[Q]save-temps` is specified, the following occurs:

- The object .o file (Linux and macOS) or .obj file (Windows) is saved.
- The assembler .s file (Linux and macOS) or .asm file (Windows) is saved if you specified the `[Q]use-asm` option.

If `-no-save-temps` is specified on Linux or macOS systems, the following occurs:

- The .o file is put into `/tmp` and deleted after calling `ld`.
- The preprocessed file is not saved after it has been used by the compiler.

If `/Qsave-temps-` is specified on Windows systems, the following occurs:

- The .obj file is not saved after the linker step.
- The preprocessed file is not saved after it has been used by the compiler.

NOTE
This option only saves intermediate files that are normally created during compilation.

IDE Equivalent

None

Alternate Options

None

Example

If you compile program `my_foo.c` on a Linux or macOS system and you specify option `-save-temps` and option `-use-asm`, the compilation will produce files `my_foo.o` and `my_foo.s`.

If you compile program `my_foo.c` on a Windows system and you specify option `/Qsave-temps` and option `/Quse-asm`, the compilation will produce files `my_foo.o` and `my_foo.asm`.

showIncludes

Tells the compiler to display a list of the include files.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/showIncludes`

Arguments

None

Default

OFF	The compiler does not display a list of the include files.

Description

This option tells the compiler to display a list of the include files. Nested include files (files that are included from the files that you include) are also displayed.

IDE Equivalent

Windows

Visual Studio: **Advanced > Show Includes**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

sox

Tells the compiler to save the compilation options and version number in the executable file. It also lets you choose whether to include lists of certain functions.

Syntax

Linux OS:

`-sox[=`*`keyword`*`[,`*`keyword`*`]]`

`-no-sox`

macOS:

`None`

Windows OS:

None

Arguments

keyword	Is the function information to include. Possible values are:	
	`inline`	Includes a list of the functions that were inlined in each object.
	`profile`	Includes a list of the functions that were compiled with the `-prof-use` option and for which the .dpi file had profile information, and an indication for each as to whether the profile information was USED (matched) or IGNORED (mismatched).

Default

`-no-sox`	The compiler does not save these informational strings in the object file.

Description

This option tells the compiler to save the compilation options and version number in the executable file. It also lets you choose whether to include lists of certain functions. The information is embedded as a string in each object file or assembly output.

If you specify option `sox` with no *keyword*, the compiler saves the compiler options and version number used in the compilation of the objects that make up the executable.

When you specify this option, the size of the executable on disk is increased slightly. Each *keyword* you specify increases the size of the executable. When you link the object files into an executable file, the linker places each of the information strings into the header of the executable. It is then possible to use a tool, such as a strings utility, to determine what options were used to build the executable file.

IDE Equivalent

None

Alternate Options

None

Example

The following commands are equivalent:

```
-sox=profile -sox=inline
-sox=profile,inline
```

You can use the negative form of the option to disable and reset the option. For example:

```
-sox=profile -no-sox -sox=inline      ! This means -sox=inline
```

See Also

`prof-use, Qprof-use` compiler option

sysroot

Specifies the root directory where headers and libraries are located.

Syntax

Linux OS:

`--sysroot=`*`dir`*

macOS:

`None`

Windows OS:

`None`

Arguments

dir	Specifies the local directory that contains copies of target libraries in the corresponding subdirectories.

Default

Off	The compiler uses default settings to search for headers and libraries.

Description

This option specifies the root directory where headers and libraries are located.

For example, if the headers and libraries are normally located in `/usr/include` and `/usr/lib` respectively, `--sysroot=/mydir` will cause the compiler to search in `/mydir/usr/include` and `/mydir/usr/lib` for the headers and libraries.

This option is provided for compatibility with gcc.

> **NOTE**
> Even though this option is not supported for a Windows-to-Windows native compiler, it is supported for a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

Tc

Tells the compiler to process a file as a C source file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Tc`*`filename`*

Arguments

filename	Is the file name to be processed as a C source file.

Default

OFF	The compiler uses default rules for determining whether a file is a C source file.

Description

This option tells the compiler to process a file as a C source file.

IDE Equivalent

None

Alternate Options

None

See Also

`TC` compiler option
`Tp` compiler option

TC

Tells the compiler to process all source or unrecognized file types as C source files.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/TC`

Arguments

None

Default

OFF	The compiler uses default rules for determining whether a file is a C source file.

Description

This option tells the compiler to process all source or unrecognized file types as C source files.

IDE Equivalent

Windows

Visual Studio: **Advanced > Compile As**

Linux

Eclipse: None

OS X

Xcode: None

Alternate Options

None

See Also

`TP` compiler option

`Tc` compiler option

Tp

Tells the compiler to process a file as a C++ source file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Tp`*`filename`*

Arguments

filename	Is the file name to be processed as a C++ source file.

Default

OFF	The compiler uses default rules for determining whether a file is a C++ source file.

Description

This option tells the compiler to process a file as a C++ source file.

IDE Equivalent

None

Alternate Options

None

See Also

`TP` compiler option

`Tc` compiler option

V

Displays the compiler version information.

Syntax

Linux OS:

`-V`

macOS:

`-V`

Windows OS:

`/QV`

Arguments

None

Default

OFF	The compiler version information is not displayed.

Description

This option displays the startup banner, which contains the following compiler information:

- The name of the compiler and its applicable architecture
- The major and minor version of the compiler, the update number, and the package number(for example, Version 12.1.0.047)
- The specific build and build date (for example, Build <builddate>)
- The copyright date of the software

This option can be placed anywhere on the command line.

IDE Equivalent

Windows

Visual Studio: None

Linux

Eclipse: **General > Show Startup Banner**

OS X

Xcode: **General > Show Startup Banner**

Alternate Options

None

version

Tells the compiler to display GCC-style version information.

Syntax

Linux OS:

`--version`

macOS:

`--version`

Windows OS:

None

Arguments

None

Default

OFF

Description

Tells the compiler to display GCC-style version information.

IDE Equivalent

None

Alternate Options

None

watch

Tells the compiler to display certain information to the console output window.

Syntax

Linux OS:

`-watch[=`*keyword*`[, `*keyword*`...]]`

`-nowatch`

macOS:

`-watch[=`*keyword*`[, `*keyword*`...]]`

`-nowatch`

Windows OS:

`/watch[:`*keyword*`[, `*keyword*`...]]`

`/nowatch`

Arguments

keyword	Determines what information is displayed. Possible values are:	
	`none`	Disables `cmd` and `source`.

`[no]cmd`	Determines whether driver tool commands are displayed and executed.
`[no]source`	Determines whether the name of the file being compiled is displayed.
`all`	Enables `cmd` and `source`.

Default

`nowatch`	Pass information and source file names are not displayed to the console output window.

Description

Tells the compiler to display processing information (pass information and source file names) to the console output window.

Option `watch` *`keyword`*	**Description**
`none`	Tells the compiler to not display pass information and source file names to the console output window. This is the same as specifying `nowatch`.
`cmd`	Tells the compiler to display and execute driver tool commands.
`source`	Tells the compiler to display the name of the file being compiled.
`all`	Tells the compiler to display pass information and source file names to the console output window. This is the same as specifying `watch` with no *keyword*. For heterogeneous compilation, the tool commands for the host and the offload compilations will be displayed.

IDE Equivalent

None

Alternate Options

`watch cmd`	Linux and macOS: `-v` Windows: None

See Also

`v` compiler option

Deprecated and Removed Compiler Options

This topic lists deprecated and removed compiler options and suggests replacement options, if any are available.

For more information on compiler options, see the detailed descriptions of the individual option descriptions in this section.

Deprecated Options

Occasionally, compiler options are marked as "deprecated." Deprecated options are still supported in the current release, but are planned to be unsupported in future releases.

The following two tables list options that are currently deprecated.

Note that deprecated options are not limited to these lists.

Deprecated Linux* and macOSOptions	Suggested Replacement
axS	axSSE4.1
axT	Linux*: axSSSE3 macOS: axSSSE3
fmudflap	None; consider using the Pointer Checker options (such as option check pointers)
Kc++	x c++
march=pentiumii	None
march=pentiumiii	march=pentium3
mcpu	mtune
msse	Linux* only: mia32
prof-gen-sampling	None
prof-use-sampling	None
rcd	None
use-asm	None
wd	diag-disable
we	diag-error
wn	diag-error-limit
wo	diag-once id[,id,...]
wr	diag-remark
ww	diag-warning
xH	xSSE4.2
xS	xSSE4.1
xT	Linux*: xSSSE3 macOS: xSSSE3

Deprecated Windows* Options	Suggested Replacement
arch:SSE	arch:IA32
Fr	FR
Ge	Gs0
GX	EHsc
GZ	RTC1
H	None

Deprecated Windows* Options	Suggested Replacement
QaxS	QaxSSE4.1
QaxT	QaxSSSE3
QIfist	Qrcd
Qrcd	None
Qsox	None
Quse-asm	None
Qwd	Qdiag-disable
Qwe	Qdiag-error
Qwn	Qdiag-error-limit:<n>
Qwo	Qdiag-once
Qwr	Qdiag-remark
Qww	Qdiag-warning
QxH	QxSSE4.2
QxS	QxSSE4.1
QxT	QxSSSE3
Yd	Z7, Zi, or Zl
Ze	None
Zg	None

Removed Options

Some compiler options are no longer supported and have been removed. If you use one of these options, the compiler issues a warning, ignores the option, and then proceeds with compilation.

The following two tables list options that are no longer supported.

Note that removed options are not limited to these lists.

Removed Linux* and macOSOptions	Suggested Replacement
A-	undef
0f_check	None
alias-args	fargument-alias
axB	axSSE2
axH	axSSE4.2
axi	None

Removed Linux* and macOSOptions	Suggested Replacement
axK	No exact replacement; upgrade to msse2
axM	None
axN	Linux*: axSSE2 macOS: None
axP	Linux*: axSSE3 macOS: None
axW	msse2
c99	std=c99
check-uninit	check=uninit
create-pch	pch-create
cxxlib-gcc[=dir]	cxxlib[=dir]
cxxlib-icc	None
export	None
export-dir	None
F	P
falign-stack=mode	None; this option is only removed on macOS
fdiv_check	None
fms-dialect (macOS only)	None
fms-dialect=11 fms-dialect=10 fms-dialect=9	None
fp	fno-omit-frame-pointer
fpstkchk	fp-stack-check
func-groups	prof-func-groups
fvisibility=internal	fvisibility=hidden
fwritable-strings	None
gcc-version	No exact replacement; use gcc-name
guide-profile	None
i-dynamic	shared-intel
i-static	static-intel
inline-debug-info	debug inline-debug-info

Removed Linux* and macOSOptions	Suggested Replacement
ipo-obj (and ipo_obj)	None
ipp-link=static-thread	None
Knopic, -KNOPIC	fpic
Kpic, -KPIC	fpic
mp	fp-model
no-alias-args	fargument-noalias
no-c99	std=c89
no-cpprt	no-cxxlib
nobss-init	no-bss-init
norestrict	no-restrict
Ob	inline-level
openmp	qopenmp
openmp-lib	qopenmp-lib
openmp-lib legacy	None
openmp-link and qopenmp-link	None
openmpP	qopenmp
openmp-profile	None
openmp-report	qopt-report-phase=openmp
openmpS	qopenmp-stubs
openmp-simd	qopenmp-simd
openmp-stubs	qopenmp-stubs
openmp-task	qopenmp-task
openmp-threadprivate	qopenmp-threadprivate
opt-args-in-regs	qopt-args-in-regs
opt-assume-safe-padding	qopt-assume-safe-padding
opt-block-factor	qopt-block-factor
opt-calloc	qopt-calloc
opt-class-analysis	qopt-class-analysis
opt-dynamic-align	qopt-dynamic-align
opt-gather-scatter-unroll	None

Removed Linux* and macOSOptions	Suggested Replacement
opt-jump-tables	qopt-jump-tables
opt-malloc-options	qopt-malloc-options
opt-matmul	qopt-matmul
opt-mem-layout-trans	qopt-mem-layout-trans
opt-multi-version-aggressive	qopt-multi-version-aggressive
opt-prefetch	qopt-prefetch
opt-prefetch-distance	qopt-prefetch-distance
opt-ra-region-strategy	qopt-ra-region-strategy
opt-report	qopt-report
opt-report-embed	qopt-report-embed
opt-report-file	qopt-report-file
opt-report-filter	qopt-report-filter
opt-report-format	qopt-report-format
opt-report-help	qopt-report-help
opt-report-level	qopt-report
opt-report-per-object	qopt-report-per-object
opt-report-phase	qopt-report-phase
opt-report-routine	qopt-report-routine
opt-streaming-cache-evict	None
opt-streaming-stores	qopt-streaming-stores
opt-subscript-in-range	qopt-subscript-in-range
par-report	qopt-report-phase=par
prefetch	qopt-prefetch
prof-format-32	None
prof-genx	prof-gen=srcpos
profile-functions	None
profile-loops	None
profile-loops-report	None
qoffload	None
qoffload-arch	None

Removed Linux* and macOSOptions	Suggested Replacement
qoffload-attribute-target	None
qoffload-option	None
qopenmp-offload	None
qopenmp-report	qopt-report-phase=openmp
qopenmp-task	None
qp	p
rct	None
shared-libcxa	shared-libgcc
ssp	None
static-libcxa	static-libgcc
std=c9x	std=c99
syntax	fsyntax-only
tcheck	None
tpp1	None
tpp2	mtune=itanium2
tpp5	None
tpp6	None
tpp7	mtune=pentium4
tprofile	None
use-pch	pch-use
vec-report	qopt-report-phase=vec
Wpragma-once	None
xB	xSSE2
xi	None
xK	No exact replacement; upgrade to msse2
xM	None
xN	Linux*: xSSE2 macOS: None
xO	-msse3
xP	Linux*: xSSE3 macOS: None

Removed Linux* and macOSOptions	Suggested Replacement
xSSE3_ATOM	xATOM_SSSE3
xSSSE3_ATOM	xATOM_SSSE3
xW	msse2

Removed Windows* Options	Suggested Replacement
debug:parallel	None
G5	None
G6 (or GB)	None
G7	None
Gf	GF
ML[d]	Upgrade to MT[d]
Og	O1, O2, or O3
Op	fp:precise
QA-	u
QaxB	QaxSSE2
QaxH	QaxSSE4.2
Qaxi	None
QaxK	Upgrade to arch:SSE2
QaxM	None
QaxN	QaxSSE2
QaxP	QaxSSE3
QaxW	arch:SSE2
Qc99	Qstd=c99
Qfpstkchk	Qfp-stack-check
Qguide-profile	None
Qgpu-arch:ivybridge	None
QI0f	None
QIfdiv	None
Qinline-debug-info	debug:inline-debug-info
Qipo-obj (and Qipo_obj)	None

Removed Windows* Options	Suggested Replacement
`Qipp-link:static-thread`	None
`Qmspp`	None
`Qopenmp-lib:legacy`	None
`Qopenmp-link`	None
`Qopenmp-offload`	None
`Qopenmp-profile`	None
`Qopenmp-report`	`Qopt-report-phase:openmp`
`Qopenmp-task`	None
`Qopt-report-level`	`Qopt-report`
`Qpar-report`	`Qopt-report-phase:par`
`Qprefetch`	`Qopt-prefetch`
`Qprof-format-32`	None
`Qprof-gen-sampling`	None
`Qprof-genx`	`Qprof-gen=srcpos`
`Qprofile-functions`	None
`Qprofile-loops`	None
`Qprofile-loops-report`	None
`Qrct`	None
`Qssp`	None
`Qtprofile`	None
`Qtcheck`	None

`Qvc11`
`Qvc10`
`Qvc9` and earlier
	None

`Qvec-report`	`Qopt-report-phase:vec`
`QxB`	`QxSSE2`
`Qxi`	None
`QxK`	Upgrade to `arch:SSE2`
`QxM`	None
`QxN`	`QxSSE2`

Removed Windows* Options	Suggested Replacement
`QxO`	`arch:SSE3`
`QxP`	`QxSSE3`
`QxSSE3_ATOM`	`QxATOM_SSSE3`
`QxSSSE3_ATOM`	`QxATOM_SSSE3`
`QxW`	`arch:SSE2`
`YX`	None
`Zd`	`debug:minimal`

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Display Option Information

To display a list of all available compiler options, specify option `help` on the command line.

To display functional groupings of compiler options, specify a functional category for option `help`. For example, to display a list of options that affect diagnostic messages, enter one of the following commands:

Linux and macOS

```
-help diagnostics
```

Windows

```
/help diagnostics
```

For details on other categories you can specify, see help.

Alternate Compiler Options

This topic lists alternate names for compiler options and show the primary option name. Some of the alternate option names are deprecated and may be removed in future releases.

For more information on compiler options, see the detailed descriptions of the individual, primary options.

Some of these options are deprecated. For more information, see Deprecated and Removed Options.

Linux

Alternate Linux* and macOS Options	Primary Option Name
Code Generation:	
`-fp`	-fomit-frame-pointer
`-mcpu`	-mtune
Advanced Optimizations:	
`-fstrict-aliasing`	-ansi-alias

Alternate Linux* and macOS Options	Primary Option Name
`-funroll-loops`	-unroll
Profile Guided Optimization (PGO):	
`-qp`	-p (Linux* only)
OpenMP* and Parallel Processing Options:	
`-fopenmp`	-qopenmp
Output, Debug, Precompiled Header (PCH):	
`-fvar-tracking`	-debug variable-locations
`-fvar-tracking-assignments`	-debug semantic-stepping
Linking or Linker:	
`-i-dynamic`	-shared-intel
`-i-static`	-static-intel

Windows

Alternate Windows* Options	Primary Option Name
OpenMP* and Parallel Processing Options:	
`/openmp`	/Qopenmp
Floating Point:	
`/QIfist`	/Qrcd

Portability and GCC-compatible Warning Options

This section discusses portability options and GCC-compatible warning options.

Portability Options

A challenge in porting applications from one compiler to another is making sure that there is support for the compiler options you use to build your application. The Intel® compiler supports many of the options that are valid on other compilers you may be using.

The first table lists compiler options that are supported by the Intel® compiler and the GCC Compiler. Following this table, you will see information about GCC-compatible warning options.

The second table lists compiler options that are supported by the Intel® compiler and the Microsoft C++ Compiler .

Options that are unique to either compiler are not listed in this topic.

Linux

This table lists compiler options that are supported by both the Intel® compiler and the GCC Compiler.

`-A`
`-ansi`
`-B`

-C

-c

-D

-dD

-dM

-dN

-E

-fargument-noalias

-fargument-noalias-global

-fcf-protection

-fdata-sections

-ffunction-sections

-fmudflap (this is a deprecated option)

-f[no-]builtin

-f[no-]common

-f[no-]freestanding

-f[no-]gnu-keywords

-fno-implicit-inline-templates

-fno-implicit-templates

-f[no-]inline

-f[no-]inline-functions

-f[no-]math-errno

-f[no-]operator-names

-f[no-]stack-protector

-f[no-]unsigned-bitfields

-fpack-struct

-fpermissive

-fPIC

-fpic

-freg-struct-return

-fshort-enums

-fsyntax-only

-ftemplate-depth

-ftls-model=global-dynamic

-ftls-model=initial-exec

-ftls-model=local-dynamic

-ftls-model=local-exec

-funroll-loops

-funsigned-char

-fverbose-asm

-fvisibility=default

-fvisibility=hidden

-fvisibility=internal

-fvisibility=protected

-H

-help

-I

-idirafter

-imacros

-iprefix

-iwithprefix

-iwithprefixbefore

-l

-L

-M

-malign-double

-march

-mcpu

-MD

-MF

-MG

-MM

-MMD

-m[no-]ieee-fp

-MP

-MQ

-msse

-msse2

-msse3

-MT

-mtune

-nodefaultlibs

-nostartfiles

-nostdinc

-nostdinc++

-nostdlib

-o

-O

-O0

-O1

-O2

-O3

-Os

-p

-P

-S

-shared

-static

-std

-trigraphs

-U

-u

-v

-V

-w

-Wall

-Werror

-Winline

-W[no-]cast-qual

-W[no-]comment
-W[no-]comments
-W[no-]deprecated
-W[no-]fatal-errors
-W[no-]format-security
-W[no-]main
-W[no-]missing-declarations
-W[no-]missing-prototypes
-W[no-]overflow
-W[no-]overloaded-virtual
-W[no-]pointer-arith
-W[no-]return-type
-W[no-]strict-prototypes
-W[no-]trigraphs
-W[no-]uninitialized
-W[no-]unknown-pragmas
-W[no-]unused-function
-W[no-]unused-variable
-X
-x assembler-with-cpp
-x c
-x c++
-Xlinker

The Intel® compiler recognizes many GCC-compatible warning options, but many are not documented.

In general, if a GCC-compatible option is accepted by the compiler, but not documented, the implementation of the option is the same as described in the GCC documentation.

To find the GCC documentation about GCC warning options, you can do any of the following:

- Enter the command:

```
man gcc
```

- Check the GCC website.
- Search the web for "gcc warning options".

Windows

This table lists compiler options that are supported by both the Intel® compiler and the Microsoft C++ Compiler.

For complete details about these options, such as the possible values for <n> when it appears below, see the Microsoft Visual Studio C++ documentation.

```
/arch

/C

/c

/D<name>{=|#}<text>

/E

/EH{a|s|c|r}

/EP

/F<n>

/Fa[file]

/FA[{c|s|cs}]

/FC

/Fe<file>

/FI<file>

/Fm[<file>]

/Fo<file>

/fp:<model>

/Fp<file>

/FR[<file>]

/Fr[<file>]

/GA

/Gd

/Ge

/GF

/Gh

/GH

/Gr

/GR[-]

/GS[-]

/Gs[<n>]

/GT

/Gy[-]

/Gz
```

/GZ

/H<n>

/help

/I<dir>

/J

/LD

/LDd

/link

/MD

/MDd

/MP

/MT

/MTd

/nologo

/O1

/O2

/Ob<n>

/Od

/Oi[-]

/Os

/Ot

/Ox

/Oy[-]

/P

/QIfist[-]

/RTC{1|c|s|u}

/showIncludes

/TC

/Tc<source file>

/TP

/Tp<source file>

/u

/U<name>

```
/V<string>
/vd<n>
/vmb
/vmg
/vmm
/vms
/vmv
/w
/W<n>
/Wall
/wd<n>
/we<n>
/WL
/Wp64
/WX
/X
/Y-
/Yc[<file>]
/Yu[<file>]
/Z7
/Za
/Zc:<arg1>[, <arg2>]
/Ze
/Zg
/Zi
/ZI
/Zl
/Zp[<n>]
/Zs
```

Floating-Point Operations

This section contains information about floating-point operations, including IEEE floating-point operations, and it provides guidelines that can help you improve the performance of floating-point applications.

Programming Tradeoffs in Floating-Point Applications

In general, the programming objectives for floating-point applications fall into the following categories:

- **Accuracy:** The application produces results that are close to the correct result.
- **Reproducibility and portability:** The application produces consistent results across different runs, different sets of build options, different compilers, different platforms, and different architectures.
- **Performance:** The application produces fast, efficient code.

Based on the goal of an application, you will need to make tradeoffs among these objectives. For example, if you are developing a 3D graphics engine, performance may be the most important factor to consider, with reproducibility and accuracy as secondary concerns.

The compiler provides several options that allow you to tune your applications based on specific objectives. Broadly speaking, there are the floating-point specific options, such as the `-fp-model` (Linux* and macOS) or `/fp` (Windows*) option, and the fast-but-low-accuracy options, such as the `[Q]imf-max-error` option. The compiler optimizes and generates code differently when you specify these different compiler options. Select appropriate compiler options by carefully balancing your programming objectives and making tradeoffs among these objectives. Some of these options may influence the choice of math routines that are invoked.

Many routines in the *libirc*, *libm*, and *svml* library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.

Use Floating-Point Options

Take the following code as an example:

```
float t0, t1, t2;
 ...
t0=t1+t2+4.0f+0.1f;
```

If you specify the `-fp-model extended` (Linux and macOS) or `/fp:extended` (Windows) option in favor of accuracy, the compiler generates the following assembly code:

```
fld       DWORD PTR _t1
fadd      DWORD PTR _t2
fadd      DWORD PTR _Cnst4.0
fadd      DWORD PTR _Cnst0.1
fstp      DWORD PTR _t0
```

This code maximizes accuracy because it utilizes the highest mantissa precision available on the target platform. The code performance might suffer when managing the x87 stack, and it might yield results that cannot be reproduced on other platforms that do not have an equivalent extended precision type.

If you specify the `-fp-model source` (Linux and macOS) or `/fp:source` (Windows) option in favor of reproducibility and portability, the compiler generates the following assembly code:

```
movss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
addss     xmm0, DWORD PTR _Cnst4.0
addss     xmm0, DWORD PTR _Cnst0.1
movss     DWORD PTR _t0, xmm0
```

This code maximizes portability by preserving the original order of the computation, and by using the IEEE single-precision type for all computations. It is not as accurate as the previous implementation, because the intermediate rounding error is greater compared to extended precision. It is not the highest performance implementation, because it does not take advantage of the opportunity to pre-compute `4.0f + 0.1f`.

If you specify the `-fp-model fast` (Linux and macOS) or `/fp:fast` (Windows) option in favor of performance, the compiler generates the following assembly code:

```
movss     xmm0, DWORD PTR _Cnst4.1
addss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
movss     DWORD PTR _t0, xmm0
```

This code maximizes performance using Intel® Streaming SIMD Extensions (Intel® SSE) instructions and pre-computing `4.0f + 0.1f`. It is not as accurate as the first implementation, due to the greater intermediate rounding error. It does not provide reproducible results like the second implementation, because it must reorder the addition to pre-compute `4.0f + 0.1f`. All compilers, on all platforms, at all optimization levels do not reorder the addition in the same way.

For many other applications, the considerations may be more complicated.

Use Fast-But-Low-Accuracy Options

The fast-but-low-accuracy options provide an easy way to control the accuracy of mathematical functions and utilize performance/accuracy tradeoffs offered by the Intel® oneAPI Math Kernel Library (oneMKL). You can specify accuracy, via a command line interface, for all math functions or a selected set of math functions at the level more precise than low, medium, or high.

You specify the accuracy requirements as a set of function attributes that the compiler uses for selecting an appropriate function implementation in the math libraries. Examples using the attribute, `max-error`, are presented here. For example, use the following option to specify the relative error of two ULPs for all single, double, long double, and quad precision functions:

```
-fimf-max-error=2
```

To specify twelve bits of accuracy for a `sin` function, use:

```
-fimf-accuracy-bits=12:sin
```

To specify relative error of ten ULPs for a `sin` function, and four ULPs for other math functions called in the source file you are compiling, use:

```
-fimf-max-error=10:sin-fimf-max-error=4
```

On Windows systems, the compiler defines the default value for the `max-error` attribute depending on the `/fp` option and `/Qfast-transcendentals` settings. In `/fp:fast` mode, or if fast but less accurate math functions are explicitly enabled by `/Qfast-transcendentals-`, then the compiler sets a `max-error=`*4.0* for the call. Otherwise, it sets a `max-error=`*0.6*.

Dispatching of Math Routines

The compiler optimizes calls to routines from the *libm* and *svml* libraries into direct CPU-specific calls, when the compilation configuration specifies the target CPU where the code is tuned, and if the set of instructions available for the code compilation is not narrower than the set of instructions available in the tuning target CPU.

For example:

- The code containing calls to the *exp()* library function and compiled with `-mtune=`*`corei7-avx`* (specifies tuning target CPU that supports Intel® Advanced Vector Extensions (Intel® AVX)) and `-QxCORE-AVX2/-march=`*`core-avx2`* (specifies Intel® Advanced Vector Extensions 2 (Intel® AVX2) instructions set) call the *exp()* routine that is optimized for processors with Intel® AVX support. This code provides the best performance for these processors.
- The same code, compiled with `-mtune=`*`core-avx2`* and `-QxAVX/-march=`*`corei7-avx`*, calls a library dispatch routine that picks the optimal CPU specific version of the *exp()* routine in runtime. Dispatching cannot be avoided because the instruction set does not allow the use of Intel® AVX2. Dynamic dispatching provides the best performance with the Intel® AVX2 CPU.

In the second example, if some portions of code extend the available instructions set by means of the `_allow_cpu_features()` or the `_may_i_use_cpu_feature()` intrinsic, then the compiler might produce direct calls to Intel® AVX2 specific versions of *exp()*.

The dispatching optimization applies to the *exp()* routine, and to the other math routines with CPU specific implementations in the libraries. The dispatching optimization can be disabled using the `-fimf-force-dynamic-target` (or `Qimf-force-dynamic-target`) option. This option specifies a list of math routines that are improved with a dynamic dispatcher.

See Also

Using -fp-model(/fp) Options
fimf-max-error, Qimf-max-error compiler option

Floating-point Optimizations

Application performance is an important goal of the Intel® C++ Compiler, even at default optimization levels. A number of optimizations involve transformations that might affect the floating-point behavior of the application, such as evaluation of constant expressions at compile time, hoisting invariant expressions out of loops, or changes in the order of evaluation of expressions. These optimizations usually help the compiler to produce the most efficient code possible. However, the optimizations might be contrary to the floating-point requirements of the application.

Some optimizations are not consistent with strict interpretation of the ANSI or ISO standards for C and C++. Such optimizations can cause differences in rounding and small variations in floating-point results that may be more or less accurate than the ANSI-conformant result.

The Intel® C++ Compiler provides the `-fp-model` (Linux* and macOS) or `/fp` (Windows*) option, which allows you to control the optimizations performed when you build an application. The option allows you to specify the compiler rules for:

- **Value safety:** Whether the compiler may perform transformations that could affect the result. For example, in the SAFE mode, the compiler won't transform `x/x` to 1.0 because the value of `x` at runtime might be a zero or a NaN . The UNSAFE mode is the default.
- **Floating-point expression evaluation:** How the compiler should handle the rounding of intermediate expressions. For example, when double precision is specified, the compiler interprets the statement `t0=4.0f+0.1f+t1+t2;` as `t0=(float)(4.1+(double)t1+(double)t2);`
- **Floating-point contractions:** Whether the compiler should generate fused multiply-add (FMA) instructions on processors that support them. When enabled, the compiler may generate FMA instructions for combining multiply and add operations; when disabled, the compiler must generate separate multiply and add instructions with intermediate rounding.
- **Floating-point environment access:** Whether the compiler must account for the possibility that the program might access the floating-point environment, either by changing the default floating-point control settings or by reading the floating-point status flags. This is disabled by default. You can use the `-fp-model:strict` (Linux* and macOS) `/fp:strict` (Windows*) option to enable it.
- **Precise floating-point exceptions:** Whether the compiler should account for the possibility that floating-point operations might produce an exception. This is disabled by default. You can use `-fp-model:strict` (Linux* and macOS) or `/fp:strict` (Windows*); or `-fp-model:except` (Linux* and macOS) or `/fp:except` (Windows*) to enable it.

Consider the following example:

```
double a=1.5; int x=0; ...
  __try {
    int t0=a; //raises inexact
    x=1;
    a*=2;
  } __except(1) {
  printf("SEH Exception: x=%d\n", x);
}
```

Without precise floating-point exceptions, the result is `SEH Exception: x=1`; with precision floating-point exceptions, the result is `SEH Exception: x=0`.

The following table describes the impact of different keywords of the option on compiler rules and optimizations:

Keyword	Value Safety	Floating-Point Expression Evaluation	Floating-Point Contractions	Floating-Point Environment Access	Precise Floating-Point Exceptions

`precise`
`source`
`double`
`extended`
	Safe	Varies
		Source
		Double
		Extended
			Yes	No	No

`strict`	Safe	Varies	No	Yes	Yes
`consistent`	Safe	Varies	No	No	No
`fast=1` (default)	Unsafe	Unknown	Yes	No	No
`fast=2`	Very unsafe	Unknown	Yes	No	No
`except` `except-`	Unaffected Unaffected	Unaffected Unaffected	Unaffected Unaffected	Unaffected Unaffected	Yes No

NOTE
It is illegal to specify the `except` keyword in an unsafe safety mode.

Based on the objectives of an application, you can choose to use different sets of compiler options and keywords to enable or disable certain optimizations, so that you can get the desired result.

See Also

Using -fp-model (/fp) Option

Use the -fp-model, /fp Option

The `-fp-model` (Linux and macOS) or `/fp` (Windows) option allows you to control the optimizations on floating-point data. You can use this option to tune the performance, level of accuracy, or result consistency for floating-point applications across platforms and optimization levels.

For applications that do not require support for denormalized numbers, the `-fp-model` or `/fp` option can be combined with the `[Q]ftz` option to flush denormalized results to zero. This flush can improve runtime performance on processors based on all Intel® architectures.

You can use keywords to specify the semantics to be used. The keywords specified for this option may influence the choice of math routines that are invoked. Many routines in the *libirc*, *libm*, and *libsvml* libraries are more highly optimized for Intel microprocessors than for non-Intel microprocessors. Possible values of the keywords are as follows:

Keyword	Description
`precise`	Enables value-safe optimizations on floating-point data.
`fast[=1|2]`	Enables more aggressive optimizations on floating-point data.
`consistent`	Enables consistent, reproducible results for different optimization levels or between different processors of the same architecture. This setting is equivalent to the use of the following options: Windows: `/fp:precise /Qfma- /Qimf-arch-consistency:true` Linux and macOS: `-fp-model precise -no-fma -fimf-arch-consistency=true`
`strict`	Enables `precise` and `except` , disables contractions, and enables pragma `stdc fenv_access`.
`source`	Rounds intermediate results to source-defined precision and enables value-safe optimizations.
`double`	Rounds intermediate results to 53-bit (double) precision and enables value-safe optimizations.
`extended`	Rounds intermediate results to 64-bit (extended) precision and enables value-safe optimizations.
`[no-]except` (Linux and macOS) or `except[-]` (Windows)	Determines whether strict floating-point exception semantics are used.

The default value of the option is `-fp-model fast=1` or `/fp:fast=1`, which means that the compiler uses more aggressive optimizations on floating-point calculations.

NOTE
Using the default option keyword `-fp-model fast` or `/fp:fast`, you may get significant differences in your result depending on whether the compiler uses x87 or Intel® Streaming SIMD Extensions (Intel® SSE)/Intel® Advanced Vector Extensions (Intel® AVX) instructions to implement floating-point operations. Results are more consistent when the other option keywords are used.

Several examples are provided to illustrate the usage of the keywords. These examples show:

- A small example of source code.

NOTE
The same source code is considered in all the included examples.

- The semantics that are used to interpret floating-point calculations in the source code.
- One or more possible ways the compiler may interpret the source code.

NOTE
There are several ways that the compiler may interpret the code; we show just some of these possibilities.

-fp-model fast or /fp:fast

Example source code:

Example

```
float t0, t1, t2;
...
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions may be performed in any order.
- Intermediate expressions may use `single`, `double`, or `extended double` precision.
- The constant addition may be pre-computed, assuming the default rounding mode.

Using these semantics, some possible ways the compiler may interpret the original code are given below:

Example

```
float t0, t1, t2;
  ...
t0 = (float)((double)t1 + (double)t2) + 4.1f;
```

```
float t0, t1, t2;
  ...
t0 = (t1 + t2) + 4.1f;
```

```
float t0, t1, t2;
  ...
t0 = (t1 + 4.1f) + t2;
```

-fp-model extended or /fp:extended

This setting is equivalent to `-fp-model precise` on Linux operating systems based on the IA-32 architecture.

Example source code:

```
float t0, t1, t2;
  ...
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order
- Intermediate expressions use extended double precision
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, a possible way the compiler may interpret the original code is shown below:

```
float t0, t1, t2;
  ...
t0 = (float)(((long double)4.1 + (long double)t1) + (long double)t2);
```

-fp-model source or /fp:source

This setting is equivalent to `-fp-model precise` or `/fp:precise` on systems based on the Intel® 64 architecture.

Source code example

```
float t0, t1, t2;
...
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order.
- Intermediate expressions use the precision specified in the source code, that is, single precision.
- The constant addition may be pre-computed, assuming the default rounding mode.

Using these semantics, a possible way the compiler may interpret the original code is shown below:

Example

```
float t0, t1, t2;
  ...
t0 = ((4.1f + t1) + t2);
```

-fp-model double or /fp:double

This setting is equivalent to `-fp-model precise` or `/fp:precise` on Windows systems based on the IA-32 architecture.

Example source code:

```
float t0, t1, t2;
  ...
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order
- Intermediate expressions use double precision
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, a possible way the compiler may interpret the original code is shown below:

```
float t0, t1, t2;
  ...
t0 = (float)(((double)4.1 + (double)t1) + (double)t
```

-fp-model strict or /fp:strict

Source code example

```
float t0, t1, t2;
  ...
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order
- Expression evaluation matches expression evaluation under keyword `precise`.
- The constant addition is not pre-computed because there is no way to tell what rounding mode will be active when the program runs.

Using these semantics, a possible way the compiler may interpret the original code is shown below:

Example

```
float t0, t1, t2;
  ...
t0 = (float)((((long double)4.0f + (long double)0.1f) + (long double)t1) + (long double)t2);
```

See Also

`fp-model, fp` compiler option

Denormal Numbers

A normalized number is a number for which both the exponent (including bias) and the most significant bit of the mantissa are non-zero. For such numbers, all the bits of the mantissa contribute to the precision of the representation.

The smallest normalized single-precision floating-point number greater than zero is about 1.1754943^{-38}. Smaller numbers are possible, but those numbers must be represented with a zero exponent and a mantissa whose leading bit(s) are zero, which leads to a loss of precision. These numbers are called denormalized numbers or denormals(newer specifications refer to these as subnormal numbers).

Denormal computations use hardware and/or operating system resources to handle denormals; these can cost hundreds of clock cycles. Denormal computations take much longer to calculate than normal computations.

There are several ways to avoid denormals and increase the performance of your application:

- Scale the values into the normalized range.
- Use a higher precision data type with a larger range.
- Flush denormals to zero.

See Also

Reducing Impact of Denormal Exceptions

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Institute of Electrical and Electronics Engineers, Inc*. (IEEE) web site for information about the current floating-point standards and recommendations

Floating-Point Environment

The floating-point environment is a collection of registers that control the behavior of the floating-point machine instructions and indicate the current floating-point status. The floating-point environment can include rounding mode controls, exception masks, flush-to-zero (`FTZ`) controls, exception status flags, and other floating-point related features.

For example, bit 15 of the `MXCSR` register enables the flush-to-zero mode, which controls the masked response to an single-instruction multiple-data (SIMD) floating-point underflow condition.

The floating-point environment affects most floating-point operations; therefore, correct configuration to meet your specific needs is important. For example, the exception mask bits define which exceptional conditions will be raised as exceptions by the processor. In general, the default floating-point environment is set by the operating system. You don't need to configure the floating-point environment unless the default floating-point environment does not suit your needs.

There are several methods available if you want to modify the default floating-point environment. For example, you can use inline assembly, compiler built-in functions, library functions, or command line options.

Changing the default floating-point environment affects runtime results only. This does not affect any calculations that are pre-computed at compile time.

If strict reproducibility and consistency are important do not change the floating point environment without also using either `-fp-model strict` (Linux* or macOS) or `/fp:strict` (Windows*) option or `pragma fenv_access`.

See Also

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Set the FTZ and DAZ Flags

In Intel® processors, the flush-to-zero (`FTZ`) and denormals-are-zero (`DAZ`) flags in the `MXCSR` register are used to control floating-point calculations. Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instructions, including scalar and vector instructions, benefit from enabling the `FTZ` and `DAZ` flags. Floating-point computations using the Intel® SSE and Intel® AVX instructions are accelerated when the `FTZ` and `DAZ` flags are enabled. This improves the application's performance.

Use the `[Q]ftz` option to flush denormal results to zero when the application is in the gradual underflow mode. This option may improve performance if the denormal values are not critical to the application's behavior. The `[Q]ftz` option, when applied to the main program, sets the `FTZ` and the `DAZ` hardware flags. The negative forms of the `[Q]ftz` option (`-no-ftz` for Linux* and macOS*, and `/Qftz-` for Windows*) leave the flags as they are.

The following table describes how the compiler processes denormal values based on the status of the `FTZ` and `DAZ` flags:

Flag	When set to ON, the compiler...	When set to OFF, the compiler...	Supported on
`FTZ`	...sets denormal results from floating-point calculations to zero.	...does not change the denormal results.	Intel® 64 and some IA-32 architectures
`DAZ`	...treats denormal values used as input to floating-point instructions as zero.	...does not change the denormal instruction inputs.	Intel® 64 and some IA-32 architectures

- `FTZ` and `DAZ` are not supported on all IA-32 architectures. The `FTZ` flag is supported only on IA-32 architectures that support Intel® SSE instructions.
- On systems based on the IA-32 and Intel® 64 architectures, `FTZ` only applies to Intel® SSE and Intel® AVX instructions. If the application generates denormals using x87 instructions, `FTZ` does not apply.
- `DAZ` and `FTZ` flags are not compatible with the IEEE 754 standard, and should only be enabled when compliance to the IEEE standard is not required.

Options for `[Q]ftz` are performance options. Setting these options does not guarantee that all denormals in a program are flushed to zero. They only cause denormals generated at run-time to be flushed to zero.

On Intel® 64 and IA-32 systems, the compiler, by default, inserts code into the main routine to set the `FTZ` and `DAZ` flags. When the `[Q]ftz` option is used on IA-32 systems with the option `-msse2` or `/arch:sse2`, the compiler inserts code to that conditionally sets the `FTZ`/`DAZ` flags based on a run-time processor check. Using the negative form of `[Q]ftz` prevents the compiler from inserting any code that sets `FTZ` or `DAZ` flags.

When the `[Q]ftz` option is used in combination with an Intel® SSE-enabling option on systems based on the IA-32 architecture (for example, `-msse2` or `/arch:sse2`), the compiler inserts code in the main routine to set `FTZ` and `DAZ`. When the option `[Q]ftz` is used without an Intel® SSE-enabling option, the compiler inserts code that conditionally sets `FTZ` or `DAZ` based on a run-time processor check. The negative form of `[Q]ftz` prevents the compiler from inserting any code that might set `FTZ` or `DAZ`.

The `[Q]ftz` option only has an effect when the main program is being compiled. It sets the `FTZ`/`DAZ` mode for the process. The initial thread, and any subsequently created threads, operate in the `FTZ`/`DAZ` mode.

On systems based on Intel® 64 and IA-32 architectures, every optimization option `O` level, except `O0`, sets `[Q]ftz`.

If this option produces undesirable results of the numerical behavior of the program, turn the `FTZ`/`DAZ` mode off by using the negative form of `[Q]ftz` in the command line while still benefitting from the `O3` optimizations.

Manually set the `FTZ` flags with the following macros:

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)
```

Manually set the `DAZ` flags with the following macros:

```
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON)
```

The prototypes for these macros are in `xmmintrin.h` (`FTZ`) and `pmmintrin.h` (`DAZ`).

See Also

ftz, Qftz compiler option

Checking the Floating-point Stack State

On systems based on the IA-32 architecture, when an application calls a function that returns a floating-point value, the returned floating-point value is supposed to be on the top of the floating-point stack. If the return value is not used, the compiler must pop the value off of the floating-point stack in order to keep the floating-point stack in the correct state.

On systems based on Intel® 64 architecture, floating-point values are usually returned in the `xmm0` register. The floating-point stack is used only when the return value is a long double on Linux* and macOS systems.

If the application calls a function without defining or incorrectly defining the function's prototype, the compiler cannot determine if the function must return a floating-point value. Consequently, the return value is not popped off the floating-point stack if it is not used. This can cause the floating-point stack to overflow.

The overflow of the stack results in two undesirable situations:

- A `NaN` value gets involved in the floating-point calculations
- The program results become unpredictable; the point where the program starts making errors can be arbitrarily far away from the point of the actual error.

For systems based on the IA-32 and Intel® 64 architectures, the [Q]fp-stack-check option checks whether a program makes a correct call to a function that should return a floating-point value. If an incorrect call is detected, the option places a code that marks the incorrect call in the program. The [Q]fp-stack-check option marks the incorrect call and makes it easy to find the error.

> **NOTE**
> The [Q]fp-stack-check option causes significant code generation after every function/subroutine call to ensure that the floating-point stack is maintained in the correct state. Therefore, using this option slows down the program being compiled. Use the option only as a debugging aid to find floating point stack underflow/overflow problems, which can be otherwise hard to find.

See Also

fp-stack-check, Qfp-stack-check compiler option

Tuning Performance

This section describes several programming guidelines that can help you improve the performance of floating-point applications, including:

- Handling Floating-point Array Operations in a Loop Body
- Reducing the Impact of Denormal Exceptions
- Avoiding Mixed Data Type Arithmetic Expressions

- Using Efficient Data Types

Handling Floating-point Array Operations in a Loop Body

Following the guidelines below will help auto-vectorization of the loop.

- Statements within the loop body may contain float or double operations (typically on arrays). The following arithmetic operations are supported: addition, subtraction, multiplication, division, negation, square root, MAX, MIN, and mathematical functions such as SIN and COS.
- Writing to a single-precision scalar/array and a double scalar/array within the same loop decreases the chance of auto-vectorization due to the differences in the vector length (that is, the number of elements in the vector register) between float and double types. If auto-vectorization fails, try to avoid using mixed data types.

> **NOTE**
> The special `__m64`, `__m128`, and `__m256` datatypes are not vectorizable. The loop body cannot contain any function calls. Use of the Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) intrinsics (for example, `mm_add_ps`) is not allowed.

Reducing the Impact of Denormal Exceptions

Denormalized floating-point values are those that are too small to be represented in the normal manner; that is, the mantissa cannot be left-justified. Denormal values require hardware or operating system interventions to handle the computation, so floating-point computations that result in denormal values may have an adverse impact on performance.

There are several ways to handle denormals to increase the performance of your application:

- Scale the values into the normalized range
- Use a higher precision data type with a larger range
- Flush denormals to zero

For example, you can translate them to normalized numbers by multiplying them using a large scalar number, doing the remaining computations in the normal space, then scaling back down to the denormal range. Consider using this method when the small denormal values benefit the program design.

Consider using a higher precision data type with a larger range; for example, by converting variables declared as `float` to be declared as `double`. Understand that making the change can potentially slow down your program. Storage requirements will increase, which will increase the amount of time for loading and storing data from memory. Higher precision data types can also decrease the potential throughput of Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) operations.

If you change the type declaration of a variable, you might also need to change associated library calls, unless these are generic; ; for example, `cos()` instead of `cosf()`.. Another strategy that might result in increased performance is to increase the amount of precision of intermediate values using the `-fp-model [double|extended]` option. However, this strategy might not eliminate all denormal exceptions, so you must experiment with the performance of your application. You should verify that the gain in performance from eliminating denormals is greater than the overhead of using a data type with higher precision and greater dynamic range.

In many cases, denormal numbers can be treated safely as zero without adverse effects on program results. Depending on the target architecture, use flush-to-zero (`FTZ`) options.

IA-32 and Intel® 64 Architectures

IA-32 and Intel® 64 architectures take advantage of the `FTZ` (flush-to-zero) and `DAZ` (denormals-are-zero) capabilities of Intel® Streaming SIMD Extensions (Intel® SSE) instructions.

By default, the Intel® C++ Compiler inserts code into the main routine to enable `FTZ` and `DAZ` at optimization levels higher than `O0`. To enable `FTZ` and `DAZ` at `O0`, compile the source file containing main() PROGRAM using compiler option `[Q]ftz`. When the `[Q]ftz` option is used on IA-32-based systems with the option `-mia32` (Linux*) or `/arch:IA32` (Windows*), the compiler inserts code to conditionally enable `FTZ` and `DAZ` flags based on a run-time processor check. IA-32 is not available on macOS*.

NOTE
After using flush-to-zero, ensure that your program still gives correct results when treating denormal values as zero.

Avoiding Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (float, double, or long double) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that `I` and `J` are both int variables, expressing a constant number (2.0) as an integer value (2) eliminates the need to convert the data. The following examples demonstrate inefficient and efficient code.

Inefficient code:

```
int I, J;
  I = J / 2.0
;
```

Efficient code:

```
int I, J;
  I = J / 2;
```

Using Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- char
- short
- int
- long
- long long
- float
- double
- long double

NOTE
In an arithmetic expression, you should avoid mixing integer and floating-point data.

You can use integer data types (*int*, *int long*, etc.) in loops to improve floating point performance. Convert the data type to integer data types, process the data, then convert the data to the old type.

See Also

Programming Guidelines for Vectorization
Setting the FTZ and DAZ Flags

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

IEEE Floating-point Operations

Understanding the IEEE Standard for Floating-point Arithmetic, IEEE 754-2008

This version of the compiler uses a close approximation to the IEEE Standard for Floating-point Arithmetic, version IEEE 754-2008, unless otherwise stated. This standard is common to many microcomputer-based systems due to the availability of fast processors that implement the required characteristics.

This section outlines the characteristics of the IEEE 754-2008 standard and its implementation in the compiler. Except as noted, the description refers to both the IEEE 754-2008 standard and the compiler implementation.

Floating-point Formats

This IEEE 754-2008 standard specifies formats and methods for Floating-point representation in computer systems, and recommends formats for data interchange. The exception conditions are defined, and the standard handling of these conditions is specified below. The binary counterpart Floating-point exception functions are described in ISO C99. The decimal Floating-point exception functions are defined in the `fenv.h` header file. The compiler supports decimal floating point types in C and C++. The decimal floating point formats are defined in the IEEE 754-2008 standard.

In C, these decimal floating types are supported:

- `_Decimal32`
- `_Decimal64`
- `_Decimal128`

In C++ for Windows and Linux, these decimal classes are supported:

- `decimal32`
- `decimal64`
- `decimal128`

NOTE To use this feature in C++ on Linux, GCC 4.5 or later is required.

The decimal Floating-point is not supported in C++ for macOS.

To ensure correct decimal Floating-point behavior, you must define `__STDC_WANT_DEC_FP__` before any standard headers are included. This is required for the declaration of decimal macros and library functions in order to ensure correct decimal Floating-point results at run-time.

Example: Linux

```
#include <iostream>
#define __STDC_WANT_DEC_FP__
#include <decimal/decimal>
typedef std::decimal::decimal32 _Decimal32;
typedef std::decimal::decimal64 _Decimal64;
typedef std::decimal::decimal128 _Decimal128;
#include <dfp754.h>

using namespace std;
using namespace std::decimal;

int main() {
```

```
    std::decimal::decimal32 d = 4.7df;
    std::cout << decimal_to_long_double(d) << std::endl;
    return 0;
}
```

Example: Windows

```
#include <iostream>
#define __STDC_WANT_DEC_FP__
#include <decimal>
#include <dfp754.h>

using namespace std;
using namespace std::decimal;

int main() {
    std::decimal::decimal32 d = 4.7df;
    std::cout << decimal_to_long_double(d) << std::endl;
    return 0;
}
```

Functions to Check Decimal Floating-point Status

Use these Floating-point exception functions to detect exceptions that occur during decimal Floating-point arithmetic:

Floating-point Functions

Function	Brief Description
`fe_dec_feclearexcept()`	Clears the supported Floating-point exceptions.
`fe_dec_fegetexceptflag`	Stores an implementation-defined representation of the states of the Floating-point status flags.
`fe_dec_feraiseexcept`	Raises the supported Floating-point exceptions.
`fe_dec_fesetexceptflag`	Sets the Floating-point status flags.
`fe_dec_fetestexcept()`	Determines which of a specified subset of the floating point exception flags are currently set.

Special Values

The following list provides a brief description of the special values that the Intel® C++ Compiler supports.

- **Signed Zero:** The sign of zero is the same as the sign of a nonzero number. Comparisons consider +0 to be equal to -0. A signed zero is useful in certain numerical analysis algorithms, but in most applications the sign of zero is invisible.
- **Denormalized Numbers:** Denormalized numbers (denormals) fill the gap between the smallest positive and the smallest negative normalized number, otherwise only (+/-) 0 occurs in the interval. Denormalized numbers extend the range of computable results by allowing for gradual underflow.

  Systems based on the IA-32 architecture support a Denormal Operand status flag. When this is set, at least one of the input operands to a Floating-point operation is a denormal. The Underflow status flag is set when a number loses precision and becomes a denormal.
- **Signed Infinity:** Infinities are the result of arithmetic in the limiting case of operands with arbitrarily large magnitude. They provide a way to continue when an overflow occurs. The sign of an infinity is simply the sign you obtain for a finite number in the same operation as the finite number approaches an infinite value.

By retrieving the status flags, you can differentiate between an infinity that results from an overflow and one that results from division by zero. The compiler treats infinity as signed by default. The output value of infinity is +Infinity or -Infinity.

- **Not a Number:** Not a Number (NaN) may result from an invalid operation. For example, `0/0` and `SQRT(-1)` result in NaN. In general, an operation involving a NaN produces another NaN. Because the fraction of a NaN is unspecified, there are many possible NaNs

  The compiler treats all NaNs identically, but there are two classes of NaNs:

  - Signaling NaNs: Have an initial mantissa bit of 0. They usually raise an invalid exception when used in an operation.
  - Quiet NaNs: Have an initial mantissa bit of 1.

  The floating-point hardware usually converts a signaling NaN into a quiet NaN during computational operations. An invalid exception is raised and the resulting Floating-point value is a quiet NaN.

Attributes

Attributes are a way to provide additional information about a declaration to the compiler. The C+11 attribute syntax is consistent with the C2x standard.

Use Attributes

The compiler supports three ways to add attributes to your program:

- **Gnu Syntax**

```
__attribute__((attribute_name(arguments)))
```

- **Microsoft Syntax**

```
__declspec(attribute_name(argument))
```

- **C++11 Standardized Attribute Syntax** (part of the C++11 language standard)

```
[[attribute_name(arguments)]]
```

```
[[attribute-namespace :: attribute_name(arguments)]]
```

Some attributes are available for both Intel® microprocessors and non-Intel microprocessors but they may perform additional optimizations for Intel® microprocessors than they perform for non-Intel microprocessors. Refer to the individual attribute name for a detailed description.

align

Directs the compiler to align the variable to a specified boundary and a specified offset.

Syntax

Windows* OS:

`__declspec(align(n[,off]))`

Linux* OS:

`__attribute__((aligned(n[,off])))`

`__attribute__((align(n[,off])))`

For portability on Linux* OS, you should use the syntax form `__attribute__((aligned(n[, off])))`. This form is compatible with the GNU compiler.

Arguments

n	Specifies the alignment. The compiler will align the variable to an *n*-byte boundary.
off	Optional. Specifies the offset. If this argument is omitted, the value is 0.

Description

This keyword directs the compiler to align the variable to an *n*-byte boundary with offset *off* within each *n*-byte boundary. The address of the variable is `address mod n=off`.

> **NOTE**
> If you require 8-byte alignment, we recommend you specify 16 for *n*, instead of 8. When 8 is used, the compiler interprets the value as a suggestion and you may not get the requested 8-byte alignment, depending on various heuristics.

align_value

Provides the ability to add a pointer alignment value to a pointer typedef declaration.

Syntax

Windows* OS:

```
__declspec(align_value(alignment))
```

Linux* OS:

```
__attribute__((align_value(alignment)))
```

Arguments

alignment	Specifies the alignment (8, 16, 32, 64, 128, 256,...) for what the pointer points to.

Description

This keyword can be added to a pointer typedef declaration to specify the alignment value of pointers declared for that pointer type.

This indicates to the compiler that the data referenced by the designated pointer is aligned by the indicated value, and the compiler can generate code based on that assumption. If this attribute is used incorrectly, and the data is not aligned to the designated value, the behavior is undefined.

avoid_false_share

Provides the ability to pad and/or align the defined variable such that it will not be subject to false cache line sharing with any other variable.

Syntax

Windows* OS:

```
__declspec(avoid_false_share(identifier))variable definition
```

Linux* OS:

`__attribute__((avoid_false_share(identifier)))variable definition`

Arguments

identifier	Specifies the string that will be used to identify one or more variables. Variables with the same identifier do not need protection from false sharing.
variable definition	Specifies the variable to be padded or aligned.

Description

This keyword indicates to the compiler that it should allocate the *variable* through padding and/or alignment such that it will not share the cache line with other variables unless they share the same *identifier*. This keyword must occur on a *variable definition* in function, global, or namespace scope. It is not permitted on a non-static class member or on a function argument.

If you specify an *identifier*, the *variable definition* does not need to be protected from false sharing with other variables that are similarly declared with the same identifier. If the *variable definition* is in function scope, the scope of the *identifier* is the current function. If the *variable definition* is in namespace or global scope, the scope of the *identifier* is the current compilation unit.

This keyword is supported for scalars and arrays and is not supported for structure fields, function arguments, functions, and references.

code_align

Specifies the byte alignment for a routine.

Syntax

Windows* OS:

`__declspec(code_align(n))`

Linux* OS, macOS:

`__attribute__((code_align(n)))`

Arguments

n	Optional. A positive integer indicating the number of bytes for the minimum desired alignment boundary. Its value must be a power of 2, between 1 and 4096, such as 1, 2, 4, 8, and so on. If you specify 1 for *n*, no alignment is performed. If you do not specify *n*, the default alignment is 16 bytes.

Description

This keyword must be placed on the routine to be aligned.

If anything inside the routine requires specific alignment *k*, the final routine alignment will be *max(n,k)*.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Product and Performance Information
Notice revision #20201201

See Also

cpu_dispatch, cpu_specific attribute
Processor Targeting

concurrency_safe

Guides the compiler to parallelize more loops and straight-line code.

Syntax

Windows* OS:

`__declspec(concurrency_safe(clause))`

Linux* OS:

`__attribute__((concurrency_safe(clause)))`

Arguments

clause	Is one of the following: *cost(cycles)*: Specifies the execution cycles of the annotated function for the compiler to perform parallelization profitability analysis while compiling its enclosing loops or blocks. The value of `cycles` is a 2-byte unsigned integer (unsigned short); its maximal value is 2^16-1. If the cycle count is greater than 2^16-1, you should use *profitable*. *profitable*: Specifies that the loops or blocks that contain calls to the annotated function are profitable to parallelize.

Description

This keyword indicates to the compiler that there are no incorrect side-effects and no illegal (or improperly synchronized) memory access interferences among multiple invocations of the annotated function or between an invocation of this annotated function and other statements in the program, if they are executed concurrently.

For every function that is marked with this keyword, you must ensure that its side effects (if any) are acceptable (or expected), and the memory access interferences are properly synchronized.

const

Indicates that a function has no effect other than returning a value and that it uses only its arguments to generate that return value.

Syntax

Windows* OS:

`__declspec(const)`

Linux* OS:

`__attribute__((const))`

Arguments

None

Description

This keyword is equivalent to the gcc* attribute `const` and applies to function declarations.

cpu_dispatch, cpu_specific

Provides the ability to write one or more versions of a function that execute only on a list of targeted processors (cpu_dispatch). Provides the ability to declare that a version of a function is targeted at particular type(s) of processors (cpu_specific).

Syntax

Windows* OS:

`__declspec(cpu_dispatch(`*`cpuid, cpuid, ...`*`))`

`__declspec(cpu_specific(`*`cpuid`*`))`

Linux* OS:

`__attribute__((cpu_dispatch(`*`cpuid, cpuid, ...`*`)))`

`__attribute__((cpu_specific(`*`cpuid`*`)))`

Arguments

cpuid	Possible values are: `atom`: Intel® Atom™ processors with Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) `atom_sse4_2`: Intel® Atom™ processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) `atom_sse4_2_movbe`: Intel® Atom™ processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) with MOVBE instructions enabled `broadwell`: This is a synonym for `core_5th_gen_avx` `core_2nd_gen_avx`: 2nd generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions (Intel® AVX) `core_3rd_gen_avx`: 3rd generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions (Intel® AVX) including the RDRND instruction `core_4th_gen_avx`: 4th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDRND instruction

`core_4th_gen_avx_tsx`: 4th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDRND instruction, and support for Intel® Transactional Synchronization Extensions (Intel® TSX)

`core_5th_gen_avx`: 5th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDSEED and Multi-Precision Add-Carry Instruction Extensions (ADX) instructions

`core_5th_gen_avx_tsx`: 5th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDSEED and Multi-Precision Add-Carry Instruction Extensions (ADX) instructions, and support for Intel® Transactional Synchronization Extensions (Intel® TSX)

`core_aes_pclmulqdq`: Intel® Core™ processors with support for Advanced Encryption Standard (AES) instructions and carry-less multiplication instruction

`core_i7_sse4_2`: Intel® Core™ i7 processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) instructions

`generic`: Other Intel processors for IA-32 or Intel® 64 architecture or compatible processors not provided by Intel Corporation

`haswell`: This is a synonym for `core_4th_gen_avx`

`pentium`: Intel® Pentium® processor

`pentium_4`: Intel® Pentium® 4 processors

`pentium_4_sse3`: Intel® Pentium® 4 processor with Intel® Streaming SIMD Extensions 3 (Intel® SSE3) instructions, Intel® Core™ Duo processors, Intel® Core™ Solo processors

`pentium_ii`: Intel® Pentium® II processors

`pentium_iii`: Intel® Pentium® III processors

`pentium_iii_no_xmm_regs`: Intel® Pentium® III processors with no XMM registers

`pentium_m`: Intel® Pentium® M processors

`pentium_mmx`: Intel® Pentium® processors with MMX™ technology

`pentium_pro`: Intel® Pentium® Pro processors

Description

Use the cpu_dispatch keyword to provide a list of targeted processors, along with an empty function body/ function stub.

Use the cpu_specific keyword to declare each function version targeted at particular type(s) of processors

These feature are available only for Intel processors based on IA-32 or Intel® 64 architecture. They are not available for non-Intel processors. Applications built using the manual processor dispatch feature may be more highly optimized for Intel processors than for non-Intel processors.

See Also

Processor Targeting

mpx

Directs the compiler to pass Intel® Memory Protection Extensions (Intel® MPX) bounds information along with any pointer-typed parameters.

Syntax

Windows* OS:

`__declspec(mpx)`

Arguments

None

Description

When a function declared with this keyword is called, any pointer-typed parameters passed to the function will also have Intel® MPX bounds information passed. If the called function returns a pointer-typed object, the compiler will expect the function to return Intel® MPX bounds information along with the pointer object. Similarly, if this keyword is applied to a function definition, the function will expect the caller to pass Intel® MPX bounds information along with any pointer-type parameters. If the function returns a pointer-typed object, Intel® MPX bounds information will be returned with the object.

> **NOTE**
> The usage of this attribute is intended for Windows code that contains hand-written Intel® MPX enhancements based on Intel® MPX inline assembly or calls to Intel® MPX intrinsics, and where the user does not wish to enable automatic Intel® MPX code generation.

target

Specifies a target for called functions or variables.

Syntax

Windows* OS:

`__declspec(target(`*`target-name`*`))`

Linux* OS:

`__attribute__((target(`*`target-name`*`)))`

Arguments

target-name	Specifies the target name. Possible values are: • arch=corei7 • arch=core2 • arch=atom • mmx • sse • sse2 • sse3 • ssse3

- `sse4`
- `sse4a`
- `sse4.1`
- `sse4.2`
- `popcnt`
- `aes`
- `pclmul`
- `avx`
- `avx2`
- `avx512f`

Description

This keyword specifies that the called function or variable is also available on the target. Only functions or variables marked with this attribute are available on the target, and only these functions can be called on the target.

vector

Provides the ability to vectorize user-defined functions and loops.

Syntax

Windows* OS:

`__declspec(vector(`*`clauses`*`))`

Linux* OS:

`__attribute__((vector(`*`clauses`*`)))`

Arguments

`clauses`	Is one of the following: `processor clause`, in the form *processor(cpuid)*. This clause creates a vector version of the function for the given target processor (*cpuid*). See cpu_dispatch, cpu_specific for a list of supported values. The default processor is determined by the implicit or explicit process- or architecture-specific flag in the compiler command line. `vector length clause`, in the form *vectorlength(n)*, where *n* is a vectorlength (vl) and must be an integer with the value 2, 4, 8, or 16. This clause tells the compiler that each routine invocation at the call site should execute the computation equivalent to *n* times the scalar function execution. `linear clause`, in the form *linear(param1:step1 [, param2:step2] …)*, where *param* is a scalar variable and *step* is a compile-time integer constant expression. This clause tells the compiler that for each consecutive invocation of the routine in a serial execution, the value of *param1* is incremented by *step1*, *param2* is incremented by *step2*, and so on. If more than one step is specified for a particular variable, a compile-time error occurs. Multiple linear clauses are merged as a union.

	`uniform clause`, in the form *uniform(param [, param,]…)*, where param is a formal parameter of the specified function. This clause tells the compiler that the values of the specified arguments can be broadcast to all iterations as a performance optimization. `mask clause`, in the form *[no]mask*. This clause tells the compiler to generate a masked vector version of the routine.

Description

This keyword combines with the `map` operation at the call site to provide the data parallel semantics. When multiple instances of the vector declaration are invoked in a parallel context, the execution order among them is not sequenced.

vector_variant

Specifies a vector variant function that corresponds to its original C/C++ scalar function. This vector variant function can be invoked under vector context at call sites.

Syntax

Windows* OS:

`__declspec(vector_variant(`*`clauses`*`))`

Linux* OS:

`__attribute__((vector_variant(`*`clauses`*`)))`

Arguments

`clauses`	Is the following: `implements` clause, in the form *implements (<function declarator>) [, <simd-clauses>])*, where *function declarator* is the original scalar function, and *simd-clauses* is one or more of the clauses allowed for the vector attribute. The *simd-clauses* are optional.

Description

This attribute provides a means for programmers to describe the association between the vector variant function and its corresponding scalar function. The compiler will use the vector variant to replace the scalar call for a vectorized loop.

The following are restrictions for this attribute:

- A vector variant function can have only one `vector_variant` annotation.
- A vector variant annotation can have only one `implements` clause.
- A vector variant annotation applies to only one vector variant function, which must not have both `mask` and `nomask` clauses specified. It can be specified with either `mask` or `nomask`; the default is `nomask`.
- A vector variant function should have the `__regcall` attribute.

If the user-defined vector variant function is a variant with `mask`, the `mask` argument should be the last argument.

Example

The following shows an example of a vector variant function:

```
#include <immintrin.h>
__declspec(noinline)
float MyAdd(float* a, int b) { return *a + b; }
__declspec(vector_variant(implements(MyAdd(float *a, int b)),
                          linear(a), vectorlength(8),
                          nomask, processor(core_2nd_gen_avx)))
__m256 __regcall MyAddVec(float* v_a, __m128i v_b, __m128i v_b2) {
  __m256i t96 = _mm256_castsi128_si256(v_b);
  __m256i tmp = _mm256_insertf128_si256(t96, v_b2, 1);
  __m256  t95 = _mm256_cvtepi32_ps(tmp);
  return _mm256_add_ps(*((__m256*)v_a), t95);
}
float x[2000], y[2000];
float foo(float y[]) {
#pragma omp simd
  for (int k=0; k< 2000; k++) {
    x[k] = MyAdd(&y[k], k);
  }
  return x[0] + x[1999];
```

If the return value contains more than one register, the following technique can be used for the correct definition of the function:

```
#include <immintrin.h>

typedef struct {
 __m256d r1;
 __m256d r2;
} __m256dx2;

__declspec(noinline)
double MyAdd(double* a, int b) { return *a + b; }

__declspec(vector_variant(implements(MyAdd(double *a, int b)),
                          linear(a), vectorlength(8),
                          nomask, processor(core_2nd_gen_avx)))
__m256dx2 __regcall MyAddVec(double* v_a, __m128i v_b, __m128i v_b2) {
  __m256d t1 = _mm256_cvtepi32_pd(v_b);
  __m256d t2 = _mm256_cvtepi32_pd(v_b2);
  __m256dx2 ret;
  ret.r1 = _mm256_mul_pd(t1,*((__m256d*)v_a));
  ret.r2 = _mm256_mul_pd(t2,*(((__m256d*)v_a)+1));
  return ret;
}

__declspec(align(32)) double x[2000], y[2000];
double foo(double* y) {
#pragma omp simd
  for (int k=0; k< 2000; k++) {
    x[k] = MyAdd(y, k);
    y++;
  }
  return x[0] + x[1999];
}
```

See Also

simd pragma
vector attribute

Intrinsics

This intrinsics section provides an introduction and information on Intel specific intrinsics. The Intel® Intrinsics Guide provides detailed information and a lookup tool for viewing the available Intel intrinsics.

The following is some general information:

- Intrinsics are assembly-coded functions that let you use C++ function calls and variables in place of assembly instructions.
- Intrinsics can be used only on the host.
- Intrinsics are expanded inline eliminating function call overhead. Providing the same benefit as using inline assembly, intrinsics improve code readability, assist instruction scheduling, and help reduce debugging.
- Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C and C++ languages.

> **NOTE**
> When developing and debugging your program with the Intel® C++ Compiler Classic, compile your sources with `-D__INTEL_COMPILER_USE_INTRINSIC_PROTOTYPES` to take advantage of improved compile-time checking of the intrinsics functions. When done be sure to remove this option as it significantly increases compile time.

Availability of Intrinsics on Intel Processors

Not all Intel® processors support all intrinsics. For information on which intrinsics are supported on Intel® processors, visit the Product Specification, Processors page. The Processor Spec Finder tool links directly to all processor documentation and the datasheets list the features, including intrinsics, supported by each processor.

Details about Intrinsics

All instructions use the following features:

- Registers
- Data Types

Registers

Intel® processors provide special register sets for different instructions.

- Intel® MMX™ instructions use eight 64-bit registers (`mm0` to `mm7`) which are aliased on the floating-point stack registers.
- Intel® Streaming SIMD Extensions (Intel® SSE) and the Advanced Encryption Standard (AES) instructions use eight 128-bit registers (`xmm0` to `xmm7`).
- Intel® Advanced Vector Extensions (Intel® AVX) instructions use 256-bit registers which are extensions of the 128-bit SIMD registers.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions use 512-bit registers.

Because each of these registers can hold more than one data element, the processor can process more than one data element simultaneously. This processing capability is also known as single-instruction multiple data processing (SIMD).

For each computational and data manipulation instruction in the new extension sets, there is a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

Data Types

Intrinsic functions use new C data types as operands, representing the new registers that are used as the operands to these intrinsic functions.

The following table details for which instructions each of the new data types are available. A 'Yes' indicates that the data type is available for that group of intrinsics; an 'NA' indicates that the data type is not available for that group of intrinsics.

Data Types --> Technology	__m64	__m128	__m128d	__m128i	__m256	__m256d	__m256i	__m512	__m512d	__m512i
Intel® MMX™ Technology Intrinsics	Yes	NA	NA	NA	NA	NA	NA	NA	NA	NA
Intel® Streaming SIMD Extensions Intrinsics	Yes	Yes	NA	NA	NA	NA	NA	NA	NA	NA
Intel® Streaming SIMD Extensions 2 Intrinsics	Yes	Yes	Yes	Yes	NA	NA	NA	NA	NA	NA
Intel® Streaming SIMD Extensions	Yes	Yes	Yes	Yes	NA	NA	NA	NA	NA	NA

Data Types --> Technology	__m64	__m128	__m128d	__m128i	__m256	__m256d	__m256i	__m512	__m512d	__m512i
3 Intrinsics										
Advanced Encryption Standard Intrinsics + Carry-less Multiplication Intrinsic	Yes	Yes	Yes	Yes	NA	NA	NA	NA	NA	NA
Half-Float Intrinsics	Yes	Yes	Yes	Yes	NA	NA	NA	NA	NA	NA
Intel® Advanced Vector Extensions Intrinsics	Yes	Yes	Yes	Yes	Yes	Yes	Yes	NA	NA	NA
Intel® Advanced Vector Extensions 512 Intrinsics	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

__m64 Data Type

The `__m64` data type is used to represent the contents of an MMX register, which is the register that is used by the MMX™ technology intrinsics. The `__m64` data type can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value.

__m128 Data Types

The __m128 data type is used to represent the contents of a SSE register used by the Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics.

Conventionally, the __m128 data type can hold four 32-bit floating-point values, while the __m128d data type can hold two 64-bit floating-point values, and the __m128i data type can hold sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integer values.

The compiler aligns __m128d and _m128i local and global data to 16-byte boundaries on the stack. To align integer, float, or double arrays, use the __declspec(align) statement.

Accessing __m128i Data

To access 8-bit data on IA-32 and Intel® 64 architecture-based systems, use the mm_extract intrinsics as follows:

```
#define _mm_extract_epi8(x, imm) \
((((imm) & 0x1) == 0) ?   \
_mm_extract_epi16((x), (imm) >> 1) & 0xff : \
_mm_extract_epi16(_mm_srli_epi16((x), 8), (imm) >> 1))
```

To access 16-bit data, use:

```
int _mm_extract_epi16(__m128i a, int imm)
```

To access 32-bit data, use:

```
#define _mm_extract_epi32(x, imm) \
_mm_cvtsi128_si32(_mm_srli_si128((x), 4 * (imm)))
```

To access 64-bit data (Intel® 64 architecture only), use:

```
#define _mm_extract_epi64(x, imm) \
_mm_cvtsi128_si64(_mm_srli_si128((x), 8 * (imm)))
```

__m256 Data Types

The __m256 data type is used to represent the contents of the extended SSE register - the YMM register, used by the Intel® AVX intrinsics.

The __m256 data type can hold eight 32-bit floating-point values, while the __m256d data type can hold four 64-bit double precision floating-point values, and the __m256i data type can hold thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit integer values. See Details for Intel® AVX Intrinsics for more information.

__m512 Data Types

The __m512 data type is used to represent the contents of the extended SSE register - the ZMM register, used by the Intel® AVX-512 intrinsics.

The __m512 data type can hold sixteen 32-bit floating-point values, while the __m512d data type can hold eight 64-bit double precision floating-point values, and the __m512i data type can hold sixty-four 8-bit, thirty-two 16-bit, sixteen 32-bit, or eight 64-bit integer values. See Overview: Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Instructions for more information.

Data Types Usage Guidelines

These data types are not basic ANSI C data types. You must observe the following usage restrictions:

- Use data types as objects in aggregates, such as unions, to access the byte elements and structures.

See Also

__declspec(align) declaration

Naming and Usage Syntax

Most intrinsic names use the following notational convention:

```
_mm_<intrin_op>_<suffix>
```

The following table explains each item in the syntax.

<intrin_op>	Indicates the basic operation of the intrinsic; for example, add for addition and sub for subtraction.
<suffix>	Denotes the type of data the instruction operates on. The first one or two letters of each suffix denote whether the data is packed (p), extended packed (ep), or scalar (s). The remaining letters and numbers denote the type, with notation as follows: • s single-precision floating point • d double-precision floating point • i128 signed 128-bit integer • i64 signed 64-bit integer • u64 unsigned 64-bit integer • i32 signed 32-bit integer • u32 unsigned 32-bit integer • i16 signed 16-bit integer • u16 unsigned 16-bit integer • i8 signed 8-bit integer • u8 unsigned 8-bit integer

A number appended to a variable name indicates the element of a packed object. For example, r0 is the lowest word of r. Some intrinsics are "composites" because they require more than one instruction to implement them.

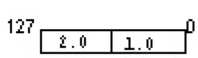
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the xmm register that holds the value t appears as follows:



The "scalar" element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

References

See the following publications and internet locations for more information about intrinsics and the Intel® architectures that support them. You can find all publications on the Intel website.

Internet Location or Publication	Description
http://www.intel.com/software/products	Technical resource center for hardware designers and developers; contains links to product pages and documentation.
Intel® 64 and IA-32 architecture manuals	Intel website for Intel® 64 and IA-32 architecture manuals.
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M	Describes the format of the instruction set of Intel® 64 and IA-32 architectures and covers the instructions from A to M.
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-U	Describes the format of the instruction set of Intel® 64 and IA-32 architectures and covers the instructions from N to U.
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, V-Z	Describes the format of the instruction set of Intel® 64 and IA-32 architectures and covers the instructions from V to Z.
https://software.intel.com/sites/landingpage/IntrinsicsGuide/	An interactive Intrinsics Guide that provides Intel intrinsic instructions.

Intrinsics for All Intel® Architectures

Most of the intrinsics documented in this section function for all supported Intel® architectures.

Some of the intrinsics documented in this section function across a subset of Intel® architectures.

Integer Arithmetic Intrinsics

The following table lists and describes integer arithmetic intrinsics that you can use across Intel® architectures.

Intrinsic Syntax	Description
Intrinsics for all Supported Intel® Architectures	
`int abs(int)`	Returns the absolute value of an integer.
`long labs(long)`	Returns the absolute value of a long integer.
`unsigned long _lrotl(unsigned long` *value*`, int` *shift*`)`	Implements 64-bit left rotate of *value* by *shift* positions.
`unsigned long _lrotr(unsigned long` *value*`, int` *shift*`)`	Implements 64-bit right rotate of *value* by *shift* positions.
`unsigned int _rotl(unsigned int` *value*`, int` *shift*`)`	Implements 32-bit left rotate of *value* by *shift* positions.
`unsigned int _rotr(unsigned int` *value*`, int` *shift*`)`	Implements 32-bit right rotate of *value* by *shift* positions.

Intrinsics for IA-32 and Intel® 64 Architectures	
`unsigned short _rotwl(unsigned short value, int shift)`	Implements 16-bit left rotate of *value* by *shift* positions.
`unsigned short _rotwr(unsigned short value, int shift)`	Implements 16-bit right rotate of *value* by *shift* positions.

NOTE
Passing a constant *shift* value in the rotate intrinsics results in higher performance.

Floating-point Intrinsics

The following table lists and describes floating point intrinsics that you can use across all Intel® and compatible architectures. Floating-point intrinsic functions may invoke library functions that are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Intrinsic	Description
`double fabs(double)`	Returns the absolute value of a floating-point value.
`double log(double)`	Returns the natural logarithm ln(x), x>0, with double precision.
`float logf(float)`	Returns the natural logarithm ln(x), x>0, with single precision.
`double log10(double)`	Returns the base 10 logarithm log10(x), x>0, with double precision.
`float log10f(float)`	Returns the base 10 logarithm log10(x), x>0, with single precision.
`double exp(double)`	Returns the exponential function with double precision.
`float expf(float)`	Returns the exponential function with single precision.
`double pow(double, double)`	Returns the value of *x* to the power *y* with double precision.
`float powf(float, float)`	Returns the value of *x* to the power *y* with single precision.
`double sin(double)`	Returns the sine of *x* with double precision.
`float sinf(float)`	Returns the sine of *x* with single precision.
`double cos(double)`	Returns the cosine of *x* with double precision.
`float cosf(float)`	Returns the cosine of *x* with single precision.
`double tan(double)`	Returns the tangent of *x* with double precision.
`float tanf(float)`	Returns the tangent of *x* with single precision.
`double acos(double)`	Returns the inverse cosine of *x* with double precision.
`float acosf(float)`	Returns the inverse cosine of *x* with single precision.
`double acosh(double)`	Compute the inverse hyperbolic cosine of the argument with double precision.

Intrinsic	Description
`float acoshf(float)`	Compute the inverse hyperbolic cosine of the argument with single precision.
`double asin(double)`	Compute inverse sine of the argument with double precision.
`float asinf(float)`	Compute inverse sine of the argument with single precision.
`double asinh(double)`	Compute inverse hyperbolic sine of the argument with double precision.
`float asinhf(float)`	Compute inverse hyperbolic sine of the argument with single precision.
`double atan(double)`	Compute inverse tangent of the argument with double precision.
`float atanf(float)`	Compute inverse tangent of the argument with single precision.
`double atanh(double)`	Compute inverse hyperbolic tangent of the argument with double precision.
`float atanhf(float)`	Compute inverse hyperbolic tangent of the argument with single precision.
`double cabs(double complex z)`	Computes absolute value of complex number. The intrinsic argument is a complex number made up of two double precision elements, one real and one imaginary. The input parameter *z* is made up of two values of double type passed together as a single argument.
`float cabsf(float complex z)`	Computes absolute value of complex number. The intrinsic argument is a complex number made up of two single precision elements, one real and one imaginary. The input parameter *z* is made up of two values of float type passed together as a single argument.
`double ceil(double)`	Computes smallest integral value of double precision argument not less than the argument.
`float ceilf(float)`	Computes smallest integral value of single precision argument not less than the argument.
`double cosh(double)`	Computes the hyperbolic cosine of double precision argument.
`float coshf(float)`	Computes the hyperbolic cosine of single precision argument.
`float fabsf(float)`	Computes absolute value of single precision argument.
`double floor(double)`	Computes the largest integral value of the double precision argument not greater than the argument.
`float floorf(float)`	Computes the largest integral value of the single precision argument not greater than the argument.
`double fmod(double)`	Computes the floating-point remainder of the division of the first argument by the second argument with double precision.
`float fmodf(float)`	Computes the floating-point remainder of the division of the first argument by the second argument with single precision.

Intrinsic	Description
`double hypot(double, double)`	Computes the length of the hypotenuse of a right angled triangle with double precision.
`float hypotf(float, float)`	Computes the length of the hypotenuse of a right angled triangle with single precision.
`double rint(double)`	Computes the integral value represented as double using the IEEE rounding mode.
`float rintf(float)`	Computes the integral value represented with single precision using the IEEE rounding mode.
`double sinh(double)`	Computes the hyperbolic sine of the double precision argument.
`float sinhf(float)`	Computes the hyperbolic sine of the single precision argument.
`float sqrtf(float)`	Computes the square root of the single precision argument.
`double tanh(double)`	Computes the hyperbolic tangent of the double precision argument.
`float tanhf(float)`	Computes the hyperbolic tangent of the single precision argument.

String and Block Copy Intrinsics

The following table lists and describes string and block copy intrinsics that you can use on systems based on IA-32 and Intel® 64 architectures. They may invoke library functions that are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
`strncpy()` and `strncmp()` functions are implemented as intrinsics depending on compiler version and compiler switches like optimization level.

Intrinsic	Description
`char *_strset(char *, _int32)`	Sets all characters in a string to a fixed value.
`int memcmp(const void *cs, const void *ct, size_t n)`	Compares two regions of memory. Return: • <0 if *cs*<*ct* • 0 if *cs*=*ct* • >0 if *cs*>*ct*
`void *memcpy(void *s, const void *ct, size_t n)`	Copies from memory. Returns *s*.
`void *memset(void *s, int c, size_t n)`	Sets memory to a fixed value. Returns *s*.
`char *strcat(char *s, const char *ct)`	Appends to a string. Returns *s*.
`int strcmp(const char *, const char *)`	Compares two strings. Return <0 if `cs`<*ct*, 0 if *cs*=*ct*, or >0 if *cs*>*ct*.

Intrinsic	Description
`char *strcpy(char *s, const char *ct)`	Copies a string. Returns `s`.
`size_t strlen(const char *cs)`	Returns the length of string *cs*.
`int strncmp(char *, char *, int)`	Compare two strings, but only specified number of characters.
`int strncpy(char *, char *, int)`	Copies a string, but only specified number of characters.

Miscellaneous Intrinsics

The following tables list and describe intrinsics that you can use across all Intel® architectures, except where noted. These intrinsics are available for both Intel® and non-Intel microprocessors but they may perform additional optimizations for Intel® microprocessors than they perform for non-Intel microprocessors.

NOTE
Casting functions for various `INT` and `FP` types for use in intrinsic functions across Intel® architectures only change the type; they *do not* convert between integer and floating point values.

Intrinsic	Description
Intrinsics for all Supported Intel® Architectures	
`__cpuid`	Queries the processor for information about processor type and supported features. The Intel® C++ Compiler Classic supports the Microsoft* implementation of this intrinsic. See the Microsoft documentation for details.
`void *_alloca(int)`	Allocates memory in the local stack frame. The memory is automatically freed upon return from the function.
`int _bit_scan_forward(int x)`	Returns the bit index of the least significant set bit of *x*. If *x* is 0, the result is undefined.
`int _bit_scan_reverse(int)`	Returns the bit index of the most significant set bit of *x*. If *x*is 0, the result is undefined.

`unsigned char _BitScanForward(unsigned __int32 *p, unsigned __int32 b);`
and for Intel® 64 architecture only:
`unsigned char _BitScanForward64(unsigned __int32 *p, unsigned __int64 b);`
	Sets **p* to the bit index of the least significant set bit of *b* or leaves it unchanged if *b* is zero. The function returns a non-zero result when *b* is non-zero and returns zero when *b* is zero.

`unsigned char _BitScanReverse(unsigned __int32 *p, unsigned __int32 b);` and for Intel® 64 architecture only:	Sets **p* to the bit index of the most significant set bit of *b* or leaves it unchanged if *b* is zero. The function returns a non-zero result when *b* is non-zero and returns zero when *b* is zero.

Intrinsic	Description
Intrinsics for all Supported Intel® Architectures	
`unsigned char _BitScanReverse64(unsigned __int32 *p, unsigned __int64 b);`	

`unsigned char _bittest(__int32 *p, __int32 b);`
and for Intel® 64 architecture only:
`unsigned char _bittest64(__int64 *p, __int64 b);`
	Returns the bit in position *b* of the memory addressed by *p*.

`unsigned char _bittestandcomplement(__int32 *p, __int32 b);`
and for Intel® 64 architecture only:
`unsigned char _bittestandcomplement64(__int64 *p, __int64 b);`
	Returns the bit in position *b* of the memory addressed by *p*, then compliments that bit.

`unsigned char _bittestandreset(__int32 *p, __int32 b);`
and for Intel® 64 architecture only:
`unsigned char _bittestandreset64(__int64 *p, __int64 b);`
	Returns the bit in position *b* of the memory addressed by *p*, then resets that bit to 0.

`unsigned char _bittestandset(__int32 *p, __int32 b);`
and for Intel® 64 architecture only:
`unsigned char _bittestandset64(__int64 *p, __int64 b);`
	Returns the bit in position *b* of the memory addressed by *p*, then sets the bit to 1.

`int _bswap(int)`	Reverses the byte order of *x*. Swaps 4 bytes; bits 0-7 are swapped with bits 24-31, bits 8-15 are swapped with bits 16-23.
`__int64 _bswap64(__int64 x)`	Reverses the byte order of *x*. Swaps 8 bytes; bits 0-7 are swapped with bits 56-63, bits 8-15 are swapped with bits 48-55, bits 16-23 are swapped with bits 40-47, and bits 24-31 are swapped with bits 32-39.

Intrinsic	Description
Intrinsics for all Supported Intel® Architectures	
`unsigned int __cacheSize(unsigned int cacheLevel)`	`__cacheSize(n)` returns the size in kilobytes of the cache at level n. 1 represents the first-level cache. 0 is returned for a non-existent cache level. For example, an application may query the cache size and use it to select block sizes in algorithms that operate on matrices.
`void _enable(void)`	Enables the interrupt.
`unsigned __int32 _castf32_u32(float)`	Casts *float* value to unsigned 32-bit integer.
`unsigned __int64 _castf64_u64(double)`	Casts *double* value to unsigned 64-bit integer.
`float _castu32_f32(unsigned __int32)`	Casts unsigned 32-bit integer to *float32*.
`double _castu64_f64(unsigned __int64)`	Casts unsigned 32-bit integer to *float64*.
`void _disable(void)`	Disables the interrupt.
`int _in_byte(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer data byte from port specified by argument.
`int _in_dword(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer double word from port specified by argument.
`int _in_word(int)`	Intrinsic that maps to the IA-32 instruction `IN`. Transfer word from port specified by argument.
`int _inp(int)`	Same as `_in_byte`.
`int _inpd(int)`	Same as `_in_dword`.
`int _inpw(int)`	Same as `_in_word`.
`int _out_byte(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer data byte in second argument to port specified by first argument.
`int _out_dword(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer double word in second argument to port specified by first argument.
`int _out_word(int, int)`	Intrinsic that maps to the IA-32 instruction `OUT`. Transfer word in second argument to port specified by first argument.
`int _outp(int, int)`	Same as `_out_byte`.
`int _outpw(int, int)`	Same as `_out_word`.
`int _outpd(int, int)`	Same as `_out_dword`.
`int _popcnt32(int` *x*`)`	Returns the number of set bits in *x*.

Intrinsic	Description
Intrinsics for all Supported Intel® Architectures	
`int _popcnt64(__int64 x)`	Returns the number of set bits in *x*.
`__int64 _rdpmc(int p)`	Returns the current value of the 40-bit performance monitoring counter specified by *p*.
Intrinsics for IA-32 and Intel® 64 Architectures	
`__int64 _rdtsc(void)`	Returns the current value of the processor's 64-bit time stamp counter.
`int _setjmp(jmp_buf)`	A fast version of `setjmp()`, which bypasses the termination handling. Saves the callee-save registers, stack pointer and return address.
`int __pin_value(char *annotation)`	Bypasses code that executes only in PIN mode. Especially useful with Intel® oneAPI Threading Building Blocks (oneTBB), which adds specified annotations to the object file so that code is executed in PIN mode.

_may_i_use_cpu_feature

Queries the processor dynamically at the source level (this intrinsic does not perform a vendor check) to determine if processor-specific features are available.

Syntax

```
extern int _may_i_use_cpu_feature(unsigned __int64);
```

Arguments

unsigned __int64	An unsigned __int64 bitset representing one or more cpuid features. The arguments for feature query accepted by this intrinsic is:
	`_FEATURE_GENERIC_IA32`
	`_FEATURE_FPU`
	`_FEATURE_CMOV`
	`_FEATURE_MMX`
	`_FEATURE_FXSAVE`
	`_FEATURE_SSE`
	`_FEATURE_SSE2`
	`_FEATURE_SSE3`
	`_FEATURE_SSSE3`
	`_FEATURE_SSE4_1`
	`_FEATURE_SSE4_2`
	`_FEATURE_POPCNT`

_FEATURE_MOVBE

_FEATURE_PCLMULQDQ

_FEATURE_AES

_FEATURE_F16C

_FEATURE_AVX

_FEATURE_RDRND

_FEATURE_FMA

_FEATURE_BMI

_FEATURE_LZCNT

_FEATURE_HLE

_FEATURE_RTM

_FEATURE_AVX2

_FEATURE_ADX

_FEATURE_RDSEED

_FEATURE_AVX512DQ

_FEATURE_AVX512F

_FEATURE_AVX512ER

_FEATURE_AVX512PF

_FEATURE_AVX512CD

_FEATURE_AVX512BW

_FEATURE_AVX512VL

_FEATURE_SHA

_FEATURE_MPX

_FEATURE_AVX512IFMA52

_FEATURE_AVX512VBMI

_FEATURE_AVX512_4FMAPS

_FEATURE_AVX512_4VNNIW

Description

This intrinsic queries the processor on which it is running to check the availability of the given features. This check is dynamically performed at the point in the source where it is called. For example:

```
if (_may_i_use_cpu_feature(_FEATURE_SSE4_2)) {
            Use SSE4.2 intrinsics;
} Else {
            Use generic code;
}
```

The `_may_i_use_feature` intrinsic, in this case, dynamically checks if the code is being executed on a processor that supports SSE4.2, and returns **true** if it is supported, or **false**. The `_may_i_use_feature` also accepts multiple features within a single argument, for example:

```
if (_may_i_use_cpu_feature(_FEATURE_SSE |
                    _FEATURE_SSE2 |
                    _FEATURE_SSE3 |
                    _FEATURE_SSSE3 |
                    _FEATURE_MOVBE) &&
                    !_may_i_use_cpu_feature(_FEATURE_SSE4_1)) {
printf("\nYou are running on an Atom processor."\n");
}
```

This intrinsic does not perform processor vendor checks that other features do (`-m <cpu>` type option).

Returns

Result of the feature query, true or false (1 or 0) for whether the set of features is available on the machine on which the intrinsic is executed.

See Also

m
  compiler option
Processor Targeting

[Q]ax
  compiler option
optimization_parameter

_allow_cpu_features

Tells the compiler that the code region may be targeted for processors with the specified features. The compiler may then generate optimized code for the specified features.

Syntax

`extern void _allow_cpu_features(unsigned __int64);`

Arguments

unsigned __int64	an unsigned __int64 bitset representing one or more cpuid features: _FEATURE_GENERIC_IA32 _FEATURE_FPU _FEATURE_CMOV _FEATURE_MMX _FEATURE_FXSAVE _FEATURE_SSE _FEATURE_SSE2 _FEATURE_SSE3 _FEATURE_SSSE3

_FEATURE_SSE4_1

_FEATURE_SSE4_2

_FEATURE_MOVBE

_FEATURE_POPCNT

_FEATURE_PCLMULQDQ

_FEATURE_AES

_FEATURE_F16C

_FEATURE_AVX

_FEATURE_RDRND

_FEATURE_FMA

_FEATURE_BMI

_FEATURE_LZCNT

_FEATURE_HLE

_FEATURE_RTM

_FEATURE_AVX2

`_FEATURE_ADX`

`_FEATURE_RDSEED`

`_FEATURE_AVX512DQ`

`_FEATURE_AVX512F`

`_FEATURE_AVX512ER`

`_FEATURE_AVX512PF`

`_FEATURE_AVX512CD`

`_FEATURE_AVX512BW`

`_FEATURE_AVX512VL`

`_FEATURE_SHA`

`_FEATURE_MPX`

`_FEATURE_AVX512IFMA52`

`_FEATURE_AVX512VBMI`

`_FEATURE_AVX512_4FMAPS`

`_FEATURE_AVX512_4VNNIW`

Description

Use this intrinsic function to use the specified processor feature at a code block level. The function only affects the scope of the code following the function call. Ensure that the code block will run only on processors with the specified features. If the code runs on a processor without the specified feature, the program may fail with an illegal instruction exception.

The function accepts a single argument that is a bitmask. In cases where one ISA depends on another, the higher ISA typically implies the lower. For example, the following arguments produce the same assembly code:

- `_FEATURE_SSE2|_FEATURE_AVX|_FEATURE_AVX512F`
- `_FEATURE_AVX512F`

The argument can only add features to those specified by the `[Q]x` or `-m` (Linux* and macOS) or `/arch` (Windows*) options, it cannot remove features.

This function does not itself cause the compiler to generate multiple code paths. To do that, you need to use `_may_i_use_cpu_feature()`.

> **NOTE**
> See the Release Notes for the latest information about this function.

To use specified processor features at a function level, use the `cpu_dispatch` or the `cpu_specific` attribute or the `optimization_parameter` pragma.

To use specified processor features at the file level, use the `[Q]x` compiler option.

The following example demonstrates how to use this intrinsic function to allow the compiler to generate the necessary code to use the Advanced Vector Extensions (AVX) and Streaming SIMD Extensions 2 (SSE2) features in the processor.

```
#include <string.h>
#include <immintrin.h>

#define MAXIMGS            20
#define MAXNAME            512

typedef struct {
  int x;                  /* image X axis size              */
  int y;                  /* image Y axis size              */
  int bpp;                /* image bits */
  char name[MAXNAME];     /* image full filename */
  unsigned char * data;   /* pointer to raw byte image data */
} rawimage;

extern rawimage * imagelist[MAXIMGS];
extern int numimages;

rawimage* CreateImage(char * filename)
{
       rawimage* newimage = NULL;
       int i, len, intable;

       intable=0;
       if (numimages!=0) {

             _allow_cpu_features(_FEATURE_SSE2 | _FEATURE_AVX);
             for (i=0; i<numimages; i++) {
                    if (!strcmp(filename, imagelist[i]->name)) {
                           newimage=imagelist[i];
                           intable=1;
                    }
             }
       }

       if (!intable) {
             newimage=(rawimage *)malloc(sizeof(rawimage));
             if (newimage != NULL) {
```

```
                    strcpy(newimage->name, filename);

                    imagelist[numimages]=newimage;  /* add new one to the table       */
                    numimages++;                    /* increment the number of images */
            }
    }

    return newimage;
}
```

Returns

Returns nothing.

See Also

_may_i_use_cpu_feature

cpu_dispatch, cpu_specific

optimization_parameter

Processor Targeting

x, Qx

Data Alignment, Memory Allocation Intrinsics, and Inline Assembly

This section describes features that support usage of the intrinsics.

Alignment Support

Aligning data improves the performance of intrinsics. When using the Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics, you should align data to 16 bytes in memory operations. Specifically, you must align `__m128` objects as addresses passed to the `_mm_load` and `_mm_store` intrinsics. If you want to declare arrays of floats and treat them as `__m128` objects by casting, you need to ensure that the float arrays are properly aligned.

Use `__declspec(align)` to direct the compiler to align data more strictly than it otherwise would. For example, a data object of type `int` is allocated at a byte address which is a multiple of 4 by default. By using `__declspec(align)`, you can direct the compiler to instead use an address which is a multiple of 8, 16, or 32 (with the following restriction on IA-32 architecture: 16-byte addresses can be locally or statically allocated).

You can use this data alignment support as an advantage in optimizing cache line usage. By clustering small objects that are commonly used together into a `struct`, and forcing the `struct` to be allocated at the beginning of a cache line, you can effectively guarantee that each object is loaded into the cache as soon as any one is accessed, resulting in a significant performance benefit.

For 16-byte alignment, you can use the macro `_MM_ALIGN16`, which other compilers can support by including header files. This macro enables you to write portable code that does not rely on compiler support for `__declspec(align)`.

See Also

__declspec(align) declaration

Programming Example includes example of _MM_ALIGN16

Allocating and Freeing Aligned Memory Blocks

To allocate and free aligned blocks of memory use the _mm_malloc and _mm_free intrinsics. These intrinsics are based on malloc and free, which are in the libirc.a library. You need to include malloc.h. The syntax for these intrinsics is as follows:

```
void* _mm_malloc (size_t size, size_t align )
```

```
void _mm_free (void *p)
```

The _mm_malloc routine takes an extra parameter, which is the alignment constraint. This constraint must be a power of two. The pointer that is returned from _mm_malloc is guaranteed to be aligned on the specified boundary.

> **NOTE**
> Memory that is allocated using _mm_malloc must be freed using _mm_free . Calling free on memory allocated with _mm_malloc or calling _mm_free on memory allocated with malloc will cause unpredictable behavior.

Inline Assembly

Microsoft* Style Inline Assembly

The Intel® C++ Compiler supports Microsoft-style inline assembly on Windows*. The Intel® C++ Compiler supports Microsoft-style inline assembly on Linux* when used with the -use-msasm option. See the Microsoft documentation for the proper syntax.

GNU*-like Style Inline Assembly (IA-32 architecture and Intel® 64 architecture only)

The Intel® C++ Compiler supports GNU-like style inline assembly. The syntax is as follows:

```
asm-keyword [ volatile-keyword ] ( asm-template [ asm-interface ] ) ;
```

The Intel® C++ Compiler also supports mixing UNIX* and Microsoft* style asms. Use the __asm__ keyword for GNU-style ASM when using the -use_msasm switch.

> **NOTE**
> The Intel® C++ Compiler supports gcc-style inline ASM if the assembler code uses AT&T* System V/386 syntax.

Syntax Element	Description
asm-keyword	Assembly statements begin with the keyword asm. Alternatively, either __asm or __asm__ may be used for compatibility. When mixing UNIX* and Microsoft* style asm, use the __asm__ keyword.

Syntax Element	Description
	The compiler only accepts the `__asm__` keyword. The `asm` and `__asm` keywords are reserved for Microsoft* style assembly statements.
`volatile-keyword`	If the optional keyword `volatile` is given, the `asm` is volatile. Two volatile `asm` statements are never moved past each other, and a reference to a volatile variable is not moved relative to a volatile `asm`. Alternate keywords `__volatile` and `__volatile__` may be used for compatibility.
`asm-template`	The `asm-template` is a C language ASCII string that specifies how to output the assembly code for an instruction. Most of the template is a fixed string; everything but the substitution-directives, if any, is passed through to the assembler. The syntax for a substitution directive is a `%` followed by one or two characters.
`asm-interface`	The `asm-interface` consists of three parts: **1.** An optional `output-list` **2.** An optional `input-list` **3.** An optional `clobber-list` These are separated by colon (:) characters. If the `output-list` is missing, but an `input-list` is given, the input list may be preceded by two colons (::) to take the place of the missing `output-list`. If the `asm-interface` is omitted altogether, the `asm` statement is considered volatile regardless of whether a `volatile-keyword` was specified.
`output-list`	An `output-list` consists of one or more output-specs separated by commas. For the purposes of substitution in the `asm-template`, each output-spec is numbered. The first operand in the `output-list` is numbered 0, the second is 1, and so on. Numbering is continuous through the `output-list` and into the `input-list`. The total number of operands is limited to 30 (i.e. 0-29).
`input-list`	Similar to an `output-list`, an `input-list` consists of one or more input-specs separated by commas. For the purposes of substitution in the `asm-template`, each input-spec is numbered, with the numbers continuing from those in the `output-list`.
`clobber-list`	A `clobber-list` tells the compiler that the `asm` uses or changes a specific machine register that is either coded directly into the `asm` or is changed implicitly by the assembly instruction. The `clobber-list` is a comma-separated list of `clobber-specs`.
`input-spec`	The `input-specs` tell the compiler about expressions whose values may be needed by the inserted assembly instruction. In order to describe fully the input requirements of the `asm`, you can list input-specs that are not actually referenced in the `asm-template`.
`clobber-spec`	Each `clobber-spec` specifies the name of a single machine register that is clobbered. The register name may optionally be preceded by a `%`. You can specify any valid machine register name. It is also legal to specify "memory" in a `clobber-spec`. This prevents the compiler from keeping data cached in registers across the `asm` statement.

When compiling an assembly statement on Linux*, the compiler simply emits the asm-template to the assembly file after making any necessary operand substitutions. The compiler then calls the GNU* assembler to generate machine code. In contrast, on Windows* the compiler itself must assemble the text contained in the asm-template string into machine code. In essence, the compiler contains a built-in assembler.

The compiler's built-in assembler supports the GNU* `.byte` directive but does not support other functionality of the GNU* assembler, so there are limitations in the contents of the asm-template. The following assembler features are not currently supported.

- Directives other than the `.byte` directive
- Symbols*

NOTE
* Direct symbol references in the asm-template are not supported. To access a C++ object, use the `asm-interface` with a substitution directive.

Example

Incorrect method for accessing a C++ object:

```
__asm__("addl $5, _x");
```

Proper method for accessing a C++ object:

```
__asm__("addl $5, %0" : "+rm" (x));
```

Additionally, there are some restrictions on the usage of labels. The compiler only allows local labels, and only references to labels within the same assembly statement are permitted. A local label has the form "*N:*", where *N* is a non-negative integer. *N* does not have to be unique, even within the same assembly statement. To reference the most recent definition of label *N*, use "*Nb*". To reference the next definition of label *N*, use "*Nf*". In this context, "*b*" means backward and "*f*" means forward. For more information, refer to the GNU assembler documentation.

GNU-style inline assembly statements on Windows* use the same assembly instruction format as on Linux* which is often referenced as AT&T* assembly syntax. This means that destination operands are on the right and source operands are on the left. This operand order is the reverse of Intel assembly syntax.

Due to the limitations of the compiler's built-in assembler, many assembly statements that compile and run on Linux* will not compile on Windows*. On the other hand, assembly statements that compile and run on Windows* should also compile and run on Linux*.

This feature provides a high-performance alternative to Microsoft-style inline assembly statements when portability between operating systems is important. Its intended use is in small primitives where high-performance integration with the surrounding C++ code is essential.

```
#ifdef _WIN64
#define INT64_PRINTF_FORMAT "I64"
#else
#define __int64 long long
#define INT64_PRINTF_FORMAT "L"
#endif
#include <stdio.h>
typedef struct {
    __int64 lo64;
    __int64 hi64;
} my_i128;
#define ADD128(out, in1, in2)                    \
    __asm__("addq %2, %0; adcq %3, %1" :         \
            "=r"(out.lo64), "=r"(out.hi64) :     \
```

```
            "emr" (in2.lo64), "emr"(in2.hi64),     \
            "0" (in1.lo64), "1" (in1.hi64));

extern int
main()
{
    my_i128 val1, val2, result;
    val1.lo64 = ~0;
    val1.hi64 = 0;

    val2.hi64 = 65;
    ADD128(result, val1, val2);
    printf("0x%016" INT64_PRINTF_FORMAT "x%016"   INT64_PRINTF_FORMAT "x\n",
            val1.hi64, val1.lo64);

    printf("+0x%016" INT64_PRINTF_FORMAT "x%016" INT64_PRINTF_FORMAT "x\n",
            val2.hi64, val2.lo64);

    printf("------------------------------------\n");
    printf("0x%016" INT64_PRINTF_FORMAT "x%016" INT64_PRINTF_FORMAT "x\n",
            result.hi64, result.lo64);
    return 0;
}
```

This example, written for Intel® 64 architecture, shows how to use a GNU-style inline assembly statement to add two 128-bit integers. In this example, a 128-bit integer is represented as two `__int64` objects in the `my_i128` structure. The inline assembly statement used to implement the addition is contained in the `ADD128` macro, which takes three `my_i128` arguments representing three 128-bit integers. The first argument is the output. The next two arguments are the inputs. The example compiles and runs using the Intel® C++ Compiler on Linux* or Windows*, producing the following output.

```
  0x0000000000000000ffffffffffffffff
+ 0x00000000000000410000000000000001
------------------------------------
+ 0x00000000000000420000000000000000
```

In the GNU-style inline assembly implementation, the asm interface specifies all the inputs, outputs, and side effects of the `asm` statement, enabling the compiler to generate very efficient code.

```
mov       r13, 0xffffffffffffffff
mov       r12, 0x000000000
add       r13, 1
adc       r12, 65
```

It is worth noting that when the compiler generates an assembly file on Windows*, it uses Intel syntax even though the assembly statement was written using AT&T* assembly syntax.

The compiler moves in`1.lo64` into a register to match the constraint of operand 4. Operand 4's constraint of "0" indicates that it must be assigned the same location as output operand 0. And operand 0's constraint is "`=r`", indicating that it must be assigned an integer register. In this case, the compiler chooses r13. In the same way, the compiler moves in `1.hi64` into register r12.

The constraints for input operands 2 and 3 allow the operands to be assigned a register location ("`r`"), a memory location ("`m`"), or a constant signed 32-bit integer value ("`e`"). In this case, the compiler chooses to match operands 2 and 3 with the constant values `1` and `65`, enabling the `add` and `adc` instructions to utilize the "register-immediate" forms.

The same operation is much more expensive using a Microsoft-style inline assembly statement, because the interface between the assembly statement and the surrounding C++ code is entirely through memory. Using Microsoft* assembly, the `ADD128` macro might be written as follows.

```
#define ADD128(out, in1, in2)                     \
    {                                             \
        __asm mov rax, in1.lo64                   \
        __asm mov rdx, in1.hi64                   \
        __asm add rax, in2.lo64                   \
        __asm adc rdx, in2.hi64                   \
        __asm mov out.lo64, rax                   \
        __asm mov out.hi64, rdx                   \
     }
```

The compiler must add code before the assembly statement to move the inputs into memory, and it must add code after the assembly statement to retrieve the outputs from memory. This prevents the compiler from exploiting some optimization opportunities. Thus, the following assembly code is produced.

```
        mov       QWORD PTR [rsp+32], -1
        mov       QWORD PTR [rsp+40], 0
        mov       QWORD PTR [rsp+48], 1
        mov       QWORD PTR [rsp+56], 65

; Begin ASM

        mov       rax, QWORD PTR [rsp+32]
        mov       rdx, QWORD PTR [rsp+40]
        add       rax, QWORD PTR [rsp+48]
        adc       rdx, QWORD PTR [rsp+56]
        mov       QWORD PTR [rsp+64], rax
        mov       QWORD PTR [rsp+72], rdx

; End ASM

        mov       rdx, QWORD PTR [rsp+72]
        mov       r8, QWORD PTR [rsp+64]
```

The operation that took only four instructions and no memory references using GNU-style inline assembly takes twelve instructions with twelve memory references using Microsoft-style inline assembly.

Intrinsics for Managing Extended Processor States and Registers

The Intel® C++ Compiler Classic provides twelve intrinsics for managing the extended processor states and extended registers. These intrinsics are available for IA-32 and Intel® 64 architectures running on supported operating systems.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The intrinsics map directly to the hardware system instructions described in "Intel® 64 and IA-32 Architectures Software Developer's Manual, volumes 1, 2a, and 2b" and "Intel® Advanced Vector Extensions Programming Reference".

Functional Overview

The intrinsics for managing the extended processor states and extended registers include:

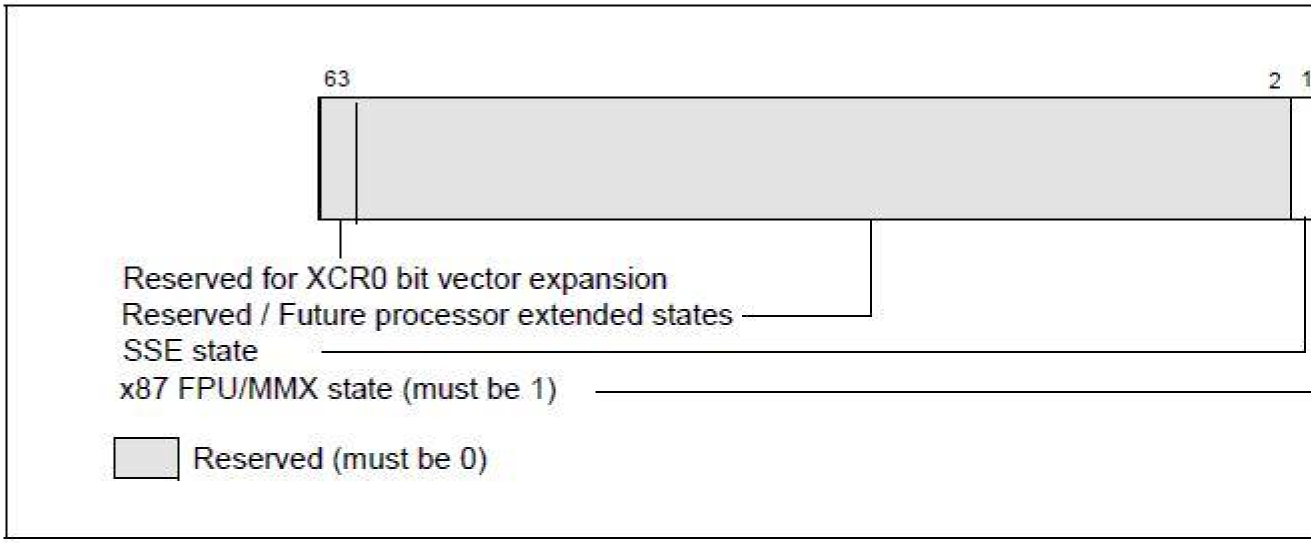
- Two intrinsics to read from and write to the specified extended control register. These intrinsics map to `XGETBV` and `XSETBV` instructions.
- Four intrinsics to save and restore the current state of the `x87 FPU`, `MMX`, `XMM`, and `MXCSR` registers. These intrinsics map to `FXSAVE`, `FXSAVE64`, `FXRSTOR`, and `FXRSTOR64` instructions.
- Six intrinsics to save and restore the current state of the `x87 FPU`, `MMX`, `XMM`, `YMM`, and `MXCSR` registers. These intrinsics map to `XSAVE`, `XSAVE64`, `XSAVEOPT`, `XSAVEOPT64`, `XRSTOR`, and `XRSTOR64` instructions.

Intrinsics for Reading and Writing the Content of Extended Control Registers

This group of intrinsics includes two intrinsics to read from and write to extended control registers (XCRs). Currently, the only such register defined is `XCR0`, `XFEATURE_ENABLED_MASK` register. This register specifies the set of processor states that the operating system enables on that processor, for example x87 FPU states, SSE states, and other processor extended states that Intel® 64 architecture may introduce in the future.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```



_xgetbv()

Reads the content of an extended control register.

Syntax

```
extern unsigned __int64 _xgetbv(unsigned int xcr);
```

Arguments

xcr	An extended control register to be read. Currently, only the value '0' is allowed.

Description

This intrinsic reads from extended control registers. Currently, the only control register allowed/defined is `(XCR0) XFEATURE_ENABLED_MASK` register. The corresponding constant is defined in the `immintrin.h` file to refer to this register:

```
#define _XCR_XFEATURE_ENABLED_MASK 0
```

This intrinsic maps to `XGETBV` instruction.

Returns

Returns the content of a specified extended control register.

_xsetbv()

Writes the given value to a specified extended control register.

Syntax

```
extern void _xsetbv(unsigned int xcr, unsigned __int64 val);
```

Arguments

xcr	An extended control register to be written. Currently, only the value '0' is allowed.
val	Value to be written to the specified extended control register.

Description

This intrinsic writes the given value to the specified extended control register. Currently, the only control register allowed/defined is `(XCR0) XFEATURE_ENABLED_MASK` register. The corresponding constant is defined in the `immintrin.h` file to refer to this register:

```
#define _XCR_XFEATURE_ENABLED_MASK 0
```

This intrinsic maps to `XSETBV` instruction.

Intrinsics for Saving and Restoring the Extended Processor States

To use any of these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsics that map to FXSAVE[64] and FXRSTOR[64] instructions

This group of intrinsics includes four intrinsics to save and restore the current state of the `x87 FPU`, `MMX`, `XMM`, and `MXCSR` registers.

These intrinsics accept a memory reference to a 16-byte aligned 512-byte memory chunk. The layout of the memory is shown below in Table 1.

Table 1 - `FXSAVE` save area layout.

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Rsrvd	CS	FPU IP		FOP		FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Rsrvd	DS		FPU DP		16
Reserved			ST0/MM0						32
Reserved			ST1/MM1						48
Reserved			ST2/MM2						64
Reserved			ST3/MM3						80
Reserved			ST4/MM4						96
Reserved			ST5/MM5						112
Reserved			ST6/MM6						128
Reserved			ST7/MM7						144
XMM0									160
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
XMM8									288
XMM9									304
XMM10									320
XMM11									336
XMM12									352
XMM13									368
XMM14									384
XMM15									400
Reserved									416
Reserved									432
Reserved									448

Intrinsics that map to XSAVE[64], XSAVEOPT[64], and XRSTOR[64] instructions

This group of intrinsics includes six intrinsics to fully or partially save and restore the current state of the `x87 FPU`, `MMX`, `XMM`, `YMM`, and `MXCSR` registers.

These intrinsics accept a memory reference to a 64-byte aligned memory. The layout of the register fields for the first 512 bytes is the same as the `FXSAVE` save area layout. The intrinsics saving the states do not write to bytes 464:511. The save area layout is shown in Tables 2a and 2b below.

The second operand is a save/restore mask specifying the saved/restored extended states. The value of the mask is `AND`ed with `XFEATURE_ENABLED_MASK(XCR0)`. A particular extended state is saved/restored only if the corresponding bit of both save/restore mask and `XFEATURE_ENABLED_MASK` is set to '1'.

Table 2a - `XSAVE` save area layout (first 512 bytes)

31........28	27........24	23........20	19........16	15........12	11........8	7........4	3........0	
MXCSR and MASK		x87 FPU operation states (see FXSAVE save area layout)						0
X87 FPU data registers (see FXSAVE instruction)								32
X87 FPU data registers (see FXSAVE instruction)								48
X87 FPU data registers (see FXSAVE instruction)								64
X87 FPU data registers (see FXSAVE instruction)								96
X87 FPU data registers (see FXSAVE instruction)								128
XMM1				XMM0				160
XMM3				XMM2				102
XMM5				XMM4				224
XMM7				XMM6				256
XMM9				XMM8				288
XMM11				XMM10				320
XMM13				XMM12				352
XMM15				XMM14				384
Reserved				Reserved				416
Reserved				Available				448
Available				Available				480

Table 2b - `XSAVE` save area layout for `YMM` registers

31........16	15........0	Byte offset from YMM save area	Byte offset from XSAVE save area
YMM1[255:128]	YMM0[255:128]	0	576
YMM3[255:128]	YMM2[255:128]	32	608
YMM5[255:128]	YMM4[255:128]	64	640
YMM7[255:128]	YMM6[255:128]	96	672
YMM9[255:128]	YMM8[255:128]	128	704
YMM11[255:128]	YMM10[255:128]	160	736
YMM13[255:128]	YMM12[255:128]	192	768
YMM15[255:128]	YMM14[255:128]	224	800

_fxsave()

Saves the states of x87 FPU, MMX, XMM, and MXCSR registers to memory.

Syntax

```
extern void _fxsave(void *mem);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.

Description

Saves the states of `x87 FPU`, `MMX`, `XMM`, and `MXCSR` registers to memory. This intrinsic maps to `FXSAVE` instruction.

_fxsave64()

Saves the states of x87 FPU, MMX, XMM, and MXCSR registers to memory.

Syntax

```
extern void _fxsave64(void *mem);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.

Description

Saves the states of `x87 FPU`, `MMX`, `XMM`, and `MXCSR` registers to memory. This intrinsic maps to `FXSAVE64` instruction.

_fxrstor()

Restores the states of x87 FPU, MMX, XMM, and MXCSR registers from memory.

Syntax

```
extern void _fxrstor(void *mem);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.

Description

Restores the states of `x87 FPU`, `MMX`, `XMM`, and `MXCSR` registers from memory. This intrinsic maps to `FXRSTOR` instruction.

_fxrstor64()

Restores the states of x87 FPU, MMX, XMM, and MXCSR registers from memory.

Syntax

```
extern void _fxrstor64(void *mem);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.

Description

Restores the states of x87 FPU, MMX, XMM, and MXCSR registers from memory. This intrinsic maps to FXRSTOR64 instruction.

_xsave()/_xsavec()/_xsaves()

Saves the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers to memory.

Syntax

```
extern void _xsave(void *mem, unsigned __int64 save_mask);
extern void _xsavec(void *mem, unsigned __int64 save_mask);
extern void _xsaves(void *mem, unsigned __int64 save_mask);
```

Arguments

mem	A memory reference to FXSAVE area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.
save_mask	A bit mask specifying the extended states to be saved.

Description

Saves the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers to memory. The xsave intrinsic maps to XSAVE instruction, the xsavec intrinsic maps to XSAVEC instruction, and the xsaves intrinsic maps to XSAVES instruction. See the Intel® 64 and IA-32 Architectures Software Developer's Manual for information on how the three instructions differ.

_xsave64()/ _xsavec64()/ _xsaves64()

Saves the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers to memory.

Syntax

```
extern void _xsave64(void *mem, unsigned __int64 save_mask);
extern void _xsavec64(void *mem, unsigned __int64 save_mask);
extern void _xsaves64(void *mem, unsigned __int64 save_mask);
```

Arguments

mem	A memory reference to FXSAVE area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.
save_mask	A bit mask specifying the extended states to be saved.

Description

Saves the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers to memory. The xsave64 intrinsic maps to XSAVE64 instruction, the xsavec64 intrinsic maps to XSAVEC64 instruction, and the xsaves64 intrinsic maps to XSAVES64 instruction. See the Intel® 64 and IA-32 Architectures Software Developer's Manual for information on how the three instructions differ.

_xsaveopt()

Saves the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers to memory, optimizing the save operation if possible.

Syntax

```
extern void _xsaveopt(void *mem, unsigned __int64 save_mask);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.
save_mask	A bit mask specifying the extended states to be saved.

Description

Saves the states of `x87 FPU`, `MMX`, `XMM`, `YMM`, and `MXCSR` registers to memory, optimizing the save operation if possible. This intrinsic maps to `XSAVEOPT` instruction.

_xsaveopt64()

Saves the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers to memory, optimizing the save operation if possible.

Syntax

```
extern void _xsaveopt64(void *mem, unsigned __int64 save_mask);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.
save_mask	A bit mask specifying the extended states to be saved.

Description

Saves the states of `x87 FPU`, `MMX`, `XMM`, `YMM`, and `MXCSR` registers to memory, optimizing the save operation if possible. This intrinsic maps to `XSAVEOPT64` instruction.

_xrstor()/xrstors()

Restores the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers from memory.

Syntax

```
extern void _xrstor(void *mem, unsigned __int64 rstor_mask);
```

```
extern void _xrstors(const void *mem, unsigned __int64 rstor_mask);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.

rstor_mask	A bit mask specifying the extended states to be restored.

Description

Restores the states of `x87 FPU`, `MMX`, `XMM`, `YMM`, and `MXCSR` registers from memory. The `xrstor` intrinsic maps to `XRSTOR` instruction, and the `xrstors` intrinsic maps to `XRSTORS` instruction. See the Intel® 64 and IA-32 Architectures Software Developer's Manual for information on how the instructions differ.

_xrstor64()/xrstors64()

Restores the states of x87 FPU, MMX, XMM, YMM, and MXCSR registers from memory.

Syntax

```
extern void _xrstor64(void *mem, unsigned __int64 rstor_mask);
extern void _xrstors64(const void *mem, unsigned __int64 rstor_mask);
```

Arguments

mem	A memory reference to `FXSAVE` area. The 512-bytes memory addressed by the reference must be 16-bytes aligned.
rstor_mask	A bit mask specifying the extended states to be restored.

Description

Restores the states of `x87 FPU`, `MMX`, `XMM`, `YMM`, and `MXCSR` registers from memory. The `xrstor64` intrinsic maps to `XRSTOR64` instruction, and the `xrstors64` intrinsic maps to `XRSTORS64` instruction. See the Intel® 64 and IA-32 Architectures Software Developer's Manual for information on how the instructions differ.

Intrinsics for the Short Vector Random Number Generator Library

The Short Vector Random Number Generator (SVRNG) library provides intrinsics for the IA-32 and Intel® 64 architectures running on supported operating systems. The SVRNG library partially covers both standard C++ (as referenced here: http://www.cplusplus.com/reference/random/) and the random number generation functionality of the Intel® oneAPI Math Kernel Library (oneMKL). The SVRNG library allows users to produce random numbers using a combination of engines and distributions. Engines are basic generators which produce uniformly distributed 32-bit or 64-bit unsigned integer numbers. Distributions transform the sequences of numbers generated by an engine into sequences of numbers with specific random variable distributions, such as uniform, normal, binomial and others. The distributions support single- or double-precision floating point and 32-bit signed integer outputs.

Both scalar and vector implementations are available for SVRNG generation functions. Scalar versions return native C++ data types: float, double, and both 32- and 64-bit integers. Vector versions produce packed results using SIMD-vector registers via corresponding data types as outlined in Data Types and Calling Conventions. Scalar versions called in loops can be vectorized by the compiler.

Unlike simple random number generators such as `rand()`, SVRNG engines and distributions require initialization routines which allocate memory and pre-compute constants required for fast vector generation. Finalization routines are provided to deallocate memory. Some engines support skip-ahead and leap-frog techniques for use in parallel computing environment. The Parallel Computation Support section discusses how these are used to obtain a random number sequence in parallel that is identical to the random number sequence that is generated in the sequential case. Error handling in SVRNG is done via status set and get functions. Additionally NULL pointers are returned on errors when possible.

SVRNG SIMD-vector functions and corresponding vectorized scalar calls are highly optimized for the following instructions sets:

- Intel® Streaming SIMD Extensions 2 (Intel® SSE2) (default)
- Intel® Advanced Vector Extensions 2 (Intel® AVX2)
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Instructions (on Intel® Many Integrated Core Architecture (Intel® MIC Architecture) and elsewhere)

Further Reference

The following documents are referenced in this section to provide further detail:

- Developer Reference for Intel® oneAPI Math Kernel Library - C: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html
- Notes for Intel® oneAPI Math Kernel Library Vector Statistics: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-vsnotes/top.html
- _vectorcall and __regcall demystified: https://software.intel.com/content/www/us/en/develop/articles/vectorcall-and-regcall-demystified.html

Data Types and Calling Conventions

Data types specific to the Short Vector Random Number Generator (SVRNG) Library

There are two types of SVRNG functions: the initialization and service routines and generation functions. The initialization and service routines introduce two new data types:

svrng_engine_t	A pointer to the engine-specific data structure created by the engine initialization routine. The structure contains pre-computed constants necessary for fast and precise random number vector generation by the engine. The structure size is engine-dependent.
svrng_distribution_t	A pointer to the distribution-specific data structure created by the distribution initialization routine. The structure contains pre-computed loop invariant constants to perform distribution transformations efficiently. The structure size is distribution-dependent.

While scalar SVRNG generation functions return native "C" data types (*float, double, 32-bit and 64-bit integers*), the SIMD-vector versions produce 1 [1] , 2, 4, 8, 16, or 32 packed results in one or several SIMD-vector registers. A set of SVRNG-specific vector types have been introduced to return these packed results. These types are CPU-specific and mapped to different numbers of SIMD-registers depending on the architecture where the program runs:

Type name	Number of packed values	SSE2 (default)	AVX2[1]
		Unsigned 32-bit integer	
svrng_uint1_t	1	__m128i	__m128i
svrng_uint2_t	2	__m128i	__m128i
svrng_uint4_t	4	__m128i	__m128i
svrng_uint8_t	8	struct { __m128i r[2]; }	__m256i

svrng_uint16_t	16	struct { __m128i r[4]; }	struct { __m256i r[2]; }
svrng_uint32_t	32	struct { __m128i r[8]; }	struct { __m256i r[4]; }
	Unsigned 64-bit integer		
svrng_ulong1_t	1	__m128i	__m128i
svrng_ulong2_t	2	__m128i	__m128i
svrng_ulong4_t	4	struct { __m128i r[2]; }	__m256i
svrng_ulong8_t	8	struct { __m128i r[4]; }	struct { __m256i r[2]; }
svrng_ulong16_t	16	struct { __m128i r[8]; }	struct { __m256i r[4]; }
svrng_ulong32_t	32	struct { __m128i r[16]; }	struct { __m256i r[8]; }
	Signed 32-bit integer		
svrng_int1_t	1	__m128i	__m128i
svrng_int2_t	2	__m128i	__m128i
svrng_int4_t	4	__m128i	__m128i
svrng_int8_t	8	struct { __m128i r[2]; }	__m256i
svrng_int16_t	16	struct { __m128i r[4]; }	struct { __m256i r[2]; }
svrng_int32_t	32	struct { __m128i r[8]; }	struct { __m256i r[4]; }
	Single-precision floating point		
svrng_float1_t	1	__m128	__m128
svrng_float2_t	2	__m128	__m128
svrng_float4_t	4	__m128	__m128
svrng_float8_t	8	struct { __m128 r[2]; }	__m256
svrng_float16_t	16	struct { __m128 r[4]; }	struct { __m256 r[2]; }
svrng_float32_t	32	struct { __m128 r[8]; }	struct { __m256 r[4]; }
	Double-precision floating point		
svrng_double1_t	1	__m128d	__m128d
svrng_double2_t	2	__m128d	__m128d
svrng_double4_t	4	struct { __m128d r[2]; }	__m256d
svrng_double8_t	8	struct { __m128d r[4]; }	struct { __m256d r[2]; }
svrng_double16_t	16	struct { __m128d r[8]; }	struct { __m256d r[4]; }
svrng_double32_t	32	struct { __m128d r[16]; }	struct { __m256d r[8]; }

[1] Note that SVRNG does not have optimizations specific to the Intel® Advanced Vector Extensions (Intel® AVX) instruction set. On hardware that supports Intel® AVX the Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instruction default versions are called, so you must use the Intel® SSE2 data types to interpret the results.

SVRNG calling conventions

All SVRNG routines use the regcall calling convention which provides the most use of hardware vector registers for passing parameters and returning results. See the "C/C++ Calling Conventions" section and the "_vectorcall and __regcall demystified" article referenced in the Introduction. This avoids unnecessary memory spills and fills of registers and improves performance.

In addition this convention provides the opportunity to deploy the "vector variant" declaration feature specific to the Intel® compiler. The declaration specifies a vector variant function that corresponds to its original C/C++ scalar function. This vector variant function can be invoked in vector context at the site of the call. See the `vector_variant` section for more detail. All SIMD-vector SVRNG intrinsics (except packed length = 1) are declared in the `svrng.h` header file as vector_variant to support automatic vectorization.

See Also

Introduction
C/C++ Calling Conventions
vector_variant

Usage Model

A typical usage model for using the intrinsics in the Short Vector Random Number Generator (SVRNG) library is the same as for standard C++ or Intel® oneAPI Math Kernel Library (oneMKL) vector statistics random number generator and looks something like the following:

- Include `svrng.h` header file
- Create and initialize basic SVRNG generator engine, create and initialize distribution (if necessary).
- Call one or more SVRNG generation function.
- Process the output.
- Delete the SVRNG engines and distributions.

On Windows*, users will need to explicitly link the static or dynamic libraries: static: `libirng.lib`, dynamic: `libirngmd.lib`. On Linux* and macOS the compiler driver will link automatically.

The following example demonstrates generation of a random stream that is output of basic generator engine MT19937 with seed equal to 777. The engine is used to generate two arrays: 1024 uniformly distributed random numbers between <a = 0.0, b = 4.0> via scalar generator call which should be vectorized by the compiler and 1024 normally distributed with parameters <mean = 2.0, standard deviation = 1.0> random numbers in blocks by 16 elements via direct call of SIMD-vector implementation. Delete engines and distributions after completing the generation. Check status for possible errors happened. The purpose of the example is to calculate the sample mean for both distributions with the given parameters.

```
#include <stdio.h>
#include <svrng.h>

int main( void )
{
  int                    i, st = SVRNG_STATUS_OK;
  double                 res1[1024],  res2[1024];
  double                 sum1 = 0, sum2 = 0;
  double                 mean1, mean2;
  svrng_engine_t         engine;
  svrng_distribution_t   distr1, distr2;
```

```
    /* Create mt19937 engine */
    engine = svrng_new_mt19937_engine( 777 );

    /* Create uniform distribution */
    distr1 = svrng_new_uniform_distribution_double( 0.0, 4.0 );

    /* Create normal distribution */
    distr2 = svrng_new_normal_distribution_double( 2.0, 1.0 );

    /* Scalar generator call, can be vectorized by compiler */
    #pragma ivdep
    #pragma vector always
    for( i = 0; i < 1024; i ++ ) {
        res1[i] = svrng_generate_double( engine, distr1 );
    }

    /* Direct call to SIMD-vector implementation */
    /* generating 16 packed elements */
    for( i = 0; i < 1024; i += 16 ) {
        *((svrng_double16_t*)(&res2[i])) =
        svrng_generate16_double( engine, distr2 );
    }

    /* Compute mean values */
    for( i = 0; i < 1024; i++ ) {
        sum1 += res1[i];
        sum2 += res2[i];
    }

    mean1 = sum1 / 1024.0;
    mean2 = sum2 / 1024.0;

    /* Printing results */
    printf( "Sample mean of uniform distribution = %f\n", mean1 );
    printf( "Sample mean of normal  distribution = %f\n", mean2 );

    /* Check for resulted status */
    st = svrng_get_status();

    if(st != SVRNG_STATUS_OK) {
        printf("FAILED: status error %d returned\n", st);
    }

    /* Delete distributions */
    svrng_delete_distribution( distr1 );
    svrng_delete_distribution( distr2 );

    /* Delete engine */
    svrng_delete_engine( engine );

return st;
}
```

Another example demonstrates the "skip-ahead" technique which ensures identical random number sequences in cases of parallel and sequential generation for certain engines. The rand0 engine is being created and copied to T "threads" with the "skip-ahead" adjustments applied. Each "thread" generates N uniformly distributed unsigned integer random values and then all LEN=T*N numbers are compared to the sequential call:

```
#include <stdio.h>
#include <stdint.h>
#include <svrng.h>

#define LEN        1024
#define T          8
#define N          (LEN/T)

int main( void ) {
  uint32_t       seq_res[LEN+32], parallel_res[LEN+32];
  svrng_engine_t seq_engine;
  svrng_engine_t parallel_engine[T];
  int            l, n, t, errs = 0, st = SVRNG_STATUS_OK;

    /* Create sequential engine and distr */
    seq_engine = svrng_new_rand0_engine( 777 );

    /* Copy existing sequential engine to new T parallel ones */
    /* with t*N offsets using skipahead method */
    for( t = 0; t < T; t++ ) {
        int thr_offset = t*N;
        parallel_engine[t] = svrng_copy_engine( seq_engine );
        parallel_engine[t] = \
          svrng_skipahead_engine( parallel_engine[t], thr_offset );
    }

    /* Sequential loop using scalar function (can be vectorized) */
    #pragma ivdep
    #pragma vector always
    for( l = 0; l < LEN; l++ ) {
        seq_res[l] = svrng_generate_uint( seq_engine );
    }

    /* Parallel loop using SIMD-vector function, */
    /* may be spreaded by threads */
    for( t = 0; t < T; t++ ) {
        for( n = 0; n < N; n += 8 ) {
            *((svrng_uint8_t*)(&(parallel_res[t*N+n]))) = \
            svrng_generate8_uint( parallel_engine[t] );
        }
    }

    /* Compare seq and parallel results */
    for(l = 0; l < LEN; l++) {
        if( parallel_res[l] != seq_res[l]) {
            errs++;
        }
    }

    /* Check for resulted status */
    st = svrng_get_status();
```

```
    /* Print overall result */
    if(st != SVRNG_STATUS_OK) {
        printf("FAILED: status error %d returned\n", st);
    }
    else if(errs) {
        printf("FAILED: %d skipahead errors\n",  errs);
    }
    else {
        printf("PASSED\n");
    }

    /* Delete engines */
    svrng_delete_engine( seq_engine );
    for( t = 0; t < T; t++ ) {
        svrng_delete_engine(parallel_engine[t]);
    }

  return (errs-st);
}
```

Engine Initialization and Finalization

Unlike the simple `rand()` function, vector random number generators in the Short Vector Random Number Generator (SVRNG) library require the initialization of an engine before the first generator run. This is due to the fact that a number of initialization values called the vector state of the engine must be pre-computed to perform effective vector generation. Once computed that vector state is retained and updated in memory as more numbers are generated. When no more random numbers are needed, that memory can be deallocated. The next few topics provide the functions used to allocate memory, initialize, and deallocate memory for all supported SVRNG engines.

The SVRNG library supports the following engines from the C++11 standard and the Intel® oneAPI Math Kernel Library (oneMKL) vector statistics random number generator collection:

- **rand0** (C++11 standard)
- **rand** (C++11 standard)
- **mcg31m1** (oneMKL)
- **mcg59** (oneMKL)
- **mt19937** (oneMKL and C++11 standard)

For more information on the figures of merit for these random number generator engines read the Basic Random Generator Properties and Testing Results section of the *Notes for Intel® oneAPI Math Kernel Library Vector Statistics* document (see the introduction).

For each engine there is a simple and extended version of the initialization function. Simple initialization has one parameter, the seed, and constructs the rest of the vector state to generate the proper sequence for the engine type. The extended versions of the initialization functions, with the `_ex` suffix, use multiple constants to set generator state values. The application notes in the description of each engine provide more detail on how these constants are used. The usual case for extended initialization requires enough constants to fill a SIMD register with 64-bit values on the system which the program is intended to run. The following table sums up the width (`SIMD_WIDTH`) of the SIMD registers used by the instructions sets for which the SVRNG intrinsics are optimized:

Instruction set	***SIMD_WIDTH***
Intel® Streaming SIMD Extensions 2 (Intel® SSE2)	2

Instruction set	***SIMD_WIDTH***
Intel® Advanced Vector Extensions 2 (Intel® AVX2)	4
Intel® Advanced Vector Extensions 512 (Intel® AVX-512)	8

See Also

Introduction

svrng_new_rand0_engine/svrng_new_rand0_ex

Routines for allocating memory for a rand0 engine and initializing with one or multiple seeds

Syntax

`svrng_engine_t svrng_new_rand0_engine( uint32_t seed )`

`svrng_engine_t svrng_new_rand0_engine_ex( int num, uint32_t *pseed )`

Input Parameters

seed	Initial condition for the engine.
num	Number of initialization values for the extended routine. May be 0 (*seed* set to 1), 1 (*seed* set to *pseed[0]*), or `SIMD_WIDTH`.
pseed	Pointer to an array with initialization values for the extended routine.

Description

The `svrng_new_rand0_engine` function allocates memory for the rand0 engine originated from C++11 standard and initializes it using one *seed* value. The extended version of the function, `svrng_new_rand0_engine_ex`, accepts several values for complex initialization cases where the user needs to fill the whole vector state with their own constants.

Status flags set

Name	**Description**
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_PARAM1	Bad parameter: `num`
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `pseed`

Return Values

A pointer to an initialized engine or NULL on error.

Application Notes

The rand0 engine is a simple 32-bit multiplicative congruential pseudo-random number generator represented by formula:

$x_{i+1} = (a*x_i) \bmod m$

multiplier *a* = *16807 (=7^5)*

modulus *m* = *2147483647 (=2^{31}-1)*

Range: [0,MAX), where MAX = m

svrng_new_rand_engine/svrng_new_rand_ex

Routines for allocating memory for a rand engine and initializing with one or multiple seeds

Syntax

```
svrng_engine_t svrng_new_rand_engine( uint32_t seed )
```

```
svrng_engine_t svrng_new_rand_engine_ex( int num, uint32_t *pseed )
```

Input Parameters

seed	Initial condition for the engine.
num	Number of initialization values for the extended routine. May be 0 (*seed* set to 1), 1 (*seed* set to *pseed[0]*), or `SIMD_WIDTH`.
pseed	Pointer to an array with initialization values for the extended routine.

Description

The `svrng_new_rand_engine` function allocates memory for the rand engine (originated from C++ 11 standard) and initializes it using one *seed* value. The extended version of the function, `svrng_new_rand_engine_ex`, accepts several values for complex initialization cases where the user needs to fill the whole vector state with their own constants.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_PARAM1	Bad parameter: `num`
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `pseed`

Return Values

A pointer to an initialized engine or NULL on error.

Application Notes

The rand is a simple 32-bit multiplicative congruential pseudo-random number generator represented by formula:

$x_{i+1} = (a*x_i) \bmod m$

multiplier *a* = *48271*

modulus *m* = *2147483647 (=2^{31}-1)*

Range: [0,MAX), where MAX = m

svrng_new_mcg31m1_engine/svrng_new_mcg31m1_ex

Routines for allocating memory for a mcg31m1 engine and initializing with one or multiple seeds

Syntax

```
svrng_engine_t svrng_new_mcg31m1_engine( uint32_t seed )
```

```
svrng_engine_t svrng_new_mcg31m1_engine_ex( int num, uint32_t *pseed )
```

Input Parameters

seed	Initial condition for the engine.
num	Number of initialization values for the extended routine. May be 0 (*seed* set to 1), 1 (*seed* set to *pseed[0]*), or `SIMD_WIDTH`.
pseed	Pointer to an array with initialization values for the extended routine.

Description

The `svrng_new_mcg31m1_engine` function allocates memory for the mcg31m1 engine (originated from C++ 11 standard) and initializes it using one *seed* value. The extended version of the function, `svrng_new_mcg31m1_engine_ex`, accepts several values for complex initialization cases where the user needs to fill the whole vector state with their own constants.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_PARAM1	Bad parameter: `num`
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `pseed`

Return Values

A pointer to an initialized engine or NULL on error.

Application Notes

The mcg31m1 is a simple 32-bit multiplicative congruential pseudo-random number generator represented by formula:

$x_{i+1} = (a*x_i) \bmod m$

multiplier $a = 1132489760$

modulus $m = 2147483647$ $(=2^{31}-1)$ Range: [0,MAX), where MAX = m

svrng_new_mcg59_engine/svrng_new_mcg59_ex

Routines for allocating memory for a mcg59 engine and initializing with one or multiple seeds

Syntax

`svrng_engine_t svrng_new_mcg59_engine( uint32_t seed )`

`svrng_engine_t svrng_new_mcg59_engine_ex( int num, uint32_t *pseed )`

Input Parameters

seed	Initial condition for the engine.
num	Number of initialization values for the extended routine. May be 0 (*seed* set to 1), 1 (*seed* set to *pseed[0]*), or `SIMD_WIDTH`.
pseed	Pointer to an array with initialization values for the extended routine.

Description

The `svrng_new_mcg59_engine` function allocates memory for the mcg59 engine (originated from C++ 11 standard) and initializes it using one *seed* value. The extended version of the function, `svrng_new_mcg59_engine_ex`, accepts several values for complex initialization cases where the user needs to fill the whole vector state with their own constants.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_PARAM1	Bad parameter: `num`
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `pseed`

Return Values

A pointer to an initialized engine or NULL on error.

Application Notes

The mcg59 is a simple 64-bit multiplicative congruential pseudo-random number generator represented by formula:

$x_{i+1} = (a*x_i) \bmod m$

multiplier $a = 13^{13}$

modulus $m = 2^{59}$ Range: [0,MAX), where MAX = m

svrng_new_mt19937_engine/svrng_new_mt19937_ex

Routines for allocating memory for an mt19937 engine and initializing with one or multiple seeds

Syntax

`svrng_engine_t svrng_new_mt19937_engine( uint32_t seed )`

`svrng_engine_t svrng_new_mt19937_engine_ex( int num, uint32_t *pseed )`

Input Parameters

seed	Initial condition for the engine.
num	Number of initialization values for the extended routine. `num`>=0. See VSL Notes for further details on extended initialization of the mt19937 engine.
pseed	Pointer to an array with initialization values for the extended routine.

Description

The `svrng_new_mt19937_engine` function allocates memory for the mt19937 engine (from C++ 11 standard) and initializes it using one *seed* value. The extended version of the function, `svrng_new_mt19937_engine_ex`, accepts several values for complex initialization cases. Because the mt19937 engine has 19937 bits of state in memory, its initialization differs from the other engines. See the *Notes for Intel® oneAPI Math Kernel Library Vector Statistics* document for detailed information on this engine.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_PARAM1	Bad parameter: `num`
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `pseed`

Return Values

A pointer to an initialized engine or NULL on error.

Application Notes

The mt19937 is a Mersenne Twister pseudo-random generator of 32-bit numbers with a state size of 19937 bits that is a modification of twisted generalized feedback shift register generator. Range: [0,MAX), where MAX = 2^{32}.

See Also

Introduction

svrng_delete_engine

Deallocates memory for the specified engine

Syntax

```
svrng_engine_t svrng_delete_engine( svrng_engine_t engine )
```

Input Parameters

engine	Pointer to the engine to be deallocated.

Description

The `svrng_delete_engine` function deallocates memory for the specified engine.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_BAD_ENGINE	Bad engine (NULL pointer)

Return Values

NULL pointer.

Distribution Initialization and Finalization

The Short Vector Random Number Generator (SVRNG) library supports the following distributions:

- **uniform** (*single and double floating point and 32-bit integer*)
- **normal** (*single and double floating point*)

SVRNG distributions must be initialized before random numbers can be generated. The initialization and finalization routines in this section allocate memory, pre-compute loop-invariant values and broadcast scalar constants for fast vector generation. Update functions are also provided to re-compute these numbers without memory re-allocation. More detail on the figures of merit for these distributions can be found in the Figures of Merit for Random Number Generators and Testing of Distribution Random Number Generators sections of the *Notes for Intel® oneAPI Math Kernel Library Vector Statistics* document referenced in the Introduction.

See Also

Introduction

svrng_new_uniform_distribution_[int|float|double]/svrng_update_uniform_distribution_[int|float|double]

Allocates and initializes constants for the uniform distribution with specified parameters

Syntax

```
svrng_distribution_t svrng_new_uniform_distribution_int( int a, int b )

svrng_distribution_t svrng_new_uniform_distribution_float( float a, float b )

svrng_distribution_t svrng_new_uniform_distribution_double( double a, double b )

svrng_distribution_t svrng_update_uniform_distribution_int( svrng_distribution_t distr,
int a, int b )

svrng_distribution_t svrng_update_uniform_distribution_float( svrng_distribution_t
distr, float a, float b )

svrng_distribution_t svrng_update_uniform_distribution_double( svrng_distribution_t
distr, double a, double b )
```

Input Parameters

a	Left bound of interval
b	Right bound of interval
distr	Pointer to the distribution to be updated

Description

The **svrng_new_uniform_distribution_[int|float|double]** function allocates memory for a uniform distribution and pre-computes and broadcasts loop-invariant constants required for vector generation of uniformly distributed values over the interval [a, b), where a, b are the real left and right bounds of the interval respectively with a < b. 32-bit `int`, `float` and `double` types are supported. The **svrng_update_uniform_distribution_[int|float|double]** functions give the same result, but by modifying existing distributions instead of allocating memory for new distributions.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_PARAMS	`a>=b`
SVRNG_STATUS_ERROR_BAD_DISTR	Bad distribution (NULL pointer)

Return Values

A pointer to the distribution created or updated by the function, or NULL on error.

svrng_new_normal_distribution_[float|double]/svrng_update_normal_distribution_[float|double]

Allocates and initializes constants for the normal distribution with specified parameters

Syntax

`svrng_distribution_t svrng_new_normal_distribution_float( float mean, float stddev )`

`svrng_distribution_t svrng_new_normal_distribution_double( double mean, double stddev )`

`svrng_distribution_t svrng_update_normal_distribution_float( svrng_distribution_t distr, float mean, float stddev )`

`svrng_distribution_t svrng_update_normal_distribution_double( svrng_distribution_t distr, double mean, double stddev )`

Input Parameters

mean	Mean value of the normal distribution.
stddev	Standard deviation of the normal distribution
distr	Pointer to the distribution to be updated

Description

The **svrng_new_normal_distribution_[float|double]** functions allocate memory for a normal distribution of either 32- or 64-bit floating pont numbers with the specified `mean` and positive, real `stddev` using the ICDF method. The function pre-computes and broadcasts loop-invariant constants required for vector generation. The **svrng_update_normal_distribution_[float|double]** functions give the same result, but by modifying existing distributions instead of allocating memory for new distributions.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `stddev`
SVRNG_STATUS_ERROR_BAD_DISTR	Bad distribution (NULL pointer)

Return Values

A pointer to the distribution created or updated by the function, or NULL on error.

svrng_delete_distribution

Deallocates memory for the specified distribution

Syntax

```
svrng_distribution_t svrng_delete_distribution( svrng_distribution_t distr)
```

Input Parameters

distr	Pointer to the distribution to be deallocated.

Description

The **svrng_delete_distribution** function deallocates memory for the specified distribution.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_BAD_DISTR	`distr` is a NULL pointer.

Return Values

NULL pointer.

Random Values Generation

Once the engines and distributions are created by the appropriate initialization routines, the SVRNG generation functions may be called. Both scalar and vector implementations are available. Scalar functions return random values of native "C" types such as `int32_t`, `uint32_t`, `uint64_t`, `float`, and `double`, while vector functions produce packed results in SIMD registers through CPU-specific types (see the "Data Types and Calling Conventions" sections). Calls to scalar SVRNG intrinsics in loops can be vectorized by the compiler via the "vector_variant" feature when the `svrng.h` header file is used. The compiler vectorizer replaces scalar calls by a corresponding SIMD version.

See Also

Data Types and Calling Conventions

svrng_generate[1|2|4|8|16|32]_[uint|ulong]

Generates uniform random bits over the a specified range

Syntax

```
uint32_t svrng_generate_uint( svrng_engine_t engine )
svrng_uint1_t svrng_generate1_uint( svrng_engine_t engine )
svrng_uint2_t svrng_generate2_uint( svrng_engine_t engine )
svrng_uint4_t svrng_generate4_uint( svrng_engine_t engine )
svrng_uint8_t svrng_generate8_uint( svrng_engine_t engine )
svrng_uint16_t svrng_generate16_uint( svrng_engine_t engine )
svrng_uint32_t svrng_generate32_uint( svrng_engine_t engine )
uint64_t svrng_generate_ulong( svrng_engine_t engine )
svrng_ulong1_t svrng_generate1_ulong( svrng_engine_t engine )
svrng_ulong2_t svrng_generate2_ulong( svrng_engine_t engine )
svrng_ulong4_t svrng_generate4_ulong( svrng_engine_t engine )
svrng_ulong8_t svrng_generate8_ulong( svrng_engine_t engine )
svrng_ulong16_t svrng_generate16_ulong( svrng_engine_t engine )
svrng_ulong32_t svrng_generate32_ulong( svrng_engine_t engine )
```

Input Parameters

engine	Pointer to the engine.

Description

The `svrng_generate[n]_[uint|ulong]` functions generate uniform random bits over the range [0, MAX) with engine-dependent maximum value. The uint versions are available for 32-bit engines only (rand0, rand, mcg31m1, mt19937), the ulong versions are available for 64-bit engines only (mcg59). The number `n` if specified expresses the number of packed unsigned integer elements in returned SIMD registers.

Status flags set

Name	**Description**
SVRNG_STATUS_ERROR_UNSUPPORTED	Unmatched engine and result type. See the Description section for supported combinations.
SVRNG_STATUS_ERROR_BAD_ENGINE	Bad engine (NULL pointer)

Return Values

Unsigned integer random value(s). The `svrng_generate_[uint|ulong]` functions return a single unsigned 32- or 64-bit integer random value. The `svrng_generate[n]_[uint|ulong]` functions, for n=1, 2, 4, 8, 16, or 32 return as many unsigned, 32- or 64-bit integer random values packed in a SIMD register.

svrng_generate[1|2|4|8|16|32]_[int|float|double]

Generates distributed random values for the specified engine and distribution

Syntax

`int32_t svrng_generate_int( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_int1_t svrng_generate1_int( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_int2_t svrng_generate2_int( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_int4_t svrng_generate4_int( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_int8_t svrng_generate8_int( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_int16_t svrng_generate16_int( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_int32_t svrng_generate32_int( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`float svrng_generate_float( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_float1_t svrng_generate1_float( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_float2_t svrng_generate2_float( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_float4_t svrng_generate4_float( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_float8_t svrng_generate8_float( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_float16_t svrng_generate16_float( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_float32_t svrng_generate32_float( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`double svrng_generate_double( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_double1_t svrng_generate1_double( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_double2_t svrng_generate2_double( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_double4_t svrng_generate4_double( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_double8_t svrng_generate8_double( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_double16_t svrng_generate16_double( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

`svrng_double32_t svrng_generate32_double( svrng_engine_t` *`engine`*`, svrng_distribution_t` *`distr`* `)`

Input Parameters

`engine`	Pointer to the engine.

distr	Pointer to the distribution.

Description

The **svrng_generate[n]_[int|float|double]** functions generate distributed random values based on the input engine and distribution specified. The output types that are supported—int, float, or double—depend on the distribution used. The number `n` if specified expresses the number of packed elements desired in the returned SIMD registers.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_UNSUPPORTED	Unmatched engine and result type. See the Description section for supported combinations.
SVRNG_STATUS_ERROR_BAD_ENGINE	Bad engine (NULL pointer)
SVRNG_STATUS_ERROR_BAD_DISTR	Bad distribution (NULL pointer)

Return Values

The `svrng_generate_[int|long|double]` functions return a single random value of the specified type. The `svrng_generate[n]_[int|long|double]` functions, for n=1, 2, 4, 8, 16, or 32, return as many signed random values packed in a SIMD register.

Service Routines

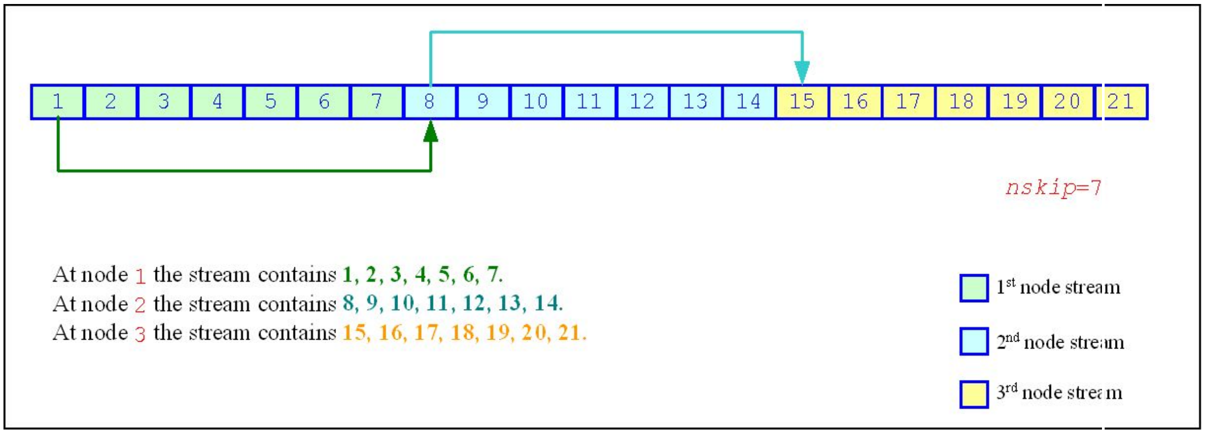
There are two types of service routines available to support the short vector random number generator library:

1. Functions for parallel computations
2. Error handling functions

Parallel Computation Support

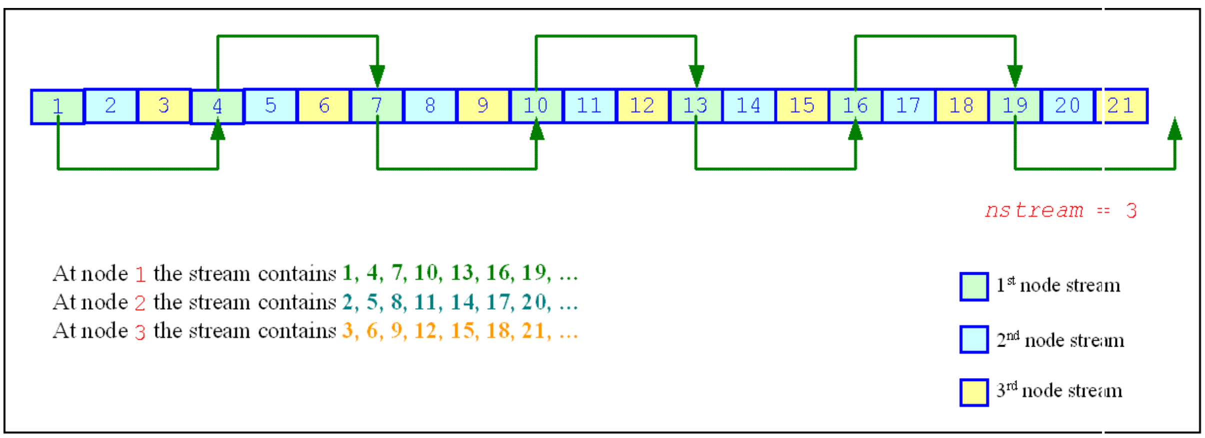
One of the basic requirements for the random number sequences generated by the engines is their mutual independence and lack of inter-correlation. Even if you want random number samplings to be correlated, such correlation should be controllable. The Short Vector Random Number Generator (SVRNG) library provides two techniques: skip-ahead and leap-frog.

Skip-ahead	The skip-ahead method splits the original sequence into *k* non-overlapping blocks, where *k* is the number of independent sequences. Each of the sequences generates random numbers only from the corresponding block of contiguous random numbers.



Leap-frog

The leap-frog method splits the original sequence into *k* disjoint subsequences in such a way that the first stream would generate the random numbers x_1, x_{k+1}, x_{2k+1}, x_{3k+1}, ..., the second stream would generate the random numbers x_2, x_{k+2}, x_{2k+2}, x_{3k+2}, ..., and, finally, the *k*-th stream would generate the random numbers x_k, x_{2k}, x_{3k}, ... The multi-dimensional uniformity properties of each subsequence deteriorate seriously as *k* grows so this method is only useful if *k* is less than about 25.



The following sequence outlines the typical usage model for creating independent sequences of random numbers in a parallel computation environment:

- Create the original engine
- Create a copy of the original engine in each thread
- Apply one of techniques above to re-initialize the individual engines to provide an independent sequence on each thread

For detailed information on the use of SVRNG intrinsics in a parallel computation environment see the Random Streams and RNGs in Parallel Computation section of the *Notes for Intel® oneAPI Math Kernel Library Vector Statistics* document listed in Intrinsics for the Short Vector Random Number Generator Library.

Note: Currently skip-ahead and leap-frog methods are supported by the rand0, rand, mcg31m1, and mcg59 engines. The skip-ahead and leap-frog methods of splitting a stream are not yet implemented for the ***mt19937*** *engine, but* ***mt19937*** *naturally provides parallel support during initialization. See the MT19937 section of the Notes for Intel® oneAPI Math Kernel Library Vector Statistics document listed in the introduction. .*

See Also

Intrinsics for the Short Vector Random Number Generator Library

svrng_copy_engine

Allocates memory for a new engine and copies over all parameters

Syntax

```
svrng_engine_t svrng_copy_engine( svrng_engine_t orig_engine )
```

Input Parameters

orig_engine	Pointer to the engine to be copied

Description

The `svrng_copy_engine` function allocates memory for a new engine then copies all parameters from original engine to the new engine.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_BAD_ENGINE	Bad engine (NULL pointer)

Return Values

Pointer to the newly created copy of the original engine, or NULL on error.

svrng_skipahead_engine

Re-initialize engine parameters for use of the skip-ahead method

Syntax

```
svrng_engine_t svrng_skipahead_engine( svrng_engine_t orig_engine, long long nskip )
```

Input Parameters

orig_engine	Pointer to the engine to be re-initialized using the skip-ahead technique
nskip	Number of skipped elements

Description

Re-initializes engine parameters using the block-splitting ("skip-ahead") method. The function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is `nskip`, then the original random sequence may be split by this function into non-overlapping blocks of `nskip` size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_NON_SUPPORTED	Memory allocation procedure failure
SVRNG_STATUS_ERROR_BAD_ENGINE	Bad engine (NULL pointer)
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `nskip`

Return Values

Pointer to the same input engine or NULL on error

svrng_leapfrog_engine

Re-initialize engine parameters for use of the leap-frog method

Syntax

```
svrng_engine_t svrng_leapfrog_engine( svrng_engine_t orig_engine, int k, int nstreams )
```

Input Parameters

orig_engine	Pointer to the engine to be re-initialized using the leap-frog technique.
k	Index of the computational node, or sequence number.
nstreams	Largest number of computational nodes, or stride.

Description

The `svrng_skipahead_engine` function re-initializes the engine parameters using the leap-frog method. The leap-frogged engine generates random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the *nstreams* buffers without generating the original random sequence with subsequent manual distribution.

Status flags set

Name	Description
SVRNG_STATUS_ERROR_UNSUPPORTED	Function or method non supported
SVRNG_STATUS_ERROR_BAD_ENGINE	Bad engine (NULL pointer)
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter: `k`
SVRNG_STATUS_ERROR_BAD_PARAM3	Bad parameter: `nstreams`

Return Values

Pointer to the same input engine or NULL on error

Error Handling

The Short Vector Random Number Generator (SVRNG) library supports error handling via status variables and corresponding set and get functions. NULL pointers are returned for errors when possible. The following table contains the status constants defined in `svrng.h`:

Macro Name	Description
SVRNG_STATUS_OK	No errors
SVRNG_STATUS_ERROR_BAD_PARAM1	Bad parameter #1
SVRNG_STATUS_ERROR_BAD_PARAM2	Bad parameter #2
SVRNG_STATUS_ERROR_BAD_PARAM3	Bad parameter #3
SVRNG_STATUS_ERROR_BAD_PARAM4	Bad parameter #4
SVRNG_STATUS_ERROR_BAD_PARAMS	Bad combination of parameters
SVRNG_STATUS_ERROR_BAD_ENGINE	Bad engine (NULL pointer)
SVRNG_STATUS_ERROR_BAD_DISTR	Bad distribution (NULL pointer)
SVRNG_STATUS_ERROR_MEMORY_ALLOC	Memory allocation failure
SVRNG_STATUS_ERROR_UNSUPPORTED	Function or method not supported

svrng_set_status

Sets the status variable to a specified value and returns the previous status value

Syntax

```
int32_t svrng_set_status( int32_t new_status )
```

Input Parameters

new_status	The new status.

Description

The `svrng_set_status` function sets the status variable to a specific constant value and returns the previous status value. See the Error Handling page for a table of values defined in `svrng.h`.

Return Values

Returns the previous status value.

svrng_get_status

Returns the current status value

Syntax

`int32_t svrng_get_status()`

Description

The `svrng_get_status` function returns the current status value.

Return Values

The current status value.

Intrinsics for Instruction Set Architecture (ISA) Instructions

SERIALIZE

_serialize

Synopsis

```
void _serialize ()
```

Header file	#include <immintrin.h>
Instruction	SERIALIZE
CPUID flags	SERIALIZE

Description

Serialize instruction execution, ensuring all modifications to flags, registers, and memory by previous instructions are completed before the next instruction is fetched.

Technology

Other

Category

General Support

TSXLDTRK

_xresldtrk

Synopsis

```
void _xresldtrk ()
```

Header file	#include <immintrin.h>
Instruction	XRESLDTRK
CPUID flags	TSXLDTRK

Description

Mark the end of a TSX (HLE/RTM) suspend load address tracking region. If this is used inside a suspend load address tracking region it will end the suspend region and all following load addresses will be added to the transaction read set. If this is used inside an active transaction but not in a suspend region it will cause transaction abort. If this is used outside of a transactional region it behaves like a NOP.

Technology

Other

Category

Miscellaneous

_xsusldtrk

Synopsis

```
void _xsusldtrk ()
```

Header file	`#include <immintrin.h>`
Instruction	`XSUSLDTRK`
CPUID flags	`TSXLDTRK`

Description

Mark the start of a TSX (HLE/RTM) suspend load address tracking region. If this is used inside a transactional region, subsequent loads are not added to the read set of the transaction. If this is used inside a suspend load address tracking region it will cause transaction abort. If this is used outside of a transactional region it behaves like a NOP.

Technology

Other

Category

Miscellaneous

Intrinsics for Intel® Advanced Matrix Extensions (Intel(R) AMX) Instructions

Intel® Advanced Matrix Extensions (Intel® AMX) is a new 64-bit programming paradigm consisting of two components:

- A set of 2-dimensional registers (tiles) representing sub-arrays from a larger 2-dimensional memory image
- An accelerator that is able to operate on tiles; the first implementation of this accelerator is called TMUL (tile matrix multiply unit).

The following sections show intrinsics that are available for Intel(R) Advanced Matrix Extension Instructions.

Intrinsic for Intel® Advanced Matrix Extensions AMX-BF16 Instructions

This intrinsic supports tile computational operations on bfloat16 numbers.

_tile_dpbf16ps

Synopsis

```
void _tile_dpbf16ps (__tile dst, __tile a, __tile b)
```

Type	TileFloating Point
Header file	`#include <immintrin.h>`
Instruction	`TDPBF16PS tmm, tmm, tmm`
CPUID flags	`AMXBF16`

Description

Compute dot-product of BF16 (16-bit) floating-point pairs in tiles "a" and "b", accumulating the intermediate single-precision (32-bit) floating-point elements with elements in "dst", and store the 32-bit result back to tile "dst".

Technology

AMX

Category

Application-Targeted

Operation

```
FOR m := 0 TO dst.rows - 1
    tmp := dst.row[m]
    FOR k := 0 TO (a.colsb / 4) - 1
        FOR n := 0 TO (dst.colsb / 4) - 1
            tmp.fp32[n] += FP32(a.row[m].bf16[2*k+0]) * FP32(b.row[k].bf16[2*n+0])
            tmp.fp32[n] += FP32(a.row[m].bf16[2*k+1]) * FP32(b.row[k].bf16[2*n+1])
        ENDFOR
    ENDFOR
    write_row_and_zero(dst, m, tmp, dst.colsb)
ENDFOR

zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

Intrinsics for Intel® Advanced Matrix Extensions AMX-INT8 Instructions

These intrinsics support tile computational operations on 8-bit integers.

_tile_dpbssd

Synopsis

```
void _tile_dpbssd (__tile dst, __tile a, __tile b)
```

Type	Tile
Header file	`#include <immintrin.h>`

Instruction	TDPBSSD tmm, tmm, tmm
CPUID flags	AMXINT8

Description

Compute dot-product of bytes in tiles with a source/destination accumulator. Multiply groups of 4 adjacent pairs of signed 8-bit integers in "a" with corresponding signed 8-bit integers in "b", producing 4 intermediate 32-bit results. Sum these 4 results with the corresponding 32-bit integer in "dst", and store the 32-bit result back to tile "dst".

Technology

AMX

Category

Application-Targeted

Operation

```
DEFINE DPBD(c, x, y) {
    tmp1 := SignExtend32(x.byte[0]) * SignExtend32(y.byte[0])
    tmp2 := SignExtend32(x.byte[1]) * SignExtend32(y.byte[1])
    tmp3 := SignExtend32(x.byte[2]) * SignExtend32(y.byte[2])
    tmp4 := SignExtend32(x.byte[3]) * SignExtend32(y.byte[3])

    RETURN c + tmp1 + tmp2 + tmp3 + tmp4
}

FOR m := 0 TO dst.rows - 1
    tmp := dst.row[m]
    FOR k := 0 TO (a.colsb / 4) - 1
        FOR n := 0 TO (dst.colsb / 4) - 1
            tmp.dword[n] := DPBD(tmp.dword[n], a.row[m].dword[k], b.row[k].dword[n])
        ENDFOR
    ENDFOR
    write_row_and_zero(dst, m, tmp, dst.colsb)
ENDFOR

zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

_tile_dpbsud

Synopsis

```
void _tile_dpbsud (__tile dst, __tile a, __tile b)
```

Type	Tile
Header file	#include <immintrin.h>
Instruction	TDPBSUD tmm, tmm, tmm
CPUID flags	AMXINT8

Description

Compute dot-product of bytes in tiles with a source/destination accumulator. Multiply groups of 4 adjacent pairs of signed 8-bit integers in "a" with corresponding unsigned 8-bit integers in "b", producing 4 intermediate 32-bit results. Sum these 4 results with the corresponding 32-bit integer in "dst", and store the 32-bit result back to tile "dst".

Technology

AMX

Category

Application-Targeted

Operation

```
DEFINE DPBD(c, x, y) {
    tmp1 := SignExtend32(x.byte[0]) * ZeroExtend32(y.byte[0])
    tmp2 := SignExtend32(x.byte[1]) * ZeroExtend32(y.byte[1])
    tmp3 := SignExtend32(x.byte[2]) * ZeroExtend32(y.byte[2])
    tmp4 := SignExtend32(x.byte[3]) * ZeroExtend32(y.byte[3])

    RETURN c + tmp1 + tmp2 + tmp3 + tmp4
}

FOR m := 0 TO dst.rows - 1
    tmp := dst.row[m]
    FOR k := 0 TO (a.colsb / 4) - 1
        FOR n := 0 TO (dst.colsb / 4) - 1
            tmp.dword[n] := DPBD(tmp.dword[n], a.row[m].dword[k], b.row[k].dword[n])
        ENDFOR
    ENDFOR
    write_row_and_zero(dst, m, tmp, dst.colsb)
ENDFOR

zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

_tile_dpbusd

Synopsis

```
void _tile_dpbusd (__tile dst, __tile a, __tile b)
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`TDPBUSD tmm, tmm, tmm`
CPUID flags	`AMXINT8`

Description

Compute dot-product of bytes in tiles with a source/destination accumulator. Multiply groups of 4 adjacent pairs of unsigned 8-bit integers in "a" with corresponding signed 8-bit integers in "b", producing 4 intermediate 32-bit results. Sum these 4 results with the corresponding 32-bit integer in "dst", and store the 32-bit result back to tile "dst".

Technology

AMX

Category

Application-Targeted

Operation

```
DEFINE DPBD(c, x, y) {
    tmp1 := ZeroExtend32(x.byte[0]) * SignExtend32(y.byte[0])
    tmp2 := ZeroExtend32(x.byte[1]) * SignExtend32(y.byte[1])
    tmp3 := ZeroExtend32(x.byte[2]) * SignExtend32(y.byte[2])
    tmp4 := ZeroExtend32(x.byte[3]) * SignExtend32(y.byte[3])

    RETURN c + tmp1 + tmp2 + tmp3 + tmp4
}

FOR m := 0 TO dst.rows - 1
    tmp := dst.row[m]
    FOR k := 0 TO (a.colsb / 4) - 1
        FOR n := 0 TO (dst.colsb / 4) - 1
            tmp.dword[n] := DPBD(tmp.dword[n], a.row[m].dword[k], b.row[k].dword[n])
        ENDFOR
    ENDFOR
    write_row_and_zero(dst, m, tmp, dst.colsb)
ENDFOR

zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

_tile_dpbuud

Synopsis

```
void _tile_dpbuud (__tile dst, __tile a, __tile b)
```

Type	Tile
Header file	#include <immintrin.h>
Instruction	TDPBUUD tmm, tmm, tmm
CPUID flags	AMXINT8

Description

Compute dot-product of bytes in tiles with a source/destination accumulator. Multiply groups of 4 adjacent pairs of unsigned 8-bit integers in "a" with corresponding unsigned 8-bit integers in "b", producing 4 intermediate 32-bit results. Sum these 4 results with the corresponding 32-bit integer in "dst", and store the 32-bit result back to tile "dst".

Technology

AMX

Category

Application-Targeted

Operation

```
DEFINE DPBD(c, x, y) {
    tmp1 := ZeroExtend32(x.byte[0]) * ZeroExtend32(y.byte[0])
    tmp2 := ZeroExtend32(x.byte[1]) * ZeroExtend32(y.byte[1])
    tmp3 := ZeroExtend32(x.byte[2]) * ZeroExtend32(y.byte[2])
    tmp4 := ZeroExtend32(x.byte[3]) * ZeroExtend32(y.byte[3])

    RETURN c + tmp1 + tmp2 + tmp3 + tmp4
}

FOR m := 0 TO dst.rows - 1
    tmp := dst.row[m]
    FOR k := 0 TO (a.colsb / 4) - 1
        FOR n := 0 TO (dst.colsb / 4) - 1
            tmp.dword[n] := DPBD(tmp.dword[n], a.row[m].dword[k], b.row[k].dword[n])
        ENDFOR
    ENDFOR
    write_row_and_zero(dst, m, tmp, dst.colsb)
ENDFOR

zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

Intrinsics for Intel(R) Advanced Matrix Extensions AMX-TILE Instructions

These intrinsics support tile architecture.

_tile_loadconfig

Synopsis

```
void _tile_loadconfig (const void * mem_addr)
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`LDTILECFG m512`
CPUID flags	`AMXTILE`

Description

Load tile configuration from a 64-byte memory location specified by "mem_addr". The tile configuration format is specified below, and includes the tile type pallette, the number of bytes per row, and the number of rows. If the specified pallette_id is zero, that signifies the init state for both the tile config and the tile data, and the tiles are zeroed. Any invalid configurations will result in #GP fault.

Technology

AMX

Category

Application-Targeted

Operation

```
//    format of memory payload. each field is a byte.
//         0: palette_id
//         1: startRow (8b)
//     2-15: reserved (must be zero)
//    16-17: tile0.colsb -- bytes_per_row
//    18-19: tile1.colsb
//    20-21: tile2.colsb
//            ...
//    46-47: tile15.colsb
//        48: tile0.rows
//        49: tile1.rows
//        50: tile2.rows
//             ...
//        63: tile15.rows
```

_tile_loadd

Synopsis

```
void _tile_loadd (__tile dst, const void * base, int stride)
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`TILELOADD tmm, sibmem`
CPUID flags	`AMXTILE`

Description

Load tile rows from memory specifieid by "base" address and "stride" into destination tile "dst" using the tile configuration previously configured via "_tile_loadconfig".

Technology

AMX

Category

Application-Targeted

Operation

```
start := tileconfig.startRow
IF start == 0 // not restarting, zero incoming state
    tilezero(dst)
FI

nbytes := dst.colsb
DO WHILE start < dst.rows
    memptr := base + start * stride
    write_row_and_zero(dst, start, read_memory(memptr, nbytes), nbytes)
```

```
    start := start + 1
OD

zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

_tile_release

Synopsis

```
void _tile_release ()
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`TILERELEASE`
CPUID flags	`AMXTILE`

Description

Release the tile configuration to return to the init state, which releases all storage it currently holds.

Technology

AMX

Category

Application-Targeted

_tile_storeconfig

Synopsis

```
void _tile_storeconfig (void * mem_addr)
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`STTILECFG m512`
CPUID flags	`AMXTILE`

Description

Stores the current tile configuration to a 64-byte memory location specified by "mem_addr". The tile configuration format is specified below, and includes the tile type pallette, the number of bytes per row, and the number of rows. If tiles are not configured, all zeroes will be stored to memory.

Technology

AMX

Category

Application-Targeted

Operation

```
//    format of memory payload. each field is a byte.
//         0: palette_id
//         1: startRow (8b)
//     2-15: reserved (must be zero)
//    16-17: tile0.colsb -- bytes_per_row
//    18-19: tile1.colsb
//    20-21: tile2.colsb
//            ...
//    46-47: tile15.colsb
//        48: tile0.rows
//        49: tile1.rows
//        50: tile2.rows
//             ...
//        63: tile15.rows
```

_tile_stored

Synopsis

```
void _tile_stored (__tile src, void * base, int stride)
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`TILESTORED sibmem, tmm`
CPUID flags	`AMXTILE`

Description

Store the tile specified by "src" to memory specifieid by "base" address and "stride" using the tile configuration previously configured via "_tile_loadconfig".

Technology

AMX

Category

Application-Targeted

Operation

```
start := tileconfig.startRow
DO WHILE start < src.rows
    memptr := base + start * stride
    write_memory(memptr, src.colsb, src.row[start])
    start := start + 1
OD
zero_tileconfig_start()
```

_tile_stream_loadd

Synopsis

```
void _tile_stream_loadd (__tile dst, const void * base, int stride)
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`TILELOADDT1 tmm, sibmem`
CPUID flags	`AMXTILE`

Description

Load tile rows from memory specifieid by "base" address and "stride" into destination tile "dst" using the tile configuration previously configured via "_tile_loadconfig". This intrinsic provides a hint to the implementation that the data will likely not be reused in the near future and the data caching can be optimized accordingly.

Technology

AMX

Category

Application-Targeted

Operation

```
start := tileconfig.startRow
IF start == 0 // not restarting, zero incoming state
    tilezero(dst)
FI

nbytes := dst.colsb
DO WHILE start < dst.rows
    memptr := base + start * stride
    write_row_and_zero(dst, start, read_memory(memptr, nbytes), nbytes)
    start := start + 1
OD

zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

_tile_zero

Synopsis

```
void _tile_zero (__tile tdest)
```

Type	Tile
Header file	`#include <immintrin.h>`
Instruction	`TILEZERO tmm`
CPUID flags	`AMXTILE`

Description

Zero the tile specified by "tdest".

Technology

AMX

Category

Application-Targeted

Operation

```
nbytes := palette_table[tileconfig.palette_id].bytes_per_row
FOR i := 0 TO palette_table[tileconfig.palette_id].max_rows-1
    FOR j := 0 TO nbytes-1
        tdest.row[i].byte[j] := 0
    ENDFOR
ENDFOR
```

Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) BF16 Instructions

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) BF16 instruction intrinsics are located in the zmmintrin.h header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
a	a source vector element
b	a second source vector element
k	mask used as a selector; depending on the intrinsic, it may be a writemask or a zeromask

_mm_cvtne2ps_pbh

```
__m128bh _mm_cvtne2ps_pbh (__m128 a, __m128 b)
```

Instructions: vcvtne2ps2bf16 xmm, xmm, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst.

_mm_mask_cvtne2ps_pbh

```
__m128bh _mm_mask_cvtne2ps_pbh (__m128bh src, __mmask8 k, __m128 a, __m128 b)
```

Instructions: vcvtne2ps2bf16 xmm {k}, xmm, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm_maskz_cvtne2ps_pbh

```
__m128bh _mm_maskz_cvtne2ps_pbh (__mmask8 k, __m128 a, __m128 b)
```

Instructions: vcvtne2ps2bf16 xmm {k}{z}, xmm, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set.

_mm256_cvtne2ps_pbh

```
__m256bh _mm256_cvtne2ps_pbh (__m256 a, __m256 b)
```

Instructions: vcvtne2ps2bf16 ymm, ymm, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst.

_mm256_mask_cvtne2ps_pbh

```
__m256bh _mm256_mask_cvtne2ps_pbh (__m256bh src, __mmask16 k, __m256 a, __m256 b)
```

Instructions: vcvtne2ps2bf16 ymm {k}, ymm, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm256_maskz_cvtne2ps_pbh

```
__m256bh _mm256_maskz_cvtne2ps_pbh (__mmask16 k, __m256 a, __m256 b)
```

Instructions: vcvtne2ps2bf16 ymm {k}{z}, ymm, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and store the results in single vector dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set.

_mm512_cvtne2ps_pbh

```
__m512bh _mm512_cvtne2ps_pbh (__m512 a, __m512 b)
```

Instructions: vcvtne2ps2bf16 zmm, zmm, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst.

_mm512_mask_cvtne2ps_pbh

```
__m512bh _mm512_mask_cvtne2ps_pbh (__m512bh src, __mmask32 k, __m512 a, __m512 b)
```

Instructions: vcvtne2ps2bf16 zmm {k}, zmm, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm512_maskz_cvtne2ps_pbh

```
__m512bh _mm512_maskz_cvtne2ps_pbh (__mmask32 k, __m512 a, __m512 b)
```

Instructions: vcvtne2ps2bf16 zmm {k}{z}, zmm, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Converts packed single-precision (32-bit) floating-point elements in two vectors *a* and *b* to packed BF16 (16-bit) floating-point elements, and stores the results in a single vector dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set.

_mm_cvtneps_pbh

```
__m128bh _mm_cvtneps_pbh (__m128 a)
```

Instructions: vcvtneps2bf16 xmm, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst.

_mm_mask_cvtneps_pbh

```
__m128bh _mm_mask_cvtneps_pbh (__m128bh src, __mmask8 k, __m128 a)
```

Instructions: vcvtneps2bf16 xmm {k}, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm_maskz_cvtneps_pbh

```
__m128bh _mm_maskz_cvtneps_pbh (__mmask8 k, __m128 a)
```

Instructions: vcvtneps2bf16 xmm {k}{z}, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set.

_mm256_cvtneps_pbh

```
__m128bh _mm256_cvtneps_pbh (__m256 a)
```

Instructions: vcvtneps2bf16 xmm, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst.

_mm256_mask_cvtneps_pbh

```
__m128bh _mm256_mask_cvtneps_pbh (__m128bh src, __mmask8 k, __m256 a)
```

Instructions: vcvtneps2bf16 xmm {k}, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm256_maskz_cvtneps_pbh

```
__m128bh _mm256_maskz_cvtneps_pbh (__mmask8 k, __m256 a)
```

Instructions: vcvtneps2bf16 xmm {k}{z}, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set.

_mm512_cvtneps_pbh

```
__m256bh _mm512_cvtneps_pbh (__m512 a)
```

Instructions: vcvtneps2bf16 ymm, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst.

_mm512_mask_cvtneps_pbh

```
__m256bh _mm512_mask_cvtneps_pbh (__m256bh src, __mmask16 k, __m512 a)
```

Instructions: vcvtneps2bf16 ymm {k}, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm512_maskz_cvtneps_pbh

```
__m256bh _mm512_maskz_cvtneps_pbh (__mmask16 k, __m512 a)
```

Instructions: vcvtneps2bf16 ymm {k}{z}, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Converts packed single-precision (32-bit) floating-point elements in *a* to packed BF16 (16-bit) floating-point elements, and stores the results in dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set.

_mm_dpbf16_ps

```
__m128 _mm_dpbf16_ps (__m128 src, __m128bh a, __m128bh b)
```

Instructions: vdpbf16ps xmm, xmm, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst.

_mm_mask_dpbf16_ps

```
__m128 _mm_mask_dpbf16_ps (__m128 src, __mmask8 k, __m128bh a, __m128bh b)
```

Instructions: vdpbf16ps xmm {k}, xmm, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm_maskz_dpbf16_ps

```
__m128 _mm_maskz_dpbf16_ps (__mmask8 k, __m128 src, __m128bh a, __m128bh b)
```

Instructions: vdpbf16ps xmm {k}{z}, xmm, xmm

CPUID Flags: AVX512_BF16 + AVX512VL

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set).

_mm256_dpbf16_ps

```
__m256 _mm256_dpbf16_ps (__m256 src, __m256bh a, __m256bh b)
```

Instructions: vdpbf16ps ymm, ymm, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst.

_mm256_mask_dpbf16_ps

```
__m256 _mm256_mask_dpbf16_ps (__m256 src, __mmask8 k, __m256bh a, __m256bh b)
```

Instructions: vdpbf16ps ymm {k}, ymm, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm256_maskz_dpbf16_ps

```
__m256 _mm256_maskz_dpbf16_ps (__mmask8 k, __m256 src, __m256bh a, __m256bh b)
```

Instructions: vdpbf16ps ymm {k}{z}, ymm, ymm

CPUID Flags: AVX512_BF16 + AVX512VL

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set).

_mm512_dpbf16_ps

```
__m512 _mm512_dpbf16_ps (__m512 src, __m512bh a, __m512bh b)
```

Instructions: vdpbf16ps zmm, zmm, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst.

_mm512_mask_dpbf16_ps

```
__m512 _mm512_mask_dpbf16_ps (__m512 src, __mmask16 k, __m512bh a, __m512bh b)
```

Instructions: vdpbf16ps zmm {k}, zmm, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst using writemask *k*. Elements are copied from src when the corresponding mask bit is not set.

_mm512_maskz_dpbf16_ps

```
__m512 _mm512_maskz_dpbf16_ps (__mmask16 k, __m512 src, __m512bh a, __m512bh b)
```

Instructions: vdpbf16ps zmm {k}{z}, zmm, zmm

CPUID Flags: AVX512_BF16 + AVX512F

Computes the dot-product of BF16 (16-bit) floating-point pairs in *a* and *b*, accumulating the intermediate single-precision (32-bit) floating-point elements with elements in src, and stores the results in dst using zeromask *k*. Elements are zeroed out when the corresponding mask bit is not set.

Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) 4VNNIW Instructions

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) 4VNNIW instruction intrinsics are located in the zmmintrin.h header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm512_4dpwssd_epi32

```
__mm512i _mm512_4dpwssd_epi32 (__m512 c, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator

Instructions: vp4dpwssd zmm1, zmm2+3, m128

Computes 4 vector source-block dot-products of two signed word operands with doubleword accumulation in *c*. The memory operand is sequentially selected in each of the four steps.

_mm512_mask_4dpwssd_epi32

```
__mm512i _mm512_mask_4dpwssd_epi32 (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2,
__m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: vp4dpwssd zmm1 {k}, zmm2+3, m128

Computes 4 vector source-block dot-products of two signed word operands with doubleword accumulation using mask *k*, with accumulation in *c*. The memory operand is sequentially selected in each of the four steps. Elements are copied from *c* when the corresponding mask bit is not set.

_mm512_maskz_4dpwssd_epi32

```
__mm512i _mm512_maskz_4dpwssd_epi32 (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2,
__m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: vp4dpwssd zmm1 {k}, zmm2+3, m128

Computes 4 vector source-block dot-products of two signed word operands with doubleword accumulation using mask *k*, with accumulation in *c*. The memory operand is sequentially selected in each of the four steps. Elements are zeroed out when the corresponding mask bit is not set.

_mm512_4dpwssds_epi32

```
__mm512i _mm512_4dpwssds_epi32 (__m512 c, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator

Instructions: vp4dpwssds zmm1, zmm2+3, m128

Computes 4 vector source-block dot-products of two signed word operands with doubleword accumulation and signed saturation in *c*. The memory operand is sequentially selected in each of the four steps.

_mm512_mask_4dpwssds_epi32

```
__mm512i _mm512_mask_4dpwssds_epi32 (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2,
__m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors

variable	definition
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: vp4dpwssds zmm1 {k}, zmm2+3, m128

Computes 4 vector source-block dot-products of two signed word operands with doubleword accumulation and signed saturation using mask *k*, with accumulation in *c*. The memory operand is sequentially selected in each of the four steps. Elements are copied from *c* when the corresponding mask bit is not set.

_mm512_maskz_4dpwssds_epi32

```
__mm512i _mm512_maskz_4dpwssds_epi32 (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2,
__m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: vp4dpwssds zmm1 {k}, zmm2+3, m128

Computes 4 vector source-block dot-products of two signed word operands with doubleword accumulation and signed saturation using mask *k*, with accumulation in *c*. The memory operand is sequentially selected in each of the four steps. Elements are zeroed out when the corresponding mask bit is not set.

Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) 4FMAPS Instructions

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) 4FMAPS instruction intrinsics are located in the zmmintrin.h header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm512_4fmadd_ps

```
__mm512i _mm512_4fmadd_ps (__m512 c, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator

Instructions: v4fmaddps zmm1, zmm2+3, m128

Multiplies packed single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} by floating-point values pointed to by *b* and accumulates the result in *c*.

_mm512_mask_4fmadd_ps

```
__mm512i _mm512_mask_4fmadd_ps (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512
a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fmaddps zmm1 {k}, zmm2+3, m128

Multiplies packed single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the result in *c*. Elements are copied from *c* when the corresponding mask bit is not set.

_mm512_maskz_4fmadd_ps

```
__mm512i _mm512 _maskz_4fmadd_ps (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512
a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fmaddps zmm {k}, zmm+3, m128

Multiplies packed single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the result in *c*. Elements are zeroed out when the corresponding mask bit is not set.

_mm512_4fnmadd_ps

```
__mm512i _mm512_4fnmadd_ps (__m512 c, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator

Instructions: v4fnmaddps zmm1, zmm2+3, m128

Multiplies and negates packed single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} by floating-point values pointed to by *b* and accumulates the result in *c*.

_mm512_mask_4fnmadd_ps

```
__mm512i _mm512_mask_4fnmadd_ps (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512
a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors

variable	definition
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fnmaddps zmm1 {k}, zmm2+3, m128

Multiplies and negates packed single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the result in *c*. Elements are copied from *c* when the corresponding mask bit is not set.

_mm512_maskz_4fnmadd_ps

```
__mm512i _mm512_maskz_4fnmadd_ps (__m512 c, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512
a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fnmaddps zmm1 {k}, zmm2+3, m128

Multiplies and negates packed single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the result in *c*. Elements are zeroed out when the corresponding mask bit is not set.

_mm_4fmadd_ss

```
__mm512i _mm_4fmadd_ss (__m128 c, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator

Instructions: v4fmaddss xmm1, xmm2+3, m128

Multiplies the lower packed scalar single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} by floating-point values pointed to by *b* and accumulates the lower element result in *c*.

_mm_mask_4fmadd_ss

```
__mm512i _mm_mask_4fmadd_ss (__m128 c, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3,
__m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fmaddss xmm1 {k}, xmm2+3, m128

Multiplies the lower packed scalar single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the lower element result in *c*. Elements are copied from *c* when the corresponding mask bit is not set.

_mm_maskz_4fmadd_ss

```
__mm512i _mm_maskz_4fmadd_ss (__m128 c, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3,
__m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fmaddss xmm1 {k}, xmm2+3, m128

Multiplies the lower packed scalar single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the lower element result in *c*. Elements are zeroed out when the corresponding mask bit is not set.

_mm_4fnmadd_ss

```
__mm512i _mm_4fnmadd_ss (__m128 c, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator

Instructions: v4fnmaddss xmm1, xmm2+3, m128

Multiplies and negates the lower packed scalar single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} by floating-point values pointed to by *b* and accumulates the lower element result in *c*.

_mm_mask_4fnmadd_ss

```
__mm512i _mm_mask_4fnmadd_ss (__m128 c, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3,
__m128 * b)
```

variable	definition
a*n*	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fnmaddss xmm1 {k}, xmm2+3, m128

Multiplies and negates the lower packed scalar single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the lower element result in *c*. Elements are copied from *c* when the corresponding mask bit is not set.

_mm_maskz_4fnmadd_ss

```
__mm512i _mm_maskz_4fnmadd_ss (__m128 c, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3,
__m128 * b)
```

variable	definition
an	first source block 4 vectors
b	pointer to the second source block
c	third source; accumulator
k	mask used as a selector

Instructions: v4fnmaddss xmm1 {k}, xmm2+3, m128

Multiplies and negates the lower packed scalar single-precision floating-point values from source register block {*a*0, *a*1, *a*2, *a*3} using mask *k* by floating-point values pointed to by *b* and accumulates the lower element result in *c*. Elements are zeroed out when the corresponding mask bit is not set.

Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) VPOPCNTDQ Instructions

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) VPOPCNTDQ instruction intrinsics are located in the zmmintrin.h header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm512_popcnt_epi32

```
__mm512i _mm512_popcnt_epi32 (__m512i a)
```

variable	definition
a	a source vector

Instructions: vpopcntd zmm1, zmm2

Counts the number of bits set to one in each dword element of *a* and places it in the corresponding elements of the result.

_mm512_mask_popcnt_epi32

```
__mm512i _mm512_mask_popcnt_epi32 (__m512 b, __mmask16 k, __m512i a)
```

variable	definition
a	a source vector
b	a second source vector
k	mask used as a selector

Instructions: vpopcntd zmm1 {k}, zmm2

Counts the number of bits set to one in each dword element of *a* using mask *k* and places it in the corresponding elements of the result. Elements are copied from *b* when the corresponding mask bit is not set.

_mm512_maskz_popcnt_epi32

```
__mm512i _mm512_maskz_popcnt_epi32 (__mmask16 k, __m512i a)
```

variable	definition
a	a source vector
k	mask used as a selector

Instructions: vpopcntd zmm1 {k}, zmm2

Counts the number of bits set to one in each dword element of *a* using mask *k* and places it in the corresponding elements of the result. Elements are zeroed out when the corresponding mask bit is not set.

_mm512_popcnt_epi64

```
__mm512i _mm512_popcnt_epi64 (__m512i a)
```

variable	definition
a	a source vector

Instructions: vpopcntd zmm1, zmm2

Counts the number of bits set to one in each quad word element of *a* and places it in the corresponding elements of the result.

_mm512_mask_popcnt_epi64

```
__mm512i _mm512_mask_popcnt_epi64 (__m512 b, __mmask16 k, __m512i a)
```

variable	definition
a	a source vector
b	a second source vector
k	mask used as a selector

Instructions: vpopcntd zmm1 {k}, zmm2

Counts the number of bits set to one in each quad word element of *a* using mask *k* and places it in the corresponding elements of the result. Elements are copied from *b* when the corresponding mask bit is not set.

_mm512_maskz_popcnt_epi64

```
__mm512i _mm512_maskz_popcnt_epi64 (__m512 b, __mmask16 k, __m512 a)
```

variable	definition
a	a source vector
b	a second source vector
k	mask used as a selector

Instructions: vpopcntd zmm2 {k}, zmm2

Counts the number of bits set to one in each quad word element of *a* using mask *k* and places it in the corresponding elements of the result. Elements are zeroed out when the corresponding mask bit is not set.

Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Additional Instructions

Additional Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Instructions

The additional instructions documented in this section enrich the operations available as part of Intel® AVX-512 Foundation instructions. A large portion of these instructions can be divided into two groups: Byte and Word Instructions, and Doubleword and Quadword Instructions. The group of byte and word (8 and 16-bit) operations, indicated by the AVX512BW and AVX512VBMI CPUID flags, enhance small integer operations. The group of doubleword and quadword (32 and 64-bit) operations indicated by the AVX512DQ and AVX512IFMA52 CPUID flags, enhance integer and floating-point operations.

An additional orthogonal capability known as Vector Length Extensions provide for most AVX-512 instructions to operate on 128 or 256 bits, instead of only 512. Vector Length Extensions can currently be applied to most Foundation Instructions, the Conflict Detection Instructions as well as the new byte, word, doubleword and quadword instructions. These AVX-512 Vector Length Extensions are indicated by the AVX512VL CPUID flag. The use of Vector Length Extensions extends most AVX-512 operations to also operate on XMM (128-bit, SSE) registers and YMM (256-bit, AVX) registers. The use of Vector Length Extensions allows the capabilities of EVEX encodings, including the use of mask registers and access to registers 16..31, to be applied to XMM and YMM registers instead of only to ZMM registers.

Byte and Word Instructions

The byte and word instructions, indicated by the AVX512BW CPUID flag, extend write-masking and zero-masking to support smaller element sizes. The original AVX-512 Foundation instructions supported such masking with vector element sizes of 32 or 64 bits. As a 512-bit vector register could hold at most 16 32-bit elements, a write-mask size of 16 bits was sufficient.

With an instruction indicated by an AVX512BW CPUID flag, a 512-bit vector can hold 64 8-bit elements or 32 16-bit elements, so write masks must be able to hold 64 bits. To support this, two new mask types, `__mmask32` and `__mmask64` have been introduced, along with additional maskable intrinsics that operate on vectors of 8 and 16-bit elements. For example,

```
__m512i _mm512_mask_abs_epi8(__m512i src, __mmask64 k, __m512i a);
```

will compute the absolute value of 8-bit elements in *a* corresponding to the set bits of write mask *k*. Elements corresponding to a zero bit in *k* are blended in from *src*.

Doubleword and Quadword Instructions

The doubleword and quadword instructions, indicated by the AVX512DQ CPUID flag, consist of additional instructions along the lines of the Foundation instructions indicated by the AVX512F CPUID flag in that they operate on 512-bit vectors whose elements are 16 32-bit elements or 8 64-bit elements. Some of these instructions provide new functionality such as the conversion of floating point numbers to 64-bit integers. Other instructions promote existing instructions (e.g., vxorps) to use 512-bit registers.

Vector Length Extensions

The vector length extensions indicated by CPUID flag AVX512VL add write-masking, zero-masking, and embedded broadcast features to 128- and 256-bit vector lengths. So for example,

```
__m256 _mm256_maskz_add_ps(__mmask8 k, __m256 a, __m256 b);
```

will add corresponding float32 elements of *a* and *b* where the mask bit from *k* is set, and will produce zero in the elements where the bit from *k* is clear.

Intrinsics for Arithmetic Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element
b	second source vector element
c	third source vector element

_mm_mask_add_pd

```
__m128d _mm_mask_add_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddpd

Add packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_add_pd

```
__m128d _mm_maskz_add_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddpd

Add packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_add_pd

```
__m256d _mm256_mask_add_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddpd

Add packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_add_pd

```
__m256d _mm256_maskz_add_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddpd

Add packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_add_ps

```
__m128 _mm_mask_add_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddps

Add packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_add_ps

```
__m128 _mm_maskz_add_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddps

Add packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_add_ps

```
__m256 _mm256_mask_add_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddps

Add packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_add_ps

```
__m256 _mm256_maskz_add_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vaddps

Add packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_div_pd

```
__m128d _mm_mask_div_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivpd

Divide packed double-precision (64-bit) floating-point elements in *a* by packed elements in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_div_pd

```
__m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivpd

Divide packed double-precision (64-bit) floating-point elements in *a* by packed elements in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_div_pd

```
__m256d _mm256_mask_div_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivpd

Divide packed double-precision (64-bit) floating-point elements in *a* by packed elements in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_div_pd

```
__m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivpd

Divide packed double-precision (64-bit) floating-point elements in *a* by packed elements in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_div_ps

```
__m128 _mm_mask_div_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivps

Divide packed single-precision (32-bit) floating-point elements in *a* by packed elements in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_div_ps

```
__m128 _mm_maskz_div_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivps

Divide packed single-precision (32-bit) floating-point elements in *a* by packed elements in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_div_ps

```
__m256 _mm256_mask_div_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivps

Divide packed single-precision (32-bit) floating-point elements in *a* by packed elements in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_div_ps

```
__m256 _mm256_maskz_div_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vdivps

Divide packed single-precision (32-bit) floating-point elements in *a* by packed elements in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmadd_pd

```
__m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132pd, vfmadd213pd, vfmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmadd_pd

```
__m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132pd, vfmadd213pd, vfmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmadd_pd

```
__m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132pd, vfmadd213pd, vfmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmadd_pd

```
__m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132pd, vfmadd213pd, vfmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmadd_pd

```
__m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132pd, vfmadd213pd, vfmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmadd_pd

```
__m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132pd, vfmadd213pd, vfmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmadd_ps

```
__m128 _mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132ps, vfmadd213ps, vfmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmadd_ps

```
__m128 _mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132ps, vfmadd213ps, vfmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmadd_ps

```
__m128 _mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132ps, vfmadd213ps, vfmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmadd_ps

```
__m256 _mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132ps, vfmadd213ps, vfmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmadd_ps

```
__m256 _mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132ps, vfmadd213ps, vfmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmadd_ps

```
__m256 _mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmadd132ps, vfmadd213ps, vfmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmaddsub_pd

```
__m128d _mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132pd, vfmaddsub213pd, vfmaddsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmaddsub_pd

```
__m128d _mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132pd, vfmaddsub213pd, vfmaddsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmaddsub_pd

```
__m128d _mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132pd, vfmaddsub213pd, vfmaddsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmaddsub_pd

```
__m256d _mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132pd, vfmaddsub213pd, vfmaddsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmaddsub_pd

```
__m256d _mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132pd, vfmaddsub213pd, vfmaddsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmaddsub_pd

```
__m256d _mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132pd, vfmaddsub213pd, vfmaddsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmaddsub_ps

```
__m128 _mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132ps, vfmaddsub213ps, vfmaddsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmaddsub_ps

```
__m128 _mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132ps, vfmaddsub213ps, vfmaddsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmaddsub_ps

```
__m128 _mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132ps, vfmaddsub213ps, vfmaddsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmaddsub_ps

```
__m256 _mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132ps, vfmaddsub213ps, vfmaddsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmaddsub_ps

```
__m256 _mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132ps, vfmaddsub213ps, vfmaddsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmaddsub_ps

```
__m256 _mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmaddsub132ps, vfmaddsub213ps, vfmaddsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmsub_pd

```
__m128d _mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132pd, vfmsub213pd, vfmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmsub_pd

```
__m128d _mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132pd, vfmsub213pd, vfmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmsub_pd

```
__m128d _mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132pd, vfmsub213pd, vfmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmsub_pd

```
__m256d _mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132pd, vfmsub213pd, vfmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmsub_pd

```
__m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132pd, vfmsub213pd, vfmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmsub_pd

```
__m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132pd, vfmsub213pd, vfmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmsub_ps

```
__m128 _mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132ps, vfmsub213ps, vfmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmsub_ps

```
__m128 _mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132ps, vfmsub213ps, vfmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmsub_ps

```
__m128 _mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132ps, vfmsub213ps, vfmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmsub_ps

```
__m256 _mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132ps, vfmsub213ps, vfmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmsub_ps

```
__m256 _mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132ps, vfmsub213ps, vfmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmsub_ps

```
__m256 _mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsub132ps, vfmsub213ps, vfmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmsubadd_pd

```
__m128d _mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132pd, vfmsubadd213pd, vfmsubadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmsubadd_pd

```
__m128d _mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132pd, vfmsubadd213pd, vfmsubadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmsubadd_pd

```
__m128d _mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132pd, vfmsubadd213pd, vfmsubadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmsubadd_pd

```
__m256d _mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132pd, vfmsubadd213pd, vfmsubadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmsubadd_pd

```
__m256d _mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132pd, vfmsubadd213pd, vfmsubadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmsubadd_pd

```
__m256d _mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132pd, vfmsubadd213pd, vfmsubadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmsubadd_ps

```
__m128 _mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132ps, vfmsubadd213ps, vfmsubadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fmsubadd_ps

```
__m128 _mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132ps, vfmsubadd213ps, vfmsubadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fmsubadd_ps

```
__m128 _mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132ps, vfmsubadd213ps, vfmsubadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fmsubadd_ps

```
__m256 _mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132ps, vfmsubadd213ps, vfmsubadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fmsubadd_ps

```
__m256 _mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132ps, vfmsubadd213ps, vfmsubadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fmsubadd_ps

```
__m256 _mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfmsubadd132ps, vfmsubadd213ps, vfmsubadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fnmadd_pd

```
__m128d _mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132pd, vfnmadd213pd, vfnmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fnmadd_pd

```
__m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132pd, vfnmadd213pd, vfnmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fnmadd_pd

```
__m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132pd, vfnmadd213pd, vfnmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fnmadd_pd

```
__m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132pd, vfnmadd213pd, vfnmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fnmadd_pd

```
__m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132pd, vfnmadd213pd, vfnmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fnmadd_pd

```
__m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132pd, vfnmadd213pd, vfnmadd231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fnmadd_ps

```
__m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132ps, vfnmadd213ps, vfnmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fnmadd_ps

```
__m128 _mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132ps, vfnmadd213ps, vfnmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fnmadd_ps

```
__m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132ps, vfnmadd213ps, vfnmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fnmadd_ps

```
__m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132ps, vfnmadd213ps, vfnmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fnmadd_ps

```
__m256 _mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132ps, vfnmadd213ps, vfnmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fnmadd_ps

```
__m256 _mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmadd132ps, vfnmadd213ps, vfnmadd231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, add the negated intermediate result to packed elements in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fnmsub_pd

```
__m128d _mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132pd, vfnmsub213pd, vfnmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fnmsub_pd

```
__m128d _mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132pd, vfnmsub213pd, vfnmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fnmsub_pd

```
__m128d _mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132pd, vfnmsub213pd, vfnmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fnmsub_pd

```
__m256d _mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132pd, vfnmsub213pd, vfnmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fnmsub_pd

```
__m256d _mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132pd, vfnmsub213pd, vfnmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fnmsub_pd

```
__m256d _mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132pd, vfnmsub213pd, vfnmsub231pd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fnmsub_ps

```
__m128 _mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132ps, vfnmsub213ps, vfnmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask3_fnmsub_ps

```
__m128 _mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132ps, vfnmsub213ps, vfnmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fnmsub_ps

```
__m128 _mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132ps, vfnmsub213ps, vfnmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_fnmsub_ps

```
__m256 _mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132ps, vfnmsub213ps, vfnmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask3_fnmsub_ps

```
__m256 _mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132ps, vfnmsub213ps, vfnmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm256_maskz_fnmsub_ps

```
__m256 _mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfnmsub132ps, vfnmsub213ps, vfnmsub231ps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, subtract packed elements in *c* from the negated intermediate result, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_max_pd

```
__m128d _mm_mask_max_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_pd

```
__m128d _mm_maskz_max_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_pd

```
__m256d _mm256_mask_max_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_pd

```
__m256d _mm256_maskz_max_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_max_ps

```
__m128 _mm_mask_max_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_ps

```
__m128 _mm_maskz_max_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_ps

```
__m256 _mm256_mask_max_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_ps

```
__m256 _mm256_maskz_max_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmaxps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_min_pd

```
__m128d _mm_mask_min_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_pd

```
__m128d _mm_maskz_min_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_pd

```
__m256d _mm256_mask_min_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_pd

```
__m256d _mm256_maskz_min_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminpd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_min_ps

```
__m128 _mm_mask_min_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_ps

```
__m128 _mm_maskz_min_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_ps

```
__m256 _mm256_mask_min_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_ps

```
__m256 _mm256_maskz_min_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vminps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mul_pd

```
__m128d _mm_mask_mul_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulpd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). RM.

_mm_maskz_mul_pd

```
__m128d _mm_maskz_mul_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulpd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mul_pd

```
__m256d _mm256_mask_mul_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulpd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mul_pd

```
__m256d _mm256_maskz_mul_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulpd

Multiply packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mul_ps

```
__m128 _mm_mask_mul_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). RM.

_mm_maskz_mul_ps

```
__m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mul_ps

```
__m256 _mm256_mask_mul_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). RM.

_mm256_maskz_mul_ps

```
__m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmulps

Multiply packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_rcp14_pd

```
__m128d _mm_mask_rcp14_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14pd

Compute the approximate reciprocal of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_maskz_rcp14_pd

```
__m128d _mm_maskz_rcp14_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14pd

Compute the approximate reciprocal of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_rcp14_pd

```
__m128d _mm_rcp14_pd(__m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14pd

Compute the approximate reciprocal of packed double-precision (64-bit) floating-point elements in *a*, and return the results. The maximum relative error for this approximation is less than 2^-14.

_mm256_mask_rcp14_pd

```
__m256d _mm256_mask_rcp14_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14pd

Compute the approximate reciprocal of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_maskz_rcp14_pd

```
__m256d _mm256_maskz_rcp14_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14pd

Compute the approximate reciprocal of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_rcp14_pd

```
__m256d _mm256_rcp14_pd(__m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14pd

Compute the approximate reciprocal of packed double-precision (64-bit) floating-point elements in *a*, and return the results. The maximum relative error for this approximation is less than 2^-14.

_mm_mask_rcp14_ps

```
__m128 _mm_mask_rcp14_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14ps

Compute the approximate reciprocal of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_maskz_rcp14_ps

```
__m128 _mm_maskz_rcp14_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14ps

Compute the approximate reciprocal of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_rcp14_ps

```
__m128 _mm_rcp14_ps(__m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14ps

Compute the approximate reciprocal of packed single-precision (32-bit) floating-point elements in *a*, and return the results. The maximum relative error for this approximation is less than 2^-14.

_mm256_mask_rcp14_ps

```
__m256 _mm256_mask_rcp14_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14ps

Compute the approximate reciprocal of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_maskz_rcp14_ps

```
__m256 _mm256_maskz_rcp14_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14ps

Compute the approximate reciprocal of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_rcp14_ps

```
__m256 _mm256_rcp14_ps(__m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrcp14ps

Compute the approximate reciprocal of packed single-precision (32-bit) floating-point elements in *a*, and return the results. The maximum relative error for this approximation is less than 2^-14.

_mm_mask_rsqrt14_pd

```
__m128d _mm_mask_rsqrt14_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14pd

Compute the approximate reciprocal square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_maskz_rsqrt14_pd

```
__m128d _mm_maskz_rsqrt14_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14pd

Compute the approximate reciprocal square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_mask_rsqrt14_pd

```
__m256d _mm256_mask_rsqrt14_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14pd

Compute the approximate reciprocal square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_maskz_rsqrt14_pd

```
__m256d _mm256_maskz_rsqrt14_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14pd

Compute the approximate reciprocal square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_mask_rsqrt14_ps

```
__m128 _mm_mask_rsqrt14_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14ps

Compute the approximate reciprocal square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_maskz_rsqrt14_ps

```
__m128 _mm_maskz_rsqrt14_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14ps

Compute the approximate reciprocal square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_mask_rsqrt14_ps

```
__m256 _mm256_mask_rsqrt14_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14ps

Compute the approximate reciprocal square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm256_maskz_rsqrt14_ps

```
__m256 _mm256_maskz_rsqrt14_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrsqrt14ps

Compute the approximate reciprocal square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). The maximum relative error for this approximation is less than 2^-14.

_mm_mask_sqrt_pd

```
__m128d _mm_mask_sqrt_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtpd

Compute the square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sqrt_pd

```
__m128d _mm_maskz_sqrt_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtpd

Compute the square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sqrt_pd

```
__m256d _mm256_mask_sqrt_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtpd

Compute the square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sqrt_pd

```
__m256d _mm256_maskz_sqrt_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtpd

Compute the square root of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sqrt_ps

```
__m128 _mm_mask_sqrt_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtps

Compute the square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sqrt_ps

```
__m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtps

Compute the square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sqrt_ps

```
__m256 _mm256_mask_sqrt_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtps

Compute the square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sqrt_ps

```
__m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsqrtps

Compute the square root of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sub_pd

```
__m128d _mm_mask_sub_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubpd

Subtract packed double-precision (64-bit) floating-point elements in *b* from packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sub_pd

```
__m128d _mm_maskz_sub_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubpd

Subtract packed double-precision (64-bit) floating-point elements in *b* from packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sub_pd

```
__m256d _mm256_mask_sub_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubpd

Subtract packed double-precision (64-bit) floating-point elements in *b* from packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sub_pd

```
__m256d _mm256_maskz_sub_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubpd

Subtract packed double-precision (64-bit) floating-point elements in *b* from packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sub_ps

```
__m128 _mm_mask_sub_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubps

Subtract packed single-precision (32-bit) floating-point elements in *b* from packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sub_ps

```
__m128 _mm_maskz_sub_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubps

Subtract packed single-precision (32-bit) floating-point elements in *b* from packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sub_ps

```
__m256 _mm256_mask_sub_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubps

Subtract packed single-precision (32-bit) floating-point elements in *b* from packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sub_ps

```
__m256 _mm256_maskz_sub_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vsubps

Subtract packed single-precision (32-bit) floating-point elements in *b* from packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_abs_epi8

```
__m128i _mm_mask_abs_epi8(__m128i src, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsb

Compute the absolute value of packed 8-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_abs_epi8

```
__m128i _mm_maskz_abs_epi8(__mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsb

Compute the absolute value of packed 8-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_abs_epi8

```
__m256i _mm256_mask_abs_epi8(__m256i src, __mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsb

Compute the absolute value of packed 8-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_abs_epi8

```
__m256i _mm256_maskz_abs_epi8(__mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsb

Compute the absolute value of packed 8-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_abs_epi8

```
__m512i _mm512_abs_epi8(__m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpabsb

Compute the absolute value of packed 8-bit integers in *a*, and store the unsigned results in the return value.

_mm512_mask_abs_epi8

```
__m512i _mm512_mask_abs_epi8(__m512i src, __mmask64 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpabsb

Compute the absolute value of packed 8-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_abs_epi8

```
__m512i _mm512_maskz_abs_epi8(__mmask64 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpabsb

Compute the absolute value of packed 8-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_abs_epi32

```
__m128i _mm_mask_abs_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsd

Compute the absolute value of packed 32-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_abs_epi32

```
__m128i _mm_maskz_abs_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsd

Compute the absolute value of packed 32-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_abs_epi32

```
__m256i _mm256_mask_abs_epi32(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsd

Compute the absolute value of packed 32-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_abs_epi32

```
__m256i _mm256_maskz_abs_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsd

Compute the absolute value of packed 32-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_abs_epi64

```
__m128i _mm_abs_epi64(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsq

Compute the absolute value of packed 64-bit integers in *a*, and store the unsigned results in the return value.

_mm_mask_abs_epi64

```
__m128i _mm_mask_abs_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsq

Compute the absolute value of packed 64-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_abs_epi64

```
__m128i _mm_maskz_abs_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsq

Compute the absolute value of packed 64-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_abs_epi64

```
__m256i _mm256_abs_epi64(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsq

Compute the absolute value of packed 64-bit integers in *a*, and store the unsigned results in the return value.

_mm256_mask_abs_epi64

```
__m256i _mm256_mask_abs_epi64(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsq

Compute the absolute value of packed 64-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_abs_epi64

```
__m256i _mm256_maskz_abs_epi64(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpabsq

Compute the absolute value of packed 64-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_abs_epi16

```
__m128i _mm_mask_abs_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsw

Compute the absolute value of packed 16-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_abs_epi16

```
__m128i _mm_maskz_abs_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsw

Compute the absolute value of packed 16-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_abs_epi16

```
__m256i _mm256_mask_abs_epi16(__m256i src, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsw

Compute the absolute value of packed 16-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_abs_epi16

```
__m256i _mm256_maskz_abs_epi16(__mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpabsw

Compute the absolute value of packed 16-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_abs_epi16

```
__m512i _mm512_abs_epi16(__m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpabsw

Compute the absolute value of packed 16-bit integers in *a*, and store the unsigned results in the return value.

_mm512_mask_abs_epi16

```
__m512i _mm512_mask_abs_epi16(__m512i src, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpabsw

Compute the absolute value of packed 16-bit integers in *a*, and store the unsigned results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_abs_epi16

```
__m512i _mm512_maskz_abs_epi16(__mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpabsw

Compute the absolute value of packed 16-bit integers in *a*, and store the unsigned results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_add_epi8

```
__m128i _mm_mask_add_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddb

Add packed 8-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_add_epi8

```
__m128i _mm_maskz_add_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddb

Add packed 8-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_add_epi8

```
__m256i _mm256_mask_add_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddb

Add packed 8-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_add_epi8

```
__m256i _mm256_maskz_add_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddb

Add packed 8-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_add_epi8

```
__m512i _mm512_add_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddb

Add packed 8-bit integers in *a* and *b*, and return the results.

_mm512_mask_add_epi8

```
__m512i _mm512_mask_add_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddb

Add packed 8-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_epi8

```
__m512i _mm512_maskz_add_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddb

Add packed 8-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_add_epi32

```
__m128i _mm_mask_add_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddd

Add packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_add_epi32

```
__m128i _mm_maskz_add_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddd

Add packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_add_epi32

```
__m256i _mm256_mask_add_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddd

Add packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_add_epi32

```
__m256i _mm256_maskz_add_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddd

Add packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_add_epi64

```
__m128i _mm_mask_add_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddq

Add packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_add_epi64

```
__m128i _mm_maskz_add_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddq

Add packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_add_epi64

```
__m256i _mm256_mask_add_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddq

Add packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_add_epi64

```
__m256i _mm256_maskz_add_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpaddq

Add packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_adds_epi8

```
__m128i _mm_mask_adds_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsb

Add packed 8-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_adds_epi8

```
__m128i _mm_maskz_adds_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsb

Add packed 8-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_adds_epi8

```
__m256i _mm256_mask_adds_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsb

Add packed 8-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_adds_epi8

```
__m256i _mm256_maskz_adds_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsb

Add packed 8-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_adds_epi8

```
__m512i _mm512_adds_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddsb

Add packed 8-bit integers in *a* and *b* using saturation, and return the results.

_mm512_mask_adds_epi8

```
__m512i _mm512_mask_adds_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddsb

Add packed 8-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_adds_epi8

```
__m512i _mm512_maskz_adds_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddsb

Add packed 8-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_adds_epi16

```
__m128i _mm_mask_adds_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsw

Add packed 16-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_adds_epi16

```
__m128i _mm_maskz_adds_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsw

Add packed 16-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_adds_epi16

```
__m256i _mm256_mask_adds_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsw

Add packed 16-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_adds_epi16

```
__m256i _mm256_maskz_adds_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddsw

Add packed 16-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_adds_epi16

```
__m512i _mm512_adds_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddsw

Add packed 16-bit integers in *a* and *b* using saturation, and return the results.

_mm512_mask_adds_epi16

```
__m512i _mm512_mask_adds_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddsw

Add packed 16-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_adds_epi16

```
__m512i _mm512_maskz_adds_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddsw

Add packed 16-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_adds_epu8

```
__m128i _mm_mask_adds_epu8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusb

Add packed unsigned 8-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_adds_epu8

```
__m128i _mm_maskz_adds_epu8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusb

Add packed unsigned 8-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_adds_epu8

```
__m256i _mm256_mask_adds_epu8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusb

Add packed unsigned 8-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_adds_epu8

```
__m256i _mm256_maskz_adds_epu8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusb

Add packed unsigned 8-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_adds_epu8

```
__m512i _mm512_adds_epu8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddusb

Add packed unsigned 8-bit integers in *a* and *b* using saturation, and return the results.

_mm512_mask_adds_epu8

```
__m512i _mm512_mask_adds_epu8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddusb

Add packed unsigned 8-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_adds_epu8

```
__m512i _mm512_maskz_adds_epu8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddusb

Add packed unsigned 8-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_adds_epu16

```
__m128i _mm_mask_adds_epu16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusw

Add packed unsigned 16-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_adds_epu16

```
__m128i _mm_maskz_adds_epu16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusw

Add packed unsigned 16-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_adds_epu16

```
__m256i _mm256_mask_adds_epu16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusw

Add packed unsigned 16-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_adds_epu16

```
__m256i _mm256_maskz_adds_epu16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddusw

Add packed unsigned 16-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_adds_epu16

```
__m512i _mm512_adds_epu16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddusw

Add packed unsigned 16-bit integers in *a* and *b* using saturation, and return the results.

_mm512_mask_adds_epu16

```
__m512i _mm512_mask_adds_epu16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddusw

Add packed unsigned 16-bit integers in *a* and *b* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_adds_epu16

```
__m512i _mm512_maskz_adds_epu16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddusw

Add packed unsigned 16-bit integers in *a* and *b* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_add_epi16

```
__m128i _mm_mask_add_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddw

Add packed 16-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_add_epi16

```
__m128i _mm_maskz_add_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddw

Add packed 16-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_add_epi16

```
__m256i _mm256_mask_add_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddw

Add packed 16-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_add_epi16

```
__m256i _mm256_maskz_add_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpaddw

Add packed 16-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_add_epi16

```
__m512i _mm512_add_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddw

Add packed 16-bit integers in *a* and *b*, and return the results.

_mm512_mask_add_epi16

```
__m512i _mm512_mask_add_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddw

Add packed 16-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_epi16

```
__m512i _mm512_maskz_add_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpaddw

Add packed 16-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_avg_epu8

```
__m128i _mm_mask_avg_epu8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgb

Average packed unsigned 8-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_avg_epu8

```
__m128i _mm_maskz_avg_epu8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgb

Average packed unsigned 8-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_avg_epu8

```
__m256i _mm256_mask_avg_epu8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgb

Average packed unsigned 8-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_avg_epu8

```
__m256i _mm256_maskz_avg_epu8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgb

Average packed unsigned 8-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_avg_epu8

```
__m512i _mm512_avg_epu8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpavgb

Average packed unsigned 8-bit integers in *a* and *b*, and return the results.

_mm512_mask_avg_epu8

```
__m512i _mm512_mask_avg_epu8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpavgb

Average packed unsigned 8-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_avg_epu8

```
__m512i _mm512_maskz_avg_epu8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpavgb

Average packed unsigned 8-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_avg_epu16

```
__m128i _mm_mask_avg_epu16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgw

Average packed unsigned 16-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_avg_epu16

```
__m128i _mm_maskz_avg_epu16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgw

Average packed unsigned 16-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_avg_epu16

```
__m256i _mm256_mask_avg_epu16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgw

Average packed unsigned 16-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_avg_epu16

```
__m256i _mm256_maskz_avg_epu16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpavgw

Average packed unsigned 16-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_avg_epu16

```
__m512i _mm512_avg_epu16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpavgw

Average packed unsigned 16-bit integers in *a* and *b*, and return the results.

_mm512_mask_avg_epu16

```
__m512i _mm512_mask_avg_epu16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpavgw

Average packed unsigned 16-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_avg_epu16

```
__m512i _mm512_maskz_avg_epu16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpavgw

Average packed unsigned 16-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_maddubs_epi16

```
__m128i _mm_mask_maddubs_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddubsw

Multiply packed unsigned 8-bit integers in *a* by packed signed 8-bit integers in *b*, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_maddubs_epi16

```
__m128i _mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddubsw

Multiply packed unsigned 8-bit integers in *a* by packed signed 8-bit integers in *b*, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_maddubs_epi16

```
__m256i _mm256_mask_maddubs_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddubsw

Multiply packed unsigned 8-bit integers in *a* by packed signed 8-bit integers in *b*, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_maddubs_epi16

```
__m256i _mm256_maskz_maddubs_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddubsw

Multiply packed unsigned 8-bit integers in *a* by packed signed 8-bit integers in *b*, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maddubs_epi16

```
__m512i _mm512_maddubs_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaddubsw

Vertically multiply each unsigned 8-bit integer from *a* with the corresponding signed 8-bit integer from *b*, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in the return value.

_mm512_mask_maddubs_epi16

```
__m512i _mm512_mask_maddubs_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaddubsw

Multiply packed unsigned 8-bit integers in *a* by packed signed 8-bit integers in *b*, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_maddubs_epi16

```
__m512i _mm512_maskz_maddubs_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaddubsw

Multiply packed unsigned 8-bit integers in *a* by packed signed 8-bit integers in *b*, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_madd_epi16

```
__m128i _mm_mask_madd_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddwd

Multiply packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the saturated results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_madd_epi16

```
__m128i _mm_maskz_madd_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddwd

Multiply packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the saturated results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_madd_epi16

```
__m256i _mm256_mask_madd_epi16(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddwd

Multiply packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the saturated results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_madd_epi16

```
__m256i _mm256_maskz_madd_epi16(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaddwd

Multiply packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the saturated results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_madd_epi16

```
__m512i _mm512_madd_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaddwd

Multiply packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the saturated results in the return value.

_mm512_mask_madd_epi16

```
__m512i _mm512_mask_madd_epi16(__m512i src, __mmask16 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaddwd

Multiply packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the saturated results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_madd_epi16

```
__m512i _mm512_maskz_madd_epi16(__mmask16 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaddwd

Multiply packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the saturated results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_max_epi8

```
__m128i _mm_mask_max_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsb

Compare packed 8-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epi8

```
__m128i _mm_maskz_max_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsb

Compare packed 8-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_epi8

```
__m256i _mm256_mask_max_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsb

Compare packed 8-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epi8

```
__m256i _mm256_maskz_max_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsb

Compare packed 8-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_max_epi8

```
__m512i _mm512_mask_max_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxsb

Compare packed 8-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epi8

```
__m512i _mm512_maskz_max_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxsb

Compare packed 8-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_epi8

```
__m512i _mm512_max_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxsb

Compare packed 8-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm_mask_max_epi32

```
__m128i _mm_mask_max_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsd

Compare packed 32-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epi32

```
__m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsd

Compare packed 32-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_epi32

```
__m256i _mm256_mask_max_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsd

Compare packed 32-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epi32

```
__m256i _mm256_maskz_max_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsd

Compare packed 32-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_max_epi64

```
__m128i _mm_mask_max_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsq

Compare packed 64-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epi64

```
__m128i _mm_maskz_max_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsq

Compare packed 64-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_max_epi64

```
__m128i _mm_max_epi64(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsq

Compare packed 64-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm256_mask_max_epi64

```
__m256i _mm256_mask_max_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsq

Compare packed 64-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epi64

```
__m256i _mm256_maskz_max_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsq

Compare packed 64-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_max_epi64

```
__m256i _mm256_max_epi64(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxsq

Compare packed 64-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm_mask_max_epi16

```
__m128i _mm_mask_max_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsw

Compare packed 16-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epi16

```
__m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsw

Compare packed 16-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_epi16

```
__m256i _mm256_mask_max_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsw

Compare packed 16-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epi16

```
__m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxsw

Compare packed 16-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_max_epi16

```
__m512i _mm512_mask_max_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxsw

Compare packed 16-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epi16

```
__m512i _mm512_maskz_max_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxsw

Compare packed 16-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_epi16

```
__m512i _mm512_max_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxsw

Compare packed 16-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm_mask_max_epu8

```
__m128i _mm_mask_max_epu8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epu8

```
__m128i _mm_maskz_max_epu8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_epu8

```
__m256i _mm256_mask_max_epu8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epu8

```
__m256i _mm256_maskz_max_epu8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_max_epu8

```
__m512i _mm512_mask_max_epu8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epu8

```
__m512i _mm512_maskz_max_epu8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_epu8

```
__m512i _mm512_max_epu8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm_mask_max_epu32

```
__m128i _mm_mask_max_epu32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epu32

```
__m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_epu32

```
__m256i _mm256_mask_max_epu32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epu32

```
__m256i _mm256_maskz_max_epu32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_max_epu64

```
__m128i _mm_mask_max_epu64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epu64

```
__m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_max_epu64

```
__m128i _mm_max_epu64(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm256_mask_max_epu64

```
__m256i _mm256_mask_max_epu64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epu64

```
__m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_max_epu64

```
__m256i _mm256_max_epu64(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmaxuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm_mask_max_epu16

```
__m128i _mm_mask_max_epu16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_max_epu16

```
__m128i _mm_maskz_max_epu16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_max_epu16

```
__m256i _mm256_mask_max_epu16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_max_epu16

```
__m256i _mm256_maskz_max_epu16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmaxuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_max_epu16

```
__m512i _mm512_mask_max_epu16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed maximum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epu16

```
__m512i _mm512_maskz_max_epu16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed maximum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_epu16

```
__m512i _mm512_max_epu16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmaxuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed maximum values in the return value.

_mm_mask_min_epi8

```
__m128i _mm_mask_min_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsb

Compare packed 8-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epi8

```
__m128i _mm_maskz_min_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsb

Compare packed 8-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_epi8

```
__m256i _mm256_mask_min_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsb

Compare packed 8-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epi8

```
__m256i _mm256_maskz_min_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsb

Compare packed 8-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_min_epi8

```
__m512i _mm512_mask_min_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminsb

Compare packed 8-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epi8

```
__m512i _mm512_maskz_min_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminsb

Compare packed 8-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epi8

```
__m512i _mm512_min_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminsb

Compare packed 8-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm_mask_min_epi32

```
__m128i _mm_mask_min_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsd

Compare packed 32-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epi32

```
__m128i _mm_maskz_min_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsd

Compare packed 32-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_epi32

```
__m256i _mm256_mask_min_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsd

Compare packed 32-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epi32

```
__m256i _mm256_maskz_min_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsd

Compare packed 32-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_min_epi64

```
__m128i _mm_mask_min_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsq

Compare packed 64-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epi64

```
__m128i _mm_maskz_min_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsq

Compare packed 64-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_min_epi64

```
__m128i _mm_min_epi64(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsq

Compare packed 64-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm256_mask_min_epi64

```
__m256i _mm256_mask_min_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsq

Compare packed 64-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epi64

```
__m256i _mm256_maskz_min_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsq

Compare packed 64-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_min_epi64

```
__m256i _mm256_min_epi64(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminsq

Compare packed 64-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm_mask_min_epi16

```
__m128i _mm_mask_min_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsw

Compare packed 16-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epi16

```
__m128i _mm_maskz_min_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsw

Compare packed 16-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_epi16

```
__m256i _mm256_mask_min_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsw

Compare packed 16-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epi16

```
__m256i _mm256_maskz_min_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminsw

Compare packed 16-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_min_epi16

```
__m512i _mm512_mask_min_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminsw

Compare packed 16-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epi16

```
__m512i _mm512_maskz_min_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminsw

Compare packed 16-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epi16

```
__m512i _mm512_min_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminsw

Compare packed 16-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm_mask_min_epu8

```
__m128i _mm_mask_min_epu8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epu8

```
__m128i _mm_maskz_min_epu8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_epu8

```
__m256i _mm256_mask_min_epu8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epu8

```
__m256i _mm256_maskz_min_epu8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_min_epu8

```
__m512i _mm512_mask_min_epu8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epu8

```
__m512i _mm512_maskz_min_epu8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epu8

```
__m512i _mm512_min_epu8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminub

Compare packed unsigned 8-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm_mask_min_epu32

```
__m128i _mm_mask_min_epu32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epu32

```
__m128i _mm_maskz_min_epu32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_epu32

```
__m256i _mm256_mask_min_epu32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epu32

```
__m256i _mm256_maskz_min_epu32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminud

Compare packed unsigned 32-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_min_epu64

```
__m128i _mm_mask_min_epu64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epu64

```
__m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_min_epu64

```
__m128i _mm_min_epu64(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm256_mask_min_epu64

```
__m256i _mm256_mask_min_epu64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epu64

```
__m256i _mm256_maskz_min_epu64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_min_epu64

```
__m256i _mm256_min_epu64(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpminuq

Compare packed unsigned 64-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm_mask_min_epu16

```
__m128i _mm_mask_min_epu16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_min_epu16

```
__m128i _mm_maskz_min_epu16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_min_epu16

```
__m256i _mm256_mask_min_epu16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_min_epu16

```
__m256i _mm256_maskz_min_epu16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpminuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_min_epu16

```
__m512i _mm512_mask_min_epu16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed minimum values in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epu16

```
__m512i _mm512_maskz_min_epu16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed minimum values in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epu16

```
__m512i _mm512_min_epu16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpminuw

Compare packed unsigned 16-bit integers in *a* and *b*, and store packed minimum values in the return value.

_mm_mask_mul_epi32

```
__m128i _mm_mask_mul_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuldq

Multiply the low 32-bit integers from each packed 64-bit element in *a* and *b*, and store the signed 64-bit results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mul_epi32

```
__m128i _mm_maskz_mul_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuldq

Multiply the low 32-bit integers from each packed 64-bit element in *a* and *b*, and store the signed 64-bit results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mul_epi32

```
__m256i _mm256_mask_mul_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuldq

Multiply the low 32-bit integers from each packed 64-bit element in *a* and *b*, and store the signed 64-bit results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mul_epi32

```
__m256i _mm256_maskz_mul_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuldq

Multiply the low 32-bit integers from each packed 64-bit element in *a* and *b*, and store the signed 64-bit results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mulhrs_epi16

```
__m128i _mm_mask_mulhrs_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhrsw

Multiply packed 16-bit integers in *a* and *b*, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mulhrs_epi16

```
__m128i _mm_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhrsw

Multiply packed 16-bit integers in *a* and *b*, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mulhrs_epi16

```
__m256i _mm256_mask_mulhrs_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhrsw

Multiply packed 16-bit integers in *a* and *b*, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mulhrs_epi16

```
__m256i _mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhrsw

Multiply packed 16-bit integers in *a* and *b*, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mulhrs_epi16

```
__m512i _mm512_mask_mulhrs_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhrsw

Multiply packed 16-bit integers in *a* and *b*, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mulhrs_epi16

```
__m512i _mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhrsw

Multiply packed 16-bit integers in *a* and *b*, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mulhrs_epi16

```
__m512i _mm512_mulhrs_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhrsw

Multiply packed 16-bit integers in *a* and *b*, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to the return value.

_mm_mask_mulhi_epu16

```
__m128i _mm_mask_mulhi_epu16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhuw

Multiply the packed unsigned 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mulhi_epu16

```
__m128i _mm_maskz_mulhi_epu16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhuw

Multiply the packed unsigned 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mulhi_epu16

```
__m256i _mm256_mask_mulhi_epu16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhuw

Multiply the packed unsigned 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mulhi_epu16

```
__m256i _mm256_maskz_mulhi_epu16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhuw

Multiply the packed unsigned 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mulhi_epu16

```
__m512i _mm512_mask_mulhi_epu16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhuw

Multiply the packed unsigned 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mulhi_epu16

```
__m512i _mm512_maskz_mulhi_epu16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhuw

Multiply the packed unsigned 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mulhi_epu16

```
__m512i _mm512_mulhi_epu16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhuw

Multiply the packed unsigned 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value.

_mm_mask_mulhi_epi16

```
__m128i _mm_mask_mulhi_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mulhi_epi16

```
__m128i _mm_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mulhi_epi16

```
__m256i _mm256_mask_mulhi_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mulhi_epi16

```
__m256i _mm256_maskz_mulhi_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmulhw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mulhi_epi16

```
__m512i _mm512_mask_mulhi_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mulhi_epi16

```
__m512i _mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mulhi_epi16

```
__m512i _mm512_mulhi_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmulhw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in the return value.

_mm_mask_mullo_epi32

```
__m128i _mm_mask_mullo_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmulld

Multiply the packed 32-bit integers in *a* and *b*, producing intermediate 64-bit integers, and store the low 32 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mullo_epi32

```
__m128i _mm_maskz_mullo_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmulld

Multiply the packed 32-bit integers in *a* and *b*, producing intermediate 64-bit integers, and store the low 32 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mullo_epi32

```
__m256i _mm256_mask_mullo_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmulld

Multiply the packed 32-bit integers in *a* and *b*, producing intermediate 64-bit integers, and store the low 32 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mullo_epi32

```
__m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmulld

Multiply the packed 32-bit integers in *a* and *b*, producing intermediate 64-bit integers, and store the low 32 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mullo_epi64

```
__m128i _mm_mask_mullo_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mullo_epi64

```
__m128i _mm_maskz_mullo_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mullo_epi64

```
__m128i _mm_mullo_epi64(__m128i a, __m128i b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value.

_mm256_mask_mullo_epi64

```
__m256i _mm256_mask_mullo_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mullo_epi64

```
__m256i _mm256_maskz_mullo_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mullo_epi64

```
__m256i _mm256_mullo_epi64(__m256i a, __m256i b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value.

_mm512_mask_mullo_epi64

```
__m512i _mm512_mask_mullo_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512DQ

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mullo_epi64

```
__m512i _mm512_maskz_mullo_epi64(__mmask8 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512DQ

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mullo_epi64

```
__m512i _mm512_mullo_epi64(__m512i a, __m512i b)
```

CPUID Flags: AVX512DQ

Instruction(s): vpmullq

Multiply the packed 64-bit integers in *a* and *b*, producing intermediate 128-bit integers, and store the low 64 bits of the intermediate integers in the return value.

_mm_mask_mullo_epi16

```
__m128i _mm_mask_mullo_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmullw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mullo_epi16

```
__m128i _mm_maskz_mullo_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmullw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mullo_epi16

```
__m256i _mm256_mask_mullo_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmullw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mullo_epi16

```
__m256i _mm256_maskz_mullo_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmullw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mullo_epi16

```
__m512i _mm512_mask_mullo_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmullw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mullo_epi16

```
__m512i _mm512_maskz_mullo_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmullw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mullo_epi16

```
__m512i _mm512_mullo_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpmullw

Multiply the packed 16-bit integers in *a* and *b*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in the return value.

_mm_mask_mul_epu32

```
__m128i _mm_mask_mul_epu32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuludq

Multiply the low unsigned 32-bit integers from each packed 64-bit element in *a* and *b*, and store the unsigned 64-bit results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mul_epu32

```
__m128i _mm_maskz_mul_epu32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuludq

Multiply the low unsigned 32-bit integers from each packed 64-bit element in *a* and *b*, and store the unsigned 64-bit results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mul_epu32

```
__m256i _mm256_mask_mul_epu32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuludq

Multiply the low unsigned 32-bit integers from each packed 64-bit element in *a* and *b*, and store the unsigned 64-bit results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mul_epu32

```
__m256i _mm256_maskz_mul_epu32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmuludq

Multiply the low unsigned 32-bit integers from each packed 64-bit element in *a* and *b*, and store the unsigned 64-bit results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sub_epi8

```
__m128i _mm_mask_sub_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sub_epi8

```
__m128i _mm_maskz_sub_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sub_epi8

```
__m256i _mm256_mask_sub_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sub_epi8

```
__m256i _mm256_maskz_sub_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_sub_epi8

```
__m512i _mm512_mask_sub_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sub_epi8

```
__m512i _mm512_maskz_sub_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sub_epi8

```
__m512i _mm512_sub_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a*, and return the results.

_mm_mask_sub_epi32

```
__m128i _mm_mask_sub_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubd

Subtract packed 32-bit integers in *b* from packed 32-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sub_epi32

```
__m128i _mm_maskz_sub_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubd

Subtract packed 32-bit integers in *b* from packed 32-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sub_epi32

```
__m256i _mm256_mask_sub_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubd

Subtract packed 32-bit integers in *b* from packed 32-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sub_epi32

```
__m256i _mm256_maskz_sub_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubd

Subtract packed 32-bit integers in *b* from packed 32-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sub_epi64

```
__m128i _mm_mask_sub_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubq

Subtract packed 64-bit integers in *b* from packed 64-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sub_epi64

```
__m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubq

Subtract packed 64-bit integers in *b* from packed 64-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sub_epi64

```
__m256i _mm256_mask_sub_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubq

Subtract packed 64-bit integers in *b* from packed 64-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sub_epi64

```
__m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsubq

Subtract packed 64-bit integers in *b* from packed 64-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_subs_epi8

```
__m128i _mm_mask_subs_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_subs_epi8

```
__m128i _mm_maskz_subs_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_subs_epi8

```
__m256i _mm256_mask_subs_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_subs_epi8

```
__m256i _mm256_maskz_subs_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_subs_epi8

```
__m512i _mm512_mask_subs_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubsb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_subs_epi8

```
__m512i _mm512_maskz_subs_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubsb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_subs_epi8

```
__m512i _mm512_subs_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubsb

Subtract packed 8-bit integers in *b* from packed 8-bit integers in *a* using saturation, and return the results.

_mm_mask_subs_epi16

```
__m128i _mm_mask_subs_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_subs_epi16

```
__m128i _mm_maskz_subs_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_subs_epi16

```
__m256i _mm256_mask_subs_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_subs_epi16

```
__m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubsw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_subs_epi16

```
__m512i _mm512_mask_subs_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubsw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_subs_epi16

```
__m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubsw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_subs_epi16

```
__m512i _mm512_subs_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubsw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a* using saturation, and return the results.

_mm_mask_subs_epu8

```
__m128i _mm_mask_subs_epu8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusb

Subtract packed unsigned 8-bit integers in *b* from packed unsigned 8-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_subs_epu8

```
__m128i _mm_maskz_subs_epu8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusb

Subtract packed unsigned 8-bit integers in *b* from packed unsigned 8-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_subs_epu8

```
__m256i _mm256_mask_subs_epu8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusb

Subtract packed unsigned 8-bit integers in *b* from packed unsigned 8-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_subs_epu8

```
__m256i _mm256_maskz_subs_epu8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusb

Subtract packed unsigned 8-bit integers in *b* from packed unsigned 8-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_subs_epu8

```
__m512i _mm512_mask_subs_epu8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubusb

Subtract packed unsigned 8-bit integers in *b* from packed unsigned 8-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_subs_epu8

```
__m512i _mm512_maskz_subs_epu8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubusb

Subtract packed unsigned 8-bit integers in *b* from packed unsigned 8-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_subs_epu8

```
__m512i _mm512_subs_epu8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubusb

Subtract packed unsigned 8-bit integers in *b* from packed unsigned 8-bit integers in *a* using saturation, and return the results.

_mm_mask_subs_epu16

```
__m128i _mm_mask_subs_epu16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusw

Subtract packed unsigned 16-bit integers in *b* from packed unsigned 16-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_subs_epu16

```
__m128i _mm_maskz_subs_epu16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusw

Subtract packed unsigned 16-bit integers in *b* from packed unsigned 16-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_subs_epu16

```
__m256i _mm256_mask_subs_epu16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusw

Subtract packed unsigned 16-bit integers in *b* from packed unsigned 16-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_subs_epu16

```
__m256i _mm256_maskz_subs_epu16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubusw

Subtract packed unsigned 16-bit integers in *b* from packed unsigned 16-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_subs_epu16

```
__m512i _mm512_mask_subs_epu16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubusw

Subtract packed unsigned 16-bit integers in *b* from packed unsigned 16-bit integers in *a* using saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_subs_epu16

```
__m512i _mm512_maskz_subs_epu16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubusw

Subtract packed unsigned 16-bit integers in *b* from packed unsigned 16-bit integers in *a* using saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_subs_epu16

```
__m512i _mm512_subs_epu16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubusw

Subtract packed unsigned 16-bit integers in *b* from packed unsigned 16-bit integers in *a* using saturation, and return the results.

_mm_mask_sub_epi16

```
__m128i _mm_mask_sub_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sub_epi16

```
__m128i _mm_maskz_sub_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sub_epi16

```
__m256i _mm256_mask_sub_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sub_epi16

```
__m256i _mm256_maskz_sub_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsubw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_sub_epi16

```
__m512i _mm512_mask_sub_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sub_epi16

```
__m512i _mm512_maskz_sub_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sub_epi16

```
__m512i _mm512_sub_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsubw

Subtract packed 16-bit integers in *b* from packed 16-bit integers in *a*, and return the results.

_mm_madd52hi_epu64

```
__m128i _mm_madd52hi_epu64(__m128i a, __m128i b, __m128i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result.

_mm_mask_madd52hi_epu64

```
__m128i _mm_mask_madd52hi_epu64(__m128i a, __mmask8 k, __m128i b, __m128i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_maskz_madd52hi_epu64

```
__m128i _mm_maskz_madd52hi_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_madd52hi_epu64

```
__m256i _mm256_madd52hi_epu64(__m256i a, __m256i b, __m256i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result.

_mm256_mask_madd52hi_epu64

```
__m256i _mm256_mask_madd52hi_epu64(__m256i a, __mmask8 k, __m256i b, __m256i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_maskz_madd52hi_epu64

```
__m256i _mm256_maskz_madd52hi_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_madd52hi_epu64

```
__m512i _mm512_madd52hi_epu64(__m512i a, __m512i b, __m512i c);
```

CPUID Flags: AVX512IFMA52

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result.

_mm512_mask_madd52hi_epu64

```
__m512i _mm512_mask_madd52hi_epu64(__m512i a, __mmask8 k, __m512i b, __m512i c);
```

CPUID Flags: AVX512IFMA52

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_maskz_madd52hi_epu64

```
__m512i _mm512_maskz_madd52hi_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);
```

CPUID Flags: AVX512IFMA52

Instruction(s): vpmadd52huq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the high 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_madd52lo_epu64

```
__m128i _mm_madd52lo_epu64(__m128i a, __m128i b, __m128i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result.

_mm_mask_madd52lo_epu64

```
__m128i _mm_mask_madd52lo_epu64(__m128i a, __mmask8 k, __m128i b, __m128i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_maskz_madd52lo_epu64

```
__m128i _mm_maskz_madd52lo_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_madd52lo_epu64

```
__m256i _mm256_madd52lo_epu64(__m256i a, __m256i b, __m256i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result.

_mm256_mask_madd52lo_epu64

```
__m256i _mm256_mask_madd52lo_epu64(__m256i a, __mmask8 k, __m256i b, __m256i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_maskz_madd52lo_epu64

```
__m256i _mm256_maskz_madd52lo_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);
```

CPUID Flags: AVX512IFMA52, AVX512VL

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_madd52lo_epu64

```
__m512i _mm512_madd52lo_epu64(__m512i a, __m512i b, __m512i c);
```

CPUID Flags: AVX512IFMA52

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result.

_mm512_mask_madd52lo_epu64

```
__m512i _mm512_mask_madd52lo_epu64(__m512i a, __mmask8 k, __m512i b, __m512i c);
```

CPUID Flags: AVX512IFMA52

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_maskz_madd52lo_epu64

```
__m512i _mm512_maskz_madd52lo_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);
```

CPUID Flags: AVX512IFMA52

Instruction(s): vpmadd52luq

Multiply packed unsigned 52-bit integers in each 64-bit element of *b* and *c* to form a 104-bit intermediate result. Add the low 52-bit unsigned integer from the intermediate result with the corresponding unsigned 64-bit integer in *a*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Bit Manipulation Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element

_mm_lzcnt_epi32

```
__m128i _mm_lzcnt_epi32(__m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntd

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and return the results.

_mm_mask_lzcnt_epi32

```
__m128i _mm_mask_lzcnt_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntd

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_lzcnt_epi32

```
__m128i _mm_maskz_lzcnt_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntd

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_lzcnt_epi32

```
__m256i _mm256_lzcnt_epi32(__m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntd

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and return the results.

_mm256_mask_lzcnt_epi32

```
__m256i _mm256_mask_lzcnt_epi32(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntd

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_lzcnt_epi32

```
__m256i _mm256_maskz_lzcnt_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntd

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_lzcnt_epi64

```
__m128i _mm_lzcnt_epi64(__m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntq

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and return the results.

_mm_mask_lzcnt_epi64

```
__m128i _mm_mask_lzcnt_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntq

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_lzcnt_epi64

```
__m128i _mm_maskz_lzcnt_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntq

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_lzcnt_epi64

```
__m256i _mm256_lzcnt_epi64(__m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntq

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and return the results.

_mm256_mask_lzcnt_epi64

```
__m256i _mm256_mask_lzcnt_epi64(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntq

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_lzcnt_epi64

```
__m256i _mm256_maskz_lzcnt_epi64(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vplzcntq

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_multishift_epi64_epi8

```
__m128i _mm_multishift_epi64_epi8(__m128i a, __m128i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value.

_mm_mask_multishift_epi64_epi8

```
__m128i _mm_mask_multishift_epi64_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_multishift_epi64_epi8

```
__m128i _mm_maskz_multishift_epi64_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_multishift_epi64_epi8

```
__m256i _mm256_multishift_epi64_epi8(__m256i a, __m256i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value.

_mm256_mask_multishift_epi64_epi8

```
__m256i _mm256_mask_multishift_epi64_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_multishift_epi64_epi8

```
__m256i _mm256_maskz_multishift_epi64_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_multishift_epi64_epi8

```
__m512i _mm512_multishift_epi64_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value.

_mm512_mask_multishift_epi64_epi8

```
__m512i _mm512_mask_multishift_epi64_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_multishift_epi64_epi8

```
__m512i _mm512_maskz_multishift_epi64_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpmultishiftqb

For each 64-bit element in *b*, select 8 unaligned bytes using a byte-granular shift control within the corresponding 64-bit element of *a*, and store the 8 assembled bytes to the corresponding 64-bit element of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Comparison Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element
b	second source vector element
imm	comparison predicate, which can be any of the following values: • `_MM_CMPINT_EQ` - Equal • `_MM_CMPINT_LT` - Less than • `_MM_CMPINT_LE` - Less than or Equal • `_MM_CMPINT_NE` - Not Equal • `_MM_CMPINT_NLT` - Not Less than • `_MM_CMPINT_GE` - Greater than or Equal • `_MM_CMPINT_NLE` - Not Less than or Equal • `_MM_CMPINT_GT` - Greater than

_mm_conflict_epi32

```
__m128i _mm_conflict_epi32(__m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictd

Test each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant element. Each element's comparison forms a zero extended bit vector in the return value.

_mm_mask_conflict_epi32

```
__m128i _mm_mask_conflict_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictd

Test each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant element using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm_maskz_conflict_epi32

```
__m128i _mm_maskz_conflict_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictd

Test each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant element using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm256_conflict_epi32

```
__m256i _mm256_conflict_epi32(__m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictd

Test each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant element. Each element's comparison forms a zero extended bit vector in the return value.

_mm256_mask_conflict_epi32

```
__m256i _mm256_mask_conflict_epi32(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictd

Test each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant element using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm256_maskz_conflict_epi32

```
__m256i _mm256_maskz_conflict_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictd

Test each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant element using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm_conflict_epi64

```
__m128i _mm_conflict_epi64(__m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictq

Test each 64-bit element of *a* for equality with all other elements in *a* closer to the least significant element. Each element's comparison forms a zero extended bit vector in the return value.

_mm_mask_conflict_epi64

```
__m128i _mm_mask_conflict_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictq

Test each 64-bit element of *a* for equality with all other elements in *a* closer to the least significant element using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm_maskz_conflict_epi64

```
__m128i _mm_maskz_conflict_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictq

Test each 64-bit element of *a* for equality with all other elements in *a* closer to the least significant element using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm256_conflict_epi64

```
__m256i _mm256_conflict_epi64(__m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictq

Test each 64-bit element of *a* for equality with all other elements in *a* closer to the least significant element. Each element's comparison forms a zero extended bit vector in the return value.

_mm256_mask_conflict_epi64

```
__m256i _mm256_mask_conflict_epi64(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictq

Test each 64-bit element of *a* for equality with all other elements in *a* closer to the least significant element using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm256_maskz_conflict_epi64

```
__m256i _mm256_maskz_conflict_epi64(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpconflictq

Test each 64-bit element of *a* for equality with all other elements in *a* closer to the least significant element using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Each element's comparison forms a zero extended bit vector in the return value.

_mm_cmp_pd_mask

```
__mmask8 _mm_cmp_pd_mask(__m128d a, __m128d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmppd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_pd_mask

```
__mmask8 _mm_mask_cmp_pd_mask(__mmask8 k1, __m128d a, __m128d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmppd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_pd_mask

```
__mmask8 _mm256_cmp_pd_mask(__m256d a, __m256d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmppd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_pd_mask

```
__mmask8 _mm256_mask_cmp_pd_mask(__mmask8 k1, __m256d a, __m256d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmppd

Compare packed double-precision (64-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_ps_mask

```
__mmask8 _mm_cmp_ps_mask(__m128 a, __m128 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmpps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_ps_mask

```
__mmask8 _mm_mask_cmp_ps_mask(__mmask8 k1, __m128 a, __m128 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmpps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_ps_mask

```
__mmask8 _mm256_cmp_ps_mask(__m256 a, __m256 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmpps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_ps_mask

```
__mmask8 _mm256_mask_cmp_ps_mask(__mmask8 k1, __m256 a, __m256 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcmpps

Compare packed single-precision (32-bit) floating-point elements in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epi8_mask

```
__mmask16 _mm_cmp_epi8_mask(__m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epi8_mask

```
__mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epi8_mask

```
__mmask16 _mm_cmpge_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epi8_mask

```
__mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epi8_mask

```
__mmask16 _mm_cmple_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epi8_mask

```
__mmask16 _mm_cmplt_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epi8_mask

```
__mmask16 _mm_cmpneq_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epi8_mask

```
__mmask16 _mm_mask_cmp_epi8_mask(__mmask16 k1, __m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epi8_mask

```
__mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epi8_mask

```
__mmask16 _mm_mask_cmpge_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epi8_mask

```
__mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epi8_mask

```
__mmask16 _mm_mask_cmple_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epi8_mask

```
__mmask16 _mm_mask_cmplt_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epi8_mask

```
__mmask16 _mm_mask_cmpneq_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epi8_mask

```
__mmask32 _mm256_cmp_epi8_mask(__m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epi8_mask

```
__mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epi8_mask

```
__mmask32 _mm256_cmpge_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epi8_mask

```
__mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epi8_mask

```
__mmask32 _mm256_cmple_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epi8_mask

```
__mmask32 _mm256_cmplt_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epi8_mask

```
__mmask32 _mm256_cmpneq_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epi8_mask

```
__mmask32 _mm256_mask_cmp_epi8_mask(__mmask32 k1, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epi8_mask

```
__mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epi8_mask

```
__mmask32 _mm256_mask_cmpge_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epi8_mask

```
__mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epi8_mask

```
__mmask32 _mm256_mask_cmple_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epi8_mask

```
__mmask32 _mm256_mask_cmplt_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epi8_mask

```
__mmask32 _mm256_mask_cmpneq_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_epi8_mask

```
__mmask64 _mm512_cmp_epi8_mask(__m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpeq_epi8_mask

```
__mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpge_epi8_mask

```
__mmask64 _mm512_cmpge_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpgt_epi8_mask

```
__mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmple_epi8_mask

```
__mmask64 _mm512_cmple_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmplt_epi8_mask

```
__mmask64 _mm512_cmplt_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpneq_epi8_mask

```
__mmask64 _mm512_cmpneq_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_mask_cmp_epi8_mask

```
__mmask64 _mm512_mask_cmp_epi8_mask(__mmask64 k1, __m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpeq_epi8_mask

```
__mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpge_epi8_mask

```
__mmask64 _mm512_mask_cmpge_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpgt_epi8_mask

```
__mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmple_epi8_mask

```
__mmask64 _mm512_mask_cmple_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmplt_epi8_mask

```
__mmask64 _mm512_mask_cmplt_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpneq_epi8_mask

```
__mmask64 _mm512_mask_cmpneq_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpb

Compare packed 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epi32_mask

```
__mmask8 _mm_cmp_epi32_mask(__m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epi32_mask

```
__mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epi32_mask

```
__mmask8 _mm_cmpge_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epi32_mask

```
__mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epi32_mask

```
__mmask8 _mm_cmple_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epi32_mask

```
__mmask8 _mm_cmplt_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epi32_mask

```
__mmask8 _mm_cmpneq_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epi32_mask

```
__mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k1, __m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epi32_mask

```
__mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epi32_mask

```
__mmask8 _mm_mask_cmpge_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epi32_mask

```
__mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epi32_mask

```
__mmask8 _mm_mask_cmple_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epi32_mask

```
__mmask8 _mm_mask_cmplt_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epi32_mask

```
__mmask8 _mm_mask_cmpneq_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epi32_mask

```
__mmask8 _mm256_cmp_epi32_mask(__m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epi32_mask

```
__mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epi32_mask

```
__mmask8 _mm256_cmpge_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epi32_mask

```
__mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epi32_mask

```
__mmask8 _mm256_cmple_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epi32_mask

```
__mmask8 _mm256_cmplt_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epi32_mask

```
__mmask8 _mm256_cmpneq_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epi32_mask

```
__mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k1, __m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epi32_mask

```
__mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epi32_mask

```
__mmask8 _mm256_mask_cmpge_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epi32_mask

```
__mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epi32_mask

```
__mmask8 _mm256_mask_cmple_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epi32_mask

```
__mmask8 _mm256_mask_cmplt_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epi32_mask

```
__mmask8 _mm256_mask_cmpneq_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpd

Compare packed 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epi64_mask

```
__mmask8 _mm_cmp_epi64_mask(__m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epi64_mask

```
__mmask8 _mm_cmpeq_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epi64_mask

```
__mmask8 _mm_cmpge_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epi64_mask

```
__mmask8 _mm_cmpgt_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epi64_mask

```
__mmask8 _mm_cmple_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epi64_mask

```
__mmask8 _mm_cmplt_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epi64_mask

```
__mmask8 _mm_cmpneq_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epi64_mask

```
__mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k1, __m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epi64_mask

```
__mmask8 _mm_mask_cmpeq_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epi64_mask

```
__mmask8 _mm_mask_cmpge_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epi64_mask

```
__mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epi64_mask

```
__mmask8 _mm_mask_cmple_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epi64_mask

```
__mmask8 _mm_mask_cmplt_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epi64_mask

```
__mmask8 _mm_mask_cmpneq_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epi64_mask

```
__mmask8 _mm256_cmp_epi64_mask(__m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epi64_mask

```
__mmask8 _mm256_cmpeq_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epi64_mask

```
__mmask8 _mm256_cmpge_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epi64_mask

```
__mmask8 _mm256_cmpgt_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epi64_mask

```
__mmask8 _mm256_cmple_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epi64_mask

```
__mmask8 _mm256_cmplt_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epi64_mask

```
__mmask8 _mm256_cmpneq_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epi64_mask

```
__mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k1, __m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epi64_mask

```
__mmask8 _mm256_mask_cmpeq_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epi64_mask

```
__mmask8 _mm256_mask_cmpge_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epi64_mask

```
__mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epi64_mask

```
__mmask8 _mm256_mask_cmple_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epi64_mask

```
__mmask8 _mm256_mask_cmplt_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epi64_mask

```
__mmask8 _mm256_mask_cmpneq_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpq

Compare packed 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epu8_mask

```
__mmask16 _mm_cmp_epu8_mask(__m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epu8_mask

```
__mmask16 _mm_cmpeq_epu8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epu8_mask

```
__mmask16 _mm_cmpge_epu8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epu8_mask

```
__mmask16 _mm_cmpgt_epu8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epu8_mask

```
__mmask16 _mm_cmple_epu8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epu8_mask

```
__mmask16 _mm_cmplt_epu8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epu8_mask

```
__mmask16 _mm_cmpneq_epu8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epu8_mask

```
__mmask16 _mm_mask_cmp_epu8_mask(__mmask16 k1, __m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epu8_mask

```
__mmask16 _mm_mask_cmpeq_epu8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epu8_mask

```
__mmask16 _mm_mask_cmpge_epu8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epu8_mask

```
__mmask16 _mm_mask_cmpgt_epu8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epu8_mask

```
__mmask16 _mm_mask_cmple_epu8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epu8_mask

```
__mmask16 _mm_mask_cmplt_epu8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epu8_mask

```
__mmask16 _mm_mask_cmpneq_epu8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epu8_mask

```
__mmask32 _mm256_cmp_epu8_mask(__m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epu8_mask

```
__mmask32 _mm256_cmpeq_epu8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epu8_mask

```
__mmask32 _mm256_cmpge_epu8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epu8_mask

```
__mmask32 _mm256_cmpgt_epu8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epu8_mask

```
__mmask32 _mm256_cmple_epu8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epu8_mask

```
__mmask32 _mm256_cmplt_epu8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epu8_mask

```
__mmask32 _mm256_cmpneq_epu8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epu8_mask

```
__mmask32 _mm256_mask_cmp_epu8_mask(__mmask32 k1, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epu8_mask

```
__mmask32 _mm256_mask_cmpeq_epu8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epu8_mask

```
__mmask32 _mm256_mask_cmpge_epu8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epu8_mask

```
__mmask32 _mm256_mask_cmpgt_epu8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epu8_mask

```
__mmask32 _mm256_mask_cmple_epu8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epu8_mask

```
__mmask32 _mm256_mask_cmplt_epu8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epu8_mask

```
__mmask32 _mm256_mask_cmpneq_epu8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_epu8_mask

```
__mmask64 _mm512_cmp_epu8_mask(__m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpeq_epu8_mask

```
__mmask64 _mm512_cmpeq_epu8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpge_epu8_mask

```
__mmask64 _mm512_cmpge_epu8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpgt_epu8_mask

```
__mmask64 _mm512_cmpgt_epu8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmple_epu8_mask

```
__mmask64 _mm512_cmple_epu8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmplt_epu8_mask

```
__mmask64 _mm512_cmplt_epu8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpneq_epu8_mask

```
__mmask64 _mm512_cmpneq_epu8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_mask_cmp_epu8_mask

```
__mmask64 _mm512_mask_cmp_epu8_mask(__mmask64 k1, __m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpeq_epu8_mask

```
__mmask64 _mm512_mask_cmpeq_epu8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpge_epu8_mask

```
__mmask64 _mm512_mask_cmpge_epu8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpgt_epu8_mask

```
__mmask64 _mm512_mask_cmpgt_epu8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmple_epu8_mask

```
__mmask64 _mm512_mask_cmple_epu8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmplt_epu8_mask

```
__mmask64 _mm512_mask_cmplt_epu8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpneq_epu8_mask

```
__mmask64 _mm512_mask_cmpneq_epu8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpub

Compare packed unsigned 8-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epu32_mask

```
__mmask8 _mm_cmp_epu32_mask(__m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epu32_mask

```
__mmask8 _mm_cmpeq_epu32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epu32_mask

```
__mmask8 _mm_cmpge_epu32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epu32_mask

```
__mmask8 _mm_cmpgt_epu32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epu32_mask

```
__mmask8 _mm_cmple_epu32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epu32_mask

```
__mmask8 _mm_cmplt_epu32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epu32_mask

```
__mmask8 _mm_cmpneq_epu32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epu32_mask

```
__mmask8 _mm_mask_cmp_epu32_mask(__mmask8 k1, __m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epu32_mask

```
__mmask8 _mm_mask_cmpeq_epu32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epu32_mask

```
__mmask8 _mm_mask_cmpge_epu32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epu32_mask

```
__mmask8 _mm_mask_cmpgt_epu32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epu32_mask

```
__mmask8 _mm_mask_cmple_epu32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epu32_mask

```
__mmask8 _mm_mask_cmplt_epu32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epu32_mask

```
__mmask8 _mm_mask_cmpneq_epu32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epu32_mask

```
__mmask8 _mm256_cmp_epu32_mask(__m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epu32_mask

```
__mmask8 _mm256_cmpeq_epu32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epu32_mask

```
__mmask8 _mm256_cmpge_epu32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epu32_mask

```
__mmask8 _mm256_cmpgt_epu32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epu32_mask

```
__mmask8 _mm256_cmple_epu32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epu32_mask

```
__mmask8 _mm256_cmplt_epu32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epu32_mask

```
__mmask8 _mm256_cmpneq_epu32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epu32_mask

```
__mmask8 _mm256_mask_cmp_epu32_mask(__mmask8 k1, __m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epu32_mask

```
__mmask8 _mm256_mask_cmpeq_epu32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epu32_mask

```
__mmask8 _mm256_mask_cmpge_epu32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epu32_mask

```
__mmask8 _mm256_mask_cmpgt_epu32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epu32_mask

```
__mmask8 _mm256_mask_cmple_epu32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epu32_mask

```
__mmask8 _mm256_mask_cmplt_epu32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epu32_mask

```
__mmask8 _mm256_mask_cmpneq_epu32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpud

Compare packed unsigned 32-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epu64_mask

```
__mmask8 _mm_cmp_epu64_mask(__m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epu64_mask

```
__mmask8 _mm_cmpeq_epu64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epu64_mask

```
__mmask8 _mm_cmpge_epu64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epu64_mask

```
__mmask8 _mm_cmpgt_epu64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epu64_mask

```
__mmask8 _mm_cmple_epu64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epu64_mask

```
__mmask8 _mm_cmplt_epu64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epu64_mask

```
__mmask8 _mm_cmpneq_epu64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epu64_mask

```
__mmask8 _mm_mask_cmp_epu64_mask(__mmask8 k1, __m128i a, __m128i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epu64_mask

```
__mmask8 _mm_mask_cmpeq_epu64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epu64_mask

```
__mmask8 _mm_mask_cmpge_epu64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epu64_mask

```
__mmask8 _mm_mask_cmpgt_epu64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epu64_mask

```
__mmask8 _mm_mask_cmple_epu64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epu64_mask

```
__mmask8 _mm_mask_cmplt_epu64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epu64_mask

```
__mmask8 _mm_mask_cmpneq_epu64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epu64_mask

```
__mmask8 _mm256_cmp_epu64_mask(__m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epu64_mask

```
__mmask8 _mm256_cmpeq_epu64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epu64_mask

```
__mmask8 _mm256_cmpge_epu64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epu64_mask

```
__mmask8 _mm256_cmpgt_epu64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epu64_mask

```
__mmask8 _mm256_cmple_epu64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epu64_mask

```
__mmask8 _mm256_cmplt_epu64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epu64_mask

```
__mmask8 _mm256_cmpneq_epu64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epu64_mask

```
__mmask8 _mm256_mask_cmp_epu64_mask(__mmask8 k1, __m256i a, __m256i b, const _MM_CMPINT_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epu64_mask

```
__mmask8 _mm256_mask_cmpeq_epu64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epu64_mask

```
__mmask8 _mm256_mask_cmpge_epu64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epu64_mask

```
__mmask8 _mm256_mask_cmpgt_epu64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epu64_mask

```
__mmask8 _mm256_mask_cmple_epu64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epu64_mask

```
__mmask8 _mm256_mask_cmplt_epu64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epu64_mask

```
__mmask8 _mm256_mask_cmpneq_epu64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcmpuq

Compare packed unsigned 64-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epu16_mask

```
__mmask8 _mm_cmp_epu16_mask(__m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epu16_mask

```
__mmask8 _mm_cmpeq_epu16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epu16_mask

```
__mmask8 _mm_cmpge_epu16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epu16_mask

```
__mmask8 _mm_cmpgt_epu16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epu16_mask

```
__mmask8 _mm_cmple_epu16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epu16_mask

```
__mmask8 _mm_cmplt_epu16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epu16_mask

```
__mmask8 _mm_cmpneq_epu16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epu16_mask

```
__mmask8 _mm_mask_cmp_epu16_mask(__mmask8 k1, __m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epu16_mask

```
__mmask8 _mm_mask_cmpeq_epu16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epu16_mask

```
__mmask8 _mm_mask_cmpge_epu16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epu16_mask

```
__mmask8 _mm_mask_cmpgt_epu16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epu16_mask

```
__mmask8 _mm_mask_cmple_epu16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epu16_mask

```
__mmask8 _mm_mask_cmplt_epu16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epu16_mask

```
__mmask8 _mm_mask_cmpneq_epu16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epu16_mask

```
__mmask16 _mm256_cmp_epu16_mask(__m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epu16_mask

```
__mmask16 _mm256_cmpeq_epu16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epu16_mask

```
__mmask16 _mm256_cmpge_epu16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epu16_mask

```
__mmask16 _mm256_cmpgt_epu16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epu16_mask

```
__mmask16 _mm256_cmple_epu16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epu16_mask

```
__mmask16 _mm256_cmplt_epu16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epu16_mask

```
__mmask16 _mm256_cmpneq_epu16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epu16_mask

```
__mmask16 _mm256_mask_cmp_epu16_mask(__mmask16 k1, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epu16_mask

```
__mmask16 _mm256_mask_cmpeq_epu16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epu16_mask

```
__mmask16 _mm256_mask_cmpge_epu16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epu16_mask

```
__mmask16 _mm256_mask_cmpgt_epu16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epu16_mask

```
__mmask16 _mm256_mask_cmple_epu16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epu16_mask

```
__mmask16 _mm256_mask_cmplt_epu16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epu16_mask

```
__mmask16 _mm256_mask_cmpneq_epu16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_epu16_mask

```
__mmask32 _mm512_cmp_epu16_mask(__m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpeq_epu16_mask

```
__mmask32 _mm512_cmpeq_epu16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpge_epu16_mask

```
__mmask32 _mm512_cmpge_epu16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpgt_epu16_mask

```
__mmask32 _mm512_cmpgt_epu16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmple_epu16_mask

```
__mmask32 _mm512_cmple_epu16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmplt_epu16_mask

```
__mmask32 _mm512_cmplt_epu16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpneq_epu16_mask

```
__mmask32 _mm512_cmpneq_epu16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_mask_cmp_epu16_mask

```
__mmask32 _mm512_mask_cmp_epu16_mask(__mmask32 k1, __m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpeq_epu16_mask

```
__mmask32 _mm512_mask_cmpeq_epu16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpge_epu16_mask

```
__mmask32 _mm512_mask_cmpge_epu16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpgt_epu16_mask

```
__mmask32 _mm512_mask_cmpgt_epu16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmple_epu16_mask

```
__mmask32 _mm512_mask_cmple_epu16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmplt_epu16_mask

```
__mmask32 _mm512_mask_cmplt_epu16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpneq_epu16_mask

```
__mmask32 _mm512_mask_cmpneq_epu16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpuw

Compare packed unsigned 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_epi16_mask

```
__mmask8 _mm_cmp_epi16_mask(__m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpeq_epi16_mask

```
__mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpge_epi16_mask

```
__mmask8 _mm_cmpge_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpgt_epi16_mask

```
__mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmple_epi16_mask

```
__mmask8 _mm_cmple_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_cmplt_epi16_mask

```
__mmask8 _mm_cmplt_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm_cmpneq_epi16_mask

```
__mmask8 _mm_cmpneq_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_cmp_epi16_mask

```
__mmask8 _mm_mask_cmp_epi16_mask(__mmask8 k1, __m128i a, __m128i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpeq_epi16_mask

```
__mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpge_epi16_mask

```
__mmask8 _mm_mask_cmpge_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpgt_epi16_mask

```
__mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmple_epi16_mask

```
__mmask8 _mm_mask_cmple_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmplt_epi16_mask

```
__mmask8 _mm_mask_cmplt_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cmpneq_epi16_mask

```
__mmask8 _mm_mask_cmpneq_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cmp_epi16_mask

```
__mmask16 _mm256_cmp_epi16_mask(__m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpeq_epi16_mask

```
__mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpge_epi16_mask

```
__mmask16 _mm256_cmpge_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpgt_epi16_mask

```
__mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmple_epi16_mask

```
__mmask16 _mm256_cmple_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmplt_epi16_mask

```
__mmask16 _mm256_cmplt_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm256_cmpneq_epi16_mask

```
__mmask16 _mm256_cmpneq_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_cmp_epi16_mask

```
__mmask16 _mm256_mask_cmp_epi16_mask(__mmask16 k1, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpeq_epi16_mask

```
__mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpge_epi16_mask

```
__mmask16 _mm256_mask_cmpge_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpgt_epi16_mask

```
__mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmple_epi16_mask

```
__mmask16 _mm256_mask_cmple_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmplt_epi16_mask

```
__mmask16 _mm256_mask_cmplt_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cmpneq_epi16_mask

```
__mmask16 _mm256_mask_cmpneq_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_epi16_mask

```
__mmask32 _mm512_cmp_epi16_mask(__m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpeq_epi16_mask

```
__mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpge_epi16_mask

```
__mmask32 _mm512_cmpge_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpgt_epi16_mask

```
__mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmple_epi16_mask

```
__mmask32 _mm512_cmple_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmplt_epi16_mask

```
__mmask32 _mm512_cmplt_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value.

_mm512_cmpneq_epi16_mask

```
__mmask32 _mm512_cmpneq_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value.

_mm512_mask_cmp_epi16_mask

```
__mmask32 _mm512_mask_cmp_epi16_mask(__mmask32 k1, __m512i a, __m512i b, const int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* based on the comparison operand specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpeq_epi16_mask

```
__mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for equality, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpge_epi16_mask

```
__mmask32 _mm512_mask_cmpge_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpgt_epi16_mask

```
__mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for greater-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmple_epi16_mask

```
__mmask32 _mm512_mask_cmple_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than-or-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmplt_epi16_mask

```
__mmask32 _mm512_mask_cmplt_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for less-than, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpneq_epi16_mask

```
__mmask32 _mm512_mask_cmpneq_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpcmpw

Compare packed 16-bit integers in *a* and *b* for not-equal, and and put each result in the corresponding bit of the returned mask value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_test_epi8_mask

```
__mmask16 _mm_mask_test_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm_test_epi8_mask

```
__mmask16 _mm_test_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm256_mask_test_epi8_mask

```
__mmask32 _mm256_mask_test_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm256_test_epi8_mask

```
__mmask32 _mm256_test_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm512_mask_test_epi8_mask

```
__mmask64 _mm512_mask_test_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm512_test_epi8_mask

```
__mmask64 _mm512_test_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm_mask_test_epi32_mask

```
__mmask8 _mm_mask_test_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm_test_epi32_mask

```
__mmask8 _mm_test_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm256_mask_test_epi32_mask

```
__mmask8 _mm256_mask_test_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm256_test_epi32_mask

```
__mmask8 _mm256_test_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm_mask_test_epi64_mask

```
__mmask8 _mm_mask_test_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm_test_epi64_mask

```
__mmask8 _mm_test_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm256_mask_test_epi64_mask

```
__mmask8 _mm256_mask_test_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm256_test_epi64_mask

```
__mmask8 _mm256_test_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm_mask_test_epi16_mask

```
__mmask8 _mm_mask_test_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm_test_epi16_mask

```
__mmask8 _mm_test_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm256_mask_test_epi16_mask

```
__mmask16 _mm256_mask_test_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm256_test_epi16_mask

```
__mmask16 _mm256_test_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm512_mask_test_epi16_mask

```
__mmask32 _mm512_mask_test_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is non-zero.

_mm512_test_epi16_mask

```
__mmask32 _mm512_test_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value if the intermediate value is non-zero.

_mm_mask_testn_epi8_mask

```
__mmask16 _mm_mask_testn_epi8_mask(__mmask16 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm_testn_epi8_mask

```
__mmask16 _mm_testn_epi8_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm256_mask_testn_epi8_mask

```
__mmask32 _mm256_mask_testn_epi8_mask(__mmask32 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm256_testn_epi8_mask

```
__mmask32 _mm256_testn_epi8_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm512_mask_testn_epi8_mask

```
__mmask64 _mm512_mask_testn_epi8_mask(__mmask64 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestnmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm512_testn_epi8_mask

```
__mmask64 _mm512_testn_epi8_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestnmb

Compute the bitwise AND of packed 8-bit integers in *a* and *b*, producing intermediate 8-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm_mask_testn_epi32_mask

```
__mmask8 _mm_mask_testn_epi32_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm_testn_epi32_mask

```
__mmask8 _mm_testn_epi32_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm256_mask_testn_epi32_mask

```
__mmask8 _mm256_mask_testn_epi32_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm256_testn_epi32_mask

```
__mmask8 _mm256_testn_epi32_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm_mask_testn_epi64_mask

```
__mmask8 _mm_mask_testn_epi64_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm_testn_epi64_mask

```
__mmask8 _mm_testn_epi64_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm256_mask_testn_epi64_mask

```
__mmask8 _mm256_mask_testn_epi64_mask(__mmask8 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm256_testn_epi64_mask

```
__mmask8 _mm256_testn_epi64_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vptestnmq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm_mask_testn_epi16_mask

```
__mmask8 _mm_mask_testn_epi16_mask(__mmask8 k1, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm_testn_epi16_mask

```
__mmask8 _mm_testn_epi16_mask(__m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm256_mask_testn_epi16_mask

```
__mmask16 _mm256_mask_testn_epi16_mask(__mmask16 k1, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm256_testn_epi16_mask

```
__mmask16 _mm256_testn_epi16_mask(__m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vptestnmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

_mm512_mask_testn_epi16_mask

```
__mmask32 _mm512_mask_testn_epi16_mask(__mmask32 k1, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestnmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value (subject to writemask *k*) if the intermediate value is zero.

_mm512_testn_epi16_mask

```
__mmask32 _mm512_testn_epi16_mask(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vptestnmw

Compute the bitwise AND of packed 16-bit integers in *a* and *b*, producing intermediate 16-bit values, and set the corresponding bit in the returned mask value if the intermediate value is zero.

Intrinsics for Conversion Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element
rounding	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm_mask_cvtpd_ps

```
__m128 _mm_mask_cvtpd_ps(__m128 src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2ps

Convert packed double-precision (64-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtpd_ps

```
__m128 _mm_maskz_cvtpd_ps(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2ps

Convert packed double-precision (64-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtpd_ps

```
__m128 _mm256_mask_cvtpd_ps(__m128 src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2ps

Convert packed double-precision (64-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtpd_ps

```
__m128 _mm256_maskz_cvtpd_ps(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2ps

Convert packed double-precision (64-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtph_ps

```
__m128 _mm_mask_cvtph_ps(__m128 src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtph2ps

Convert packed half-precision (16-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtph_ps

```
__m128 _mm_maskz_cvtph_ps(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtph2ps

Convert packed half-precision (16-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtph_ps

```
__m256 _mm256_mask_cvtph_ps(__m256 src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtph2ps

Convert packed half-precision (16-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtph_ps

```
__m256 _mm256_maskz_cvtph_ps(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtph2ps

Convert packed half-precision (16-bit) floating-point elements in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvt_roundps_ph

```
__m128i _mm_mask_cvt_roundps_ph(__m128i src, __mmask8 k, __m128 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_cvtps_ph

```
__m128i _mm_mask_cvtps_ph(__m128i src, __mmask8 k, __m128 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvt_roundps_ph

```
__m128i _mm_maskz_cvt_roundps_ph(__mmask8 k, __m128 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_cvtps_ph

```
__m128i _mm_maskz_cvtps_ph(__mmask8 k, __m128 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvt_roundps_ph

```
__m128i _mm256_mask_cvt_roundps_ph(__m128i src, __mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_cvtps_ph

```
__m128i _mm256_mask_cvtps_ph(__m128i src, __mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvt_roundps_ph

```
__m128i _mm256_maskz_cvt_roundps_ph(__mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_cvtps_ph

```
__m128i _mm256_maskz_cvtps_ph(__mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2ph

Convert packed single-precision (32-bit) floating-point elements in *a* to packed half-precision (16-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi32_pd

```
__m128d _mm_mask_cvtepi32_pd(__m128d src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2pd

Convert packed 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi32_pd

```
__m128d _mm_maskz_cvtepi32_pd(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2pd

Convert packed 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi32_pd

```
__m256d _mm256_mask_cvtepi32_pd(__m256d src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2pd

Convert packed 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi32_pd

```
__m256d _mm256_maskz_cvtepi32_pd(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2pd

Convert packed 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi32_ps

```
__m128 _mm_mask_cvtepi32_ps(__m128 src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2ps

Convert packed 32-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi32_ps

```
__m128 _mm_maskz_cvtepi32_ps(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2ps

Convert packed 32-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi32_ps

```
__m256 _mm256_mask_cvtepi32_ps(__m256 src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2ps

Convert packed 32-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi32_ps

```
__m256 _mm256_maskz_cvtepi32_ps(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtdq2ps

Convert packed 32-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtpd_epi32

```
__m128i _mm_mask_cvtpd_epi32(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtpd_epi32

```
__m128i _mm_maskz_cvtpd_epi32(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtpd_epi32

```
__m128i _mm256_mask_cvtpd_epi32(__m128i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtpd_epi32

```
__m128i _mm256_maskz_cvtpd_epi32(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtpd_epi64

```
__m128i _mm_cvtpd_epi64(__m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm_mask_cvtpd_epi64

```
__m128i _mm_mask_cvtpd_epi64(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtpd_epi64

```
__m128i _mm_maskz_cvtpd_epi64(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtpd_epi64

```
__m256i _mm256_cvtpd_epi64(__m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm256_mask_cvtpd_epi64

```
__m256i _mm256_mask_cvtpd_epi64(__m256i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtpd_epi64

```
__m256i _mm256_maskz_cvtpd_epi64(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundpd_epi64

```
__m512i _mm512_cvt_roundpd_epi64(__m512d a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm512_cvtpd_epi64

```
__m512i _mm512_cvtpd_epi64(__m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm512_mask_cvt_roundpd_epi64

```
__m512i _mm512_mask_cvt_roundpd_epi64(__m512i src, __mmask8 k, __m512d a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtpd_epi64

```
__m512i _mm512_mask_cvtpd_epi64(__m512i src, __mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundpd_epi64

```
__m512i _mm512_maskz_cvt_roundpd_epi64(__mmask8 k, __m512d a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtpd_epi64

```
__m512i _mm512_maskz_cvtpd_epi64(__mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtpd_epu32

```
__m128i _mm_cvtpd_epu32(__m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results.

_mm_mask_cvtpd_epu32

```
__m128i _mm_mask_cvtpd_epu32(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtpd_epu32

```
__m128i _mm_maskz_cvtpd_epu32(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtpd_epu32

```
__m128i _mm256_cvtpd_epu32(__m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results.

_mm256_mask_cvtpd_epu32

```
__m128i _mm256_mask_cvtpd_epu32(__m128i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtpd_epu32

```
__m128i _mm256_maskz_cvtpd_epu32(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtpd_epu64

```
__m128i _mm_cvtpd_epu64(__m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm_mask_cvtpd_epu64

```
__m128i _mm_mask_cvtpd_epu64(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtpd_epu64

```
__m128i _mm_maskz_cvtpd_epu64(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtpd_epu64

```
__m256i _mm256_cvtpd_epu64(__m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm256_mask_cvtpd_epu64

```
__m256i _mm256_mask_cvtpd_epu64(__m256i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtpd_epu64

```
__m256i _mm256_maskz_cvtpd_epu64(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundpd_epu64

```
__m512i _mm512_cvt_roundpd_epu64(__m512d a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm512_cvtpd_epu64

```
__m512i _mm512_cvtpd_epu64(__m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm512_mask_cvt_roundpd_epu64

```
__m512i _mm512_mask_cvt_roundpd_epu64(__m512i src, __mmask8 k, __m512d a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtpd_epu64

```
__m512i _mm512_mask_cvtpd_epu64(__m512i src, __mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundpd_epu64

```
__m512i _mm512_maskz_cvt_roundpd_epu64(__mmask8 k, __m512d a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtpd_epu64

```
__m512i _mm512_maskz_cvtpd_epu64(__mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtps_epi32

```
__m128i _mm_mask_cvtps_epi32(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtps_epi32

```
__m128i _mm_maskz_cvtps_epi32(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtps_epi32

```
__m256i _mm256_mask_cvtps_epi32(__m256i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtps_epi32

```
__m256i _mm256_maskz_cvtps_epi32(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtps_epi64

```
__m128i _mm_cvtps_epi64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm_mask_cvtps_epi64

```
__m128i _mm_mask_cvtps_epi64(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtps_epi64

```
__m128i _mm_maskz_cvtps_epi64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtps_epi64

```
__m256i _mm256_cvtps_epi64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm256_mask_cvtps_epi64

```
__m256i _mm256_mask_cvtps_epi64(__m256i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtps_epi64

```
__m256i _mm256_maskz_cvtps_epi64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_epi64

```
__m512i _mm512_cvt_roundps_epi64(__m256 a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm512_cvtps_epi64

```
__m512i _mm512_cvtps_epi64(__m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results.

_mm512_mask_cvt_roundps_epi64

```
__m512i _mm512_mask_cvt_roundps_epi64(__m512i src, __mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtps_epi64

```
__m512i _mm512_mask_cvtps_epi64(__m512i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundps_epi64

```
__m512i _mm512_maskz_cvt_roundps_epi64(__mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtps_epi64

```
__m512i _mm512_maskz_cvtps_epi64(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtps_epu32

```
__m128i _mm_cvtps_epu32(__m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results.

_mm_mask_cvtps_epu32

```
__m128i _mm_mask_cvtps_epu32(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtps_epu32

```
__m128i _mm_maskz_cvtps_epu32(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtps_epu32

```
__m256i _mm256_cvtps_epu32(__m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results.

_mm256_mask_cvtps_epu32

```
__m256i _mm256_mask_cvtps_epu32(__m256i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtps_epu32

```
__m256i _mm256_maskz_cvtps_epu32(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtps_epu64

```
__m128i _mm_cvtps_epu64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm_mask_cvtps_epu64

```
__m128i _mm_mask_cvtps_epu64(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtps_epu64

```
__m128i _mm_maskz_cvtps_epu64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtps_epu64

```
__m256i _mm256_cvtps_epu64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm256_mask_cvtps_epu64

```
__m256i _mm256_mask_cvtps_epu64(__m256i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtps_epu64

```
__m256i _mm256_maskz_cvtps_epu64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_epu64

```
__m512i _mm512_cvt_roundps_epu64(__m256 a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm512_cvtps_epu64

```
__m512i _mm512_cvtps_epu64(__m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results.

_mm512_mask_cvt_roundps_epu64

```
__m512i _mm512_mask_cvt_roundps_epu64(__m512i src, __mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtps_epu64

```
__m512i _mm512_mask_cvtps_epu64(__m512i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundps_epu64

```
__m512i _mm512_maskz_cvt_roundps_epu64(__mmask8 k, __m256 a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtps_epu64

```
__m512i _mm512_maskz_cvtps_epu64(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi64_pd

```
__m128d _mm_cvtepi64_pd(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm_mask_cvtepi64_pd

```
__m128d _mm_mask_cvtepi64_pd(__m128d src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi64_pd

```
__m128d _mm_maskz_cvtepi64_pd(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi64_pd

```
__m256d _mm256_cvtepi64_pd(__m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm256_mask_cvtepi64_pd

```
__m256d _mm256_mask_cvtepi64_pd(__m256d src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi64_pd

```
__m256d _mm256_maskz_cvtepi64_pd(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundepi64_pd

```
__m512d _mm512_cvt_roundepi64_pd(__m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm512_cvtepi64_pd

```
__m512d _mm512_cvtepi64_pd(__m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm512_mask_cvt_roundepi64_pd

```
__m512d _mm512_mask_cvt_roundepi64_pd(__m512d src, __mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepi64_pd

```
__m512d _mm512_mask_cvtepi64_pd(__m512d src, __mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundepi64_pd

```
__m512d _mm512_maskz_cvt_roundepi64_pd(__mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtepi64_pd

```
__m512d _mm512_maskz_cvtepi64_pd(__mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2pd

Convert packed 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi64_ps

```
__m128 _mm_cvtepi64_ps(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm_mask_cvtepi64_ps

```
__m128 _mm_mask_cvtepi64_ps(__m128 src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi64_ps

```
__m128 _mm_maskz_cvtepi64_ps(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi64_ps

```
__m128 _mm256_cvtepi64_ps(__m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm256_mask_cvtepi64_ps

```
__m128 _mm256_mask_cvtepi64_ps(__m128 src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi64_ps

```
__m128 _mm256_maskz_cvtepi64_ps(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundepi64_ps

```
__m256 _mm512_cvt_roundepi64_ps(__m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm512_cvtepi64_ps

```
__m256 _mm512_cvtepi64_ps(__m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm512_mask_cvt_roundepi64_ps

```
__m256 _mm512_mask_cvt_roundepi64_ps(__m256 src, __mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepi64_ps

```
__m256 _mm512_mask_cvtepi64_ps(__m256 src, __mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundepi64_ps

```
__m256 _mm512_maskz_cvt_roundepi64_ps(__mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtepi64_ps

```
__m256 _mm512_maskz_cvtepi64_ps(__mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtqq2ps

Convert packed 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvttpd_epi32

```
__m128i _mm_mask_cvttpd_epi32(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttpd_epi32

```
__m128i _mm_maskz_cvttpd_epi32(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvttpd_epi32

```
__m128i _mm256_mask_cvttpd_epi32(__m128i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttpd_epi32

```
__m128i _mm256_maskz_cvttpd_epi32(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2dq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvttpd_epi64

```
__m128i _mm_cvttpd_epi64(__m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results.

_mm_mask_cvttpd_epi64

```
__m128i _mm_mask_cvttpd_epi64(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttpd_epi64

```
__m128i _mm_maskz_cvttpd_epi64(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvttpd_epi64

```
__m256i _mm256_cvttpd_epi64(__m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results.

_mm256_mask_cvttpd_epi64

```
__m256i _mm256_mask_cvttpd_epi64(__m256i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttpd_epi64

```
__m256i _mm256_maskz_cvttpd_epi64(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundpd_epi64

```
__m512i _mm512_cvtt_roundpd_epi64(__m512d a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results. Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_cvttpd_epi64

```
__m512i _mm512_cvttpd_epi64(__m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results.

_mm512_mask_cvtt_roundpd_epi64

```
__m512i _mm512_mask_cvtt_roundpd_epi64(__m512i src, __mmask8 k, __m512d a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvttpd_epi64

```
__m512i _mm512_mask_cvttpd_epi64(__m512i src, __mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundpd_epi64

```
__m512i _mm512_maskz_cvtt_roundpd_epi64(__mmask8 k, __m512d a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_maskz_cvttpd_epi64

```
__m512i _mm512_maskz_cvttpd_epi64(__mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2qq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvttpd_epu32

```
__m128i _mm_cvttpd_epu32(__m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results.

_mm_mask_cvttpd_epu32

```
__m128i _mm_mask_cvttpd_epu32(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttpd_epu32

```
__m128i _mm_maskz_cvttpd_epu32(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvttpd_epu32

```
__m128i _mm256_cvttpd_epu32(__m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results.

_mm256_mask_cvttpd_epu32

```
__m128i _mm256_mask_cvttpd_epu32(__m128i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttpd_epu32

```
__m128i _mm256_maskz_cvttpd_epu32(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttpd2udq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvttpd_epu64

```
__m128i _mm_cvttpd_epu64(__m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results.

_mm_mask_cvttpd_epu64

```
__m128i _mm_mask_cvttpd_epu64(__m128i src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttpd_epu64

```
__m128i _mm_maskz_cvttpd_epu64(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvttpd_epu64

```
__m256i _mm256_cvttpd_epu64(__m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results.

_mm256_mask_cvttpd_epu64

```
__m256i _mm256_mask_cvttpd_epu64(__m256i src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttpd_epu64

```
__m256i _mm256_maskz_cvttpd_epu64(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundpd_epu64

```
__m512i _mm512_cvtt_roundpd_epu64(__m512d a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results. Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_cvttpd_epu64

```
__m512i _mm512_cvttpd_epu64(__m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results.

_mm512_mask_cvtt_roundpd_epu64

```
__m512i _mm512_mask_cvtt_roundpd_epu64(__m512i src, __mmask8 k, __m512d a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvttpd_epu64

```
__m512i _mm512_mask_cvttpd_epu64(__m512i src, __mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundpd_epu64

```
__m512i _mm512_maskz_cvtt_roundpd_epu64(__mmask8 k, __m512d a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_maskz_cvttpd_epu64

```
__m512i _mm512_maskz_cvttpd_epu64(__mmask8 k, __m512d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttpd2uqq

Convert packed double-precision (64-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvttps_epi32

```
__m128i _mm_mask_cvttps_epi32(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttps_epi32

```
__m128i _mm_maskz_cvttps_epi32(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvttps_epi32

```
__m256i _mm256_mask_cvttps_epi32(__m256i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttps_epi32

```
__m256i _mm256_maskz_cvttps_epi32(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2dq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvttps_epi64

```
__m128i _mm_cvttps_epi64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results.

_mm_mask_cvttps_epi64

```
__m128i _mm_mask_cvttps_epi64(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttps_epi64

```
__m128i _mm_maskz_cvttps_epi64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvttps_epi64

```
__m256i _mm256_cvttps_epi64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results.

_mm256_mask_cvttps_epi64

```
__m256i _mm256_mask_cvttps_epi64(__m256i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttps_epi64

```
__m256i _mm256_maskz_cvttps_epi64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundps_epi64

```
__m512i _mm512_cvtt_roundps_epi64(__m256 a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results. Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_cvttps_epi64

```
__m512i _mm512_cvttps_epi64(__m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results.

_mm512_mask_cvtt_roundps_epi64

```
__m512i _mm512_mask_cvtt_roundps_epi64(__m512i src, __mmask8 k, __m256 a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvttps_epi64

```
__m512i _mm512_mask_cvttps_epi64(__m512i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundps_epi64

```
__m512i _mm512_maskz_cvtt_roundps_epi64(__mmask8 k, __m256 a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_maskz_cvttps_epi64

```
__m512i _mm512_maskz_cvttps_epi64(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2qq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvttps_epu32

```
__m128i _mm_cvttps_epu32(__m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results.

_mm_mask_cvttps_epu32

```
__m128i _mm_mask_cvttps_epu32(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2udq

Convert packed double-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttps_epu32

```
__m128i _mm_maskz_cvttps_epu32(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2udq

Convert packed double-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvttps_epu32

```
__m256i _mm256_cvttps_epu32(__m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2udq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results.

_mm256_mask_cvttps_epu32

```
__m256i _mm256_mask_cvttps_epu32(__m256i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2udq

Convert packed double-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttps_epu32

```
__m256i _mm256_maskz_cvttps_epu32(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvttps2udq

Convert packed double-precision (32-bit) floating-point elements in *a* to packed unsigned 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvttps_epu64

```
__m128i _mm_cvttps_epu64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results.

_mm_mask_cvttps_epu64

```
__m128i _mm_mask_cvttps_epu64(__m128i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvttps_epu64

```
__m128i _mm_maskz_cvttps_epu64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvttps_epu64

```
__m256i _mm256_cvttps_epu64(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results.

_mm256_mask_cvttps_epu64

```
__m256i _mm256_mask_cvttps_epu64(__m256i src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvttps_epu64

```
__m256i _mm256_maskz_cvttps_epu64(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundps_epu64

```
__m512i _mm512_cvtt_roundps_epu64(__m256 a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results. Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_cvttps_epu64

```
__m512i _mm512_cvttps_epu64(__m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results.

_mm512_mask_cvtt_roundps_epu64

```
__m512i _mm512_mask_cvtt_roundps_epu64(__m512i src, __mmask8 k, __m256 a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvttps_epu64

```
__m512i _mm512_mask_cvttps_epu64(__m512i src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundps_epu64

```
__m512i _mm512_maskz_cvtt_roundps_epu64(__mmask8 k, __m256 a, int sae)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). Pass __MM_FROUND_NO_EXC to *sae* to suppress all exceptions.

_mm512_maskz_cvttps_epu64

```
__m512i _mm512_maskz_cvttps_epu64(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvttps2uqq

Convert packed single-precision (32-bit) floating-point elements in *a* to packed unsigned 64-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepu32_pd

```
__m128d _mm_cvtepu32_pd(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtudq2pd

Convert packed unsigned 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm_mask_cvtepu32_pd

```
__m128d _mm_mask_cvtepu32_pd(__m128d src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtudq2pd

Convert packed unsigned 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu32_pd

```
__m128d _mm_maskz_cvtepu32_pd(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtudq2pd

Convert packed unsigned 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepu32_pd

```
__m256d _mm256_cvtepu32_pd(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtudq2pd

Convert packed unsigned 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm256_mask_cvtepu32_pd

```
__m256d _mm256_mask_cvtepu32_pd(__m256d src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtudq2pd

Convert packed unsigned 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu32_pd

```
__m256d _mm256_maskz_cvtepu32_pd(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcvtudq2pd

Convert packed unsigned 32-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepu64_pd

```
__m128d _mm_cvtepu64_pd(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm_mask_cvtepu64_pd

```
__m128d _mm_mask_cvtepu64_pd(__m128d src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu64_pd

```
__m128d _mm_maskz_cvtepu64_pd(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepu64_pd

```
__m256d _mm256_cvtepu64_pd(__m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm256_mask_cvtepu64_pd

```
__m256d _mm256_mask_cvtepu64_pd(__m256d src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu64_pd

```
__m256d _mm256_maskz_cvtepu64_pd(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundepu64_pd

```
__m512d _mm512_cvt_roundepu64_pd(__m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm512_cvtepu64_pd

```
__m512d _mm512_cvtepu64_pd(__m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results.

_mm512_mask_cvt_roundepu64_pd

```
__m512d _mm512_mask_cvt_roundepu64_pd(__m512d src, __mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepu64_pd

```
__m512d _mm512_mask_cvtepu64_pd(__m512d src, __mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundepu64_pd

```
__m512d _mm512_maskz_cvt_roundepu64_pd(__mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtepu64_pd

```
__m512d _mm512_maskz_cvtepu64_pd(__mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2pd

Convert packed unsigned 64-bit integers in *a* to packed double-precision (64-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepu64_ps

```
__m128 _mm_cvtepu64_ps(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm_mask_cvtepu64_ps

```
__m128 _mm_mask_cvtepu64_ps(__m128 src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu64_ps

```
__m128 _mm_maskz_cvtepu64_ps(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepu64_ps

```
__m128 _mm256_cvtepu64_ps(__m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm256_mask_cvtepu64_ps

```
__m128 _mm256_mask_cvtepu64_ps(__m128 src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu64_ps

```
__m128 _mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundepu64_ps

```
__m256 _mm512_cvt_roundepu64_ps(__m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm512_cvtepu64_ps

```
__m256 _mm512_cvtepu64_ps(__m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results.

_mm512_mask_cvt_roundepu64_ps

```
__m256 _mm512_mask_cvt_roundepu64_ps(__m256 src, __mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepu64_ps

```
__m256 _mm512_mask_cvtepu64_ps(__m256 src, __mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundepu64_ps

```
__m256 _mm512_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtepu64_ps

```
__m256 _mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vcvtuqq2ps

Convert packed unsigned 64-bit integers in *a* to packed single-precision (32-bit) floating-point elements, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi32_epi8

```
__m128i _mm_cvtepi32_epi8(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and return the results.

_mm_mask_cvtepi32_epi8

```
__m128i _mm_mask_cvtepi32_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi32_epi8

```
__m128i _mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi32_epi8

```
__m128i _mm256_cvtepi32_epi8(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and return the results.

_mm256_mask_cvtepi32_epi8

```
__m128i _mm256_mask_cvtepi32_epi8(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi32_epi8

```
__m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi32_epi16

```
__m128i _mm_cvtepi32_epi16(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and return the results.

_mm_mask_cvtepi32_epi16

```
__m128i _mm_mask_cvtepi32_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi32_epi16

```
__m128i _mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi32_epi16

```
__m128i _mm256_cvtepi32_epi16(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and return the results.

_mm256_mask_cvtepi32_epi16

```
__m128i _mm256_mask_cvtepi32_epi16(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi32_epi16

```
__m128i _mm256_maskz_cvtepi32_epi16(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi64_epi8

```
__m128i _mm_cvtepi64_epi8(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and return the results.

_mm_mask_cvtepi64_epi8

```
__m128i _mm_mask_cvtepi64_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi64_epi8

```
__m128i _mm_maskz_cvtepi64_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi64_epi8

```
__m128i _mm256_cvtepi64_epi8(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and return the results.

_mm256_mask_cvtepi64_epi8

```
__m128i _mm256_mask_cvtepi64_epi8(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi64_epi8

```
__m128i _mm256_maskz_cvtepi64_epi8(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi64_epi32

```
__m128i _mm_cvtepi64_epi32(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and return the results.

_mm_mask_cvtepi64_epi32

```
__m128i _mm_mask_cvtepi64_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi64_epi32

```
__m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi64_epi32

```
__m128i _mm256_cvtepi64_epi32(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and return the results.

_mm256_mask_cvtepi64_epi32

```
__m128i _mm256_mask_cvtepi64_epi32(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi64_epi32

```
__m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi64_epi16

```
__m128i _mm_cvtepi64_epi16(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and return the results.

_mm_mask_cvtepi64_epi16

```
__m128i _mm_mask_cvtepi64_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi64_epi16

```
__m128i _mm_maskz_cvtepi64_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi64_epi16

```
__m128i _mm256_cvtepi64_epi16(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and return the results.

_mm256_mask_cvtepi64_epi16

```
__m128i _mm256_mask_cvtepi64_epi16(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi64_epi16

```
__m128i _mm256_maskz_cvtepi64_epi16(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtsepi32_epi8

```
__m128i _mm_cvtsepi32_epi8(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results.

_mm_mask_cvtsepi32_epi8

```
__m128i _mm_mask_cvtsepi32_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtsepi32_epi8

```
__m128i _mm_maskz_cvtsepi32_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtsepi32_epi8

```
__m128i _mm256_cvtsepi32_epi8(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results.

_mm256_mask_cvtsepi32_epi8

```
__m128i _mm256_mask_cvtsepi32_epi8(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtsepi32_epi8

```
__m128i _mm256_maskz_cvtsepi32_epi8(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtsepi32_epi16

```
__m128i _mm_cvtsepi32_epi16(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results.

_mm_mask_cvtsepi32_epi16

```
__m128i _mm_mask_cvtsepi32_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtsepi32_epi16

```
__m128i _mm_maskz_cvtsepi32_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtsepi32_epi16

```
__m128i _mm256_cvtsepi32_epi16(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results.

_mm256_mask_cvtsepi32_epi16

```
__m128i _mm256_mask_cvtsepi32_epi16(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtsepi32_epi16

```
__m128i _mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtsepi64_epi8

```
__m128i _mm_cvtsepi64_epi8(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results.

_mm_mask_cvtsepi64_epi8

```
__m128i _mm_mask_cvtsepi64_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtsepi64_epi8

```
__m128i _mm_maskz_cvtsepi64_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtsepi64_epi8

```
__m128i _mm256_cvtsepi64_epi8(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results.

_mm256_mask_cvtsepi64_epi8

```
__m128i _mm256_mask_cvtsepi64_epi8(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtsepi64_epi8

```
__m128i _mm256_maskz_cvtsepi64_epi8(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtsepi64_epi32

```
__m128i _mm_cvtsepi64_epi32(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and return the results.

_mm_mask_cvtsepi64_epi32

```
__m128i _mm_mask_cvtsepi64_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtsepi64_epi32

```
__m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtsepi64_epi32

```
__m128i _mm256_cvtsepi64_epi32(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and return the results.

_mm256_mask_cvtsepi64_epi32

```
__m128i _mm256_mask_cvtsepi64_epi32(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtsepi64_epi32

```
__m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtsepi64_epi16

```
__m128i _mm_cvtsepi64_epi16(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results.

_mm_mask_cvtsepi64_epi16

```
__m128i _mm_mask_cvtsepi64_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtsepi64_epi16

```
__m128i _mm_maskz_cvtsepi64_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtsepi64_epi16

```
__m128i _mm256_cvtsepi64_epi16(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results.

_mm256_mask_cvtsepi64_epi16

```
__m128i _mm256_mask_cvtsepi64_epi16(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtsepi64_epi16

```
__m128i _mm256_maskz_cvtsepi64_epi16(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtsepi16_epi8

```
__m128i _mm_cvtsepi16_epi8(__m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results.

_mm_mask_cvtsepi16_epi8

```
__m128i _mm_mask_cvtsepi16_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtsepi16_epi8

```
__m128i _mm_maskz_cvtsepi16_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtsepi16_epi8

```
__m128i _mm256_cvtsepi16_epi8(__m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results.

_mm256_mask_cvtsepi16_epi8

```
__m128i _mm256_mask_cvtsepi16_epi8(__m128i src, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtsepi16_epi8

```
__m128i _mm256_maskz_cvtsepi16_epi8(__mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtsepi16_epi8

```
__m256i _mm512_cvtsepi16_epi8(__m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results.

_mm512_mask_cvtsepi16_epi8

```
__m256i _mm512_mask_cvtsepi16_epi8(__m256i src, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtsepi16_epi8

```
__m256i _mm512_maskz_cvtsepi16_epi8(__mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi8_epi32

```
__m128i _mm_mask_cvtepi8_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbd

Sign extend packed 8-bit integers in the low 4 bytes of *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi8_epi32

```
__m128i _mm_maskz_cvtepi8_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbd

Sign extend packed 8-bit integers in the low 4 bytes of *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi8_epi32

```
__m256i _mm256_mask_cvtepi8_epi32(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbd

Sign extend packed 8-bit integers in the low 8 bytes of *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi8_epi32

```
__m256i _mm256_maskz_cvtepi8_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbd

Sign extend packed 8-bit integers in the low 8 bytes of *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi8_epi64

```
__m128i _mm_mask_cvtepi8_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbq

Sign extend packed 8-bit integers in the low 2 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi8_epi64

```
__m128i _mm_maskz_cvtepi8_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbq

Sign extend packed 8-bit integers in the low 2 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi8_epi64

```
__m256i _mm256_mask_cvtepi8_epi64(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbq

Sign extend packed 8-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi8_epi64

```
__m256i _mm256_maskz_cvtepi8_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxbq

Sign extend packed 8-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi8_epi16

```
__m128i _mm_mask_cvtepi8_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovsxbw

Sign extend packed 8-bit integers in *a* to packed 16-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi8_epi16

```
__m128i _mm_maskz_cvtepi8_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovsxbw

Sign extend packed 8-bit integers in *a* to packed 16-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi8_epi16

```
__m256i _mm256_mask_cvtepi8_epi16(__m256i src, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovsxbw

Sign extend packed 8-bit integers in *a* to packed 16-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi8_epi16

```
__m256i _mm256_maskz_cvtepi8_epi16(__mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovsxbw

Sign extend packed 8-bit integers in *a* to packed 16-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi8_epi16

```
__m512i _mm512_cvtepi8_epi16(__m256i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovsxbw

Sign extend packed 8-bit integers in *a* to packed 16-bit integers, and return the results.

_mm512_mask_cvtepi8_epi16

```
__m512i _mm512_mask_cvtepi8_epi16(__m512i src, __mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovsxbw

Sign extend packed 8-bit integers in *a* to packed 16-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepi8_epi16

```
__m512i _mm512_maskz_cvtepi8_epi16(__mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovsxbw

Sign extend packed 8-bit integers in *a* to packed 16-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi32_epi64

```
__m128i _mm_mask_cvtepi32_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxdq

Sign extend packed 32-bit integers in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi32_epi64

```
__m128i _mm_maskz_cvtepi32_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxdq

Sign extend packed 32-bit integers in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi32_epi64

```
__m256i _mm256_mask_cvtepi32_epi64(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxdq

Sign extend packed 32-bit integers in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi32_epi64

```
__m256i _mm256_maskz_cvtepi32_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxdq

Sign extend packed 32-bit integers in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi16_epi32

```
__m128i _mm_mask_cvtepi16_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwd

Sign extend packed 16-bit integers in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi16_epi32

```
__m128i _mm_maskz_cvtepi16_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwd

Sign extend packed 16-bit integers in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi16_epi32

```
__m256i _mm256_mask_cvtepi16_epi32(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwd

Sign extend packed 16-bit integers in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi16_epi32

```
__m256i _mm256_maskz_cvtepi16_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwd

Sign extend packed 16-bit integers in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepi16_epi64

```
__m128i _mm_mask_cvtepi16_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwq

Sign extend packed 16-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi16_epi64

```
__m128i _mm_maskz_cvtepi16_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwq

Sign extend packed 16-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepi16_epi64

```
__m256i _mm256_mask_cvtepi16_epi64(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwq

Sign extend packed 16-bit integers in the low 8 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi16_epi64

```
__m256i _mm256_maskz_cvtepi16_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsxwq

Sign extend packed 16-bit integers in the low 8 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtusepi32_epi8

```
__m128i _mm_cvtusepi32_epi8(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results.

_mm_mask_cvtusepi32_epi8

```
__m128i _mm_mask_cvtusepi32_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtusepi32_epi8

```
__m128i _mm_maskz_cvtusepi32_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtusepi32_epi8

```
__m128i _mm256_cvtusepi32_epi8(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results.

_mm256_mask_cvtusepi32_epi8

```
__m128i _mm256_mask_cvtusepi32_epi8(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtusepi32_epi8

```
__m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtusepi32_epi16

```
__m128i _mm_cvtusepi32_epi16(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results.

_mm_mask_cvtusepi32_epi16

```
__m128i _mm_mask_cvtusepi32_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtusepi32_epi16

```
__m128i _mm_maskz_cvtusepi32_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtusepi32_epi16

```
__m128i _mm256_cvtusepi32_epi16(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results.

_mm256_mask_cvtusepi32_epi16

```
__m128i _mm256_mask_cvtusepi32_epi16(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtusepi32_epi16

```
__m128i _mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtusepi64_epi8

```
__m128i _mm_cvtusepi64_epi8(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results.

_mm_mask_cvtusepi64_epi8

```
__m128i _mm_mask_cvtusepi64_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtusepi64_epi8

```
__m128i _mm_maskz_cvtusepi64_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtusepi64_epi8

```
__m128i _mm256_cvtusepi64_epi8(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results.

_mm256_mask_cvtusepi64_epi8

```
__m128i _mm256_mask_cvtusepi64_epi8(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtusepi64_epi8

```
__m128i _mm256_maskz_cvtusepi64_epi8(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtusepi64_epi32

```
__m128i _mm_cvtusepi64_epi32(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and return the results.

_mm_mask_cvtusepi64_epi32

```
__m128i _mm_mask_cvtusepi64_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtusepi64_epi32

```
__m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtusepi64_epi32

```
__m128i _mm256_cvtusepi64_epi32(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and return the results.

_mm256_mask_cvtusepi64_epi32

```
__m128i _mm256_mask_cvtusepi64_epi32(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtusepi64_epi32

```
__m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtusepi64_epi16

```
__m128i _mm_cvtusepi64_epi16(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results.

_mm_mask_cvtusepi64_epi16

```
__m128i _mm_mask_cvtusepi64_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtusepi64_epi16

```
__m128i _mm_maskz_cvtusepi64_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtusepi64_epi16

```
__m128i _mm256_cvtusepi64_epi16(__m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results.

_mm256_mask_cvtusepi64_epi16

```
__m128i _mm256_mask_cvtusepi64_epi16(__m128i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtusepi64_epi16

```
__m128i _mm256_maskz_cvtusepi64_epi16(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtusepi16_epi8

```
__m128i _mm_cvtusepi16_epi8(__m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results.

_mm_mask_cvtusepi16_epi8

```
__m128i _mm_mask_cvtusepi16_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtusepi16_epi8

```
__m128i _mm_maskz_cvtusepi16_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtusepi16_epi8

```
__m128i _mm256_cvtusepi16_epi8(__m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results.

_mm256_mask_cvtusepi16_epi8

```
__m128i _mm256_mask_cvtusepi16_epi8(__m128i src, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtusepi16_epi8

```
__m128i _mm256_maskz_cvtusepi16_epi8(__mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtusepi16_epi8

```
__m256i _mm512_cvtusepi16_epi8(__m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results.

_mm512_mask_cvtusepi16_epi8

```
__m256i _mm512_mask_cvtusepi16_epi8(__m256i src, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtusepi16_epi8

```
__m256i _mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtepi16_epi8

```
__m128i _mm_cvtepi16_epi8(__m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results.

_mm_mask_cvtepi16_epi8

```
__m128i _mm_mask_cvtepi16_epi8(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepi16_epi8

```
__m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_cvtepi16_epi8

```
__m128i _mm256_cvtepi16_epi8(__m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results.

_mm256_mask_cvtepi16_epi8

```
__m128i _mm256_mask_cvtepi16_epi8(__m128i src, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepi16_epi8

```
__m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi16_epi8

```
__m256i _mm512_cvtepi16_epi8(__m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results.

_mm512_mask_cvtepi16_epi8

```
__m256i _mm512_mask_cvtepi16_epi8(__m256i src, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepi16_epi8

```
__m256i _mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepu8_epi32

```
__m128i _mm_mask_cvtepu8_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbd

Zero extend packed unsigned 8-bit integers in the low 4 bytes of *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu8_epi32

```
__m128i _mm_maskz_cvtepu8_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbd

Zero extend packed unsigned 8-bit integers in th elow 4 bytes of *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepu8_epi32

```
__m256i _mm256_mask_cvtepu8_epi32(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbd

Zero extend packed unsigned 8-bit integers in the low 8 bytes of *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu8_epi32

```
__m256i _mm256_maskz_cvtepu8_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbd

Zero extend packed unsigned 8-bit integers in the low 8 bytes of *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepu8_epi64

```
__m128i _mm_mask_cvtepu8_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbq

Zero extend packed unsigned 8-bit integers in the low 2 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu8_epi64

```
__m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbq

Zero extend packed unsigned 8-bit integers in the low 2 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepu8_epi64

```
__m256i _mm256_mask_cvtepu8_epi64(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbq

Zero extend packed unsigned 8-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu8_epi64

```
__m256i _mm256_maskz_cvtepu8_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxbq

Zero extend packed unsigned 8-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepu8_epi16

```
__m128i _mm_mask_cvtepu8_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovzxbw

Zero extend packed unsigned 8-bit integers in *a* to packed 16-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu8_epi16

```
__m128i _mm_maskz_cvtepu8_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovzxbw

Zero extend packed unsigned 8-bit integers in *a* to packed 16-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepu8_epi16

```
__m256i _mm256_mask_cvtepu8_epi16(__m256i src, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovzxbw

Zero extend packed unsigned 8-bit integers in *a* to packed 16-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu8_epi16

```
__m256i _mm256_maskz_cvtepu8_epi16(__mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovzxbw

Zero extend packed unsigned 8-bit integers in *a* to packed 16-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepu8_epi16

```
__m512i _mm512_cvtepu8_epi16(__m256i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovzxbw

Zero extend packed unsigned 8-bit integers in *a* to packed 16-bit integers, and return the results.

_mm512_mask_cvtepu8_epi16

```
__m512i _mm512_mask_cvtepu8_epi16(__m512i src, __mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovzxbw

Zero extend packed unsigned 8-bit integers in *a* to packed 16-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepu8_epi16

```
__m512i _mm512_maskz_cvtepu8_epi16(__mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovzxbw

Zero extend packed unsigned 8-bit integers in *a* to packed 16-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepu32_epi64

```
__m128i _mm_mask_cvtepu32_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxdq

Zero extend packed unsigned 32-bit integers in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu32_epi64

```
__m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxdq

Zero extend packed unsigned 32-bit integers in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepu32_epi64

```
__m256i _mm256_mask_cvtepu32_epi64(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxdq

Zero extend packed unsigned 32-bit integers in *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu32_epi64

```
__m256i _mm256_maskz_cvtepu32_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxdq

Zero extend packed unsigned 32-bit integers in *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepu16_epi32

```
__m128i _mm_mask_cvtepu16_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwd

Zero extend packed unsigned 16-bit integers in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu16_epi32

```
__m128i _mm_maskz_cvtepu16_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwd

Zero extend packed unsigned 16-bit integers in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepu16_epi32

```
__m256i _mm256_mask_cvtepu16_epi32(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwd

Zero extend packed unsigned 16-bit integers in *a* to packed 32-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu16_epi32

```
__m256i _mm256_maskz_cvtepu16_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwd

Zero extend packed unsigned 16-bit integers in *a* to packed 32-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_cvtepu16_epi64

```
__m128i _mm_mask_cvtepu16_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwq

Zero extend packed unsigned 16-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_cvtepu16_epi64

```
__m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwq

Zero extend packed unsigned 16-bit integers in the low 4 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_cvtepu16_epi64

```
__m256i _mm256_mask_cvtepu16_epi64(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwq

Zero extend packed unsigned 16-bit integers in the low 8 bytes of *a* to packed 64-bit integers, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_cvtepu16_epi64

```
__m256i _mm256_maskz_cvtepu16_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovzxwq

Zero extend packed unsigned 16-bit integers in the low 8 bytes of *a* to packed 64-bit integers, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_packs_epi32

```
__m128i _mm_mask_packs_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackssdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_packs_epi32

```
__m128i _mm_maskz_packs_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackssdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_packs_epi32

```
__m256i _mm256_mask_packs_epi32(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackssdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_packs_epi32

```
__m256i _mm256_maskz_packs_epi32(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackssdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_packs_epi32

```
__m512i _mm512_mask_packs_epi32(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackssdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_packs_epi32

```
__m512i _mm512_maskz_packs_epi32(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackssdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_packs_epi32

```
__m512i _mm512_packs_epi32(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackssdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using signed saturation, and return the results.

_mm_mask_packs_epi16

```
__m128i _mm_mask_packs_epi16(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpacksswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_packs_epi16

```
__m128i _mm_maskz_packs_epi16(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpacksswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_packs_epi16

```
__m256i _mm256_mask_packs_epi16(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpacksswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_packs_epi16

```
__m256i _mm256_maskz_packs_epi16(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpacksswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_packs_epi16

```
__m512i _mm512_mask_packs_epi16(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpacksswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using signed saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_packs_epi16

```
__m512i _mm512_maskz_packs_epi16(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpacksswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using signed saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_packs_epi16

```
__m512i _mm512_packs_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpacksswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using signed saturation, and return the results.

_mm_mask_packus_epi32

```
__m128i _mm_mask_packus_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackusdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_packus_epi32

```
__m128i _mm_maskz_packus_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackusdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_packus_epi32

```
__m256i _mm256_mask_packus_epi32(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackusdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_packus_epi32

```
__m256i _mm256_maskz_packus_epi32(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackusdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_packus_epi32

```
__m512i _mm512_mask_packus_epi32(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackusdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_packus_epi32

```
__m512i _mm512_maskz_packus_epi32(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackusdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_packus_epi32

```
__m512i _mm512_packus_epi32(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackusdw

Convert packed 32-bit integers from *a* and *b* to packed 16-bit integers using unsigned saturation, and return the results.

_mm_mask_packus_epi16

```
__m128i _mm_mask_packus_epi16(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackuswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_packus_epi16

```
__m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackuswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_packus_epi16

```
__m256i _mm256_mask_packus_epi16(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackuswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_packus_epi16

```
__m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpackuswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_packus_epi16

```
__m512i _mm512_mask_packus_epi16(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackuswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using unsigned saturation, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_packus_epi16

```
__m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackuswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using unsigned saturation, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_packus_epi16

```
__m512i _mm512_packus_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpackuswb

Convert packed 16-bit integers from *a* and *b* to packed 8-bit integers using unsigned saturation, and return the results.

_mm_mask_cvtepi32_storeu_epi8

```
void _mm_mask_cvtepi32_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtepi32_storeu_epi8

```
void _mm256_mask_cvtepi32_storeu_epi8(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtepi32_storeu_epi16

```
void _mm_mask_cvtepi32_storeu_epi16(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtepi32_storeu_epi16

```
void _mm256_mask_cvtepi32_storeu_epi16(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtepi64_storeu_epi8

```
void _mm_mask_cvtepi64_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtepi64_storeu_epi8

```
void _mm256_mask_cvtepi64_storeu_epi8(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtepi64_storeu_epi32

```
void _mm_mask_cvtepi64_storeu_epi32(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtepi64_storeu_epi32

```
void _mm256_mask_cvtepi64_storeu_epi32(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtepi64_storeu_epi16

```
void _mm_mask_cvtepi64_storeu_epi16(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtepi64_storeu_epi16

```
void _mm256_mask_cvtepi64_storeu_epi16(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtsepi32_storeu_epi8

```
void _mm_mask_cvtsepi32_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtsepi32_storeu_epi8

```
void _mm256_mask_cvtsepi32_storeu_epi8(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdb

Convert packed 32-bit integers in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtsepi32_storeu_epi16

```
void _mm_mask_cvtsepi32_storeu_epi16(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtsepi32_storeu_epi16

```
void _mm256_mask_cvtsepi32_storeu_epi16(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsdw

Convert packed 32-bit integers in *a* to packed 16-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtsepi64_storeu_epi8

```
void _mm_mask_cvtsepi64_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtsepi64_storeu_epi8

```
void _mm256_mask_cvtsepi64_storeu_epi8(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqb

Convert packed 64-bit integers in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtsepi64_storeu_epi32

```
void _mm_mask_cvtsepi64_storeu_epi32(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtsepi64_storeu_epi32

```
void _mm256_mask_cvtsepi64_storeu_epi32(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqd

Convert packed 64-bit integers in *a* to packed 32-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtsepi64_storeu_epi16

```
void _mm_mask_cvtsepi64_storeu_epi16(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtsepi64_storeu_epi16

```
void _mm256_mask_cvtsepi64_storeu_epi16(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovsqw

Convert packed 64-bit integers in *a* to packed 16-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtsepi16_storeu_epi8

```
void _mm_mask_cvtsepi16_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtsepi16_storeu_epi8

```
void _mm256_mask_cvtsepi16_storeu_epi8(void* base_addr, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_mask_cvtsepi16_storeu_epi8

```
void _mm512_mask_cvtsepi16_storeu_epi8(void* base_addr, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovswb

Convert packed 16-bit integers in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtusepi32_storeu_epi8

```
void _mm_mask_cvtusepi32_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtusepi32_storeu_epi8

```
void _mm256_mask_cvtusepi32_storeu_epi8(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdb

Convert packed unsigned 32-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtusepi32_storeu_epi16

```
void _mm_mask_cvtusepi32_storeu_epi16(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtusepi32_storeu_epi16

```
void _mm256_mask_cvtusepi32_storeu_epi16(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusdw

Convert packed unsigned 32-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtusepi64_storeu_epi8

```
void _mm_mask_cvtusepi64_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtusepi64_storeu_epi8

```
void _mm256_mask_cvtusepi64_storeu_epi8(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqb

Convert packed unsigned 64-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtusepi64_storeu_epi32

```
void _mm_mask_cvtusepi64_storeu_epi32(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtusepi64_storeu_epi32

```
void _mm256_mask_cvtusepi64_storeu_epi32(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqd

Convert packed unsigned 64-bit integers in *a* to packed unsigned 32-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtusepi64_storeu_epi16

```
void _mm_mask_cvtusepi64_storeu_epi16(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtusepi64_storeu_epi16

```
void _mm256_mask_cvtusepi64_storeu_epi16(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpmovusqw

Convert packed unsigned 64-bit integers in *a* to packed unsigned 16-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtusepi16_storeu_epi8

```
void _mm_mask_cvtusepi16_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtusepi16_storeu_epi8

```
void _mm256_mask_cvtusepi16_storeu_epi8(void* base_addr, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_mask_cvtusepi16_storeu_epi8

```
void _mm512_mask_cvtusepi16_storeu_epi8(void* base_addr, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovuswb

Convert packed unsigned 16-bit integers in *a* to packed unsigned 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_cvtepi16_storeu_epi8

```
void _mm_mask_cvtepi16_storeu_epi8(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_cvtepi16_storeu_epi8

```
void _mm256_mask_cvtepi16_storeu_epi8(void* base_addr, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_mask_cvtepi16_storeu_epi8

```
void _mm512_mask_cvtepi16_storeu_epi8(void* base_addr, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovwb

Convert packed 16-bit integers in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

Intrinsics for Load Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
mem_addr	pointer to base address in memory
base_addr	pointer to base address in memory to begin load or store operation

_mm_mask_expandloadu_pd

```
__m128d _mm_mask_expandloadu_pd(__m128d src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load as many contiguous double-precision (64-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 2 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expandloadu_pd

```
__m128d _mm_maskz_expandloadu_pd(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load as many contiguous double-precision (64-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 2 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expandloadu_pd

```
__m256d _mm256_mask_expandloadu_pd(__m256d src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load as many contiguous double-precision (64-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expandloadu_pd

```
__m256d _mm256_maskz_expandloadu_pd(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load as many contiguous double-precision (64-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_expandloadu_ps

```
__m128 _mm_mask_expandloadu_ps(__m128 src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load as many contiguous single-precision (32-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expandloadu_ps

```
__m128 _mm_maskz_expandloadu_ps(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load as many contiguous single-precision (32-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expandloadu_ps

```
__m256 _mm256_mask_expandloadu_ps(__m256 src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load as many contiguous single-precision (32-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 8 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expandloadu_ps

```
__m256 _mm256_maskz_expandloadu_ps(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load as many contiguous single-precision (32-bit) floating-point elements from unaligned memory at *mem_addr* as there are ones in the low 8 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm_mmask_i32gather_pd

```
__m128d _mm_mmask_i32gather_pd(__m128d src, __mmask8 k, __m128i vindex, void const* base_addr,
const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherdpd

Gather double-precision (64-bit) floating-point elements from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i32gather_pd

```
__m256d _mm256_mmask_i32gather_pd(__m256d src, __mmask8 k, __m128i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherdpd

Gather double-precision (64-bit) floating-point elements from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mmask_i32gather_ps

```
__m128 _mm_mmask_i32gather_ps(__m128 src, __mmask8 k, __m128i vindex, void const* base_addr,
const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherdps

Gather single-precision (32-bit) floating-point elements from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i32gather_ps

```
__m256 _mm256_mmask_i32gather_ps(__m256 src, __mmask8 k, __m256i vindex, void const* base_addr,
const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherdps

Gather single-precision (32-bit) floating-point elements from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mmask_i64gather_pd

```
__m128d _mm_mmask_i64gather_pd(__m128d src, __mmask8 k, __m128i vindex, void const* base_addr,
const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherqpd

Gather double-precision (64-bit) floating-point elements from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i64gather_pd

```
__m256d _mm256_mmask_i64gather_pd(__m256d src, __mmask8 k, __m256i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherqpd

Gather double-precision (64-bit) floating-point elements from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mmask_i64gather_ps

```
__m128 _mm_mmask_i64gather_ps(__m128 src, __mmask8 k, __m128i vindex, void const* base_addr,
const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherqps

Gather single-precision (32-bit) floating-point elements from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i64gather_ps

```
__m128 _mm256_mmask_i64gather_ps(__m128 src, __mmask8 k, __m256i vindex, void const* base_addr,
const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgatherqps

Gather single-precision (32-bit) floating-point elements from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mask_load_pd

```
__m128d _mm_mask_load_pd(__m128d src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Load packed double-precision (64-bit) floating-point elements from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm_maskz_load_pd

```
__m128d _mm_maskz_load_pd(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Load packed double-precision (64-bit) floating-point elements from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_load_pd

```
__m256d _mm256_mask_load_pd(__m256d src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Load packed double-precision (64-bit) floating-point elements from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm256_maskz_load_pd

```
__m256d _mm256_maskz_load_pd(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Load packed double-precision (64-bit) floating-point elements from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_load_ps

```
__m128 _mm_mask_load_ps(__m128 src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Load packed single-precision (32-bit) floating-point elements from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm_maskz_load_ps

```
__m128 _mm_maskz_load_ps(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Load packed single-precision (32-bit) floating-point elements from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_load_ps

```
__m256 _mm256_mask_load_ps(__m256 src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Load packed single-precision (32-bit) floating-point elements from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm256_maskz_load_ps

```
__m256 _mm256_maskz_load_ps(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Load packed single-precision (32-bit) floating-point elements from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_loadu_pd

```
__m128d _mm_mask_loadu_pd(__m128d src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovupd

Load packed double-precision (64-bit) floating-point elements from memoy into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_maskz_loadu_pd

```
__m128d _mm_maskz_loadu_pd(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovupd

Load packed double-precision (64-bit) floating-point elements from memoy into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_loadu_pd

```
__m256d _mm256_mask_loadu_pd(__m256d src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovupd

Load packed double-precision (64-bit) floating-point elements from memoy into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_maskz_loadu_pd

```
__m256d _mm256_maskz_loadu_pd(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovupd

Load packed double-precision (64-bit) floating-point elements from memoy into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_loadu_ps

```
__m128 _mm_mask_loadu_ps(__m128 src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovups

Load packed single-precision (32-bit) floating-point elements from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_maskz_loadu_ps

```
__m128 _mm_maskz_loadu_ps(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovups

Load packed single-precision (32-bit) floating-point elements from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_loadu_ps

```
__m256 _mm256_mask_loadu_ps(__m256 src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovups

Load packed single-precision (32-bit) floating-point elements from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_maskz_loadu_ps

```
__m256 _mm256_maskz_loadu_ps(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovups

Load packed single-precision (32-bit) floating-point elements from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_load_epi32

```
__m128i _mm_mask_load_epi32(__m128i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Load packed 32-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm_maskz_load_epi32

```
__m128i _mm_maskz_load_epi32(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Load packed 32-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_load_epi32

```
__m256i _mm256_mask_load_epi32(__m256i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Load packed 32-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm256_maskz_load_epi32

```
__m256i _mm256_maskz_load_epi32(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Load packed 32-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_load_epi64

```
__m128i _mm_mask_load_epi64(__m128i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Load packed 64-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm_maskz_load_epi64

```
__m128i _mm_maskz_load_epi64(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Load packed 64-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_load_epi64

```
__m256i _mm256_mask_load_epi64(__m256i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Load packed 64-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm256_maskz_load_epi64

```
__m256i _mm256_maskz_load_epi64(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Load packed 64-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_loadu_epi16

```
__m128i _mm_mask_loadu_epi16(__m128i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Load packed 16-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_maskz_loadu_epi16

```
__m128i _mm_maskz_loadu_epi16(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Load packed 16-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_loadu_epi16

```
__m256i _mm256_mask_loadu_epi16(__m256i src, __mmask16 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Load packed 16-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_maskz_loadu_epi16

```
__m256i _mm256_maskz_loadu_epi16(__mmask16 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Load packed 16-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm512_mask_loadu_epi16

```
__m512i _mm512_mask_loadu_epi16(__m512i src, __mmask32 k, void const* mem_addr)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu16

Load packed 16-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm512_maskz_loadu_epi16

```
__m512i _mm512_maskz_loadu_epi16(__mmask32 k, void const* mem_addr)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu16

Load packed 16-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_loadu_epi32

```
__m128i _mm_mask_loadu_epi32(__m128i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu32

Load packed 32-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_maskz_loadu_epi32

```
__m128i _mm_maskz_loadu_epi32(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu32

Load packed 32-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_loadu_epi32

```
__m256i _mm256_mask_loadu_epi32(__m256i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu32

Load packed 32-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_maskz_loadu_epi32

```
__m256i _mm256_maskz_loadu_epi32(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu32

Load packed 32-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_loadu_epi64

```
__m128i _mm_mask_loadu_epi64(__m128i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu64

Load packed 64-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_maskz_loadu_epi64

```
__m128i _mm_maskz_loadu_epi64(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu64

Load packed 64-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_loadu_epi64

```
__m256i _mm256_mask_loadu_epi64(__m256i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu64

Load packed 64-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_maskz_loadu_epi64

```
__m256i _mm256_maskz_loadu_epi64(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu64

Load packed 64-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_loadu_epi8

```
__m128i _mm_mask_loadu_epi8(__m128i src, __mmask16 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Load packed 8-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_maskz_loadu_epi8

```
__m128i _mm_maskz_loadu_epi8(__mmask16 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Load packed 8-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_loadu_epi8

```
__m256i _mm256_mask_loadu_epi8(__m256i src, __mmask32 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Load packed 8-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm256_maskz_loadu_epi8

```
__m256i _mm256_maskz_loadu_epi8(__mmask32 k, void const* mem_addr)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Load packed 8-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm512_mask_loadu_epi8

```
__m512i _mm512_mask_loadu_epi8(__m512i src, __mmask64 k, void const* mem_addr)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu8

Load packed 8-bit integers from memory into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm512_maskz_loadu_epi8

```
__m512i _mm512_maskz_loadu_epi8(__mmask64 k, void const* mem_addr)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu8

Load packed 8-bit integers from memory into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_expandloadu_epi32

```
__m128i _mm_mask_expandloadu_epi32(__m128i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load as many contiguous 32-bit integers from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expandloadu_epi32

```
__m128i _mm_maskz_expandloadu_epi32(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load as many contiguous 32-bit integers from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expandloadu_epi32

```
__m256i _mm256_mask_expandloadu_epi32(__m256i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load as many contiguous 32-bit integers from unaligned memory at *mem_addr* as there are ones in the low 8 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expandloadu_epi32

```
__m256i _mm256_maskz_expandloadu_epi32(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load as many contiguous 32-bit integers from unaligned memory at *mem_addr* as there are ones in the low 8 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_expandloadu_epi64

```
__m128i _mm_mask_expandloadu_epi64(__m128i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load as many contiguous 64-bit integers from unaligned memory at *mem_addr* as there are ones in the low 2 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expandloadu_epi64

```
__m128i _mm_maskz_expandloadu_epi64(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load as many contiguous 64-bit integers from unaligned memory at *mem_addr* as there are ones in the low 2 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expandloadu_epi64

```
__m256i _mm256_mask_expandloadu_epi64(__m256i src, __mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load as many contiguous 64-bit integers from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expandloadu_epi64

```
__m256i _mm256_maskz_expandloadu_epi64(__mmask8 k, void const* mem_addr)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load as many contiguous 64-bit integers from unaligned memory at *mem_addr* as there are ones in the low 4 bits of mask *k*, and place them in the result element positions corresponding to the positions of the ones in the mask (elements are zeroed out when the corresponding mask bit is not set).

_mm_mmask_i32gather_epi32

```
__m128i _mm_mmask_i32gather_epi32(__m128i src, __mmask8 k, __m128i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherdd

Gather 32-bit integers from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i32gather_epi32

```
__m256i _mm256_mmask_i32gather_epi32(__m256i src, __mmask8 k, __m256i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherdd

Gather 32-bit integers from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mmask_i32gather_epi64

```
__m128i _mm_mmask_i32gather_epi64(__m128i src, __mmask8 k, __m128i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherdq

Gather 64-bit integers from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i32gather_epi64

```
__m256i _mm256_mmask_i32gather_epi64(__m256i src, __mmask8 k, __m128i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherdq

Gather 64-bit integers from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mmask_i64gather_epi32

```
__m128i _mm_mmask_i64gather_epi32(__m128i src, __mmask8 k, __m128i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherqd

Gather 32-bit integers from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i64gather_epi32

```
__m128i _mm256_mmask_i64gather_epi32(__m128i src, __mmask8 k, __m256i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherqd

Gather 32-bit integers from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mmask_i64gather_epi64

```
__m128i _mm_mmask_i64gather_epi64(__m128i src, __mmask8 k, __m128i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherqq

Gather 64-bit integers from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mmask_i64gather_epi64

```
__m256i _mm256_mmask_i64gather_epi64(__m256i src, __mmask8 k, __m256i vindex, void const*
base_addr, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpgatherqq

Gather 64-bit integers from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

Intrinsics for Logical Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element
b	second source vector element
c	third source vector element
imm	comparison predicate, which can be any of the following values: • `_MM_CMPINT_EQ` - Equal • `_MM_CMPINT_LT` - Less than • `_MM_CMPINT_LE` - Less than or Equal • `_MM_CMPINT_NE` - Not Equal • `_MM_CMPINT_NLT` - Not Less than • `_MM_CMPINT_GE` - Greater than or Equal • `_MM_CMPINT_NLE` - Not Less than or Equal • `_MM_CMPINT_GT` - Greater than

_mm_mask_andnot_pd

```
__m128d _mm_mask_andnot_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnpd

Compute the bitwise AND NOT of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_andnot_pd

```
__m128d _mm_maskz_andnot_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnpd

Compute the bitwise AND NOT of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_andnot_pd

```
__m256d _mm256_mask_andnot_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnpd

Compute the bitwise AND NOT of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_andnot_pd

```
__m256d _mm256_maskz_andnot_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnpd

Compute the bitwise AND NOT of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_andnot_pd

```
__m512d _mm512_andnot_pd(__m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandnpd

Compute the bitwise AND NOT of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_mask_andnot_pd

```
__m512d _mm512_mask_andnot_pd(__m512d src, __mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandnpd

Compute the bitwise AND NOT of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_andnot_pd

```
__m512d _mm512_maskz_andnot_pd(__mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandnpd

Compute the bitwise AND NOT of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_andnot_ps

```
__m128 _mm_mask_andnot_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnps

Compute the bitwise AND NOT of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_andnot_ps

```
__m128 _mm_maskz_andnot_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnps

Compute the bitwise AND NOT of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_andnot_ps

```
__m256 _mm256_mask_andnot_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnps

Compute the bitwise AND NOT of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_andnot_ps

```
__m256 _mm256_maskz_andnot_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandnps

Compute the bitwise AND NOT of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_andnot_ps

```
__m512 _mm512_andnot_ps(__m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandnps

Compute the bitwise AND NOT of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_mask_andnot_ps

```
__m512 _mm512_mask_andnot_ps(__m512 src, __mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandnps

Compute the bitwise AND NOT of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_andnot_ps

```
__m512 _mm512_maskz_andnot_ps(__mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandnps

Compute the bitwise AND NOT of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_and_pd

```
__m128d _mm_mask_and_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandpd

Compute the bitwise AND of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_and_pd

```
__m128d _mm_maskz_and_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandpd

Compute the bitwise AND of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_and_pd

```
__m256d _mm256_mask_and_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandpd

Compute the bitwise AND of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_and_pd

```
__m256d _mm256_maskz_and_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandpd

Compute the bitwise AND of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_and_pd

```
__m512d _mm512_and_pd(__m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandpd

Compute the bitwise AND of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_mask_and_pd

```
__m512d _mm512_mask_and_pd(__m512d src, __mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandpd

Compute the bitwise AND of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_and_pd

```
__m512d _mm512_maskz_and_pd(__mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandpd

Compute the bitwise AND of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_and_ps

```
__m128 _mm_mask_and_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandps

Compute the bitwise AND of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_and_ps

```
__m128 _mm_maskz_and_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandps

Compute the bitwise AND of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_and_ps

```
__m256 _mm256_mask_and_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandps

Compute the bitwise AND of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_and_ps

```
__m256 _mm256_maskz_and_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vandps

Compute the bitwise AND of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_and_ps

```
__m512 _mm512_and_ps(__m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandps

Compute the bitwise AND of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_mask_and_ps

```
__m512 _mm512_mask_and_ps(__m512 src, __mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandps

Compute the bitwise AND of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_and_ps

```
__m512 _mm512_maskz_and_ps(__mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vandps

Compute the bitwise AND of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_or_pd

```
__m128d _mm_mask_or_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorpd

Compute the bitwise OR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_or_pd

```
__m128d _mm_maskz_or_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorpd

Compute the bitwise OR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_or_pd

```
__m256d _mm256_mask_or_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorpd

Compute the bitwise OR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_or_pd

```
__m256d _mm256_maskz_or_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorpd

Compute the bitwise OR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_or_pd

```
__m512d _mm512_mask_or_pd(__m512d src, __mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vorpd

Compute the bitwise OR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_or_pd

```
__m512d _mm512_maskz_or_pd(__mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vorpd

Compute the bitwise OR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_or_pd

```
__m512d _mm512_or_pd(__m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vorpd

Compute the bitwise OR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm_mask_or_ps

```
__m128 _mm_mask_or_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorps

Compute the bitwise OR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_or_ps

```
__m128 _mm_maskz_or_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorps

Compute the bitwise OR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_or_ps

```
__m256 _mm256_mask_or_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorps

Compute the bitwise OR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_or_ps

```
__m256 _mm256_maskz_or_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vorps

Compute the bitwise OR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_or_ps

```
__m512 _mm512_mask_or_ps(__m512 src, __mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vorps

Compute the bitwise OR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_or_ps

```
__m512 _mm512_maskz_or_ps(__mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vorps

Compute the bitwise OR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_or_ps

```
__m512 _mm512_or_ps(__m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vorps

Compute the bitwise OR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm_mask_xor_pd

```
__m128d _mm_mask_xor_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorpd

Compute the bitwise XOR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_xor_pd

```
__m128d _mm_maskz_xor_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorpd

Compute the bitwise XOR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_xor_pd

```
__m256d _mm256_mask_xor_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorpd

Compute the bitwise XOR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_xor_pd

```
__m256d _mm256_maskz_xor_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorpd

Compute the bitwise XOR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_xor_pd

```
__m512d _mm512_mask_xor_pd(__m512d src, __mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vxorpd

Compute the bitwise XOR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_xor_pd

```
__m512d _mm512_maskz_xor_pd(__mmask8 k, __m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vxorpd

Compute the bitwise XOR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_xor_pd

```
__m512d _mm512_xor_pd(__m512d a, __m512d b)
```

CPUID Flags: AVX512DQ

Instruction(s): vxorpd

Compute the bitwise XOR of packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm_mask_xor_ps

```
__m128 _mm_mask_xor_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorps

Compute the bitwise XOR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_xor_ps

```
__m128 _mm_maskz_xor_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorps

Compute the bitwise XOR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_xor_ps

```
__m256 _mm256_mask_xor_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorps

Compute the bitwise XOR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_xor_ps

```
__m256 _mm256_maskz_xor_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vxorps

Compute the bitwise XOR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_xor_ps

```
__m512 _mm512_mask_xor_ps(__m512 src, __mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vxorps

Compute the bitwise XOR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_xor_ps

```
__m512 _mm512_maskz_xor_ps(__mmask16 k, __m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vxorps

Compute the bitwise XOR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_xor_ps

```
__m512 _mm512_xor_ps(__m512 a, __m512 b)
```

CPUID Flags: AVX512DQ

Instruction(s): vxorps

Compute the bitwise XOR of packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm_mask_and_epi32

```
__m128i _mm_mask_and_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_and_epi32

```
__m128i _mm_maskz_and_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_and_epi32

```
__m256i _mm256_mask_and_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_and_epi32

```
__m256i _mm256_maskz_and_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandd

Compute the bitwise AND of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_andnot_epi32

```
__m128i _mm_mask_andnot_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnd

Compute the bitwise AND NOT of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_andnot_epi32

```
__m128i _mm_maskz_andnot_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnd

Compute the bitwise AND NOT of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_andnot_epi32

```
__m256i _mm256_mask_andnot_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnd

Compute the bitwise AND NOT of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_andnot_epi32

```
__m256i _mm256_maskz_andnot_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnd

Compute the bitwise AND NOT of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_andnot_epi64

```
__m128i _mm_mask_andnot_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnq

Compute the bitwise AND NOT of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_andnot_epi64

```
__m128i _mm_maskz_andnot_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnq

Compute the bitwise AND NOT of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_andnot_epi64

```
__m256i _mm256_mask_andnot_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnq

Compute the bitwise AND NOT of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_andnot_epi64

```
__m256i _mm256_maskz_andnot_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandnq

Compute the bitwise AND NOT of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_and_epi64

```
__m128i _mm_mask_and_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_and_epi64

```
__m128i _mm_maskz_and_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_and_epi64

```
__m256i _mm256_mask_and_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_and_epi64

```
__m256i _mm256_maskz_and_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpandq

Compute the bitwise AND of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_or_epi32

```
__m128i _mm_mask_or_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpord

Compute the bitwise OR of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_or_epi32

```
__m128i _mm_maskz_or_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpord

Compute the bitwise OR of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_or_epi32

```
__m256i _mm256_mask_or_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpord

Compute the bitwise OR of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_or_epi32

```
__m256i _mm256_maskz_or_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpord

Compute the bitwise OR of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_or_epi64

```
__m128i _mm_mask_or_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vporq

Compute the bitwise OR of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_or_epi64

```
__m128i _mm_maskz_or_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vporq

Compute the bitwise OR of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_or_epi64

```
__m256i _mm256_mask_or_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vporq

Compute the bitwise OR of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_or_epi64

```
__m256i _mm256_maskz_or_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vporq

Compute the bitwise OR of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_ternarylogic_epi32

```
__m128i _mm_mask_ternarylogic_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogd

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 32-bit integer, the corresponding bit from *src*, *a*, and *b* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using writemask *k* at 32-bit granularity (32-bit elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_ternarylogic_epi32

```
__m128i _mm_maskz_ternarylogic_epi32(__mmask8 k, __m128i a, __m128i b, __m128i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogd

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 32-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using zeromask *k* at 32-bit granularity (32-bit elements are zeroed out when the corresponding mask bit is not set).

_mm_ternarylogic_epi32

```
__m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, __m128i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogd

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 32-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value.

_mm256_mask_ternarylogic_epi32

```
__m256i _mm256_mask_ternarylogic_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogd

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 32-bit integer, the corresponding bit from *src*, *a*, and *b* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using writemask *k* at 32-bit granularity (32-bit elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_ternarylogic_epi32

```
__m256i _mm256_maskz_ternarylogic_epi32(__mmask8 k, __m256i a, __m256i b, __m256i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogd

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 32-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using zeromask *k* at 32-bit granularity (32-bit elements are zeroed out when the corresponding mask bit is not set).

_mm256_ternarylogic_epi32

```
__m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, __m256i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogd

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 32-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value.

_mm_mask_ternarylogic_epi64

```
__m128i _mm_mask_ternarylogic_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogq

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 64-bit integer, the corresponding bit from *src*, *a*, and *b* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using writemask *k* at 64-bit granularity (64-bit elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_ternarylogic_epi64

```
__m128i _mm_maskz_ternarylogic_epi64(__mmask8 k, __m128i a, __m128i b, __m128i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogq

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 64-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using zeromask *k* at 64-bit granularity (64-bit elements are zeroed out when the corresponding mask bit is not set).

_mm_ternarylogic_epi64

```
__m128i _mm_ternarylogic_epi64(__m128i a, __m128i b, __m128i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogq

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 64-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value.

_mm256_mask_ternarylogic_epi64

```
__m256i _mm256_mask_ternarylogic_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogq

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 64-bit integer, the corresponding bit from *src*, *a*, and *b* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using writemask *k* at 64-bit granularity (64-bit elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_ternarylogic_epi64

```
__m256i _mm256_maskz_ternarylogic_epi64(__mmask8 k, __m256i a, __m256i b, __m256i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogq

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 64-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value using zeromask *k* at 64-bit granularity (64-bit elements are zeroed out when the corresponding mask bit is not set).

_mm256_ternarylogic_epi64

```
__m256i _mm256_ternarylogic_epi64(__m256i a, __m256i b, __m256i c, int imm8)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpternlogq

Bitwise ternary logic that provides the capability to implement any three-operand binary function; the specific binary function is specified by value in *imm8*. For each bit in each packed 64-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding bit in the return value.

_mm_mask_xor_epi32

```
__m128i _mm_mask_xor_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxord

Compute the bitwise XOR of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_xor_epi32

```
__m128i _mm_maskz_xor_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxord

Compute the bitwise XOR of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_xor_epi32

```
__m256i _mm256_mask_xor_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxord

Compute the bitwise XOR of packed 32-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_xor_epi32

```
__m256i _mm256_maskz_xor_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxord

Compute the bitwise XOR of packed 32-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_xor_epi64

```
__m128i _mm_mask_xor_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxorq

Compute the bitwise XOR of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_xor_epi64

```
__m128i _mm_maskz_xor_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxorq

Compute the bitwise XOR of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_xor_epi64

```
__m256i _mm256_mask_xor_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxorq

Compute the bitwise XOR of packed 64-bit integers in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_xor_epi64

```
__m256i _mm256_maskz_xor_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpxorq

Compute the bitwise XOR of packed 64-bit integers in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Miscellaneous Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element
b	second source vector element
c	third source vector element
rounding	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register
interv	Where `_MM_MANTISSA_NORM_ENUM` can be one of the following: • `_MM_MANT_NORM_1_2` - interval [1, 2) • `_MM_MANT_NORM_p5_2` - interval [1.5, 2) • `_MM_MANT_NORM_p5_1` - interval [1.5, 1) • `_MM_MANT_NORM_p75_1p5` - interval [0.75, 1.5)

variable	definition
sc	Where `_MM_MANTISSA_SIGN_ENUM` can be one of the following: • `_MM_MANT_SIGN_src` - sign = sign(SRC) • `_MM_MANT_SIGN_zero` - sign = 0 • `_MM_MANT_SIGN_nan` - DEST = NaN if sign(SRC) = 1

_mm_broadcast_i32x2

```
__m128i _mm_broadcast_i32x2(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of "dst.

_mm_mask_broadcast_i32x2

```
__m128i _mm_mask_broadcast_i32x2(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_broadcast_i32x2

```
__m128i _mm_maskz_broadcast_i32x2(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_broadcast_i32x2

```
__m256i _mm256_broadcast_i32x2(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of "dst.

_mm256_mask_broadcast_i32x2

```
__m256i _mm256_mask_broadcast_i32x2(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcast_i32x2

```
__m256i _mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_i32x2

```
__m512i _mm512_broadcast_i32x2(__m128i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of "dst.

_mm512_mask_broadcast_i32x2

```
__m512i _mm512_mask_broadcast_i32x2(__m512i src, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_i32x2

```
__m512i _mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti32x2

Broadcast the lower 2 packed 32-bit integers from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_broadcast_i32x4

```
__m256i _mm256_broadcast_i32x4(__m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcasti32x4

Broadcast the 4 packed 32-bit integers from *a* to all elements of the return value.

_mm256_mask_broadcast_i32x4

```
__m256i _mm256_mask_broadcast_i32x4(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcasti32x4

Broadcast the 4 packed 32-bit integers from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcast_i32x4

```
__m256i _mm256_maskz_broadcast_i32x4(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcasti32x4

Broadcast the 4 packed 32-bit integers from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_i32x8

```
__m512i _mm512_broadcast_i32x8(__m256i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti32x8

Broadcast the 8 packed 32-bit integers from *a* to all elements of the return value.

_mm512_mask_broadcast_i32x8

```
__m512i _mm512_mask_broadcast_i32x8(__m512i src, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti32x8

Broadcast the 8 packed 32-bit integers from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_i32x8

```
__m512i _mm512_maskz_broadcast_i32x8(__mmask16 k, __m256i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti32x8

Broadcast the 8 packed 32-bit integers from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_broadcast_i64x2

```
__m256i _mm256_broadcast_i64x2(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti64x2

Broadcast the 2 packed 64-bit integers from *a* to all elements of the return value.

_mm256_mask_broadcast_i64x2

```
__m256i _mm256_mask_broadcast_i64x2(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti64x2

Broadcast the 2 packed 64-bit integers from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcast_i64x2

```
__m256i _mm256_maskz_broadcast_i64x2(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcasti64x2

Broadcast the 2 packed 64-bit integers from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_i64x2

```
__m512i _mm512_broadcast_i64x2(__m128i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti64x2

Broadcast the 2 packed 64-bit integers from *a* to all elements of the return value.

_mm512_mask_broadcast_i64x2

```
__m512i _mm512_mask_broadcast_i64x2(__m512i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti64x2

Broadcast the 2 packed 64-bit integers from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_i64x2

```
__m512i _mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcasti64x2

Broadcast the 2 packed 64-bit integers from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_inserti32x4

```
__m256i _mm256_inserti32x4(__m256i a, __m128i b, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vinserti32x4

Copy *a* to the return value, then insert 128 bits (composed of 4 packed 32-bit integers) from *b* into *dst* at the location specified by *imm*.

_mm256_mask_inserti32x4

```
__m256i _mm256_mask_inserti32x4(__m256i src, __mmask8 k, __m256i a, __m128i b, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vinserti32x4

Copy *a* to *tmp*, then insert 128 bits (composed of 4 packed 32-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_inserti32x4

```
__m256i _mm256_maskz_inserti32x4(__mmask8 k, __m256i a, __m128i b, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vinserti32x4

Copy *a* to *tmp*, then insert 128 bits (composed of 4 packed 32-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_inserti32x8

```
__m512i _mm512_inserti32x8(__m512i a, __m256i b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinserti32x8

Copy *a* to the return value, then insert 256 bits (composed of 8 packed 32-bit integers) from *b* into *dst* at the location specified by *imm*.

_mm512_mask_inserti32x8

```
__m512i _mm512_mask_inserti32x8(__m512i src, __mmask16 k, __m512i a, __m256i b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinserti32x8

Copy *a* to *tmp*, then insert 256 bits (composed of 8 packed 32-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_inserti32x8

```
__m512i _mm512_maskz_inserti32x8(__mmask16 k, __m512i a, __m256i b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinserti32x8

Copy *a* to *tmp*, then insert 256 bits (composed of 8 packed 32-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_inserti64x2

```
__m256i _mm256_inserti64x2(__m256i a, __m128i b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vinserti64x2

Copy *a* to the return value, then insert 128 bits (composed of 2 packed 64-bit integers) from *b* into *dst* at the location specified by *imm*.

_mm256_mask_inserti64x2

```
__m256i _mm256_mask_inserti64x2(__m256i src, __mmask8 k, __m256i a, __m128i b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vinserti64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed 64-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_inserti64x2

```
__m256i _mm256_maskz_inserti64x2(__mmask8 k, __m256i a, __m128i b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vinserti64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed 64-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_inserti64x2

```
__m512i _mm512_inserti64x2(__m512i a, __m128i b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinserti64x2

Copy *a* to the return value, then insert 128 bits (composed of 2 packed 64-bit integers) from *b* into *dst* at the location specified by *imm*.

_mm512_mask_inserti64x2

```
__m512i _mm512_mask_inserti64x2(__m512i src, __mmask8 k, __m512i a, __m128i b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinserti64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed 64-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_inserti64x2

```
__m512i _mm512_maskz_inserti64x2(__mmask8 k, __m512i a, __m128i b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinserti64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed 64-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_shuffle_i32x4

```
__m256i _mm256_mask_shuffle_i32x4(__m256i src, __mmask8 k, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufi32x4

Shuffle 128-bits (composed of 4 32-bit integers) selected by *imm* from *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_i32x4

```
__m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufi32x4

Shuffle 128-bits (composed of 4 32-bit integers) selected by *imm* from *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_shuffle_i32x4

```
__m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufi32x4

Shuffle 128-bits (composed of 4 32-bit integers) selected by *imm* from *a* and *b*, and return the results.

_mm256_mask_shuffle_i64x2

```
__m256i _mm256_mask_shuffle_i64x2(__m256i src, __mmask8 k, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufi64x2

Shuffle 128-bits (composed of 2 64-bit integers) selected by *imm* from *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_i64x2

```
__m256i _mm256_maskz_shuffle_i64x2(__mmask8 k, __m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufi64x2

Shuffle 128-bits (composed of 2 64-bit integers) selected by *imm* from *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_shuffle_i64x2

```
__m256i _mm256_shuffle_i64x2(__m256i a, __m256i b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufi64x2

Shuffle 128-bits (composed of 2 64-bit integers) selected by *imm* from *a* and *b*, and return the results.

_mm_mask_blend_pd

```
__m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vblendmpd

Blend packed double-precision (64-bit) floating-point elements from *a* and *b* using control mask *k*, and return the results.

_mm256_mask_blend_pd

```
__m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vblendmpd

Blend packed double-precision (64-bit) floating-point elements from *a* and *b* using control mask *k*, and return the results.

_mm_mask_blend_ps

```
__m128 _mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vblendmps

Blend packed single-precision (32-bit) floating-point elements from *a* and *b* using control mask *k*, and return the results.

_mm256_mask_blend_ps

```
__m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vblendmps

Blend packed single-precision (32-bit) floating-point elements from *a* and *b* using control mask *k*, and return the results.

_mm256_broadcast_f32x2

```
__m256 _mm256_broadcast_f32x2(__m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcastf32x2

Broadcast the lower 2 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value.

_mm256_mask_broadcast_f32x2

```
__m256 _mm256_mask_broadcast_f32x2(__m256 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcastf32x2

Broadcast the lower 2 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcast_f32x2

```
__m256 _mm256_maskz_broadcast_f32x2(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcastf32x2

Broadcast the lower 2 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_f32x2

```
__m512 _mm512_broadcast_f32x2(__m128 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf32x2

Broadcast the lower 2 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value.

_mm512_mask_broadcast_f32x2

```
__m512 _mm512_mask_broadcast_f32x2(__m512 src, __mmask16 k, __m128 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf32x2

Broadcast the lower 2 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_f32x2

```
__m512 _mm512_maskz_broadcast_f32x2(__mmask16 k, __m128 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf32x2

Broadcast the lower 2 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_broadcast_f32x4

```
__m256 _mm256_broadcast_f32x4(__m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastf32x4

Broadcast the 4 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value.

_mm256_mask_broadcast_f32x4

```
__m256 _mm256_mask_broadcast_f32x4(__m256 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastf32x4

Broadcast the 4 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcast_f32x4

```
__m256 _mm256_maskz_broadcast_f32x4(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastf32x4

Broadcast the 4 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_f32x8

```
__m512 _mm512_broadcast_f32x8(__m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf32x8

Broadcast the 8 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value.

_mm512_mask_broadcast_f32x8

```
__m512 _mm512_mask_broadcast_f32x8(__m512 src, __mmask16 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf32x8

Broadcast the 8 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_f32x8

```
__m512 _mm512_maskz_broadcast_f32x8(__mmask16 k, __m256 a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf32x8

Broadcast the 8 packed single-precision (32-bit) floating-point elements from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_broadcast_f64x2

```
__m256d _mm256_broadcast_f64x2(__m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcastf64x2

Broadcast the 2 packed double-precision (64-bit) floating-point elements from *a* to all elements of the return value.

_mm256_mask_broadcast_f64x2

```
__m256d _mm256_mask_broadcast_f64x2(__m256d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcastf64x2

Broadcast the 2 packed double-precision (64-bit) floating-point elements from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcast_f64x2

```
__m256d _mm256_maskz_broadcast_f64x2(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vbroadcastf64x2

Broadcast the 2 packed double-precision (64-bit) floating-point elements from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_f64x2

```
__m512d _mm512_broadcast_f64x2(__m128d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf64x2

Broadcast the 2 packed double-precision (64-bit) floating-point elements from *a* to all elements of the return value.

_mm512_mask_broadcast_f64x2

```
__m512d _mm512_mask_broadcast_f64x2(__m512d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf64x2

Broadcast the 2 packed double-precision (64-bit) floating-point elements from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_f64x2

```
__m512d _mm512_maskz_broadcast_f64x2(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512DQ

Instruction(s): vbroadcastf64x2

Broadcast the 2 packed double-precision (64-bit) floating-point elements from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_broadcastsd_pd

```
__m256d _mm256_mask_broadcastsd_pd(__m256d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastsd

Broadcast the low double-precision (64-bit) floating-point element from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcastsd_pd

```
__m256d _mm256_maskz_broadcastsd_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastsd

Broadcast the low double-precision (64-bit) floating-point element from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_broadcastss_ps

```
__m128 _mm_mask_broadcastss_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastss

Broadcast the low single-precision (32-bit) floating-point element from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_broadcastss_ps

```
__m128 _mm_maskz_broadcastss_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastss

Broadcast the low single-precision (32-bit) floating-point element from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_broadcastss_ps

```
__m256 _mm256_mask_broadcastss_ps(__m256 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastss

Broadcast the low single-precision (32-bit) floating-point element from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcastss_ps

```
__m256 _mm256_maskz_broadcastss_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vbroadcastss

Broadcast the low single-precision (32-bit) floating-point element from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_compress_pd

```
__m128d _mm_mask_compress_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompresspd

Contiguously store the active double-precision (64-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm_maskz_compress_pd

```
__m128d _mm_maskz_compress_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompresspd

Contiguously store the active double-precision (64-bit) floating-point elements in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm256_mask_compress_pd

```
__m256d _mm256_mask_compress_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompresspd

Contiguously store the active double-precision (64-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm256_maskz_compress_pd

```
__m256d _mm256_maskz_compress_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompresspd

Contiguously store the active double-precision (64-bit) floating-point elements in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm_mask_compress_ps

```
__m128 _mm_mask_compress_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompressps

Contiguously store the active single-precision (32-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm_maskz_compress_ps

```
__m128 _mm_maskz_compress_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompressps

Contiguously store the active single-precision (32-bit) floating-point elements in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm256_mask_compress_ps

```
__m256 _mm256_mask_compress_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompressps

Contiguously store the active single-precision (32-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm256_maskz_compress_ps

```
__m256 _mm256_maskz_compress_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompressps

Contiguously store the active single-precision (32-bit) floating-point elements in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm_mask_expand_pd

```
__m128d _mm_mask_expand_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load contiguous active double-precision (64-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expand_pd

```
__m128d _mm_maskz_expand_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load contiguous active double-precision (64-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expand_pd

```
__m256d _mm256_mask_expand_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load contiguous active double-precision (64-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expand_pd

```
__m256d _mm256_maskz_expand_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandpd

Load contiguous active double-precision (64-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_expand_ps

```
__m128 _mm_mask_expand_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load contiguous active single-precision (32-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expand_ps

```
__m128 _mm_maskz_expand_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load contiguous active single-precision (32-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expand_ps

```
__m256 _mm256_mask_expand_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load contiguous active single-precision (32-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expand_ps

```
__m256 _mm256_maskz_expand_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vexpandps

Load contiguous active single-precision (32-bit) floating-point elements from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_extractf32x4_ps

```
__m128 _mm256_extractf32x4_ps(__m256 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vextractf32x4

Extract 128 bits (composed of 4 packed single-precision (32-bit) floating-point elements) from *a*, selected with *imm*, and store the result in the return value.

_mm256_mask_extractf32x4_ps

```
__m128 _mm256_mask_extractf32x4_ps(__m128 src, __mmask8 k, __m256 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vextractf32x4

Extract 128 bits (composed of 4 packed single-precision (32-bit) floating-point elements) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_extractf32x4_ps

```
__m128 _mm256_maskz_extractf32x4_ps(__mmask8 k, __m256 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vextractf32x4

Extract 128 bits (composed of 4 packed single-precision (32-bit) floating-point elements) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_extractf32x8_ps

```
__m256 _mm512_extractf32x8_ps(__m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextractf32x8

Extract 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from *a*, selected with *imm*, and store the result in the return value.

_mm512_mask_extractf32x8_ps

```
__m256 _mm512_mask_extractf32x8_ps(__m256 src, __mmask8 k, __m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextractf32x8

Extract 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extractf32x8_ps

```
__m256 _mm512_maskz_extractf32x8_ps(__mmask8 k, __m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextractf32x8

Extract 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_extractf64x2_pd

```
__m128d _mm256_extractf64x2_pd(__m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vextractf64x2

Extract 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *a*, selected with *imm*, and store the result in the return value.

_mm256_mask_extractf64x2_pd

```
__m128d _mm256_mask_extractf64x2_pd(__m128d src, __mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vextractf64x2

Extract 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_extractf64x2_pd

```
__m128d _mm256_maskz_extractf64x2_pd(__mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vextractf64x2

Extract 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_extractf64x2_pd

```
__m128d _mm512_extractf64x2_pd(__m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextractf64x2

Extract 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *a*, selected with *imm*, and store the result in the return value.

_mm512_mask_extractf64x2_pd

```
__m128d _mm512_mask_extractf64x2_pd(__m128d src, __mmask8 k, __m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextractf64x2

Extract 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extractf64x2_pd

```
__m128d _mm512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextractf64x2

Extract 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_fixupimm_pd

```
__m128d _mm_fixupimm_pd(__m128d a, __m128d b, __m128i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmpd

Fix up packed double-precision (64-bit) floating-point elements in *a* and *b* using packed 64-bit integers in *c*, and return the results. *imm* is used to set the required flags reporting.

_mm_mask_fixupimm_pd

```
__m128d _mm_mask_fixupimm_pd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmpd

Fix up packed double-precision (64-bit) floating-point elements in *a* and *b* using packed 64-bit integers in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm_maskz_fixupimm_pd

```
__m128d _mm_maskz_fixupimm_pd(__mmask8 k, __m128d a, __m128d b, __m128i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmpd

Fix up packed double-precision (64-bit) floating-point elements in *a* and *b* using packed 64-bit integers in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm256_fixupimm_pd

```
__m256d _mm256_fixupimm_pd(__m256d a, __m256d b, __m256i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmpd

Fix up packed double-precision (64-bit) floating-point elements in *a* and *b* using packed 64-bit integers in *c*, and return the results. *imm* is used to set the required flags reporting.

_mm256_mask_fixupimm_pd

```
__m256d _mm256_mask_fixupimm_pd(__m256d a, __mmask8 k, __m256d b, __m256i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmpd

Fix up packed double-precision (64-bit) floating-point elements in *a* and *b* using packed 64-bit integers in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm256_maskz_fixupimm_pd

```
__m256d _mm256_maskz_fixupimm_pd(__mmask8 k, __m256d a, __m256d b, __m256i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmpd

Fix up packed double-precision (64-bit) floating-point elements in *a* and *b* using packed 64-bit integers in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm_fixupimm_ps

```
__m128 _mm_fixupimm_ps(__m128 a, __m128 b, __m128i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmps

Fix up packed single-precision (32-bit) floating-point elements in *a* and *b* using packed 32-bit integers in *c*, and return the results. *imm* is used to set the required flags reporting.

_mm_mask_fixupimm_ps

```
__m128 _mm_mask_fixupimm_ps(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmps

Fix up packed single-precision (32-bit) floating-point elements in *a* and *b* using packed 32-bit integers in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm_maskz_fixupimm_ps

```
__m128 _mm_maskz_fixupimm_ps(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmps

Fix up packed single-precision (32-bit) floating-point elements in *a* and *b* using packed 32-bit integers in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm256_fixupimm_ps

```
__m256 _mm256_fixupimm_ps(__m256 a, __m256 b, __m256i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmps

Fix up packed single-precision (32-bit) floating-point elements in *a* and *b* using packed 32-bit integers in *c*, and return the results. *imm* is used to set the required flags reporting.

_mm256_mask_fixupimm_ps

```
__m256 _mm256_mask_fixupimm_ps(__m256 a, __mmask8 k, __m256 b, __m256i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmps

Fix up packed single-precision (32-bit) floating-point elements in *a* and *b* using packed 32-bit integers in *c*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm256_maskz_fixupimm_ps

```
__m256 _mm256_maskz_fixupimm_ps(__mmask8 k, __m256 a, __m256 b, __m256i c, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vfixupimmps

Fix up packed single-precision (32-bit) floating-point elements in *a* and *b* using packed 32-bit integers in *c*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). *imm* is used to set the required flags reporting.

_mm_getexp_pd

```
__m128d _mm_getexp_pd(__m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexppd

Convert the exponent of each packed double-precision (64-bit) floating-point element in *a* to a double-precision (64-bit) floating-point number representing the integer exponent, and return the results. This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm_mask_getexp_pd

```
__m128d _mm_mask_getexp_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexppd

Convert the exponent of each packed double-precision (64-bit) floating-point element in *a* to a double-precision (64-bit) floating-point number representing the integer exponent, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm_maskz_getexp_pd

```
__m128d _mm_maskz_getexp_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexppd

Convert the exponent of each packed double-precision (64-bit) floating-point element in *a* to a double-precision (64-bit) floating-point number representing the integer exponent, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm256_getexp_pd

```
__m256d _mm256_getexp_pd(__m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexppd

Convert the exponent of each packed double-precision (64-bit) floating-point element in *a* to a double-precision (64-bit) floating-point number representing the integer exponent, and return the results. This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm256_mask_getexp_pd

```
__m256d _mm256_mask_getexp_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexppd

Convert the exponent of each packed double-precision (64-bit) floating-point element in *a* to a double-precision (64-bit) floating-point number representing the integer exponent, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm256_maskz_getexp_pd

```
__m256d _mm256_maskz_getexp_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexppd

Convert the exponent of each packed double-precision (64-bit) floating-point element in *a* to a double-precision (64-bit) floating-point number representing the integer exponent, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm_getexp_ps

```
__m128 _mm_getexp_ps(__m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexpps

Convert the exponent of each packed single-precision (32-bit) floating-point element in *a* to a single-precision (32-bit) floating-point number representing the integer exponent, and return the results. This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm_mask_getexp_ps

```
__m128 _mm_mask_getexp_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexpps

Convert the exponent of each packed single-precision (32-bit) floating-point element in *a* to a single-precision (32-bit) floating-point number representing the integer exponent, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm_maskz_getexp_ps

```
__m128 _mm_maskz_getexp_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexpps

Convert the exponent of each packed single-precision (32-bit) floating-point element in *a* to a single-precision (32-bit) floating-point number representing the integer exponent, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm256_getexp_ps

```
__m256 _mm256_getexp_ps(__m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexpps

Convert the exponent of each packed single-precision (32-bit) floating-point element in *a* to a single-precision (32-bit) floating-point number representing the integer exponent, and return the results. This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm256_mask_getexp_ps

```
__m256 _mm256_mask_getexp_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexpps

Convert the exponent of each packed single-precision (32-bit) floating-point element in *a* to a single-precision (32-bit) floating-point number representing the integer exponent, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm256_maskz_getexp_ps

```
__m256 _mm256_maskz_getexp_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetexpps

Convert the exponent of each packed single-precision (32-bit) floating-point element in *a* to a single-precision (32-bit) floating-point number representing the integer exponent, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates *floor(log2(x))* for each element.

_mm_getmant_pd

```
__m128d _mm_getmant_pd(__m128d a, _MM_MANTISSA_NORM_ENUM interv, _MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantpd

Normalize the mantissas of packed double-precision (64-bit) floating-point elements in *a*, and return the results. This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm_mask_getmant_pd

```
__m128d _mm_mask_getmant_pd(__m128d src, __mmask8 k, __m128d a, _MM_MANTISSA_NORM_ENUM interv,
_MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantpd

Normalize the mantissas of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm_maskz_getmant_pd

```
__m128d _mm_maskz_getmant_pd(__mmask8 k, __m128d a, _MM_MANTISSA_NORM_ENUM interv,
_MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantpd

Normalize the mantissas of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm256_getmant_pd

```
__m256d _mm256_getmant_pd(__m256d a, _MM_MANTISSA_NORM_ENUM interv, _MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantpd

Normalize the mantissas of packed double-precision (64-bit) floating-point elements in *a*, and return the results. This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm256_mask_getmant_pd

```
__m256d _mm256_mask_getmant_pd(__m256d src, __mmask8 k, __m256d a, _MM_MANTISSA_NORM_ENUM
interv, _MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantpd

Normalize the mantissas of packed double-precision (64-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm256_maskz_getmant_pd

```
__m256d _mm256_maskz_getmant_pd(__mmask8 k, __m256d a, _MM_MANTISSA_NORM_ENUM interv,
_MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantpd

Normalize the mantissas of packed double-precision (64-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm_getmant_ps

```
__m128 _mm_getmant_ps(__m128 a, _MM_MANTISSA_NORM_ENUM interv, _MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantps

Normalize the mantissas of packed single-precision (32-bit) floating-point elements in *a*, and return the results. This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm_mask_getmant_ps

```
__m128 _mm_mask_getmant_ps(__m128 src, __mmask8 k, __m128 a, _MM_MANTISSA_NORM_ENUM interv,
_MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantps

Normalize the mantissas of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm_maskz_getmant_ps

```
__m128 _mm_maskz_getmant_ps(__mmask8 k, __m128 a, _MM_MANTISSA_NORM_ENUM interv,
_MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantps

Normalize the mantissas of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm256_getmant_ps

```
__m256 _mm256_getmant_ps(__m256 a, _MM_MANTISSA_NORM_ENUM interv, _MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantps

Normalize the mantissas of packed single-precision (32-bit) floating-point elements in *a*, and return the results. This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm256_mask_getmant_ps

```
__m256 _mm256_mask_getmant_ps(__m256 src, __mmask8 k, __m256 a, _MM_MANTISSA_NORM_ENUM interv,
_MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantps

Normalize the mantissas of packed single-precision (32-bit) floating-point elements in *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm256_maskz_getmant_ps

```
__m256 _mm256_maskz_getmant_ps(__mmask8 k, __m256 a, _MM_MANTISSA_NORM_ENUM interv,
_MM_MANTISSA_SIGN_ENUM sc)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vgetmantps

Normalize the mantissas of packed single-precision (32-bit) floating-point elements in *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates ±*(2^k)**|*x.significand*|, where *k* depends on the interval range defined by *interv* and the sign depends on *sc* and the source sign.

_mm256_insertf32x4

```
__m256 _mm256_insertf32x4(__m256 a, __m128 b, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vinsertf32x4

Copy *a* to the return value, then insert 128 bits (composed of 4 packed single-precision (32-bit) floating-point elements) from *b* into *dst* at the location specified by *imm*.

_mm256_mask_insertf32x4

```
__m256 _mm256_mask_insertf32x4(__m256 src, __mmask8 k, __m256 a, __m128 b, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vinsertf32x4

Copy *a* to *tmp*, then insert 128 bits (composed of 4 packed single-precision (32-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_insertf32x4

```
__m256 _mm256_maskz_insertf32x4(__mmask8 k, __m256 a, __m128 b, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vinsertf32x4

Copy *a* to *tmp*, then insert 128 bits (composed of 4 packed single-precision (32-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_insertf32x8

```
__m512 _mm512_insertf32x8(__m512 a, __m256 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinsertf32x8

Copy *a* to the return value, then insert 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from *b* into *dst* at the location specified by *imm*.

_mm512_mask_insertf32x8

```
__m512 _mm512_mask_insertf32x8(__m512 src, __mmask16 k, __m512 a, __m256 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinsertf32x8

Copy *a* to *tmp*, then insert 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_insertf32x8

```
__m512 _mm512_maskz_insertf32x8(__mmask16 k, __m512 a, __m256 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinsertf32x8

Copy *a* to *tmp*, then insert 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_insertf64x2

```
__m256d _mm256_insertf64x2(__m256d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vinsertf64x2

Copy *a* to the return value, then insert 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *b* into *dst* at the location specified by *imm*.

_mm256_mask_insertf64x2

```
__m256d _mm256_mask_insertf64x2(__m256d src, __mmask8 k, __m256d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vinsertf64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_insertf64x2

```
__m256d _mm256_maskz_insertf64x2(__mmask8 k, __m256d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vinsertf64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_insertf64x2

```
__m512d _mm512_insertf64x2(__m512d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinsertf64x2

Copy *a* to the return value, then insert 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *b* into *dst* at the location specified by *imm*.

_mm512_mask_insertf64x2

```
__m512d _mm512_mask_insertf64x2(__m512d src, __mmask8 k, __m512d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinsertf64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_insertf64x2

```
__m512d _mm512_maskz_insertf64x2(__mmask8 k, __m512d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vinsertf64x2

Copy *a* to *tmp*, then insert 128 bits (composed of 2 packed double-precision (64-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask2_permutex2var_pd

```
__m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set)

_mm256_mask2_permutex2var_pd

```
__m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm_maskz_permutex2var_pd

```
__m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2pd, vpermt2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_permutex2var_pd

```
__m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2pd, vpermt2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm256_maskz_permutex2var_pd

```
__m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2pd, vpermt2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex2var_pd

```
__m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2pd, vpermt2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm_mask2_permutex2var_ps

```
__m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm256_mask2_permutex2var_ps

```
__m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm_maskz_permutex2var_ps

```
__m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2ps, vpermt2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_permutex2var_ps

```
__m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2ps, vpermt2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm256_maskz_permutex2var_ps

```
__m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2ps, vpermt2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex2var_ps

```
__m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2ps, vpermt2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm_mask_permute_pd

```
__m128d _mm_mask_permute_pd(__m128d src, __mmask8 k, __m128d a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_permutevar_pd

```
__m128d _mm_mask_permutevar_pd(__m128d src, __mmask8 k, __m128d a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* using the control in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_permute_pd

```
__m128d _mm_maskz_permute_pd(__mmask8 k, __m128d a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_permutevar_pd

```
__m128d _mm_maskz_permutevar_pd(__mmask8 k, __m128d a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* using the control in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_permute_pd

```
__m256d _mm256_mask_permute_pd(__m256d src, __mmask8 k, __m256d a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* within 128-bit lanes using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_permutevar_pd

```
__m256d _mm256_mask_permutevar_pd(__m256d src, __mmask8 k, __m256d a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* within 128-bit lanes using the control in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permute_pd

```
__m256d _mm256_maskz_permute_pd(__mmask8 k, __m256d a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* within 128-bit lanes using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_permutevar_pd

```
__m256d _mm256_maskz_permutevar_pd(__mmask8 k, __m256d a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilpd

Shuffle double-precision (64-bit) floating-point elements in *a* within 128-bit lanes using the control in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_permute_ps

```
__m128 _mm_mask_permute_ps(__m128 src, __mmask8 k, __m128 a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_permutevar_ps

```
__m128 _mm_mask_permutevar_ps(__m128 src, __mmask8 k, __m128 a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* using the control in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_permute_ps

```
__m128 _mm_maskz_permute_ps(__mmask8 k, __m128 a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_permutevar_ps

```
__m128 _mm_maskz_permutevar_ps(__mmask8 k, __m128 a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* using the control in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_permute_ps

```
__m256 _mm256_mask_permute_ps(__m256 src, __mmask8 k, __m256 a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* within 128-bit lanes using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_permutevar_ps

```
__m256 _mm256_mask_permutevar_ps(__m256 src, __mmask8 k, __m256 a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* within 128-bit lanes using the control in *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permute_ps

```
__m256 _mm256_maskz_permute_ps(__mmask8 k, __m256 a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* within 128-bit lanes using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_permutevar_ps

```
__m256 _mm256_maskz_permutevar_ps(__mmask8 k, __m256 a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermilps

Shuffle single-precision (32-bit) floating-point elements in *a* within 128-bit lanes using the control in *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_permutex_pd

```
__m256d _mm256_mask_permutex_pd(__m256d src, __mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermpd

Shuffle double-precision (64-bit) floating-point elements in *a* across lanes using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_permutexvar_pd

```
__m256d _mm256_mask_permutexvar_pd(__m256d src, __mmask8 k, __m256i idx, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermpd

Shuffle double-precision (64-bit) floating-point elements in *a* across lanes using the corresponding index in *idx*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permutex_pd

```
__m256d _mm256_maskz_permutex_pd(__mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermpd

Shuffle double-precision (64-bit) floating-point elements in *a* across lanes using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_permutexvar_pd

```
__m256d _mm256_maskz_permutexvar_pd(__mmask8 k, __m256i idx, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermpd

Shuffle double-precision (64-bit) floating-point elements in *a* across lanes using the corresponding index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex_pd

```
__m256d _mm256_permutex_pd(__m256d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermpd

Shuffle double-precision (64-bit) floating-point elements in *a* across lanes using the control in *imm*, and return the results.

_mm256_permutexvar_pd

```
__m256d _mm256_permutexvar_pd(__m256i idx, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermpd

Shuffle double-precision (64-bit) floating-point elements in *a* across lanes using the corresponding index in *idx*, and return the results.

_mm256_mask_permutexvar_ps

```
__m256 _mm256_mask_permutexvar_ps(__m256 src, __mmask8 k, __m256i idx, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermps

Shuffle single-precision (32-bit) floating-point elements in *a* across lanes using the corresponding index in *idx*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permutexvar_ps

```
__m256 _mm256_maskz_permutexvar_ps(__mmask8 k, __m256i idx, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermps

Shuffle single-precision (32-bit) floating-point elements in *a* across lanes using the corresponding index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutexvar_ps

```
__m256 _mm256_permutexvar_ps(__m256i idx, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermps

Shuffle single-precision (32-bit) floating-point elements in *a* across lanes using the corresponding index in *idx*.

_mm_mask_permutex2var_pd

```
__m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask_permutex2var_pd

```
__m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2pd

Shuffle double-precision (64-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask_permutex2var_ps

```
__m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask_permutex2var_ps

```
__m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2ps

Shuffle single-precision (32-bit) floating-point elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask_range_pd

```
__m128d _mm_mask_range_pd(__m128d src, __mmask8 k, __m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_range_pd

```
__m128d _mm_maskz_range_pd(__mmask8 k, __m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_range_pd

```
__m128d _mm_range_pd(__m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm256_mask_range_pd

```
__m256d _mm256_mask_range_pd(__m256d src, __mmask8 k, __m256d a, __m256d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_range_pd

```
__m256d _mm256_maskz_range_pd(__mmask8 k, __m256d a, __m256d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_range_pd

```
__m256d _mm256_range_pd(__m256d a, __m256d b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_mask_range_pd

```
__m512d _mm512_mask_range_pd(__m512d src, __mmask8 k, __m512d a, __m512d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_range_round_pd

```
__m512d _mm512_mask_range_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b, int imm, int
rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_range_pd

```
__m512d _mm512_maskz_range_pd(__mmask8 k, __m512d a, __m512d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_range_round_pd

```
__m512d _mm512_maskz_range_round_pd(__mmask8 k, __m512d a, __m512d b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_range_pd

```
__m512d _mm512_range_pd(__m512d a, __m512d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_range_round_pd

```
__m512d _mm512_range_round_pd(__m512d a, __m512d b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangepd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed double-precision (64-bit) floating-point elements in *a* and *b*, and return the results.

_mm_mask_range_ps

```
__m128 _mm_mask_range_ps(__m128 src, __mmask8 k, __m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_range_ps

```
__m128 _mm_maskz_range_ps(__mmask8 k, __m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_range_ps

```
__m128 _mm_range_ps(__m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm256_mask_range_ps

```
__m256 _mm256_mask_range_ps(__m256 src, __mmask8 k, __m256 a, __m256 b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_range_ps

```
__m256 _mm256_maskz_range_ps(__mmask8 k, __m256 a, __m256 b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_range_ps

```
__m256 _mm256_range_ps(__m256 a, __m256 b, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_mask_range_ps

```
__m512 _mm512_mask_range_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_range_round_ps

```
__m512 _mm512_mask_range_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int imm, int
rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_range_ps

```
__m512 _mm512_maskz_range_ps(__mmask16 k, __m512 a, __m512 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_range_round_ps

```
__m512 _mm512_maskz_range_round_ps(__mmask16 k, __m512 a, __m512 b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_range_ps

```
__m512 _mm512_range_ps(__m512 a, __m512 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm512_range_round_ps

```
__m512 _mm512_range_round_ps(__m512 a, __m512 b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangeps

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for packed single-precision (32-bit) floating-point elements in *a* and *b*, and return the results.

_mm_mask_range_round_sd

```
__m128d _mm_mask_range_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int imm, int
rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangesd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower double-precision (64-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper element of *dst*.

_mm_mask_range_sd

```
__m128d _mm_mask_range_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangesd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower double-precision (64-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper element of *dst*.

_mm_maskz_range_round_sd

```
__m128d _mm_maskz_range_round_sd(__mmask8 k, __m128d a, __m128d b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangesd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower double-precision (64-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper element of *dst*.

_mm_maskz_range_sd

```
__m128d _mm_maskz_range_sd(__mmask8 k, __m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangesd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower double-precision (64-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper element of *dst*.

_mm_range_round_sd

```
__m128d _mm_range_round_sd(__m128d a, __m128d b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangesd

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower double-precision (64-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value, and copy the upper element from *a* to the upper element of *dst*.

_mm_mask_range_round_ss

```
__m128 _mm_mask_range_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangess

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower single-precision (32-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper 3 packed elements from *a* to the upper elements of *dst*.

_mm_mask_range_ss

```
__m128 _mm_mask_range_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangess

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower single-precision (32-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper 3 packed elements from *a* to the upper elements of *dst*.

_mm_maskz_range_round_ss

```
__m128 _mm_maskz_range_round_ss(__mmask8 k, __m128 a, __m128 b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangess

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower single-precision (32-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper 3 packed elements from *a* to the upper elements of *dst*.

_mm_maskz_range_ss

```
__m128 _mm_maskz_range_ss(__mmask8 k, __m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangess

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower single-precision (32-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper 3 packed elements from *a* to the upper elements of *dst*.

_mm_range_round_ss

```
__m128 _mm_range_round_ss(__m128 a, __m128 b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vrangess

Calculate the max, min, absolute max, or absolute min (depending on control in *imm*) for the lower single-precision (32-bit) floating-point element in *a* and *b*, store the result in the lower element of the return value, and copy the upper 3 packed elements from *a* to the upper elements of *dst*.

_mm_mask_reduce_pd

```
__m128d _mm_mask_reduce_pd(__m128d src, __mmask8 k, __m128d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_reduce_pd

```
__m128d _mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_reduce_pd

```
__m128d _mm_reduce_pd(__m128d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm256_mask_reduce_pd

```
__m256d _mm256_mask_reduce_pd(__m256d src, __mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_reduce_pd

```
__m256d _mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_reduce_pd

```
__m256d _mm256_reduce_pd(__m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm512_mask_reduce_pd

```
__m512d _mm512_mask_reduce_pd(__m512d src, __mmask8 k, __m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_reduce_round_pd

```
__m512d _mm512_mask_reduce_round_pd(__m512d src, __mmask8 k, __m512d a, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_reduce_pd

```
__m512d _mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_reduce_round_pd

```
__m512d _mm512_maskz_reduce_round_pd(__mmask8 k, __m512d a, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_reduce_pd

```
__m512d _mm512_reduce_pd(__m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm512_reduce_round_pd

```
__m512d _mm512_reduce_round_pd(__m512d a, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducepd

Extract the reduced argument of packed double-precision (64-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm_mask_reduce_ps

```
__m128 _mm_mask_reduce_ps(__m128 src, __mmask8 k, __m128 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_reduce_ps

```
__m128 _mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_reduce_ps

```
__m128 _mm_reduce_ps(__m128 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm256_mask_reduce_ps

```
__m256 _mm256_mask_reduce_ps(__m256 src, __mmask8 k, __m256 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_reduce_ps

```
__m256 _mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_reduce_ps

```
__m256 _mm256_reduce_ps(__m256 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm512_mask_reduce_ps

```
__m512 _mm512_mask_reduce_ps(__m512 src, __mmask16 k, __m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_reduce_round_ps

```
__m512 _mm512_mask_reduce_round_ps(__m512 src, __mmask16 k, __m512 a, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_reduce_ps

```
__m512 _mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_reduce_round_ps

```
__m512 _mm512_maskz_reduce_round_ps(__mmask16 k, __m512 a, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_reduce_ps

```
__m512 _mm512_reduce_ps(__m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm512_reduce_round_ps

```
__m512 _mm512_reduce_round_ps(__m512 a, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreduceps

Extract the reduced argument of packed single-precision (32-bit) floating-point elements in *a* by the number of bits specified by *imm*, and return the results.

_mm_mask_reduce_round_sd

```
__m128d _mm_mask_reduce_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int imm, int
rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducesd

Extract the reduced argument of the lower double-precision (64-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *b* to the upper element of *dst*.

_mm_mask_reduce_sd

```
__m128d _mm_mask_reduce_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducesd

Extract the reduced argument of the lower double-precision (64-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *b* to the upper element of *dst*.

_mm_maskz_reduce_round_sd

```
__m128d _mm_maskz_reduce_round_sd(__mmask8 k, __m128d a, __m128d b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducesd

Extract the reduced argument of the lower double-precision (64-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *b* to the upper element of *dst*.

_mm_maskz_reduce_sd

```
__m128d _mm_maskz_reduce_sd(__mmask8 k, __m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducesd

Extract the reduced argument of the lower double-precision (64-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *b* to the upper element of *dst*.

_mm_reduce_round_sd

```
__m128d _mm_reduce_round_sd(__m128d a, __m128d b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducesd

Extract the reduced argument of the lower double-precision (64-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value, and copy the upper element from *b* to the upper element of *dst*.

_mm_reduce_sd

```
__m128d _mm_reduce_sd(__m128d a, __m128d b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducesd

Extract the reduced argument of the lower double-precision (64-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value, and copy the upper element from *b* to the upper element of *dst*.

_mm_mask_reduce_round_ss

```
__m128 _mm_mask_reduce_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int imm, int
rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducess

Extract the reduced argument of the lower single-precision (32-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper 3 packed elements from *b* to the upper elements of *dst*.

_mm_mask_reduce_ss

```
__m128 _mm_mask_reduce_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducess

Extract the reduced argument of the lower single-precision (32-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper 3 packed elements from *b* to the upper elements of *dst*.

_mm_maskz_reduce_round_ss

```
__m128 _mm_maskz_reduce_round_ss(__mmask8 k, __m128 a, __m128 b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducess

Extract the reduced argument of the lower single-precision (32-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper 3 packed elements from *b* to the upper elements of *dst*.

_mm_maskz_reduce_ss

```
__m128 _mm_maskz_reduce_ss(__mmask8 k, __m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducess

Extract the reduced argument of the lower single-precision (32-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper 3 packed elements from *b* to the upper elements of *dst*.

_mm_reduce_round_ss

```
__m128 _mm_reduce_round_ss(__m128 a, __m128 b, int imm, int rounding)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducess

Extract the reduced argument of the lower single-precision (32-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value, and copy the upper 3 packed elements from *b* to the upper elements of *dst*.

_mm_reduce_ss

```
__m128 _mm_reduce_ss(__m128 a, __m128 b, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vreducess

Extract the reduced argument of the lower single-precision (32-bit) floating-point element in *a* by the number of bits specified by *imm*, store the result in the lower element of the return value, and copy the upper 3 packed elements from *b* to the upper elements of *dst*.

_mm_mask_roundscale_pd

```
__m128d _mm_mask_roundscale_pd(__m128d src, __mmask8 k, __m128d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscalepd

Round packed double-precision (64-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_roundscale_pd

```
__m128d _mm_maskz_roundscale_pd(__mmask8 k, __m128d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscalepd

Round packed double-precision (64-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_roundscale_pd

```
__m128d _mm_roundscale_pd(__m128d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscalepd

Round packed double-precision (64-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results.

_mm256_mask_roundscale_pd

```
__m256d _mm256_mask_roundscale_pd(__m256d src, __mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscalepd

Round packed double-precision (64-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_roundscale_pd

```
__m256d _mm256_maskz_roundscale_pd(__mmask8 k, __m256d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscalepd

Round packed double-precision (64-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_roundscale_pd

```
__m256d _mm256_roundscale_pd(__m256d a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscalepd

Round packed double-precision (64-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results.

_mm_mask_roundscale_ps

```
__m128 _mm_mask_roundscale_ps(__m128 src, __mmask8 k, __m128 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscaleps

Round packed single-precision (32-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_roundscale_ps

```
__m128 _mm_maskz_roundscale_ps(__mmask8 k, __m128 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscaleps

Round packed single-precision (32-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_roundscale_ps

```
__m128 _mm_roundscale_ps(__m128 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscaleps

Round packed single-precision (32-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results.

_mm256_mask_roundscale_ps

```
__m256 _mm256_mask_roundscale_ps(__m256 src, __mmask8 k, __m256 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscaleps

Round packed single-precision (32-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_roundscale_ps

```
__m256 _mm256_maskz_roundscale_ps(__mmask8 k, __m256 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscaleps

Round packed single-precision (32-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_roundscale_ps

```
__m256 _mm256_roundscale_ps(__m256 a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vrndscaleps

Round packed single-precision (32-bit) floating-point elements in *a* to the number of fraction bits specified by *imm*, and return the results.

_mm_mask_scalef_pd

```
__m128d _mm_mask_scalef_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefpd

Scale the packed double-precision (64-bit) floating-point elements in *a* using values from *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_scalef_pd

```
__m128d _mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefpd

Scale the packed double-precision (64-bit) floating-point elements in *a* using values from *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_scalef_pd

```
__m128d _mm_scalef_pd(__m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefpd

Scale the packed double-precision (64-bit) floating-point elements in *a* using values from *b*, and return the results.

_mm256_mask_scalef_pd

```
__m256d _mm256_mask_scalef_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefpd

Scale the packed double-precision (64-bit) floating-point elements in *a* using values from *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_scalef_pd

```
__m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefpd

Scale the packed double-precision (64-bit) floating-point elements in *a* using values from *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_scalef_pd

```
__m256d _mm256_scalef_pd(__m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefpd

Scale the packed double-precision (64-bit) floating-point elements in *a* using values from *b*, and return the results.

_mm_mask_scalef_ps

```
__m128 _mm_mask_scalef_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefps

Scale the packed single-precision (32-bit) floating-point elements in *a* using values from *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_scalef_ps

```
__m128 _mm_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefps

Scale the packed single-precision (32-bit) floating-point elements in *a* using values from *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_scalef_ps

```
__m128 _mm_scalef_ps(__m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefps

Scale the packed single-precision (32-bit) floating-point elements in *a* using values from *b*, and return the results.

_mm256_mask_scalef_ps

```
__m256 _mm256_mask_scalef_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefps

Scale the packed single-precision (32-bit) floating-point elements in *a* using values from *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_scalef_ps

```
__m256 _mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefps

Scale the packed single-precision (32-bit) floating-point elements in *a* using values from *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_scalef_ps

```
__m256 _mm256_scalef_ps(__m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscalefps

Scale the packed single-precision (32-bit) floating-point elements in *a* using values from *b*, and return the results.

_mm256_mask_shuffle_f32x4

```
__m256 _mm256_mask_shuffle_f32x4(__m256 src, __mmask8 k, __m256 a, __m256 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshuff32x4

Shuffle 128-bits (composed of 4 single-precision (32-bit) floating-point elements) selected by *imm* from *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_f32x4

```
__m256 _mm256_maskz_shuffle_f32x4(__mmask8 k, __m256 a, __m256 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshuff32x4

Shuffle 128-bits (composed of 4 single-precision (32-bit) floating-point elements) selected by *imm* from *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_shuffle_f32x4

```
__m256 _mm256_shuffle_f32x4(__m256 a, __m256 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshuff32x4

Shuffle 128-bits (composed of 4 single-precision (32-bit) floating-point elements) selected by *imm* from *a* and *b*, and return the results.

_mm256_mask_shuffle_f64x2

```
__m256d _mm256_mask_shuffle_f64x2(__m256d src, __mmask8 k, __m256d a, __m256d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshuff64x2

Shuffle 128-bits (composed of 2 double-precision (64-bit) floating-point elements) selected by *imm* from *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_f64x2

```
__m256d _mm256_maskz_shuffle_f64x2(__mmask8 k, __m256d a, __m256d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshuff64x2

Shuffle 128-bits (composed of 2 double-precision (64-bit) floating-point elements) selected by *imm* from *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_shuffle_f64x2

```
__m256d _mm256_shuffle_f64x2(__m256d a, __m256d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshuff64x2

Shuffle 128-bits (composed of 2 double-precision (64-bit) floating-point elements) selected by *imm* from *a* and *b*, and return the results.

_mm_mask_shuffle_pd

```
__m128d _mm_mask_shuffle_pd(__m128d src, __mmask8 k, __m128d a, __m128d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufpd

Shuffle double-precision (64-bit) floating-point elements using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_shuffle_pd

```
__m128d _mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufpd

Shuffle double-precision (64-bit) floating-point elements using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_shuffle_pd

```
__m256d _mm256_mask_shuffle_pd(__m256d src, __mmask8 k, __m256d a, __m256d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufpd

Shuffle double-precision (64-bit) floating-point elements within 128-bit lanes using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_pd

```
__m256d _mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufpd

Shuffle double-precision (64-bit) floating-point elements within 128-bit lanes using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_shuffle_ps

```
__m128 _mm_mask_shuffle_ps(__m128 src, __mmask8 k, __m128 a, __m128 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufps

Shuffle single-precision (32-bit) floating-point elements in *a* using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_shuffle_ps

```
__m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufps

Shuffle single-precision (32-bit) floating-point elements in *a* using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_shuffle_ps

```
__m256 _mm256_mask_shuffle_ps(__m256 src, __mmask8 k, __m256 a, __m256 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufps

Shuffle single-precision (32-bit) floating-point elements in *a* within 128-bit lanes using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_ps

```
__m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vshufps

Shuffle single-precision (32-bit) floating-point elements in *a* within 128-bit lanes using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpackhi_pd

```
__m128d _mm_mask_unpackhi_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhpd

Unpack and interleave double-precision (64-bit) floating-point elements from the high half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpackhi_pd

```
__m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhpd

Unpack and interleave double-precision (64-bit) floating-point elements from the high half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpackhi_pd

```
__m256d _mm256_mask_unpackhi_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhpd

Unpack and interleave double-precision (64-bit) floating-point elements from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpackhi_pd

```
__m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhpd

Unpack and interleave double-precision (64-bit) floating-point elements from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpackhi_ps

```
__m128 _mm_mask_unpackhi_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhps

Unpack and interleave single-precision (32-bit) floating-point elements from the high half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpackhi_ps

```
__m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhps

Unpack and interleave single-precision (32-bit) floating-point elements from the high half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpackhi_ps

```
__m256 _mm256_mask_unpackhi_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhps

Unpack and interleave single-precision (32-bit) floating-point elements from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpackhi_ps

```
__m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpckhps

Unpack and interleave single-precision (32-bit) floating-point elements from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpacklo_pd

```
__m128d _mm_mask_unpacklo_pd(__m128d src, __mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklpd

Unpack and interleave double-precision (64-bit) floating-point elements from the low half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpacklo_pd

```
__m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklpd

Unpack and interleave double-precision (64-bit) floating-point elements from the low half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpacklo_pd

```
__m256d _mm256_mask_unpacklo_pd(__m256d src, __mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklpd

Unpack and interleave double-precision (64-bit) floating-point elements from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpacklo_pd

```
__m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklpd

Unpack and interleave double-precision (64-bit) floating-point elements from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpacklo_ps

```
__m128 _mm_mask_unpacklo_ps(__m128 src, __mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklps

Unpack and interleave single-precision (32-bit) floating-point elements from the low half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpacklo_ps

```
__m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklps

Unpack and interleave single-precision (32-bit) floating-point elements from the low half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpacklo_ps

```
__m256 _mm256_mask_unpacklo_ps(__m256 src, __mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklps

Unpack and interleave single-precision (32-bit) floating-point elements from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpacklo_ps

```
__m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vunpcklps

Unpack and interleave single-precision (32-bit) floating-point elements from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_alignr_epi32

```
__m128i _mm_alignr_epi32(__m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignd

Concatenate *a* and *b* into a 32-byte immediate result, shift the result right by *count* 32-bit elements, and store the low 16 bytes (4 elements) in the return value.

_mm_mask_alignr_epi32

```
__m128i _mm_mask_alignr_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignd

Concatenate *a* and *b* into a 32-byte immediate result, shift the result right by *count* 32-bit elements, and store the low 16 bytes (4 elements) in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_alignr_epi32

```
__m128i _mm_maskz_alignr_epi32(__mmask8 k, __m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignd

Concatenate *a* and *b* into a 32-byte immediate result, shift the result right by *count* 32-bit elements, and store the low 16 bytes (4 elements) in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_alignr_epi32

```
__m256i _mm256_alignr_epi32(__m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignd

Concatenate *a* and *b* into a 64-byte immediate result, shift the result right by *count* 32-bit elements, and store the low 32 bytes (8 elements) in the return value.

_mm256_mask_alignr_epi32

```
__m256i _mm256_mask_alignr_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignd

Concatenate *a* and *b* into a 64-byte immediate result, shift the result right by *count* 32-bit elements, and store the low 32 bytes (8 elements) in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_alignr_epi32

```
__m256i _mm256_maskz_alignr_epi32(__mmask8 k, __m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignd

Concatenate *a* and *b* into a 64-byte immediate result, shift the result right by *count* 32-bit elements, and store the low 32 bytes (8 elements) in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_alignr_epi64

```
__m128i _mm_alignr_epi64(__m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignq

Concatenate *a* and *b* into a 32-byte immediate result, shift the result right by *count* 64-bit elements, and store the low 16 bytes (2 elements) in the return value.

_mm_mask_alignr_epi64

```
__m128i _mm_mask_alignr_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignq

Concatenate *a* and *b* into a 32-byte immediate result, shift the result right by *count* 64-bit elements, and store the low 16 bytes (2 elements) in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_alignr_epi64

```
__m128i _mm_maskz_alignr_epi64(__mmask8 k, __m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignq

Concatenate *a* and *b* into a 32-byte immediate result, shift the result right by *count* 64-bit elements, and store the low 16 bytes (2 elements) in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_alignr_epi64

```
__m256i _mm256_alignr_epi64(__m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignq

Concatenate *a* and *b* into a 64-byte immediate result, shift the result right by *count* 64-bit elements, and store the low 32 bytes (4 elements) in the return value.

_mm256_mask_alignr_epi64

```
__m256i _mm256_mask_alignr_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignq

Concatenate *a* and *b* into a 64-byte immediate result, shift the result right by *count* 64-bit elements, and store the low 32 bytes (4 elements) in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_alignr_epi64

```
__m256i _mm256_maskz_alignr_epi64(__mmask8 k, __m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): valignq

Concatenate *a* and *b* into a 64-byte immediate result, shift the result right by *count* 64-bit elements, and store the low 32 bytes (4 elements) in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_dbsad_epu8

```
__m128i _mm_dbsad_epu8(__m128i a, __m128i b, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value.

_mm_mask_dbsad_epu8

```
__m128i _mm_mask_dbsad_epu8(__m128i src, __mmask8 k, __m128i a, __m128i b, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_dbsad_epu8

```
__m128i _mm_maskz_dbsad_epu8(__mmask8 k, __m128i a, __m128i b, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_dbsad_epu8

```
__m256i _mm256_dbsad_epu8(__m256i a, __m256i b, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value.

_mm256_mask_dbsad_epu8

```
__m256i _mm256_mask_dbsad_epu8(__m256i src, __mmask16 k, __m256i a, __m256i b, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_dbsad_epu8

```
__m256i _mm256_maskz_dbsad_epu8(__mmask16 k, __m256i a, __m256i b, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_dbsad_epu8

```
__m512i _mm512_dbsad_epu8(__m512i a, __m512i b, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value.

_mm512_mask_dbsad_epu8

```
__m512i _mm512_mask_dbsad_epu8(__m512i src, __mmask32 k, __m512i a, __m512i b, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_dbsad_epu8

```
__m512i _mm512_maskz_dbsad_epu8(__mmask32 k, __m512i a, __m512i b, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vdbpsadbw

Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in *a* compared to those in *b*, and store the 16-bit results in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_extracti32x4_epi32

```
__m128i _mm256_extracti32x4_epi32(__m256i a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vextracti32x4

Extract 128 bits (composed of 4 packed 32-bit integers) from *a*, selected with *imm*, and store the result in the return value.

_mm256_mask_extracti32x4_epi32

```
__m128i _mm256_mask_extracti32x4_epi32(__m128i src, __mmask8 k, __m256i a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vextracti32x4

Extract 128 bits (composed of 4 packed 32-bit integers) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_extracti32x4_epi32

```
__m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vextracti32x4

Extract 128 bits (composed of 4 packed 32-bit integers) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_extracti32x8_epi32

```
__m256i _mm512_extracti32x8_epi32(__m512i a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextracti32x8

Extract 256 bits (composed of 8 packed 32-bit integers) from *a*, selected with *imm*, and store the result in the return value.

_mm512_mask_extracti32x8_epi32

```
__m256i _mm512_mask_extracti32x8_epi32(__m256i src, __mmask8 k, __m512i a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextracti32x8

Extract 256 bits (composed of 8 packed 32-bit integers) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extracti32x8_epi32

```
__m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextracti32x8

Extract 256 bits (composed of 8 packed 32-bit integers) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_extracti64x2_epi64

```
__m128i _mm256_extracti64x2_epi64(__m256i a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vextracti64x2

Extract 128 bits (composed of 2 packed 64-bit integers) from *a*, selected with *imm*, and store the result in the return value.

_mm256_mask_extracti64x2_epi64

```
__m128i _mm256_mask_extracti64x2_epi64(__m128i src, __mmask8 k, __m256i a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vextracti64x2

Extract 128 bits (composed of 2 packed 64-bit integers) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_extracti64x2_epi64

```
__m128i _mm256_maskz_extracti64x2_epi64(__mmask8 k, __m256i a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vextracti64x2

Extract 128 bits (composed of 2 packed 64-bit integers) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_extracti64x2_epi64

```
__m128i _mm512_extracti64x2_epi64(__m512i a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextracti64x2

Extract 128 bits (composed of 2 packed 64-bit integers) from *a*, selected with *imm*, and store the result in the return value.

_mm512_mask_extracti64x2_epi64

```
__m128i _mm512_mask_extracti64x2_epi64(__m128i src, __mmask8 k, __m512i a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextracti64x2

Extract 128 bits (composed of 2 packed 64-bit integers) from *a*, selected with *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extracti64x2_epi64

```
__m128i _mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vextracti64x2

Extract 128 bits (composed of 2 packed 64-bit integers) from *a*, selected with *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_alignr_epi8

```
__m128i _mm_mask_alignr_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpalignr

Concatenate pairs of 16-byte blocks in *a* and *b* into a 32-byte temporary result, shift the result right by *count* bytes, and store the low 16 bytes in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_alignr_epi8

```
__m128i _mm_maskz_alignr_epi8(__mmask16 k, __m128i a, __m128i b, const int count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpalignr

Concatenate pairs of 16-byte blocks in *a* and *b* into a 32-byte temporary result, shift the result right by *count* bytes, and store the low 16 bytes in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_alignr_epi8

```
__m256i _mm256_mask_alignr_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpalignr

Concatenate pairs of 16-byte blocks in *a* and *b* into a 32-byte temporary result, shift the result right by *count* bytes, and store the low 16 bytes in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_alignr_epi8

```
__m256i _mm256_maskz_alignr_epi8(__mmask32 k, __m256i a, __m256i b, const int count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpalignr

Concatenate pairs of 16-byte blocks in *a* and *b* into a 32-byte temporary result, shift the result right by *count* bytes, and store the low 16 bytes in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_alignr_epi8

```
__m512i _mm512_alignr_epi8(__m512i a, __m512i b, const int count)
```

CPUID Flags: AVX512BW

Instruction(s): vpalignr

Concatenate pairs of 16-byte blocks in *a* and *b* into a 32-byte temporary result, shift the result right by *count* bytes, and store the low 16 bytes in the return value.

_mm512_mask_alignr_epi8

```
__m512i _mm512_mask_alignr_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b, const int count)
```

CPUID Flags: AVX512BW

Instruction(s): vpalignr

Concatenate pairs of 16-byte blocks in *a* and *b* into a 32-byte temporary result, shift the result right by *count* bytes, and store the low 16 bytes in the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_alignr_epi8

```
__m512i _mm512_maskz_alignr_epi8(__mmask64 k, __m512i a, __m512i b, const int count)
```

CPUID Flags: AVX512BW

Instruction(s): vpalignr

Concatenate pairs of 16-byte blocks in *a* and *b* into a 32-byte temporary result, shift the result right by *count* bytes, and store the low 16 bytes in the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_blend_epi8

```
__m128i _mm_mask_blend_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpblendmb

Blend packed 8-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm256_mask_blend_epi8

```
__m256i _mm256_mask_blend_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpblendmb

Blend packed 8-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm512_mask_blend_epi8

```
__m512i _mm512_mask_blend_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpblendmb

Blend packed 8-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm_mask_blend_epi32

```
__m128i _mm_mask_blend_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpblendmd

Blend packed 32-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm256_mask_blend_epi32

```
__m256i _mm256_mask_blend_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpblendmd

Blend packed 32-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm_mask_blend_epi64

```
__m128i _mm_mask_blend_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpblendmq

Blend packed 64-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm256_mask_blend_epi64

```
__m256i _mm256_mask_blend_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpblendmq

Blend packed 64-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm_mask_blend_epi16

```
__m128i _mm_mask_blend_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpblendmw

Blend packed 16-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm256_mask_blend_epi16

```
__m256i _mm256_mask_blend_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpblendmw

Blend packed 16-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm512_mask_blend_epi16

```
__m512i _mm512_mask_blend_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpblendmw

Blend packed 16-bit integers from *a* and *b* using control mask *k*, and return the results.

_mm_mask_broadcastb_epi8

```
__m128i _mm_mask_broadcastb_epi8(__m128i src, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast the low packed 8-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_broadcastb_epi8

```
__m128i _mm_maskz_broadcastb_epi8(__mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast the low packed 8-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_broadcastb_epi8

```
__m256i _mm256_mask_broadcastb_epi8(__m256i src, __mmask32 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast the low packed 8-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcastb_epi8

```
__m256i _mm256_maskz_broadcastb_epi8(__mmask32 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast the low packed 8-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcastb_epi8

```
__m512i _mm512_broadcastb_epi8(__m128i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastb

Broadcast the low packed 8-bit integer from *a* to all elements of the return value.

_mm512_mask_broadcastb_epi8

```
__m512i _mm512_mask_broadcastb_epi8(__m512i src, __mmask64 k, __m128i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastb

Broadcast the low packed 8-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcastb_epi8

```
__m512i _mm512_maskz_broadcastb_epi8(__mmask64 k, __m128i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastb

Broadcast the low packed 8-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_broadcastd_epi32

```
__m128i _mm_mask_broadcastd_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast the low packed 32-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_broadcastd_epi32

```
__m128i _mm_maskz_broadcastd_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast the low packed 32-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_broadcastd_epi32

```
__m256i _mm256_mask_broadcastd_epi32(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast the low packed 32-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcastd_epi32

```
__m256i _mm256_maskz_broadcastd_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast the low packed 32-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_broadcastmb_epi64

```
__m128i _mm_broadcastmb_epi64(__mmask8 k)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpbroadcastmb2q

Broadcast the low 8-bits from input mask *k* to all 64-bit elements of the return value.

_mm256_broadcastmb_epi64

```
__m256i _mm256_broadcastmb_epi64(__mmask8 k)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpbroadcastmb2q

Broadcast the low 8-bits from input mask *k* to all 64-bit elements of the return value.

_mm_broadcastmw_epi32

```
__m128i _mm_broadcastmw_epi32(__mmask16 k)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpbroadcastmw2d

Broadcast the low 16-bits from input mask *k* to all 32-bit elements of the return value.

_mm256_broadcastmw_epi32

```
__m256i _mm256_broadcastmw_epi32(__mmask16 k)
```

CPUID Flags: AVX512CD, AVX512VL

Instruction(s): vpbroadcastmw2d

Broadcast the low 16-bits from input mask *k* to all 32-bit elements of the return value.

_mm_mask_broadcastq_epi64

```
__m128i _mm_mask_broadcastq_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast the low packed 64-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_broadcastq_epi64

```
__m128i _mm_maskz_broadcastq_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast the low packed 64-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_broadcastq_epi64

```
__m256i _mm256_mask_broadcastq_epi64(__m256i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast the low packed 64-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcastq_epi64

```
__m256i _mm256_maskz_broadcastq_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast the low packed 64-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_broadcastw_epi16

```
__m128i _mm_mask_broadcastw_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_broadcastw_epi16

```
__m128i _mm_maskz_broadcastw_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_broadcastw_epi16

```
__m256i _mm256_mask_broadcastw_epi16(__m256i src, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_broadcastw_epi16

```
__m256i _mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcastw_epi16

```
__m512i _mm512_broadcastw_epi16(__m128i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value.

_mm512_mask_broadcastw_epi16

```
__m512i _mm512_mask_broadcastw_epi16(__m512i src, __mmask32 k, __m128i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcastw_epi16

```
__m512i _mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_compress_epi32

```
__m128i _mm_mask_compress_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressd

Contiguously store the active 32-bit integers in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm_maskz_compress_epi32

```
__m128i _mm_maskz_compress_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressd

Contiguously store the active 32-bit integers in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm256_mask_compress_epi32

```
__m256i _mm256_mask_compress_epi32(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressd

Contiguously store the active 32-bit integers in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm256_maskz_compress_epi32

```
__m256i _mm256_maskz_compress_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressd

Contiguously store the active 32-bit integers in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm_mask_compress_epi64

```
__m128i _mm_mask_compress_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressq

Contiguously store the active 64-bit integers in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm_maskz_compress_epi64

```
__m128i _mm_maskz_compress_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressq

Contiguously store the active 64-bit integers in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm256_mask_compress_epi64

```
__m256i _mm256_mask_compress_epi64(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressq

Contiguously store the active 64-bit integers in *a* (those with their respective bit set in writemask *k*) to the return value, and pass through the remaining elements from *src*.

_mm256_maskz_compress_epi64

```
__m256i _mm256_maskz_compress_epi64(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressq

Contiguously store the active 64-bit integers in *a* (those with their respective bit set in zeromask *k*) to the return value, and set the remaining elements to zero.

_mm256_mask_permutexvar_epi32

```
__m256i _mm256_mask_permutexvar_epi32(__m256i src, __mmask8 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermd

Shuffle 32-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permutexvar_epi32

```
__m256i _mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermd

Shuffle 32-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutexvar_epi32

```
__m256i _mm256_permutexvar_epi32(__m256i idx, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermd

Shuffle 32-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results.

_mm_mask2_permutex2var_epi32

```
__m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm256_mask2_permutex2var_epi32

```
__m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm_maskz_permutex2var_epi32

```
__m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2d, vpermt2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_permutex2var_epi32

```
__m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2d, vpermt2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm256_maskz_permutex2var_epi32

```
__m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2d, vpermt2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex2var_epi32

```
__m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2d, vpermt2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm_mask2_permutex2var_epi64

```
__m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm256_mask2_permutex2var_epi64

```
__m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm_maskz_permutex2var_epi64

```
__m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2q, vpermt2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_permutex2var_epi64

```
__m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2q, vpermt2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm256_maskz_permutex2var_epi64

```
__m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2q, vpermt2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex2var_epi64

```
__m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermi2q, vpermt2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm_mask2_permutex2var_epi16

```
__m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermi2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm256_mask2_permutex2var_epi16

```
__m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermi2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm512_mask2_permutex2var_epi16

```
__m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpermi2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm_maskz_permutex2var_epi16

```
__m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermi2w, vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_permutex2var_epi16

```
__m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermi2w, vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm256_maskz_permutex2var_epi16

```
__m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermi2w, vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex2var_epi16

```
__m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermi2w, vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm512_maskz_permutex2var_epi16

```
__m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpermi2w, vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutex2var_epi16

```
__m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpermi2w, vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results.

_mm256_mask_permutex_epi64

```
__m256i _mm256_mask_permutex_epi64(__m256i src, __mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermq

Shuffle 64-bit integers in *a* across lanes lanes using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_permutexvar_epi64

```
__m256i _mm256_mask_permutexvar_epi64(__m256i src, __mmask8 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermq

Shuffle 64-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permutex_epi64

```
__m256i _mm256_maskz_permutex_epi64(__mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermq

Shuffle 64-bit integers in *a* across lanes using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_permutexvar_epi64

```
__m256i _mm256_maskz_permutexvar_epi64(__mmask8 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermq

Shuffle 64-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex_epi64

```
__m256i _mm256_permutex_epi64(__m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermq

Shuffle 64-bit integers in *a* across lanes using the control in *imm*, and return the results.

_mm256_permutexvar_epi64

```
__m256i _mm256_permutexvar_epi64(__m256i idx, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermq

Shuffle 64-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results.

_mm_mask_permutex2var_epi32

```
__m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask_permutex2var_epi32

```
__m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2d

Shuffle 32-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask_permutex2var_epi64

```
__m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask_permutex2var_epi64

```
__m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpermt2q

Shuffle 64-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask_permutex2var_epi16

```
__m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask_permutex2var_epi16

```
__m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask_permutex2var_epi16

```
__m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpermt2w

Shuffle 16-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the results using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask_permutexvar_epi16

```
__m128i _mm_mask_permutexvar_epi16(__m128i src, __mmask8 k, __m128i idx, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_permutexvar_epi16

```
__m128i _mm_maskz_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_permutexvar_epi16

```
__m128i _mm_permutexvar_epi16(__m128i idx, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results.

_mm256_mask_permutexvar_epi16

```
__m256i _mm256_mask_permutexvar_epi16(__m256i src, __mmask16 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permutexvar_epi16

```
__m256i _mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutexvar_epi16

```
__m256i _mm256_permutexvar_epi16(__m256i idx, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results.

_mm512_mask_permutexvar_epi16

```
__m512i _mm512_mask_permutexvar_epi16(__m512i src, __mmask32 k, __m512i idx, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutexvar_epi16

```
__m512i _mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutexvar_epi16

```
__m512i _mm512_permutexvar_epi16(__m512i idx, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpermw

Shuffle 16-bit integers in *a* across lanes using the corresponding index in *idx*, and return the results.

_mm_mask_expand_epi32

```
__m128i _mm_mask_expand_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load contiguous active 32-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expand_epi32

```
__m128i _mm_maskz_expand_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load contiguous active 32-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expand_epi32

```
__m256i _mm256_mask_expand_epi32(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load contiguous active 32-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expand_epi32

```
__m256i _mm256_maskz_expand_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandd

Load contiguous active 32-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_expand_epi64

```
__m128i _mm_mask_expand_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load contiguous active 64-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_expand_epi64

```
__m128i _mm_maskz_expand_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load contiguous active 64-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_expand_epi64

```
__m256i _mm256_mask_expand_epi64(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load contiguous active 64-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_expand_epi64

```
__m256i _mm256_maskz_expand_epi64(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpexpandq

Load contiguous active 64-bit integers from *a* (those with their respective bit set in mask *k*), and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_movm_epi8

```
__m128i _mm_movm_epi8(__mmask16 k)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovm2b

Set each packed 8-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm256_movm_epi8

```
__m256i _mm256_movm_epi8(__mmask32 k)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovm2b

Set each packed 8-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm512_movm_epi8

```
__m512i _mm512_movm_epi8(__mmask64 k)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovm2b

Set each packed 8-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm_movm_epi32

```
__m128i _mm_movm_epi32(__mmask8 k)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovm2d

Set each packed 32-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm256_movm_epi32

```
__m256i _mm256_movm_epi32(__mmask8 k)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovm2d

Set each packed 32-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm512_movm_epi32

```
__m512i _mm512_movm_epi32(__mmask16 k)
```

CPUID Flags: AVX512DQ

Instruction(s): vpmovm2d

Set each packed 32-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm_movm_epi64

```
__m128i _mm_movm_epi64(__mmask8 k)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovm2q

Set each packed 64-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm256_movm_epi64

```
__m256i _mm256_movm_epi64(__mmask8 k)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovm2q

Set each packed 64-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm512_movm_epi64

```
__m512i _mm512_movm_epi64(__mmask8 k)
```

CPUID Flags: AVX512DQ

Instruction(s): vpmovm2q

Set each packed 64-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm_movm_epi16

```
__m128i _mm_movm_epi16(__mmask8 k)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovm2w

Set each packed 16-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm256_movm_epi16

```
__m256i _mm256_movm_epi16(__mmask16 k)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovm2w

Set each packed 16-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm512_movm_epi16

```
__m512i _mm512_movm_epi16(__mmask32 k)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovm2w

Set each packed 16-bit integer in the return value to all ones or all zeros based on the value of the corresponding bit in *k*.

_mm512_sad_epu8

```
__m512i _mm512_sad_epu8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpsadbw

Compute the absolute differences of packed unsigned 8-bit integers in *a* and *b*, then horizontally sum each consecutive 8 differences to produce four unsigned 16-bit integers, and pack these unsigned 16-bit integers in the low 16 bits of 64-bit elements in the return value.

_mm_mask_shuffle_epi8

```
__m128i _mm_mask_shuffle_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufb

Shuffle packed 8-bit integers in *a* according to shuffle control mask in the corresponding 8-bit element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_shuffle_epi8

```
__m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufb

Shuffle packed 8-bit integers in *a* according to shuffle control mask in the corresponding 8-bit element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_shuffle_epi8

```
__m256i _mm256_mask_shuffle_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufb

Shuffle packed 8-bit integers in *a* according to shuffle control mask in the corresponding 8-bit element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_epi8

```
__m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufb

Shuffle packed 8-bit integers in *a* according to shuffle control mask in the corresponding 8-bit element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_shuffle_epi8

```
__m512i _mm512_mask_shuffle_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpshufb

Shuffle 8-bit integers in *a* within 128-bit lanes using the control in the corresponding 8-bit element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shuffle_epi8

```
__m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpshufb

Shuffle packed 8-bit integers in *a* according to shuffle control mask in the corresponding 8-bit element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shuffle_epi8

```
__m512i _mm512_shuffle_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpshufb

Shuffle packed 8-bit integers in *a* according to shuffle control mask in the corresponding 8-bit element of *b*, and return the results.

_mm_mask_shuffle_epi32

```
__m128i _mm_mask_shuffle_epi32(__m128i src, __mmask8 k, __m128i a, _MM_PERM_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpshufd

Shuffle 32-bit integers in *a* using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_shuffle_epi32

```
__m128i _mm_maskz_shuffle_epi32(__mmask8 k, __m128i a, _MM_PERM_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpshufd

Shuffle 32-bit integers in *a* using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_shuffle_epi32

```
__m256i _mm256_mask_shuffle_epi32(__m256i src, __mmask8 k, __m256i a, _MM_PERM_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpshufd

Shuffle 32-bit integers in *a* within 128-bit lanes using the control in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shuffle_epi32

```
__m256i _mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, _MM_PERM_ENUM imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpshufd

Shuffle 32-bit integers in *a* within 128-bit lanes using the control in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_shufflehi_epi16

```
__m128i _mm_mask_shufflehi_epi16(__m128i src, __mmask8 k, __m128i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufhw

Shuffle 16-bit integers in the high 64 bits of *a* using the control in *imm*. Store the results in the high 64 bits of the return value, with the low 64 bits being copied from from *a* to *dst*, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_shufflehi_epi16

```
__m128i _mm_maskz_shufflehi_epi16(__mmask8 k, __m128i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufhw

Shuffle 16-bit integers in the high 64 bits of *a* using the control in *imm*. Store the results in the high 64 bits of the return value, with the low 64 bits being copied from from *a* to *dst*, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_shufflehi_epi16

```
__m256i _mm256_mask_shufflehi_epi16(__m256i src, __mmask16 k, __m256i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufhw

Shuffle 16-bit integers in the high 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the high 64 bits of 128-bit lanes of the return value, with the low 64 bits of 128-bit lanes being copied from from *a* to *dst*, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shufflehi_epi16

```
__m256i _mm256_maskz_shufflehi_epi16(__mmask16 k, __m256i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshufhw

Shuffle 16-bit integers in the high 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the high 64 bits of 128-bit lanes of the return value, with the low 64 bits of 128-bit lanes being copied from from *a* to *dst*, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_shufflehi_epi16

```
__m512i _mm512_mask_shufflehi_epi16(__m512i src, __mmask32 k, __m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpshufhw

Shuffle 16-bit integers in the high 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the high 64 bits of 128-bit lanes of the return value, with the low 64 bits of 128-bit lanes being copied from from *a* to *dst*, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shufflehi_epi16

```
__m512i _mm512_maskz_shufflehi_epi16(__mmask32 k, __m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpshufhw

Shuffle 16-bit integers in the high 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the high 64 bits of 128-bit lanes of the return value, with the low 64 bits of 128-bit lanes being copied from from *a* to *dst*, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shufflehi_epi16

```
__m512i _mm512_shufflehi_epi16(__m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpshufhw

Shuffle 16-bit integers in the high 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the high 64 bits of 128-bit lanes of the return value, with the low 64 bits of 128-bit lanes being copied from from *a* to *dst*.

_mm_mask_shufflelo_epi16

```
__m128i _mm_mask_shufflelo_epi16(__m128i src, __mmask8 k, __m128i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshuflw

Shuffle 16-bit integers in the low 64 bits of *a* using the control in *imm*. Store the results in the low 64 bits of the return value, with the high 64 bits being copied from from *a* to *dst*, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_shufflelo_epi16

```
__m128i _mm_maskz_shufflelo_epi16(__mmask8 k, __m128i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshuflw

Shuffle 16-bit integers in the low 64 bits of *a* using the control in *imm*. Store the results in the low 64 bits of the return value, with the high 64 bits being copied from from *a* to *dst*, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_shufflelo_epi16

```
__m256i _mm256_mask_shufflelo_epi16(__m256i src, __mmask16 k, __m256i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshuflw

Shuffle 16-bit integers in the low 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the low 64 bits of 128-bit lanes of the return value, with the high 64 bits of 128-bit lanes being copied from from *a* to *dst*, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_shufflelo_epi16

```
__m256i _mm256_maskz_shufflelo_epi16(__mmask16 k, __m256i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpshuflw

Shuffle 16-bit integers in the low 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the low 64 bits of 128-bit lanes of the return value, with the high 64 bits of 128-bit lanes being copied from from *a* to *dst*, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_shufflelo_epi16

```
__m512i _mm512_mask_shufflelo_epi16(__m512i src, __mmask32 k, __m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpshuflw

Shuffle 16-bit integers in the low 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the low 64 bits of 128-bit lanes of the return value, with the high 64 bits of 128-bit lanes being copied from from *a* to *dst*, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shufflelo_epi16

```
__m512i _mm512_maskz_shufflelo_epi16(__mmask32 k, __m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpshuflw

Shuffle 16-bit integers in the low 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the low 64 bits of 128-bit lanes of the return value, with the high 64 bits of 128-bit lanes being copied from from *a* to *dst*, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shufflelo_epi16

```
__m512i _mm512_shufflelo_epi16(__m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpshuflw

Shuffle 16-bit integers in the low 64 bits of 128-bit lanes of *a* using the control in *imm*. Store the results in the low 64 bits of 128-bit lanes of the return value, with the high 64 bits of 128-bit lanes being copied from from *a* to *dst*.

_mm_mask_unpackhi_epi8

```
__m128i _mm_mask_unpackhi_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhbw

Unpack and interleave 8-bit integers from the high half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpackhi_epi8

```
__m128i _mm_maskz_unpackhi_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhbw

Unpack and interleave 8-bit integers from the high half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpackhi_epi8

```
__m256i _mm256_mask_unpackhi_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhbw

Unpack and interleave 8-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpackhi_epi8

```
__m256i _mm256_maskz_unpackhi_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhbw

Unpack and interleave 8-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_unpackhi_epi8

```
__m512i _mm512_mask_unpackhi_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpckhbw

Unpack and interleave 8-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpackhi_epi8

```
__m512i _mm512_maskz_unpackhi_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpckhbw

Unpack and interleave 8-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpackhi_epi8

```
__m512i _mm512_unpackhi_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpckhbw

Unpack and interleave 8-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results.

_mm_mask_unpackhi_epi32

```
__m128i _mm_mask_unpackhi_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhdq

Unpack and interleave 32-bit integers from the high half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpackhi_epi32

```
__m128i _mm_maskz_unpackhi_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhdq

Unpack and interleave 32-bit integers from the high half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpackhi_epi32

```
__m256i _mm256_mask_unpackhi_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhdq

Unpack and interleave 32-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpackhi_epi32

```
__m256i _mm256_maskz_unpackhi_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhdq

Unpack and interleave 32-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpackhi_epi64

```
__m128i _mm_mask_unpackhi_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhqdq

Unpack and interleave 64-bit integers from the high half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpackhi_epi64

```
__m128i _mm_maskz_unpackhi_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhqdq

Unpack and interleave 64-bit integers from the high half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpackhi_epi64

```
__m256i _mm256_mask_unpackhi_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhqdq

Unpack and interleave 64-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpackhi_epi64

```
__m256i _mm256_maskz_unpackhi_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckhqdq

Unpack and interleave 64-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpackhi_epi16

```
__m128i _mm_mask_unpackhi_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhwd

Unpack and interleave 16-bit integers from the high half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpackhi_epi16

```
__m128i _mm_maskz_unpackhi_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhwd

Unpack and interleave 16-bit integers from the high half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpackhi_epi16

```
__m256i _mm256_mask_unpackhi_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhwd

Unpack and interleave 16-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpackhi_epi16

```
__m256i _mm256_maskz_unpackhi_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpckhwd

Unpack and interleave 16-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_unpackhi_epi16

```
__m512i _mm512_mask_unpackhi_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpckhwd

Unpack and interleave 16-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpackhi_epi16

```
__m512i _mm512_maskz_unpackhi_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpckhwd

Unpack and interleave 16-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpackhi_epi16

```
__m512i _mm512_unpackhi_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpckhwd

Unpack and interleave 16-bit integers from the high half of each 128-bit lane in *a* and *b*, and return the results.

_mm_mask_unpacklo_epi8

```
__m128i _mm_mask_unpacklo_epi8(__m128i src, __mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklbw

Unpack and interleave 8-bit integers from the low half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpacklo_epi8

```
__m128i _mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklbw

Unpack and interleave 8-bit integers from the low half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpacklo_epi8

```
__m256i _mm256_mask_unpacklo_epi8(__m256i src, __mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklbw

Unpack and interleave 8-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpacklo_epi8

```
__m256i _mm256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklbw

Unpack and interleave 8-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_unpacklo_epi8

```
__m512i _mm512_mask_unpacklo_epi8(__m512i src, __mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpcklbw

Unpack and interleave 8-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpacklo_epi8

```
__m512i _mm512_maskz_unpacklo_epi8(__mmask64 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpcklbw

Unpack and interleave 8-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpacklo_epi8

```
__m512i _mm512_unpacklo_epi8(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpcklbw

Unpack and interleave 8-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results.

_mm_mask_unpacklo_epi32

```
__m128i _mm_mask_unpacklo_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckldq

Unpack and interleave 32-bit integers from the low half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpacklo_epi32

```
__m128i _mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckldq

Unpack and interleave 32-bit integers from the low half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpacklo_epi32

```
__m256i _mm256_mask_unpacklo_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckldq

Unpack and interleave 32-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpacklo_epi32

```
__m256i _mm256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpckldq

Unpack and interleave 32-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpacklo_epi64

```
__m128i _mm_mask_unpacklo_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpcklqdq

Unpack and interleave 64-bit integers from the low half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpacklo_epi64

```
__m128i _mm_maskz_unpacklo_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpcklqdq

Unpack and interleave 64-bit integers from the low half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpacklo_epi64

```
__m256i _mm256_mask_unpacklo_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpcklqdq

Unpack and interleave 64-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpacklo_epi64

```
__m256i _mm256_maskz_unpacklo_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpunpcklqdq

Unpack and interleave 64-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_unpacklo_epi16

```
__m128i _mm_mask_unpacklo_epi16(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklwd

Unpack and interleave 16-bit integers from the low half of *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_unpacklo_epi16

```
__m128i _mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklwd

Unpack and interleave 16-bit integers from the low half of *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_unpacklo_epi16

```
__m256i _mm256_mask_unpacklo_epi16(__m256i src, __mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklwd

Unpack and interleave 16-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_unpacklo_epi16

```
__m256i _mm256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpunpcklwd

Unpack and interleave 16-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_unpacklo_epi16

```
__m512i _mm512_mask_unpacklo_epi16(__m512i src, __mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpcklwd

Unpack and interleave 16-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpacklo_epi16

```
__m512i _mm512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpcklwd

Unpack and interleave 16-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpacklo_epi16

```
__m512i _mm512_unpacklo_epi16(__m512i a, __m512i b)
```

CPUID Flags: AVX512BW

Instruction(s): vpunpcklwd

Unpack and interleave 16-bit integers from the low half of each 128-bit lane in *a* and *b*, and return the results.

_mm512_kunpackd

```
__mmask64 _mm512_kunpackd(__mmask64 a, __mmask64 b)
```

CPUID Flags: AVX512BW

Instruction(s): kunpckdq

Unpack and interleave 32 bits from masks *a* and *b*, and return the 64-bit result.

_mm512_kunpackw

```
__mmask32 _mm512_kunpackw(__mmask32 a, __mmask32 b)
```

CPUID Flags: AVX512BW

Instruction(s): kunpckwd

Unpack and interleave 16 bits from masks *a* and *b*, and store the 32-bit result in *k*.

_mm_fpclass_pd_mask

```
__mmask8 _mm_fpclass_pd_mask(__m128d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclasspd

Test packed double-precision (64-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_fpclass_pd_mask

```
__mmask8 _mm_mask_fpclass_pd_mask(__mmask8 k1, __m128d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclasspd

Test packed double-precision (64-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_fpclass_pd_mask

```
__mmask8 _mm256_fpclass_pd_mask(__m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclasspd

Test packed double-precision (64-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_fpclass_pd_mask

```
__mmask8 _mm256_mask_fpclass_pd_mask(__mmask8 k1, __m256d a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclasspd

Test packed double-precision (64-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fpclass_pd_mask

```
__mmask8 _mm512_fpclass_pd_mask(__m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclasspd

Test packed double-precision (64-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm512_mask_fpclass_pd_mask

```
__mmask8 _mm512_mask_fpclass_pd_mask(__mmask8 k1, __m512d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclasspd

Test packed double-precision (64-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_fpclass_ps_mask

```
__mmask8 _mm_fpclass_ps_mask(__m128 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclassps

Test packed single-precision (32-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm_mask_fpclass_ps_mask

```
__mmask8 _mm_mask_fpclass_ps_mask(__mmask8 k1, __m128 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclassps

Test packed single-precision (32-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_fpclass_ps_mask

```
__mmask8 _mm256_fpclass_ps_mask(__m256 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclassps

Test packed single-precision (32-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm256_mask_fpclass_ps_mask

```
__mmask8 _mm256_mask_fpclass_ps_mask(__mmask8 k1, __m256 a, int imm)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vfpclassps

Test packed single-precision (32-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fpclass_ps_mask

```
__mmask16 _mm512_fpclass_ps_mask(__m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclassps

Test packed single-precision (32-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value.

_mm512_mask_fpclass_ps_mask

```
__mmask16 _mm512_mask_fpclass_ps_mask(__mmask16 k1, __m512 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclassps

Test packed single-precision (32-bit) floating-point elements in *a* for special categories specified by *imm*, and and put each result in the corresponding bit of the returned mask value using zeromask *k1* (elements are zeroed out when the corresponding mask bit is not set).

_mm_fpclass_sd_mask

```
__mmask8 _mm_fpclass_sd_mask(__m128d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclasssd

Test the lower double-precision (64-bit) floating-point element in *a* for special categories specified by *imm*, and and put the result in the returned mask value.

_mm_mask_fpclass_sd_mask

```
__mmask8 _mm_mask_fpclass_sd_mask(__mmask8 k1, __m128d a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclasssd

Test the lower double-precision (64-bit) floating-point element in *a* for special categories specified by *imm*, and and put the result in the returned mask value using zeromask *k1* (the element is zeroed out when mask bit 0 is not set).

_mm_fpclass_ss_mask

```
__mmask8 _mm_fpclass_ss_mask(__m128 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclassss

Test the lower single-precision (32-bit) floating-point element in *a* for special categories specified by *imm*, and store the result in mask vector "k.

_mm_mask_fpclass_ss_mask

```
__mmask8 _mm_mask_fpclass_ss_mask(__mmask8 k1, __m128 a, int imm)
```

CPUID Flags: AVX512DQ

Instruction(s): vfpclassss

Test the lower single-precision (32-bit) floating-point element in *a* for special categories specified by *imm*, and and put the result in the returned mask value using zeromask *k1* (the element is zeroed out when mask bit 0 is not set).

_mm_movepi8_mask

```
__mmask16 _mm_movepi8_mask(__m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovb2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 8-bit integer in *a*.

_mm256_movepi8_mask

```
__mmask32 _mm256_movepi8_mask(__m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovb2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 8-bit integer in *a*.

_mm512_movepi8_mask

```
__mmask64 _mm512_movepi8_mask(__m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovb2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 8-bit integer in *a*.

_mm_movepi32_mask

```
__mmask8 _mm_movepi32_mask(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovd2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 32-bit integer in *a*.

_mm256_movepi32_mask

```
__mmask8 _mm256_movepi32_mask(__m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovd2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 32-bit integer in *a*.

_mm512_movepi32_mask

```
__mmask16 _mm512_movepi32_mask(__m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vpmovd2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 32-bit integer in *a*.

_mm_movepi64_mask

```
__mmask8 _mm_movepi64_mask(__m128i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovq2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 64-bit integer in *a*.

_mm256_movepi64_mask

```
__mmask8 _mm256_movepi64_mask(__m256i a)
```

CPUID Flags: AVX512DQ, AVX512VL

Instruction(s): vpmovq2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 64-bit integer in *a*.

_mm512_movepi64_mask

```
__mmask8 _mm512_movepi64_mask(__m512i a)
```

CPUID Flags: AVX512DQ

Instruction(s): vpmovq2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 64-bit integer in *a*.

_mm_movepi16_mask

```
__mmask8 _mm_movepi16_mask(__m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovw2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 16-bit integer in *a*.

_mm256_movepi16_mask

```
__mmask16 _mm256_movepi16_mask(__m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpmovw2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 16-bit integer in *a*.

_mm512_movepi16_mask

```
__mmask32 _mm512_movepi16_mask(__m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vpmovw2m

Set each bit of the returned mask value based on the most significant bit of the corresponding packed 16-bit integer in *a*.

_mm_permutexvar_epi8

```
__m128i _mm_permutexvar_epi8(__m128i idx, __m128i a)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result.

_mm_mask_permutexvar_epi8

```
__m128i _mm_mask_permutexvar_epi8(__m128i src, __mmask16 k, __m128i idx, __m128i a)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_permutexvar_epi8

```
__m128i _mm_maskz_permutexvar_epi8(__mmask16 k, __m128i idx, __m128i a)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutexvar_epi8

```
__m256i _mm256_permutexvar_epi8(__m256i idx, __m256i a)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result.

_mm256_mask_permutexvar_epi8

```
__m256i _mm256_mask_permutexvar_epi8(__m256i src, __mmask32 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_permutexvar_epi8

```
__m256i _mm256_maskz_permutexvar_epi8(__mmask32 k, __m256i idx, __m256i a)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutexvar_epi8

```
__m512i _mm512_permutexvar_epi8(__m512i idx, __m512i a)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpermb

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding selector and index in *idx*, and return the result.

_mm512_mask_permutexvar_epi8

```
__m512i _mm512_mask_permutexvar_epi8(__m512i src, __mmask64 k, __m512i idx, __m512i a)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutexvar_epi8

```
__m512i _mm512_maskz_permutexvar_epi8(__mmask64 k, __m512i idx, __m512i a)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpermb

Shuffle 8-bit integers in *a* across lanes using the corresponding index in *idx*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_permutex2var_epi8

```
__m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermi2b

Shuffle 8-bit integers in *a* and *b* using the corresponding index in *idx*, and return the result.

_mm_mask_permutex2var_epi8

```
__m128i _mm_mask_permutex2var_epi8(__m128i a, __mmask16 k, __m128i idx, __m128i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermt2b

Shuffle 8-bit integers in *a* and *b* using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm_mask2_permutex2var_epi8

```
__m128i _mm_mask2_permutex2var_epi8(__m128i a, __m128i idx, __mmask16 k, __m128i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermi2b

Shuffle 8-bit integers in *a* and *b* using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm_maskz_permutex2var_epi8

```
__m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermi2b, vpermt2b

Shuffle 8-bit integers in *a* and *b* using the corresponding index in *idx*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_permutex2var_epi8

```
__m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermi2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result.

_mm256_mask_permutex2var_epi8

```
__m256i _mm256_mask_permutex2var_epi8(__m256i a, __mmask32 k, __m256i idx, __m256i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermt2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm256_mask2_permutex2var_epi8

```
__m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i idx, __mmask32 k, __m256i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermi2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm256_maskz_permutex2var_epi8

```
__m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b)
```

CPUID Flags: AVX512VBMI, AVX512VL

Instruction(s): vpermi2b, vpermt2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutex2var_epi8

```
__m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpermi2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result.

_mm512_mask_permutex2var_epi8

```
__m512i _mm512_mask_permutex2var_epi8(__m512i a, __mmask64 k, __m512i idx, __m512i b)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpermt2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask2_permutex2var_epi8

```
__m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i idx, __mmask64 k, __m512i b)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpermi2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm512_maskz_permutex2var_epi8

```
__m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b)
```

CPUID Flags: AVX512VBMI

Instruction(s): vpermi2b, vpermt2b

Shuffle 8-bit integers in *a* and *b* across lanes using the corresponding index in *idx*, and return the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Move Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element

_mm_mask_mov_pd

```
__m128d _mm_mask_mov_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Move packed double-precision (64-bit) floating-point elements from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mov_pd

```
__m128d _mm_maskz_mov_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Move packed double-precision (64-bit) floating-point elements from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mov_pd

```
__m256d _mm256_mask_mov_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Move packed double-precision (64-bit) floating-point elements from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mov_pd

```
__m256d _mm256_maskz_mov_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Move packed double-precision (64-bit) floating-point elements from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mov_ps

```
__m128 _mm_mask_mov_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Move packed single-precision (32-bit) floating-point elements from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mov_ps

```
__m128 _mm_maskz_mov_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Move packed single-precision (32-bit) floating-point elements from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mov_ps

```
__m256 _mm256_mask_mov_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Move packed single-precision (32-bit) floating-point elements from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mov_ps

```
__m256 _mm256_maskz_mov_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Move packed single-precision (32-bit) floating-point elements from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_movedup_pd

```
__m128d _mm_mask_movedup_pd(__m128d src, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovddup

Duplicate even-indexed double-precision (64-bit) floating-point elements from *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_movedup_pd

```
__m128d _mm_maskz_movedup_pd(__mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovddup

Duplicate even-indexed double-precision (64-bit) floating-point elements from *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_movedup_pd

```
__m256d _mm256_mask_movedup_pd(__m256d src, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovddup

Duplicate even-indexed double-precision (64-bit) floating-point elements from *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_movedup_pd

```
__m256d _mm256_maskz_movedup_pd(__mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovddup

Duplicate even-indexed double-precision (64-bit) floating-point elements from *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_movehdup_ps

```
__m128 _mm_mask_movehdup_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovshdup

Duplicate odd-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_movehdup_ps

```
__m128 _mm_maskz_movehdup_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovshdup

Duplicate odd-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_movehdup_ps

```
__m256 _mm256_mask_movehdup_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovshdup

Duplicate odd-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_movehdup_ps

```
__m256 _mm256_maskz_movehdup_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovshdup

Duplicate odd-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_moveldup_ps

```
__m128 _mm_mask_moveldup_ps(__m128 src, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovsldup

Duplicate even-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_moveldup_ps

```
__m128 _mm_maskz_moveldup_ps(__mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovsldup

Duplicate even-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_moveldup_ps

```
__m256 _mm256_mask_moveldup_ps(__m256 src, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovsldup

Duplicate even-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_moveldup_ps

```
__m256 _mm256_maskz_moveldup_ps(__mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovsldup

Duplicate even-indexed single-precision (32-bit) floating-point elements from *a*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mov_epi32

```
__m128i _mm_mask_mov_epi32(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Move packed 32-bit integers from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mov_epi32

```
__m128i _mm_maskz_mov_epi32(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Move packed 32-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mov_epi32

```
__m256i _mm256_mask_mov_epi32(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Move packed 32-bit integers from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mov_epi32

```
__m256i _mm256_maskz_mov_epi32(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Move packed 32-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mov_epi64

```
__m128i _mm_mask_mov_epi64(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Move packed 64-bit integers from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mov_epi64

```
__m128i _mm_maskz_mov_epi64(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Move packed 64-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mov_epi64

```
__m256i _mm256_mask_mov_epi64(__m256i src, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Move packed 64-bit integers from *a* to the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mov_epi64

```
__m256i _mm256_maskz_mov_epi64(__mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Move packed 64-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mov_epi16

```
__m128i _mm_mask_mov_epi16(__m128i src, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Move packed 16-bit integers from *a* into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mov_epi16

```
__m128i _mm_maskz_mov_epi16(__mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Move packed 16-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mov_epi16

```
__m256i _mm256_mask_mov_epi16(__m256i src, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Move packed 16-bit integers from *a* into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mov_epi16

```
__m256i _mm256_maskz_mov_epi16(__mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Move packed 16-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mov_epi16

```
__m512i _mm512_mask_mov_epi16(__m512i src, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu16

Move packed 16-bit integers from *a* into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mov_epi16

```
__m512i _mm512_maskz_mov_epi16(__mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu16

Move packed 16-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_mov_epi8

```
__m128i _mm_mask_mov_epi8(__m128i src, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Move packed 8-bit integers from *a* into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_mov_epi8

```
__m128i _mm_maskz_mov_epi8(__mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Move packed 8-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_mov_epi8

```
__m256i _mm256_mask_mov_epi8(__m256i src, __mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Move packed 8-bit integers from *a* into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_mov_epi8

```
__m256i _mm256_maskz_mov_epi8(__mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Move packed 8-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mov_epi8

```
__m512i _mm512_mask_mov_epi8(__m512i src, __mmask64 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu8

Move packed 8-bit integers from *a* into the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mov_epi8

```
__m512i _mm512_maskz_mov_epi8(__mmask64 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu8

Move packed 8-bit integers from *a* into the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Set Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element

_mm_mask_set1_epi8

```
__m128i _mm_mask_set1_epi8(__m128i src, __mmask16 k, char a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast 8-bit integer *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_set1_epi8

```
__m128i _mm_maskz_set1_epi8(__mmask16 k, char a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast 8-bit integer *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_set1_epi8

```
__m256i _mm256_mask_set1_epi8(__m256i src, __mmask32 k, char a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast 8-bit integer *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_set1_epi8

```
__m256i _mm256_maskz_set1_epi8(__mmask32 k, char a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastb

Broadcast 8-bit integer *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_set1_epi8

```
__m512i _mm512_mask_set1_epi8(__m512i src, __mmask64 k, char a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastb

Broadcast 8-bit integer *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_set1_epi8

```
__m512i _mm512_maskz_set1_epi8(__mmask64 k, char a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastb

Broadcast 8-bit integer *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_set1_epi32

```
__m128i _mm_mask_set1_epi32(__m128i src, __mmask8 k, int a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast 32-bit integer *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_set1_epi32

```
__m128i _mm_maskz_set1_epi32(__mmask8 k, int a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast 32-bit integer *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_set1_epi32

```
__m256i _mm256_mask_set1_epi32(__m256i src, __mmask8 k, int a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast 32-bit integer *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_set1_epi32

```
__m256i _mm256_maskz_set1_epi32(__mmask8 k, int a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastd

Broadcast 32-bit integer *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_set1_epi64

```
__m128i _mm_mask_set1_epi64(__m128i src, __mmask8 k, __int64 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast 64-bit integer *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_set1_epi64

```
__m128i _mm_maskz_set1_epi64(__mmask8 k, __int64 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast 64-bit integer *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_set1_epi64

```
__m256i _mm256_mask_set1_epi64(__m256i src, __mmask8 k, __int64 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast 64-bit integer *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_set1_epi64

```
__m256i _mm256_maskz_set1_epi64(__mmask8 k, __int64 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpbroadcastq

Broadcast 64-bit integer *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_set1_epi16

```
__m128i _mm_mask_set1_epi16(__m128i src, __mmask8 k, short a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_set1_epi16

```
__m128i _mm_maskz_set1_epi16(__mmask8 k, short a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_set1_epi16

```
__m256i _mm256_mask_set1_epi16(__m256i src, __mmask16 k, short a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_set1_epi16

```
__m256i _mm256_maskz_set1_epi16(__mmask16 k, short a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_set1_epi16

```
__m512i _mm512_mask_set1_epi16(__m512i src, __mmask32 k, short a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_set1_epi16

```
__m512i _mm512_maskz_set1_epi16(__mmask32 k, short a)
```

CPUID Flags: AVX512BW

Instruction(s): vpbroadcastw

Broadcast the low packed 16-bit integer from *a* to all elements of the return value using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Shift Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
src	source element to use based on writemask result
k	writemask used as a selector
a	first source vector element
b	second source vector element

_mm_mask_rol_epi32

```
__m128i _mm_mask_rol_epi32(__m128i src, __mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprold

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_rol_epi32

```
__m128i _mm_maskz_rol_epi32(__mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprold

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_rol_epi32

```
__m128i _mm_rol_epi32(__m128i a, int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprold

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results.

_mm256_mask_rol_epi32

```
__m256i _mm256_mask_rol_epi32(__m256i src, __mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprold

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_rol_epi32

```
__m256i _mm256_maskz_rol_epi32(__mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprold

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_rol_epi32

```
__m256i _mm256_rol_epi32(__m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprold

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results.

_mm_mask_rol_epi64

```
__m128i _mm_mask_rol_epi64(__m128i src, __mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_rol_epi64

```
__m128i _mm_maskz_rol_epi64(__mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_rol_epi64

```
__m128i _mm_rol_epi64(__m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results.

_mm256_mask_rol_epi64

```
__m256i _mm256_mask_rol_epi64(__m256i src, __mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_rol_epi64

```
__m256i _mm256_maskz_rol_epi64(__mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_rol_epi64

```
__m256i _mm256_rol_epi64(__m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in *imm*, and return the results.

_mm_mask_rolv_epi32

```
__m128i _mm_mask_rolv_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvd

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_rolv_epi32

```
__m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvd

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_rolv_epi32

```
__m128i _mm_rolv_epi32(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvd

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results.

_mm256_mask_rolv_epi32

```
__m256i _mm256_mask_rolv_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvd

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_rolv_epi32

```
__m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvd

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_rolv_epi32

```
__m256i _mm256_rolv_epi32(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvd

Rotate the bits in each packed 32-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results.

_mm_mask_rolv_epi64

```
__m128i _mm_mask_rolv_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_rolv_epi64

```
__m128i _mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_rolv_epi64

```
__m128i _mm_rolv_epi64(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results.

_mm256_mask_rolv_epi64

```
__m256i _mm256_mask_rolv_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_rolv_epi64

```
__m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_rolv_epi64

```
__m256i _mm256_rolv_epi64(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprolvq

Rotate the bits in each packed 64-bit integer in *a* to the left by the number of bits specified in the corresponding element of *b*, and return the results.

_mm_mask_ror_epi32

```
__m128i _mm_mask_ror_epi32(__m128i src, __mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprord

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_ror_epi32

```
__m128i _mm_maskz_ror_epi32(__mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprord

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_ror_epi32

```
__m128i _mm_ror_epi32(__m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprord

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results.

_mm256_mask_ror_epi32

```
__m256i _mm256_mask_ror_epi32(__m256i src, __mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprord

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_ror_epi32

```
__m256i _mm256_maskz_ror_epi32(__mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprord

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_ror_epi32

```
__m256i _mm256_ror_epi32(__m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprord

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results.

_mm_mask_ror_epi64

```
__m128i _mm_mask_ror_epi64(__m128i src, __mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_ror_epi64

```
__m128i _mm_maskz_ror_epi64(__mmask8 k, __m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_ror_epi64

```
__m128i _mm_ror_epi64(__m128i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results.

_mm256_mask_ror_epi64

```
__m256i _mm256_mask_ror_epi64(__m256i src, __mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_ror_epi64

```
__m256i _mm256_maskz_ror_epi64(__mmask8 k, __m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_ror_epi64

```
__m256i _mm256_ror_epi64(__m256i a, const int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in *imm*, and return the results.

_mm_mask_rorv_epi32

```
__m128i _mm_mask_rorv_epi32(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvd

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_rorv_epi32

```
__m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvd

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_rorv_epi32

```
__m128i _mm_rorv_epi32(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvd

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results.

_mm256_mask_rorv_epi32

```
__m256i _mm256_mask_rorv_epi32(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvd

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_rorv_epi32

```
__m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvd

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_rorv_epi32

```
__m256i _mm256_rorv_epi32(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvd

Rotate the bits in each packed 32-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results.

_mm_mask_rorv_epi64

```
__m128i _mm_mask_rorv_epi64(__m128i src, __mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_rorv_epi64

```
__m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_rorv_epi64

```
__m128i _mm_rorv_epi64(__m128i a, __m128i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results.

_mm256_mask_rorv_epi64

```
__m256i _mm256_mask_rorv_epi64(__m256i src, __mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_rorv_epi64

```
__m256i _mm256_maskz_rorv_epi64(__mmask8 k, __m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_rorv_epi64

```
__m256i _mm256_rorv_epi64(__m256i a, __m256i b)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vprorvq

Rotate the bits in each packed 64-bit integer in *a* to the right by the number of bits specified in the corresponding element of *b*, and return the results.

_mm_mask_sll_epi32

```
__m128i _mm_mask_sll_epi32(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_slli_epi32

```
__m128i _mm_mask_slli_epi32(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sll_epi32

```
__m128i _mm_maskz_sll_epi32(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_slli_epi32

```
__m128i _mm_maskz_slli_epi32(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sll_epi32

```
__m256i _mm256_mask_sll_epi32(__m256i src, __mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_slli_epi32

```
__m256i _mm256_mask_slli_epi32(__m256i src, __mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sll_epi32

```
__m256i _mm256_maskz_sll_epi32(__mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_slli_epi32

```
__m256i _mm256_maskz_slli_epi32(__mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpslld

Shift packed 32-bit integers in *a* left by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_bslli_epi128

```
__m512i _mm512_bslli_epi128(__m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpslldq

Shift 128-bit lanes in *a* left by *imm* bytes while shifting in zeros, and return the results.

_mm_mask_sll_epi64

```
__m128i _mm_mask_sll_epi64(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_slli_epi64

```
__m128i _mm_mask_slli_epi64(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sll_epi64

```
__m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_slli_epi64

```
__m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sll_epi64

```
__m256i _mm256_mask_sll_epi64(__m256i src, __mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_slli_epi64

```
__m256i _mm256_mask_slli_epi64(__m256i src, __mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sll_epi64

```
__m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_slli_epi64

```
__m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllq

Shift packed 64-bit integers in *a* left by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sllv_epi32

```
__m128i _mm_mask_sllv_epi32(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvd

Shift packed 32-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sllv_epi32

```
__m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvd

Shift packed 32-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sllv_epi32

```
__m256i _mm256_mask_sllv_epi32(__m256i src, __mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvd

Shift packed 32-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sllv_epi32

```
__m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvd

Shift packed 32-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sllv_epi64

```
__m128i _mm_mask_sllv_epi64(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvq

Shift packed 64-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sllv_epi64

```
__m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvq

Shift packed 64-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sllv_epi64

```
__m256i _mm256_mask_sllv_epi64(__m256i src, __mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvq

Shift packed 64-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sllv_epi64

```
__m256i _mm256_maskz_sllv_epi64(__mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsllvq

Shift packed 64-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sllv_epi16

```
__m128i _mm_mask_sllv_epi16(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sllv_epi16

```
__m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_sllv_epi16

```
__m128i _mm_sllv_epi16(__m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results.

_mm256_mask_sllv_epi16

```
__m256i _mm256_mask_sllv_epi16(__m256i src, __mmask16 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sllv_epi16

```
__m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_sllv_epi16

```
__m256i _mm256_sllv_epi16(__m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results.

_mm512_mask_sllv_epi16

```
__m512i _mm512_mask_sllv_epi16(__m512i src, __mmask32 k, __m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sllv_epi16

```
__m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sllv_epi16

```
__m512i _mm512_sllv_epi16(__m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllvw

Shift packed 16-bit integers in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results.

_mm_mask_sll_epi16

```
__m128i _mm_mask_sll_epi16(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_slli_epi16

```
__m128i _mm_mask_slli_epi16(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sll_epi16

```
__m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_slli_epi16

```
__m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sll_epi16

```
__m256i _mm256_mask_sll_epi16(__m256i src, __mmask16 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_slli_epi16

```
__m256i _mm256_mask_slli_epi16(__m256i src, __mmask16 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sll_epi16

```
__m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_slli_epi16

```
__m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_sll_epi16

```
__m512i _mm512_mask_sll_epi16(__m512i src, __mmask32 k, __m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_slli_epi16

```
__m512i _mm512_mask_slli_epi16(__m512i src, __mmask32 k, __m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sll_epi16

```
__m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_slli_epi16

```
__m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sll_epi16

```
__m512i _mm512_sll_epi16(__m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *count* while shifting in zeros, and return the results.

_mm512_slli_epi16

```
__m512i _mm512_slli_epi16(__m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsllw

Shift packed 16-bit integers in *a* left by *imm* while shifting in zeros, and return the results.

_mm_mask_sra_epi32

```
__m128i _mm_mask_sra_epi32(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_srai_epi32

```
__m128i _mm_mask_srai_epi32(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sra_epi32

```
__m128i _mm_maskz_sra_epi32(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_srai_epi32

```
__m128i _mm_maskz_srai_epi32(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sra_epi32

```
__m256i _mm256_mask_sra_epi32(__m256i src, __mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_srai_epi32

```
__m256i _mm256_mask_srai_epi32(__m256i src, __mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sra_epi32

```
__m256i _mm256_maskz_sra_epi32(__mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_srai_epi32

```
__m256i _mm256_maskz_srai_epi32(__mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrad

Shift packed 32-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sra_epi64

```
__m128i _mm_mask_sra_epi64(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_srai_epi64

```
__m128i _mm_mask_srai_epi64(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sra_epi64

```
__m128i _mm_maskz_sra_epi64(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_srai_epi64

```
__m128i _mm_maskz_srai_epi64(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_sra_epi64

```
__m128i _mm_sra_epi64(__m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *count* while shifting in sign bits, and return the results.

_mm_srai_epi64

```
__m128i _mm_srai_epi64(__m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *imm* while shifting in sign bits, and return the results.

_mm256_mask_sra_epi64

```
__m256i _mm256_mask_sra_epi64(__m256i src, __mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_srai_epi64

```
__m256i _mm256_mask_srai_epi64(__m256i src, __mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sra_epi64

```
__m256i _mm256_maskz_sra_epi64(__mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_srai_epi64

```
__m256i _mm256_maskz_srai_epi64(__mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_sra_epi64

```
__m256i _mm256_sra_epi64(__m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *count* while shifting in sign bits, and return the results.

_mm256_srai_epi64

```
__m256i _mm256_srai_epi64(__m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsraq

Shift packed 64-bit integers in *a* right by *imm* while shifting in sign bits, and return the results.

_mm_mask_srav_epi32

```
__m128i _mm_mask_srav_epi32(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srav_epi32

```
__m128i _mm_maskz_srav_epi32(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_srav_epi32

```
__m256i _mm256_mask_srav_epi32(__m256i src, __mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srav_epi32

```
__m256i _mm256_maskz_srav_epi32(__mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_srav_epi64

```
__m128i _mm_mask_srav_epi64(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srav_epi64

```
__m128i _mm_maskz_srav_epi64(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_srav_epi64

```
__m128i _mm_srav_epi64(__m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results.

_mm256_mask_srav_epi64

```
__m256i _mm256_mask_srav_epi64(__m256i src, __mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srav_epi64

```
__m256i _mm256_maskz_srav_epi64(__mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_srav_epi64

```
__m256i _mm256_srav_epi64(__m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsravq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results.

_mm_mask_srav_epi16

```
__m128i _mm_mask_srav_epi16(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srav_epi16

```
__m128i _mm_maskz_srav_epi16(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_srav_epi16

```
__m128i _mm_srav_epi16(__m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results.

_mm256_mask_srav_epi16

```
__m256i _mm256_mask_srav_epi16(__m256i src, __mmask16 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srav_epi16

```
__m256i _mm256_maskz_srav_epi16(__mmask16 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_srav_epi16

```
__m256i _mm256_srav_epi16(__m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results.

_mm512_mask_srav_epi16

```
__m512i _mm512_mask_srav_epi16(__m512i src, __mmask32 k, __m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srav_epi16

```
__m512i _mm512_maskz_srav_epi16(__mmask32 k, __m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srav_epi16

```
__m512i _mm512_srav_epi16(__m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsravw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and return the results.

_mm_mask_sra_epi16

```
__m128i _mm_mask_sra_epi16(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_srai_epi16

```
__m128i _mm_mask_srai_epi16(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_sra_epi16

```
__m128i _mm_maskz_sra_epi16(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_srai_epi16

```
__m128i _mm_maskz_srai_epi16(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_sra_epi16

```
__m256i _mm256_mask_sra_epi16(__m256i src, __mmask16 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_srai_epi16

```
__m256i _mm256_mask_srai_epi16(__m256i src, __mmask16 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_sra_epi16

```
__m256i _mm256_maskz_sra_epi16(__mmask16 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_srai_epi16

```
__m256i _mm256_maskz_srai_epi16(__mmask16 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_sra_epi16

```
__m512i _mm512_mask_sra_epi16(__m512i src, __mmask32 k, __m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *count* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_srai_epi16

```
__m512i _mm512_mask_srai_epi16(__m512i src, __mmask32 k, __m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sra_epi16

```
__m512i _mm512_maskz_sra_epi16(__mmask32 k, __m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *count* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_srai_epi16

```
__m512i _mm512_maskz_srai_epi16(__mmask32 k, __m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *imm* while shifting in sign bits, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sra_epi16

```
__m512i _mm512_sra_epi16(__m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *count* while shifting in sign bits, and return the results.

_mm512_srai_epi16

```
__m512i _mm512_srai_epi16(__m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsraw

Shift packed 16-bit integers in *a* right by *imm* while shifting in sign bits, and return the results.

_mm_mask_srl_epi32

```
__m128i _mm_mask_srl_epi32(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_srli_epi32

```
__m128i _mm_mask_srli_epi32(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srl_epi32

```
__m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_srli_epi32

```
__m128i _mm_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_srl_epi32

```
__m256i _mm256_mask_srl_epi32(__m256i src, __mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_srli_epi32

```
__m256i _mm256_mask_srli_epi32(__m256i src, __mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srl_epi32

```
__m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_srli_epi32

```
__m256i _mm256_maskz_srli_epi32(__mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrld

Shift packed 32-bit integers in *a* right by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_bsrli_epi128

```
__m512i _mm512_bsrli_epi128(__m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrldq

Shift 128-bit lanes in *a* right by *imm* bytes while shifting in zeros, and return the results.

_mm_mask_srl_epi64

```
__m128i _mm_mask_srl_epi64(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_srli_epi64

```
__m128i _mm_mask_srli_epi64(__m128i src, __mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srl_epi64

```
__m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_srli_epi64

```
__m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_srl_epi64

```
__m256i _mm256_mask_srl_epi64(__m256i src, __mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_srli_epi64

```
__m256i _mm256_mask_srli_epi64(__m256i src, __mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srl_epi64

```
__m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_srli_epi64

```
__m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlq

Shift packed 64-bit integers in *a* right by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_srlv_epi32

```
__m128i _mm_mask_srlv_epi32(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srlv_epi32

```
__m128i _mm_maskz_srlv_epi32(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_srlv_epi32

```
__m256i _mm256_mask_srlv_epi32(__m256i src, __mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srlv_epi32

```
__m256i _mm256_maskz_srlv_epi32(__mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvd

Shift packed 32-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_srlv_epi64

```
__m128i _mm_mask_srlv_epi64(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srlv_epi64

```
__m128i _mm_maskz_srlv_epi64(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_srlv_epi64

```
__m256i _mm256_mask_srlv_epi64(__m256i src, __mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srlv_epi64

```
__m256i _mm256_maskz_srlv_epi64(__mmask8 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpsrlvq

Shift packed 64-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_srlv_epi16

```
__m128i _mm_mask_srlv_epi16(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srlv_epi16

```
__m128i _mm_maskz_srlv_epi16(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_srlv_epi16

```
__m128i _mm_srlv_epi16(__m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results.

_mm256_mask_srlv_epi16

```
__m256i _mm256_mask_srlv_epi16(__m256i src, __mmask16 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srlv_epi16

```
__m256i _mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_srlv_epi16

```
__m256i _mm256_srlv_epi16(__m256i a, __m256i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results.

_mm512_mask_srlv_epi16

```
__m512i _mm512_mask_srlv_epi16(__m512i src, __mmask32 k, __m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srlv_epi16

```
__m512i _mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srlv_epi16

```
__m512i _mm512_srlv_epi16(__m512i a, __m512i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlvw

Shift packed 16-bit integers in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and return the results.

_mm_mask_srl_epi16

```
__m128i _mm_mask_srl_epi16(__m128i src, __mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_mask_srli_epi16

```
__m128i _mm_mask_srli_epi16(__m128i src, __mmask8 k, __m128i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm_maskz_srl_epi16

```
__m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_maskz_srli_epi16

```
__m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_mask_srl_epi16

```
__m256i _mm256_mask_srl_epi16(__m256i src, __mmask16 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_mask_srli_epi16

```
__m256i _mm256_mask_srli_epi16(__m256i src, __mmask16 k, __m256i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm256_maskz_srl_epi16

```
__m256i _mm256_maskz_srl_epi16(__mmask16 k, __m256i a, __m128i count)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm256_maskz_srli_epi16

```
__m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, int imm)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_srl_epi16

```
__m512i _mm512_mask_srl_epi16(__m512i src, __mmask32 k, __m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *count* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_srli_epi16

```
__m512i _mm512_mask_srli_epi16(__m512i src, __mmask32 k, __m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *imm* while shifting in zeros, and return the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srl_epi16

```
__m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *count* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_srli_epi16

```
__m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *imm* while shifting in zeros, and return the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srl_epi16

```
__m512i _mm512_srl_epi16(__m512i a, __m128i count)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *count* while shifting in zeros, and return the results.

_mm512_srli_epi16

```
__m512i _mm512_srli_epi16(__m512i a, unsigned int imm)
```

CPUID Flags: AVX512BW

Instruction(s): vpsrlw

Shift packed 16-bit integers in *a* right by *imm* while shifting in zeros, and return the results.

Intrinsics for Store Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

variable	definition
base_addr	pointer to base address in memory to begin load or store operation
mem_addr	pointer to base address in memory
k	writemask used as a selector
a	first source vector element

_mm_mask_compressstoreu_pd

```
void _mm_mask_compressstoreu_pd(void* base_addr, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompresspd

Contiguously store the active double-precision (64-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_compressstoreu_pd

```
void _mm256_mask_compressstoreu_pd(void* base_addr, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompresspd

Contiguously store the active double-precision (64-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_compressstoreu_ps

```
void _mm_mask_compressstoreu_ps(void* base_addr, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompressps

Contiguously store the active single-precision (32-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_compressstoreu_ps

```
void _mm256_mask_compressstoreu_ps(void* base_addr, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vcompressps

Contiguously store the active single-precision (32-bit) floating-point elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_store_pd

```
void _mm_mask_store_pd(void* mem_addr, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Store packed double-precision (64-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_store_pd

```
void _mm256_mask_store_pd(void* mem_addr, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovapd

Store packed double-precision (64-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_store_ps

```
void _mm_mask_store_ps(void* mem_addr, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Store packed single-precision (32-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_store_ps

```
void _mm256_mask_store_ps(void* mem_addr, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovaps

Store packed single-precision (32-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_storeu_pd

```
void _mm_mask_storeu_pd(void* mem_addr, __mmask8 k, __m128d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovupd

Store packed double-precision (64-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_storeu_pd

```
void _mm256_mask_storeu_pd(void* mem_addr, __mmask8 k, __m256d a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovupd

Store packed double-precision (64-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_storeu_ps

```
void _mm_mask_storeu_ps(void* mem_addr, __mmask8 k, __m128 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovups

Store packed single-precision (32-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_storeu_ps

```
void _mm256_mask_storeu_ps(void* mem_addr, __mmask8 k, __m256 a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovups

Store packed single-precision (32-bit) floating-point elements from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm_i32scatter_pd

```
void _mm_i32scatter_pd(void* base_addr, __m128i vindex, __m128d a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm_mask_i32scatter_pd

```
void _mm_mask_i32scatter_pd(void* base_addr, __mmask8 k, __m128i vindex, __m128d a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i32scatter_pd

```
void _mm256_i32scatter_pd(void* base_addr, __m128i vindex, __m256d a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i32scatter_pd

```
void _mm256_mask_i32scatter_pd(void* base_addr, __mmask8 k, __m128i vindex, __m256d a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_i32scatter_ps

```
void _mm_i32scatter_ps(void* base_addr, __m128i vindex, __m128 a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm_mask_i32scatter_ps

```
void _mm_mask_i32scatter_ps(void* base_addr, __mmask8 k, __m128i vindex, __m128 a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i32scatter_ps

```
void _mm256_i32scatter_ps(void* base_addr, __m256i vindex, __m256 a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i32scatter_ps

```
void _mm256_mask_i32scatter_ps(void* base_addr, __mmask8 k, __m256i vindex, __m256 a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterdps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_i64scatter_pd

```
void _mm_i64scatter_pd(void* base_addr, __m128i vindex, __m128d a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm_mask_i64scatter_pd

```
void _mm_mask_i64scatter_pd(void* base_addr, __mmask8 k, __m128i vindex, __m128d a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i64scatter_pd

```
void _mm256_i64scatter_pd(void* base_addr, __m256i vindex, __m256d a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i64scatter_pd

```
void _mm256_mask_i64scatter_pd(void* base_addr, __mmask8 k, __m256i vindex, __m256d a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqpd

Scatter double-precision (64-bit) floating-point elements from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_i64scatter_ps

```
void _mm_i64scatter_ps(void* base_addr, __m128i vindex, __m128 a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mask_i64scatter_ps

```
void _mm_mask_i64scatter_ps(void* base_addr, __mmask8 k, __m128i vindex, __m128 a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i64scatter_ps

```
void _mm256_i64scatter_ps(void* base_addr, __m256i vindex, __m128 a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i64scatter_ps

```
void _mm256_mask_i64scatter_ps(void* base_addr, __mmask8 k, __m256i vindex, __m128 a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vscatterqps

Scatter single-precision (32-bit) floating-point elements from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_mask_store_epi32

```
void _mm_mask_store_epi32(void* mem_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Store packed 32-bit integers from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_store_epi32

```
void _mm256_mask_store_epi32(void* mem_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa32

Store packed 32-bit integers from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_store_epi64

```
void _mm_mask_store_epi64(void* mem_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Store packed 64-bit integers from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 16-byte boundary or a general-protection exception may be generated.

_mm256_mask_store_epi64

```
void _mm256_mask_store_epi64(void* mem_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqa64

Store packed 64-bit integers from *a* into memory using writemask *k*. *mem_addr* must be aligned on a 32-byte boundary or a general-protection exception may be generated.

_mm_mask_storeu_epi16

```
void _mm_mask_storeu_epi16(void* mem_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Store packed 16-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_storeu_epi16

```
void _mm256_mask_storeu_epi16(void* mem_addr, __mmask16 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu16

Store packed 16-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm512_mask_storeu_epi16

```
void _mm512_mask_storeu_epi16(void* mem_addr, __mmask32 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu16

Store packed 16-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_storeu_epi32

```
void _mm_mask_storeu_epi32(void* mem_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu32

Store packed 32-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_storeu_epi32

```
void _mm256_mask_storeu_epi32(void* mem_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu32

Store packed 32-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_storeu_epi64

```
void _mm_mask_storeu_epi64(void* mem_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu64

Store packed 64-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_storeu_epi64

```
void _mm256_mask_storeu_epi64(void* mem_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vmovdqu64

Store packed 64-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_storeu_epi8

```
void _mm_mask_storeu_epi8(void* mem_addr, __mmask16 k, __m128i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Store packed 8-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm256_mask_storeu_epi8

```
void _mm256_mask_storeu_epi8(void* mem_addr, __mmask32 k, __m256i a)
```

CPUID Flags: AVX512BW, AVX512VL

Instruction(s): vmovdqu8

Store packed 8-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm512_mask_storeu_epi8

```
void _mm512_mask_storeu_epi8(void* mem_addr, __mmask64 k, __m512i a)
```

CPUID Flags: AVX512BW

Instruction(s): vmovdqu8

Store packed 8-bit integers from *a* into memory using writemask *k*. *mem_addr* does not need to be aligned on any particular boundary.

_mm_mask_compressstoreu_epi32

```
void _mm_mask_compressstoreu_epi32(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressd

Contiguously store the active 32-bit integers in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_compressstoreu_epi32

```
void _mm256_mask_compressstoreu_epi32(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressd

Contiguously store the active 32-bit integers in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_mask_compressstoreu_epi64

```
void _mm_mask_compressstoreu_epi64(void* base_addr, __mmask8 k, __m128i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressq

Contiguously store the active 64-bit integers in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm256_mask_compressstoreu_epi64

```
void _mm256_mask_compressstoreu_epi64(void* base_addr, __mmask8 k, __m256i a)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpcompressq

Contiguously store the active 64-bit integers in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm_i32scatter_epi32

```
void _mm_i32scatter_epi32(void* base_addr, __m128i vindex, __m128i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdd

Scatter 32-bit integers from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm_mask_i32scatter_epi32

```
void _mm_mask_i32scatter_epi32(void* base_addr, __mmask8 k, __m128i vindex, __m128i a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdd

Scatter 32-bit integers from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i32scatter_epi32

```
void _mm256_i32scatter_epi32(void* base_addr, __m256i vindex, __m256i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdd

Scatter 32-bit integers from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i32scatter_epi32

```
void _mm256_mask_i32scatter_epi32(void* base_addr, __mmask8 k, __m256i vindex, __m256i a, const
int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdd

Scatter 32-bit integers from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_i32scatter_epi64

```
void _mm_i32scatter_epi64(void* base_addr, __m128i vindex, __m128i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdq

Scatter 64-bit integers from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm_mask_i32scatter_epi64

```
void _mm_mask_i32scatter_epi64(void* base_addr, __mmask8 k, __m128i vindex, __m128i a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdq

Scatter 64-bit integers from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i32scatter_epi64

```
void _mm256_i32scatter_epi64(void* base_addr, __m128i vindex, __m256i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdq

Scatter 64-bit integers from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i32scatter_epi64

```
void _mm256_mask_i32scatter_epi64(void* base_addr, __mmask8 k, __m128i vindex, __m256i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterdq

Scatter 64-bit integers from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_i64scatter_epi32

```
void _mm_i64scatter_epi32(void* base_addr, __m128i vindex, __m128i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqd

Scatter 32-bit integers from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm_mask_i64scatter_epi32

```
void _mm_mask_i64scatter_epi32(void* base_addr, __mmask8 k, __m128i vindex, __m128i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqd

Scatter 32-bit integers from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i64scatter_epi32

```
void _mm256_i64scatter_epi32(void* base_addr, __m256i vindex, __m128i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqd

Scatter 32-bit integers from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i64scatter_epi32

```
void _mm256_mask_i64scatter_epi32(void* base_addr, __mmask8 k, __m256i vindex, __m128i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqd

Scatter 32-bit integers from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm_i64scatter_epi64

```
void _mm_i64scatter_epi64(void* base_addr, __m128i vindex, __m128i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqq

Scatter 64-bit integers from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm_mask_i64scatter_epi64

```
void _mm_mask_i64scatter_epi64(void* base_addr, __mmask8 k, __m128i vindex, __m128i a, const int
scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqq

Scatter 64-bit integers from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

_mm256_i64scatter_epi64

```
void _mm256_i64scatter_epi64(void* base_addr, __m256i vindex, __m256i a, const int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqq

Scatter 64-bit integers from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). *scale* should be 1, 2, 4 or 8.

_mm256_mask_i64scatter_epi64

```
void _mm256_mask_i64scatter_epi64(void* base_addr, __mmask8 k, __m256i vindex, __m256i a, const
int scale)
```

CPUID Flags: AVX512F, AVX512VL

Instruction(s): vpscatterqq

Scatter 64-bit integers from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k* (elements are not stored when the corresponding mask bit is not set). *scale* should be 1, 2, 4 or 8.

Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Instructions

Functional Overview

Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Instructions extend Intel® Advanced Vector Extensions (Intel® AVX) and Intel® Advanced Vector Extensions 2 (Intel® AVX2) by promoting most of the 256-bit SIMD instructions with 512-bit numeric processing capabilities.

The Intel® AVX-512 instructions follow the same programming model as the Intel® AVX2 instructions, providing enhanced functionality for broadcast, embedded masking to enable predication, embedded floating point rounding control, embedded floating-point fault suppression, scatter instructions, high speed math instructions, and compact representation of large displacement values. Unlike Intel® SSE and Intel® AVX, which cannot be mixed without performance penalties, the mixing of Intel® AVX and Intel® AVX-512 instructions is supported without penalty.

Intel® AVX-512 intrinsics are supported on IA-32 and Intel® 64 architectures built from 32nm process technology. They map directly to the new Intel® AVX-512 instructions and other enhanced 128-bit and 256-bit SIMD instructions.

Intel® AVX-512 Registers

512-bit Register state is managed by the operating system using `XSAVE` / `XRSTOR` / `XSAVEOPT` instructions, introduced in 45nm Intel® 64 processors (see Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, and Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

- Support for sixteen new 512-bit SIMD registers in 64-bit mode (for a total of 32 SIMD registers, representing 2K of register space, `ZMM0` through `ZMM31`).
- Support for eight new opmask registers (`k0` through `k7`) used for conditional execution and efficient merging of destination operands.

Intel® AVX registers `YMM0-YMM15` map into Intel® AVX-512 registers `ZMM0-ZMM15`, very much like Intel® SSE registers map into Intel® AVX registers. In processors with Intel® AVX-512 support, Intel® AVX and Intel® AVX2 instructions operate on the lower 128- or 256-bits of the first sixteen `ZMM` registers.

Prefix Instruction Encoding Support for Intel® AVX-512

A new encoding prefix (referred to as EVEX) to support additional vector length encoding up to 512 bits. The EVEX prefix builds upon the foundations of VEX prefix, to provide compact, efficient encoding for functionality available to VEX encoding while enhancing vector capabilities.

The Intel® AVX-512 intrinsic functions use three C data types as operands, representing the new registers used as operands to the intrinsic functions. These are `__m512`, `__m512d`, and `__m512i` data types. The `__m512` data type is used to represent the contents of the extended SSE register, the `ZMM` register, used by the Intel® AVX-512 intrinsics. The `__m512` data type can hold sixteen 32-bit floating-point values. The `__m512d` data type can hold eight 64-bit double precision floating-point values. The `__m512i` data type can hold sixty-four 8-bit, thirty-two 16-bit, sixteen 32-bit, or eight 64-bit integer values.

The compiler aligns the `__m512`, `__m512d`, and `__m512i` local and global data to 64-byte boundaries on the stack. To align `integer`, `float`, or `double` arrays, use the `__declspec(align)` statement.

Data Types for Intel® AVX-512 Intrinsics

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intel® AVX-512 intrinsics have vector variants that use `__m128`, `__m128i`, `__m128d`, `__m256`, `__m256i`, `__m256d`, `__m512`, `__m512i`, and `__m512d` data types.

Naming and Usage Syntax

Most Intel® AVX-512 intrinsic names use the following notational convention:

```
_mm512[_<maskprefix>]_<intrin_op>_<suffix>
```

The following table explains each item in the syntax.

`_mm512`	Prefix representing the size of the largest vector in the operation considering any of the parameters or the result.
`<maskprefix>`	When present, indicates write-masked (`_mask`) or zero-masked (`_maskz`) predication.
`<intrin_op>`	Indicates the basic operation of the intrinsic; for example, add for addition and sub for subtraction.
`<suffix>`	Denotes the type of data the instruction operates on. The first one or two letters of each suffix denote whether the data is packed (*p*), extended packed (*ep*), or scalar (*s*). The remaining letters and numbers denote the type, with notation as follows: • **s:** single-precision floating point • **d:** double-precision floating point • **i512:** signed 512-bit integer • **i256:** signed 256-bit integer • **i128:** signed 128-bit integer • **i64:** signed 64-bit integer • **u64:** unsigned 64-bit integer • **i32:** signed 32-bit integer • **u32:** unsigned 32-bit integer • **i16:** signed 16-bit integer • **u16:** unsigned 16-bit integer • **i8:** signed 8-bit integer • **u8:** unsigned 8-bit integer

Programs can pack eight double precision and sixteen single precision floating-point numbers within the 512-bit vectors, as well as eight 64-bit and sixteen 32-bit integers. This enables processing of twice the number of data elements that Intel® AVX or Intel® AVX2 can process with a single instruction and four times the capabilities of Intel® SSE.

Example: Write-Masking

Write-masking allows an intrinsic to perform its operation on selected SIMD elements of a source operand, with blending of the other elements from an additional SIMD operand. Consider the declarations below, where the write-mask *k* has a 1 in the even numbered bit positions 0, 3, 5, 7, 9, 11, 13 and 15, and a 0 in the odd numbered bit positions.

```
__m512 res, src, a, b;
__mmask16  k = 0x5555;
```

Then, given an intrinsic invocation such as this:

```
res = _mm512_mask_add_ps(src, k, a, b);
```

every even-numbered float32 element of the result *res* is computed as the sum of the corresponding elements in *a* and *b*, while every odd-numbered element is passed through (i.e., blended) from the corresponding float32 element in *src*.

Typical write-masked intrinsics are declared with a parameter order such that the values to be blended (*src* in the example above) are in the first parameter, and the write mask *k* immediately follows this parameter. Some intrinsics provide the blended values from a different SIMD parameter, for example: `_mm512_mask2_permutex2var_epi32`. In this case too, the mask will follow that parameter.

Example: Zero-Masking

Zero-masking is a simplified form of write-masking where there are no blended values. Instead result elements corresponding to zero bits in the write mask are simply set to zero. Given:

```
res = _mm512_maskz_add_ps(k, a, b);
```

the float32 elements of res corresponding to zeros in the write-mask *k*, are set to zero. The elements corresponding to ones in *k*, have the expected sum of corresponding elements in *a* and *b*.

Zero-masked intrinsics are typically declared with the write-mask as the first parameter, as there is no parameter for blended values.

Example: Embedded Rounding and Suppress All Exceptions (SAE)

Embedded rounding allows the floating point rounding mode to be explicitly specified for an individual operation, without having to modify the rounding controls in the MXCSR control register. The Suppress All Exceptions feature allows signaling of FP exceptions to be suppressed.

AVX-512 provides these capabilities on most 512-bit and scalar floating point operations. An intrinsic supporting these features will typically have "_round" in its name, for example:

```
__m512d _mm512_add_round_pd(__m512d a, __m512d b, int rounding);
```

To specify round-towards-zero and SAE, an invocation would appear as follows:

```
__m512d res, a, b;
res = _mm512_add_round_pd(a, b, _MM_FROUND_TO_ZERO | _MM_FROUND_NO_EXC);
```

Example: Embedded Broadcasting

Embedded broadcasting allows a single value to be broadcast across a source operand, without requiring an extra instruction. The "set1" family of intrinsics represent a broadcast operation, and the compiler can embed such operations into the EVEX prefix of an AVX-512 instruction. For example,

```
__m512 res, a;
res = _mm512_add_ps(a, _mm512_set1_ps(3.0f));
```

will add 3.0 to each float32 element of *a*.

See Also

Details of Intrinsics (general)

`__declspec(align)`
declaration

Intel® AVX site at https://software.intel.com/en-us/isa-extensions

Details of Intel® Advanced Vector Extensions Intrinsics

Intrinsics for Arithmetic Operations

Intrinsics for Addition Operations

Intrinsics for FP Addition Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_add_round_pd`, `_mm512_mask_add_round_pd`, `_mm512_maskz_add_round_pd`	Add rounded float64 vectors.	`VADDPD`
`_mm512_add_pd`, `_mm512_mask_add_pd`, `_mm512_maskz_add_pd`	Add rounded float64 vectors.	`VADDPD`
`_mm512_add_round_ps`, `_mm512_mask_add_round_ps`, `_mm512_maskz_add_round_ps`	Add rounded float32 vectors.	`VADDPS`
`_mm512_add_ps`, `_mm512_mask_add_ps`, `_mm512_maskz_add_ps`	Add rounded float32 vectors.	`VADDPS`
`_mm_add_round_sd`, `_mm_mask_add_round_sd`, `_mm_maskz_add_round_sd`	Add scalar float64 vectors.	`VADDSD`
`_mm_mask_add_sd`, `_mm_maskz_add_sd`	Add scalar float64 vectors.	`VADDSD`
`_mm_add_round_ss`, `_mm_mask_add_round_ss`, `_mm_maskz_add_round_ss`	Add scalar float32 vectors.	`VADDSS`
`_mm_mask_add_ss`, `_mm_maskz_add_ss`	Add scalar float32 vectors.	`VADDPD`

variable	definition
k	writemask used as a selector
a	first source vector element

variable	definition
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_add_pd

```
extern __m512d __cdecl _mm512_add_pd(__m512d a, __m512d b);
```

Adds packed float64 elements in *a* and *b*, and stores the result.

_mm512_mask_add_pd

```
extern __m512d __cdecl _mm512_mask_add_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Adds packed float64 elements in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_pd

```
extern __m512d __cdecl _mm512_maskz_add_pd(__mmask8 k, __m512d a, __m512d b);
```

Adds packed float64 elements in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_add_round_pd

```
extern __m512d __cdecl _mm512_add_round_pd(__m512d a, __m512d b, int round);
```

Adds packed float64 elements in *a* and *b* using rounding control *round*, and stores the result.

_mm512_mask_add_round_pd

```
extern __m512d __cdecl _mm512_mask_add_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
int round);
```

Adds packed float64 elements in *a* and *b* using rounding control *round*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_round_pd

```
extern __m512d __cdecl _mm512_maskz_add_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Adds packed float64 elements in *a* and *b* using rounding control *round*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_add_ps

```
extern __m512 __cdecl _mm512_add_ps(__m512 a, __m512 b);
```

Adds packed float32 elements in *a* and *b*, and stores the result.

_mm512_mask_add_ps

```
extern __m512 __cdecl _mm512_mask_add_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Adds packed float32 elements in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_ps

```
extern __m512 __cdecl _mm512_maskz_add_ps(__mmask16 k, __m512 a, __m512 b);
```

Adds packed float32 elements in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_add_round_ps

```
extern __m512 __cdecl _mm512_add_round_ps(__m512 a, __m512 b, int round);
```

Adds packed float32 elements in *a* and *b* using rounding control *round*, and stores the result.

_mm512_mask_add_round_ps

```
extern __m512 __cdecl _mm512_mask_add_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int
round);
```

Adds packed float32 elements in *a* and *b* using rounding control *round*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_round_ps

```
extern __m512 __cdecl _mm512_maskz_add_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Adds packed float32 elements in *a* and *b* using rounding control *round*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_add_round_sd

```
extern __m128d __cdecl _mm_add_round_sd(__m128d a, __m128d b, int round);
```

Adds the lower float64 element in *a* and *b* using rounding control *round*, stores the result in the lower destination element, and copies the upper element from *a* to the upper destination element.

_mm_mask_add_round_sd

```
extern __m128d __cdecl _mm_mask_add_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int
round);
```

Adds the lower float64 element in *a* and *b* using rounding control *round*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_add_round_sd

```
extern __m128d __cdecl _mm_maskz_add_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Adds the lower float64 element in *a* and *b* using rounding control *round*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_mask_add_sd

```
extern __m128d __cdecl _mm_mask_add_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Adds the lower float64 element in *a* and *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_add_sd

```
extern __m128d __cdecl _mm_maskz_add_sd(__mmask8 k, __m128d a, __m128d b);
```

Adds the lower float64 element in *a* and *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_add_round_ss

```
extern __m128 __cdecl _mm_add_round_ss(__m128 a, __m128 b, int round);
```

Add the lower float32 element in *a* and *b* using rounding control *round*, stores the result in the lower destination element, and copies the upper three packed elements from *a* to the upper destination elements.

_mm_mask_add_round_ss

```
extern __m128 __cdecl _mm_mask_add_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Add the lower float32 element in *a* and *b* using rounding control *round*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_add_round_ss

```
extern __m128 __cdecl _mm_maskz_add_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Add the lower float32 element in *a* and *b* using rounding control *round*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_mask_add_ss

```
extern __m128 __cdecl _mm_mask_add_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Add the lower float32 element in *a* and *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_add_ss

```
extern __m128 __cdecl _mm_maskz_add_ss(__mmask8 k, __m128 a, __m128 b);
```

Add the lower float32 element in *a* and *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

Intrinsics for Integer Addition Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_add_epi32`, `_mm512_mask_add_epi32`, `_mm512_maskz_add_epi32`	Add int32 vectors.	`VPADDD`
`_mm512_add_epi64`, `_mm512_mask_add_epi64`, `_mm512_maskz_add_epi64`	Add int64 vectors.	`VPADDQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_add_epi32

```
extern __m512i __cdecl _mm512_add_epi32(__m512i a, __m512i b);
```

Adds packed int32 elements in *a* and *b*, and stores the result.

_mm512_mask_add_epi32

```
extern __m512i __cdecl _mm512_mask_add_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Adds packed int32 elements in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_epi32

```
extern __m512i __cdecl _mm512_maskz_add_epi32(__mmask16 k, __m512i a, __m512i b);
```

Adds packed int32 elements in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_add_epi64

```
extern __m512i __cdecl _mm512_add_epi64(__m512i a, __m512i b);
```

Adds packed int64 elements in *a* and *b*, and stores the result.

_mm512_mask_add_epi64

```
extern __m512i __cdecl _mm512_mask_add_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Adds packed int64 elements in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_add_epi64

```
extern __m512i __cdecl _mm512_maskz_add_epi64(__mmask8 k, __m512i a, __m512i b);
```

Adds packed int64 elements in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Determining Minimum and Maximum Values

Intrinsics for Determining Minimum and Maximum FP Values

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_max_round_pd_mm512_mask_max_round_pd_mm512_maskz_max_round_pd _mm512_max_pd_mm512_mask_max_pd_mm512_maskz_max_pd	Calculate maximum of rounded float64 values.	VMAXPD
_mm512_max_round_ps_mm512_mask_max_round_ps_mm512_maskz_max_round_ps _mm512_max_ps_mm512_mask_max_ps_mm512_maskz_max_ps	Calculate maximum of rounded float32 values.	VMAXPS
_mm_mask_max_sd_mm_maskz_max_sd _mm_max_round_sd, _mm_mask_max_round_sd, _mm_maskz_max_round_sd	Calculate maximum of scalar float64 values.	VMAXSD
_mm_mask_max_ss_mm_maskz_max_ss _mm_max_round_ss, _mm_mask_max_round_ss, _mm_maskz_max_round_ss	Calculate maximum of scalar float32 values.	VMAXSS
_mm512_min_pd, _mm512_mask_min_pd, _mm512_maskz_min_pd	Calculate minimum of packed float64 values.	VMINPD

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_min_round_pd, _mm512_mask_min_round_pd, _mm512_maskz_min_round_pd		
_mm512_min_ps, _mm512_mask_min_ps, _mm512_maskz_min_ps _mm512_min_round_ps, _mm512_mask_min_round_ps, _mm512_maskz_min_round_ps	Calculate minimum of packed float32 values.	VMINPS
_mm_mask_min_sd_mm_maskz_min_sd _mm_min_round_sd, _mm_mask_min_round_sd, _mm_maskz_min_round_sd	Calculate minimum of scalar float64 values.	VMINSD
_mm_mask_min_ss_mm_maskz_min_ss _mm_min_round_ss, _mm_mask_min_round_ss, _mm_maskz_min_round_ss	Calculate minimum of scalar float32 values.	VMINSS

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the sae suppress all exceptions flag): • _MM_FROUND_TO_NEAREST_INT - rounds to nearest even • _MM_FROUND_TO_NEG_INF - rounds to negative infinity • _MM_FROUND_TO_POS_INF - rounds to positive infinity • _MM_FROUND_TO_ZERO - rounds to zero • _MM_FROUND_CUR_DIRECTION - rounds using default from MXCSR register

_mm512_max_pd

```
extern __m512d __cdecl _mm512_max_pd(__m512d a, __m512d b);
```

Compares packed float64 elements in *a* and *b*, and stores packed maximum values.

_mm512_mask_max_pd

```
extern __m512d __cdecl _mm512_mask_max_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Compares packed float64 elements in *a* and *b*, and stores packed maximum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_pd

```
extern __m512d __cdecl _mm512_maskz_max_pd(__mmask8 k, __m512d a, __m512d b);
```

Compares packed float64 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_round_pd

```
extern __m512d __cdecl _mm512_max_round_pd(__m512d a, __m512d b, int round);
```

Compares packed float64 elements in *a* and *b*, and stores packed maximum values.

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_mask_max_round_pd

```
extern __m512d __cdecl _mm512_mask_max_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
int round);
```

Compares packed float64 elements in *a* and *b*, and stores packed maximum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_maskz_max_round_pd

```
extern __m512d __cdecl _mm512_maskz_max_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Compares packed float64 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_max_ps

```
extern __m512 __cdecl _mm512_max_ps(__m512 a, __m512 b);
```

Compares packed float32 elements in *a* and *b*, and stores packed maximum values.

_mm512_mask_max_ps

```
extern __m512 __cdecl _mm512_mask_max_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Compares packed float32 elements in *a* and *b*, and stores packed maximum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_ps

```
extern __m512 __cdecl _mm512_maskz_max_ps(__mmask16 k, __m512 a, __m512 b);
```

Compares packed float32 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_round_ps

```
extern __m512 __cdecl _mm512_max_round_ps(__m512 a, __m512 b, int round);
```

Compares packed float32 elements in *a* and *b*, and stores packed maximum values.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_mask_max_round_ps

```
extern __m512 __cdecl _mm512_mask_max_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int
round);
```

Compares packed float32 elements in *a* and *b*, and stores packed maximum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_maskz_max_round_ps

```
extern __m512 __cdecl _mm512_maskz_max_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Compares packed float32 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_max_sd

```
extern __m128d __cdecl _mm_mask_max_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Compares the lower float64 elements in *a* and *b*, stores the maximum value in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_max_sd

```
extern __m128d __cdecl _mm_maskz_max_sd(__mmask8 k, __m128d a, __m128d b);
```

Compares the lower float64 elements in *a* and *b*, stores the maximum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_max_round_sd

```
extern __m128d __cdecl _mm_max_round_sd(__m128d a, __m128d b, int round);
```

Compares the lower float64 elements in *a* and *b*, stores the maximum value in the lower destination element, and copies the upper element from *a* to the upper destination element.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_max_round_sd

```
extern __m128d __cdecl _mm_mask_max_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int
round);
```

Compares the lower float64 elements in *a* and *b*, stores the maximum value in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_maskz_max_round_sd

```
extern __m128d __cdecl _mm_maskz_max_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Compares the lower float64 elements in *a* and *b*, stores the maximum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_max_ss

```
extern __m128 __cdecl _mm_mask_max_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Compares the lower float32 elements in *a* and *b*, stores the maximum value in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_max_ss

```
extern __m128 __cdecl _mm_maskz_max_ss(__mmask8 k, __m128 a, __m128 b);
```

Compares the lower float32 elements in *a* and *b*, stores the maximum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_max_round_ss

```
extern __m128 __cdecl _mm_max_round_ss(__m128 a, __m128 b, int round);
```

Compares the lower float32 elements in *a* and *b*, stores the maximum value in the lower destination element, and copies the upper element from *a* to the upper destination element.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_max_round_ss

```
extern __m128 __cdecl _mm_mask_max_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Compares the lower float32 elements in *a* and *b*, stores the maximum value in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_maskz_max_round_ss

```
extern __m128 __cdecl _mm_maskz_max_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Compares the lower float32 elements in *a* and *b*, stores the maximum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_min_pd

```
extern __m512d __cdecl _mm512_min_pd(__m512d a, __m512d b);
```

Compares packed float64 elements in *a* and *b*, and stores packed minimum values.

_mm512_mask_min_pd

```
extern __m512d __cdecl _mm512_mask_min_pd(__m512d src, __mmask8 k, __m512d a,__m512d b);
```

Compares packed float64 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_pd

```
extern __m512d __cdecl _mm512_maskz_min_pd(__mmask8 k, __m512d a, __m512d b);
```

Compares packed float64 elements in *a* and *b*, and store packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_round_pd

```
extern __m512d __cdecl _mm512_min_round_pd(__m512d a, __m512d b, int round);
```

Compares packed float64 elements in *a* and *b*, and stores packed minimum values.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_mask_min_round_pd

```
extern __m512d __cdecl _mm512_mask_min_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
int round);
```

Compares packed float64 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_maskz_min_round_pd

```
extern __m512d __cdecl _mm512_maskz_min_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Compares packed float64 elements in *a* and *b*, and stores packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_min_ps

```
extern __m512 __cdecl _mm512_min_ps(__m512 a, __m512 b);
```

Compares packed float32 elements in *a* and *b*, and stores packed minimum values.

_mm512_mask_min_ps

```
extern __m512 __cdecl _mm512_mask_min_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Compares packed float32 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_ps

```
extern __m512 __cdecl _mm512_maskz_min_ps(__mmask16 k, __m512 a, __m512 b;
```

Compares packed float32 elements in *a* and *b*, and store packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_round_ps

```
extern __m512 __cdecl _mm512_min_round_ps( __m512 a, __m512 b, int round);
```

Compares packed float32 elements in *a* and *b*, and stores packed minimum values.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_mask_min_round_ps

```
extern __m512 __cdecl _mm512_mask_min_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int
round);
```

Compares packed float32 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_maskz_min_round_ps

```
extern __m512 __cdecl _mm512_maskz_min_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Compares packed float32 elements in *a* and *b*, and stores packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_min_sd

```
extern __m128d __cdecl _mm_mask_min_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Compares the lower float64 elements in *a* and *b*, stores the minimum value in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_min_sd

```
extern __m128d __cdecl _mm_maskz_min_sd(__mmask8 k, __m128d a, __m128d b);
```

Compares the lower float64 elements in *a* and *b*, stores the minimum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_min_round_sd

```
extern __m128d __cdecl _mm_min_round_sd(__m128d a, __m128d b, int round);
```

Compares the lower float64 elements in *a* and *b*, stores the minimum value in the lower destination element, and copies the upper element from *a* to the upper destination element.

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_min_round_sd

```
extern __m128d __cdecl _mm_mask_min_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int
round);
```

Compares the lower float64 elements in *a* and *b*, stores the minimum value in the lower destination element of using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_maskz_min_round_sd

```
extern __m128d __cdecl _mm_maskz_min_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Compares the lower float64 elements in *a* and *b*, stores the minimum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_min_ss

```
extern __m128 __cdecl _mm_mask_min_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Compares the lower float32 elements in *a* and *b*, stores the minimum value in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_min_ss

```
extern __m128 __cdecl _mm_maskz_min_ss(__mmask8 k, __m128 a, __m128 b);
```

Compares the lower float32 elements in *a* and *b*, stores the minimum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_min_round_ss

```
extern __m128 __cdecl _mm_min_round_ss(__m128 a, __m128 b, int round);
```

Compares the lower float32 elements in *a* and *b*, stores the minimum value in the lower destination element, and copies the upper element from *a* to the upper destination element.

NOTE
Pass `_MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_min_round_ss

```
extern __m128 __cdecl _mm_mask_min_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Compares the lower float32 elements in *a* and *b*, stores the minimum value in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

NOTE
Pass `_MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_maskz_min_round_ss

```
extern __m128 __cdecl _mm_maskz_min_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Compares the lower float32 elements in *a* and *b*, stores the minimum value in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

NOTE
Pass `_MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

Intrinsics for Determining Minimum and Maximum Integer Values

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_max_epi32`, `_mm512_mask_max_epi32`, `_mm512_maskz_max_epi32`	Calculate maximum of packed int32 values.	`VPMAXSD`
`_mm512_min_epi32`, `_mm512_mask_min_epi32`, `_mm512_maskz_min_epi32`	Calculate minimum of packed int32 values.	`VPMINSD`
`_mm512_max_epu32`, `_mm512_mask_max_epu32`, `_mm512_maskz_max_epu32`	Calculate maximum of unpacked int32 values.	`VPMAXUD`
`_mm512_min_epu32`, `_mm512_mask_min_epu32`, `_mm512_maskz_min_epu32`	Calculate minimum of unpacked int32 values.	`VPMINUD`
`_mm512_max_epi64`, `_mm512_mask_max_epi64`, `_mm512_maskz_max_epi64`	Calculate maximum of packed signed int64 values.	`VPMAXSQ`
`_mm512_max_epu64`, `_mm512_mask_max_epu64`, `_mm512_maskz_max_epu64`	Calculate maximum of unpacked unsigned int64 values.	`VPMAXUQ`
`_mm512_min_epi64`, `_mm512_mask_min_epi64`, `_mm512_maskz_min_epi64`	Calculate minimum of packed signed int64 values.	`VPMINSQ`
`_mm512_min_epu64`, `_mm512_mask_min_epu64`, `_mm512_maskz_min_epu64`	Calculate minimum of unpacked unsigned int64 values.	`VPMINUQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_max_epi32

```
extern __m512i __cdecl _mm512_max_epi32(__m512i a, __m512i b);
```

Compares packed int32 elements in *a* and *b*, and stores packed maximum values.

_mm512_mask_max_epi32

```
extern __m512i __cdecl _mm512_mask_max_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Compares packed int32 elements in *a* and *b*, and stores packed maximum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epi32

```
extern __m512i __cdecl _mm512_maskz_max_epi32(__mmask16 k, __m512i a, __m512i b);
```

Compares packed int32 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_epi64

```
extern __m512i __cdecl _mm512_max_epi64(__m512i a, __m512i b);
```

Compares packed int64 elements in *a* and *b*, and stores packed maximum values.

_mm512_mask_max_epi64

```
extern __m512i __cdecl _mm512_mask_max_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Compares packed int64 elements in *a* and *b*, and stores packed maximum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epi64

```
extern __m512i __cdecl _mm512_maskz_max_epi64(__mmask8 k, __m512i a, __m512i b);
```

Compares packed int64 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_epu32

```
extern __m512i __cdecl _mm512_max_epu32(__m512i a,__m512i b);
```

Compares packed uint32 elements in *a* and *b*, and stores packed maximum values.

_mm512_mask_max_epu32

```
extern __m512i __cdecl _mm512_mask_max_epu32(__m512i src, __mmask16 k, __m512i a,__m512i b);
```

Compares packed uint32 elements in *a* and *b*, and stores packed maximum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epu32

```
extern __m512i __cdecl _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
```

Compares packed uint32 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_max_epu64

```
extern __m512i __cdecl _mm512_max_epu64(__m512i a, __m512i b);
```

Compares packed uint64 elements in *a* and *b*, and stores packed maximum values.

_mm512_mask_max_epu64

```
extern __m512i __cdecl _mm512_mask_max_epu64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Compares packed uint64 elements in *a* and *b*, and stores packed maximum values in using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_max_epu64

```
extern __m512i __cdecl _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
```

Compares packed uint64 elements in *a* and *b*, and stores packed maximum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epi32

```
extern __m512i __cdecl _mm512_min_epi32(__m512i a, __m512i b);
```

Compares packed int32 elements in *a* and *b*, and stores packed minimum values.

_mm512_mask_min_epi32

```
extern __m512i __cdecl _mm512_mask_min_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Compares packed int32 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epi32

```
extern __m512i __cdecl _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
```

Compares packed int32 elements in *a* and *b*, and stores packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epi64

```
extern __m512i __cdecl _mm512_min_epi64(__m512i a, __m512i b);
```

Compares packed int64 elements in *a* and *b*, and stores packed minimum values.

_mm512_mask_min_epi64

```
extern __m512i __cdecl _mm512_mask_min_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Compares packed int64 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epi64

```
extern __m512i __cdecl _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
```

Compares packed int64 elements in *a* and *b*, and stores packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epu32

```
extern __m512i __cdecl _mm512_min_epu32(__m512i a, __m512i b);
```

Compares packed uint32 elements in *a* and *b*, and stores packed minimum values.

_mm512_mask_min_epu32

```
extern __m512i __cdecl _mm512_mask_min_epu32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Compares packed uint32 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epu32

```
extern __m512i __cdecl _mm512_maskz_min_epu32(__mmask16 k, __m512i a, __m512i b);
```

Compares packed uint32 elements in *a* and *b*, and stores packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_min_epu64

```
extern __m512i __cdecl _mm512_min_epu64(__m512i a, __m512i b);
```

Compares packed uint64 elements in *a* and *b*, and stores packed minimum values.

_mm512_mask_min_epu64

```
extern __m512i __cdecl _mm512_mask_min_epu64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Compares packed uint64 elements in *a* and *b*, and stores packed minimum values using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_min_epu64

```
extern __m512i __cdecl _mm512_maskz_min_epu64(__mmask8 k, __m512i a, __m512i b);
```

Compares packed uint64 elements in *a* and *b*, and stores packed minimum values using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for FP Fused Multiply-Add (FMA) Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_fmadd_pd`, `_mm512_mask3_fmadd_pd`, `_mm512_mask_fmadd_pd`, `_mm512_maskz_fmadd_pd`	Multiplies float64 element vector elements, then adds the intermediate result to float64 vector elements.	`VFMADD132PD`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_fmadd_round_pd, _mm512_mask3_fmadd_round_pd, _mm512_mask_fmadd_round_pd, _mm512_maskz_fmadd_round_pd		
_mm512_fmadd_ps, _mm512_mask3_fmadd_ps, _mm512_mask_fmadd_ps, _mm512_maskz_fmadd_ps	Multiplies float32 element vector elements, then adds the intermediate result to float32 vector elements.	VFMADD132PS
_mm512_fmadd_round_ps, _mm512_mask3_fmadd_round_ps, _mm512_mask_fmadd_round_ps, _mm512_maskz_fmadd_round_ps		
_mm_mask3_fmadd_sd, _mm_mask_fmadd_sd, _mm_maskz_fmadd_sd	Multiplies float64 element vector elements, then adds the intermediate result to float64 vector elements.	VFMADD132SD
_mm_mask3_fmadd_round_sd, _mm_mask_fmadd_round_sd, _mm_maskz_fmadd_round_sd		
_mm_mask3_fmadd_ss, _mm_mask_fmadd_ss, _mm_maskz_fmadd_ss	Multiplies float32 element vector elements, then adds the intermediate result to float32 vector elements.	VFMADD132SS
_mm_mask3_fmadd_round_ss, _mm_mask_fmadd_round_ss, _mm_maskz_fmadd_round_ss		
_mm512_fmaddsub_pd, _mm512_mask3_fmaddsub_pd, _mm512_mask_fmaddsub_pd, _mm512_maskz_fmaddsub_pd	Multiplies float64 element vector elements, then alternatively add and subtract to/from the intermediate result.	VFMADDSUB132PD
_mm512_fmaddsub_round_pd, _mm512_mask3_fmaddsub_round_pd, _mm512_mask_fmaddsub_round_pd, _mm512_maskz_fmaddsub_round_pd		
_mm512_fmaddsub_ps, _mm512_mask3_fmaddsub_ps, _mm512_mask_fmaddsub_ps, _mm512_maskz_fmaddsub_ps	Multiplies float32 element vector elements, then alternatively add and subtract to/from the intermediate result.	VFMADDSUB132PS
_mm512_fmaddsub_round_ps, _mm512_mask3_fmaddsub_round_ps, _mm512_mask_fmaddsub_round_ps, _mm512_maskz_fmaddsub_round_ps		

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_fmsub_pd, _mm512_mask3_fmsub_pd, _mm512_mask_fmsub_pd, _mm512_maskz_fmsub_pd _mm512_fmsub_round_pd, _mm512_mask3_fmsub_round_pd, _mm512_mask_fmsub_round_pd, _mm512_maskz_fmsub_round_pd	Multiplies packed float64 element vector elements, then subtracts the intermediate result to float64 vector elements.	VFMSUB132PD
_mm512_fmsub_ps, _mm512_mask3_fmsub_ps, _mm512_mask_fmsub_ps, _mm512_maskz_fmsub_ps _mm512_fmsub_round_ps, _mm512_mask3_fmsub_round_ps, _mm512_mask_fmsub_round_ps, _mm512_maskz_fmsub_round_ps	Multiplies packed float32 element vector elements, then subtracts the intermediate result to float32 vector elements.	VFMSUB132PS
_mm_mask3_fmsub_sd, _mm_mask_fmsub_sd, _mm_maskz_fmsub_sd _mm_mask3_fmsub_round_sd, _mm_mask_fmsub_round_sd, _mm_maskz_fmsub_round_sd	Multiplies scalar float64 element vector elements, then subtracts the intermediate result to float64 vector elements.	VFMSUB132SD
_mm_mask3_fmsub_ss, _mm_mask_fmsub_ss, _mm_maskz_fmsub_ss _mm_mask3_fmsub_round_ss, _mm_mask_fmsub_round_ss, _mm_maskz_fmsub_round_ss	Multiplies scalar float32 element vector elements, then subtracts the intermediate result to float32 vector elements.	VFMSUB132SS
_mm512_fmsubadd_pd, _mm512_mask3_fmsubadd_pd, _mm512_mask_fmsubadd_pd, _mm512_maskz_fmsubadd_pd _mm512_fmsubadd_round_pd, _mm512_mask3_fmsubadd_round_pd, _mm512_mask_fmsubadd_round_pd, _mm512_maskz_fmsubadd_round_pd	Multiplies float64 element vector elements, then alternatively subtract and add to/from the intermediate result.	VFMSUBADD132PD
_mm512_fmsubadd_ps, _mm512_mask3_fmsubadd_ps, _mm512_mask_fmsubadd_ps, _mm512_maskz_fmsubadd_ps	Multiplies float32 element vector elements, then alternatively subtract and add to/from the intermediate result.	VFMSUBADD132PS

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_fmsubadd_round_ps, _mm512_mask3_fmsubadd_round_ps, _mm512_mask_fmsubadd_round_ps, _mm512_maskz_fmsubadd_round_ps		
_mm512_fnmadd_pd, _mm512_mask3_fnmadd_pd, _mm512_mask_fnmadd_pd, _mm512_maskz_fnmadd_pd _mm512_fnmadd_round_pd, _mm512_mask3_fnmadd_round_pd, _mm512_mask_fnmadd_round_pd, _mm512_maskz_fnmadd_round_pd	Multiplies packed float64 element vector elements, then adds the negated intermediate result to float64 vector elements.	VFNMADD132PD
_mm512_fnmadd_ps, _mm512_mask3_fnmadd_ps, _mm512_maskz_fnmadd_ps, _mm512_mask_fnmadd_ps _mm512_fnmadd_round_ps, , _mm512_mask3_fnmadd_round_ps, _mm512_mask_fnmadd_round_ps, _mm512_maskz_fnmadd_round_ps	Multiplies packed float32 element vector elements, then adds the negated intermediate result to float32 vector elements.	VFNMADD132PS
_mm_mask3_fnmadd_round_sd, _mm_mask_fnmadd_round_sd, _mm_maskz_fnmadd_round_sd _mm_maskz_fnmadd_sd, _mm_mask_fnmadd_sd, _mm_mask3_fnmadd_sd	Multiplies scalar float64 element vector elements, then adds the negated intermediate result to float64 vector elements.	VFNMADD132SD
_mm_mask3_fnmadd_ss, _mm_mask_fnmadd_ss, _mm_maskz_fnmadd_ss _mm_mask3_fnmadd_round_ss, _mm_mask_fnmadd_round_ss, _mm_maskz_fnmadd_round_ss	Multiplies scalar float32 element vector elements, then adds the negated intermediate result to float32 vector elements.	VFNMADD132SS
_mm512_fnmsub_pd, _mm512_mask3_fnmsub_pd, _mm512_mask_fnmsub_pd, _mm512_maskz_fnmsub_pd _mm512_fnmsub_round_pd, _mm512_mask3_fnmsub_round_pd, _mm512_mask_fnmsub_round_pd, _mm512_maskz_fnmsub_round_pd	Multiplies packed float64 element vector elements, then subtracts the negated intermediate result to float64 vector elements.	VFNMSUB132PD

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction

`_mm512_fnmsub_ps`, `_mm512_mask3_fnmsub_ps`, `_mm512_maskz_fnmsub_ps`, `_mm512_mask_fnmsub_ps`
 	
`_mm512_fnmsub_round_ps`, `_mm512_mask3_fnmsub_round_ps`, `_mm512_maskz_fnmsub_round_ps`, `_mm512_mask_fnmsub_round_ps`
	Multiplies packed float32 element vector elements, then subtracts the negated intermediate result to float32 vector elements.	`VFNMSUB132PS`

`_mm_maskz_fnmsub_round_sd`, `_mm_mask_fnmsub_round_sd`, `_mm_mask3_fnmsub_round_sd`
 	
`_mm_mask_fnmsub_sd`, `_mm_mask3_fnmsub_sd`, `_mm_maskz_fnmsub_sd`
	Multiplies scalar float64 element vector elements, then subtracts the negated intermediate result to float64 vector elements.	`VFNMSUB132SD`

`_mm_maskz_fnmsub_round_ss`, `_mm_mask_fnmsub_round_ss`, `_mm_mask3_fnmsub_round_ss`
 	
`_mm_mask_fnmsub_ss`, `_mm_maskz_fnmsub_ss`, `_mm_mask3_fnmsub_ss`
	Multiplies scalar float32 element vector elements, then subtracts the negated intermediate result to float32 vector elements.	`VFNMSUB132SS`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_fmadd_pd

```
extern __m512d __cdecl _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fmadd_pd

```
extern __m512d __cdecl _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmadd_pd

```
extern __m512d __cdecl _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmadd_pd

```
extern __m512d __cdecl _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmadd_round_pd

```
extern __m512d __cdecl _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fmadd_round_pd

```
extern __m512d __cdecl _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmadd_round_pd

```
extern __m512d __cdecl _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k,
int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmadd_round_pd

```
extern __m512d __cdecl _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmadd_round_ps

```
extern __m512 __cdecl _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fmadd_round_ps

```
extern __m512 __cdecl _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int
round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmadd_round_ps

```
extern __m512 __cdecl _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int
round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmadd_round_ps

```
extern __m512 __cdecl _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c,
const int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result *a* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmadd_ps

```
extern __m512 __cdecl _mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fmadd_ps

```
extern __m512 __cdecl _mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmadd_ps

```
extern __m512 __cdecl _mm512_mask3_fmadd_ps(__m512, __m512 b, __m512 c, __mmask16 k);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmadd_ps

```
extern __m512 __cdecl _mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmadd_round_ps

```
extern __m512 __cdecl _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fmadd_round_ps

```
extern __m512 __cdecl _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmadd_round_ps

```
extern __m512 __cdecl _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmadd_round_ps

```
extern __m512 __cdecl _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the intermediate result to packed elements in *c*, and stores the result *a* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmadd_sd

```
extern __m128d __cdecl _mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
```

Multiplies lower float64 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fmadd_sd

```
extern __m128d __cdecl _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
```

Multiplies lower float64 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fmadd_sd

```
extern __m128d __cdecl _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
```

Multiplies lower float64 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fmadd_round_sd

```
extern __m128d __cdecl _mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int round);
```

Multiplies lower float64 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fmadd_round_sd

```
extern __m128d __cdecl _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int
round);
```

Multiplies lower float64 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fmadd_round_sd

```
extern __m128d __cdecl _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int
round);
```

Multiplies lower float64 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fmadd_ss

```
extern __m128 __cdecl _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fmadd_ss

```
extern __m128 __cdecl _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
```

Multiplies lower float32 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_fmadd_ss

```
extern __m128 __cdecl _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask_fmadd_round_ss

```
extern __m128 __cdecl _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fmadd_round_ss

```
extern __m128 __cdecl _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_fmadd_round_ss

```
extern __m128 __cdecl _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and adds the intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm512_fmaddsub_pd

```
extern __m512d __cdecl _mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result.

_mm512_mask_fmaddsub_pd

```
extern __m512d __cdecl _mm512_mask_fmaddsub_pd(__m512d, __mmask8 k, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmaddsub_pd

```
extern __m512d __cdecl _mm512_mask3_fmaddsub_pd(__m512d a, __m512d k, __m512d b, __mmask8 c);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmaddsub_pd

```
extern __m512d __cdecl _mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmaddsub_round_pd

```
extern __m512d __cdecl _mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result.

_mm512_mask_fmaddsub_round_pd

```
extern __m512d __cdecl _mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d
c, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmaddsub_round_pd

```
extern __m512d __cdecl _mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8
k, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmaddsub_round_pd

```
extern __m512d __cdecl _mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d
c, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmaddsub_ps

```
extern __m512 __cdecl _mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result.

_mm512_mask_fmaddsub_ps

```
extern __m512 __cdecl _mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmaddsub_ps

```
extern __m512 __cdecl _mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmaddsub_ps

```
extern __m512 __cdecl _mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmaddsub_round_ps

```
extern __m512 __cdecl _mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result.

_mm512_mask_fmaddsub_round_ps

```
extern __m512 __cdecl _mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c,
int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmaddsub_round_ps

```
extern __m512 __cdecl _mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k,
int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmaddsub_round_ps

```
extern __m512 __cdecl _mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c,
int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively add and subtract packed elements in *c* to/from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmsub_pd

```
extern __m512d __cdecl _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result.

_mm512_mask_fmsub_pd

```
extern __m512d __cdecl _mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsub_pd

```
extern __m512d __cdecl _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsub_pd

```
extern __m512d __cdecl _mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmsub_round_pd

```
extern __m512d __cdecl _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result.

_mm512_mask_fmsub_round_pd

```
extern __m512d __cdecl _mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsub_round_pd

```
extern __m512d __cdecl _mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k,
int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsub_round_pd

```
extern __m512d __cdecl _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmsub_ps

```
extern __m512 __cdecl _mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result.

_mm512_mask_fmsub_ps

```
extern __m512 __cdecl _mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsub_ps

```
extern __m512 __cdecl _mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsub_ps

```
extern __m512 __cdecl _mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmsub_round_ps

```
extern __m512 __cdecl _mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result.

_mm512_mask_fmsub_round_ps

```
extern __m512 __cdecl _mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int
round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsub_round_ps

```
extern __m512 __cdecl _mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int
round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsub_round_ps

```
extern __m512 __cdecl _mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int
round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fmsub_sd

```
extern __m128d __cdecl _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fmsub_sd

```
extern __m128d __cdecl _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fmsub_sd

```
extern __m128d __cdecl _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fmsub_round_sd

```
extern __m128d __cdecl _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int
round);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fmsub_round_sd

```
extern __m128d __cdecl _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int
round);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fmsub_round_sd

```
extern __m128d __cdecl _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int
round);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fmsub_ss

```
extern __m128 __cdecl _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fmsub_ss

```
extern __m128 __cdecl _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_fmsub_ss

```
extern __m128 __cdecl _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask_fmsub_round_ss

```
extern __m128 __cdecl _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fmsub_round_ss

```
extern __m128 __cdecl _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_fmsub_round_ss

```
extern __m128 __cdecl _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the intermediate result. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm512_fmsubadd_pd

```
extern __m512d __cdecl _mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result.

_mm512_mask_fmsubadd_pd

```
extern __m512d __cdecl _mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsubadd_pd

```
extern __m512d __cdecl _mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsubadd_pd

```
extern __m512d __cdecl _mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmsubadd_round_pd

```
extern __m512d __cdecl _mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result.

_mm512_mask_fmsubadd_round_pd

```
extern __m512d __cdecl _mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d
c, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsubadd_round_pd

```
extern __m512d __cdecl _mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8
k, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsubadd_round_pd

```
extern __m512d __cdecl _mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d
c, int round);
```

Multiplies packed float64 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmsubadd_ps

```
extern __m512 __cdecl _mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result.

_mm512_mask_fmsubadd_ps

```
extern __m512 __cdecl _mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsubadd_ps

```
extern __m512 __cdecl _mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsubadd_ps

```
extern __m512 __cdecl _mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fmsubadd_round_ps

```
extern __m512 __cdecl _mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result.

_mm512_mask_fmsubadd_round_ps

```
extern __m512 __cdecl _mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c,
int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fmsubadd_round_ps

```
extern __m512 __cdecl _mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k,
int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fmsubadd_round_ps

```
extern __m512 __cdecl _mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c,
int round);
```

Multiplies packed float32 elements in *a* and *b*, alternatively subtract and add packed elements in *c* from/to the intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fnmadd_pd

```
extern __m512d __cdecl _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fnmadd_pd

```
extern __m512d __cdecl _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmadd_pd

```
extern __m512d __cdecl _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmadd_pd

```
extern __m512d __cdecl _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fnmadd_round_pd

```
extern __m512d __cdecl _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fnmadd_round_pd

```
extern __m512d __cdecl _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmadd_round_pd

```
extern __m512d __cdecl _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k,
int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmadd_round_pd

```
extern __m512d __cdecl _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fnmadd_ps

```
extern __m512 __cdecl _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fnmadd_ps

```
extern __m512 __cdecl _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmadd_ps

```
extern __m512 __cdecl _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmadd_ps

```
extern __m512 __cdecl _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fnmadd_round_ps

```
extern __m512 __cdecl _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result.

_mm512_mask_fnmadd_round_ps

```
extern __m512 __cdecl _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int
round);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmadd_round_ps

```
extern __m512 __cdecl _mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k,
int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmadd_round_ps

```
extern __m512 __cdecl _mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c,
int round);
```

Multiplies packed float32 elements in *a* and *b*, adds the negated intermediate result to packed elements in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fnmadd_sd

```
extern __m128d __cdecl _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
```

Multiplies lower float64 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fnmadd_sd

```
extern __m128d __cdecl _mm_mask3_fnmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
```

Multiplies lower float64 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fnmadd_sd

```
extern __m128d __cdecl _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
```

Multiplies lower float64 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fnmadd_round_sd

```
extern __m128d __cdecl _mm_mask_fnmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int round);
```

Multiplies lower float64 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fnmadd_round_sd

```
extern __m128d __cdecl _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int round);
```

Multiplies lower float64 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fnmadd_round_sd

```
extern __m128d __cdecl _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int round);
```

Multiplies lower float64 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fnmadd_ss

```
extern __m128 __cdecl _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fnmadd_ss

```
extern __m128 __cdecl _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
```

Multiplies lower float32 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_fnmadd_ss

```
extern __m128 __cdecl _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask_fnmadd_round_ss

```
extern __m128 __cdecl _mm_mask_fnmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int round);
```

Multiplies lower float32 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fnmadd_round_ss

```
extern __m128 __cdecl _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int round);
```

Multiplies lower float32 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_fnmadd_round_ss

```
extern __m128 __cdecl _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int round);
```

Multiplies lower float32 elements in *a* and *b*, and adds the negated intermediate result to lower element in *c*. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm512_fnmsub_pd

```
extern __m512d __cdecl _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result.

_mm512_mask_fnmsub_pd

```
extern __m512d __cdecl _mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmsub_pd

```
extern __m512d __cdecl _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmsub_pd

```
extern __m512d __cdecl _mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fnmsub_round_pd

```
extern __m512d __cdecl _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result.

_mm512_mask_fnmsub_round_pd

```
extern __m512d __cdecl _mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmsub_round_pd

```
extern __m512d __cdecl _mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k,
int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmsub_round_pd

```
extern __m512d __cdecl _mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c,
int round);
```

Multiplies packed float64 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fnmsub_ps

```
extern __m512 __cdecl _mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result.

_mm512_mask_fnmsub_ps

```
extern __m512 __cdecl _mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmsub_ps

```
extern __m512 __cdecl _mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmsub_ps

```
extern __m512 __cdecl _mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_fnmsub_round_ps

```
extern __m512 __cdecl _mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result.

_mm512_mask_fnmsub_round_ps

```
extern __m512 __cdecl _mm512_mask_fnmsub_round_ps(__m512 c, __mmask16 k, __m512 a, __m512 b, int
round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask3_fnmsub_round_ps

```
extern __m512 __cdecl _mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k,
int round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm512_maskz_fnmsub_round_ps

```
extern __m512 __cdecl _mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c,
int round);
```

Multiplies packed float32 elements in *a* and *b*, subtracts packed elements in *c* from the negated intermediate result, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_fnmsub_sd

```
extern __m128d __cdecl _mm_mask_fnmsub_sd(__m128d c, __mmask8 k, __m128d a, __m128d b);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fnmsub_sd

```
extern __m128d __cdecl _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fnmsub_sd

```
extern __m128d __cdecl _mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fnmsub_ss

```
extern __m128 __cdecl _mm_mask_fnmsub_ss(__m128 c, __mmask8 k, __m128 a, __m128 b);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fnmsub_ss

```
extern __m128 __cdecl _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element, and copies upper element from *a* to upper destination element using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fnmsub_ss

```
extern __m128 __cdecl _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask_fnmsub_round_ss

```
extern __m128 __cdecl _mm_mask_fnmsub_round_ss(__m128 c, __mmask8 k, __m128 a, __m128 b, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fnmsub_round_ss

```
extern __m128 __cdecl _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int
round);
```

Multiplies lower float32 elements in *a* and *b*, subtract lower element in *c* from the negated intermediate result, Stores the result in lower destination element, and copies upper element from *a* to upper destination element using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fnmsub_round_ss

```
extern __m128 __cdecl _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int
round);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask_fnmsub_round_sd

```
extern __m128d __cdecl _mm_mask_fnmsub_round_sd(__m128d c, __mmask8 k, __m128d a, __m128d b, int
round);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask3_fnmsub_round_sd

```
extern __m128d __cdecl _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, 
int round);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_fnmsub_round_sd

```
extern __m128d __cdecl _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, 
int round);
```

Multiplies lower float64 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_mask_fnmsub_ss

```
extern __m128 __cdecl _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using writemask *k* (the element is copied from *c* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask3_fnmsub_ss

```
extern __m128 __cdecl _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element, and copies upper element from *a* to upper destination element using writemask *k* (elements are copied from *c* when the corresponding mask bit is not set).

_mm_maskz_fnmsub_ss

```
extern __m128 __cdecl _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
```

Multiplies lower float32 elements in *a* and *b*, and subtracts lower element in *c* from the negated intermediate result. Stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

Intrinsics for Multiplication Operations

Intrinsics for FP Multiplication Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction

`_mm_mul_round_sd`, `_mm_mask_mul_round_sd`, `_mm_maskz_mul_round_sd`
 	
`_mm_mask_mul_sd`, `_mm_maskz_mul_sd`
	Multiplies rounded vectors.	`VMULSD`

`_mm_mul_round_ss`, `_mm_mask_mul_round_ss`, `_mm_maskz_mul_round_ss`
 	
`_mm_mask_mul_ss`, `_mm_maskz_mul_ss`
	Multiplies rounded vectors.	`VMULSS`

`_mm512_mul_round_pd`, `_mm512_mask_mul_round_pd`, `_mm512_maskz_mul_round_pd`
 	
`_mm512_mul_pd`, `_mm512_mask_mul_pd`, `_mm512_maskz_mul_pd`
	Multiplies rounded float64 vectors.	`VMULPD`

`_mm512_mul_round_ps`, `_mm512_mask_mul_round_ps`, `_mm512_maskz_mul_round_ps`
 	
`_mm512_mul_ps`, `_mm512_mask_mul_ps`, `_mm512_maskz_mul_ps`
	Multiplies rounded float32 vectors.	`VMULPS`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_mul_pd

```
extern __m512d __cdecl _mm512_mul_pd(__m512d a, __m512d b);
```

Multiplies packed float64 elements in *a* and *b*, stores the result.

_mm512_mask_mul_pd

```
extern __m512d __cdecl _mm512_mask_mul_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Multiplies packed float64 elements in *a* and *b*, stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mul_pd

```
extern __m512d __cdecl _mm512_maskz_mul_pd(__mmask8 k, __m512d a, __m512d b);
```

Multiplies packed float64 elements in *a* and *b*, stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mul_round_pd

```
extern __m512d __cdecl _mm512_mul_round_pd(__m512d a, __m512d b, int round);
```

Multiplies packed float64 elements in *a* and *b*, stores the result.

_mm512_mask_mul_round_pd

```
extern __m512d __cdecl _mm512_mask_mul_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
int round);
```

Multiplies packed float64 elements in *a* and *b*, stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mul_round_pd

```
extern __m512d __cdecl _mm512_maskz_mul_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Multiplies packed float64 elements in *a* and *b*, stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mul_ps

```
extern __m512 __cdecl _mm512_mul_ps(__m512 a, __m512 b);
```

Multiplies packed float32 elements in *a* and *b*, stores the result.

_mm512_mask_mul_ps

```
extern __m512 __cdecl _mm512_mask_mul_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Multiplies packed float32 elements in *a* and *b*, stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mul_ps

```
extern __m512 __cdecl _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
```

Multiplies packed float32 elements in *a* and *b*, stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mul_round_ps

```
extern __m512 __cdecl _mm512_mul_round_ps(__m512 a, __m512 b, int round);
```

Multiplies packed float32 elements in *a* and *b*, stores the result.

_mm512_mask_mul_round_ps

```
extern __m512 __cdecl _mm512_mask_mul_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int round);
```

Multiplies packed float32 elements in *a* and *b*, stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mul_round_ps

```
extern __m512 __cdecl _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Multiplies packed float32 elements in *a* and *b*, stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mul_round_sd

```
extern __m128d __cdecl _mm_mul_round_sd(__m128d a, __m128d b, int round);
```

Multiplies the lower float64 element in *a* and *b*, stores the result in the lower destination element, and copy the upper element from *a* to the upper destination element.

_mm_mask_mul_round_sd

```
extern __m128d __cdecl _mm_mask_mul_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int round);
```

Multiplies the lower float64 element in *a* and *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_maskz_mul_round_sd

```
extern __m128d __cdecl _mm_maskz_mul_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Multiplies the lower float64 element in *a* and *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_mask_mul_sd

```
extern __m128d __cdecl  _mm_mask_mul_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Multiplies the lower float64 element in *a* and *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_maskz_mul_sd

```
extern __m128d __cdecl _mm_maskz_mul_sd(__mmask8 k, __m128d a, __m128d b);
```

Multiplies the lower float64 element in *a* and *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_mul_round_ss

```
extern __m128 __cdecl _mm_mul_round_ss(__m128 a, __m128 b, int round);
```

Multiplies the lower float32 element in *a* and *b*, stores the result in the lower destination element, and copy the upper three packed elements from *a* to the upper destination elements.

_mm_mask_mul_round_ss

```
extern __m128 __cdecl _mm_mask_mul_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Multiplies the lower float32 element in *a* and *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_mul_round_ss

```
extern __m128 __cdecl _mm_maskz_mul_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Multiplies the lower float32 element in *a* and *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper three packed elements from *a* to the upper destination elements.

_mm_mask_mul_ss

```
extern __m128 __cdecl _mm_mask_mul_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Multiplies the lower float32 element in *a* and *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_mul_ss

```
extern __m128 __cdecl _mm_maskz_mul_ss(__mmask8 k, __m128 a, __m128 b);
```

Multiplies the lower float32 element in *a* and *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper three packed elements from *a* to the upper destination elements.

Intrinsics for Integer Multiplication Operations

The prototypes for Intel© Advanced Vector Extensions 512 (Intel© AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_mul_epi32`, `_mm512_mask_mul_epi32`, `_mm512_maskz_mul_epi32`	Multiplies alternating int32 vectors together to produce int64.	`VPMULDQ`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_mul_epu32`, `_mm512_mask_mul_epu32`, `_mm512_maskz_mul_epu32`	Multiplies alternating unsigned int32 vectors together to produce int64.	`VPMULUDQ`
`_mm512_mullo_epi32`, `_mm512_mask_mullo_epi32`	Multiplies int32 vectors together to produce int64.	`VPMULLD`
`_mm512_mullox_epi64`, `_mm512_mask_mullox_epi64`	Multiplies int64 vectors together to produce int64.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_mul_epi32

```
extern __m512i __cdecl _mm512_mul_epi32(__m512i a, __m512i b);
```

Multiplies the low int32 elements from each packed 64-bit element in *a* and *b*, and stores the signed 64-bit result.

_mm512_mask_mul_epi32

```
extern __m512i __cdecl _mm512_mask_mul_epi32(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Multiplies the low int32 elements from each packed 64-bit element in *a* and *b*, and stores the signed 64-bit result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mul_epi32

```
extern __m512i __cdecl _mm512_maskz_mul_epi32(__mmask8 k, __m512i a, __m512i b);
```

Multiplies the low int32 elements from each packed 64-bit element in *a* and *b*, and stores the signed 64-bit result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mullo_epi32

```
extern __m512i __cdecl _mm512_mullo_epi32(__m512i a, __m512i b);
```

Multiplies the packed int32 elements in *a* and *b*, producing intermediate int64 elements, and stores the low 32 bits of the intermediate integers.

_mm512_mask_mullo_epi32

```
extern __m512i __cdecl _mm512_mask_mullo_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Multiplies the packed int32 elements in *a* and *b*, producing intermediate int64 elements, and stores the low 32 bits of the intermediate integers in destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mul_epu32

```
extern __m512i __cdecl _mm512_mul_epu32(__m512i a, __m512i b);
```

Multiplies the low unsigned int32 elements from each packed 64-bit element in *a* and *b*, and stores the unsigned 64-bit result.

_mm512_mask_mul_epu32

```
extern __m512i __cdecl _mm512_mask_mul_epu32(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Multiplies the low unsigned int32 elements from each packed 64-bit element in *a* and *b*, and stores the unsigned 64-bit result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mul_epu32

```
extern __m512i __cdecl _mm512_maskz_mul_epu32(__mmask8 k, __m512i a, __m512i b);
```

Multiplies the low unsigned int32 elements from each packed 64-bit element in *a* and *b*, and stores the unsigned 64-bit result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mullox_epi64

```
extern __m512i __cdecl _mm512_mullox_epi64(__m512i a, __m512i b);
```

Multiplies each packed int64 element in *a* and *b*, and selects the low bits of each product.

_mm512_mask_mullox_epi64

```
extern __m512i __cdecl _mm512_mask_mullox_epi64(__m512i, __mmask8 k, __m512i a, __m512i b);
```

Multiplies each packed int64 element in *a* and *b*, and selects the low bits of each product, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Subtraction Operations

Intrinsics for FP Subtraction Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_sub_pd`, `_mm512_mask_sub_pd`, `_mm512_maskz_sub_pd` `_mm512_sub_round_pd`, `_mm512_mask_sub_round_pd`, `_mm512_maskz_sub_round_pd`	Subtracts float64 vectors.	`VSUBPD`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_sub_ps`, `_mm512_mask_sub_ps`, `_mm512_maskz_sub_ps` `_mm512_sub_round_ps`, `_mm512_mask_sub_round_ps`, `_mm512_maskz_sub_round_ps`	Subtracts float32 vectors.	`VSUBPS`
`_mm_mask_sub_sd`, `_mm_maskz_sub_sd` `_mm_sub_round_sd`, `_mm_mask_sub_round_sd`, `_mm_maskz_sub_round_sd`	Subtracts float64 vectors.	`VSUBSD`
`_mm_mask_sub_ss`, `_mm_maskz_sub_ss` `_mm_sub_round_ss`, `_mm_mask_sub_round_ss`, `_mm_maskz_sub_round_ss`	Subtracts float32 vectors.	`VSUBSS`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_sub_pd

```
extern __m512d __cdecl _mm512_sub_pd(__m512d a, __m512d b);
```

Subtracts packed float64 elements in vector *b* from vector *a*, and stores the result.

_mm512_mask_sub_pd

```
extern __m512d __cdecl _mm512_mask_sub_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Subtracts packed float64 elements in *b* from packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sub_pd

```
extern __m512d __cdecl _mm512_maskz_sub_pd(__mmask8 k, __m512d a, __m512d b);
```

Subtracts packed float64 elements in *b* from packed float64 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sub_round_pd

```
extern __m512d __cdecl _mm512_sub_round_pd(__m512d a, __m512d b, int round);
```

Subtracts packed float64 elements in *b* from packed float64 elements in *a*using rounding control *round*, and stores the result.

_mm512_mask_sub_round_pd

```
extern __m512d __cdecl _mm512_mask_sub_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
int round);
```

Subtracts packed float64 elements in *b* from packed float64 elements in *a* using rounding control *round*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sub_round_pd

```
extern __m512d __cdecl _mm512_maskz_sub_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Subtracts packed float64 elements in *b* from packed float64 elements in *a* using rounding control *round*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sub_ps

```
extern __m512 __cdecl _mm512_sub_ps(__m512 a, __m512 b);
```

Subtracts packed float32 elements in *b* from packed float32 elements in *a*, and stores the result.

_mm512_mask_sub_ps

```
extern __m512 __cdecl _mm512_mask_sub_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Subtracts packed float32 elements in *b* from packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sub_ps

```
extern __m512 __cdecl _mm512_maskz_sub_ps(__mmask16 k, __m512 a, __m512 b);
```

Subtracts packed float32 elements in *b* from packed float32 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sub_round_ps

```
extern __m512 __cdecl _mm512_sub_round_ps(__m512 a, __m512 b, int round);
```

Subtracts packed float32 elements in *b* from packed float32 elements in *a*, and stores the result.

_mm512_mask_sub_round_ps

```
extern __m512 __cdecl _mm512_mask_sub_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int
round);
```

Subtracts packed float32 elements in *b* from packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sub_round_ps

```
extern __m512 __cdecl _mm512_maskz_sub_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Subtracts packed float32 elements in *b* from packed float32 elements in *a* using rounding control *round*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_sub_sd

```
extern __m128d __cdecl _mm_mask_sub_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Subtracts the lower float64 element in *b* from the lower float64 element in *a*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_sub_sd

```
extern __m128d __cdecl _mm_maskz_sub_sd(__mmask8 k, __m128d a, __m128d b);
```

Subtracts the lower float64 element in *b* from the lower float64 element in *a*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_sub_round_sd

```
extern __m128d __cdecl _mm_sub_round_sd(__m128d a, __m128d b, int round);
```

Subtracts the lower float64 element in *b* from the lower float64 element in *a*, stores the result in the lower destination element, and copies the upper element from *a* to the upper destination element.

_mm_mask_sub_round_sd

```
extern __m128d __cdecl _mm_mask_sub_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int
round);
```

Subtracts the lower float64 element in *b* from the lower float64 element in *a* using rounding control *round*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_sub_round_sd

```
extern __m128d __cdecl _mm_maskz_sub_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Subtracts the lower float64 element in *b* from the lower float64 element in *a*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_sub_round_ss

```
extern __m128 __cdecl _mm_sub_round_ss(__m128 a, __m128 b, int round);
```

Subtracts the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element, and copies the upper three packed elements from *a* to the upper destination elements.

_mm_mask_sub_round_ss

```
extern __m128 __cdecl _mm_mask_sub_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Subtracts the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_sub_round_ss

```
extern __m128 __cdecl _mm_maskz_sub_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Subtracts the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_mask_sub_ss

```
extern __m128 __cdecl _mm_mask_sub_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Subtracts the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_sub_ss

```
extern __m128 __cdecl _mm_maskz_sub_ss(__mmask8 k, __m128 a, __m128 b);
```

Subtracts the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_sub_round_ss

```
extern __m128 __cdecl _mm_sub_round_ss(__m128 a, __m128 b, int round);
```

Subtracts the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element, and copy the upper three packed elements from *a* to the upper destination elements.

_mm_mask_sub_round_ss

```
extern __m128 __cdecl _mm_mask_sub_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Subtract the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_sub_round_ss

```
extern __m128 __cdecl _mm_maskz_sub_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Subtracts the lower float32 element in *b* from the lower float32 element in *a*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

Intrinsics for Integer Subtraction Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_sub_epi32`, `_mm512_maskz_sub_epi32`	Subtracts int32 elements.	`VPSUBD`
`_mm512_sub_epi64`, `_mm512_mask_sub_epi64`, `_mm512_maskz_sub_epi64`	Subtracts int64 elements.	`VPSUBQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_sub_epi32

```
extern __m512i __cdecl _mm512_sub_epi32(__m512i a, __m512i b);
```

Subtracts packed 32-bit integers in *b* from packed 32-bit integers in *a*, and stores the result.

_mm512_maskz_sub_epi32

```
extern __m512i __cdecl _mm512_maskz_sub_epi32(__mmask16 k, __m512i a, __m512i b);
```

Subtracts packed 32-bit integers in *b* from packed 32-bit integers in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sub_epi64

```
extern __m512i __cdecl _mm512_sub_epi64(__m512i a, __m512i b);
```

Subtracts packed 64-bit integers in *b* from packed 64-bit integers in *a*, and stores the result.

_mm512_mask_sub_epi64

```
extern __m512i __cdecl _mm512_mask_sub_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Subtracts packed 64-bit integers in *b* from packed 64-bit integers in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sub_epi64

```
extern __m512i __cdecl _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
```

Subtracts packed 64-bit integers in *b* from packed 64-bit integers in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Other Mathematics Operations

Intrinsics for FP Division Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_div_round_pd, _mm512_mask_div_round_pd, _mm512_maskz_div_round_pd	Calculates quotient of a rounded division operation of packed float64 elements.	VDIVPD
_mm512_div_pd, _mm512_mask_div_pd, _mm512_maskz_div_pd		
_mm512_div_round_ps, _mm512_mask_div_round_ps, _mm512_maskz_div_round_ps	Calculates quotient of a rounded division operation of packed float32 elements.	VDIVPS
_mm512_div_ps, _mm512_mask_div_ps, _mm512_maskz_div_ps		
_mm_div_round_sd, _mm_mask_div_round_sd, _mm_maskz_div_round_sd	Calculates quotient of a rounded division operation of scalar float64 elements.	VDIVSD
_mm_mask_div_sd, _mm_maskz_div_sd		
_mm_div_round_ss, _mm_mask_div_round_ss, _mm_maskz_div_round_ss	Calculates quotient of a rounded division operation of scalar float32 elements.	VDIVSS
_mm_mask_div_ss, _mm_maskz_div_ss		

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element

variable	definition
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_div_pd

```
extern __m512d __cdecl _mm512_div_pd(__m512d a, __m512d b);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_pd

```
extern __m512d __cdecl _mm512_mask_div_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_pd

```
extern __m512d __cdecl _mm512_maskz_div_pd(__mmask8 k, __m512d a, __m512d b);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_div_round_pd

```
extern __m512d __cdecl _mm512_div_round_pd(__m512d a, __m512d b, int round);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_round_pd

```
extern __m512d __cdecl _mm512_mask_div_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
int round);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_round_pd

```
extern __m512d __cdecl _mm512_maskz_div_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_div_ps

```
extern __m512 __cdecl _mm512_div_ps(__m512 a, __m512 b);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_ps

```
extern __m512 __cdecl _mm512_mask_div_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_ps

```
extern __m512 __cdecl _mm512_maskz_div__ps(__mmask16 k, __m512 a, __m512 b);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_div_round_ps

```
extern __m512 __cdecl _mm512_div_round_ps(__m512 a, __m512 b, int round);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_round_ps

```
extern __m512 __cdecl _mm512_mask_div_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int
round);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_round_ps

```
extern __m512 __cdecl _mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_div_sd

```
extern __m128d __cdecl _mm_mask_div_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Divides the lower float64 element in *a* by the lower float64 element in *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_div_sd

```
extern __m128d __cdecl _mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
```

Divides the lower float64 element in *a* by the lower float64 element in *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_div_round_sd

```
extern __m128d __cdecl _mm_div_round_sd(__m128d a, __m128d b, int round);
```

Divides the lower float64 element in *a* by the lower float64 element in *b*, stores the result in the lower destination element, and copies the upper element from *a* to the upper destination element.

_mm_mask_div_round_sd

```
extern __m128d __cdecl _mm_maskz_div_round_sd(__mmask8 src, __m128d k, __m128d a, int round);
```

Divides the lower float64 element in *a* by the lower float64 element in *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_div_round_sd

```
extern __m128d __cdecl _mm_mask_div_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Divides the lower float64 element in *a* by the lower float64 element in *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_div_round_ss

```
extern __m128 __cdecl _mm_div_round_ss(__m128 a, __m128 b, int round);
```

Divides the lower float32 element in *a* by the lower float32 element in *b*, stores the result in the lower destination element, and copies the upper three packed elements from *a* to the upper destination elements.

_mm_mask_div_round_ss

```
extern __m128 __cdecl _mm_mask_div_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Divides the lower float32 element in *a* by the lower float32 element in *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_div_round_ss

```
extern __m128 __cdecl _mm_maskz_div_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Divides the lower float32 element in *a* by the lower float32 element in *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_mask_div_ss

```
extern __m128 __cdecl _mm_mask_div_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Divides the lower float32 element in *a* by the lower float32 element in *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_div_ss

```
extern __m128 __cdecl _mm_maskz_div_ss(__mmask8 k, __m128 a, __m128 b);
```

Divides the lower float32 element in *a* by the lower float32 element in *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

Intrinsics for Absolute Value Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_abs_epi32`, `_mm512_mask_abs_epi32`, `_mm512_maskz_abs_epi32`	Computes absolute value of int32 vector elements.	`VPABSD`
`_mm512_abs_epi64`, `_mm512_mask_abs_epi64`, `_mm512_maskz_abs_epi64`	Computes absolute value of int64 vector elements.	`VPABSQ`

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result

_mm512_abs_epi32

```
extern __m512i __cdecl _mm512_abs_epi32(__m512i a);
```

Computes absolute value of packed int32 elements in *a*, and stores unsigned results in destination.

_mm512_mask_abs_epi32

```
extern __m512i __cdecl _mm512_mask_abs_epi32(__m512i src, __mmask16 k, __m512i a);
```

Computes absolute value of packed int32 elements in *a*, and stores unsigned results in destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_abs_epi32

```
extern __m512i __cdecl _mm512_maskz_abs_epi32(__mmask16 k, __m512i a);
```

Computes absolute value of packed int32 elements in *a*, and stores unsigned results in destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_abs_epi64

```
extern __m512i __cdecl _mm512_abs_epi64(__m512i a);
```

Computes absolute value of packed int64 elements in *a*, and stores unsigned results in destination.

_mm512_mask_abs_epi64

```
extern __m512i __cdecl _mm512_mask_abs_epi64(__m512i src, __mmask8 k, __m512i a);
```

Computes absolute value of packed int64 elements in *a*, and stores unsigned results in destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_abs_epi64

```
extern __m512i __cdecl _mm512_maskz_abs_epi64(__mmask8 k, __m512i a);
```

Computes absolute value of packed int64 elements in *a*, and stores unsigned results in destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Scale Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction

`_mm512_scalef_pd`, `_mm512_mask_scalef_pd`, `_mm512_maskz_scalef_pd`
 	
`_mm512_scalef_round_pd`, `_mm512_mask_scalef_round_pd`, `_mm512_maskz_scalef_round_pd`
	Scale packed float64 values with float64 values.	`VSCALEFPD`

`_mm512_scalef_ps`, `_mm512_mask_scalef_ps`, `_mm512_maskz_scalef_ps`
 	
`_mm512_scalef_round_ps`, `_mm512_mask_scalef_round_ps`, `_mm512_maskz_scalef_round_ps`
	Scale packed float32 values with float32 values.	`VSCALEFSD`

`_mm_scalef_sd`, `_mm_mask_scalef_sd`, `_mm_maskz_scalef_sd`
 	
`_mm_scalef_round_sd`, `_mm_mask_scalef_round_sd`, `_mm_maskz_scalef_round_sd`
	Scale scalar float64 values with float64 values.	`VSCALEFPS`

`_mm_scalef_round_ss`, `_mm_mask_scalef_round_ss`, `_mm_maskz_scalef_round_ss`
 	
`_mm_scalef_ss`, `_mm_mask_scalef_ss`, `_mm_maskz_scalef_ss`
	Scale scalar float32 values with float32 values.	`VSCALEFSS`

`_mm512_roundscale_pd`, `_mm512_mask_roundscale_pd`, `_mm512_maskz_roundscale_pd`	Scale packed float64 values with float64 values.	`VRNDSCALEPD`
`_mm512_roundscale_ps`, `_mm512_mask_roundscale_ps`, `_mm512_maskz_roundscale_ps`	Scale packed float32 values with float32 values.	`VRNDSCALEPS`
`_mm_roundscale_sd`, `_mm_mask_roundscale_sd`, `_mm_maskz_roundscale_sd`	Scale scalar float64 values with float64 values.	`VRNDSCALESD`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm_roundscale_round_sd`, `_mm_mask_roundscale_round_sd`, `_mm_maskz_roundscale_round_sd`		
`_mm_roundscale_ss`, `_mm_mask_roundscale_ss`, `_mm_maskz_roundscale_ss`	Scale scalar float32 values with float32 values.	`VRNDSCALE`
`_mm_roundscale_round_ss`, `_mm_mask_roundscale_round_ss`, `_mm_maskz_roundscale_round_ss`		

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register
imm	8-bit immediate integer specifies offset for destination

_mm512_roundscale_pd

```
extern __m512d __cdecl _mm512_roundscale_pd(__m512d a, int imm);
```

Performs a floating point scale of packed float64 elements in *a* to the number of fraction bits specified by *imm*, and stores the result.

_mm512_mask_roundscale_pd

```
extern __m512d __cdecl _mm512_mask_roundscale_pd(__m512d src, __mmask8 k, __m512d a, int imm);
```

Performs a floating point scale of packed float64 elements in *a* to the number of fraction bits specified by *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_roundscale_pd

```
extern __m512d __cdecl _mm512_maskz_roundscale_pd(__mmask8 k, __m512d a, int imm);
```

Performs a floating point scale of packed float64 elements in *a* to the number of fraction bits specified by *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_roundscale_ps

```
extern __m512 __cdecl _mm512_roundscale_ps(__m512 a, int imm);
```

Performs a floating point scale of packed float32 elements in *a* to the number of fraction bits specified by *imm*, and stores the result.

_mm512_mask_roundscale_ps

```
extern __m512 __cdecl _mm512_mask_roundscale_ps(__m512 src, __mmask16 k, __m512 a, int imm);
```

Performs a floating point scale of packed float32 elements in *a* to the number of fraction bits specified by *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_roundscale_ps

```
extern __m512 __cdecl _mm512_maskz_roundscale_ps(__mmask16 k, __m512 a, int imm);
```

Performs a floating point scale of packed float32 elements in *a* to the number of fraction bits specified by *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_roundscale_round_sd

```
extern __m128d __cdecl _mm_roundscale_round_sd(__m128d a, __m128d b, const int imm, const int
round);
```

Rounds the lower float64 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element, and copies the upper element from *b* to the upper destination element.

_mm_mask_roundscale_round_sd

```
extern __m128d __cdecl _mm_mask_roundscale_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d
b, const int imm, const int round);
```

Rounds the lower float64 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_maskz_roundscale_round_sd

```
extern __m128d __cdecl _mm_maskz_roundscale_round_sd(__mmask8 k, __m128d a, __m128d b, const int
imm, const int round);
```

Rounds the lower float64 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_roundscale_sd

```
extern __m128d __cdecl _mm_roundscale_sd(__m128d a, __m128d b, const int imm);
```

Rounds the lower float64 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element, and copies the upper element from *b* to the upper destination element.

_mm_mask_roundscale_sd

```
extern __m128d __cdecl _mm_mask_roundscale_sd(__m128d old, __mmask8 k, __m128d a, __m128d b,
const int imm);
```

Rounds the lower float64 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_maskz_roundscale_sd

```
extern __m128d __cdecl _mm_maskz_roundscale_sd(__mmask8 k, __m128d a, __m128d b, const int imm);
```

Rounds the lower float64 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_roundscale_round_ss

```
extern __m128 __cdecl _mm_roundscale_round_ss(__m128 a, __m128 b, const int imm, const int
round);
```

Rounds the lower float32 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element, and copies the upper three packed elements from *b* to the upper destination elements.

_mm_mask_roundscale_round_ss

```
extern __m128 __cdecl _mm_mask_roundscale_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b,
const int imm, const int round);
```

Rounds the lower float32 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

_mm_maskz_roundscale_round_ss

```
extern __m128 __cdecl _mm_maskz_roundscale_round_ss(__mmask8 k, __m128 a, __m128 b, const int
imm, const int round);
```

Rounds the lower float32 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

_mm_roundscale_ss

```
extern __m128 __cdecl _mm_roundscale_ss(__m128 a, __m128 b, const int imm);
```

Rounds the lower float32 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element, and copies the upper three packed elements from *b* to the upper destination elements.

_mm_mask_roundscale_ss

```
extern __m128 __cdecl _mm_mask_roundscale_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, const
int imm);
```

Rounds the lower float32 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

_mm_maskz_roundscale_ss

```
extern __m128 __cdecl _mm_maskz_roundscale_ss(__mmask8 k, __m128 a, __m128 b, const int imm);
```

Rounds the lower float32 element in *a* to the number of fraction bits specified by *imm*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

_mm512_scalef_pd

```
extern __m512d __cdecl  _mm512_scalef_pd(__m512d a, __m512d b);
```

Performs a floating point scale of the packed float64 elements in source *a* by multiplying by 2^b, and stores the result.

_mm512_mask_scalef_pd

```
extern __m512d __cdecl _mm512_mask_scalef_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Performs a floating point scale of the packed float64 elements in source *a* by multiplying by 2^b, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_scalef_pd

```
extern __m512d __cdecl _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
```

Performs a floating point scale of the packed float64 elements in source *a* by multiplying by 2^b, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_scalef_round_pd

```
extern __m512d __cdecl _mm512_scalef_round_pd(__m512d a, __m512d b, int round);
```

Performs a floating point scale of the rounded packed float64 elements in source *a* by multiplying by 2^b, and stores the result.

_mm512_mask_scalef_round_pd

```
extern __m512d __cdecl _mm512_mask_scalef_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d
b, int round);
```

Performs a floating point scale of the rounded packed float64 elements in source *a* by multiplying by 2^b, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_scalef_round_pd

```
extern __m512d __cdecl _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Performs a floating point scale of the rounded packed float64 elements in source *a* by multiplying by 2^b, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_scalef_ps

```
extern __m512 __cdecl _mm512_scalef_ps(__m512 a, __m512 b);
```

Performs a floating point scale of the packed float32 elements in source *a* by multiplying by 2^b, and stores the result.

_mm512_mask_scalef_ps

```
extern __m512 __cdecl _mm512_mask_scalef_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Performs a floating point scale of the packed float32 elements in source *a* by multiplying by 2^b, and stores the result, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_scalef_ps

```
extern __m512 __cdecl _mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
```

Performs a floating point scale of the packed single-precision (64-bit) floating-point elements in source *a* by multiplying by 2^b, and stores the result, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_scalef_round_ps

```
extern __m512 __cdecl _mm512_scalef_round_ps(__m512 a, __m512 b, int round);
```

Performs a floating point scale of the rounded packed single-precision (64-bit) floating-point elements in source *a* by multiplying by 2^b, and stores the results.

_mm512_mask_scalef_round_ps

```
extern __m512 __cdecl _mm512_mask_scalef_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b,
int round);
```

Performs a floating point scale of the rounded packed single-precision (64-bit) floating-point elements in source *a* by multiplying by 2^b, and stores the results, using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_scalef_round_ps

```
extern __m512 __cdecl _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Performs a floating point scale of the rounded packed single-precision (64-bit) floating-point elements in source *a* by multiplying by 2^b, and stores the results, using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_scalef_round_sd

```
extern __m128d __cdecl _mm_scalef_round_sd(__m128d a, __m128d b, int round);
```

Performs a floating point scale of the rounded scalar float64 elements in source *a* by multiplying by 2^b, stores the result in the lower destination element, and copies the upper element from *b* to the upper destination element.

_mm_mask_scalef_round_sd

```
extern __m128d __cdecl _mm_mask_scalef_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b,
int round);
```

Performs a floating point scale of the rounded scalar float64 elements in source *a* by multiplying by 2^b, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_maskz_scalef_round_sd

```
extern __m128d __cdecl _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Performs a floating point scale of the rounded scalar float64 elements in source *a* by multiplying by 2^b, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_scalef_sd

```
extern __m128d __cdecl _mm_scalef_sd(__m128d a, __m128d b);
```

Performs a floating point scale of the scalar float64 elements in source *a* using values from *b*, stores the result in the lower destination element, and copies the upper element from *b* to the upper destination element.

_mm_mask_scalef_sd

```
extern __m128d __cdecl _mm_mask_scalef_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Performs a floating point scale of the scalar float64 elements in source *a* using values from *b*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_maskz_scalef_sd

```
extern __m128d __cdecl _mm_maskz_scalef_sd(__mmask8 k, __m128d a, __m128d b);
```

Performs a floating point scale of the scalar float64 elements in source *a* using values from *b*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element.

_mm_scalef_round_ss

```
extern __m128 __cdecl _mm_scalef_round_ss(__m128 a, __m128 b, int  round);
```

Performs a floating point scale of the rounded scalar float32 elements in source *a* by multiplying by 2^b, stores the result in the lower destination element, and copies the upper element from *b* to the upper destination elements.

_mm_mask_scalef_round_ss

```
extern __m128 __cdecl _mm_mask_scalef_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Performs a floating point scale of the rounded scalar float32 elements in source *a* by multiplying by 2^b, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

_mm_maskz_scalef_round_ss

```
extern __m128 __cdecl _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int  round);
```

Performs a floating point scale of the rounded scalar float32 elements in source *a* by multiplying by 2^b, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

_mm_scalef_ss

```
extern __m128 __cdecl _mm_scalef_ss(__m128 a, __m128 b);
```

Performs a floating point scale of the scalar float32 elements in source *a* using values from *b*, stores the result in the lower destination element, and copies the upper three packed elements from *b* to the upper destination elements.

_mm_mask_scalef_ss

```
extern __m128 __cdecl _mm_mask_scalef_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Performs a floating point scale of the scalar float32 elements in source *a* using values from *b*, stores the result in the lower destination element, using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

_mm_maskz_scalef_ss

```
extern __m128 __cdecl _mm_maskz_scalef_ss(__mmask8 k, __m128 a, __m128 b);
```

Performs a floating point scale of the scalar float32 elements in source *a* using values from *b*, stores the result in the lower destination element, using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements.

Intrinsics for Blend Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

NOTE
The opmask register is not used as a writemask for these instructions. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit ('0' for the first source operand, '1' for the second source operand), the elements with corresponding mask bit value of '0' in the destination operand are zeroed.

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_mask_blend_pd`	Blend float64 vector elements using instruction mask.	`VBLENDMPD`
`_mm512_mask_blend_ps`	Blend float32 vector elements using instruction mask.	`VBLENDMPS`
`_mm512_mask_blend_epi32`	Blend int32 vectors using instruction mask.	`VPBLENDMD`
`_mm512_mask_blend_epi64`	Blend int64 vectors using instruction mask.	`VPBLENDMQ`

variable	definition
k	instruction mask used as a selector
a	first source vector element
b	second source vector element

_mm512_mask_blend_pd

```
extern m512d __cdecl _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
```

Performs element-by-element blending of float64 source vectors *a* and *b*, using the instruction mask *k* as selector.

The result is written into float64 vector destination register.

_mm512_mask_blend_ps

```
extern m512 __cdecl _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
```

Performs element-by-element blending of float32 source vectors *a* and *b*, using the instruction mask *k* as selector.

The result is written into an float32 vector register.

_mm512_mask_blend_epi32

```
extern m512i __cdecl _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
```

Performs element-by-element blending of int32 source vectors *a* and *b*, using the instruction mask *k* as selector.

The result is written into an int32 vector register.

_mm512_mask_blend_epi64

```
extern m512i __cdecl _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
```

Performs element-by-element blending of int64 source vectors *a* and *b*, using the instruction mask *k* as selector.

The result is written into an int64 vector register.

Intrinsics for Bit Manipulation Operations

Intrinsics for Integer Bit Manipulation and Conflict Detection Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_lzcnt_epi32`, `_mm512_mask_lzcnt_epi32`, `_mm512_maskz_lzcnt_epi32`	Counts the leading zero bits in source int32 elements.	`VPLZCNTD`
`_mm512_lzcnt_epi64`, `_mm512_mask_lzcnt_epi64`, `_mm512_maskz_lzcnt_epi64`	Counts the leading zero bits in source int64 elements.	`VPLZCNTQ`
`_mm512_ternarylogic_epi32`, `_mm512_mask_ternarylogic_epi32`, `_mm512_maskz_ternarylogic_epi32`	Implements three-operand binary function specified by immediate value.	`VPTERNLOGD`
`_mm512_ternarylogic_epi64`, `_mm512_mask_ternarylogic_epi64`, `_mm512_maskz_ternarylogic_epi64`	Implements three-operand binary function specified by immediate value.	`VPTERNLOGQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
c	third source vector element
imm8	binary function specifier
src	source element to use based on writemask result

_mm512_lzcnt_epi32

```
extern __m512i __cdecl _mm512_lzcnt_epi32(__m512i a);
```

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and store the results in destination.

_mm512_mask_lzcnt_epi32

```
extern __m512i __cdecl _mm512_mask_lzcnt_epi32(__m512i src, __mmask16 k, __m512i a);
```

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and store the results in destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_lzcnt_epi32

```
extern __m512i __cdecl _mm512_maskz_lzcnt_epi32(__mmask16 k, __m512i a);
```

Counts the number of leading zero bits in each packed 32-bit integer in *a*, and store the results in destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_lzcnt_epi64

```
extern __m512i __cdecl _mm512_lzcnt_epi64(__m512i a);
```

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and store the results.

_mm512_mask_lzcnt_epi64

```
extern __m512i __cdecl _mm512_mask_lzcnt_epi64(__m512i src, __mmask8 k, __m512i a);
```

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and store the results in using writemask *k*.

Elements are copied from *src* when the corresponding mask bit is not set.

_mm512_maskz_lzcnt_epi64

```
extern __m512i __cdecl _mm512_maskz_lzcnt_epi64(__mmask8 k, __m512i a);
```

Counts the number of leading zero bits in each packed 64-bit integer in *a*, and store the results in destination using zeromask *k*.

Elements are zeroed out when the corresponding mask bit is not set.

_mm512_ternarylogic_epi32

```
extern __m512i __cdecl _mm512_ternarylogic_epi32(__m512i a, __m512i b, __m512i c, int imm8);
```

Bitwise ternary logic to implement three-operand binary functions; the specific binary function is specified by value in *imm8*.

For each bit in each packed 32-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding destination bit.

_mm512_mask_ternarylogic_epi32

```
extern __m512i __cdecl _mm512_mask_ternarylogic_epi32(__m512i a, __mmask16 k, __m512i, __m512i
b, int imm8);
```

Bitwise ternary logic to implement three-operand binary functions; the specific binary function is specified by value in *imm8*.

For each bit in each packed 32-bit integer, the corresponding bit from *src*, *a*, and *b* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding destination bit using writemask *k* at 32-bit granularity (32-bit elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_ternarylogic_epi32

```
extern __m512i __cdecl _mm512_maskz_ternarylogic_epi32(__mmask16 k, __m512i a, __m512i b,
__m512i c, int imm8);
```

Bitwise ternary logic to implement three-operand binary functions; the specific binary function is specified by value in *imm8*.

For each bit in each packed 32-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding destination bit using zeromask *k* at 32-bit granularity (32-bit elements are zeroed out when the corresponding mask bit is not set).

_mm512_ternarylogic_epi64

```
extern __m512i __cdecl _mm512_ternarylogic_epi64(__m512i a, __m512i b, __m512i c, int imm8);
```

Bitwise ternary logic to implement three-operand binary functions; the specific binary function is specified by value in *imm8*.

For each bit in each packed 64-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3-bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding destination bit.

_mm512_mask_ternarylogic_epi64

```
extern __m512i __cdecl _mm512_mask_ternarylogic_epi64(__m512i src, __mmask8 k, __m512i a,
__m512i b, int imm8);
```

Bitwise ternary logic to implement three-operand binary functions; the specific binary function is specified by value in *imm8*.

For each bit in each packed 64-bit integer, the corresponding bit from *src*, *a*, and *b* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding destination bit using writemask *k* at 64-bit granularity (64-bit elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_ternarylogic_epi64

```
extern __m512i __cdecl _mm512_maskz_ternarylogic_epi64(__mmask8 k, __m512i a, __m512i b, __m512i
c, int imm8);
```

Bitwise ternary logic to implement three-operand binary functions; the specific binary function is specified by value in *imm8*.

For each bit in each packed 64-bit integer, the corresponding bit from *a*, *b*, and *c* are used to form a 3 bit index into *imm8*, and the value at that bit in *imm8* is written to the corresponding destination bit using zeromask *k* at 64-bit granularity (64-bit elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Bitwise Logical Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_and_epi32, _mm512_mask_and_epi32_mm512_maskz_and_epi32	Computes the bitwise AND of packed int32 elements.	VPANDD
_mm512_and_epi64, _mm512_mask_and_epi64, _mm512_maskz_and_epi64	Computes the bitwise AND of packed int64 elements.	VPANDQ
_mm512_or_epi32, _mm512_mask_or_epi32, _mm512_maskz_or_epi32	Computes the bitwise OR of packed int32 elements.	VPORD
_mm512_or_epi64, _mm512_mask_or_epi64, _mm512_maskz_or_epi64	Computes the bitwise OR of packed int64 elements.	VPORQ
_mm512_andnot_epi32, _mm512_mask_andnot_epi32, _mm512_maskz_andnot_epi32	Computes the bitwise AND NOT of packed int32 elements.	VPANDND
_mm512_andnot_epi64, _mm512_mask_andnot_epi64, _mm512_maskz_andnot_epi64	Computes the bitwise AND NOT of packed int64 elements.	VPANDNQ
_mm512_xor_epi32, _mm512_mask_xor_epi32, _mm512_maskz_xor_epi32	Computes the bitwise XOR of packed int 32 elements.	VPXORD
_mm512_xor_epi64, _mm512_mask_xor_epi64, _mm512_maskz_xor_epi64	Computes the bitwise XOR of packed int64 elements.	VPXORQ

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_and_epi32

```
extern __m512i __cdecl _mm512_and_epi32(__m512i a, __m512i b);
```

Computes the bitwise AND of packed 32-bit integers in *a* and *b*, and stores the result.

_mm512_mask_and_epi32

```
extern __m512i __cdecl _mm512_mask_and_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise AND of packed 32-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_and_epi32

```
extern __m512i __cdecl _mm512_maskz_and_epi32(__mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise AND of packed 32-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_and_epi64

```
extern __m512i __cdecl _mm512_and_epi64(__m512i a, __m512i b);
```

Computes the bitwise AND of 512 bits (composed of packed 64-bit integers) in *a* and *b*, and stores the result.

_mm512_mask_and_epi64

```
extern __m512i __cdecl _mm512_mask_and_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise AND of packed 64-bit integers in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_and_epi64

```
extern __m512i __cdecl _mm512_maskz_and_epi64(__mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise AND of packed 64-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_andnot_epi32

```
extern __m512i __cdecl _mm512_andnot_epi32(__m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 32-bit integers in *a* and *b*, and stores the result.

_mm512_mask_andnot_epi32

```
extern __m512i __cdecl _mm512_mask_andnot_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 32-bit integers in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_andnot_epi32

```
extern __m512i __cdecl _mm512_maskz_andnot_epi32(__mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 32-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_andnot_epi64

```
extern __m512i __cdecl _mm512_andnot_epi64(__m512i a, __m512i b);
```

Computes the bitwise AND NOT of 512 bits (composed of packed 64-bit integers) in *a* and *b*, and stores the result.

_mm512_mask_andnot_epi64

```
extern __m512i __cdecl _mm512_mask_andnot_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 64-bit integers in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_andnot_epi64

```
extern __m512i __cdecl _mm512_maskz_andnot_epi64(__mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise `AND NOT` of packed 64-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_or_epi32

```
extern __m512i __cdecl _mm512_or_epi32(__m512i a, __m512i b);
```

Computes the bitwise `OR` of packed 32-bit integers in *a* and *b*, and stores the result.

_mm512_mask_or_epi32

```
extern __m512i __cdecl _mm512_mask_or_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise `OR` of packed 32-bit integers in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_or_epi32

```
extern __m512i __cdecl _mm512_maskz_or_epi32( __mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise `OR` of packed 32-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_or_epi64

```
extern __m512i __cdecl _mm512_or_epi64(__m512i a, __m512i b);
```

Computes the bitwise `OR` of packed 64-bit integers in *a* and *b*, and store the result.

_mm512_mask_or_epi64

```
extern __m512i __cdecl _mm512_mask_or_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise `OR` of packed 64-bit integers in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_or_epi64

```
extern __m512i __cdecl _mm512_maskz_or_epi64(__mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise `OR` of packed 64-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_xor_epi32

```
extern __m512i __cdecl _mm512_xor_epi32(__m512i a, __m512i b);
```

Computes the bitwise `XOR` of packed 32-bit integers in *a* and *b*, and stores the result.

_mm512_mask_xor_epi32

```
extern __m512i __cdecl _mm512_mask_xor_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise `XOR` of packed 32-bit integers in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_xor_epi32

```
extern __m512i __cdecl _mm512_maskz_xor_epi32(__mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise XOR of packed 32-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_xor_epi64

```
extern __m512i __cdecl _mm512_xor_epi64(__m512i a, __m512i b);
```

Computes the bitwise XOR of packed 64-bit integers in *a* and *b*, and stores the result.

_mm512_mask_xor_epi64

```
extern __m512i __cdecl _mm512_mask_xor_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise XOR of packed 64-bit integers in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_xor_epi64

```
extern __m512i __cdecl _mm512_maskz_xor_epi64(__mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise XOR of packed 64-bit integers in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Integer Bit Rotation Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_rol_epi32, _mm512_mask_rol_epi32, _mm512_maskz_rol_epi32	Rotates bits of int32 source elements left by specified count.	VPROLD
_mm512_rol_epi64, _mm512_mask_rol_epi64, _mm512_maskz_rol_epi64	Rotates bits of int64 source elements left by specified count.	VPROLQ
_mm512_rolv_epi32, _mm512_mask_rolv_epi32, _mm512_maskz_rolv_epi32	Rotates bits of int32 source elements left by specified count.	VPROLVD
_mm512_rolv_epi64, _mm512_mask_rolv_epi64, _mm512_maskz_rolv_epi64	Rotates bits of int64 source elements left by specified count.	VPROLVQ
_mm512_ror_epi32, _mm512_mask_ror_epi32, _mm512_maskz_ror_epi32	Rotates bits of int32 source elements right by specified count.	VPRORD
_mm512_ror_epi64, _mm512_mask_ror_epi64, _mm512_maskz_ror_epi64	Rotates bits of int64 source elements right by specified count.	VPRORQ

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_rorv_epi32`, `_mm512_mask_rorv_epi32`, `_mm512_maskz_rorv_epi32`	Rotates bits of int32 source elements right by specified count.	`VPRORVD`
`_mm512_rorv_epi64`, `_mm512_mask_rorv_epi64`, `_mm512_maskz_rorv_epi64`	Rotates bits of int64 source elements right by specified count.	`VPRORVQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
imm	8-bit immediate integer specifies offset for destination

_mm512_rol_epi32

```
extern __m512i __cdecl _mm512_rol_epi32(__m512i a, const int imm);
```

Rotates bits in each packed int32 element in *a* to the left by the number of bits specified in *imm*, and stores the results.

_mm512_mask_rol_epi32

```
extern __m512i __cdecl _mm512_mask_rol_epi32(__m512i src, __mmask16 k, __m512i a, const int imm);
```

Rotates bits in each packed int32 element in *a* to the left by the number of bits specified in *imm*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_rol_epi32

```
extern __m512i __cdecl _mm512_maskz_rol_epi32(__mmask16 k, __m512i a, const int imm);
```

Rotates bits in each packed int32 element in *a* to the left by the number of bits specified in *imm*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_rol_epi64

```
extern __m512i __cdecl _mm512_rol_epi64(__m512i a, const int imm);
```

Rotates bits in each packed int64 element in *a* to the left by the number of bits specified in *imm*, and stores the results.

_mm512_mask_rol_epi64

```
extern __m512i __cdecl _mm512_mask_rol_epi64(__m512i src, __mmask8 k, __m512i a, const int imm);
```

Rotates bits in each packed int64 element in *a* to the left by the number of bits specified in *imm*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_rol_epi64

```
extern __m512i __cdecl _mm512_maskz_rol_epi64(__mmask8 k, __m512i a, const int imm);
```

Rotates bits in each packed int64 element in *a* to the left by the number of bits specified in *imm*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_rolv_epi32

```
extern __m512i __cdecl _mm512_rolv_epi32(__m512i a, __m512i b);
```

Rotates bits in each packed int32 element in *a* to the left by the number of bits specified in the corresponding element of *b*, and stores the results.

_mm512_mask_rolv_epi32

```
extern __m512i __cdecl _mm512_mask_rolv_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Rotates bits in each packed int32 element in *a* to the left by the number of bits specified in the corresponding element of *b*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_rolv_epi32

```
extern __m512i __cdecl _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i b);
```

Rotates bits in each packed int32 element in *a* to the left by the number of bits specified in the corresponding element of *b*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_rolv_epi64

```
extern __m512i __cdecl _mm512_rolv_epi64(__m512i a, __m512i b);
```

Rotates bits in each packed int64 element in *a* to the left by the number of bits specified in the corresponding element of *b*, and stores the results.

_mm512_mask_rolv_epi64

```
extern __m512i __cdecl _mm512_mask_rolv_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Rotates bits in each packed int64 element in *a* to the left by the number of bits specified in the corresponding element of *b*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_rolv_epi64

```
extern __m512i __cdecl _mm512_maskz_rolv_epi64(__mmask8 k, __m512i a, __m512i b);
```

Rotates bits in each packed int64 element in *a* to the left by the number of bits specified in the corresponding element of *b*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_ror_epi32

```
extern __m512i __cdecl _mm512_ror_epi32(__m512i a, int imm);
```

Rotates bits in each packed int32 element in *a* to the right by the number of bits specified in *imm*, and stores the results.

_mm512_mask_ror_epi32

```
extern __m512i __cdecl _mm512_mask_ror_epi32(__m512i src, __mmask16 k, __m512i a, int imm);
```

Rotates bits in each packed int32 element in *a* to the right by the number of bits specified in *imm*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_ror_epi32

```
extern __m512i __cdecl _mm512_maskz_ror_epi32(__mmask16 k, __m512i a, int imm);
```

Rotates bits in each packed int32 element in *a* to the right by the number of bits specified in *imm*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_ror_epi64

```
extern __m512i __cdecl _mm512_ror_epi64(__m512i a, int imm);
```

Rotates bits in each packed int64 element in *a* to the right by the number of bits specified in *imm*, and stores the results.

_mm512_mask_ror_epi64

```
extern __m512i __cdecl _mm512_mask_ror_epi64(__m512i src, __mmask8 k, __m512i a, int imm);
```

Rotates bits in each packed int64 element in *a* to the right by the number of bits specified in *imm*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_ror_epi64

```
extern __m512i __cdecl _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
```

Rotates bits in each packed int64 element in *a* to the right by the number of bits specified in *imm*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_rorv_epi32

```
extern __m512i __cdecl _mm512_rorv_epi32(__m512i a, __m512i b);
```

Rotates bits in each packed int32 element in *a* to the right by the number of bits specified in the corresponding element of *b*, and stores the results.

_mm512_mask_rorv_epi32

```
extern __m512i __cdecl _mm512_mask_rorv_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b);
```

Rotates bits in each packed int32 element in *a* to the right by the number of bits specified in the corresponding element of *b*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_rorv_epi32

```
extern __m512i __cdecl _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i b);
```

Rotates bits in each packed int32 element in *a* to the right by the number of bits specified in the corresponding element of *b*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_rorv_epi64

```
extern __m512i __cdecl _mm512_rorv_epi64(__m512i a, __m512i b);
```

Rotates bits in each packed int64 element in *a* to the right by the number of bits specified in the corresponding element of *b*, and stores the results.

_mm512_mask_rorv_epi64

```
extern __m512i __cdecl _mm512_mask_rorv_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Rotates bits in each packed int64 element in *a* to the right by the number of bits specified in the corresponding element of *b*, and stores the results using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_rorv_epi64

```
extern __m512i __cdecl _mm512_maskz_rorv_epi64(__mmask8 k, __m512i a, __m512i b);
```

Rotates bits in each packed int64 element in *a* to the right by the number of bits specified in the corresponding element of *b*, and stores the results using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Integer Bit Shift Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_sll_epi32`, `_mm512_mask_sll_epi32`, `_mm512_maskz_sll_epi32` `_mm512_slli_epi32`, `_mm512_mask_slli_epi32`, `_mm512_maskz_slli_epi32`	Logical left shift of int32 elements.	`VPSLLD`
`_mm512_srl_epi32`, `_mm512_mask_srl_epi32`, `_mm512_maskz_srl_epi32` `_mm512_srli_epi32`, `_mm512_mask_srli_epi32`, `_mm512_maskz_srli_epi32`	Logical right shift of int32 elements.	`VPSRLD`
`_mm512_sll_epi64`, `_mm512_mask_sll_epi64`, `_mm512_maskz_sll_epi64` `_mm512_slli_epi64`, `_mm512_mask_slli_epi64`, `_mm512_maskz_slli_epi64`	Logical left shift of int64 elements.	`VPSLLQ`
`_mm512_srl_epi64`, `_mm512_mask_srl_epi64`, `_mm512_maskz_srl_epi64`	Logical right shift of int64 elements.	`VPSRLQ`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_srli_epi64, _mm512_mask_srli_epi64, _mm512_maskz_srli_epi64		
_mm512_sllv_epi32, _mm512_mask_sllv_epi32, _mm512_maskz_sllv_epi32	Variable logical left shift of int32 elements.	VPSLLVD
_mm512_srlv_epi32, _mm512_mask_srlv_epi32, _mm512_maskz_srlv_epi32	Variable logical right shift of int32 elements.	VPSRLVD
_mm512_sllv_epi64, _mm512_mask_sllv_epi64, _mm512_maskz_sllv_epi64	Variable logical bit shift left of int64 elements.	VPSLLVQ
_mm512_srlv_epi64, _mm512_mask_srlv_epi64, _mm512_maskz_srlv_epi64	Variable logical bit shift right of int64 elements.	VPSRLVQ
_mm512_sra_epi32, _mm512_mask_sra_epi32, _mm512_maskz_sra_epi32	Arithmetic right shift of int32 elements.	VPSRAD
_mm512_srai_epi32, _mm512_mask_srai_epi32, _mm512_maskz_srai_epi32		
_mm512_srav_epi32, _mm512_mask_srav_epi32, _mm512_maskz_srav_epi32	Variable arithmetic right shift of int32 elements.	VPSRAVD
_mm512_srav_epi64, _mm512_mask_srav_epi64, _mm512_maskz_srav_epi64	Variable arithmetic bit shift right of int64 elements.	VPSRAVQ
_mm512_sra_epi64, _mm512_mask_sra_epi64, _mm512_maskz_sra_epi64	Arithmetic right shift of int64 elements.	VPSRAQ
_mm512_srai_epi64, _mm512_mask_srai_epi64, _mm512_maskz_srai_epi64		

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result
count	specifies the number of bits for shift operation

variable	definition
imm	8-bit immediate integer specifies offset for destination

_mm512_sll_epi32

```
extern __m512i __cdecl _mm512_sll_epi32(__m512i a, __m128i count);
```

Shifts packed int32 elements in *a* left by *count* while shifting in zeros, and stores the result.

_mm512_mask_sll_epi32

```
extern __m512i __cdecl _mm512_mask_sll_epi32(__m512i src, __mmask16 k, __m512i a, __m128i count);
```

Shifts packed int32 elements in *a* left by *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sll_epi32

```
extern __m512i __cdecl _mm512_maskz_sll_epi32(__mmask16 k, __m512i a, __m128i count);
```

Shifts packed int32 elements in *a* left by *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_slli_epi32

```
extern __m512i __cdecl _mm512_slli_epi32(__m512i a, unsigned int imm);
```

Shifts packed int32 elements in *a* left by *imm* while shifting in zeros, and stores the result.

_mm512_mask_slli_epi32

```
extern __m512i __cdecl _mm512_mask_slli_epi32(__m512i src, __mmask16 k, __m512i a, unsigned int
imm);
```

Shifts packed int32 elements in *a* left by *imm* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_slli_epi32

```
extern __m512i __cdecl _mm512_maskz_slli_epi32(__mmask16 k, __m512i a, unsigned int imm);
```

Shifts packed int32 elements in *a* left by *imm* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sll_epi64

```
extern __m512i __cdecl _mm512_sll_epi64(__m512i a, __m128i count);
```

Shifts packed int64 elements in *a* left by *count* while shifting in zeros, and stores the result.

_mm512_mask_sll_epi64

```
extern __m512i __cdecl _mm512_mask_sll_epi64(__m512i src, __mmask8 k, __m512i a, __m128i count);
```

Shifts packed int64 elements in *a* left by *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sll_epi64

```
extern __m512i __cdecl _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i count);
```

Shifts packed int64 elements in *a* left by *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_slli_epi64

```
extern __m512i __cdecl _mm512_slli_epi64(__m512i a, unsigned int imm);
```

Shifts packed int64 elements in *a* left by *imm* while shifting in zeros, and stores the result.

_mm512_mask_slli_epi64

```
extern __m512i __cdecl _mm512_mask_slli_epi64(__m512i src, __mmask8 k, __m512i a, unsigned int
imm);
```

Shifts packed int64 elements in *a* left by *imm* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_slli_epi64

```
extern __m512i __cdecl _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
```

Shifts packed int64 elements in *a* left by *imm* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sllv_epi32

```
extern __m512i __cdecl _mm512_sllv_epi32(__m512i a, __m512i count);
```

Shifts packed int32 elements in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result.

_mm512_mask_sllv_epi32

```
extern __m512i __cdecl _mm512_mask_sllv_epi32(__m512i src, __mmask16 k, __m512i a, __m512i
count);
```

Shifts packed int32 elements in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sllv_epi32

```
extern __m512i __cdecl _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i count);
```

Shifts packed int32 elements in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sllv_epi64

```
extern __m512i __cdecl _mm512_sllv_epi64(__m512i a, __m512i count);
```

Shifts packed int64 elements in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result.

_mm512_mask_sllv_epi64

```
extern __m512i __cdecl _mm512_mask_sllv_epi64(__m512i src, __mmask8 k, __m512i a, __m512i count);
```

Shifts packed int64 elements in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sllv_epi64

```
extern __m512i __cdecl _mm512_maskz_sllv_epi64(__mmask8 k, __m512i a, __m512i count);
```

Shifts packed int64 elements in *a* left by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sra_epi32

```
extern __m512i __cdecl _mm512_sra_epi32(__m512i a, __m128i count);
```

Shifts packed int32 elements in *a* right by *count* while shifting in sign bits, and stores the result.

_mm512_mask_sra_epi32

```
extern __m512i __cdecl _mm512_mask_sra_epi32(__m512i src, __mmask16 k, __m512i a, __m128i count);
```

Shifts packed int32 elements in *a* right by *count* while shifting in sign bits, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sra_epi32

```
extern __m512i __cdecl _mm512_maskz_sra_epi32(__mmask16 k, __m512i a, __m128i count);
```

Shifts packed int32 elements in *a* right by *count* while shifting in sign bits, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_sra_epi64

```
extern __m512i __cdecl _mm512_sra_epi64(__m512i a, __m128i count);
```

Shifts packed int64 elements in *a* right by *count* while shifting in sign bits, and stores the result.

_mm512_mask_sra_epi64

```
extern __m512i __cdecl _mm512_mask_sra_epi64(__m512i src, __mmask8 k, __m512i a, __m128i count);
```

Shifts packed int64 elements in *a* right by *count* while shifting in sign bits, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_sra_epi64

```
extern __m512i __cdecl _mm512_maskz_sra_epi64(__mmask8 k, __m512i a, __m128i count);
```

Shifts packed int64 elements in *a* right by *count* while shifting in sign bits, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srai_epi32

```
extern __m512i __cdecl _mm512_srai_epi32(__m512i a, unsigned int imm);
```

Shifts packed int32 elements in *a* right by *imm* while shifting in sign bits, and stores the result.

_mm512_mask_srai_epi32

```
extern __m512i __cdecl _mm512_mask_srai_epi32(__m512i src, __mmask16 k, __m512i a, unsigned int
imm);
```

Shifts packed int32 elements in *a* right by *imm* while shifting in sign bits, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srai_epi32

```
extern __m512i __cdecl _mm512_maskz_srai_epi32(__mmask16 k, __m512i a, unsigned int imm);
```

Shifts packed int32 elements in *a* right by *imm* while shifting in sign bits, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srai_epi64

```
extern __m512i __cdecl _mm512_srai_epi64(__m512i a, unsigned int imm);
```

Shifts packed int64 elements in *a* right by *imm* while shifting in sign bits, and stores the result.

_mm512_mask_srai_epi64

```
extern __m512i __cdecl _mm512_mask_srai_epi64(__m512i src, __mmask8 k, __m512i a, unsigned int
imm);
```

Shifts packed int64 elements in *a* right by *imm* while shifting in sign bits, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srai_epi64

```
extern __m512i __cdecl _mm512_maskz_srai_epi64(__mmask8 k, __m512i a, unsigned int imm);
```

Shifts packed int64 elements in *a* right by *imm* while shifting in sign bits, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srav_epi32

```
extern __m512i __cdecl _mm512_srav_epi32(__m512i a, __m512i count);
```

Shifts packed int32 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and stores the result.

_mm512_mask_srav_epi32

```
extern __m512i __cdecl _mm512_mask_srav_epi32(__m512i src, __mmask16 k, __m512i a, __m512i
count);
```

Shifts packed int32 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srav_epi32

```
extern __m512i __cdecl _mm512_maskz_srav_epi32(__mmask16 k, __m512i a, __m512i count);
```

Shifts packed int32 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srav_epi64

```
extern __m512i __cdecl _mm512_srav_epi64(__m512i a, __m512i count);
```

Shifts packed int64 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and stores the result.

_mm512_mask_srav_epi64

```
extern __m512i __cdecl _mm512_mask_srav_epi64(__m512i src, __mmask8 k, __m512i a, __m512i count);
```

Shifts packed int64 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srav_epi64

```
extern __m512i __cdecl _mm512_maskz_srav_epi64(__mmask8 k, __m512i a, __m512i count);
```

Shifts packed int64 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in sign bits, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srl_epi32

```
extern __m512i __cdecl _mm512_srl_epi32(__m512i a, __m128i count);
```

Shifts packed int32 elements in *a* right by *count* while shifting in zeros, and stores the result.

_mm512_mask_srl_epi32

```
extern __m512i __cdecl _mm512_mask_srl_epi32(__m512i src, __mmask16 k, __m512i a, __m128i count);
```

Shifts packed int32 elements in *a* right by *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srl_epi32

```
extern __m512i __cdecl _mm512_maskz_srl_epi32(__mmask16 k, __m512i a, __m128i count);
```

Shifts packed int32 elements in *a* right by *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srli_epi32

```
extern __m512i __cdecl _mm512_srli_epi32(__m512i a, unsigned int imm);
```

Shifts packed int32 elements in *a* right by *imm* while shifting in zeros, and stores the result.

_mm512_mask_srli_epi32

```
extern __m512i __cdecl _mm512_mask_srli_epi32(__m512i src, __mmask16 k, __m512i a, unsigned int
imm);
```

Shifts packed int32 elements in *a* right by *imm* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srli_epi32

```
extern __m512i __cdecl _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, unsigned int imm);
```

Shifts packed int32 elements in *a* right by *imm* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srl_epi64

```
extern __m512i __cdecl _mm512_srl_epi64(__m512i a, __m128i count);
```

Shifts packed int64 elements in *a* right by *count* while shifting in zeros, and stores the result.

_mm512_mask_srl_epi64

```
extern __m512i __cdecl _mm512_mask_srl_epi64(__m512i src, __mmask8 k, __m512i a, __m128i count);
```

Shifts packed int64 elements in *a* right by *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srl_epi64

```
extern __m512i __cdecl _mm512_maskz_srl_epi64(__mmask8 k, __m512i a, __m128i count);
```

Shifts packed int64 elements in *a* right by *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srli_epi64

```
extern __m512i __cdecl _mm512_srli_epi64(__m512i a, unsigned int imm);
```

Shifts packed int64 elements in *a* right by *imm* while shifting in zeros, and stores the result.

_mm512_mask_srli_epi64

```
extern __m512i __cdecl _mm512_mask_srli_epi64(__m512i src, __mmask8 k, __m512i a, unsigned int
imm);
```

Shifts packed int64 elements in *a* right by *imm* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srli_epi64

```
extern __m512i __cdecl _mm512_maskz_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
```

Shifts packed int64 elements in *a* right by *imm* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srlv_epi32

```
extern __m512i __cdecl _mm512_srlv_epi32(__m512i a, __m512i count);
```

Shifts packed int32 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result.

_mm512_mask_srlv_epi32

```
extern __m512i __cdecl _mm512_mask_srlv_epi32(__m512i src, __mmask16 k, __m512i a, __m512i
count);
```

Shifts packed int32 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srlv_epi32

```
extern __m512i __cdecl _mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i count);
```

Shifts packed int32 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_srlv_epi64

```
extern __m512i __cdecl _mm512_srlv_epi64(__m512i a, __m512i count);
```

Shifts packed int64 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result.

_mm512_mask_srlv_epi64

```
extern __m512i __cdecl _mm512_mask_srlv_epi64(__m512i src, __mmask8 k, __m512i a, __m512i count);
```

Shifts packed int64 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_srlv_epi64

```
extern __m512i __cdecl _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i count);
```

Shifts packed int64 elements in *a* right by the amount specified by the corresponding element in *count* while shifting in zeros, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Broadcast Operations

Intrinsics for FP Broadcast Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_broadcast_f32x4`, `_mm512_mask_broadcast_f32x4`, `_mm512_maskz_broadcast_f32x4`	Broadcast float32 element to four destination locations.	`VBROADCASTF32X4`
`_mm512_broadcast_f64x4`, `_mm512_mask_broadcast_f64x4`, `_mm512_maskz_broadcast_f64x4`	Broadcast float64 element to four destination locations.	`VBROADCASTF64X4`
`_mm512_broadcastsd_pd`, `_mm512_mask_broadcastsd_pd`, `_mm512_maskz_broadcastsd_pd`	Broadcast packed float64 element to all destination locations.	`VBROADCASTSD`
`_mm512_broadcastss_ps`, `_mm512_mask_broadcastss_ps`, `_mm512_maskz_broadcastss_ps`	Broadcast packed float32 element to all destination locations.	`VBROADCASTSS`

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result

_mm512_broadcast_f32x4

```
extern __m512 __cdecl _mm512_broadcast_f32x4(__m128 a);
```

Broadcasts four packed float32 elements from *a* to all destination elements.

_mm512_mask_broadcast_f32x4

```
extern __m512 __cdecl _mm512_mask_broadcast_f32x4(__m512 src, __mmask16 k, __m128 a);
```

Broadcasts four packed float32 elements from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_f32x4

```
extern __m512 __cdecl _mm512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
```

Broadcasts four packed float32 elements from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_f64x4

```
extern __m512d __cdecl _mm512_broadcast_f64x4(__m256d a);
```

Broadcasts four packed float64 elements from *a* to all destination elements.

_mm512_mask_broadcast_f64x4

```
extern __m512d __cdecl _mm512_mask_broadcast_f64x4(__m512d src, __mmask8 k, __m256d a);
```

Broadcasts four packed float64 elements from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_f64x4

```
extern __m512d __cdecl _mm512_maskz_broadcast_f64x4(__mmask8 k, __m256d a);
```

Broadcasts four packed float64 elements from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcastsd_pd

```
extern __m512d __cdecl _mm512_broadcastsd_pd(__m128d a);
```

Broadcasts low float64 element from *a* to all destination elements.

_mm512_mask_broadcastsd_pd

```
extern __m512d __cdecl _mm512_mask_broadcastsd_pd(__m512d src, __mmask8 k, __m128d a);
```

Broadcasts low float64 element from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcastsd_pd

```
extern __m512d __cdecl _mm512_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
```

Broadcasts low float64 element from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcastss_ps

```
extern __m512 __cdecl _mm512_broadcastss_ps(__m128 a);
```

Broadcasts low float32 element from *a* to all destination elements.

_mm512_mask_broadcastss_ps

```
extern __m512 __cdecl _mm512_mask_broadcastss_ps(__m512 src, __mmask16 k, __m128 a);
```

Broadcasts low float32 element from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcastss_ps

```
extern __m512 __cdecl _mm512_maskz_broadcastss_ps(__mmask16 k, __m128 a);
```

Broadcasts low float32 element from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Integer Broadcast Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_broadcast_i32x4`, `_mm512_mask_broadcast_i32x4`, `_mm512_maskz_broadcast_i32x4`	Broadcasts source int32 element to four destinations.	`VBROADCASTI32X4`
`_mm512_broadcast_i64x4`, `_mm512_mask_broadcast_i64x4`, `_mm512_maskz_broadcast_i64x4`	Broadcasts source int64 element to four destinations.	`VBROADCASTI64X4`
`_mm512_broadcastd_epi32`, `_mm512_mask_broadcastd_epi32`, `_mm512_maskz_broadcastd_epi32`	Broadcasts source int32 element to doubleword destinations.	`VPBROADCASTD`
`_mm512_broadcastq_epi64`, `_mm512_mask_broadcastq_epi64`, `_mm512_maskz_broadcastq_epi64`	Broadcasts source int64 element to quadword destinations.	`VPBROADCASTQ`
`_mm512_broadcastmb_epi64`	Broadcasts low byte from input mask to all int64 destination elements.	`VPBROADCASTMB2Q`
`_mm512_broadcastmw_epi32`	Broadcasts low word from input mask to all int32 destination elements.	`VPBROADCASTMD2W`

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result

_mm512_broadcast_i32x4

```
extern __m512i __cdecl _mm512_broadcast_i32x4(__m128i a);
```

Broadcasts four packed int32 elements from *a* to all destination elements.

_mm512_mask_broadcast_i32x4

```
extern __m512i __cdecl _mm512_mask_broadcast_i32x4(__m512i src, __mmask16 k, __m128i a);
```

Broadcasts four packed int32 elements from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_i32x4

```
extern __m512i __cdecl _mm512_maskz_broadcast_i32x4(__mmask16 k, __m128i a);
```

Broadcasts four packed int32 elements from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcast_i64x4

```
extern __m512i __cdecl _mm512_broadcast_i64x4(__m256i a);
```

Broadcasts four packed int64 elements from *a* to all destination elements.

_mm512_mask_broadcast_i64x4

```
extern __m512i __cdecl _mm512_mask_broadcast_i64x4(__m512i src, __mmask8 k, __m256i a);
```

Broadcasts four packed int64 elements from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcast_i64x4

```
extern __m512i __cdecl _mm512_maskz_broadcast_i64x4(__mmask8 k, __m256i a);
```

Broadcasts four packed int64 elements from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcastd_epi32

```
extern __m512i __cdecl _mm512_broadcastd_epi32(__m128i a);
```

Broadcasts low packed 32-bit integer from *a* to all elements.

_mm512_mask_broadcastd_epi32

```
extern __m512i __cdecl _mm512_mask_broadcastd_epi32(__m512i src, __mmask16 k, __m128i a);
```

Broadcasts low packed 32-bit integer from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcastd_epi32

```
extern __m512i __cdecl _mm512_maskz_broadcastd_epi32(__mmask16 k, __m128i a);
```

Broadcasts low packed 32-bit integer from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcastq_epi64

```
extern __m512i __cdecl _mm512_broadcastq_epi64(__m128i a);
```

Broadcasts low packed 64-bit integer from *a* to all destination elements.

_mm512_mask_broadcastq_epi64

```
extern __m512i __cdecl _mm512_mask_broadcastq_epi64(__m512i src, __mmask8 k, __m128i a);
```

Broadcasts low packed 64-bit integer from *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_broadcastq_epi64

```
extern __m512i __cdecl _mm512_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
```

Broadcasts low packed 64-bit integer from *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_broadcastmw_epi32

```
extern __m512i __cdecl _mm512_broadcastmw_epi32(__mmask16 k);
```

Broadcasts low 16-bits from input mask *k* to all 32-bit elements of destination.

_mm512_broadcastmb_epi64

```
extern __m512i __cdecl _mm512_broadcastmb_epi64(__mmask8 k);
```

Broadcasts the low 8-bits from input mask *k* to all 64-bit elements of destination.

Intrinsics for Comparison Operations

Intrinsics for FP Comparison Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_cmp_round_pd_mask`, `_mm512_mask_cmp_round_pd_mask` `_mm512_cmp_pd_mask`, `_mm512_mask_cmp_pd_mask`, `_mm512_cmp_round_pd_mask`,	Compares float64 vector elements based on comparison operand.	`VCMPPD`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_mask_cmp_round_pd_mask, _mm512_cmpeq_pd_mask, _mm512_mask_cmpeq_pd_mask, _mm512_cmple_pd_mask, _mm512_mask_cmple_pd_mask, _mm512_cmplt_pd_mask, _mm512_mask_cmplt_pd_mask, _mm512_cmpneq_pd_mask, _mm512_cmplt_pd_mask, _mm512_cmpneq_pd_mask, _mm512_mask_cmpneq_pd_mask, _mm512_cmpnle_pd_mask, _mm512_mask_cmpnle_pd_mask, _mm512_cmpnlt_pd_mask, _mm512_mask_cmpnlt_pd_mask, _mm512_cmpord_pd_mask, _mm512_mask_cmpord_pd_mask, _mm512_cmpunord_pd_mask, _mm512_mask_cmpunord_pd_mask		
_mm512_cmp_round_ps_mask, _mm512_mask_cmp_round_ps_mask	Compares float32 vector elements based on comparison operand.	VCMPPS
_mm512_cmp_ps_mask, _mm512_mask_cmp_ps_mask, _mm512_cmp_round_ps_mask, _mm512_mask_cmp_round_ps_mask, _mm512_cmpeq_ps_mask, _mm512_mask_cmpeq_ps_mask, _mm512_cmple_ps_mask, _mm512_mask_cmple_ps_mask, _mm512_cmplt_ps_mask, _mm512_mask_cmplt_ps_mask, _mm512_cmpneq_ps_mask, _mm512_mask_cmpneq_ps_mask, _mm512_cmpnle_ps_mask, _mm512_mask_cmpnle_ps_mask, _mm512_cmpnlt_ps_mask, _mm512_mask_cmpnlt_ps_mask, _mm512_cmpord_ps_mask, _mm512_mask_cmpord_ps_mask, _mm512_cmpunord_ps_mask, _mm512_mask_cmpunord_ps_mask		
_mm_cmp_sd_mask, _mm_mask_cmp_sd_mask, _mm_cmp_round_sd_mask, _mm_mask_cmp_sd_mask	Compares lower float64 vector elements based on comparison operand.	VCMPSD

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm_cmp_ss_mask`, `_mm_mask_cmp_ss_mask`, `_mm_cmp_round_ss_mask`, `_mm_mask_cmp_ss_mask`	Compares lower float32 vector elements based on comparison operand.	`VCMPSS`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register
imm	comparison predicate, which can be any of the following values: • `_MM_CMPINT_EQ` - Equal • `_MM_CMPINT_LT` - Less than • `_MM_CMPINT_LE` - Less than or Equal • `_MM_CMPINT_NE` - Not Equal • `_MM_CMPINT_NLT` - Not Less than • `_MM_CMPINT_GE` - Greater than or Equal • `_MM_CMPINT_NLE` - Not Less than or Equal • `_MM_CMPINT_GT` - Greater than

_mm512_cmp_pd_mask

```
extern __mmask8 __cdecl _mm512_cmp_pd_mask(__m512d a, __m512d b, const int imm);
```

Compares float64 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

_mm512_cmp_round_pd_mask

```
extern __mmask8 __cdecl _mm512_cmp_round_pd_mask(__m512d a, __m512d b, const int imm, const int
round);
```

Compares float64 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_mask_cmp_round_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmp_round_pd_mask(__mmask8 k, __m512d a, __m512d b, const
int imm, const int round);
```

Compares float64 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_mask_cmp_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmp_pd_mask(__mmask8 k, __m512d a, __m512d b, const int imm);
```

Compares float64 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpeq_pd_mask

```
extern __mmask8 __cdecl _mm512_cmp_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for equality.

The result is stored in mask vector.

_mm512_mask_cmpeq_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpeq_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for equality.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmple_pd_mask

```
extern __mmask8 __cdecl _mm512_cmple_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for less-than-or-equal.

The result is stored in mask vector.

_mm512_mask_cmple_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmple_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for less-than-or-equal.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmplt_pd_mask

```
extern __mmask8 __cdecl _mm512_cmplt_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for less-than.

The result is stored in mask vector.

_mm512_mask_cmplt_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmplt_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for less-than.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpneq_pd_mask

```
extern __mmask8 __cdecl _mm512_cmpneq_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for not-equal.

The result is stored in mask vector.

_mm512_mask_cmpneq_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpneq_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for not-equal.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpnle_pd_mask

```
extern __mmask8 __cdecl _mm512_cmpnle_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for not-less-than-or-equal.

The result is stored in mask vector.

_mm512_mask_cmpnle_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpnle_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for not-less-than-or-equal.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpnlt_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpnlt_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for not-less-than.

The result is stored in mask vector.

_mm512_mask_cmpnlt_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpnlt_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* for not-less-than.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpord_pd_mask

```
extern __mmask8 __cdecl _mm512_cmpord_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* to see if neither is NaN.

The result is stored in mask vector.

_mm512_mask_cmpord_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpord_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* to see if neither is NaN.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpunord_pd_mask

```
extern __mmask8 __cdecl _mm512_cmpunord_pd_mask(__m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* to see if either is NaN.

The result is stored in mask vector.

_mm512_mask_cmpord_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpord_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* to see if neither is NaN.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_cmpunord_pd_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpord_pd_mask(__mmask8 k, __m512d a, __m512d b);
```

Compares float64 elements in *a* and *b* to see if either is NaN.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_ps_mask

```
extern __mmask16 __cdecl _mm512_cmp_ps_mask(__m512 a, __m512 b, const int imm);
```

Compares float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

_mm512_mask_cmp_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmp_ps_mask(__mmask16 k, __m512 a, __m512 b, const int imm);
```

Compares float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_round_ps_mask

```
extern __mmask16 __cdecl _mm512_cmp_round_ps_mask(__m512 a, __m512 b, const int imm, const int
round);
```

Compares float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_mask_cmp_round_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmp_round_ps_mask(__mmask16 k, __m512 a, __m512 b, const
int imm, const int round);
```

Compares float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm512_cmpeq_ps_mask

```
extern __mmask16 __cdecl _mm512_cmpeq_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for equality.

The result is stored in mask vector.

_mm512_mask_cmpeq_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmpeq_ps_mask(__mmask16 k, __m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for equality.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmple_ps_mask

```
extern __mmask16 __cdecl _mm512_cmple_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for less-than-or-equal.

The result is stored in mask vector.

_mm512_mask_cmple_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmple_ps_mask(__mmask16 k, __m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for less-than-or-equal.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpunord_ps_mask

```
extern __mmask16 __cdecl _mm512_cmpunord_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* to see if either is NaN.

The result is stored in mask vector.

_mm512_mask_cmpunord_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmpunord_ps_mask(__mmask16 k, __m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* to see if neither is NaN.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmplt_ps_mask

```
extern __mmask16 __cdecl _mm512_cmplt_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for less-than.

The result is stored in mask vector.

_mm512_mask_cmplt_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmplt_ps_mask(__mmask16 k, __m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for less-than.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpneq_ps_mask

```
extern __mmask16 __cdecl _mm512_cmpneq_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for not-equal.

The result is stored in mask vector.

_mm512_mask_cmpneq_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmpneq_ps_mask(__mmask16 k, __m512 a, __m512 b, const int
round);
```

Compares float32 elements in *a* and *b* for not-equal.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpnle_ps_mask

```
extern __mmask16 __cdecl _mm512_cmpnle_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for not-less-than-or-equal.

The result is stored in mask vector.

_mm512_mask_cmpnle_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmpnle_ps_mask(__mmask16 k, __m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for not-less-than-or-equal.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpnlt_ps_mask

```
extern __mmask16 __cdecl _mm512_cmpnlt_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for not-less-than.

The result is stored in mask vector.

_mm512_mask_cmpnlt_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmpnlt_ps_mask(__mmask16 k, __m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* for not-less-than.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpord_ps_mask

```
extern __mmask16 __cdecl _mm512_cmpord_ps_mask(__m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* to see if either is NaN.

The result is stored in mask vector.

_mm512_mask_cmpord_ps_mask

```
extern __mmask16 __cdecl _mm512_mask_cmpord_ps_mask(__mmask16 k, __m512 a, __m512 b);
```

Compares float32 elements in *a* and *b* to see if either is NaN.

The result is stored in mask vector using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cmp_round_sd_mask

```
extern __mmask8 __cdecl _mm_cmp_round_sd_mask(__m128d a, __m128d b, const int imm, const round);
```

Compares lower float64 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_cmp_round_sd_mask

```
extern __mmask8 __cdecl _mm_mask_cmp_round_sd_mask(__mmask8 k, __m128d a, __m128d b, const int
imm, const int round);
```

Compares lower float64 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the result in mask vector *k* using zeromask *k* (the element is zeroed out when mask bit 0 is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_cmp_sd_mask

```
extern __mmask8 __cdecl _mm_cmp_sd_mask(__m128d a, __m128d b, const int imm);
```

Compares lower float64 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

_mm_mask_cmp_sd_mask

```
extern __mmask8 __cdecl _mm_mask_cmp_sd_mask(__mmask8 k, __m128d a, __m128d b, const int imm);
```

Compares lower float64 elements in *a* and *b* based on the comparison operand specified by *imm*

The result is stored in mask vector using zeromask *k* (the element is zeroed out when mask bit 0 is not set).

_mm_cmp_round_ss_mask

```
extern __mmask8 __cdecl _mm_cmp_round_ss_mask(__m128 a, __m128 b, const int imm, const int
round);
```

Compares lower float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_mask_cmp_round_ss_mask

```
extern __mmask8 __cdecl _mm_mask_cmp_round_ss_mask(__mmask8 k, __m128 a, __m128 b, const int
imm, const int round);
```

Compares lower float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector using zeromask *k* (the element is zeroed out when mask bit 0 is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *round* to suppress all exceptions.

_mm_cmp_ss_mask

```
extern __mmask8 __cdecl _mm_cmp_ss_mask(__m128 a, __m128 b, const int imm);
```

Compares lower float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector.

_mm_mask_cmp_ss_mask

```
extern __mmask8 __cdecl _mm_mask_cmp_ss_mask(__mmask8 k, __m128 a, __m128 b, const int imm);
```

Compares lower float32 elements in *a* and *b* based on the comparison operand specified by *imm*.

The result is stored in mask vector using zeromask *k* (the element is zeroed out when mask bit 0 is not set).

Intrinsics for Integer Comparison Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_cmp_epi32_mask` `_mm512_cmpeq_epi32_mask`, `_mm512_cmpge_epi32_mask`, `_mm512_cmpgt_epi32_mask`, `_mm512_cmple_epi32_mask`, `_mm512_cmplt_epi32_mask`, `_mm512_cmpneq_epi32_mask`, `_mm512_mask_cmp_epi32_mask`, `_mm512_mask_cmpeq_epi32_mask`, `_mm512_mask_cmpge_epi32_mask`, `_mm512_mask_cmpgt_epi32_mask`, `_mm512_mask_cmple_epi32_mask`, `_mm512_mask_cmplt_epi32_mask`, `_mm512_mask_cmpneq_epi32_mask`	Compare signed int32 elements based on the comparison operand.	`VPCMPD`
`_mm512_cmp_epi64_mask`, `_mm512_mask_cmp_epi64_mask` `_mm512_cmpeq_epi64_mask`, `_mm512_cmpge_epi64_mask`, `_mm512_cmpgt_epi64_mask`, `_mm512_cmple_epi64_mask`, `_mm512_cmplt_epi64_mask`, `_mm512_cmpneq_epi64_mask`, `_mm512_mask_cmp_epi64_mask`, `_mm512_mask_cmpeq_epi64_mask`, `_mm512_mask_cmpge_epi64_mask`,	Compare signed int64 elements based on the comparison operand.	`VPCMPQ`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_mask_cmpgt_epi64_mask, _mm512_mask_cmple_epi64_mask, _mm512_mask_cmplt_epi64_mask, _mm512_mask_cmpneq_epi64_mask		
_mm512_cmp_epu32_mask _mm512_cmpeq_epu32_mask, _mm512_cmpge_epu32_mask, _mm512_cmpgt_epu32_mask, _mm512_cmple_epu32_mask, _mm512_cmplt_epu32_mask, _mm512_cmpneq_epu32_mask, _mm512_mask_cmp_epu32_mask, _mm512_mask_cmpeq_epu32_mask, _mm512_mask_cmpge_epu32_mask, _mm512_mask_cmpgt_epu32_mask, _mm512_mask_cmple_epu32_mask, _mm512_mask_cmplt_epu32_mask, _mm512_mask_cmpneq_epu32_mask	Compare unsigned int32 elements based on the comparison operand.	VPCMPUD
_mm512_cmp_epu64_mask, _mm512_mask_cmp_epu64_mask _mm512_cmpeq_epu64_mask, _mm512_cmpge_epu64_mask, _mm512_cmpgt_epu64_mask, _mm512_cmple_epu64_mask, _mm512_cmplt_epu64_mask, _mm512_cmpneq_epu64_mask, _mm512_mask_cmp_epu64_mask, _mm512_mask_cmpeq_epu64_mask, _mm512_mask_cmpge_epu64_mask, _mm512_mask_cmpgt_epu64_m	Compare unsigned int64 elements based on the comparison operand.	VPCMPUQ

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
ask, _mm512_mask_cmple_epu64_mask, _mm512_mask_cmplt_epu64_mask, _mm512_mask_cmpneq_epu64_mask		

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
imm	comparison operand
src	source element

_mm_comi_round_sd

```
extern int __cdecl _mm_comi_round_sd(__m128d a, __m128d b, const int imm, const int sae);
```

Compare the lower double-precision (64-bit) floating-point element in *a* and *b* based on the comparison operand specified by *imm*, and return the boolean result (0 or 1).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm_comi_round_ss

```
extern int __cdecl _mm_comi_round_ss(__m128 a, __m128 b, const int imm, const int sae);
```

Compare the lower single-precision (32-bit) floating-point element in *a* and *b* based on the comparison operand specified by *imm*, and return the boolean result (0 or 1).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cmp_epi32_mask

```
extern __mmask16 __cdecl _mm512_cmp_epi32_mask(__m512i a, __m512i b, const int imm);
```

Compare packed int32 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k*.

_mm512_mask_cmp_epi32_mask

```
extern __mmask16 __cdecl _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, const int
imm);
```

Compare packed int32 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpeq_epi32_mask

```
extern __mmask16 __cdecl _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for equality, and store the results in mask vector *k*.

_mm512_mask_cmpeq_epi32_mask

```
 extern __mmask16 __cdecl  _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for equality, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpge_epi32_mask

```
extern __mmask16 __cdecl  _mm512_cmpge_epi32_mask(__m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmpge_epi32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmpge_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpgt_epi32_mask

```
extern __mmask16 __cdecl  _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for greater-than, and store the results in mask vector *k*.

_mm512_mask_cmpgt_epi32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for greater-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmple_epi32_mask

```
extern __mmask16 __cdecl  _mm512_cmple_epi32_mask(__m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmple_epi32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmple_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for less-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmplt_epi32_mask

```
extern __mmask16 __cdecl  _mm512_cmplt_epi32_mask(__m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for less-than, and store the results in mask vector *k*.

_mm512_mask_cmplt_epi32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmplt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpneq_epi32_mask

```
extern __mmask16 __cdecl  _mm512_cmpneq_epi32_mask(__m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for not-equal, and store the results in mask vector *k*.

_mm512_mask_cmpneq_epi32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmpneq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed int32 elements in *a* and *b* for not-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_epi64_mask

```
extern __mmask8 __cdecl _mm512_cmp_epi64_mask(__m512i a, __m512i b, const int imm);
```

Compare packed int64 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k*.

_mm512_mask_cmp_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, const int
imm);
```

Compare packed int64 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpeq_epi64_mask

```
extern __mmask8 __cdecl _mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for equality, and store the results in mask vector *k*.

_mm512_mask_cmpeq_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for equality, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpge_epi64_mask

```
extern __mmask8 __cdecl _mm512_cmpge_epi64_mask(__m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmpge_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpge_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpgt_epi64_mask

```
extern __mmask8 __cdecl _mm512_cmpgt_epi64_mask(__m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for greater-than, and store the results in mask vector *k*.

_mm512_mask_cmpgt_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for greater-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmple_epi64_mask

```
extern __mmask8 __cdecl _mm512_cmple_epi64_mask(__m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmple_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmple_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmplt_epi64_mask

```
extern __mmask8 __cdecl _mm512_cmplt_epi64_mask(__m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for less-than, and store the results in mask vector *k*.

_mm512_mask_cmplt_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmplt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for less-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpneq_epi64_mask

```
extern __mmask8 __cdecl _mm512_cmpneq_epi64_mask(__m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for not-equal, and store the results in mask vector *k*.

_mm512_mask_cmpneq_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpneq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed int64 elements in *a* and *b* for not-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_epu32_mask

```
extern __mmask16 __cdecl _mm512_cmp_epu32_mask(__m512i a, __m512i b, const int imm);
```

Compare packed unsigned int32 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k*.

_mm512_mask_cmp_epu32_mask

```
extern __mmask16 __cdecl _mm512_mask_cmp_epu32_mask(__mmask16 k, __m512i a, __m512i b, const int
imm);
```

Compare packed unsigned int32 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpeq_epu32_mask

```
extern __mmask16 __cdecl  _mm512_cmpeq_epu32_mask(__m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for equality, and store the results in mask vector *k*.

_mm512_mask_cmpeq_epu32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmpeq_epu32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for equality, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpge_epu32_mask

```
extern __mmask16 __cdecl  _mm512_cmpge_epu32_mask(__m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmpge_epu32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmpge_epu32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpgt_epu32_mask

```
extern __mmask16 __cdecl  _mm512_cmpgt_epu32_mask(__m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for greater-than, and store the results in mask vector *k*.

_mm512_mask_cmpgt_epu32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmpgt_epu32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for greater-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmple_epu32_mask

```
extern __mmask16 __cdecl  _mm512_cmple_epu32_mask(__m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmple_epu32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmple_epu32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for less-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmplt_epu32_mask

```
extern __mmask16 __cdecl  _mm512_cmplt_epu32_mask(__m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for less-than, and store the results in mask vector *k*.

_mm512_mask_cmplt_epu32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmplt_epu32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpneq_epu32_mask

```
extern __mmask16 __cdecl  _mm512_cmpneq_epu32_mask(__m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for not-equal, and store the results in mask vector *k*.

_mm512_mask_cmpneq_epu32_mask

```
extern __mmask16 __cdecl  _mm512_mask_cmpneq_epu32_mask(__mmask16 k, __m512i a, __m512i b);
```

Compare packed unsigned int32 elements in *a* and *b* for not-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmp_epu64_mask

```
extern __mmask8 __cdecl _mm512_cmp_epu64_mask(__m512i a, __m512i b, const _MM_CMPINT_ENUM imm);
```

Compare packed unsigned int64 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k*.

_mm512_mask_cmp_epu64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, const
_MM_CMPINT_ENUM imm);
```

Compare packed unsigned int64 elements in *a* and *b* based on the comparison operand specified by *imm*, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpge_epu64_mask

```
extern __mmask8 __cdecl _mm512_cmpge_epu64_mask(__m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmpge_epu64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpge_epu64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for greater-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpgt_epu64_mask

```
extern __mmask8 __cdecl _mm512_cmpgt_epu64_mask(__m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for greater-than, and store the results in mask vector *k*.

_mm512_mask_cmpgt_epu64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpgt_epu64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for greater-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmple_epu64_mask

```
extern __mmask8 __cdecl _mm512_cmple_epu64_mask(__m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k*.

_mm512_mask_cmple_epu64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmple_epu64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for less-than-or-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmplt_epu64_mask

```
extern __mmask8 __cdecl _mm512_cmplt_epu64_mask(__m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for less-than, and store the results in mask vector *k*.

_mm512_mask_cmplt_epu64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmplt_epu64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for less-than, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpeq_epu64_mask

```
extern __mmask8 __cdecl _mm512_cmpeq_epu64_mask(__m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for equality, and store the results in mask vector *k*.

_mm512_mask_cmpeq_epu64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpeq_epu64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for equality, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cmpneq_epu64_mask

```
extern __mmask8 __cdecl _mm512_cmpneq_epu64_mask(__m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for not-equal, and store the results in mask vector *k*.

_mm512_mask_cmpneq_epu64_mask

```
extern __mmask8 __cdecl _mm512_mask_cmpneq_epu64_mask(__mmask8 k, __m512i a, __m512i b);
```

Compare packed unsigned int64 elements in *a* and *b* for not-equal, and store the results in mask vector *k* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Compression Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_mask_compress_pd`, `_mm512_maskz_compress_pd`	Contiguously store active float32 elements.	VCOMPRESSPD

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_mask_compress_ps`, `_mm512_maskz_compress_ps`	Contiguously store active float64 elements.	`VCOMPRESSPS`
`_mm512_mask_compress_epi32`, `_mm512_maskz_compress_epi32`, `_mm512_mask_compressstoreu_epi32`	Contiguously store active int32 elements.	`VPCOMPRESSD`
`_mm512_mask_compress_epi64`, `_mm512_maskz_compress_epi64`	Contiguously store active int64 elements.	`VPCOMPRESSQ`

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result
base_addr	pointer to base address in memory to begin load or store operation

_mm512_mask_compress_pd

```
extern __m512d __cdecl _mm512_mask_compress_pd(__m512d a, __mmask8 k, __m512d src);
```

Contiguously stores the active float64 elements in *a* (those with their respective bit set in writemask *k*) to destination, and passes through the remaining elements from *src*.

_mm512_maskz_compress_pd

```
extern __m512d __cdecl _mm512_maskz_compress_pd(__mmask8 k, __m512d a);
```

Contiguously stores the active float64 elements in *a* (those with their respective bit set in zeromask *k*) to destination, and set the remaining elements to zero.

_mm512_mask_compress_ps

```
extern __m512 __cdecl _mm512_mask_compress_ps(__m512 a, __mmask16 k, __m512 src);
```

Contiguously stores the active float32 elements in *a* (those with their respective bit set in writemask *k*) to destination, and passes through the remaining elements from *src*.

_mm512_maskz_compress_ps

```
extern __m512 __cdecl _mm512_maskz_compress_ps(__mmask16 k, __m512 a);
```

Contiguously stores the active float32 elements in *a* (those with their respective bit set in zeromask *k*) to destination, and set the remaining elements to zero.

_mm512_mask_compressstoreu_pd

```
extern void __cdecl _mm512_mask_compressstoreu_pd(void* base_addr, __mmask8 k, __m512d a);
```

Contiguously stores the active float64 elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_mask_compressstoreu_ps

```
extern void __cdecl _mm512_mask_compressstoreu_ps(void* base_addr, __mmask16 k, __m512 a);
```

Contiguously stores the active float32 elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_mask_compress_epi32

```
extern __m512i __cdecl _mm512_mask_compress_epi32(__m512i a, __mmask16 k, __m512i src);
```

Contiguously stores the active int32 elements in *a* (those with their respective bit set in writemask *k*) to destination, and passes through the remaining elements from *src*.

_mm512_maskz_compress_epi32

```
extern __m512i __cdecl _mm512_maskz_compress_epi32(__mmask16 k, __m512i a);
```

Contiguously stores the active int32 elements in *a* (those with their respective bit set in zeromask *k*) to destination, and set the remaining elements to zero.

_mm512_mask_compress_epi64

```
extern __m512i __cdecl _mm512_mask_compress_epi64(__m512i a, __mmask8 k, __m512i src);
```

Contiguously stores the active int64 elements in *a* (those with their respective bit set in writemask *k*) to destination, and passes through the remaining elements from *src*.

_mm512_maskz_compress_epi64

```
extern __m512i __cdecl _mm512_maskz_compress_epi64(__mmask8 k, __m512i a);
```

Contiguously stores the active int64 elements in *a* (those with their respective bit set in zeromask *k*) to destination, and set the remaining elements to zero.

_mm512_mask_compressstoreu_epi32

```
extern void __cdecl _mm512_mask_compressstoreu_epi32(void* base_addr, __mmask16 k, __m512i a);
```

Contiguously stores the active int32 elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_mask_compressstoreu_epi64

```
extern void __cdecl _mm512_mask_compressstoreu_epi64(void* base_addr, __mmask8 k, __m512i a);
```

Contiguously stores the active int64 elements in *a* (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

Intrinsics for Conversion Operations

Intrinsics for FP Conversion Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_cvtps_pd_mm512_mask_cvtps_pd_mm512_maskz_cvtps_pd _mm512_cvt_roundps_pd, _mm512_mask_cvt_roundps_pd, _mm512_maskz_cvt_roundps_pd	Converts rounded float32 to float 64.	VCVTPS2PD
_mm512_cvt_roundps_epi32, _mm512_mask_cvt_roundps_epi32, _mm512_maskz_cvt_roundps_epi32 _mm512_cvtt_roundps_epi32, _mm512_mask_cvtt_roundps_epi32, _mm512_maskz_cvtt_roundps_epi32	Converts rounded float32 to int32.	VCVTPS2DQ/VCVTTPS2DQ
_mm512_cvt_roundps_epu32, _mm512_mask_cvt_roundps_epu32, _mm512_maskz_cvt_roundps_epu32 _mm512_cvtt_roundps_epu32, _mm512_mask_cvtt_roundps_epu32, _mm512_maskz_cvtt_roundps_epu32	Converts rounded float32 to unsigned int32.	VCVTPS2UDQ/VCVTTPS2UDQ
_mm_cvt_roundsd_i32, _mm_cvt_roundsd_i64 _mm_cvtt_roundsd_i32, _mm_cvtt_roundsd_i64	Converts rounded scalar float64 to int32/int64.	VCVTSD2SI/VCVTTSD2SI
_mm_cvt_roundsd_u32, _mm_cvt_roundsd_u64 _mm_cvtt_roundsd_u32, _mm_cvtt_roundsd_u64	Converts rounded scalar float64 to unsigned int32/int64.	VCVTSD2USI/VCVTTSD2USI
_mm_cvt_roundss_i32, _mm_cvt_roundss_i64 _mm_cvtt_roundss_i32, _mm_cvtt_roundss_i64	Converts rounded scalar float32 to int32/int64.	VCVTSS2SI/VCVTTSS2SI

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm_cvt_roundss_u32, _mm_cvt_roundss_u64 _mm_cvtt_roundss_u32, _mm_cvtt_roundss_u64	Converts rounded scalar float32 to unsigned int32/int64.	VCVTSS2USI/VCVTTSS2USI
_mm512_cvtpd_ps_mm512_mask_cvtpd_ps_mm512_maskz_cvtpd_ps _mm512_cvt_roundpd_ps, _mm512_mask_cvt_roundpd_ps, _mm512_maskz_cvt_roundpd_ps	Converts rounded float64 to float32.	VCVTPD2PS
_mm512_cvt_roundpd_epi32, _mm512_mask_cvt_roundpd_epi32, _mm512_maskz_cvt_roundpd_epi32 _mm512_cvtt_roundpd_epi32, _mm512_mask_cvtt_roundpd_epi32, _mm512_maskz_cvtt_roundpd_epi32	Converts rounded float64 to int32.	VCVTPD2DQ/VCVTTPD2DQ
_mm512_cvt_roundpd_epu32, _mm512_mask_cvt_roundpd_epu32, _mm512_maskz_cvt_roundpd_epu32 _mm512_cvtt_roundpd_epu32, _mm512_mask_cvtt_roundpd_epu32, _mm512_maskz_cvtt_roundpd_epu32	Converts rounded float64 to unsigned int32.	VCVTPD2UDQ/VCVTTPD2UDQ
_mm512_cvtph_ps_mm512_mask_cvtph_ps_mm512_maskz_cvtph_ps _mm512_cvt_roundph_ps, _mm512_mask_cvt_roundph_ps, _mm512_maskz_cvt_roundph_ps	Converts rounded float64 to float32.	VCVTPH2PS
_mm512_cvt_roundps_ph, _mm512_mask_cvt_roundps_ph, _mm512_maskz_cvt_roundps_ph	Converts rounded float32 to scalar float32.	VCVTPS2PH
_mm_mask_cvtss_sd_mm_maskz_cvtss_sd _mm_cvt_roundss_sd, _mm_mask_cvt_roundss_sd, _mm_maskz_cvt_roundss_sd	Converts rounded scalar float32 to scalar float64.	VCVTSS2SD

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm_cvt_roundsd_ss`, `_mm_mask_cvt_roundsd_ss`, `_mm_maskz_cvt_roundsd_ss`	Converts rounded scalar float64 to scalar float32.	`VCVTSD2SS`
`_mm512_cvtepu32_ps_mm512_mask_cvtepu32_ps_mm512_maskz_cvtepu32_ps` `_mm512_cvt_roundepu32_ps_mm512_mask_cvt_roundepu32_ps_mm512_maskz_cvt_roundepu32_ps`	Converts packed unsigned int32 to float32.	`VCVTUDQ2PS`
`_mm512_cvtss_f32`	Extracts a float32 value from the first vector element of an `__m512`. It does so in the most efficient manner possible in the context used.	`MOVSS`/`VMOVSS`
`_mm512_cvtsd_f64`	Extracts a float64 value from first vector element of an `__m512d`. It does so in the most efficient manner possible in the context used.	`MOVSD`/`VMOVSD`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_cvt_roundpd_ps

```
extern __m256 __cdecl _mm512_cvt_roundpd_ps(__m512d a, int round);
```

Converts float64 elements in *a* to float32 elements, and stores the result.

_mm512_mask_cvt_roundpd_ps

```
extern __m256 __cdecl _mm512_mask_cvt_roundpd_ps(__m256 src, __mmask8 k, __m512d a, int round);
```

Converts float64 elements in *a* to float32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundpd_ps

```
extern __m256 __cdecl _mm512_maskz_cvt_roundpd_ps(__mmask8 k, __m512d a, int round);
```

Converts float64 elements in *a* to float32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtpd_ps

```
extern __m256 __cdecl _mm512_cvt_pd_ps(__m512d a);
```

Converts float64 elements in *a* to float32 elements, and stores the result.

_mm512_mask_cvtpd_ps

```
extern __m256 __cdecl _mm512_mask_cvt_pd_ps(__m256 src, __mmask8 k, __m512d a);
```

Converts float64 elements in *a* to float32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtpd_ps

```
extern __m256 __cdecl _mm512_maskz_cvt_pd_ps(__mmask8 k, __m512d a);
```

Converts float64 elements in *a* to float32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundpd_epi32

```
extern __m512i __cdecl _mm512_cvt_roundpd_epi32(__m512d a, int round);
```

Converts float64 elements in *a* to int32 elements, and stores the results.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvt_roundpd_epi32

```
extern __m512i __cdecl _mm512_mask_cvt_roundpd_epi32(__m256i src, __mmask8 k, __m512d a, int
round);
```

Converts float64 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvt_roundpd_epi32

```
extern __m512i __cdecl _mm512_maskz_cvt_roundpd_epi32(__mmask8 k, __m512d a, int round);
```

Converts float64 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvtpd_epi32

```
extern __m512i __cdecl _mm512_cvtpd_epi32(__m512d a);
```

Converts float64 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvtpd_epi32

```
extern __m512i __cdecl _mm512_mask_cvtpd_epi32(__m256i src, __mmask8 k, __m512d a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtpd_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtpd_epi32(__mmask8 k, __m512d a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundpd_epi32

```
extern __m512i __cdecl _mm512_cvtt_roundpd_epi32(__m512d a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the results.

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvtt_roundpd_epi32

```
extern __m512i __cdecl _mm512_mask_cvtt_roundpd_epi32(__m256i src, __mmask8 k, __m512d a, int
round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvtt_roundpd_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtt_roundpd_epi32(__mmask8 k, __m512d a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttpd_epi32

```
extern __m512i __cdecl _mm512_cvttpd_epi32(__m512d a);
```

Converts float64 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvttpd_epi32

```
extern __m512i __cdecl _mm512_mask_cvttpd_epi32(__m256i src, __mmask8 k, __m512d a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvttpd_epi32

```
extern __m512i __cdecl _mm512_maskz_cvttpd_epi32(__mmask8 k, __m512d a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundpd_epu32

```
extern __m512i __cdecl _mm512_cvt_roundpd_epu32(__m512 a, int round);
```

Converts float64 elements in *a* to int32 elements, and stores the results.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvt_roundpd_epu32

```
extern __m512i __cdecl _mm512_mask_cvt_roundpd_epu32(__m256i src, __mmask16 k, __m512 a, int
round);
```

Converts float64 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvt_roundpd_epu32

```
extern __m512i __cdecl _mm512_maskz_cvt_roundpd_epu32(__mmask16 k, __m512 a, int round);
```

Converts float64 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvtpd_epu32

```
extern __m512i __cdecl _mm512_cvtpd_epu32(__m512 a);
```

Converts float64 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvtpd_epu32

```
extern __m512i __cdecl _mm512_mask_cvtpd_epu32(__m256i src, __mmask16 k, __m512 a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtpd_epu32

```
extern __m512i __cdecl _mm512_maskz_cvtpd_epu32(__mmask16 k, __m512 a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundpd_epu32

```
extern __m512i __cdecl _mm512_cvtt_roundpd_epu32(__m512 a, int round);
```

Converts float64 elements in *a* to int32 elements, and stores the results.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvtt_roundpd_epu32

```
extern __m512i __cdecl _mm512_mask_cvtt_roundpd_epu32(__m256i src, __mmask16 k, __m512 a, int
round);
```

Converts float64 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvtt_roundpd_epu32

```
extern __m512i __cdecl _mm512_maskz_cvtt_roundpd_epu32(__mmask16 k, __m512 a, int round);
```

Converts float64 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttpd_epu32

```
extern __m512i __cdecl _mm512_cvttpd_epu32(__m512 a);
```

Converts float64 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvttpd_epu32

```
extern __m512i __cdecl _mm512_mask_cvttpd_epu32(__m256i src, __mmask16 k, __m512 a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvttpd_epu32

```
extern __m512i __cdecl _mm512_maskz_cvttpd_epu32(__mmask16 k, __m512 a);
```

Converts float64 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtph_ps

```
extern __m512 __cdecl _mm512_cvtph_ps(__m256i a);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the results.

_mm512_mask_cvtph_ps

```
extern __m512 __cdecl _mm512_mask_cvtph_ps(__m512 src, __mmask16 k, __m256i a);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtph_ps

```
extern __m512 __cdecl _mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundph_ps

```
extern __m512 __cdecl _mm512_cvt_roundph_ps(__m256i a, int round);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the result.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvt_roundph_ps

```
extern __m512 __cdecl _mm512_mask_cvt_roundph_ps(__m512 src, __mmask16 k, __m256i a, int round);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvt_roundph_ps

```
extern __m512 __cdecl _mm512_maskz_cvt_roundph_ps(__mmask16 k, __m256i a, int round);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvt_roundps_ph

```
extern __m256i __cdecl _mm512_cvt_roundps_ph(__m512 a, int round);
```

Converts float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the results.

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvt_roundps_ph

```
extern __m256i __cdecl _mm512_mask_cvt_roundps_ph(__m256i src, __mmask16 k, __m512 a, int round);
```

Converts float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvt_roundps_ph

```
extern __m256i __cdecl _mm512_maskz_cvt_roundps_ph(__mmask16 k, __m512 a, int round);
```

Converts float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvtph_ps

```
extern __m256i __cdecl _mm512_cvtps_ph(__m512 a);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the results.

_mm512_mask_cvtph_ps

```
extern __m256i __cdecl _mm512_mask_cvtps_ph(__m256i src, __mmask16 k, __m512 a);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtph_ps

```
extern __m256i __cdecl _mm512_maskz_cvtps_ph(__mmask16 k, __m512 a);
```

Converts packed half-precision (16-bit) floating-point elements in *a* to float32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_pd

```
extern __m512d __cdecl _mm512_cvt_roundps_pd(__m256 a, int round);
```

Converts float32 elements in *a* to float64 elements, and stores the results.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvt_roundps_pd

```
extern __m512d __cdecl _mm512_mask_cvt_roundps_pd(__m512d src, __mmask8 k, __m256 a, int round);
```

Converts float32 elements in *a* to float64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvt_roundps_pd

```
extern __m512d __cdecl _mm512_maskz_cvt_roundps_pd(__mmask8 k, __m256 a, int round);
```

Converts float32 elements in *a* to float64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvtps_pd

```
extern __m512d __cdecl _mm512_cvt_ps_pd(__m256 a);
```

Converts float32 elements in *a* to float64 elements, and stores the result.

_mm512_mask_cvtps_pd

```
extern __m512d __cdecl _mm512_mask_cvt_ps_pd(__m512d src, __mmask8 k, __m256 a);
```

Converts float32 elements in *a* to float64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtps_pd

```
extern __m512d __cdecl _mm512_maskz_cvt_ps_pd(__mmask8 k, __m256 a);
```

Converts float32 elements in *a* to float64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_epi32

```
extern __m512i __cdecl _mm512_cvt_roundps_epi32(__m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the results.

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvt_roundps_epi32

```
extern __m512i __cdecl _mm512_mask_cvt_roundps_epi32(__m512i src, __mmask16 k, __m512 a, int
round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvt_roundps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvt_roundps_epi32(__mmask16 k, __m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvtps_epi32

```
extern __m512i __cdecl _mm512_cvtps_epi32(__m512 a);
```

Converts float32 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvtps_epi32

```
extern __m512i __cdecl _mm512_mask_cvtps_epi32(__m512i src, __mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtps_epi32(__mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundps_epi32

```
extern __m512i __cdecl _mm512_cvtt_roundps_epi32(__m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the results.

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvtt_roundps_epi32

```
extern __m512i __cdecl _mm512_mask_cvtt_roundps_epi32(__m512i src, __mmask16 k, __m512 a, int
round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvtt_roundps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtt_roundps_epi32(__mmask16 k, __m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttps_epi32

```
extern __m512i __cdecl _mm512_cvttps_epi32(__m512 a);
```

Converts float32 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvttps_epi32

```
extern __m512i __cdecl _mm512_mask_cvttps_epi32(__m512i src, __mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvttps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvttps_epi32(__mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_epu32

```
extern __m512i __cdecl _mm512_cvt_roundps_epu32(__m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the results.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvt_roundps_epu32

```
extern __m512i __cdecl _mm512_mask_cvt_roundps_epu32(__m512i src, __mmask16 k, __m512 a, int
round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvt_roundps_epu32

```
extern __m512i __cdecl _mm512_maskz_cvt_roundps_epu32(__mmask16 k, __m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvtps_epu32

```
extern __m512i __cdecl _mm512_cvtps_epu32(__m512 a);
```

Converts float32 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvtps_epu32

```
extern __m512i __cdecl _mm512_mask_cvtps_epu32(__m512i src, __mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtps_epu32

```
extern __m512i __cdecl _mm512_maskz_cvtps_epu32(__mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundps_epu32

```
extern __m512i __cdecl _mm512_cvtt_roundps_epu32(__m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the results.

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvtt_roundps_epu32

```
extern __m512i __cdecl _mm512_mask_cvtt_roundps_epu32(__m512i src, __mmask16 k, __m512 a, int
round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvtt_roundps_epu32

```
extern __m512i __cdecl _mm512_maskz_cvtt_roundps_epu32(__mmask16 k, __m512 a, int round);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttps_epu32

```
extern __m512i __cdecl _mm512_cvttps_epu32(__m512 a);
```

Converts float32 elements in *a* int32 elements, and stores the result.

_mm512_mask_cvttps_epu32

```
extern __m512i __cdecl _mm512_mask_cvttps_epu32(__m512i src, __mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvttps_epu32

```
extern __m512i __cdecl _mm512_maskz_cvttps_epu32(__mmask16 k, __m512 a);
```

Converts float32 elements in *a* to int32 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvt_roundss_sd

```
extern __m128d __cdecl _mm_cvt_roundss_sd(__m128d a, __m128 b, int round);
```

Converts the lower float32 element in *b* to a float64 element, stores the result in the lower destination element, and copies the upper element from *a* to the upper destination element .

_mm_mask_cvt_roundss_sd

```
extern __m128d __cdecl _mm_mask_cvt_roundss_sd(__m128d src, __mmask8 k, __m128d a, __m128 b, int
round);
```

Converts the lower float32 element in *b* to a float64 element, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_maskz_cvt_roundss_sd

```
extern __m128d __cdecl _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 b, int round);
```

Converts the lower float32 element in *b* to a float64 element, store the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_mask_cvtss_sd

```
extern __m128d __cdecl _mm_mask_cvt_ss_sd(__m128d src, __mmask8 k, __m128d a, __m128 b);
```

Converts the lower float32 element in *b* to a float64 element, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_maskz_cvtss_sd

```
extern __m128d __cdecl _mm_maskz_cvt_ss_sd(__mmask8 k, __m128d a, __m128 b);
```

Converts the lower float32 element in *b* to a float64 element, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_cvt_roundsd_ss

```
extern __m128 __cdecl _mm_cvt_roundsd_ss(__m128 a, __m128d b, int round);
```

Converts float64 elements in *b* to a single-precision (64-bit) floating-point elements, stores the result in the lower destination element, and copies the upper element from *a* to the upper destination element .

_mm_mask_cvt_roundsd_ss

```
extern __m128 __cdecl _mm_mask_cvt_roundsd_ss(__m128 src, __mmask8 k, __m128 a, __m128d b, int
round);
```

Converts float64 elements in *b* to a single-precision (64-bit) floating-point elements, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_maskz_cvt_roundsd_ss

```
extern __m128 __cdecl _mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a, __m128d b, int round);
```

Converts float64 elements in *b* to a single-precision (64-bit) floating-point elements, the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_mask_cvtsd_ss

```
extern __m128 __cdecl _mm_mask_cvt_sd_ss(__m128 src, __mmask8 k, __m128 a, __m128d b);
```

Converts float64 elements in *b* to a single-precision (64-bit) floating-point elements, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_maskz_cvtsd_ss

```
extern __m128 __cdecl _mm_maskz_cvt_sd_ss(__mmask8 k, __m128 a, __m128d b);
```

Converts float64 elements in *b* to a single-precision (64-bit) floating-point elements, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy the upper element from *a* to the upper destination element.

_mm_cvt_roundsd_i32 / _mm_cvt_roundsd_si32

```
extern int __cdecl _mm_cvt_roundsd_i32(__m128d a, int round);
extern int __cdecl _mm_cvt_roundsd_si32(__m128d a, int round);
```

_mm_cvt_sd_i32 / _mm_cvt_sd_si32

```
extern __int64 __cdecl _mm_cvt_sd_i32(__m128d);
extern __int64 __cdecl _mm_cvt_sd_i64(__m128d);
```

_mm_cvt_roundsd_i64 / _mm_cvt_roundsd_si64

```
extern __int64 __cdecl _mm_cvt_roundsd_i64(__m128d, int round);
extern __int64 __cdecl _mm_cvt_roundsd_si64(__m128d, int round);
```

_mm_cvti64 / _mm_cvtsd_si64

```
extern __m128d __cdecl _mm_cvt_i64_sd(__m128d a, __int64);
extern __m128d __cdecl _mm_cvt_si64_sd(__m128d a, __int64);
```

_mm_cvt_roundsd_u32 / _mm_cvt_roundsd_u64

```
extern unsigned int __cdecl _mm_cvt_roundsd_u32(__m128d a, int round);
extern unsigned __int64 __cdecl _mm_cvt_roundsd_u64(__m128d a, int round);
```

_mm_cvt_sd_u32 / _mm_cvt_sd_u64

```
extern unsigned int __cdecl _mm_cvt_sd_u32(__m128d a);
extern unsigned __int64 __cdecl _mm_cvt_sd_u64(__m128d a);
```

_mm_cvt_roundsd_i32 / _mm_cvt_roundsd_si32

```
extern int __cdecl _mm_cvt_roundsd_i32(__m128d a, int round);
extern int __cdecl _mm_cvt_roundsd_si32(__m128d a, int round);
```

_mm_cvtt_sd_i32 / _mm_cvtt_sd_si32

```
extern __int64 __cdecl _mm_cvtt_sd_i32(__m128d);
extern __int64 __cdecl _mm_cvtt_sd_i64(__m128d);
```

_mm_cvtt_roundsd_i64 / _mm_cvtt_roundsd_si64

```
extern __int64 __cdecl _mm_cvtt_roundsd_i64(__m128d, int round);
extern __int64 __cdecl _mm_cvtt_roundsd_si64(__m128d, int round);
```

_mm_cvtti64 / _mm_cvttsd_si64

```
extern __m128d __cdecl _mm_cvtt_i64_sd(__m128d a, __int64);
extern __m128d __cdecl _mm_cvtt_si64_sd(__m128d a, __int64);
```

_mm_cvtt_roundsd_u32 / _mm_cvtt_roundsd_u64

```
extern unsigned int __cdecl _mm_cvtt_roundsd_u32(__m128d a, int round);
extern unsigned __int64 __cdecl _mm_cvtt_roundsd_u64(__m128d a, int round);
```

_mm_cvtt_sd_u32 / _mm_cvtt_sd_u64

```
extern unsigned int __cdecl _mm_cvtt_sd_u32(__m128d a);
extern unsigned __int64 __cdecl _mm_cvtt_sd_u64(__m128d a);
```

_mm_cvt_roundss_i32 / _mm_cvt_roundss_si32

```
extern int __cdecl _mm_cvt_roundss_i32(__m128d a, int round);
extern int __cdecl _mm_cvt_roundss_si32(__m128d a, int round);
```

_mm_cvt_ss_i32 / _mm_cvt_ss_si32

```
extern __int64 __cdecl _mm_cvt_ss_i32(__m128d);
extern __int64 __cdecl _mm_cvt_ss_i64(__m128d);
```

_mm_cvt_roundss_i64 / _mm_cvt_roundss_si64

```
extern __int64 __cdecl _mm_cvt_roundss_i64(__m128d, int round);
extern __int64 __cdecl _mm_cvt_roundss_si64(__m128d, int round);
```

_mm_cvti64 / _mm_cvtss_si64

```
extern __m128d __cdecl _mm_cvt_i64_sd(__m128d a, __int64);
extern __m128d __cdecl _mm_cvt_si64_sd(__m128d a, __int64);
```

_mm_cvt_roundss_u32 / _mm_cvt_roundss_u64

```
extern unsigned int __cdecl _mm_cvt_roundss_u32(__m128d a, int round);
extern unsigned __int64 __cdecl _mm_cvt_roundss_u64(__m128d a, int round);
```

_mm_cvt_ss_u32 / _mm_cvt_ss_u64

```
extern unsigned int __cdecl _mm_cvt_ss_u32(__m128d a);
extern unsigned __int64 __cdecl _mm_cvt_ss_u64(__m128d a);
```

_mm_cvt_roundss_i32 / _mm_cvt_roundss_si32

```
extern int __cdecl _mm_cvt_roundss_i32(__m128d a, int round);
extern int __cdecl _mm_cvt_roundss_si32(__m128d a, int round);
```

_mm_cvtt_ss_i32 / _mm_cvtt_ss_si32

```
extern __int64 __cdecl _mm_cvtt_ss_i32(__m128d);
extern __int64 __cdecl _mm_cvtt_ss_i64(__m128d);
```

_mm_cvtt_roundss_i64 / _mm_cvtt_roundss_si64

```
extern __int64 __cdecl _mm_cvtt_roundss_i64(__m128d, int round);
extern __int64 __cdecl _mm_cvtt_roundss_si64(__m128d, int round);
```

_mm_cvtti64 / _mm_cvttss_si64

```
extern __m128d __cdecl _mm_cvtt_i64_sd(__m128d a, __int64);
extern __m128d __cdecl _mm_cvtt_si64_sd(__m128d a, __int64);
```

_mm_cvtt_roundss_u32 / _mm_cvtt_roundss_u64

```
extern unsigned int __cdecl _mm_cvtt_roundss_u32(__m128d a, int round);
extern unsigned __int64 __cdecl _mm_cvtt_roundss_u64(__m128d a, int round);
```

_mm_cvtt_ss_u32 / _mm_cvtt_ss_u64

```
extern unsigned int __cdecl _mm_cvtt_ss_u32(__m128d a);
extern unsigned __int64 __cdecl _mm_cvtt_ss_u64(__m128d a);
```

_mm512_cvtss_f32

```
float _mm512_cvtss_f32(__m512 a);
```

_mm512_cvtsd_f64

```
double _mm512_cvtsd_f64(__m512d a);
```

Intrinsics for Integer Conversion Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_cvtepi8_epi32`, `_mm512_mask_cvtepi8_epi32`, `_mm512_maskz_cvtepi8_epi32`	Up-converts int8 to int32.	`VPMOVSXBD`
`_mm512_cvtepi8_epi64`, `_mm512_mask_cvtepi8_epi64`, `_mm512_maskz_cvtepi8_epi64`	Up-converts int8 to int64.	`VPMOVSXBQ`
`_mm512_cvtepi16_epi32`, `_mm512_mask_cvtepi16_epi32`, `_mm512_maskz_cvtepi16_epi32`	Up-converts int16 to int32.	`VPMOVSXWD`
`_mm512_cvtepi16_epi64`, `_mm512_mask_cvtepi16_epi64`, `_mm512_maskz_cvtepi16_epi64`	Up-converts int16 to int64.	`VPMOVSXWQ`
`_mm512_cvtepi32_epi8`, `_mm512_mask_cvtepi32_epi8`, `_mm512_maskz_cvtepi32_epi8`	Down-converts int32 to int8.	`VPMOVDB`
`_mm512_cvtsepi32_epi8`, `_mm512_mask_cvtsepi32_epi8`, `_mm512_maskz_cvtsepi32_epi8`	Down-converts signed int32 to int8.	`VPMOVSDB`
`_mm512_cvtusepi32_epi8`, `_mm512_mask_cvtusepi32_epi8`, `_mm512_maskz_cvtusepi32_epi8`	Down-converts unsigned int32 to int8.	`VPMOVUSDB`
`_mm512_cvtepi32_epi16`, `_mm512_mask_cvtepi32_epi16`, `_mm512_maskz_cvtepi32_epi16`	Down-converts int32 to int16.	`VPMOVDW`
`_mm512_cvtsepi32_epi16`, `_mm512_mask_cvtsepi32_epi16`, `_mm512_maskz_cvtsepi32_epi16`	Down-converts signed int32 to int16.	`VPMOVSDW`
`_mm512_cvtusepi32_epi16`, `_mm512_mask_cvtusepi32_epi16`, `_mm512_maskz_cvtusepi32_epi16`	Down-converts unsigned int32 to int16.	`VPMOVUSDW`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_cvtepi32_epi64, _mm512_mask_cvtepi32_epi64, _mm512_maskz_cvtepi32_epi64	Up-converts int32 to int64.	VPMOVSXDQ
_mm512_cvtepi64_epi8, _mm512_mask_cvtepi64_epi8, _mm512_maskz_cvtepi64_epi8	Down-converts int64 to int8.	VPMOVQB
_mm512_cvtsepi64_epi8, _mm512_mask_cvtsepi64_epi8, _mm512_maskz_cvtsepi64_epi8	Down-converts signed int64 to int8.	VPMOVSQB
_mm512_cvtusepi64_epi8, _mm512_mask_cvtusepi64_epi8, _mm512_maskz_cvtusepi64_epi8	Down-converts unsigned int64 to int8.	VPMOVUSQB
_mm512_cvtepi64_epi16, _mm512_mask_cvtepi64_epi16, _mm512_maskz_cvtepi64_epi16	Down-converts int64 to int16.	VPMOVQW
_mm512_cvtsepi64_epi16, _mm512_mask_cvtsepi64_epi16, _mm512_maskz_cvtsepi64_epi16	Down-converts signed int64 to int16.	VPMOVSQW
_mm512_cvtusepi64_epi16, _mm512_mask_cvtusepi64_epi16, _mm512_maskz_cvtusepi64_epi16	Down-converts unsigned int64 to int16.	VPMOVUSQW
_mm512_cvtepi64_epi32, _mm512_mask_cvtepi64_epi32, _mm512_maskz_cvtepi64_epi32	Down-converts int64 to int32.	VPMOVQD
_mm512_cvtsepi64_epi32, _mm512_mask_cvtsepi64_epi32, _mm512_maskz_cvtsepi64_epi32	Down-converts signed int64 to int32.	VPMOVSQD
_mm512_cvtusepi64_epi32, _mm512_mask_cvtusepi64_epi32, _mm512_maskz_cvtusepi64_epi32	Down-converts unsigned int64 to int32.	VPMOVUSQD
_mm512_cvtepu8_epi64, _mm512_mask_cvtepu8_epi64, _mm512_maskz_cvtepu8_epi64	Up-converts unsigned int8 to int64.	VPMOVZXBQ
_mm512_cvtepu16_epi32, _mm512_mask_cvtepu16_epi32, _mm512_maskz_cvtepu16_epi32	Up-converts unsigned int16 to int32.	VPMOVZXWD
_mm512_cvtepu32_epi64, _mm512_mask_cvtepu32_epi64, _mm512_maskz_cvtepu32_epi64	Up-converts unsigned int32 to int64.	VPMOVZXDQ

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_cvtepi32_pd, _mm512_mask_cvtepi32_pd, _mm512_maskz_cvtepi32_pd	Converts int32 to float64.	VCVTDQ2PD
_mm512_cvt_roundepi32_ps, _mm512_mask_cvt_roundepi32_ps, _mm512_maskz_cvt_roundepi32_ps	Converts int32 to float32.	VCVTDQ2PS
_mm512_cvt_roundepu32_ps, _mm512_mask_cvt_roundepu32_ps, _mm512_maskz_cvt_roundepu32_ps	Converts unsigned int32 to float32.	VCVTUDQ2PS
_mm512_cvtepu32_pd, _mm512_mask_cvtepu32_pd, _mm512_maskz_cvtepu32_pd	Converts unsigned int32 to float64.	VCVTUQD2PD
_mm_cvtu32_sd	Converts unsigned int32 to scalar float64.	VCVTUSI2SD
_mm_cvt_roundi64_sd, _mm_cvt_roundu64_sd	Converts rounded int64 to scalar float64.	VCVTSI2SD
_mm_cvt_roundi32_ss, _mm_cvt_roundi64_ss	Converts unsigned int32 to scalar float32.	VCVTSI2SS
_mm_cvt_roundu32_ss, _mm_cvt_roundu64_ss	Converts rounded int64 to scalar float32.	VCVTUSI2SS
_mm512_cvtsi512_si32	Moves the least significant vector element to a scalar 32-bit integer.	MOVD/VMOVD

variable	definition
k	zeromask used as a selector
k	writemask used as a selector
a	first source vector element
b	second source vector element
c	third source vector element
round	Rounding control values; these can be one of the following (along with the sae suppress all exceptions flag): • _MM_FROUND_TO_NEAREST_INT - rounds to nearest even • _MM_FROUND_TO_NEG_INF - rounds to negative infinity • _MM_FROUND_TO_POS_INF - rounds to positive infinity • _MM_FROUND_TO_ZERO - rounds to zero • _MM_FROUND_CUR_DIRECTION - rounds using default from MXCSR register

src	source element

_mm512_cvt_roundpd_epi32

```
extern __m256i __cdecl _mm512_cvt_roundpd_epi32(__m512d a, int round);
```

Converts packed float64 elements in *a* to packed 32-bit integers, and stores the result.

_mm512_cvtpd_epi32

```
extern __m256i __cdecl _mm512_cvtpd_epi32(__m512d a);
```

Converts packed float64 elements in *a* to packed 32-bit integers, and stores the result.

_mm512_mask_cvt_roundpd_epi32

```
extern __m256i __cdecl _mm512_mask_cvt_roundpd_epi32(__m256i src, __mmask8 k, __m512d a, int
round);
```

Converts packed float64 elements in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtpd_epi32

```
extern __m256i __cdecl _mm512_mask_cvtpd_epi32(__m256i src, __mmask8 k, __m512d a);
```

Converts packed float64 elements in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundpd_epi32

```
extern __m256i __cdecl _mm512_maskz_cvt_roundpd_epi32(__mmask8 k, __m512d a, int round);
```

Converts packed float64 elements in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtpd_epi32

```
extern __m256i __cdecl _mm512_maskz_cvtpd_epi32(__mmask8 k, __m512d a);
```

Converts packed float64 elements in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundpd_epu32

```
extern __m256i __cdecl _mm512_cvt_roundpd_epu32(__m512d a, int round);
```

Converts packed float64 elements in *a* to packed unsigned 32-bit integers, and stores the result.

_mm512_cvtpd_epu32

```
extern __m256i __cdecl _mm512_cvtpd_epu32(__m512d a);
```

Converts packed float64 elements in *a* to packed unsigned 32-bit integers, and stores the result.

_mm512_mask_cvt_roundpd_epu32

```
extern __m256i __cdecl _mm512_mask_cvt_roundpd_epu32(__m256i src, __mmask8 k, __m512d a, int round);
```

Converts packed float64 elements in *a* to packed unsigned 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtpd_epu32

```
extern __m256i __cdecl _mm512_mask_cvtpd_epu32(__m256i src, __mmask8 k, __m512d a);
```

Converts packed float64 elements in *a* to packed unsigned 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundpd_epu32

```
extern __m256i __cdecl _mm512_maskz_cvt_roundpd_epu32(__mmask8 k, __m512d a, int round);
```

Converts packed float64 elements in *a* to packed unsigned 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtpd_epu32

```
extern __m256i __cdecl _mm512_maskz_cvtpd_epu32(__mmask8 k, __m512d a);
```

Converts packed float64 elements in *a* to packed unsigned 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_epi32

```
extern __m512i __cdecl _mm512_cvt_roundps_epi32(__m512 a, int round);
```

Converts packed float32 elements in *a* to packed 32-bit integers, and stores the result.

_mm512_cvtps_epi32

```
extern __m512i __cdecl _mm512_cvtps_epi32(__m512 a);
```

Converts packed float32 elements in *a* to packed 32-bit integers, and stores the result.

_mm512_mask_cvt_roundps_epi32

```
extern __m512i __cdecl _mm512_mask_cvt_roundps_epi32(__m512i src, __mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtps_epi32

```
extern __m512i __cdecl _mm512_mask_cvtps_epi32(__m512i src, __mmask16 k, __m512 a);
```

Converts packed float32 elements in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvt_roundps_epi32(__mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtps_epi32(__mmask16 k, __m512 a);
```

Converts packed float32 elements in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_ph

```
extern __m256i __cdecl _mm512_cvt_roundps_ph(__m512 a, int round);
```

Converts packed float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result.

_mm512_cvtps_ph

```
extern __m256i __cdecl _mm512_cvtps_ph(__m512 a, int round);
```

Converts packed float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result.

_mm512_mask_cvt_roundps_ph

```
extern __m256i __cdecl _mm512_mask_cvt_roundps_ph(__m256i src, __mmask16 k, __m512 a, int
round);
```

Converts packed float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtps_ph

```
extern __m256i __cdecl _mm512_mask_cvtps_ph(__m256i src, __mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundps_ph

```
extern __m256i __cdecl _mm512_maskz_cvt_roundps_ph(__mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtps_ph

```
extern __m256i __cdecl _mm512_maskz_cvtps_ph(__mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed half-precision (16-bit) floating-point elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvt_roundps_epu32

```
extern __m512i __cdecl _mm512_cvt_roundps_epu32(__m512 a, int round);
```

Converts packed float32 elements in *a* to packed unsigned 32-bit integers, and stores the result.

_mm512_cvtps_epu32

```
extern __m512i __cdecl _mm512_cvtps_epu32(__m512 a);
```

Converts packed float32 elements in *a* to packed unsigned 32-bit integers, and stores the result.

_mm512_mask_cvt_roundps_epu32

```
extern __m512i __cdecl _mm512_mask_cvt_roundps_epu32(__m512i src, __mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed unsigned 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtps_epu32

```
extern __m512i __cdecl _mm512_mask_cvtps_epu32(__m512i src, __mmask16 k, __m512 a);
```

Converts packed float32 elements in *a* to packed unsigned 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvt_roundps_epu32

```
extern __m512i __cdecl _mm512_maskz_cvt_roundps_epu32(__mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed unsigned 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_maskz_cvtps_epu32

```
extern __m512i __cdecl _mm512_maskz_cvtps_epu32(__mmask16 k, __m512 a);
```

Converts packed float32 elements in *a* to packed unsigned 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvt_roundsd_i32

```
extern int __cdecl _mm_cvt_roundsd_i32(__m128d, int);
```

Convert the lower float64 element in *a* to a 32-bit integer, and stores the result.

_mm_cvt_roundsd_i64

```
extern __int64 __cdecl _mm_cvt_roundsd_i64(__m128d, int);
```

Convert the lower float64 element in *a* to a 64-bit integer, and stores the result.

_mm_cvt_roundsd_si32

```
extern int __cdecl _mm_cvt_roundsd_i32(__m128d, int);
```

Convert the lower float64 element in *a* to a 32-bit integer, and stores the result.

_mm_cvt_roundsd_si64

```
extern __int64 __cdecl _mm_cvt_roundsd_i64(__m128d, int);
```

Convert the lower float64 element in *a* to a 64-bit integer, and stores the result.

_mm_cvtsd_i32

```
extern int __cdecl _mm_cvtsd_i32(__m128d, int);
```

Convert the lower float64 element in *a* to a 32-bit integer, and stores the result.

_mm_cvtsd_i64

```
extern __int64 __cdecl _mm_cvtsd_i64(__m128d, int);
```

Convert the lower float64 element in *a* to a 64-bit integer, and stores the result.

_mm_cvt_roundsd_ss

```
extern __m128 __cdecl _mm_cvtsd_ss(__m128 a, __m128d b, int round);
```

Convert the lower float64 element in *b* to a float32 element, and stores the result in the lower destination element, and copies the upper three packed elements from *a* to the upper destination elements.

_mm_mask_cvt_roundsd_ss

```
extern __m128 __cdecl _mm_mask_cvt_roundsd_ss(__m128 src, __mmask8 k, __m128 a, __m128d b,
int round);
```

Convert the lower float64 element in *b* to a float32 element, and stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_mask_cvtsd_ss

```
extern __m128 __cdecl _mm_mask_cvtsd_ss(__m128 src, __mmask8 k, __m128 a, __m128d b, int
round);
```

Convert the lower float64 element in *b* to a float32 element, and stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *a* to the upper destination element.

_mm_maskz_cvt_roundsd_ss

```
extern __m128 __cdecl _mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a, __m128d b, int round);
```

Convert the lower float64 element in *b* to a float32 element, and stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_maskz_cvtsd_ss

```
extern __m128 __cdecl _mm_maskz_cvtsd_ss(__mmask8 k, __m128 a, __m128d b, int round);
```

Convert the lower float64 element in *b* to a float32 element, and stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *a* to the upper destination elements.

_mm_cvt_roundsd_u32

```
extern unsigned int __cdecl _mm_cvt_roundsd_u32(__m128d a, int round);
```

Convert the lower float64 element in *a* to an unsigned 32-bit integer, and stores the result.

_mm_cvt_roundsd_u64

```
extern unsigned __int64 __cdecl _mm_cvt_roundsd_u64(__m128d a, int round);
```

Convert the lower float64 element in *a* to an unsigned 64-bit integer, and stores the result.

_mm_cvtsd_u32

```
extern unsigned int __cdecl _mm_cvtsd_u32(__m128d a);
```

Convert the lower float64 element in *a* to an unsigned 32-bit integer, and stores the result.

_mm_cvtsd_u64

```
extern unsigned __int64 __cdecl _mm_cvtsd_u64(__m128d a);
```

Convert the lower float64 element in *a* to an unsigned 64-bit integer, and stores the result.

_mm_cvt_roundss_i32

```
extern int __cdecl _mm_cvt_roundss_i32(__m128 a, int round);
```

Convert the lower float32 element in *a* to a 32-bit integer, and stores the result.

_mm_cvt_roundss_i64

```
extern __int64 __cdecl _mm_cvt_roundss_i64(__m128 a, int round);
```

Convert the lower float32 element in *a* to a 64-bit integer, and stores the result.

_mm_cvt_roundss_si32

```
extern int __cdecl _mm_cvt_roundss_si32(__m128 a, int round);
```

Convert the lower float32 element in *a* to a 32-bit integer, and stores the result.

_mm_cvt_roundss_si64

```
extern __int64 __cdecl _mm_cvt_roundss_si64(__m128 a, int round);
```

Convert the lower float32 element in *a* to a 64-bit integer, and stores the result.

_mm_cvtss_i32

```
extern
```

Convert the lower float32 element in *a* to a 32-bit integer, and stores the result.

_mm_cvtss_i64

```
extern
```

Convert the lower float32 element in *a* to a 64-bit integer, and stores the result.

_mm_cvt_roundss_u32

```
extern unsigned int __cdecl _mm_cvt_roundss_u32(__m128 a, int round);
```

Convert the lower float32 element in *a* to an unsigned 32-bit integer, and stores the result.

_mm_cvt_roundss_u64

```
extern unsigned __int64 __cdecl _mm_cvt_roundss_u64(__m128 a, int round);
```

Convert the lower float32 element in *a* to an unsigned 64-bit integer, and stores the result.

_mm_cvtss_u32

```
extern unsigned int __cdecl _mm_cvtss_u32(__m128 a);
```

Convert the lower float32 element in *a* to an unsigned 32-bit integer, and stores the result.

_mm_cvtss_u64

```
extern unsigned __int64 __cdecl _mm_cvtss_u64(__m128 a);
```

Convert the lower float32 element in *a* to an unsigned 64-bit integer, and stores the result.

_mm512_cvtt_roundpd_epi32

```
extern __m256i __cdecl _mm512_cvtt_roundpd_epi32(__m512d a, int round);
```

Converts packed float64 elements in *a* to packed int32 elements with truncation, and stores the result.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttpd_epi32

```
extern __m256i __cdecl _mm512_cvttpd_epi32(__m512d a);
```

Converts packed float64 elements in *a* to packed int32 elements with truncation, and stores the result in *dst*.

_mm512_mask_cvtt_roundpd_epi32

```
extern __m256i __cdecl _mm512_mask_cvtt_roundpd_epi32(__m256i src, __mmask8 k, __m512d a,
int round);
```

Converts packed float64 elements in *a* to packed int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvttpd_epi32

```
extern __m256i __cdecl _mm512_mask_cvttpd_epi32(__m256i src, __mmask8 k, __m512d a, int round);
```

Converts packed float64 elements in *a* to packed int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundpd_epi32

```
extern __m256i __cdecl _mm512_maskz_cvtt_roundpd_epi32(__mmask8 k, __m512d a, int round);
```

Converts packed float64 elements in *a* to packed int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvttpd_epi32

```
extern __m256i __cdecl _mm512_maskz_cvttpd_epi32(__mmask8 k, __m512d a);
```

Converts packed float64 elements in *a* to packed int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundpd_epu32

```
extern __m256i __cdecl _mm512_cvtt_roundpd_epu32(__m512d a, int round);
```

Converts packed float64 elements in *a* to packed unsigned int32 elements with truncation, and stores the result.

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttpd_epu32

```
extern __m256i __cdecl _mm512_cvttpd_epu32(__m512d a);
```

Converts packed float64 elements in *a* to packed unsigned int32 elements with truncation, and stores the result.

_mm512_mask_cvtt_roundpd_epu32

```
extern __m256i __cdecl _mm512_mask_cvtt_roundpd_epu32(__m256i src, __mmask8 k, __m512d a, int round);
```

Converts packed float64 elements in *a* to packed unsigned int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvttpd_epu32

```
extern __m256i __cdecl _mm512_mask_cvttpd_epu32(__m256i src, __mmask8 k, __m512d a);
```

Converts packed float64 elements in *a* to packed unsigned int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundpd_epu32

```
extern __m256i __cdecl _mm512_maskz_cvtt_roundpd_epu32(__mmask8 k, __m512d a, int round);
```

Converts packed float64 elements in *a* to packed unsigned int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvttpd_epu32

```
extern __m256i __cdecl _mm512_maskz_cvtt_roundpd_epu32(__mmask8 k, __m512d a);
```

Converts packed float64 elements in *a* to packed unsigned int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundps_epi32

```
extern __m512i __cdecl _mm512_cvtt_roundps_epi32(__m512 a, int round);
```

Converts packed float32 elements in *a* to packed int32 elements with truncation, and stores the result.

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttps_epi32

```
extern __m512i __cdecl _mm512_cvttps_epi32(__m512 a);
```

Converts packed float32 elements in *a* to packed int32 elements with truncation, and stores the result.

_mm512_mask_cvtt_roundps_epi32

```
extern __m512i __cdecl _mm512_mask_cvtt_roundps_epi32(__m512i src, __mmask16 k, __m512 a,
int round);
```

Converts packed float32 elements in *a* to packed int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvttps_epi32

```
extern __m512i __cdecl _mm512_mask_cvttps_epi32(__m512i src, __mmask16 k, __m512 a);
```

Converts packed float32 elements in *a* to packed int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtt_roundps_epi32(__mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvttps_epi32

```
extern __m512i __cdecl _mm512_maskz_cvttps_epi32(__mmask16 k, __m512 a);
```

Converts packed float32 elements in *a* to packed int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtt_roundps_epu32

```
extern __m512i __cdecl _mm512_cvtt_roundps_epu32(__m512 a, int round);
```

Converts packed float32 elements in *a* to packed unsigned int32 elements with truncation, and stores the result.

NOTE
Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_cvttps_epu32

```
extern __m512i __cdecl _mm512_cvttps_epu32(__m512 a);
```

Converts packed float32 elements in *a* to packed unsigned int32 elements with truncation, and stores the result.

_mm512_mask_cvtt_roundps_epu32

```
    extern __m512i __cdecl _mm512_mask_cvtt_roundps_epu32(__m512i src, __mmask16 k, __m512 a,
int round);
```

Converts packed float32 elements in *a* to packed unsigned int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_mask_cvttps_epu32

```
    extern __m512i __cdecl _mm512_mask_cvttps_epu32(__m512i src, __mmask16 k, __m512 a);
```

Converts packed double-precision (32-bit) floating-point elements in *a* to packed unsigned int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtt_roundps_epu32

```
    extern __m512i __cdecl _mm512_maskz_cvtt_roundps_epu32(__mmask16 k, __m512 a, int round);
```

Converts packed float32 elements in *a* to packed unsigned int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

> **NOTE**
> Pass `__MM_FROUND_NO_EXC` to *sae* to suppress all exceptions.

_mm512_maskz_cvttps_epu32

```
    extern __m512i __cdecl _mm512_maskz_cvttps_epu32(__mmask16 k, __m512 a);
```

Converts packed double-precision (32-bit) floating-point elements in *a* to packed unsigned int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_cvtt_roundsd_i32

```
    extern int __cdecl _mm_cvtt_roundsd_i32(__m128d a, int round);
```

Convert the lower float64 element in *a* to a 32-bit integer with truncation, and stores the result.

_mm_cvtt_roundsd_i64

```
    extern __int64 __cdecl _mm_cvtt_roundsd_i64(__m128d a, int round);
```

Convert the lower float64 element in *a* to a 64-bit integer with truncation, and stores the result.

_mm_cvtt_roundsd_si32

```
    extern int __cdecl _mm_cvtt_roundsd_si32(__m128d a, int round);
```

Convert the lower float64 element in *a* to a 32-bit integer with truncation, and stores the result.

_mm_cvtt_roundsd_si64

```
extern __int64 __cdecl _mm_cvtt_roundsd_si64(__m128d a, int round);
```

Convert the lower float64 element in *a* to a 64-bit integer with truncation, and stores the result.

_mm_cvttsd_i32

```
extern int __cdecl _mm_cvttsd_i32(__m128d a);
```

Convert the lower float64 element in *a* to a 32-bit integer with truncation, and stores the result.

_mm_cvttsd_i64

```
extern __int64 __cdecl _mm_cvttsd_i64(__m128d a);
```

Convert the lower float64 element in *a* to a 64-bit integer with truncation, and stores the result.

_mm_cvtt_roundsd_u32

```
extern unsigned int __cdecl _mm_cvtt_roundsd_u32(__m128d a, int);
```

Convert the lower float64 element in *a* to an unsigned 32-bit integer with truncation, and stores the result.

_mm_cvtt_roundsd_u64

```
extern unsigned __int64 __cdecl _mm_cvtt_roundsd_u64(__m128d a, int);
```

Convert the lower float64 element in *a* to an unsigned 64-bit integer with truncation, and stores the result.

_mm_cvttsd_u32

```
extern unsigned int __cdecl _mm_cvtt_u32(__m128d a, int);
```

Convert the lower float64 element in *a* to an unsigned 32-bit integer with truncation, and stores the result.

_mm_cvttsd_u64

```
extern unsigned __int64 __cdecl _mm_cvttsd_u64(__m128d a, int);
```

Convert the lower float64 element in *a* to an unsigned 64-bit integer with truncation, and stores the result.

_mm_cvtt_roundss_i32

```
extern int __cdecl _mm_cvtt_roundss_i32(__m128 a, int);
```

Convert the lower float32 element in *a* to a 32-bit integer with truncation, and stores the result.

_mm_cvtt_roundss_i64

```
extern __int64 __cdecl _mm_cvtt_roundss_i64(__m128 a, int);
```

Convert the lower float32 element in *a* to a 64-bit integer with truncation, and stores the result.

_mm_cvtt_roundss_si32

```
extern int __cdecl _mm_cvtt_roundss_si32(__m128 a, int);
```

Convert the lower float32 element in *a* to a 32-bit integer with truncation, and stores the result.

_mm_cvtt_roundss_si64

```
extern __int64 __cdecl _mm_cvtt_roundss_si64(__m128 a, int);
```

Convert the lower float32 element in *a* to a 64-bit integer with truncation, and stores the result.

_mm_cvttss_i32

```
extern int __cdecl _mm_cvttss_i32(__m128 a);
```

Convert the lower float32 element in *a* to a 32-bit integer with truncation, and stores the result.

_mm_cvttss_i64

```
extern __int64 __cdecl _mm_cvttss_i64(__m128 a);
```

Convert the lower float32 element in *a* to a 64-bit integer with truncation, and stores the result.

_mm_cvtt_roundss_u32

```
extern unsigned int __cdecl _mm_cvtt_roundss_u32(__m128 a, int);
```

Convert the lower float32 element in *a* to an unsigned 32-bit integer with truncation, and stores the result.

_mm_cvtt_roundss_u64

```
extern unsigned __int64 __cdecl _mm_cvtt_roundss_u64(__m128, int);
```

Convert the lower float32 element in *a* to an unsigned 64-bit integer with truncation, and stores the result.

_mm_cvttss_u32

```
extern unsigned int __cdecl _mm_cvttss_u32(__m128 a);
```

Convert the lower float32 element in *a* to an unsigned 32-bit integer with truncation, and stores the result.

_mm_cvttss_u64

```
extern unsigned __int64 __cdecl _mm_cvttss_u64(__m128);
```

Convert the lower float32 element in *a* to an unsigned 64-bit integer with truncation, and stores the result.

_mm512_cvtepi32_epi8

```
extern __m128i __cdecl _mm512_cvtepi32_epi8(__m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with truncation, and stores the result.

_mm512_mask_cvtepi32_epi8

```
extern __m128i __cdecl _mm512_mask_cvtepi32_epi8(__m128i src, __mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepi32_storeu_epi8

```
extern void __cdecl _mm512_mask_cvtepi32_storeu_epi8(void* base_addr, __mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtepi32_epi8

```
extern __m128i __cdecl _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi32_epi16

```
extern __m256i __cdecl _mm512_cvtepi32_epi16(__m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with truncation, and stores the result.

_mm512_mask_cvtepi32_epi16

```
extern __m256i __cdecl _mm512_mask_cvtepi32_epi16(__m256i src, __mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepi32_storeu_epi16

```
extern void __cdecl _mm512_mask_cvtepi32_storeu_epi16(void* base_addr, __mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtepi32_epi16

```
extern __m256i __cdecl _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi64_epi8

```
extern __m128i __cdecl _mm512_cvtepi64_epi8(__m512i a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with truncation, and stores the result.

_mm512_mask_cvtepi64_epi8

```
extern __m128i __cdecl _mm512_mask_cvtepi64_epi8(__m128i src, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepi64_storeu_epi8

```
extern void __cdecl _mm512_mask_cvtepi64_storeu_epi8(void* base_addr, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtepi64_epi8

```
extern __m128i __cdecl _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi64_epi32

```
extern __m256i __cdecl _mm512_cvtepi64_epi32(__m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with truncation, and stores the result.

_mm512_mask_cvtepi64_epi32

```
extern __m256i __cdecl _mm512_mask_cvtepi64_epi32(__m256i src, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepi64_storeu_epi32

```
extern void __cdecl _mm512_mask_cvtepi64_storeu_epi32(void* base_addr, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtepi64_epi32

```
extern __m256i __cdecl _mm512_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with truncation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi64_epi16

```
extern __m128i __cdecl _mm512_cvtepi64_epi16(__m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with truncation, and stores the result.

_mm512_mask_cvtepi64_epi16

```
extern __m128i __cdecl _mm512_mask_cvtepi64_epi16(__m128i src, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with truncation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtepi64_storeu_epi16

```
extern void __cdecl _mm512_mask_cvtepi64_storeu_epi16(void* base_addr, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with truncation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtepi64_epi16

```
extern __m128i __cdecl _mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with truncation, and stores the result in *dst* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtsepi32_epi8

```
extern __m128i __cdecl _mm512_cvtsepi32_epi8(__m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with signed saturation, and stores the result.

_mm512_mask_cvtsepi32_epi8

```
extern __m128i __cdecl _mm512_mask_cvtsepi32_epi8(__m128i src, __mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with signed saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtsepi32_storeu_epi8

```
extern void __cdecl _mm512_mask_cvtsepi32_storeu_epi8(void* base_addr, __mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtsepi32_epi8

```
extern __m128i __cdecl _mm512_maskz_cvtsepi32_epi8(__mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 8-bit integers with signed saturation, and stores the result in *dst* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtsepi32_epi16

```
extern __m256i __cdecl _mm512_cvtsepi32_epi16(__m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with signed saturation, and stores the result.

_mm512_mask_cvtsepi32_epi16

```
extern __m256i __cdecl _mm512_mask_cvtsepi32_epi16(__m256i src, __mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with signed saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtsepi32_storeu_epi16

```
    extern void __cdecl _mm512_mask_cvtsepi32_storeu_epi16(void* base_addr, __mmask16 k,
__m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtsepi32_epi16

```
    extern __m256i __cdecl _mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
```

Converts packed int32 elements in *a* to packed 16-bit integers with signed saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtsepi64_epi8

```
    extern __m128i __cdecl _mm512_cvtsepi64_epi8(__m512i a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with signed saturation, and stores the result.

_mm512_mask_cvtsepi64_epi8

```
    extern __m128i __cdecl _mm512_mask_cvtsepi64_epi8(__m128i src, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with signed saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtsepi64_storeu_epi8

```
    extern void __cdecl _mm512_mask_cvtsepi64_storeu_epi8(void* base_addr, __mmask8 k, __m512i
a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtsepi64_epi8

```
    extern __m128i __cdecl _mm512_maskz_cvtsepi64_epi8(__mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 8-bit integers with signed saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtsepi64_epi32

```
    extern __m256i __cdecl _mm512_cvtsepi64_epi32(__m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with signed saturation, and stores the result.

_mm512_mask_cvtsepi64_epi32

```
    extern __m256i __cdecl _mm512_mask_cvtsepi64_epi32(__m256i src, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with signed saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtsepi64_storeu_epi32

```
extern void __cdecl _mm512_mask_cvtsepi64_storeu_epi32(void* base_addr, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with signed saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtsepi64_epi32

```
extern __m256i __cdecl _mm512_maskz_cvtsepi64_epi32(__mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed int32 elements with signed saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtsepi64_epi16

```
extern __m128i __cdecl _mm512_cvtsepi64_epi16(__m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with signed saturation, and stores the result.

_mm512_mask_cvtsepi64_epi16

```
extern __m128i __cdecl _mm512_mask_cvtsepi64_epi16(__m128i src, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with signed saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtsepi64_storeu_epi16

```
extern void __cdecl _mm512_mask_cvtsepi64_storeu_epi16(void* base_addr, __mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with signed saturation, and stores the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtsepi64_epi16

```
extern __m128i __cdecl _mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
```

Converts packed int64 elements in *a* to packed 16-bit integers with signed saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi8_epi32

```
extern __m512i __cdecl _mm512_cvtepi8_epi32(__m128i a);
```

Sign extend packed 8-bit integers in *a* to packed 32-bit integers, and stores the result.

_mm512_mask_cvtepi8_epi32

```
extern __m512i __cdecl _mm512_mask_cvtepi8_epi32(__m512i src, __mmask16 k, __m128i a);
```

Sign extend packed 8-bit integers in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepi8_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtepi8_epi32(__mmask16 k, __m128i a);
```

Sign extend packed 8-bit integers in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi8_epi64

```
extern __m512i __cdecl _mm512_cvtepi8_epi64(__m128i a);
```

Sign extend packed 8-bit integers in the low 8 bytes of *a* to packed int64 elements, and stores the result.

_mm512_mask_cvtepi8_epi64

```
extern __m512i __cdecl _mm512_mask_cvtepi8_epi64(__m512i src, __mmask8 k, __m128i a);
```

Sign extend packed 8-bit integers in the low 8 bytes of *a* to packed int64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepi8_epi64

```
extern __m512i __cdecl _mm512_maskz_cvtepi8_epi64(__mmask8 k, __m128i a);
```

Sign extend packed 8-bit integers in the low 8 bytes of *a* to packed int64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi32_epi64

```
extern __m512i __cdecl _mm512_cvtepi32_epi64(__m256i a);
```

Sign extend packed int32 elements in *a* to packed int64 elements, and stores the result.

_mm512_mask_cvtepi32_epi64

```
extern __m512i __cdecl _mm512_mask_cvtepi32_epi64(__m512i src, __mmask8 k, __m256i a);
```

Sign extend packed int32 elements in *a* to packed int64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepi32_epi64

```
extern __m512i __cdecl _mm512_maskz_cvtepi32_epi64(__mmask8 k, __m256i a);
```

Sign extend packed int32 elements in *a* to packed int64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi16_epi32

```
extern __m512i __cdecl _mm512_cvtepi16_epi32(__m256i a);
```

Sign extend packed 16-bit integers in *a* to packed 32-bit integers, and stores the result.

_mm512_mask_cvtepi16_epi32

```
extern __m512i __cdecl _mm512_mask_cvtepi16_epi32(__m512i src, __mmask16 k, __m256i a);
```

Sign extend packed 16-bit integers in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepi16_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtepi16_epi32(__mmask16 k, __m256i a);
```

Sign extend packed 16-bit integers in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepi16_epi64

```
extern __m512i __cdecl _mm512_cvtepi16_epi64(__m128i a);
```

Sign extend packed 16-bit integers in *a* to packed int64 elements, and stores the result.

_mm512_mask_cvtepi16_epi64

```
extern __m512i __cdecl _mm512_mask_cvtepi16_epi64(__m512i src, __mmask8 k, __m128i a);
```

Sign extend packed 16-bit integers in *a* to packed int64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepi16_epi64

```
extern __m512i __cdecl _mm512_maskz_cvtepi16_epi64(__mmask8 k, __m128i a);
```

Sign extend packed 16-bit integers in *a* to packed int64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtusepi32_epi8

```
extern __m128i __cdecl _mm512_cvtusepi32_epi8(__m512i a);
```

Converts packed unsigned int32 elements in *a* to packed unsigned 8-bit integers with unsigned saturation, and stores the result.

_mm512_mask_cvtusepi32_epi8

```
extern __m128i __cdecl _mm512_mask_cvtusepi32_epi8(__m128i src, __mmask16 k, __m512i a);
```

Converts packed unsigned int32 elements in *a* to packed unsigned 8-bit integers with unsigned saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtusepi32_storeu_epi8

```
extern void __cdecl _mm512_mask_cvtusepi32_storeu_epi8(void* base_addr, __mmask16 k,
__m512i a);
```

Converts packed unsigned int32 elements in *a* to packed 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtusepi32_epi8

```
extern __m128i __cdecl _mm512_maskz_cvtusepi32_epi8(__mmask16 k, __m512i a);
```

Converts packed unsigned int32 elements in *a* to packed unsigned 8-bit integers with unsigned saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtusepi32_epi16

```
extern __m256i __cdecl _mm512_cvtusepi32_epi16(__m512i a);
```

Converts packed unsigned int32 elements in *a* to packed unsigned 16-bit integers with unsigned saturation, and stores the result.

_mm512_mask_cvtusepi32_epi16

```
extern __m256i __cdecl _mm512_mask_cvtusepi32_epi16(__m256i src, __mmask16 k, __m512i a);
```

Converts packed unsigned int32 elements in *a* to packed unsigned 16-bit integers with unsigned saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtusepi32_storeu_epi16

```
extern void __cdecl _mm512_mask_cvtusepi32_storeu_epi16(void* base_addr, __mmask16 k,
__m512i a);
```

Converts packed unsigned int32 elements in *a* to packed 16-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtusepi32_epi16

```
extern __m256i __cdecl _mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
```

Converts packed unsigned int32 elements in *a* to packed unsigned 16-bit integers with unsigned saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtusepi64_epi8

```
extern __m128i __cdecl _mm512_cvtusepi64_epi8(__m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned 8-bit integers with unsigned saturation, and stores the result.

_mm512_mask_cvtusepi64_epi8

```
extern __m128i __cdecl _mm512_mask_cvtusepi64_epi8(__m128i src, __mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned 8-bit integers with unsigned saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtusepi64_storeu_epi8

```
extern void __cdecl _mm512_mask_cvtusepi64_storeu_epi8(void* base_addr, __mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed 8-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtusepi64_epi8

```
extern __m128i __cdecl _mm512_maskz_cvtusepi64_epi8(__mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned 8-bit integers with unsigned saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtusepi64_epi32

```
extern __m256i __cdecl _mm512_cvtusepi64_epi32(__m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned int32 elements with unsigned saturation, and stores the result.

_mm512_mask_cvtusepi64_epi32

```
extern __m256i __cdecl _mm512_mask_cvtusepi64_epi32(__m256i src, __mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned int32 elements with unsigned saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtusepi64_storeu_epi32

```
extern void __cdecl _mm512_mask_cvtusepi64_storeu_epi32(void* base_addr, __mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed int32 elements with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtusepi64_epi32

```
extern __m256i __cdecl _mm512_maskz_cvtusepi64_epi32(__mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned int32 elements with unsigned saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtusepi64_epi16

```
extern __m128i __cdecl _mm512_cvtusepi64_epi16(__m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned 16-bit integers with unsigned saturation, and stores the result.

_mm512_mask_cvtusepi64_epi16

```
extern __m128i __cdecl _mm512_mask_cvtusepi64_epi16(__m128i src, __mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned 16-bit integers with unsigned saturation, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_cvtusepi64_storeu_epi16

```
extern void __cdecl _mm512_mask_cvtusepi64_storeu_epi16(void* base_addr, __mmask8 k,
__m512i a);
```

Converts packed unsigned int64 elements in *a* to packed 16-bit integers with unsigned saturation, and store the active results (those with their respective bit set in writemask *k*) to unaligned memory at *base_addr*.

_mm512_maskz_cvtusepi64_epi16

```
extern __m128i __cdecl _mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
```

Converts packed unsigned int64 elements in *a* to packed unsigned 16-bit integers with unsigned saturation, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepu8_epi32

```
extern __m512i __cdecl _mm512_cvtepu8_epi32(__m128i a);
```

Zero extend packed unsigned 8-bit integers in *a* to packed 32-bit integers, and stores the result.

_mm512_mask_cvtepu8_epi32

```
extern __m512i __cdecl _mm512_mask_cvtepu8_epi32(__m512i src, __mmask16 k, __m128i a);
```

Zero extend packed unsigned 8-bit integers in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepu8_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtepu8_epi32(__mmask16 k, __m128i a);
```

Zero extend packed unsigned 8-bit integers in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepu8_epi64

```
extern __m512i __cdecl _mm512_cvtepu8_epi64(__m128i a);
```

Zero extend packed unsigned 8-bit integers in the low 8 byte sof *a* to packed int64 elements, and stores the result.

_mm512_mask_cvtepu8_epi64

```
extern __m512i __cdecl _mm512_mask_cvtepu8_epi64(__m512i src, __mmask8 k, __m128i a);
```

Zero extend packed unsigned 8-bit integers in the low 8 bytes of *a* to packed int64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepu8_epi64

```
extern __m512i __cdecl _mm512_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
```

Zero extend packed unsigned 8-bit integers in the low 8 bytes of *a* to packed int64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepu32_epi64

```
extern __m512i __cdecl _mm512_cvtepu32_epi64(__m256i a);
```

Zero extend packed unsigned int32 elements in *a* to packed int64 elements, and stores the result.

_mm512_mask_cvtepu32_epi64

```
extern __m512i __cdecl _mm512_mask_cvtepu32_epi64(__m512i src, __mmask8 k, __m256i a);
```

Zero extend packed unsigned int32 elements in *a* to packed int64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepu32_epi64

```
extern __m512i __cdecl _mm512_maskz_cvtepu32_epi64(__mmask8 k, __m256i a);
```

Zero extend packed unsigned int32 elements in *a* to packed int64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepu16_epi32

```
extern __m512i __cdecl _mm512_cvtepu16_epi32(__m256i a);
```

Zero extend packed unsigned 16-bit integers in *a* to packed 32-bit integers, and stores the result.

_mm512_mask_cvtepu16_epi32

```
extern __m512i __cdecl _mm512_mask_cvtepu16_epi32(__m512i src, __mmask16 k, __m256i a);
```

Zero extend packed unsigned 16-bit integers in *a* to packed 32-bit integers, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepu16_epi32

```
extern __m512i __cdecl _mm512_maskz_cvtepu16_epi32(__mmask16 k, __m256i a);
```

Zero extend packed unsigned 16-bit integers in *a* to packed 32-bit integers, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtepu16_epi64

```
extern __m512i __cdecl _mm512_cvtepu16_epi64(__m128i a);
```

Zero extend packed unsigned 16-bit integers in *a* to packed int64 elements, and stores the result.

_mm512_mask_cvtepu16_epi64

```
extern __m512i __cdecl _mm512_mask_cvtepu16_epi64(__m512i src, __mmask8 k, __m128i a);
```

Zero extend packed unsigned 16-bit integers in *a* to packed int64 elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_cvtepu16_epi64

```
extern __m512i __cdecl _mm512_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
```

Zero extend packed unsigned 16-bit integers in *a* to packed int64 elements, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_cvtsi512_si32

```
int _mm512_cvtsi512_si32(__m512i a);
```

Moves the least significant 32 bits of *a* to a 32-bit integer.

Intrinsics for Expand and Load Operations

Intrinsics for FP Expand and Load Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_expand_pd`, `_mm512_mask_expand_pd`, `_mm512_maskz_expand_pd`	Load packed float64 values from dense memory.	`VEXPANDPD`
`_mm512_mask_expandloadu_pd`, `_mm512_maskz_expandloadu_pd`	Load packed float64 values from dense memory.	`VEXPANDPD`
`_mm512_expand_ps`, `_mm512_mask_expand_ps`, `_mm512_maskz_expand_ps`	Load packed float32 values from dense memory.	`VEXPANDPS`
`_mm512_mask_expandloadu_ps`, `_mm512_maskz_expandloadu_ps`	Load packed float32 values from dense memory.	`VEXPANDPS`

variable	definition
k	writemask used as a selector

variable	definition
a	first source vector element
src	source element to use based on writemask result
mem_addr	pointer to memory address

_mm512_expand_pd

```
extern __m512d __cdecl _mm512_expand_pd(__m512d a);
```

Loads contiguous active float64 elements from *a* (those with their respective bit set in mask *k*), and stores the result.

_mm512_mask_expand_pd

```
extern __m512d __cdecl _mm512_mask_expand_pd(__m512d src, __mmask8 k, __m512d a);
```

Loads contiguous active float64 elements from *a* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expand_pd

```
extern __m512d __cdecl _mm512_maskz_expand_pd(__mmask8 k, __m512d a);
```

Loads contiguous active float64 elements from *a* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_expand_ps

```
extern __m512 __cdecl _mm512_expand_ps(__m512 a);
```

Loads contiguous active float32 elements from *a* (those with their respective bit set in mask *k*), and stores the result.

_mm512_mask_expand_ps

```
extern __m512 __cdecl _mm512_mask_expand_ps(__m512 src, __mmask16 k, __m512 a);
```

Loads contiguous active float32 elements from *a* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expand_ps

```
extern __m512 __cdecl _mm512_maskz_expand_ps(__mmask16 k, __m512 a);
```

Loads contiguous active float32 elements from *a* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_expandloadu_pd

```
extern __m512d __cdecl _mm512_mask_expandloadu_pd(__m512d src, __mmask8 k, void * mem_addr);
```

Loads contiguous active float64 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expandloadu_pd

```
extern __m512d __cdecl _mm512_maskz_expandloadu_pd(__mmask8 k, void * mem_addr);
```

Loads contiguous active float64 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_expandloadu_ps

```
extern __m512 __cdecl _mm512_mask_expandloadu_ps(__m512 src, __mmask16 k, void * mem_addr);
```

Loads contiguous active float32 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expandloadu_ps

```
extern __m512 __cdecl _mm512_maskz_expandloadu_ps(__mmask16 k, void * mem_addr);
```

Loads contiguous active float32 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Integer Expand and Load Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_mask_expandloadu_epi32`, `_mm512_maskz_expandloadu_epi32` `_mm512_mask_expand_epi32`, `_mm512_maskz_expand_epi32`	Load packed int32 values from dense memory or register.	`VPEXPANDD`
`_mm512_mask_expandloadu_epi64`, `_mm512_maskz_expandloadu_epi64` `_mm512_mask_expand_epi64`, `_mm512_maskz_expand_epi64`	Load packed int64 values from dense memory or register.	`VPEXPANDQ`

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result
mem_addr	pointer to base address in memory

_mm512_mask_expand_epi32

```
extern __m512i __cdecl _mm512_mask_expand_epi32(__m512i src, __mmask16 k, __m512i a);
```

Loads contiguous active int32 elements from *a* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expand_epi32

```
extern __m512i __cdecl _mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
```

Loads contiguous active int32 elements from *a* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_expandloadu_epi32

```
extern __m512i __cdecl _mm512_mask_expandloadu_epi32(__m512i src, __mmask16 k, void * mem_addr);
```

Loads contiguous active int32 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expandloadu_epi32

```
extern __m512i __cdecl _mm512_maskz_expandloadu_epi32( __mmask16 k, void * mem_addr);
```

Loads contiguous active int32 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_expandloadu_epi64

```
extern __m512i __cdecl _mm512_mask_expandloadu_epi64(__m512i src, __mmask8 k, void * mem_addr);
```

Loads contiguous active int64 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expandloadu_epi64

```
extern __m512i __cdecl _mm512_maskz_expandloadu_epi64(__mmask8 k, void * mem_addr);
```

Loads contiguous active int64 elements from unaligned memory at *mem_addr* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_expand_epi64

```
extern __m512i __cdecl _mm512_mask_expand_epi64(__m512i src, __mmask8 k, __m512i a);
```

Loads contiguous active int64 elements from *a* (those with their respective bit set in mask *k*), and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_expand_epi64

```
extern __m512i __cdecl _mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
```

Loads contiguous active int64 elements from *a* (those with their respective bit set in mask *k*), and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Gather and Scatter Operations

Intrinsics for FP Gather and Scatter Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_i32gather_pd`, `_mm512_mask_i32gather_pd`	Gathers double-precision (64-bit) floating-point elements from memory with 32-bit integer indices.	`VGATHERDPD`
`_mm512_i32gather_ps`, `_mm512_mask_i32gather_ps`	Gathers single-precision (32-bit) vector elements from memory with 32-bit integer indices.	`VGATHERDPS`
`_mm512_i32extgather_ps`, `_mm512_mask_i32extgather_ps`	Up-converts single-precision (32-bit) floating-point elements from memory with 32-bit integer indices.	`VGATHERDPS`
`_mm512_i64gather_pd`, `_mm512_mask_i64gather_pd`	Gathers double-precision (64-bit) floating-point elements from memory with 64-bit integer indices.	`VGATHERQPD`
`_mm512_i64gather_ps`, `_mm512_mask_i64gather_ps`	Gathers single-precision (32-bit) vector elements from memory with 64-bit integer indices.	`VGATHERQPS`
`_mm512_prefetch_i32gather_pd`, `_mm512_mask_prefetch_i32gather_pd`	Gathers prefetch double-precision (64-bit) floating-point elements with 32-bit integer indices.	`VGATHERPF0DPD`, `VGATHERPF1DPD`
`_mm512_prefetch_i32gather_ps`, `_mm512_mask_prefetch_i32gather_ps`	Gathers prefetch double-precision (64-bit) floating-point elements with 32-bit integer indices.	`VGATHERPF0DPS`, `VGATHERPF1DPS`
`_mm512_prefetch_i64gather_pd`, `_mm512_mask_prefetch_i64gather_pd`	Gathers prefetch double-precision (64-bit) floating-point elements with 64-bit integer indices.	`VGATHERPF0QPD`, `VGATHERPF1QPD`
`_mm512_prefetch_i64gather_ps`, `_mm512_mask_prefetch_i64gather_ps`	Gathers prefetch double-precision (64-bit) floating-point elements with 64-bit integer indices.	`VGATHERPF0QPS`, `VGATHERPF1QPS`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_i32scatter_pd`, `_mm512_mask_i32scatter_pd`	Scatters double-precision (64-bit) floating-point elements from memory with 32-bit integer indices.	`VSCATTERDPD`
`_mm512_i32scatter_ps`, `_mm512_mask_i32scatter_ps`	Scatters single-precision (32-bit) floating-point elements from memory with 32-bit integer indices.	`VSCATTERDPD`
`_mm512_i32extscatter_ps`, `_mm512_mask_i32extscatter_ps`	Down-converts single-precision (32-bit) floating-point elements from memory with 32-bit integer indices.	`VSCATTERDPS`
`_mm512_i64scatter_pd`, `_mm512_mask_i64scatter_pd`	Scatters double-precision (64-bit) floating-point elements from memory with 64-bit integer indices.	`VSCATTERQPD`
`_mm512_i64scatter_ps`, `_mm512_mask_i64scatter_ps`	Scatters single-precision (32-bit) floating-point elements from memory with 64-bit integer indices.	`VSCATTERQPS`
`_mm512_prefetch_i32scatter_pd`, `_mm512_mask_prefetch_i32scatter_pd`	Scatters prefetch double-precision (64-bit) floating-point elements with 32-bit integer indices.	`VSCATTERPF0DPD`, `VSCATTERPF1DPD`
`_mm512_prefetch_i32scatter_ps`, `_mm512_mask_prefetch_i32scatter_ps`	Scatters prefetch double-precision (64-bit) floating-point elements with 32-bit integer indices.	`VSCATTERPF0DPS`, `VSCATTERPF1DPS`
`_mm512_prefetch_i64scatter_pd`, `_mm512_mask_prefetch_i64scatter_pd`	Scatters prefetch double-precision (64-bit) floating-point elements with 64-bit integer indices.	`VSCATTERPF0QPD`, `VSCATTERPF1QPD`
`_mm512_prefetch_i64scatter_ps`, `_mm512_mask_prefetch_i64scatter_ps`	Scatters prefetch double-precision (64-bit) floating-point elements with 64-bit integer indices.	`VSCATTERPF0QPS`, `VSCATTERPF1QPS`

variable	definition
vindex	a vector of indices
base_addr	a pointer to the base address in memory
scale	a compilation-time literal constant that is used as the vector indices scale. Possible values are 1, 2, 4, or 8.
k	mask used as a selector
a	first source vector element
src	source element to use based on the mask result
upconv	Where _MM_UPCONV_PS_ENUM is the following:

variable	definition
	• _MM_UPCONV_PS_NONE - no conversion
index	a vector containing indexes in memory *mv*
downconv	Where _MM_DOWNCONV_PS_ENUM is the following: • _MM_DOWNCONV_PS_NONE - no conversion
hint	Indicates which cache level to bring values into. _MM_HINT_ENUM is the following: • _MM_HINT_NONE 0x0 - Off

_mm512_i32gather_pd

```
__m512d _mm512_i32gather_pd (__m256i vindex, void const* base_addr, int scale)
```

Gather double-precision (64-bit) floating-point elements from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32gather_pd

```
__m512d _mm512_mask_i32gather_pd (__m512d src, __mmask8 k, __m256i vindex, void const*
base_addr, int scale)
```

Gathers double-precision (64-bit) floating-point elements from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i32gather_ps

```
__m512 _mm512_i32gather_ps (__m512i vindex, void const* base_addr, int scale)
```

Gathers single-precision (32-bit) floating-point elements from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32gather_ps

```
__m512 _mm512_mask_i32gather_ps (__m512 src, __mmask16 k, __m512i vindex, void const* base_addr,
int scale)
```

Gathers single-precision (32-bit) floating-point elements from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i32extgather_ps

```
__m512 _mm512_i32extgather_ps (__m512i index, void const * mv, _MM_UPCONV_PS_ENUM upconv, int
scale, int hint)
```

Up-converts 16 memory locations starting at location *mv* using packed 32-bit integer indices stored in *index* scaled by *scale* using *upconv* to single-precision (32-bit) floating-point elements and stores them in dst.

_mm512_mask_i32extgather_ps

```
__m512 _mm512_mask_i32extgather_ps (__m512 src, __mmask16 k, __m512i index, void const * mv,
_MM_UPCONV_PS_ENUM upconv, int scale, int hint)
```

Up-converts 16 single-precision memory locations starting at location *mv* at packed 32-bit integer indices stored in *index* scaled by *scale* using *upconv* to single-precision (32-bit) floating-point elements and merges them with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i64gather_pd

```
__m512d _mm512_i64gather_pd (__m512i vindex, void const* base_addr, int scale)
```

Gathers double-precision (64-bit) floating-point elements from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64gather_pd

```
__m512d _mm512_mask_i64gather_pd (__m512d src, __mmask8 k, __m512i vindex, void const*
base_addr, int scale)
```

Gathers double-precision (64-bit) floating-point elements from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i64gather_ps

```
__m256 _mm512_i64gather_ps (__m512i vindex, void const* base_addr, int scale)
```

Gathers single-precision (32-bit) floating-point elements from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64gather_ps

```
__m256 _mm512_mask_i64gather_ps (__m256 src, __mmask8 k, __m512i vindex, void const* base_addr,
int scale)
```

Gathers single-precision (32-bit) floating-point elements from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_prefetch_i32gather_pd

```
void _mm512_prefetch_i32gather_pd (__m256i vindex, void const* base_addr, int scale, int hint)
```

Prefetches double-precision (64-bit) floating-point elements from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_prefetch_i32gather_pd

```
void _mm512_mask_prefetch_i32gather_pd (__m256i vindex, __mmask8 mask, void const* base_addr,
int scale, int hint)
```

Prefetches double-precision (64-bit) floating-point elements from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with cache using mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_prefetch_i32gather_ps

```
void _mm512_prefetch_i32gather_ps (__m512i index, void const* mv, int scale, int hint)
```

Prefetches 16 single-precision (32-bit) floating-point elements in memory starting at location *mv* at packed 32-bit integer indices stored in *index* (each index is scaled by the factor in *scale*).

_mm512_mask_prefetch_i32gather_ps

```
void _mm512_mask_prefetch_i32gather_ps (__m512i vindex, __mmask16 mask, void const* base_addr,
int scale, int hint)
```

Prefetches single-precision (32-bit) floating-point elements from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with cache using mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_prefetch_i64gather_pd

```
void _mm512_prefetch_i64gather_pd (__m512i vindex, void const* base_addr, int scale, int hint)
```

Prefetches double-precision (64-bit) floating-point elements from memory into cache level specified by *hint* using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_prefetch_i64gather_pd

```
void _mm512_mask_prefetch_i64gather_pd (__m512i vindex, __mmask8 mask, void const* base_addr,
int scale, int hint)
```

Prefetches double-precision (64-bit) floating-point elements from memory into cache level specified by *hint* using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Prefetched elements are merged with cache using mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_prefetch_i64gather_ps

```
void _mm512_prefetch_i32gather_pd (__m256i vindex, void const* base_addr, int scale, int hint)
```

Prefetches single-precision (32-bit) floating-point elements from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_prefetch_i64gather_ps

```
void _mm512_mask_prefetch_i64gather_ps (__m512i vindex, __mmask8 mask, void const* base_addr,
int scale, int hint)
```

Prefetches single-precision (32-bit) floating-point elements from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with cache using mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_i32scatter_pd

```
void _mm512_i32scatter_pd (void* base_addr, __m256i vindex, __m512d a, int scale)
```

Scatters double-precision (64-bit) floating-point elements from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32scatter_pd

```
void _mm512_mask_i32scatter_pd (void* base_addr, __mmask8 k, __m256i vindex, __m512d a, int
scale)
```

Scatters double-precision (64-bit) floating-point elements from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_i32scatter_ps

```
void _mm512_i32scatter_ps (void* base_addr, __m512i vindex, __m512 a, int scale)
```

Scatters single-precision (32-bit) floating-point elements from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32scatter_ps

```
void _mm512_mask_i32scatter_ps (void* base_addr, __mmask16 k, __m512i vindex, __m512 a, int
scale)
```

Scatters single-precision (32-bit) floating-point elements from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_i32extscatter_ps

```
void _mm512_i32extscatter_ps (void * mv, __m512i index, __m512 v1, _MM_DOWNCONV_PS_ENUM
downconv, int scale, int hint)
```

Down-converts 16 packed single-precision (32-bit) floating-point elements in *v1* and stores them in memory locations starting at location *mv* at packed 32-bit integer indices stored in *index* scaled by *scale* using *downconv*.

_mm512_mask_i32extscatter_ps

```
void _mm512_mask_i32extscatter_ps (void * mv, __mmask16 k, __m512i index, __m512 v1,
_MM_DOWNCONV_PS_ENUM downconv, int scale, int hint)
```

Down-converts 16 packed single-precision (32-bit) floating-point elements in *v1* according to *downconv* and stores them in memory locations starting at location *mv* at packed 32-bit integer indices stored in *index* scaled by *scale* using mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_i64scatter_pd

```
void _mm512_i64scatter_pd (void* base_addr, __m512i vindex, __m512d a, int scale)
```

Scatters double-precision (64-bit) floating-point elements from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64scatter_pd

```
void _mm512_mask_i64scatter_pd (void* base_addr, __mmask8 k, __m512i vindex, __m512d a, int
scale)
```

Scatters double-precision (64-bit) floating-point elements from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_i64scatter_ps

```
void _mm512_i64scatter_ps (void* base_addr, __m512i vindex, __m256 a, int scale)
```

Scatters single-precision (32-bit) floating-point elements from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64scatter_ps

```
void _mm512_mask_i64scatter_ps (void* base_addr, __mmask8 k, __m512i vindex, __m256 a, int scale)
```

Scatters single-precision (32-bit) floating-point elements from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_prefetch_i32scatter_pd

```
void _mm512_prefetch_i32scatter_pd (void* base_addr, __m256i vindex, int scale, int hint)
```

Prefetches double-precision (64-bit) floating-point elements with intent to write using 32-bit indices. 64-bit elements are brought into cache from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_prefetch_i32scatter_pd

```
extern void __cdecl _mm512_mask_prefetch_i32gather_pd(__m256i vindex, __mmask8 k, void const*
base_addr, int scale, int hint);
```

Prefetches double-precision (64-bit) floating-point elements with intent to write using 32-bit indices. 64-bit elements are brought into cache from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_prefetch_i32scatter_ps

```
void _mm512_prefetch_i32scatter_ps (void* mv, __m512i index, int scale, int hint)
```

Prefetches 16 single-precision (32-bit) floating-point elements in memory starting at location *mv* at packed 32-bit integer indices stored in *index* scaled by *scale*.

_mm512_mask_prefetch_i32scatter_ps

```
void _mm512_mask_prefetch_i32scatter_ps (void* mv, __mmask16 k, __m512i index, int scale, int
hint)
```

Prefetches 16 single-precision (32-bit) floating-point elements in memory starting at location *mv* at packed 32-bit integer indices stored in *index* scaled by *scale*. Elements are brought into cache only when their corresponding mask bits in mask *k* are set.

_mm512_prefetch_i64scatter_pd

```
void _mm512_prefetch_i64scatter_pd (void* base_addr, __m512i vindex, int scale, int hint)
```

Prefetches double-precision (64-bit) floating-point elements with intent to write into memory using 64-bit indices. 64-bit elements are brought into cache from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_prefetch_i64scatter_pd

```
void _mm512_mask_prefetch_i64scatter_pd (void* base_addr, __mmask8 mask, __m512i vindex, int
scale, int hint)
```

Prefetches double-precision (64-bit) floating-point elements with intent to write into memory using 64-bit indices. 64-bit elements are brought into cache from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

_mm512_prefetch_i64scatter_ps

```
void _mm512_prefetch_i64scatter_ps (void* base_addr, __m512i vindex, int scale, int hint)
```

Prefetches single-precision (32-bit) floating-point elements with intent to write into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_prefetch_i64scatter_ps

```
void _mm512_mask_prefetch_i64scatter_ps (void* base_addr, __mmask8 mask, __m512i vindex, int
scale, int hint)
```

Prefetches single-precision (32-bit) floating-point elements with intent to write into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. Elements are brought into cache only when their corresponding mask bits are set.

Intrinsics for Integer Gather and Scatter Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_i32gather_epi32`, `_mm512_mask_i32gather_epi32`	Gathers 32-bit integers from memory using 32-bit indices.	`VPGATHERDD`
`_mm512_i32gather_epi64`, `_mm512_mask_i32gather_epi64`	Gathers 64-bit integers from memory using 32-bit indices.	`VPGATHERDQ`
`_mm512_i64gather_epi32`, `_mm512_mask_i64gather_epi32`	Gathers 32-bit integers from memory using 64-bit indices.	`VPGATHERQD`
`_mm512_i64gather_epi64`, `_mm512_mask_i64gather_epi64`	Gathers 64-bit integers from memory using 64-bit indices.	`VPGATHERQQ`
`_mm512_i32scatter_epi32`, `_mm512_mask_i32scatter_epi32`	Scatters 32-bit integers into memory using 32-bit indices.	`VPSCATTERDD`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_i32scatter_epi64`, `_mm512_mask_i32scatter_epi64`	Scatters 64-bit integers into memory using 32-bit indices.	`VPSCATTERDQ`
`_mm512_i64scatter_epi32`, `_mm512_mask_i64scatter_epi32`	Scatters 32-bit integers into memory using 64-bit indices.	`VPSCATTERQD`
`_mm512_i64scatter_epi64`, `_mm512_mask_i64scatter_epi64`	Scatters 64-bit integers into memory using 64-bit indices.	`VPSCATTERQQ`

variable	definition
vindex	a vector of indices
base_addr	a pointer to the base address in memory
scale	a compilation-time literal constant that is used as the vector indices scale. Possible values are 1, 2, 4, or 8.
k	mask used as a selector
a	first source vector element
src	source element to use based on the mask result

_mm512_i32gather_epi32

```
__m512i _mm512_i32gather_epi32(__m512i vindex, void const* base_addr, int scale)
```

Gathers 32-bit integers from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32gather_epi32

```
__m512i _mm512_mask_i32gather_epi32(__m512i src, __mmask16 k, __m512i vindex, void const*
base_addr, int scale)
```

Gathers 32-bit integers from memory using 32-bit indices. 32-bit elements are loaded from addresses starting at `base_addr` and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i32gather_epi64

```
__m512i _mm512_mask_i32gather_epi64 (__m512i src, __mmask8 k, __m256i vindex, void const*
base_addr, int scale)
```

Gathers 64-bit integers from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32gather_epi64

```
__m512i _mm512_mask_i32gather_epi64 (__m512i src, __mmask8 k, __m256i vindex, void const*
base_addr, int scale)
```

Gathers 64-bit integers from memory using 32-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i64gather_epi32

```
__m256i _mm512_i64gather_epi32 (__m512i vindex, void const* base_addr, int scale)
```

Gathers 32-bit integers from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64gather_epi32

```
__m256i _mm512_mask_i64gather_epi32 (__m256i src, __mmask8 k, __m512i vindex, void const*
base_addr, int scale)
```

Gathers 32-bit integers from memory using 64-bit indices. 32-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i64gather_epi64

```
__m512i _mm512_i64gather_epi64 (__m512i vindex, void const* base_addr, int scale)
```

Gathers 64-bit integers from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64gather_epi64

```
__m512i _mm512_mask_i64gather_epi64 (__m512i src, __mmask8 k, __m512i vindex, void const*
base_addr, int scale)
```

Gathers 64-bit integers from memory using 64-bit indices. 64-bit elements are loaded from addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*). Gathered elements are merged with *src* using mask *k*. When the corresponding mask bit is not set, elements are copied from *src*.

_mm512_i32scatter_epi32

```
void mm512_i32scatter_epi32(void* base_addr, __m512i vindex, __m512i a, int scale)
```

Scatters 32-bit integers from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32scatter_epi32

```
void _mm512_mask_i32scatter_epi32(void* base_addr, __mmask16 k, __m512i vindex, __m512i a, int
scale
```

Scatters 32-bit integers from *a* into memory using 32-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. When the corresponding mask bit is not set, elements are not stored.

_mm512_i32scatter_epi64

```
void _mm512_i32scatter_epi64 (void* base_addr, __m256i vindex, __m512i a, int scale)
```

Scatters 64-bit integers from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i32scatter_epi64

```
void _mm512_mask_i32scatter_epi64 (void* base_addr, __mmask8 k, __m256i vindex, __m512i a, int
scale)
```

Scatters 64-bit integers from *a* into memory using 32-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 32-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. When the corresponding mask bit is not set, elements are not stored.

_mm512_i64scatter_epi32

```
void _mm512_i64scatter_epi32 (void* base_addr, __m512i vindex, __m256i a, int scale)
```

Scatters 32-bit integers from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64scatter_epi32

```
void _mm512_mask_i64scatter_epi32 (void* base_addr, __mmask8 k, __m512i vindex, __m256i a, int
scale)
```

Scatters 32-bit integers from *a* into memory using 64-bit indices. 32-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. When the corresponding mask bit is not set, elements are not stored.

_mm512_i64scatter_epi64

```
void _mm512_i64scatter_epi64 (void* base_addr, __m512i vindex, __m512i a, int scale)
```

Scatters 64-bit integers from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*).

_mm512_mask_i64scatter_epi64

```
void _mm512_mask_i64scatter_epi64 (void* base_addr, __mmask8 k, __m512i vindex, __m512i a, int
scale)
```

Scatters 64-bit integers from *a* into memory using 64-bit indices. 64-bit elements are stored at addresses starting at *base_addr* and offset by each 64-bit element in *vindex* (each index is scaled by the factor in *scale*) subject to mask *k*. When the corresponding mask bit is not set, elements are not stored.

Intrinsics for Insert and Extract Operations

Intrinsics for FP Insert and Extract Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_extractf32x4_ps, _mm512_mask_extractf32x4_ps, _mm512_maskz_extractf32x4_ps	Extract float32 values.	VEXTRACTF32X4
_mm512_extractf64x4_pd_mm512_mask_extractf64x4_pd, _mm512_maskz_extractf64x4_pd	Extract float64 values.	VEXTRACTF64X4
_mm_extract_ps	Extract packed float32 values.	EXTRACTPS
_mm512_getmant_pd, _mm512_mask_getmant_pd, _mm512_maskz_getmant_pd _mm512_getmant_round_pd, _mm512_mask_getmant_round_pd, _mm512_maskz_getmant_round_pd	Extract float64 vector of normalized mantissas from float64 vector.	VGETMANTPD
_mm512_getmant_ps, _mm512_mask_getmant_ps, _mm512_maskz_getmant_ps _mm512_getmant_round_ps, _mm512_mask_getmant_round_ps, _mm512_maskz_getmant_round_ps	Extract float32 vector of normalized mantissas from float32 vector.	VGETMANTPS
_mm512_getmant_ss, _mm512_mask_getmant_ss, _mm512_maskz_getmant_ss _mm512_getmant_round_ss, _mm512_mask_getmant_round_ss, _mm512_maskz_getmant_round_ss	Extract float32 vector of normalized mantissas from float32 scalar.	VGETMANTSS
_mm512_getmant_sd, _mm512_mask_getmant_sd, _mm512_maskz_getmant_sd _mm512_getmant_round_sd, _mm512_mask_getmant_round_sd, _mm512_maskz_getmant_round_sd	Extract float64 of normalized mantissas from float64 scalar.	VGETMANTSD
_mm512_insertf32x4, _mm512_mask_insertf32x4, _mm512_maskz_insertf32x4	Insert float32 values.	VINSERTF32X4
_mm512_insertf64x4, _mm512_mask_insertf64x4, _mm512_mask_insertf64x4	Insert float64 values.	VINSERTF64X4
_mm_insert_ps	Insert scalar float32 values.	VINSERTPS/INSERTPS

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
imm	8-bit immediate integer specifies offset for destination
tmp	temporary storage location used during operation
interval	Where `_MM_MANTISSA_NORM_ENUM` can be one of the following: • `_MM_MANT_NORM_1_2` - interval [1, 2) • `_MM_MANT_NORM_p5_2` - interval [1.5, 2) • `_MM_MANT_NORM_p5_1` - interval [1.5, 1) • `_MM_MANT_NORM_p75_1p5` - interval [0.75, 1.5)
sign	Where `_MM_MANTISSA_SIGN_ENUM` can be one of the following: • `_MM_MANT_SIGN_src` - sign = sign(SRC) • `_MM_MANT_SIGN_zero` - sign = 0 • `_MM_MANT_SIGN_nan` - DEST = NaN if sign(SRC) = 1
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_extractf32x4_ps

```
extern __m128 __cdecl _mm512_extractf32x4_ps(__m512 a, int imm);
```

Extracts 128 bits (composed of four packed float32 elements) from *a*, selected with *imm*, and stores the result.

_mm512_mask_extractf32x4_ps

```
extern __m128 __cdecl _mm512_mask_extractf32x4_ps(__m128 src, __mmask8 k, __m512 a, int imm);
```

Extracts 128 bits (composed of four packed float32 elements) from *a*, selected with *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extractf32x4_ps

```
extern __m128 __cdecl _mm512_maskz_extractf32x4_ps(__mmask8 k, __m512, int imm);
```

Extracts 128 bits (composed of four packed float32 elements) from *a*, selected with *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_extractf64x4_pd

```
extern __m256d __cdecl _mm512_extractf64x4_pd(__m512d a, int imm);
```

Extracts 256 bits (composed of four packed float64 elements) from *a*, selected with *imm*, and stores the result.

_mm512_mask_extractf64x4_pd

```
extern __m256d __cdecl _mm512_mask_extractf64x4_pd(__m256d src, __mmask8 k, __m512d a, int imm);
```

Extracts 256 bits (composed of four packed float64 elements) from *a*, selected with *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extractf64x4_pd

```
extern __m256d __cdecl _mm512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, int imm);
```

Extracts 256 bits (composed of four packed float64 elements) from *a*, selected with *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_insertf32x4

```
extern __m512 __cdecl _mm512_insertf32x4(__m512 a, __m128 b, int imm);
```

Copies *a* to destination, then inserts 128 bits (composed of four packed float32 elements) from *b* into destination at the location specified by *imm*.

_mm512_mask_insertf32x4

```
extern __m512 __cdecl _mm512_mask_insertf32x4(__m512 src, __mmask16 k, __m512 a, __m128 b, int
imm);
```

Copies *a* to destination, then inserts 128 bits (composed of four packed float32 elements) from *b* into destination at the location specified by *imm*.

_mm512_maskz_insertf32x4

```
extern __m512 __cdecl _mm512_maskz_insertf32x4(__mmask16 k, __m512 a, __m128 b, int imm);
```

Copies *a* to *tmp*, then inserts 128 bits (composed of four packed float32 elements) from *b* into *tmp* at the location specified by *imm*. Stores *tmp* to destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_insertf64x4

```
extern __m512d __cdecl _mm512_insertf64x4(__m512d a, __m256d b, int imm);
```

Copies *a* to *tmp*, then inserts 128 bits (composed of four packed float32 elements) from *b* into *tmp* at the location specified by *imm*. Stores *tmp* to destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_insertf64x4

```
extern __m512d __cdecl _mm512_mask_insertf64x4(__m512d src, __mmask8 k, __m512d a, __m256d b,
int imm);
```

Copies *a* to destination, then inserts 256 bits (composed of four packed float64 elements) from *b* into destination at the location specified by *imm*.

_mm512_maskz_insertf64x4

```
extern __m512d __cdecl _mm512_maskz_insertf64x4(__mmask8 k, __m512d a, __m256d b, int imm);
```

Copies *a* to *tmp*, then inserts 256 bits (composed of four packed float64 elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_getmant_pd

```
extern __m512d __cdecl _mm512_getmant_pd(__m512d a, _MM_MANTISSA_NORM_ENUM interval,
_MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of packed float64 elements in *a*, and stores the result. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_mask_getmant_pd

```
extern __m512d __cdecl _mm512_mask_getmant_pd(__m512d src, __mmask8 k, __m512d a,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of packed float64 elements in *a*, and stores the result. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_maskz_getmant_pd

```
extern __m512d __cdecl _mm512_maskz_getmant_pd(__mmask8 k, __m512d a, _MM_MANTISSA_NORM_ENUM
interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_getmant_round_pd

```
extern __m512d __cdecl _mm512_getmant_round_pd(__m512d a, _MM_MANTISSA_NORM_ENUM interval,
_MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of packed float64 elements in *a*, and stores the result. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_mask_getmant_round_pd

```
extern __m512d __cdecl _mm512_mask_getmant_round_pd(__m512d src, __mmask8 k, __m512d a,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_maskz_getmant_round_pd

```
extern __m512d __cdecl _mm512_maskz_getmant_round_pd(__mmask8 k, __m512d a,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of packed float64 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_getmant_ps

```
extern __m512 __cdecl _mm512_getmant_ps(__m512 a, _MM_MANTISSA_NORM_ENUM interval,
_MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of packed float32 elements in *a*, and stores the result. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_mask_getmant_ps

```
extern __m512 __cdecl _mm512_mask_getmant_ps(__m512 src, __mmask16 k, __m512 a,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of packed float32 elements in *a*, and stores the result. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_maskz_getmant_ps

```
extern __m512 __cdecl _mm512_maskz_getmant_ps(__mmask16 k, __m512 a, _MM_MANTISSA_NORM_ENUM
interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_getmant_round_ps

```
extern __m512 __cdecl _mm512_getmant_round_ps(__m512 a, _MM_MANTISSA_NORM_ENUM interval,
_MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of packed float32 elements in *a*, and stores the result. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_mask_getmant_round_ps

```
extern __m512 __cdecl _mm512_mask_getmant_round_ps(__m512 src, __mmask16 k, __m512 a,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm512_maskz_getmant_round_ps

```
extern __m512 __cdecl _mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of packed float32 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_getmant_round_sd

```
extern __m128d __cdecl _mm_getmant_round_sd(__m128d a, __m128d b, _MM_MANTISSA_NORM_ENUM
interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of the lower float64 element in *a*, stores the result in the lower destination element, and copies the upper element from *b* to the upper destination element. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_mask_getmant_round_sd

```
extern __m128d __cdecl _mm_mask_getmant_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of the lower float64 element in *a*, store the result in the lower destination element, and copies the upper element from *b* to the upper destination element. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_maskz_getmant_round_sd

```
extern __m128d __cdecl _mm_maskz_getmant_round_sd(__mmask8 k, __m128d a, __m128d b,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of the lower float64 element in *a*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_getmant_sd

```
extern __m128d __cdecl _mm_getmant_sd(__m128d a, __m128d b, _MM_MANTISSA_NORM_ENUM interval,
_MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of the lower float64 element in *a*, store the result in the lower destination element, and copies the upper element from *b* to the upper destination element. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_mask_getmant_sd

```
extern __m128d __cdecl _mm_mask_getmant_sd(__m128d a, __mmask8 k, __m128d b, __m128d c,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalize the mantissas of the lower float64 element in *a*, store the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy the upper element from *b* to the upper destination element. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_maskz_getmant_sd

```
extern __m128d __cdecl _mm_maskz_getmant_sd(__mmask8 k, __m128d a, __m128d b,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of the lower float64 element in *a*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper element from *b* to the upper destination element. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_getmant_round_ss

```
extern __m128 __cdecl _mm_getmant_round_ss(__m128 a, __m128 b, _MM_MANTISSA_NORM_ENUM interval,
_MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of the lower float32 element in *a*, stores the result in the lower destination element, and copies the upper three packed elements from *b* to the upper destination elements. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_mask_getmant_round_ss

```
extern __m128 __cdecl _mm_mask_getmant_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of the lower float32 element in *a*, stores the result in the lower destination element, and copies the upper three packed elements from *b* to the upper destination elements. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_maskz_getmant_round_ss

```
extern __m128 __cdecl _mm_maskz_getmant_round_ss(__mmask8 k, __m128 a, __m128 b,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign, int round);
```

Normalizes the mantissas of the lower float32 element in *a*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_getmant_ss

```
extern __m128 __cdecl _mm_getmant_ss(__m128 a, __m128 b, _MM_MANTISSA_NORM_ENUM interval,
_MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of the lower float32 element in *a*, stores the result in the lower destination element, and copies the upper three packed elements from *b* to the upper destination elements. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_mask_getmant_ss

```
extern __m128 __cdecl _mm_mask_getmant_ss(__m128 a, __mmask8 k, __m128 b, __m128 c,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of the lower float32 element in *a*, stores the result in the lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

_mm_maskz_getmant_ss

```
extern __m128 __cdecl _mm_maskz_getmant_ss(__mmask8 k, __m128 a, __m128 b,
_MM_MANTISSA_NORM_ENUM interval, _MM_MANTISSA_SIGN_ENUM sign);
```

Normalizes the mantissas of the lower float32 element in *a*, stores the result in the lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies the upper three packed elements from *b* to the upper destination elements. This intrinsic essentially calculates `±(2^k)*|x.significand|`, where *k* depends on the interval range defined by *interval* and the sign depends on *sign* and the source sign.

Intrinsics for Integer Insert and Extract Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_extracti32x4_epi32`, `_mm512_mask_extracti32x4_epi32`, `_mm512_maskz_extracti32x4_epi32`	Extracts int32 values.	`VEXTRACTI32X4`
`_mm512_extracti64x4_epi64`, `_mm512_mask_extracti64x4_epi64`, `_mm512_maskz_extracti64x4_epi64`	Extracts int64 values.	`VEXTRACTI64X4`
`_mm512_inserti32x4_epi32`, `_mm512_mask_inserti32x4_epi32`, `_mm512_maskz_inserti32x4_epi32`	Inserts int32 values.	`VINSERTI32X4`
`_mm512_inserti64x4_epi64`, `_mm512_mask_inserti64x4_epi64`, `_mm512_maskz_inserti64x4_epi64`	Inserts int64 values.	`VINSERTI64X4`

variable	definition
k	writemask used as a selector
a	first source vector element
mem_addr	pointer to base address in memory
src	source element to use based on writemask result

variable	definition
tmp	temporary location specified by imm
imm	specifies temporary location *tmp*

_mm512_extracti32x4_epi32

```
extern __m128i __cdecl _mm512_extracti32x4_epi32(__m512i a, int imm);
```

Extracts 128 bits (composed of four packed 32-bit integers) from *a*, selected with *imm*, and stores the result.

_mm512_mask_extracti32x4_epi32

```
extern __m128i __cdecl _mm512_mask_extracti32x4_epi32(__m128i src, __mmask8 k, __m512i a, int
imm);
```

Extracts 128 bits (composed of four packed 32-bit integers) from *a*, selected with *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extracti32x4_epi32

```
extern __m128i __cdecl _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, int imm);
```

Extracts 128 bits (composed of four packed 32-bit integers) from *a*, selected with *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_extracti64x4_epi64

```
extern __m256i __cdecl _mm512_extracti64x4_epi64(__m512i a, int imm);
```

Extracts 256 bits (composed of four packed int64 elements) from *a*, selected with *imm*, and stores the result.

_mm512_mask_extracti64x4_epi64

```
extern __m256i __cdecl _mm512_mask_extracti64x4_epi64(__m256i src, __mmask8 k, __m512i a, int
imm);
```

Extracts 256 bits (composed of four packed int64 elements) from *a*, selected with *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_extracti64x4_epi64

```
extern __m256i __cdecl _mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, int imm);
```

Extracts 256 bits (composed of four packed int64 elements) from *a*, selected with *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_inserti32x4

```
extern __m512i __cdecl _mm512_inserti32x4(__m512i a, __m128i b, int imm);
```

Copies *a* to destination, then inserts 128 bits (composed of four packed 32-bit integers) from *b* into destination at the location specified by *imm*.

_mm512_mask_inserti32x4

```
extern __m512i __cdecl _mm512_mask_inserti32x4(__m512i src, __mmask16 k, __m512i a, __m128i b,
int imm);
```

Copies *a* to *tmp*, then inserts 128 bits (composed of four packed 32-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_inserti32x4

```
extern __m512i __cdecl _mm512_maskz_inserti32x4(__mmask16 k, __m512i a, __m128i b, int imm);
```

Copies *a* to *tmp*, then inserts 256 bits (composed of four packed double-precision (64-bit) floating-point elements) from *b* into *tmp* at the location specified by *imm*.

Store *tmp* to destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_inserti64x4

```
extern __m512i __cdecl _mm512_inserti64x4(__m512i a, __m256i b, int imm);
```

Copies *a* to *tmp*, then inserts 256 bits (composed of four packed int64 elements) from *b* into *tmp* at the location specified by *imm*.

_mm512_mask_inserti64x4

```
extern __m512i __cdecl _mm512_mask_inserti64x4(__m512i src, __mmask8 k, __m512i a, __m256i b,
int imm);
```

Copies *a* to *tmp*, then inserts 256 bits (composed of four packed int64 elements) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_inserti64x4

```
extern __m512i __cdecl _mm512_maskz_inserti64x4(__mmask8 k, __m512i a, __m256i b, int imm);
```

Copies *a* to *tmp*, then inserts 128 bits (composed of four packed 32-bit integers) from *b* into *tmp* at the location specified by *imm*. Store *tmp* to using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Load and Store Operations

Intrinsics for FP Loads and Store Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_load_pd, _mm512_mask_load_pd, _mm512_maskz_load_pd _mm512_store_pd, _mm512_mask_store_pd	Load/store aligned float64 values from memory.	MOVAPD
_mm512_load_ps, _mm512_mask_load_ps, _mm512_maskz_load_ps _mm512_store_ps, _mm512_mask_store_ps	Load/store aligned float32 values from memory.	MOVAPS
_mm_mask_load_sd, _mm_maskz_load_sd _mm_mask_store_sd	Load/store lower float64 values from memory.	VMOVSD
_mm_mask_load_ss, _mm_maskz_load_ss _mm_mask_store_ss	Load/store lower float32 values from memory.	VMOVSS
_mm512_loadu_pd, _mm512_mask_loadu_pd, _mm512_maskz_loadu_pd _mm512_storeu_pd, _mm512_mask_storeu_pd	Load/store unaligned float64 values from memory.	VMOVUPD
_mm512_loadu_ps , _mm512_mask_loadu_ps, _mm512_maskz_loadu_ps _mm512_storeu_ps, _mm512_mask_storeu_ps	Load/store unaligned float32 values from memory.	VMOVUPS
_mm512_stream_pd	Store float64 values using non-temporal hint.	VMOVNTPD
_mm512_stream_ps	Store float32 values using non-temporal hint.	VMOVNTPS

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result
mem_addr	pointer to base address in memory

_mm512_load_pd

```
extern __m512d __cdecl _mm512_load_pd(void const* mem_addr);
```

Loads 512-bits (composed of eight packed float64 elements) from *mem_addr* into destination.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_load_pd

```
extern __m512d __cdecl _mm512_mask_load_pd(__m512d src, __mmask8 k, void const* mem_addr);
```

Loads packed float64 elements from *mem_addr* into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_maskz_load_pd

```
extern __m512d __cdecl _mm512_maskz_load_pd(__mmask8 k, void const* mem_addr);
```

Loads packed float64 elements from *mem_addr* into destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_load_ps

```
extern __m512 __cdecl _mm512_load_ps(void const* mem_addr);
```

Loads 512-bits (composed of sixteen packed float32 elements) from *mem_addr* into destination.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_load_ps

```
extern __m512 __cdecl _mm512_mask_load_ps(__m512 src, __mmask16 k, void const* mem_addr);
```

Loads packed float32 elements from *mem_addr* into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_maskz_load_ps

```
extern __m512 __cdecl _mm512_maskz_load_ps(__mmask16 k, void const*  mem_addr);
```

Loads packed float32 elements from *mem_addr* into destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm_mask_load_sd

```
extern __m128d __cdecl _mm_mask_load_sd(__m128d src, __mmask8 k, const double* mem_addr);
```

Loads float64 element from *mem_addr* into lower element of destination using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and sets upper destination element to zero.

mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

_mm_maskz_load_sd

```
extern __m128d __cdecl _mm_maskz_load_sd(__mmask8 k, const double* mem_addr);
```

Loads a float64 element from *mem_addr* into lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and sets upper destination elements to zero.

mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

_mm_mask_load_ss

```
extern __m128 __cdecl _mm_mask_load_ss(__m128 src, __mmask8 k, const float* mem_addr);
```

Loads float32 element from *mem_addr* into lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and sets upper destination elements to zero.

mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

_mm_maskz_load_ss

```
extern __m128 __cdecl _mm_maskz_load_ss(__mmask8 k, const float* mem_addr);
```

Loads float32 element from *mem_addr* into lower element of destination using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and sets upper destination elements to zero.

mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

_mm512_loadu_pd

```
extern __m512d __cdecl _mm512_loadu_pd(void const* mem_addr);
```

Loads 512-bits (composed of eight packed float64 elements) from *mem_addr* into destination.

mem_addr does not need to be aligned on any particular boundary.

_mm512_mask_loadu_pd

```
extern __m512d __cdecl _mm512_mask_loadu_pd(__m512d src, __mmask8 k, void const* mem_addr);
```

Loads packed float64 elements from *mem_addr* into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

mem_addr does not need to be aligned on any particular boundary.

_mm512_maskz_loadu_pd

```
extern __m512d __cdecl _mm512_maskz_loadu_pd(__mmask8 k, void const* mem_addr);
```

Loads packed float64 elements from *mem_addr* into destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

mem_addr does not need to be aligned on any particular boundary.

_mm512_loadu_ps

```
extern __m512 __cdecl _mm512_loadu_ps(void const* mem_addr);
```

Loads 512-bits (composed of sixteen packed float32 elements) from *mem_addr* into destination.

_mm512_mask_loadu_ps

```
extern __m512 __cdecl _mm512_mask_loadu_ps(__m512 src, __mmask16 k, void const* mem_addr);
```

Loads packed float32 elements from *mem_addr* into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_loadu_ps

```
extern __m512 __cdecl _mm512_maskz_loadu_ps(__mmask16 k, void const* mem_addr);
```

Loads packed float32 elements from *mem_addr* into destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

mem_addr does not need to be aligned on any particular boundary.

_mm512_store_pd

```
extern void __cdecl _mm512_store_pd(void* mem_addr, __m512d a);
```

Stores 512-bits (composed of eight packed float64 elements) from *a* into *mem_addr*.

_mm512_mask_store_pd

```
extern void __cdecl _mm512_mask_store_pd(void* mem_addr, __mmask8 k, __m512d a);
```

Stores packed float64 elements from *a* into *mem_addr* using writemask *k*.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_store_ps

```
extern void __cdecl _mm512_store_ps(void* mem_addr, __m512 a);
```

Store 512-bits (composed of sixteen packed float32 elements) from *a* into *mem_addr*.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_store_ps

```
extern void __cdecl _mm512_mask_store_ps(void* mem_addr, __mmask16 k, __m512 a);
```

Store packed float32 elements from *a* into *mem_addr* using writemask *k*.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_stream_pd

```
extern void __cdecl _mm512_stream_pd(void* mem_addr, __m512d a);
```

Stores 512-bits (composed of eight packed float64 elements) from *a* into *mem_addr* using a non-temporal memory hint.

_mm512_stream_ps

```
extern void __cdecl _mm512_stream_ps(void* mem_addr, __m512 a);
```

Stores 512-bits (composed of sixteen packed float32 elements) from *a* into *mem_addr* using a non-temporal memory hint.

_mm_mask_store_sd

```
extern void __cdecl _mm_mask_store_sd(double* mem_addr, __mmask8 k, __m128d a);
```

Stores lower float64 element from *a* into *mem_addr* using writemask *k*.

mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

_mm_mask_store_ss

```
extern void __cdecl _mm_mask_store_ss(float* mem_addr, __mmask8 k, __m128 a);
```

Stores lower float32 element from *a* into *mem_addr* using writemask *k*.

mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

_mm512_storeu_pd

```
extern void __cdecl _mm512_storeu_pd(void* mem_addr, __m512d a);
```

Stores 512-bits (composed of eight packed float64 elements) from *a* into *mem_addr*.

mem_addr does not need to be aligned on any particular boundary.

_mm512_mask_storeu_pd

```
extern void __cdecl _mm512_mask_storeu_pd(void* mem_addr, __mmask8 k, __m512d a);
```

Stores packed float64 elements from *a* into *mem_addr* using writemask *k*.

mem_addr does not need to be aligned on any particular boundary.

_mm512_storeu_ps

```
extern void __cdecl _mm512_storeu_ps(void* mem_addr, __m512 a);
```

Stores 512-bits (composed of sixteen packed float32 elements) from *a* into *mem_addr*.

mem_addr does not need to be aligned on any particular boundary.

_mm512_mask_storeu_ps

```
extern void __cdecl _mm512_mask_storeu_ps(void* mem_addr, __mmask16 k, __m512 a);
```

Stores packed float32 elements from *a* into *mem_addr* using writemask *k*.

Intrinsics for Integer Load and Store Operations

The prototypes for Intel© Advanced Vector Extensions 512 (Intel© AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_load_epi32`, `_mm512_mask_load_epi32`, `_mm512_maskz_load_epi32`	Load packed int32 elements from memory	`VMOVDQA32`
`_mm512_load_epi64`, `_mm512_mask_load_epi64`, `_mm512_maskz_load_epi64`	Load packed int64 elements from memory	`VMOVDQA64`
`_mm512_loadu_si512`	Unaligned load of 512-bit scalar integer	`VMOVDQU32`
`_mm512_mask_loadu_epi32`, `_mm512_maskz_loadu_epi32`	Unaligned load of packed int32 elements	`VMOVDQU32`
`_mm512_mask_loadu_epi64`, `_mm512_maskz_loadu_epi64`	Unaligned load of packed int64 elements	`VMOVDQU64`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_stream_load_si512`	Load double quadword using non-temporal aligned hint.	`MOVNTDQA`
`_mm512_mask_storeu_epi64`	Store unaligned packed int64 elements	`VMOVDQU64`
`_mm512_stream_si512`	Store packed integer values using non-temporal hint.	`VMOVNTDQA`

variable	definition
k	writemask used as a selector
a	first source vector element
mem_addr	pointer to base address in memory
src	source element to use based on writemask result

_mm512_load_si512

```
extern __m512i __cdecl _mm512_load_si512(void const* mem_addr);
```

Load 512-bits of integer data from memory into destination.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_loadu_si512

```
extern __m512i __cdecl _mm512_loadu_si512(void const* mem_addr);
```

Load 512-bits of integer data from memory into destination.

mem_addr does not need to be aligned on any particular boundary.

_mm512_load_epi32

```
extern __m512i __cdecl _mm512_load_epi32(void const* mem_addr);
```

Load 512-bits (composed of sixteen packed 32-bit integers) from memory into destination.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_load_epi32

```
extern __m512i __cdecl _mm512_mask_load_epi32(__m512i src, __mmask16 k, void const* mem_addr);
```

Load packed int32 elements from memory into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_maskz_load_epi32

```
extern __m512i __cdecl _mm512_maskz_load_epi32(__mmask16 k, void const* mem_addr);
```

Load packed int32 elements from memory into destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_load_epi64

```
extern __m512i __cdecl _mm512_load_epi64(void const* mem_addr);
```

Load 512-bits (composed of eight packed int64 elements) from memory into destination.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_load_epi64

```
extern __m512i __cdecl _mm512_mask_load_epi64(__m512i src, __mmask8 k, void const* mem_addr);
```

Load packed int64 elements from memory into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_maskz_load_epi64

```
extern __m512i __cdecl _mm512_maskz_load_epi64(__mmask8 k, void const* mem_addr);
```

Load packed int64 elements from memory into destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_loadu_epi32

```
extern __m512i __cdecl _mm512_mask_loadu_epi32(__m512i src, __mmask16 k, void const* mem_addr);
```

Load packed int32 elements from memory into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

mem_addr does not need to be aligned on any particular boundary.

_mm512_maskz_loadu_epi32

```
extern __m512i __cdecl _mm512_maskz_loadu_epi32(__mmask16 k, void const* mem_addr);
```

Load packed int32 elements from memory into destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

mem_addr does not need to be aligned on any particular boundary.

_mm512_mask_loadu_epi64

```
extern __m512i __cdecl _mm512_mask_loadu_epi64(__m512i src, __mmask8 k, void const* mem_addr);
```

Load packed int64 elements from memory into destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

mem_addr does not need to be aligned on any particular boundary.

_mm512_stream_load_si512

```
extern __m512i __cdecl _mm512_stream_load_si512(void * mem_addr);
```

Load 512-bits of integer data from memory into destination using a non-temporal memory hint.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_store_epi32

```
extern void __cdecl _mm512_store_epi32(void* mem_addr, __m512i a);
```

Store 512-bits (composed of sixteen packed 32-bit integers) from *a* into memory.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_store_epi32

```
extern void __cdecl _mm512_mask_store_epi32(void* mem_addr, __mmask16 k, __m512i a);
```

Store packed int32 elements from *a* into memory using writemask *k*.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_store_si512

```
extern void __cdecl _mm512_store_si512(void* mem_addr, __m512i a);
```

Store 512-bits of integer data from *a* into memory.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_store_epi64

```
extern void __cdecl _mm512_store_epi64(void* mem_addr, __m512i a);
```

Store 512-bits (composed of eight packed int64 elements) from *a* into memory.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_store_epi64

```
extern void __cdecl _mm512_mask_store_epi64(void* mem_addr, __mmask8 k, __m512i a);
```

Store packed int64 elements from *a* into memory using writemask *k*.

mem_addr must be aligned on a 64-byte boundary or a general-protection exception will be generated.

_mm512_mask_storeu_epi32

```
extern void __cdecl _mm512_mask_storeu_epi32(void* mem_addr, __mmask16 k, __m512i a);
```

Store packed int32 elements from *a* into memory using writemask *k*.

mem_addr does not need to be aligned on any particular boundary.

_mm512_mask_storeu_epi64

```
extern void __cdecl _mm512_mask_storeu_epi64(void* mem_addr, __mmask8 k, __m512i a);
```

Store packed int64 elements from *a* into memory using writemask *k*.

mem_addr does not need to be aligned on any particular boundary.

_mm512_storeu_si512

```
extern void __cdecl _mm512_storeu_si512(void* mem_addr, __m512i a);
```

Store 512-bits of integer data from *a* into memory.

mem_addr does not need to be aligned on any particular boundary.

_mm512_stream_si512

```
extern void __cdecl _mm512_stream_si512(void* mem_addr, __m512i a);
```

Store 512-bits of integer data from *a* into memory using a non-temporal memory hint.

Intrinsics for Miscellaneous Operations

Intrinsics for Miscellaneous FP Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_fixupimm_pd, _mm512_mask_fixupimm_pd, _mm512_maskz_fixupimm_pd _mm512_fixupimm_round_pd, _mm512_mask_fixupimm_round_pd, _mm512_maskz_fixupimm_round_pd	Fixes up packed float64 elements.	VFIXUPIMMPD
_mm512_fixupimm_ps, _mm512_mask_fixupimm_ps, _mm512_maskz_fixupimm_ps _mm512_fixupimm_round_ps, _mm512_mask_fixupimm_round_ps, _mm512_maskz_fixupimm_round_ps	Fixes up packed float32 elements.	VFIXUPIMMPS
_mm_fixupimm_round_sd, _mm_mask_fixupimm_round_sd, _mm_maskz_fixupimm_round_sd	Fixes up scalar float64 elements.	VFIXUPIMMSD
_mm_fixupimm_round_ss, _mm_mask_fixupimm_round_ss, _mm_maskz_fixupimm_round_ss	Fixes up scalar float32 elements.	VFIXUPIMMSS

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm_getexp_round_pd`, `_mm_mask_getexp_round_pd`, `_mm_maskz_getexp_round_pd` `_mm_maskz_getexp_round_pd`	Converts exponent of each packed float64 element to a rounded float64 number representing the integer exponent.	`VGETEXPPD`
`_mm_getexp_round_ps`, `_mm_mask_getexp_round_ps`, `_mm_maskz_getexp_round_ps` `_mm_maskz_getexp_round_ps`	Converts exponent of each packed float32 element to a rounded float32 number representing the integer exponent.	`VGETEXPPS`
`_mm_getexp_round_sd`, `_mm_mask_getexp_round_sd`, `_mm_maskz_getexp_round_sd` `_mm_getexp_round_sd`, `_mm_mask_getexp_round_sd`, `_mm_maskz_getexp_round_sd`	Converts exponent of each scalar float64 element to a rounded scalar float64 number representing the integer exponent.	`VGETEXPSD`
`_mm_getexp_round_ss`, `_mm_mask_getexp_round_ss`, `_mm_maskz_getexp_round_ss`	Converts exponent of each scalar float32 element to a rounded scalar float32 number representing the integer exponent.	`VGETEXPSS`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
c	third source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register
src	source element
imm	reporting flag
src	source element

_mm512_fixupimm_pd

```
extern __m512d __cdecl _mm512_fixupimm_pd(__m512d a, __m512d b, __m512i c, int imm);
```

Fixes up packed float64 elements in *a* and *b* using packed 64-bit integers in *c*, and stores the result.

imm is used to set the required flags reporting.

_mm512_mask_fixupimm_pd

```
extern __m512d __cdecl _mm512_mask_fixupimm_pd(__m512d a, __mmask8 k, __m512d b, __m512i c, int
imm);
```

Fixes up packed float64 elements in *a* and *b* using packed 64-bit integers in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm512_maskz_fixupimm_pd

```
extern __m512d __cdecl _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512d b, __m512i c, int
imm);
```

Fixes up packed float64 elements in *a* and *b* using packed 64-bit integers in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm512_fixupimm_round_pd

```
extern __m512d __cdecl _mm512_fixupimm_round_pd(__m512d a, __m512d b, __m512i c, int imm, int
round);
```

Fixes up packed float64 elements in *a* and *b* using packed 64-bit integers in *c*, and stores the result.

imm is used to set the required flags reporting.

_mm512_mask_fixupimm_round_pd

```
extern __m512d __cdecl _mm512_mask_fixupimm_round_pd(__m512d a, __mmask8 k, __m512d b, __m512i
c, int imm, int round);
```

Fixes up packed float64 elements in *a* and *b* using packed 64-bit integers in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm512_maskz_fixupimm_round_pd

```
extern __m512d __cdecl _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512d b, __m512i
c, int imm, int round);
```

Fixes up packed float64 elements in *a* and *b* using packed 64-bit integers in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm512_fixupimm_ps

```
extern __m512 __cdecl _mm512_fixupimm_ps(__m512 a, __m512 b, __m512i c, int imm);
```

Fixes up packed float32 elements in *a* and *b* using packed 32-bit integers in *c*, and stores the result. *imm* is used to set the required flags reporting.

_mm512_mask_fixupimm_ps

```
extern __m512 __cdecl _mm512_mask_fixupimm_ps(__m512 a, __mmask16 k, __m512 b, __m512i c, int
imm);
```

Fixes up packed float32 elements in *a* and *b* using packed 32-bit integers in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm512_maskz_fixupimm_ps

```
extern __m512 __cdecl _mm512_maskz_fixupimm_ps(__mmask16 k, __m512 a, __m512 b, __m512i c, int
imm);
```

Fixes up packed float32 elements in *a* and *b* using packed 32-bit integers in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm512_fixupimm_round_ps

```
extern __m512 __cdecl _mm512_fixupimm_round_ps(__m512 a, __m512 b, __m512i c, int imm, int
round);
```

Fixes up packed float32 elements in *a* and *b* using packed 32-bit integers in *c*, and stores the result. *imm* is used to set the required flags reporting.

_mm512_mask_fixupimm_round_ps

```
extern __m512 __cdecl _mm512_mask_fixupimm_round_ps(__m512 a, __mmask16 k, __m512 b, __m512i,
int imm, int round);
```

Fixes up packed float32 elements in *a* and *b* using packed 32-bit integers in *c*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm512_maskz_fixupimm_round_ps

```
extern __m512 __cdecl _mm512_maskz_fixupimm_round_ps(__mmask16 k, __m512 a, __m512 b, __m512i,
int imm, int round);
```

Fixes up packed float32 elements in *a* and *b* using packed 32-bit integers in *c*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

imm is used to set the required flags reporting.

_mm_fixupimm_sd

```
extern __m128d __cdecl _mm_fixupimm_sd(__m128d a, __m128d b, __m128i c, int imm);
```

Fixes up lower float64 elements in *a* and *b* using lower 64-bit integer in *c*, stores the result in lower destination element, and copies upper element from *a* to upper destination element.

imm is used to set the required flags reporting.

_mm_mask_fixupimm_sd

```
extern __m128d __cdecl _mm_mask_fixupimm_sd(__m128d a, __mmask8 k, __m128d b, __m128i c, int
imm);
```

Fixes up lower float64 elements in *a* and *b* using lower 64-bit integer in *c*, stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

imm is used to set the required flags reporting.

_mm_maskz_fixupimm_sd

```
extern __m128d __cdecl _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128d b, __m128i c, int
imm);
```

Fixes up lower float64 elements in *a* and *b* using lower 64-bit integer in *c*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

imm is used to set the required flags reporting.

_mm_fixupimm_round_sd

```
extern __m128d __cdecl _mm_fixupimm_round_sd(__m128d a, __m128d b, __m128i c, int imm, int
round);
```

Fixes up lower float64 elements in *a* and *b* using lower 64-bit integer in *c*, stores the result in lower destination element, and copies upper element from *a* to upper destination element.

imm is used to set the required flags reporting.

_mm_mask_fixupimm_round_sd

```
extern __m128d __cdecl _mm_mask_fixupimm_round_sd(__m128d a, __mmask8 k, __m128d b, __m128i c,
int imm, int round);
```

Fixes up lower float64 elements in *a* and *b* using lower 64-bit integer in *c*, stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

imm is used to set the required flags reporting.

_mm_maskz_fixupimm_round_sd

```
extern __m128d __cdecl _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128d b, __m128i c,
int imm, int round);
```

Fixes up lower float64 elements in *a* and *b* using lower 64-bit integer in *c*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

imm is used to set the required flags reporting.

_mm_fixupimm_round_ss

```
extern __m128 __cdecl _mm_fixupimm_round_ss(__m128 a, __m128 b, __m128i c, int imm, int round);
```

Fixes up lower float32 elements in *a* and *b* using lower 32-bit integer in *c*, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements.

imm is used to set the required flags reporting.

_mm_mask_fixupimm_round_ss

```
extern __m128 __cdecl _mm_mask_fixupimm_round_ss(__m128 a, __mmask8 k, __m128 b, __m128i c, int
imm, int round);
```

Fixes up lower float32 elements in *a* and *b* using lower 32-bit integer in *c*, stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

imm is used to set the required flags reporting.

_mm_maskz_fixupimm_round_ss

```
extern __m128 __cdecl _mm_maskz_fixupimm_round_ss(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm, int round);
```

Fixes up lower float32 elements in *a* and *b* using lower 32-bit integer in *c*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

imm is used to set the required flags reporting.

_mm_fixupimm_ss

```
extern __m128 __cdecl _mm_fixupimm_ss(__m128 a, __m128 b, __m128i c, int imm);
```

Fixes up lower float32 elements in *a* and *b* using lower 32-bit integer in *c*, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements.

imm is used to set the required flags reporting.

_mm_mask_fixupimm_ss

```
extern __m128 __cdecl _mm_mask_fixupimm_ss(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm);
```

Fixes up lower float32 elements in *a* and *b* using lower 32-bit integer in *c*, stores the result in lower destination element using writemask *k* (the element is copied from *a* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

imm is used to set the required flags reporting.

_mm_maskz_fixupimm_ss

```
extern __m128 __cdecl _mm_maskz_fixupimm_ss(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm);
```

Fixes up lower float32 elements in *a* and *b* using lower 32-bit integer in *c*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

imm is used to set the required flags reporting.

_mm512_getexp_pd

```
extern __m512d __cdecl _mm512_getexp_pd(__m512d a);
```

Converts the exponent of each packed float64 element in *a* to a float64 number representing the integer exponent, and stores the result. This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_mask_getexp_pd

```
extern __m512d __cdecl _mm512_mask_getexp_pd(__m512d src, __mmask8 k, __m512d a);
```

Converts the exponent of each packed float64 element in *a* to a float64 number representing the integer exponent, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_maskz_getexp_pd

```
extern __m512d __cdecl _mm512_maskz_getexp_pd(__mmask8 k, __m512d a);
```

Converts the exponent of each packed float64 element in *a* to a float64 number representing the integer exponent, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_getexp_round_pd

```
extern __m512d __cdecl _mm512_getexp_round_pd(__m512d a, int round);
```

Converts the exponent of each packed float64 element in *a* to a float64 number representing the integer exponent, and stores the result. This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_mask_getexp_round_pd

```
extern __m512d __cdecl _mm512_mask_getexp_round_pd(__m512d src, __mmask8 a, __m512d, int round);
```

Converts the exponent of each packed float64 element in *a* to a float64 number representing the integer exponent, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_maskz_getexp_round_pd

```
extern __m512d __cdecl _mm512_maskz_getexp_round_pd(__mmask8 k, __m512d a, int round);
```

Converts the exponent of each packed float64 element in *a* to a float64 number representing the integer exponent, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_getexp_ps

```
extern __m512 __cdecl _mm512_getexp_ps(__m512 a);
```

Converts the exponent of each packed float32 element in *a* to a float32 number representing the integer exponent, and stores the result. This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_mask_getexp_ps

```
extern __m512 __cdecl _mm512_mask_getexp_ps(__m512 src, __mmask16 k, __m512 a);
```

Converts the exponent of each packed float32 element in *a* to a float32 number representing the integer exponent, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_maskz_getexp_ps

```
extern __m512d __cdecl _mm512_maskz_getexp_ps(__mmask16 k, __m512 a);
```

Converts the exponent of each packed float32 element in *a* to a float32 number representing the integer exponent, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_getexp_round_ps

```
extern __m512 __cdecl _mm512_getexp_round_ps(__m512 a, int round);
```

Converts the exponent of each packed float32 element in *a* to a float32 number representing the integer exponent, and stores the result. This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_mask_getexp_round_ps

```
extern __m512 __cdecl _mm512_mask_getexp_round_ps(__m512 src, __mmask16 k, __m512 a, int round);
```

Converts the exponent of each packed float32 element in *a* to a float32 number representing the integer exponent, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm512_maskz_getexp_round_ps

```
extern __m512 __cdecl _mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int round);
```

Converts the exponent of each packed float32 element in *a* to a float32 number representing the integer exponent, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set). This intrinsic essentially calculates `floor(log2(x))` for each element.

_mm_getexp_round_sd

```
extern __m128d __cdecl _mm_getexp_round_sd(__m128d a, __m128d b, int round);
```

Converts lower exponent of float64 element in *b* to a float64 number representing the integer exponent, stores the result in lower destination element, and copies upper element from *a* to upper destination element. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_mask_getexp_round_sd

```
extern __m128d __cdecl _mm_mask_getexp_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b,
int round);
```

Converts lower exponent of float64 element in *b* to a float64 number representing the integer exponent, stores the result in lower destination element, and copies upper element from *a* to upper destination element. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_maskz_getexp_round_sd

```
extern __m128d __cdecl _mm_maskz_getexp_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Converts lower exponent of float64 element in *b* to a float64 number representing the integer exponent, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper element from *a* to upper destination element. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_getexp_sd

```
extern __m128d __cdecl _mm_getexp_sd(__m128d a, __m128d b);
```

Converts lower exponent of float64 element in *b* to a float64 number representing the integer exponent, stores the result in lower destination element, and copies upper element from *a* to upper destination element. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_mask_getexp_sd

```
extern __m128d __cdecl _mm_mask_getexp_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Converts lower exponent of float64 element in *b* to a float64 number representing the integer exponent, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper element from *a* to upper destination element. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_maskz_getexp_sd

```
extern __m128d __cdecl _mm_maskz_getexp_sd(__mmask8 k, __m128d a, __m128d b);
```

Converts lower exponent of float64 element in *b* to a float64 number representing the integer exponent, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_getexp_round_ss

```
extern __m128 __cdecl _mm_getexp_round_ss(__m128 a, __m128 b, int round);
```

Converts lower exponent of float32 element in *b* to a float32 number representing the integer exponent, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_mask_getexp_round_ss

```
extern __m128 __cdecl _mm_mask_getexp_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Converts lower exponent of float32 element in *b* to a float32 number representing the integer exponent, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper elements. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_maskz_getexp_round_ss

```
extern __m128 __cdecl _mm_maskz_getexp_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Converts lower exponent of float32 element in *b* to a float32 number representing the integer exponent, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper elements. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_getexp_ss

```
extern __m128 __cdecl _mm_getexp_ss(__m128 a, __m128 b);
```

Converts lower exponent of float32 element in *b* to a float32 number representing the integer exponent, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_mask_getexp_ss

```
extern __m128 __cdecl _mm_mask_getexp_ss(__m128 src, __mmask8 k, __m128 a, __m128b);
```

Converts lower exponent of float32 element in *b* to a float32 number representing the integer exponent, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements. This intrinsic essentially calculates `floor(log2(x))` for lower element.

_mm_maskz_getexp_ss

```
extern __m128 __cdecl _mm_maskz_getexp_ss(__mmask8 k, __m128 a, __m128 b);
```

Converts lower exponent of float32 element in *b* to a float32 number representing the integer exponent, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements. This intrinsic essentially calculates `floor(log2(x))` for lower element.

Intrinsics for Miscellaneous Integer Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_alignr_epi32`, `_mm512_mask_alignr_epi32`, `_mm512_maskz_alignr_epi32`	Aligns elements of two source vectors depending on bits in a mask.	`VALIGND`
`_mm512_alignr_epi64`, `_mm512_mask_alignr_epi64`, `_mm512_maskz_alignr_epi64`	Aligns elements of two source vectors depending on bits in a mask.	`VALIGNQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
count	specifies the number of bits for shift operation

_mm512_alignr_epi32

```
extern __m512i __cdecl _mm512_alignr_epi32(__m512i a, __m512i b, const int count);
```

Concatenates vector elements from *a* and *b* into a 128-byte immediate result, shifts the result right by *count* of 32-bit elements, and stores the low 64 bytes (sixteen elements).

_mm512_mask_alignr_epi32

```
extern __m512i __cdecl _mm512_mask_alignr_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b,
const int count);
```

Concatenates vector elements from *a* and *b* into a 128-byte immediate result, shifts the result right by *count* of 32-bit elements, and stores the low 64 bytes (sixteen elements) using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_alignr_epi32

```
extern __m512i __cdecl _mm512_maskz_alignr_epi32(__mmask16 k, __m512i a, __m512i b, const int
count);
```

Concatenates vector elements from *a* and *b* into a 128-byte immediate result, shifts the result right by *count* of 32-bit elements, and stores the low 64 bytes (sixteen elements) using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_alignr_epi64

```
extern __m512i __cdecl _mm512_alignr_epi64(__m512i a, __m512i b, const int count);
```

Concatenates vector elements from *a* and *b* into a 128-byte immediate result, shifts the result right by *count* of 64-bit elements, and stores the low 64 bytes (eight elements).

_mm512_mask_alignr_epi64

```
extern __m512i __cdecl _mm512_mask_alignr_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b,
const int count);
```

Concatenates vector elements from *a* and *b* into a 128-byte immediate result, shifts the result right by *count* of 64-bit elements, and stores the low 64 bytes (eight elements) using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_alignr_epi64

```
extern __m512i __cdecl _mm512_maskz_alignr_epi64(__mmask8 k, __m512i a, __m512i b, const int
count);
```

Concatenates vector elements from *a* and *b* into a 128-byte immediate result, shifts the result right by *count* of 64-bit elements, and stores the low 64 bytes (eight elements) using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Move Operations

Intrinsics for FP Move Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_mask_mov_pd`, `_mm512_maskz_mov_pd`	Moves packed float64 elements.	`VMOVAPD`
`_mm512_mask_mov_ps`, `_mm512_maskz_mov_ps`	Moves packed float32 elements.	`VMOVAPS`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm_mask_move_sd`, `_mm_maskz_move_sd`	Moves scalar float64 elements.	VMOVSD
`_mm_mask_move_ss`, `_mm_maskz_move_ss`	Moves scalar float32 elements.	VMOVSS
`_mm512_movedup_pd`, `_mm512_mask_movedup_pd`, `_mm512_maskz_movedup_pd`	Duplicates even-indexed float64 elements.	VMOVDDUP
`_mm512_movehdup_ps`, `_mm512_mask_movehdup_ps`, `_mm512_maskz_movehdup_ps`	Duplicates odd-indexed float32 elements.	VMOVSHDUP
`_mm512_moveldup_ps`, `_mm512_mask_moveldup_ps`, `_mm512_maskz_moveldup_ps`	Moves lower float32 element.	VMOVSLDUP

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_mask_mov_pd

```
extern __m512d __cdecl _mm512_mask_mov_pd(__m512d src, __mmask8 k, __m512d a);
```

Moves packed float64 elements from *a* using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mov_pd

```
extern __m512d __cdecl _mm512_maskz_mov_pd(__mmask8 k, __m512d a);
```

Moves packed float64 elements from *a* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mov_ps

```
extern __m512 __cdecl _mm512_mask_mov_ps(__m512 src, __mmask16 k, __m512 a);
```

Moves packed float32 elements from *a* using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mov_ps

```
extern __m512 __cdecl _mm512_maskz_mov_ps(__mmask16 k, __m512 a);
```

Moves packed float32 elements from *a* using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_movedup_pd

```
extern __m512d __cdecl _mm512_movedup_pd(__m512d a);
```

Duplicates even-indexed float64 elements from *a*, and stores the result.

_mm512_mask_movedup_pd

```
extern __m512d __cdecl _mm512_mask_movedup_pd(__m512d src, __mmask8 k, __m512d a);
```

Duplicates even-indexed float64 elements from *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_movedup_pd

```
extern __m512d __cdecl _mm512_maskz_movedup_pd(__mmask8 k, __m512d a);
```

Duplicates even-indexed float64 elements from *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_move_sd

```
extern __m128d __cdecl _mm_mask_move_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Moves lower float64 element from *b* to lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copy upper element from *a* to upper destination element.

_mm_maskz_move_sd

```
extern __m128d __cdecl _mm_maskz_move_sd(__mmask8 k, __m128d a, __m128d b);
```

Moves lower float64 element from *b* to lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copy upper element from *a* to upper destination element.

_mm512_movehdup_ps

```
extern __m512 __cdecl _mm512_movehdup_ps(__m512 a);
```

Duplicates odd-indexed float32 elements from *a*, and stores the result.

_mm512_mask_movehdup_ps

```
extern __m512 __cdecl _mm512_mask_movehdup_ps(__m512 src, __mmask16 k, __m512 a);
```

Duplicates odd-indexed float32 elements from *a*, and store the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_movehdup_ps

```
extern __m512 __cdecl _mm512_maskz_movehdup_ps(__mmask16 k, __m512 a);
```

Duplicates odd-indexed float32 elements from *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_moveldup_ps

```
extern __m512 __cdecl _mm512_moveldup_ps(__m512 a);
```

Duplicates even-indexed float32 elements from *a*, and stores the result.

_mm512_mask_moveldup_ps

```
extern __m512 __cdecl _mm512_mask_moveldup_ps(__m512 src, __mmask16 k, __m512 a);
```

Duplicates even-indexed float32 elements from *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_moveldup_ps

```
extern __m512 __cdecl _mm512_maskz_moveldup_ps(__mmask16 k, __m512 a);
```

Duplicates even-indexed float32 elements from *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_move_ss

```
extern __m128 __cdecl _mm_mask_move_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Moves lower float32 element from *b* to lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_move_ss

```
extern __m128 __cdecl _mm_maskz_move_ss(__mmask8 k, __m128 a, __m128 b);
```

Moves lower float32 element from *b* to lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

Intrinsics for Integer Move Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_mask_mov_epi32, _mm512_maskz_mov_epi32	Move packed int32 elements.	VMOVDQA32
_mm512_mask_mov_epi64, _mm512_maskz_mov_epi64	Move packed int64 elements.	VMOVQA64

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result

_mm512_mask_mov_epi32

```
extern __m512i __cdecl _mm512_mask_mov_epi32(__m512i a, __mmask16 k, __m512i src);
```

Move packed int32 elements from *a* to destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mov_epi32

```
extern __m512i __cdecl _mm512_maskz_mov_epi32(__mmask16 k, __m512i a);
```

Move packed int32 elements from *a* to destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_mask_mov_epi64

```
extern __m512i __cdecl _mm512_mask_mov_epi64(__m512i a, __mmask16 k, __m512i src);
```

Move packed int64 elements from *a* to destination using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_mov_epi64

```
extern __m512i __cdecl _mm512_maskz_mov_epi64(__mmask8 k, __m512i a);
```

Move packed int64 elements from *a* to destination using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Pack and Unpack Operations

Intrinsics for FP Pack and Unpack Operations

The prototypes for Intel© Advanced Vector Extensions 512 (Intel© AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_unpackhi_pd, _mm512_mask_unpackhi_pd, _mm512_maskz_unpackhi_pd	Unpacks and interleaves high packed float64 values.	VPUNPCKHPD
_mm512_unpackhi_ps, _mm512_mask_unpackhi_ps, _mm512_maskz_unpackhi_ps	Unpacks and interleaves high packed float32 values.	VPUNPCKHPS
_mm512_unpacklo_pd, _mm512_mask_unpacklo_pd, _mm512_maskz_unpacklo_pd	Unpacks and interleaves low packed float64 values.	VPUNPCKLPD
_mm512_unpacklo_ps, _mm512_mask_unpacklo_ps, _mm512_maskz_unpacklo_ps	Unpacks and interleaves low packed float32 values.	VPUNPCKLPS

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_unpackhi_pd

```
extern __m512d __cdecl _mm512_unpackhi_pd(__m512d a, __m512d b);
```

Unpacks and interleaves float64 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpackhi_pd

```
extern __m512d __cdecl _mm512_mask_unpackhi_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Unpacks and interleaves float64 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpackhi_pd

```
extern __m512d __cdecl _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
```

Unpacks and interleaves float64 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpackhi_ps

```
extern __m512 __cdecl _mm512_unpackhi_ps(__m512 a, __m512 b);
```

Unpacks and interleaves float32 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpackhi_ps

```
extern __m512 __cdecl _mm512_mask_unpackhi_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Unpacks and interleaves float32 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpackhi_ps

```
extern __m512 __cdecl _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
```

Unpacks and interleaves float32 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpacklo_pd

```
extern __m512d __cdecl _mm512_unpacklo_pd(__m512d a, __m512d b);
```

Unpacks and interleaves float64 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpacklo_pd

```
extern __m512d __cdecl _mm512_mask_unpacklo_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Unpacks and interleaves float64 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpacklo_pd

```
extern __m512d __cdecl _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
```

Unpacks and interleaves float64 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpacklo_ps

```
extern __m512 __cdecl _mm512_unpacklo_ps(__m512 a, __m512 b);
```

Unpacks and interleaves float32 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpacklo_ps

```
extern __m512 __cdecl _mm512_mask_unpacklo_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Unpacks and interleaves float32 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpacklo_ps

```
extern __m512 __cdecl _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
```

Unpacks and interleaves float32 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Integer Pack and Unpack Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_unpackhi_epi32`, `_mm512_mask_unpackhi_epi32`, `_mm512_maskz_unpackhi_epi32`	Unpacks and interleaves high packed int32 values.	`VPUNPCKHQD`, `VUNPCKHQD`, `UNPCKHQD`
`_mm512_unpackhi_epi64`, `_mm512_mask_unpackhi_epi64`, `_mm512_maskz_unpackhi_epi64`	Unpacks and interleaves high packed int64 values.	`VPUNPCKHQDQ`, `VUNPCKHQDQ`, `UNPCKHQDQ`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_unpacklo_epi32`, `_mm512_mask_unpacklo_epi32`, `_mm512_maskz_unpacklo_epi32`	Unpacks and interleaves low packed int32 values.	`VPUNPCKLQD`, `VUNPCKLQD`, `UNPCKLQD`
`_mm512_unpacklo_epi64`, `_mm512_mask_unpacklo_epi64`, `_mm512_maskz_unpacklo_epi64`	Unpacks and interleaves low packed int64 values.	`VPUNPCKLQDQ`, `VUNPCKLQDQ`, `UNPCKLQDQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_unpackhi_epi32

```
extern __m512i __cdecl _mm512_unpackhi_epi32(__m512i a, __m512i b);
```

Unpacks and interleaves int32 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpackhi_epi32

```
extern __m512i __cdecl _mm512_mask_unpackhi_epi32(__m512i src, __mmask16 k, __m512i a, __m512i
b);
```

Unpacks and interleaves int32 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpackhi_epi32

```
extern __m512i __cdecl _mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
```

Unpacks and interleaves int32 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpackhi_epi64

```
extern __m512i __cdecl _mm512_unpackhi_epi64(__m512i a, __m512i b);
```

Unpacks and interleaves int64 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpackhi_epi64

```
extern __m512i __cdecl _mm512_mask_unpackhi_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Unpacks and interleaves int64 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpackhi_epi64

```
extern __m512i __cdecl _mm512_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
```

Unpacks and interleaves int64 elements from the high half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpacklo_epi32

```
extern __m512i __cdecl _mm512_unpacklo_epi32(__m512i a, __m512i b);
```

Unpacks and interleaves int32 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpacklo_epi32

```
extern __m512i __cdecl _mm512_mask_unpacklo_epi32(__m512i src, __mmask16 k, __m512i a, __m512i
b);
```

Unpacks and interleaves int32 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpacklo_epi32

```
extern __m512i __cdecl _mm512_maskz_unpacklo_epi32(__mmask16 k, __m512i a, __m512i b);
```

Unpacks and interleaves int32 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_unpacklo_epi64

```
extern __m512i __cdecl _mm512_unpacklo_epi64(__m512i a, __m512i b);
```

Unpacks and interleaves int64 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result.

_mm512_mask_unpacklo_epi64

```
extern __m512i __cdecl _mm512_mask_unpacklo_epi64(__m512i src, __mmask8 k, __m512i a, __m512i b);
```

Unpacks and interleaves int64 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_unpacklo_epi64

```
extern __m512i __cdecl _mm512_maskz_unpacklo_epi64(__mmask8 k, __m512i a, __m512i b);
```

Unpacks and interleaves int64 elements from the low half of each 128-bit lane in *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Permutation Operations

Intrinsics for FP Permutation Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_permutex2var_pd, _mm512_mask_permutex2var_pd, _mm512_mask2_permutex2var_pd, _mm512_maskz_permutex2var_pd	Shuffle float64 elements across lanes.	VPERMI2PD
_mm512_permutex2var_ps, _mm512_mask_permutex2var_ps, _mm512_mask2_permutex2var_ps, _mm512_maskz_permutex2var_ps	Shuffle float32 elements across lanes.	VPERMI2PS
_mm512_permute_pd, _mm512_mask_permute_pd, _mm512_maskz_permute_pd	Shuffle float64 elements within 128-bit lanes.	VPERMILPD, VPERMPD
_mm512_permutevar_pd, _mm512_mask_permutevar_pd, _mm512_maskz_permutevar_pd	Shuffle float64 elements within 128-bit lanes.	VPERMPD
_mm512_permutex_pd, _mm512_mask_permutex_pd, _mm512_maskz_permutex_pd	Shuffle float64 elements within lanes.	VPERMPD
_mm512_permutexvar_pd, _mm512_mask_permutexvar_pd, _mm512_maskz_permutexvar_pd	Shuffle float64 elements across lanes.	VPERMPD
_mm512_permute_ps, _mm512_mask_permute_ps, _mm512_maskz_permute_ps	Shuffle float32 elements within lanes.	VPERMILPS
_mm512_permutevar_ps, _mm512_mask_permutevar_ps, _mm512_maskz_permutevar_ps	Shuffle float32 elements within lanes.	VPERMPS, VPERMILPS
_mm512_permutexvar_ps, _mm512_mask_permutexvar_ps, _mm512_maskz_permutexvar_ps	Shuffle float32 elements across lanes.	VPERMPS

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
idx	index

variable	definition

_mm512_permutex2var_pd

```
extern __m512d __cdecl _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
```

Shuffles float64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result.

_mm512_mask_permutex2var_pd

```
extern __m512d __cdecl _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d
b);
```

Shuffles float64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask2_permutex2var_pd

```
extern __m512d __cdecl _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d
b);
```

Shuffles float64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the results using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set)

_mm512_maskz_permutex2var_pd

```
extern __m512d __cdecl _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d
b);
```

Shuffles float64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutex2var_ps

```
extern __m512 __cdecl _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
```

Shuffles float32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result.

_mm512_mask2_permutex2var_ps

```
extern __m512 __cdecl _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
```

Shuffles float32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm512_mask_permutex2var_ps

```
extern __m512 __cdecl _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
```

Shuffles float32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_maskz_permutex2var_ps

```
extern __m512 __cdecl _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
```

Shuffles float32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permute_pd

```
extern __m512d __cdecl _mm512_permute_pd(__m512d a, const int imm);
```

Shuffles float64 elements in *a* within 128-bit lanes using the control in *imm*, and stores the result.

_mm512_mask_permute_pd

```
extern __m512d __cdecl _mm512_mask_permute_pd(__m512d src, __mmask8 k, __m512d a, const int imm);
```

Shuffles float64 elements in *a* within 128-bit lanes using the control in *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permute_pd

```
extern __m512d __cdecl _mm512_maskz_permute_pd(__mmask8 k, __m512d a, const int imm);
```

Shuffles float64 elements in *a* within 128-bit lanes using the control in *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutevar_pd

```
extern __m512d __cdecl _mm512_permutevar_pd(__m512d a, __m512i b);
```

Shuffles float64 elements in *a* within 128-bit lanes using the control in *b*, and stores the result.

_mm512_mask_permutevar_pd

```
extern __m512d __cdecl _mm512_mask_permutevar_pd(__m512d src, __mmask8 k, __m512d a, __m512i b);
```

Shuffles float64 elements in *a* within 128-bit lanes using the control in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutevar_pd

```
extern __m512d __cdecl _mm512_maskz_permutevar_pd(__mmask8 k, __m512d a, __m512i b);
```

Shuffles float64 elements in *a* within 128-bit lanes using the control in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permute_ps

```
extern __m512 __cdecl _mm512_permute_ps(__m512 a, const int imm);
```

Shuffles float32 elements in *a* within 128-bit lanes using the control in *imm*, and stores the result.

_mm512_mask_permute_ps

```
extern __m512 __cdecl _mm512_mask_permute_ps(__m512 src, __mmask16 k, __m512 a, const int imm);
```

Shuffles float32 elements in *a* within 128-bit lanes using the control in *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permute_ps

```
extern __m512 __cdecl _mm512_maskz_permute_ps(__mmask16 k, __m512 a, const int imm);
```

Shuffles float32 elements in *a* within 128-bit lanes using the control in *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutevar_ps

```
extern __m512 __cdecl _mm512_permutevar_ps(__m512 a, __m512i b);
```

Shuffles float32 elements in *a* within 128-bit lanes using the control in *b*, and stores the result.

_mm512_mask_permutevar_ps

```
extern __m512 __cdecl _mm512_mask_permutevar_ps(__m512 src, __mmask16 k, __m512 a, __m512i b);
```

Shuffles float32 elements in *a* within 128-bit lanes using the control in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutevar_ps

```
extern __m512 __cdecl _mm512_maskz_permutevar_ps(__mmask16 k, __m512 a, __m512i b);
```

Shuffles float32 elements in *a* within 128-bit lanes using the control in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutex_pd

```
extern __m512d __cdecl _mm512_permutex_pd(__m512d a, const int imm);
```

Shuffles float64 elements in *a* within 256-bit lanes using the control in *imm*, and stores the result.

_mm512_mask_permutex_pd

```
extern __m512d __cdecl _mm512_mask_permutex_pd(__m512d src, __mmask8 k, __m512d a, const int
imm);
```

Shuffles float64 elements in *a* within 256-bit lanes using the control in *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutex_pd

```
extern __m512d __cdecl _mm512_maskz_permutex_pd(__mmask8 k, __m512d a, const int imm);
```

Shuffles float64 elements in *a* within 256-bit lanes using the control in *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutexvar_pd

```
extern __m512d __cdecl _mm512_permutexvar_pd(__m512i idx, __m512d a);
```

Shuffles float64 elements in *a* across lanes using the corresponding index in *idx*, and stores the result.

_mm512_mask_permutexvar_pd

```
extern __m512d __cdecl _mm512_mask_permutexvar_pd(__m512d src, __mmask8 k, __m512i idx, __m512d
a);
```

Shuffles float64 elements in *a* across lanes using the corresponding index in *idx*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutexvar_pd

```
extern __m512d __cdecl _mm512_maskz_permutexvar_pd(__mmask8 k, __m512i idx, __m512d a);
```

Shuffles float64 elements in *a* across lanes using the corresponding index in *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutexvar_ps

```
extern __m512 __cdecl _mm512_permutexvar_ps(__m512i idx, __m512 a);
```

Shuffles float32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result.

_mm512_mask_permutexvar_ps

```
extern __m512 __cdecl _mm512_mask_permutexvar_ps(__m512 src, __mmask16 k, __m512i idx, __m512 a);
```

Shuffles float32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutexvar_ps

```
extern __m512 __cdecl _mm512_maskz_permutexvar_ps(__mmask16 k, __m512i idx, __m512 a);
```

Shuffles float32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Integer Permutation Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_permutex2var_epi32`, `_mm512_mask_permutex2var_epi32`, `_mm512_mask2_permutex2var_epi32`, `_mm512_maskz_permutex2var_epi32`	Shuffle int32 elements across lanes.	`VPERMI2D`
`_mm512_permutex2var_epi64`, `_mm512_mask_permutex2var_epi64`, `_mm512_mask2_permutex2var_epi64`, `_mm512_maskz_permutex2var_epi64`	Shuffle int64 elements across lanes.	`VPERMI2Q`, `VPERMT2Q`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction

`_mm512_permutevar_epi32`, `_mm512_mask_permutevar_epi32`
 	
`_mm512_permutexvar_epi32`, `_mm512_mask_permutexvar_epi32`, `_mm512_maskz_permutexvar_epi32`
	Shuffle int32 elements across lanes.	`VPERMD`

`_mm512_permutex_epi64`, `_mm512_mask_permutex_epi64`, `_mm512_maskz_permutex_epi64`
 	
`_mm512_permutexvar_epi64`, `_mm512_mask_permutexvar_epi64`, `_mm512_maskz_permutexvar_epi64`
	Shuffle int64 elements across lanes.	`VPERMQ`

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result
idx	int32 vector containing indices in memory

_mm512_permutevar_epi32

```
extern __m512i __cdecl _mm512_permutevar_epi32(__m512i a, __m512i idx);
```

Shuffle int32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result.

> **NOTE**
> This intrinsic shuffles across 128-bit lanes, unlike past intrinsics that use the `permutevar` name. This intrinsic is identical to `_mm512_mask_permutexvar_epi32`, and it is recommended that you use that intrinsic name.

_mm512_mask_permutevar_epi32

```
extern __m512i __cdecl _mm512_mask_permutevar_epi32(__m512i src, __mmask16 k, __m512i a, __m512i
idx);
```

Shuffle int32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
This intrinsic shuffles across 128-bit lanes, unlike past intrinsics that use the `permutevar` name. This intrinsic is identical to `_mm512_mask_permutexvar_epi32`, and it is recommended that you use that intrinsic name.

_mm512_permutexvar_epi32

```
extern __m512i __cdecl _mm512_permutexvar_epi32(__m512i idx, __m512i a);
```

Shuffles int32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result.

_mm512_mask_permutexvar_epi32

```
extern __m512i __cdecl _mm512_mask_permutexvar_epi32(__m512i src, __mmask16 k, __m512i idx,
__m512i a);
```

Shuffles int32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutexvar_epi32

```
extern __m512i __cdecl _mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
```

Shuffles int32 elements in *a* across lanes using the corresponding index in *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutex2var_epi32

```
extern __m512i __cdecl _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
```

Shuffles int32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result.

_mm512_mask_permutex2var_epi32

```
extern __m512i __cdecl _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx,
__m512i b);
```

Shuffles int32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask2_permutex2var_epi32

```
extern __m512i __cdecl _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k,
__m512i b);
```

Shuffles int32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm512_maskz_permutex2var_epi32

```
extern __m512i __cdecl _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx,
__m512i b);
```

Shuffles int32 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutex2var_epi64

```
extern __m512i __cdecl _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
```

Shuffles int64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result.

_mm512_mask_permutex2var_epi64

```
extern __m512i __cdecl _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx,
__m512i b);
```

Shuffles int64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using writemask *k* (elements are copied from *a* when the corresponding mask bit is not set).

_mm512_mask2_permutex2var_epi64

```
extern __m512i __cdecl _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k,
__m512i b);
```

Shuffles int64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using writemask *k* (elements are copied from *idx* when the corresponding mask bit is not set).

_mm512_maskz_permutex2var_epi64

```
extern __m512i __cdecl _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx,
__m512i b);
```

Shuffles int64 elements in *a* and *b* across lanes using the corresponding selector and index in *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutex_epi64

```
extern __m512i __cdecl _mm512_permutex_epi64(__m512i a, const int imm);
```

Shuffles int64 elements in *a* within 256-bit lanes using the control in *imm*, and stores the result.

_mm512_mask_permutex_epi64

```
extern __m512i __cdecl _mm512_mask_permutex_epi64(__m512i src, __mmask8 k, __m512i a, const int
imm);
```

Shuffles int64 elements in *a* within 256-bit lanes using the control in *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutex_epi64

```
extern __m512i __cdecl _mm512_maskz_permutex_epi64(__mmask8 k, __m512i a, const int imm);
```

Shuffles int64 elements in *a* within 256-bit lanes using the control in *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_permutexvar_epi64

```
extern __m512i __cdecl _mm512_permutexvar_epi64(__m512i idx, __m512i a);
```

Shuffles int64 elements in *a* across lanes using the corresponding index *idx*, and stores the result.

_mm512_mask_permutexvar_epi64

```
extern __m512i __cdecl _mm512_mask_permutexvar_epi64(__m512i src, __mmask8 k, __m512i idx,
__m512i a);
```

Shuffles int64 elements in *a* across lanes using the corresponding index *idx*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_permutexvar_epi64

```
extern __m512i __cdecl _mm512_maskz_permutexvar_epi64(__mmask8 k, __m512i idx, __m512i a);
```

Shuffles int64 elements in *a* across lanes using the corresponding index *idx*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Reduction Operations

Intrinsics for FP Reduction Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_reduce_add_pd`, `_mm512_mask_reduce_add_pd`	Reduce float64 elements by addition.	**None.**
`_mm512_reduce_add_ps`, `_mm512_mask_reduce_add_ps`	Reduce float32 elements by addition.	**None.**
`_mm512_reduce_max_pd`, `_mm512_mask_reduce_max_pd`	Reduce float64 elements by maximum.	**None.**
`_mm512_reduce_max_ps`, `_mm512_mask_reduce_max_ps`	Reduce float32 elements by maximum.	**None.**
`_mm512_reduce_min_pd`, `_mm512_mask_reduce_min_pd`	Reduce float64 elements by minimum.	**None.**
`_mm512_reduce_min_ps`, `_mm512_mask_reduce_min_ps`	Reduce float32 elements by minimum.	**None.**
`_mm512_reduce_mul_pd`, `_mm512_mask_reduce_mul_pd`	Reduce float64 elements by multiplication.	**None.**
`_mm512_reduce_mul_ps`, `_mm512_mask_reduce_mul_ps`	Reduce float32 elements by multiplication.	**None.**

variable	definition
k	writemask

variable	definition
a	first source vector element

_mm512_reduce_add_pd

```
extern double __cdecl _mm512_reduce_add_pd(__m512d a);
```

Reduces packed float64 elements in *a* by addition.

Returns the sum of all elements in *a*.

_mm512_mask_reduce_add_pd

```
extern double __cdecl _mm512_mask_reduce_add_pd(__mmask8 k, __m512d a);
```

Reduces packed float64 elements in *a* by addition using writemask *k*.

Returns the sum of all active elements in *a*.

_mm512_reduce_add_ps

```
extern float __cdecl _mm512_reduce_add_ps(__m512 a);
```

Reduces packed float32 elements in *a* by addition.

Returns the sum of all elements in *a*.

_mm512_mask_reduce_add_ps

```
extern float __cdecl _mm512_mask_reduce_add_ps(__mmask16 k, __m512 a);
```

Reduces packed float32 elements in *a* by addition using writemask *k*.

Returns the sum of all active elements in *a*.

_mm512_reduce_max_pd

```
extern double __cdecl _mm512_reduce_max_pd(__m512d a);
```

Reduces packed float64 elements in *a* by maximum.

Returns the maximum of all elements in *a*.

_mm512_mask_reduce_max_pd

```
extern double __cdecl _mm512_mask_reduce_max_pd(__mmask8 k, __m512d a);
```

Reduces packed float64 elements in *a* by maximum, using writemask *k*.

Returns the maximum of all active elements in *a*.

_mm512_reduce_max_ps

```
extern float __cdecl _mm512_reduce_max_ps(__m512 a);
```

Reduces packed float32 elements in *a* by maximum.

Returns the maximum of all elements in *a*.

_mm512_mask_reduce_max_ps

```
extern float __cdecl _mm512_mask_reduce_max_ps(__mmask16 k, __m512 a);
```

Reduces packed float32 elements in *a* by maximum, using writemask *k*.

Returns the maximum of all active elements in *a*.

_mm512_reduce_min_pd

```
extern double __cdecl _mm512_reduce_min_pd(__m512d a);
```

Reduces packed float64 elements in *a* by minimum.

Returns the minimum of all elements in *a*.

_mm512_mask_reduce_min_pd

```
extern double __cdecl _mm512_mask_reduce_min_pd(__mmask8 k, __m512d a);
```

Reduces packed float64 elements in *a* by minimum, using writemask *k*.

Returns the minimum of all active elements in *a*.

_mm512_reduce_min_ps

```
extern float __cdecl _mm512_reduce_min_ps(__m512 a);
```

Reduces packed float32 elements in *a* by minimum.

Returns the minimum of all elements in *a*.

_mm512_mask_reduce_min_ps

```
extern float __cdecl _mm512_mask_reduce_min_ps(__mmask16 k, __m512 a);
```

Reduces packed float32 elements in *a* by minimum, using writemask *k*.

Returns the minimum of all active elements in *a*.

_mm512_reduce_mul_pd

```
extern double __cdecl _mm512_reduce_mul_pd(__m512d a);
```

Reduces packed float64 elements in *a* by multiplication.

Returns the product of all elements in *a*.

_mm512_mask_reduce_mul_pd

```
extern double __cdecl _mm512_mask_reduce_mul_pd(__mmask8 k, __m512d a);
```

Reduces packed float64 elements in *a* by multiplication, using writemask *k*.

Returns the product of all active elements in *a*.

_mm512_reduce_mul_ps

```
extern float __cdecl _mm512_reduce_mul_ps(__m512 a);
```

Reduces packed float32 elements in *a* by multiplication.

Returns the product of all elements in *a*.

_mm512_mask_reduce_mul_ps

```
extern float __cdecl _mm512_mask_reduce_mul_ps(__mmask16 k, __m512 a);
```

Reduces packed float32 elements in *a* by multiplication, using writemask *k*.

Returns the product of all active elements in *a*.

Intrinsics for Integer Reduction Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_reduce_add_epi32`, `_mm512_mask_reduce_add_epi32`	Reduces int32 elements of an addition operation.	**None.**
`_mm512_reduce_add_epi64`, `_mm512_mask_reduce_add_epi64`	Reduces int64 elements of an addition operation.	**None.**
`_mm512_reduce_mul_epi32`, `_mm512_mask_reduce_mul_epi32`	Reduces int32 elements of a multiplication operation.	**None.**
`_mm512_reduce_mul_epi64`, `_mm512_mask_reduce_mul_epi64`	Reduces int64 elements of a multiplication operation.	**None.**
`_mm512_reduce_min_epi32`, `_mm512_mask_reduce_min_epi32`	Reduces signed int32 elements of a minimum value operation.	**None.**
`_mm512_reduce_min_epi64`, `_mm512_mask_reduce_min_epi64`	Reduces signed int64 elements of a minimum value operation.	**None.**
`_mm512_reduce_min_epu32`, `_mm512_mask_reduce_min_epu32`	Reduces unsigned int32 elements of a minimum value operation.	**None.**
`_mm512_reduce_min_epu64`, `_mm512_mask_reduce_min_epu64`	Reduces unsigned int64 elements of a minimum value operation.	**None.**
`_mm512_reduce_max_epi32`, `_mm512_mask_reduce_max_epi32`	Reduces signed int32 elements of a maximum value operation.	**None.**
`_mm512_reduce_max_epi64`, `_mm512_mask_reduce_max_epi64`	Reduces signed int64 elements of a maximum value operation.	**None.**
`_mm512_reduce_max_epu32`, `_mm512_mask_reduce_max_epu32`	Reduces unsigned int32 elements of a maximum value operation.	**None.**
`_mm512_reduce_max_epu64`, `_mm512_mask_reduce_max_epu64`	Reduces unsigned int64 elements of a maximum value operation.	**None.**
`_mm512_reduce_or_epi32`, `_mm512_mask_reduce_or_epi32`	Reduces int32 elements of a bitwise `OR` operation.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_reduce_or_epi64`, `_mm512_mask_reduce_or_epi64`	Reduces int64 elements of a bitwise `OR` operation.	**None.**
`_mm512_reduce_and_epi32`, `_mm512_mask_reduce_and_epi32`	Reduces int32 elements of a bitwise `AND` operation.	**None.**
`_mm512_reduce_and_epi64`, `_mm512_mask_reduce_and_epi64`	Reduces int64 elements of a bitwise `AND` operation.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result

_mm512_reduce_and_epi32

```
extern int __cdecl _mm512_reduce_and_epi32(__m512i a);
```

Reduces the packed int32 elements in *a* by bitwise `AND`.

Returns the bitwise `AND` of all elements in *a*.

_mm512_mask_reduce_and_epi32

```
extern int __cdecl _mm512_mask_reduce_and_epi32(__mmask16 k, __m512i a);
```

Reduces the packed int32 elements in *a* by bitwise `AND` using mask *k*.

Returns the bitwise `AND` of all active elements in *a*.

_mm512_reduce_and_epi64

```
extern __int64 __cdecl _mm512_reduce_and_epi64(__m512i a);
```

Reduces the packed int64 elements in *a* by bitwise `AND`.

Returns the bitwise `AND` of all elements in *a*.

_mm512_mask_reduce_and_epi64

```
extern __int64 __cdecl _mm512_mask_reduce_and_epi64(__mmask8 k, __m512i a);
```

Reduces the packed int64 elements in *a* by bitwise `AND` using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the bitwise `AND` of all active elements in *a*.

_mm512_reduce_add_epi32

```
extern int __cdecl _mm512_reduce_add_epi32(__m512i a);
```

Reduces the packed int32 elements in *a* by addition.

Returns the sum of all elements in *a*.

_mm512_mask_reduce_add_epi32

```
extern int __cdecl _mm512_mask_reduce_add_epi32(__mmask16 k, __m512i a);
```

Reduces the packed int32 elements in *a* by addition using mask *k*.

Returns the sum of all active elements in *a*.

_mm512_reduce_add_epi64

```
extern __int64 __cdecl _mm512_reduce_add_epi64(__m512i a);
```

Reduces the packed int64 elements in *a* by addition.

Returns the sum of all elements in *a*.

_mm512_mask_reduce_add_epi64

```
extern __int64 __cdecl _mm512_mask_reduce_add_epi64(__mmask8 k, __m512i a);
```

Reduce the packed int64 elements in *a* by addition, using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the sum of all active elements in *a*.

_mm512_reduce_max_epi32

```
extern int __cdecl _mm512_reduce_max_epi32(__m512i a);
```

Reduce the packed int32 elements in *a* by maximum.

Returns the maximum of all elements in *a*.

_mm512_mask_reduce_max_epi32

```
extern int __cdecl _mm512_mask_reduce_max_epi32(__mmask16 k, __m512i a);
```

Reduce the packed int32 elements in *a* by maximum using mask *k*.

Returns the maximum of all active elements in *a*.

_mm512_reduce_max_epi64

```
extern __int64 __cdecl _mm512_reduce_max_epi64(__m512i a);
```

Reduce the packed int64 elements in *a* by maximum.

Returns the maximum of all elements in *a*.

_mm512_mask_reduce_max_epi64

```
extern __int64 __cdecl _mm512_mask_reduce_max_epi64(__mmask8 k, __m512i a);
```

Reduce the packed int64 elements in *a* by maximum using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the maximum of all active elements in *a*.

_mm512_reduce_max_epu32

```
extern unsigned int __cdecl _mm512_reduce_max_epu32(__m512i a);
```

Reduce the packed unsigned int32 elements in *a* by maximum.

Returns the maximum of all elements in *a*.

_mm512_mask_reduce_max_epu32

```
extern unsigned int __cdecl _mm512_mask_reduce_max_epu32(__mmask16 k, __m512i a);
```

Reduce the packed unsigned int32 elements in *a* by maximum using mask *k*.

Returns the maximum of all active elements in *a*.

_mm512_reduce_max_epu64

```
extern unsigned __int64 __cdecl _mm512_reduce_max_epu64(__m512i a);
```

Reduce the packed unsigned int64 elements in *a* by maximum.

Returns the maximum of all elements in *a*.

_mm512_mask_reduce_max_epu64

```
extern unsigned __int64 __cdecl _mm512_mask_reduce_max_epu64(__mmask8 k, __m512i a);
```

Reduce the packed unsigned int64 elements in *a* by maximum using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the maximum of all active elements in *a*.

_mm512_reduce_min_epi32

```
extern int __cdecl _mm512_reduce_min_epi32(__m512i a);
```

Reduce the packed int32 elements in *a* by minimum.

Returns the minimum of all elements in *a*.

_mm512_mask_reduce_min_epi32

```
extern int __cdecl _mm512_mask_reduce_min_epi32(__mmask16 k, __m512i a);
```

Reduce the packed int32 elements in *a* by maximum using mask *k*.

Returns the minimum of all active elements in *a*.

_mm512_reduce_min_epi64

```
extern __int64 __cdecl _mm512_reduce_min_epi64(__m512i a);
```

Reduce the packed int64 elements in *a* by minimum.

Returns the minimum of all elements in *a*.

_mm512_mask_reduce_min_epi64

```
extern __int64 __cdecl _mm512_mask_reduce_min_epi64(__mmask8 k, __m512i a);
```

Reduce the packed int64 elements in *a* by maximum, using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the minimum of all active elements in *a*.

_mm512_reduce_min_epu32

```
extern unsigned int __cdecl _mm512_reduce_min_epu32(__m512i a);
```

Reduce the packed unsigned int32 elements in *a* by minimum.

Returns the minimum of all elements in *a*.

_mm512_mask_reduce_min_epu32

```
extern unsigned int __cdecl _mm512_mask_reduce_min_epu32(__mmask16 k, __m512i a);
```

Reduce the packed unsigned int32 elements in *a* by maximum using mask *k*.

Returns the minimum of all active elements in *a*.

_mm512_reduce_min_epu64

```
extern unsigned __int64 __cdecl _mm512_reduce_min_epu64(__m512i a);
```

Reduce the packed unsigned int64 elements in *a* by minimum.

Returns the minimum of all elements in *a*.

_mm512_mask_reduce_min_epu64

```
extern unsigned __int64 __cdecl _mm512_mask_reduce_min_epu64(__mmask8 k, __m512i a);
```

Reduce the packed unsigned int64 elements in *a* by minimum using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the minimum of all active elements in *a*.

_mm512_reduce_mul_epi32

```
extern int __cdecl _mm512_reduce_mul_epi32(__m512i a);
```

Reduce the packed int32 elements in *a* by multiplication.

Returns the product of all elements in *a*.

_mm512_mask_reduce_mul_epi32

```
extern int __cdecl _mm512_mask_reduce_mul_epi32(__mmask16 k, __m512i a);
```

Reduce the packed int32 elements in *a* by multiplication using mask *k*.

Returns the product of all active elements in *a*.

_mm512_reduce_mul_epi64

```
extern __int64 __cdecl _mm512_reduce_mul_epi64(__m512i a);
```

Reduce the packed int64 elements in *a* by multiplication.

Returns the product of all elements in *a*.

_mm512_mask_reduce_mul_epi64

```
extern __int64 __cdecl _mm512_mask_reduce_mul_epi64(__mmask8 k, __m512i a);
```

Reduce the packed int64 elements in *a* by multiplication using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the product of all active elements in *a*.

_mm512_reduce_or_epi32

```
extern int __cdecl _mm512_reduce_or_epi32(__m512i a);
```

Reduce the packed int32 elements in *a* by bitwise `OR`.

Returns the bitwise `OR` of all elements in *a*.

_mm512_mask_reduce_or_epi32

```
extern int __cdecl _mm512_mask_reduce_or_epi32(__mmask16 k, __m512i a);
```

Reduce the packed int32 elements in *a* by bitwise `OR` using mask *k*.

Returns the bitwise `OR` of all active elements in *a*.

_mm512_reduce_or_epi64

```
extern __int64 __cdecl _mm512_reduce_or_epi64(__m512i a);
```

Reduce the packed int64 elements in *a* by bitwise `OR`.

Returns the bitwise `OR` of all elements in *a*.

_mm512_mask_reduce_or_epi64

```
extern __int64 __cdecl _mm512_mask_reduce_or_epi64(__mmask8 k, __m512i a);
```

Reduce the packed int64 elements in *a* by bitwise `OR` using mask *k*.

Only those elements in the source registers with the corresponding bit set in vector mask *k* are used for computing. Elements in *a* with corresponding bit clear in *k* are copied as is to the resulting vector.

Returns the bitwise `OR` of all active elements in *a*.

Intrinsics for Set Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Setting Vectors of Undefined Value

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_undefined`	Returns vector of type `__m512i` with undefined elements.	**None.**
`_mm512_undefined_epi32`	Returns vector of type `__m512i` with undefined elements.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_undefined_pd`	Returns vector of type `__m512d` with undefined elements.	**None.**
`_mm512_undefined_ps`	Returns vector of type `__m512` with undefined elements.	**None.**
`_mm512_set1_pd`	Broadcast float64 element to all destination elements.	**None.**
`_mm512_set1_ps`	Broadcast float32 element to all destination elements.	**None.**
`_mm512_set4_pd`	Broadcast float64 element to destination elements with repeated four element sequence.	**None.**
`_mm512_set4_ps`	Broadcast float32 element to destination elements with repeated four element sequence.	**None.**
`_mm512_set_pd`	Broadcast packed float64 elements with supplied values.	**None.**
`_mm512_set_ps`	Broadcast packed float32 elements with supplied values.	**None.**
`_mm512_setr4_pd`	Broadcast packed float64 elements with the repeated four element sequence in reverse order.	**None.**
`_mm512_setr4_ps`	Broadcast packed float32 elements with the repeated four element sequence in reverse order.	**None.**
`_mm512_setr_pd`	Broadcast packed float64 elements with supplied values in reverse order.	**None.**
`_mm512_setr_ps`	Broadcast packed float32 elements with supplied values in reverse order.	**None.**
`_mm512_setzero_pd`	Returns vector of type `__m512d` with all elements set to zero.	`VXORPD`
`_mm512_setzero_ps`	Returns vector of type `__m512` with all elements set to zero.	`VXORPS`
`_mm512_undefined_pd`	Returns vector of type `__m512d` with undefined elements.	**None.**
`_mm512_undefined_ps`	Returns vector of type `__m512` with undefined elements.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_set1_epi8	Broadcast 8-bit integer *a* to all destination elements.	VPBROADCASTB
_mm512_set1_epi32, _mm512_mask_set1_epi32, _mm512_maskz_set1_epi32	Broadcast a single int32 element to all destination elements.	VPBROADCASTD
_mm512_set1_epi64, _mm512_mask_set1_epi64, _mm512_maskz_set1_epi64	Broadcast a single int64 element to all destination elements.	VPBROADCASTQ
_mm512_set1_epi16	Broadcast a single int16 element to all destination elements	VPBROADCASTW
_mm512_set4_epi32	Broadcast packed int32 elements with repeated four element sequence.	**None.**
_mm512_set4_epi64	Broadcast packed int64 elements in with repeated four element sequence.	**None.**
_mm512_set_epi32	Broadcast packed int32 elements with supplied values.	**None.**
_mm512_set_epi64	Broadcast packed int64 elements with supplied values.	**None.**
_mm512_setr4_epi32	Broadcast packed int32 elements with repeated four element sequence in reverse order.	**None.**
_mm512_setr4_epi64	Broadcast packed int64 elements with repeated four element sequence in reverse order.	**None.**
_mm512_setr_epi32	Broadcast packed int32 elements with supplied values in reverse order.	**None.**
_mm512_setr_epi64	Broadcast packed int64 elements with supplied values in reverse order.	**None.**
_mm512_setzero_epi32 _mm512_setzero_si512	Returns vector of type __m512i with all elements set to zero.	VPXOR

variable	definition
k	writemask used as a selector
a	first source vector element

variable	definition
e	elements for set operation
a, *b*, *c*, *d*	elements for set operation

_mm512_set1_pd

```
extern __m512d __cdecl _mm512_set1_pd(double a);
```

Broadcasts float64 value *a* to all destination elements.

_mm512_set1_ps

```
extern __m512 __cdecl _mm512_set1_ps(float a);
```

Broadcasts float32 value *a* to all destination elements.

_mm512_set4_pd

```
extern __m512d __cdecl _mm512_set4_pd(double d, double c, double b, double a);
```

Sets packed float64 elements in destination with the repeated four element sequence (dcba dcba).

_mm512_set4_ps

```
extern __m512 __cdecl _mm512_set4_ps(float d, float c, float b, float a);
```

Sets packed float32 elements in destination with the repeated four element sequence. (dcba dcba dcba dcba).

_mm512_set_pd

```
extern __m512 __cdecl _mm512_set_pd(float e7, float e6, float e5, float e4, float e3, float e2,
float e1, float e0);
```

Sets packed float64 elements in destination with supplied values.

_mm512_set_ps

```
extern __m512 __cdecl _mm512_set_ps(float e15, float e14, float e13, float e12, float e11, float
e10, float e9, float e8, float e7, float e6, float e5, float e4, float e3, float e2, float e1,
float e0);
```

Sets packed float32 elements in destination with supplied values.

_mm512_setr4_pd

```
extern __m512d __cdecl _mm512_setr4_pd(double a, double b, double c, double d);
```

Broadcast packed float64 elements in destination with the repeated four element sequence in reverse order.

_mm512_setr4_ps

```
extern __m512 __cdecl _mm512_setr4_ps(float a, float b, float c, float d);
```

Sets packed float32 elements in destination with the repeated four element sequence in reverse order.

_mm512_setr_pd

```
extern __m512d __cdecl _mm512_setr_pd(double e0, double e1, double e2, double e3, double e4,
double e5, double e6, double e7);
```

Sets packed float64 elements in destination with supplied values in reverse order.

_mm512_setr_ps

```
extern __m512 __cdecl _mm512_setr_ps(float e0, float e1, float e2, float e3, float e4, float e5,
float e6, float e7, float e8, float e9, float e10, float e11, float e12, float e13, float e14,
float e15);
```

Sets packed float32 elements in destination with supplied values in reverse order.

_mm512_setzero_pd

```
extern __m512 __cdecl _mm512_setzero_pd(void);
```

Returns vector of type `__m512d` with all elements set to zero.

_mm512_setzero_ps

```
extern __m512 __cdecl _mm512_setzero_ps(void);
```

Returns vector of type `__m512` with all elements set to zero.

_mm512_undefined_pd

```
extern __m512 __cdecl _mm512_undefined_pd(void)
```

Returns vector of type `__m512d` with undefined elements.

_mm512_undefined_ps

```
extern __m512 __cdecl _mm512_undefined_ps(void)
```

Returns vector of type `__m512` with undefined elements.

_mm512_set1_epi16

```
extern __m512i __cdecl _mm512_set1_epi16(short a);
```

Broadcast int16 *a* to all destination elements.

_mm512_set1_epi8

```
extern __m512i __cdecl _mm512_set1_epi8(char a);
```

Broadcasts int8 *a* to all destination elements.

_mm512_set1_epi32

```
extern __m512i __cdecl _mm512_set1_epi32(int a);
```

Broadcasts int32 *a* to all destination elements.

_mm512_mask_set1_epi32

```
extern __m512i __cdecl _mm512_mask_set1_epi32(__m512i src, __mmask16 k, int a);
```

Broadcasts int32 *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_set1_epi32

```
extern __m512i __cdecl _mm512_maskz_set1_epi32(__mmask16 k, int a);
```

Broadcasts int32 *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_set1_epi64

```
extern __m512i __cdecl _mm512_set1_epi64(__int64 a);
```

Broadcasts int64 *a* to all destination elements.

_mm512_mask_set1_epi64

```
extern __m512i __cdecl _mm512_mask_set1_epi64(__m512i src, __mmask8 k, __int64 a);
```

Broadcasts int64 *a* to all destination elements using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_set1_epi64

```
extern __m512i __cdecl _mm512_maskz_set1_epi64(__mmask8 k, __int64 a);
```

Broadcasts int64 *a* to all destination elements using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_set4_epi32

```
extern __m512i __cdecl _mm512_set4_epi32(int d, int c, int b, int a);
```

Sets packed int32 in destination with the repeated four element sequence.

_mm512_set4_epi64

```
extern __m512i __cdecl _mm512_set4_epi64(__int64 d, __int64 c, __int64 b, __int64 a);
```

Sets packed int64 in destination with the repeated four element sequence.

_mm512_set_epi32

```
extern __m512i __cdecl _mm512_set_epi32(int e15, int e14, int e13, int e12, int e11, int e10,
int e9, int e8, int e7, int e6, int e5, int e4, int e3, int e2, int e1, int e0);
```

Sets packed int32 in destination with supplied values.

_mm512_set_epi64

```
extern __m512i __cdecl _mm512_set_epi64(__int64 e7, __int64 e6, __int64 e5, __int64 e4, __int64
e3, __int64 e2, __int64 e1, __int64 e0);
```

Sets packed int64 in destination with supplied values.

_mm512_setr4_epi32

```
extern __m512i __cdecl _mm512_setr4_epi32(int a, int b, int c, int d);
```

Sets packed int32 in destination with the repeated four element sequence in reverse order.

_mm512_setr4_epi64

```
extern __m512i __cdecl _mm512_setr4_epi64(__int64 a, __int64 b, __int64 c, __int64 d);
```

Sets packed int64 in destination with the repeated four element sequence in reverse order.

_mm512_setr_epi32

```
extern __m512i __cdecl _mm512_setr4_epi32(int a, int b, int c, int d);
```

Sets packed int32 in destination with supplied values in reverse order.

_mm512_setr_epi64

```
extern __m512i __cdecl _mm512_setr_epi64(__int64 e0, __int64 e1, __int64 e2, __int64 e3, __int64
e4, __int64 e5, __int64 e6, __int64 e7);
```

Sets packed int64 in destination with supplied values in reverse order.

_mm512_setzero_epi32

```
extern __m512 __cdecl _mm512_setzero(void);
```

Returns vector of type `__m512i` with all elements set to zero.

_mm512_setzero_si512

```
extern __m512 __cdecl _mm512_setzero(void);
```

Returns vector of type `__m512i` with all elements set to zero.

_mm512_undefined_epi32

```
extern __m512 __cdecl _mm512_undefined_epi32(void);
```

Returns vector of type `__m512i` with undefined elements.

_mm512_setzero

```
extern __m512 __cdecl _mm512_setzero(void);
```

Returns vector of type `__m512` with all elements set to zero.

_mm512_undefined

```
extern __m512 __cdecl _mm512_undefined(void);
```

Returns vector of type `__m512` with undefined elements.

Intrinsics for Shuffle Operations

Intrinsics for FP Shuffle Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_shuffle_pd`, `_mm512_mask_shuffle_pd`, `_mm512_maskz_shuffle_pd`	Shuffle float64 values.	`VSHUFPD`
`_mm512_shuffle_ps`, `_mm512_mask_shuffle_ps`, `_mm512_maskz_shuffle_ps`	Shuffle float32 values.	`VSHUFPS`
`_mm512_shuffle_f64x2`, `_mm512_mask_shuffle_f64x2`, `_mm512_maskz_shuffle_f64x2`	Shuffle float64 values and store using mask.	`VSHUFF64X2`
`_mm512_shuffle_f32x4`, `_mm512_mask_shuffle_f32x4`, `_mm512_maskz_shuffle_f32x4`	Shuffle float32 values and store using mask.	`VSHUFF32X4`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
imm	vector element selector

_mm512_shuffle_f32x4

```
extern __m512 __cdecl _mm512_shuffle_f32x4(__m512 a, __m512 b, const int imm);
```

Shuffles four float32 elements from *a* and *b*, selected by *imm*, and stores the result.

_mm512_mask_shuffle_f32x4

```
extern __m512 __cdecl _mm512_mask_shuffle_f32x4(__m512 src, __mmask16 k, __m512 a, __m512 b,
const int imm);
```

Shuffles four float32 elements from *a* and *b*, selected by *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shuffle_f32x4

```
extern __m512 __cdecl _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, const int imm);
```

Shuffles four float32 elements from *a* and *b*, selected by *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shuffle_f64x2

```
extern __m512d __cdecl _mm512_shuffle_f64x2(__m512d a, __m512d b, const int imm);
```

Shuffles 128-bits (composed of two float64 elements from *a* and *b*, selected by *imm*, and stores the result.

_mm512_mask_shuffle_f64x2

```
extern __m512d __cdecl _mm512_mask_shuffle_f64x2(__m512d src, __mmask8 k, __m512d a, __m512d b,
const int imm);
```

Shuffles 128-bits (composed of two float64 elements from *a* and *b*, selected by *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shuffle_f64x2

```
extern __m512d __cdecl _mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, const int
imm);
```

Shuffles 128-bits (composed of two float64 elements from *a* and *b*, selected by *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shuffle_pd

```
extern __m512d __cdecl _mm512_shuffle_pd(__m512d a, __m512d b, const int imm);
```

Shuffles float64 elements from vectors *a* and *b* within 128-bit lanes using the control in *imm*, and stores the result.

_mm512_mask_shuffle_pd

```
extern __m512d __cdecl _mm512_mask_shuffle_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
const int imm);
```

Shuffle float64 elements from vectors *a* and *b* within 128-bit lanes using the control in *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shuffle_pd

```
extern __m512d __cdecl _mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, const int imm);
```

Shuffle float64 elements from vectors *a* and *b* within 128-bit lanes using the control in *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shuffle_ps

```
extern __m512 __cdecl _mm512_shuffle_ps(__m512 a, __m512 b, const int imm);
```

Shuffles float32 elements from vectors *a* and *b* within 128-bit lanes using the control in *imm*, and stores the result.

_mm512_mask_shuffle_ps

```
extern __m512 __cdecl _mm512_mask_shuffle_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, const
int imm);
```

Shuffle float32 elements from vectors *a* and *b* within 128-bit lanes using the control in *imm*, and stores the result using writemask *k*.

Elements are copied from *src* when the corresponding mask bit is not set.

_mm512_maskz_shuffle_ps

```
extern __m512 __cdecl _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, const int imm);
```

Shuffle float32 elements from vectors *a* and *b* within 128-bit lanes using the control in *imm*, and stores the result using zeromask *k*.

Elements are zeroed out when the corresponding mask bit is not set.

Intrinsics for Integer Shuffle Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_shuffle_epi32, _mm512_mask_shuffle_epi32, _mm512_maskz_shuffle_epi32	Shuffle int32 vectors within 128-bit lanes using control value.	VPSHUFD
_mm512_shuffle_i32x4, _mm512_mask_shuffle_i32x4, _mm512_maskz_shuffle_i32x4	Shuffle four int32 values by specified value.	VSHUFI32X4
_mm512_shuffle_i64x2, _mm512_mask_shuffle_i64x2, _mm512_maskz_shuffle_i64x2	Shuffle two int64 values by specified value.	VSHUFI64X2

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
imm	control value for shuffle operation

_mm512_shuffle_epi32

```
extern __m512i _cdecl __m512_shuffle_epi32(__m512i a, _MM_PERM_ENUM imm);
```

Shuffles int32 in *a* within 128-bit lanes using the control in *imm*, and stores the result.

_mm512_mask_shuffle_epi32

```
extern __m512i _cdecl __m512_mask_shuffle_epi32(__m512i src, __mmask16 k, __m512i a,
_MM_PERM_ENUM imm);
```

Shuffles int32 in *a* within 128-bit lanes using the control in *imm*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shuffle_epi32

```
extern__m512i _cdecl __m512_maskz_shuffle_epi32(__mmask16 k, __m512i a, _MM_PERM_ENUM imm);
```

Shuffles int32 in *a* within 128-bit lanes using the control in *imm*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shuffle_i32x4

```
extern __m512i _cdecl __m512_shuffle_i32x4(__m512i a, __m512i b, _MM_PERM_ENUM imm);
```

Shuffles 128-bits (composed of four int32) selected by *imm* from *a* and *b*, and stores the result.

_mm512_mask_shuffle_i32x4

```
extern __m512i _cdecl __m512_mask_shuffle_i32x4(__m512i src, __mmask16 k, __m512i a, __m512i b,
_MM_PERM_ENUM imm);
```

Shuffles 128-bits (composed of four int32) selected by *imm* from *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shuffle_i32x4

```
extern __m512i _cdecl __m512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b,
_MM_PERM_ENUM imm);
```

Shuffles 128-bits (composed of four int32) selected by *imm* from *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_shuffle_i64x2

```
extern __m512i _cdecl __m512_shuffle_i64x2(__m512i a, __m512i b, _MM_PERM_ENUM imm);
```

Shuffles 128-bits (composed of two int64) selected by *imm* from *a* and *b*, and stores the result.

_mm512_mask_shuffle_i64x2

```
extern __m512i _cdecl __m512_mask_shuffle_i64x2(__m512i src, __mmask8 k, __m512i b, __m512i b,
_MM_PERM_ENUM imm);
```

Shuffles 128-bits (composed of two int64) selected by *imm* from *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_shuffle_i64x2

```
extern __m512i _cdecl __m512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, _MM_PERM_ENUM
imm);
```

Shuffles 128-bits (composed of two int64) selected by *imm* from *a* and *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Intrinsics for Test Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_conflict_epi32`, `_mm512_mask_conflict_epi32`, `_mm512_maskz_conflict_epi32`	Test int32 elements for equality.	`VPCONFLICTD`
`_mm512_conflict_epi64`, `_mm512_mask_conflict_epi64`, `_mm512_maskz_conflict_epi64`	Test int64 elements for equality.	`VPCONFLICTQ`
`_mm512_test_epi32_mask`, `_mm512_mask_test_epi32_mask`	Performs a bitwise logical `AND` operation and stores the logical comparison result.	`VPTESTMD`
`_mm512_testn_epi32_mask`, `_mm512_mask_testn_epi32_mask`	Performs a bitwise logical `AND NOT` operation and stores the logical comparison result.	`VPTESTNMD`
`_mm512_test_epi64_mask`, `_mm512_mask_test_epi64_mask`	Performs a bitwise logical `AND` operation and stores the logical comparison result.	`VPTESTMQ`
`_mm512_testn_epi64_mask`, `_mm512_mask_testn_epi64_mask`	Performs a bitwise logical `AND NOT` operation and stores the logical comparison result.	`VPTESTNMQ`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_conflict_epi32

```
extern __m512i __cdecl _mm512_conflict_epi32(__m512i a);
```

Tests each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant bit.

Each element's comparison forms a zero extended bit vector in the destination.

_mm512_mask_conflict_epi32

```
extern __m512i __cdecl _mm512_mask_conflict_epi32(__m512i src, __mmask16 k, __m512i a);
```

Tests each 32-bit element of *a* for equality with all other elements in *a* closer to the least significant bit using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

Each element's comparison forms a zero extended bit vector in the destination.

_mm512_maskz_conflict_epi32

```
extern __m512i __cdecl _mm512_maskz_conflict_epi32(__mmask16 k, __m512i a);
```

Tests each 32-bit element of *a* for equality with other elements in *a* closer to the least significant bit using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Each element's comparison forms a zero extended bit vector in the destination.

_mm512_conflict_epi64

```
extern __m512i __cdecl _mm512_conflict_epi64(__m512i a);
```

Tests each 64-bit element of *a* for equality with other elements in *a* closer to the least significant bit.

Each element's comparison forms a zero extended bit vector in the destination.

_mm512_mask_conflict_epi64

```
extern __m512i __cdecl _mm512_mask_conflict_epi64(__m512i src, __mmask8 k, __m512i a);
```

Tests each 64-bit element of *a* for equality with other elements in *a* closer to the least significant bit using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

Each element's comparison forms a zero extended bit vector in the destination.

_mm512_maskz_conflict_epi64

```
extern __m512i __cdecl _mm512_maskz_conflict_epi64(__mmask8 k, __m512i a);
```

Tests each 64-bit element of *a* for equality with other elements in *a* closer to the least significant bit using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

Each element's comparison forms a zero extended bit vector the destination.

_mm512_test_epi32_mask

```
extern __mmask16 __cdecl _mm512_test_epi32_mask(__m512i a, __m512i b);
```

Computes the bitwise `AND` of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and sets the corresponding bit in result mask *k* if the intermediate value is non-zero.

_mm512_mask_test_epi32_mask

```
extern __mmask16 __cdecl _mm512_mask_test_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise `AND` of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and sets the corresponding bit in result mask *k* (subject to writemask *k*) if the intermediate value is non-zero.

_mm512_test_epi64_mask

```
extern __mmask8 __cdecl _mm512_test_epi64_mask(__m512i a, __m512i b);
```

Computes the bitwise `AND` of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and sets the corresponding bit in result mask *k* if the intermediate value is non-zero.

_mm512_mask_test_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_test_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise AND of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and sets the corresponding bit in result mask *k* (subject to writemask *k*) if the intermediate value is non-zero.

_mm512_testn_epi32_mask

```
extern __mmask16 __cdecl _mm512_testn_epi32_mask(__m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and sets the corresponding bit in result mask *k* if the intermediate value is zero.

_mm512_mask_testn_epi32_mask

```
extern __mmask16 __cdecl _mm512_mask_testn_epi32_mask(__mmask16 k, __m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 32-bit integers in *a* and *b*, producing intermediate 32-bit values, and sets the corresponding bit in result mask *k* (subject to writemask *k*) if the intermediate value is zero.

_mm512_testn_epi64_mask

```
extern __mmask8 __cdecl _mm512_testn_epi64_mask(__m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and sets the corresponding bit in result mask *k* if the intermediate value is zero.

_mm512_mask_testn_epi64_mask

```
extern __mmask8 __cdecl _mm512_mask_testn_epi64_mask(__mmask8 k, __m512i a, __m512i b);
```

Computes the bitwise AND NOT of packed 64-bit integers in *a* and *b*, producing intermediate 64-bit values, and sets the corresponding bit in result mask *k* (subject to writemask *k*) if the intermediate value is zero.

Intrinsics for Typecast Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the zmmintrin.h header file.
To use these intrinsics, include the immintrin.h file as follows:

```
#include <immintrin.h>
```

NOTE
These intrinsics are used for compilation and do not generate any instructions.

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_castpd512_pd128	Casts from larger type to smaller type.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_castps512_ps128	Casts from larger type to smaller type.	**None.**
_mm512_castsi512_si128	Casts from larger type to smaller type.	**None.**
_mm512_castpd512_pd256	Casts from larger type to smaller type.	**None.**
_mm512_castps512_ps256	Casts from larger type to smaller type.	**None.**
_mm512_castsi512_si256	Casts from larger type to smaller type.	**None.**
_mm512_castpd256_pd512	Casts from smaller type to larger type.	**None.**
_mm512_castps128_ps512	Casts from smaller type to larger type.	**None.**
_mm512_castsi128_si512	Casts from smaller type to larger type.	**None.**
_mm512_castpd256_pd512	Casts from smaller type to larger type.	**None.**
_mm512_castps256_ps512	Casts from smaller type to larger type.	**None.**
_mm512_castsi256_si512	Casts from smaller type to larger type.	**None.**
_mm512_castpd_ps	Casts from smaller type to larger type.	**None.**
_mm512_castpd_si512	Casts from smaller type to larger type.	**None.**
_mm512_castps_pd	Casts from smaller type to larger type.	**None.**
_mm512_castps_si512	Casts from smaller type to larger type.	**None.**
_mm512_castsi512_pd	Casts from smaller type to larger type.	**None.**
_mm512_castsi512_ps	Casts from smaller type to larger type.	**None.**

variable	definition
a	vector element for casting operation

_mm512_castpd_ps

```
extern __m512 __cdecl _mm512_castpd_ps(__m512d a);
```

Casts vector *a* of type `__m512d` to type `__m512`, with no change in value.

_mm512_castpd_si512

```
extern __m512i __cdecl _mm512_castpd_si512(__m512d a);
```

Casts vector *a* of type `__m512d` to type `__m512i`, with no change in value.

_mm512_castps_pd

```
extern __m512d __cdecl _mm512_castps_pd(__m512 a);
```

Casts vector *a* of type `__m512` to type `__m512d`, with no change in value.

_mm512_castps_si512

```
extern __m512i __cdecl _mm512_castps_si512(__m512 a);
```

Casts vector *a* of type `__m512` to type `__m512i`, with no change in value.

_mm512_castpd128_pd512

```
extern __m512d __cdecl _mm512_castpd128_pd512(__m128d a);
```

Casts vector *a* of type `__m128d` to type `__m512d`.

> **NOTE**
> The upper 384-bits of the result are undefined.

_mm512_castpd256_pd512

```
extern __m512d __cdecl _mm512_castpd256_pd512(__m256d a);
```

Casts vector *a* of type `__m256d` to type `__m512d`.

> **NOTE**
> The upper 256-bits of the result are undefined.

_mm512_castpd512_pd128

```
extern __m128d __cdecl _mm512_castpd512_pd128(__m512d a);
```

Casts vector *a* of type `__m512d` to type `__m128d`.

_mm512_castps512_ps128

```
extern __m128 __cdecl _mm512_castps512_ps128(__m512 a);
```

Casts vector *a* of type `__m512` to type `__m128`.

_mm512_castpd512_pd256

```
extern __m256d __cdecl _mm512_castpd512_pd256(__m512d a);
```

Casts vector *a* of type `__m512d` to type `__m256d`.

_mm512_castps128_ps512

```
extern __m512 __cdecl _mm512_castps128_ps512(__m128 a);
```

Casts vector *a* of type `__m128` to type `__m512`.

NOTE
The upper 384-bits of the result are undefined.

_mm512_castps256_ps512

```
extern __m512 __cdecl _mm512_castps256_ps512(__m256 a);
```

Casts vector *a* of type `__m256` to type `__m512`.

NOTE
The upper 256-bits of the result are undefined.

_mm512_castps512_ps256

```
extern __m256 __cdecl _mm512_castps512_ps256(__m512 a);
```

Casts vector *a* of type `__m512` to type `__m256`.

_mm512_castsi128_si512

```
extern __m512i __cdecl _mm512_castsi128_si512(__m128i a);
```

Casts vector *a* of type `__m128i` to type `__m512i`.

NOTE
The upper 384-bits of the result are undefined.

_mm512_castsi256_si512

```
extern __m512i __cdecl _mm512_castsi256_si512(__m256i a);
```

Casts vector *a* of type `__m256i` to type `__m512i`.

NOTE
The upper 256-bits of the result are undefined.

_mm512_castsi512_pd

```
extern __m512d __cdecl _mm512_castsi512_pd(__m512i a);
```

Casts vector *a* of type `__m512i` to type `__m512d`, with no change in value.

_mm512_castsi512_ps

```
extern __m512 __cdecl _mm512_castsi512_ps(__m512i a);
```

Casts vector *a* of type `__m512i` to type `__m512`, with no change in value.

_mm512_castsi512_si128

```
extern __m128i __cdecl _mm512_castsi512_si128(__m512i a);
```

Casts vector *a* of type `__m512d` to type `__m256d`.

_mm512_castsi512_si256

```
extern __m256i __cdecl _mm512_castsi512_si256(__m512i a);
```

Casts vector *a* of type `__m512i` to type `__m256i`.

Intrinsics for Vector Mask Operations

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_kand`	Bitwise logical `AND` masks.	`KANDW`
`_mm512_kandn`	Bitwise logical `AND NOT` masks.	`KANDNW`
`_mm512_kmov`	Move to and from mask registers.	`KMOVW`
`_mm512_kunpackb`	Unpack mask registers.	`KUNPCKBW`
`_mm512_knot`	Bitwise logical `NOT` masks.	`KNOTW`
`_mm512_kor`	Bitwise logical `OR` masks.	`KORW`
`_mm512_kortestc` `_mm512_kortestz`	Bitwise logical `OR` mask and set flag.	`KORTESTW`
`_mm512_kxnor`	Bitwise logical `XNOR` masks.	`KXNORW`
`_mm512_kxor`	Bitwise logical `XOR` masks.	`KXORW`

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element

_mm512_kand

```
extern __mmask16 __cdecl _mm512_kand(__mmask16 a, __mmask16 b);
```

Computes the bitwise `AND` of 16-bit masks *a* and *b*, and stores the result in *k*.

_mm512_kandn

```
extern __mmask16 __cdecl _mm512_kandn(__mmask16 a, __mmask16 b);
```

Computes the bitwise `AND NOT` of 16-bit masks *a* and *b*, and stores the result in *k*.

_mm512_knot

```
extern __mmask16 __cdecl _mm512_knot(__mmask16 a);
```

Computes the bitwise `NOT` of 16-bit mask *a*, and stores the result in *k*.

_mm512_kor

```
extern __mmask16 __cdecl _mm512_kor(__mmask16 a, __mmask16 b);
```

Computes the bitwise `OR` of 16-bit masks *a* and *b*, and stores the result in *k*.

_mm512_kxnor

```
extern __mmask16 __cdecl _mm512_kxnor(__mmask16 a, __mmask16 b);
```

Computes the bitwise `XNOR` of 16-bit masks *a* and *b*, and stores the result in *k*.

_mm512_kxor

```
extern __mmask16 __cdecl _mm512_kxor(__mmask16 a, __mmask16 b);
```

Computes the bitwise `XOR` of 16-bit masks *a* and *b*, and stores the result in *k*.

_mm512_kmov

```
extern __mmask16 __cdecl _mm512_kmov(__mmask16 a);
```

Copies 16-bit mask *a* to *k*.

_mm512_kunpackb

```
extern __mmask16 __cdecl _mm512_kunpackb(__mmask16 a, __mmask16 b);
```

Unpacks and interleaves eight bits from 16-bit masks *a* and *b*, and stores the 16-bit result in mask register.

Intrinsics for Later Generation Intel® Core™ Processor Instruction Extensions

This section discusses intrinsics for 3rd and 4th generation Intel® Core™ Processor instruction extensions, intrinsics for converting half floats, intrinsics that generate random numbers, intrinsics for multi-precision arithmetic, and intrinsics that allow reads and writes to the FS base and GS base registers.

Intrinsics for 3rd Generation Intel® Core™ Processor Instruction Extensions

The 3rd Generation Intel® Core™ Processor Instruction Extension intrinsics are assembly-coded functions that call on 3rd Generation Intel® Core™ Processor Instructions that include new vector SIMD and scalar instructions targeted for Intel® 64 architecture processors in process technology smaller than 32nm.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Functional Overview

The 3rd Generation Intel® Core™ Processor Instruction Extensions include:

- Four intrinsics that map to two hardware instructions `VCVTPS2PH` and `VCVTPH2PS` performing 16-bit floating-point data type conversion to and from single-precision floating-point data type. The intrinsics for conversion to packed 16-bit floating-point values from packed single-precision floating-point values also provide rounding control using an immediate byte.
- Three intrinsics that map to the hardware instruction `RDRAND`. The intrinsics generate random numbers of 16/32/64 bit wide random integers.
- Eight intrinsics that map to the hardware instructions `RDFSBASE`, `RDGSBASE`, `WRFSBASE`, and `WRGSBASE`. The intrinsics allow software that works in the 64-bit environment to read and write the FS base and GS base registers at all privileged levels.

Intrinsics for 4th Generation Intel® Core™ Processor Instruction Extensions

The 4th Generation Intel® Core™ Processor Instruction Extensions intrinsics are assembly-coded functions that call on 4th Generation Intel® Core™ Processor Instructions that include scalar instructions targeted for Intel® 64 architecture processors in process technology smaller than 32nm.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Functional Overview

The 4th Generation Intel® Core™ Processor Instruction Extensions include:

- Four intrinsics that map to two hardware instructions `ADOX` and `ADCX` performing 32-bit or 64-bit arithmetic operations with flags.
- One intrinsic that maps to the hardware instruction, `PREFETCHW`. This intrinsic allows data to be to prefetched into the cache in anticipation of a write. This intrinsic can be found in the Cacheability Support Intrinsics section.
- Three intrinsics that map to the hardware instruction `RDSEED`. The intrinsics generate random numbers of 16/32/64 bit wide random integers.

Intrinsics for Converting Half Floats that Map to 3rd Generation Intel® Core™ Processor Instructions

There are four intrinsics for converting the half-float values.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

These intrinsics convert packed half-precision values starting from the first CPUs with the Intel® AVX instructions support that do not really have any special instructions performing FP16 conversions. Therefore, the intrinsics are lowered to runtime library function calls and map to 3rd Generation Intel® Core™ Processor instructions only when such a processor is specified as target CPU using the `-Qx<CPU>/-x<CPU>` option, where <CPU> is the name of a CPU with support of 3rd Generation Intel® Core™ Processor Instruction Extensions.

See Also

Intrinsics for Converting Half Floats

_mm_cvtph_ps()

Converts four half-precision (16-bit) floating point values to single-precision floating point values. The corresponding 3rd Generation Intel® Core™ Processor extension instruction is `VCVTPH2PS`.

Syntax

```
extern __m128 _mm_cvtph_ps(__m128i x, int imm);
```

Arguments

X	a vector containing four 16-bit FP values
Imm	a conversion control constant

Description

This intrinsic converts four half-precision (16-bit) floating point values to single-precision floating point values.

Returns

A vector containing four single-precision floating point elements.

_mm256_cvtph_ps()

Converts eight half-precision (16-bit) floating point values to single-precision floating point values. The corresponding 3rd Generation Intel® Core™ Processor extension instruction is `VCVTPH2PS`.

Syntax

```
extern __m256 _mm256_cvtph_ps(__m128i x);
```

Arguments

x	a vector containing eight 16-bit FP values

Description

This intrinsic converts eight half-precision (16-bit) floating point values to single-precision floating point values.

Returns

A vector containing eight single-precision floating point elements.

_mm_cvtps_ph()

Converts four single-precision floating point values to half-precision (16-bit) floating point values. The corresponding 3rd Generation Intel® Core™ Processor extension instruction is `VCVTPS2PH`.

Syntax

```
extern __m128i _mm_cvtps_ph(__m128 x, int imm);
```

Arguments

X	a vector containing four single-precision FP values
Imm	a conversion control constant

Description

This intrinsic converts four single-precision floating point values to half-precision (16-bit) floating point values.

Returns

A vector containing eight half-precision (16-bit) floating point elements.

_mm256_cvtps_ph()

Converts eight single-precision floating point values to half-precision (16-bit) floating point values. The corresponding 3rd Generation Intel® Core™ Processor extension instruction is `VCVTPS2PH`.

Syntax

```
extern __m128i _mm_cvtps_ph(__m256 x, int imm);
```

Arguments

X	a vector containing eight single-precision FP values
Imm	a conversion control constant

Description

This intrinsic converts eight single-precision floating point values to half-precision (16-bit) floating point values.

Intrinsics that Generate Random Numbers of 16/32/64 Bit Wide Random Integers

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

RDRAND

There are three intrinsics returning a hardware-generated random value.

These intrinsics are mapped to a single `RDRAND` instruction. The exception is the intrinsic `_rdrand64_step()`, which is mapped to two 32-bit `RDRAND` instructions and several shift and conditional jump/move instructions on 32-bit platforms.

RDSEED

There are three intrinsics returning a hardware-generated random value.

These intrinsics are mapped to a code-sequence based on the `RDSEED` instruction. The result code depends on the context in which the intrinsics were used and on the target OS.

_rdrand16_step(), _rdrand32_step(), _rdrand64_step()

Generate random numbers of 16/32/64 bit wide random integers. These intrinsics are mapped to the hardware instruction `RDRAND`.

Syntax

```
extern int _rdrand16_step(unsigned short *random_val);

extern int _rdrand32_step(unsigned int *random_val);

extern int _rdrand64_step(unsigned __int64 *random_val);
```

Description

These intrinsics generate random numbers of 16/32/64 bit wide random integers. The generated random value is written to the given memory location and the success status is returned: '1' if the hardware returned a valid random value, and '0' otherwise.

Returns

1 : if the hardware returns a 16/32/64 random value.

0 : if the hardware does not return any random value.

_rdseed16_step/ _rdseed32_step/ _rdseed64_step

Generates random numbers of 16/32/64 bit wide random integers. The corresponding 4th Generation Intel® Core™ instruction is `RDSEED`.

Syntax

```
extern int _rdseed16_step(unsigned short *random_val);

extern int _rdseed32_step(unsigned int *random_val);

extern int _rdseed64_step(unsigned __int64 *random_val);
```

Parameters

**random_val*	Random value written to the given memory location

Description

These intrinsics generate random numbers of 16/32/64 bit wide random integers. These intrinsics are mapped to a code-sequence based on the `RDSEED` instruction. The result code depends on the context in which the intrinsics were used and on the target operating system.

NOTE
The `_rdrand64_step()` intrinsic can be used only on systems with the 64-bit registers support.

The generated random value is written to the given memory location and the success status is returned: '1' if the hardware returned a valid random value, and '0' otherwise.

NOTE The difference between `RDSEED` and `RDRAND` intrinsics is that `RDSEED` intrinsics meet the NIST SP 800-90B and NIST SP 800-90C standards, while the `RDRAND` meets the NIST SP 800-90A standard.

Returns

The generated random value is written to the given memory location and the success status is returned. Returns '1' if the hardware returns a random 16/32/64 bit value (success). Returns '0' otherwise (failure).

Intrinsics for Multi-Precision Arithmetic

_addcarry_u32(), _addcarry_u64()

Computes sum of two 32/64 bit wide unsigned integer values and a carry-in and returns the value of carry-out produced by the sum. The corresponding 4th Generation Intel® Core™ Processor extension instructions are `ADCX` and `ADOX`.

Syntax

```
extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned int
src2, unsigned int *sum_out);
```

```
extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned
__int64 src2, unsigned __int64 *sum_out);
```

Parameters

c_in	Value used for determining carry-in value
src1	32/64 bit source integer
src2	32/64 bit source integer
**sum_out*	Pointer to memory location where result is stored

Description

The intrinsic computes sum of two 32/64 bit wide integer values, *src1* and *src2*, and a carry-in value. The carry-in value is considered '1' for any non-zero *c_in* input value or '0' otherwise. The sum is stored to a memory location referenced by *sum_out* argument:

```
*sum_out = src1 + src2 + (c_in !=0 ? 1 : 0)
```

NOTE
This intrinsic does not perform validity checking of the memory address pointed to by *sum_out*, thus it cannot be used to find out if the sum produces carry-out without storing result of the sum.

Returns

Returns the value of the intrinsic is a carry-out value generated by sum. The sum result is stored into memory location pointed by *sum_out* argument.

_addcarryx_u32(), _addcarryx_u64()

Computes sum of two 32/64 bit wide unsigned integer values and a carry-in and returns the value of carry-out produced by the sum. The corresponding 4th Generation Intel® Core™ Processor extension instructions are `ADCX` and `ADOX`.

Syntax

```
extern unsigned char _addcarryx_u32(unsigned char c_in, unsigned int src1, unsigned int
src2, unsigned int *sum_out);
```

```
extern unsigned char _addcarryx_u64(unsigned char c_in, unsigned __int64 src1, unsigned
__int64 src2, unsigned __int64 *sum_out);
```

Parameters

c_in	Value used for determining carry-in value
src1	32/64 bit source integer
src2	32/64 bit source integer
**sum_out*	Pointer to memory location where result is stored

Description

Computes the sum of two 32/64 bit wide integer values (*src1*, *src2*) and a carry-in value. The carry-in value is considered '1' for any non-zero *c_in* input value, or '0' otherwise. The sum is stored to a memory location referenced by *sum_out* argument:

```
*sum_out = src1 + src2 + (c_in !=0 ? 1 : 0)
```

NOTE
This intrinsic does not perform validity checking of the memory address pointed to by *sum_out*, thus it cannot be used to find out if the sum produces carry-out without storing result of the sum.

The intrinsic is translated to either `ADCX`/`ADOX` instruction, chosen by the compiler. By design, these instructions allow running of two interleaved add-with-carry instruction sequences in parallel via using `ADCX` and `ADOX` instructions for these sequences respectively.

Returns

Returns carry-out value generated by the sum. The sum result is stored into memory location pointed by *sum_out* argument.

_subborrow_u32(), _subborrow_u64()

Computes sum of 32/64 bit unsigned integer value with borrow-in value and then subtracts the result from a 32/64 bit unsigned integer value. The corresponding 4th Generation Intel® Core™ Processor extension instruction is `V`.

Syntax

```
extern unsigned char _subborrow_u32(unsigned char b_in, unsigned int src1, unsigned int
src2, unsigned int *diff_out);
```

```
extern unsigned char _subborrow_u64(unsigned char b_in, unsigned __int64 src1, unsigned
__int64 src2, unsigned __int64 *diff_out);
```

Parameters

b_in	Borrow-in value for addition operation
src1	32/64 bit source integer for addition operation
src2	32/64 bit source integer for subtraction operation
**diff_out*	Pointer to memory location where result is stored

Description

Computes the sum of a 32/64 bit wide unsigned integer value *src2* and a borrow-in value *b_in* and then subtracts result of the sum from 32/64 bit wide unsigned integer value *src1*. The borrow-in value is considered '1' for any non-zero *b_in* input value, or '0' otherwise. The difference is then stored to a memory location referenced by *diff_out* argument:

```
*diff_out = src1 - (src2 + (b_in !=0 ? 1 : 0))
```

NOTE
This intrinsic does not perform validity checking of the memory address pointed to by *diff_out*, thus it cannot be used to find out if a subtraction produces borrow-out without storing the result of the subtraction.

Returns

Returns borrow-out value generated by subtraction operation. The result of the subtraction is stored into memory location pointed by *diff_out* argument.

Intrinsics that Allow Reading from and Writing to the FS Base and GS Base Registers

There are eight intrinsics that allow reading and writing the value of the FS base and GS base registers at all privilege levels and in 64-bit mode only.

These intrinsics are mapped to corresponding 3rd Generation Intel® Core™ Processor extension instructions.

_readfsbase_u32(), _readfsbase_u64()

Read the value of the FS base register. Both intrinsics are mapped to `RDFSBASE` *instruction.*

Syntax

```
extern unsigned int _readfsbase_u32();
extern unsigned __int64 _readfsbase_u64();
```

Description

These intrinsics read the value of the FS base register.

Returns

The value of the FS base segment register.

_readgsbase_u32(), _readgsbase_u64()

Read the value of the GS base register. Both intrinsics are mapped to `RDGSBASE` instruction.

Syntax

```
extern unsigned int _readgsbase_u32();
extern unsigned int _readgsbase_u32();
```

Description

These intrinsics read the value of the GS base register.

Returns

The value of the GS base segment register.

_writefsbase_u32(), _writefsbase_u64()

Write the value to the FS base register. Both intrinsics are mapped to `WRFSBASE` instruction.

Syntax

```
extern void _writefsbase_u32(int 32 val);
extern void _writefsbase_u64(unsigned __int64 val);
```

Arguments

val	the value to be written to the FS base register

Description

These intrinsics write the given value to the FS segment base register.

_writegsbase_u32(), _writegsbase_u64()

Write the value to the GS base register. Both intrinsics are mapped to `WRGSBASE` instruction.

Syntax

```
_writegsbase_u32(int 32 val)
extern void _writegsbase_u64(unsigned __int64 val);
```

Arguments

val	the value to be written to the GS base register

Description

These intrinsics write the given value to the GS segment base register.

Intrinsics for Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Intel® Advanced Vector Extensions 2 (Intel® AVX2) extends Intel® Advanced Vector Extensions (Intel® AVX) by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. The Intel® AVX2 instructions follow the same programming model as the Intel® AVX instructions.

Intel® AVX2 also provides enhanced functionality for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

Intel® AVX2 intrinsics have vector variants that use `__m128`, `__m128i`, `__m256`, and `__m256i` data types.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The Intel® AVX2 intrinsics are supported on IA-32 and Intel® 64 architectures built from 32nm process technology. They map directly to the Intel® AVX2 new instructions and other enhanced 128-bit SIMD instructions.

Functional Overview

Intel® AVX2 instructions promote the vast majority of 128-bit integer SIMD instruction sets to operate with 256-bit wide YMM registers. Intel® AVX2 instructions are encoded using the VEX prefix and require the same operating system support as Intel® AVX. Generally, most of the promoted 256-bit vector integer instructions follow the 128-bit lane operation, similar to the promoted 256-bit floating-point SIMD instructions in Intel® AVX.

The Intel® AVX2 instructions may be broadly categorized as follows:

- **Intel® AVX complementary integer instructions:** Intel® AVX2 instructions complement the Intel® AVX instructions that are typed for integer operations with a full complement of equivalent instructions for operating with integer data elements.
- **`BROADCAST` and `PERMUTE` instructions:** These instructions provide cross-lane functionality for floating-point and integer operations. In addition, some of the Intel® AVX2 256-bit vector integer instructions promoted from legacy SSE instruction sets also exhibiting cross-lane behavior fall into this category; for example, instructions of the `VPMOVZ`/`VPMOVS` family.
- **`SHIFT` instructions:** Intel® AVX2 vector SHIFT instructions operate with per-element shift count and support data element sizes of 32- and 64-bits.
- **`GATHER` instructions:** The Intel® AVX2 vector GATHER instructions are used for fetching non-contiguous data elements from memory using vector-index memory addressing. They introduce a new memory addressing form consisting of a base register and multiple indices specified by a vector register (`XMM` or `YMM`). Data element sizes of 32- and 64-bits are supported as well as data types for floating-point and integer elements.

See Also

Instruction Set Architecture (ISA) site at https://software.intel.com/content/www/us/en/develop/tools/isa-extensions.html

Details about the Intel® AVX Intrinsics

Intrinsics for Arithmetic Operations

_mm256_abs_epi8/16/32

Computes the absolute value of the signed packed integer data elements of a given vector. The corresponding Intel® AVX2 instruction is `VPABSB`*,* `VPABSW`*, or* `VPABSD`*.*

Syntax

```
extern __m256i _mm256_abs_epi8(__m256i s1);
```

```
extern __m256i _mm256_abs_epi16(__m256i s1);
```

```
extern __m256i _mm256_abs_epi32(__m256i s1);
```

Arguments

s1	integer source vector used for the operation

Description

Computes the absolute value of each data element, either signed bytes, 16-bit words, or 32-bit integers, of the source vector and stores the UNSIGNED results in the destination vector.

Returns

Result of the operation.

_mm256_add_epi8/16/32/64

Adds signed/unsigned packed 8/16/32/64-bit integer data elements of two vectors. The corresponding Intel® AVX2 instruction is `VPADDB`*,* `VPADDW`*,* `VPADDD`*, or* `VPADDQ`*.*

Syntax

```
extern __m256i _mm256_add_epi8(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_add_epi16(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_add_epi32(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_add_epi64(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Adds packed signed/unsigned 8-, 16-, 32-, or 64-bit integers from source vector *s1* and corresponding bits of source vector *s2* and stores the packed integer result in the destination vector. When an individual result is too large to be represented in 8/16/32/64 bits (overflow), the result is wrapped around and the low 8/16/32/64 bits are written to the destination vector (that is, the carry is ignored).

You must control the range of values operated upon to prevent undetected overflow conditions.

Returns

Result of the addition operation.

_mm256_adds_epi8/16

Adds the signed 8/16-bit integer data elements with saturation of two vectors. The corresponding Intel® AVX2 instruction is `VPADDSB` *or* `VPADDSW`*.*

Syntax

```
extern __m256i _mm256_adds_epi8(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_adds_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD add of the packed, signed, 8- or 16-bit integer data elements with saturation from the first source vector, *s1*, and corresponding elements of the second source vector, *s2*, and stores the packed integer results in the destination vector. When an individual byte/word result is beyond the range of a signed byte/word integer (that is, greater than 7FH/7FFFH or less than 80H/8000H), the saturated value of 7FH/7FFFH or 80H/8000H, respectively, is written to the destination vector.

Returns

Result of the addition operation.

_mm256_adds_epu8/16

Adds the unsigned 8/16-bit integer data elements with saturation of two vectors. The corresponding Intel® AVX2 instruction is `VPADDUSB` *or* `VPADDUSW`*.*

Syntax

```
extern __m256i _mm256_adds_epu8(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_adds_epu16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD add of the packed, unsigned, 8- or 16-bit integer data elements with saturation from the first source vector, *s1*, and corresponding elements of the second source vector, *s2*, and stores the packed integer results in the destination vector. When an individual byte/word result is beyond the range of a unsigned byte/word integer (that is, greater than FFH/FFFFH), the saturated value of FFH/FFFFH, respectively, is written to the destination vector.

Returns

Result of the addition operation.

_mm256_sub_epi8/16/32/64

Subtracts signed/unsigned packed 8/16/32/64-bit integer data elements of two vectors. The corresponding Intel® AVX2 instruction is `VPSUBB`*,* `VPSUBW`*,* `VPADDD`*, or* `VPSUBQ`*.*

Syntax

```
extern __m256i _mm256_sub_epi8(__m256i s1, __m256i s2);
extern __m256i _mm256_sub_epi16(__m256i s1, __m256i s2);
extern __m256i _mm256_sub_epi32(__m256i s1, __m256i s2);
extern __m256i _mm256_sub_epi64(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Subtracts packed signed/unsigned 8-, 16-, 32-, or 64-bit integers of the second source vector *s2* from corresponding bits of the first source vector *s1* and stores the packed integer result in the destination vector. When an individual result is too large to be represented in 8/16/32/64 bits (overflow), the result is wrapped around and the low 8/16/32/64 bits are written to the destination vector (that is, the carry is ignored).

You must control the range of values operated upon to prevent undetected overflow conditions.

Returns

Result of the subtraction operation.

_mm256_subs_epi8/16

Subtracts the signed 8/16-bit integer data elements with saturation of two vectors. The corresponding Intel® AVX2 instruction is `VPSUBSB` *or* `VPSUBSW`*.*

Syntax

```
extern __m256i _mm256_subs_epi8(__m256i s1, __m256i s2);
extern __m256i _mm256_subs_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD subtract of the packed, signed, 8- or 16-bit integer data elements with saturation of the second source vector *s2* from the corresponding elements of the first source vector *s1* and stores the packed integer results in the destination vector. When an individual byte/word result is beyond the range of a signed byte/word integer (that is, greater than 7FH/7FFFH or less than 80H/8000H), the saturated value of 7FH/7FFFH or 80H/8000H, respectively, is written to the destination vector.

Returns

Result of the subtraction operation.

_mm256_subs_epu8/16

Subtracts the unsigned 8/16-bit integer data elements with saturation of two vectors. The corresponding Intel® AVX2 instruction is `VPSUBUSB` *or* `VPSUBUSW`*.*

Syntax

```
extern __m256i _mm256_subs_epu8(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_subs_epu16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD subtract of the packed, unsigned, 8- or 16-bit integer data elements with saturation of the second source vector *s2* from the corresponding elements of the first source vector *s1* and stores the packed integer results in the destination vector. When an individual byte/word result is less than zero (that is, 00H/0000H), the saturated value of 00H/0000H is written to the destination vector.

Returns

Result of the subtraction operation.

_mm256_avg_epu8/16

Computes the average of unsigned 8/16-bit integer data elements of two vectors. The corresponding Intel® AVX2 instruction is `VPAVGB` *or* `VPAVGW`*.*

Syntax

```
extern __m256i _mm256_avg_epu8(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_avg_epu16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD average of the packed unsigned integers from source vector *s2* and source vector *s1* and stores the results in the destination vector. For each corresponding pair of data elements in the first and second vectors, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right by one bit position.

Returns

Result of the operation.

_mm256_hadd_epi16/32

Horizontally adds adjacent signed packed 16/32-bit integer data elements of two vectors. The corresponding Intel® AVX2 instruction is `VPHADDW` *or* `VPHADDD`*.*

Syntax

```
extern __m256i _mm256_hadd_epi16(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_hadd_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Adds two adjacent 16- or 32-bit signed integers horizontally from source vectors, *s1* and *s2* and packs the 16 or 32-bit signed results to the destination vector.

Horizontal addition of two adjacent data elements of the low 16- or 32-bytes of the first and second source vectors are packed into the low 16- or 32-bytes of the destination vector. Horizontal addition of two adjacent data elements of the high 16- or 32-bytes of the first and second source vectors are packed into the high 16- or 32-bytes of the destination vector.

Returns

Result of the horizontal addition operation.

_mm256_hadds_epi16

Horizontally adds adjacent signed packed 16-bit integer data elements of two vectors with saturation. The corresponding Intel® AVX2 instruction is `VPHADDSW`*.*

Syntax

```
extern __m256i _mm256_hadds_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Adds two adjacent, signed 16-bit integers horizontally from the source vectors *s1* and *s2*, saturates the signed results, and packs the signed, saturated 16-bit results to the destination vector.

Returns

Result of the horizontal addition operation with saturation.

_mm256_hsub_epi16/32

Horizontally subtracts adjacent signed packed 16/32-bit integer data elements of two vectors. The corresponding Intel® AVX2 instruction is `VPHSUBW` *or* `VPHSUBD`*.*

Syntax

```
extern __m256i _mm256_hsub_epi16(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_hsub_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs horizontal subtraction on each adjacent pair of 16- or 32-bit signed integers by subtracting the most significant word from the least significant word of each pair source vectors *s2* and *s2*, and packs the signed 16- or 32-bit results to the destination vector.

Returns

Result of the horizontal subtraction operation.

_mm256_hsubs_epi16

Horizontally subtracts adjacent signed packed 16-bit integer data elements of two vectors with saturation. The corresponding Intel® AVX2 instruction is `VPHSUBSW`*.*

Syntax

```
extern __m256i _mm256_hsubs_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in source vectors *s1* and *s2*. The signed, saturated 16-bit results are packed to the destination vector.

Returns

Result of the horizontal subtraction operation with saturation.

_mm256_madd_epi16

Multiplies signed packed 16-bit integer data elements of two vectors. The corresponding Intel® AVX2 instruction is `VPMADDW`*.*

Syntax

```
extern __m256i _mm256_madd_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Multiplies individual, signed 16-bit integers of source vector *s1* by the corresponding signed 16-bit integers of source vector *s2*, producing temporary, signed, 32-bit [doubleword] results. The adjacent doubleword results are then summed and stored in the destination vector.

For example, the corresponding low-order words `(15:0)` and `(31-16)` in *s2* and *s1* vectors are multiplied, and the doubleword results are added together and stored in the low doubleword of the destination vector `(31-0)`. The same operation is performed on the other pairs of adjacent words.

Returns

Result of the multiplication operation.

_mm256_maddubs_epi16

Multiplies unsigned packed 16-bit integer data elements of one vector with signed elements of second vector. The corresponding Intel® AVX2 instruction is `VPMADDUBSW`*.*

Syntax

```
extern __m256i _mm256_maddubs_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Multiplies vertically each unsigned byte of source vector *s1* with the corresponding signed byte of source vector *s2*, producing intermediate, signed 16-bit integers. Each adjacent pair of signed words is added, and the saturated result is packed to the destination vector.

For example, the lowest-order bytes (bits `7:0`) in *s1* and *s2* vectors are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits `15:8`) of the vectors. The sign-saturated result is stored in the lowest word of the destination vector (`15:0`). The same operation is performed on the other pairs of adjacent bytes.

Returns

Result of the multiplication operation.

_mm256_mul_epi32

Multiplies two vectors with packed doubleword signed integer values. The corresponding Intel® AVX2 instruction is `VPMULDQ`*.*

Syntax

```
extern __m256i _mm256_mul_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Multiplies the value of packed signed doubleword integer in source vector *s1* by the value in source vector *s2* and stores the result in the destination vector.

When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Returns

Result of the multiplication operation.

_mm256_mul_epu32

Multiplies two vectors with packed doubleword unsigned integer values. The corresponding Intel® AVX2 instruction is `VPMULUDQ`*.*

Syntax

```
extern __m256i _mm256_mul_epu32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Multiplies the value of packed unsigned doubleword integer in source vector *s1* by the value in source vector *s2* and stores the result in the destination vector.

When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Returns

Result of the multiplication operation.

_mm256_mulhi_epi16

Multiplies signed packed 16/32-bit integer data elements of two vectors and stores high bits. The corresponding Intel® AVX2 instruction is `VPMULHW`*.*

Syntax

```
extern __m256i _mm256_mulhi_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD signed multiply of the packed signed 16-bit integers in source vectors *s1* and *s2* and stores the high 16 bits of each intermediate 32-bit result in the destination vector.

Returns

Result of the multiplication operation.

_mm256_mulhi_epu16

Multiplies unsigned packed 16/32-bit integer data elements of two vectors and stores high bits. The corresponding Intel® AVX2 instruction is `VPMULHUW`*.*

Syntax

```
extern __m256i _mm256_mulhi_epu16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD unsigned multiply of the packed unsigned 16-bit integers in source vectors *s1* and *s2* and stores the high 16 bits of each intermediate 32-bit result in the destination vector.

Returns

Result of the multiplication operation.

_mm256_mullo_epi16/32

Multiplies signed packed 16/32-bit integer data elements of two vectors and stores low bits. The corresponding Intel® AVX2 instruction is `VPMULLW` *or* `VPMULLD`*.*

Syntax

```
extern __m256i _mm256_mullo_epi16(__m256i s1, __m256i s2);

extern __m256i _mm256_mullo_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD signed multiply of the packed signed 16- or 32-bit integers in source vectors *s1* and *s2* and stores the low 16- or 32-bits of each intermediate 32- or 64-bit result in the destination vector.

Returns

Result of the multiplication operation.

_mm256_mulhrs_epi16

Multiplies extended packed unsigned integers of two vectors with round and scale. The corresponding Intel® AVX2 instruction is `VPMULHRSW`.

Syntax

```
extern __m256i _mm256_mulhrs_epi16(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Vertically multiplies each signed 16-bit integer from *s1*vector with the corresponding signed 16-bit integer of *s2* vector, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most-significant-bits. Rounding is performed by adding 1 to the least-significant-bit of the 18-bit intermediate result.

The final result is obtained by selecting the 16 bits immediately to the right of the most-significant-bit of each 18-bit intermediate result and packing them to the destination operand.

Returns

Result of the multiply, round, and scale operation.

_mm256_sign_epi8/16/32

Changes the sign of elements in one source vector depending on the sign of corresponding element in other source vector. The corresponding Intel® AVX2 instruction is `VPSIGNB, VPSIGNW, or VPSIGND`.

Syntax

```
extern __m256i _mm256_sign_epi8(__m256i s1, __m256i s2);

extern __m256i _mm256_sign_epi16(__m256i s1, __m256i s2);

extern __m256i _mm256_sign_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Modifies the sign of data elements in the source vector *s1* depending on the sign of corresponding element in vector *s2* as follows:

- If sign of data element in *s2* vector is < 0 then the sign of corresponding data element in vector *s1* is made negative.
- If sign of data element in *s2* vector is > 0 then the sign of corresponding data element in vector *s1* is left unchanged.
- If the data element in *s2* vector is = 0 then the corresponding data element in vector *s1* is set to 0.

The `_mm256_sign_epi8` intrinsic operates on 8-bit signed bytes. The `_mm256_sign_epi16` intrinsic operates on 16-bit signed words. The `_mm256_sign_epi32` intrinsic operates on 32-bit signed integers.

Returns

Result of the operation.

_mm256_mpsadbw_epu8

Performs multiple sum of absolute differences on extended packed unsigned integer values ofs two vectors. The corresponding Intel® AVX2 instruction is `VMPSADBW`*.*

Syntax

```
extern __m256i _mm256_mpsadbw_epu8(__m256i s1, __m256i s2, const int mask);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation
mask	integer constant specifying offset

Description

Performs multiple sum operations of the absolute difference of blocks of four packed unsigned bytes of vector *s2* with sequential blocks of four packed unsigned bytes in vector *s1*. The offset granularity in both vectors is 32 bits.

The sum-absolute-difference (SAD) operation is repeated 16 times by the intrinsic between the *s2* vector with a fixed offset and a variable *s1* vector where the offset is shifted by eight bits for each SAD operation. The integer constant specified in *mask* provides bit fields that specify the initial offset for *s2* and *s1* vectors. Each 16-bit result of eight SAD operations is written to the respective word in the result vector.

Returns

Result of the multiple sum-absolute-difference operation.

_mm256_sad_epu8

Computes sum of absolute differences between extended packed unsigned values of two vectors. The corresponding Intel© AVX2 instruction is `VPSADBW`.

Syntax

```
extern __m256i _mm256_sad_epu8(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Computes the absolute value of the difference of packed groups of eight unsigned byte integers from the source vectors *s1* and *s2*. Four blocks of eight differences are stored at specific locations in the destination vector. Remaining bits in the destination vector are set to zero.

Returns

Result of the single sum-absolute-difference operation.

Intrinsics for Arithmetic Shift Operations

_mm256_sra_epi16/32

Arithmetic shift of word/doubleword elements to right according to specified number. The corresponding Intel© AVX2 instruction is `VPSRAW`*, or* `VPSRAD`.

Syntax

```
extern __m256i _mm256_sra_epi16(__m256i s1, __m128i count);
extern __m256i _mm256_sra_epi32(__m256i s1, __m128i count);
```

Arguments

s1	integer source vector used for the operation
count	128-bit memory location used for the operation

Description

Performs an arithmetic shift of bits in the individual data elements (16-bit word or 32-bit doubleword) in the first source vector *s1* to the right by the number of bits specified in *count*. The empty high-order bits are filled with the initial value of the sign bit. If the value specified by *count* is greater than 15/31/63 (depending on the intrinsic being used), the destination vector is filled with the initial value of the sign bit.

The *count* argument is a 128-bit memory location. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

Returns

Result of the right-shift operation.

_mm256_srai_epi16/32

Arithmetic shift of word/doubleword elements to right according to specified number. The corresponding Intel® AVX2 instruction is `VPSRAW` or `VPSRAD`.

Syntax

```
extern __m256i _mm256_srai_epi16(__m256i s1, const int count);
extern __m256i _mm256_srai_epi32(__m256i s1, const int count);
```

Arguments

s1	integer source vector used for the operation
count	8-bit immediate used for the operation

Description

Performs an arithmetic shift of 16-bit [word] or 32-bit [doubleword] elements within a 128-bit lane of source vector *s1* to the right by the number of bits specified in *count*. The empty high-order bits are filled with the initial value of the sign bit. If the value specified by *count* is greater than 15 or 31, the whole destination vector is filled with the initial value of the sign bit. The *count* argument is an 8-bit immediate.

Returns

Result of the right-shift operation.

_mm256_srav_epi32

Arithmetic shift of doubleword elements to right according variable values. The corresponding Intel® AVX2 instruction is `VPSRAVD`.

Syntax

```
extern __m256i _mm256_srav_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector providing variable values for shift operation

Description

Performs an arithmetic shift of 32 bits (doublewords) in the individual data elements in source vector *s1* to the right by the count value of corresponding data elements in source vector *s2*. As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit.

The count values are specified individually in each data element of the second source vector. If the unsigned integer value specified in the respective data element of the second source vector is greater than 31 (for a doubleword), then the destination data elements are filled with the initial value of the sign bit.

Returns

Result of the right-shift operation.

_mm_srav_epi32

Arithmetic shift of doubleword elements to right according variable values. The corresponding Intel® AVX2 instruction is `VPSRAVD`.

Syntax

```
extern __m128i _mm_srav_epi32(__m128i s1, __m128i s2);
```

Arguments

s1	128-bit integer source vector used for the operation
s2	128-bit integer source vector providing variable values for shift operation

Description

Performs an arithmetic shift of the 32 bits (doublewords) in the individual data elements in source vector *s1* to the right by the count value of corresponding data elements in source vector *s2*. As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit.

The count values are specified individually in each data element of the second source vector. If the unsigned integer value specified in the respective data element of the second source vector is greater than 31 (for a doubleword), then the destination data elements are filled with the initial value of the sign bit.

Returns

Result of the right-shift operation.

Intrinsics for Blend Operations

_mm_blend_epi32, _mm256_blend_epi16/32

Conditionally blends data elements of source vector depending on bits in a mask. The corresponding Intel® AVX2 instruction is `VPBLENDD` or `VPBLENDW`.

Syntax

```
extern __m128i _mm_blend_epi32(__m128i s1, __m128i s2, const int mask);

extern __m256i _mm256_blend_epi16(__m256i s1, __m256i s2, const int mask);

extern __m256i _mm256_blend_epi32(__m256i s1, __m256i s2, const int mask);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation
mask	8-bit immediate used for the operation

Description

Performs a blend operation by conditionally copying 16/32-bit [word/doubleword] elements from source vectors *s2* and *s1*, depending on mask bits defined in *mask*. The mask bits are the least significant 8 bits in *mask* when the 256-bit intrinsics, `_mm256_blend_epi16`/`_mm256_blend_epi32`, are used, and 4 bits when the 128-bit intrinsic, `_mm_blend_epi32`, is used.

Each word/doubleword element of the destination vector is copied from the corresponding word/doubleword element in *s2* if a mask bit is 1, or is copied from the corresponding word/doubleword element in *s1* if a mask bit is 0.

Returns

Result of the blend operation.

_mm256_blendv_epi8

Conditionally blends word elements of source vector depending on bits in a mask vector. The corresponding Intel® AVX2 instruction is `VPBLENDVB`*.*

Syntax

```
extern __m256i _mm256_blendv_epi8(__m256i s1, __m256i s2, __m256i mask);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation
mask	integer vector used for the operation

Description

Performs a blend operation by conditionally copying 8-bit byte elements from source vectors *s2* and *s1*, depending on mask bits defined in *mask* vector. The mask bits are the most significant bit in each byte element of *mask.*

Each byte element of the destination vector is copied from the corresponding byte element in *s2* if a mask bit is 1, or the corresponding byte element in *s1* if a mask bit is 0.

Returns

Result of the blend operation.

Intrinsics for Bitwise Operations

_mm256_and_si256

Performs bitwise logical `AND` *operation on signed integer vectors. The corresponding Intel® AVX2 instruction is* `VPAND`*.*

Syntax

```
extern __m256i _mm256_and_si256(__m256i s1, __m256i s2);
```

Arguments

s1	signed integer vector used for the operation
s2	signed integer vector also used for the operation

Description

Performs a bitwise logical AND operation of the signed integer elements of source vector *s1* and corresponding elements in source vector *s2*, and stores the result in the destination vector. If the corresponding bits of the first and second vectors are 1, each bit of the result is set to 1, otherwise it is set to 0.

Returns

Result of the bitwise logical `AND` operation.

_mm256_andnot_si256

Performs bitwise logical `AND NOT` operation on signed integer vectors. The corresponding Intel® AVX2 instruction is `VPANDN`.

Syntax

```
extern __m256i _mm256_andnot_si256(__m256i s1, __m256i s2);
```

Arguments

s1	signed integer vector used for the operation
s2	signed integer vector also used for the operation

Description

Performs a bitwise logical NOT operation on source vector *s1* then performs bitwise AND with source vector *s2* and stores the result in the destination vector. If the corresponding bit in the first vector is 0 and the corresponding bit in the second vector is 1, each bit of the result is set to 1, otherwise it is set to 0.

Returns

Result of the bitwise logical `AND NOT` operation.

_mm256_or_si256

Performs bitwise logical `OR` operation on signed integer vectors. The corresponding Intel® AVX2 instruction is `VPOR`.

Syntax

```
extern __m256i _mm256_or_si256(__m256i s1, __m256i s2);
```

Arguments

s1	signed integer vector used for the operation
s2	signed integer vector also used for the operation

Description

Performs a bitwise logical OR operation of the signed integer elements of source vector *s2* and the corresponding elements in source vector *s1* and stores the result in the destination vector. If either of the corresponding bits of the first and second vectors are 1, each bit of the result is set to 1, otherwise it is set to 0.

Returns

Result of the bitwise logical `OR` operation.

_mm256_xor_si256

Performs bitwise logical `XOR` *operation on signed integer vectors. The corresponding Intel® AVX2 instruction is* `VPXOR`*.*

Syntax

```
extern __m256i _mm256_xor_si256(__m256i s1, __m256i s2);
```

Arguments

s1	signed integer vector used for the operation
s2	signed integer vector also used for the operation

Description

Performs a bitwise logical `XOR` operation of the signed integer elements of source vector *s2* and the corresponding elements in source vector *s1* and stores the result in the destination vector. If the corresponding bits of the first and second vectors differ, each bit of the result is set to 1, otherwise it is set to 0.

Returns

Result of the bitwise logical `XOR` operation.

Intrinsics for Broadcast Operations

_mm_broadcastss_ps, _mm256_broadcastss_ps

Take the low packed single-precision floating-point data element from the source operand and broadcast to all elements of the result vector. The corresponding Intel® AVX2 instruction is `VBROADCASTSS`*.*

Syntax

```
extern __m128 _mm_broadcastss_ps(__m128 val);
extern __m256 _mm256_broadcastss_ps(__m128 val);
```

Arguments

val	__m128 vector containing the 32-bit element to be broadcasted

Description

Takes the low packed single-precision floating-point (float32) data element from the source operand and broadcasts it to all elements of the result vector. The source operand is `__m128`; only the low 32 bits of this operand are broadcasted.

Returns

Return result of the broadcast operation.

_mm256_broadcastsd_pd

Takes the low packed double-precision floating-point data element from the source operand and broadcast to all elements of the result vector. The corresponding Intel® AVX2 instruction is `VBROADCASTSD`.

Syntax

```
extern __m256d _mm256_broadcastsd_pd(__m128d val);
```

Arguments

val	__m128 vector containing the 64-bit element to be broadcasted

Description

Takes the low packed double-precision floating-point (float64) data element from the source operand and broadcasts it to all elements of the result vector. The source operand is `__m128d`; only the low 64 bits of this operand are broadcasted.

The 128-bit version of this intrinsics is `_mm_broadcastsd_pd`. The intrinsic's syntax is `extern __m128d _mm_broadcastsd_pd(__m128d val);` This intrinsic is an alias for `_mm_movedup_pd()` intrinsic. Please, see Double-precision Floating-point Vector Intrinsics for details.

Returns

Return result of the broadcast operation.

_mm_broadcastb_epi8, _mm256_broadcastb_epi8

Take byte elements from the source operand and broadcast to all elements of the result vector. The corresponding Intel® AVX2 instruction is `VPBROADCASTB`.

Syntax

```
extern __m128i _mm_broadcastb_epi8(__m128i val);

extern __m256i _mm256_broadcastb_epi8(__m128i val);
```

Arguments

val	__m128i vector containing the elements to be broadcasted

Description

Takes a byte integer in the low bits of the source operand and broadcasts to all elements of the result vector.

Returns

Returns result of the broadcast operation.

_mm_broadcastw_epi16, _mm256_broadcastw_epi16

Take word elements from the source operand and broadcast to all elements of the result vector. The corresponding Intel© AVX2 instruction is `VPBROADCASTW`.

Syntax

```
extern __m128i _mm_broadcastw_epi16(__m128i val);
extern __m256i _mm256_broadcastw_epi16(__m128i val);
```

Arguments

val	__m128i vector containing the elements to be broadcasted

Description

Takes a word integer in the low bits of the source operand and broadcasts to all eight or sixteen elements of the result vector.

Returns

Returns result of the broadcast operation.

_mm_broadcastd_epi32, _mm256_broadcastd_epi32

Take doublewords from the source operand and broadcast to all elements of the result vector. The corresponding Intel© AVX2 instruction is `VPBROADCASTD`.

Syntax

```
extern __m128i _mm_broadcastd_epi32(__m128i val);
extern __m256i _mm256_broadcastd_epi32(__m128i val);
```

Arguments

val	__m128i vector containing 32-bit element to be broadcasted

Description

Takes a dword integer in the low bits of the source operand and broadcasts to all elements of the result vector.

Returns

Returns result of the broadcast operation.

_mm_broadcastq_epi64, _mm256_broadcastq_epi64

Take qwords from the source operand and broadcast to all elements of the result vector. The corresponding Intel® AVX2 instruction is `VPBROADCASTQ`.

Syntax

```
extern __m128i _mm_broadcastq_epi64(__m128i val);
extern __m256i _mm256_broadcastq_epi64(__m128i val);
```

Arguments

val	__m128i vector containing 64-bit element to be broadcasted

Description

Takes a qword integer in the low bits of the source operand and broadcasts to all elements of the result vector.

Returns

Returns result of the broadcast operation.

_mm256_broadcastsi128_si256

Takes 128-bit data from the source operand and broadcasts it to all 128-bit elements of the result 256-bit vector. The corresponding Intel® AVX2 instructions are `VBROADCASTI128` *and* `VPERM2I128`.

Syntax

```
extern __m256i _mm256_broadcastsi128_si256(__m128i val);
```

Arguments

val	the value to be broadcasted

Description

Takes 128-bit data from the source operand and broadcasts it to all 128-bit elements of the result 256-bit vector.

Returns

Returns result of the broadcast operation.

Intrinsics for Compare Operations

_mm256_cmpeq_epi8/16/32/64

Compares packed bytes/words/doublewords/quadwords of two source vectors. The corresponding Intel® AVX2 instruction is `VPCMPEQB`, `VPCMPEQW`, `VPCMPEQD`, *or* `VPCMPEQQ`.

Syntax

```
extern __m256i _mm256_cmpeq_epi8(__m256i s1, __m256i s2);
extern __m256i _mm256_cmpeq_epi16(__m256i s1, __m256i s2);
extern __m256i _mm256_cmpeq_epi32(__m256i s1, __m256i s2);
extern __m256i _mm256_cmpeq_epi64(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD compare for equality of packed bytes, words, doublewords, or quadwords in source vectors *s1* and *s2*. If a pair of data elements is equal, the corresponding data element in the destination vector is set to all 1s. If a pair of data elements is unequal, the corresponding data element in the destination vector is set to 0.

Returns

Destination vector with result of the compare equal operation.

_mm256_cmpgt_epi8/16/32/64

Compares packed bytes/words/doublewords/ quadwords of two source vectors. The corresponding Intel® AVX2 instruction is `VPCMPGTB`*,* `VPCMPGTW`*,* `VPCMPGTD`*, or* `VPCMPGTQ`*.*

Syntax

```
extern __m256i _mm256_cmpgt_epi8(__m256i s1, __m256i s2);
extern __m256i _mm256_cmpgt_epi16(__m256i s1, __m256i s2);
extern __m256i _mm256_cmpgt_epi32(__m256i s1, __m256i s2);
extern __m256i _mm256_cmpgt_epi64(__m256i s1, __m256i s2);
```

Arguments

s1	integer destination vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD signed compare to determine which of the data elements [packed bytes, words, doublewords, or quadwords] in destination vector *s1* is greater than the corresponding element in the source vector *s2*.

For each pair of data elements in *s1* and *s2*, if the *s1* data element is greater than the corresponding element in *s2*, then the corresponding element in the destination vector is set to all 1s. If the *s1* data element is less than the corresponding data element in *s2*, then the corresponding data element in destination vector is set to all 0s.

If the data elements are equal, the destination vector is set to 0.

Returns

Destination vector with result of the compare greater-than operation.

_mm256_max_epi8/16/32

Determines the maximum value between two vectors with packed signed byte/word/doubleword integers. The corresponding Intel® AVX2 instruction is `VPMAXSB`*,* `VPMAXSW`*, or* `VPMAXSD`*.*

Syntax

```
extern __m256i _mm256_max_epi8(__m256i s1, __m256i s2);
extern __m256i _mm256_max_epi16(__m256i s1, __m256i s2);
extern __m256i _mm256_max_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD compare of the packed signed byte, word, or doubleword integers in source vectors *s1* and *s2* and returns the maximum value for each pair of integers to the destination vector.

Returns

Destination vector with result of the compare operation.

_mm256_max_epu8/16/32

Determines the maximum value between two vectors with packed unsigned byte/word/doubleword integers. The corresponding Intel® AVX2 instruction is `VPMAXUB`*,* `VPMAXUW`*, or* `VPMAXUD`*.*

Syntax

```
extern __m256i _mm256_max_epu8(__m256i s1, __m256i s2);
extern __m256i _mm256_max_epu16(__m256i s1, __m256i s2);
extern __m256i _mm256_max_epu32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD compare of the packed unsigned byte, word, or doubleword integers in source vectors *s1* and *s2* and returns the maximum value for each pair of integers to the destination vector.

Returns

Destination vector with result of the compare operation.

_mm256_min_epi8/16/32

Determines the minimum value between two vectors with packed signed byte/word/doubleword integers. The corresponding Intel® AVX2 instruction is `VPMINSB`*,* `VPMINSW`*, or* `VPMINSD`*.*

Syntax

```
extern __m256i _mm256_min_epi8(__m256i s1, __m256i s2);
extern __m256i _mm256_min_epi16(__m256i s1, __m256i s2);
extern __m256i _mm256_min_epi32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD compare of the packed signed byte, word, or doubleword integers in source vectors *s1* and *s2* and returns the minimum value for each pair of integers to the destination vector.

Returns

Destination vector with result of the compare operation.

_mm256_min_epu8/16/32

Determines the minimum value between two vectors with packed unsigned byte/word/doubleword integers. The corresponding Intel® AVX2 instruction is `VPMINUB`*,* `VPMINUW`*, or* `VPMINUD`*.*

Syntax

```
extern __m256i _mm256_min_epu8(__m256i s1, __m256i s2);
extern __m256i _mm256_min_epu16(__m256i s1, __m256i s2);
extern __m256i _mm256_min_epu32(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation

Description

Performs a SIMD compare of the packed unsigned byte, word, or doubleword integers in source vectors *s1* and *s2* and returns the minimum value for each pair of integers to the destination vector.

Returns

Destination vector with result of the compare operation.

Intrinsics for Fused Multiply Add Operations

_mm_fmadd_pd, _mm256_fmadd_pd

Multiply-adds packed double-precision floating-point values using three float64 vectors. The corresponding FMA instruction is `VFMADD<XXX>PD`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128d _mm_fmadd_pd(__m128d a, __m128d b, __m128d c);
```

For 256-bit vector

```
extern __m256d _mm256_fmadd_pd(__m256d a, __m256d b, __m256d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the infinite precision intermediate results are added to corresponding values in the third operand, after which the final results are rounded to the nearest float64 values.

The compiler defaults to using the `VFMADD213PD` instruction and uses the other forms `VFMADD132PD` or `VFMADD231PD` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-add operation.

_mm_fmadd_ps, _mm256_fmadd_ps

Multiply-adds packed single-precision floating-point values using three float32 vectors. The corresponding FMA instruction is `VFMADD<XXX>PS`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128 _mm_fmadd_ps(__m128 a, __m128 b, __m128 c);
```

For 256-bit vector

```
extern __m256 _mm256_fmadd_ps(__m256 a, __m256 b, __m256 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation
c	float32 vector also used for the operation

Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the infinite precision intermediate results are added to corresponding values in the third operand, after which the final results are rounded to the nearest float32 values.

The compiler defaults to using the `VFMADD213PS` instruction and uses the other forms `VFMADD132PS` or `VFMADD231PS` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-add operation.

_mm_fmadd_sd

Multiply-adds scalar double-precision floating-point values using three float64 vectors. The corresponding FMA instruction is `VFMADD<XXX>SD`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128d _mm_fmadd_sd(__m128d a, __m128d b, __m128d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of scalar SIMD multiply-add computation on scalar double-precision floating-point values in the low 32-bits of three source operands, *a*, *b*, and *c*. The float64 values in two operands, *a* and *b*, are multiplied and the infinite precision intermediate result is obtained. The float64 value in the third operand, *c*, is added to the infinite precision intermediate result. The final result is rounded to the nearest float64 value.

The compiler defaults to using the `VFMADD213SD` instruction and uses the other forms `VFMADD132SD` or `VFMADD231SD` only if a low level optimization decides it is useful/necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-add operation.

_mm_fmadd_ss

Multiply-adds scalar single-precision floating-point values using three float32vectors. The corresponding FMA instruction is `VFMADD<XXX>SS`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128 _mm_fmadd_ss(__m128 a, __m128 b, __m128 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation
c	float32 vector also used for the operation

Description

Performs a set of scalar SIMD multiply-add computation on scalar single-precision floating-point values in the low 32-bits of three source operands, *a*, *b*, and *c*. The float32 values in two operands, a and b, are multiplied and the infinite precision intermediate result is obtained. The float32 value in the third operand, *c*, is added to the infinite precision intermediate result. The final result is rounded to the nearest float32 value.

The compiler defaults to using the `VFMADD213SS` instruction and uses the other forms `VFMADD132SS` or `VFMADD231SS` only if a low level optimization decides it is useful/necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-add operation.

_mm_fmaddsub_pd, _mm256_fmaddsub_pd

Multiply-adds and subtracts packed double-precision floating-point values using three float64 vectors. The corresponding FMA instruction is `VFMADDSUB<XXX>PD`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128d _mm_fmaddsub_pd(__m128d a, __m128d b, __m128d c);
```

For 256-bit vector

```
extern __m256d _mm256_fmaddsub_pd(__m256d a, __m256d b, __m256d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of SIMD multiply-add-subtract computation on packed double-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and infinite precision intermediate results are obtained. The odd values in the third operand, `c`, are added to the intermediate results while the even values are subtracted from them. The final results are rounded to the nearest float64 values.

The compiler defaults to using the `VFMADDSUB213PD` instruction and uses the other forms `VFMADDSUB132PD` or `VFMADDSUB231PD` only if a low-level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-add-subtract operation.

_mm_fmaddsub_ps, _mm256_fmaddsub_ps

Multiply-adds and subtracts packed single-precision floating-point values using three float32 vectors. The corresponding FMA instruction is `VFMADDSUB<XXX>PS`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128 _mm_fmaddsub_ps(__m128 a, __m128 b, __m128 c);
```

For 256-bit vector

```
extern __m256 _mm256_fmaddsub_ps(__m256 a, __m256 b, __m256 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation
c	float32 vector also used for the operation

Description

Performs a set of SIMD multiply-add-subtract computation on packed single-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and infinite precision intermediate results are obtained. The odd values in the third operand, `c`, are added to the intermediate results while the even values are subtracted from them. The final results are rounded to the nearest float32 values.

The compiler defaults to using the `VFMADDSUB213PS` instruction and uses the other forms `VFMADDSUB132PS` or `VFMADDSUB231PS` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-add-subtract operation.

_mm_fmsub_pd, _mm256_fmsub_pd

Multiply-subtracts packed double-precision floating-point values using three float64 vectors. The corresponding FMA instruction is `VFMSUB<XXX>PD`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128d _mm_fmsub_pd(__m128d a, __m128d b, __m128d c);
```

For 256-bit vector

```
extern __m256d _mm256_fmsub_pd(__m256d a, __m256d b, __m256d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the infinite precision intermediate results are obtained. From the infinite precision intermediate results, the values in the third operand, `c`, are subtracted. The final results are rounded to the nearest float64 values.

The compiler defaults to using the `VFMSUB213PD` instruction and uses the other forms `VFMSUB132PD` or `VFMSUB231PD` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-subtract operation.

_mm_fmsub_ps, _mm256_fmsub_ps

Multiply-subtracts packed single-precision floating-point values using three float32 vectors. The corresponding FMA instruction is `VFMSUB<XXX>PS`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128 _mm_fmsub_ps(__m128 a, __m128 b, __m128 c);
```

For 256-bit vector

```
extern __m256 _mm256_fmsub_ps(__m256 a, __m256 b, __m256 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation

c	float32 vector also used for the operation

Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the infinite precision intermediate results are obtained. From the infinite precision intermediate results, the values in the third operand, `c`, are subtracted. The final results are rounded to the nearest float32 values.

The compiler defaults to using the `VFMSUB213PS` instruction and uses the other forms `VFMSUB132PS` or `VFMSUB231PS` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-subtract operation.

_mm_fmsub_sd

Multiply-subtracts scalar double-precision floating-point values using three float64 vectors. The corresponding FMA instruction is `VFMSUB<XXX>SD`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128d _mm_fmsub_sd(__m128d a, __m128d b, __m128d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of scalar SIMD multiply-subtract computation on scalar double-precision floating-point values in the low 64-bits of three source operands, *a*, *b*, and *c*. The float64 values in two operands, a and b, are multiplied and the infinite precision intermediate result is obtained. From the infinite precision intermediate result, the float64 value in the third operand, *c*, is subtracted. The final result is rounded to the nearest float64 value.

The compiler defaults to using the `VFMSUB213SD` instruction and uses the other forms `VFMSUB132SD` or `VFMSUB231SD` only if a low level optimization decides it is useful/necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-subtract operation.

_mm_fmsub_ss

Multiply-subtracts scalar single-precision floating-point values using three float32vectors. The corresponding FMA instruction is `VFMSUB<XXX>SS`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128 _mm_fmsub_ss(__m128 a, __m128 b, __m128 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation
c	float32 vector also used for the operation

Description

Performs a set of scalar SIMD multiply-subtract computation on scalar single-precision floating-point values in the low 32-bits of three source operands, `a`, `b`, and `c`. The float32 values in two operands, `a` and `b`, are multiplied and the infinite precision intermediate result is obtained. From the infinite precision intermediate result, the float32 value in the third operand, `c`, is subtracted. The final result is rounded to the nearest float32 value.

The compiler defaults to using the `VFMSUB213SS` instruction and uses the other forms `VFMSUB132SS` or `VFMSUB231SS` only if a low level optimization decides it is useful/necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-subtract operation.

_mm_fmsubadd_pd, _mm256_fmsubadd_pd

Multiply-subtracts and adds packed double-precision floating-point values using three float64 vectors. The corresponding FMA instruction is `VFMSUBADD<XXX>PD`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128d _mm_fmsubadd_pd(__m128d a, __m128d b, __m128d c);
```

For 256-bit vector

```
extern __m256d _mm256_fmsubadd_pd(__m256d a, __m256d b, __m256d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of SIMD multiply-subtract-add computation on packed double-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and infinite precision intermediate results are obtained. The odd values in the third operand, `c`, are subtracted from the intermediate results while the even values are added to them. The final results are rounded to the nearest float64 values.

The compiler defaults to using the `VFMSUBADD213PD` instruction and uses the other forms `VFMSUBADD132PD` or `VFMSUBSADD231PD` only if a low-level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-subtract-add operation.

_mm_fmsubadd_ps, _mm256_fmsubadd_ps

Multiply-subtracts and adds packed single-precision floating-point values using three float32 vectors. The corresponding FMA instruction is `VFMSUBADD<XXX>PS`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128 _mm_fmsubadd_ps(__m128 a, __m128 b, __m128 c);
```

For 256-bit vector

```
extern __m256 _mm256_fmsubadd_ps(__m256 a, __m256 b, __m256 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation
c	float32 vector also used for the operation

Description

Performs a set of SIMD multiply-subtract-add computation on packed single-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and infinite precision intermediate results are obtained. The odd values in the third operand, `c`, are subtracted from the intermediate results while the even values are added to them. The final results are rounded to the nearest float32 values.

The compiler defaults to using the `VFMSUBADD213PS` instruction and uses the other forms `VFMSUBADD132PS` or `VFMSUBADDS231PS` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the multiply-add-subtract operation.

_mm_fnmadd_pd, _mm256_fnmadd_pd

Multiply-adds negated packed double-precision floating-point values of three float64 vectors. The corresponding FMA instruction is `VFNMADD<XXX>PD`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128d _mm_fnmadd_pd(__m128d a, __m128d b, __m128d c);
```

For 256-bit vector

```
extern __m256d _mm256_fnmadd_pd(__m256d a, __m256d b, __m256d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of SIMD negated multiply-add computation on packed double-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the negated infinite precision intermediate results are added to the values in the third operand, `c`, after which the final results are rounded to the nearest float64 values.

The compiler defaults to using the `VFNMADD213PD` instruction and uses the other forms `VFNMADD132PD` or `VFNMADD231PD` only if a low level optimization decides it as useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-add operation.

_mm_fnmadd_ps, _mm256_fnmadd_ps

Multiply-adds negated packed single-precision floating-point values of three float32 vectors. The corresponding FMA instruction is `VFNMADD<XXX>PS`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128 _mm_fnmadd_ps(__m128 a, __m128 b, __m128 c);
```

For 256-bit vector

```
extern __m256 _mm256_fnmadd_ps(__m256 a, __m256 b, __m256 c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of SIMD negated multiply-add computation on packed single-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the negated infinite precision intermediate results are added to the values in the third operand, `c`, after which the final results are rounded to the nearest float32 values.

The compiler defaults to using the `VFNMADD213PS` instruction and uses the other forms `VFNMADD132PS` or `VFNMADD231PS` only if a low level optimization decides it as useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-add operation.

_mm_fnmadd_sd

Multiply-adds negated scalar double-precision floating-point values of three float64 vectors. The corresponding FMA instruction is `VFNMADD<XXX>SD`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128d _mm_fnmadd_sd(__m128d a, __m128d b, __m128d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of scalar SIMD negated multiply-add computation on scalar double-precision floating-point values in the low 64-bits of three source operands, *a*, *b*, and *c*. The float64 values in two operands, *a* and *b*, are multiplied and the negated infinite precision intermediate result obtained is added to the float64 value in the third operand, *c*. The final result is rounded to the nearest float64 value.

The compiler defaults to using the `VFNMADD213SD` instruction and uses the other forms `VFNMADD132SD` or `VFNMADD231SD` only if a low level optimization decides it as useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-add operation.

_mm_fnmadd_ss

Multiply-adds negated scalar single-precision floating-point values of three float32 vectors. The corresponding FMA instruction is `VFNMADD<XXX>SS`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128 _mm_fnmadd_ss(__m128 a, __m128 b, __m128 c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of scalar SIMD negated multiply-add computation on scalar single-precision floating-point values in the low 32-bits of three source operands, *a*, *b*, and *c*. The float32 values in two operands, *a* and *b*, are multiplied and the negated infinite precision intermediate result obtained is added to the float32 value in the third operand, *c*. The final result is rounded to the nearest float32 value.

The compiler defaults to using the `VFNMADD213SS` instruction and uses the other forms `VFNMADD132SS` or `VFNMADD231SS` only if a low level optimization decides it as useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-add operation.

_mm_fnmsub_pd, _mm256_fnmsub_pd

Multiply-subtracts negated packed double-precision floating-point values of three float64 vectors. The corresponding FMA instruction is `VFNMSUB<XXX>PD`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128d _mm_fnmsub_pd(__m128d a, __m128d b, __m128d c);
```

For 256-bit vector

```
extern __m256d _mm256_fnmsub_pd(__m256d a, __m256d b, __m256d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of SIMD negated multiply-subtract computation on packed double-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the negated infinite precision intermediate result is obtained. From this intermediate result the value in the third operand, `c`, is subtracted, after which the final results are rounded to the nearest float64 values.

The compiler defaults to using the `VFNMSUB213PD` instruction and uses the other forms `VFNMSUB132PD` or `VFNMSUB231PD` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-subtract operation.

_mm_fnmsub_ps, _mm256_fnmsub_ps

Multiply-subtracts negated packed single-precision floating-point values of three float32 vectors. The corresponding FMA instruction is `VFNMSUB<XXX>PS`*, where XXX could be 132, 213, or 231.*

Syntax

For 128-bit vector

```
extern __m128 _mm_fnmsub_ps(__m128 a, __m128 b, __m128 c);
```

For 256-bit vector

```
extern __m256 _mm256_fnmsub_ps(__m256 a, __m256 b, __m256 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation
c	float32 vector also used for the operation

Description

Performs a set of SIMD negated multiply-subtract computation on packed single-precision floating-point values using three source vectors/operands, `a`, `b`, and `c`. Corresponding values in two operands, `a` and `b`, are multiplied and the negated infinite precision intermediate result is obtained. From this intermediate result the value in the third operand, `c`, is subtracted, after which the final results are rounded to the nearest float32 values.

The compiler defaults to using the `VFNMSUB213PS` instruction and uses the other forms `VFNMSUB132PS` or `VFNMSUB231PS` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-subtract operation.

_mm_fnmsub_sd

Multiply-subtracts negated scalar double-precision floating-point values of three float64 vectors. The corresponding FMA instruction is `VFNMSUB<XXX>SD`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128d _mm_fnmsub_sd(__m128d a, __m128d b, __m128d c);
```

Arguments

a	float64 vector used for the operation
b	float64 vector also used for the operation
c	float64 vector also used for the operation

Description

Performs a set of scalar SIMD negated multiply-subtract computation on scalar double-precision floating-point values in the low 64-bits of three source operands, *a*, *b*, and *c*. The float64 values in two operands, *a* and *b*, are multiplied and the negated infinite precision intermediate result is obtained. From this negated intermediate result, the float64 value in the third operand, *c*, is subtracted. The final result is rounded to the nearest float64 value.

The compiler defaults to using the `VFNMSUB213SD` instruction and uses the other forms `VFNMSUB132SD` or `VFNMSUB231SD` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-subtract operation.

_mm_fnmsub_ss

Multiply-subtracts negated scalar single-precision floating-point values of three float32 vectors. The corresponding FMA instruction is `VFNMSUB<XXX>SS`*, where XXX could be 132, 213, or 231.*

Syntax

```
extern __m128 _mm_fnmsub_ss(__m128 a, __m128 b, __m128 c);
```

Arguments

a	float32 vector used for the operation
b	float32 vector also used for the operation
c	float32 vector also used for the operation

Description

Performs a set of scalar SIMD negated multiply-subtract computation on scalar single-precision floating-point values in the low 32-bits of three source operands, *a*, *b*, and *c*. The float32 values in two operands, *a* and *b*, are multiplied and the negated infinite precision intermediate result is obtained. From this negated intermediate result, the float32 value in the third operand, *c*, is subtracted. The final result is rounded to the nearest float32 value.

The compiler defaults to using the `VFNMSUB213SS` instruction and uses the other forms `VFNMSUB132SS` or `VFNMSUB231SS` only if a low level optimization decides it is useful or necessary. For example, the compiler could change the default if it finds that another instruction form saves a register or eliminates a move.

Returns

Result of the negated multiply-subtract operation.

Intrinsics for GATHER Operations

_mm_mask_i32gather_pd, _mm256_mask_i32gather_pd

Gathers 2/4 packed double-precision floating point values from memory referenced by the given base address, dword indices and scale, and using the given double-precision FP mask values. The corresponding Intel© AVX2 instruction is `VGATHERDPD`.

Syntax

```
extern __m128d _mm_mask_i32gather_pd(__m128d def_vals, double const * base, __m128i
vindex __m128d vmask, const int scale);
```

```
extern __m256d _mm256_mask_i32gather_pd(__m256d def_vals, double const * base, __m128i
vindex __m256d vmask, const int scale);
```

Arguments

def_vals	the vector of double-precision FP values copied to the destination when the corresponding element of the double-precision FP mask is '0'.
base	the base address used to reference the loaded FP elements.
vindex	the vector of dword indices used to reference the loaded FP elements.
vmask	the vector of FP elements used as a vector mask; only the most significant bit of each data element is used as a mask.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally load 2/4 packed double-precision floating-point values from memory using dword indices according to mask values and updates the destination operand.

Below is the pseudo-code for the intrinsics:

`_mm_mask_i32gather_pd()`:

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[127:64]);
```

`_mm256_mask_i32gather_pd()`:

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[127:64]);
result[191:128] = (vmask[191]==1) ? (mem[base+vindex[95:64]*scale]) : (def_vals[191:128]);
result[255:192] = (vmask[255]==1) ? (mem[base+vindex[127:96]*scale]) : (def_vals[255:192]);
```

Returns

A 128/256-bit vector with conditionally gathered double-precision FP values.

_mm_i32gather_pd, _mm256_i32gather_pd

Gathers 2/4 packed double-precision floating point values from memory referenced by the given base address, dword indices and scale. The corresponding Intel® AVX2 instruction is `VGATHERDPD`.

Syntax

`extern __m128d _mm_i32gather_pd(double const * base, __m128i vindex, const int scale);`

`extern __m256d _mm256_i32gather_pd(double const * base, __m128i vindex, const int scale);`

Arguments

base	the base address used to reference the loaded FP elements.
vindex	the vector of dword indices used to reference the loaded FP elements.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 packed double-precision floating-point values from memory using dword indices and updates the destination operand.

Below is the pseudo-code for the intrinsics:

`_mm_i32gather_pd()`:

```
result[63:0] = mem[base+vindex[31:0]*scale];
result[127:64] = mem[base+vindex[63:32]*scale];
```

`_mm256_i32gather_pd()`:

```
result[63:0] = mem[base+vindex[31:0]*scale];
result[127:64] = mem[base+vindex[63:32]*scale];
result[191:128] = mem[base+vindex[95:64]*scale];
result[255:192] = mem[base+vindex[127:96]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered double-precision FP values.

_mm_mask_i64gather_pd, _mm256_mask_i64gather_pd

Gathers 2/4 packed double-precision floating point values from memory referenced by the given base address, qword indices and scale, and using the given double precision FP mask values. The corresponding Intel® AVX2 instruction is `VGATHERQPD`.

Syntax

`extern __m128d _mm_mask_i64gather_pd(__m128d def_vals, double const * base, __m128i vindex __m128d vmask, const int scale);`

```
extern __m256d _mm256_mask_i64gather_pd(__m256d def_vals, double const * base, __m128i
vindex __m256d vmask, const int scale);
```

Arguments

def_vals	the vector of double-precision FP values copied to the destination when the corresponding element of the double-precision FP mask is '0'.
base	the base address used to reference the loaded FP elements.
vindex	the vector of qword indices used to reference the loaded FP elements.
vmask	the vector of FP elements used as a vector mask; only the most significant bit of each data element is used as a mask.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally load 2/4 packed double-precision floating-point values from memory using qword indices according to mask values.

Below is the pseudo-code for the intrinsics:

`_mm_mask_i64gather_pd():`

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[127:64]);
```

`_mm256_mask_i64gather_pd():`

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[127:64]);
result[191:128] = (vmask[191]==1) ? (mem[base+vindex[191:128]*scale]) : (def_vals[191:128]);
result[255:192] = (vmask[255]==1) ? (mem[base+vindex[255:192]*scale]) : (def_vals[255:192]);
```

Returns

A 128/256-bit vector with conditionally gathered double-precision values.

_mm_i64gather_pd, _mm256_i64gather_pd

Gathers 2/4 packed double-precision floating point values from memory referenced by the given base address, qword indices, and scale. The corresponding Intel® AVX2 instruction is `VGATHERQPD`.

Syntax

```
extern __m128d _mm_i64gather_pd(double const * base, __m128i vindex, const int scale);
```

```
extern __m256d _mm256_mask_i64gather_pd(double const * base, __m128i vindex, const int
scale);
```

Arguments

base	the base address used to reference the loaded FP elements.
vindex	the vector of qword indices used to reference the loaded FP elements.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 packed double-precision floating-point values from memory using qword indices and updates the destination operand.

Below is the pseudo-code for the intrinsics:

`_mm_i64gather_pd()`:

```
result[63:0] = mem[base+vindex[63:0]*scale];
result[127:64] = mem[base+vindex[127:64]*scale];
```

`_mm256_i64gather_pd()`:

```
result[63:0] = mem[base+vindex[63:0]*scale];
result[127:64] = mem[base+vindex[127:64]*scale];
result[191:128] = mem[base+vindex[191:128]*scale];
result[255:192] = mem[base+vindex[255:192]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered double-precision FP values.

_mm_mask_i32gather_ps, _mm256_mask_i32gather_ps

Gathers 2/4 packed single-precision floating point values from memory referenced by the given base address, dword indices and scale, and using the given single-precision FP mask values. The corresponding Intel® AVX2 instruction is `VGATHERDPS`.

Syntax

```
extern __m128 _mm_mask_i32gather_ps(__m128 def_vals, float const * base, __m128i vindex
__m128 vmask, const int scale);
```

```
extern __m256 _mm256_mask_i32gather_ps(__m256 def_vals, float const * base, __m256i
vindex __m256 vmask, const int scale);
```

Arguments

def_vals	the vector of single-precision FP values copied to the destination when the corresponding element of the single-precision FP mask is '0'.
base	the base address used to reference the loaded FP elements.

vindex	the vector of dword indices used to reference the loaded FP elements.
vmask	the vector of FP elements used as a vector mask; only the most significant bit of each data element is used as a mask.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally load 2/4 packed single-precision floating-point values from memory using dword indices according to mask values.

Below is the pseudo-code for the intrinsics:

`_mm_mask_i32gather_ps():`

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[63:32]);
result[95:64] = (vmask[95]==1) ? (mem[base+vindex[95:64]*scale]) : (def_vals[95:64]);
result127:96] = (vmask[127]==1) ? (mem[base+vindex[127:96]*scale]) : (def_vals[127:96]);
```

`_mm256_mask_i32gather_ps():`

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[63:32]);
result[95:64] = (vmask[95]==1) ? (mem[base+vindex[95:64]*scale]) : (def_vals[95:64]);
result127:96] = (vmask[127]==1) ? (mem[base+vindex[127:96]*scale]) : (def_vals[127:96]);
result[159:128] = (vmask[159]==1) ? (mem[base+vindex[159:128]*scale]) : (def_vals[159:128]);
result[191:160] = (vmask[191]==1) ? (mem[base+vindex[191:160]*scale]) : (def_vals[191:160]);
result[223:192] = (vmask[223]==1) ? (mem[base+vindex[223:192]*scale]) : (def_vals[223:192]);
result[255:224] = (vmask[255]==1) ? (mem[base+vindex[255:224]*scale]) : (def_vals[255:224]);
```

Returns

A 128/256-bit vector with conditionally gathered single-precision FP values.

_mm_i32gather_ps, _mm256_i32gather_ps

Gathers 2/4 packed single-precision floating point values from memory referenced by the given base address, dword indices, and scale. The corresponding Intel© AVX2 instruction is `VGATHERDPS`.

Syntax

```
extern __m128 _mm_mask_i32gather_ps(float const * base, __m128i vindex, const int
scale);
```

```
extern __m256 _mm256_mask_i32gather_ps(float const * base, __m256i vindex, const int
scale);
```

Arguments

base	the base address used to reference the loaded FP elements.

vindex	the vector of dword indices used to reference the loaded FP elements.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 packed single-precision floating-point values from memory using dword indices.

Below is the pseudo-code for the intrinsics:

`_mm_i32gather_ps()`:

```
result[31:0] = mem[base+vindex[31:0]*scale];
result[63:32] = mem[base+vindex[63:32]*scale];
result[95:64] = mem[base+vindex[95:64]*scale];
result127:96] = mem[base+vindex[127:96]*scale];
```

`_mm256_i32gather_ps()`:

```
result[31:0] = mem[base+vindex[31:0]*scale];
result[63:32] = mem[base+vindex[63:32]*scale];
result[95:64] = mem[base+vindex[95:64]*scale];
result127:96] = mem[base+vindex[127:96]*scale];
result[159:128] = mem[base+vindex[159:128]*scale];
result[191:160] = mem[base+vindex[191:160]*scale];
result[223:192] = mem[base+vindex[223:192]*scale];
result[255:224] = mem[base+vindex[255:224]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered single-precision FP values.

_mm_mask_i64gather_ps, _mm256_mask_i64gather_ps

Gathers 2/4 packed single-precision floating point values from memory referenced by the given base address, qword indices and scale, and using the given single-precision FP mask values. The corresponding Intel® AVX2 instruction is `VGATHERQPS`.

Syntax

```
extern __m128 _mm_mask_i64gather_ps(__m128 def_vals, float const * base, __m128i vindex, __m128 vmask, const int scale);
```

```
extern __m128 _mm256_mask_i64gather_ps(float const * base, __m256i vindex, __m256i vmask, const int scale);
```

Arguments

def_vals	the vector of single-precision FP values copied to the destination when the corresponding element of the single-precision FP mask is '0'.
base	the base address used to reference the loaded FP elements.

vindex	the vector of qword indices used to reference the loaded FP elements.
vmask	the vector of FP elements used as a vector mask; only the most significant bit of each data element is used as a mask.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally load 2/4 packed single-precision floating-point values from memory using qword indices and updates the destination operand. The intrinsic `_mm_mask_i64gather_ps()` also sets the upper 64-bits of the result to '0'.

Below is the pseudo-code for the intrinsics:

`_mm_mask_i64gather_ps():`

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[63:32]);
result[127:64] = 0;
```

`_mm256_mask_i64gather_ps():`

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[63:32]);
result[95:64] = (vmask[95]==1) ? (mem[base+vindex[191:128]*scale]) : (def_vals[95:64]);
result[127:96] = (vmask[127]==1) ? (mem[base+vindex[255:192]*scale]) : (def_vals[127:96]);
```

Returns

A 256/128-bit vector with conditionally gathered single-precision FP values.

_mm_i64gather_ps, _mm256_i64gather_ps

Gathers 2/4 packed single-precision floating point values from memory referenced by the given base address, qword indices and scale. The corresponding Intel® AVX2 instruction is `VGATHERQPS`.

Syntax

```
extern __m128 _mm_mask_i64gather_ps(float const * base, __m128i vindex, const int
scale);
```

```
extern __m128 _mm256_mask_i64gather_ps(float const * base, __m256i vindex, const int
scale);
```

Arguments

base	the base address used to reference the loaded FP elements.
vindex	the vector of qword indices used to reference the loaded FP elements.

scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 packed single-precision floating-point values from memory using qword indices and updates the destination operand. The intrinsic `_mm_i64gather_ps()` also sets the upper 64-bits of the result to '0'.

Below is the pseudo-code for the intrinsics:

`_mm_i64gather_ps()`:

```
result[31:0] = mem[base+vindex[63:0]*scale];
result[63:32] = mem[base+vindex[127:64]*scale];
result[127:64] = 0;
```

`_mm256_i64gather_ps()`:

```
result[31:0] = mem[base+vindex[63:0]*scale];
result[63:32] = mem[base+vindex[127:64]*scale];
result[95:64] = mem[base+vindex[191:128]*scale];
result[127:96] = mem[base+vindex[255:192]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered single-precision FP values.

_mm_mask_i32gather_epi32, _mm256_mask_i32gather_epi32

Gathers 2/4 doubleword values from memory referenced by the given base address, dword indices, and scale, using the given dword mask values. The corresponding Intel® AVX2 instruction is `VPGATHERDD`.

Syntax

```
extern __m128i _mm_mask_i32gather_epi32(__m128i def_vals, int const * base, __m128i vindex, __m128i vmask, const int scale);
```

```
extern __m256i _mm256_mask_i32gather_epi32(__m256i def_vals, int const * base, __m256i vindex, __m256i vmask, const int scale);
```

Arguments

def_val	the vector of dword values copied to the destination when the corresponding element of the vector mask is '0'.
base	the base address used to reference the loaded dword elements.
vindex	the vector of dword indices used to reference the loaded dword elements.
vmask	the vector of dword elements used as a vector mask; only the most significant bit of each dword is used as a mask.

scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally loads 2/4 doubleword values from memory referenced by the given base address, dword indices and scale, and using the given dword mask values.

Below is the pseudo-code for the intrinsics:

`_mm_mask_i32gather_epi32()`:

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[63:32]);
result[95:64] = (vmask[95]==1) ? (mem[base+vindex[95:64]*scale]) : (def_vals[95:64]);
result127:96] = (vmask[127]==1) ? (mem[base+vindex[127:96]*scale]) : (def_vals[127:96]);
```

`_mm256_mask_i32gather_epi32()`:

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[63:32]);
result[95:64] = (vmask[95]==1) ? (mem[base+vindex[95:64]*scale]) : (def_vals[95:64]);
result127:96] = (vmask[127]==1) ? (mem[base+vindex[127:96]*scale]) : (def_vals[127:96]);
result[159:128] = (vmask[159]==1) ? (mem[base+vindex[159:128]*scale]) : (def_vals[159:128]);
result[191:160] = (vmask[191]==1) ? (mem[base+vindex[191:160]*scale]) : (def_vals[191:160]);
result[223:192] = (vmask[223]==1) ? (mem[base+vindex[223:192]*scale]) : (def_vals[223:192]);
result[255:224] = (vmask[255]==1) ? (mem[base+vindex[255:224]*scale]) : (def_vals[255:224]);
```

Returns

A 128/256 vector with conditionally gathered integer32 values.

_mm_i32gather_epi32, _mm256_i32gather_epi32

Gathers 2/4 doubleword values from memory referenced by the given base address, dword indices, and scale. The corresponding Intel® AVX2 instruction is `VPGATHERDD`.

Syntax

`extern __m128i _mm_i32gather_epi32(int const * base, __m128i vindex, const int scale);`

`extern __m256i _mm256_i32gather_epi32(int const * base, __m256i vindex, const int scale);`

Arguments

base	the base address used to reference the loaded dword elements.
vindex	the vector of dword indices used to reference the loaded dword elements.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 doubleword values from memory using the base address, qword indices, and 32-bit scale.

Below is the pseudo-code for the intrinsics:

`_mm_i32gather_epi32()`:

```
result[31:0] = mem[base+vindex[31:0]*scale];
result[63:32] = mem[base+vindex[63:32]*scale];
result[95:64] = mem[base+vindex[95:64]*scale];
result127:96] = mem[base+vindex[127:96]*scale];
```

`_mm256_i32gather_epi32()`:

```
result[31:0] = mem[base+vindex[31:0]*scale];
result[63:32] = mem[base+vindex[63:32]*scale];
result[95:64] = mem[base+vindex[95:64]*scale];
result127:96] = mem[base+vindex[127:96]*scale];
result[159:128] = mem[base+vindex[159:128]*scale];
result[191:160] = mem[base+vindex[191:160]*scale];
result[223:192] = mem[base+vindex[223:192]*scale];
result[255:224] = mem[base+vindex[255:224]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered integer32 values.

_mm_mask_i32gather_epi64,_mm256_mask_i32gather_epi64

Gathers 2/4 quadword values from memory referenced by the given base address, dword indices, and scale, and using the given qword mask values. The corresponding Intel® AVX2 instruction is `VPGATHERDQ`.

Syntax

```
extern __m128i _mm_mask_i32gather_epi64(__m128i def_vals, __int64 const * base, __m128i vindex, __m128i vmask, const int scale);
```

```
extern __m256i _mm256_mask_i32gather_epi64(__m256i def_vals, __int64 const * base, __m128i vindex, __m256i vmask, const int scale);
```

Arguments

def_val	the vector of qword values copied to the destination when the corresponding element of the vector mask is '0'.
base	the base address used to reference the loaded qword elements.
vindex	the vector of dword indices used to reference the loaded qword elements.
vmask	the vector of qword elements used as a vector mask; only the most significant bit of each qword is used as a mask.

scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally load 2/4 quadword values from memory referenced by the given base address, dword indices and scale, and using the given qword mask values.

Below is the pseudo-code for the intrinsics:

`_mm_mask_i32gather_epi64():`

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[127:64]);
```

`_mm256_mask_i32gather_epi64():`

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[31:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[63:32]*scale]) : (def_vals[127:64]);
result[191:128] = (vmask[191]==1) ? (mem[base+vindex[95:64]*scale]) : (def_vals[191:128]);
result[255:192] = (vmask[255]==1) ? (mem[base+vindex[127:96]*scale]) : (def_vals[255:192]);
```

Returns

A 256/128-bit vector with conditionally gathered interger64 values.

_mm_i32gather_epi64,_mm256_i32gather_epi64

Gathers 2/4 quadword values from memory referenced by the given base address, dword indices and scale. The corresponding Intel® AVX2 instruction is `VPGATHERDQ`.

Syntax

```
extern __m128i _mm_i32gather_epi64(__int64 const * base, __m128i vindex, const int scale);
```

```
extern __m256i _mm256_i32gather_epi64(__int64 const * base, __m128i vindex, const int scale);
```

Arguments

base	the base address used to reference the loaded qword elements.
vindex	the vector of dword indices used to reference the loaded qword elements.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 quadword values from memory using the base address, dword indices, and 64-bit scale.

Below is the pseudo-code for the intrinsics:

`_mm_i32gather_epi64()`:

```
result[63:0] = mem[base+vindex[31:0]*scale];
result[127:64] = mem[base+vindex[63:32]*scale];
```

`_mm256_i32gather_epi64()`:

```
result[63:0] = mem[base+vindex[31:0]*scale];
result[127:64] = mem[base+vindex[63:32]*scale];
result[191:128] = mem[base+vindex[95:64]*scale];
result[255:192] = mem[base+vindex[127:96]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered integer64 values.

_mm_mask_i64gather_epi32,_mm256_mask_i64gather_epi32

Gathers 2/4 doubleword values from memory referenced by the given base address, qword indices and scale, and using the given dword mask values. The corresponding Intel© AVX2 instruction is `VPGATHERQD`.

Syntax

```
extern __m128i _mm_mask_i64gather_epi32(__m128i def_vals, int const * base, __m128i vindex, __m128i vmask, const int scale);
```

```
extern __m256i _mm256_mask_i64gather_epi32(__m128i def_vals, int const * base, __m256i vindex, __m128i vmask, const int scale);
```

Arguments

def_val	the vector of dword values copied to the destination when the corresponding element of the vector mask is '0'.
base	the base address used to reference the loaded dword elements.
vindex	the vector of qword indices used to reference the loaded dword elements.
vmask	the vector of dword elements used as a vector mask; only the most significant bit of each dword is used as a mask.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally load 2/4 doubleword values from memory using the base address, qword indices and 32-bit scale. The intrinsic `_mm_mask_i64gather_epi32()` also sets the upper 64-bits of the result to '0'.

Below is the pseudo-code for the intrinsics:

_mm_mask_i64gather_epi32():

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[63:32]);
result[127:64] = 0;
```

_mm256_mask_i64gather_epi32():

```
result[31:0] = (vmask[31]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[31:0]);
result[63:32] = (vmask[63]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[63:32]);
result[95:64] = (vmask[95]==1) ? (mem[base+vindex[191:128]*scale]) : (def_vals[95:64]);
result[127:96] = (vmask[127]==1) ? (mem[base+vindex[255:192]*scale]) : (def_vals[127:96]);
```

Returns

A 128/256-bit vector with conditionally gathered integer32 values.

_mm_i64gather_epi32,_mm256_i64gather_epi32

Gathers 2/4 doubleword values from memory referenced by the given base address, qword indices, and scale. The corresponding Intel® AVX2 instruction is `VPGATHERQD`.

Syntax

`extern __m128i _mm_i64gather_epi32(int const * base, __m128i vindex, const int scale);`

`extern __m128i _mm256_i64gather_epi32(int const * base, __m256i vindex, const int scale);`

Arguments

base	the base address used to reference the loaded dword elements.
vindex	the vector of qword indices used to reference the loaded dword elements.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 doubleword values from memory using the base address, qword indices, and 32-bit scale. The intrinsic `_mm_i64gather_epi32()` also sets the upper 64-bits of the result to '0'.

Below is the pseudo-code for the intrinsics:

_mm_i64gather_epi32():

```
result[31:0] = mem[base+vindex[63:0]*scale];
result[63:32] = mem[base+vindex[127:64]*scale];
result[127:64] = 0;
```

_mm256_i64gather_epi32():

```
result[31:0] = mem[base+vindex[63:0]*scale];
result[63:32] = mem[base+vindex[127:64]*scale];
result[95:64] = mem[base+vindex[191:128]*scale];
result[127:96] = mem[base+vindex[255:192]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered integer32 values.

_mm_mask_i64gather_epi64,_mm256_mask_i64gather_epi64

Gathers 2/4 quadword values from memory referenced by the given base address, qword indices and scale, and using the given qword mask values. The corresponding Intel® AVX2 instruction is `VPGATHERQQ`.

Syntax

```
extern __m128i _mm_mask_i64gather_epi64(__m128i def_vals, __int64 const * base, __m128i
vindex, __m128i vmask, const int scale);
```

```
extern __m256i _mm256_mask_i64gather_epi64(__m128i def_vals, __int64 const * base,
__m256i vindex, __m256i vmask, const int scale);
```

Arguments

def_val	the vector of qword values copied to the destination when the corresponding element of the vector mask is '0'.
base	the base address used to reference the loaded qword elements.
vindex	the vector of qword indices used to reference the loaded qword elements.
vmask	the vector of qword elements used as a vector mask; only the most significant bit of each qword is used as a mask.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics conditionally load 2/4 quadword values from memory using the base address, qword indices and 64-bit scale.

Below is the pseudo-code for the intrinsics:

`_mm_mask_i64gather_epi64()`:

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[127:64]);
```

`_mm256_mask_i64gather_epi64()`:

```
result[63:0] = (vmask[63]==1) ? (mem[base+vindex[63:0]*scale]) : (def_vals[63:0]);
result[127:64] = (vmask[127]==1) ? (mem[base+vindex[127:64]*scale]) : (def_vals[127:64]);
result[191:128] = (vmask[191]==1) ? (mem[base+vindex[191:128]*scale]) : (def_vals[191:128]);
result[255:192] = (vmask[255]==1) ? (mem[base+vindex[255:192]*scale]) : (def_vals[255:192]);
```

Returns

A 128/256-bit vector with conditionally gathered integer64 values.

_mm_i64gather_epi64,_mm256_i64gather_epi64

Gathers 2/4 quadword values from memory referenced by the given base address, qword indices, and scale. The corresponding Intel® AVX2 instruction is `VPGATHERQQ`.

Syntax

```
extern __m128i _mm_i64gather_epi64(__int64 const * base, __m128i vindex, const int scale);
```

```
extern __m256i _mm256_i64gather_epi64(__int64 const * base, __m256i vindex, const int scale);
```

Arguments

base	the base address used to reference the loaded qword elements.
vindex	the vector of qword indices used to reference the loaded qword elements.
scale	The compilation time literal constant, which is used as the vector indices scale to address the loaded elements. Possible values are one of the following: 1, 2, 4, 8.

Description

The intrinsics load 2/4 quadword values from memory using the base address, qword indices, and 64-bit scale.

Below is the pseudo-code for the intrinsics:

`_mm_i64gather_epi64()`:

```
result[63:0] = mem[base+vindex[63:0]*scale];
result[127:64] = mem[base+vindex[127:64]*scale];
```

`_mm256_i64gather_epi64()`:

```
result[63:0] = mem[base+vindex[63:0]*scale];
result[127:64] = mem[base+vindex[127:64]*scale];
result[191:128] = mem[base+vindex[191:128]*scale];
result[255:192] = mem[base+vindex[255:192]*scale];
```

Returns

A 128/256-bit vector with unconditionally gathered integer64 values.

Intrinsics for Logical Shift Operations

_mm256_sll_epi16/32/64

Logical shift of word/doubleword/quadword elements to left according to specified number. The corresponding Intel® AVX2 instruction is `VPSLLW`, `VPSLLD`, or `VPSLLQ`.

Syntax

```
extern __m256i _mm256_sll_epi16(__m256i s1, __m128i count);
extern __m256i _mm256_sll_epi32(__m256i s1, __m128i count);
extern __m256i _mm256_sll_epi64(__m256i s1, __m128i count);
```

Arguments

s1	integer source vector used for the operation
count	128-bit memory location used for the operation

Description

Performs logical shift of bits in the individual data elements (16-bit word, 32-bit doubleword, or 64-bit quadword) in source vector *s1* to the left by the number of bits specified in *count*. The empty low-order bytes are cleared (set to all '0'). If the value specified by *count* is greater than 15/31/63 (depending on the intrinsic being used), the destination vector is set to all '0'.

The *count* argument is a 128-bit memory location. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

Returns

Result of the left-shift operation.

_mm256_slli_epi16/32/64

Logical shift of word/doubleword/quadword elements to left according to specified number. The corresponding Intel© AVX2 instruction is `VPSLLW`, `VPSLLD`, or `VPSLLQ`.

Syntax

```
extern __m256i _mm256_slli_epi16(__m256i s1, int count);
extern __m256i _mm256_slli_epi32(__m256i s1, int count);
extern __m256i _mm256_slli_epi64(__m256i s1, int count);
```

Arguments

s1	integer source vector used for the operation
count	8-bit immediate used for the operation

Description

Performs a logical shift of bits in the individual data elements (words, doublewords, or quadword) in source vector *s1* to the left by the number of bits specified in *count*. The empty low-order bytes are cleared (set to all '0'). If the value specified by *count* is greater than 15/31/63 (depending on the intrinsic being used), the destination vector is set to all 0s. The *count* argument is an 8-bit immediate.

Returns

Result of the left-shift operation.

_mm256_sllv_epi32/64

Logical shift of doubleword/quadword elements to left according variable values. The corresponding Intel® AVX2 instruction is `VPSLLVD` *or* `VPSLLVQ`*.*

Syntax

```
extern __m256i _mm256_sllv_epi32(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_sllv_epi64(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector providing variable values for shift operation

Description

Performs a logical shift of 32 or 64 bits (doublewords, or quadword) in the individual data elements in source vector *s1* to the left by the count value of corresponding data elements in source vector *s2*. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to '0').

The count values are specified individually in each data element of the second source vector. If the unsigned integer value specified in the respective data element of the second source vector is greater than 31 (for a doubleword), or 63 (for a quadword), then the destination data elements are set to '0'.

Returns

Result of the left-shift operation.

_mm_sllv_epi32/64

Logical shift of word/doubleword elements in a 128-bit vector to left according variable values. The corresponding Intel® AVX2 instruction is `VPSLLVD` *or* `VPSLLVQ`*.*

Syntax

```
extern __m128i _mm_sllv_epi32(__m128i s1, __m128i s2);
```

```
extern __m128i _mm_sllv_epi64(__m128i s1, __m128i s2);
```

Arguments

s1	128-bit integer source vector used for the operation
s2	128-bit integer source vector providing variable values for shift operation

Description

Performs a logical shift of 32 or 64 bits (doublewords, or quadword) in the individual data elements in source vector *s1* to the left by the count value of corresponding data elements in source vector *s2*. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to '0').

The count values are specified individually in each data element of the second source vector. If the unsigned integer value specified in the respective data element of the second source vector is greater than 31 (for a doubleword), or 63 (for a quadword), then the destination data elements are set to '0'.

Returns

Result of the left-shift operation.

_mm256_slli_si256

Logical shift of byte elements to left according to specified number. The corresponding Intel® AVX2 instruction is `VPSLLDQ`.

Syntax

```
extern __m256i _mm256_slli_si256(__m256i s1, const int count);
```

Arguments

s1	integer source vector used for the operation
count	8-bit immediate used for the operation

Description

Performs a logical shift of 8-bit [byte] elements within a 128-bit lane of the source vector *s1* to the left by the number of bytes specified in *count*. The empty low-order bytes are cleared (set to all '0'). If the value specified by *count* is greater than 15, the destination vector is set to all 0s. The *count* argument is an 8-bit immediate.

Returns

Result of the left-shift operation.

_mm256_srli_si256

Logical shift of byte elements to right according to specified number. The corresponding Intel® AVX2 instruction is `VPSRLDQ`.

Syntax

```
extern __m256i _mm256_srli_si256(__m256i s1, const int count);
```

Arguments

s1	integer source vector used for the operation
count	8-bit immediate used for the operation

Description

Performs a logical shift of 8-bit [byte] elements within a 128-bit lane of source vector *s1* to the right by the number of bytes specified in *count*. The empty low-order bytes are cleared (set to all '0'). If the value specified by *count* is greater than 15, the destination vector is set to all '0'. The *count* argument is an 8-bit immediate.

Returns

Result of the right-shift operation.

_mm256_srl_epi16/32/64

Logical shift of word/doubleword/quadword elements to right according to specified number. The corresponding Intel® AVX2 instruction is `VPSRLW`*,* `VPSRLD`*, or* `VPSRLQ`*.*

Syntax

```
extern __m256i _mm256_srl_epi16(__m256i s1, __m128i count);
```

```
extern __m256i _mm256_srl_epi32(__m256i s1, __m128i count);
```

```
extern __m256i _mm256_srl_epi64(__m256i s1, __m128i count);
```

Arguments

s1	integer source vector used for the operation
count	128-bit memory location used for the operation

Description

Performs a logical shift of the bits in the individual data elements (16-bit word, 32-bit doubleword, or 64-bit quadword) in source vector *s1* to the right by the number of bits specified in *count*. The empty low-order bytes are cleared (set to all '0'). If the value specified by *count* is greater than 15/31/63 (depending on the intrinsic being used), the destination vector is set to all '0'.

The *count* argument is a 128-bit memory location. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

Returns

Result of the right-shift operation.

_mm256_srli_epi16/32/64

Logical shift of word/doubleword/quadword elements to right according to specified number. The corresponding Intel® AVX2 instruction is `VPSRLW`*,* `VPSRLD`*, or* `VPSRLQ`*.*

Syntax

```
extern __m256i _mm256_srli_epi16(__m256i s1, int count);
```

```
extern __m256i _mm256_srli_epi32(__m256i s1, int count);
```

```
extern __m256i _mm256_srli_epi64(__m256i s1, int count);
```

Arguments

s1	integer source vector used for the operation
count	8-bit immediate used for the operation

Description

Performs a logical shift of bits in the individual data elements (16-bit word, 32-bit doubleword, or 64-bit quadword) in source vector *s1* to the right by the number of bits specified in *count*. The empty low-order bytes are cleared (set to all '0'). If the value specified by *count* is greater than 15/31/63 (depending on the intrinsic being used), the destination vector is set to all '0'. The *count* argument is an 8-bit immediate.

Returns

Result of the right-shift operation.

_mm256_srlv_epi32/64

Logical shift of doubleword/quadword elements to right according variable values. The corresponding Intel© AVX2 instruction is `VPSRLVD` or `VPSRLVQ`.

Syntax

```
extern __m256i _mm256_srlv_epi32(__m256i s1, __m256i s2);
```

```
extern __m256i _mm256_srlv_epi64(__m256i s1, __m256i s2);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector providing variable values for shift operation

Description

Performs a logical shift of 32 or 64 bits (doublewords, or quadword) in the individual data elements in source vector *s1* to the right by the count value of corresponding data elements in source vector *s2*. As the bits in the data elements are shifted right, the empty low-order bits are cleared (set to '0').

The count values are specified individually in each data element of the second source vector. If the unsigned integer value specified in the respective data element of the second source vector is greater than 31 (for a doubleword), or 63 (for a quadword), then the destination data elements are set to '0'.

Returns

Result of the right-shift operation.

_mm_srlv_epi32/64

Logical shift of word/doubleword elements in a 128-bit vector to right according variable values. The corresponding Intel© AVX2 instruction is `VPSRLVD` or `VPSRLVQ`.

Syntax

```
extern __m128i _mm_srlv_epi32(__m128i s1, __m128i s2);
```

```
extern __m128i _mm_srlv_epi64(__m128i s1, __m128i s2);
```

Arguments

s1	128-bit integer source vector used for the operation
s2	128-bit integer source vector providing variable values for shift operation

Description

Performs a logical shift of 32 or 64 bits (doublewords, or quadword) in the individual data elements in source vector *s1* to the right by the count value of corresponding data elements in the source vector *s2*. As the bits in the data elements are shifted right, the empty low-order bits are cleared (set to '0').

The count values are specified individually in each data element of the second source vector. If the unsigned integer value specified in the respective data element of the second source vector is greater than 31 (for a doubleword), or 63 (for a quadword), then the destination data element are set to '0'.

Returns

Result of the right-shift operation.

Intrinsics for Insert/Extract Operations

_mm256_inserti128_si256

Inserts 128-bits of packed integer data of the second source vector into the destination vector at a 128-bit offset from imm8[0]. The corresponding Intel® AVX2 instruction is `VINSERTI128`*.*

Syntax

```
extern __m256i _mm256_inserti128_si256(__m256i a, __m128i b, const int mask);
```

Arguments

a	integer source vector
b	integer source vector
mask	integer constant specifying offset

Description

Inserts 128-bits of packed integer data from the second source operand (third operand) into the destination operand (first operand) at a 128-bit offset from `imm8[0]`. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The high 7 bits of the immediate are ignored.

Returns

_mm256_extracti128_si256

Extracts 128-bits of packed integer data of the second source vector into the destination vector at a 128-bit offset from imm8[0]. The corresponding Intel® AVX2 instruction is `VEXTRACTI128`*.*

Syntax

```
extern __m128i _mm256_extracti128_si256(__m256i a, int offset);
```

Arguments

a	integer source vector
offset	integer constant specifying offset

Description

Extract 128 bits (composed of integer data) from a, selected with imm, and store the result in dst.

Extracts 128-bits of packed integer data from source vector *a* with *offset*. The remaining portions of the destination are written by the corresponding fields of the source vector. The destination may be either an *XMM* register or a 128-bit memory location. The high 7 bits of the immediate are ignored.

Returns

_mm256_insert_epi8/16/32/64

Insert 8/16/32/64-bit integer into a vector of integers at the position specified by index.

Syntax

```
extern __m256i _mm256_insert_epi8(__m256i a, int8 i, const int index);
extern __m256i _mm256_insert_epi16(__m256i a, int16 i, const int index);
extern __m256i _mm256_insert_epi32(__m256i a, int32 i, const int index);
extern __m256i _mm256_insert_epi64(__m256i a, int64 i, const int index);
```

Arguments

a	integer source vector
i	integer value to insert
offset	integer constant specifying offset

Description

Insert an integer value, *i* into the corresponding position of an integer source vector, *a*, and return the resulting vector.

_mm256_extract_epi8/16/32/64

Extract integer byte or word from packed integer array element selected by index.

Syntax

```
extern int _mm256_extract_epi8(__m256i a, int offset);
extern int _mm256_extract_epi16(__m256i a, int offset);
extern int _mm256_extract_epi32(__m256i a, int offset);
extern int _mm256_extract_epi64(__m256i a, int offset);
```

Arguments

a	integer source vector
offset	integer constant specifying offset

Description

Returns extracted 8/16/32/64 bits of data of the source vector at offset position. Offset counts with element size granularity.

Upper bits of returned integer value are cleared.

Intrinsics for Masked Load/Store Operations

_mm_maskload_epi32/64, _mm256_maskload_epi32/64

Conditionally loads dwords/qwords from the specified memory location, depending on the mask bits associated with each data element. The corresponding Intel® AVX2 instruction is `VPMASKMOVD` *or* `VPMASKMOVQ`.

Syntax

```
extern __m128i _mm_maskload_epi32(int const * addr, __m128i mask);
extern __m256i _mm256_maskload_epi32(int const * addr, __m256i mask);
extern __m128i _mm_maskload_epi64(__int64 const * addr, __m128i mask);
extern __m256i _mm256_maskload_epi64(__int64 const * addr, __m256i mask);
```

Arguments

addr	pointer to data to be loaded
mask	integer source vector

Description

Conditionally loads 32/64-bit data elements from the memory referenced by the `addr` and stores it into the corresponding data element of the result vector. If an element of mask is 0, the 32/64-bit zero is written to the corresponding element of the result vector. The mask bit for each data element is the most significant bit of that element in mask.

Returns

Result of the masked load operation.

_mm_maskstore_epi32/64, _mm256_maskstore_epi32/64

Conditionally stores dwords/qwords from the source vector to the specified memory location, depending on the given mask bits associated with each data element. The corresponding Intel® AVX2 instruction is `VPMASKMOVD` *or* `VPMASKMOVQ`.

Syntax

```
extern void _mm_maskstore_epi32(int * addr, __m128i vmask, __m128i val);
extern void _mm256_maskstore_epi32(int * addr, __m256i vmask, __m256i val);
extern void _mm_maskstore_epi64(__int64 * addr, __m128i vmask, __m128i val);
extern void _mm256_maskstore_epi64(__int64 * addr, __m256i vmask, __m256i val);
```

Arguments

addr	pointer to data to be loaded
vmask	vector mask. If element of vmask is 0, then the value in the memory is unchanged

val	location from where the elements are written to vector located in memory and referenced by "*addr*"

Description

Conditionally stores 32/64-bit data elements from the source vector into the corresponding elements of the vector in memory referenced by `addr`. If an element of mask is 0, corresponding element of the result vector in memory stays unchanged. Only the most significant bit of each element in the vector mask is used.

Returns

Result of the masked store operation.

Intrinsics for Miscellaneous Operations

_mm256_alignr_epi8

Aligns elements of two source vectors depending on bits in a mask. The corresponding Intel® AVX2 instruction is `VPALIGNR`*.*

Syntax

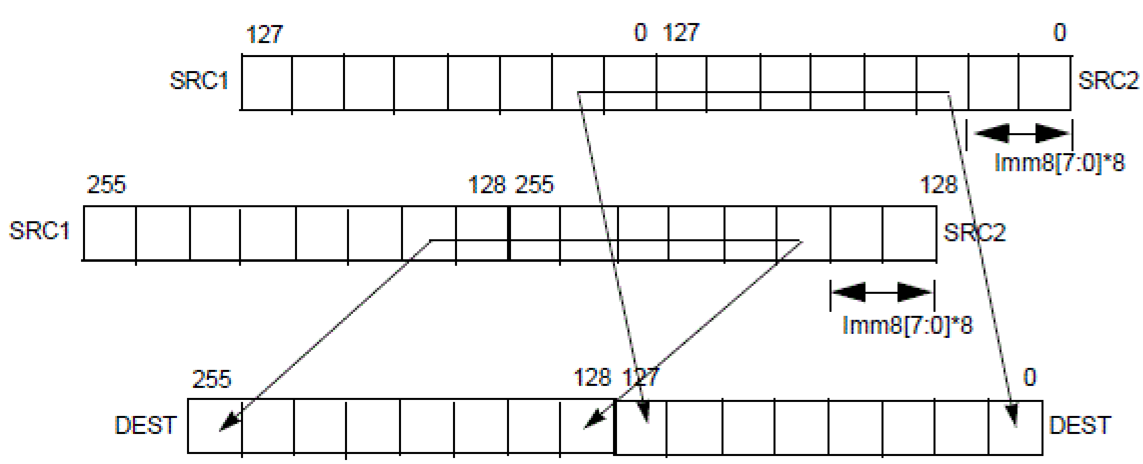
```
extern __m256i _mm256_alignr_epi8(__m256i s1, __m256i s2, const int mask);
```

Arguments

s1	integer source vector used for the operation
s2	integer source vector used for the operation
mask	8-bit immediate bits used for the operation

Description

Performs an alignment operation by concatenating two blocks of 16-byte data from the first and second source vectors, *s1* and *s2*, into an intermediate 32-byte composite, shifting the composite at byte granularity to the right by a constant immediate specified by mask, and extracting the right-aligned 16-byte result into the destination vector. The immediate value is considered unsigned.



Returns

Result of the alignment operation.

_mm256_movemask_epi8

Moves byte mask from source vector. The corresponding Intel® AVX2 instruction is `VPMOVMSKB`*.*

Syntax

```
extern int _mm256_movemask_epi8(__m256i s1);
```

Arguments

s1	integer source vector used for the operation.

Description

Creates a mask from the most significant bit of each byte of source vector *s1* and stores the result in the in the returned doubleword value/mask.

Returns

Result of the move mask operation.

_mm256_stream_load_si256

Loads 256-bit data from memory using non-temporal aligned hint. The corresponding Intel® AVX2 instruction is `VMOVNTDQA`*.*

Syntax

```
extern __m256i _mm256_stream_load_si256(__m256i const *);
```

Arguments

s1	integer source vector used for the operation.

Description

Loads 256-bit data from the source operand to the destination operand using a non-temporal hint if the memory source is write combining memory type.

Intrinsics for Operations to Manipulate Integer Data at Bit-Granularity

_bextr_u32/64

Extracts contiguous bits from the first source operand to the destination using an index value and length value specified in the second source operand. The corresponding Intel® AVX2 instruction is `BEXTR`*.*

Syntax

```
extern unsigned int _bextr_u32(unsigned int source, unsigned int sb, unsigned int bl);
```

```
extern unsigned __int64 _bextr_u64(unsigned __int64 s1, unsigned int sb, unsigned int
bl);
```

Arguments

source	the source from where the bits are extracted
sb	start bit, the number of the bit from where the contiguous bits are extracted
bl	bit length, the number of bits to be extracted

Description

Extracts contiguous bits from the first source operand to the destination using an index value and length value specified in the second source operand. The extracted bits are written to the destination starting from the least significant bit. All higher order bits in the destination starting at bit position *bl* are zeroed. The destination is cleared if no bits are extracted.

Returns

Result of the operation.

_blsi_u32/64

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination. The corresponding Intel® AVX2 instruction is `BLSI`*.*

Syntax

```
extern unsigned int _blsi_u32(unsigned int source);
```

```
extern unsigned __int64 _blsi_u64(unsigned __int64 source);
```

Arguments

source	the source from where the bits are extracted

Description

Extracts the lowest set bit from the source operand and sets the corresponding bit in the destination. All other bits in the destination are set to 0. If no bits are set in the source operand, all the bits in the destination are set to 0.

Returns

Result of the operation.

_blsmsk_u32/64

Sets all the lower bits of the destination to "1" up to and including lowest set bit (=1) in the source operand. The corresponding Intel® AVX2 instruction is `BLSMSK`*.*

Syntax

```
extern unsigned int _blsmsk_u32(unsigned int s1);
```

```
extern unsigned __int64 _blsmsk_u64(unsigned __int64 s1);
```

Arguments

s1	the source operand used for the operation

Description

Sets all the lower bits of the destination to "1" up to and including lowest set bit (=1) in the source operand. If source operand is 0, all bits of the destination are set to 1.

Returns

Result of the operation

_blsr_u32/64

Copies all bits from the source operand to the destination and resets (=0) the bit position in the destination that corresponds to the lowest set bit of the source operand. The corresponding Intel® AVX2 instruction is `BLSR`*.*

Syntax

```
extern unsigned int _blsr_u32(unsigned int s1);
```

```
extern unsigned __int64 _blsr_u64(unsigned __int64 s1);
```

Arguments

s1	the source operand from where the bits are copied

Description

Copies all bits from the source operand to the destination and resets (=0) the bit position in the destination that corresponds to the lowest set bit of the source operand.

Returns

Result of the operation

_bzhi_u32/64

Copies the bits of the first source operand into the destination and clears the higher bits in the destination according to the index value specified by the second source operand. The corresponding Intel® AVX2 instruction is `BZHI`*.*

Syntax

```
extern unsigned int _bzhi_u32(unsigned int source, unsigned int index);
```

```
extern unsigned __int64 _bzhi_u64(unsigned __int64 source, unsigned int index);
```

Arguments

source	the source operand from where the bits are copied
index	index value according to which the bits are copied

Description

Copies the bits of the first source operand into the destination and clears the higher bits in the destination according to the index value. The index value is specified by bits `7:0` of the second source operand.

Returns

Result of the operation.

_pext_u32/64

Transfer either contiguous or non-contiguous bits in the first source operand to contiguous low order bit positions in the destination according to the mask values. The corresponding Intel® AVX2 instruction is `PEXT`.

Syntax

```
extern unsigned int _pext_u32(unsigned int source, unsigned int mask);
```

```
extern unsigned __int64 _pext_u64(unsigned __int64 s1, unsigned __int64 mask);
```

Arguments

source	the source operand from where the bits are transferred
mask	mask value according to which the bits are transferred

Description

The intrinsics use a mask in the second source operand to transfer either contiguous or non-contiguous bits in the first source operand to contiguous, low-order bit positions in the destination. For each bit set in the mask, the intrinsic extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of the destination. The remaining upper bits of the destination are set to 0.

Returns

Result of the operation

_pdep_u32/64

Transfer/scatter contiguous low order bits in the first source operand into the destination according to the mask in the second source operand. The corresponding Intel® AVX2 instruction is `PDEP`.

Syntax

```
extern unsigned int _pdep_u32(unsigned int source, unsigned int mask);
```

```
extern unsigned __int64 _pdep_u64(unsigned __int64 source, unsigned __int64 mask);
```

Arguments

source	the source operand from where the bits are transferred

mask	the mask value according to which the bits are transferred

Description

The intrinsics use a mask in the second source operand to transfer/scatter contiguous, low-order bits in the first source operand into the destination. It takes the low bits from the first source operand and deposit them in the destination at the corresponding bit locations that are set in the mask. All other bits (bits not set in mask) in the destination are set to 0.

Returns

Result of the operation

_lzcnt_u32/64

Counts the number of leading zero bits in a source operand. Returns operand size as output when source operand is zero. The corresponding Intel® AVX2 instruction is `LZCNT`*.*

Syntax

```
extern unsigned int _lzcnt_u32(unsigned int source);
```

```
extern unsigned __int64 _lzcnt_u64(unsigned __int64 source);
```

Arguments

source	the source operand used for the operation

Description

Counts the number of leading most significant zero bits in a source operand returning the result into a destination when source operand is 0.

Returns

Result of the operation.

_tzcnt_u32/64

Counts the number of trailing least significant zero bits in source operand and returns the result in destination. When source operand is 0, it returns its size in bits. The corresponding Intel® AVX2 instruction is `TZCNT`*.*

Syntax

```
extern unsigned int _tzcnt_u32(unsigned int source);
```

```
extern unsigned __int64 _tzcnt_u64(unsigned __int64 source);
```

Arguments

source	the source operand used for the operation

Description

Searches the source operand for the least significant set bit. If a least significant 1 bit is found, its bit index is returned, otherwise the result is the number of bits in the operand size.

Returns

Result of the operation.

Intrinsics for Pack/Unpack Operations

_mm256_packs_epi16/32

Pack signed word/doubleword integers to signed byte/ words integers and saturates. The corresponding Intel© AVX2 instruction is `VPACKSSWB` *or* `VPACKSSDW`*.*

Syntax

```
extern __m256i _mm256_packs_epi16(__m256i a, __m256i b);
extern __m256i _mm256_packs_epi32(__m256i a, __m256i b);
```

Arguments

a	integer source vector used for the operation
b	integer source vector used for the operation

Description

The `_mm256_packs_epi16` intrinsic converts 16 packed signed word integers from the first and the second source operands into 32 packed signed byte integers. The `_mm256_packs_epi32` intrinsic converts eight packed signed doubleword integers from the first and the second source operands into 16 packed signed word integers.

Returns

Result of the pack operation.

_mm256_packus_epi16/32

Pack signed word/doubleword integers to unsigned byte/word integers and saturates. The corresponding Intel© AVX2 instruction is `VPACKUSWB` *or* `VPACKUSDW`*.*

Syntax

```
extern __m256i _mm256_packus_epi16(__m256i a, __m256i b);
extern __m256i _mm256_packus_epi32(__m256i a, __m256i b);
```

Arguments

a	integer source operand used for the operation
b	integer source operand used for the operation

Description

The `_mm256_packus_epi16` intrinsic converts 16 packed signed word integers from source operands *a* and *b* into 32 packed unsigned byte integers. The `_mm256_packus_epi32` intrinsic converts eight packed signed doubleword integers from the source operands *a* and *b* into 16 packed unsigned word integers.

Returns

Result of the pack operation.

_mm256_unpackhi_epi8/16/32/64

Unpacks and interleaves the high-order data elements of the source vector with the high-order data elements in the destination vector. The corresponding Intel® AVX2 instruction is `VPUNPCKHBW`*,* `VPUNPCKHWD`*,* `VPUNPCKHDQ`*, or* `VPUNPCKHQDQ`*.*

Syntax

```
extern __m256i _mm256_unpackhi_epi8(__m256i a, __m256i b);
extern __m256i _mm256_unpackhi_epi16(__m256i a, __m256i b);
extern __m256i _mm256_unpackhi_epi32(__m256i a, __m256i b);
extern __m256i _mm256_unpackhi_epi64(__m256i a, __m256i b);
```

Arguments

a	integer source operand used for the operation
b	integer source operand used for the operation

Description

Unpacks and interleaves the high-order signed or unsigned data elements (bytes, words, doublewords, and quadwords) of the source vector and the high-order signed or unsigned data elements (bytes, words, doublewords, and quadwords) in the destination vector. The low-order data elements are ignored.

Returns

Result of the interleave operation

_mm256_unpacklo_epi8/16/32/64

Unpacks and interleaves the low-order data elements of the source vector with the low-order data elements in the destination vector. The corresponding Intel® AVX2 instruction is `VPUNPCKLBW`*,* `VPUNPCKLWD`*,* `VPUNPCKLDQ`*, or* `VPUNPCKLQDQ`*.*

Syntax

```
extern __m256i _mm256_unpacklo_epi8(__m256i a, __m256i b);
extern __m256i _mm256_unpacklo_epi16(__m256i a, __m256i b);
extern __m256i _mm256_unpacklo_epi32(__m256i a, __m256i b);
extern __m256i _mm256_unpacklo_epi64(__m256i a, __m256i b);
```

Arguments

a	integer source vector used for the operation
b	integer source vector used for the operation

Description

Unpacks and interleaves the low-order signed or unsigned data elements (bytes, words, doublewords, and quadwords) of the source vector and the low-order signed or unsigned data elements (bytes, words, doublewords, and quadwords) in the destination operand. The high-order data elements are ignored.

Returns

Result of the interleave operation

Intrinsics for Packed Move with Extend Operations

_mm256_cvtepi8_epi16/32/64

Performs packed move with sign-extend on 8-bit signed integers to 16/32/64-bit integers. The corresponding Intel© AVX2 instruction is `VPMOVSXBW`*,*`VPMOVSXBD`*, or* `VPMOVSXBQ`*.*

Syntax

```
extern __m256i _mm256_cvtepi8_epi16(__m128i s1);
```

```
extern __m256i _mm256_cvtepi8_epi32(__m128i s1);
```

```
extern __m256i _mm256_cvtepi8_epi64(__m128i s1);
```

Arguments

s1	128-bit integer source vector used for the operation

Description

Performs a packed move with sign-extend operation to convert 8-bit [byte] integers in the low bytes of the source vector, *s1*, to 16-bit [word], 32-bit [doubleword], or 64-bit [quadword] integers, which are stored as packed signed word/doubleword/quadword integers in the destination vector.

Returns

Result of the sign-extend operation.

_mm256_cvtepi16_epi32/64

Performs packed move with sign-extend on 16-bit signed integers to 32/64-bit integers. The corresponding Intel© AVX2 instruction is `VPMOVSXWD` *or*`VPMOVSXWQ`*.*

Syntax

```
extern __m256i _mm256_cvtepi16_epi32(__m128i s1);
```

```
extern __m256i _mm256_cvtepi16_epi64(__m128i s1);
```

Arguments

s1	128-bit integer source vector used for the operation

Description

Performs a packed move with sign-extend operation to convert 16-bit [word] integers in the low bytes of the source vector, *s1*, to 32-bit [doubleword] or 64-bit [quadword] integers and stored as packed signed doubleword/quadword integers in the destination vector.

Returns

Result of the sign-extend operation.

_mm256_cvtepi32_epi64

Performs packed move with sign-extend on 32-bit signed integers to 64-bit integers. The corresponding Intel® AVX2 instruction is `VPMOVSXDQ`.

Syntax

```
extern __m256i _mm256_cvtepi32_epi64(__m128i s1);
```

Arguments

s1	128-bit integer source vector used for the operation

Description

Performs a packed move with sign-extend operation to convert 32-bit [doubleword] integers in the low bytes of the source vector, *s1*, to 64-bit [quadword] integers and stored as packed signed quadword integers in the destination vector.

Returns

Result of the sign-extend operation.

_mm256_cvtepu8_epi16/32/64

Performs packed move with zero-extend on 8-bit unsigned integers to 16/32/64-bit integers. The corresponding Intel® AVX2 instruction is `VPMOVZXBW`,`VPMOVZXBD`, *or* `VPMOVZXBQ`.

Syntax

```
extern __m256i _mm256_cvtepu8_epi16(__m128i s1);
extern __m256i _mm256_cvtepu8_epi32(__m128i s1);
extern __m256i _mm256_cvtepu8_epi64(__m128i s1);
```

Arguments

s1	128-bit integer source vector used for the operation

Description

Performs a packed move with zero-extend operation to convert 8-bit [byte] integers in the low bytes of the source vector, *s1*, to 16-bit [word], 32-bit [doubleword], or 64-bit [quadword] integers and stored as packed unsigned word/doubleword/quadword integers in the destination vector.

Returns

Result of the zero-extend operation.

_mm256_cvtepu16_epi32/64

Performs packed move with zero-extend on 16-bit unsigned integers to 32/64-bit integers. The corresponding Intel® AVX2 instruction is `VPMOVZXWD` *or* `VPMOVZXWQ`*.*

Syntax

```
extern __m256i _mm256_cvtepu16_epi32(__m128i s1);
```

```
extern __m256i _mm256_cvtepu16_epi64(__m128i s1);
```

Arguments

s1	128-bit integer source vector used for the operation

Description

Performs a packed move with zero-extend operation to convert 16-bit [word] integers in the low bytes of the source vector, *s1*, to 32-bit [doubleword] or 64-bit [quadword] integers and stored as packed signed doubleword/quadword integers in the destination vector.

Returns

Result of the zero-extend operation.

_mm256_cvtepu32_epi64

Performs packed move with zero-extend on 32-bit unsigned integers to 64-bit integers. The corresponding Intel® AVX2 instruction is `VPMOVZXDQ`*.*

Syntax

```
extern __m256i _mm256_cvtepu32_epi64(__m128i s1);
```

Arguments

s1	128-bit integer source vector used for the operation

Description

Performs a packed move with zero-extend operation to convert 32-bit [doubleword] integers in the low bytes of the source vector, *s1*, to 64-bit [quadword] integers and stored as packed signed quadword integers in the destination vector.

Returns

Result of the zero-extend operation.

Intrinsics for Permute Operations

_mm256_permutevar8x32_epi32

Permutes doubleword elements of the source vector into the destination vector. The corresponding Intel® AVX2 instruction is `VPERMD`*.*

Syntax

```
extern __m256i _mm256_permutevar8x32_epi32(__m256i val, __m256i offsets);
```

Arguments

val	the vector of 32-bit integer elements to be permuted
offsets	the vector of eight 3-bit offsets (specifying values in range [0 - 7]) for the permuted elements of 256-bit vector

Description

Use the offset values in each dword element of the vector *offsets* to select a dword element from the source vector *val*. The result element is copied to the corresponding element of destination vector. The intrinsic does NOT allow to copy the same element of the source vector to more than one element of the destination vector.

Below is the pseudo-code for the intrinsic:

```
RESULT[31:0] <- (VAL[255:0] >> (OFFSETS[2:0] * 32))[31:0];
RESULT[63:32] <- (VAL[255:0] >> (OFFSETS[34:32] * 32))[31:0];
RESULT[95:64] <- (VAL[255:0] >> (OFFSETS[66:64] * 32))[31:0];
RESULT[127:96] <- (VAL[255:0] >> (OFFSETS[98:96] * 32))[31:0];
RESULT[159:128] <- (VAL[255:0] >> (OFFSETS[130:128] * 32))[31:0];
RESULT[191:160] <- (VAL[255:0] >> (OFFSETS[162:160] * 32))[31:0];
RESULT[223:192] <- (VAL[255:0] >> (OFFSETS[194:192] * 32))[31:0];
RESULT[255:224] <- (VAL[255:0] >> (OFFSETS[226:224] * 32))[31:0];
```

Returns

Result of the permute operation.

_mm256_permutevar8x32_ps

Permutes single-precision floating-point elements of the source vector into the destination vector. The corresponding Intel® AVX2 instruction is `VPERMPS`.

Syntax

```
extern __m256i _mm256_permutevar8x32_ps(__m256 val, __m256i offsets);
```

Arguments

val	the vector of 32-bit single-precision floating-point elements to be permuted
offsets	the vector of eight 3-bit offsets (specifying values in range [0 - 7]) for the permuted elements of 256-bit vector

Description

Use the offset values in each dword element of the vector *offsets* to select a single-precision floating-point element from the source vector *val*. The result element is copied to the corresponding element of destination vector. The intrinsic does NOT allow to copy the same element of the source vector to more than one element of the destination vector.

Below is the pseudo-code for the intrinsic:

```
RESULT[31:0] <- (VAL[255:0] >> (OFFSETS[2:0] * 32))[31:0];
RESULT[63:32] <- (VAL[255:0] >> (OFFSETS[34:32] * 32))[31:0];
RESULT[95:64] <- (VAL[255:0] >> (OFFSETS[66:64] * 32))[31:0];
RESULT[127:96] <- (VAL[255:0] >> (OFFSETS[98:96] * 32))[31:0];
RESULT[159:128] <- (VAL[255:0] >> (OFFSETS[130:128] * 32))[31:0];
RESULT[191:160] <- (VAL[255:0] >> (OFFSETS[162:160] * 32))[31:0];
RESULT[223:192] <- (VAL[255:0] >> (OFFSETS[194:192] * 32))[31:0];
RESULT[255:224] <- (VAL[255:0] >> (OFFSETS[226:224] * 32))[31:0];
```

Returns

Result of the permute operation.

_mm256_permute4x64_epi64

Permutes quadword integer values of the source vector into the destination vector. The corresponding Intel© AVX2 instruction is `VPERMQ`*.*

Syntax

```
extern __m256i _mm256_permute4x64_epi64(__m256i val, const int control);
```

Arguments

val	the vector of 64-bit quadword integer elements to be permuted
control	an integer specified as an 8-bit immediate

Description

Use two-bit index values in the immediate byte to select a qword integer element from the source vector *val*. The result element is copied to the corresponding element of destination vector. The intrinsic allows to copy the same element of the source vector to more than one element of the destination vector.

Below is the pseudo-code for the intrinsic:

```
RESULT[63:0] <- (VAL[255:0] >> (CONTROL[1:0] * 64))[63:0];
RESULT[127:64] <- (VAL[255:0] >> (CONTROL[3:2] * 64))[63:0];
RESULT[191:128] <- (VAL[255:0] >> (CONTROL[5:4] * 64))[63:0];
RESULT[255:192] <- (VAL[255:0] >> (CONTROL[7:6] * 64))[63:0];
```

Returns

Result of the permute operation.

_mm256_permute4x64_pd

Permutes quadword double-precision floating-point values of the source vector into the destination vector. The corresponding Intel© AVX2 instruction is `VPERMPD`*.*

Syntax

```
extern __m256i _mm256_permute4x64_epi64(__m256d val, const int control);
```

Arguments

val	the vector of 64-bit qword double-precision floating-point elements to be permuted
control	an integer specified as an 8-bit immediate

Description

Use two-bit index values in the immediate byte to select a qword double-precision floating-point element from the source vector *val*. The result element is copied to the corresponding element of destination vector. The intrinsic allows to copy the same element of the source vector to more than one element of the destination vector.

Below is the pseudo-code for the intrinsic:

```
RESULT[63:0] <- (VAL[255:0] >> (CONTROL[1:0] * 64))[63:0];
RESULT[127:64] <- (VAL[255:0] >> (CONTROL[3:2] * 64))[63:0];
RESULT[191:128] <- (VAL[255:0] >> (CONTROL[5:4] * 64))[63:0];
RESULT[255:192] <- (VAL[255:0] >> (CONTROL[7:6] * 64))[63:0];
```

Returns

Result of the permute operation.

_mm256_permute2x128_si256

Permutes 128-bit integer data from the first source vector and the second source vector in the destination vector. The corresponding Intel® AVX2 instruction is `VPERM2I128`*.*

Syntax

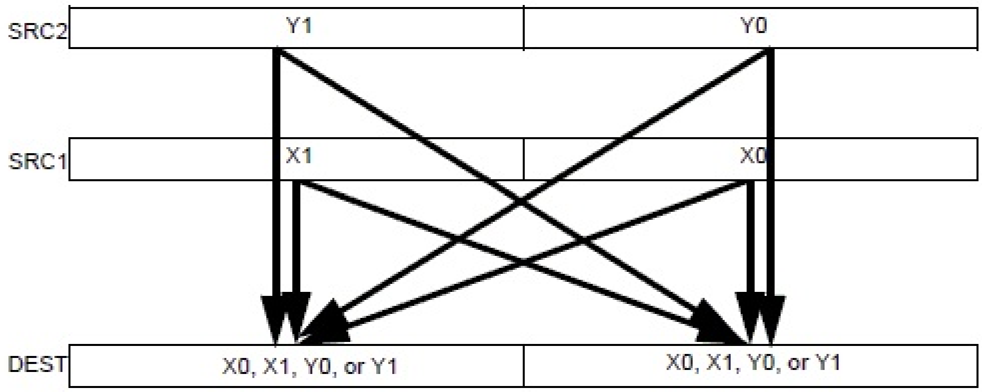
```
extern __m256i _mm256_permute2x128_si256(__m256i a, __m256i b, int control);
```

Arguments

a	integer source vector
b	integer source vector
control	8-bit immediate used for the operation

Description

Permutes 128-bit integer data from source vector `a` and source vector `b` using bits in the 8-bit immediate and stores results in the destination vector.



Returns

Result of the permute operation.

Intrinsics for Shuffle Operations

_mm256_shuffle_epi8

Shuffles bytes in the first source vector according to the shuffle control mask in the second source vector. The corresponding Intel© AVX2 instruction is `VPSHUFB`.

Syntax

`extern __m256i _mm256_shuffle_epi8(__m256i a, __m256i b);`

Arguments

a	integer source vector
b	integer source vector

Description

Performs shuffle operations of the signed or unsigned 8-bit integers in the source vector as specified by the shuffle control mask in the second source operand.

Below is the pseudocode interpreting `a`, `b`, and `r` as arrays of unsigned 8-bit integers:

```
for (i = 0; i < 16; i++){
  if (b[i] & 0x80){
    r[i] =  0;
  }
  else{
    r[i] = a[b[i] & 0x0F];
  }
  if (b[16+i] & 0x80){
    r[16+i] =  0;
  }
  else{
    r[16+i] = a[16+(b[16+i] & 0x0F)];
  }
}
```

Returns

Result of the shuffle operation.

_mm256_shuffle_epi32

Conditionally shuffles doublewords of the source vector in the destination vector at the locations selected with the immediate control operand. The corresponding Intel® AVX2 instruction is `VPSHUFD`.

Syntax

```
extern __m256i _mm256_shuffle_epi32(__m256i val, const int control);
```

Arguments

val	integer source vector
control	immediate control operand

Description

Performs shuffle operations of the signed or unsigned 32-bit integers in the source vector as specified by the control operand. The shuffle value must be an immediate.

Returns

Result of the shuffle operation.

_mm256_shufflehi_epi16

Shuffles the upper 4 high signed or unsigned words in each 128-bit lane of the source operand according to the shuffle control operand. The low qwords in each of 2 128-bit lanes of the source operand are copied to the corresponding low qwords of the result value. The corresponding Intel® AVX 2 instruction is `VPSHUFHW`.

Syntax

```
extern __m256i _mm256_shufflehi_epi16(__m256i val, const int control);
```

Arguments

val	integer source operand
control	immediate control mask

Description

Shuffles the upper four high signed or unsigned words in each 128-bit lane of the source operand according to the shuffle control operand. The low qwords in each of two 128-bit lanes of the source operand are copied to the corresponding low qwords of the result value. The shuffle control operand must be an immediate.

Below is the pseudo-code for the intrinsic:

```
RESULT[63:0] <- VAL[63:0]
RESULT[79:64] <- (VAL >> (CONTROL[1:0] *16))[79:64]
RESULT[95:80] <- (VAL >> (CONTROL[3:2] * 16))[79:64]
RESULT[111:96] <- (VAL >> (CONTROL[5:4] * 16))[79:64]
RESULT[127:112] <- (VAL >> (CONTROL[7:6] * 16))[79:64]
```

```
RESULT[191:128] <- VAL[191:128]
RESULT[207192] <- (VAL >> (CONTROL[1:0] *16))[207:192]
RESULT[223:208] <- (VAL >> (CONTROL[3:2] * 16))[207:192]
RESULT[239:224] <- (VAL >> (CONTROL[5:4] * 16))[207:192]

RESULT[255:240] <- (VAL >> (CONTROL[7:6] * 16))[207:192]
```

Returns

Result of the shuffle operation.

_mm256_shufflelo_epi16

Shuffles the low 4 signed or unsigned words in each 128-bit lane of the source operand according to the shuffle control operand. The high qwords in each of 2 128-bit lanes of the source operand are copied to the corresponding high qwords of the result value. The corresponding Intel© AVX 2 instruction is `VPSHUFLW`.

Syntax

```
extern __m256i _mm256_shufflelo_epi16(__m256i val, const int control);
```

Arguments

val	integer source vector
control	immediate control operand

Description

Shuffles the low four signed or unsigned words in each 128-bit lane of the source operand according to the shuffle control operand. The high qwords in each of 2 128-bit lanes of the source operand are copied to the corresponding high qwords of the result value. The shuffle value must be an immediate.

Below is the pseudo-code for the intrinsic:

```
RESULT[15:0] <- (VAL >> (CONTROL[1:0] *16))[15:0]
RESULT[31:16] <- (VAL >> (CONTROL[3:2] * 16))[15:0]
RESULT[47:32] <- (VAL >> (CONTROL[5:4] * 16))[15:0]
RESULT[63:48] <- (VAL >> (CONTROL[7:6] * 16))[15:0]
RESULT[127:64] <- VAL[127:64]
RESULT[143:128] <- (VAL >> (CONTROL[1:0] *16))[143:128]
RESULT[159:144] <- (VAL >> (CONTROL[3:2] * 16))[143:128]
RESULT[175:160] <- (VAL >> (CONTROL[5:4] * 16))[143:128]
RESULT[191:176] <- (VAL >> (CONTROL[7:6] * 16))[143:128]

RESULT[255:192] <- VAL[255:192]
```

Returns

Result of the shuffle operation.

Intrinsics for Intel® Transactional Synchronization Extensions (Intel® TSX)

Intel® Transactional Synchronization Extensions (Intel® TSX) Overview

Intel® Transactional Synchronization Extensions (Intel® TSX) allow the processor to determine dynamically whether threads need to serialize through lock-protected critical sections, and to perform serialization only when required. This lets the processor to expose and exploit concurrence hidden in an application due to dynamically unnecessary synchronization.

With Intel® TSX, programmer-specified code regions (also referred to as transactional regions) are executed transactionally. If the transactional execution completes successfully, then all memory operations performed within the transactional region will appear to have occurred instantaneously when viewed from other logical processors. A processor makes architectural updates performed within the region visible to other logical processors only on a successful commit, a process referred to as an atomic commit.

Intel® TSX also provides an `XTEST` instruction, allowing software to query whether the logical processor is transactionally executing in a transactional region identified by either Hardware Lock Elision (HLE) or Restricted Transactional Memory (RTM).

Since a successful transactional execution ensures an atomic commit, the processor executes the code region optimistically without explicit synchronization. If synchronization was unnecessary for that specific execution, execution can commit without any cross-thread serialization. If the processor cannot commit atomically, the optimistic execution fails. When this happens, the processor will roll back the execution, a process referred to as a transactional abort. On a transactional abort, the processor will discard all updates performed in the region, restore architectural state to appear as if the optimistic execution never occurred, and resume execution non-transactionally.

A processor can perform a transactional abort for numerous reasons. A primary cause is due to conflicting accesses between the transactionally executing logical processor and another logical processor. Such conflicting accesses may prevent a successful transactional execution. Memory addresses read from within a transactional region constitute the read-set of the transactional region and addresses written to within the transactional region constitute the write-set of the transactional region. Intel® TSX maintains the read- and write-sets at the granularity of a cache line. A conflicting access occurs if another logical processor either reads a location that is part of the transactional region’s write-set or writes a location that is a part of either the read- or write-set of the transactional region.

Conflicting access typically means serialization is required for this code region. Intel® TSX detects data conflicts at the granularity of a cache line, so unrelated data locations placed in the same cache line will be detected as conflicts. Transactional aborts may also occur due to limited transactional resources. The amount of data accessed in the region may exceed an implementation-specific capacity. Some instructions and system events may also cause transactional aborts. Frequent transactional aborts cause wasted cycles.

Intel® TSX provide two software interfaces to specify regions of code for transactional execution.

Hardware Lock Elision (HLE)

HLE is a legacy-compatible instruction set extension (comprising the `XACQUIRE` and `XRELEASE` prefixes) to specify transactional regions. HLE is for programmers who prefer the backward compatibility of the conventional mutual-exclusion programming model and would like to run HLE-enabled software on legacy hardware, but would like to take advantage of new lock elision capabilities on hardware with HLE support.

NOTE
Hardware Lock Elision (HLE) intrinsic functions apply to C/C++ applications for Windows* only.

Restricted Transactional Memory (RTM)

RTM is a new instruction set interface (comprising the `XBEGIN`, `XEND`, and `XABORT` instructions) for programmers to define transactional regions in a more flexible manner than that possible with HLE.

RTM is for programmers who prefer a flexible interface to the transactional execution hardware.

Intel® Transactional Synchronization Extensions (Intel® TSX) Programming Considerations

Typical programmer-identified regions are expected to transactionally execute and commit successfully. However, Intel® Transactional Synchronization Extensions (Intel® TSX) does not provide any such guarantee. A transactional execution may abort for many reasons. To take full advantage of the transactional capabilities, programmers should take into account programming considerations to increase the probability of their transactional execution committing successfully.

This section discusses various events that may cause transactional aborts. The architecture ensures that updates performed within a transaction that subsequently aborts execution will not become visible: Only a committed transactional execution updates architectural state. Transactional aborts never cause functional failures and only affect performance.

Instruction Based Considerations

Programmers can use any instruction safely inside a transaction, Hardware Lock Elision (HLE) or Restricted Transactional Memory (RTM) and can use transactions at any privilege level. Some instructions will always abort the transactional execution and cause execution to seamlessly and safely transition to a non-transactional path.

Intel® TSX allows for most common instructions to be used inside transactions without causing aborts. The following operations inside a transaction do not typically cause an abort:

- Operations on the instruction pointer register.
- Operations on general purpose registers (`GPR`s).
- Operations on the status flags (CF, OF, SF, PF, AF, and ZF).
- Operations on `XMM` and `YMM` registers.
- Operations on the `MXCSR` register.

> **NOTE**
> Programmers must be careful when intermixing Intel® Supplemental Streaming Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) operations inside a transactional region. Intermixing Intel® SSE instructions accessing `XMM` registers and Intel® AVX instructions accessing `YMM` registers may cause transactions to abort.

Programmers may use `REP`/`REPNE` prefixed string operations inside transactions. However, long strings may cause aborts. Further, the use of `CLD` and `STD` instructions may cause aborts if they change the value of the DF flag. If DF is '1', the `STD` instruction will not cause an abort. Similarly, if DF is '0', the `CLD` instruction will not cause an abort.

Instructions not enumerated here as causing abort when used inside a transaction will typically not cause a transaction to abort (examples include but are not limited to `MFENCE`, `LFENCE`, `SFENCE`, `RDTSC`, `RDTSCP`, etc.).

The following instructions will abort transactional execution on any implementation:

- `XABORT`
- `CPUID`
- `PAUSE`

In some implementations, the following instructions may always cause transactional aborts. These instructions are not expected to be commonly used inside typical transactional regions. Programmers must not rely on these instructions to force a transactional abort, since whether they cause transactional aborts is implementation dependent.

- Operations on X87 and MMX™ architecture state. This includes all MMX™ and X87 instructions, including the `FXRSTOR` and `FXSAVE` instructions.
- **Update to non-status portion of EFLAGS:**`CLI`, `STI`, `POPFD`, `POPFQ`, `CLTS`.
- **Instructions that update segment registers, debug registers and/or control registers:**`MOV` to `DS`/`ES`/`FS`/`GS`/`SS`, `POPDS`/`ES`/`FS`/`GS`/`SS`, `LDS`, `LES`, `LFS`, `LGS`, `LSS`, `SWAPGS`, `WRFSBASE`, `WRGSBASE`, `LGDT`, `SGDT`, `LIDT`, `SIDT`, `LLDT`, `SLDT`, `LTR`, `STR`, `Far CALL`, `Far JMP`, `Far RET`, `IRET`, `MOV` to DRx, `MOV` to `CR0`/`CR2`/`CR3`/`CR4`/`CR8`, and `LMSW`.
- **Ring transitions:**`SYSENTER`, `SYSCALL`, `SYSEXIT`, and `SYSRET`.

- **TLB and Cacheability control:**`CLFLUSH`, `INVD`, `WBINVD`, `INVLPG`, `INVPCID`, and memory instructions with a non-temporal hint (`MOVNTDQA`, `MOVNTDQ`, `MOVNTI`, `MOVNTPD`, `MOVNTPS`, and `MOVNTQ`).
- **Processor state save:**`XSAVE`, `XSAVEOPT`, and `XRSTOR`.
- **Interrupts:**`INTn`, `INTO`.
- **IO:**`IN`, `INS`, `REP INS`, `OUT`, `OUTS`, `REP OUTS` and their variants.
- **VMX:** `VMPTRLD`, `VMPTRST`, `VMCLEAR`, `VMREAD`, `VMWRITE`, `VMCALL`, `VMLAUNCH`, `VMRESUME`, `VMXOFF`, `VMXON`, `INVEPT`, and `INVVPID`.
- **SMX:**`GETSEC`, `UD2`, `RSM`, `RDMSR`, `WRMSR`, `HLT`, `MONITOR`, `MWAIT`, `XSETBV`, `VZEROUPPER`, `MASKMOVQ`, and `V/MASKMOVDQU`.

Runtime Considerations

In addition to instruction-based considerations, runtime events may cause transactional execution to abort. These may be due to data access patterns or micro-architectural implementation causes. The following list is not a comprehensive discussion of all abort causes:

Any fault or trap in a transaction that must be exposed to software will be suppressed. Transactional execution will abort and execution will transition to a nontransactional execution as if the fault or trap had never occurred. If any exception is not masked, that will result in a transactional abort and it will be as if the exception had never occurred.

Synchronous exception events (#DE, #OF, #NP, #SS, #GP, #BR, #UD, #AC, #XF, #PF, #NM, #TS, #MF, #DB, #BP/INT3) that occur during transactional execution may cause an execution not to commit transactionally, and require a non-transactional execution. These events are suppressed as if they had never occurred. With HLE, since the non-transactional code path is identical to the transactional code path, these events will typically re-appear when the instruction that caused the exception is re-executed non-transactionally, causing the associated synchronous events to be delivered appropriately in the non-transactional execution.

Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to nontransactional execution. The asynchronous events will be queued and handled after the transactional abort is processed.

Transactions only support write-back cacheable memory type operations. A transaction may always abort if it includes operations on any other memory type. This includes instruction fetches to `UC` memory type.

Memory accesses within a transactional region may require the processor to set the Accessed and Dirty flags of the referenced page table entry. The behavior of how the processor handles this is implementation specific. Some implementations may allow the updates to these flags to become externally visible even if the transactional region subsequently aborts. Some Intel® TSX implementations may choose to abort the transactional execution if these flags need to be updated. Further, a processor's page-table walk may generate accesses to its own transactionally written but uncommitted state. Some Intel® TSX implementations may choose to abort the execution of a transactional region in such situations. The architecture ensures that if the transactional region aborts, the transactionally written state will not be made architecturally visible through the behavior of structures such as TLBs.

Executing self-modifying code transactionally may also cause transactional aborts. Programmers must continue to follow Intel recommended guidelines for writing self-modifying and cross-modifying code even when employing HLE and RTM.

While an implementation of RTM and HLE will typically provide sufficient resources for executing common transactional regions, implementation constraints and excessive sizes for transactional regions may cause a transactional execution to abort and transition to a non-transactional execution. The architecture provides no guarantee of the amount of resources available to do transactional execution and does not guarantee that a transactional execution will ever succeed.

Conflicting requests to a cache line accessed within a transactional region may prevent the transaction from executing successfully. For example, if logical processor *P0* reads line *A* in a transactional region and another logical processor *P1* writes *A* (either inside or outside a transactional region) then logical processor *P0* may abort if logical processor *P1*'s write interferes with processor *P0*'s ability to execute transactionally.

Similarly, if *P0* writes line *A* in a transactional region and *P1* reads or writes *A* (either inside or outside a transactional region), then *P0* may abort if *P1*'s access to *A* interferes with *P0*'s ability to execute transactionally. Other coherence traffic may at times appear as conflicting requests and may cause aborts. While these false conflicts may happen, they are expected to be uncommon. The conflict resolution policy to determine whether *P0* or *P1* aborts in the above scenarios implementation specific.

Intrinsics for Restricted Transactional Memory Operations

Restricted Transactional Memory Overview

Restricted Transactional Memory (RTM) provides a software interface for transactional execution. RTM provides three new instructions— `XBEGIN`, `XEND`, and `XABORT`—for programmers to start, commit, and abort transactional execution.

The programmer uses the `XBEGIN` instruction to specify the start of the transactional code region and the `XEND` instruction to specify the end of the transactional code region. The `XBEGIN` instruction takes an operand that provides a relative offset to the fallback instruction address if the RTM region could not be successfully executed transactionally.

A processor may abort RTM transactional execution for many reasons. The hardware automatically detects transactional abort conditions and restarts execution from the fallback instruction address with the architectural state corresponding to that at the start of the `XBEGIN` instruction and the `EAX` register updated to describe the abort status.

The `XABORT` instruction allows programmers to abort the execution of an RTM region explicitly. The `XABORT` instruction takes an 8-bit immediate argument that is loaded into the `EAX` register becoming available to software following an RTM abort.

RTM instructions do not have any data memory location associated with them. While the hardware provides no guarantees as to whether an RTM region will ever successfully commit transactionally, most transactions that follow the recommended guidelines are expected to successfully commit transactionally.

Programmers must always provide an alternative code sequence in the fallback path to guarantee the code completes execution. This may be as simple as acquiring a lock and executing the specified code region non-transactionally. Further, a transaction that always aborts on a given implementation may complete transactionally on a future implementation. Therefore, programmers must ensure the code paths for the transactional region and the alternative code sequence are functionally tested.

Detection of RTM Support

A processor supports RTM execution if CPUID.07H.EBX.RTM [bit 11] = 1. An application must check if the processor supports RTM before it uses the RTM instructions (`XBEGIN`, `XEND`, `XABORT`). These instructions will generate a #UD exception when used on a processor that does not support RTM.

RTM Abort Status Definition

RTM uses the `EAX` register to communicate abort status to software. Following an RTM abort the `EAX` register has the following definition.

EAX register bit position	**Meaning**
0	Set if abort caused by `XABORT` instruction.
1	If set, the transaction may succeed on a retry. This bit is always clear if bit '0' is set.

EAX register bit position	Meaning
2	Set if another logical processor conflicted with a memory address that was part of the transaction that aborted.
3	Set if an internal buffer overflowed.
4	Set if a debug breakpoint was hit.
5	Set if an abort occurred during execution of a nested transaction.
23:6	Reserved.
31:24	`XABORT` argument (only valid if bit '0' set, otherwise reserved).

The `EAX` abort status for RTM only provides the cause for aborts. It does not by itself encode whether an abort or commit occurred for the RTM region. The value of `EAX` can be '0' following an RTM abort. For example, a CPUID instruction when used inside an RTM region causes a transactional abort and may not satisfy the requirements for setting any of the `EAX` bits. This may result in an `EAX` value of '0'.

RTM Memory Ordering

A successful RTM commit causes all memory operations in the RTM region to appear to execute atomically. A successfully committed RTM region consisting of an `XBEGIN` followed by an `XEND`, even with no memory operations in the RTM region, has the same ordering semantics as a `LOCK` prefixed instruction. The `XBEGIN` instruction does not have fencing semantics. However, if an RTM execution aborts, all memory updates from within the RTM region are discarded and never made visible to any other logical processor.

See Also

Intel® Transactional Synchronization Extensions Programming Considerations

_xtest

Queries whether the processor is executing in a transactional region identified by restricted transactional memory (RTM) or hardware lock elision (HLE). The corresponding Intel® AVX2 instruction is `XTEST`*.*

Syntax

```
unsigned char _xtest(void);
```

Arguments

None.

Description

Queries the RTM or HLE execution status. If the `xtest` function is called inside an RTM or an active HLE region, then the `ZF` flag is cleared, else it is set.

NOTE

A processor supports the `xtest` instruction if it supports either HLE or RTM. An application must check either of these feature flags before using the `xtest` instruction. While the HLE prefixes are ignored on processors that do not support HLE, this instruction will generate a #UD exception when used on a processor that does not support either HLE or RTM.

Returns

Result of the query.

_xbegin

Specifies the start of a restricted transactional memory (RTM) code region and returns a value indicating status. The corresponding Intel® AVX2 instruction is `XBEGIN`*.*

Syntax

`unsigned int _xbegin(void);`

Arguments

None.

Description

Starts a RTM code region and returns a value indicating transaction successfully started or status from a transaction abort.

If the logical processor was not already in transactional execution, then the `xbegin` instruction causes the logical processor to start transactional execution. The `xbegin` instruction that transitions the logical processor into transactional execution is referred to as the outermost `xbegin` instruction.

The `xbegin` instruction specifies a relative offset to the fallback code path executed following a transactional abort. To promote proper program structure, this is not exposed in C++ code and the intrinsic function operates as if it invoked the following model code:

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
    move   eax, 0xFFFFFFFF
    xbegin _txnL1
    _txnL1:
    move   status, eax
  }
  return status;
}
```

When a transaction is successfully created the function will return 0xFFFFFFFF, which is never a valid status code for an aborted transaction. If the transaction aborts for any reason, the logical processor discards all architectural register and memory updates performed during the transaction execution and restores the architectural state to that corresponding to the outermost `xbegin` instruction. The EAX register is then updated with the status code of the aborted transaction, which can be used to transfer control to a fallback handler.

The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort. On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost `xbegin` instruction. The abort destination operand of the `xbegin` instruction is targeted to the following instruction so that there is no change in control flow whether the transaction aborts or not.

Returns

Returns value indicating transaction successfully started or status from a transaction abort.

_xend

Specifies the end of a restricted transactional memory (RTM) code region. The corresponding Intel® AVX2 instruction is `XEND`*.*

Syntax

`void _xend(void);`

Arguments

None.

Description

Specifies the end of restricted transactional memory code region. If this is the outermost transaction (including this `xend` instruction, the number of `xbegin` matches the number of `xend` instructions) then the processor will attempt to commit processor state automatically.

If the commit fails, the processor will rollback all register and memory updates performed during the RTM execution.

The logical processor will resume execution at the fallback address computed from the outermost `xbegin` instruction. The EAX register is updated to reflect RTM abort information. When `xend` is executed outside a transaction will cause a general protection fault (#GP).

The model instruction sequence for `xend` support is:

```
__inline void _xend() {
  __asm { xend }
}
```

Returns

Result of the query.

_xabort

Forces a restricted transactional memory (RTM) region to abort. The corresponding Intel® AVX2 instruction is `XABORT`*.*

Syntax

`void _xabort(const unsigned int imm);`

Arguments

None.

Description

Forces a RTM region to abort. All outstanding transactions are aborted and the logical processor resumes execution at the fallback address computed through the outermost `xbegin` transaction.

The EAX register is updated to reflect an `xabort` instruction caused the abort, and the imm8 argument will be provided in the upper eight bits of the return value (EAX register bits 31:24) containing the indicated immediate value. The argument of `xabort` function must be a compile time constant.

The model instruction sequence for `xabort` support is:

```
__inline void _xabort() {     __asm { xabort } }
```

Returns

Result of the query.

Intrinsics for Hardware Lock Elision Operations

Hardware Lock Elision Overview

> **NOTE**
> Hardware Lock Elision (HLE) intrinsic functions apply to C/C++ applications for Windows* only.

Hardware Lock Elision (HLE) provides a legacy compatible instruction set interface for transactional execution. HLE provides two new instruction prefix hints: `XACQUIRE` and `XRELEASE`.

The programmer uses the `XACQUIRE` prefix in front of the instruction that is used to acquire the lock that is protecting the critical section. The processor treats the indication as a hint to elide the write associated with the lock acquire operation. Even though the lock acquire has an associated write operation to the lock, the processor does not add the address of the lock to the transactional region's write-set nor does it issue any write requests to the lock. Instead, the address of the lock is added to the read-set and the logical processor enters transactional execution. If the lock was available before the `XACQUIRE` prefixed instruction, all other processors will continue to see it as available afterwards. Since the transactionally executing logical processor neither added the address of the lock to its write-set, nor performed externally visible write operations to it, other logical processors can read the lock without causing a data conflict. This allows other logical processors to enter and concurrently execute the lock-protected section. The processor automatically detects data conflicts that occur during the transactional execution and will perform a transactional abort if necessary.

The hardware ensures program order of operations on the lock, even though the eliding processor did not perform external write operations to the lock. If the eliding processor itself reads the value of the lock in the critical section, it will appear as if the processor had acquired the lock (the read will return the non-elided value). This behavior makes an HLE execution functionally equivalent to an execution without the HLE prefixes.

The programmer uses the `XRELEASE` prefix in front of the instruction that is used to release the lock protecting the critical section. This involves a write to the lock. If the instruction is restoring the value of the lock to the value it had prior to the `XACQUIRE` prefixed lock-acquire operation on the same lock, the processor elides the external write request associated with the release of the lock and does not add the address of the lock to the write-set. The processor then attempts to commit the transactional execution.

If multiple threads execute critical sections protected by the same lock, but they do not perform conflicting data operations, the threads can execute concurrently and without serialization. Even though the software uses lock acquisition operations on a common lock, the hardware recognizes this, elides the lock, and executes the critical sections on the two threads without requiring any communication through the lock — if such communication was dynamically unnecessary.

If the processor is unable to execute the region transactionally, it will execute the region non-transactionally and without elision. HLE-enabled software has the same forward progress guarantees as the underlying non-HLE lock-based execution. For successful HLE execution, the lock and the critical section code must follow certain guidelines. These guidelines only affect performance; not following these guidelines will not cause functional failure.

Hardware without HLE support will ignore the `XACQUIRE` and `XRELEASE` prefix hints and will not perform any elision. These prefixes correspond to the `REPNE`/`REPE` IA-32 architecture prefixes ignored on the instructions where `XACQUIRE` and `XRELEASE` are valid. Importantly, HLE is compatible with the existing lock-based programming model. Improper use of hints will not cause functional bugs though it may expose latent bugs already in the code.

See Also

Intel® Transactional Synchronization Extensions Programming Considerations

HLE Acquire _InterlockedCompareExchange Functions

Performs an atomic compare-and-exchange operation on the specified values and attempts to begin a HLE transaction if supported by the executing platform. This intrinsic function applies to C/C++ applications for Windows only.

Syntax

```
long _InterlockedCompareExchange_HLEAcquire(long volatile *Destination, long Exchange,
long Comparand);
```

```
__int64 _InterlockedCompareExchange64_HLEAcquire(__int64 volatile *Destination, __int64
Exchange, __int64 Comparand);
```

```
void * _InterlockedCompareExchangePointer_HLEAcquire(void * volatile *Destination, void
* Exchange, void * Comparand);
```

Parameters

`Destination [in, out]`	pointer to the destination value
`Exchange [in]`	value which will be written at Destination if the comparison matches.
`Comparand [in]`	value to compare to the value referenced by Destination.

Description

Performs an atomic compare-and-exchange operation on the specified values, and also attempts to begin a HLE transaction if the executing platform supports it.

These functions compare two specified values and replaces one of them with a third value if the compared values are equal.

Returns

Returns the initial value referenced by the `Destination` parameter.

HLE Acquire _InterlockedExchangeAdd Functions

Performs an atomic addition of two values and attempts to begin a HLE transaction if supported by the executing platform. This intrinsic function applies to C/C++ applications for Windows only.

Syntax

```
long _InterlockedExchangeAdd_HLEAcquire(long volatile *Addend, long Value);
```

```
__int64 _InterlockedExchangeAdd64_HLEAcquire(__int64 volatile *Addend, __int64 Value);
```

Parameters

`Addend [in, out]`	pointer to the addend which will be replaced with the result of the addition
`Value [in]`	value to be added to the value referenced by the `Addend`parameter

Description

Performs an atomic addition of two values, and also attempts to begin a HLE transaction if the executing platform supports it.

Returns

The function returns the initial value referenced by the `Addend` parameter.

HLE Release _InterlockedCompareExchange Functions

Performs an atomic compare-and-exchange operation on the specified values and releases pending active HLE transaction. This intrinsic function applies to C/C++ applications for Windows only.

Syntax

```
long _InterlockedCompareExchange_HLERelease(long volatile *Destination, long Exchange, long Comparand);
```

```
__int64 _InterlockedCompareExchange64_HLERelease(__int64 volatile *Destination, __int64 Exchange, __int64 Comparand);
```

```
void * _InterlockedCompareExchangePointer_HLERelease(void * volatile *Destination, void * Exchange, void * Comparand);
```

Parameters

`Destination [in, out]`	pointer to the destination value
`Exchange [in]`	value which will be written at Destination if the comparison matches.
`Comparand [in]`	value to compare to the value referenced by Destination.

Description

Performs an atomic compare-and-exchange operation on the specified values and releases a pending HLE transaction (if one is active).

The function compares two specified values and replaces one of them with a third value if the compared values are equal.

Returns

Returns the initial value referenced by the `Destination` parameter.

HLE Release _InterlockedExchangeAdd Functions

Performs an atomic addition of two values and releases pending active HLE transaction. This intrinsic function applies to C/C++ applications for Windows only.

Syntax

```
long _InterlockedExchangeAdd_HLERelease(long volatile *Addend, long Value);
```

```
__int64 _InterlockedExchangeAdd64_HLERelease(__int64 volatile *Addend, __int64 Value);
```

Parameters

`Addend [in, out]`	pointer to the addend which will be replaced with the result of the addition
`Value [in]`	value to be added to the value referenced by the `Addend` parameter

Description

Performs an atomic addition of two values and releases a pending HLE transaction (if one is active).

Returns

Returns the initial value referenced by the `Addend` parameter.

HLE Release _Store Functions

Stores the specified value at the specified address and releases pending active HLE transaction. This intrinsic function applies to C/C++ applications for Windows only.

Syntax

`void _Store_HLERelease(long volatile *Destination, long Value);`

`void _Store64_HLERelease(__int64 volatile *Destination, __int64 Value);`

`void _StorePointer_HLERelease(void * volatile *Destination, void * Value);`

Parameters

`Destination[out]`	pointer to the destination value
`Value [in]`	value to store at `Destination`.

Description

Stores the specified value at the specified address and releases a pending HLE transaction if one is active.

Returns

Nothing.

Function Prototype and Macro Definitions

To use the prototypes and macro definitions shown in Group 1, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Group 1: Function Prototypes:

```
unsigned int _xbegin(void);
void _xend(void);
void _xabort(const unsigned int imm);
unsigned char _xtest(void);
```

The following macro definitions are included in the `immintrin.h` header file:

Group 1: Macro Definitions:

```
#define _XBEGIN_STARTED (~0u)
#define _XABORT_EXPLICIT (1 << 0)
#define _XABORT_RETRY (1 << 1)
#define _XABORT_CONFLICT (1 << 2)
#define _XABORT_CAPACITY (1 << 3)
#define _XABORT_DEBUG (1 << 4)
#define _XABORT_NESTED (1 << 5)
```

Group 2: Function Macros

The following function Macros are *not* included in `immintrin.h` header file. If you want to use them, you need to define them in your applications.

For the HW with RTM support

```
#define __try_transaction(x) if ((x =_xbegin()) == _XBEGIN_STARTED)
#define __try_else _xend() } else
#define __transaction_abort(c) _xabort(c)
```

For the HW with no RTM support

```
#define __try_transaction(x) if (0) {
#define __try_else } else
#define __transaction_abort(c)
```

x is an unsigned integer type local variable for programmers to access RTM transaction abort code and holds the return value of `_xbegin()`. *c* is an unsigned integer compile-time constant value that is returned in the upper bits of *x* when `_xabort(c)` is executed.

A usage sample code of macros

```
 foo() { // user macros
int status;
__try_transaction (status) {
''' ''' '''
transaction code ….
 }
__try_else {
if (status & _XABORT_CONFLICT) {
 … code
}
}
}
```

Pseudo-ASM code

```
foo() { or eax 0xffffffff
xbegin L1 L1: mov status, eax
cmp eax 0xffffffff jnz
L2 transaction code

// when abort happens, HW restarts from L1
xend jmp L3 L2: abort
handler code L3: ret
}
```

The compiler will convert the macros to the instruction sequence with a proper branching for speculative execution path and alternative execution path.

The above example is similar to the usage example, except `__try_transaction` and `__try_else` macros are used instead of RTM intrinsic functions.

Intrinsics for Intel® Advanced Vector Extensions

Intel® Advanced Vector Extensions (Intel® AVX) intrinsics are assembly-coded functions that call on Intel® AVX instructions, which are new vector SIMD instruction extensions for IA-32 and Intel® 64 architectures. Intel® AVX intrinsics are architecturally similar to Intel® Streaming SIMD Extensions (Intel® SSE) and double-precision floating-point portions of Intel® Streaming SIMD Extensions 2 (Intel® SSE2).

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intel® AVX intrinsics introduce 256-bit vector processing capability, and are supported on IA-32 and Intel® 64 architectures built from 32nm process technology and beyond. They map directly to Intel® AVX new instructions and other enhanced 128-bit SIMD instructions.

The first generation Intel® AVX provides 256-bit SIMD register support, 256-bit vector floating-point instructions, enhancements to 128-bit SIMD instructions, and support for three and four operand syntax.

Functional Overview

Intel® AVX provides comprehensive functional improvements over previous generations of SIMD instruction extensions. The functional improvements include:

- **256-bit floating-point arithmetic primitives:** Intel® AVX enhances existing 128-bit floating-point arithmetic instructions with 256-bit capabilities for floating-point processing.
- **Enhancements for flexible SIMD data movements:** Intel® AVX provides a number of new data movement primitives to enable efficient SIMD programming in relation to loading non-unit-strided data into SIMD registers, intra-register SIMD data manipulation, conditional expression and branch handling, etc. Enhancements for SIMD data movement primitives cover 256-bit and 128-bit vector floating-point data, and across 128-bit integer SIMD data processing using VEX-encoded instructions.

See Also

Instruction Set Architecture (ISA) site at https://software.intel.com/en-us/isa-extensions
Details about the Intel® AVX Intrinsics

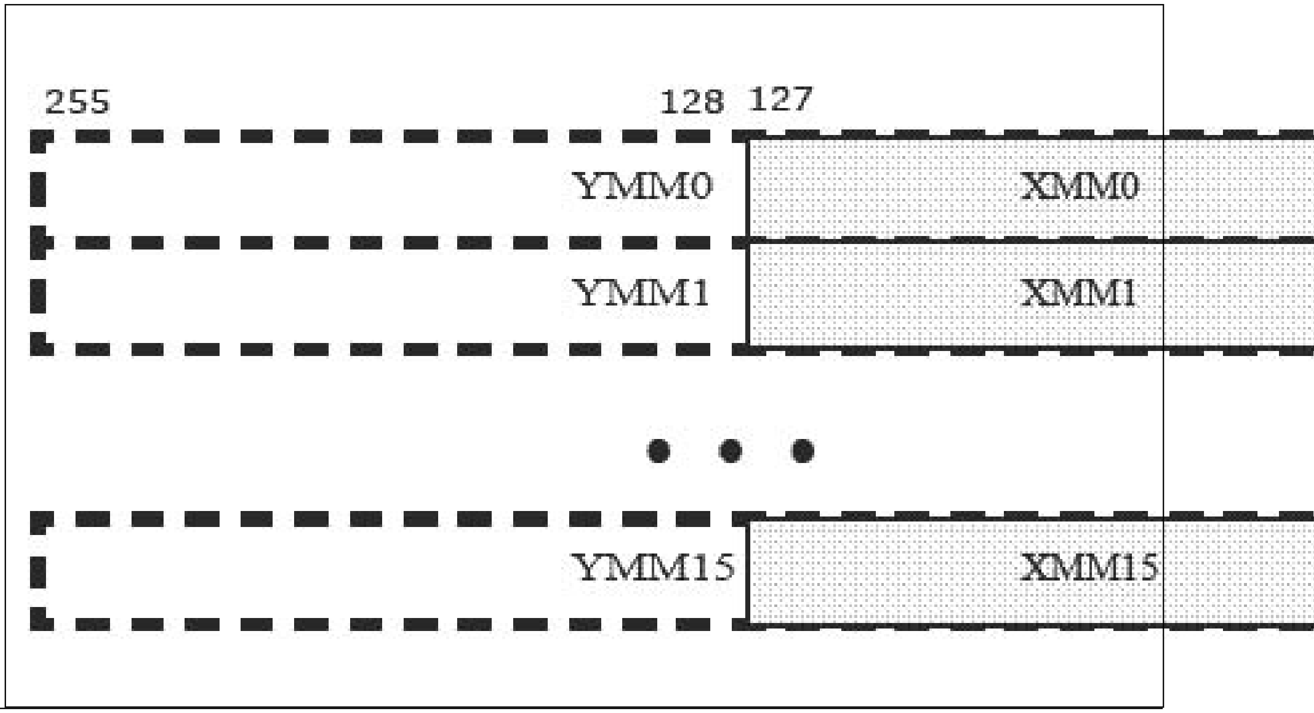
Details of Intel® Advanced Vector Extensions Intrinsics

Intel® Advanced Vector Extensions (Intel® AVX) intrinsics map directly to Intel® AVX instructions and other enhanced 128-bit single-instruction multiple data processing (SIMD) instructions. Intel® AVX instructions are architecturally similar to extensions of the existing Intel® 64 architecture-based vector streaming SIMD portions of Intel® Streaming SIMD Extensions (Intel® SSE) instructions, and double-precision floating-point portions of Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instructions. However, Intel® AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- Instruction syntax support three and four operand syntax, to improve instruction programming flexibility and efficiency for new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- Instruction encoding format using a new prefix (referred to as VEX) to provide compact, efficient encoding for three-operand syntax, vector lengths, compaction of legacy SIMD prefixes and REX functionality.
- Intel® AVX data types allow packing of up to 32 elements in a register if bytes are used. The number of elements depends upon the element type: eight single-precision floating point types or four double-precision floating point types.

Intel® Advanced Vector Extensions Registers

Intel® AVX adds 16 registers (`YMM0-YMM15`), each 256 bits wide, aliased onto the 16 SIMD (`XMM0-XMM15`) registers. The Intel® AVX new instructions operate on the `YMM` registers. Intel® AVX extends certain existing instructions to operate on the `YMM` registers, defining a new way of encoding up to three sources and one destination in a single instruction.



Because each of these registers can hold more than one data element, the processor can process more than one data element simultaneously. This processing capability is also known as single-instruction multiple data processing (SIMD).

For each computational and data manipulation instruction in the new extension sets, there is a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

Intel® Advanced Vector Extensions Types

The Intel® AVX intrinsic functions use three new C data types as operands, representing the new registers used as operands to the intrinsic functions. These are `__m256`, `__m256d`, and `__m256i` data types.

The `__m256` data type is used to represent the contents of the extended SSE register, the `YMM` register, used by the Intel® AVX intrinsics. The `__m256` data type can hold eight 32-bit floating-point values.

The `__m256d` data type can hold four 64-bit double precision floating-point values.

The `__m256i` data type can hold thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit integer values.

The compiler aligns the `__m256`, `__m256d`, and `__m256i` local and global data to 32-byte boundaries on the stack. To align `integer`, `float`, or `double` arrays, use the `__declspec(align)` statement.

The Intel® AVX intrinsics also use Intel® SSE2 data types like `__m128`, `__m128d`, and `__m128i` for some operations. See Details of Intrinsics topic for more information.

VEX Prefix Instruction Encoding Support for Intel® AVX

Intel® AVX introduces a new prefix, referred to as VEX, in the Intel® 64 and IA-32 instruction encoding format. Instruction encoding using the VEX prefix provides several capabilities:

- direct encoding of a register operand within the VEX prefix.
- efficient encoding of instruction syntax operating on 128-bit and 256-bit register sets.
- compaction of REX prefix functionality.
- compaction of SIMD prefix functionality and escape byte encoding.

- providing relaxed memory alignment requirements for most VEX-encoded SIMD numeric and data processing instruction semantics with memory operand as compared to instructions encoded using SIMD prefixes.

The VEX prefix encoding applies to SIMD instructions operating on YMM registers, XMM registers, and in some cases with a general-purpose register as one of the operands. The VEX prefix is not supported for instructions operating on MMX™ or x87 registers.

It is recommended to use Intel® AVX intrinsics with option `[Q]xAVX`, because their corresponding instructions are encoded with the VEX-prefix. The `[Q]xAVX` option forces other packed instructions to be encoded with VEX too. As a result there are fewer performance stalls due to Intel® AVX to legacy Intel® SSE code transitions.

Naming and Usage Syntax

Most Intel® AVX intrinsic names use the following notational convention:

```
_mm256_<intrin_op>_<suffix>(<data type> <parameter1>, <data type> <parameter2>, <data type>
<parameter3>)
```

The following table explains each item in the syntax.

`_mm256/` `_mm128`	Prefix representing the size of the result. Usually, this corresponds to the Intel® AVX vector register size of 256 bits, but certain comparison and conversion intrinsics yield a 128-bit result.
`<intrin_op>`	Indicates the basic operation of the intrinsic; for example, add for addition and sub for subtraction.
`<suffix>`	Denotes the type of data the instruction operates on. The first one or two letters of each suffix denote whether the data is packed (*p*), extended packed (*ep*), or scalar (*s*). The remaining letters and numbers denote the type, with notation as follows: • **s:** single-precision floating point • **d:** double-precision floating point • **i128:** signed 128-bit integer • **i64:** signed 64-bit integer • **u64:** unsigned 64-bit integer • **i32:** signed 32-bit integer • **u32:** unsigned 32-bit integer • **i16:** signed 16-bit integer • **u16:** unsigned 16-bit integer • **i8:** signed 8-bit integer • **u8:** unsigned 8-bit integer • **ps:** packed single-precision floating point • **pd:** packed double-precision floating point • **sd:** scalar double-precision floating point • **epi32:** extended packed 32-bit integer • **si256:** scalar 256-bit integer
`<data type>`	Parameter data types: `__m256`, `__m256d`, `__m256i`, `__m128`, `__m128d`, `__m128i`, `const`, `int`, etc.
`<parameter1>`	Represents a source vector register: `m1`/`s1`/`a`
`<parameter2>`	Represents another source vector register: `m2`/`s2`/`b`
`<parameter3>`	Represents an integer value: `mask`/`select`/`offset`

The third parameter is an integer value whose bits represent a conditionality based on which the intrinsic performs an operation.

Example Usage

```
extern __m256d _mm256_add_pd(__m256d m1, __m256d m2);
```

where,

add	indicates that an addition operation must be performed
pd	indicates packed double-precision floating-point value

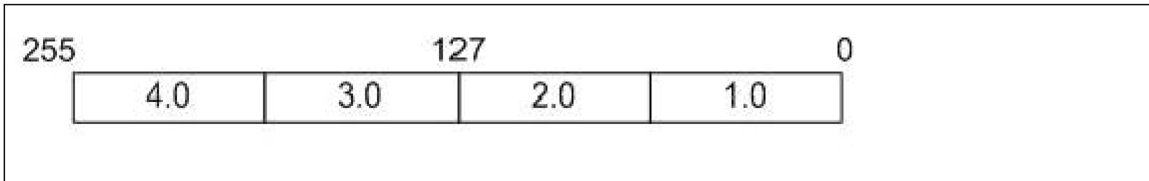
The packed values are represented in right-to-left order, with the lowest value used for scalar operations. Consider the following example operation:

```
double a[4] = {1.0, 2.0, 3.0, 4.0};
__m256d t = _mm256_load_pd(a);
```

The result is the following:

```
__m256d t = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
```

In other words, the `YMM` register that holds the value `t` appears as follows:



The " scalar " element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

See Also

`__declspec(align)` declaration

Details of Intrinsics (general)

Intrinsics for Arithmetic Operations

_mm256_add_pd

Adds float64 vectors. The corresponding Intel® AVX instruction is `VADDPD`.

Syntax

```
extern __m256d _mm256_add_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD addition of four packed double-precision floating-point elements (float64 elements) in the first source vector *m1* with four float64 elements in the second source vector *m2*.

Returns

Result of the addition operation.

_mm256_add_ps

Adds float32 vectors. The corresponding Intel® AVX instruction is `VADDPS`*.*

Syntax

```
extern __m256 _mm256_add_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD addition of eight packed single-precision floating-point elements (float32 elements) in the first source vector *m1* with eight float32 elements in the second source vector *m2*.

Returns

Result of the addition operation.

_mm256_addsub_pd

Adds odd float64 elements and subtracts even float64 elements of vectors. The corresponding Intel® AVX instruction is `VADDSUBPD`*.*

Syntax

```
extern __m256d _mm256_addsub_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD addition of the odd packed double-precision floating-point elements (float64 elements) from the first source vector *m1* to the odd float64 elements of the second source vector *m2*.

Simultaneously, the intrinsic performs subtraction of the even double-precision floating-point elements of the second source vector *m2* from the even float64 elements of the first source vector *m1*.

Returns

Result of the operation is stored in the result vector, which is returned by the intrinsic.

_mm256_addsub_ps

Adds odd float32 elements and subtracts even float32 elements of vectors. The corresponding Intel® AVX instruction is `VADDSUBPS`*.*

Syntax

```
extern __m256 _mm256_addsub_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD addition of the odd single-precision floating-point elements (float32 elements) in the first source vector *m1* with the odd float32 elements in the second source vector *m2*.

Simultaneously, the intrinsic performs subtraction of the even single-precision floating-point elements (float32 elements) in the second source vector, *m2*, from the even float32 elements in the first source vector, *m1*.

Returns

Result of the operation stored in the result vector.

_mm256_hadd_pd

Adds horizontal pairs of float64 elements of two vectors. The corresponding Intel® AVX instruction is `VHADDPD`*.*

Syntax

```
extern __m256d _mm256_hadd_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD addition of adjacent (horizontal) pairs of double-precision floating-point elements (float64 elements) ins the first source vector *m1* with adjacent pairs of float64 elements in the second source vector *m2*.

Returns

Result of the addition operation.

_mm256_hadd_ps

Adds horizontal pairs of float32 elements of two vectors. The corresponding Intel® AVX instruction is `VHADDPS`*.*

Syntax

```
extern __m256 _mm256_hadd_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD addition of adjacent (horizontal) pairs of single-precision floating-point elements (float32 elements) in the first source vector *m1* with adjacent pairs of float32 elements in the second source vector *m2*.

Returns

Returns the result of the addition operation.

_mm256_sub_pd

Subtracts float64 vectors. The corresponding Intel® AVX instruction is `VSUBPD`*.*

Syntax

```
extern __m256d _mm256_sub_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD subtraction of four packed double-precision floating-point elements (float64 elements) of the second source vector *m2* from the first source vector *m1*.

Returns

Returns the result of the subtraction operation.

_mm256_sub_ps

Subtracts float32 vectors. The corresponding Intel® AVX instruction is `VSUBPS`*.*

Syntax

```
extern __m256 _mm256_sub_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD subtraction of eight packed single-precision floating-point elements (float32 elements) of the second source vector *m2* from the first source vector *m1*.

Returns

Returns the result of the subtraction operation.

_mm256_hsub_pd

Subtracts horizontal pairs of float64 elements of two vectors. The corresponding Intel® AVX instruction is `VHSUBPD`.

Syntax

```
extern __m256d _mm256_hsub_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD subtraction of adjacent (horizontal) pairs of double-precision floating-point elements (float64 elements) in the second source vector *m2* from adjacent pairs of float64 elements in the first source vector *m1*.

Returns

Result of the subtraction operation.

_mm256_hsub_ps

Subtracts horizontal pairs of float32 elements of two vectors. The corresponding Intel® AVX instruction is `VHSUBPS`.

Syntax

```
extern __m256 _mm256_hsub_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD subtraction of adjacent (horizontal) pairs of single-precision floating-point elements (float32 elements) in the second source vector *m2* from adjacent pairs of float32 elements in the first source vector *m1*.

Returns

Returns the result of the subtraction operation.

_mm256_mul_pd

Multiplies float64 vectors. The corresponding Intel® AVX instruction is `VMULPD`.

Syntax

```
extern __m256d _mm256_mul_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD multiplication of four packed double-precision floating-point elements (float64 elements) in the first source vector, *m1* , with four float64 elements in the second source vector, *m2*.

Returns

Result of the multiplication operation.

_mm256_mul_ps

Multiplies float32 vectors. The corresponding Intel® AVX instruction is `VMULPS`.

Syntax

```
extern __m256 _mm256_mul_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD multiplication of eight packed single-precision floating-point elements (float32 elements) in the first source vector *m1* with eight float32 elements in the second source vector *m2*.

Returns

Result of the multiplication operation.

_mm256_div_pd

Divides float64 vectors. The corresponding Intel® AVX instruction is `VDIVPD`.

Syntax

```
extern __m256d _mm256_div_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation

m2	float64 vector also used for the operation

Description

Performs a SIMD division of four packed double-precision floating-point elements (float64 elements) in the first source vector *m1* with four float64 elements in the second source vector *m2*.

Returns

Result of the division operation.

_mm256_div_ps

Divides float32 vectors. The corresponding Intel® AVX instruction is `VDIVPS`*.*

Syntax

```
extern __m256 _mm256_div_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD division of eight packed single-precision floating-point elements (float32 elements) in the first source vector *m1* with eight float32 elements in the second source vector *m2*.

Returns

Result of the division operation.

_mm256_dp_ps

Calculates the dot product of float32 vectors. The corresponding Intel® AVX instruction is `VDPPS`*.*

Syntax

```
extern __m256 _mm256_dp_ps(__m256 m1, __m256 m2, const int mask);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation
mask	a constant of integer type where the high four bits of the mask determine how the resultant elements are summed and the low four bits determine whether the summed resultant value is to be broadcast to the destination vector or not

Description

First performs a SIMD multiplication of the lower four packed single-precision floating-point elements (float32 elements) from the first source vector *m1* with corresponding elements in the second source vector *m2*.

Each of the four resulting single-precision elements is conditionally summed depending on the high four bits in the *mask* parameter.

The resulting summed value is broadcast to each of the lower 4 positions in the destination vector, if the corresponding lower bit of the *mask* is "1". If the corresponding lower bit of the *mask* is zero, the corresponding lower element in the destination vector is set to zero.

The process is then replicated with the high elements of the source vectors.

Returns

Result of the operation.

_mm256_sqrt_pd

Computes the square root of double-precision floating point values. The corresponding Intel® AVX instruction is `VSQRTPD`*.*

Syntax

```
extern __m256d _mm256_sqrt_pd(__m256d a);
```

Arguments

a	float64 source vector

Description

Performs a SIMD computation of the square roots of the two or four packed double-precision floating-point values (float64 values) in the source vector and returns the result of the square root operation.

Returns

Result of the square root operation.

_mm256_sqrt_ps

Computes the square root of single-precision floating point values. The corresponding Intel® AVX instruction is `VSQRTPS`*.*

Syntax

```
extern __m256 _mm256_sqrt_ps(__m256 a);
```

Arguments

a	float32 source vector

Description

Performs a SIMD computation of the square roots of the eight packed single-precision floating-point values (float32 values) in the source vector and returns the result of the square root operation.

Returns

Result of the square root operation.

_mm256_rsqrt_ps

Computes approximate reciprocals of square roots of float32 values.The corresponding Intel® AVX instruction is `VRSQRTPS`*.*

Syntax

```
extern __m256 _mm256_rsqrt_ps(__m256 a);
```

Arguments

a	float32 source vector

Description

Performs a computation of the reciprocal of the square roots of eight single-precision floating point elements of the source vector *a* and returns the result as a vector.

Returns

Result of the reciprocal square root operation.

_mm256_rcp_ps

Computes approximate reciprocals of float32 values. The corresponding Intel® AVX instruction is `VRCPPS`*.*

Syntax

```
extern __m256 _mm256_rcp_ps(__m256 a);
```

Arguments

a	float32 source vector

Description

Performs a computation of the reciprocal of eight single-precision floating point elements of the source vector *a* and returns the result as a vector.

Returns

Result of the reciprocal operation.

Intrinsics for Bitwise Operations

_mm256_and_pd

Performs bitwise logical AND operation on float64 vectors. The corresponding Intel® AVX instruction is `VANDPD`*.*

Syntax

```
extern __m256d _mm256_and_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a bitwise logical AND of the four packed double-precision floating-point elements (float64 elements) of the first source vector *m1*, and corresponding elements in the second source vector *m2*.

Returns

Result of the bitwise operation.

_mm256_and_ps

Performs bitwise logical AND operation on float32 vectors. The corresponding Intel® AVX instruction is `VANDPS`.

Syntax

```
extern __m256 _mm256_and_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a bitwise logical AND of the eight packed single-precision floating-point elements (float32 elements) of the first source vector *m1*, and corresponding elements in the second source vector *m2.*

Returns

Result of the bitwise operation.

_mm256_andnot_pd

Performs bitwise logical AND NOT operation on float64 vectors. The corresponding Intel® AVX instruction is `VANDNPD`.

Syntax

```
extern __m256d _mm256_andnot_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a bitwise logical AND NOT of the four packed double-precision floating-point elements (float64 elements) of the first source vector *m1*, and corresponding elements in the second source vector *m2*.

Returns

Result of the bitwise operation.

_mm256_andnot_ps

Performs bitwise logical AND NOT operation on float32 vectors.The corresponding Intel© AVX instruction is `VANDNPS`.

Syntax

```
extern __m256 _mm256_andnot_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a bitwise logical AND NOT of the eight packed single-precision floating-point elements (float32 elements) of the first source vector *m1*, and corresponding elements in the second source vector *m2*.

Returns

Result of the bitwise operation.

_mm256_or_pd

Performs bitwise logical OR operation on float64 vectors. The corresponding Intel© AVX instruction is `VORPD`.

Syntax

```
extern __m256d _mm256_or_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a bitwise logical OR of the four packed double-precision floating-point elements (float64 elements) of the first source vector *m1*, and corresponding elements in the second source vector *m2*.

Returns

Result of the bitwise operation.

_mm256_or_ps

Performs bitwise logical OR operation on float32 vectors. The corresponding Intel© AVX instruction is `VORPS`.

Syntax

```
extern __m256 _mm256_or_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a bitwise logical OR of the eight packed single-precision floating-point elements (float32 elements) of the first source vector *m1*, and corresponding elements in the second source vector *m2*.

Returns

Result of the bitwise operation.

_mm256_xor_pd

Performs bitwise logical XOR operation on float64 vectors. The corresponding Intel® AVX instruction is `VXORPD`*.*

Syntax

```
extern __m256d _mm256_xor_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a bitwise logical XOR of the four packed double-precision floating-point elements (float64 elements) of the first source vector *m1*, and corresponding elements in the second source vector *m2*.

Returns

Result of the bitwise operation.

_mm256_xor_ps

Performs bitwise logical XOR operation on float32 vectors. The corresponding Intel® AVX instruction is `VXORPS`*.*

Syntax

```
extern __m256 _mm256_xor_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a bitwise logical XOR of the eight packed single-precision floating-point elements (float32 elements) of the first source vector *m1*, and corresponding elements in the second source vector, *m2*.

Returns

Result of the bitwise operation.

Intrinsics for Blend and Conditional Merge Operations

_mm256_blend_pd

Performs a conditional blend/merge of float64 vectors. The corresponding Intel© AVX instruction is `VBLENDPD`*.*

Syntax

```
extern __m256d _mm256_blend_pd(__m256d m1, __m256d m2, const int mask);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation
mask	a constant of integer type that is the mask for the operation

Description

Performs a conditional merge of four packed double-precision floating point elements (float64 elements) of two vectors according to the immediate bits of the *mask* parameter.

The *mask* parameter defines a constant integer. The immediate bits [3:0] in the *mask* determine from which source vector elements are copied into the resulting vector.

If the bits in *mask* are "1" then the corresponding elements of the second source vector are copied into the resulting vector. If the bits are "0" then the corresponding elements of the first source vector are copied into the resulting vector. Thus a merging/blending of the elements of the two source vectors occurs when this intrinsic is used.

Returns

Result of the merge/blend operation.

_mm256_blend_ps

Performs a conditional blend/merge of float32 vectors. The corresponding Intel© AVX instruction is `VBLENDPS`*.*

Syntax

```
extern __m256 _mm256_blend_ps(__m256 m1, __m256 m2, const int mask);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

mask	a constant of integer type that is the mask for the operation

Description

Performs a conditional merge of eight packed single-precision floating point elements (float32 elements) of two vectors according to the immediate bits of the *mask* parameter.

The *mask* parameter defines a constant integer. The immediate bits [7:0] in the mask determine from which source vector elements are copied into the resulting vector.

If the bits in *mask* are "1" then the corresponding elements of the second source vector are copied into the resulting vector. If the bits are "0" then the corresponding elements of the first source vector are copied into the resulting vector. Thus a merging/blending of the elements of the two source vectors occurs when this intrinsic is used.

Returns

Result of the merge/blend operation.

_mm256_blendv_pd

Performs conditional blend/merge of float64 vectors. The corresponding Intel® AVX instruction is `VBLENDVPD`*.*

Syntax

```
extern __m256d _mm256_blendv_pd(__m256d m1, __m256d m2, __m256d mask);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation
mask	float64 vector with mask for the operation

Description

Performs a conditional merge of four packed double-precision floating point elements (float64 elements) of two vectors according to the most significant bits of the *mask* parameter elements.

The *mask* parameter defines a mask for the operation. The most significant bit of the corresponding double-precision floating-point elements in the *mask* determines whether the corresponding double-precision floating-point element in the resulting vector is copied from the second source or first source.

If the bit in the *mask* is "1" then the corresponding element of the second source vector is copied into the resulting vector. If the bit is "0" then the corresponding element of the first source vector is copied into the resulting vector. Thus a merging/blending of the elements of the two source vectors occurs when this intrinsic is used.

Returns

Result of the blend operation.

_mm256_blendv_ps

Performs conditional blend/merge of float32 vectors. The corresponding Intel® AVX instruction is `VBLENDVPS`*.*

Syntax

```
extern __m256 _mm256_blendv_ps(__m256 m1, __m256 m2, __m256 mask);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation
mask	float32 vector with the mask for the operation; defined such that the "1" in the most significant bits of an element indicate that corresponding elements of the second source vector are copied into the result, while "0" bits indicate that corresponding elements of the first source vector are copied into the result

Description

Performs a conditional merge of eight packed single-precision floating point elements (float32 elements) of two vectors according to the most significant bits of the *mask* parameter.

The *mask* parameter defines a mask for the operation. The most significant bit of the corresponding single-precision floating-point elements in the *mask* determines whether the corresponding single-precision floating-point element in the resulting vector is copied from the second source or first source.

If the bit in the *mask* is "1" then the corresponding element of the second source vector is copied into the resulting vector. If the bit is "0" then the corresponding element of the first source vector is copied into the resulting vector. Thus a merging/blending of the elements of the two source vectors occurs when this intrinsic is used.

Returns

Result of the blend operation.

Intrinsics for Compare Operations

_mm_cmp_pd, _mm256_cmp_pd

Compares packed 128-bit and 256-bit float64 vector elements. The corresponding Intel® AVX instruction is `VCMPPD`*.*

Syntax

```
extern __m128d _mm_cmp_pd(__m128d m1, __m128d m2, const int predicate);
extern __m256d _mm256_cmp_pd(__m256d m1, __m256d m2, const int predicate);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation
predicate	an immediate operand that specifies the type of comparison to be performed of the packed values; see `immintrin.h` file for the values to specify the type of comparison

Description

Performs a SIMD compare of the four packed double-precision floating-point (float64) values in the first source operand, *m1*, and the second source operand, *m2*, and returns the results of the comparison.

The `_mm_cmp_pd` intrinsic is used for comparing 128-bit float64 values while the `_m256_cmp_pd` intrinsic is used for comparing 256-bit float64 values.

The comparison *predicate* parameter (immediate) specifies the type of comparison performed on each of the pairs of packed values.

Returns

Result of the compare operation.

_mm_cmp_ps, _mm256_cmp_ps

Compares packed float32 elements of two vectors. The corresponding Intel® AVX instruction is `VCMPPS`*.*

Syntax

```
extern __m128 _mm_cmp_ps(__m128 m1, __m128 m2, const int predicate);
```

```
extern __m256 _mm256_cmp_ps(__m256 m1, __m256 m2, const int predicate);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation
predicate	an immediate operand that specifies the type of comparison to be performed of the packed values; see `immintrin.h` file for the values to specify the type of comparison

Description

Performs a SIMD compare of the eight packed single-precision floating-point (float32) values in the first source operand, *m1*, and the second source operand, *m2*, and returns the results of the comparison.

The `_mm_cmp_ps` intrinsic is used for comparing 128-bit float32 values while the `_mm256_cmp_ps` intrinsic is used for comparing 256-bit float32 values.

The comparison *predicate* parameter (immediate) specifies the type of comparison performed on each of the pairs of packed values.

Returns

Result of the compare operation.

_mm_cmp_sd

Compares scalar float64 vectors. The corresponding Intel® AVX instruction is `VCMPSD`*.*

Syntax

```
extern __m128d _mm_cmp_sd(__m128d m1, __m128d m2, const int predicate);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation
predicate	an immediate operand that specifies the type of comparison to be performed of the packed values; see `immintrin.h` file for the values to specify the type of comparison

Description

Performs a compare operation of the low double-precision floating-point values (float64 values) in the first source operand, *m1*, and the second source operand, *m2*, and returns the result.

The comparison *predicate* parameter (immediate operand) specifies the type of comparison performed on each of the pairs of values.

Returns

Result of the compare operation.

_mm_cmp_ss

Compares scalar float32 values. The corresponding Intel© AVX instruction is `VCMPSS`*.*

Syntax

```
extern __m128 _mm_cmp_ss(__m128 m1, __m128 m2, const int predicate);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation
predicate	an immediate operand that specifies the type of comparison to be performed of the packed values; see `immintrin.h` file for the values to specify the type of comparison

Description

Performs a compare operation of the low single-precision floating-point values (float32 values) in the first source operand, *m1*, and the second source operand, *m2*, and returns the results.

The comparison predicate parameter (immediate) specifies the type of comparison performed.

Returns

Result of the compare operation.

Intrinsics for Conversion Operations

_mm256_cvtepi32_pd

Converts extended packed 32-bit integer values to packed double-precision floating point values.The corresponding Intel® AVX instruction is `VCVTDQ2PD`*.*

Syntax

```
extern __m256 _mm256_cvtepi32_pd(__m128i m1);
```

Arguments

m1	integer source vector/operand

Description

Converts four packed signed doubleword integers in the source vector *m1* into four packed double-precision floating-point values.

Returns

Result of the conversion operation.

_mm256_cvtepi32_ps

Converts extended packed 32-bit integer values to packed single-precision floating point values. The corresponding Intel® AVX instruction is `VCVTDQ2PS`*.*

Syntax

```
extern __m256 _mm256_cvtepi32_ps(__m256i m1);
```

Arguments

m1	integer source vector /operand

Description

Converts eight packed signed doubleword integers in the source vector *m1* to eight packed single-precision floating-point values.

Returns

Result of the conversion operation.

_mm256_cvtpd_epi32

Converts packed double-precision float values to extended 32-bit integer values.The corresponding Intel® AVX instruction is `VCVTPD2DQ`*.*

Syntax

```
extern __m128i _mm256_cvtpd_epi32(__m256d m1);
```

Arguments

m1	float64 source vector

Description

Converts four packed double-precision floating-point values in the source vector *m1* to four packed signed doubleword integer (extended 32-bit integer) values in the destination.

Returns

Result of the conversion operation.

_mm256_cvtps_epi32

Converts packed single-precision float values to extended 32-bit integer values. The corresponding Intel® AVX instruction is `VCVTPS2DQ`*.*

Syntax

```
extern __m256i _mm256_cvtps_epi32(__m256 m1);
```

Arguments

m1	float32 source vector

Description

Converts eight packed single-precision floating point values in the source vector *m1* to eight signed doubleword integer (extended 32-bit integer) values.

Returns

Result of the conversion operation.

_mm256_cvtpd_ps

Converts packed float64 values to packed float32 values. The corresponding Intel® AVX instruction is `VCVTPD2PS`*.*

Syntax

```
extern __m128 _mm256_cvtpd_ps(__m256d m1);
```

Arguments

m1	float64 source vector

Description

Converts four packed double-precision floating point values (float64 values) in the source vector *m1* to eight packed single-precision floating-point values (float32 values).

Returns

Result of the conversion operation.

_mm256_cvtps_pd

Converts packed float32 values to packed floati64 values. The corresponding Intel® AVX instruction is `VCVTPS2PD`*.*

Syntax

```
extern __m256d _mm256_cvtps_pd(__m128 m1);
```

Arguments

m1	128-bit float32 source vector

Description

Converts four packed single-precision floating point values (float32 values) in the source vector *m1* to four packed double-precision floating point values (float64 values).

Returns

Result of the conversion operation.

_mm256_cvttp_epi32

Converts packed float64 values to truncated extended 32-bit integer values. The corresponding Intel® AVX instruction is `VCVTTPD2DQ`*.*

Syntax

```
extern __m128i _mm256_cvttpd_epi32(__m256d m1);
```

Arguments

m1	float64 source vector

Description

Converts four packed double-precision floating-point values (float64 values) in the source vector to four packed signed doubleword integer (extended 32-bit integer) values in the destination.

When a conversion is inexact, a truncated (round towards zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Returns

Result of the conversion operation.

_mm256_cvttps_epi32

Converts packed float32 values to truncated extended 32-bit integer values. The corresponding Intel® AVX instruction is `VCVTTPS2DQ`*.*

Syntax

```
extern __m256i _mm256_cvttps_epi32(__m256 m1);
```

Arguments

m1	float32 source vector

Description

Converts eight packed single-precision floating-point values (float32 values) in the source vector to eight packed signed doubleword integer (extended 32-bit integer) values in the destination.

When a conversion is inexact, a truncated (round towards zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised; if this exception is masked, the indefinite integer value (80000000H) is returned.

Returns

Result of the conversion operation.

_mm256_cvtsi256_si32

Extracts a 32-bit integer value.

Syntax

```
int _mm256_cvtsi256_si32(__m256i a);
```

Arguments

a	256-bit integer source vector

Description

Copies the least significant 32 bits of *a* to a 32-bit integer.

Returns

Result of the conversion operation.

_mm256_cvtsd_f64

Extracts a double precision floating point value.

Syntax

```
double _mm256_cvtsd_f64(__m256d a);
```

Arguments

a	float64 source vector

Description

This intrinsic extracts a double precision floating point value from the first vector element of an `__m256d`. It does so in the most efficient manner possible in the context used.

Returns

Result of the conversion operation.

_mm256_cvtss_f32

Extracts a single precision floating point value.

Syntax

```
float _mm256_cvtss_f32(__m256 a);
```

Arguments

a	float32 source vector

Description

Extracts a single precision floating point value from the first vector element of an `__m256`. It does so in the most efficient manner possible in the context used.

Returns

Result of the conversion operation.

Intrinsics to Determine Minimum and Maximum Values

_mm256_max_pd

Determines the maximum of float64 vectors. The corresponding Intel© AVX instruction is `VMAXPD`.

Syntax

```
extern __m256d _mm256_max_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD compare of the packed double-precision floating-point (float64) elements in the first source vector *m1* and the second source vector *m2*, and returns the maximum value for each pair.

Returns

Maximum value of the compare operation.

_mm256_max_ps

Determines the maximum of float32 vectors. The corresponding Intel© AVX instruction is `VMAXPS`.

Syntax

```
extern __m256 _mm256_max_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD compare of the packed single-precision floating-point (float32) elements in the first source vector *m1* and the second source vector *m2*, and returns the maximum value for each pair.

Returns

Maximum value of the compare operation.

_mm256_min_pd

Determines the minimum of float64 vectors. The corresponding Intel® AVX instruction is `VMINPD`*.*

Syntax

```
extern __m256d _mm256_min_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation

Description

Performs a SIMD compare of the packed double-precision floating-point (float64) elements in the first source vector *m1* and the second source vector *m2*, and returns the minimum value for each pair.

Returns

Minimum value of the compare operation.

_mm256_min_ps

Determines the minimum of float32 vectors.The corresponding Intel® AVX instruction is `VMINPS`*.*

Syntax

```
extern __m256 _mm256_min_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Description

Performs a SIMD compare of the packed single-precision floating-point (float32) elements in the first source vector *m1* and the second source vector *m2*, and returns the minimum value for each pair.

Returns

Minimum value of the compare operation.

Intrinsics for Load and Store Operations

_mm256_broadcast_pd

Loads and broadcasts packed double-precision floating point values.The corresponding Intel® AVX instruction is `VBROADCASTF128`*.*

Syntax

```
extern __m256d _mm256_broadcast_pd(__m128d const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant float64 values

Description

Loads 128-bit float64 values from the specified address pointed to by *a*, and broadcasts it to two elements in the destination 256-bit vector.

Returns

Result of the load and broadcast operation.

_mm256_broadcast_ps

Loads and broadcasts packed single-precision floating point values. The corresponding Intel® AVX instruction is `VBROADCASTF128`*.*

Syntax

```
extern __m256 _mm256_broadcast_ps(__m128 const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant 128-bit float32 values

Description

Loads 128-bit float32 values from the specified address pointed to by *a*, and broadcasts it to all elements in the destination 256-bit vector.

Returns

Result of the load and broadcast operation.

_mm256_broadcast_sd

Loads and broadcasts scalar double-precision floating point values to a 256-bit destination operand. The corresponding Intel® AVX instruction is `VBROADCASTSD`*.*

Syntax

```
extern __m256d _mm256_broadcast_sd(double const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant scalar float64 values

Description

Loads scalar double-precision floating-point values from the specified address *a*, and broadcasts it to all four elements in the destination vector.

Returns

Result of the load and broadcast operation.

_mm256_broadcast_ss, _mm_broadcast_ss

Loads and broadcasts 256/128-bit scalar single-precision floating point values to a 256/128-bit destination operand. The corresponding Intel® AVX instruction is `VBROADCASTSS`.

Syntax

```
extern __m256 _mm256_broadcast_ss(float const *a);
```

```
extern __m128 _mm_broadcast_ss(float const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant 256-bit or 128-bit float32 values

Description

Loads scalar single-precision floating-point values from the specified address pointed to by *a*, and broadcasts it to elements in the destination vector.

The `_m256_broadcast_ss` intrinsic broadcasts the loaded values to all eight elements in the 256-bit destination vector.

The `_mm_broadcast_ss` intrinsic broadcasts the loaded values to all four elements in the 128-bit destination vector.

Returns

Result of the load and broadcast operation.

_mm256_load_pd

Moves packed double-precision floating point values from aligned memory location to a destination vector. The corresponding Intel® AVX instruction is`VMOVAPD`.

Syntax

```
extern __m256d _mm256_load_pd(double const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant float64 values; the address must be 32-byte aligned

Description

Loads packed double-precision floating point values (float64 values) from the 256-bit aligned memory location pointed to by *a*, into a destination float64 vector, which is retured by the intrinsic.

Returns

A 256-bit vector with float64 values.

_mm256_load_ps

Moves packed single-precision floating point values from aligned memory location to a destination vector. The corresponding Intel© AVX instruction is`VMOVAPS`*.*

Syntax

```
extern __m256 _mm256_load_ps(float const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant float32 values; the address must be 32-byte aligned

Description

Loads packed single-precision floating point values (float32 values) from the 256-bit aligned memory location pointed to by *a*, into a destination float32 vector, which is retured by the intrinsic.

Returns

A 256-bit vector with float32 values.

_mm256_load_si256

Moves integer values from aligned memory location to a destination vector. The corresponding Intel© AVX instruction is`VMOVDQA`*.*

Syntax

```
extern __m256i _mm256_load_si256(__m256i const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant integer values; the address must be 32-byte aligned

Description

Loads integer values from the 256-bit aligned memory location pointed to by **a*, into a destination integer vector, which is returned by the intrinsic.

Returns

A 256-bit vector with integer values.

_mm256_loadu_pd

Moves packed double-precision floating point values from unaligned memory location to a destination vector. The corresponding Intel© AVX instruction is `VMOVUPD`*.*

Syntax

```
extern __m256d _mm256_loadu_pd(double const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant float64 values;

Description

Loads packed double-precision floating point values (float64 values) from the 256-bit unaligned memory location pointed to by *a*, into a destination float64 vector, which is retured by the intrinsic.

Returns

A 256-bit vector with float64 values.

_mm256_loadu_ps

Moves packed single-precision floating point values from unaligned memory location to a destination vector. The corresponding Intel® AVX instruction is`VMOVUPS`*.*

Syntax

```
extern __m256 _mm256_loadu_ps(float const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant float32 values

Description

Loads packed single-precision floating point values (float32 values) from the 256-bit unaligned memory location pointed to by *a*, into a destination float32 vector, which is retured by the intrinsic.

Returns

A 256-bit vector with float32 values.

_mm256_loadu_si256

Moves integer values from unaligned memory location to a destination vector. The corresponding Intel® AVX instruction is`VMOVDQU`*.*

Syntax

```
extern __m256i _mm256_loadu_si256(__m256i const *a);
```

Arguments

**a*	pointer to a memory location that can hold constant integer values

Description

Loads integer values from the 256-bit unaligned memory location pointed to by **a*, into a destination integer vector, which is returned by the intrinsic.

Returns

A 256-bit vector with integer values.

_mm256_maskload_pd, _mm_maskload_pd

Loads packed double-precision floating point values according to mask values. The corresponding Intel® AVX instruction is `VMASKMOVPD`.

Syntax

```
extern __m256d _mm256_maskload_pd(double const *a, __m256i mask);
```

```
extern __m128d _mm_maskload_pd(double const *a, __m128i mask);
```

Arguments

**a*	pointer to a 256/128-bit memory location that can hold constant float64 values
mask	integer value calculated based on the most-significant-bit of each quadword of a mask register

Description

Loads packed double-precision floating point (float64) values from the 256/128-bit memory location pointed to by *a*, into a destination register using the *mask* value.

The *mask* is calculated from the most significant bit of each qword of the *mask* register. If any of the bits of the *mask* is set to zero, the corresponding value from the memory location is not loaded, and the corresponding field of the destination vector is set to zero.

Returns

A 256/128-bit register with float64 values.

_mm256_maskload_ps, _mm_maskload_ps

Loads packed single-precision floating point values according to mask values. The corresponding Intel® AVX instruction is `VMASKMOVPS`.

Syntax

```
extern __m256 _mm256_maskload_ps(float const *a, __m256i mask);
```

```
extern __m128 _mm_maskload_ps(float const *a, __m128i mask);
```

Arguments

**a*	pointer to a 256/128-bit memory location that can hold constant float32 values
mask	integer value calculated based on the most-significant-bit of each doubleword of a mask register

Description

Loads packed single-precision floating point (float32) values from the 256/128-bit memory location pointed to by *a*, into a destination register using the *mask* value.

The *mask* is calculated from the most significant bit of each dword of the *mask* register. If any of the bits of the *mask* is set to zero, the corresponding value from the memory location is not loaded, and the corresponding field of the destination vector is set to zero.

Returns

A 256/128-bit register with float32 values.

_mm256_store_pd

Moves packed double-precision floating point values from a float64 vector to an aligned memory location. The corresponding Intel© AVX instruction is`VMOVAPD`*.*

Syntax

```
extern void _mm256_store_pd(double *a, __m256d b);
```

Arguments

**a*	pointer to a memory location that can hold double-precision floating point (float64) values; the address must be 32-byte aligned
b	float64 vector

Description

Performs a store operation by moving packed double-precision floating point values (float64 values) from a float64 vector, *b*, to a 256-bit aligned memory location, pointed to by *a*.

Returns

Nothing

_mm256_store_ps

Moves packed single-precision floating point values from a float32 vector to an aligned memory location. The corresponding Intel© AVX instruction is`VMOVAPS`*.*

Syntax

```
extern void _mm256_store_ps(float *a, __m256 b);
```

Arguments

**a*	pointer to a memory location that can hold single-precision floating point (float32) values; the address must be 32-byte aligned
b	float32 vector

Description

Performs a store operation by moving packed single-precision floating point values (float32 values) from a float32 vector, *b*, to a 256-bit aligned memory location, pointed to by *a*.

Returns

Nothing.

_mm256_store_si256

Moves values from a integer vector to an aligned memory location. The corresponding Intel® AVX instruction is `VMOVDQA`*.*

Syntax

```
extern void _mm256_store_si256(__m256i *a, __m256i b);
```

Arguments

**a*	pointer to a memory location that can hold scalar integer values; the address must be 32-byte aligned.
b	integer vector

Description

Performs a store operation by moving integer values from a 256-bit integer vector, *b*, to a 256-bit aligned memory location, pointed to by *a*.

Returns

Nothing.

_mm256_storeu_pd

Moves packed double-precision floating point values from a float64 vector to an unaligned memory location. The corresponding Intel® AVX instruction is`VMOVUPD`*.*

Syntax

```
extern void _mm256_storeu_pd(double *a, __m256d b);
```

Arguments

**a*	pointer to a memory location that can hold double-precision floating point (float64) values
b	float64 vector

Description

Performs a store operation by moving packed double-precision floating point values (float64 values) from a float64 vector, *b*, to a 256-bit unaligned memory location, pointed to by *a*.

Returns

Nothing.

_mm256_storeu_ps

Moves packed single-precision floating point values from a float32 vector to an unaligned memory location. The corresponding Intel® AVX instruction is`VMOVUPS`*.*

Syntax

```
extern void _mm256_storeu_ps(float *a, __m256 b);
```

Arguments

**a*	pointer to a memory location that can hold single-precision floating point (float32) values
b	float32 vector

Description

Performs a store operation by moving packed single-precision floating point values (float32 values) from a float32 vector, *b*, to a 256-bit unaligned memory location, pointed to by *a*.

Returns

Nothing.

_mm256_storeu_si256

Moves values from a integer vector to an unaligned memory location. The corresponding Intel® AVX instruction is `VMOVDQU`*.*

Syntax

```
extern void _mm256_storeu_si256(__m256i *a, __m256i b);
```

Arguments

**a*	pointer to a memory location that can hold scalar integer values
b	integer vector

Description

Performs a store operation by moving integer values from a 256-bit integer vector, *b*, to a 256-bit unaligned memory location, pointed to by *a*.

Returns

Nothing.

_mm256_stream_pd

Moves packed double-precision floating-point values using non-temporal hint. The corresponding Intel® AVX instruction is `VMOVNTPD`*.*

Syntax

```
extern void _mm256_stream_pd(double *p, __m256d a);
```

Arguments

**p*	pointer to a memory location that can hold double-precision floating point (float64) values; the address must be 32-byte aligned
a	float64 vector

Description

Performs a store operation by moving packed double-precision floating point values (float64 values) from a float64 vector, *a*, to a 256-bit aligned memory location, pointed to by *p*, using a non-temporal hint to prevent caching of the data during the write to memory.

Returns

Result of the streaming/store operation.

_mm256_stream_ps

Moves packed single-precision floating-point values using non-temporal hint. The corresponding Intel® AVX instruction is `VMOVNTPS`*.*

Syntax

```
extern void _mm256_stream_ps(float *p, __m256 a);
```

Arguments

**p*	pointer to a memory location that can hold single-precision floating point (float32) values; the address must be 32-byte aligned
a	float32 vector

Description

Performs a store operation by moving packed single-precision floating point values (float32 values) from a float32 vector, *a*, to a 256-bit aligned memory location, pointed to by *p*, using a non-temporal hint to prevent caching of the data during the write to memory.

Returns

Result of the streaming/store operation.

_mm256_stream_si256

Moves packed integer values using non-temporal hint. The corresponding Intel® AVX instruction is `VMOVNTDQ`*.*

Syntax

```
extern void _mm256_stream_si256(__m256i *p, __m256i a);
```

Arguments

**p*	pointer to a memory location that can hold scalar integer values; the address must be 32-byte aligned

a	integer vector

Description

Performs a store operation by moving scalar integer values from an integer vector *a*, to a 256-bit aligned memory location, pointed to by *p*, using a non-temporal hint to prevent caching of the data during the write to memory.

Returns

Result of the streaming/store operation.

_mm256_maskstore_pd, _mm_maskstore_pd

Stores packed double-precision floating point values according to mask values. The corresponding Intel® AVX instruction is `VMASKMOVPD`*.*

Syntax

```
extern void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);
```

```
extern void _mm_maskstore_pd(double *a, __m128i mask, __m128d b);
```

Arguments

**a*	pointer to a 256/128-bit memory location that can hold constant double-precision floating point (float64) values
mask	integer value calculated based on the most-significant-bit of each quadword of a mask register
b	a 256/128-bit float64 vector

Description

Performs a store operation by moving packed double-precision floating point (float64) values from a vector, *b*, to a 256/128-bit memory location, pointed to by *a*, using a *mask*.

The *mask* is calculated from the most significant bit of each qword of the *mask* register. If any of the bits of the mask are set to zero, the corresponding value from the float64 vector is not loaded, and the corresponding field of the destination memory location is left unchanged.

> **NOTE**
> Stores are atomic. Faults do not occur for memory locations for which all corresponding mask bits are set to zero.

Returns

Nothing.

_mm256_maskstore_ps, _mm_maskstore_ps

Stores packed single-precision floating point values according to mask values. The corresponding Intel® AVX instruction is `VMASKMOVPS`*.*

Syntax

```
extern void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b);
```

```
extern void _mm_maskstore_ps(float *a, __m128i mask, __m128 b);
```

Arguments

**a*	pointer to a 256/128-bit memory location that can hold constant single-precision floating point (float32) values
mask	integer value calculated based on the most-significant-bit of each quadword of a mask register
b	a 256/128-bit float32 vector

Description

Performs a store operation by moving packed single-precision floating point (float32) values from a vector, *b*, to a 256/128-bit memory location, pointed to by *a*, using a *mask*.

The *mask* is calculated from the most significant bit of each qword of the *mask* register. If any of the bits of the mask are set to zero, the corresponding value from the float32 vector is not loaded, and the corresponding field of the destination memory location is left unchanged.

> **NOTE**
> Stores are atomic. Faults do not occur for memory locations for which all corresponding mask bits are set to zero.

Returns

Nothing.

Intrinsics for Miscellaneous Operations

_mm256_extractf128_pd

Extracts 128-bit packed float64 values. The corresponding Intel© AVX instruction is `VEXTRACTF128`*.*

Syntax

```
extern __m128d _mm256_extractf128_pd(__m256d m1, const int offset);
```

Arguments

m1	float64 source vector
offset	a constant integer value that represents the 128-bit offset from where extraction must start

Description

Extracts 128-bit packed double-precision floating point values (float64 values) from the source vector *m1*, starting from the location specified by the value in the *offset* parameter.

Returns

Result of the extraction operation.

_mm256_extractf128_ps

Extracts 128-bit float32 values. The corresponding Intel© AVX instruction is `VEXTRACTF128`.

Syntax

```
extern __m128 _mm256_extractf128_ps(__m256 m1, const int offset);
```

Arguments

m1	float32 source vector
offset	a constant integer value that represents the 128-bit offset from where extraction must start

Description

Extracts 128-bit packed single-precision floating point values (float32 values) from the source vector *m1*, starting from the location specified by the value in the *offset* parameter.

Returns

Result of the extraction operation.

_mm256_extractf128_si256

Extracts 128-bit scalar integer values. The corresponding Intel© AVX instruction is `VEXTRACTF128`.

Syntax

```
extern __m128i _mm256_extractf128_si256(__m256i m1, const int offset);
```

Arguments

m1	integer source vector
offset	a constant integer value that represents the offset from where extraction must start

Description

Extracts 128-bit scalar integer values from the source vector *m1*, starting from the location specified by the value in the *offset* parameter.

Returns

Result of the extraction operation.

_mm256_insertf128_pd

Inserts 128 bits of packed float64 values. The corresponding Intel© AVX instruction is `VINSERTF128`.

Syntax

```
extern __m256d _mm256_insertf128_pd(__m256d a, __m128d b, int offset);
```

Arguments

a	256-bit float64 source vector
b	128-bit float64 source vector
offset	an integer value that represents the 128-bit offset where the insertion must start

Description

Performs an insertion of 128 bits of packed double-precision floating-point values (float64 values) from the second source vector *b* into a destination at a 128-bit offset specified by the *offset* parameter. The remaining portions of the destination are written by the corresponding elements of the first source vector *a*.

Returns

Result of the insertion operation.

_mm256_insertf128_ps

Inserts 128 bits of packed float32 values. The corresponding Intel® AVX instruction is `VINSERTF128`*.*

Syntax

```
extern __m256 _mm256_insertf128_ps(__m256 a, __m128 b, int offset);
```

Arguments

a	256-bit float32 source vector
b	128-bit float32 source vector
offset	an integer value that represents the 128-bit offset where the insertion must start

Description

Performs an insertion of 128 bits of packed single-precision floating-point values (float32 values) from the second source vector *b*, into a destination at a 128-bit offset specified by the *offset* parameter. The remaining portions of the destination are written by the corresponding elements of the first source vector *a*.

Returns

Result of the insertion operation.

_mm256_insertf128_si256

Inserts 128 bits of packed scalar integer values . The corresponding Intel® AVX instruction is `VINSERTF128`*.*

Syntax

```
extern __m256i _mm256_insertf128_si256(__m256i a, __m128i b, int offset);
```

Arguments

a	256-bit integer source vector
b	128-bit integer source vector
offset	an integer value that represents the 128-bit offset where the insertion must start

Description

Performs an insertion of 128 bits of packed scalar integer values from the second source vector *b*, into a destination at a 128-bit offset specified by the *offset* parameter. The remaining portions of the destination are written by the corresponding elements of the first source vector *a*.

Returns

Result of the insertion operation.

_mm256_lddqu_si256

Moves unaligned integer from memory. The corresponding Intel® AVX instruction is `VLDDQU`.

Syntax

```
extern __m256i _mm256_lddqu_si256(__m256i const *a);
```

Arguments

**a*	points to a memory location from where unaligned integer value must be moved

Description

Fetches 32 bytes of data, starting at a memory address specified by the *a* parameter, and places them in a destination. This intrinsic calls the corresponding instruction `VLDDQU`, which performs an operation functionally similar to the `VMOVDQU` instruction.

Returns

Result of the move operation.

_mm256_movedup_pd

Duplicates even-indexed double-precision floating point values. The corresponding Intel® AVX instruction is `VMOVDDUP`.

Syntax

```
extern __m256d _mm256_movedup_pd(__m256d a);
```

Arguments

a	float64 source vector

Description

Performs a duplication of even-indexed double-precision floating-point values (float64 values) in the source vector *a*, and returns the result.

Returns

Result of the duplication operation.

_mm256_movehdup_ps

Duplicates odd-indexed single-precision floating point values. The corresponding Intel® AVX instruction is `VMOVSHDUP`.

Syntax

```
extern __m256 _mm256_movehdup_ps(__m256 a);
```

Arguments

a	float32 source vector

Description

Performs a duplication of odd-indexed single-precision floating-point values (float32 values) in the source vector *a*, and returns the result.

Returns

Result of the duplication operation.

_mm256_moveldup_ps

Duplicates even-indexed single-precision floating point values. The corresponding Intel® AVX instruction is `VMOVSLDUP`.

Syntax

```
extern __m256 _mm256_moveldup_ps(__m256 a);
```

Arguments

a	float32 source vector

Description

Performs a duplication of even-indexed single-precision floating-point values (float32 values) in the source vector *a*, and returns the result.

Returns

Result of the duplication operation.

_mm256_movemask_pd

Extracts float64 sign mask. The corresponding Intel® AVX instruction is `VMOVMSKPD`.

Syntax

```
extern int _mm256_movemask_pd(__m256d a);
```

Arguments

a	float64 source vector

Description

Performs an extract operation of sign bits from four double-precision floating point elements (float64 elements) of the source vector *a*, and composes them into bitmasks.

Returns

An integer bitmask of four meaningful bits.

_mm256_movemask_ps

Extracts float32 sign mask. The corresponding Intel® AVX instruction is `VMOVMSKPS`*.*

Syntax

```
extern int _mm256_movemask_ps(__m256 a;
```

Arguments

a	float32 source vector

Description

Performs an extract operation of sign bits from eight single-precision floating point elements (float32 elements) of the source vector *a*, and composes them into bitmasks.

Returns

An integer bitmask of eight meaningful bits.

_mm256_round_pd

Rounds off double-precision floating point values to nearest upper/lower integer depending on rounding mode. The corresponding Intel® AVX instruction is`VROUNDPD`*.*

Syntax

```
extern __m256d _mm256_round_pd(__m256d a, int iRoundMode);
```

Arguments

a	float64 vector
iRoundMode	A hexadecimal value dependent on rounding mode: • For rounding off to upper integer, the value is `0x0A` • For rounding off to lower integer, the value is `0x09`

Description

Rounds off the elements of a float64 vector *a* to the nearest upper/lower integer value. Two shortcuts, in the form of `#defines`, are used to achieve these two separate operations:

```
#define _mm256_ceil_pd(a)    _mm256_round_pd((a), 0x0A)
```

```
#define _mm256_floor_pd(a)   _mm256_round_pd((a), 0x09)
```

These `#defines` tell the preprocessor to replace all instances of `_mm256_ceil_pd(a)` with `_mm256_round_pd((a), 0x0A)` and all instances of `_mm256_floor_pd(a)` with `_mm256_round_pd((a), 0x09)`.

For example, if you write the following:

```
__m256 a, b;
```

```
a = _mm256_ceil_pd(b);
```

the preprocessor will modify it to:

```
a = _mm256_round_pd(b, 0x0a);
```

The Precision Floating Point Exception is signaled according to the (immediate operand) *iRoundMode* value.

Returns

Result of the rounding off operation as a vector with double-precision floating point values.

_mm256_round_ps

Rounds off single-precision floating point values to nearest upper/lower integer depending on rounding mode. The corresponding Intel© AVX instruction is`VROUNDPS`*.*

Syntax

```
extern __m256 _mm256_round_ps(__m256 a, int iRoundMode );
```

Arguments

a	float32 vector
iRoundMode	A hexadecimal value dependent on rounding mode: • For rounding off to upper integer, the value is `0x0A` • For rounding off to lower integer, the value is `0x09`

Description

Rounds off the elements of a float32 vector *a* to the nearest upper/lower integer value. Two shortcuts, in the form of `#defines`, are used to achieve these two separate operations:

```
#define _mm256_ceil_ps(a)    _mm256_round_ps((a), 0x0A)
#define _mm256_floor_ps(a)   _mm256_round_ps((a), 0x09)
```

These `#defines` tells the preprocessor to replace all instances of `_mm256_ceil_ps(a)` with `_mm256_round_ps((a), 0x0A)` and all instances of `_mm256_floor_ps(a)` with `_mm256_round_ps((a), 0x09)`.

For example, if you write the following:

```
__m256 a, b;
a = _mm256_ceil_ps(b);
```

the preprocessor will modify it to:

```
a = _mm256_round_ps(b, 0x0a);
```

The Precision Floating Point Exception is signaled according to the (immediate operand) *iRoundMode* value.

Returns

Result of the rounding off operation as a vector with single-precision floating point values.

_mm256_set_pd

Initializes 256-bit vector with float64 values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256d _mm256_set_pd(double, double, double, double);
```

Arguments

double	a float64 value to be initialized into the 256-bit vector; there are four *double* parameters, one for each float64 vector element

Description

Initializes a 256-bit vector with double-precision floating point values (float64 values) as specified by the *double* parameter.

Returns

A float64 vector initialized with specified double-precision floating point values.

_mm256_set_ps

Initializes 256-bit vector with float32 values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256 _mm256_set_ps(float, float, float, float, float, float, float, float);
```

Arguments

float	a float32 value to be initialized into the 256-bit vector; there are eight *float* parameters, one for each float32 vector element

Description

Initializes a 256-bit vector with single-precision floating point values (float32 values) as specified by the *float* parameter.

Returns

A float32 vector initialized with specified single-precision floating point values.

_mm256_set_epi8/16/32/64x

Initializes 256-bit vector with integer values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_set_epi8(char e31, char e30, char e29, char e28, char e27, char
e26, char e25, char e24, char e23, char e22, char e21, char e20, char e19, char e18,
char e17, char e16, char e15, char e14, char e13, char e12, char e11, char e10, char
e9, char e8, char e7, char e6, char e5, char e4, char e3, char e2, char e1, char e0);
```

```
extern __m256i _mm256_set_epi316(short e15, short e14, short e13, short e12, short e11,
short e10, short e9, short e8, short e7, short e6, short e5, short e4, short e3, short
e2, short e1, short e0);
```

```
extern __m256i _mm256_set_epi32(int e7, int e6, int e5, int e4, int e3, int e2, int e1,
int e0);
```

```
extern __m256i _mm256_set_epi64x(__int64 e3, __int64 e2, __int64 e1, __int64 e0);
```

Arguments

e_n	An 8/16/32/64-bit integer value to be initialized into the 256-bit vector. For each variant, there is one integer parameter for each 8/16/32/64-bit integer vector element.

Description

Initializes a 256-bit vector with extended packed integer values (8/16/32/64-bit values) as specified by the e_n parameter.

Returns

An 8/16/32/64-bit integer vector initialized with specified extended packed integer values.

_mm256_setr_pd

Initializes 256-bit vector with float64 values in reverse of specified order. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256d _mm256_setr_pd(double, double, double, double);
```

Arguments

double	a float64 value to be initialized into the 256-bit vector; there are four *double* parameters, one for each float64 vector element

Description

Initializes a 256-bit vector with double-precision floating point values (float64 values) in reverse order as specified by the *double* parameter.

Returns

A float64 vector initialized with specified double-precision floating point values in reverse.

_mm256_setr_ps

Initializes 256-bit vector with float32 values in reverse of specified order. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256 _mm256_setr_ps(float, float, float, float, float, float, float, float);
```

Arguments

float	a float32 value to be initialized into the 256-bit vector; there are eight *float* parameters, one for each float32 vector element

Description

Initializes a 256-bit vector with single-precision floating point values (float32 values) in reverse order as specified by the *float* parameter.

Returns

A float32 vector initialized with specified single-precision floating point values in reverse.

_mm256_setr_epi32

Initializes 256-bit vector with integer values in reverse of specified order. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_setr_epi32(int, int, int, int, int, int, int, int);
```

```
extern __m256i _mm256_setr_epi8(char e31, char e30, char e29, char e28, char e27, char
e26, char e25, char e24, char e23, char e22, char e21, char e20, char e19, char e18,
char e17, char e16, char e15, char e14, char e13, char e12, char e11, char e10, char
e9, char e8, char e7, char e6, char e5, char e4, char e3, char e2, char e1, char e0);
```

```
extern __m256i _mm256_setr_epi316(short e15, short e14, short e13, short e12, short
e11, short e10, short e9, short e8, short e7, short e6, short e5, short e4, short e3,
short e2, short e1, short e0);
```

```
extern __m256i _mm256_setr_epi32(int e7, int e6, int e5, int e4, int e3, int e2, int
e1, int e0);
```

```
extern __m256i _mm256_setr_epi64x(__int64 e3, __int64 e2, __int64 e1, __int64 e0);
```

Arguments

e_n	An 8/16/32/64-bit integer value to be initialized into the 256-bit vector. For each variant, there is one integer parameter for each 8/16/32/64-bit integer vector element.

Description

Initializes a 256-bit vector with extended packed integer values (8/16/32/64-bit values), in reverse order, as specified by the e_n parameter.

Returns

An 8/16/32/64-bit integer vector initialized with specified extended packed integer values in reverse.

_mm256_set1_pd

Initializes 256-bit vector with scalar double-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256d _mm256_set1_pd(double);
```

Arguments

double	a float64 value to be initialized into the 256-bit vector; there is one *double* parameter to initialize each float64 vector element

Description

Initializes a 256-bit vector with scalar double-precision floating point values (float64 values) as specified by the *double* parameter.

Returns

A float64 vector with all elements set to the specified scalar double-precision floating point value.

_mm256_set1_ps

Initializes 256-bit vector with scalar single-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256 _mm256_set1_ps(float);
```

Arguments

float	a float32 value to be initialized into the 256-bit vector; there is one *float* parameter to initialize each float32 vector element

Description

Initializes a 256-bit vector with scalar single-precision floating point values (float32 values) as specified by the *float* parameter.

Returns

A float32 vector with all elements set to the specified scalar single-precision floating point value.

_mm256_set1_epi32

Initializes 256-bit vector with scalar integer values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_set1_epi8(char a);
```

```
extern __m256i _mm256_set1_epi16(short a);
extern __m256i _mm256_set1_epi32(int a);
extern __m256i _mm256_set1_epi64x(long long a);
```

Arguments

a	An 8/16/32/64-bit integer value to be initialized into the 256-bit vector. For each variant, there is one integer parameter for each 8/16/32/64-bit integer vector element.

Description

Initializes a 256-bit vector with scalar integer values (8/16/32/64-bit values) as specified by the *a* parameter.

Returns

An 8/16/32/64-bit integer vector with all elements set to the specified scalar integer value.

_mm256_setzero_pd

Sets float64 YMM registers to zero. No corresponding Intel© AVX instruction.

Syntax

```
extern __m256d _mm256_setzero_pd(void);
```

Arguments

None

Description

Sets all the elements of a float64 vector to zero and returns the float64 vector. This is a utility intrinsic that helps during programming.

Returns

A float64 vector with all elements set to zero.

_mm256_setzero_ps

Sets float32 YMM registers to zero. No corresponding Intel© AVX instruction.

Syntax

```
extern __m256 _mm256_setzero_ps(void);
```

Arguments

None

Description

Sets all the elements of a float32 vector to zero and returns the float32 vector. This is a utility intrinsic that helps during programming.

Returns

A float32 vector with all elements set to zero.

_mm256_setzero_si256

Sets integer YMM registers to zero. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_setzero_si256(void);
```

Arguments

None

Description

Sets all the elements of an integer vector to zero and returns the integer vector. This is a utility intrinsic that helps during programming.

Returns

An integer vector with all elements set to zero.

_mm256_zeroall

Zeroes all YMM registers. The corresponding Intel® AVX instruction is `VZEROALL`.

Syntax

```
extern void _mm256_zeroall(void);
```

Arguments

None

Description

Zeroes all `YMM` registers. This intrinsic is useful to clear all the YMM registers when transitioning between Intel® Advanced Vector Extensions (Intel® AVX) instructions and legacy Intel® Supplemental SIMD Extensions (Intel® SSE) instructions.

There is no transition penalty if an application clears the bits of all `YMM` registers (sets to '0') via `VZEROALL`, the corresponding instruction for this intrinsic, before transitioning between Intel® Advanced Vector Extensions (Intel® AVX) instructions and legacy Intel® Supplemental SIMD Extensions (Intel® SSE) instructions.

Returns

Nothing

_mm256_zeroupper

Zeroes the upper bits of the YMM registers. The corresponding Intel® AVX instruction is `VZEROUPPER`.

Syntax

```
extern void _mm256_zeroupper(void);
```

Arguments

None

Description

Zeroes the upper 128 bits of all YMM registers. The lower 128 bits that correspond to the XMM registers are left unmodified.

This intrinsic is useful to clear the upper bits of the YMM registers when transitioning between Intel® Advanced Vector Extensions (Intel® AVX) instructions and legacy Intel® Supplemental SIMD Extensions (Intel® SSE) instructions. There is no transition penalty if an application clears the upper bits of all YMM registers (sets to '0') via VZEROUPPER, the corresponding instruction for this intrinsic, before transitioning between Intel® Advanced Vector Extensions (Intel® AVX) instructions and legacy Intel® Supplemental SIMD Extensions (Intel® SSE) instructions.

Returns

Result of the operation.

Intrinsics for Packed Test Operations

_mm256_testz_si256

Performs a packed bit test of two integer vectors to set the ZF flag. The corresponding Intel® AVX instruction is VPTEST*.*

Syntax

```
extern int _mm256_testz_si256(__m256i s1, __m256i s2);
```

Arguments

s1	first source integer vector
s2	second source integer vector

Description

Allows setting of the ZF flag. The ZF flag is set based on the result of a bitwise AND operation between the first and second source vectors. The corresponding instruction VPTEST sets the ZF flag if all the resulting bits are 0. If the resulting bits are non-zeros, the instruction clears the ZF flag.

Returns

Non-zero if ZF flag is set

Zero if the ZF flag is not set

_mm256_testc_si256

Performs a packed bit test of two integer vectors to set the CF flag. The corresponding Intel® AVX instruction is VPTEST*.*

Syntax

```
extern int _mm256_testc_si256(__m256i s1, __m256i s2);
```

Arguments

s1	first source integer vector

s2	second source integer vector

Description

Allows setting of the CF flag. The CF flag is set based on the result of a bitwise AND and logical NOT operation between the first and second source vectors. The corresponding instruction, `VPTEST`, sets the CF flag if all the resulting bits are 0. If the resulting bits are non-zeros, the instruction clears the CF flag.

Returns

Non-zero if CF flag is set

Zero if the CF flag is not set

_mm256_testnzc_si256

Performs a packed bit test of two integer vectors to set ZF and CF flags. The corresponding Intel® AVX instruction is `VPTEST`*.*

Syntax

```
extern int _mm256_testnzc_si256(__m256i s1, __m256i s2);
```

Arguments

s1	first source integer vector
s2	second source integer vector

Description

Performs a packed bit test of *s1* and *s2* vectors using `VTESTPD`*s1*, *s2* instruction and checks the status of the ZF and CF flags. The intrinsic returns 1 if both ZF and CF flags are not 1 (that is, both flags are not set), otherwise returns 0 (that is, one of the flags is set) .

The `VTESTPD` instruction performs a bitwise comparison of all the sign bits of the integer elements in the first source operand and corresponding sign bits in the second source operand. If the AND of the first source operand sign bits with the second source operand sign bits produces all zeros, the ZF flag is set else the ZF flag is clear. If the AND of the inverted first source operand sign bits with the second source operand sign bits produces all zeros the CF flag is set, else the CF flag is clear.

Returns

1: indicates that both ZF and CF flags are clear

0: indicates that either ZF or CF flag is set

_mm256_testz_pd, _mm_testz_pd

Performs a packed bit test of two float64 256-bit or 128-bit vectors to set the ZF flag. The corresponding Intel® AVX instruction is `VTESTPD`*.*

Syntax

```
extern int _mm256_testz_pd(__m256d s1, __m256d s2);

extern int _mm_testz_pd(__m128d s1, __m128d s2);
```

Arguments

s1	first float64 source vector
s2	second float64 source vector

Description

Compute the bitwise AND of the two vectors s1 and s2, representing double-precision (64-bit) floating-point elements, producing an intermediate value, and set ZF to 1 if the sign bit of each 64-bit element in the intermediate value is zero, otherwise set ZF to 0. Compute the bitwise AND NOT of s1 and s2, producing an intermediate value, and set CF to 1 if the sign bit of each 64-bit element in the intermediate value is zero, otherwise set CF to 0. Return the ZF value.

> **NOTE**
> Intel® Advanced Vector Extensions (Intel® AVX) instructions include a full compliment of 128-bit SIMD instructions. Such Intel® AVX instructions, with vector length of 128-bits, zeroes the upper 128 bits of the `YMM` register. The lower 128 bits of the `YMM` register is aliased to the corresponding SIMD `XMM` register.

Returns

Non-zero if ZF flag is set

Zero if the ZF flag is not set

_mm256_testz_ps, _mm_testz_ps

Performs a packed bit test of two float32 256-bit or 128-bit vectors to set the ZF flag. The corresponding Intel® AVX instruction is `VTESTPS`.

Syntax

```
extern int _mm256_testz_ps(__m256 s1, __m256 s2);
```

```
extern int _mm_testz_ps(__m128 s1, __m128 s2);
```

Arguments

s1	first source float32 vector
s2	second source float32 vector

Description

Compute the bitwise AND of the two vectors s1 and s2, representing single-precision (32-bit) floating-point elements, producing an intermediate value, and set ZF to 1 if the sign bit of each 32-bit element in the intermediate value is zero, otherwise set ZF to 0. Compute the bitwise AND NOT of s1 and s2, producing an intermediate value, and set CF to 1 if the sign bit of each 32-bit element in the intermediate value is zero, otherwise set CF to 0. Return the ZF value.

NOTE
Intel® Advanced Vector Extensions (Intel® AVX) instructions include a full compliment of 128-bit SIMD instructions. Such Intel® AVX instructions, with vector length of 128-bits, zeroes the upper 128 bits of the YMM register. The lower 128 bits of the YMM register is aliased to the corresponding SIMD XMM register.

Returns

Non-zero if ZF flag is set

Zero if the ZF flag is not set

_mm256_testc_pd, _mm_testc_pd

Performs a packed bit test of two 256-bit or 128-bit float64 vectors to set the CF flag. The corresponding Intel® AVX instruction is `VTESTPD`.

Syntax

```
extern int _mm256_testc_pd(__m256d s1, __m256d s2);
extern int _mm_testc_pd(__m128d s1, __m128d s2);
```

Arguments

s1	first source float64 vector
s2	second source float64 vector

Description

Compute the bitwise AND of the two vectors s1 and s2, representing double-precision (64-bit) floating-point elements, producing an intermediate value, and set ZF to 1 if the sign bit of each 64-bit element in the intermediate value is zero, otherwise set ZF to 0. Compute the bitwise AND NOT of s1 and s2, producing an intermediate value, and set CF to 1 if the sign bit of each 64-bit element in the intermediate value is zero, otherwise set CF to 0. Return the CF value.

NOTE
Intel® Advanced Vector Extensions (Intel® AVX) instructions include a full compliment of 128-bit SIMD instructions. Such Intel® AVX instructions, with vector length of 128-bits, zeroes the upper 128 bits of the YMM register. The lower 128 bits of the YMM register is aliased to the corresponding SIMD XMM register.

Returns

Non-zero if CF flag is set

Zero if the CF flag is not set

_mm256_testc_ps, _mm_testc_ps

Performs a packed bit test of two 256-bit or 128-bit float32 vectors to set the CF flag. The corresponding Intel® AVX instruction is `VTESTPS`.

Syntax

```
extern int _mm256_testc_ps(__m256 s1, __m256 s2);
extern int _mm_testc_ps(__m128 s1, __m128 s2);
```

Arguments

s1	first source float32 vector
s2	second source float32 vector

Description

Compute the bitwise AND of the two vectors s1 and s2, representing single-precision (32-bit) floating-point elements, producing an intermediate value, and set ZF to 1 if the sign bit of each 32-bit element in the intermediate value is zero, otherwise set ZF to 0. Compute the bitwise AND NOT of s1 and s2, producing an intermediate value, and set CF to 1 if the sign bit of each 32-bit element in the intermediate value is zero, otherwise set CF to 0. Return the ZF value.

> **NOTE**
> Intel® Advanced Vector Extensions (Intel® AVX) instructions include a full compliment of 128-bit SIMD instructions. Such Intel® AVX instructions, with vector length of 128-bits, zeroes the upper 128 bits of the YMM register. The lower 128 bits of the YMM register is aliased to the corresponding SIMD XMM register.

Returns

Non-zero if CF flag is set

Zero if the CF flag is not set

_mm256_testnzc_pd, _mm_testnzc_pd

Performs a packed bit test of two 256-bit float64 or 128-bit float64 vectors to check ZF and CF flag settings. The corresponding Intel® AVX instruction is `VTESTPD`*.*

Syntax

```
extern int _mm256_testnzc_pd(__m256d s1, __m256d s2);
extern int _mm_testnzc_pd(__m128d s1, __m256d s2);
```

Arguments

s1	first source float64 vector
s2	second source float64 vector

Description

Performs a packed bit test of *s1* and *s2* vectors using `VTESTPD`*s1*, *s2* instruction and checks the status of the ZF and CF flags. The intrinsic returns 1 if both ZF and CF flags are not 1 (that is, both flags are not set), otherwise returns 0 (that is, one of the flags is set).

The `VTESTPD` instruction performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operand and corresponding sign bits in the second source operand. If the AND of the first source operand sign bits with the second source operand sign bits produces all zeros, the ZF flag is set else the ZF flag is clear. If the AND of the inverted first source operand sign bits with the second source operand sign bits produces all zeros the CF flag is set, else the CF flag is clear.

The `_mm_testnzc_pd` intrinsic checks the ZF and CF flags according to results of the 128-bit float64 source vectors. The `_m256_testnzc_pd` intrinsic checks the ZF and CF flags according to the results of the 256-bit float64 source vectors.

> **NOTE**
> Intel® Advanced Vector Extensions (Intel® AVX) instructions include a full compliment of 128-bit SIMD instructions. Such Intel® AVX instructions, with vector length of 128-bits, zeroes the upper 128 bits of the YMM register. The lower 128 bits of the YMM register is aliased to the corresponding SIMD XMM register.

Returns

1: indicates that both ZF and CF flags are clear

0: indicates that either ZF or CF flag is set

_mm256_testnzc_ps, _mm_testnzc_ps

Performs a packed bit test of two 256-bit or 128-bit float32 vectors to check ZF and CF flag settings. The corresponding Intel® AVX instruction is `VTESTPS`*.*

Syntax

```
extern int _mm256_testnzc_ps(__m256 s1, __m256 s2);
```

```
extern int _mm_testnzc_ps(__m128 s1, __m128 s2);
```

Arguments

s1	first source float32 vector
s2	second source float32 vector

Description

Performs a packed bit test of *s1* and *s2* vectors using `VTESTPD`*s1*, *s2* instruction and checks the status of the ZF and CF flags. The intrinsic returns 1 if both ZF and CF flags are not 1 (that is, both flags are not set), otherwise returns 0 (that is, one of the flags is set).

The `VTESTPD` instruction performs a bitwise comparison of all the sign bits of the single-precision elements in the first source operand and corresponding sign bits in the second source operand. If the AND of the first source operand sign bits with the second source operand sign bits produces all zeros, the ZF flag is set else the ZF flag is clear. If the AND of the inverted first source operand sign bits with the second source operand sign bits produces all zeros the CF flag is set, else the CF flag is clear.

The `_mm_testnzc_ps` intrinsic checks the ZF and CF flags according to results of the 128-bit float32 source vectors. The `_m256_testnzc_ps` intrinsic checks the ZF and CF flags according to the results of the 256-bit float32 source vectors.

NOTE
Intel® Advanced Vector Extensions (Intel® AVX) instructions include a full compliment of 128-bit SIMD instructions. Such Intel® AVX instructions, with vector length of 128-bits, zeroes the upper 128 bits of the YMM register. The lower 128 bits of the YMM register is aliased to the corresponding SIMD XMM register.

Returns

1: indicates that both ZF and CF flags are clear

0: indicates that either ZF or CF flag is set

Intrinsics for Permute Operations

_mm256_permute_pd, _mm_permute_pd

Permutes 256-bit or 128-bit float64 values into a 256-bit or 128-bit destination vector. The corresponding Intel® AVX instruction is `VPERMILPD`.

Syntax

```
extern __m256d _mm256_permute_pd(__m256d m1, int control);
```

```
extern __m128d _mm_permute_pd(__m128d m1, int control);
```

Arguments

m1	a 256-bit or 128-bit float64 vector
control	an integer specified as an 8-bit immediate; • for the 256-bit *m1* vector this integer contains four 1-bit control fields in the low 4 bits of the immediate • for the 128-bit *m1* vector this integer contains two 1-bit control fields in the low 2 bits of the immediate

Description

The `_mm256_permute_pd` intrinsic permutes double-precision floating point elements (float64 elements) in the 256-bit source vector *m1*, according to a specified 1-bit control field, *control*, and stores the result in a destination vector.

The `_mm_permute_pd` intrinsic permutes double-precision floating point elements (float64 elements) in the 128-bit source vector *m1*, according to a specified 1-bit control field, *control*, and stores the result in a destination vector.

Returns

A 256-bit or 128-bit float64 vector with permuted values.

_mm256_permute_ps, _mm_permute_ps

Permutes 256-bit or 128-bit float32 values into a 256-bit or 128-bit destination vector. The corresponding Intel® AVX instruction is `VPERMILPS`.

Syntax

```
extern __m256 _mm256_permute_ps(__m256 m1, int control);
```

```
extern __m128 _mm_permute_ps(__m128 m1, int control);
```

Arguments

m1	a 256-bit or 128-bit float32 vector
control	an integer specified as an 8-bit immediate; • for the 256-bit *m1* vector this integer contains four 2-bit control fields in the low 8 bits of the immediate • for the 128-bit *m1* vector this integer contains two 2-bit control fields in the low 4 bits of the immediate

Description

The `_mm256_permute_ps` intrinsic permutes single-precision floating point elements (float32 elements) in the 256-bit source vector *m1*, according to a specified 2-bit control field, *control*, and stores the result in a destination vector.

The `_mm_permute_ps` intrinsic permutes single-precision floating point elements (float32 elements) in the 128-bit source vector *m1*, according to a specified 2-bit control field, *control*, and stores the result in a destination vector.

Returns

A 256-bit or 128-bit float32 vector with permuted values.

_mm256_permutevar_pd, _mm_permutevar_pd

Permutes float64 values into a 256-bit or 128-bit destination vector. The corresponding Intel® AVX instruction is `VPERMILPD`.

Syntax

```
extern __m256d _mm256_permutevar_pd(__m256d m1, __m256i control);
```

```
extern __m128d _mm_permutevar_pd(__m128d m1, __m128i control);
```

Arguments

m1	a 256-bit or 128-bit float64 vector
control	a vector with 1-bit control fields, one for each corresponding element of the source vector

- for the 256-bit *m1* source vector this *control* vector contains four 1-bit control fields in the low 4 bits of the immediate
- for the 128-bit *m1* source vector this *control* vector contains two 1-bit control fields in the low 2 bits of the immediate

Description

Permutes double-precision floating-point values in the source vector, *m1*, according to the the 1-bit control fields in the low bytes of corresponding elements of a shuffle control. The result is stored in a destination vector.

Returns

A 256-bit or 128-bit float64 vector with permuted values.

_mm_permutevar_ps, _mm256_permutevar_ps

Permutes float32 values into a 256-bit or 128-bit destination vector. The corresponding Intel® AVX instruction is `VPERMILPS`*.*

Syntax

```
extern __m256 _mm256_permutevar_ps(__m256 m1, __m256i control);
```

```
extern __m128 _mm_permutevar_ps(__m128 m1, __m128i control);
```

Arguments

m1 — a 256-bit or 128-bit float32 vector

control — a vector with 2-bit control fields, one for each corresponding element of the source vector

- for the 256-bit *m1* source vector this *control* vector contains eight 2-bit control fields
- for the 128-bit *m1* source vector this *control* vector contains four 2-bit control fields

Description

Permutes single-precision floating-point values in the source vector, *m1*, according to the the 2-bit control fields in the low bytes of corresponding elements of a shuffle control. The result is stored in a destination vector.

Returns

A 256-bit or 128-bit float32 vector with permuted values.

_mm256_permute2f128_pd

Permutes 128-bit double-precision floating point containing fields into a 256-bit destination vector. The corresponding Intel® AVX instruction is `VPERM2F128`*.*

Syntax

```
extern __m256d _mm256_permute2f128_pd(__m256d m1, __m256d m2, int control);
```

Arguments

m1	a 256-bit float64 source vector
m2	a 256-bit float64 source vector
control	an immediate byte that specifies two 2-bit control fields and two additional bits which specify zeroing behavior.

Description

Permutes 128-bit floating-point-containing fields from the first source vector *m1* and second source vector *m2*, by using bits in the 8-bit *control* argument.

Returns

A 256-bit float64 vector with permuted values.

_mm256_permute2f128_ps

Permutes 128-bit single-precision floating point containing fields into a 256-bit destination vector. The corresponding Intel® AVX instruction is `VPERM2F128`.

Syntax

```
extern __m256 _mm256_permute2f128_ps(__m256 m1, __m256 m2, int control);
```

Arguments

m1	a 256-bit float32 source vector
m2	a 256-bit float32 source vector
control	an immediate byte that specifies two 2-bit control fields and two additional bits which specify zeroing behavior.

Description

Permutes 128-bit floating-point-containing fields from the first source vector *m1* and second source vector *m2*, by using bits in the 8-bit *control* argument.

Returns

A 256-bit float32 vector with permuted values.

_mm256_permute2f128_si256

Permutes 128-bit integer containing fields into a 256-bit destination vector. The corresponding Intel® AVX instruction is `VPERM2F128`.

Syntax

```
extern __m256i _mm256_permute2f128_si256(__m256i m1, __m256i m2, int control);
```

Arguments

m1	a 256-bit integer source vector
m2	a 256-bit integer source vector
control	an immediate byte that specifies two 2-bit control fields and two additional bits which specify zeroing behavior.

Description

Permutes 128-bit integer-containing fields from the first source vector *m1* and second source vector *m2*, by using bits in the 8-bit *control* argument.

Returns

A 256-bit integer vector with permuted values.

Intrinsics for Shuffle Operations

_mm256_shuffle_pd

Shuffles float64 vectors. The corresponding Intel® AVX instruction is `VSHUFPD`*.*

Syntax

```
extern __m256d _mm256_shuffle_pd(__m256d m1, __m256d m2, const int select);
```

Arguments

m1	float64 vector used for the operation
m2	float64 vector also used for the operation
select	a constant of integer type that determines which elements of the source vectors are moved to the result

Description

Moves or shuffles either of the two packed double-precision floating-point elements (float64 elements) from the double quadword in the source vectors to the low and high quadwords of the double quadword of the result.

The elements of the first source vector are moved to the low quadword while the elements of the second source vector are moved to the high quadword of the result. The constant defined by the *select* parameter determines which of the two elements of the source vectors are moved to the result.

Returns

Result of the shuffle operation.

_mm256_shuffle_ps

Shuffles float32 vectors. The corresponding Intel® AVX instruction is `VSHUFPS`*.*

Syntax

```
extern __m256 _mm256_shuffle_ps(__m256 m1, __m256 m2, const int select);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation
select	a constant of integer type which determines which elements of the source vectors move to the result

Description

Moves or shuffles two of the packed single-precision floating-point elements (float32 elements) from the double quadword in the source vectors to the low and high quadwords of the double quadword of the result.

The elements of the first source vector are moved to the low quadword while the elements of the second source vector are moved to the high quadword of the result. The constant defined by the *select* parameter determines which of the two elements of the source vectors are moved to the result.

Returns

Result of the shuffle operation.

Intrinsics for Unpack and Interleave Operations

_mm256_unpackhi_pd

Unpacks and interleaves high packed double-precision floating point values. The corresponding Intel® AVX instruction is `VUNPCKHPD`*.*

Syntax

```
extern __m256d _mm256_unpackhi_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 source vector
m2	float64 source vector

Description

Performs an interleaved unpack operation of high double-precision floating point values of the two source vectors *m1* and *m2*, and returns the result of the operation.

Returns

A vector with unpacked interleaved double-precision floating point values.

_mm256_unpackhi_ps

Unpacks and interleaves high packed single-precision floating point values. The corresponding Intel® AVX instruction is `VUNPCKHPS`*.*

Syntax

```
extern __m256 _mm256_unpackhi_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 source vector
m2	float32 source vector

Description

Performs an interleaved unpack operation of high single-precision floating point values of the two source vectors *m1* and *m2*, and returns the result of the operation.

Returns

A vector with unpacked interleaved single-precision floating point values.

_mm256_unpacklo_pd

Unpacks and interleaves low packed double-precision floating point values. The corresponding Intel® AVX instruction is `VUNPCKLPD`*.*

Syntax

```
extern __m256d _mm256_unpacklo_pd(__m256d m1, __m256d m2);
```

Arguments

m1	float64 source vector
m2	float64 source vector

Description

Performs an interleaved unpack operation of low double-precision floating point values of the two source vectors *m1* and *m2*, and returns the result of the operation.

Returns

A vector with unpacked interleaved double-precision floating point values.

_mm256_unpacklo_ps

Unpacks and interleaves low packed single-precision floating point values. The corresponding Intel® AVX instruction is `VUNPCKLPS`*.*

Syntax

```
extern __m256 _mm256_unpacklo_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 source vector
m2	float32 source vector

Description

Performs an interleaved unpack operation of low single-precision floating point values of the two source vectors *m1* and *m2*, and returns the result of the operation.

Returns

A vector with unpacked interleaved single-precision floating point values.

Support Intrinsics for Vector Typecasting Operations

_mm256_castpd_ps

Typecasts double-precision floating point values to single-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256 _mm256_castpd_ps(__m256d a);
```

Arguments

a	float64 source vector

Description

Performs a typecast operation from double-precision floating point values (float64 values) to single-precision floating point values (float32 values). This intrinsic does not introduce extra moves to the generated code. Source operand bits are passed unchanged to the result.

Returns

A vector with single-precision floating point values.

_mm256_castps_pd

Typecasts single-precision floating point values to double-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256d _mm256_castps_pd(__m256 a);
```

Arguments

a	float32 source vector

Description

Performs a typecast operation from single-precision floating point values (float32 values) to double-precision floating point values (float64 values). This intrinsic does not introduce extra moves to the generated code. Source operand bits are passed unchanged to the result.

Returns

A vector with double-precision floating point values.

_mm256_castpd_si256

Typecasts double-precision floating point values to integer values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_castpd_si256(__m256d a);
```

Arguments

a	float64 source vector

Description

Performs a typecast operation from double-precision floating point values (float64 values) to integer values. This intrinsic does not introduce extra moves to the generated code. Source operand bits are passed unchanged to the result.

Returns

A vector with 256-bit integer values.

_mm256_castps_si256

Typecasts single-precision floating point values to integer values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_castps_si256(__m256 a);
```

Arguments

a	float32 source vector

Description

Performs a typecast operation from single-precision floating point values (float32 values) to integer values. This intrinsic does not introduce extra moves to the generated code. Source operand bits are passed unchanged to the result.

Returns

A vector with 256-bit integer values.

_mm256_castsi256_pd

Typecasts 256-bit integer values to double-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256d _mm256_castsi256_pd(__m256i a);
```

Arguments

a	256-bit integer vector

Description

Performs a typecast operation from 256-bit integer values to double-precision floating point values (float64 values). This intrinsic does not introduce extra moves to the generated code. Source operand bits are passed unchanged to the result.

Returns

A vector with double-precision floating point values.

_mm256_castsi256_ps

Typecasts 256-bit integer values to single-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256 _mm256_castsi256_ps(__m256i a);
```

Arguments

a	256-bit integer source vector

Description

Performs a typecast operation from 256-bit integer values to single-precision floating point values (float32 values). This intrinsic does not introduce extra moves to the generated code. Source operand bits are passed unchanged to the result.

Returns

A vector with single-precision floating point values.

_mm256_castpd128_pd256

Typecasts 128-bit double-precision floating point values to 256-bit double-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256d _mm256_castpd128_pd256(__m128d a);
```

Arguments

a	128-bit float64 vector

Description

Performs a typecast operation from 128-bit double-precision floating point values to 256-bit double-precision floating point values.

The lower 128-bits of the 256-bit resulting vector contains the source vector values; the upper 128-bits of the resulting vector are undefined. This intrinsic does not introduce extra moves to the generated code.

Returns

A vector with 256-bit double-precision floating point values. The upper bits of the resulting vector are undefined.

_mm256_castpd256_pd128

Typecasts 256-bit double-precision floating point values to 128-bit double-precision floating point values. No corresponding Intel© AVX instruction.

Syntax

```
extern __m128d _mm256_castpd256_pd128(__m256d a);
```

Arguments

a	256-bit float64 source vector

Description

Performs a typecast operation from 256-bit double-precision floating point values to 128-bit double-precision floating point values.

The lower 128-bits of the source vector are passed unchanged to the result. This intrinsic does not introduce extra moves to the generated code.

Returns

A vector with 128-bit double-precision floating point values.

_mm256_castps128_ps256

Typecasts 128-bit single-precision floating point values to 256-bit single-precision floating point values. No corresponding Intel© AVX instruction.

Syntax

```
extern __m256 _mm256_castps128_ps256(__m128 a);
```

Arguments

a	128-bit float32 source vector

Description

Performs a typecast operation from 128-bit single-precision floating point values to 256-bit single-precision floating point values.

The lower 128-bits of the 256-bit resulting vector contains the source vector values; the upper 128-bits of the resulting vector are undefined. This intrinsic does not introduce extra moves to the generated code.

Returns

A vector with 256-bit single-precision floating point values. The upper bits of the resulting vector are undefined.

_mm256_castps256_ps128

Typecasts 256-bit single-precision floating point values to 128-bit single-precision floating point values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m128 _mm256_castps256_ps128(__m256 a);
```

Arguments

a	256-bit float32 source vector

Description

Performs a typecast operation from 256-bit single-precision floating point values to 128-bit single-precision floating point values.

The lower 128-bits of the source vector are passed unchanged to the result. This intrinsic does not introduce extra moves to the generated code.

Returns

A vector with 128-bit single-precision floating point values.

_mm256_castsi128_si256

Typecasts 128-bit integer values to 256-bit integer values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_castsi128_si256(__m128i a);
```

Arguments

a	128-bit integer source vector

Description

Performs a typecast operation from 128-bit integer values to 256-bit integer values.

The lower 128-bits of the 256-bit resulting vector contains the source vector values; the upper 128-bits of the resulting vector are undefined. This intrinsic does not introduce extra moves to the generated code.

Returns

A vector with 256-bit integer values. The upper bits of the resulting vector are undefined.

_mm256_castsi256_si128

Typecasts 256-bit integer values to 128-bit integer values. No corresponding Intel® AVX instruction.

Syntax

```
extern __m128i _mm256_castsi256_si128(__m256i a);
```

Arguments

a	256-bit integer source vector

Description

Performs a typecast operation from 256-bit integer values to 128-bit integer values.

The lower 128-bits of the source vector are passed unchanged to the result. This intrinsic does not introduce extra moves to the generated code.

Returns

A vector with 128-bit integer values.

Intrinsics Generating Vectors of Undefined Values

_mm256_undefined_ps()

Returns a vector of eight single precision floating point elements. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256 _mm256_undefined_ps(void);
```

Description

This intrinsic returns a vector of eight single precision floating point elements. The content of the vector is not specified. The prototype of this intrinsic is in the `immintrin.h` header file.

Returns

A vector of eight single precision floating point elements.

See Also

Intrinsics Returning Vectors of Undefined Values

_mm256_undefined_pd()

Returns a vector of four double precision floating point elements. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256d _mm256_undefined_pd(void);
```

Description

This intrinsic returns a vector of four double precision floating point elements. The content of the vector is not specified. The prototype of this intrinsic is in the `immintrin`.h header file.

Returns

A vector of four double precision floating point elements.

See Also

Intrinsics Returning Vectors of Undefined Values

_mm256_undefined_si256

Returns a vector of eight packed doubleword integer elements. No corresponding Intel® AVX instruction.

Syntax

```
extern __m256i _mm256_undefined_si256(void);
```

Description

This intrinsic returns a vector of eight packed doubleword integer elements. The content of the vector is not specified. The prototype of this intrinsic is in the `immintrin.h` header file.

Returns

A vector of eight packed doubleword integer elements.

See Also

Intrinsics Returning Vectors of Undefined Values

Intrinsics for Intel® Streaming SIMD Extensions 4 (Intel® SSE4)

The intrinsics in this section correspond to Intel® Streaming SIMD Extensions 4 (Intel® SSE4) instructions.

Efficient Accelerated String and Text Processing

Overview: Efficient Accelerated String and Text Processing

The intrinsics in this section correspond to Intel® Streaming SIMD Extensions 4 (Intel® SSE4) Efficient Accelerated String and Text Processing instructions.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Packed Compare Intrinsics

These Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsics perform packed comparisons. Some of these intrinsics could map to more than one instruction; the Intel® C++ Compiler selects the instruction to generate.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® SSE4 Instruction
`_mm_cmpestri`	Packed comparison, generates index	`PCMPESTRI`
`_mm_cmpestrm`	Packed comparison, generates mask	`PCMPESTRM`

Intrinsic Name	Operation	Corresponding Intel® SSE4 Instruction
`_mm_cmpistri`	Packed comparison, generates index	`PCMPISTRI`
`_mm_cmpistrm`	Packed comparison, generates mask	`PCMPISTRM`
`_mm_cmpestrz`	Packed comparison	`PCMPESTRM` or `PCMPESTRI`
`_mm_cmpestrc`	Packed comparison	`PCMPESTRM` or `PCMPESTRI`
`_mm_cmpestrs`	Packed comparison	`PCMPESTRM` or `PCMPESTRI`
`_mm_cmpestro`	Packed comparison	`PCMPESTRM` or `PCMPESTRI`
`_mm_cmpestra`	Packed comparison	`PCMPESTRM` or `PCMPESTRI`
`_mm_cmpistrz`	Packed comparison	`PCMPISTRM` or`PCMPISTRI`
`_mm_cmpistrc`	Packed comparison	`PCMPISTRM` or `PCMPISTRI`
`_mm_cmpistrs`	Packed comparison	`PCMPISTRM` or `PCMPISTRI`
`_mm_cmpistro`	Packed comparison	`PCMPISTRM` or `PCMPISTRI`
`_mm_cmpistra`	Packed comparison	`PCMPISTRM` or `PCMPISTRI`

_mm_cmpestri

```
int _mm_cmpestri(__m128i src1, int len1, __m128i src2, int len2, const int mode);
```

Performs a packed comparison of string data with explicit lengths, generating an index and storing the result in `ECX`.

_mm_cmpestrm

```
__m128i _mm_cmpestrm(__m128i src1, int len1, __m128i src2, int len2, const int mode);
```

Performs a packed comparison of string data with explicit lengths, generating a mask and storing the result in `XMM0`.

_mm_cmpistri

```
int _mm_cmpistri(__m128i src1, __m128i src2, const int mode);
```

Performs a packed comparison of string data with implicit lengths, generating an index and storing the result in `ECX`.

_mm_cmpistrm

```
__m128i _mm_cmpistrm(__m128i src1, __m128i src2, const int mode);
```

Performs a packed comparison of string data with implicit lengths, generating a mask and storing the result in `XMM0`.

_mm_cmpestrz

`int _mm_cmpestrz(__m128i src1, int len1, __m128i src2, int len2, const int mode);`

Performs a packed comparison of string data with explicit lengths. Returns '1' if `ZFlag == 1`, otherwise '0'.

_mm_cmpestrc

`int _mm_cmpestrc(__m128i src1, int len1, __m128i src2, int len2, const int mode);`

Performs a packed comparison of string data with explicit lengths. Returns '1' if `CFlag == 1`, otherwise '0'.

_mm_cmpestrs

`int _mm_cmpestrs(__m128i src1, int len1, __m128i src2, int len2, const int mode);`

Performs a packed comparison of string data with explicit lengths. Returns '1' if `SFlag == 1`, otherwise '0'.

_mm_cmpestro

`int _mm_cmpestro(__m128i src1, int len1, __m128i src2, int len2, const int mode);`

Performs a packed comparison of string data with explicit lengths. Returns '1' if `OFlag == 1`, otherwise '0'.

_mm_cmpestra

`int _mm_cmpestra(__m128i src1, int len1, __m128i src2, int len2, const int mode);`

Performs a packed comparison of string data with explicit lengths. Returns '1' if `CFlag == 0` and `ZFlag == 0`, otherwise '0'.

_mm_cmpistrz

`int _mm_cmpistrz(__m128i src1, __m128i src2, const int mode);`

Performs a packed comparison of string data with implicit lengths. Returns '1' if `(ZFlag == 1)`, otherwise '0'.

_mm_cmpistrc

`int _mm_cmpistrc(__m128i src1, __m128i src2, const int mode);`

Performs a packed comparison of string data with implicit lengths. Returns '1' if `(CFlag == 1)`, otherwise '0'.

_mm_cmpistrs

`int _mm_cmpistrs(__m128i src1, __m128i src2, const int mode);`

Performs a packed comparison of string data with implicit lengths. Returns '1' if `(SFlag == 1)`, otherwise '0'.

_mm_cmpistro

`int _mm_cmpistro(__m128i src1, __m128i src2, const int mode);`

Performs a packed comparison of string data with implicit lengths. Returns '1' if `(OFlag == 1)`, otherwise '0'.

_mm_cmpistra

`int _mm_cmpistra(__m128i src1, __m128i src2, const int mode);`

Performs a packed comparison of string data with implicit lengths. Returns '1' if `(ZFlag == 0 and CFlag == 0)`, otherwise '0'.

Application Targeted Accelerators Intrinsics

These Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsics extend the capabilities of Intel® architectures by adding performance-optimized, low-latency, lower power fixed-function accelerators on the processor die to benefit specific applications.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsics marked with * are implemented only on Intel® 64 architecture. The rest of the intrinsics are implemented on both IA-32 and Intel® 64 architectures.

Intrinsic Name	Operation	Corresponding Intel® SSE4 Instruction
`_mm_popcnt_u32`	Counts number of set bits in a data operation	`POPCNT`
`_mm_popcnt_u64*`	Counts number of set bits in a data operation	`POPCNT`
`_mm_crc32_u8`	Accumulates cyclic redundancy check	`CRC32`
`_mm_crc32_u16`	Performs cyclic redundancy check	`CRC32`
`_mm_crc32_u32`	Performs cyclic redundancy check	`CRC32`
`_mm_crc32_u64*`	Performs cyclic redundancy check	`CRC32`

_mm_popcnt_u32

`unsigned int _mm_popcnt_u32(unsigned int v);`

Counts the number of set bits in a data operation.

_mm_popcnt_u64

`__int64 _mm_popcnt_u64(unsigned __int64 v);`

Counts the number of set bits in a data operation.

NOTE
Use only on Intel® 64 architecture.

_mm_crc32_u8

`unsigned int _mm_crc32_u8(unsigned int crc, unsigned char v);`

Starting with initial value in the first operand, accumulates CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m8.

_mm_crc32_u16

```
unsigned int _mm_crc32_u16(unsigned int crc, unsigned short v);
```

Starting with initial value in the first operand, accumulates CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m16.

_mm_crc32_u32

```
unsigned int _mm_crc32_u32(unsigned int crc, unsigned int v);
```

Starting with initial value in the first operand, accumulates CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m32.

_mm_crc32_u64

```
unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64 v);
```

Starting with initial value in the first operand, accumulates CRC32 value for the second operand and stores the result in the destination operand. Accumulates CRC32 on r/m64.

NOTE
Use only on Intel® 64 architecture.

Vectorizing Compiler and Media Accelerators

Overview: Vectorizing Compiler and Media Accelerators

The intrinsics in this section correspond to Intel® Streaming SIMD Extensions 4 (Intel® SSE4) Vectorizing Compiler and Media Accelerators instructions.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Packed Blending Intrinsics

These Intel® Streaming SIMD Extensions 4 (Intel® SSE4) intrinsics pack multiple operations in a single instruction. Blending conditionally copies one field in the source onto the corresponding field in the destination. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction
`__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask);`	Selects single precision float data from two sources using constant mask	BLENDPS

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction
`__m128d _mm_blend_pd(__m128d v1, __m128d v2, const int mask)`	Selects double precision float data from two sources using constant mask	BLENDPD
`__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)`	Selects single precision float data from two sources using variable mask	BLENDVPS
`__m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3)`	Selects double precision float data from two sources using variable mask	BLENDVPD
`__m128i _mm_blendv_epi8(__m128i v1, __m128i v2, __m128i mask)`	Selects integer bytes from two sources using variable mask	PBLENDVB
`__m128i _mm_blend_epi16(__m128i v1, __m128i v2, const int mask)`	Selects integer words from two sources using constant mask	PBLENDW

Floating Point Dot Product Intrinsics

These Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsics enable floating point single-precision and double-precision dot products. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic	Operation	Corresponding Intel® SSE4 Instruction
`_mm_dp_pd`	Double-precision dot product	DPPD
`_mm_dp_ps`	Single-precision dot product	DPPS

_mm_dp_pd

```
__m128d _mm_dp_pd( __m128d a, __m128d b, const int mask);
```

Calculates the dot product of double-precision packed values with mask-defined summing and zeroing of the parts of the result.

_mm_dp_ps

```
__m128 _mm_dp_ps( __m128 a, __m128 b, const int mask);
```

Calculates the dot product of single-precision packed values with mask-defined summing and zeroing of the parts of the result.

Packed Format Conversion Intrinsics

These Intel® Streaming SIMD Extensions 4 (Intel® SSE4) intrinsics convert a packed integer to a zero-extended or sign-extended integer with wider type. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction
`__m128i _mm_cvtepi8_epi32(__m128i a)`	Sign extend four bytes into four doublewords	`PMOVSXBD`
`__m128i _mm_cvtepi8_epi64 (__m128i a)`	Sign extend two bytes into two quadwords	`PMOVSXBQ`
`__m128i _mm_cvtepi8_epi16(__m128i a)`	Sign extend eight bytes into eight words	`PMOVSXBW`
`__m128i _mm_cvtepi32_epi64(__m128i a)`	Sign extend two doublewords into two quadwords	`PMOVSXDQ`
`__m128i _mm_cvtepi16_epi32(__m128i a)`	Sign extend four words into four doublewords	`PMOVSXWD`
`__m128i _mm_cvtepi16_epi64(__m128i a)`	Sign extend two words into two quadwords	`PMOVSXWQ`
`__m128i _mm_cvtepu8_epi32(__m128i a)`	Zero extend four bytes into four doublewords	`PMOVZXBD`
`__m128i _mm_cvtepu8_epi64(__m128i a)`	Zero extend two bytes into two quadwords	`PMOVZXBQ`
`__m128i _mm_cvtepu8_epi16(__m128i a)`	Zero extend eight bytes into eight words	`PMOVZXBW`
`__m128i _mm_cvtepu32_epi64(__m128i a)`	Zero extend two doublewords into two quadwords	`PMOVZXDQ`
`__m128i _mm_cvtepu16_epi32(__m128i a)`	Zero extend four words into four doublewords	`PMOVZXWD`
`__m128i _mm_cvtepu16_epi64(__m128i a)`	Zero extend two words into two quadwords	`PMOVZXWQ`

Packed Integer Min/Max Intrinsics

These Intel® Streaming SIMD Extensions 4 (Intel® SSE4) intrinsics compare packed integers in the destination operand and the source operand, and return the minimum or maximum for each packed operand in the destination operand. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction
`__m128i _mm_max_epi8( __m128i a, __m128i b)`	Calculates maximum of signed packed integer bytes	`PMAXSB`
`__m128i _mm_max_epi32( __m128i a, __m128i b)`	Calculates maximum of signed packed integer doublewords	`PMAXSD`
`__m128i _mm_max_epu32( __m128i a, __m128i b)`	Calculates maximum of unsigned packed integer doublewords	`PMAXUD`
`__m128i _mm_max_epu16( __m128i a, __m128i b)`	Calculates maximum of unsigned packed integer words	`PMAXUW`
`__m128i _mm_min_epi8( __m128i a, __m128i b)`	Calculates minimum of signed packed integer bytes	`PMINSB`
`__m128i _mm_min_epi32( __m128i a, __m128i b)`	Calculates minimum of signed packed integer doublewords	`PMINSD`
`__m128i _mm_min_epu32( __m128i a, __m128i b)`	Calculates minimum of unsigned packed integer double words	`PMINUD`
`__m128i _mm_min_epu16( __m128i a, __m128i b)`	Calculates minimum of unsigned packed integer words	`PMINUW`

Floating Point Rounding Intrinsics

These Intel® Streaming SIMD Extensions 4 (Intel® SSE4) rounding intrinsics cover scalar and packed single-precision and double-precision floating-point operands. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The `floor` and `ceil` intrinsics correspond to the definitions of `floor` and `ceil` in the ISO 9899:1999 standard for the C programming language.

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction

`__m128d _mm_round_pd(__m128d s1, int iRoundMode)`
`__m128d _mm_floor_pd(__m128d s1)`
`__m128d _mm_ceil_pd(__m128d s1)`
	Packed double-precision float rounding	`ROUNDPD`

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction

`__m128 _mm_round_ps(__m128 s1, int iRoundMode)`
`__m128 _mm_floor_ps(__m128 s1)`
`__m128 _mm_ceil_ps(__m128 s1)`
	Packed single-precision float rounding	`ROUNDPS`

`__m128d _mm_round_sd(__m128d dst, __m128d s1, int iRoundMode)`
`__m128d _mm_floor_sd(__m128d dst, __m128d s1)`
`__m128d _mm_ceil_sd(__m128d dst, __m128d s1)`
	Single double-precision float rounding	`ROUNDSD`

`__m128 _mm_round_ss(__m128 dst, __m128d s1, int iRoundMode)`
`__m128 _mm_floor_ss(__m128d dst, __m128 s1)`
`__m128 _mm_ceil_ss(__m128d dst, __m128 s1)`
	Single single-precision float rounding	`ROUNDSS`

DWORD Multiply Intrinsics

These Intel® Streaming SIMD Extensions (Intel® SSE4) DWORD multiply intrinsics are designed to aid vectorization. They enable four simultaneous 32-bit by 32-bit multiplies. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction
`__m128i _mm_mul_epi32( __m128i a, __m128i b)`	Packed integer 32-bit multiplication of two low pairs of operands producing two 64-bit results	`PMULDQ`
`__m128i _mm_mullo_epi32( __m128i a, __m128i b)`	Packed integer 32-bit multiplication with truncation of upper halves of results	`PMULLD`

Register Insertion/Extraction Intrinsics

These Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsics enable data insertion and extraction between general purpose registers and `XMM` registers. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsics marked with * are implemented only on Intel® 64 architectures. The rest of the intrinsics are implemented on both IA-32 and Intel® 64 architectures.

Intrinsic Syntax	Operation	Corresponding Intel® SSE4 Instruction
`__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)`	Insert single precision float into packed single precision array element selected by index.	`INSERTPS`
`int _mm_extract_ps(__m128 src, const int ndx)`	Extract single precision float from packed single precision array element selected by index.	`EXTRACTPS`
`__m128i _mm_insert_epi8(__m128i s1, int s2, const int ndx)`	Insert integer byte into packed integer array element selected by index.	`PINSRB`
`int _mm_extract_epi8(__m128i src, const int ndx)`	Extract integer byte from packed integer array element selected by index.	`PEXTRB`
`int _mm_extract_epi16(__m128i src, int ndx)`	Extract integer word from packed integer array element selected by index.	`PEXTRW`
`__m128i _mm_insert_epi32(__m128i s1, int s2, const int ndx)`	Insert integer doubleword into packed integer array element selected by index.	`PINSRD`
`int _mm_extract_epi32(__m128i src, const int ndx)`	Extract integer doubleword from packed integer array element selected by index.	`PEXTRD`
`__m128i _mm_insert_epi64(__m128i s2, int s, const int ndx)`	Insert integer quadword into packed integer array element selected by index. Use only on Intel® 64 architectures.	`PINSRQ`
`__int64 _mm_extract_epi64(__m128i src, const int ndx)`	Extract integer quad word from packed integer array element selected by index. Use only on Intel® 64 architectures.	`PEXTRQ`

Test Intrinsics

These Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsics perform packed integer 128-bit comparisons. The prototypes for these intrinsics are in the `smmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® SSE4 Instruction
`_mm_testz_si128`	Checks for all zeros in specified bits of a 128-bit value	`PTEST`
`_mm_testc_si128`	Checks for all ones in specified bits of a 128-bit value	`PTEST`

Intrinsic Name	Operation	Corresponding Intel® SSE4 Instruction
`_mm_testnzc_si128`	Checks for at least one '0' and at least one '1' in the specified bits of a 128-bit value	`PTEST`

_mm_testz_si128

```
int _mm_testz_si128(__m128i s1, __m128i s2);
```

Returns '1' if the bitwise AND operation on `s1` and `s2` results in all zeros, else returns '0'. That is,

```
_mm_testz_si128 := ( (s1 & s2) == 0 ? 1 : 0 )
```

This intrinsic checks if the `ZF` flag equals '1' as a result of the instruction `PTEST` *`s1`*, *`s2`*. For example, it allows you to check if all set bits in *s2* (mask) are zeros in *s1*.

Corresponding instruction: `PTEST`

_mm_testc_si128

```
int _mm_testc_si128(__m128i s1, __m128i s2);
```

Returns '1' if the bitwise AND operation on *s2* and logical NOT *s1* results in all zeros, else returns '0'. That is,

```
_mm_testc_si128 := ( (~s1 & s2) == 0 ? 1 : 0 )
```

This intrinsic checks if the `CF` flag equals '1' as a result of the instruction `PTEST` *`s1`*, *`s2`*. For example it allows you to check if all set bits in *s2* (mask) are also set in *s1*.

Corresponding instruction: `PTEST`

_mm_testnzc_si128

```
int _mm_testnzc_si128(__m128i s1, __m128i s2);
```

Returns '1' if the following conditions are true: bitwise operation of *s1* AND *s2* does not equal all zeros and bitwise operation of NOT *s1* AND *s2* does not equal all zeros, otherwise returns '0'. That is,

```
_mm_testnzc_si128 := ( ( (s1 & s2) != 0 && (~s1 & s2) != 0 ) ? 1 : 0 )
```

This intrinsic checks if both the `CF` and `ZF` flags are not '1' as a result of the instruction `PTEST` *`s1`*, *`s2`*. For example, it allows you to check that the result has both zeros and ones in *s1* on positions specified as set bits in *s2* (mask).

Corresponding instruction: `PTEST`

Packed DWORD to Unsigned WORD Intrinsic

The prototype for this Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsic is in the `smmintrin.h` file.

To use this intrinsic, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_packus_epi32

```
__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
```

Converts eight packed signed doublewords into eight packed unsigned words, using unsigned saturation to handle overflow condition.

Corresponding instruction: `PACKUSDW`

Packed Compare for Equal Intrinsic

The prototype for this Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsic is in the `smmintrin.h` file.

To use this intrinsic, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_cmpeq_epi64

```
__m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
```

Performs a packed integer 64-bit comparison for equality. The intrinsic fills the corresponding parts of the result with zeroes or ones based on equality.

Corresponding instruction: `PCMPEQQ`

Cacheability Support Intrinsic

The prototype for this Intel® Streaming SIMD Extensions (Intel® SSE4) intrinsic is in the `smmintrin.h` file.

To use this intrinsic, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_stream_load_si128

```
extern __m128i _mm_stream_load_si128(__m128i* v1);
```

Loads `__m128` data from a 16-byte aligned address, *v1*, to the destination operand, `m128i` without polluting the caches.

Corresponding instruction: `MOVNTDQA`

Intrinsics for Intel® Supplemental Streaming SIMD Extensions 3 (SSSE3)

Intel® C++ intrinsics listed in this section correspond to the Supplemental Streaming SIMD Extensions 3 (SSSE3) instructions. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The topics in this section summarize these intrinsics.

Addition Intrinsics

These Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics are used for horizontal addition. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_hadd_epi16

```
extern __m128i _mm_hadd_epi16(__m128i a, __m128i b);
```

Adds horizontally packed signed words. Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
r[i] = a[2*i] + a[2i+1];
r[i+4] = b[2*i] + b[2*i+1];
}
```

_mm_hadd_epi32

`extern __m128i _mm_hadd_epi32(__m128i a, __m128i b);`

Adds horizontally packed signed doublewords. Interpreting *a*, *b*, and *r* as arrays of 32-bit signed integers:

```
for (i = 0; i < 2; i++) {
r[i] = a[2*i] + a[2i+1];
r[i+2] = b[2*i] + b[2*i+1];
}
```

_mm_hadds_epi16

`extern __m128i _mm_hadds_epi16(__m128i a, __m128i b);`

Adds horizontally packed signed words with signed saturation. Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
r[i] = signed_saturate_to_word(a[2*i] + a[2i+1]);
r[i+4] = signed_saturate_to_word(b[2*i] + b[2*i+1]);
}
```

_mm_hadd_pi16

`extern __m64 _mm_hadd_pi16(__m64 a, __m64 b);`

Adds horizontally packed signed words. Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
r[i] = a[2*i] + a[2i+1];
r[i+2] = b[2*i] + b[2*i+1];
}
```

_mm_hadd_pi32

`extern __m64 _mm_hadd_pi32(__m64 a, __m64 b);`

Adds horizontally packed signed doublewords. Interpreting *a*, *b*, and *r* as arrays of 32-bit signed integers:

```
r[0] = a[1] + a[0];
r[1] = b[1] + b[0];
```

_mm_hadds_pi16

`extern __m64 _mm_hadds_pi16(__m64 a, __m64 b);`

Adds horizontally packed signed words with signed saturation. Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
r[i] = signed_saturate_to_word(a[2*i] + a[2i+1]);
r[i+2] = signed_saturate_to_word(b[2*i] + b[2*i+1]);
}
```

Subtraction Intrinsics

These Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics are used for horizontal subtraction. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_hsub_epi16

`extern __m128i _mm_hsub_epi16(__m128i a, __m128i b);`

Subtract horizontally packed signed words.

Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
r[i] = a[2*i] - a[2i+1];
r[i+4] = b[2*i] - b[2*i+1];
}
```

_mm_hsub_epi32

`extern __m128i _mm_hsub_epi32(__m128i a, __m128i b);`

Subtracts horizontally packed signed doublewords.

Interpreting *a*, *b*, and *r* as arrays of 32-bit signed integers:

```
for (i = 0; i < 2; i++) {
r[i] = a[2*i] - a[2i+1];
r[i+2] = b[2*i] - b[2*i+1];
}
```

_mm_hsubs_epi16

`extern __m128i _mm_hsubs_epi16(__m128i a, __m128i b);`

Subtracts horizontally packed signed words with signed saturation.

Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
r[i] = signed_saturate_to_word(a[2*i] - a[2i+1]);
r[i+4] = signed_saturate_to_word(b[2*i] - b[2*i+1]);
}
```

_mm_hsub_pi16

`extern __m64 _mm_hsub_pi16(__m64 a, __m64 b);`

Subtracts horizontally packed signed words.

Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
r[i] = a[2*i] - a[2i+1];
r[i+2] = b[2*i] - b[2*i+1];
}
```

_mm_hsub_pi32

```
extern __m64 _mm_hsub_pi32(__m64 a, __m64 b);
```

Subtracts horizontally packed signed doublewords.

Interpreting *a*, *b*, and *r* as arrays of 32-bit signed integers:

```
r[0] = a[0] - a[1];
r[1] = b[0] - b[1];
```

_mm_hsubs_pi16

```
extern __m64 _mm_hsubs_pi16(__m64 a, __m64 b);
```

Subtracts horizontally packed signed words with signed saturation.

Interpreting *a*, *b*, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 2; i++) {
r[i] = signed_saturate_to_word(a[2*i] - a[2i+1]);
r[i+2] = signed_saturate_to_word(b[2*i] - b[2*i+1]);
}
```

Multiplication Intrinsics

These Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics are used for multiplication. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_maddubs_epi16

```
extern __m128i _mm_maddubs_epi16(__m128i a, __m128i b);
```

Multiplies signed and unsigned bytes, adds horizontal pair of signed words, and packs saturated signed words.

Interpreting *a* as array of unsigned 8-bit integers, *b* as arrays of signed 8-bit integers, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 8; i++) {
r[i] = signed_saturate_to_word(a[2*i+1] * b[2*i+1] + a[2*i]*b[2*i]);
}
```

_mm_maddubs_pi16

```
extern __m64 _mm_maddubs_pi16(__m64 a, __m64 b);
```

Multiplies signed and unsigned bytes, adds horizontal pair of signed words, and packs saturated signed words.

Interpreting *a* as array of unsigned 8-bit integers, *b* as arrays of signed 8-bit integers, and *r* as arrays of 16-bit signed integers:

```
for (i = 0; i < 4; i++) {
r[i] = signed_saturate_to_word(a[2*i+1] * b[2*i+1] + a[2*i]*b[2*i]);
}
```

_mm_mulhrs_epi16

`extern __m128i _mm_mulhrs_epi16(__m128i a, __m128i b);`

Multiplies signed words, scales and rounds signed doublewords, and packs high 16-bits.

Interpreting *a*, *b*, and *r* as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++) {
r[i] =  (( (int32)((a[i] * b[i]) >> 14) + 1) >> 1) & 0xFFFF;
}
```

_mm_mulhrs_pi16

`extern __m64 _mm_mulhrs_pi16(__m64 a, __m64 b);`

Multiplies signed words, scales and rounds signed doublewords, and packs high 16-bits.

Interpreting *a*, *b*, and *r* as arrays of signed 16-bit integers:

```
for (i = 0; i < 4; i++) {
r[i] =  (( (int32)((a[i] * b[i]) >> 14) + 1) >> 1) & 0xFFFF;
}
```

Absolute Value Intrinsics

These Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics are used to compute absolute values. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_abs_epi8

`extern __m128i _mm_abs_epi8(__m128i a);`

Computes absolute value of signed bytes. Interpreting *a* and *r* as arrays of signed 8-bit integers:

```
for (i = 0; i < 16; i++) {
r[i] = abs(a[i]);
}
```

_mm_abs_epi16

`extern __m128i _mm_abs_epi16(__m128i a);`

Computes absolute value of signed words. Interpreting *a* and *r* as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++) {
r[i] = abs(a[i]);
}
```

_mm_abs_epi32

`extern __m128i _mm_abs_epi32(__m128i a);`

Computes absolute value of signed doublewords. Interpreting *a* and *r* as arrays of signed 32-bit integers:

```
for (i = 0; i < 4; i++) {
r[i] = abs(a[i]);
}
```

_mm_abs_pi8

`extern __m64 _mm_abs_pi8(__m64 a);`

Computes absolute value of signed bytes. Interpreting *a* and *r* as arrays of signed 8-bit integers:

```
for (i = 0; i < 8; i++) {
r[i] = abs(a[i]);
}
```

_mm_abs_pi16

`extern __m64 _mm_abs_pi16(__m64 a);`

Computes absolute value of signed words. Interpreting *a* and *r* as arrays of signed 16-bit integers:

```
for (i = 0; i < 4; i++) {
r[i] = abs(a[i]);
}
```

_mm_abs_pi32

`extern __m64 _mm_abs_pi32(__m64 a);`

Computes absolute value of signed doublewords. Interpreting *a* and *r* as arrays of signed 32-bit integers:

```
for (i = 0; i < 2; i++) {
r[i] = abs(a[i]);
}
```

Shuffle Intrinsics

These Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics are used to perform shuffle operations. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_shuffle_epi8

`extern __m128i _mm_shuffle_epi8(__m128i a, __m128i b);`

Shuffle bytes from *a* according to contents of *b*.

Interpreting *a*, *b*, and *r* as arrays of unsigned 8-bit integers:

```
for (i = 0; i < 16; i++){
 if (b[i] & 0x80){
  r[i] =  0;
 }
 else {
  r[i] = a[b[i] & 0x0F];
 }
}
```

_mm_shuffle_pi8

`extern __m64 _mm_shuffle_pi8(__m64 a, __m64 b);`

Shuffle bytes from *a* according to contents of *b*.

Interpreting *a*, *b*, and *r* as arrays of unsigned 8-bit integers:

```
for (i = 0; i < 8; i++){
 if (b[i] & 0x80){
  r[i] =  0;
 }
else {
 r[i] = a[b[i] & 0x07];
 }
}
```

Concatenate Intrinsics

These Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics are used concatenation. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_alignr_epi8

`extern __m128i _mm_alignr_epi8(__m128i a, __m128i b, int n);`

Concatenates *a* and *b*, extracts byte-aligned result shifted to the right by *n*.

Interpreting `t1` as 256-bit unsigned integer, *a*, *b*, and *r* as 128-bit unsigned integers:

```
t1[255:128] = a;
t1[127:0] = b;
t1[255:0] = t1[255:0] >> (8 * n); // unsigned shift
r[127:0] = t1[127:0];
```

_mm_alignr_pi8

`extern __m64 _mm_alignr_pi8(__m64 a, __m64 b, int n);`

Concatenates *a* and *b*, extracts byte-aligned result shifted to the right by *n*.

Interpreting `t1` as 128-bit unsigned integer, *a*, *b*, and *r* as 64-bit unsigned integers:

```
t1[127:64] = a;
t1[63:0] = b;
t1[127:0] = t1[127:0] >> (8 * n); // unsigned shift
r[63:0] = t1[63:0];
```

Negation Intrinsics

These Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics are used for negation. The prototypes for these intrinsics are in `tmmintrin.h`.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_sign_epi8

`extern __m128i _mm_sign_epi8(__m128i a, __m128i b);`

Negates packed bytes in *a* if corresponding sign in *b* is less than zero.

Interpreting *a*, *b*, and *r* as arrays of signed 8-bit integers:

```
for (i = 0; i < 16; i++){
 if (b[i] < 0){
  r[i] =  -a[i];
 }
 else
 if (b[i] == 0){
  r[i] = 0;
 }
 else {
  r[i] = a[i];
 }
}
```

_mm_sign_epi16

`extern __m128i _mm_sign_epi16(__m128i a, __m128i b);`

Negates packed words in *a* if corresponding sign in *b* is less than zero.

Interpreting *a*, *b*, and *r* as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++){
 if (b[i] < 0){
  r[i] =  -a[i];
 }
else
 if (b[i] == 0){
  r[i] = 0;
 }
 else
 {
  r[i] = a[i];
 }
}
```

_mm_sign_epi32

`extern __m128i _mm_sign_epi32(__m128i a, __m128i b);`

Negates packed doublewords in *a* if corresponding sign in *b* is less than zero.

Interpreting *a*, *b*, and *r* as arrays of signed 32-bit integers:

```
for (i = 0; i < 4; i++){
 if (b[i] < 0){
  r[i] = -a[i];
 }
 else
 if (b[i] == 0){
  r[i] = 0;
 }
 else {
  r[i] = a[i];
 }
}
```

_mm_sign_pi8

```
extern __m64 _mm_sign_pi8(__m64 a, __m64 b);
```

Negates packed bytes in *a* if corresponding sign in *b* is less than zero.

Interpreting *a*, *b*, and *r* as arrays of signed 8-bit integers:

```
for (i = 0; i < 16; i++){
 if (b[i] < 0){
  r[i] =  -a[i];
 }
 else
 if (b[i] == 0){
  r[i] = 0;
 }
 else {
  r[i] = a[i];
 }
}
```

_mm_sign_pi16

```
extern __m64 _mm_sign_pi16(__m64 a, __m64 b);
```

Negates packed words in *a* if corresponding sign in *b* is less than zero.

Interpreting *a*, *b*, and *r* as arrays of signed 16-bit integers:

```
for (i = 0; i < 8; i++){
 if (b[i] < 0){
  r[i] =  -a[i];
  }
 else
 if (b[i] == 0){
  r[i] = 0;
  }
 else {
  r[i] = a[i];
  }
}
```

_mm_sign_pi32

```
extern __m64 _mm_sign_pi32(__m64 a, __m64 b);
```

Negates packed doublewords in *a* if corresponding sign in *b* is less than zero.

Interpreting *a*, *b*, and *r* as arrays of signed 32-bit integers:

```
for (i = 0; i < 2; i++){
 if (b[i] < 0){
  r[i] =  -a[i];
 }
 else
 if (b[i] == 0){
  r[i] = 0;
 }
 else {
  r[i] = a[i];
 }
}
```

Intrinsics for Intel® Streaming SIMD Extensions 3 (Intel® SSE3)

The Intel® C++ intrinsics listed in this section are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (Intel® SSE3). The prototypes for these intrinsics are in the `pmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The topics in this section summarize these intrinsics.

Integer Vector Intrinsic

The integer vector intrinsic listed here is designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (Intel® SSE3). The prototype for this intrinsic is in the `pmmintrin.h` header file.

To use this intrinsic, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

`R` represents the register into which the returns are placed.

_mm_lddqu_si128

`__m128i _mm_lddqu_si128(__m128i const *p);`

Loads an unaligned 128-bit value. This differs from `MOVDQU` in that it can provide higher performance in some cases. However, it also may provide lower performance than `MOVDQU` if the memory value being read was just written.

R
`*p;`

Single-precision Floating-point Vector Intrinsics

The single-precision floating-point vector intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (Intel® SSE3). The prototypes for these intrinsics are in the `pmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in the registers `R0`, `R1`, `R2`, and `R3`.

Intrinsic Name	Operation	Corresponding Intel® SSE3 Instruction
`_mm_addsub_ps`	Subtract and add	`ADDSUBPS`
`_mm_hadd_ps`	Add	`HADDPS`
`_mm_hsub_ps`	Subtracts	`HSUBPS`
`_mm_movehdup_ps`	Duplicates	`MOVSHDUP`

Intrinsic Name	Operation	Corresponding Intel® SSE3 Instruction
`_mm_moveldup_ps`	Duplicates	`MOVSLDUP`

_mm_addsub_ps

```
extern __m128 _mm_addsub_ps(__m128 a, __m128 b);
```

Subtracts even vector elements while adding odd vector elements.

R0	R1	R2	R3
`a0 - b0;`	`a1 + b1;`	`a2 - b2;`	`a3 + b3;`

_mm_hadd_ps

```
extern __m128 _mm_hadd_ps(__m128 a, __m128 b);
```

Adds adjacent vector elements.

R0	R1	R2	R3
`a0 + a1;`	`a2 + a3;`	`b0 + b1;`	`b2 + b3;`

_mm_hsub_ps

```
extern __m128 _mm_hsub_ps(__m128 a, __m128 b);
```

Subtracts adjacent vector elements.

R0	R1	R2	R3
`a0 - a1;`	`a2 - a3;`	`b0 - b1;`	`b2 - b3;`

_mm_movehdup_ps

```
extern __m128 _mm_movehdup_ps(__m128 a);
```

Duplicates odd vector elements into even vector elements.

R0	R1	R2	R3
`a1;`	`a1;`	`a3;`	`a3;`

_mm_moveldup_ps

```
extern __m128 _mm_moveldup_ps(__m128 a);
```

Duplicates even vector elements into odd vector elements.

R0	R1	R2	R3
`a0;`	`a0;`	`a2;`	`a2;`

Double-precision Floating-point Vector Intrinsics

The double-precision floating-point intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (Intel® SSE3). The prototypes for these intrinsics are in the `pmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in the registers `R0` and `R1`.

Intrinsic Name	Operation	Corresponding Intel® SSE3 Instruction
`_mm_addsub_pd`	Subtract and add	`ADDSUBPD`
`_mm_hadd_pd`	Add	`HADDPD`
`_mm_hsub_pd`	Subtract	`HSUBPD`
`_mm_loaddup_pd`	Duplicate	`MOVDDUP`
`_mm_movedup_pd`	Duplicate	`MOVDDUP`

_mm_addsub_pd

```
extern __m128d _mm_addsub_pd(__m128d a, __m128d b);
```

Adds upper vector element while subtracting lower vector element.

R0	R1
`a0 - b0;`	`a1 + b1;`

_mm_hadd_pd

```
extern __m128d _mm_hadd_pd(__m128d a, __m128d b);
```

Adds adjacent vector elements.

R0	R1
`a0 + a1;`	`b0 + b1;`

_mm_hsub_pd

```
extern __m128d _mm_hsub_pd(__m128d a, __m128d b);
```

Subtracts adjacent vector elements.

R0	R1
`a0 - a1;`	`b0 - b1;`

_mm_loaddup_pd

```
extern __m128d _mm_loaddup_pd(double const * dp);
```

Duplicates a double value into upper and lower vector elements.

R0	R1
`*dp;`	`*dp;`

_mm_movedup_pd

```
extern __m128d _mm_movedup_pd(__m128d a);
```

Duplicates lower vector element into upper vector element.

R0	R1
`a0;`	`a0;`

Miscellaneous Intrinsics

The intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (Intel® SSE3). The prototypes for these intrinsics are in the `pmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_mm_monitor

```
extern void _mm_monitor(void const *p, unsigned extensions, unsigned hints);
```

Generates the `MONITOR` instruction. This sets up an address range for the monitor hardware using *p* to provide the logical address, and will be passed to the monitor instruction in register `EAX`. The *extensions* parameter contains optional extensions to the monitor hardware which will be passed in `ECX`. The *hints* parameter will contain hints to the monitor hardware, which will be passed in `EDX`. A non-zero value for *extensions* will cause a general protection fault.

_mm_mwait

```
extern void _mm_mwait(unsigned extensions, unsigned hints);
```

Generates the `MWAIT` instruction. This instruction is a hint that allows the processor to stop execution and enter an implementation-dependent optimized state until occurrence of a class of events. In future processor designs, *extensions* and *hints* parameters may be used to convey additional information to the processor. All non-zero values of *extensions* and *hints* are reserved. A non-zero value for *extensions* will cause a general protection fault.

Intrinsics for Intel® Streaming SIMD Extensions 2 (Intel® SSE2)

This section describes the C++ language-level features supporting the Intel® Streaming SIMD Extensions 2 (Intel® SSE2) in the Intel® C++ Compiler Classic. The features are divided into two categories:

- Floating-Point Intrinsics -- describes the arithmetic, logical, compare, conversion, memory, and initialization intrinsics for the double-precision floating-point data type (`__m128d`).

- Integer Intrinsics -- describes the arithmetic, logical, compare, conversion, memory, and initialization intrinsics for the extended-precision integer data type (`__m128i`).

The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

NOTE
There are no intrinsics for floating-point move operations. To move data from one register to another, a simple assignment, `A = B`, suffices, where `A` and `B` are the source and target registers for the move operation.

Some intrinsics are "composites" - they require more than one instruction to implement them. Intrinsics that require one instruction to implement them are referred to as "simple".

You should be familiar with the hardware features provided by Intel® SSE2 when writing programs with the intrinsics. The following are three important issues to keep in mind:

- Certain intrinsics, such as `_mm_loadr_pd` and `_mm_cmpgt_sd`, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful of their implementation cost.
- Data loaded or stored as `__m128d` objects must be generally 16-byte-aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.

Macro Functions

The macro function intrinsics listed here were designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (Intel® SSE3). They are also compatible with Streaming SIMD Extensions 2 (Intel® SSE2).

The prototypes for these intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

_MM_SET_DENORMALS_ZERO_MODE

`_MM_SET_DENORMALS_ZERO_MODE(x);`

Macro arguments: either `_MM_DENORMALS_ZERO_ON`, `_MM_DENORMALS_ZERO_OFF`.

This macro causes "denormals are zero" mode to be turned ON or OFF by setting the appropriate bit of the control register.

_MM_GET_DENORMALS_ZERO_MODE

`_MM_GET_DENORMALS_ZERO_MODE();`

No arguments.

This macro returns the current value of the denormals are zero mode bit of the control register.

Floating-point Intrinsics

Arithmetic Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for floating-point arithmetic operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in a register. The information about what is placed in each register appears in the tables below, in the detailed explanation for each intrinsic. For each intrinsic, the resulting register is represented by `R0` and `R1`, where `R0` and `R1` each represent one piece of the result register.

Intrinsic Name	**Operation**	**Corresponding Intel® SSE2 Instruction**
`_mm_add_sd`	Addition	`ADDSD`
`_mm_add_pd`	Addition	`ADDPD`
`_mm_sub_sd`	Subtraction	`SUBSD`
`_mm_sub_pd`	Subtraction	`SUBPD`
`_mm_mul_sd`	Multiplication	`MULSD`
`_mm_mul_pd`	Multiplication	`MULPD`
`_mm_div_sd`	Division	`DIVSD`
`_mm_div_pd`	Division	`DIVPD`
`_mm_sqrt_sd`	Computes Square Root	`SQRTSD`
`_mm_sqrt_pd`	Computes Square Root	`SQRTPD`
`_mm_min_sd`	Computes Minimum	`MINSD`
`_mm_min_pd`	Computes Minimum	`MINPD`
`_mm_max_sd`	Computes Maximum	`MAXSD`
`_mm_max_pd`	Computes Maximum	`MAXPD`

_mm_add_sd

```
__m128d _mm_add_sd(__m128d a, __m128d b);
```

Adds the lower double-precision FP (floating-point) values of *a* and *b*; the upper double-precision FP value is passed through from *a*.

R0	**R1**
`a0 + b0`	`a1`

_mm_add_pd

```
__m128d _mm_add_pd(__m128d a, __m128d b);
```

Adds the two DP FP values of *a* and *b*.

R0	R1
`a0 + b0`	`a1 + b1`

_mm_sub_sd

`__m128d _mm_sub_sd(__m128d a, __m128d b);`

Subtracts the lower DP FP value of *b* from *a*. The upper DP FP value is passed through from *a*.

R0	R1
`a0 - b0`	`a1`

_mm_sub_pd

`__m128d _mm_sub_pd(__m128d a, __m128d b);`

Subtracts the two DP FP values of *b* from *a*.

R0	R1
`a0 - b0`	`a1 - b1`

_mm_mul_sd

`__m128d _mm_mul_sd(__m128d a, __m128d b);`

Multiplies the lower DP FP values of *a* and *b*. The upper DP FP is passed through from *a*.

R0	R1
`a0 * b0`	`a1`

_mm_mul_pd

`__m128d _mm_mul_pd(__m128d a, __m128d b);`

Multiplies the two DP FP values of *a* and `b`.

R0	R1
`a0 * b0`	`a1 * b1`

_mm_div_sd

`__m128d _mm_div_sd(__m128d a, __m128d b);`

Divides the lower DP FP values of *a* and *b*. The upper DP FP value is passed through from *a*.

R0	R1
`a0 / b0`	`a1`

_mm_div_pd

```
__m128d _mm_div_pd(__m128d a, __m128d b);
```

Divides the two DP FP values of *a* and *b*.

R0	R1
`a0 / b0`	`a1 / b1`

_mm_sqrt_sd

```
__m128d _mm_sqrt_sd(__m128d a, __m128d b);
```

Computes the square root of the lower DP FP value of *b*. The upper DP FP value is passed through from *a*.

R0	R1
`sqrt(b0)`	`a1`

_mm_sqrt_pd

```
__m128d _mm_sqrt_pd(__m128d a);
```

Computes the square root of the two DP FP values of *a*.

R0	R1
`sqrt(a0)`	`sqrt(a1)`

_mm_min_sd

```
__m128d _mm_min_sd(__m128d a, __m128d b);
```

Computes the minimum of the lower DP FP values of *a* and *b*. The upper DP FP value is passed through from *a*.

R0	R1
`min (a0, b0)`	`a1`

_mm_min_pd

```
__m128d _mm_min_pd(__m128d a, __m128d b);
```

Computes the minima of the two DP FP values of *a* and *b*.

R0	R1
`min (a0, b0)`	`min(a1, b1)`

_mm_max_sd

```
__m128d _mm_max_sd(__m128d a, __m128d b);
```

Computes the maximum of the lower DP FP values of *a* and *b*. The upper DP FP value is passed through from *a*.

R0	R1
`max (a0, b0)`	`a1`

_mm_max_pd

```
__m128d _mm_max_pd(__m128d a, __m128d b);
```

Computes the maxima of the two DP FP values of *a* and *b*.

R0	R1
`max (a0, b0)`	`max (a1, b1)`

Logical Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for floating-point logical operations are listed in the following table. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation for each intrinsic. For each intrinsic, the resulting register is represented by `R0` and `R1`, where `R0` and `R1` each represent one piece of the result register.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_and_pd`	Computes AND	`ANDPD`
`_mm_andnot_pd`	Computes AND and NOT	`ANDNPD`
`_mm_or_pd`	Computes OR	`ORPD`
`_mm_xor_pd`	Computes XOR	`XORPD`

_mm_and_pd

```
__m128d _mm_and_pd(__m128d a, __m128d b);
```

Computes the bitwise `AND` of the two DP FP values of *a* and *b*.

R0	R1
`a0 & b0`	`a1 & b1`

_mm_andnot_pd

```
__m128d _mm_andnot_pd(__m128d a, __m128d b);
```

Computes the bitwise `AND` of the 128-bit value in *b* and the bitwise `NOT` of the 128-bit value in *a*.

R0	R1
`(~a0) & b0`	`(~a1) & b1`

_mm_or_pd

`__m128d _mm_or_pd(__m128d a, __m128d b);`

Computes the bitwise `OR` of the two DP FP values of `a` and *b*.

R0	R1
`a0 | b0`	`a1 | b1`

_mm_xor_pd

`__m128d _mm_xor_pd(__m128d a, __m128d b);`

Computes the bitwise `XOR` of the two DP FP values of *a* and *b*.

R0	R1
`a0 ^ b0`	`a1 ^ b1`

Compare Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for floating-point comparision operations are listed in the following table. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Each comparison intrinsic performs a comparison of *a* and *b*. For the packed form, the two double-precision FP values of *a* and *b* are compared, and a 128-bit mask is returned. For the scalar form, the lower double-precision FP values of *a* and *b* are compared, and a 64-bit mask is returned; the upper double-precision FP value is passed through from *a*.

The mask is set to `0xffffffffffffffff` for each element where the comparison is true, and set to `0x0` where the comparison is false. The `r` following the instruction name indicates that the operands to the instruction are reversed in the actual implementation.

The results of each intrinsic operation are placed in a register. The information about what is placed in each register appears in the tables below, in the detailed explanation for each intrinsic. For each intrinsic, the resulting register is represented by `R`, `R0`, and `R1`, where `R`, `R0`, and `R1` each represent one piece of the result register.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_cmpeq_pd`	Equality	`CMPEQPD`
`_mm_cmplt_pd`	Less Than	`CMPLTPD`
`_mm_cmple_pd`	Less Than or Equal	`CMPLEPD`
`_mm_cmpgt_pd`	Greater Than	`CMPLTPDr`

Intrinsic Name	**Operation**	**Corresponding Intel® SSE2 Instruction**
`_mm_cmpge_pd`	Greater Than or Equal	`CMPLEPDr`
`_mm_cmpord_pd`	Ordered	`CMPORDPD`
`_mm_cmpunord_pd`	Unordered	`CMPUNORDPD`
`_mm_cmpneq_pd`	Inequality	`CMPNEQPD`
`_mm_cmpnlt_pd`	Not Less Than	`CMPNLTPD`
`_mm_cmpnle_pd`	Not Less Than or Equal	`CMPNLEPD`
`_mm_cmpngt_pd`	Not Greater Than	`CMPNLTPDr`
`_mm_cmpnge_pd`	Not Greater Than or Equal	`CMPNLEPDr`
`_mm_cmpeq_sd`	Equality	`CMPEQSD`
`_mm_cmplt_sd`	Less Than	`CMPLTSD`
`_mm_cmple_sd`	Less Than or Equal	`CMPLESD`
`_mm_cmpgt_sd`	Greater Than	`CMPLTSDr`
`_mm_cmpge_sd`	Greater Than or Equal	`CMPLESDr`
`_mm_cmpord_sd`	Ordered	`CMPORDSD`
`_mm_cmpunord_sd`	Unordered	`CMPUNORDSD`
`_mm_cmpneq_sd`	Inequality	`CMPNEQSD`
`_mm_cmpnlt_sd`	Not Less Than	`CMPNLTSD`
`_mm_cmpnle_sd`	Not Less Than or Equal	`CMPNLESD`
`_mm_cmpngt_sd`	Not Greater Than	`CMPNLTSDr`
`_mm_cmpnge_sd`	Not Greater Than or Equal	`CMPNLESDr`
`_mm_comieq_sd`	Equality	`COMISD`
`_mm_comilt_sd`	Less Than	`COMISD`
`_mm_comile_sd`	Less Than or Equal	`COMISD`
`_mm_comigt_sd`	Greater Than	`COMISD`
`_mm_comige_sd`	Greater Than or Equal	`COMISD`
`_mm_comineq_sd`	Not Equal	`COMISD`
`_mm_ucomieq_sd`	Equality	`UCOMISD`
`_mm_ucomilt_sd`	Less Than	`UCOMISD`
`_mm_ucomile_sd`	Less Than or Equal	`UCOMISD`

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_ucomigt_sd`	Greater Than	`UCOMISD`
`_mm_ucomige_sd`	Greater Than or Equal	`UCOMISD`
`_mm_ucomineq_sd`	Not Equal	`UCOMISD`

_mm_cmpeq_pd

```
__m128d _mm_cmpeq_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for equality.

R0	R1
`(a0 == b0) ? 0xffffffffffffffff : 0x0`	`(a1 == b1) ? 0xffffffffffffffff : 0x0`

_mm_cmplt_pd

```
__m128d _mm_cmplt_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for *a* less than *b*.

R0	R1
`(a0 < b0) ? 0xffffffffffffffff : 0x0`	`(a1 < b1) ? 0xffffffffffffffff : 0x0`

_mm_cmple_pd

```
__m128d _mm_cmple_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for *a* less than or equal to *b*.

R0	R1
`(a0 <= b0) ? 0xffffffffffffffff : 0x0`	`(a1 <= b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpgt_pd

```
__m128d _mm_cmpgt_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for *a* greater than *b*.

R0	R1
`(a0 > b0) ? 0xffffffffffffffff : 0x0`	`(a1 > b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpge_pd

```
__m128d _mm_cmpge_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for *a* greater than or equal to *b*.

R0	R1
`(a0 >= b0) ? 0xffffffffffffffff : 0x0`	`(a1 >= b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpord_pd

```
__m128d _mm_cmpord_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for ordered.

R0	R1
`(a0 ord b0) ? 0xffffffffffffffff : 0x0`	`(a1 ord b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpunord_pd

```
__m128d _mm_cmpunord_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for unordered.

R0	R1
`(a0 unord b0) ? 0xffffffffffffffff : 0x0`	`(a1 unord b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpneq_pd

```
__m128d _mm_cmpneq_pd( __m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for inequality.

R0	R1
`(a0 != b0) ? 0xffffffffffffffff : 0x0`	`(a1 != b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpnlt_pd

```
__m128d _mm_cmpnlt_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for *a* not less than *b*.

R0	R1
`!(a0 < b0) ? 0xffffffffffffffff : 0x0`	`!(a1 < b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpnle_pd

```
__m128d _mm_cmpnle_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for *a* not less than or equal to *b*.

R0	R1
`!(a0 <= b0) ? 0xffffffffffffffff : 0x0`	`!(a1 <= b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpngt_pd

```
__m128d _mm_cmpngt_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and `b` for *a* not greater than *b*.

b **R0**	**R1**
`!(a0 > b0) ? 0xffffffffffffffff : 0x0`	`!(a1 > b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpnge_pd

```
__m128d _mm_cmpnge_pd(__m128d a, __m128d b);
```

Compares the two DP FP values of *a* and *b* for *a* not greater than or equal to *b*.

R0	**R1**
`!(a0 >= b0) ? 0xffffffffffffffff : 0x0`	`!(a1 >= b1) ? 0xffffffffffffffff : 0x0`

_mm_cmpeq_sd

```
__m128d _mm_cmpeq_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for equality. The upper DP FP value is passed through from *a*.

R0	**R1**
`(a0 == b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmplt_sd

```
__m128d _mm_cmplt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* less than *b*. The upper DP FP value is passed through from *a*.

R0	**R1**
`(a0 < b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmple_sd

```
__m128d _mm_cmple_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* less than or equal to *b*. The upper DP FP value is passed through from *a*.

R0	**R1**
`(a0 <= b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmpgt_sd

```
__m128d _mm_cmpgt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* greater than *b*. The upper DP FP value is passed through from *a*.

R0	R1
`(a0 > b0) ? 0xffffffffffffffff : 0x0`	a1

_mm_cmpge_sd

```
__m128d _mm_cmpge_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* greater than or equal to *b*. The upper DP FP value is passed through from *a*.

R0	R1
`(a0 >= b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmpord_sd

```
__m128d _mm_cmpord_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for ordered. The upper DP FP value is passed through from *a*.

R0	R1
`(a0 ord b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmpunord_sd

```
__m128d _mm_cmpunord_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for unordered. The upper DP FP value is passed through from *a*.

R0	R1
`(a0 unord b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmpneq_sd

```
__m128d _mm_cmpneq_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for inequality. The upper DP FP value is passed through from *a*.

R0	R1
`(a0 != b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmpnlt_sd

```
__m128d _mm_cmpnlt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* not less than *b*. The upper DP FP value is passed through from *a*.

R0	R1
`!(a0 < b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmpnle_sd

```
__m128d _mm_cmpnle_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* not less than or equal to *b*. The upper DP FP value is passed through from *a*.

R0	R1
!(a0 <= b0) ? 0xffffffffffffffff : 0x0	a1

_mm_cmpngt_sd

```
__m128d _mm_cmpngt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* not greater than *b*. The upper DP FP value is passed through from *a*.

R0	R1
`!(a0 > b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_cmpnge_sd

```
__m128d _mm_cmpnge_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* not greater than or equal to *b*. The upper DP FP value is passed through from *a*.

R0	R1
`!(a0 >= b0) ? 0xffffffffffffffff : 0x0`	`a1`

_mm_comieq_sd

```
int _mm_comieq_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* equal to *b*. If *a* and *b* are equal, 1 is returned. Otherwise, 0 is returned.

R
`(a0 == b0) ? 0x1 : 0x0`

_mm_comilt_sd

```
int _mm_comilt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* less than *b*. If *a* is less than *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 < b0) ? 0x1 : 0x0`

_mm_comile_sd

```
int _mm_comile_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for a less than or equal to *b*. If *a* is less than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 <= b0) ? 0x1 : 0x0`

_mm_comigt_sd

```
int _mm_comigt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* greater than *b*. If *a* is greater than *b* are equal, 1 is returned. Otherwise, 0 is returned.

R
`(a0 > b0) ? 0x1 : 0x0`

_mm_comige_sd

```
int _mm_comige_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* greater than or equal to *b*. If *a* is greater than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 >= b0) ? 0x1 : 0x0`

_mm_comineq_sd

```
int _mm_comineq_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of `a` and *b* for *a* not equal to *b*. If *a* and *b* are not equal, 1 is returned. Otherwise, 0 is returned.

R
`(a0 != b0) ? 0x1 : 0x0`

_mm_ucomieq_sd

```
int _mm_ucomieq_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for `a` equal to *b*. If *a* and *b* are equal, 1 is returned. Otherwise, 0 is returned.

R
(a0 == b0) ? 0x1 : 0x0

_mm_ucomilt_sd

```
int _mm_ucomilt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* less than *b*. If a is less than *b*, 1 is returned. Otherwise, 0 is returned.

R
(a0 < b0) ? 0x1 : 0x0

_mm_ucomile_sd

```
int _mm_ucomile_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for a less than or equal to *b*. If *a* is less than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
(a0 <= b0) ? 0x1 : 0x0

_mm_ucomigt_sd

```
int _mm_ucomigt_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for a greater than *b*. If *a* is greater than *b* are equal, 1 is returned. Otherwise, 0 is returned.

R
(a0 > b0) ? 0x1 : 0x0

_mm_ucomige_sd

```
int _mm_ucomige_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* greater than or equal to *b*. If *a* is greater than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
(a0 >= b0) ? 0x1 : 0x0

_mm_ucomineq_sd

```
int _mm_ucomineq_sd(__m128d a, __m128d b);
```

Compares the lower DP FP value of *a* and *b* for *a* not equal to *b*. If *a* and *b* are not equal, 1 is returned. Otherwise, 0 is returned.

R
(a0 != b0) ? 0x1 : 0x0

Conversion Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for floating-point conversion operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Each conversion intrinsic takes one data type and performs a conversion to a different type. Some conversions, such as those performed by the `_mm_cvtpd_ps` intrinsic, result in a loss of precision. The rounding mode used in such cases is determined by the value in the `MXCSR` register. The default rounding mode is round-to-nearest.

> **NOTE**
> The rounding mode used by the C and C++ languages when performing a type conversion is to truncate. The `_mm_cvttpd_epi32` and `_mm_cvttsd_si32` intrinsics use the truncate rounding mode regardless of the mode specified by the `MXCSR` register.

The results of each intrinsic operation are placed in a register. The information about what is placed in each register appears in the tables below, in the detailed explanation for each intrinsic. For each intrinsic, the resulting register is represented by `R`, `R0`, `R1`, `R2`, and `R3`, where each represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_cvtpd_ps`	Convert DP FP to SP FP	`CVTPD2PS`
`_mm_cvtps_pd`	Convert from SP FP to DP FP	`CVTPS2PD`
`_mm_cvtepi32_pd`	Convert lower integer values to DP FP	`CVTDQ2PD`
`_mm_cvtpd_epi32`	Convert DP FP values to integer values	`CVTPD2DQ`
`_mm_cvtsd_si32`	Convert lower DP FP value to integer value	`CVTSD2SI`
`_mm_cvtsd_ss`	Convert lower DP FP value to SP FP	`CVTSD2SS`
`_mm_cvtsi32_sd`	Convert signed integer value to DP FP	`CVTSI2SD`
`_mm_cvtss_sd`	Convert lower SP FP value to DP FP	`CVTSS2SD`
`_mm_cvttpd_epi32`	Convert DP FP values to signed integers	`CVTTPD2DQ`

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_cvttsd_si32`	Convert lower DP FP to signed integer	`CVTTSD2SI`
`_mm_cvtpd_pi32`	Convert two DP FP values to signed integer values	`CVTPD2PI`
`_mm_cvttpd_pi32`	Convert two DP FP values to signed integer values using truncate	`CVTTPD2PI`
`_mm_cvtpi32_pd`	Convert two signed integer values to DP FP	`CVTPI2PD`
`_mm_cvtsd_f64`	Extract DP FP value from first vector element	None

_mm_cvtpd_ps

```
__m128 _mm_cvtpd_ps(__m128d a);
```

Converts the two DP FP values of *a* to SP FP values.

R0	R1	R2	R3
`(float) a0`	`(float) a1`	`0.0`	`0.0`

_mm_cvtps_pd

```
__m128d _mm_cvtps_pd(__m128 a);
```

Converts the lower two SP FP values of *a* to DP FP values.

R0	R1
`(double) a0`	`(double) a1`

_mm_cvtepi32_pd

```
__m128d _mm_cvtepi32_pd(__m128i a);
```

Converts the lower two signed 32-bit integer values of *a* to DP FP values.

R0	R1
`(double) a0`	`(double) a1`

_mm_cvtpd_epi32

```
__m128i _mm_cvtpd_epi32(__m128d a);
```

Converts the two DP FP values of *a* to 32-bit signed integer values.

R0	R1	R2	R3
`(int) a0`	`(int) a1`	`0x0`	`0x0`

_mm_cvtsd_si32

```
int _mm_cvtsd_si32(__m128d a);
```

Converts the lower DP FP value of *a* to a 32-bit signed integer value.

R
`(int) a0`

_mm_cvtsd_ss

```
__m128 _mm_cvtsd_ss(__m128 a, __m128d b);
```

Converts the lower DP FP value of *b* to an SP FP value. The upper SP FP values in *a* are passed through.

R0	R1	R2	R3
`(float) b0`	`a1`	`a2`	`a3`

_mm_cvtsi32_sd

```
__m128d _mm_cvtsi32_sd(__m128d a, int b);
```

Converts the signed integer value in *b* to a DP FP value. The upper DP FP value in *a* is passed through.

R0	R1
`(double) b`	`a1`

_mm_cvtss_sd

```
__m128d _mm_cvtss_sd(__m128d a, __m128 b);
```

Converts the lower SP FP value of *b* to a DP FP value. The upper value DP FP value in *a* is passed through.

R0	R1
`(double) b0`	`a1`

_mm_cvttpd_epi32

```
__m128i _mm_cvttpd_epi32(__m128d a);
```

Converts the two DP FP values of *a* to 32-bit signed integers using truncate.

R0	R1	R2	R3
`(int) a0`	`(int) a1`	`0x0`	`0x0`

_mm_cvttsd_si32

```
int _mm_cvttsd_si32(__m128d a);
```

Converts the lower DP FP value of *a* to a 32-bit signed integer using truncate.

R
(int) a0

_mm_cvtpd_pi32

```
__m64 _mm_cvtpd_pi32(__m128d a);
```

Converts the two DP FP values of *a* to 32-bit signed integer values.

R0	R1
(int)a0	(int) a1

_mm_cvttpd_pi32

```
__m64 _mm_cvttpd_pi32(__m128d a);
```

Converts the two DP FP values of *a* to 32-bit signed integer values using truncate.

R0	R1
(int)a0	(int) a1

_mm_cvtpi32_pd

```
__m128d _mm_cvtpi32_pd(__m64 a);
```

Converts the two 32-bit signed integer values of *a* to DP FP values.

R0	R1
(double)a0	(double)a1

_mm_cvtsd_f64

```
double _mm_cvtsd_f64(__m128d a);
```

This intrinsic extracts a double precision floating point value from the first vector element of an `__m128d`. It does so in the most efficient manner possible in the context used.

> **NOTE**
> This intrinsic does not map to any specific Intel® SSE2 instruction.

Load Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for floating-point load operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The load and set operations are similar in that both initialize `__m128d` data. However, the set operations take a double argument and are intended for initialization with constants, while the load operations take a double pointer argument and are intended to mimic the instructions for loading data from memory.

The results of each intrinsic operation are placed in a register. The information about what is placed in each register appears in the tables below, in the detailed explanation for each intrinsic. For each intrinsic, the resulting register is represented by R0 and R1, where R0 and R1 each represent one piece of the result register.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
_mm_load_pd	Loads two DP FP values	MOVAPD
_mm_load1_pd	Loads a single DP FP value, copying to both elements	MOVSD + shuffling
_mm_loadr_pd	Loads two DP FP values in reverse order	MOVAPD + shuffling
_mm_loadu_pd	Loads two DP FP values	MOVUPD
_mm_load_sd	Loads a DP FP value, sets upper DP FP to zero	MOVSD
_mm_loadh_pd	Loads a DP FP value as the upper DP FP value of the result	MOVHPD
_mm_loadl_pd	Loads a DP FP value as the lower DP FP value of the result	MOVLPD

_mm_load_pd

`__m128d _mm_load_pd(double const*dp);`

Loads two DP FP values. The address *p* must be 16-byte aligned.

R0	R1
p[0]	p[1]

_mm_load1_pd

`__m128d _mm_load1_pd(double const*dp);`

Loads a single DP FP value, copying to both elements. The address *p* need not be 16-byte aligned.

R0	R1
*p	*p

_mm_loadr_pd

`__m128d _mm_loadr_pd(double const*dp);`

Loads two DP FP values in reverse order. The address *p* must be 16-byte aligned.

R0	R1
p[1]	p[0]

_mm_loadu_pd

```
__m128d _mm_loadu_pd(double const*dp);
```

Loads two DP FP values. The address *p* need not be 16-byte aligned.

R0	R1
p[0]	p[1]

_mm_load_sd

```
__m128d _mm_load_sd(double const*dp);
```

Loads a DP FP value. The upper DP FP is set to zero. The address *p* need not be 16-byte aligned.

R0	R1
*p	0.0

_mm_loadh_pd

```
__m128d _mm_loadh_pd(__m128d a, double const*dp);
```

Loads a DP FP value as the upper DP FP value of the result. The lower DP FP value is passed through from *a*. The address *p* need not be 16-byte aligned.

R0	R1
a0	*p

_mm_loadl_pd

```
__m128d _mm_loadl_pd(__m128d a, double const*dp);
```

Loads a DP FP value as the lower DP FP value of the result. The upper DP FP value is passed through from *a*. The address *p* need not be 16-byte aligned.

R0	R1
*p	a1

Set Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for floating-point set operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The load and set operations are similar in that both initialize `__m128d` data. However, the set operations take a double argument and are intended for initialization with constants, while the load operations take a double pointer argument and are intended to mimic the instructions for loading data from memory.

Some of the these intrinsics are composite intrinsics because they require more than one instruction to implement them.

The results of each intrinsic operation are placed in a register. The information about what is placed in each register appears in the tables below, in the detailed explanation for each intrinsic. For each intrinsic, the resulting register is represented by `R0` and `R1`, where `R0` and `R1` each represent one piece of the result register.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_set_sd`	Sets lower DP FP value to *w* and upper to zero	`Composite`
`_mm_set1_pd`	Sets two DP FP values to *w*	`Composite`
`_mm_set_pd`	Sets lower DP FP to *x* and upper to *w*	`Composite`
`_mm_setr_pd`	Sets lower DP FP to *w* and upper to *x*	`Composite`
`_mm_setzero_pd`	Sets two DP FP values to zero	`XORPD`
`_mm_move_sd`	Sets lower DP FP value to the lower DP FP value of *b*	`MOVSD`

_mm_set_sd

```
__m128d _mm_set_sd(double w);
```

Sets the lower DP FP value to *w* and sets the upper DP FP value to zero.

R0	R1
`w`	`0.0`

_mm_set1_pd

```
__m128d _mm_set1_pd(double w);
```

Sets the two DP FP values to *w*.

R0	R1
`w`	`w`

_mm_set_pd

```
__m128d _mm_set_pd(double w, double x);
```

Sets the lower DP FP value to *x* and sets the upper DP FP value to *w*.

R0	R1
x	w

_mm_setr_pd

```
__m128d _mm_setr_pd(double w, double x);
```

Sets the lower DP FP value to *w* and sets the upper DP FP value to *x*. `r0 := w r1 := x`

R0	R1
w	x

_mm_setzero_pd

```
__m128d _mm_setzero_pd(void);
```

Sets the two DP FP values to zero.

R0	R1
0.0	0.0

_mm_move_sd

```
__m128d _mm_move_sd( __m128d a, __m128d b);
```

Sets the lower DP FP value to the lower DP FP value of *b*. The upper DP FP value is passed through from *a*.

R0	R1
b0	a1

Store Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for floating-point store operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The store operations assign the initialized data to the address.

The detailed description of each intrinsic contains a table detailing the returns. In these tables, *dp[n]* is an access to the *n* element of the result.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_stream_pd`	Store	`MOVNTPD`
`_mm_store_sd`	Store	`MOVSD`
`_mm_store1_pd`	Store	`MOVAPD` + shuffling
`_mm_store_pd`	Store	`MOVAPD`

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_storeu_pd`	Store	`MOVUPD`
`_mm_storer_pd`	Store	`MOVAPD` + shuffling
`_mm_storeh_pd`	Store	`MOVHPD`
`_mm_storel_pd`	Store	`MOVLPD`

_mm_store_sd

```
void _mm_store_sd(double *dp, __m128d a);
```

Stores the lower DP FP value of *a*. The address *dp* needs not be 16-byte aligned.

*dp
a0

_mm_store1_pd

```
void _mm_store1_pd(double *dp, __m128d a);
```

Stores the lower DP FP value of *a* twice. The address *dp* must be 16-byte aligned.

dp[0]	dp[1]
a0	a0

_mm_store_pd

```
void _mm_store_pd(double *dp, __m128d a);
```

Stores two DP FP values. The address *dp* must be 16-byte aligned.

dp[0]	dp[1]
a0	a1

_mm_storeu_pd

```
void _mm_storeu_pd(double *dp, __m128d a);
```

Stores two DP FP values. The address *dp* need not be 16-byte aligned.

dp[0]	dp[1]
a0	a1

_mm_storer_pd

```
void _mm_storer_pd(double *dp, __m128d a);
```

Stores two DP FP values in reverse order. The address *dp* must be 16-byte aligned.

dp[0]	dp[1]
a1	a0

_mm_storeh_pd

```
void _mm_storeh_pd(double *dp, __m128d a);
```

Stores the upper DP FP value of *a*.

*dp
a1

```
void _mm_storel_pd(double *dp, __m128d a);
```

Stores the lower DP FP value of `a`.

*dp
a0

Integer Intrinsics

Arithmetic Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer arithmetic operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, `R1`, ..., `R15` represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
_mm_add_epi8	Addition	PADDB
_mm_add_epi16	Addition	PADDW
_mm_add_epi32	Addition	PADDD
_mm_add_si64	Addition	PADDQ
_mm_add_epi64	Addition	PADDQ
_mm_adds_epi8	Addition	PADDSB
_mm_adds_epi16	Addition	PADDSW
_mm_adds_epu8	Addition	PADDUSB
_mm_adds_epu16	Addition	PADDUSW

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
_mm_avg_epu8	Computes Average	PAVGB
_mm_avg_epu16	Computes Average	PAVGW
_mm_madd_epi16	Multiplication and Addition	PMADDWD
_mm_max_epi16	Computes Maxima	PMAXSW
_mm_max_epu8	Computes Maxima	PMAXUB
_mm_min_epi16	Computes Minima	PMINSW
_mm_min_epu8	Computes Minima	PMINUB
_mm_mulhi_epi16	Multiplication	PMULHW
_mm_mulhi_epu16	Multiplication	PMULHUW
_mm_mullo_epi16	Multiplication	PMULLW
_mm_mul_su32	Multiplication	PMULUDQ
_mm_mul_epu32	Multiplication	PMULUDQ
_mm_sad_epu8	Computes Difference/Adds	PSADBW
_mm_sub_epi8	Subtraction	PSUBB
_mm_sub_epi16	Subtraction	PSUBW
_mm_sub_epi32	Subtraction	PSUBD
_mm_sub_si64	Subtraction	PSUBQ
_mm_sub_epi64	Subtraction	PSUBQ
_mm_subs_epi8	Subtraction	PSUBSB
_mm_subs_epi16	Subtraction	PSUBSW
_mm_subs_epu8	Subtraction	PSUBUSB
_mm_subs_epu16	Subtraction	PSUBUSW

_mm_add_epi8

```
__m128i _mm_add_epi8(__m128i a, __m128i b);
```

Adds the 16 signed or unsigned 8-bit integers in *a* to the 16 signed or unsigned 8-bit integers in *b*.

R0	R1	...	R15
a0 + b0	a1 + b1;	...	a15 + b15

_mm_add_epi16

```
__m128i _mm_add_epi16(__m128i a, __m128i b);
```

Adds the eight signed or unsigned 16-bit integers in *a* to the eight signed or unsigned 16-bit integers in *b*.

R0	R1	...	R7
a0 + b0	a1 + b1	...	a7 + b7

_mm_add_epi32

```
__m128i _mm_add_epi32(__m128i a, __m128i b);
```

Adds the four signed or unsigned 32-bit integers in *a* to the four signed or unsigned 32-bit integers in *b*.

R0	R1	R2	R3
a0 + b0	a1 + b1	a2 + b2	a3 + b3

_mm_add_si64

```
__m64 _mm_add_si64(__m64 a, __m64 b);
```

Adds the signed or unsigned 64-bit integer *a* to the signed or unsigned 64-bit integer *b*.

R0
a + b

_mm_add_epi64

```
__m128i _mm_add_epi64(__m128i a, __m128i b);
```

Adds the two signed or unsigned 64-bit integers in *a* to the two signed or unsigned 64-bit integers in *b*.

R0	R1
a0 + b0	a1 + b1

_mm_adds_epi8

```
__m128i _mm_adds_epi8(__m128i a, __m128i b);
```

Adds the 16 signed 8-bit integers in *a* to the 16 signed 8-bit integers in *b* using saturating arithmetic.

R0	R1	...	R15
SignedSaturate (a0 + b0)	SignedSaturate (a1 + b1)	...	SignedSaturate (a15 + b15)

_mm_adds_epi16

```
__m128i _mm_adds_epi16(__m128i a, __m128i b);
```

Adds the eight signed 16-bit integers in *a* to the eight signed 16-bit integers in *b* using saturating arithmetic.

R0	R1	...	R7
SignedSaturate (a0 + b0)	SignedSaturate (a1 + b1)	...	SignedSaturate (a7 + b7)

_mm_adds_epu8

```
__m128i _mm_adds_epu8(__m128i a, __m128i b);
```

Adds the 16 unsigned 8-bit integers in *a* to the 16 unsigned 8-bit integers in *b* using saturating arithmetic.

R0	R1	...	R15
UnsignedSaturate (a0 + b0)	UnsignedSaturate (a1 + b1)	...	UnsignedSaturate (a15 + b15)

_mm_adds_epu16

```
__m128i _mm_adds_epu16(__m128i a, __m128i b);
```

Adds the eight unsigned 16-bit integers in *a* to the eight unsigned 16-bit integers in *b* using saturating arithmetic.

R0	R1	...	R7
UnsignedSaturate (a0 + b0)	UnsignedSaturate (a1 + b1)	...	UnsignedSaturate (a7 + b7)

_mm_avg_epu8

```
__m128i _mm_avg_epu8(__m128i a, __m128i b);
```

Computes the average of the 16 unsigned 8-bit integers in *a* and the 16 unsigned 8-bit integers in *b* and rounds.

R0	R1	...	R15
(a0 + b0) / 2	(a1 + b1) / 2	...	(a15 + b15) / 2

_mm_avg_epi16

```
__m128i _mm_avg_epu16(__m128i a, __m128i b);
```

Computes the average of the eight unsigned 16-bit integers in *a* and the eight unsigned 16-bit integers in *b* and rounds.

R0	R1	...	R7
(a0 + b0) / 2	(a1 + b1) / 2	...	(a7 + b7) / 2

_mm_madd_epi16

```
__m128i _mm_madd_epi16(__m128i a, __m128i b);
```

Multiplies the eight signed 16-bit integers from *a* by the eight signed 16-bit integers from *b*. Adds the signed 32-bit integer results pairwise and packs the four signed 32-bit integer results.

R0	R1	R2	R3
`(a0 * b0) + (a1 * b1)`	`(a2 * b2) + (a3 * b3)`	`(a4 * b4) + (a5 * b5)`	`(a6 * b6) + (a7 * b7)`

_mm_max_epi16

```
__m128i _mm_max_epi16(__m128i a, __m128i b);
```

Computes the pairwise maxima of the eight signed 16-bit integers from *a* and the eight signed 16-bit integers from *b*.

R0	R1	...	R7
`max(a0, b0)`	`max(a1, b1)`	...	`max(a7, b7)`

_mm_max_epu8

```
__m128i _mm_max_epu8(__m128i a, __m128i b);
```

Computes the pairwise maxima of the 16 unsigned 8-bit integers from *a* and the 16 unsigned 8-bit integers from *b*.

R0	R1	...	R15
`max(a0, b0)`	`max(a1, b1)`	...	`max(a15, b15)`

_mm_min_epi16

```
__m128i _mm_min_epi16(__m128i a, __m128i b);
```

Computes the pairwise minima of the eight signed 16-bit integers from *a* and the eight signed 16-bit integers from *b*.

R0	R1	...	R7
`min(a0, b0)`	`min(a1, b1)`	...	`min(a7, b7)`

_mm_min_epu8

```
__m128i _mm_min_epu8(__m128i a, __m128i b);
```

Computes the pairwise minima of the 16 unsigned 8-bit integers from *a* and the 16 unsigned 8-bit integers from *b*.

R0	R1	...	R15
`min(a0, b0)`	`min(a1, b1)`	...	`min(a15, b15)`

_mm_mulhi_epi16

```
__m128i _mm_mulhi_epi16(__m128i a, __m128i b);
```

Multiplies the eight signed 16-bit integers from *a* by the eight signed 16-bit integers from *b*. Packs the upper 16-bits of the eight signed 32-bit results.

R0	R1	...	R7
(a0 * b0)[31:16]	(a1 * b1)[31:16]	...	(a7 * b7)[31:16]

_mm_mulhi_epu16

```
__m128i _mm_mulhi_epu16(__m128i a, __m128i b);
```

Multiplies the eight unsigned 16-bit integers from *a* by the eight unsigned 16-bit integers from *b*. Packs the upper 16-bits of the eight unsigned 32-bit results.

R0	R1	...	R7
(a0 * b0)[31:16]	(a1 * b1)[31:16]	...	(a7 * b7)[31:16]

_mm_mullo_epi16

```
__m128i _mm_mullo_epi16(__m128i a, __m128i b);
```

Multiplies the eight signed or unsigned 16-bit integers from *a* by the eight signed or unsigned 16-bit integers from *b*. Packs the lower 16-bits of the eight signed or unsigned 32-bit results.

R0	R1	...	R7
(a0 * b0)[15:0]	(a1 * b1)[15:0]	...	(a7 * b7)[15:0]

_mm_mul_su32

```
__m64 _mm_mul_su32(__m64 a, __m64 b);
```

Multiplies the lower 32-bit integer from *a* by the lower 32-bit integer from *b*, and returns the 64-bit integer result.

R0
a0 * b0

_mm_mul_epu32

```
__m128i _mm_mul_epu32(__m128i a, __m128i b);
```

Multiplies two unsigned 32-bit integers from *a* by two unsigned 32-bit integers from *b*. Packs the two unsigned 64-bit integer results.

R0	R1
a0 * b0	a2 * b2

_mm_sad_epu8

```
__m128i _mm_sad_epu8(__m128i a, __m128i b);
```

Computes the absolute difference of the 16 unsigned 8-bit integers from *a* and the 16 unsigned 8-bit integers from *b*. Sums the upper eight differences and lower eight differences, and packs the resulting two unsigned 16-bit integers into the upper and lower 64-bit elements.

R0	R1 to R3	R4	R5 to R7
abs(a0 - b0) + abs(a1 - b1) +...+ abs(a7 - b7)	0x0	abs(a8 - b8) + abs(a9 - b9) +...+ abs(a15 - b15)	0x0

_mm_sub_epi8

```
__m128i _mm_sub_epi8(__m128i a, __m128i b);
```

Subtracts the 16 signed or unsigned 8-bit integers of *b* from the 16 signed or unsigned 8-bit integers of *a*.

R0	R1	...	R15
a0 - b0	a1 - b1	...	a15 - b15

_mm_sub_epi16

```
__m128i _mm_sub_epi16(__m128i a, __m128i b);
```

Subtracts the eight signed or unsigned 16-bit integers of *b* from the eight signed or unsigned 16-bit integers of *a*.

R0	R1	...	R7
a0 - b0	a1 - b1	...	a7 - b7

_mm_sub_epi32

```
__m128i _mm_sub_epi32(__m128i a, __m128i b);
```

Subtracts the four signed or unsigned 32-bit integers of *b* from the four signed or unsigned 32-bit integers of *a*.

R0	R1	R2	R3
a0 - b0	a1 - b1	a2 - b2	a3 - b3

_mm_sub_si64

```
__m64 _mm_sub_si64 (__m64 a, __m64 b);
```

Subtracts the signed or unsigned 64-bit integer *b* from the signed or unsigned 64-bit integer *a*.

R
a - b

_mm_sub_epi64

```
__m128i _mm_sub_epi64(__m128i a, __m128i b);
```

Subtracts the two signed or unsigned 64-bit integers in *b* from the two signed or unsigned 64-bit integers in *a*.

R0	R1
a0 - b0	a1 - b1

_mm_subs_epi8

```
__m128i _mm_subs_epi8(__m128i a, __m128i b);
```

Subtracts the 16 signed 8-bit integers of *b* from the 16 signed 8-bit integers of *a* using saturating arithmetic.

R0	R1	...	R15
SignedSaturate (a0 - b0)	SignedSaturate (a1 - b1)	...	SignedSaturate (a15 - b15)

_mm_subs_epi16

```
__m128i _mm_subs_epi16(__m128i a, __m128i b);
```

Subtracts the eight signed 16-bit integers of *b* from the eight signed 16-bit integers of *a* using saturating arithmetic.

R0	R1	...	R15
SignedSaturate (a0 - b0)	SignedSaturate (a1 - b1)	...	SignedSaturate (a7 - b7)

_mm_subs_epu8

```
__m128i _mm_subs_epu8(__m128i a, __m128i b);
```

Subtracts the 16 unsigned 8-bit integers of *b* from the 16 unsigned 8-bit integers of *a* using saturating arithmetic.

R0	R1	...	R15
UnsignedSaturate (a0 - b0)	UnsignedSaturate (a1 - b1)	...	UnsignedSaturate (a15 - b15)

_mm_subs_epu16

```
__m128i _mm_subs_epu16(__m128i a, __m128i b);
```

Subtracts the eight unsigned 16-bit integers of *b* from the eight unsigned 16-bit integers of *a* using saturating arithmetic.

R0	R1	...	R7
UnsignedSaturate (a0 - b0)	UnsignedSaturate (a1 - b1)	...	UnsignedSaturate (a7 - b7)

Logical Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer logical operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in register `R`. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_and_si128`	Computes AND	`PAND`
`_mm_andnot_si128`	Computes AND and NOT	`PANDN`
`_mm_or_si128`	Computes OR	`POR`
`_mm_xor_si128`	Computes XOR	`PXOR`

_mm_and_si128

```
__m128i _mm_and_si128(__m128i a, __m128i b);
```

Computes the bitwise AND of the 128-bit value in *a* and the 128-bit value in *b*.

R0
`a & b`

_mm_andnot_si128

```
__m128i _mm_andnot_si128(__m128i a, __m128i b);
```

Computes the bitwise AND of the 128-bit value in *b* and the bitwise NOT of the 128-bit value in *a*.

R0
`(~a) & b`

_mm_or_si128

```
__m128i _mm_or_si128(__m128i a, __m128i b);
```

Computes the bitwise OR of the 128-bit value in *a* and the 128-bit value in *b*.

R0
`a | b`

_mm_xor_si128

```
__m128i _mm_xor_si128(__m128i a, __m128i b);
```

Computes the bitwise XOR of the 128-bit value in *a* and the 128-bit value in *b*.

R0
`a ^ b`

Shift Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer shift operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, `R1`...`R7` represent the registers in which results are placed.

NOTE
The `count` argument is one shift count that applies to all elements of the operand being shifted. It is not a vector shift count that shifts each element by a different amount.

Intrinsic	Operation	Shift Type	Corresponding Intel® SSE2 Instruction
`_mm_slli_si128`	Shift left	Logical	`PSLLDQ`
`_mm_slli_epi16`	Shift left	Logical	`PSLLW`
`_mm_sll_epi16`	Shift left	Logical	`PSLLW`
`_mm_slli_epi32`	Shift left	Logical	`PSLLD`
`_mm_sll_epi32`	Shift left	Logical	`PSLLD`
`_mm_slli_epi64`	Shift left	Logical	`PSLLQ`
`_mm_sll_epi64`	Shift left	Logical	`PSLLQ`
`_mm_srai_epi16`	Shift right	Arithmetic	`PSRAW`
`_mm_sra_epi16`	Shift right	Arithmetic	`PSRAW`
`_mm_srai_epi32`	Shift right	Arithmetic	`PSRAD`
`_mm_sra_epi32`	Shift right	Arithmetic	`PSRAD`
`_mm_srli_si128`	Shift right	Logical	`PSRLDQ`
`_mm_srli_epi16`	Shift right	Logical	`PSRLW`
`_mm_srl_epi16`	Shift right	Logical	`PSRLW`
`_mm_srli_epi32`	Shift right	Logical	`PSRLD`
`_mm_srl_epi32`	Shift right	Logical	`PSRLD`
`_mm_srli_epi64`	Shift right	Logical	`PSRLQ`
`_mm_srl_epi64`	Shift right	Logical	`PSRLQ`

_mm_slli_si128

`__m128i _mm_slli_si128(__m128i a, int imm);`

Shifts the 128-bit value in *a* left by *imm* bytes while shifting in zeros. *imm* must be an immediate.

R
a << (imm * 8)

_mm_slli_epi16

`__m128i _mm_slli_epi16(__m128i a, int count);`

Shifts the eight signed or unsigned 16-bit integers in *a* left by count bits while shifting in zeros.

R0	R1	...	R7
`a0 << count`	`a1 << count`	...	`a7 << count`

_mm_sll_epi16

`__m128i _mm_sll_epi16(__m128i a, __m128i count);`

Shifts the eight signed or unsigned 16-bit integers in *a* left by *count* bits while shifting in zeros.

R0	R1	...	R7
`a0 << count`	`a1 << count`	...	`a7 << count`

_mm_slli_epi32

`__m128i _mm_slli_epi32(__m128i a, int count);`

Shifts the four signed or unsigned 32-bit integers in *a* left by *count* bits while shifting in zeros.

R0	R1	R2	R3
`a0 << count`	`a1 << count`	`a2 << count`	`a3 << count`

_mm_sll_epi32

`__m128i _mm_sll_epi32(__m128i a, __m128i count);`

Shifts the four signed or unsigned 32-bit integers in *a* left by *count* bits while shifting in zeros.

R0	R1	R2	R3
`a0 << count`	`a1 << count`	`a2 << count`	`a3 << count`

_mm_slli_epi64

`__m128i _mm_slli_epi64(__m128i a, int count);`

Shifts the two signed or unsigned 64-bit integers in *a* left by *count* bits while shifting in zeros.

R0	R1
a0 << count	a1 << count

_mm_sll_epi64

```
__m128i _mm_sll_epi64(__m128i a, __m128i count);
```

Shifts the two signed or unsigned 64-bit integers in *a* left by *count* bits while shifting in zeros.

R0	R1
a0 << count	a1 << count

_mm_srai_epi16

```
__m128i _mm_srai_epi16(__m128i a, int count);
```

Shifts the eight signed 16-bit integers in *a* right by *count* bits while shifting in the sign bit.

R0	R1	...	R7
a0 >> count	a1 >> count	...	a7 >> count

_mm_sra_epi16

```
__m128i _mm_sra_epi16(__m128i a, __m128i count);
```

Shifts the eight signed 16-bit integers in *a* right by *count* bits while shifting in the sign bit.

R0	R1	...	R7
a0 >> count	a1 >> count	...	a7 >> count

_mm_srai_epi32

```
__m128i _mm_srai_epi32(__m128i a, int count);
```

Shifts the four signed 32-bit integers in *a* right by *count* bits while shifting in the sign bit.

R0	R1	R2	R3
a0 >> count	a1 >> count	a2 >> count	a3 >> count

_mm_sra_epi32

```
__m128i _mm_sra_epi32(__m128i a, __m128i count);
```

Shifts the four signed 32-bit integers in *a* right by *count* bits while shifting in the sign bit.

R0	R1	R2	R3
a0 >> count	a1 >> count	a2 >> count	a3 >> count

_mm_srli_si128

```
__m128i _mm_srli_si128(__m128i a, int imm);
```

Shifts the 128-bit value in *a* right by *imm* bytes while shifting in zeros. *imm* must be an immediate.

R
`srl(a, imm*8)`

_mm_srli_epi16

```
__m128i _mm_srli_epi16(__m128i a, int count);
```

Shifts the eight signed or unsigned 16-bit integers in *a* right by *count* bits while shifting in zeros.

R0	R1	...	R7
`srl(a0, count)`	`srl(a1, count)`	...	`srl(a7, count)`

_mm_srl_epi16

```
__m128i _mm_srl_epi16(__m128i a, __m128i count);
```

Shifts the eight signed or unsigned 16-bit integers in *a* right by `count` bits while shifting in zeros.

R0	R1	...	R7
`srl(a0, count)`	`srl(a1, count)`	...	`srl(a7, count)`

_mm_srli_epi32

```
__m128i _mm_srli_epi32(__m128i a, int count);
```

Shifts the four signed or unsigned 32-bit integers in *a* right by *count* bits while shifting in zeros.

R0	R1	R2	R3
`srl(a0, count)`	`srl(a1, count)`	`srl(a2, count)`	`srl(a3, count)`

_mm_srl_epi32

```
__m128i _mm_srl_epi32(__m128i a, __m128i count);
```

Shifts the four signed or unsigned 32-bit integers in *a* right by *count* bits while shifting in zeros.

R0	R1	R2	R3
`srl(a0, count)`	`srl(a1, count)`	`srl(a2, count)`	`srl(a3, count)`

_mm_srli_epi64

```
__m128i _mm_srli_epi64(__m128i a, int count)
```

Shifts the two signed or unsigned 64-bit integers in *a* right by *count* bits while shifting in zeros.

R0	R1
`srl(a0, count)`	`srl(a1, count)`

_mm_srl_epi64

```
__m128i _mm_srl_epi64(__m128i a, __m128i count)
```

Shifts the two signed or unsigned 64-bit integers in *a* right by *count* bits while shifting in zeros.

R0	R1
`srl(a0, count)`	`srl(a1, count)`

Compare Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer comparison operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, `R1`, ...,`R15` represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_cmpeq_epi8`	Equality	`PCMPEQB`
`_mm_cmpeq_epi16`	Equality	`PCMPEQW`
`_mm_cmpeq_epi32`	Equality	`PCMPEQD`
`_mm_cmpgt_epi8`	Greater Than	`PCMPGTB`
`_mm_cmpgt_epi16`	Greater Than	`PCMPGTW`
`_mm_cmpgt_epi32`	Greater Than	`PCMPGTD`
`_mm_cmplt_epi8`	Less Than	`PCMPGTBr`
`_mm_cmplt_epi16`	Less Than	`PCMPGTWr`
`_mm_cmplt_epi32`	Less Than	`PCMPGTDr`

_mm_cmpeq_epi8

```
__m128i _mm_cmpeq_epi8(__m128i a, __m128i b);
```

Compares the 16 signed or unsigned 8-bit integers in *a* and the 16 signed or unsigned 8-bit integers in *b* for equality.

R0	R1	...	R15
`(a0 == b0) ? 0xff : 0x0`	`(a1 == b1) ? 0xff : 0x0`	...	`(a15 == b15) ? 0xff : 0x0`

_mm_cmpeq_epi16

```
__m128i _mm_cmpeq_epi16(__m128i a, __m128i b);
```

Compares the eight signed or unsigned 16-bit integers in *a* and the eight signed or unsigned 16-bit integers in *b* for equality.

R0	R1	...	R7
`(a0 == b0) ? 0xffff : 0x0`	`(a1 == b1) ? 0xffff : 0x0`	...	`(a7 == b7) ? 0xffff : 0x0`

_mm_cmpeq_epi32

```
__m128i _mm_cmpeq_epi32(__m128i a, __m128i b);
```

Compares the four signed or unsigned 32-bit integers in *a* and the four signed or unsigned 32-bit integers in *b* for equality.

R0	R1	R2	R3
`(a0 == b0) ? 0xffffffff : 0x0`	`(a1 == b1) ? 0xffffffff : 0x0`	`(a2 == b2) ? 0xffffffff : 0x0`	`(a3 == b3) ? 0xffffffff : 0x0`

_mm_cmpgt_epi8

```
__m128i _mm_cmpgt_epi8(__m128i a, __m128i b);
```

Compares the 16 signed 8-bit integers in *a* and the 16 signed 8-bit integers in *b* for greater than.

R0	R1	...	R15
`(a0 > b0) ? 0xff : 0x0`	`(a1 > b1) ? 0xff : 0x0`	...	`(a15 > b15) ? 0xff : 0x0`

_mm_cmpgt_epi16

```
__m128i _mm_cmpgt_epi16(__m128i a, __m128i b);
```

Compares the eight signed 16-bit integers in *a* and the eight signed 16-bit integers in *b* for greater than.

R0	R1	...	R7
`(a0 > b0) ? 0xffff : 0x0`	`(a1 > b1) ? 0xffff : 0x0`	...	`(a7 > b7) ? 0xffff : 0x0`

_mm_cmpgt_epi32

```
__m128i _mm_cmpgt_epi32(__m128i a, __m128i b);
```

Compares the four signed 32-bit integers in *a* and the four signed 32-bit integers in *b* for greater than.

R0	R1	R2	R3
`(a0 > b0) ? 0xffffffff : 0x0`	`(a1 > b1) ? 0xffffffff : 0x0`	`(a2 > b2) ? 0xffffffff : 0x0`	`(a3 > b3) ? 0xffffffff : 0x0`

_mm_cmplt_epi8

```
__m128i _mm_cmplt_epi8( __m128i a, __m128i b);
```

Compares the 16 signed 8-bit integers in *a* and the 16 signed 8-bit integers in *b* for less than.

R0	R1	...	R15
`(a0 < b0) ? 0xff : 0x0`	`(a1 < b1) ? 0xff : 0x0`	...	`(a15 < b15) ? 0xff : 0x0`

_mm_cmplt_epi16

```
__m128i _mm_cmplt_epi16( __m128i a, __m128i b);
```

Compares the eight signed 16-bit integers in *a* and the eight signed 16-bit integers in *b* for less than.

R0	R1	...	R7
`(a0 < b0) ? 0xffff : 0x0`	`(a1 < b1) ? 0xffff : 0x0`	...	`(a7 < b7) ? 0xffff : 0x0`

_mm_cmplt_epi32

```
__m128i _mm_cmplt_epi32( __m128i a, __m128i b);
```

Compares the four signed 32-bit integers in *a* and the four signed 32-bit integers in *b* for less than.

R0	R1	R2	R3
`(a0 < b0) ? 0xffffffff : 0x0`	`(a1 < b1) ? 0xffffffff : 0x0`	`(a2 < b2) ? 0xffffffff : 0x0`	`(a3 < b3) ? 0xffffffff : 0x0`

Conversion Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer conversion operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, `R1`, `R2`, and `R3` represent the registers in which results are placed.

Intrinsics marked with * are implemented only on Intel® 64 architecture. The rest of the intrinsics are implemented on both IA-32 and Intel® 64 architectures.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_cvtsi64_sd*`	Convert and pass through	`CVTSI2SD`
`_mm_cvtsd_si64*`	Convert according to rounding	`CVTSD2SI`
`_mm_cvttsd_si64*`	Convert using truncation	`CVTTSD2SI`
`_mm_cvtepi32_ps`	Convert to SP FP	None
`_mm_cvtps_epi32`	Convert from SP FP	None
`_mm_cvttps_epi32`	Convert from SP FP using truncate	None

_mm_cvtsi64_sd

```
__m128d _mm_cvtsi64_sd(__m128d a, __int64 b);
```

Converts the signed 64-bit integer value in *b* to a DP FP value. The upper DP FP value in *a* is passed through.

> **NOTE**
> Use only on Intel® 64 architectures.

R0	R1
`(double)b`	`a1`

_mm_cvtsd_si64

```
__int64 _mm_cvtsd_si64(__m128d a);
```

Converts the lower DP FP value of *a* to a 64-bit signed integer value according to the current rounding mode.

> **NOTE**
> Use only on Intel® 64 architectures.

R
`(__int64) a0`

_mm_cvttsd_si64

```
__int64 _mm_cvttsd_si64(__m128d a);
```

Converts the lower DP FP value of *a* to a 64-bit signed integer value using truncation.

> **NOTE**
> Use only on Intel® 64 architectures.

R
`(__int64) a0`

_mm_cvtepi32_ps

```
__m128 _mm_cvtepi32_ps(__m128i a);
```

Converts the four signed 32-bit integer values of *a* to SP FP values.

R0	R1	R2	R3
`(float) a0`	`(float) a1`	`(float) a2`	`(float) a3`

_mm_cvtps_epi32

```
__m128i _mm_cvtps_epi32(__m128 a);
```

Converts the four SP FP values of *a* to signed 32-bit integer values.

R0	R1	R2	R3
`(int) a0`	`(int) a1`	`(int) a2`	`(int) a3`

_mm_cvttps_epi32

```
__m128i _mm_cvttps_epi32(__m128 a);
```

Converts the four SP FP values of *a* to signed 32 bit integer values using truncate.

R0	R1	R2	R3
`(int) a0`	`(int) a1`	`(int) a2`	`(int) a3`

Move Intrinsics

The Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer move operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, `R1`, `R2` and `R3` represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_cvtsi32_si128`	Move and zero	`MOVD`
`_mm_cvtsi64_si128`	Move and zero	`MOVQ`
`_mm_cvtsi128_si32`	Move lowest 32 bits	`MOVD`
`_mm_cvtsi128_si64`	Move lowest 64 bits	`MOVQ`

_mm_cvtsi32_si128

`__m128i _mm_cvtsi32_si128(int a);`

Moves 32-bit integer *a* to the least significant 32 bits of an `__m128i` object. Zeroes the upper 96 bits of the `__m128i` object.

R0	R1	R2	R3
a	0x0	0x0	0x0

_mm_cvtsi64_si128

`__m128i _mm_cvtsi64_si128(__int64 a);`

Moves 64-bit integer *a* to the lower 64 bits of an `__m128i` object, zeroing the upper bits.

R0	R1
a	0x0

_mm_cvtsi128_si32

`int _mm_cvtsi128_si32(__m128i a);`

Moves the least significant 32 bits of *a* to a 32-bit integer.

R
a0

_mm_cvtsi128_si64

`__int64 _mm_cvtsi128_si64(__m128i a);`

Moves the lower 64 bits of *a* to a 64-bit integer.

R
a0

Load Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer load operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, and `R1` represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_load_si128`	Load	`MOVDQA`

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
_mm_loadu_si128	Load	MOVDQU
_mm_loadl_epi64	Load and zero	MOVQ

_mm_load_si128

`__m128i _mm_load_si128(__m128i const*p);`

Loads 128-bit value. Address *p* must be 16-byte aligned.

R
*p

_mm_loadu_si128

`__m128i _mm_loadu_si128(__m128i const*p);`

Loads 128-bit value. Address *p* not need be 16-byte aligned.

R
*p

_mm_loadl_epi64

`__m128i _mm_loadl_epi64(__m128i const*p);`

Load the lower 64 bits of the value pointed to by *p* into the lower 64 bits of the result, zeroing the upper 64 bits of the result.

R0	R1
*p[63:0]	0x0

Set Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer set operations are listed in this topic. These intrinsics are composite intrinsics because they require more than one instruction to implement them. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, `R1`...`R15` represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
_mm_set_epi64	Set two integer values	Composite

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_set_epi64x`	Set two integer values	`Composite`
`_mm_set_epi32`	Set four integer values	`Composite`
`_mm_set_epi16`	Set eight integer values	`Composite`
`_mm_set_epi8`	Set sixteen integer values	`Composite`
`_mm_set1_epi64`	Set two integer values	`Composite`
`_mm_set1_epi64x`	Set two integer values	`Composite`
`_mm_set1_epi32`	Set four integer values	`Composite`
`_mm_set1_epi16`	Set eight integer values	`Composite`
`_mm_set1_epi8`	Set sixteen integer values	`Composite`
`_mm_setr_epi64`	Set two integer values in reverse order	`Composite`
`_mm_setr_epi32`	Set four integer values in reverse order	`Composite`
`_mm_setr_epi16`	Set eight integer values in reverse order	`Composite`
`_mm_setr_epi8`	Set sixteen integer values in reverse order	`Composite`
`_mm_setzero_si128`	Set to zero	`Composite`

_mm_set_epi64

```
__m128i _mm_set_epi64(__m64 q1, __m64 q0);
```

Sets the two 64-bit integer values.

R0	R1
`q0`	`q1`

_mm_set_epi64x

```
__m128i _mm_set_epi64x(__int64 b, __int64 a);
```

Sets the two 64-bit integer values.

R0	R1
`a`	`b`

_mm_set_epi32

```
__m128i _mm_set_epi32(int i3, int i2, int i1, int i0);
```

Sets the four signed 32-bit integer values.

R0	R1	R2	R3
i0	i1	i2	i3

_mm_set_epi16

```
__m128i _mm_set_epi16(short w7, short w6, short w5, short w4, short w3, short w2, short
w1, short w0);
```

Sets the eight signed 16-bit integer values.

R0	R1	...	R7
w0	w1	...	w7

_mm_set_epi8

```
__m128i _mm_set_epi8(char b15, char b14, char b13, char b12, char b11, char b10, char
b9, char b8, char b7, char b6, char b5, char b4, char b3, char b2, char b1, char b0);
```

Sets the 16 signed 8-bit integer values.

R0	R1	...	R15
b0	b1	...	b15

_mm_set1_epi64

```
__m128i _mm_set1_epi64(__m64 q);
```

Sets the two 64-bit integer values to *q*.

R0	R1
q	q

_mm_set1_epi64x

```
__m128i _mm_set1_epi64x(__int64 a);
```

Sets the two 64-bit integer values to *a*.

R0	R1
a	a

_mm_set1_epi32

```
__m128i _mm_set1_epi32(int i);
```

Sets the four signed 32-bit integer values to *i*.

R0	R1	R2	R3
i	i	i	i

_mm_set1_epi16

```
__m128i _mm_set1_epi16(short w);
```

Sets the eight signed 16-bit integer values to *w*.

R0	R1	...	R7
w	w	w	w

_mm_set1_epi8

```
__m128i _mm_set1_epi8(char b);
```

Sets the 16 signed 8-bit integer values to *b*.

R0	R1	...	R15
b	b	b	b

_mm_setr_epi64

```
__m128i _mm_setr_epi64(__m64 q0, __m64 q1);
```

Sets the two 64-bit integer values in reverse order.

R0	R1
q0	q1

_mm_setr_epi32

```
__m128i _mm_setr_epi32(int i0, int i1, int i2, int i3);
```

Sets the four signed 32-bit integer values in reverse order.

R0	R1	R2	R3
i0	i1	i2	i3

_mm_setr_epi16

```
__m128i _mm_setr_epi16(short w0, short w1, short w2, short w3, short w4, short w5,
short w6, short w7);
```

Sets the eight signed 16-bit integer values in reverse order.

R0	R1	...	R7
w0	w1	...	w7

_mm_setr_epi8

```
__m128i _mm_setr_epi8(char b0, char b1, char b2, char b3, char b4, char b5, char b6,
char b7, char b8, char b9, char b10, char b11, char b12, char b13, char b14, char b15);
```

Sets the 16 signed 8-bit integer values in reverse order.

R0	R1	...	R15
b0	b1	...	b15

_mm_setzero_si128

```
__m128i _mm_setzero_si128();
```

Sets the 128-bit value to zero.

R
0x0

Store Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer store operations are listed in this topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The detailed description of each intrinsic contains a table detailing the returns. In these tables, *p* is an access to the result.

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
_mm_stream_si128	Store	MOVNTDQ
_mm_stream_si32	Store	MOVNTI
_mm_store_si128	Store	MOVDQA
_mm_storeu_si128	Store	MOVDQU
_mm_maskmoveu_si128	Conditional store	MASKMOVDQU
_mm_storel_epi64	Store lowest	MOVQ

_mm_stream_si128

```
void _mm_stream_si128(__m128i *p, __m128i a);
```

Stores the data in *a* to the address *p* without polluting the caches. If the cache line containing address *p* is already in the cache, the cache will be updated. Address *p* must be 16 byte aligned.

*p
a

_mm_stream_si32

```
void _mm_stream_si32(int *p, int a);
```

Stores the data in *a* to the address *p* without polluting the caches. If the cache line containing address *p* is already in the cache, the cache will be updated.

*p
a

_mm_store_si128

```
void _mm_store_si128(__m128i *p, __m128i b);
```

Stores 128-bit value. Address *p* must be 16 byte aligned.

*p
a

_mm_storeu_si128

```
void _mm_storeu_si128(__m128i *p, __m128i b);
```

Stores 128-bit value. Address *p* need not be 16-byte aligned.

*p
a

_mm_maskmoveu_si128

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p);
```

Conditionally store byte elements of *d* to address *p*. The high bit of each byte in the selector *n* determines whether the corresponding byte in *d* will be stored. Address *p* need not be 16-byte aligned.

if (n0[7])	if (n1[7]	...	if (n15[7])
p[0] := d0	p[1] := d1	...	p[15] := d15

_mm_storel_epi64

```
void _mm_storel_epi64(__m128i *p, __m128i a);
```

Stores the lower 64 bits of the value pointed to by *p*.

*p[63:0]
a0

Miscellaneous Functions and Intrinsics

Cacheability Support Intrinsics

The prototypes for Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for cacheability support are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_stream_pd`	Store	`MOVNTPD`
`_mm256_stream_pd`	Store	`VMOVNTPD`
`_mm_stream_si128`	Store	`MOVNTDQ`
`_mm256_stream_si256`	Store	`VMOVNTDQ`
`_mm_stream_si32`	Store	`MOVNTI`
`_mm_stream_si64*`	Store	`MOVNTI`
`_mm_clflush`	Flush	`CLFLUSH`
`_mm_clflushopt`	Flush	`CLFLUSHOPT`
`_mm_lfence`	Guarantee visibility	`LFENCE`
`_mm_mfence`	Guarantee visibility	`MFENCE`

_mm_stream_pd

```
void _mm_stream_pd(double *p, __m128d a);
```

Stores the data in *a* to the address *p* without polluting caches. The address *p* must be 16-byte (128-bit version) aligned. If the cache line containing address *p* is already in the cache, the cache will be updated. `p[0] := a0 p[1] := a1`

p[0]	p[1]
a0	a1

_mm256_stream_pd

```
void _mm256_stream_pd(double *p, __m256d a);
```

Stores the data in *a* to the address *p* without polluting caches. The address *p* must be 32-byte (VEX.256 encoded version) aligned. If the cache line containing address *p* is already in the cache, the cache will be updated. `p[0] := a0 p[1] := a1`

p[0]	p[1]
a0	a1

_mm_stream_si128

```
void _mm_stream_si128(__m128i *p, __m128i a);
```

Stores the data in *a* to the address *p* without polluting the caches. If the cache line containing address *p* is already in the cache, the cache will be updated. Address *p* must be 16-byte (128-bit version) aligned.

*p
a

_mm256_stream_si256

```
void _mm256_stream_si256(__m256i *p, __m256i a);
```

Stores the data in *a* to the address *p* without polluting the caches. If the cache line containing address *p* is already in the cache, the cache will be updated. Address *p* must be 32-byte (VEX.256 encoded version) aligned.

*p
a

_mm_stream_si32

```
void _mm_stream_si32(int *p, int a);
```

Stores the 32-bit integer data in *a* to the address *p* without polluting the caches. If the cache line containing address *p* is already in the cache, the cache will be updated.

*p
a

_mm_stream_si64

```
void _mm_stream_si64(__int64 *p, __int64 a);
```

Stores the 64-bit integer data in *a* to the address *p* without polluting the caches. If the cache line containing address *p* is already in the cache, the cache is updated.

*p
a

_mm_clflush

```
void _mm_clflush(void const*p);
```

Cache line containing *p* is flushed and invalidated from all caches in the coherency domain.

*p
a

_mm_clflushopt

```
void _mm_clflushopt(void const *p);
```

Cache line containing *p* is flushed and invalidated from all caches in the coherency domain. This optimized version of the _mm_clflush is available if indicated by the CPUID feature flag `CLFLUSHOPT`.

*p
a

_mm_lfence

```
void _mm_lfence(void);
```

Guarantees that every load instruction that precedes, in program order, the load fence instruction is globally visible before any load instruction which follows the fence in program order.

_mm_mfence

```
void _mm_mfence(void);
```

Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction which follows the fence in program order.

Miscellaneous Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for miscellaneous operations are listed in the following table followed by descriptions.

The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic	Operation	Corresponding Intel® SSE 2 Instruction
_mm_packs_epi16	Packed Saturation	PACKSSWB
_mm_packs_epi32	Packed Saturation	PACKSSDW
_mm_packus_epi16	Packed Saturation	PACKUSWB
_mm_extract_epi16	Extraction	PEXTRW
_mm_insert_epi16	Insertion	PINSRW
_mm_movemask_epi8	Mask Creation	PMOVMSKB
_mm_shuffle_epi32	Shuffle	PSHUFD
_mm_shufflehi_epi16	Shuffle	PSHUFHW
_mm_shufflelo_epi16	Shuffle	PSHUFLW
_mm_unpackhi_epi8	Interleave	PUNPCKHBW
_mm_unpackhi_epi16	Interleave	PUNPCKHWD

Intrinsic	Operation	Corresponding Intel® SSE 2 Instruction
`_mm_unpackhi_epi32`	Interleave	`PUNPCKHDQ`
`_mm_unpackhi_epi64`	Interleave	`PUNPCKHQDQ`
`_mm_unpacklo_epi8`	Interleave	`PUNPCKLBW`
`_mm_unpacklo_epi16`	Interleave	`PUNPCKLWD`
`_mm_unpacklo_epi32`	Interleave	`PUNPCKLDQ`
`_mm_unpacklo_epi64`	Interleave	`PUNPCKLQDQ`
`_mm_movepi64_pi64`	Move	`MOVDQ2Q`
`_mm_movpi64_epi64`	Move	`MOVDQ2Q`
`_mm_move_epi64`	Move	`MOVQ`
`_mm_unpackhi_pd`	Interleave	`UNPCKHPD`
`_mm_unpacklo_pd`	Interleave	`UNPCKLPD`
`_mm_movemask_pd`	Create mask	`MOVMSKPD`
`_mm_shuffle_pd`	Select values	`SHUFPD`

_mm_packs_epi16

```
__m128i _mm_packs_epi16(__m128i a, __m128i b);
```

Packs the 16 signed 16-bit integers from *a* and *b* into 8-bit integers and saturates.

R0	...	R7	R8	...	R15
`Signed Saturate(a0)`	`...`	`Signed Saturate(a7)`	`Signed Saturate(b0)`	`...`	`Signed Saturate(b7)`

_mm_packs_epi32

```
__m128i _mm_packs_epi32(__m128i a, __m128i b);
```

Packs the eight signed 32-bit integers from *a* and *b* into signed 16-bit integers and saturates.

R0	...	R3	R4	...	R7
`Signed Saturate(a0)`	`...`	`Signed Saturate(a3)`	`Signed Saturate(b0)`	`...`	`Signed Saturate(b3)`

_mm_packus_epi16

```
__m128i _mm_packus_epi16(__m128i a, __m128i b);
```

Packs the 16 signed 16-bit integers from *a* and *b* into 8-bit unsigned integers and saturates.

R0	...	R7	R8	...	R15
Unsigned Saturate(a0)	...	Unsigned Saturate(a7)	Unsigned Saturate(b0)	...	Unsigned Saturate(b15)

_mm_extract_epi16

```
int _mm_extract_epi16(__m128i a, int imm);
```

Extracts the selected signed or unsigned 16-bit integer from *a* and zero extends. The selector *imm* must be an immediate.

R0
(imm == 0) ? a0: ((imm == 1) ? a1: ... (imm==7) ? a7)

_mm_insert_epi16

```
__m128i _mm_insert_epi16(__m128i a, int b, int imm);
```

Inserts the least significant 16 bits of *b* into the selected 16-bit integer of *a*. The selector *imm* must be an immediate.

R0	R1	...	R7
(imm == 0) ? b : a0;	(imm == 1) ? b : a1;	...	(imm == 7) ? b : a7;

_mm_movemask_epi8

```
int _mm_movemask_epi8(__m128i a);
```

Creates a 16-bit mask from the most significant bits of the 16 signed or unsigned 8-bit integers in *a* and zero extends the upper bits.

R0
a15[7] << 15 | a14[7] << 14 | ... a1[7] << 1 | a0[7]

_mm_shuffle_epi32

```
__m128i _mm_shuffle_epi32(__m128i a, int imm);
```

Shuffles the four signed or unsigned 32-bit integers in *a* as specified by *imm*. The shuffle value, *imm*, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

_mm_shufflehi_epi16

```
__m128i _mm_shufflehi_epi16(__m128i a, int imm);
```

Shuffles the upper four signed or unsigned 16-bit integers in *a* as specified by *imm*. The shuffle value, *imm*, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

_mm_shufflelo_epi16

```
__m128i _mm_shufflelo_epi16(__m128i a, int imm);
```

Shuffles the lower four signed or unsigned 16-bit integers in *a* as specified by *imm*. The shuffle value, *imm*, must be an immediate. See Macro Function for Shuffle for a description of shuffle semantics.

_mm_unpackhi_epi8

```
__m128i _mm_unpackhi_epi8(__m128i a, __m128i b);
```

Interleaves the upper eight signed or unsigned 8-bit integers in *a* with the upper eight signed or unsigned 8-bit integers in *b*.

R0	R1	R2	R3	...	R14	R15
a8	b8	a9	b9	...	a15	b15

_mm_unpackhi_epi16

```
__m128i _mm_unpackhi_epi16(__m128i a, __m128i b);
```

Interleaves the upper four signed or unsigned 16-bit integers in *a* with the upper four signed or unsigned 16-bit integers in *b*.

R0	R1	R2	R3	R4	R5	R6	R7
a4	b4	a5	b5	a6	b6	a7	b7

_mm_unpackhi_epi32

```
__m128i _mm_unpackhi_epi32(__m128i a, __m128i b);
```

Interleaves the upper two signed or unsigned 32-bit integers in *a* with the upper two signed or unsigned 32-bit integers in *b*.

R0	R1	R2	R3
a2	b2	a3	b3

_mm_unpackhi_epi64

```
__m128i _mm_unpackhi_epi64(__m128i a, __m128i b);
```

Interleaves the upper signed or unsigned 64-bit integer in *a* with the upper signed or unsigned 64-bit integer in *b*.

R0	R1
a1	b1

_mm_unpacklo_epi8

```
__m128i _mm_unpacklo_epi8(__m128i a, __m128i b);
```

Interleaves the lower eight signed or unsigned 8-bit integers in *a* with the lower eight signed or unsigned 8-bit integers in *b*.

R0	R1	R2	R3	...	R14	R15
a0	b0	a1	b1	...	a7	b7

_mm_unpacklo_epi16

```
__m128i _mm_unpacklo_epi16(__m128i a, __m128i b);
```

Interleaves the lower four signed or unsigned 16-bit integers in *a* with the lower four signed or unsigned 16-bit integers in *b*.

R0	R1	R2	R3	R4	R5	R6	R7
a0	b0	a1	b1	a2	b2	a3	b3

_mm_unpacklo_epi32

```
__m128i _mm_unpacklo_epi32(__m128i a, __m128i b);
```

Interleaves the lower two signed or unsigned 32-bit integers in *a* with the lower two signed or unsigned 32-bit integers in *b*.

R0	R1	R2	R3
a0	b0	a1	b1

_mm_unpacklo_epi64

```
__m128i _mm_unpacklo_epi64(__m128i a, __m128i b);
```

Interleaves the lower signed or unsigned 64-bit integer in *a* with the lower signed or unsigned 64-bit integer in *b*.

R0	R1
a0	b0

_mm_movepi64_pi64

```
__m64 _mm_movepi64_pi64(__m128i a);
```

Returns the lower 64 bits of *a* as an `__m64` type.

R0
a0

_mm_movpi64_pi64

```
__m128i _mm_movpi64_pi64(__m64 a);
```

Moves the 64 bits of *a* to the lower 64 bits of the result, zeroing the upper bits.

R0	R1
a0	0X0

_mm_move_epi64

```
__m128i _mm_move_epi64(__m128i a);
```

Moves the lower 64 bits of *a* to the lower 64 bits of the result, zeroing the upper bits.

R0	R1
a0	0X0

_mm_unpackhi_pd

```
__m128d _mm_unpackhi_pd(__m128d a, __m128d b);
```

Interleaves the upper DP FP values of *a* and *b*.

R0	R1
a1	b1

_mm_unpacklo_pd

```
__m128d _mm_unpacklo_pd(__m128d a, __m128d b);
```

Interleaves the lower DP FP values of *a* and *b*.

R0	R1
a0	b0

_mm_movemask_pd

```
int _mm_movemask_pd(__m128d a);
```

Creates a two-bit mask from the sign bits of the two DP FP values of *a*.

R
sign(a1) << 1 | sign(a0)

_mm_shuffle_pd

```
__m128d _mm_shuffle_pd(__m128d a, __m128d b, int i)
```

Selects two specific DP FP values from *a* and *b*, based on the mask *i*. The mask must be an immediate. See Macro Function for Shuffle for a description of the shuffle semantics.

Casting Support Intrinsics

Intel® C++ Compiler supports casting between various single-precision, double-precision, and integer vector types. These intrinsics do not convert values; they change one data type to another without changing the value.

The intrinsics for casting support do not have any corresponding Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instructions. The syntax for the Casting Support intrinsics are as follows:

`__m128 _mm_castpd_ps(__m128d in);`

`__m128i _mm_castpd_si128(__m128d in);`

`__m128d _mm_castps_pd(__m128 in);`

`__m128i _mm_castps_si128(__m128 in);`

`__m128 _mm_castsi128_ps(__m128i in);`

`__m128d _mm_castsi128_pd(__m128i in);`

Pause Intrinsic

The prototype for this Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsic is in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

`PAUSE` Intrinsic

`void _mm_pause(void);`

The `pause` intrinsic is used in spin-wait loops with the processors implementing dynamic execution (especially out-of-order execution). In the spin-wait loop, the `pause` intrinsic improves the speed at which the code detects the release of the lock and provides especially significant performance gain.

The execution of the next instruction is delayed for an implementation-specific amount of time. The `PAUSE` instruction does not modify the architectural state. For dynamic scheduling, the `PAUSE` instruction reduces the penalty of exiting from the spin-loop.

Example of loop with the PAUSE instruction:

In this example, the program spins until memory location `A` matches the value in register `eax`. The code sequence that follows shows a test-and-test-and-set.

```
spin_loop:pause
cmp eax, A
jne spin_loop
```

In this example, the spin occurs only after the attempt to get a lock has failed.

```
get_lock: mov eax, 1
xchg eax, A ; Try to get lock
cmp eax, 0 ; Test if successful
jne spin_loop
```

Critical Section

```
// critical_section code
mov A, 0 ; Release lock
jmp continue
spin_loop: pause;
// spin-loop hint
cmp 0, A ;
// check lock availability
```

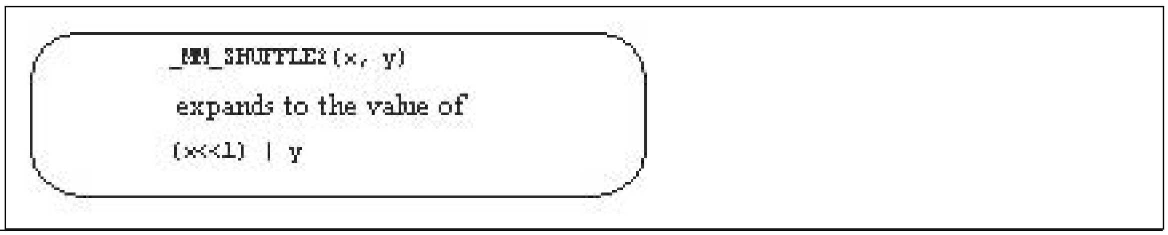
```
jne spin_loop
jmp get_lock
// continue: other code
```

NOTE
The first branch is predicted to fall-through to the critical section in anticipation of successfully gaining access to the lock. It is highly recommended that all spin-wait loops include the `PAUSE` instruction. Since `PAUSE` is backwards compatible to all existing IA-32 architecture-based processor generations, a test for processor type (a CPUID test) is not needed. All legacy processors execute `PAUSE` instruction as a `NOP`, but in processors that use the `PAUSE` instruction as a hint there can be significant performance benefit.

Macro Function for Shuffle

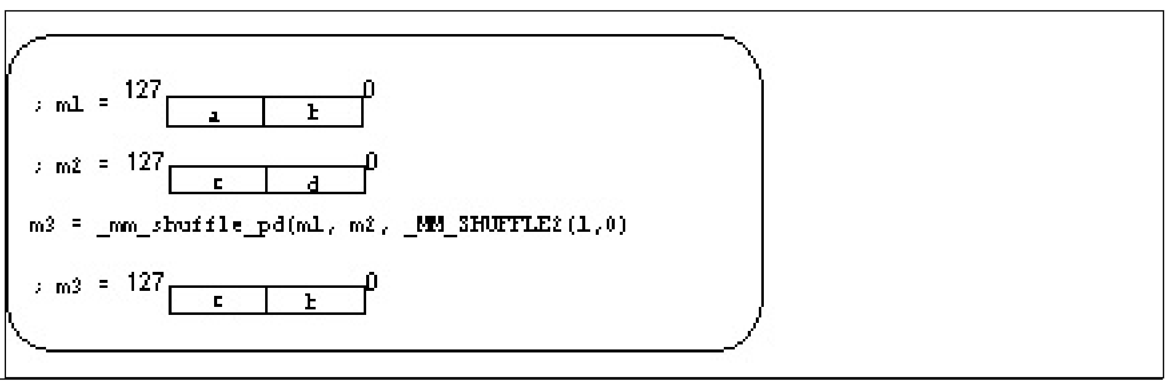
Intel® Streaming SIMD Extensions 2 (Intel® SSE2) provide a macro function to help create constants that describe shuffle operations. The macro takes two small integers (in the range of 0 to 1) and combines them into an 2-bit immediate value used by the `SHUFPD` instruction. See the following example.

Shuffle Function Macro



You can view the two integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word.

View of Original and Result Words with Shuffle Function Macro



Intrinsics Returning Vectors of Undefined Values

These intrinsics generate vectors of undefined values. The result of the intrinsics is usually used as an argument to another intrinsic that requires all operands to be initialized, and when the content of a particular argument does not matter.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

For example, you can use such an intrinsic when you need to calculate a sum of packed double-precision floating-point values located in the *xmm* register. To avoid unnecessary moves, you can use the following code to obtain the required result at the low 64 bits:

```
__m128d HILO    = doSomeWork();
__m128d HI = _mm_unpackhi_pd(HILO, _mm_undefined_pd());
__m128d result    = _mm_add_sd(HI, HILO);
```

_mm_undefined_pd

`extern __m128d _mm_undefined_pd(void);`

Returns a vector of two double precision floating point elements. The content of the vector is not specified.

_mm_undefined_si128

`extern __m128i _mm_undefined_si128(void);`

Returns a vector of four packed doubleword integer elements. The content of the vector is not specified.

See Also

_mm256_undefined_si128() Returns a vector of eight packed doubleword integer elements. No corresponding Intel® AVX instruction.

_mm256_undefined_pd() Returns a vector of four double precision floating point elements. No corresponding Intel® AVX instruction.

Intrinsics for Intel® Streaming SIMD Extensions (Intel® SSE)

This section describes the C++ language-level features supporting the Intel® Streaming SIMD Extensions (Intel® SSE) in the Intel® C++ Compiler Classic. The prototypes for Intel® SSE intrinsics are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The following topics summarize these intrinsics.

Details about Intel® Streaming SIMD Extensions Intrinsics

Intel® Streaming SIMD Extensions (Intel® SSE) instructions use the following features:

- Registers – Enable packed data of up to 128 bits in length for optimal SIMD processing
- Data Types – Enable packing of up to 16 elements of data in one register

Registers

Intel® Streaming SIMD Extensions use eight 128-bit registers (`XMM0` to `XMM7`).

Because each of these registers can hold more than one data element, the processor can process more than one data element simultaneously. This processing capability is also known as single-instruction multiple data processing (SIMD).

For each computational and data manipulation instruction in the new extension sets, there is a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

NOTE
The MM and XMM registers are the SIMD registers used by the systems based on IA-32 architecture to implement MMX™ technology and Intel® SSE or Intel® Streaming SIMD Extensions 2 (Intel® SSE2).

Data Types

These intrinsic functions use four new C data types as operands, representing the new registers that are used as the operands to these intrinsic functions.

New Data Types

The following table details for which instructions each of the new data types are available.

New Data Type	Intel® Streaming SIMD Extensions Intrinsics	Intel® Streaming SIMD Extensions 2 Intrinsics	Intel® Streaming SIMD Extensions 3 Intrinsics
__m64	Available	Available	Available
__m128	Available	Available	Available
__m128d	Not available	Available	Available
__m128i	Not available	Available	Available

__m128 Data Types

The __m128 data type is used to represent the contents of a Intel® SSE register used by Intel® SSE intrinsics. The __m128 data type can hold four 32-bit floating-point values.

The __m128d data type can hold two 64-bit floating-point values.

The __m128i data type can hold sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integer values.

The compiler aligns __m128d and __m128ilocal and global data to 16-byte boundaries on the stack. To align integer, float, or double arrays, you can use the __declspec(align) statement.

Data Types Usage Guidelines

These data types are not basic ANSI C data types. You must observe the following usage restrictions:

- Use data types only on either side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions (+, -, etc).
- Use data types as objects in aggregates, such as unions, to access the byte elements and structures.
- Use data types only with the respective intrinsics described in this documentation.

Accessing __m128i Data

To access 8-bit data:

```
#define  _mm_extract_epi8(x, imm) \
((((imm) & 0x1) == 0) ?   \
 _mm_extract_epi16((x), (imm) >> 1) & 0xff : \
 _mm_extract_epi16( _mm_srli_epi16((x), 8), (imm) >> 1))
```

For 16-bit data, use the following intrinsic:

```
int  _mm_extract_epi16(__m128i a, int imm)
```

To access 32-bit data:

```
#define _mm_extract_epi32(x, imm) \
 _mm_cvtsi128_si32( _mm_srli_si128((x), 4 * (imm)))
```

See Also

`__declspec(align)` declaration

Writing Programs with Intel® Streaming SIMD Extensions (Intel® SSE) Intrinsics

You should be familiar with the hardware features provided by Intel® Streaming SIMD Extensions (Intel® SSE) when writing programs with the intrinsics. The following are four important issues to keep in mind:

- Certain intrinsics, such as `_mm_loadr_ps` and `_mm_cmpgt_ss`, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful that they may consist of more than one machine-language instruction.
- Floating-point data loaded or stored as `__m128` objects must be generally 16-byte-aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.
- The result of arithmetic operations acting on two `NaN` (Not a Number) arguments is undefined. Therefore, FP operations using `NaN` arguments will not match the expected behavior of the corresponding assembly instructions.

Arithmetic Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for arithmetic operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. R0, R1, R2, and R3 each represent one of the four 32-bit pieces of the result register.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_add_ss`	Addition	`ADDSS`
`_mm_add_ps`	Addition	`ADDPS`
`_mm_sub_ss`	Subtraction	`SUBSS`
`_mm_sub_ps`	Subtraction	`SUBPS`
`_mm_mul_ss`	Multiplication	`MULSS`
`_mm_mul_ps`	Multiplication	`MULPS`
`_mm_div_ss`	Division	`DIVSS`
`_mm_div_ps`	Division	`DIVPS`
`_mm_sqrt_ss`	Squared Root	`SQRTSS`

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_sqrt_ps`	Squared Root	`SQRTPS`
`_mm_rcp_ss`	Reciprocal	`RCPSS`
`_mm_rcp_ps`	Reciprocal	`RCPPS`
`_mm_rsqrt_ss`	Reciprocal Squared Root	`RSQRTSS`
`_mm_rsqrt_ps`	Reciprocal Squared Root	`RSQRTPS`
`_mm_min_ss`	Computes Minimum	`MINSS`
`_mm_min_ps`	Computes Minimum	`MINPS`
`_mm_max_ss`	Computes Maximum	`MAXSS`
`_mm_max_ps`	Computes Maximum	`MAXPS`

_mm_add_ss

```
__m128 _mm_add_ss(__m128 a, __m128 b);
```

Adds the lower single-precision, floating-point (FP) values of *a* and *b*; the upper three single-precision FP values are passed through from *a*.

R0	R1	R2	R3
a0 + b0	a1	a2	a3

_mm_add_ps

```
__m128 _mm_add_ps(__m128 a, __m128 b);
```

Adds the four single-precision FP values of *a* and *b*.

R0	R1	R2	R3
a0 +b0	`a1 + b1`	`a2 + b2`	`a3 + b3`

_mm_sub_ss

```
__m128 _mm_sub_ss(__m128 a, __m128 b);
```

Subtracts the lower single-precision FP values of *a* and *b*. The upper three single-precision FP values are passed through from *a*.

R0	R1	R2	R3
a0 - b0	a1	a2	a3

_mm_sub_ps

```
__m128 _mm_sub_ps(__m128 a, __m128 b);
```

Subtracts the four single-precision FP values of *a* and *b*.

R0	R1	R2	R3
a0 - b0	a1 - b1	a2 - b2	a3 - b3

_mm_mul_ss

```
__m128 _mm_mul_ss(__m128 a, __m128 b);
```

Multiplies the lower single-precision FP values of *a* and *b*; the upper three single-precision FP values are passed through from *a*.

R0	R1	R2	R3
a0 * b0	a1	a2	a3

_mm_mul_ps

```
__m128 _mm_mul_ps(__m128 a, __m128 b);
```

Multiplies the four single-precision FP values of *a* and *b*.

R0	R1	R2	R3
a0 * b0	a1 * b1	a2 * b2	a3 * b3

_mm_div_ss

```
__m128 _mm_div_ss(__m128 a, __m128 b);
```

Divides the lower single-precision FP values of *a* and *b*; the upper three single-precision FP values are passed through from *a*.

R0	R1	R2	R3
a0 / b0	a1	a2	a3

_mm_div_ps

```
__m128 _mm_div_ps(__m128 a, __m128 b);
```

Divides the four single-precision FP values of *a* and *b*.

R0	R1	R2	R3
a0 / b0	a1 / b1	a2 / b2	a3 / b3

_mm_sqrt_ss

```
__m128 _mm_sqrt_ss(__m128 a);
```

Computes the square root of the lower single-precision FP value of *a* ; the upper three single-precision FP values are passed through.

R0	R1	R2	R3
`sqrt(a0)`	a1	a2	a3

_mm_sqrt_ps

```
__m128 _mm_sqrt_ps(__m128 a);
```

Computes the square roots of the four single-precision FP values of *a*.

R0	R1	R2	R3
`sqrt(a0)`	`sqrt(a1)`	`sqrt(a2)`	`sqrt(a3)`

_mm_rcp_ss

```
__m128 _mm_rcp_ss(__m128 a);
```

Computes the approximation of the reciprocal of the lower single-precision FP value of *a*; the upper the single-precision FP values are passed through.

R0	R1	R2	R3
`recip(a0)`	a1	a2	a3

_mm_rcp_ps

```
__m128 _mm_rcp_ps(__m128 a);
```

Computes the approximations of reciprocals of the four single-precision FP values of *a*.

R0	R1	R2	R3
`recip(a0)`	`recip(a1)`	`recip(a2)`	`recip(a3)`

_mm_rsqrt_ss

```
__m128 _mm_rsqrt_ss(__m128 a);
```

Computes the approximation of the reciprocal of the square root of the lower single-precision FP value of *a*; the upper three single-precision FP values are passed through.

R0	R1	R2	R3
`recip(sqrt(a0))`	a1	a2	a3

_mm_rsqrt_ps

```
__m128 _mm_rsqrt_ps(__m128 a);
```

Computes the approximations of the reciprocals of the square roots of the four single-precision FP values of *a*.

R0	R1	R2	R3
`recip(sqrt(a0))`	`recip(sqrt(a1))`	`recip(sqrt(a2))`	`recip(sqrt(a3))`

_mm_min_ss

```
__m128 _mm_min_ss(__m128 a, __m128 b);
```

Computes the minimum of the lower single-precision FP values of *a* and *b*; the upper three single-precision FP values are passed through from *a*.

R0	R1	R2	R3
`min(a0, b0)`	a1	a2	a3

_mm_min_ps

```
__m128 _mm_min_ps(__m128 a, __m128 b);
```

Computes the minimum of the four single-precision FP values of *a* and *b*.

R0	R1	R2	R3
`min(a0, b0)`	`min(a1, b1)`	`min(a2, b2)`	`min(a3, b3)`

_mm_max_ss

```
__m128 _mm_max_ss(__m128 a, __m128 b);
```

Computes the maximum of the lower single-precision FP values of *a* and *b*; the upper three single-precision FP values are passed through from *a*.

R0	R1	R2	R3
`max(a0, b0)`	a1	a2	a3

_mm_max_ps

```
__m128 _mm_max_ps(__m128 a, __m128 b);
```

Computes the maximum of the four single-precision FP values of *a* and *b*.

R0	R1	R2	R3
`max(a0, b0)`	`max(a1, b1)`	`max(a2, b2)`	`max(a3, b3)`

Logical Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for logical operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. R0, R1, R2, and R3 each represent one of the four 32-bit pieces of the result register.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_and_ps`	Bitwise AND	`ANDPS`
`_mm_andnot_ps`	Bitwise ANDNOT	`ANDNPS`
`_mm_or_ps`	Bitwise OR	`ORPS`
`_mm_xor_ps`	Bitwise Exclusive OR	`XORPS`

_mm_and_ps

```
__m128 _mm_and_ps(__m128 a, __m128 b);
```

Computes the bitwise AND of the four SP FP values of *a* and *b*.

R0	R1	R2	R3
`a0 & b0`	`a1 & b1`	`a2 & b2`	`a3 & b3`

_mm_andnot_ps

```
__m128 _mm_andnot_ps(__m128 a, __m128 b);
```

Computes the bitwise AND-NOT of the four SP FP values of *a* and *b*.

R0	R1	R2	R3
`~a0 & b0`	`~a1 & b1`	`~a2 & b2`	`~a3 & b3`

_mm_or_ps

```
__m128 _mm_or_ps(__m128 a, __m128 b);
```

Computes the bitwise OR of the four SP FP values of *a* and *b*.

R0	R1	R2	R3
`a0 | b0`	`a1 | b1`	`a2 | b2`	`a3 | b3`

_mm_xor_ps

```
__m128 _mm_xor_ps(__m128 a, __m128 b);
```

Computes bitwise XOR (exclusive-or) of the four SP FP values of *a* and *b*.

R0	R1	R2	R3
`a0 ^ b0`	`a1 ^ b1`	`a2 ^ b2`	`a3 ^ b3`

Compare Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for comparison operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Each comparison intrinsic performs a comparison of `a` and `b`. For the packed form, the four single-precision FP values of *a* and *b* are compared, and a 128-bit mask is returned. For the scalar form, the lower single-precision FP values of *a* and *b* are compared, and a 32-bit mask is returned; the upper three single-precision FP values are passed through from *a*. The mask is set to `0xffffffff` for each element where the comparison is true and `0x0` where the comparison is false.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R or R0-R3. R0, R1, R2, and R3 each represent one of the four 32-bit pieces of the result register.

Intrinsic Name	**Operation**	**Corresponding Intel® SSE Instruction**
`_mm_cmpeq_ss`	Equal	`CMPEQSS`
`_mm_cmpeq_ps`	Equal	`CMPEQPS`
`_mm_cmplt_ss`	Less Than	`CMPLTSS`
`_mm_cmplt_ps`	Less Than	`CMPLTPS`
`_mm_cmple_ss`	Less Than or Equal	`CMPLESS`
`_mm_cmple_ps`	Less Than or Equal	`CMPLEPS`
`_mm_cmpgt_ss`	Greater Than	`CMPLTSS`
`_mm_cmpgt_ps`	Greater Than	`CMPLTPS`
`_mm_cmpge_ss`	Greater Than or Equal	`CMPLESS`
`_mm_cmpge_ps`	Greater Than or Equal	`CMPLEPS`
`_mm_cmpneq_ss`	Not Equal	`CMPNEQSS`
`_mm_cmpneq_ps`	Not Equal	`CMPNEQPS`
`_mm_cmpnlt_ss`	Not Less Than	`CMPNLTSS`
`_mm_cmpnlt_ps`	Not Less Than	`CMPNLTPS`
`_mm_cmpnle_ss`	Not Less Than or Equal	`CMPNLESS`
`_mm_cmpnle_ps`	Not Less Than or Equal	`CMPNLEPS`
`_mm_cmpngt_ss`	Not Greater Than	`CMPNLTSS`
`_mm_cmpngt_ps`	Not Greater Than	`CMPNLTPS`
`_mm_cmpnge_ss`	Not Greater Than or Equal	`CMPNLESS`
`_mm_cmpnge_ps`	Not Greater Than or Equal	`CMPNLEPS`
`_mm_cmpord_ss`	Ordered	`CMPORDSS`
`_mm_cmpord_ps`	Ordered	`CMPORDPS`
`_mm_cmpunord_ss`	Unordered	`CMPUNORDSS`

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_cmpunord_ps`	Unordered	`CMPUNORDPS`
`_mm_comieq_ss`	Equal	`COMISS`
`_mm_comilt_ss`	Less Than	`COMISS`
`_mm_comile_ss`	Less Than or Equal	`COMISS`
`_mm_comigt_ss`	Greater Than	`COMISS`
`_mm_comige_ss`	Greater Than or Equal	`COMISS`
`_mm_comineq_ss`	Not Equal	`COMISS`
`_mm_ucomieq_ss`	Equal	`UCOMISS`
`_mm_ucomilt_ss`	Less Than	`UCOMISS`
`_mm_ucomile_ss`	Less Than or Equal	`UCOMISS`
`_mm_ucomigt_ss`	Greater Than	`UCOMISS`
`_mm_ucomige_ss`	Greater Than or Equal	`UCOMISS`
`_mm_ucomineq_ss`	Not Equal	`UCOMISS`

_mm_cmpeq_ss

```
__m128 __cdecl _mm_cmpeq_ss(__m128 a, __m128 b);
```

Compares for equality.

R0	R1	R2	R3
`(a0 == b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpeq_ps

```
__m128 _mm_cmpeq_ps(__m128 a, __m128 b);
```

Compares for equality.

R0	R1	R2	R3
`(a0 == b0) ? 0xffffffff : 0x0`	`(a1 == b1) ? 0xffffffff : 0x0`	`(a2 == b2) ? 0xffffffff : 0x0`	`(a3 == b3) ? 0xffffffff : 0x0`

_mm_cmplt_ss

```
__m128 _mm_cmplt_ss(__m128 a, __m128 b);
```

Compares for less-than.

R0	R1	R2	R3
`(a0 < b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmplt_ps

```
__m128 _mm_cmplt_ps(__m128 a, __m128 b);
```

Compares for less-than.

R0	R1	R2	R3
`(a0 < b0) ? 0xffffffff : 0x0`	`(a1 < b1) ? 0xffffffff : 0x0`	`(a2 < b2) ? 0xffffffff : 0x0`	`(a3 < b3) ? 0xffffffff : 0x0`

_mm_cmple_ss

```
__m128 _mm_cmple_ss(__m128 a, __m128 b);
```

Compares for less-than-or-equal.

R0	R1	R2	R3
`(a0 <= b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmple_ps

```
__m128 _mm_cmple_ps(__m128 a, __m128 b);
```

Compares for less-than-or-equal.

R0	R1	R2	R3
`(a0 <= b0) ? 0xffffffff : 0x0`	`(a1 <= b1) ? 0xffffffff : 0x0`	`(a2 <= b2) ? 0xffffffff : 0x0`	`(a3 <= b3) ? 0xffffffff : 0x0`

_mm_cmpgt_ss

```
__m128 _mm_cmpgt_ss(__m128 a, __m128 b);
```

Compares for greater-than.

R0	R1	R2	R3
`(a0 > b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpgt_ps

```
__m128 _mm_cmpgt_ps(__m128 a, __m128 b);
```

Compares for greater-than.

R0	R1	R2	R3
`(a0 > b0) ? 0xffffffff : 0x0`	`(a1 > b1) ? 0xffffffff : 0x0`	`(a2 > b2) ? 0xffffffff : 0x0`	`(a3 > b3) ? 0xffffffff : 0x0`

_mm_cmpge_ss

```
__m128 _mm_cmpge_ss(__m128 a, __m128 b);
```

Compares for greater-than-or-equal.

R0	R1	R2	R3
`(a0 >= b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpge_ps

```
__m128 _mm_cmpge_ps(__m128 a, __m128 b);
```

Compares for greater-than-or-equal.

R0	R1	R2	R3
`(a0 >= b0) ? 0xffffffff : 0x0`	`(a1 >= b1) ? 0xffffffff : 0x0`	`(a2 >= b2) ? 0xffffffff : 0x0`	`(a3 >= b3) ? 0xffffffff : 0x0`

_mm_cmpneq_ss

```
__m128 _mm_cmpneq_ss(__m128 a, __m128 b);
```

Compares for inequality.

R0	R1	R2	R3
`(a0 != b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpneq_ps

```
__m128 _mm_cmpneq_ps(__m128 a, __m128 b);
```

Compares for inequality.

R0	R1	R2	R3
`(a0 != b0) ? 0xffffffff : 0x0`	`(a1 != b1) ? 0xffffffff : 0x0`	`(a2 != b2) ? 0xffffffff : 0x0`	`(a3 != b3) ? 0xffffffff : 0x0`

_mm_cmpnlt_ss

```
__m128 _mm_cmpnlt_ss(__m128 a, __m128 b);
```

Compares for not-less-than.

R0	R1	R2	R3
`!(a0 < b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpnlt_ps

```
__m128 _mm_cmpnlt_ps(__m128 a, __m128 b);
```

Compares for not-less-than.

R0	R1	R2	R3
`!(a0 < b0) ? 0xffffffff : 0x0`	`!(a1 < b1) ? 0xffffffff : 0x0`	`!(a2 < b2) ? 0xffffffff : 0x0`	`!(a3 < b3) ? 0xffffffff : 0x0`

_mm_cmpnle_ss

```
__m128 _mm_cmpnle_ss(__m128 a, __m128 b);
```

Compares for not-less-than-or-equal.

R0	R1	R2	R3
`!(a0 <= b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpnle_ps

```
__m128 _mm_cmpnle_ps(__m128 a, __m128 b);
```

Compares for not-less-than-or-equal.

R0	R1	R2	R3
`!(a0 <= b0) ? 0xffffffff : 0x0`	`!(a1 <= b1) ? 0xffffffff : 0x0`	`!(a2 <= b2) ? 0xffffffff : 0x0`	`!(a3 <= b3) ? 0xffffffff : 0x0`

_mm_cmpngt_ss

```
__m128 _mm_cmpngt_ss(__m128 a, __m128 b);
```

Compares for not-greater-than.

R0	R1	R2	R3
`!(a0 > b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpngt_ps

```
__m128 _mm_cmpngt_ps(__m128 a, __m128 b);
```

Compares for not-greater-than.

R0	R1	R2	R3
`!(a0 > b0) ? 0xffffffff : 0x0`	`!(a1 > b1) ? 0xffffffff : 0x0`	`!(a2 > b2) ? 0xffffffff : 0x0`	`!(a3 > b3) ? 0xffffffff : 0x0`

_mm_cmpnge_ss

```
__m128 _mm_cmpnge_ss(__m128 a, __m128 b);
```

Compares for not-greater-than-or-equal.

R0	R1	R2	R3
`!(a0 >= b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpnge_ps

```
__m128 _mm_cmpnge_ps(__m128 a, __m128 b);
```

Compares for not-greater-than-or-equal.

R0	R1	R2	R3
`!(a0 >= b0) ? 0xffffffff : 0x0`	`!(a1 >= b1) ? 0xffffffff : 0x0`	`!(a2 >= b2) ? 0xffffffff : 0x0`	`!(a3 >= b3) ? 0xffffffff : 0x0`

_mm_cmpord_ss

```
__m128 _mm_cmpord_ss(__m128 a, __m128 b);
```

Compares for ordered.

R0	R1	R2	R3
`(a0 ord? b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpord_ps

```
__m128 _mm_cmpord_ps(__m128 a, __m128 b);
```

Compares for ordered.

R0	R1	R2	R3
`(a0 ord? b0) ? 0xffffffff : 0x0`	`(a1 ord? b1) ? 0xffffffff : 0x0`	`(a2 ord? b2) ? 0xffffffff : 0x0`	`(a3 ord? b3) ? 0xffffffff : 0x0`

_mm_cmpunord_ss

```
__m128 _mm_cmpunord_ss(__m128 a, __m128 b);
```

Compares for unordered.

R0	R1	R2	R3
`(a0 unord? b0) ? 0xffffffff : 0x0`	a1	a2	a3

_mm_cmpunord_ps

```
__m128 _mm_cmpunord_ps(__m128 a, __m128 b);
```

Compares for unordered.

R0	R1	R2	R3
`(a0 unord? b0) ? 0xffffffff : 0x0`	`(a1 unord? b1) ? 0xffffffff : 0x0`	`(a2 unord? b2) ? 0xffffffff : 0x0`	`(a3 unord? b3) ? 0xffffffff : 0x0`

_mm_comieq_ss

```
int _mm_comieq_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* equal to *b*. If *a* and *b* are equal, 1 is returned. Otherwise, 0 is returned.

R
`(a0 == b0) ? 0x1 : 0x0`

_mm_comilt_ss

```
int _mm_comilt_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and `b` for *a* less than *b*. If `a` is less than *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 < b0) ? 0x1 : 0x0`

_mm_comile_ss

```
int _mm_comile_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* less than or equal to *b*. If *a* is less than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 <= b0) ? 0x1 : 0x0`

_mm_comigt_ss

```
int _mm_comigt_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* greater than *b*. If *a* is greater than *b* are equal, 1 is returned. Otherwise, 0 is returned.

R
`(a0 > b0) ? 0x1 : 0x0`

_mm_comige_ss

```
int _mm_comige_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* greater than or equal to *b*. If *a* is greater than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 >= b0) ? 0x1 : 0x0`

_mm_comineq_ss

```
int _mm_comineq_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* not equal to `b`. If *a* and *b* are not equal, 1 is returned. Otherwise, 0 is returned.

R
`(a0 != b0) ? 0x1 : 0x0`

_mm_ucomieq_ss

```
int _mm_ucomieq_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* equal to `b`. If *a* and *b* are equal, 1 is returned. Otherwise, 0 is returned.

R
`(a0 == b0) ? 0x1 : 0x0`

_mm_ucomilt_ss

```
int _mm_ucomilt_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* less than `b`. If *a* is less than *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 < b0) ? 0x1 : 0x0`

_mm_ucomile_ss

```
int _mm_ucomile_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* less than or equal to `b`. If *a* is less than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 <= b0) ? 0x1 : 0x0`

_mm_ucomigt_ss

```
int _mm_ucomigt_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and `b` for *a* greater than *b*. If *a* is greater than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 > b0) ? 0x1 : 0x0`

_mm_ucominge_ss

```
int _mm_ucomige_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* greater than or equal to *b*. If *a* is greater than or equal to *b*, 1 is returned. Otherwise, 0 is returned.

R
`(a0 >= b0) ? 0x1 : 0x0`

_mm_ucomineq_ss

```
int _mm_ucomineq_ss(__m128 a, __m128 b);
```

Compares the lower SP FP value of *a* and *b* for *a* not equal to *b*. If *a* and *b* are not equal, 1 is returned. Otherwise, 0 is returned.

R
`r := (a0 != b0) ? 0x1 : 0x0`

Conversion Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for conversion operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R or R0-R3. R0, R1, R2, and R3 each represent one of the four 32-bit pieces of the result register.

Intrinsics marked with * are available only on Intel® 64 architecture. The rest of the intrinsics can be implemented on both IA-32 and Intel® 64 architectures.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_cvtss_si32`	Convert to 32-bit integer	`CVTSS2SI`

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_cvtss_si64*`	Convert to 64-bit integer	`CVTSS2SI`
`_mm_cvtps_pi32`	Convert to two 32-bit integers	`CVTPS2PI`
`_mm_cvttss_si32`	Convert to 32-bit integer	`CVTTSS2SI`
`_mm_cvttss_si64*`	Convert to 64-bit integer	`CVTTSS2SI`
`_mm_cvttps_pi32`	Convert to two 32-bit integers	`CVTTPS2PI`
`_mm_cvtsi32_ss`	Convert from 32-bit integer	`CVTSI2SS`
`_mm_cvtsi64_ss*`	Convert from 64-bit integer	`CVTSI2SS`
`_mm_cvtpi32_ps`	Convert from two 32-bit integers	`CVTTPI2PS`
`_mm_cvtpi16_ps`	Convert from four 16-bit integers	composite
`_mm_cvtpu16_ps`	Convert from four 16-bit integers	composite
`_mm_cvtpi8_ps`	Convert from four 8-bit integers	composite
`_mm_cvtpu8_ps`	Convert from four 8-bit integers	composite
`_mm_cvtpi32x2_ps`	Convert from four 32-bit integers	composite
`_mm_cvtps_pi16`	Convert to four 16-bit integers	composite
`_mm_cvtps_pi8`	Convert to four 8-bit integers	composite
`_mm_cvtss_f32`	Extract	composite

_mm_cvtss_si32

```
int _mm_cvtss_si32(__m128 a);
```

Converts the lower SP FP value of *a* to a 32-bit integer according to the current rounding mode.

R
`(int)a0`

_mm_cvtss_si64

```
__int64 _mm_cvtss_si64(__m128 a);
```

Converts the lower SP FP value of *a* to a 64-bit signed integer according to the current rounding mode.

NOTE
Use only on Intel© 64 architecture.

R
`(__int64)a0`

_mm_cvtps_pi32

```
__m64 _mm_cvtps_pi32(__m128 a);
```

Converts the two lower SP FP values of *a* to two 32-bit integers according to the current rounding mode, returning the integers in packed form.

R0	R1
`(int)a0`	`(int)a1`

_mm_cvttss_si32

```
int _mm_cvttss_si32(__m128 a);
```

Converts the lower SP FP value of *a* to a 32-bit integer with truncation.

R
`(int)a0`

_mm_cvttss_si64

```
__int64 _mm_cvttss_si64(__m128 a);
```

Converts the lower SP FP value of *a* to a 64-bit signed integer with truncation.

> **NOTE**
> Use only on Intel® 64 architecture.

R
`(__int64)a0`

_mm_cvttps_pi32

```
__m64 _mm_cvttps_pi32(__m128 a);
```

Converts the two lower SP FP values of *a* to two 32-bit integer with truncation, returning the integers in packed form.

R0	R1
`(int)a0`	`(int)a1`

_mm_cvtsi32_ss

```
__m128 _mm_cvtsi32_ss(__m128 a, int b);
```

Converts the 32-bit integer value *b* to an SP FP value; the upper three SP FP values are passed through from *a*.

R0	R1	R2	R3
`(float)b`	a1	a2	a3

_mm_cvtsi64_ss

`__m128 _mm_cvtsi64_ss(__m128 a, __int64 b);`

Converts the signed 64-bit integer value *b* to an SP FP value; the upper three SP FP values are passed through from *a*.

NOTE
Use only on Intel® 64 architecture.

R0	R1	R2	R3
`(float)b`	a1	a2	a3

_mm_cvtpi32_ps

`__m128 _mm_cvtpi32_ps(__m128 a, __m64 b);`

Converts the two 32-bit integer values in packed form in *b* to two SP FP values; the upper two SP FP values are passed through from *a*.

R0	R1	R2	R3
`(float)b0`	(float)b1	a2	a3

_mm_cvtpi16_ps

`__m128 _mm_cvtpi16_ps(__m64 a);`

Converts the four 16-bit signed integer values in *a* to four SP FP values.

R0	R1	R2	R3
`(float)a0`	(float)a1	`(float)a2`	`(float)a3`

_mm_cvtpu16_ps

`__m128 _mm_cvtpu16_ps(__m64 a);`

Converts the four 16-bit unsigned integer values in *a* to four SP FP values.

R0	R1	R2	R3
`(float)a0`	(float)a1	`(float)a2`	`(float)a3`

_mm_cvtpi8_ps

```
__m128 _mm_cvtpi8_ps(__m64 a);
```

Converts the lower four 8-bit signed integer values in *a* to four SP FP values.

R0	R1	R2	R3
(float)a0	(float)a1	(float)a2	(float)a3

_mm_cvtpu8_ps

```
__m128 _mm_cvtpu8_ps(__m64 a);
```

Converts the lower four 8-bit unsigned integer values in *a* to four SP FP values.

R0	R1	R2	R3
(float)a0	(float)a1	(float)a2	(float)a3

_mm_cvtpi32x2_ps

```
__m128 _mm_cvtpi32x2_ps(__m64 a, __m64 b);
```

Converts the two 32-bit signed integer values in *a* and the two 32-bit signed integer values in *b* to four SP FP values.

R0	R1	R2	R3
(float)a0	(float)a1	(float)b0	(float)b1

_mm_cvtps_pi16

```
__m64 _mm_cvtps_pi16(__m128 a);
```

Converts the four SP FP values in *a* to four signed 16-bit integer values.

R0	R1	R2	R3
(short)a0	(short)a1	(short)a2	(short)a3

_mm_cvtps_pi8

```
__m64 _mm_cvtps_pi8(__m128 a);
```

Converts the four SP FP values in *a* to the lower four signed 8-bit integer values of the result.

R0	R1	R2	R3
(char)a0	(char)a1	(char)a2	(char)a3

_mm_cvtss_f32

```
float _mm_cvtss_f32(__m128 a);
```

Extracts a SP floating-point value from the first vector element of an `__m128`. It does so in the most efficient manner possible in the context used.

Load Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for load operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. R0, R1, R2, and R3 each represent one of the four 32-bit pieces of the result register.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_loadh_pi`	Load high	`MOVHPS reg, mem`
`_mm_loadl_pi`	Load low	`MOVLPS reg, mem`
`_mm_load_ss`	Load the low value and clear the three high values	`MOVSS`
`_mm_load1_ps`	Load one value into all four words	`MOVSS + Shuffling`
`_mm_load_ps`	Load four values, address aligned	`MOVAPS`
`_mm_loadu_ps`	Load four values, address unaligned	`MOVUPS`
`_mm_loadr_ps`	Load four values in reverse	`MOVAPS + Shuffling`

_mm_loadh_pi

`__m128 _mm_loadh_pi(__m128 a, __m64 const *p);`

Sets the upper two SP FP values with 64 bits of data loaded from the address *p*; the lower two values are passed through from *a*.

R0	R1	R2	R3
a0	a1	*p0	*p1

_mm_loadl_pi

`__m128 _mm_loadl_pi(__m128 a, __m64 const *p);`

Sets the lower two SP FP values with 64 bits of data loaded from the address *p*; the upper two values are passed through from *a*.

R0	R1	R2	R3
a0	a1	*p0	*p1

R0	R1	R2	R3
*p0	*p1	a2	a3

_mm_load_ss

`__m128 _mm_load_ss(float * p);`

Loads a SP FP value into the low word and clears the upper three words.

R0	R1	R2	R3
*p	0.0	0.0	0.0

_mm_load1_ps

`__m128 _mm_load1_ps(float * p);`

Loads a SP FP value, copying it into all four words.

R0	R1	R2	R3
*p	*p	*p	*p

_mm_load_ps

`__m128 _mm_load_ps(float * p);`

Loads four SP FP values. The address must be 16-byte-aligned.

R0	R1	R2	R3
p[0]	p[1]	p[2]	p[3]

_mm_loadu_ps

`__m128 _mm_loadu_ps(float * p);`

Loads four SP FP values. The address need not be 16-byte-aligned.

R0	R1	R2	R3
p[0]	p[1]	p[2]	p[3]

_mm_loadr_ps

`__m128 _mm_loadr_ps(float * p);`

Loads four SP FP values in reverse order. The address must be 16-byte-aligned.

R0	R1	R2	R3
p[3]	p[2]	p[1]	p[0]

Set Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for set operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R0, R1, R2, and R3 represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_set_ss`	Set the low value and clear the three high values	Composite
`_mm_set1_ps`	Set all four words with the same value	Composite
`_mm_set_ps`	Set four values, address aligned	Composite
`_mm_setr_ps`	Set four values, in reverse order	Composite
`_mm_setzero_ps`	Clear all four values	Composite

_mm_set_ss

`__m128 _mm_set_ss(float w);`

Sets the low word of a SP FP value to *w* and clears the upper three words.

R0	R1	R2	R3
w	0.0	0.0	0.0

_mm_set1_ps

`__m128 _mm_set1_ps(float w);`

Sets the four SP FP values to *w*.

R0	R1	R2	R3
w	w	w	w

_mm_set_ps

`__m128 _mm_set_ps(float z, float y, float x, float w);`

Sets the four SP FP values to the inputs *w*, *x*, *y*, and *z*.

R0	R1	R2	R3
w	x	y	z

_mm_setr_ps

`__m128 _mm_setr_ps(float z, float y, float x, float w);`

Sets the four SP FP values to the inputs *w*, *x*, *y*, and *z* in reverse order.

R0	R1	R2	R3
z	y	x	w

_mm_setzero_ps

```
__m128 _mm_setzero_ps(void);
```

Clears the four SP FP values.

R0	R1	R2	R3
0.0	0.0	0.0	0.0

Store Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for store operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The description for each intrinsic contains a table detailing the returns. In these tables, p[*n*] is an access to the *n* element of the result.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_storeh_pi`	Store high	`MOVHPS mem, reg`
`_mm_storel_pi`	Store low	`MOVLPS mem, reg`
`_mm_store_ss`	Store the low value	`MOVSS`
`_mm_store1_ps`	Store the low value across all four words, address aligned	`Shuffling + MOVSS`
`_mm_store_ps`	Store four values, address aligned	`MOVAPS`
`_mm_storeu_ps`	Store four values, address unaligned	`MOVUPS`
`_mm_storer_ps`	Store four values, in reverse order	`MOVAPS + Shuffling`

_mm_storeh_pi

```
void _mm_storeh_pi(__m64 *p, __m128 a);
```

Stores the upper two SP FP values to the address *p*.

*p0	*p1
a2	a3

_mm_storel_pi

```
void _mm_storel_pi(__m64 *p, __m128 a);
```

Stores the lower two SP FP values of *a* to the address *p*.

*p0	*p1
a0	a1

_mm_store_ss

```
void _mm_store_ss(float * p, __m128 a);
```

Stores the lower SP FP value.

*p
a0

_mm_store1_ps

```
void _mm_store1_ps(float * p, __m128 a);
```

Stores the lower SP FP value across four words.

p[0]	p[1]	p[2]	p[3]
a0	a0	a0	a0

_mm_store_ps

```
void _mm_store_ps(float *p, __m128 a);
```

Stores four SP FP values. The address must be 16-byte-aligned.

p[0]	p[1]	p[2]	p[3]
a0	a1	a2	a3

_mm_storeu_ps

```
void _mm_storeu_ps(float *p, __m128 a);
```

Stores four SP FP values. The address need not be 16-byte-aligned.

p[0]	p[1]	p[2]	p[3]
a0	a1	a2	a3

_mm_storer_ps

```
void _mm_storer_ps(float * p, __m128 a);
```

Stores four SP FP values in reverse order. The address must be 16-byte-aligned.

p[0]	p[1]	p[2]	p[3]
a3	a2	a1	a0

Cacheability Support Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for cacheability support are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_prefetch`	Load	`PREFETCH`
`_mm_stream_pi`	Store	`MOVNTQ`
`_mm_stream_ps`	Store	`MOVNTPS`
`_mm256_stream_ps`	Store	`VMOVNTPS`
`_mm_sfence`	Store fence	`SFENCE`

_mm_prefetch

`void _mm_prefetch(char const*a, int sel);`

Loads one cache line of data from address `a` to a location "closer" to the processor. The value `sel` specifies the type of prefetch operation: the constants `_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2`, `_MM_HINT_NTA`, and `_MM_HINT_ET0` should be used for systems based on IA-32 architecture, and correspond to the type of `prefetch` instruction.

NOTE
The `_MM_HINT_ET0` hint lowers the intrinsic being to the instruction `PREFETCHW`, which is not included in Intel® SSE instructions. Check if the target CPU supports the `PREFETCHW` instruction before using `_MM_HINT_ET0`.

_mm_stream_pi

`void _mm_stream_pi(__m64 *p, __m64 a);`

Stores the data in *a* to the address *p* without polluting the caches. This intrinsic requires you to empty the multimedia state for the `MMX`™ register. See the topic The EMMS Instruction: Why You Need It.

_mm_stream_ps

`void _mm_stream_ps(float *p, __m128 a);`

Stores the data in *a* to the address *p* without polluting the caches. The address must be 16-byte-aligned.

_mm256_stream_ps

```
void _mm256_stream_ps(float *p, __m256 a);
```

Stores the data in *a* to the address *p* without polluting the caches. The address must be 32-byte (VEX.256 encoded version) aligned.

_mm_sfence

```
void _mm_sfence(void);
```

Guarantees that every preceding store is globally visible before any subsequent store.

See Also

The EMMS Instruction: Why You Need It

Integer Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for integer operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1, ..., R7 represent the registers in which results are placed.

Before using these intrinsics, you must empty the multimedia state for the MMX™ technology register. See The EMMS Instruction: Why You Need It for more details.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_extract_pi16`	Extract one of four words	`PEXTRW`
`_mm_insert_pi16`	Insert word	`PINSRW`
`_mm_max_pi16`	Compute maximum	`PMAXSW`
`_mm_max_pu8`	Compute maximum, unsigned	`PMAXUB`
`_mm_min_pi16`	Compute minimum	`PMINSW`
`_mm_min_pu8`	Compute minimum, unsigned	`PMINUB`
`_mm_movemask_pi8`	Create eight-bit mask	`PMOVMSKB`
`_mm_mulhi_pu16`	Multiply, return high bits	`PMULHUW`
`_mm_shuffle_pi16`	Return a combination of four words	`PSHUFW`
`_mm_maskmove_si64`	Conditional Store	`MASKMOVQ`
`_mm_avg_pu8`	Compute rounded average	`PAVGB`
`_mm_avg_pu16`	Compute rounded average	`PAVGW`

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_sad_pu8`	Compute sum of absolute differences	`PSADBW`

_mm_extract_pi16

```
int _mm_extract_pi16(__m64 a, int n);
```

Extracts one of the four words of *a*. The selector *n* must be an immediate.

R
`(n==0) ? a0 : ( (n==1) ? a1 : ( (n==2) ? a2 : a3 ) )`

_mm_insert_pi16

```
__m64 _mm_insert_pi16(__m64 a, int d, int n);
```

Inserts word `d` into one of four words of *a*. The selector *n* must be an immediate.

R0	R1	R2	R3
`(n==0) ? d : a0;`	`(n==1) ? d : a1;`	`(n==2) ? d : a2;`	`(n==3) ? d : a3;`

_mm_max_pi16

```
__m64 _mm_max_pi16(__m64 a, __m64 b);
```

Computes the element-wise maximum of the words in *a* and *b*.

R0	R1	R2	R3
`min(a0, b0)`	`min(a1, b1)`	`min(a2, b2)`	`min(a3, b3)`

_mm_max_pu8

```
__m64 _mm_max_pu8(__m64 a, __m64 b);
```

Computes the element-wise maximum of the unsigned bytes in *a* and *b*.

R0	R1	...	R7
`min(a0, b0)`	`min(a1, b1)`	`...`	`min(a7, b7)`

_mm_min_pi16

```
__m64 _mm_min_pi16(__m64 a, __m64 b);
```

Computes the element-wise minimum of the words in *a* and *b*.

R0	R1	R2	R3
`min(a0, b0)`	`min(a1, b1)`	`min(a2, b2)`	`min(a3, b3)`

_mm_min_pu8

```
__m64 _mm_min_pu8(__m64 a, __m64 b);
```

Computes the element-wise minimum of the unsigned bytes in *a* and *b*.

R0	R1	...	R7
`min(a0, b0)`	`min(a1, b1)`	`...`	`min(a7, b7)`

_mm_movemask_pi8

```
__m64 _mm_movemask_pi8(__m64 b);
```

Creates an 8-bit mask from the most significant bits of the bytes in *a*.

R
`sign(a7)<<7 | sign(a6)<<6 |... | sign(a0)`

_mm_mulhi_pu16

```
__m64 _mm_mulhi_pu16(__m64 a, __m64 b);
```

Multiplies the unsigned words in *a* and *b*, returning the upper 16 bits of the 32-bit intermediate results.

R0	R1	R2	R3
`hiword(a0 * b0)`	`hiword(a1 * b1)`	`hiword(a2 * b2)`	`hiword(a3 * b3)`

_mm_shuffle_pi16

```
__m64 _mm_shuffle_pi16(__m64 a, int n);
```

Returns a combination of the four words of *a*. The selector *n* must be an immediate.

R0	R1	R2	R3
`word (n&0x3) of a`	`word ((n>>2)&0x3) of a`	`word ((n>>4)&0x3) of a`	`word ((n>>6)&0x3) of a`

_mm_maskmove_si64

```
void _mm_maskmove_si64(__m64 d, __m64 n, char *p);
```

Conditionally stores byte elements of *d* to address *p*. The high bit of each byte in the selector *p* determines whether the corresponding byte in *d* will be stored.

`if (sign(n0))`	`if (sign(n1))`	...	`if (sign(n7))`
`p[0] := d0`	`p[1] := d1`	`...`	`p[7] := d7`

_mm_avg_pu8

```
__m64 _mm_avg_pu8(__m64 a, __m64 b);
```

Computes the (rounded) averages of the unsigned bytes in *a* and *b*.

R0	R1	...	R7
`(t >> 1) | (t & 0x01)`, where `t = (unsigned char)a0 + (unsigned char)b0`	`(t >> 1) | (t & 0x01)`, where `t = (unsigned char)a1 + (unsigned char)b1`	...	`((t >> 1) | (t & 0x01))`, where `t = (unsigned char)a7 + (unsigned char)b7`

_mm_avg_pu16

```
__m64 _mm_avg_pu16(__m64 a, __m64 b);
```

Computes the (rounded) averages of the unsigned short in *a* and *b*.

R0	R1	...	R7
`(t >> 1) | (t & 0x01)`, where `t = (unsigned int)a0 + (unsigned int)b0`	`(t >> 1) | (t & 0x01)`, where `t = (unsigned int)a1 + (unsigned int)b1`	...	`(t >> 1) | (t & 0x01)`, where `t = (unsigned int)a7 + (unsigned int)b7`

_mm_sad_pu8

```
__m64 _mm_sad_pu8(__m64 a, __m64 b);
```

Computes the sum of the absolute differences of the unsigned bytes in *a* and *b*, returning the value in the lower word. The upper three words are cleared.

R0	R1	R2	R3
`abs(a0-b0) +... + abs(a7-b7)`	0	0	0

Intrinsics to Read and Write Registers

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics to read from and write to registers are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_getcsr`	Return control register	`STMXCSR`
`_mm_setcsr`	Set control register	`LDMXCSR`

_mm_getcsr

```
unsigned int _mm_getcsr(void);
```

Returns the contents of the control register.

_mm_setcsr

```
void _mm_setcsr(unsigned int i);
```

Sets the control register to the value specified by *i*.

Miscellaneous Intrinsics

The prototypes for Intel® Streaming SIMD Extensions (Intel® SSE) intrinsics for miscellaneous operations are in the `xmmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R, R0, R1, R2, and R3 represent the registers in which results are placed.

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`_mm_shuffle_ps`	Shuffle	`SHUFPS`
`_mm_unpackhi_ps`	Unpack High	`UNPCKHPS`
`_mm_unpacklo_ps`	Unpack Low	`UNPCKLPS`
`_mm_move_ss`	Set low word, pass in three high values	`MOVSS`
`_mm_movehl_ps`	Move High to Low	`MOVHLPS`
`_mm_movelh_ps`	Move Low to High	`MOVLHPS`
`_mm_movemask_ps`	Create four-bit mask	`MOVMSKPS`
`_mm_undefined_ps`	Return vector of type __m128 with undefined elements.	This is a utility intrinsic that returns some arbitrary value.

_mm_shuffle_ps

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8);
```

Selects four specific SP FP values from *a* and *b*, based on the mask *imm8*. The mask must be an immediate. See Macro Function for Shuffle Using Intel® Streaming SIMD Extensions for a description of the shuffle semantics.

_mm_unpackhi_ps

```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b);
```

Selects and interleaves the upper two SP FP values from *a* and *b*.

R0	R1	R2	R3
a2	b2	a3	b3

_mm_unpacklo_ps

```
__m128 _mm_unpacklo_ps(__m128 a, __m128 b);
```

Selects and interleaves the lower two SP FP values from *a* and *b*.

R0	R1	R2	R3
a0	b0	a1	b1

_mm_move_ss

```
__m128 _mm_move_ss( __m128 a, __m128 b);
```

Sets the low word to the SP FP value of *b*. The upper three SP FP values are passed through from *a*.

R0	R1	R2	R3
b0	a1	a2	a3

_mm_movehl_ps

```
__m128 _mm_movehl_ps(__m128 a, __m128 b);
```

Moves the upper two SP FP values of *b* to the lower two SP FP values of the result. The upper two SP FP values of *a* are passed through to the result.

R0	R1	R2	R3
b2	b3	a2	a3

_mm_movelh_ps

```
__m128 _mm_movelh_ps(__m128 a, __m128 b);
```

Moves the lower two SP FP values of *b* to the upper two SP FP values of the result. The lower two SP FP values of *a* are passed through to the result.

R0	R1	R2	R3
a0	a1	b0	b1

_mm_movemask_ps

```
int _mm_movemask_ps(__m128 a);
```

Creates a 4-bit mask from the most significant bits of the four SP FP values.

R
sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)

_mm_undefined_ps

```
extern __m128 _mm_undefined_ps(void);
```

Returns a vector of four single precision floating point elements. The content of the vector is not specified. The result is usually used as an argument to another intrinsic that requires all operands to be initialized, and when the content of a particular argument does not matter. This intrinsic is declared in the `immintrin.h` header file. It typically maps to a read of some XMM register and gets whatever value happens to live in that register at the time of the read.

For example, you can use such an intrinsic when you need to calculate a sum of packed double-precision floating-point values located in the *xmm* register.

See Also

Macro Function for Shuffle Using Intel® Streaming SIMD Extensions

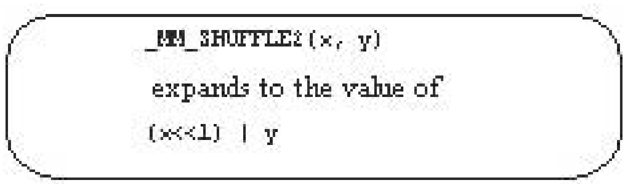
_mm256_undefined_ps() Returns a vector of eight single precision floating point elements. No corresponding Intel® AVX instruction.

Macro Functions

Macro Function for Shuffle Operations

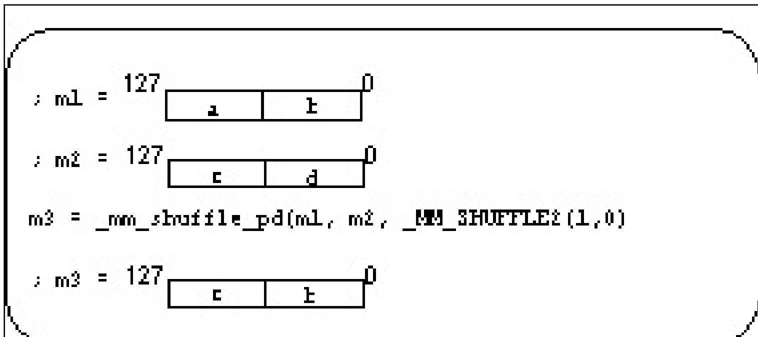
Intel® Streaming SIMD Extensions (Intel® SSE) provide a macro function to help create constants that describe shuffle operations. The macro takes four small integers (in the range of 0 to 3) and combines them into an 8-bit immediate value used by the `SHUFPS` instruction.

Shuffle Function Macro



You can view the four integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word.

View of Original and Result Words with Shuffle Function Macro



Macro Functions to Read and Write Control Registers

The following macro functions enable you to read and write bits to and from the control register.

Exception State Macros	Macro Arguments
`_MM_SET_EXCEPTION_STATE(x)`	`_MM_EXCEPT_INVALID`
`_MM_GET_EXCEPTION_STATE()`	`_MM_EXCEPT_DIV_ZERO`
	`_MM_EXCEPT_DENORM`
Macro Definitions	`_MM_EXCEPT_OVERFLOW`
Write to and read from the six least significant control register bits, respectively.	
	`_MM_EXCEPT_UNDERFLOW`
	`_MM_EXCEPT_INEXACT`

The following example tests for a divide-by-zero exception.

Exception State Macros with _MM_EXCEPT_DIV_ZERO

```
if  (_MM_GET_EXCEPTION_STATE(x)  &  _MM_EXCEPT_DIV_ZERO)  {
    /* Exception has occurred */
}
```

Exception Mask Macros	Macro Arguments
`_MM_SET_EXCEPTION_MASK(x)`	`_MM_MASK_INVALID`
`_MM_GET_EXCEPTION_MASK ()`	`_MM_MASK_DIV_ZERO`
	`_MM_MASK_DENORM`
Macro Definitions	`_MM_MASK_OVERFLOW`
Write to and read from bit 7 – 12 control register bits, respectively.	
NOTE All six exception mask bits are always affected. Bits not set explicitly are cleared.	
	`_MM_MASK_UNDERFLOW`
	`_MM_MASK_INEXACT`

To mask the overflow and underflow exceptions and unmask all other exceptions, use the macros as follows:

```
_MM_SET_EXCEPTION_MASK(MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)
```

The following table lists the macros to set and get rounding modes, and the macro arguments that can be passed with the macros.

Rounding Mode	Macro Arguments
`_MM_SET_ROUNDING_MODE(x)`	`_MM_ROUND_NEAREST`
`_MM_GET_ROUNDING_MODE()`	`_MM_ROUND_DOWN`
Macro Definition	`_MM_ROUND_UP`
Write to and read from bits 13 and 14 of the control register.	
	`_MM_ROUND_TOWARD_ZERO`

To test the rounding mode for round toward zero, use the `_MM_ROUND_TOWARD_ZERO` macro as follows.

```
if (_MM_GET_ROUNDING_MODE() == _MM_ROUND_TOWARD_ZERO) {
/* Rounding mode is round toward zero */
}
```

The following table lists the macros to set and get the flush-to-zero mode and the macro arguments that can be used.

Flush-to-Zero Mode	Macro Arguments
`_MM_SET_FLUSH_ZERO_MODE(x)`	`_MM_FLUSH_ZERO_ON`
`_MM_GET_FLUSH_ZERO_MODE()`	`_MM_FLUSH_ZERO_OFF`
Macro Definition	
Write to and read from bit 15 of the control register.	

To disable the flush-to-zero mode, use the `_MM_FLUSH_ZERO_OFF` macro.

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_OFF)
```

See Also

Intrinsics to Read and Write Registers

Macro Function for Matrix Transposition

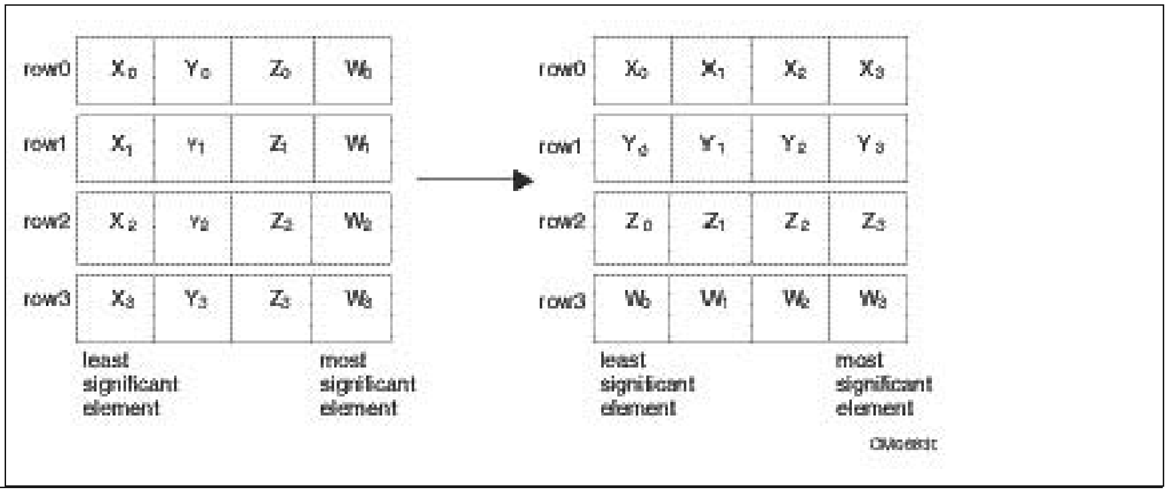
Intel® Streaming SIMD Extensions (Intel® SSE) provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

`_MM_TRANSPOSE4_PS(row0, row1, row2, row3)`

The arguments `row0`, `row1`, `row2`, and `row3` are `__m128` values whose elements form the corresponding rows of a 4 by 4 matrix. The matrix transposition is returned in arguments `row0`, `row1`, `row2`, and `row3` where `row0` now holds column 0 of the original matrix, `row1` now holds column 1 of the original matrix, and so on.

The transposition function of this macro is illustrated in the figure below.

Matrix Transposition Using _MM_TRANSPOSE4_PS Macro



Intrinsics for MMX™ Technology

MMX™ technology is an extension to the Intel® architecture instruction set. The MMX™ instruction set adds 57 opcodes and a 64-bit quadword data type, and eight 64-bit registers. Each of the eight registers can be directly addressed using the register names `MM0` to `MM7`.

The prototypes for MMX™ technology intrinsics are in the `mmintrin.h` header file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Details about MMX™ Technology Intrinsics

The MMX™ technology instructions use the following features:

- Registers – Enable packed data of up to 128 bits in length for optimal single-instruction multiple data (SIMD) processing.
- Data Types – Enable packing of up to 16 elements of data in one register.

Registers

The MMX™ instructions use eight 64-bit registers (`mm0` to `mm7`) which are aliased on the floating-point stack registers.

Because each of these registers can hold more than one data element, the processor can process more than one data element simultaneously. This processing capability is also known as single-instruction multiple data (SIMD) processing.

For each computational and data manipulation instruction in the new extension sets, there is a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

Data Types

Intrinsic functions use four new C data types as operands, representing the new registers that are used as the operands to these intrinsic functions.

__m64 Data Type

The `__m64` data type is used to represent the contents of an MMX™ register, which is the register that is used by the MMX™ technology intrinsics. The `__m64` data type can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value.

Data Types Usage Guidelines

These data types are not basic ANSI C data types. You must observe the following usage restrictions:

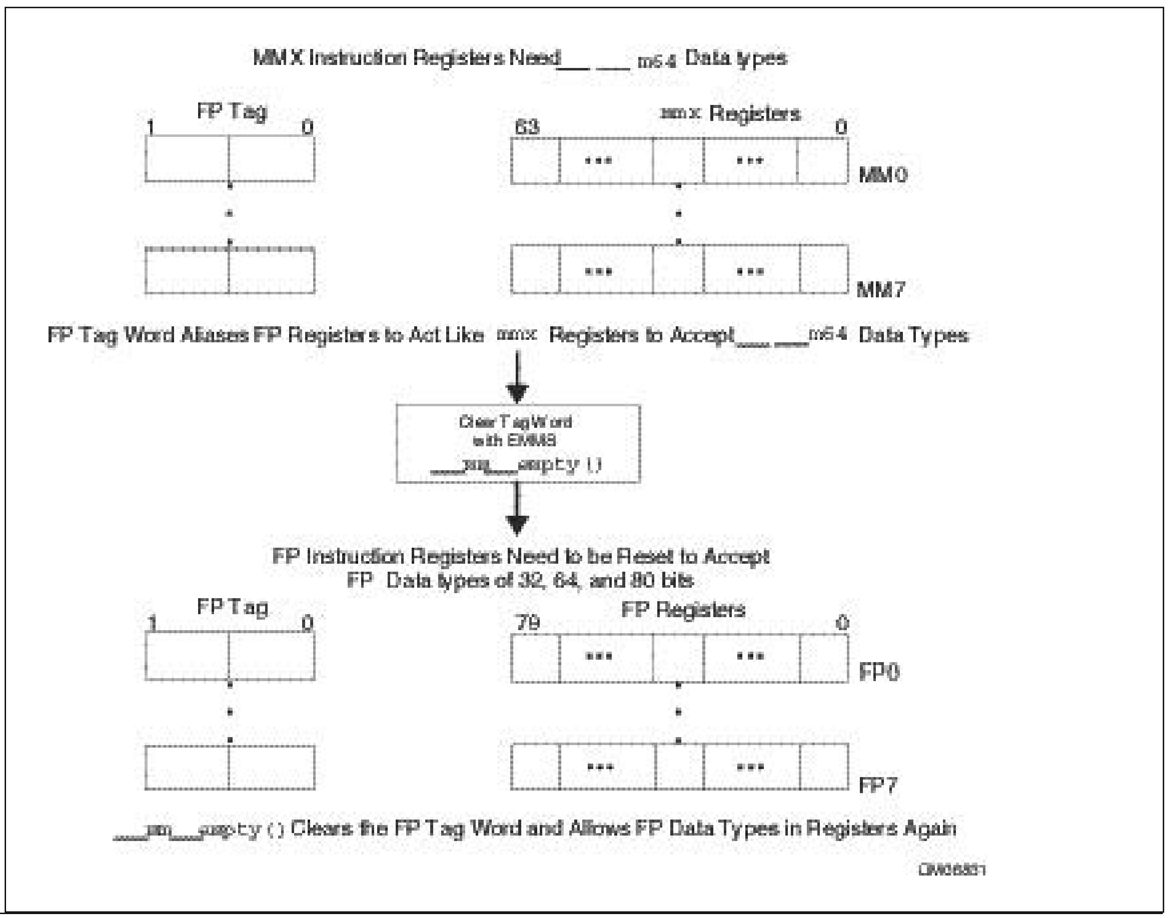
- Use data types only on either side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions (+, -, etc).
- Use data types as objects in aggregates, such as unions, to access the byte elements and structures.
- Use data types only with the respective intrinsics described in this documentation.

The EMMS Instruction: Why You Need It

Using EMMS is like emptying a container to accommodate new content. The EMMS instruction clears the MMX™ registers and sets the value of the floating-point tag word to empty.

You should clear the MMX™ registers before issuing a floating-point instruction because floating-point convention specifies that the floating-point stack be cleared after use. Insert the EMMS instruction at the end of all MMX™ code segments to avoid a floating-point overflow exception.

Why You Need EMMS to Reset After an MMX™ Instruction



Caution
Failure to empty the multimedia state after using an MMX™ technology instruction and before using a floating-point instruction can result in unexpected execution or poor performance.

EMMS Usage Guidelines

Here are guidelines for when to use the EMMS instruction:

- Use `_mm_empty()` after an MMX™ instruction if the next instruction is a floating-point (FP) instruction. For example, you should use the EMMS instruction before performing calculations on `float`, `double` or `long double`. You must be aware of all situations in which your code generates an MMX™ instruction:
  - when using an MMX™ technology intrinsic
  - when using Intel® Streaming SIMD Extensions (Intel® SSE) integer intrinsics that use the `__m64` data type
  - when referencing an `__m64` data type variable
  - when using an MMX™ instruction through inline assembly
- Use different functions for operations that use floating point instructions and those that use MMX™ instructions. This action eliminates the need to empty the multimedia state within the body of a critical loop.
- Use `_mm_empty()` during runtime initialization of `__m64` and FP data types. This ensures resetting the register between data type transitions.
- Do not use `_mm_empty()` before an MMX™ instruction, since using `_mm_empty()` before an MMX™ instruction incurs an operation with no benefit (no-op).
- See the Correct Usage and Incorrect Usage coding examples in the following table.

Incorrect Usage	Correct Usage
`__m64 x = _m_paddd(y, z);`	`__m64 x = _m_paddd(y, z);`
`float f = init();`	`float f = ( _mm_empty(), init());`

General Support Intrinsics (MMX™ technology)

This topic summarizes the MMX™ technology general support intrinsics.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_empty`	Empty `MM` state	`EMMS`
`_mm_cvtsi32_si64`	Convert from `int`	`MOVD`
`_mm_cvtsi64_si32`	Convert to `int`	`MOVD`
`_mm_cvtsi64_m64`	Convert from `__int64`	`MOVQ`
`_mm_cvtm64_si64`	Convert to `__int64`	`MOVQ`
`_mm_packs_pi16`	Pack	`PACKSSWB`

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_packs_pi32`	Pack	`PACKSSDW`
`_mm_packs_pu16`	Pack	`PACKUSWB`
`_mm_unpackhi_pi8`	Interleave	`PUNPCKHBW`
`_mm_unpackhi_pi16`	Interleave	`PUNPCKHWD`
`_mm_unpackhi_pi32`	Interleave	`PUNPCKHDQ`
`_mm_unpacklo_pi8`	Interleave	`PUNPCKLBW`
`_mm_unpacklo_pi16`	Interleave	`PUNPCKLWD`
`_mm_unpacklo_pi32`	Interleave	`PUNPCKLDQ`

_mm_empty

```
void _mm_empty(void);
```

Empties the multimedia state.

_mm_cvtsi32_si64

```
__m64 _mm_cvtsi32_si64(int i);
```

Converts the integer object *i* to a 64-bit `__m64` object. The integer value is zero-extended to 64 bits.

_mm_cvtsi64_si32

```
int _mm_cvtsi64_si32(__m64 m);
```

Converts the lower 32 bits of the `__m64` object *m* to an integer.

_mm_cvtsi64_m64

```
__m64 _mm_cvtsi64_m64(__int64 i);
```

Moves the 64-bit integer object *i* to a `__m64` object

_mm_cvtm64_si64

```
__m64 _mm_cvtm64_si64(__m64 m);
```

Moves the `__m64` object *m* to a 64-bit integer

_mm_packs_pi16

```
__m64 _mm_packs_pi16(__m64 m1, __m64 m2);
```

Packs the four 16-bit values from *m1* into the lower four 8-bit values of the result with signed saturation, and pack the four 16-bit values from *m2* into the upper four 8-bit values of the result with signed saturation.

_mm_packs_pi32

```
__m64 _mm_packs_pi32(__m64 m1, __m64 m2);
```

Packs the two 32-bit values from *m1* into the lower two 16-bit values of the result with signed saturation, and pack the two 32-bit values from *m2* into the upper two 16-bit values of the result with signed saturation.

_mm_packs_pu16

```
__m64 _mm_packs_pu16(__m64 m1, __m64 m2);
```

Packs the four 16-bit values from *m1* into the lower four 8-bit values of the result with unsigned saturation, and pack the four 16-bit values from *m2* into the upper four 8-bit values of the result with unsigned saturation.

_mm_unpackhi_pi8

```
__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2);
```

Interleaves the four 8-bit values from the high half of *m1* with the four values from the high half of *m2*. The interleaving begins with the data from *m1*.

_mm_unpackhi_pi16

```
__m64 _mm_unpackhi_pi16(__m64 m1, __m64 m2);
```

Interleaves the two 16-bit values from the high half of *m1* with the two values from the high half of *m2*. The interleaving begins with the data from *m1*.

_mm_unpackhi_pi32

```
__m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2);
```

Interleaves the 32-bit value from the high half of *m1* with the 32-bit value from the high half of *m2*. The interleaving begins with the data from *m1*.

_mm_unpacklo_pi8

```
__m64 _mm_unpacklo_pi8(__m64 m1, __m64 m2);
```

Interleaves the four 8-bit values from the low half of *m1* with the four values from the low half of *m2*. The interleaving begins with the data from *m1*.

_mm_unpacklo_pi16

```
__m64 _mm_unpacklo_pi16(__m64 m1, __m64 m2);
```

Interleaves the two 16-bit values from the low half of *m1* with the two values from the low half of *m2*. The interleaving begins with the data from *m1*.

_mm_unpacklo_pi32

```
__m64 _mm_unpacklo_pi32(__m64 m1, __m64 m2);
```

Interleaves the 32-bit value from the low half of *m1* with the 32-bit value from the low half of *m2*. The interleaving begins with the data from *m1*.

Packed Arithmetic Intrinsics (MMX™ technology)

This topic summarizes the MMX™ technology packed arithmetic intrinsics.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_add_pi8`	Addition	`PADDB`
`_mm_add_pi16`	Addition	`PADDW`
`_mm_add_pi32`	Addition	`PADDD`
`_mm_adds_pi8`	Addition	`PADDSB`
`_mm_adds_pi16`	Addition	`PADDSW`
`_mm_adds_pu8`	Addition	`PADDUSB`
`_mm_adds_pu16`	Addition	`PADDUSW`
`_mm_sub_pi8`	Subtraction	`PSUBB`
`_mm_sub_pi16`	Subtraction	`PSUBW`
`_mm_sub_pi32`	Subtraction	`PSUBD`
`_mm_subs_pi8`	Subtraction	`PSUBSB`
`_mm_subs_pi16`	Subtraction	`PSUBSW`
`_mm_subs_pu8`	Subtraction	`PSUBUSB`
`_mm_subs_pu16`	Subtraction	`PSUBUSW`
`_mm_madd_pi16`	Multiply and add	`PMADDWD`
`_mm_mulhi_pi16`	Multiplication	`PMULHW`
`_mm_mullo_pi16`	Multiplication	`PMULLW`

_mm_add_pi8

```
__m64 _mm_add_pi8(__m64 m1, __m64 m2);
```

Add the eight 8-bit values in *m1* to the eight 8-bit values in *m2*.

_mm_add_pi16

```
__m64 _mm_add_pi16(__m64 m1, __m64 m2);
```

Add the four 16-bit values in *m1* to the four 16-bit values in *m2*.

_mm_add_pi32

```
__m64 _mm_add_pi32(__m64 m1, __m64 m2);
```

Add the two 32-bit values in *m1* to the two 32-bit values in *m2*.

_mm_adds_pi8

```
__m64 _mm_adds_pi8(__m64 m1, __m64 m2);
```

Add the eight signed 8-bit values in *m1* to the eight signed 8-bit values in *m2* using saturating arithmetic.

_mm_adds_pi16

```
__m64 _mm_adds_pi16(__m64 m1, __m64 m2);
```

Add the four signed 16-bit values in *m1* to the four signed 16-bit values in *m2* using saturating arithmetic.

_mm_adds_pu8

```
__m64 _mm_adds_pu8(__m64 m1, __m64 m2);
```

Add the eight unsigned 8-bit values in *m1* to the eight unsigned 8-bit values in *m2* and using saturating arithmetic.

_mm_adds_pu16

```
__m64 _mm_adds_pu16(__m64 m1, __m64 m2);
```

Add the four unsigned 16-bit values in *m1* to the four unsigned 16-bit values in *m2* using saturating arithmetic.

_mm_sub_pi8

```
__m64 _mm_sub_pi8(__m64 m1, __m64 m2);
```

Subtract the eight 8-bit values in *m2* from the eight 8-bit values in *m1*.

_mm_sub_pi16

```
__m64 _mm_sub_pi16(__m64 m1, __m64 m2);
```

Subtract the four 16-bit values in *m2* from the four 16-bit values in *m1*.

_mm_sub_pi32

```
__m64 _mm_sub_pi32(__m64 m1, __m64 m2);
```

Subtract the two 32-bit values in *m2* from the two 32-bit values in `m1`.

_mm_subs_pi8

```
__m64 _mm_subs_pi8(__m64 m1, __m64 m2);
```

Subtract the eight signed 8-bit values in *m2* from the eight signed 8-bit values in *m1* using saturating arithmetic.

_mm_subs_pi16

```
__m64 _mm_subs_pi16(__m64 m1, __m64 m2);
```

Subtract the four signed 16-bit values in *m2* from the four signed 16-bit values in *m1* using saturating arithmetic.

_mm_subs_pu8

```
__m64 _mm_subs_pu8(__m64 m1, __m64 m2);
```

Subtract the eight unsigned 8-bit values in *m2* from the eight unsigned 8-bit values in *m1* using saturating arithmetic.

_mm_subs_pu16

```
__m64 _mm_subs_pu16(__m64 m1, __m64 m2);
```

Subtract the four unsigned 16-bit values in *m2* from the four unsigned 16-bit values in *m1* using saturating arithmetic.

_mm_madd_pi16

```
__m64 _mm_madd_pi16(__m64 m1, __m64 m2);
```

Multiply four 16-bit values in *m1* by four 16-bit values in *m2* producing four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.

_mm_mulhi_pi16

```
__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2);
```

Multiply four signed 16-bit values in *m1* by four signed 16-bit values in *m2* and produce the high 16 bits of the four results.

_mm_mullo_pi16

```
__m64 _mm_mullo_pi16(__m64 m1, __m64 m2);
```

Multiply four 16-bit values in *m1* by four 16-bit values in *m2* and produce the low 16 bits of the four results.

Shift Intrinsics (MMX™ technology)

This topic summarizes the MMX™ technology shift intrinsics.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_sll_pi16`	Logical shift left	`PSLLW`
`_mm_slli_pi16`	Logical shift left	`PSLLWI`
`_mm_sll_pi32`	Logical shift left	`PSLLD`
`_mm_slli_pi32`	Logical shift left	`PSLLDI`
`_mm_sll_pi64`	Logical shift left	`PSLLQ`
`_mm_slli_pi64`	Logical shift left	`PSLLQI`
`_mm_sra_pi16`	Arithmetic shift right	`PSRAW`
`_mm_srai_pi16`	Arithmetic shift right	`PSRAWI`

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_sra_pi32`	Arithmetic shift right	`PSRAD`
`_mm_srai_pi32`	Arithmetic shift right	`PSRADI`
`_mm_srl_pi16`	Logical shift right	`PSRLW`
`_mm_srli_pi16`	Logical shift right	`PSRLWI`
`_mm_srl_pi32`	Logical shift right	`PSRLD`
`_mm_srli_pi32`	Logical shift right	`PSRLDI`
`_mm_srl_pi64`	Logical shift right	`PSRLQ`
`_mm_srli_pi64`	Logical shift right	`PSRLQI`

_mm_sll_pi16

```
__m64 _mm_sll_pi16(__m64 m, __m64 count);
```

Shifts four 16-bit values in *m* left the amount specified by *count* while shifting in zeros.

_mm_slli_pi16

```
__m64 _mm_slli_pi16(__m64 m, int count);
```

Shifts four 16-bit values in *m* left the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

_mm_sll_pi32

```
__m64 _mm_sll_pi32(__m64 m, __m64 count);
```

Shifts two 32-bit values in *m* left the amount specified by *count* while shifting in zeros.

_mm_slli_pi32

```
__m64 _mm_slli_pi32(__m64 m, int count);
```

Shifts two 32-bit values in *m* left the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

_mm_sll_pi64

```
__m64 _mm_sll_pi64(__m64 m, __m64 count);
```

Shifts the 64-bit value in *m* left the amount specified by *count* while shifting in zeros.

_mm_slli_pi64

```
__m64 _mm_slli_pi64(__m64 m, int count);
```

Shifts the 64-bit value in *m* left the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

_mm_sra_pi16

```
__m64 _mm_sra_pi16(__m64 m, __m64 count);
```

Shifts four 16-bit values in *m* right the amount specified by *count* while shifting in the sign bit.

_mm_srai_pi16

```
__m64 _mm_srai_pi16(__m64 m, int count);
```

Shifts four 16-bit values in *m* right the amount specified by *count* while shifting in the sign bit. For the best performance, *count* should be a constant.

_mm_sra_pi32

```
__m64 _mm_sra_pi32(__m64 m, __m64 count);
```

Shifts two 32-bit values in *m* right the amount specified by *count* while shifting in the sign bit.

_mm_srai_pi32

```
__m64 _mm_srai_pi32(__m64 m, int count);
```

Shifts two 32-bit values in *m* right the amount specified by *count* while shifting in the sign bit. For the best performance, *count* should be a constant.

_mm_srl_pi16

```
__m64 _mm_srl_pi16(__m64 m, __m64 count);
```

Shifts four 16-bit values in *m* right the amount specified by *count* while shifting in zeros.

_mm_srli_pi16

```
__m64 _mm_srli_pi16(__m64 m, int count);
```

Shifts four 16-bit values in *m* right the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

_mm_srl_pi32

```
__m64 _mm_srl_pi32(__m64 m, __m64 count);
```

Shifts two 32-bit values in *m* right the amount specified by *count* while shifting in zeros.

_mm_srli_pi32

```
__m64 _mm_srli_pi32(__m64 m, int count);
```

Shifts two 32-bit values in *m* right the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

_mm_srl_pi64

```
__m64 _mm_srl_pi64(__m64 m, __m64 count);
```

Shifts the 64-bit value in *m* right the amount specified by *count* while shifting in zeros.

_mm_srli_pi64

```
__m64 _mm_srli_pi64(__m64 m, int count);
```

Shifts the 64-bit value in *m* right the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

Logical Intrinsics (MMX™ technology)

This topic summarizes the MMX™ technology logical intrinsics.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_and_si64`	Bitwise AND	`PAND`
`_mm_andnot_si64`	Bitwise ANDNOT	`PANDN`
`_mm_or_si64`	Bitwise OR	`POR`
`_mm_xor_si64`	Bitwise Exclusive OR	`PXOR`

_mm_and_si64

```
__m64 _mm_and_si64(__m64 m1, __m64 m2);
```

Perform a bitwise AND of the 64-bit value in *m1* with the 64-bit value in *m2*.

_mm_andnot_si64

```
__m64 _mm_andnot_si64(__m64 m1, __m64 m2);
```

Perform a bitwise NOT on the 64-bit value in *m1* and use the result in a bitwise AND with the 64-bit value in *m2*.

_mm_or_si64

```
__m64 _mm_or_si64(__m64 m1, __m64 m2);
```

Perform a bitwise OR of the 64-bit value in *m1* with the 64-bit value in *m2*.

_mm_xor_si64

```
__m64 _mm_xor_si64(__m64 m1, __m64 m2);
```

Perform a bitwise XOR of the 64-bit value in *m1* with the 64-bit value in *m2*.

Compare Intrinsics (MMX™ technology)

This topic summarizes the MMX™ technology compare intrinsics.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_cmpeq_pi8`	Equal	`PCMPEQB`
`_mm_cmpeq_pi16`	Equal	`PCMPEQW`
`_mm_cmpeq_pi32`	Equal	`PCMPEQD`
`_mm_cmpgt_pi8`	Greater Than	`PCMPGTB`
`_mm_cmpgt_pi16`	Greater Than	`PCMPGTW`
`_mm_cmpgt_pi32`	Greater Than	`PCMPGTD`

_mm_ cmpeq_pi8

```
__m64 _mm_ cmpeq_pi8(__m64 m1, __m64 m2);
```

Sets the corresponding 8-bit resulting values to all ones if the 8-bit values in *m1* are equal to the corresponding 8-bit values in *m2*; otherwise sets them to all zeros.

_mm_ cmpeq_pi16

```
__m64 _mm_ cmpeq_pi16(__m64 m1, __m64 m2);
```

Sets the corresponding 16-bit resulting values to all ones if the 16-bit values in *m1* are equal to the corresponding 16-bit values in *m2*; otherwise set them to all zeros.

_mm_ cmpeq_pi32

```
__m64 _mm_ cmpeq_pi32(__m64 m1, __m64 m2);
```

Sets the corresponding 32-bit resulting values to all ones if the 32-bit values in *m1* are equal to the corresponding 32-bit values in *m2*; otherwise set them to all zeros.

_mm_ cmpgt_pi8

```
__m64 _mm_ cmpgt_pi8(__m64 m1, __m64 m2);
```

Sets the corresponding 8-bit resulting values to all ones if the 8-bit signed values in *m1* are greater than the corresponding 8-bit signed values in *m2*; otherwise set them to all zeros.

_mm_ cmpgt_pi16

```
__m64 _mm_ cmpgt_pi16(__m64 m1, __m64 m2);
```

Sets the corresponding 16-bit resulting values to all ones if the 16-bit signed values in *m1* are greater than the corresponding 16-bit signed values in *m2*; otherwise set them to all zeros.

_mm_ cmpgt_pi32

```
__m64 _mm_ cmpgt_pi32(__m64 m1, __m64 m2);
```

Sets the corresponding 32-bit resulting values to all ones, if the 32-bit signed values in *m1* are greater than the corresponding 32-bit signed values in *m2*; otherwise set them all to zeros.

Set Intrinsics (MMX™ technology)

This topic summarizes the MMX™ technology intrinsics.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

NOTE
In the descriptions regarding the bits of the `MMX`™ register, bit 0 is the least significant and bit 63 is the most significant.

Intrinsic Name	Operation	Corresponding MMX™ Instruction
`_mm_setzero_si64`	set to zero	`PXOR`
`_mm_set_pi32`	set integer values	Composite
`_mm_set_pi16`	set integer values	Composite
`_mm_set_pi8`	set integer values	Composite
`_mm_set1_pi32`	set integer values	Composite
`_mm_set1_pi16`	set integer values	Composite
`_mm_set1_pi8`	set integer values	Composite
`_mm_setr_pi32`	set integer values	Composite
`_mm_setr_pi16`	set integer values	Composite
`_mm_setr_pi8`	set integer values	Composite

_mm_setzero_si64

```
__m64 _mm_setzero_si64(void);
```

Sets the 64-bit value to zero.

R
`0x0`

_mm_set_pi32

```
__m64 _mm_set_pi32(int i1, int i0);
```

Sets the two signed 32-bit integer values.

R0	R1
`i0`	`i1`

_mm_set_pi16

```
__m64 _mm_set_pi16(short s3, short s2, short s1, short s0);
```

Sets the four signed 16-bit integer values.

R0	R1	R2	R3
w0	w1	w2	w3

_mm_set_pi8

```
__m64 _mm_set_pi8(char b7, char b6, char b5, char b4, char b3, char b2, char b1, char
b0);
```

Sets the eight signed 8-bit integer values.

R0	R1	...	R7
b0	b1	...	b7

_mm_set1_pi32

```
__m64 _mm_set1_pi32(int i);
```

Sets the two signed 32-bit integer values to *i*.

R0	R1
i	i

_mm_set1_pi16

```
__m64 _mm_set1_pi16(short s);
```

Sets the four signed 16-bit integer values to *s*.

R0	R1	R2	R3
w	w	w	w

_mm_set1_pi8

```
__m64 _mm_set1_pi8(char b);
```

Sets the eight signed 8-bit integer values to *b*.

R0	R1	...	R7
b	b	...	b

_mm_setr_pi32

```
__m64 _mm_setr_pi32(int i1, int i0);
```

Sets the two signed 32-bit integer values in reverse order.

R0	R1
i1	i0

_mm_setr_pi16

```
__m64 _mm_setr_pi16(short s3, short s2, short s1, short s0);
```

Sets the four signed 16-bit integer values in reverse order.

R0	R1	R2	R3
w3	w2	w1	w0

_mm_setr_pi8

```
__m64 _mm_setr_pi8(char b7, char b6, char b5, char b4, char b3, char b2, char b1, char
b0);
```

Sets the eight signed 8-bit integer values in reverse order.

R0	R1	...	R7
b7	b6	...	b0

Intrinsics for Advanced Encryption Standard Implementation

The Intel® C++ Compiler Classic provides intrinsics to enable carry-less multiplication and encryption based on Advanced Encryption Standard (AES) specifications. The carry-less multiplication intrinsic corresponds to a single new instruction, `PCLMULQDQ`. The AES extension intrinsics correspond to AES extension intructions.

The AES extension instructions and the `PCLMULQDQ` instruction follow the same system software requirements for `XMM` state support and single-instruction multiple data (SIMD) floating-point exception support as Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Intel Supplemental Streaming SIMD Extensions 3 (SSSE3), and Intel® Streaming SIMD Extensions 4 (Intel® SSE4) extensions.

Intel®64 processors using 32nm processing technology support the AES extension instructions as well as the `PCLMULQDQ` instruction.

AES Encryption and Cryptographic Processing

AES encryption involves processing 128-bit input data (plaintext) through a finite number of iterative operation, referred to as AES round, into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the equivalent inverse cipher instead of the inverse cipher.

The cryptographic processing at each round involves two input data, one is the state, the other is the round key. Each round uses a different round key. The round keys are derived from the cipher key using a key schedule algorithm. The key schedule algorithm is independent of the data processing of encryption/ decryption, and can be carried out independently from the encryption/decryption phase.

The AES standard supports cipher key of sizes 128, 192, and 256 bits. The respective cipher key sizes corresponds to 10, 12, and 14 rounds of iteration.

Carry-less Multiplication Instruction and AES Extension Instructions

A single instruction, PCLMULQDQ, performs carry-less multiplication for two binary numbers that are up to 64-bit wide.

The AES extensions provide:

- two instructions to accelerate AES rounds on encryption (AESENC and AESENCLAST)
- two instructions for AES rounds on decryption using the equivalent inverse cipher (AESDEC and AESENCLAST)
- instructions for the generation of key schedules (AESIMC and AESENCLAST)

Detecting Support for Using Instructions

Before any application attempts to use the PCLMULQDQ or the AES extension instructions, it must first detect if the instructions are supported by the processor.

To detect support for the PCLMULQDQ instruction, your application must check the following:

```
CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1.
```

To detect support for the AES extension instructions, your application must check the following:

```
CPUID.01H:ECX.AES[bit 25] = 1.
```

Operating systems that support handling of the SSE state also support applications that use AES extension instruction and the PCLMULQDQ instruction.

Intrinsics for Carry-less Multiplication Instruction and Advanced Encryption Standard Instructions

The prototypes for the Carry-less multiplication intrinsic and the Advanced Encryption Standard (AES) intrinsics are defined in the wmmintrin.h file.

To use these intrinsics, include the immintrin.h file as follows:

```
#include <immintrin.h>
```

Carry-less Multiplication Intrinsic

The single general purpose block encryption intrinsic description is provided below.

`__m128i _mm_clmulepi64_si128(__m128i v1, __m128i v2, const int imm8);`

Performs a carry-less multiplication of one quadword of v1 by one quadword of v2, and returns the result. The imm8 value is used to determine which quadwords of v1 and v2 should be used.

Corresponding Instruction: PCLMULQDQ

Advanced Encryption Standard Intrinsics

The AES intrinsics are described below.

`__m128i _mm_aesdec_si128(__m128i v, __m128i rkey);`

Performs one round of an AES decryption flow using the Equivalent Inverse Cipher operating on a 128-bit data (state) from v with a 128-bit round key from rkey.

Corresponding Instruction: AESDEC

`__m128i _mm_aesdeclast_si128(__m128i v, __m128i rkey);`

Performs the last round of an AES decryption flow using the Equivalent Inverse Cipher operating on a 128-bit data (state) from `v` with a 128-bit round key from `rkey`.

Corresponding Instruction: `AESDECLAST`

```
__m128i _mm_aesenc_si128(__m128i v, __m128i rkey);
```

Performs one round of an AES encryption flow operating on a 128-bit data (state) from `v` with a 128-bit round key from `rkey`.

Corresponding Instruction:`AESENC`

```
__m128i _mm_aesenclast_si128(__m128i v, __m128i rkey);
```

Performs the last round of an AES encryption flow operating on a 128-bit data (state) from `v` with a 128-bit round key from `rkey`.

Corresponding Instruction: `AESENCLAST`

```
__m128i _mm_aesimc_si128(__m128i v);
```

Performs the InvMixColumn transformation on a 128-bit round key from`v` and returns the result.

Corresponding Instruction: `AESIMC`

```
__m128i _mm_aeskeygenassist_si128(__m128i ckey, const int rcon);
```

Assists in AES round key generation using an 8-bit Round Constant (RCON) specified in `rcon` operating on 128 bits of data specified in `ckey` and returns the result.

Corresponding Instruction: `AESKEYGENASSIST`

Intrinsics for Converting Half Floats

The half-float or 16-bit float is a popular type in some application domains. The half-float type is regarded as a storage type because although data is often stored as a half-float, computation is never done on values in these type. Usually values are converted to regular 32-bit floats before any computation.

Support for half-float type is restricted to just conversions to/from 32-bit floats. The main benefits of using half float type are:

- reduced storage requirements
- less consumption of memory bandwidth and cache
- accuracy and precision adequate for many applications

Half Float Intrinsics

The half-float intrinsics are provided to convert half-float values to 32-bit floats for computation purposes and, conversely, 32-bit float values to half-float values for data storage purposes.

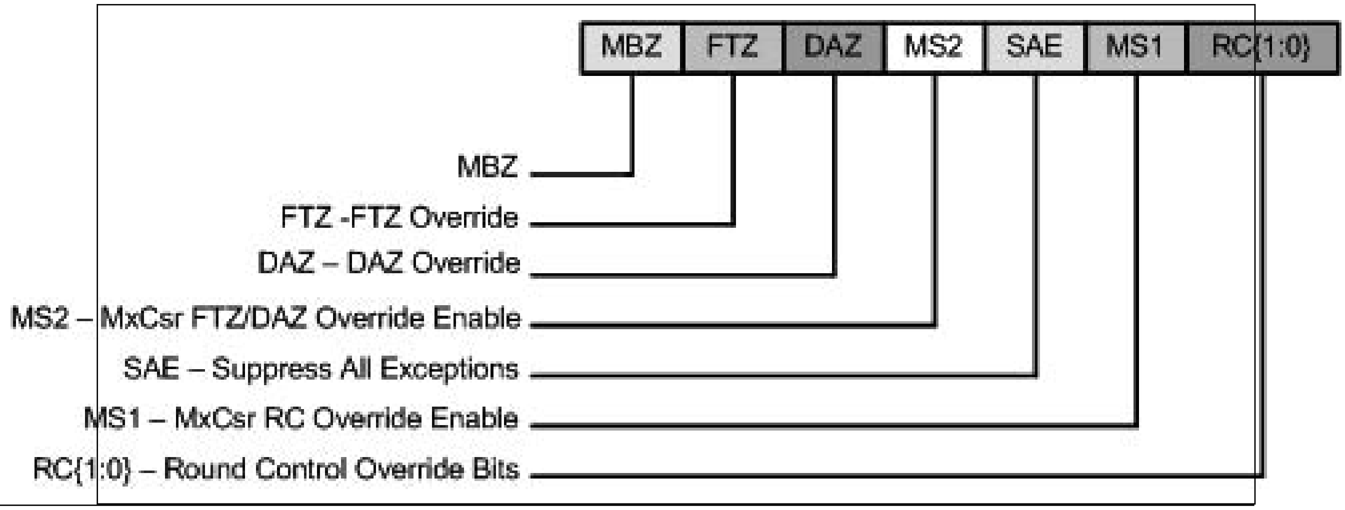
The intrinsics are translated into library calls that do the actual conversions.

The half-float intrinsics are available on IA-32 and Intel® 64 architectures running supported operating systems. The minimum processor requirement is an Intel® Pentium 4 processor and an operating system supporting Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instructions.

Role of Immediate Byte in Half Float Intrinsic Operations

For all half-float intrinsics an immediate byte controls rounding mode, flush to zero, and other non-volatile set values. The format of the `imm8` byte is as shown in the diagram below.

The `imm8` value is used for special `MXCSR` overrides.



In the diagram,

- MBZ = Most significant Bit is Zero; used for error checking
- MS1 = 1 : use MXCSR RC, else use imm8.RC
- SAE = 1 : all exceptions are suppressed
- MS2 = 1 : use MXCSR FTZ/DAZ control, else use imm8.FTZ/DAZ.

The compiler passes the bits to the library function, with error checking - the most significant bit must be zero.

Details About Intrinsics for Half Floats

There are four intrinsics for converting half-floats to 32-bit floats and 32-bit floats to half-floats. The prototypes for these half-float conversion intrinsics are in the `emmintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

`float _cvtsh_ss(unsigned short x);`

This intrinsic takes a half-float value, `x`, and converts it to a 32-bit float value, which is returned.

`unsigned short _cvtss_sh(float x, int imm);`

This intrinsic takes a 32-bit float value, `x`, and converts it to a half-float value, which is returned.

`__m128 _mm_cvtph_ps(__m128i x);`

This intrinsic takes four packed half-float values and converts them to four 32-bit float values, which are returned. The upper 64-bits of `x` are ignored. The lower 64-bits are taken as four 16-bit float values for conversion.

`__m128i _mm_cvtps_ph(_m128 x, int imm);`

This intrinsic takes four packed 32-bit float values and converts them to four half-float values, which are returned. The upper 64-bits in the returned result are all zeros. The lower 64-bits contain the four packed 16-bit float values.

Intrinsics for Short Vector Math Library Operations (SVML)

The compiler provides short vector math library (SVML) intrinsics to compute vector math functions. These intrinsics are available for IA-32 and Intel® 64 architectures running on supported operating systems. The prototypes for the SVML intrinsics are available in the `immintrin.h` file.

To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

The SVML intrinsics do not have any corresponding instructions.

The SVML intrinsics are vector variants of corresponding scalar math operations using `__m128`, `__m128d`, `__m256`, `__m256d`, and `__m256i` data types. They take packed vector arguments, perform the operation on each element of the packed vector argument, and return a packed vector result.

For example, the argument to the `_mm_sin_ps` intrinsic is a packed 128-bit vector of four 32-bit precision floating point numbers. The intrinsic computes the sine of each of these four numbers and returns the four results in a packed 128-bit vector.

Using SVML intrinsics is faster than repeatedly calling the scalar math functions. However, the intrinsics differ from the scalar functions in accuracy.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

This section also includes information about 512-bit intrinsics for SVML.

Intrinsics for Division Operations (512-bit)

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction

`_mm512_div_pd`, `_mm512_mask_div_pd`, `_mm512_maskz_div_pd`
 	
`_mm512_div_round_pd`, `_mm512_mask_div_round_pd`, `_mm512_maskz_div_round_pd`
	Calculates quotient of rounded division operation of packed float64 elements.	`VDIVPD`

`_mm512_div_ps`, `_mm512_mask_div_ps`, `_mm512_maskz_div_ps`	Calculates quotient of rounded division operation of packed float32 elements.	`VDIVPS`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
_mm512_div_round_ps, _mm512_mask_div_round_ps, _mm512_maskz_div_round_ps		
_mm_mask_div_sd, _mm_maskz_div_sd _mm_div_round_sd, _mm_mask_div_round_sd, _mm_maskz_div_round_sd	Calculates quotient of rounded division operation of scalar float64 elements.	VDIVSD
_mm_mask_div_ss, _mm_maskz_div_ss _mm_div_round_ss, _mm_mask_div_round_ss, _mm_maskz_div_round_ss	Calculates quotient of rounded division operation of scalar float32 elements.	VDIVSS

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result
round	Rounding control values; these can be one of the following (along with the `sae` suppress all exceptions flag): • `_MM_FROUND_TO_NEAREST_INT` - rounds to nearest even • `_MM_FROUND_TO_NEG_INF` - rounds to negative infinity • `_MM_FROUND_TO_POS_INF` - rounds to positive infinity • `_MM_FROUND_TO_ZERO` - rounds to zero • `_MM_FROUND_CUR_DIRECTION` - rounds using default from MXCSR register

_mm512_div_pd

```
extern __m512d __cdecl _mm512_div_pd(__m512d a, __m512d b);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_pd

```
extern __m512d __cdecl _mm512_mask_div_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_pd

```
extern __m512d __cdecl _mm512_maskz_div_pd(__mmask8 k, __m512d a, __m512d b);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_div_round_pd

```
extern __m512d __cdecl _mm512_div_round_pd(__m512d a, __m512d b, int round);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_round_pd

```
extern __m512d __cdecl _mm512_mask_div_round_pd(__m512d src, __mmask8 k, __m512d a, __m512d b,
int round);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_round_pd

```
extern __m512d __cdecl _mm512_maskz_div_round_pd(__mmask8 k, __m512d a, __m512d b, int round);
```

Divides packed float64 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_div_ps

```
extern __m512 __cdecl _mm512_div_ps(__m512 a, __m512 b);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_ps

```
extern __m512 __cdecl _mm512_mask_div_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_ps

```
extern __m512 __cdecl _mm512_maskz_div__ps(__mmask16 k, __m512 a, __m512 b);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm512_div_round_ps

```
extern __m512 __cdecl _mm512_div_round_ps(__m512 a, __m512 b, int round);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result.

_mm512_mask_div_round_ps

```
extern __m512 __cdecl _mm512_mask_div_round_ps(__m512 src, __mmask16 k, __m512 a, __m512 b, int
round);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_maskz_div_round_ps

```
extern __m512 __cdecl _mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int round);
```

Divides packed float32 elements in *a* by packed elements in *b*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

_mm_mask_div_sd

```
extern __m128d __cdecl _mm_mask_div_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Divides lower float64 element in *a* by lower float64 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_div_sd

```
extern __m128d __cdecl _mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
```

Divides lower float64 element in *a* by lower float64 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_div_round_sd

```
extern __m128d __cdecl _mm_div_round_sd(__m128d a, __m128d b, int round);
```

Divides lower float64 element in *a* by lower float64 element in *b*, stores the result in lower destination element, and copies upper element from *a* to upper destination element.

_mm_mask_div_round_sd

```
extern __m128d __cdecl _mm_mask_div_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b, int
round);
```

Divides lower float64 element in *a* by lower float64 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_maskz_div_round_sd

```
extern __m128d __cdecl _mm_maskz_div_round_sd(__mmask8 k, __m128d a, __m128d b, int round);
```

Divides lower float64 element in *a* by lower float64 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

_mm_div_round_ss

```
extern __m128 __cdecl _mm_div_round_ss(__m128 a, __m128 b, int round);
```

Divides lower float32 element in *a* by lower float32 element in *b*, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements.

_mm_mask_div_round_ss

```
extern __m128 __cdecl _mm_mask_div_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b, int
round);
```

Divides lower float32 element in *a* by lower float32 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_div_round_ss

```
extern __m128 __cdecl _mm_maskz_div_round_ss(__mmask8 k, __m128 a, __m128 b, int round);
```

Divides lower float32 element in *a* by lower float32 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_mask_div_ss

```
extern __m128 __cdecl _mm_mask_div_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Divides lower float32 element in *a* by lower float32 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

_mm_maskz_div_ss

```
extern __m128 __cdecl _mm_maskz_div_ss(__mmask8 k, __m128 a, __m128 b);
```

Divides lower float32 element in *a* by lower float32 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

Intrinsics for Division Operations

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epi8/ _mm256_div_epi8

Calculates quotient of a division operation. Vector variant of `div()` function for signed 8-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epi8(__m128i v1, __m128i v2);
extern __m256i _mm256_div_epi8(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epi16/ _mm256_div_epi16

Calculates quotient of a division operation. Vector variant of `div()` function for signed 16-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epi16(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_div_epi16(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epi32/ _mm256_div_epi32

Calculates quotient of a division operation. Vector variant of `div()` function for signed 32-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epi32(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_div_epi32(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epi64/ _mm256_div_epi64

Calculates quotient of a division operation. Vector variant of `div()` function for signed 64-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epi64(__m128i v1, __m128i v2);
extern __m256i _mm256_div_epi64(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epu8/ _mm256_div_epu8

Calculates quotient of a division operation. Vector variant of `div()` function for unsigned 8-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epu8(__m128i v1, __m128i v2);
extern __m256i _mm256_div_epu8(__m256i v1, __m256i v2);
```

Parameters

v1	unsigned integer source vector containing the dividends
v2	unsigned integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epu16/ _mm256_div_epu16

Calculates quotient of a division operation. Vector variant of `div()` function for unsigned 16-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epu16(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_div_epu16(__m256i v1, __m256i v2);
```

Parameters

v1	unsigned integer source vector containing the dividends
v2	unsigned integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epu32/ _mm256_div_epu32

Calculates quotient of a division operation. Vector variant of `div()` function for unsigned 32-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epu32(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_div_epu32(__m256i v1, __m256i v2);
```

Parameters

v1	unsigned integer source vector containing the dividends
v2	unsigned integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_div_epu64/ _mm256_div_epu64

Calculates quotient of a division operation. Vector variant of `div()` function for unsigned 64-bit integer arguments.

Syntax

```
extern __m128i _mm_div_epu64(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_div_epu64(__m256i v1, __m256i v2);
```

Parameters

v1	unsigned integer source vector containing the dividends
v2	unsigned integer source vector containing the divisors

Description

Calculates the quotient by dividing value of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epi8/ _mm256_rem_epi8

Calculates remainder of a division operation. Vector variant of `rem()` function for signed 8-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epi8(__m128i v1, __m128i v2);
extern __m256i _mm256_rem_epi8(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epi16/ _mm256_rem_epi16

Calculates remainder of a division operation. Vector variant of `rem()` function for signed 16-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epi16(__m128i v1, __m128i v2);
extern __m256i _mm256_rem_epi16(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epi32/ _mm256_rem_epi32

Calculates remainder of a division operation. Vector variant of `rem()` function for signed 32-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epi32(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_rem_epi32(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epi64/ _mm256_rem_epi64

Calculates remainder of a division operation. Vector variant of `rem()` function for signed 64-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epi64(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_rem_epi64(__m256i v1, __m256i v2);
```

Parameters

v1	signed integer source vector containing the dividends
v2	signed integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epu8/ _mm256_rem_epu8

Calculates remainder of a division operation. Vector variant of `rem()` function for unsigned 8-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epu8(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_rem_epu8(__m256i v1, __m256i v2);
```

Parameters

v1	unsigned integer source vector containing the dividends
v2	unsigned integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epu16/ _mm256_rem_epu16

Calculates remainder of a division operation. Vector variant of `rem()` function for unsigned 16-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epu16(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_rem_epu16(__m256i v1, __m256i v2);
```

Parameters

v1	unsigned integer source vector containing the dividends

v2 unsigned integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epu32/ _mm256_rem_epu32

Calculates remainder of a division operation. Vector variant of `rem()` function for unsigned 32-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epu32(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_rem_epu32(__m256i v1, __m256i v2);
```

Parameters

v1 unsigned integer source vector containing the dividends

v2 unsigned integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_rem_epu64/ _mm256_rem_epu64

Calculates remainder of a division operation. Vector variant of `rem()` function for unsigned 64-bit integer arguments.

Syntax

```
extern __m128i _mm_rem_epu64(__m128i v1, __m128i v2);
```

```
extern __m256i _mm256_rem_epu64(__m256i v1, __m256i v2);
```

Parameters

v1	unsigned integer source vector containing the dividends
v2	unsigned integer source vector containing the divisors

Description

Calculates the remainder from division of `v1` vector elements by corresponding `v2` vector elements.

Returns

Returns the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Intrinsics for Error Function Operations (512-bit)

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_cdfnorm_pd`, `_mm512_mask_cdfnorm_pd`	Calculates cumulative distribution function for float64 vector elements.	**None.**
`_mm512_cdfnorm_ps`, `_mm512_mask_cdfnorm_ps`	Calculates cumulative distribution function for float32 vector elements.	**None.**
`_mm512_cdfnorminv_pd`, `_mm512_mask_cdfnorminv_pd`	Calculates inverse cumulative distribution function for float64 vector elements.	**None.**
`_mm512_cdfnorminv_ps`, `_mm512_mask_cdfnorminv_ps`	Calculates inverse cumulative distribution function for float32 vector elements.	**None.**
`_mm512_erf_pd`, `_mm512_mask_erf_pd`	Calculates error function for float64 vector elements.	**None.**
`_mm512_erf_ps`, `_mm512_mask_erf_ps`	Calculates error function for float32 vector elements.	**None.**
`_mm512_erfc_pd`, `_mm512_mask_erfc_pd`	Calculates complementary error function for float64 vector elements.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_erfc_ps`, `_mm512_mask_erfc_ps`	Calculates complementary error function for float32 vector elements.	**None.**
`_mm512_erfinv_pd`, `_mm512_mask_erfinv_pd`	Calculates inverse error function for float64 vector elements.	**None.**
`_mm512_erfinv_ps`, `_mm512_mask_erfinv_ps`	Calculates inverse error function for float32 vector elements.	**None.**
`_mm512_erfcinv_pd`, `_mm512_mask_ercfinv_pd`	Calculates inverse complementary error function for float64 vector elements.	**None.**
`_mm512_erfcinv_ps`, `_mm512_mask_ercfinv_ps`	Calculates inverse complementary error function for float32 vector elements.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result

_mm512_cdfnorm_pd

```
extern __m512d __cdecl _mm512_cdfnorm_pd(__m512d a);
```

Computes normalized central distribution function for float64 elements in *a*, and stores the result.

_mm512_mask_cdfnorm_pd

```
extern __m512d __cdecl _mm512_mask_cdfnorm_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes normalized central distribution function for float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cdfnorm_ps

```
extern __m512 __cdecl _mm512_cdfnorm_ps(__m512 a);
```

Computes normalized central distribution function for float32 elements in *a*, and stores the result.

_mm512_mask_cdfnorm_ps

```
extern __m512 __cdecl _mm512_mask_cdfnorm_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes normalized central distribution function for float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cdfnorminv_pd

```
extern __m512d __cdecl _mm512_cdfnorminv_pd(__m512d a);
```

Computes inverse normalized central distribution function for float64 elements in *a*, and stores the result.

_mm512_mask_cdfnorminv_pd

```
extern __m512d __cdecl _mm512_mask_cdfnorminv_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes inverse normalized central distribution function for float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cdfnorminv_ps

```
extern __m512 __cdecl _mm512_cdfnorminv_ps(__m512 a);
```

Computes inverse normalized central distribution function for float32 elements in *a*, and stores the result.

_mm512_mask_cdfnorminv_ps

```
extern __m512 __cdecl _mm512_mask_cdfnorminv_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes inverse normalized central distribution function for float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_erf_pd

```
extern __m512d __cdecl _mm512_erf_pd(__m512d a);
```

Computes error function of packed float64 elements in *a*, and stores the result.

_mm512_mask_erf_pd

```
extern __m512d __cdecl _mm512_mask_erf_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes error function of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_erf_ps

```
extern __m512 __cdecl _mm512_erf_ps(__m512 a);
```

Computes error function of packed float32 elements in *a*, and stores the result.

_mm512_mask_erf_ps

```
extern __m512 __cdecl _mm512_mask_erf_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes error function of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_mask_erfc_pd

```
extern __m512d __cdecl _mm512_erfc_pd(__m512d a);
```

Computes complex error function of packed float64 elements in *a*, and stores the result.

_mm512_mask_erfc_pd

```
extern __m512d __cdecl _mm512_mask_erfc_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes complex error function of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_erfc_ps

```
extern __m512 __cdecl _mm512_erfc_ps(__m512 a);
```

Computes complex error function of packed float32 elements in *a*, and stores the result.

_mm512_mask_erfc_ps

```
extern __m512 __cdecl _mm512_mask_erfc_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes complex error function of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_erfinv_pd

```
extern __m512d __cdecl _mm512_erfinv_pd(__m512d a);
```

Calculates the inverse error function of float64 vector *a* elements.

_mm512_mask_erfinv_pd

```
extern __m512d __cdecl _mm512_mask_erfinv_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes inverse error function of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_erfinv_ps

```
extern __m512 __cdecl _mm512_erfinv_ps(__m512 a);
```

Computes inverse error function of packed float32 elements in *a*, and stores the result.

_mm512_mask_erfinv_ps

```
extern __m512 __cdecl _mm512_mask_erfinv_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes inverse error function of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_erfcinv_pd

```
extern __m512d __cdecl _mm512_erfcinv_pd(__m512d a);
```

Computes inverse complex error function of packed float64 elements in *a*, and stores the result.

_mm512_mask_erfcinv_pd

```
extern __m512d __cdecl _mm512_mask_erfcinv_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes inverse complex error function of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_erfcinv_ps

```
extern __m512 __cdecl _mm512_erfcinv_ps(__m512 a);
```

Computes inverse complex error function of packed float32 elements in *a*, and stores the result.

_mm512_mask_erfcinv_ps

```
extern __m512 __cdecl _mm512_mask_erfcinv_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes inverse complex error function of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

Intrinsics for Error Function Operations

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cdfnorminv_pd, _mm256_cdfnorminv_pd

Calculates inverse cumulative distribution function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_cdfnorminv_pd(__m128d v1);
extern __m256d _mm256_cdfnorminv_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Returns the inverse cumulative distribution function of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cdfnorminv_ps, _mm256_cdfnorminv_ps

Calculates inverse cumulative distribution function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_cdfnorminv_ps(__m128 v1);
extern __m256 _mm256_cdfnorminv_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Returns the inverse cumulative distribution function of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_erf_pd, _mm256_erf_pd

Calculates error function. Vector variant of erf(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_erf_pd(__m128d v1);
```

```
extern __m256d _mm256_erf_pd(__m256d v1);
```

Arguments

v1	float64 vector used for the operation

Description

Calculates error function of v1 elements, which is defined as:

```
erf(x) = 2/sqrt(pi)* integral from 0 to x of exp(-t*t) dt
```

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_erf_ps, _mm256_erf_ps

Calculates error function. Vector variant of erf(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_erf_ps(__m128 v1);
```

```
extern __m256 _mm256_erf_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates error function of v1 elements, which is defined as:

```
erf(x) = 2/sqrt(pi)* integral from 0 to x of exp(-t*t) dt
```

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_erfc_pd, _mm256_erfc_pd

Calculates complementary error function. Vector variant of erfc(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_erfc_pd(__m128d v1);
extern __m256d _mm256_erfc_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the complementary error function of vector *v1* elements, which is defined as:

```
1.0 - erf(x)
```

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_erfc_ps, _mm256_erfc_ps

Calculates complementary error function. Vector variant of erfc(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_erfc_ps(__m128 v1);
```

```
extern __m256 _mm256_erfc_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the complementary error function of vector *v1* elements, which is

```
1.0 - erf(x)
```

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_erfinv_pd, _mm256_erfinv_pd

Calculates inverse error function. Vector variant of erfinv(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_erfinv_pd(__m128d v1);
```

```
extern __m256d _mm256_erfinv_pd(__m256d v1);
```

Arguments

v1	float64 vector used for the operation

Description

Calculates the inverse error function of *v1* elements, which is defined as:

```
1 / erf(x)
```

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_erfinv_ps, _mm256_erfinv_ps

Calculates inverse error function. Vector variant of erfinv(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_erfinv_ps(__m128 v1);
```

```
extern __m256 _mm256_erfinv_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the inverse error function of *v1* elements, which is defined as:

```
1 / erf(x)
```

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Intrinsics for Exponential Operations (512-bit)

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_pow_pd`, `_mm512_mask_pow_pd`	Calculates exponential value of float64 vector elements raised to the power of other float64 vector elements.	**None.**
`_mm512_pow_ps`, `_mm512_mask_mm512_pow_ps`	Calculates exponential value of float32 vector elements raised to the power of other float32 vector elements.	**None.**
`_mm512_exp10_pd`, `_mm512_mask_mm512_exp10_pd`	Calculates base-10 exponential value of float64 vector elements.	**None.**
`_mm512_exp10_ps`, `_mm512_mask_mm512_exp10_ps`	Calculates base-10 exponential value of float32 vector elements.	**None.**
`_mm512_exp2_pd`, `_mm512_mask_mm512_exp2_pd`	Calculates base-2 exponential value of float64 vector elements.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_exp2_ps`, `_mm512_mask_mm512_exp2_ps`	Calculates base-2 exponential value of float32 vector elements.	**None.**
`_mm512_exp_pd`, `_mm512_mask_mm512_exp_pd`	Calculates base-e exponential value of float64 vector elements.	**None.**
`_mm512_exp_ps`, `_mm512_mask_mm512_exp_ps`	Calculates base-e exponential value of float32 vector elements.	**None.**
`_mm512_expm1_pd`, `_mm512_mask_mm512_expm1_pd`	Calculates base-e exponential value of float64 vector elements minus one.	**None.**
`_mm512_expm1_ps`, `_mm512_mask_mm512_expm1_ps`	Calculates base-e exponential value of float32 vector elements minus one.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_pow_pd

```
extern __m512d __cdecl _mm512_pow_pd(__m512d a, __m512d b);
```

Calculates the exponential value of each float64 vector *a* element raised to the power of the corresponding vector *b* element, and stores the result.

_mm512_mask_pow_pd

```
extern __m512d __cdecl _mm512_mask_pow_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Calculates the exponential value of each float64 vector *a* element raised to the power of the corresponding vector *b* element, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_pow_ps

```
extern __m512 __cdecl _mm512_pow_ps(__m512 a, __m512 b);
```

Calculates the exponential value of each float32 vector *a* element raised to the power of the corresponding vector *b* element, and stores the result.

_mm512_mask_pow_ps

```
extern __m512 __cdecl _mm512_mask_pow_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Calculates the exponential value of each float32 vector *a* element raised to the power of the corresponding vector *b* element, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_exp10_pd

```
extern __m512d __cdecl _mm512_exp10_pd(__m512d a);
```

Computes the base-10 exponent of packed float64 elements in *a*, and stores the result.

_mm512_mask_exp10_pd

```
extern __m512d __cdecl _mm512_mask_exp10_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the base-10 exponent of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_exp10_ps

```
extern __m512 __cdecl _mm512_exp10_ps(__m512 a);
```

Computes the base-10 exponent of packed float32 elements in *a*, and stores the result.

_mm512_mask_exp10_ps

```
extern __m512 __cdecl _mm512_mask_exp10_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the base-10 exponent of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_exp2_pd

```
extern __m512d __cdecl _mm512_exp2_pd(__m512d a);
```

Computes the base-2 exponent of packed float64 elements in *a*, and stores the result.

_mm512_mask_exp2_pd

```
extern __m512d __cdecl _mm512_mask_exp2_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the base-2 exponent of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_exp2_ps

```
extern __m512 __cdecl _mm512_exp2_ps(__m512 a);
```

Computes the base-2 exponent of packed float32 elements in *a*, and stores the result.

_mm512_mask_exp2_ps

```
extern __m512 __cdecl _mm512_mask_exp2_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the base-2 exponent of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_exp_pd

```
extern __m512d __cdecl _mm512_exp_pd(__m512d a);
```

Calculates the exponential value of e (base of natural logarithms) raised to the power of float64 vector *a* elements.

_mm512_mask_exp_pd

```
extern __m512d __cdecl _mm512_mask_exp_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates the exponential value of e (base of natural logarithms) raised to the power of float64 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_exp_ps

```
extern __m512 __cdecl _mm512_exp_ps(__m512 a);
```

Calculates the exponential value of e (base of natural logarithms) raised to the power of float32 vector *a* elements.

_mm512_mask_exp_ps

```
extern __m512 __cdecl _mm512_mask_exp_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates the exponential value of e (base of natural logarithms) raised to the power of float32 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_expm1_pd

```
extern __m512d __cdecl _mm512_expm1_pd(__m512d a);
```

Calculates exponential value of e (base of natural logarithms), raised to the power of float64 vector *a* elements minus one.

_mm512_mask_expm1_pd

```
extern __m512d __cdecl _mm512_mask_expm1_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates exponential value of e (base of natural logarithms), raised to the power of float64 vector *a* elements minus one, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_expm1_ps

```
extern __m512 __cdecl _mm512_expm1_ps(__m512 a);
```

Calculates exponential value of e (base of natural logarithms), raised to the power of float32 vector *a* elements minus one.

_mm512_mask_expm1_ps

```
extern __m512 __cdecl _mm512_mask_expm1_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates exponential value of e (base of natural logarithms), raised to the power of float32 vector *a* elements minus one, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

Intrinsics for Exponential Operations

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_exp2_pd, _mm256_exp2_pd

Calculates exponential value of 2. Vector variant of exp2(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_exp2_pd(__m128d v1);
```

```
extern __m256d _mm256_exp2_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the exponential value of 2 raised to the power of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_exp2_ps, _mm256_exp2_ps

Calculates exponential value of 2. Vector variant of exp2(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_exp2_ps(__m128 v1);
```

```
extern __m256 _mm256_exp2_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the exponential value of 2 raised to the power of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_exp_pd, _mm256_exp_pd

Calculates exponential value of e (base of natural logarithms). Vector variant of exp(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_exp_pd(__m128d v1);
```

```
extern __m256d _mm256_exp_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the exponential value of *e* (base of natural logarithms) raised to the power of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_exp_ps, _mm256_exp_ps

Calculates exponential value of e (base of natural logarithms). Vector variant of exp(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_exp_ps(__m128 v1);
```

```
extern __m256 _mm256_exp_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the exponential value of *e* (base of natural logarithms) raised to the power of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_exp10_pd, _mm256_exp10_pd

Calculates exponential value of 10. Vector variant of exp(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_exp10_pd(__m128d v1);
```

```
extern __m256d _mm256_exp10_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates 10 raised to the power of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_exp10_ps, _mm256_exp10_ps

Calculates exponential value of 10. Vector variant of exp(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_exp10_ps(__m128 v1);
```

```
extern __m256 _mm256_exp10_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates 10 raised to the power of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_expm1_pd, _mm256_expm1_pd

Calculates exponential value of e (base of natural logarithms), raised to the power of vector elements minus 1. Vector variant of expm1(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_expm1_pd(__m128d v1);
```

```
extern __m256d _mm256_expm1_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates exponential value of *e* (base of natural logarithms), raised to the power of vector elements minus 1.

$$e^{(x)} - 1$$

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_expm1_ps, _mm256_expm1_ps

Calculates exponential value of e (base of natural logarithms), raised to the power of vector elements minus 1. Vector variant of expm1(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_expm1_ps(__m128 v1);
```

```
extern __m256 _mm256_expm1_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates exponential value of *e* (base of natural logarithms), raised to the power of vector elements minus 1.

```
e^(x) - 1
```

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cexp_ps, _mm256_cexp_ps

Calculates complex exponential value of e (base of natural logarithms). Vector variant of exp(x) function for a 128-bit/256-bit vector argument of _Complex float32 values.

Syntax

```
extern __m128 _mm_cexp_ps(__m128 v1);
```

```
extern __m256 _mm256_cexp_ps(__m256 v1);
```

Arguments

v1	vector with _Complex float32 values

Description

Calculates the complex exponential value of *e* (base of natural logarithms) raised to the power of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_pow_pd, _mm256_pow_pd

Calculates exponential value of one argument raised to the other argument. Vector variant of pow(x, y) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_pow_pd(__m128d v1, __m128d v2);
```

```
extern __m256d _mm256_pow_pd(__m256d v1, __m256d v2);
```

Arguments

v1	vector with float64 values
v2	vector with float64 values

Description

Calculates the exponential value of each vector *v1* element raised to the power of the corresponding vector *v2* element.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_pow_ps, _mm256_pow_ps

Calculates exponential value of one argument raised to the other argument. Vector variant of pow(x, y) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_pow_ps(__m128 v1, __m128 v2);
```

```
extern __m256 _mm256_pow_ps(__m256 v1, __m256 v2);
```

Arguments

v1	vector with float32 values
v2	vector with float32 values

Description

Calculates the exponential value of each vector *v1* element raised to the power of the corresponding vector *v2* element.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_hypot_pd, _mm256_hypot_pd

Computes the length of the hypotenuse of a right angled triangle. Vector variant of hypot(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_hypot_pd(__m128d v1, __m128d v2);
```

```
extern __m256d _mm256_hypot_pd(__m256d v1, __m256d v2);
```

Arguments

v1	vector with float64 values
v2	vector with float64 values

Description

Computes the length of the hypotenuse of a right angled triangle with sides *v1* and *v2*, defined by:

$$\mathrm{sqrt}\ (v1^2 + v2^2)$$

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_hypot_ps, _mm256_hypot_ps

Computes the length of the hypotenuse of a right angled triangle. Vector variant of hypot(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_hypot_ps(__m128 v1, __m128 v2);
```

```
extern __m256 _mm256_hypot_ps(__m256 v1, __m256 v2);
```

Arguments

v1	vector with float32 values
v2	vector with float32 values

Description

Computes the length of the hypotenuse of a right angled triangle with sides *v1* and *v2*, defined by:

$$\mathrm{sqrt}\ (v1^2 + v2^2)$$

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Intrinsics for Logarithmic Operations (512-bit)

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_log10_pd`, `_mm512_mask_log10_pd`	Calculates base-10 logarithm.	**None.**
`_mm512_log10_ps`, `_mm512_mask_log10_ps`	Calculates base-10 logarithm.	**None.**
`_mm512_log1p_pd`, `_mm512_mask_log1p_pd`	Calculates natural logarithm.	**None.**
`_mm512_log1p_ps`, `_mm512_mask_log1p_ps`	Calculates signed exponent.	**None.**
`_mm512_log2_pd`, `_mm512_mask_log2_pd`	Calculates base-2 logarithm.	**None.**
`_mm512_log_pd`, `_mm512_mask_log_pd`	Calculates natural logarithm.	**None.**
`_mm512_log_ps`, `_mm512_mask_log_ps`	Calculates natural logarithm.	**None.**
`_mm512_logb_pd`, `_mm512_mask_logb_pd`	Calculates signed exponent.	**None.**
`_mm512_logb_ps`, `_mm512_mask_logb_ps`	Calculates signed exponent.	**None.**

variable	definition
k	zeromask used as a selector
k	writemask used as a selector
a	first source vector element
b	second source vector element
c	third source vector element
src	source element

_mm512_log10_pd

```
extern __m512d __cdecl _mm512_log10_pd(__m512d a);
```

Calculates the base-10 logarithm of vector *a* elements.

_mm512_mask_log10_pd

```
extern __m512d __cdecl _mm512_mask_log10_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates the base-10 logarithm of vector *a* elements.

_mm512_log10_ps

```
extern __m512 __cdecl _mm512_log10_ps(__m512 a);
```

Calculates the base-10 logarithm of vector *a* elements.

_mm512_mask_log10_ps

```
extern __m512 __cdecl _mm512_mask_log10_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates the base-10 logarithm of vector *a* elements.

_mm512_log1p_pd

```
extern __m512d __cdecl _mm512_log1p_pd(__m512d a);
```

Calculates the natural logarithm of vector *a* elements, defined by: `ln` $^{(v1 + 1)}$

_mm512_mask_log1p_pd

```
extern __m512d __cdecl _mm512_mask_log1p_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates the natural logarithm of vector *a* elements, defined by: `ln` $^{(v1 + 1)}$

_mm512_log1p_ps

```
extern __m512 __cdecl _mm512_log1p_ps(__m512 a);
```

Calculates the natural logarithm of vector *a* elements, defined by: `ln` $^{(v1 + 1)}$

_mm512_mask_log1p_ps

```
extern __m512 __cdecl _mm512_mask_log1p_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates the natural logarithm of vector *a* elements, defined by: `ln` $^{(v1 + 1)}$

_mm512_log2_pd

```
extern __m512d __cdecl _mm512_log2_pd(__m512d a);
```

Calculates the base-2 logarithm of vector *a* elements.

_mm512_mask_log2_pd

```
extern __m512d __cdecl _mm512_mask_log2_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates the base-2 logarithm of vector *a* elements.

_mm512_log_pd

```
extern __m512d __cdecl _mm512_log_pd(__m512d a);
```

Calculates the natural (base-e) logarithm of vector *a* elements.

_mm512_mask_log_pd

```
extern __m512d __cdecl _mm512_mask_log_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates the natural (base-e) logarithm of vector *a* elements.

_mm512_log_ps

```
extern __m512 __cdecl _mm512_log_ps(__m512 a);
```

Calculates the natural (base-e) logarithm of vector *a* elements.

_mm512_mask_log_ps

```
extern __m512 __cdecl _mm512_mask_log_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates the natural (base-e) logarithm of vector *a* elements.

_mm512_logb_pd

```
extern __m512d __cdecl _mm512_logb_pd(__m512d a);
```

Calculates the signed exponent of vector *a* elements.

_mm512_mask_logb_pd

```
extern __m512d __cdecl _mm512_mask_logb_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates the signed exponent of vector *a* elements.

_mm512_logb_ps

```
extern __m512 __cdecl _mm512_logb_ps(__m512 a);
```

Calculates the signed exponent of vector *a* elements.

_mm512_mask_logb_ps

```
extern __m512 __cdecl _mm512_mask_logb_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates the signed exponent of vector *a* elements.

Intrinsics for Logarithmic Operations

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log2_pd, _mm256_log2_pd

Calculates base-2 logarithm. Vector variant of log2(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_log2_pd(__m128d v1);
```

```
extern __m256d _mm256_log2_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the base-2 logarithm of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log2_ps, _mm256_log2_ps

Calculates base-2 logarithm. Vector variant of log2(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_log2_ps(__m128 v1);
extern __m256 _mm256_log2_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the base-2 logarithm of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log10_pd, _mm256_log10_pd

Calculates base-10 logarithm. Vector variant of log10(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_log10_pd(__m128d v1);
extern __m256d _mm256_log10_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the base-10 logarithm of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log10_ps, _mm256_log10_ps

Calculates base-10 logarithm. Vector variant of log10(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_log10_ps(__m128 v1);
extern __m256 _mm256_log10_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the base-10 logarithm of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log_pd, _mm256_log_pd

Calculates natural logarithm. Vector variant of log(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_log_pd(__m128d v1);
extern __m256d _mm256_log_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the natural logarithm of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log_ps, _mm256_log_ps

Calculates natural logarithm. Vector variant of log(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_log_ps(__m128 v1);
extern __m256 _mm256_log_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the natural logarithm of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_logb_pd, _mm256_logb_pd

Calculates signed exponent. Vector variant of logb(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_logb_pd(__m128d v1);
extern __m256d _mm256_logb_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Returns the signed exponent for vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_logb_ps, _mm256_logb_ps

Calculates signed exponent. Vector variant of logb(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_logb_ps(__m128 v1);
```

```
extern __m256 _mm256_logb_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Returns the signed exponent for vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log1p_pd, _mm256_log1p_pd

Calculates natural logarithm. Vector variant of log1p(x) function for a 128-bit/256-bit vector arguments with float64 values.

Syntax

```
extern __m128d _mm_log1p_pd(__m128d v1);
```

```
extern __m256d _mm256_log1p_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Returns the natural logarithm of vector *v1* elements, defined by:

```
ln (v1 + 1)
```

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_log1p_ps, _mm256_log1p_ps

Calculates natural logarithm. Vector variant of log1p(x) function for a 128-bit/256-bit vector arguments with float32 values.

Syntax

```
extern __m128 _mm_log1p_ps(__m128 v1);
```

```
extern __m256 _mm256_log1p_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Returns the natural logarithm of vector *v1* elements, defined by:

```
ln (v1 + 1)
```

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_clog_ps, _mm256_clog_ps

Calculates complex natural logarithm. Vector variant of clog(x) function for a 128-bit/256-bit vector argument of _Complex float32 values.

Syntax

```
extern __m128 _mm_clog_ps(__m128 v1);
```

```
extern __m256 _mm256_clog_ps(__m256 v1);
```

Arguments

v1	vector with _Complex float32 values

Description

Calculates the complex natural logarithm of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Intrinsics for Reciprocal Operations (512-bit)

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_rcp14_pd`, `_mm512_mask_rcp14_pd`, `_mm512_maskz_rcp14_pd`	Computes the approximate reciprocal of packed float64 elements.	`VRCP14PD`
`_mm512_rcp14_ps`, `_mm512_mask_rcp14_ps`, `_mm512_maskz_rcp14_ps`	Computes the approximate reciprocal of packed float32 elements.	`VRCP14PS`
`_mm_rcp14_sd`, `_mm_mask_rcp14_sd`, `_mm_maskz_rcp14_sd`	Computes the approximate reciprocal of scalar float64 elements.	`VRCP14SD`
`_mm_rcp14_ss`, `_mm_mask_rcp14_ss`, `_mm_maskz_rcp14_ss`	Computes the approximate reciprocal of scalar float32 elements.	`VRCP14SS`
`_mm512_rcp28_pd`, `_mm512_mask_rcp28_pd`, `_mm512_maskz_rcp28_pd` `_mm512_rcp28_round_pd`, `_mm512_mask_rcp28_round_pd`, `_mm512_maskz_rcp28_round_pd`	Computes the approximate reciprocal of packed float64 elements with bounded relative error.	`VRCP28PD`

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm_rcp28_sd`, `_mm_mask_rcp28_sd`, `_mm_maskz_rcp28_sd` `_mm_rcp28_round_sd`, `_mm_mask_rcp28_round_sd`, `_mm_maskz_rcp28_round_sd`	Computes the approximate reciprocal of scalar float64 elements with bounded relative error.	`VRCP28SD`
`_mm512_rcp28_ps`, `_mm512_mask_rcp28_ps`, `_mm512_maskz_rcp28_ps` `_mm512_rcp28_round_ps`, `_mm512_mask_rcp28_round_ps`, `_mm512_maskz_rcp28_round_ps`	Computes the approximate reciprocal of packed float32 elements with bounded relative error.	`VRCP28PS`
`_mm_rcp28_ss`, `_mm_mask_rcp28_ss`, `_mm_maskz_rcp28_ss` `_mm_rcp28_round_ss`, `_mm_mask_rcp28_round_ss`, `_mm_maskz_rcp28_round_ss`	Computes the approximate reciprocal of scalar float32 elements with bounded relative error.	`VRCP28SS`
`_mm512_recip_pd`, `_mm512_mask_recip_pd`	Computes the approximate reciprocal of packed float64 elements.	**None.**
`_mm512_recip_ps`, `_mm512_mask_recip_ps`	Computes the approximate reciprocal of packed float32 elements.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_rcp14_pd

```
extern __m512d __cdecl _mm512_rcp14_pd(__m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result.

> **NOTE**
> The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm512_mask_rcp14_pd

```
extern __m512d __cdecl _mm512_mask_rcp14_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm512_maskz_rcp14_pd

```
extern __m512d __cdecl _mm512_maskz_rcp14_pd(__mmask8 k, __m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm512_rcp14_ps

```
extern __m512 __cdecl _mm512_rcp14_ps(__m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result.

NOTE
The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm512_mask_rcp14_ps

```
extern __m512 __cdecl _mm512_mask_rcp14_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm512_maskz_rcp14_ps

```
extern __m512 __cdecl _mm512_maskz_rcp14_ps(__mmask16 k, __m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm_rcp14_sd

```
extern __m128d __cdecl _mm_rcp14_sd(__m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element, and copies upper element from *a* to upper destination element.

> **NOTE**
> The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm_mask_rcp14_sd

```
extern __m128d __cdecl _mm_mask_rcp14_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

> **NOTE**
> The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm_maskz_rcp14_sd

```
extern __m128d __cdecl _mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

> **NOTE**
> The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm_rcp14_ss

```
extern __m128 __cdecl _mm_rcp14_ss(__m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements.

> **NOTE**
> The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm_mask_rcp14_ss

```
extern __m128 __cdecl _mm_mask_rcp14_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm_maskz_rcp14_ss

```
extern __m128 __cdecl _mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-14)}$.

_mm512_rcp28_round_pd

```
extern __m512d __cdecl _mm512_rcp28_round_pd(__m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_round_pd

```
extern __m512d __cdecl _mm512_mask_rcp28_round_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_round_pd

```
extern __m512d __cdecl _mm512_maskz_rcp28_round_pd(__mmask8 k, __m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_rcp28_pd

```
extern __m512d __cdecl _mm512_rcp28_pd(__m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_pd

```
extern __m512d __cdecl _mm512_mask_rcp28_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_pd

```
extern __m512d __cdecl _mm512_maskz_rcp28_pd(__mmask8 k, __m512d a);
```

Computes the approximate reciprocal of packed float64 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_rcp28_round_ps

```
extern __m512 __cdecl _mm512_rcp28_round_ps(__m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_round_ps

```
extern __m512 __cdecl _mm512_mask_rcp28_round_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_round_ps

```
extern __m512 __cdecl _mm512_maskz_rcp28_round_ps(__mmask16 k, __m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_rcp28_ps

```
extern __m512 __cdecl _mm512_rcp28_ps(__m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_ps

```
extern __m512 __cdecl _mm512_mask_rcp28_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_ps

```
extern __m512 __cdecl _mm512_maskz_rcp28_ps(__mmask16 k, __m512 a);
```

Computes the approximate reciprocal of packed float32 elements in *a*, and stores the result using zeromask *k* (elements are zeroed out when the corresponding mask bit is not set).

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_rcp28_round_sd

```
extern __m128d __cdecl _mm512_rcp28_round_sd(__m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element, and copies upper element from *a* to upper destination element.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_round_sd

```
extern __m128d __cdecl _mm512_mask_rcp28_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_round_sd

```
extern __m128d __cdecl _mm512_maskz_rcp28_round_sd(__mmask8 k, __m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_rcp28_round_sd

```
extern __m128d __cdecl _mm512_rcp28_round_sd(__m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element, and copies upper element from *a* to upper destination element.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_round_sd

```
extern __m128d __cdecl _mm512_mask_rcp28_round_sd(__m128d src, __mmask8 k, __m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_round_sd

```
extern __m128d __cdecl _mm512_maskz_rcp28_round_sd(__mmask8 k, __m128d a, __m128d b);
```

Computes the approximate reciprocal of lower float64 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper element from *a* to upper destination element.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_rcp28_round_ss

```
extern __m128 __cdecl _mm512_rcp28_round_ss(__m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_round_ss

```
extern __m128 __cdecl _mm512_mask_rcp28_round_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_round_ss

```
extern __m128 __cdecl _mm512_maskz_rcp28_round_ss(__mmask8 k, __m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_rcp28_ss

```
extern __m128 __cdecl _mm512_rcp28_ss(__m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element, and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_mask_rcp28_ss

```
extern __m128 __cdecl _mm512_mask_rcp28_ss(__m128 src, __mmask8 k, __m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element using writemask *k* (the element is copied from *src* when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_maskz_rcp28_ss

```
extern __m128 __cdecl _mm512_maskz_rcp28_ss(__mmask8 k, __m128 a, __m128 b);
```

Computes approximate reciprocal of lower float32 element in *b*, stores the result in lower destination element using zeromask *k* (the element is zeroed out when mask bit 0 is not set), and copies upper three packed elements from *a* to upper destination elements.

NOTE
The maximum relative error for this approximation is less than $2^{(-28)}$.

_mm512_recip_pd

```
extern __m512d __cdecl _mm512_recip_pd(__m512d a);
```

Computes approximate reciprocal of float64 elements in *a*, and stores the result.

_mm512_mask_recip_pd

```
extern __m512d __cdecl _mm512_mask_recip_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes approximate reciprocal of float64 elements in *a*, and stores the result using writemask *k* (the element is copied from *src* when mask bit 0 is not set).

_mm512_recip_ps

```
extern __m512 __cdecl _mm512_recip_ps(__m512 a);
```

Computes approximate reciprocal of float32 elements in *a*, and stores the result.

_mm512_mask_recip_ps

```
extern __m512 __cdecl _mm512_mask_recip_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes approximate reciprocal of float32 elements in *a*, and stores the result using writemask *k* (the element is copied from *src* when mask bit 0 is not set).

Intrinsics for Root Function Operations (512-bit)

The prototypes for Intel© Advanced Vector Extensions 512 (Intel© AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_sqrt_pd`, `_mm512_mask_sqrt_pd`	Calculates square root of float64 vector elements.	**None.**
`_mm512_sqrt_ps`, `_mm512_mask_sqrt_ps`	Calculates square root of float32 vector elements.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_invsqrt_pd`, `_mm512_mask_invsqrt_pd`	Calculates inverse square root of float64 vector elements.	**None.**
`_mm512_invsqrt_ps`, `_mm512_mask_invsqrt_ps`	Calculates inverse square root of float32 vector elements.	**None.**
`_mm512_hypot_pd`, `_mm512_mask_hypot_pd`	Calculates square root of float64 vector elements.	**None.**
`_mm512_hypot_ps`, `_mm512_mask_hypot_ps`	Calculates square root of float32 vector elements.	**None.**
`_mm512_cbrt_pd`, `_mm512_mask_cbrt_pd`	Calculates cube root of float64 vector elements.	**None.**
`_mm512_cbrt_ps`, `_mm512_mask_cbrt_ps`	Calculates cube root of float32 vector elements.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_sqrt_pd

```
extern __m512d __cdecl _mm512_sqrt_pd(__m512d a);
```

Calculates square root value of float64 vector *a* elements.

_mm512_mask_sqrt_pd

```
extern __m512d __cdecl _mm512_mask_sqrt_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates square root value of float64 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sqrt_ps

```
extern __m512 __cdecl _mm512_sqrt_ps(__m512 a);
```

Calculates square root value of float32 vector *a* elements.

_mm512_mask_sqrt_ps

```
extern __m512 __cdecl _mm512_mask_sqrt_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates square root value of float32 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_invsqrt_pd

```
extern __m512d __cdecl _mm512_invsqrt_pd(__m512d a);
```

Calculates inverse square root value of float64 vector *a* elements.

_mm512_mask_invsqrt_pd

```
extern __m512d __cdecl _mm512_mask_invsqrt_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates inverse square root value of float64 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_invsqrt_ps

```
extern __m512 __cdecl _mm512_invsqrt_ps(__m512 a);
```

Calculates inverse square root value of float32 vector *a* elements.

_mm512_mask_invsqrt_ps

```
extern __m512 __cdecl _mm512_mask_invsqrt_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates inverse square root value of float32 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_hypot_pd

```
extern __m512d __cdecl _mm512_hypot_pd(__m512d a, __m512d b);
```

Computes the length of the hypotenuse of a right angled triangle with sides from float64 vector *a* and *b* elements.

_mm512_mask_hypot_pd

```
extern __m512d __cdecl _mm512_mask_hypot_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Computes the length of the hypotenuse of a right angled triangle with sides from float64 vector *a* and *b* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_hypot_ps

```
extern __m512 __cdecl _mm512_hypot_ps(__m512 a, __m512 b);
```

Computes the length of the hypotenuse of a right angled triangle with sides from float32 vector *a* and *b* elements.

_mm512_mask_hypot_ps

```
extern __m512 __cdecl _mm512_mask_hypot_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Computes the length of the hypotenuse of a right angled triangle with sides from float32 vector *a* and *b* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cbrt_pd

```
extern __m512d __cdecl _mm512_cbrt_pd(__m512d a);
```

Calculates the cube root of float64 vector *a* elements.

_mm512_mask_cbrt_pd

```
extern __m512d __cdecl _mm512_mask_cbrt_pd(__m512d src, __mmask8 k, __m512d a);
```

Calculates the cube root of float64 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cbrt_ps

```
extern __m512 __cdecl _mm512_cbrt_ps(__m512 a);
```

Calculates the cube root of float32 vector *a* elements.

_mm512_mask_cbrt_ps

```
extern __m512 __cdecl _mm512_mask_cbrt_ps(__m512 src, __mmask16 k, __m512 a);
```

Calculates the cube root of float32 vector *a* elements, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

Intrinsics for Rounding Operations (512-bit)

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_ceil_pd`, `_mm512_mask_ceil_pd`	Rounds float64 vector elements to nearest upper integer.	**None.**
`_mm512_ceil_ps`, `_mm512_mask_ceil_ps`	Rounds float32 vector elements to nearest upper integer.	**None.**
`_mm512_floor_pd`, `_mm512_mask_floor_pd`	Rounds float64 vector elements to nearest lower integer.	**None.**
`_mm512_floor_ps`, `_mm512_mask_floor_ps`	Rounds float32 vector elements to nearest lower integer.	**None.**
`_mm512_nearbyint_pd`, `_mm512_mask_nearbyint_pd`	Rounds float64 vector elements to nearest integer in floating point format.	**None.**
`_mm512_nearbyint_ps`, `_mm512_mask_nearbyint_ps`	Rounds float32 vector elements to nearest integer in floating point format.	**None.**
`_mm512_rint_pd`, `_mm512_mask_rint_pd`	Rounds float64 vector elements to nearest even integer.	**None.**
`_mm512_rint_ps`, `_mm512_mask_rint_ps`	Rounds float32 vector elements to nearest even integer.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_svml_round_pd`, `_mm512_mask_svml_round_pd`	Rounds float64 vector elements to nearest integer.	**None.**
`_mm512_trunc_pd`, `_mm512_mask_trunc_pd`	Rounds float64 vector elements to nearest integer not larger in absolute value.	**None.**
`_mm512_trunc_ps`, `_mm512_mask_trunc_ps`	Rounds float32 vector elements to nearest integer not larger in absolute value.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
src	source element to use based on writemask result

_mm512_ceil_pd

```
extern __m512d __cdecl _mm512_ceil_pd(__m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest upper integer value.

_mm512_mask_ceil_pd

```
extern __m512d __cdecl _mm512_mask_ceil_pd(__m512d src, __mmask8 k, __m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest upper integer value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_ceil_ps

```
extern __m512 __cdecl _mm512_ceil_ps(__m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest upper integer value.

_mm512_mask_ceil_ps

```
extern __m512 __cdecl _mm512_mask_ceil_ps(__m512 src, __mmask16 k, __m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest upper integer value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_floor_pd

```
extern __m512d __cdecl _mm512_floor_pd(__m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest lower integer value.

_mm512_mask_floor_pd

```
extern __m512d __cdecl _mm512_mask_floor_pd(__m512d src, __mmask8 k, __m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest lower integer value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_floor_ps

```
extern __m512 __cdecl _mm512_floor_ps(__m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest lower integer value.

_mm512_mask_floor_ps

```
extern __m512 __cdecl _mm512_mask_floor_ps(__m512 src, __mmask16 k, __m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest lower integer value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_nearbyint_pd

```
extern __m512d __cdecl _mm512_nearbyint_pd(__m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest integer value in floating point format without raising the inexact exception.

_mm512_mask_nearbyint_pd

```
extern __m512d __cdecl _mm512_mask_nearbyint_pd(__m512d src, __mmask8 k, __m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest integer value in floating point format without raising the inexact exception, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_nearbyint_ps

```
extern __m512 __cdecl _mm512_nearbyint_ps(__m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest integer value in floating point format without raising the inexact exception.

_mm512_mask_nearbyint_ps

```
extern __m512 __cdecl _mm512_mask_nearbyint_ps(__m512 src, __mmask16 k, __m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest integer value in floating point format without raising the inexact exception, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_rint_pd

```
extern __m512d __cdecl _mm512_rint_pd(__m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest even integer value.

_mm512_mask_rint_pd

```
extern __m512d __cdecl _mm512_mask_rint_pd(__m512d src, __mmask8 k, __m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest even integer value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_rint_ps

```
extern __m512 __cdecl _mm512_rint_ps(__m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest even integer value.

_mm512_mask_rint_ps

```
extern __m512 __cdecl _mm512_mask_rint_ps(__m512 src, __mmask16 k, __m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest even integer value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_svml_round_pd

```
extern __m512d __cdecl _mm512_svml_round_pd(__m512d a);
```

Rounds off the elements of vector *a* to the nearest integer value. This intrinsic rounds the halfway cases away from zero regardless of the current rounding direction, instead of to the nearest even integer like the `_mm512_rint_pd` intrinsic.

_mm512_mask_svml_round_pd

```
extern __m512d __cdecl _mm512_mask_svml_round_pd(__m512d src, __mmask8 k, __m512d a);
```

Rounds off the elements of vector *a* to the nearest integer value. This intrinsic rounds the halfway cases away from zero regardless of the current rounding direction, instead of to the nearest even integer like the `_mm512_rint_pd` intrinsic.

The result is stored using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set)

_mm512_trunc_pd

```
extern __m512d __cdecl _mm512_trunc_pd(__m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest integer value which is not larger in absolute value.

_mm512_mask_trunc_pd

```
extern __m512d __cdecl _mm512_mask_trunc_pd(__m512d src, __mmask8 k, __m512d a);
```

Rounds off the elements of float64 vector *a* to the nearest integer value which is not larger in absolute value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_trunc_ps

```
extern __m512 __cdecl _mm512_trunc_ps(__m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest integer value which is not larger in absolute value.

_mm512_mask_trunc_ps

```
extern __m512 __cdecl _mm512_mask_trunc_ps(__m512 src, __mmask16 k, __m512 a);
```

Rounds off the elements of float32 vector *a* to the nearest integer value which is not larger in absolute value, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

Intrinsics for Square Root and Cube Root Operations

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sqrt_pd, _mm256_sqrt_pd

Calculates square root value. Vector variant of sqrt(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_sqrt_pd(__m128d v1);
```

```
extern __m256d _mm256_sqrt_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the square root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sqrt_ps, _mm256_sqrt_ps

Calculates square root value. Vector variant of sqrt(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_sqrt_ps(__m128 v1);
```

```
extern __m256 _mm256_sqrt_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the square root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_invsqrt_pd, _mm256_invsqrt_pd

Calculates inverse square root value. Vector variant of invsqrt(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_invsqrt_pd(__m128d v1);
```

```
extern __m256d _mm256_invsqrt_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the inverse square root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_invsqrt_ps, _mm256_invsqrt_ps

Calculates inverse square root value. Vector variant of invsqrt(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_invsqrt_ps(__m128 v1);
```

```
extern __m256 _mm256_invsqrt_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the inverse square root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cbrt_pd, _mm256_cbrt_pd

Calculates cube root value. Vector variant of cbrt(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_cbrt_pd(__m128d v1);
```

```
extern __m256d _mm256_cbrt_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the cube root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cbrt_ps, _mm256_cbrt_ps

Calculates cube root value. Vector variant of cbrt(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_cbrt_ps(__m128 v1);
```

```
extern __m256 _mm256_cbrt_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the cube root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_invcbrt_pd, _mm256_invcbrt_pd

Calculates inverse cube root value. Vector variant of invcbrt(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_invcbrt_pd(__m128d v1);
```

```
extern __m256d _mm256_invcbrt_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the inverse cube root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_invcbrt_ps, _mm256_invcbrt_ps

Calculates inverse cube root value. Vector variant of invcbrt(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_invcbrt_ps(__m128 v1);
```

```
extern __m256 _mm256_invcbrt_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the inverse cube root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_csqrt_ps, _mm256_csqrt_ps

Calculates complex square root value. Vector variant of csqrt(x) function for a 128-bit/256-bit vector argument of _Complex float32 values.

Syntax

```
extern __m128 _mm_csqrt_ps(__m128 v1);
```

```
extern __m256 _mm256_csqrt_ps(__m256 v1);
```

Arguments

v1	vector with _Complex float32 values

Description

Calculates the complex square root of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Intrinsics for Trigonometric Operations (512-bit)

The prototypes for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) intrinsics are located in the `zmmintrin.h` header file.
To use these intrinsics, include the `immintrin.h` file as follows:

```
#include <immintrin.h>
```

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_acos_pd`, `_mm512_mask_acos_pd`	Calculates inverse cosine value for float64 vector elements.	**None.**
`_mm512_acos_ps`, `_mm512_mask_acos_ps`	Calculates inverse cosine value for float32 vector elements.	**None.**
`_mm512_acosh_pd`, `_mm512_mask_acosh_pd`	Calculates inverse hyperbolic cosine value for float64 vector elements.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_acosh_ps`, `_mm512_mask_acosh_ps`	Calculates inverse hyperbolic cosine value for float32 vector elements.	**None.**
`_mm512_asin_pd`, `_mm512_mask_asin_pd`	Calculates inverse sine value for float64 vector elements.	**None.**
`_mm512_asin_ps`, `_mm512_mask_asin_ps`	Calculates inverse sine value for float32 vector elements.	**None.**
`_mm512_asinh_pd`, `_mm512_mask_asinh_pd`	Calculates inverse hyperbolic sine value for float64 vector elements.	**None.**
`_mm512_asinh_ps`, `_mm512_mask_asinh_ps`	Calculates inverse hyperbolic sine value for float32 vector elements.	**None.**
`_mm512_atan_pd`, `_mm512_mask_atan_pd`	Calculates inverse tangent value for float64 vector elements.	**None.**
`_mm512_atan_ps`, `_mm512_mask_atan_ps`	Calculates inverse tangent value for float32 vector elements.	**None.**
`_mm512_atan2_pd`, `_mm512_mask_atan2_pd`	Calculates inverse tangent value for float64 elements from multiple vectors.	**None.**
`_mm512_atan2_ps`, `_mm512_mask_atan2_ps`	Calculates inverse tangent value for float32 elements from multiple vectors.	**None.**
`_mm512_atanh_pd`, `_mm512_mask_atanh_pd`	Calculates inverse hyperbolic tangent value for float64 vector elements.	**None.**
`_mm512_atanh_ps`, `_mm512_mask_atanh_ps`	Calculates inverse hyperbolic tangent value for float32 vector elements.	**None.**
`_mm512_cos_pd`, `_mm512_mask_cos_pd`	Calculates cosine value for float64 vector elements.	**None.**
`_mm512_cos_ps`, `_mm512_mask_cos_ps`	Calculates cosine value for float32 vector elements.	**None.**
`_mm512_cosd_pd`, `_mm512_mask_cosd_pd`	Calculates cosine value (in degrees) for float64 vector elements.	**None.**
`_mm512_cosd_ps`, `_mm512_mask_cosd_ps`	Calculates cosine value (in degrees) for float32 vector elements.	**None.**
`_mm512_cosh_pd`, `_mm512_mask_cosh_pd`	Calculates hyperbolic cosine value for float64 vector elements.	**None.**
`_mm512_cosh_ps`, `_mm512_mask_cosh_ps`	Calculates hyperbolic cosine value for float32 vector elements.	**None.**
`_mm512_sin_pd`, `_mm512_mask_sin_pd`	Calculates sine value for float64 vector elements.	**None.**

Intrinsic Name	Operation	Corresponding Intel® AVX-512 Instruction
`_mm512_sin_ps`, `_mm512_mask_sin_ps`	Calculates sine value for float32 vector elements.	**None.**
`_mm512_sincos_pd`, `_mm512_mask_sincos_pd`	Calculates the sine and cosine values for float64 vector elements.	**None.**
`_mm512_sincos_ps`, `_mm512_mask_sincos_ps`	Calculates the sine and cosine values for float32 vector elements.	**None.**
`_mm512_sind_pd`, `_mm512_mask_sind_pd`	Calculates sine value (in degrees) for float64 vector elements.	**None.**
`_mm512_sind_ps`, `_mm512_mask_sind_ps`	Calculates sine value (in degrees) for float32 vector elements.	**None.**
`_mm512_sinh_pd`, `_mm512_mask_sinh_pd`	Calculates hyperbolic sine value for float64 vector elements.	**None.**
`_mm512_sinh_ps`, `_mm512_mask_sinh_ps`	Calculates hyperbolic sine value for float32 vector elements.	**None.**
`_mm512_tan_pd`, `_mm512_mask_tan_pd`	Calculates tangent value for float64 vector elements.	**None.**
`_mm512_tan_ps`, `_mm512_mask_tan_ps`	Calculates tangent value for float32 vector elements.	**None.**
`_mm512_tand_pd`, `_mm512_mask_tand_pd`	Calculates tangent (in degrees) value for float64 vector elements.	**None.**
`_mm512_tand_ps`, `_mm512_mask_tand_ps`	Calculates tangent (in degrees) value for float32 vector elements.	**None.**
`_mm512_tanh_pd`, `_mm512_mask_tanh_pd`	Calculates hyperbolic tangent value for float64 vector elements.	**None.**
`_mm512_tanh_ps`, `_mm512_mask_tanh_ps`	Calculates hyperbolic tangent value for float32 vector elements.	**None.**

variable	definition
k	writemask used as a selector
a	first source vector element
b	second source vector element
src	source element to use based on writemask result

_mm512_acos_pd

```
extern __m512d __cdecl _mm512_acos_pd(__m512d a);
```

Computes the inverse cosine of packed float64 elements in *a*, and stores the result.

_mm512_mask_acos_pd

```
extern __m512d __cdecl _mm512_mask_acos_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the inverse cosine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_acos_ps

```
extern __m512 __cdecl _mm512_acos_ps(__m512 a);
```

Computes the inverse cosine of packed float32 elements in *a*, and stores the result.

_mm512_mask_acos_ps

```
extern __m512 __cdecl _mm512_mask_acos_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the inverse cosine of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_acosh_pd

```
extern __m512d __cdecl _mm512_acosh_pd(__m512d a);
```

Computes the inverse hyperbolic cosine of packed float64 elements in *a*, and stores the result.

_mm512_mask_acosh_pd

```
extern __m512d __cdecl _mm512_mask_acosh_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the inverse hyperbolic cosine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_acosh_ps

```
extern __m512 __cdecl _mm512_acosh_ps(__m512 a);
```

Computes the inverse hyperbolic cosine of packed float32 elements in *a*, and stores the result.

_mm512_mask_acosh_ps

```
extern __m512 __cdecl _mm512_mask_acosh_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the inverse hyperbolic cosine of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_asin_pd

```
extern __m512d __cdecl _mm512_asin_pd(__m512d a);
```

Computes the inverse sine of packed float64 elements in *a*, and stores the result.

_mm512_mask_asin_pd

```
extern __m512d __cdecl _mm512_mask_asin_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the inverse sine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_asin_ps

```
extern __m512 __cdecl _mm512_asin_ps(__m512 a);
```

Computes the inverse sine of packed float32 elements in *a*, and stores the result.

_mm512_mask_asin_ps

```
extern __m512 __cdecl _mm512_mask_asin_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the inverse sine of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_asinh_pd

```
extern __m512d __cdecl _mm512_asinh_pd(__m512d a);
```

Computes the inverse hyperbolic sine of packed float64 elements in *a*, and stores the result.

_mm512_mask_asinh_pd

```
extern __m512d __cdecl _mm512_mask_asinh_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the inverse hyperbolic sine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_asinh_ps

```
extern __m512 __cdecl _mm512_asinh_ps(__m512 a);
```

Computes the inverse hyperbolic sine of packed float32 elements in *a*, and stores the result.

_mm512_mask_asinh_ps

```
extern __m512 __cdecl _mm512_mask_asinh_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the inverse hyperbolic sine of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_atan2_pd

```
extern __m512d __cdecl _mm512_atan2_pd(__m512d a, __m512d b);
```

Computes the inverse tangent of packed float64 elements in *a* and *b*, and stores the result.

_mm512_mask_atan2_pd

```
extern __m512d __cdecl _mm512_mask_atan2_pd(__m512d src, __mmask8 k, __m512d a, __m512d b);
```

Computes the inverse tangent of packed float64 elements in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_atan2_ps

```
extern __m512 __cdecl _mm512_atan2_ps(__m512 a, __m512 b);
```

Computes the inverse tangent of packed float32 elements in *a* and *b*, and stores the result.

_mm512_mask_atan2_ps

```
extern __m512 __cdecl _mm512_mask_atan2_ps(__m512 src, __mmask16 k, __m512 a, __m512 b);
```

Computes the inverse tangent of packed float32 elements in *a* and *b*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_atan_pd

```
extern __m512d __cdecl _mm512_atan_pd(__m512d a);
```

Computes the inverse tangent of packed float64 elements in *a*, and stores the result.

_mm512_mask_atan_pd

```
extern __m512d __cdecl _mm512_mask_atan_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the inverse tangent of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_atan_ps

```
extern __m512 __cdecl _mm512_atan_ps(__m512 a);
```

Computes the inverse tangent of packed float32 elements in *a*, and stores the result.

_mm512_mask_atan_ps

```
extern __m512 __cdecl _mm512_mask_atan_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the inverse tangent of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_atanh_pd

```
extern __m512d __cdecl _mm512_atanh_pd(__m512d a);
```

Computes the inverse hyperbolic tangent of packed float64 elements in *a*, and stores the result.

_mm512_mask_atanh_pd

```
extern __m512d __cdecl _mm512_mask_atanh_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the inverse hyperbolic tangent of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_atanh_ps

```
extern __m512 __cdecl _mm512_atanh_ps(__m512 a);
```

Computes the inverse hyperbolic tangent of packed float32 elements in *a*, and stores the result.

_mm512_mask_atanh_ps

```
extern __m512 __cdecl _mm512_mask_atanh_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the inverse hyperbolic tangent of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cos_pd

```
extern __m512d __cdecl _mm512_cos_pd(__m512d a);
```

Computes the cosine of packed float64 elements in *a*, and stores the result.

_mm512_mask_cos_pd

```
extern __m512d __cdecl _mm512_mask_cos_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the cosine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cos_ps

```
extern __m512 __cdecl _mm512_cos_ps(__m512 a);
```

Computes the cosine of packed float32 elements in *a*, and stores the result.

_mm512_mask_cos_ps

```
extern __m512 __cdecl _mm512_mask_cos_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the cosine of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cosd_pd

```
extern __m512d __cdecl _mm512_cosd_pd(__m512d a);
```

Computes the cosine (in degrees) of packed float64 elements in *a*, and stores the result.

_mm512_mask_cosd_pd

```
extern __m512d __cdecl _mm512_mask_cosd_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the cosine (in degrees) of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cosd_ps

```
extern __m512 __cdecl _mm512_cosd_ps(__m512 a);
```

Computes the cosine (in degrees) of packed float32 elements in *a*, and stores the result.

_mm512_mask_cosd_ps

```
extern __m512 __cdecl _mm512_mask_cosd_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the cosine (in degrees) of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cosh_pd

```
extern __m512d __cdecl _mm512_cosh_pd(__m512d a);
```

Computes the hyperbolic cosine of packed float64 elements in *a*, and stores the result.

_mm512_mask_cosh_pd

```
extern __m512d __cdecl _mm512_mask_cosh_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the hyperbolic cosine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_cosh_ps

```
extern __m512 __cdecl _mm512_cosh_ps(__m512 a);
```

Computes the hyperbolic cosine of packed float32 elements in *a*, and stores the result.

_mm512_mask_cosh_ps

```
extern __m512 __cdecl _mm512_mask_cosh_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the hyperbolic cosine (in degrees) of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sin_pd

```
extern __m512d __cdecl _mm512_sin_pd(__m512d a);
```

Computes the sine of packed float64 elements in *a*, and stores the result.

_mm512_mask_sin_pd

```
extern __m512d __cdecl _mm512_mask_sin_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the sine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sin_ps

```
extern __m512 __cdecl _mm512_sin_ps(__m512 a);
```

Computes the sine of packed float32 elements in *a*, and stores the result.

_mm512_mask_sin_ps

```
extern __m512 __cdecl _mm512_mask_sin_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the sine (in degrees) of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sincos_pd

```
extern __m512d __cdecl _mm512_sincos_pd(__m512d a);
```

Computes the sine and cosine of packed float64 elements in *a*, and stores the result.

_mm512_mask_sincos_pd

```
extern __m512d __cdecl _mm512_mask_sincos_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the sine and cosine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sincos_ps

```
extern __m512 __cdecl _mm512_sincos_ps(__m512 a);
```

Computes the sine and cosine of packed float32 elements in *a*, and stores the result.

_mm512_mask_sincos_ps

```
extern __m512 __cdecl _mm512_mask_sincos_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the sineand cosine of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sind_pd

```
extern __m512d __cdecl _mm512_sind_pd(__m512d a);
```

Computes the sine (in degrees) of packed float64 elements in *a*, and stores the result.

_mm512_mask_sind_pd

```
extern __m512d __cdecl _mm512_mask_sind_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the sine (in degrees) packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sind_ps

```
extern __m512 __cdecl _mm512_sind_ps(__m512 a);
```

Computes the sine (in degrees) of packed float32 elements in *a*, and stores the result.

_mm512_mask_sind_ps

```
extern __m512 __cdecl _mm512_mask_sind_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the sine (in degrees) of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sinh_pd

```
extern __m512d __cdecl _mm512_sinh_pd(__m512d a);
```

Computes the hyperbolic sine of packed float64 elements in *a*, and stores the result.

_mm512_mask_sinh_pd

```
extern __m512d __cdecl _mm512_mask_sinh_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the hyperbolic sine of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_sinh_ps

```
extern __m512 __cdecl _mm512_sinh_ps(__m512 a);
```

Computes the hyperbolic sine of packed float32 elements in *a*, and stores the result.

_mm512_mask_sinh_ps

```
extern __m512 __cdecl _mm512_mask_sinh_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the hyperbolic cosine of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_tan_pd

```
extern __m512d __cdecl _mm512_tan_pd(__m512d a);
```

Computes the tangent of packed float64 elements in *a*, and stores the result.

_mm512_mask_tan_pd

```
extern __m512d __cdecl _mm512_mask_tan_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the tangent of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_tan_ps

```
extern __m512 __cdecl _mm512_tan_ps(__m512 a);
```

Computes the tangent of packed float32 elements in *a*, and stores the result.

_mm512_mask_tan_ps

```
extern __m512 __cdecl _mm512_mask_tan_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the tangent of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_tand_pd

```
extern __m512d __cdecl _mm512_tand_pd(__m512d a);
```

Computes the tangent (in degrees) of packed float64 elements in *a*, and stores the result.

_mm512_mask_tand_pd

```
extern __m512d __cdecl _mm512_mask_tand_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes tangent (in degrees) of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_tand_ps

```
extern __m512 __cdecl _mm512_tand_ps(__m512 a);
```

Computes the tangent (in degrees) of packed float32 elements in *a*, and stores the result.

_mm512_mask_tand_ps

```
extern __m512 __cdecl _mm512_mask_tand_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the tangent (in degrees) of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_tanh_pd

```
extern __m512d __cdecl _mm512_tanh_pd(__m512d a);
```

Computes the hyperbolic tangent of packed float64 elements in *a*, and stores the result.

_mm512_mask_tanh_pd

```
extern __m512d __cdecl _mm512_mask_tanh_pd(__m512d src, __mmask8 k, __m512d a);
```

Computes the hyperbolic tangent of packed float64 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

_mm512_tanh_ps

```
extern __m512 __cdecl _mm512_tanh_ps(__m512 a);
```

Computes the hyperbolic tangent of packed float32 elements in *a*, and stores the result.

_mm512_mask_tanh_ps

```
extern __m512 __cdecl _mm512_mask_tanh_ps(__m512 src, __mmask16 k, __m512 a);
```

Computes the hyperbolic tangent of packed float32 elements in *a*, and stores the result using writemask *k* (elements are copied from *src* when the corresponding mask bit is not set).

Intrinsics for Trigonometric Operations

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

For information about 512-bit trigonometric intrinsics for SVML, see Intrinsics for Trigonometric Operations (512-bit).

_mm_acos_pd, _mm256_acos_pd

Calculates inverse cosine value. Vector variant of acos(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_acos_pd(__m128d v1);
```

```
extern __m256d _mm256_acos_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the arc cosine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_acos_ps, _mm256_acos_ps

Calculates inverse cosine value. Vector variant of acos(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_acos_ps(__m128 v1);
```

```
extern __m256 _mm256_acos_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the arc cosine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_acosh_pd, _mm256_acosh_pd

Calculates the inverse hyperbolic cosine value. Vector variant of acosh(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_acosh_pd(__m128d v1);
```

```
extern __m256d _mm256_acosh_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the inverse hyperbolic cosine of vector *v1* values.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_acosh_ps, _mm256_acosh_ps

Calculates the inverse hyperbolic cosine value. Vector variant of acosh(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_acosh_ps(__m128 v1);
```

```
extern __m256 _mm256_acosh_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the inverse hyperbolic cosine of vector *v1* values.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_asin_pd, _mm256_asin_pd

Calculates inverse sine value. Vector variant of asin(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_asin_pd(__m128d v1);
```

```
extern __m256d _mm256_asin_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the arc sine of vector *v1* values.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_asin_ps, _mm256_asin_ps

Calculates inverse sine value. Vector variant of asin(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_asin_ps(__m128 v1);
```

```
extern __m256 _mm256_asin_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the arc sine of vector *v1* values.

Returns

128-bit/256-bit vector with result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_asinh_pd, _mm256_asinh_pd

Calculates the inverse hyperbolic sine value. Vector variant of asinh(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_asinh_pd(__m128d v1);
```

```
extern __m256d _mm256_asinh_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the inverse hyperbolic sine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_asinh_ps, _mm256_asinh_ps

Calculates the inverse hyperbolic sine value. Vector variant of asinh(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_asinh_ps(__m128 v1);
```

```
extern __m256 _mm256_asinh_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the inverse hyperbolic sine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_atan_pd, _mm256_atan_pd

Calculates inverse tangent value. Vector variant of atan(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_atan_pd(__m128d v1);
```

```
extern __m256d _mm256_atan_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the arc tangent of vector *v1* elements. In effect, the tangent of each resulting element value is the value of the corresponding *v1* vector element.

Returns

128-bit/256-bit vector with result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_atan_ps, _mm256_atan_ps

Calculates inverse tangent value. Vector variant of atan(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_atan_ps(__m128 v1);
```

```
extern __m256 _mm256_atan_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the arc tangent of vector *v1* elements. In effect, the tangent of each resulting element value is the value of the corresponding *v1* vector element.

Returns

128-bit/256-bit vector with result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_atan2_pd, _mm256_atan2_pd

Calculates the inverse tangent of float64 variables x and y. Vector variant of atan2(x, y) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_atan2_pd(__m128d v1, __m128 v2);
```

```
extern __m256d _mm256_atan2_pd(__m256d v1, __m256 v2);
```

Arguments

v1	vector with float64 values
v2	vector with float64 values

Description

Calculates the arc tangent of corresponding float64 elements of vectors *v1* and *v2*. The following is an illustration of the atan2 operation:

```
Res[0]  = atan2(v1[0], v2[0])
Res[1]  = atan2(v1[1], v2[1])
Res[2]  = atan2(v1[2], v2[2])
Res[15] = atan2(v1[15], v2[15])
...
```

NOTE
This calculation is similar to calculating the arc tangent of `y / x`, except that the signs of both arguments are used to determine the quadrant of the result.

Returns

Result of the bitwise operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_atan2_ps, _mm256_atan2_ps

Calculates the inverse tangent of float32 variables x and y. Vector variant of atan2(x, y) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_atan2_ps(__m128 v1, __m128 v2);
```

```
extern __m256 _mm256_atan2_ps(__m256 v1, __m256 v2);
```

Arguments

v1	vector with float32 values

v2 vector with float32 values

Description

Calculates the arc tangent of corresponding float32 elements of vectors *v1* and *v2*. The following is an illustration of the atan2 operation:

```
Res[0]  = atan2(v1[0], v2[0])
Res[1]  = atan2(v1[1], v2[1])
Res[2]  = atan2(v1[2], v2[2])
Res[15] = atan2(v1[15], v2[15])
...
```

NOTE
This calculation is similar to calculating the arc tangent of `y / x`, except that the signs of both arguments are used to determine the quadrant of the result.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_atanh_pd, _mm256_atanh_pd

Calculates inverse hyperbolic tangent value. Vector variant of atanh(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_atanh_pd(__m128d v1);
```

```
extern __m256d _mm256_atanh_pd(__m256d v1);
```

Arguments

v1 vector with float64 values

Description

Calculates the inverse hyperbolic tangent of vector *v1* elements. In effect, the hyperbolic tangent of each resulting element value is the value of the corresponding *v1* vector element.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_atanh_ps, _mm256_atanh_ps

Calculates inverse hyperbolic tangent value. Vector variant of atanh(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_atanh_ps(__m128 v1, __m128 v2);
```

```
extern __m256 _mm256_atanh_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the inverse hyperbolic tangent of vector *v1* elements. In effect, the hyperbolic tangent of each resulting element value is the value of the corresponding *v1* vector element.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cos_pd, _mm256_cos_pd

Calculates cosine value. Vector variant of cos(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_cos_pd(__m128d v1);
```

```
extern __m256d _mm256_cos_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the cosine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cos_ps, _mm256_cos_ps

Calculates cosine value. Vector variant of cos(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_cos_ps(__m128 v1);
```

```
extern __m256 _mm256_cos_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the cosine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cosd_pd, _mm256_cosd_pd

Calculates cosine value. Vector variant of cosd(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_cosd_pd(__m128d v1);
```

```
extern __m256d _mm256_cosd_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the cosine of vector *v1* elements, measured in degrees.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cosd_ps, _mm256_cosd_ps

Calculates cosine value. Vector variant of cosd(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_cosd_ps(__m128 v1);
```

```
extern __m256 _mm256_cosd_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the cosine of vector *v1* elements, measured in degrees.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cosh_pd, _mm256_cosh_pd

Calculates the hyperbolic cosine value. Vector variant of cosh(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_cosh_pd(__m128d v1, __m128d v2);
```

```
extern __m256d _mm256_cosh_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the hyperbolic cosine of vector *v1* elements, which is defined mathematically as:

```
(exp(x) + exp(-x)) / 2
```

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_cosh_ps, _mm256_cosh_ps

Calculates the hyperbolic cosine value. Vector variant of cosh(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_cosh_ps(__m128 v1);
```

```
extern __m256 _mm256_cosh_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the hyperbolic cosine of vector *v1* elements, which is defined mathematically as:

```
(exp(x) + exp(-x)) / 2
```

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sin_pd, _mm256_sin_pd

Calculates sine value. Vector variant of sin(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_sin_pd(__m128d v1);
```

```
extern __m256d _mm256_sin_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the sine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sin_ps, _mm256_sin_ps

Calculates sine value. Vector variant of sin(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_sin_ps(__m128 v1);
```

```
extern __m256 _mm256_sin_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the sine of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sind_pd, _mm256_sind_pd

Calculates sine value. Vector variant of sind(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_sind_pd(__m128d v1);
```

```
extern __m256d _mm256_sind_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the sine of vector *v1* elements, measured in degrees.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sind_ps, _mm256_sind_ps

Calculates sine value. Vector variant of sind(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_sind_ps(__m128 v1);
```

```
extern __m256 _mm256_sind_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the sine of vector *v1* elements, measured in degrees.

Returns

128-bit/256-bit vector with the result of the operation.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sinh_pd, _mm256_sinh_pd

Calculates the hyperbolic sine value. Vector variant of sinh(x) function for a 128-bit/256-bit vector argument of float64 values.

Syntax

```
extern __m128d _mm_sinh_pd(__m128d v1);
```

```
extern __m256d _mm256_sinh_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the hyperbolic sine of vector v1 elements, which is defined mathematically as:

$$(\exp^{(x)} - \exp^{(-x)}) / 2$$

.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sinh_ps, _mm256_sinh_ps

Calculates the hyperbolic sine value. Vector variant of sinh(x) function for a 128-bit/256-bit vector argument of float32 values.

Syntax

```
extern __m128 _mm_sinh_ps(__m128 v1);
```

```
extern __m256 _mm256_sinh_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the hyperbolic sine of vector v1 elements, which is defined mathematically as:

$$(\exp^{(x)} - \exp^{(-x)}) / 2$$

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_tan_pd, _mm256_tan_pd

Calculates tangent value. Vector variant of tan(x) function for a 128-bit/256-bit vector arguments with float64 values.

Syntax

```
extern __m128d _mm_tan_pd(__m128d v1);
```

```
extern __m256d _mm256_tan_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the tangent of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_tan_ps, _mm256_tan_ps

Calculates tangent value. Vector variant of tan(x) function for a 128-bit/256-bit vector arguments with float32 values.

Syntax

```
extern __m128 _mm_tan_ps(__m128 v1);
```

```
extern __m256 _mm256_tan_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the tangent of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_tand_pd, _mm256_tand_pd

Calculates tangent value. Vector variant of tand(x) function for a 128-bit/256-bit vector arguments with float64 values.

Syntax

```
extern __m128d _mm_tand_pd(__m128d v1);
```

```
extern __m256d _mm256_tand_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the tangent of vector *v1* elements, measured in degrees.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_tand_ps, _mm256_tand_ps

Calculates the tangent value. Vector variant of tand(x) function for a 128-bit/256-bit vector arguments with float32 values.

Syntax

```
extern __m128 _mm_tand_ps(__m128 v1);
extern __m256 _mm256_tand_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the tangent of vector *v1* elements, measured in degrees.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_tanh_pd, _mm256_tanh_pd

Calculates hyperbolic tangent value. Vector variant of tanh(x) function for a 128-bit/256-bit vector arguments with float64 values.

Syntax

```
extern __m128d _mm_tanh_pd(__m128d v1);
extern __m256d _mm256_tanh_pd(__m256d v1);
```

Arguments

v1	vector with float64 values

Description

Calculates the hyperbolic tangent of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_tanh_ps, _mm256_tanh_ps

Calculates hyperbolic tangent value. Vector variant of tanh(x) function for a 128-bit/256-bit vector arguments with float32 values.

Syntax

```
extern __m128 _mm_tanh_ps(__m128 v1);
```

```
extern __m256 _mm256_tanh_ps(__m256 v1);
```

Arguments

v1	vector with float32 values

Description

Calculates the hyperbolic tangent of vector *v1* elements.

Returns

128-bit/256-bit vector with the result of the operation.

NOTE
Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sincos_pd, _mm256_sincos_pd

Calculates the sine and cosine values. Vector variant of sincos(x, &sin_x, &cos_x) function for a 128-bit/256-bit vector with float64 values.

Syntax

```
extern __m128d _mm_sincos_pd(__m128d *p_cos, __m128d v1);
```

```
extern __m256d _mm256_sincos_pd(__m256d *p_cos, __m256d v1);
```

Arguments

**p_cos*	points to vector of cosine results (pointer must be aligned on 16 bytes, or declared as `__m128d*` instead)
v1	vector with float64 values

Description

Calculates sine and cosine values of vector *v1* elements.

The cosine and sine values cannot be returned in the result vector. Therefore, the intrinsic stores the cosine values at a location pointed to by*p_cos*, and returns only the sine values in the 128-bit result vector.

Returns

128-bit/256-bit vector with the sine results.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

_mm_sincos_ps, _mm256_sincos_ps

Calculates the sine and cosine values. Vector variant of sincos(x, &sin_x, &cos_x) function for a 128-bit/256-bit vector with float32 values.

Syntax

```
extern __m128 _mm_sincos_ps(__m128 *p_cos, __m128 v1);
```

```
extern __m256 _mm256_sincos_ps(__m256 *p_cos, __m256 v1);
```

Arguments

**p_cos*	points to vector of cosine results (pointer must be aligned on 16 bytes, or declared as `__m128*` instead)
v2	vector with float32 values

Description

Calculates sine and cosine values of vector *v1* elements.

The cosine and sine values cannot be returned in the result vector. Therefore, the intrinsic stores the cosine values at a location pointed to by*p_cos*, and returns only the sine values in the 128-bit result vector.

Returns

128-bit/256-bit vector with the sine results.

> **NOTE**
> Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Libraries

The Intel® C++ Compiler lets you use all the standard run-time libraries that are part of Microsoft* Visual C++*. The options described in this section can help you determine which libraries your application uses.

To create libraries, use the `lib.exe` tool or `xilib.exe` tool.

Create Libraries

Libraries are simply an indexed collection of object files that are included as needed in a linked program. Combining object files into a library makes it easy to distribute your code without disclosing the source. It also reduces the number of command-line entries needed to compile your project.

Static Libraries

Executables generated using static libraries are no different than executables generated from individual source or object files. Static libraries are not required at runtime, so you do not need to include them when you distribute your executable. At compile time, linking to a static library is generally faster than linking to individual source files.

These steps show how to build a static library on Linux using the `icpc` driver . See Invoke the Compiler for information about all available drivers.

1. Use the `c` option to generate object files from the source files:

```
icpc -c my_source1.cpp my_source2.cpp my_source3.cpp
```

2. Use the GNU* tool `ar` to create the library file from the object files:

```
ar rc my_lib.a my_source1.o my_source2.o my_source3.o
```

3. Compile and link your project with your new library:

```
icpc main.cpp my_lib.a
```

If your library file and source files are in different directories, use the `L`*`dir`* option to indicate where your library is located:

```
icpc -L/cpp/libs main.cpp my_lib.a
```

To build a static library on macOS:

1. Use the following command to generate object files and create the library file:

```
icpc -fpic -o mylib.a -staticlib my_source1.cpp my_source2.cpp my_source3.cpp
```

2. Compile and link your project with your new library:

```
icpc main.cpp my_lib.a
```

If your library file and source files are in different directories, use the `L`*`dir`**dir* option to indicate where your library is located:

```
icpc -L/cpp/libs main.cpp my_lib.a
```

If you are using Interprocedural Optimization, see the topic Create a Library from IPO Objects, which discusses using `xiar`.

Shared Libraries

Shared libraries, also referred to as dynamic libraries or Dynamic Shared Objects (DSO), are linked differently than static libraries. At compile time, the linker insures that all the necessary symbols are either linked into the executable, or can be linked at runtime from the shared library. Executables compiled from shared libraries are smaller, but the shared libraries must be included with the executable to function correctly. When multiple programs use the same shared library, only one copy of the library is required in memory.

Linux

These steps show how to build a shared library on Linux using the `icpc` driver . See Invoke the Compiler for information about all available drivers.

1. Use options `fPIC` and `c` to generate object files from the source files:

```
icpc -fPIC -c my_source1.cpp my_source2.cpp my_source3.cpp
```

2. Use the `shared` option to create the library file from the object files:

```
icpc -shared -o my_lib.so my_source1.o my_source2.o my_source3.o
```

3. Compile and link your project with your new library:

```
icpc main.cpp my_lib.so
```

macOS

These steps show how to build a shared library on macOS using the `icpc` driver

1. Use the following command to generate object files and create the library file:

```
icpc -fPIC -o my_lib.so -dynamiclib my_source1.cpp my_source2.cpp my_source3.cpp
```

2. Compile and link your project with your new library:

```
icpc main.cpp my_lib.dylib
```

Windows

Use the following options to create libraries on Windows:

Option	Description
`/LD`, `/LDd`	Produces a DLL. `d` indicates debug version.
`/MD`, `/MDd`	Compiles and links with the dynamic, multi-thread C run time library. `d` indicates debug version.
`/MT`, `/MTd`	Compiles and links with the static, multi-thread C run time library. `d` indicates debug version.
`/Zl`	Disables embedding default libraries in object files.

See Also

Use Intel Shared Libraries

Create a Library from IPO Objects

See Also

`/LD` compiler option
`/MD` compiler option
`/MT` compiler option

Use Intel Shared Libraries

This topic applies to Linux and macOS.

By default, the Intel® C++ Compiler Classic links Intel® C++ libraries dynamically. The GNU/Linux/macOS system libraries are also linked dynamically.

Options for Shared Libraries (Linux)

Option	Description
`-shared-intel`	Use the `shared-intel` option to link Intel®-provided libraries dynamically. This has the advantage of reducing the size of the application binary, but it also requires the libraries to be on the application's target system.
`-shared`	The `shared` option instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable. For more details, refer to the `ld` man page documentation.
`-fpic`	Use the `fpic` option when building shared libraries. It is required for the compilation of each object file included in the shared library.

Options for Shared Libraries (macOS)

Option	Description
`-dynamiclib`	Use the `dynamiclib` option to invoke the `libtool` command to generate dynamic libraries.
`-fpic`	Use the `fpic` option when building shared libraries. It is required for the compilation of each object file included in the shared library.

Using Shared Libraries on macOS

This topic only applies to macOS.

On macOS, it is possible to store path information in shared libraries to perform library searches. The compiler installation changes the path to the installation directory, but you will need to modify these paths if you move the libraries elsewhere. For example, you may want to bundle redistributable Intel® libraries with your application. This eliminates the dependency on libraries found on `DYLD_LIBRARY_PATH`.

If your compilations do not use `DYLD_LIBRARY_PATH` to find libraries, and you distribute executables that depend on shared libraries, then you will need to modify the Intel shared libraries (`../lib/*.dylib`) using the `install_name_tool` to set the correct path to the shared libraries. This also permits the end-user to launch the application by double-clicking on the executable. The command below will modify each library with the correct absolute path information:

```
for i in *.dylib
do
    echo -change $i `pwd`/$i
    done > changes
for i in *.dylib
do
    install_name_tool `cat changes` $i
done
```

You can also use the `install_name_tool` to set `@executable_path` to change path information for the libraries bundled in your application by changing the path appropriately.

Be sure to recompile your sources after modifying the libraries.

Manage Libraries

Manage Libraries on Linux and macOS

During compilation, the compiler reads the `LIBRARY_PATH` environment variable for static libraries it needs to link when building the executable. At runtime, the executable will link against dynamic libraries referenced in the `LD_LIBRARY_PATH` environment variable. Add the location of your static libraries to the `LIBRARY_PATH` environment variable so that they are available for linking during compilation.

For example, to compile `file.cpp` and link it with the library `lib.a`, located in the `/libs` directory:

1. Add the directory `/libs` to `LIBRARY_PATH` from the command line with the export command:

```
export LIBRARY_PATH=/libs:$LIBRARY_PATH
```

   Alternately, add the directory to `LIBRARY_PATH` by addiing the export command to your startup file.
2. Compile `file.cpp` and link it with `lib.a`:

```
icc file.cpp lib.a
```

To link your library during compilation without modifying the `LIBRARY_PATH` environment variable use the `-L` option. For example:

```
icc file.cpp -L /libs lib.a
```

During compilation, the compiler passes object files to the linker in the following order:

1. Object files, from files specified on the command line, in the order they are specified (left to right)
2. Objects or libraries specified in default configuration files
3. Default Intel and system libraries

For example, the command

```
icc lib1.a file.cpp lib2.a
```

would have the following link order:

1. `lib1.a`
2. `file.o`
3. `lib2.a`
4. Objects or libraries specified in default configuration files
5. Default Inel and system libraries

Manage Libraries on Windows

The `LIB` environment variable contains a semicolon-separated list of directories in which the Microsoft linker will search for library (`.lib`) files. The compiler does not specify library names to the linker but includes directives in the object file to specify the libraries to be linked with each object.

For more information on adding library names to the response file and the configuration file, see Using Response Files and Using Configuration Files.

To specify a library name on the command line, you must first add the library's path to the `LIB` environment variable. Then you can specify the library name on the command line. For example, to compile `file.cpp` and link it with the library `mylib.lib` enter the command:

```
icc file.cpp mylib.lib
```

Other Considerations

The Intel Compiler Math Libraries contain performance-optimized implementations for various Intel platforms. By default, the best implementation for the underlying hardware is selected at runtime. The library dispatch of multi-threaded code may lead to apparent data races, which may be detected by certain software analysis tools. However, as long as the threads are running on cores with the same CPUID, these data races are harmless and are not a cause for concern.

Redistribute Libraries When Deploying Applications

When you deploy your application to systems that do not have a compiler installed, you need to redistribute certain Intel® libraries where your application is linked. You can do so in one of the following ways:

- Statically link your application.

  An application built with statically-linked libraries eliminates the need to distribute runtime libraries with the application executable. By linking the application to the static libraries, you are not dependent on the Intel® Fortran or Intel® C/C++ dynamic shared libraries.
- Dynamically link your application.

  If you must build your application with dynamically linked (or shared) compiler libraries, you should address the following concerns:

  - You must build your application with shared or dynamic libraries that are redistributable.
  - Pay careful attention to the directory where the redistributables are installed and how the OS finds them.
  - You should determine which shared or dynamic libraries your application needs.

The information here is only introductory. The redistributable library installation packages are available at the following locations:

- Intel® oneAPI versions
- Older Intel® Parallel Studio XE versions

Resolve References to Shared Libraries

If you are relying on shared libraries distributed with Intel® oneAPI tools, you must make sure that your users have these shared libraries on their systems.

If you are building an application that will be deployed to your user community and you are relying on shared libraries (`.so` shared objects on Linux, `.dll` dynamic libraries on Windows, and `.dylib` dynamic libraries on macOS) distributed with Intel® oneAPI tools, you must make sure that your users have these shared libraries on their systems. To determine what shared libraries you depend on, use one of the following commands for each of your programs and components:

Linux

```
ldconfig
```

macOS

```
otool -L
```

Windows

```
dumpbin /DEPENDENTS programOrComponentName
```

Once you have done this, you must choose how your users will receive these libraries.

Shared Library Deployment

Once you have built, run, and debugged your application, you must deploy it to your users. That deployment includes any shared libraries, including libraries that are components of the Intel® oneAPI toolkits.

Deployment Models

You have two options for deploying the shared libraries from the Intel oneAPI toolkit that your application depends on:

Private Model	Copy the shared libraries from the Intel oneAPI toolkit into your application environment, and then package and deploy them with your application. Review the license and third-party files associated with the Intel oneAPI toolkits and/or components you have installed to determine the files that you can redistribute. The advantage to this model is that you have control over your library and version choice, so you only package and deploy the libraries that you have tested. The disadvantage is that the end users may see multiple libraries installed on their system, if multiple installed applications all use the private model. You are also responsible for updating these libraries whenever updates are required.
Public Model	You direct your users to runtime packages provided by Intel. Your users install these packages on their system when they install your application. The runtime packages install onto a fixed location, so all applications built with Intel oneAPI tools can be used. The advantage is that one copy of each library is shared by all applications, which results in improved performance. You can rely on updates to the runtime packages to resolve issues with libraries independently from when you update your application. The disadvantage is that the footprint of the runtime package is larger than a package from the private model. Another disadvantage is that your tested versions of the runtime libraries may not be the same as your end user's versions.

Select the model that best fits your environment, your needs, and the needs of your users.

NOTE Intel ensures that newer compiler-support libraries work with older versions of generated compiler objects, but newer versioned objects require newer versioned compiler-support libraries. If an incompatibility is introduced that causes newer compiler-support libraries not to work with older compilers, you will have sufficient warning and the library will be versioned so that deployed applications continue to work.

Additional Steps

Under either model, you must manually configure certain environment variables that are normally handled by the `setvars`/`vars` scripts or module files.

For example, with the Intel® MPI Library, you must set the following environment variables during installation:

Linux

```
I_MPI_ROOT=installPath FI_PROVIDER_PATH=installPath/intel64/libfabric:/usr/lib64/libfabric
```

Windows

```
I_MPI_ROOT=installPath
```

Compatibility in the Minor Releases of the Intel oneAPI Products

For Intel oneAPI products, each minor version of the product is compatible with the other minor version from the same release (for example, 2021). When there are breaking changes in API or ABI, the major version is increased. For example, if you tested your application with an Intel oneAPI product with a 2021.1 version, it will work with all 2021.x versions. It is not guaranteed that it will work with 2022.x or 19.x versions.

Intel's Memory Allocator Library

Intel's `libqkmalloc` library for fast memory allocation provides a C-level interface for memory allocation that is optimized for performance.

You can link the `libqkmalloc` library as a shared library only on Linux* platforms for Intel® 64 architecture. This library provides optimized implementation of standard allocation routines `malloc`, `calloc`, `realloc`, and `free`, and is C99 standard compliant.

> **NOTE** This library is limited to work only on Intel® processors and will redirect to standard C routines at runtime if used on non-Intel® processors.

Use Intel's Custom Memory Allocator Library

You can use the libqkmalloc library by linking directly to it or by using the `LD_PRELOAD` environment variable.

To ensure the application will override the standard library allocation routines with `libqkmalloc`, set the environment variable LD_PRELOAD in the command line before the application execution. This environment variable allows you to set the path of the library that will be loaded before any other library (including the C runtime library), and the application will use symbols from this specified library instead of the symbols from the standard library.

Restrictions

This library does not support threaded code such as OpenMP* and is not thread-safe. It should not be used simultaneously from multiple threads. For the best results this library should be used with large throughput workloads.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

SIMD Data Layout Templates

SIMD Data Layout Templates (SDLT) is a C++11 template library providing containers that represent arrays of "Plain Old Data" objects (a struct whose data members do not have any pointers/references and no virtual functions) using layouts that enable generation of efficient SIMD (single instruction multiple data) vector code. SDLT uses standard ISO C++11 code. It does not require a special language or compiler to be functional, but takes advantage of performance features (such as OpenMP* SIMD extensions and `pragma ivdep`) that may not be available to all compilers. It is designed to promote scalable SIMD vector programming. To use the library, specify SIMD loops and data layouts using explicit vector programming model and SDLT containers, and let the compiler generate efficient SIMD code in an efficient manner.

Many of the library interfaces employ generic programming, in which interfaces are defined by requirements on types and not specific types. The C++ Standard Template Library (STL) is an example of generic programming. Generic programming enables SDLT to be flexible yet efficient. The generic interfaces enable you to customize components to your specific needs.

The net result is that SDLT enables you to specify a preferred SIMD data layout far more conveniently than re-structuring your code completely with a new data structure for effective vectorization, and at the same time can improve performance.

Motivation

C++ programs often represent an algorithm in terms of high level objects. For many algorithms there is a set of data that the algorithm will need to process. It is common for the data set to be represented as array of "plain old data" objects. It is also common for developers to represent that array with a container from the C ++ Standard Template Library, like std::vector. For example:

```
struct Point3s
{
    float x;
    float y;
    float z;
       // helper methods
};

std::vector<Point3s> inputDataSet(count);
std::vector<Point3s> outputDataSet(count);

for(int i=0; i < count; ++i) {
  Point3s inputElement = inputDataSet[i];
  Point3s result = // transformation of inputElement that is independent of other iterations
                   // can keep algorithm high level using object helper methods
  outputDataSet[i] = result;
}
```

When possible a compiler may attempt to vectorize the loop above, however the overhead of loading the "Array of Structures" data set into vector registers may overcome any performance gain of vectorizing. Programs exhibiting the scenario above could be good candidates to use a SDLT container with a SIMD-friendly internal memory layout. SDLT containers provide *accessor* objects to import and export Primitives between the underlying memory layout and the objects original representation. For example:

```
SDLT_PRIMITIVE(Point3s, x, y, z)

sdlt::soa1d_container<Point3s> inputDataSet(count);
sdlt::soa1d_container<Point3s> outputDataSet(count);

auto inputData = inputDataSet.const_access();
auto outputData = outputDataSet.access();

#pragma forceinline recursive
#pragma omp simd
for(int i=0; i < count; ++i) {
  Point3s inputElement = inputData[i];
  Point3s result = // transformation of inputElement that is independent of other iterations
                   // can keep algorithm high level using object helper methods
  outputData[i] = result;
}
```

When a local variable inside the loop is imported from or exported to using that loop's index, the compiler's vectorizor can now access the underlying SIMD friendly data format and when possible perform unit stride loads. If the compiler can prove nothing outside the loop can access the loop's local object, then it can optimize its private representation of the loop object be "Structure of Arrays" (SOA). In our example, the container's underlying memory layout is also SOA and unit stride loads can be generated. The Container also allocates aligned memory and its accessor objects provide the compiler with the correct alignment information for it to optimize code generation accordingly.

Version Information

This documentation is for SDLT version 2, which extends version 1 by introducing support for n-dimensional containers.

Backwards Compatibility

Public interfaces of version 2 are fully backward compatible with interfaces of version 1.

The backwards compatibility includes:

- Existing source code compatibility.
  - Any source code using the SDLT v1 public API (non-internal interfaces) can be recompiled against SDLT v2 headers with no changes.
- Binary compatibility.
  - Because SDLT v2 API's exist in a new name space, sdlt::v2, all ABI linkage should not collide with any existing SDLT v1 ABI's that exist only in sdlt namespace.
  - A binary, dynamically-linked library that uses SDLT v1 internally, can be linked into a program using SDLT v2, and vice versa.
- Passing SDLT containers or accessors as part of a libraries public API (ABI). When SDLT is used as part of an ABI, that library and the calling code must use the same version of SDLT. They cannot be mixed or matched.

This compatibility doesn't cover internal implementation. Internal implementation for SDLT v1 was updated and unified with parts introduced in v2, so for codes dependent on internal interfaces backwards compatibility is not guaranteed.

Deprecated

The interfaces below are deprecated; use the replacements provided in the table.

Deprecated Interface	Deprecated in Version	Replaced By
`sdlt::fixed_offset<>`	v2	`sdlt::fixed<>`
`sdlt::aligned_offset<>`	v2	`sdlt::aligned<>`

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Usage Guidelines: Function Calls and Containers

Function Calls

Function calls are a commonly used programming construct. Follow these simple guidelines when using SDLT containers:

- If an SDLT Primitive is passed to a function by value, by pointer, or by reference, be sure to inline them
- Any Non-inlined functions should be SIMD enabled (for example, marke them with #pragma omp declare simd).

If a loop variable is passed to a non-inlined function, the current C++ Application Binary Interface (ABI) requires the memory layout match object's original which could cause additional data transformations or inhibit vectorization. For that reason, the SDLT approach works best when all the methods or functions called are inlined or use `#pragma omp declare simd`. Marking a function "inline" explicitly or implicitly is only a hint. Compilers have several limits and heuristics that could cause a function to not be inlined. To avoid this issue, we recommend utilizing the `#pragma forceinline recursive` which instructs the compiler to ignore its limits and heuristics: causing all functions in the following code block that could be inlined to actually be inlined together with any functions called, and functions they call, and so on. Please also note that this can cause the loop body and/or the function body to become too big to optimize. Under such circumstances, carefully examine and restructure the function call boundaries and consider applying non-inlined, SIMD-enabled function calls.

1-Dimensional Containers Overview

What if that std::vector<typename> could store data SIMD-friendly format internally while exposing an AOS view to the programmer?

The 1-dimensional containers in SDLT aim to achieve that goal. They can abstract the in-memory data layout of an array of objects to:

1. AOS (Array of Structures)
2. SOA (Structure of Arrays) which is SIMD friendly

Import/Export Only

As the memory layout is abstracted and may not match the original structure's layout, containers cannot provide memory references to the underlying data. Only import or export of the object to and from a particular element in the container. In use, an algorithm might require some minor code changes to follow import/export paradigm, however algorithm itself should read/flow the same.

The 1D containers in SDLT are dynamically resizable with an interface similar to std::vector<T>. To avoid accidental misuse of copying containers into C++11 lambda functions we chose to delete the container's copy constructor and instead provide explicit "clone" method instead.

Containers provide SDLT concepts of an accessor and const_accessor for use with SIMD loops, interfaces for `std::vector` compatibility are intended for ease of integration, not high performance.

Just like `std::vector`, the containers own the array data and its scope controls the life of that data.

n-Dimensional Containers Overview

Multi-dimensional containers generalize ideas from 1-dimensional containers; they separate multi-dimensional access semantics from storage logic in an abstract way. A multi-dimensional SDLT container is a generic container that handles an arbitrary number of dimensions, and at the same time internally represents data as needed. Unlike 1-dimensional containers, multi-dimensional containers are not resizable and don't have interfaces like that of `std::vector`. While 1-dimensional containers are like `std::vectors` with decoupled storage, multi-dimensional containers are more akin to arrays (statically sized or variable length).

Below is an example of an n-dimensional container parameterized by three concerns: the data item (primitive) type, the storage layout in memory, and the observed shape of the container.

```
n_container<PrimitiveT, LayoutT, ExtentsT>
```

Template Arguments	Description
`typename PrimitiveT`	The type of primitive that will be contained.
`typename LayoutT`	The type of data layout.
`typename ExtentsT`	Specifies the dimensions of the container

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Construct an n_container

Description

An N-dimensional (multi-dimensional) container must be constructed before it can be used. The data type to be contained must first be declared as a `SDLT_PRIMITIVE`, then a data layout is chosen, and finally the shape of the container is determined describing the extents of each dimension.

Specify Data Layout

Rather than defining different containers for different data layouts, the data layout to use is specified as a template parameter to the container.

Available layouts are summarized in table below. Full details can found on the table in the topic n_container.

Layout	Description
`layout::soa<>`	Structure of Arrays (SOA). Each data member of the Primitive will have its own N-dimensional array.
`layout::soa_per_row<>`	Structure of Arrays Per Row. Each data member of the Primitive will have its own 1-dimensional array per row. Layout repeats for remaining N-1 dimensions.
`layout::aos_by_struct`	Array of Structures (AOS) Accessed by Struct. Native AOS layout and data access.
`layout::aos_by_stride`	Array of Structures Accessed by Stride. Native SOA data access through pointers to the built in types of members using a stride to account for the size of the Primitive.

Numbers and Constants

In order to define shape, integer values can be provided in three different forms, each successively providing less information to compiler. It is advised to use as precise specification as possible. The compiler may optimize better with more information.

Integer Value Specification	Description

`fixed<int NumberT>`	Known at compile time.
	`foo(fixed<1080>(), fixed<1920>());`
	The suffix `_fixed` will declare an equivalent literal. For example, `(1080_fixed` is equivalent to `fixed<1080>`.
	`foo(1080_fixed, 1920_fixed); )`

`aligned<int AlignmentT>(number)`	Programmer guarantees the number is a multiple of the `AlignmentT`. `foo(aligned<8>(height), aligned<128>(width));`
`“int”`	Arbitrary integer value. `foo(width, height);`

Specify Container Shape

`n_extent_t<…>` is a variadic template that accepts any number of arguments defining dimensions. Because construction using this type may look unclear, a generator object, `n_extent` , is provided to construct extents for all dimensions using a familiar array-definition-like syntax. Extent values may be specified using the most precise representation possible, as described above, to allow the compiler to better prove any potential data alignments.

```
n_extent[height][width];                  // OK
n_extent[height][aligned<128>(width)];  // Better
n_extent[1080_fixed][1920_fixed];       // Best
```

Define an n_container

Using a previously declared primitive (same as SDLT v1),

```
struct RGBAs { float red, green, blue, alpha; };
SDLT_PRIMITIVE(RGBAs, red, green, blue, alpha)
```

A two-dimensional container of RGBAs with HD image size 1920x0180 can be declared and instantiated as in the below example.

```
typedef n_container<RGBAs, layout::soa,
                    n_extent_t<fixed<1080>, fixed<1920>>> HdImage;
HdImage image1;
```

If sizes are not known, a container may be defined with extents unknown to the compiler but known at run-time when an instance of the container is created.

```
typedef n_container<RGBAs, layout::soa, n_extent_t<int, int>> Image;
Image image2(n_extent[height][width]);
```

Additionally, the templated factory function make_n_container<PrimitiveT, LayoutT> may be used to create containers.

```
auto image1 = make_n_container<RGBAs,
                          layout::soa>(n_extent[1080_fixed][1920_fixed]);
auto image2 = make_n_container<RGBAs,
                           layout::soa>(n_extent[height][width]);
```

Access Cells

Containers own data. To get to the data inside, use an "accessor."

```
auto ca = image1.const_access();
auto a = image2.access();
```

Specify the index for each dimension with a series of calls to the array subscript operator [], similar to a multi-dimensional array in C.

```
RGBAs pixel = ca[y][x];
float greyscale = (pixel.red + pixel.green + pixel.blue)/3;
a[y][x] = RGBAs(greyscale, greyscale, greyscale);
```

Discover Extents

Accessors know their extents.

Use template function `extent_d<int DimensionT>(object).`

```
for (int y = 0; y < extent_d<0>(ca); ++y)
    for (int x = 0; x < extent_d<1>(ca); ++x) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

For convenience, non-template methods are also provided.

```
for (int y = 0; y < ca.extent_d0(); ++y)
    for (int x = 0; x < ca.extent_d1(); ++x) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

Lower Dimensions

The result of not specifying all the dimensions required by an accessor is a new accessor with a lower rank that can then be accessed.

```
auto cay = ca[y];
RGBAs pixel = cay[x];
```

Bounds

Description

`bounds_t<LowerT, UpperT>` holds the lower and upper bounds of a half-open interval. It is templated to allow the different integer representations for the lower and upper bounds. The intent is to model a valid iteration space over a single dimension.

Bounds can be used to iterate over an entire extent or to restrict iteration space within an extent

Creating Bounds

Bounds can be created using full bounds_t type, but this may be tedious.

```
bounds_t<int, int>(start, finish)
bounds_t<int, aligned<16>>(start, aligned<16>(finish))
bounds_t<fixed<0>, fixed<1920>>()
```

It is simpler and clearer to use factory function bounds to build a bounds_t<>.

```
bounds(start,finish);
bounds(start, aligned<16>(finish));
bounds(0_fixed, 1920_fixed)
```

Discovering Bounds

Accessors know their valid iteraton space. Initial bounds for an accessor are set to set the lower bound to be fixed<0> and the upper bound set to the value and type of the dimension's extent as specified during construction of the n_container(fixed<>,aligned<>, or int).

To query bounds for given dimension of the accessor use template function `bounds_d<int DimensionT>(object)`.

```
auto b0 = bounds_d<0>(ca);
auto b1 = bounds_d<1>(ca);
for (int y = b0.lower(); y < b0.upper(); ++y)
    for (int x = b1.lower(); x < b1.upper(); ++x) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

`bounds_t` can participate in C++11 range-based for loops.

```
for (auto y: bounds_d<0>(ca))
    for (auto x: bounds_d<1>(ca)) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

```
for (auto y: ca.bounds_d0())
    for (auto x: ca.bounds_d1()) {
          RGBAs pixel = ca[y][x];
         // …
    }
```

N-Dimensional Indexes and Bounds

To model index and bounds values over multiple dimensions, respectively the following template classes are provided: `n_index_t<…>` and `n_bounds_t<…>` . These are both variadic templates, accepting any number of arguments.

`n_index` is a generator to simplify creating instances of `n_index_t`.

```
n_index[540][960]
```

`n_bounds` is a generator to simplify creating instances of `n_bounds_t`.

```
n_bounds[bounds(540,1080)][bounds(960,1920)]
```

Alternatively, `n_bounds_t` can be defined in terms of a `n_index_t` and `n_extent_t`.

```
n_bounds(n_index[540][960], n_extent[540][960]);
```

Accessing Subsections

From a container's accessors, a new accessor can be created over a subsection defined by a `n_bounds_t`.

```
auto ca = c.const_access();
auto subsect = ca.section(n_bounds[bounds(540, 1080)][bounds(960,1920)]);
```

The effect is to restrict the results of `bounds_d<int Dimension>` on the subsection accessor.

You can create a new accessor translated to a different index space.

```
auto offsetnewSpace = ca.translated_to(n_index[1000][2000]);
auto zeroSpace = ca.translated_to_zero();
```

Accesses will have a translation applied that maps the `n_index` back to the lower bounds of the accessor that created it. This allows a smaller container to be reused in a larger index space that is being walked over by blocks, or to move a subsection index space back to the origin.

User-Level Interface

This section describes the user-level interface for the SIMD Data Layout Templates (SDLT). This API is defined in `sdlt.h` and its associated header files.

SDLT Primitives (SDLT_PRIMITIVE)

Primitives represent the data we want to work over in SIMD. They can be more than just data structures. As a C++ object, it can have its own methods that modify its data.

Rules:

- Must be Plain Old Data (POD)
  - Has trivial copy constructor

  - Has trivial move constructor
  - Has trivial destructor
  - No virtual functions or virtual bases
- No reference data members
- No unions
- No bit fields
- No bool types
  - Comparison semantics not efficient in SIMD
  - Use 32-bit integer and compare against known values like 0 or 1 explicitly
- Data members need to be public or declare `SDLT_PRIMITIVE_FRIEND` in the object's definition

Current limitations:

- No pointer data members
- No C++11 strongly typed enums—use integers instead.
- No array based data members.
- copy constructor and assignment operator (=) defined by individual member assignment—strongly encouraged to facilitate better code generation

They may seem like large restrictions, but often code can easily be re-factored to meet this requirement. For example:

```
class Point3d {
    // methods...
protected:
    double v[3];
};
```

can be re-factored to have a public data member for each element in the array and update methods to use the `x`, `y`, and `z` data members rather than the array `v`.

```
class Point3d {
public:
    // methods...
    double x;
    double y;
    double z;
};
```

For better code generation, explicitly define a copy constructor and assignment operator (=) by individual member assignment.

`SDLT_PRIMITIVE` Macro

Once an object meets the criteria above, we can consider it a Primitive type in SDLT. In order for Container's to import and export the Primitive, it has to understand its data layout. Unfortunately C++11 lacks compile time reflection, so the user must provide SDLT with a description of your structure's data layout. This is easily done with the SDLT_PRIMITIVE helper macro that accepts a struct type followed by a comma separated list of its data members.

```
SDLT_PRIMITIVE(STRUCT_NAME, DATA_MEMBER_1, ...)
```

Example Usage:

```
struct UserObject
{
    float x;
    float y;
    double acceleration;
    int behavior;
};

SDLT_PRIMITIVE(UserObject, x, y, acceleration, behavior)
```

An object must be declared as a Primitive before it can be used in a Container. However, built-in types like float, double, int, etc. do not need to be declared as a Primitive before use with a Container. Built-in's are automatically considered Primitives by SDLT.

Nested Primitives are supported, but the nested Primitive must be declared before the outer Primitive is. Example: Axis Aligned Bounding Box made up of two 3d points

```
struct Point3s
{
    float x;
    float y;
    float z;
};

struct AABB
{
    Point3s topLeft;
    Point3s bottomRight;
};

SDLT_PRIMITIVE(Point3s, x, y, z)
SDLT_PRIMITIVE(AABB, topLeft, bottomRight)
```

Notice the `struct` definitions themselves do not derive from SDLT or use any of its nomenclature. This independence allows classes to be used in code not using SDLT and only code that does use SDLT Containers needs to see the Primitive declarations.

soa1d_container

Template class for "Structure of Arrays" memory layout of a one-dimensional container of Primitives.
`#include <sdlt/soa1d_container.h>`

Syntax

```
template<typename PrimitiveT,
         int AlignD1OnIndexT = 0,
         class AllocatorT = allocator::default_alloc>
class soa1d_container;
```

Arguments

`typename PrimitiveT`	The type that each element in the array will store
`int AlignD1OnIndexT = 0`	[Optional] The index on which the data access will be aligned (useful for stencils)
`class AllocatorT = allocator::default_alloc`	[Optional] Specify type of allocator to be used. `allocator::default_alloc` is currently the only allocator supported.

Description

Dynamically sized container of Primitive elements with memory layout as a Structure of Arrays internally providing:

- Dynamic resizing with interface similar to `std::vector`
- Accessor objects suitable for efficient data access inside SIMD loops

Member	Description
`typedef size_t size_type;`	Type to use when specifying sizes to methods of the container.
`template <typename OffsetT = no_offset>` `using accessor;`	Template alias to an accessor for this container
`template <typename OffsetT = no_offset >` `using const_accessor;`	Template alias to an const_accessor for this container

Member Type	Description

`soa1d_container(`
`    size_type size_d1 = 0u,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator =`
`allocator_type());`
	Constructs an uninitialized container of size_d1 elements, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`soa1d_container(`
`    size_type size_d1,`
`    const PrimitiveT &a_value,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container of size_d1 elements initializing each with a_value, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`template<typename StlAllocatorT>`
`soa1d_container(`
`    const std::vector<PrimitiveT,`
`StlAllocatorT> &other,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container with a copy of each of the elements in other, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`soa1d_container(`
`    const PrimitiveT *other_array,`
`    size_type number_of_elements,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container with a copy of number_of_elements elements from the array other_array, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`template< typename IteratorT >`
`soa1d_container(`
`    IteratorT a_begin,`
`    IteratorT an_end,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container with as many elements as the range [a_begin - an_end), each with a copy of the value from its corresponding element in that range, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`soa1d_container clone() const;`	Returns: a new soa1d_container instance with its own copy of the elements

Member Type	Description
`void resize(size_type new_size_d1);`	Resize the container so that it contains *new_size_d1* elements. If the new size is greater than the current container size, the new elements are unitialized.
`accessor<> access();`	Returns: accessor with no embedded index offset.
`accessor<int> access(int offset);`	Returns: accessor with an integer based embedded index offset.
`template<int IndexAlignmentT> accessor<aligned_offset<IndexAlignmentT> > access(aligned_offset<IndexAlignmentT>);`	Returns: accessor with an aligned_offset<IndexAlignmentT> based embedded index offset.
`template<int OffsetT> accessor<fixed_offset<OffsetT> > access(fixed_offset<OffsetT>);`	Returns: accessor with a fixed_offset<OffsetT> based embedded index offset.
`const_accessor<> const_access() const;`	Returns: const_accessor with no embedded index offset.
`const_accessor<int> const_access(int offset) const;`	Returns: const_accessor with an integer based embedded index offset.
`const_accessor<aligned_offset<IndexAlignmentT> > const_access(aligned_offset<IndexAlignmentT> offset) const;`	Returns: const_accessor with an aligned_offset<IndexAlignmentT> based embedded index offset.
`template<int OffsetT> const_accessor<fixed_offset<OffsetT> > const_access(fixed_offset<OffsetT>) const;`	Returns: const_accessor with a fixed_offset<OffsetT> based embedded index offset.

STL Compatibility

In addition to the performance oriented interface explained in the table above, `soa1d_container` implements a subset of the `std::vector` interface that is intended for ease of integration, not high performance. Due to the import/export only requirement we can't return a reference to the object, instead `iterators` and `operator[]` return a Proxy object while other "const' methods return a "value_type const". Futhermore, iterators do not support the `->` operator. Despite that limitation the iterators can be passed to any STL algorithm. Also for performance reasons, resize does not initialize new elements. The following `std::vector` interface methods are implemented:

- size, max_size, capacity, empty, reserve, shrink_to_fit
- assign, push_back, pop_back, clear, insert, emplace, erase
- cbegin, cend, begin, end, begin, end, crbegin, crend, rbegin, rend, rbegin, rend
- operator[], front() const, back() const, at() const
- swap, ==, !=
- swap, soa1d_container(soa1d_container&& donor), soa1d_container & operator=(soa1d_container&& donor)

aos1d_container

Template class for "Array of Structures" memory layout of a one-dimensional container of Primitives.

#include <sdlt/aos1d_container.h>

Syntax

```
template<
    typename PrimitiveT,
    AccessBy AccessByT,
    class AllocatorT = allocator::default_alloc
>
class aos1d_container;
```

Arguments

`typename PrimitiveT`	The type that each element in the array will store
`access_by AccessByT`	Enum to control how the memory layout will be accessed. Recommend access_by_struct unless you are having issues vectorizing. See the documentation of `access_by` for more details
`class AllocatorT = allocator::default_alloc`	[Optional] Specify the type of allocator to be used. `allocator::default_alloc` is currently the only allocator supported.

Description

Provide compatible interface with `soa1d_container` while keeping the memory layout as an Array of Structures internally. User can easily switch between data layouts by changing the type of container they use. The rest of the code written against accessors and proxy elements and members can stay the same.

- Dynamic resizing with interface similar to `std::vector`
- Accessor objects suitable for efficient data access inside SIMD loops

Member	Description
`typedef size_t size_type;`	Type to use when specifying sizes to methods of the container.
`template <typename OffsetT = no_offset> using accessor;`	Template alias to an `accessor` for this container
`template <typename OffsetT = no_offset> using const_accessor;`	Template alias to a `const_accessor` for this container

Member Type	Description
`aos1d_container( size_type size_d1 = 0u, buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const allocator_type & an_allocator = allocator_type());`	Constructs an uninitialized container of size_d1 elements, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

Member Type	Description

`aos1d_container (`
`    size_type size_d1,`
`    const PrimitiveT &a_value,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container of *size_d1* elements initializing each with *a_value*, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`template<typename StlAllocatorT>`
`aos1d_container(`
`    const std::vector<PrimitiveT,`
`StlAllocatorT> &other,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container with a copy of each of the elements in *other*, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`aos1d_container(`
`    const PrimitiveT *other_array,`
`    size_type number_of_elements,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container with a copy of *number_of_elements* elements from the array *other_array*, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`template< typename IteratorT >`
`aos1d_container(`
`    IteratorT a_begin,`
`    IteratorT an_end,`
`    buffer_offset_in_cachelines buffer_offset`
`        = buffer_offset_in_cachelines(0),`
`    const allocator_type & an_allocator`
`        = allocator_type());`
	Constructs a container with as many elements as the range [*a_begin*-*an_end*), each with a copy of the value from its corresponding element in that range, in the same order, using optionally specified allocator instance, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing.

`aos1d_container clone() const;`	Returns: a new *aos1d_container* instance with its own copy of the elements
`void resize(size_type new_size_d1);`	Resize the container so that it contains *new_size_d1* elements. If the new size is greater than the current container size, the new elements are unitialized
`accessor<> access();`	Returns: *accessor* with no embedded index *offset*.
`accessor<int> access(int offset);`	Returns:*accessor* with an integer based embedded index *offset*.

`template<int IndexAlignmentT>`
`accessor<aligned_offset<IndexAlignmentT> >`
`    access(aligned_offset<IndexAlignmentT>);`
	Returns: *accessor* with an aligned_offset<IndexAlignmentT> based embedded index *offset.*

`template<int OffsetT>`
`accessor<fixed_offset<OffsetT> >`
`    access(fixed_offset<OffsetT>);`
	Returns: *accessor* with a fixed_offset<OffsetT> based embedded index *offset*.

Member Type	Description
`const_accessor<> const_access() const;`	Returns: *const_accessor* with no embedded index offset.
`const_accessor<int> const_access(int offset) const;`	Returns: *const_accessor* with an integer based embedded index *offset.*
`const_accessor<aligned_offset<IndexAlignmentT> > const_access(aligned_offset<IndexAlignmentT> offset) const;`	Returns:*const_accessor* with an aligned_offset<IndexAlignmentT> based embedded index *offset*.
`template<int OffsetT> const_accessor<fixed_offset<OffsetT> > const_access(fixed_offset<OffsetT>) const;`	Returns:*const_accessor* with a fixed_offset<OffsetT> based embedded index *offset*.

STL Compatibility

In addition to the performance oriented interface explained in the table above, `aos1d_container` implements a subset of the `std::vector` interface that is intended for ease of integration, not high performance. Due to the import/export only requirement we can't return a reference to the object, instead `iterators` and `operator[]` return a Proxy object while other "const' methods return a "value_type const". Furthermore, iterators do not support the `->` operator. Despite that limitation the iterators can be passed to any STL algorithm. Also for performance reasons, resize does not initialize new elements. The following `std::vector` interface methods are implemented:

- size, max_size, capacity, empty, reserve, shrink_to_fit
- assign, push_back, pop_back, clear, insert, emplace, erase
- cbegin, cend, begin, end, crbegin, crend, rbegin, rend, rbegin, rend
- operator[], front() const, back() const, at() const
- swap, ==, !=
- swap, aos1d_container(aos1d_container&& donor), aos1d_container & operator=(aos1d_container&& donor)

access_by

Enum to control how the memory layout will be accessed. `#include <sdlt/access_by.h>`

Syntax

```
enum access_by
{
    access_by_struct,
    access_by_stride
};
```

Description

The `access_by_struct` causes data access via structure member access. Nested structures will drill down through the structure members in a nested manner. For example an Axis Aligned Bounding Box (AABB) containing two Point3d objects (with x,y,z data members) will logically expand to something like:

```
AABB local;
local = accessor.mData[i];
```

access_by_stride will cause data access through pointers to built in types with a stride to account for the size of the primitive. For an Axis Aligned Bounding Box (AABB) containing two Point3d objects (with x,y,z data members) will logically expand to something like:

```
AABB local;
local.topLeft.x = *(accessor.mData + offsetof(AABB,topLeft) + offset(Point3d,x) +
(sizeof(AABB)*i));
local.topLeft.y = *(accessor.mData + offsetof(AABB,topLeft) + offset(Point3d,y) +
(sizeof(AABB)*i));
local.topLeft.z = *(accessor.mData + offsetof(AABB,topLeft) + offset(Point3d,z) +
(sizeof(AABB)*i));
local.topRight.x = *(accessor.mData + offsetof(AABB,topRight) + offset(Point3d,x) +
(sizeof(AABB)*i));
local.topRight.y = *(accessor.mData + offsetof(AABB,topRight) + offset(Point3d,y) +
(sizeof(AABB)*i));
local.topRight.z = *(accessor.mData + offsetof(AABB,topRight) + offset(Point3d,z) +
(sizeof(AABB)*i));
```

When vectorizing, access_by_struct can sometimes generate better code as the compiler could perform wide loads and use shuffle/insert instructions to move data into SIMD registers. However, depending on the complexity of the primitive, it can also fail to vectorize, especially when the primitive contains nested structures.

On the other hand access_by_stride has always vectorized successfully, because the data access is simplified to an array pointer with a stride. The compiler is able to handle any complexity of primitive, because it never sees the complexity and instead just sees the simple array pointer with strided access.

access_by_struct is probably the best choice as it offers a chance of better code generation especially when used outside of a SIMD loop. However if you run into issues when vectorizing, try access_by_stride to see if that alleviates the problem.

We leave this choice up to the developer and require they explicitly make a choice, so this is not hidden behavior.

n_container

Template class for N-dimensional container. The contained primitive type, exact memory layout and container shape are defined via template arguments.

Syntax

```
template <typename PrimitiveT,
          typename LayoutT,
          typename ExtentsT,
          typename AllocatorT >
class n_container;
```

Description

N-dimensional container of PrimitiveT elements with predefined memory layout and shape. Provides accessor interface suitable for flexible and efficient data access inside SIMD loops

The following table provides information on the template arguments for n_container

Template Argument	Description
`typename PrimitiveT`	The type that each cell in the multi-dimensional container will store. Requirements: PrimitiveT must be previously declared with the SDLT_PRIMITIVE macro.

Template Argument	Description
`typename LayoutT`	The in-memory data layout of cells in the container. Requirements: LayoutT must be a class from *layout* namespace.
`typename ExtentsT`	The shape of the container. Requirements: ExtentsT must be a concrete type of *n_extent_t* variadic template.
`class AllocatorT = allocator::default_alloc`	[Optional] Specify type of *allocator* to be used. allocator::default_alloc is currently the only allocator supported.

The following table provides information on the types defined as members of `n_container`

Member Type	Description
`typedef PrimitiveT primitive_type;`	Type inside each cell of the container.
`typedef PrimitiveT allocator_type;`	Type of allocator used by the container.
`typedef implementation-defined accessor`	Type of an *accessor* that can write or read cells to and from this container.
`typedef implementation-defined const_accessor;`	Type of a *const_accessor* that can read cells from this container.

The following table provides information on the methods of `n_container`

Member	Description
`n_container ( const ExtentsT &a_extents, buffer_offset_in_cachelines buffer_offset =buffer_offset_in_cachelines(0), const AllocatorT &an_allocator=AllocatorT())`	Constructs an uninitialized container of the shape defined as *a_extents*, using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing, using optionally specified allocator instance.
`n_container (buffer_offset_in_cachelines buffer_offset = buffer_offset_in_cachelines(0), const AllocatorT &an_allocator=AllocatorT())`	Constructs an uninitialized container of the shape, defined via template parameter ExtentsT using optionally specified number of cache lines to offset the start of the buffer in memory to allow management of 4k cache aliasing, using optionally specified allocator instance. ExtentsT must be default constructible. Only true when ExtentsT is made up enitrely of fixed<NumberT> types.
`n_container(n_container&& donor)`	Transfers ownership of the donor's currently owned buffers and organization, if any. Any outstanding *accessors* on the donor are no longer valid.

Member	Description
`n_container & operator = (n_container&& donor)`	Frees any existing buffers, then transfers ownership of the donor's currently owned buffers and organization, if any. Any outstanding *accessors* on the donor are no longer valid. **Returns:** Reference to this instance.
`const ExtentsT& n_extent () const`	Provides the shape of the container. Alternatively, the free template function *extent_d<int DimenstionT>(const n_container &)* could be used. **Returns:** Constant reference to ExtentsT instance describing the shape of the container.
`const_accessor const_access();`	Constructs an *const_accessor* with knowledge of the underlying data organization to read cells inside the container. **Returns:***const_accessor* for the container
`accessor access();`	Constructs an *accessor* with knowledge of the underlying data organization to write or read cells inside the container. **Returns:***accessor* for the container

The following table provides information about the friend functions of n_container.

Friend Function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_container & a_container)`	Append string representation of a_container's extents values to a_output_stream. **Returns:** Reference to a_output_stream for chained calls.

Layouts

sdlt::layout namespace

Rather than having different container types for different data layouts, the library uses the types from the layout namespace as a template parameter to the n_container.

Available layouts are defined in the namespace layout and summarized in table below.

Layout	Description
`template <typename AlignOnColumnIndexT=0> layout::soa`	Structure of Arrays: Each data member of the Primitive will have its own N-dimensional array. The arrays are placed back-to-back inside a contiguous buffer. Template parameter AlignOnColumnIndexT identifies which column of the row dimension should be cache line aligned. The AlignOnColumnIndexT of each row is cache line aligned.
`template <typename AlignOnColumnIndexT> layout::soa_per_row`	Structure of Arrays Per Row: Each data member of the Primitive will have its own 1-dimensional array for the row dimension (Soa1d) placed back to back. The AlignOnColumnIndexT of each row is cache line aligned. Multiple of these Soa1d's are laid out

Layout	Description
	sequentially to model the remaining dimensions, effectively becoming an Array of Structures of Arrays where the SOA where the size of the array is the row's extent. This can be particularly efficient when the extent of the row can be fixed<NumberT>. **Note**: If the size of the row isn't known at compile time, consider adding an additional dimension that is fixed<Number> and dividing the row up by that fixed<NumberT>.
`layout::aos_by_struct`	Array of Structures Accessed by Struct: Primitives are laid out in native format back to back in memory and access happens via structure or member access. Nested structures will drill down through the structure members in a nested manner.
`layout::aos_by_stride`	Array of Structures Accessed by Stride: Primitives are laid out in native format back to back in memory and accessed through pointers to built in types with a stride to account for the size of the Primitive. Can be useful if aos_by_struct doesn't vectorize.

Description

The classes are empty and only for specialization of containers for denoted layouts.

Shape

Variadic template class n_extent_t describes the shape of the n_dimensional container. Specifically, the number of dimensions the size of each.

Syntax

```
template<typename... TypeListT>
class n_extent_t
```

Description

`n_extent_t` represents the shape of a container as a sequence of sizes for each dimension. The size of each dimension can be represented by different types. This flexibility allows the same interface to be used to declare `n_extents_t` whose dimensions are fully known at compile time with `fixed<int NumberT>`, or to be only known at runtime with `int`, or only known at runtime but with a guarantee will be a multiple of an alignment with `aligned<int Alignment>`. For details, see the Number representation section.

The following table provides information on the template arguments for `n_extent_t`.

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the size of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array, from leftmost to rightmost.

Template Argument	Description
	Type must be `int`, `fixed<NumberT>`, or `aligned<AlignmentT>`. for each value describing corresponding dimensions size (extent) in regular order of C++ subscripts - from outer to inner.

The following table provides information on the members of `n_extent_t`

Member	Description
`static constexpr int rank;`	Number of dimensions.
`static constexpr int row_dimension = rank-1;`	Index of last dimension, *row.*
`n_extent_t()`	**Requirements:** Every type in TypeListT is default constructible. **Effects:** Construct n_extent_t, uses default values of each type in TypeListT for the dimesnion sizes. In general, only correctly initialized when every type is a fixed<NumberT>
`n_extent_t(const n_extent_t &a_other)`	**Effects:** Construct n_exent_t, copying size of each dimension from *a_other*.
`explicit n_extent_t(const TypeListT & … a_values)`	**Effects:** Construct n_exent_t, initializing each dimension with the corresponding value from the list of *a_values* passed as an argument. In use, *a_values* is a comma separate list of values whose length and types are defined by TypeListT.
`template<int DimensionT> auto get() const`	**Requirements:** DimenstionT >=0 and DimensiontT < rank. **Effects:** Determine the exent of *DimensionT*. **Returns:** In the type declared by the *DimensionT* position of 0-based TypeListT, the extent of the specified *DimensionT*
`template<int DimensionT>` `auto rightmost_dimensions() const`	**Requirements:** DimenstionT >=0 and DimensiontT <= rank. **Effects:** Construct a n_extent_t with a lower rank by copying the righmost DimensionT values from this instance. **Returns:** n_exent[get<rank - DimensionT>()] [get<rank + 1 - DimensionT>()] [get<…>()] [get<row_dimension>()]

`template<class... OtherTypeListT>`
`bool operator == (const`
`n_extent_t<OtherTypeListT...> a_other) const`
	Requirements: rank of a_other is the same as this instance's.

Member	Description
	Effects: Compare size of each dimension for equality. Only compares numeric values, not the types of each dimension. **Returns:***true* if all dimensions are numerically equal, *false* otherwise.
`template<class... OtherTypeListT> bool operator != (const n_extent_t<OtherTypeListT...> a_other) const`	**Requirements:** rank of a_other is the same as this instance's. **Effects:** Compare size of each dimension for inequality. Only compares numeric values, not the types of each dimension. **Returns:***true* if any dimensions are numerically different, *false* otherwise.
`size_t size() const`	**Returns:** Number of elements specified by extent **Effects:** Calculates the number of cells represented by the current extent values of each dimension by multiplying them all together. **Returns:** get<0>()*get<1>()*get<…>()*get<rank-1>()

The following table provides information on the friend functions of `n_extent_t`.

Friend function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_extent_t & a_extents)`	**Effects:** Append string representation of a_extents' values to a_output_stream **Returns:** Reference to a_output_stream for chained calls.

n_extent_generator
To facilitate simpler and clearer creation of `n_extent_t` *objects.*

Syntax

```
template<typename... TypeListT>
class n_extent_generator;

namespace {
    // Instance of generator object
    n_extent_generator<> n_extent;
}
```

Description

The generator object provides recursively constructing operators [] for `fixed<>`, `aligned<>`, and `integer` values allowing building of an `n_extent_t` <…> instance, one dimension at a time. The main purpose is to allow a usage syntax that is similar to C multi-dimensional array definition:

Compare the following examples, instantiating three n_extent_t instances. and using the generator object to instantiate equivalent instances.

```
n_extent_t<int, int> ext1(height, width);
n_extent_t<int, aligned<128>> ext2(height, width);
n_extent_t<fixed<1080>, fixed<1920>> ext3(1080_fixed, 1920_fixed);
```

```
auto ext1 = n_extent[height][width];
auto ext2 = n_extent[height][aligned<128>(width)];
auto ext3 = n_extent[1080_fixed][1920_fixed];
```

Class Hierarchy

It is expected that n_extent_generator < … > not be directly used as a data member or parameter, instead only n_extent_t <...> from which it is derived. The generator object n_extent can be automatically downcast any place expecting an n_extent_t<…> .

The following table provides the template arguments for n_extent_generator

Template Argument	Description
typename... TypeListT	Comma separated list of types, where the number of types provided controls how many dimensions the generator currently represent. Each type in the list identifies how the size of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. **Requirements**: Type is int, fixed<NumberT>, or aligned<AlignmentT>.

The following table provides information on the types defined as members of n_extent_generator in addition to those inherited from n_extent_t.

Member Type	Description
typedef n_extent_t<TypeListT...> value_type	Type value that the any chained [] operator calls have produced.

The following table provides information on the members of n_extent_generator in addition to those inherited from n_extent_t

Member	Description
n_extent_generator ()	**Requirements:** TypeListT is empty **Effects:** Construct generator with no extents specified
n_extent_generator (const n_extent_generator &a_other)	**Effects:** Construct generator copying any extent values from a_other
n_extent_generator<TypeListT..., int> operator [] (int a_size) const	**Requirements:** a_size >= 0 **Returns:** n_extent_generator<…> with additional rightmost integer based extent.
n_extent_generator<TypeListT..., fixed<NumberT>> operator [] (fixed<NumberT> a_size) const	**Requirements:** a_size >= 0 **Returns:** n_extent_generator<…> with additional rightmost fixed<NumberT> extent.

Member	Description
`n_extent_generator<TypeListT..., aligned<AlignmentT>> operator [] (aligned<AlignmentT> a_size)`	**Requirements:** a_size >= 0 **Returns:** n_extent_generator<...> with additional rightmost aligned<AlignmentT> based extent.
`value_type value() const`	**Returns:** n_extent_t<...> with the correct types and values of the multi-dimensional extents aggregated by the generator.

make_ n_container template function

Factory function to construct an instance of a properly-typed n_container<...> based on n_extent_t passed to it.

Syntax

```
template<
    typename PrimitiveT,
    typename LayoutT,
    typename AllocatorT = allocator::default_alloc,
    typename ExtentsT
>
auto make_n_container(const ExtentsT &_extents)
->n_container<PrimitiveT, LayoutT, ExtentsT, AllocatorT>
```

Description

Use `make_n_container` to more easily create an n-dimensional container using template argument deduction, and avoid specifying the type of extents.

An example of the instantiation of a High Definition image object is below.

```
typedef n_container<RGBAs, layout::soa,
                    n_extent_t<int, int>> HdImage;
HdImage image1(n_extent[1080][1920]);
```

Alternatively, it is possible to use factory function with the C++11 keyword `auto`, as shown below.

```
auto image1 = make_n_container<RGBAs,
                          layout::soa>(n_extent[1080][1920]);
```

extent_d template function

Syntax

```
template<int DimensionT, typename ObjT>
auto extent_d(const ObjT &a_obj)
```

Description

The template function offers a consistent way to determine the extent of a dimension for a multi-dimensional object. It can avoid extracting an entire n_extent_t<...> when only the extent of a single dimension is needed.

Template Argument	Description
`int DimensionT`	0 based index starting at the leftmost dimension indicating which n-dimensions to query the extent of.

Template Argument	Description
	Requirements: DimensionT >=0 and DimensionT < ObjT::rank
`typename ObjT`	The type of n-dimensional object from which to retrieve the extent. **Requirements:** ObtT is one of: n_container<…> n_extent_t<…> n_extent_generator<…>

Returns

The correctly typed extent corresponding to the requested DimensionT of a_obj.

Example

```
template <typename VolumeT>
void foo(const VolumeT & a_volume)
{
    int extent_z = extent_d<0>(volume);
    int extent_y = extent_d<1>(volume);
    int extent_x = extent_d<2>(volume);
    /…
}
```

Bounds

bounds_t

Class represents a half-open interval with lower and upper bounds. `#include <sdlt/bounds.h>`

Syntax

```
template<typename LowerT = int, typename UpperT = int>
struct bounds_t
```

Description

bounds_t holds the lower and upper bounds of a half open interval. It is templated to allow the different representations for the lower and upper bounds. Supported types include fixed<NumberT>, aligned<AlignmentT> and integer values. bounds_t models a valid iteration space over a single dimension.

bounds_t can be used to represent an iteration space over the entire extent of a dimension or to restrict iteration space within the extent. n_bounds_t aggregates a number of bounds_t objects to allow construction of multi-demensional subsections restricting multiple extents.

The class interface is compatible with C++ range-based loops to simplify iteration.

Template Argument	Description
`typename LowerT = int`	Type of lower bound. Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>
`typename UpperT = int`	Type of upper bound.

Template Argument	Description
	Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>

Member Types	Description
`typedef LowerT lower_type`	Type of the lower bound
`typedef UpperT upper_type`	Type of the upper bound
`typedef implementation-defined iterator`	Iterator type for C++ range-based loops support.

Member	Description
`bounds_t()`	Effects: Constructs bounds_t with uninitialized lower and upper bounds.
`bounds_t(lower_type l, upper_type u)`	Requirements: (u >= l) Effects: Constructs bounds_t representing the half-open interval [l, u)
`bounds_t(const bounds_t & a_other)`	Effects: Constructs bounds_t with lower and upper bounds initialized from those of a_other.
`template<typename OtherLowerT, typename OtherUpperT>` `bounds_t(const bounds_t<OtherLowerT, OtherUpperT> & a_other)`	Requirements: OtherLowerT and OtherUpperT can legally be converted to lower_type and upper_type. For example it would be illegal to convert an int to fixed<8>(). Effects: Constructs bounds_t with lower and upper bounds initialized from those of a_other.
`void set(lower_type l, upper_type u)`	Effects: Set index of the inclusive lower bound and the index of the exclusive upper bound.
`void set_lower(lower_type a_lower)`	Effects: Set index of the inclusive lower bound
`void set_upper(upper_type a_upper)`	Effects: Set index of the exclusive upper bound
`lower_type lower() const`	Returns: index of the inclusive lower bound
`upper_type upper() const`	Returns: index of the exclusive upper bound
`iterator begin() const`	Returns: index iterator for the inclusive lower bound. NOTE: C++11 range-based loops require begin() & end()
`iterator end() const`	Returns: index iterator for the exclusive upper bound. NOTE: C++11 range-based loops require begin() & end()
`auto width() const`	Effects: Determine width of iteration space inside the half open interval between lower() and upper() bounds. Returns: upper() – lower()

Member	Description
	NOTE: the return type depends on resulting type of a subtraction between the types of upper() and lower().

template<typename OtherLowerT,
        typename OtherUpperT>
bool contains(const bounds_t<OtherLowerT,
                OtherUpperT> &a_other) const
	Effects: Determine if interval of a_other is entirely contained inside this object's bounds
	Returns: (a_other.lower() >= lower() &&
	a_other.upper() <= upper())

template<typename T> auto operator + (const T &offset) const	Effects: create a new bounds_t instance with offset added to both lower and upper bounds. Returns: bounds(lower() + offset, upper()+offset) NOTE: The lower_type and upper_type of the returned bound_t maybe different as result of addition of the offset.
template<typename T> auto operator - (const T & offset) const	Effects: create a new bounds_t instance with offset subtracted from both lower and upper bounds. Returns: bounds(lower() - offset, upper()-offset) NOTE: The lower_type and upper_type of the returned object maybe different as result of subtraction of T.
bool operator == (const bounds_t &a_other) const	Effects: Equality comparison with same-typed bounds_t object Returns: (lower() == a_other.lower() && upper() == a_other.upper())

template<typename OtherLowerT,
        typename OtherUpperT>
bool operator == (
 const bounds_t<OtherLowerT,
                OtherUpperT> &a_other) const
	Effects: Equality comparison with bounds_t object of different lower_type or upper_type.
	Returns: (lower() == a_other.lower() && upper() == a_other.upper())

bool operator != (const bounds_t &) const	Effects: Inequality comparison with same-typed bounds_t object Returns: (lower() != a_other.lower() || upper() != a_other.upper())

template<typename OtherLowerT,
        typename OtherUpperT>
bool operator != (
 const bounds_t<OtherLowerT,
                OtherUpperT> &a_other) const
	Effects: Inequality comparison with with bounds_t object of different lower_type or upper_type
	Returns: (lower() != a_other.lower() || upper() != a_other.upper())

Friend Function	Description
std::ostream& operator << (std::ostream& a_output_stream, const bounds_t &a_bounds)	Effects: append string representation of bounds_t lower and upper values to a_output_stream

Friend Function	Description
	Returns: reference to a_output_stream for chained calls

Range-based loops support

The bounds_t provides begin() and end() methods returning iterators to enable C++11 range-based loops. The may save quite some typing and improve code clarity when iterating over bounds of a multidimensional container.

Compare:

```
auto ca = image_container.const_access();
auto b0 = bounds_d<0>(ca);
auto b1 = bounds_d<1>(ca);
for (auto y = b0.lower(); y < b0.upper(); ++y)
    for (auto x = b1.lower(); x < b1.upper(); ++x) {
          RGBAs pixel = ca[y][x];
            // …
    }
```

and

```
auto ca = image_container.const_access();
for (auto y: bounds_d<0>(ca))
    for (auto x: bounds_d<1>(ca)) {
          RGBAs pixel = ca[y][x];
            // …
    }
```

Note that iterator only gives an index value within the bounds, not an object value. It is expected to be used to index into accessors like in example above.

sdlt::bounds Template Function

Factory function provided for creation of bounds_t objects. `#include <sdlt/bounds.h>`

Syntax

```
template<typename LowerT, typename UpperT>
auto bounds(LowerT a_lower, UpperT a_upper)
```

Description

In order to make creation of objects of bounds_t cleaner the factory function bounds is provided. It basically enables LowerT and UpperT to be deduced from the arguments passed into it.

Template Argument	Description
`typename LowerT = int`	Type of lower bound. Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>
`typename UpperT = int`	Type of upper bound. Requirements: type is int, or fixed<NumberT>, or aligned<AlignmentT>

Returns:

The correctly typed bounds_t<LowerT, UpperT> corresponding to types of a_lower and a_upper passed to the factory function.

Example:

Compare two ways of instantiating a bounds:

```
bounds_t<fixed<0>, aligned<16>> my_bounds1(0_fixed, aligned<16>(upper))
auto my_bounds2 = bounds_t<fixed<0>, aligned<16>>(0_fixed, aligned<16>(upper))
```

With the factory function:

```
auto my_bounds = bounds(0_fixed, aligned<16>(upper))
```

n_bounds_t

Variadic template class to describe the valid iteration space over an N-dimensional container. `#include <sdlt/n_bounds.h>`

Syntax

```
template<typename... TypeListT>
class n_bounds_t
```

Description

n_bound_t represents the valid iteration space over a n_container or its accessor as as a sequence of bounds_t for each dimension. The bounds_t of each dimension can be represented by different types. This flexibility allows the same interface to be used to declare n_bounds_t whose dimensions are fully known at compile time with fixed<int NumberT>, or to be only known at runtime with int, or only known at runtime but with a guarantee will be a multiple of an alignment with aligned<int Alignment>. For details see the Number Representation section).

When an n_container is created, its n_bounds_t always start at fixed<0> for the inclusive lower bounds of each dimension, and exclusive upper bounds match the extent of the dimension. Accessors can be translated to different index spaces as well as restrict their iteration space to subsections, which will change the n_bounds_t those accessors provide.

The following table provides information on the template arguments for `n_bounds_t`.

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the bounds of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. Requirements: types in the list be bounds_t<LowerT, UpperT>

The following table provides information on the member types of n_bounds_t

Member Types	Description
`typedef implementation-defined lower_type`	Type of n_index_t<…> returned by method lower()
`typedef implementation-defined upper_type`	Type of n_index_t<…> returned by method upper()

The following table provides information on the members of n_bounds_t.

Member	Description
static constexpr int rank;	Number of dimensions
static constexpr int row_dimension = rank-1;	Index of last dimension considered to be the row
n_bounds_t()	Requirements: Every bounds_t in TypeListT is default constructible. Effects: Construct n_bounds_t, uses default values of each bounds_t in TypeListT for the dimesnion sizes. In general only correctly initialized when every bounds_t has an LowerT and UpperT that is a fixed<NumberT>.
n_bounds_t(const n_bounds_t &a_other)	Effects: Construct n_bounds_t, copying bounds of each dimension from a_other.
template<int DimensionT> auto get() const	Requirements: DimenstionT >=0 and DimensiontT < rank. Effects: Determine the bounds of DimensionT. Returns: In the type declared by the DimensionT position of 0-based TypeListT, the bounds_t of the specified DimensionT
lower_type lower()	Effects: build n_index<…> representing the inclusive lower bounds for all dimensions Returns: n_index[get<0>().lower()] [get<1>().lower()] [get<…>().lower()] [get<row_dimension>().lower()]
upper_type upper()	Effects: build n_index<…> representing the exclusive upper bounds for all dimensions Returns: n_index[get<0>().upper()] [get<1>(). upper ()] [get<…>(). upper ()] [get<row_dimension>().upper()]
template<typename... OtherTypeListT> bool contains(n_bounds_t<OtherTypeListT...> &a_other) const	Requirements: rank of a_other is the same as this instance's. Effects: Determine whether each dimension of the passed n_bounds_t is fully contained within bounds of each dimenson of this object. Returns: get<0>().contains(a_other.get<0>()) && get<1>().contains(a_other.get<1>()) && get<…>().contains(a_other.get<…>()) && get<row_dimension>().contains(a_other.get<row_dimension>())

Member	Description
`template<class... OtherTypeListT>` `bool operator == (const n_bounds_t<OtherTypeListT...> a_other) const`	Requirements: rank of a_other is the same as this instance's. Effects: Compare bounds each of dimension for equality. Only compares numeric values, not the types of each dimension. Returns: true if all dimensions are numerically equal, false otherwise.
`template<class... OtherTypeListT>` `bool operator != (const n_bounds_t<OtherTypeListT...> a_other) const`	Requirements: rank of a_other is the same as this instance's. Effects: Compare bounds of each dimension for inequality. Only compares numeric values, not the types of each dimension. Returns: true if any dimensions are numerically different, false otherwise.
`template<class ...OtherTypeListT>` `auto operator+ (const n_index_t<OtherTypeListT...> a_offset) const`	Requirements: rank of a_other is the same as this instance's. Effects: construct a n_bound_t whose types and bounds value for each dimension are determined by taking the bounds for each dimension and adding the an offset for that dimension from a_offset. Returns: n_bounds[get<0>() + a_offset.get<0>()] [get<1>() + a_offset.get<1>()] [get<…>() + a_offset.get<…>()] [get<row_dimension>() + a_offset.get< row_dimension >()]
`template<int DimensionT>` `auto rightmost_dimensions() const`	Requirements: DimenstionT >=0 and DimensiontT <= rank. Effects: Construct a n_bounds_t with a lower rank by copying the righmost DimensionT values from this instance. Returns: n_bounds[get<rank – DimensionT>()] [get<rank + 1 – DimensionT>()] [get<…>()] [get<row_dimension>()]
`template<class... OtherTypeListT>` `auto overlay_rightmost(const n_bounds_t<OtherTypeListT...> & a_other) const`	Requirements: rank of a_other is <= rank Effects: Construct copy of n_bounds_t where the rightmost dimensions' values are copied from a_other, effectively overlaying a_other ontop of rightmost dimensions of this instance. Returns: n_bounds[get<0>()]

Member	Description
	[get<1 >()] [get<…>()] [get<rank-a_other::rank>()] [a_other.get<0>()] [a_other.get<…>()] [a_other.get<a_other::row_dimension>()]

The following table provides information on the friend functions of n_bounds_t.

Friend Function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_bounds_t & a_bounds_list)`	Effects: append string representation of a_bounds_list values to a_output_stream Returns: reference to a_output_stream for chained calls.

n_bounds_generator

Facilitates simple creation of n_bounds_t objects.

`#include <sdlt/n_bounds.h>`

Syntax

```
template<typename... TypeListT>
class n_bounds_generator;

namespace {
    // Instance of generator object
    n_bounds_generator<> n_bounds;
}
```

Description

The generator object provides recursively constructing operators [] for bounds_t<LowerT, UpperT> values allowing building of a n_bounds_t<…> instance one dimension at a time. Its main purpose is to allow a usage syntax that is similar to C multi-dimensional array definition:

Compare creating two n_bounds_t instances:

```
n_bounds_t<bounds_t<fixed<540>, fixed<1080>>,
           bounds_t<fixed<960>, fixed<1920>>> bounds1(bounds_t<540_fixed,1080_fixed>(),
        bounds_t<960_fixed, 1920_fixed>));

n_bounds_t<bounds_t<int, int>,
           bounds_t<int, int>> bounds2(bounds_t<int, int>(540,960),
        bounds_t<int, int>(960, 1920));
```

and the equivalent instances using the generator objects and factory functions

```
auto bounds1 = n_bounds[bounds(540_fixed, 1080_fixed)]
                       [bounds(960_fixed, 1920_fixed)];
auto bounds2 = n_bounds[bounds(540, 1080)]
                       [bounds(960, 1920)];
```

or alternatively using the operator() with n_index_t and n_extent_t generator objects

```
auto bounds1 = n_bounds(n_index[540_fixed][960_fixed],
                         n_extent[540_fixed][960_fixed]);

auto bounds2 = n_bounds(n_index[540][960],
                        n_extent[540][960]);
```

Class Hierarchy

It is expected that n_bounds_generator<…> not be directly used as a data member or parameter, instead only n_bounds_t<...> from which it is derived. The generator object n_bounds can be automatically downcast any place expecting a n_bounds_t<…>.

The following table provides information on the template arguments for n_bounds_generator

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the bounds of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. Requirements: types in the list be bounds_t<LowerT, UpperT>

The following table provides information on the types defined as members of n_bounds_generator in addition to those inherited from n_bounds_t

Member Types	Description
`typedef n_bounds_t<TypeListT...> value_type`	Type value that the any chained [] operator calls have produced.

The following table provides information on the members of n_bounds_generator in addition to those inherited from n_bounds_t

Member	Description
`n_bounds_generator()`	Requirements: TypeListT is empty Effects: Construct generator with no bounds specified
`n_bounds_generator(const n_bounds_generator &a_other)`	Effects: Construct generator copying any bounds values from a_other
`template<typename LowerT, typename UpperT> auto operator [] (const bounds_t<LowerT, UpperT> & a_bounds) const`	Effects: build a n_bounds_generator<…> with additional rightmost bounds_t<LowerT, UpperT> based dimension. Returns: n_bounds_generator<TypeListT..., bounds_t< LowerT, UpperT >>
`template<class... IndexTypeListT, class... ExtentTypeListT> auto operator () ( const n_index_t<IndexTypeListT...> &`	Requirements: rank of a_indices is same as rank of a_extents and TypeListT be empty

Member	Description
`a_indices,` `const n_extent_t<ExtentTypeListT...> & a_extents) const`	Effects: build a n_bounds_generator<…> where n-lower bounds are specified by a_indices, and n-upper bounds are calculated by adding a_extents to a_indices Returns: n_bounds[bounds(a_indices.get<0>(), a_indices.get<0>() + a_extents.get<0>())] [bounds(a_indices.get<1>(), a_indices.get<1>() + a_extents.get<1>())] [bounds(a_indices.get<…>(), a_indices.get<…>() + a_extents.get<…>())] [bounds(a_indices.get<row_dimension>(), a_indices.get< row_dimension >() + a_extents.get< row_dimension >())]
`value_type value() const`	Returns: n_bounds_t<…> with the correct types and values of the multi-dimensional bounds aggregated by the generator.

bounds_d Template Function

Provides a consistent way to determine the bounds of a dimension for a multi-dimensional object. `#include <sdlt/n_extent.h>`

Syntax

```
template<int DimensionT, typename ObjT>
auto bounds_d(const ObjT &a_obj)
```

Description

Consistent way to determine the bounds of a dimension for a multi-dimensional object. Can avoid extracting an entire n_bounds_t<…> when only the extent of a single dimension is needed.

Template Argument	Description
`int DimensionT`	0 based index starting at the leftmost dimension indicating which n-dimensions to query the bounds of. Requirements: DimensionT >=0 and DimensionT < ObjT::rank
`typename ObjT`	The type of n-dimensional object from which to retrieve the extent. Requirements: ObtT is one of: n_container<…> n_bounds_t<…> n_bounds_generator<…> n_container<…>::accessor

Template Argument	Description
	n_container<…>::const_accessor or any sectioned or translated accessor.

Returns:

The correctly typed bounds_t<LowerT, UpperT> corresponding to the requested DimensionT of a_obj.

Example:

```
template <typename VolumeT>
void foo(const VolumeT & a_volume)
{
    auto bounds_z = bounds_d<0>(volume);
    auto bounds_y = bounds_d<1>(volume);
    auto bounds_x = bounds_d<2>(volume);
    for(auto z : bounds_z)
        for(auto y : bounds_y)
            for(auto x : bounds_x) {
                // …
            }
}
```

Accessors

soa1d_container::accessor and aos1d_container::accessor

Lightweight object provides efficient array subscript [] access to the read or write elements from inside a soa1d_container or aos1d_container. `#include <sdlt/soa1d_container.h>` *and* `#include <sdlt/aos1d_container.h>`

Syntax

```
template <typename OffsetT> soa1d_container::accessor;
template <typename OffsetT> aos1d_container::accessor;
```

Arguments

`typename OffsetT`	The type offset that will be applied to each operator[] call determined by the type of offset passed into `soa1d_container::access(offset)`/ `aos1d_container::access(offset)` which constructs an accessor.

Description

`accessor` provides [] operator that returns a proxy object representing an Element inside the Container that can export or import the Primitive's data. Can re-access with an offset to create a new *accessor* that when accessed at [0] will really be accessing at index corresponding to the embedded offset. Lightweight and meant to be passed by value into functions or lambda closures. Use accessors in place of pointers to access the logical array data.

Member	Description
`accessor();`	Default Constructible

Member	Description
`accessor(const accessor &);`	Copy Constructible
`accessor & operator = (const accessor &);`	Copy Assignable
`const int & get_size_d1() const;`	Returns: Number of elements in the container.
`auto operator [] (int index_d1) const`	Returns: proxy Element representing element *at index_d1* in the container..
`template<typename IndexT_D1> auto operator [] (const IndexT_D1 index_d1);`	When: IndexT_D1 is one of the SDLT defined or generated Index types, Returns: proxy Element representing element at *index_d1* in the container.
`auto reaccess(const int offset) const;`	Returns: *accessor* with an integer-based embedded index *offset*.
`template<int IndexAlignmentT> auto reaccess(aligned_offset<IndexAlignmentT> offset) const;`	Returns: *accessor* with an `aligned_offset<IndexAlignmentT>` based embedded index *offset*.
`template<int fixed_offsetT> auto reaccess(fixed_offset<fixed_offsetT>) const;`	Returns: *accessor* with a `fixed_offset<OffsetT>` based embedded index *offset*.

soa1d_container::const_accessor and aos1d_container::const_accessor

Lightweight object provides efficient array subscript [] access to the read elements from inside a soa1d_container or aos1d_container. `#include <sdlt/soa1d_container.h>` *and* `#include <sdlt/aos1d_container.h>`

Syntax

```
template <typename OffsetT> soa1d_container::const_accessor;
template <typename OffsetT> aos1d_container::const_accessor;
```

Arguments

`typename OffsetT`	The type offset that embedded offset that will be applied to each operator[] call

Description

`const_accessor` provides [] operator that returns a proxy object representing a const Element inside the Container that can export the Primitive's data. Can re-access with an offset to create a new *const_accessor* that when accessed at [0] will really be accessing at index corresponding to the embedded offset. Lightweight and meant to be passed by value into functions or lambda closures. Use *const_accessors* in place of const pointers to access the logical array data.

Member	Description
`const_accessor();`	Default Constructible

Member	Description
`const_accessor(const const_accessor &);`	Copy Constructible
`const_accessor & operator = (const const_accessor &);`	Copy Assignable
`const int & get_size_d1() const;`	Returns: Number of elements in the container.
`auto operator [] (int index_d1) const`	Returns: proxy ConstElement representing element at*index_d1* in the container..
`template<typename IndexT_D1> auto operator [] (const IndexT_D1 index_d1);`	When: IndexT_D1 is one of the SDLT defined or generated Index types. Returns: proxy ConstElement representing element at *index_d1* in the container.
`auto reaccess(const int offset) const;`	Returns: *const_accessor* with an integer-based embedded index *offset*.
`template<int IndexAlignmentT> auto reaccess(aligned_offset<IndexAlignmentT> offset) const;`	Returns: *const_accessor* with an `aligned_offset<IndexAlignmentT>` based embedded index *offset*.
`template<int fixed_offsetT> auto reaccess(fixed_offset<fixed_offsetT>) const;`	Returns: *const_accessor* with a `fixed_offset<OffsetT>` based embedded index *offset*.

Accessor Concept

Accessor and const_accessor objects obtained via `n_container::access()` *and* `n_container::const_access()` *provide access to read from or write to cells inside an* `n_container`*.*

Syntax

The following methods return objects meeting the requirements of the accessor concept.

```
auto n_container::access();
auto n_container::const_access();
auto accessor_concept::section(n_bounds_t<…>);
auto accessor_concept::translated_to(n_index_t<…>);
auto accessor_concept::translated_to_zero();
```

Description

Accessor objects provide read/write access to individual cells of an n-dimensional container. Index values passed to a sequence of array subscript operator calls will produce a proxy concept that can import to or export the primitive data the corresponding cell inside the container.

```
auto image = make_n_container<MyStruct, layout::soa>(n_extent[128][256]);
auto acc = image.access();
MyStruct in_value(100.0f, 200.0f, 300.0f);

acc[64][128] = in_value;
```

```
MyStruct out_value = acc[64][128];

assert(out_value == in_value);
```

Accessors also know their valid iteration space, which can queried using the template function bound_d<int DimensionT>(accessor).

```
assert(bounds_d<0>(acc) == bounds(0_fixed,128));
assert(bounds_d<1>(acc) == bounds(0_fixed,256));
```

An accessor may have a non-zero index space if it has a translation embedded into it, `bounds_d` will reflect any such translation.

```
auto shifted_acc = acc.translated_to(n_index[1000][2000]);
assert(bounds_d<0>(shifted_acc) == bounds(1000,1128));
assert(bounds_d<1>(shifted_acc) == bounds(2000,2256));
```

This is useful to have a smaller sized container participate in a calculation over a portion of a larger index space, simplifying programming as the same index variable can be used, and the accessor takes care of applying the necessary translation. An accessor may represent a subsection over the original extents, bounds_d will identify the valid iteration space for that accessor.

```
auto subsection_acc = a.section(n_bounds[bounds(64,96)][bounds(128,160)]);
assert(bounds_d<0>(subsection_acc) == bounds(64, 96));
assert(bounds_d<1>(subsection_acc) == bounds(128, 160);
```

It can also be useful to have subsections be translated back to start their iteration space at 0. For efficiency, the translated_to_zero() method is provided to create an accessor shifted back to zero.

```
auto zb_sub_acc = a.section( n_bounds[bounds(64, 96)][bounds(128, 160)] ).translated_to_zero();
assert(bounds_d<0>(zb_sub_acc) == bounds(0, 32));
assert(bounds_d<1>(zb_sub_acc) == bounds(0, 32));
```

If fewer array subscript calls applied to an accessor than its rank, the result is another accessor of a lower rank. This can be useful to obtain accessors suitable to pass to code expecting lower rank accessors. Such as a obtaining a 3d accessor from a 4d container by specifying only a single index via array subscript. This has the effect of embedding the index value of the dimension inside accessor. When the final dimension is sliced, the result is a proxy object to the cell inside the container corresponding to the embedded index values inside the sliced accessors

```
auto image4d = make_n_container<MyStruct, layout::soa>(n_extent[10][20][128][256]);

MyStruct in_value(100.0f, 200.0f, 300.0f);
auto acc4d = image4d.access();
auto acc3d = acc4d[5];
auto acc2d = acc3d[10];
auto acc1d = acc2d[64];
acc1d[128] = in_value;
MyStruct out_value = acc4d[5][10][64][128];
assert(out_value == in_value);
```

The following table provides information on the requirements of the accessor concept.

Pseudo-Signature	Description
`typedef PrimitiveT primitive_type;`	Data type inside the cells of the container.
`static constexpr int rank;`	Number of free dimensions of accessor

Pseudo-Signature	Description
accessor_concept(const accessor_concept &a_other)	Effects: constructs a copy of another accessor of the exact same type
template<typename IndexT> element_concept operator[] (const IndexT a_index) const	**Requirements:** rank == 1 and IndexT is one of: int, aligned<AlignmentT>, fixed<NumberT>, linear_index, or simd_index<LaneCountT> **Effects:** When only 1 free dimension is left, the operator[] will construct an element_concept which is the proxy to the cell inside the container. If this accessor was obtained with const_access(), then the proxy will provide read only interface to the cell's data. **Returns:** The proxy object to cell inside the container corresponding to the position identified by the a_index along with any embedded index values for other dimensions
template<typename IndexT> accessor_concept operator[] (const IndexT a_index) const	**Requirements:** rank > 1 and IndexT is one of: int, aligned<AlignmentT>, fixed<NumberT>, linear_index, or simd_index<LaneCountT> **Effects:** When 2 or more free dimensions are left, the operator[] will construct another accessor_concept of lower rank embeding a_index inside of it, effectively fixing that dimension's index value for any accesses made through the returned accessor_concept. **Returns:** The accessor_concept of lower rank (one less free dimension).
template<int DimensionT> auto bounds_d() const	**Requirements:** DimensionT >=0 and DimensionT < rank **Effects:** Determine the bounds of a free dimension using DimensionT as a 0 based index starting at the leftmost dimension. **Returns:** bounds_t of the DimensionT
auto bounds_d*XX*() const where XX is 0-19	**Requirements:** XX >=0 and XX < rank and XX < 20 **Effects:** Non templated methods to determine the bounds of a free dimension using XX as a 0 based index starting at the leftmost dimension. **Returns:** bounds_t of the XX dimension
template<int DimensionT> auto extent_d() const	**Requirements:** DimensionT >=0 and DimensionT < rank **Effects:** Determine the extent of a free dimension using DimensionT as a 0 based index starting at the leftmost dimension. Returns: extent of the DimensionT

Pseudo-Signature	Description
`auto extent_dXX() const` `where XX is 0-19`	**Requirements:** XX >=0 and XX < rank and XX < 20 **Effects:** Non templated methods to determine the extent of a free dimension using XX as a 0 based index starting at the leftmost dimension. Returns: extent of the XX dimension

`template<typename ...IndexListT>`
`accessor_concept translated_to(`
`    n_index_t<IndexListT...> a_n_index) const`
	Requirements: a_n_index has same rank as the accessor
	Effects: construct an accessor_concept with an embedded translation such that accessing a_n_index will corresponds back to the current lower bounds. Easy way to think of it is that current iteration space is translated to a_n_index space.
	Returns: accessor_concept whose bounds have the same extents, but whose lower bounds start at the supplied a_n_index

`template<typename ...IndexListT>` `accessor_concept translated_to_zero() const`	**Effects:** construct an accessor_concept with an embedded translation such that accessing [0] index for all dimensions will corresponds back to the current lower bounds. Easy way to think of it is that current iteration space is translated to [0] for all free dimensions. **Returns:** accessor_concept whose bounds have the same extents, but whose lower bounds start [0]…[0]

`template<typename ...BoundsTypeListT>`
`    auto`
`    section(const`
`n_bounds_t<BoundsTypeListT...> &a_n_bounds)`
`const`
	Requirements: a_n_bounds has same rank as the accessor and a_n_bounds is contained by the accessors current bounds.
	Effects: construct an accessor_concept with using the supplied a_n_bounds to represent its valid iteration space. Because a_n_bounds must be contained within the existing bounds, we are effictively creating an accessor over a section of the container. Easy way to think of it is that current bounds are being restricted to a_n_bounds. Note: can be useful to chain a call `translated_to_zero()` on to the return value.
	Returns:`accessor_concept` whose bounds are set to the supplied a_n_bounds

Proxy Objects

*accessor*s can't return a reference to the Primitive because its memory layout is abstracted. Instead a Proxy object is returned. That Proxy supports importing or exporting data to and from the Container. The actual type of Proxy objects is an implementation detail, but they all support the same public interface which we will document.

Each *accessor* [index] operator returns a Proxy object.

Each *const_accessor* [index] operator returns a ConstProxy object.

The Proxy objects provide a Data Member Interface where for each data member of *value_type* they are representing, a member access method is defined which returns a new Proxy or ConstProxy representing just that data member. Users can drill down through a complex data structure to get a Proxy representing the exact data member they need versus importing and exporting the entire Primitive value.

Proxy objects also overload the following operators if the underlying *value_type* supports the operator:

==, !=, <, >, <=, >=, +, -, *, /, %, &&, ||, &, |, ^, ~, *, +, -, !, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --

Proxy

Proxy object provides access to a specific Primitive, Primitive data member, or nested data member within a Primitive for an element in a container.

Description

accessor [index] or a Proxy object's Data Member Interfaces return Proxy objects. That Proxy object represents the Primitive, Primitive data member, or nested data member within a Primitive for an element in a container. The Proxy object has the following features:

- A value_type can be exported or imported from the Proxy.
  - Conversion operator is used to export the value_type
  - Alternatively the Proxy can be passed to the function `unproxy` to export a `value_type`
  - Assignment operator = is used to import value_type into the Proxy
- Overloads the following operators if the underlying value_type supports the operator
  - ==, !=, <, >, <=, >=, +, -, *, /, %, &&, ||, &, |, ^, ~, *, +, -, !, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --
  - When an operator is called the following occurs:
    - value_type is exported
    - The operator applied to the exported value
    - If the operator was an assignment, the result is imported back into the Member and returns the proxy
    - Otherwise a result is returned.
- Data Member Interface.
  - For each data member of value_type
    - A member access method is defined which returns a Member proxy representing just that member.

Member Type	Description
`typedef implementation-defined value_type`	The type of the data the Proxy is representing

Member	Description
`operator value_type const () const;`	Returns: exports a copy of the Proxy's value. NOTE: constant return value prevents rvalue assignment for structs offering some protection against code that expected a modifiable reference.
`const value_type &` `operator = (const value_type &a_value);`	Imports a_value into container at the position the Proxy is representing. Returns: the same constant value_type it was passed.

Member	Description
	NOTE: This behavior is different from traditional assignment operators that return *this. Choice was to enable efficient chaining of assignment operators versus returning a Proxy which would have to export the value it had just imported.
`Proxy & operator = (const Proxy &other);`	Exports value from the other Proxy and imports it. Returns: A reference to this Proxy obect.
`auto name_of_values_data_member_1()const;`	Returns: Proxy instance representing the 1st data member of the value_type NOTE: actual method name is the name of the value_type's 1st data member
`auto name_of_values_data_member_2()const;`	Returns: Proxy instance representing the 2nd data member of the value_type. NOTE: actual method name is the name of the value_type's 2nd data member.
`auto name_of_values_data_member_...()const;`	Returns: Proxy instance representing the ...th data member of the value_type. NOTE: actual method name is the name of the value_type's …th data member.
`auto name_of_values_data_member_N()const;`	Returns: Proxy instance representing the Nth data member of the value_type. NOTE: actual method name is the name of the value_type's Nth data member

ConstProxy

ConstProxy object provides access to a specific constant primitive, primitive data member, or nested data member within a primitive for an element in a container.

Description

const_accessor [index] or a ConstProxy object's Data Member Interfaces return ConstProxy objects. That ConstProxy object represents the constant primitive, primitive data member, or nested data member within a primitive for an element in a container. The ConstProxy object has the following features:

- A value_type can be exported or imported from the ConstProxy.
  - Conversion operator is used to export the value_type
  - Alternatively the ConstProxy can be passed to the function `unproxy` to export a `value_type`
- Overloads the following operators if the underlying value_type supports the operator
  - ==, !=, <, >, <=, >=, +, -, *, /, %, &&, ||, &, |, ^, ~, *, +, -, !
  - When an operator is called the following occurs:
    - value_type is exported
    - The operator applied to the exported value
    - returns the result.
- Data Member Interface.
  - For each data member of value_type

- A member access method is defined which returns a Member ConstProxy representing just that member.

Member Type	Description
`typedef implementation-defined value_type`	The type of the data the ConstProxy is representing

Member	Description
`operator value_type const () const;`	Returns: exports a copy of the ConstProxy's value. NOTE: constant return value prevents rvalue assignment for structs offering some protection against code that expected a modifiable reference.
`auto name_of_values_data_member_1()const;`	Returns: ConstProxy instance representing the 1st data member of the value_type NOTE: actual method name is the name of the value_type's 1st data member
`auto name_of_values_data_member_2()const;`	Returns: ConstProxy instance representing the 2nd data member of the value_type. NOTE: actual method name is the name of the value_type's 2nd data member.
`auto name_of_values_data_member_...()const;`	Returns: ConstProxy instance representing the ...th data member of the value_type. NOTE: actual method name is the name of the value_type's ...th data member.
`auto name_of_values_data_member_N()const;`	Returns: ConstProxy instance representing the Nth data member of the value_type. NOTE: actual method name is the name of the value_type's Nth data member

Number Representation

When specifying extents, positions inside of, or bounds of a container, numeric values can be represented three different ways: `fixed`, `aligned`, and `int`. `Fixed` is most precise and `int` is least precise. It is advised to use as precise specification as possible. The compiler may optimize better with more information.

Fixed

Represent a numerical constant whose value specified at compile time.

```
template <int NumberT> class fixed;
```

If offsets applied to index values inside a SIMD loop are known at compile time, then the compiler can use that information. For example, to maintain aligned access, if boundary is fixed and known to be aligned when accessing underlying data layout. When multiple accesses are happening near each other, the compiler will have the opportunity to detect which accesses occur in the same cache lines and potentially avoid prefetching the same cache line repeatedly. Additionally, if the start of an iteration space is known at compile time, if it's a multiple of the SIMD lane count, the compiler could skip generating a peel loop. Whenever possible, `fixed` values should be used over `aligned` or arbitrary integer values.

Although `std::integral_constant<int>` provides the same functionality, the library defines own type to provide overloaded operators and avoid collisions with any other code's interactions with `std::integral_constant<int>`.

The following table provides information about the template arguments for `fixed`.

Template Argument	Description
`int Number T`	The numerical value the fixed will represent.

The following table provides information about the members of `fixed`.

Member	Description
`static constexpr int value = NumberT`	The numerical value known at compile-time.
`constexpr operator value_type() const`	**Returns:** The numerical value
`constexpr value_type operator()() const;`	**Returns:** The numerical value

Constant expression arithmetic operators +,- (both unary and binary), * and / are defined for type `sdlt::fixed<>` and will be evaluated at compile-time.

The suffix `_fixed` is a C++11 user-defined equivalent literal. For example, `1080_fixed` is equivalent to `fixed<1080>`. Consider the readability of the two samples below.

```
foo3d(fixed<1080>(), fixed<1920>());
```

versus

```
foo3d(1080_fixed, 1920_fixed);
```

> **NOTE** The `sdlt::fixed<NumberT>` type supersedes the deprecated `sdlt::fixed_offset<OffsetT>` type found in SDLT v1. It is strongly advised to use `sdlt::fixed<NumberT>`. However, in this release, a template alias is provided mapping `sdlt::fixed_offset<OffsetT>` onto `sdlt::fixed<NumberT>`.

Aligned

Represent integer value known at compile time to be a multiple of an `IndexAlignment`.

```
template <int IndexAlignmentT> class aligned;
```

If you can tell the compiler that you know that an integer will be a multiple of known value, then, when combined with a loop index inside a SIMD loop, the compiler can use that information to maintain aligned access when accessing underlying data layout.

Internally, the integer value is converted to a block count, where:

```
block_count = value/IndexAlignmentT;
```

Overloaded math operations can then use that aligned block count as needed. The value() is represented by `AlignmentT*block_count` allowing the compiler to prove that the value() is a multiple of `AlignmentT`, which can utilize alignment optimizations.

The following table provides information about the template arguments for `aligned`.

Template Argument	Description
`int IndexAlignmentT`	The alignment the user is stating that the number is a multiple of. `IndexAlignmentT` must be a power of two.

The following table provides information about the types defined as members of aligned.

Member Type	Description
typedef int value_type	The type of the numerical value.
typedef int block_type	The type of the block_count.

The following table provides information about the members of aligned.

Member	Description
static const int index_alignment	The IndexAlignmentT value.
aligned()	Constructs empty (uninitialized) object
explicit aligned(value_type)	Constructs computing block_count=a_value/ IndexAlignmentT.
aligned(const aligned& a_other)	Constructs copying block_count from a_other. a_other must have same IndexAlignmentT.
template<int OtherAlignment> explicit aligned(const aligned& other)	Constructs computing block_count optimized by avoiding computing other.value(). Must have IndexAlignmentT of a_other < IndexAlignmentT and other.value() be multiple of IndexAlignmentT.
template<int OtherAlignment> aligned(const aligned& other)	Constructs computing block_count with a multiply instead of divide. Must have IndexAlignmentT of a_other > IndexAlignmentT
static aligned from_block_count(block_type block_count)	Creates an instance of aligned avoiding any math by directly using supplied block_count
value_type value() const	Computes the value represented by the aligned. **Returns:** aligned_block_count()*IndexAlignmentT
operator value_type()	Conversion to int. **Returns:** value()
block_type aligned_block_count() const	Conversion to int. **Returns:** The block count

The following operations are supported for the aligned type.

Operation	Description
operator *(int), commutative	Scale value. Returns: aligned<IndexAlignmentT >
operator *(fixed<V>), commutative	Scales IndexAlignment by 2^M and value by K. Must have V=2^M*K (V is a multiple of a power of 2). **Returns:** aligned<IndexAlignmentT*(2^M)>

Operation	Description
operator *(aligned<OtherAl>)	Scales IndexAlignment by OtherAl and block_count by argument. **Returns:** aligned<IndexAlignmentT*OtherAl>
int operator/(fixed<IndexAlignmentT>)	**Returns:** aligned_block_count()
int operator/(fixed<-IndexAlignmentT>)	**Returns:** -aligned_block_count();
int operator/(fixed<V>)	Must have abs(V)>IndexAlignmentT && IndexAlignmentT%V==0. **Returns:** aligned_block_count()/(V/ IndexAlignmentT)
int operator/(fixed<V>)	Must have abs(V) < IndexAlignmentT && V %IndexAlignmentT==0 **Returns:** aligned_block_count()*(IndexAlignmentT/V)
aligned operator -()	**Returns:** Same type aligned for negated value.
aligned operator -(const aligned &) const	**Returns:** Same type aligned for value of difference.
template<int OtherAl> aligned<?> operator -(const aligned<OtherAl>&) const	Difference with other alignment. Behavior and returned alignment type depend on relation between alignments of operands. **Returns:** Value for difference as lower of incoming alignments
template<int V> aligned<?> operator -(const fixed<V> &) const	Difference with fixed value. Behavior and returned alignment type depend on relation between alignments of aligned<> operand and the value of V. **Returns:** Adjusted aligned value of a difference
aligned operator +(const aligned &)const	**Returns:** Same type aligned for value of sum
template<int OtherAl> aligned<?> operator +(const aligned<OtherAl>&) const	Sum with other alignment. Behavior and returned alignment type depend on relation between alignments of operands. **Returns:** Value for sum as lower of incoming alignments
template<int V> aligned<?> operator +(const fixed<V> &) const	Sum with fixed value. Behavior and returned alignment type depend on relation between alignments of aligned<> operand and the value of V. **Returns:** Adjusted aligned value of a sum.

Operation	Description
`template<int OtherAl>` `aligned operator +=(const aligned<OtherAl> &) const`	Increments value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl` **Returns:** Aligned with incremented value.
`template<int OtherAl>` `aligned operator -=(const aligned<OtherAl> &) const`	Decrements value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl` **Returns:** Same type aligned with decremented value.
`template<int OtherAl>` `aligned operator *=(const aligned<OtherAl> &) const`	Multiplies value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl`. **Returns:** Same type aligned with multiplied value.
`template<int OtherAl>` `aligned operator /=(const aligned<OtherAl> &) const`	Divides value for the aligned object if `IndexAlignmentT` is compatible with `OtherAl` **Returns:** Same type aligned with divided value.

NOTE The `sdlt::aligned<>` type supersedes the deprecated `sdlt::aligned_offset<>` type found in SDLT v1. It is strongly advised to use `sdlt::aligned<>`, however in this release a template alias is provided mapping `sdlt::aligned_offset<>` onto `sdlt::aligned<>`.

int

Represents an arbitrary integer value. In interfaces where fixed<> and aligned<> values supported you may also use plain old integer value. It provides least information among these three and so least facilitates compiler optimizations.

aligned_offset

Represent an integer based offset whose value is a multiple of an IndexAlignment specified at compile time. `#include <sdlt/aligned_offset.h>`

Syntax

```
template<int IndexAlignmentT>
class aligned_offset;
```

Arguments

`int IndexAlignmentT`	The index alignment the user is stating that the offset have.

Description

aligned_offset is a deprecated feature.

If we can tell the compiler that we know an offset will be a multiple of known value, then when combined with a loop index inside a SIMD loop, the compiler can use that information to maintain aligned access when accessing underlying data layout.

Internally, the offset value is converted to a block count.

```
Block Count = offsetValue/IndexAlignmentT;
```

Indices can then use that aligned block count as needed.

Member	Description
`static const int IndexAlignment = IndexAlignmentT;`	The alignment the offset is a multiple of
`explicit aligned_offset(const int offset)`	Construct instance based on offset
`static aligned_offset from_block_count(int aligned_block_count);`	Returns: Instance based on aligned_block_count, where the offset value = IndexAlignment*aligned_block_count
`int aligned_block_count() const;`	Returns: number of blocks of IndexAlignment it takes to represent the offset value.
`int value() const;`	Returns: offset value

fixed_offset

Represent an integer based offset whose value specified at compile time. `#include <sdlt/fixed_offset.h>`

Syntax

```
template <int OffsetT> fixed_offset;
```

Arguments

`int OffsetT`	The value the fixed_offset will represent

Description

fixed_offset is a deprecated feature.

If we can tell the compiler that we know an offset at compile time, then when combined with a loop index inside a SIMD loop, the compiler can use that information to maintain aligned access (should the offset be aligned) when accessing underlying data layout. When multiple accesses are happening near each other, the compiler will have the opportunity to detect which accesses occur in the same cache lines and potentially avoid prefetching the same cache line repeatedly. Whenever possible, a fixed_offset should be used over an aligned_offset or integer based offset.

Member	Description
`static constexpr int value = OffsetT`	The offset value known at compile

Indexes

soa1d_container's and aos1d_container's accessors [] operator can accept an integer based loop index. However if any modifications were applied to that loop index, the fact that it's a loop index may be lost by the compiler as it is handled before being passed to the [] operator.

To avoid this situation, SDLT provides classes to wrap loop indexes that capture multiple additions or subtractions of offsets (see the Offsets section). The resulting index can be passed to [] and preserve the original loop index and track any arithmetic with Offsets to be applied to underlying data layout.

It is common for stencil based algorithms to need to apply offsets during data access.

For a regular linear loop, use linear_index to wrap your loop index.

linear_index

Wraps an integer-based loop index that is iterating linearly through an iteration space. `#include <sdlt/linear_index.h>`

Syntax

```
class linear_index;
```

Description

Inside of a linear loop, wrap the loop index with a `linear_index` to allow addition or subtraction of offsets.

Member	Description
`explicit linear_index(int an_index);`	Construct instance from a loop index
`int value() const;`	Returns the original loop index

n_index_t

Variadic template class n_index_t describes a position inside of the N-dimensional container. Specifically, the number of dimensions and the of index value of each.

Syntax

```
template<typename... TypeListT>
class n_index_t
```

Description

`n_index_t` represents a position inside an n-dimensional space as a sequence of index value for each dimension. The index of each dimension can be represented by different types. This flexibility allows the same interface to be used to declare `n_index_t` with indices that are fully known at compile time with `fixed<int NumberT>`, or to be only known at runtime with `int`, or only known at runtime but with a guarantee will be a multiple of an alignment with `aligned<int Alignment>`. For more details, see the Number representation section.

Objects of this class may be used to identify a cell in a container, describe the inclusive lower bounds for `n_bounds()`, n-dimensional position for accessor's *translated_to()*.

The following table provides information about the template arguments for `n_index_t`.

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions there are. Each type in the list identifies how the index of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array, from leftmost to rightmost. **Requirements**: Type must be `int`, or `fixed<NumberT>`, or `aligned<AlignmentT>`.

The following table provides information about the members of `n_index_t`

Member	Description
`static constexpr int rank;`	Number of dimensions.
`static constexpr int row_dimension = rank-1;`	Index of last dimension, *row.*
`n_index_t()`	Default constructor. Uses default values for extent types. **Requirements:** Every type in TypeListT is default constructible. **Effects:** Construct n_index_t, uses default values of each type in TypeListT for the dimesnion sizes. In general only correctly initialized when every type is a fixed<NumberT>.
`n_index_t(const n_extent_t &a_other)`	Copy constructor. **Effects:** Construct n_index_t, copying index value of each dimension from a_other.
`explicit n_index_t(const TypeListT & … a_values)`	**Returns:** The last extent in its native type **Effects:** Construct n_index_t, initializing each dimension with the corresponding value from the list of *a_values* passed as an argument. In use, *a_values* is a comma separate list of values whose length and types are defined by TypeListT.
`template<int DimensionT>` `auto get() const`	**Requirements:** DimenstionT >=0 and DimensiontT < rank. **Effects:** Determine the index value of DimensionT. **Returns:** In the type declared by the DimensionT position of 0-based TypeListT, the index value of the specified *DimensionT*
`n_index_t operator +() const`	**Effects:** Determine the positive unary value of each dimension's index, effectively no operation is performed **Returns:** Copy of the current instance.
`auto operator -() const`	**Effects:** Determine the negative unary value of each dimension's index **Returns:** n_index[-get<0>()] [-get<1>()] [-get<…>()] [-get<row_dimension>()]

`template<class... OtherTypeListT>`
`auto operator +(`
`   const n_index_t<OtherTypeListT...> & a_other) const`
	Requirements: Rank of a_other is the same as this instance's.
	
	Effects: Build n_index_t whose values are the result of adding the index value for each dimension with those of a_other

Member	Description
	Returns: n_index[get<0>() + a_other.get<0>()] [get<1>() + a_other.get<1>()] [get<…>() + a_other.get<…>()] [get<row_dimension>() + a_other.get<row_dimension>()]

`template<class... OtherTypeListT>`
`auto operator -(`
`   const n_index_t<OtherTypeListT...> &`
`a_other) const`
	Requirements: Rank of a_other is the same as this instance's.
	Effects: Build n_index_t whose values are the result of subtracting the index value for each dimension of a_other with this instance's.
	Returns: n_index[get<0>() - a_other.get<0>()]
	[get<1>() - a_other.get<1>()]
	[get<…>() - a_other.get<…>()]
	[get<row_dimension>() -
	a_other.get<row_dimension>()]

`template<class... OtherTypeListT>`
`bool operator == (const`
`n_index_t<OtherTypeListT...> a_other) const`
	Requirements: Rank of a_other is the same as this instance's.
	Effects: Compare index of each dimension for equality. Only compares numeric values, not the types of each dimension.
	Returns:*true* if all dimensions are numerically equal, *false* otherwise.

`template<class... OtherTypeListT>`
`bool operator != (const`
`n_index_t<OtherTypeListT...> a_other) const`
	Requirements: Rank of a_other is the same as this instance's.
	Effects: Compare index of each dimension for inequality. Only compares numeric values, not the types of each dimension.
	Returns: *true* if any dimensions are numerically different, *false* otherwise.

`template<int DimensionT>` `auto rightmost_dimensions() const`	**Requirements:** DimenstionT >=0 and DimensiontT <= rank. **Effects:** Construct a n_index_t with a lower rank by copying the righmost DimensionT values from this instance. **Returns:** n_index[get<rank - DimensionT>()] [get<rank + 1 - DimensionT>()] [get<…>()] [get<row_dimension>()]

Member	Description
`template<class... OtherTypeListT> auto overlay_rightmost(const n_index_t<OtherTypeListT...> & a_other) const`	**Requirements:** rank of a_other is <= rank **Effects:** Construct copy of n_index_t where the rightmost dimensions' values are copied from a_other, effectively overlaying a_other ontop of rightmost dimensions of this instance. **Returns**: n_index[get<0>()] [get<1 >()] [get<...>()] [get<rank-a_other::rank>()] [a_other.get<0>()] [a_other.get<...>()] [a_other.get<a_other::row_dimension>()]

The following table provides information about the friend functions of `n_index_t`

Friend Function	Description
`std::ostream& operator << (std::ostream& output_stream, const n_index_t & a_indices)`	**Effects:** Append string representation of a_indices' values to a_output_stream. **Returns:** Reference to a_output_stream for chained calls.

n_index_generator

To facilitate simpler creation of `n_index_t` *objects, the generator object* `n_index` *is provided.*

Syntax

```
template<typename... TypeListT>
class n_index_generator;

namespace {
    // Instance of generator object
    n_index_generator<> n_index;
}
```

Description

The generator object provides recursively constructing operators [] for `fixed<>`, `aligned<>`, and `integer` values allowing building of a `n_index_t<…>` instance one dimension at a time. Its main purpose is to allow a usage syntax that is similar to C multi-dimensional array definition.

Compare the following examples, instantiating three `n_index_t` instances, and using the generator object to instantiate equivalent instances.

```
n_index_t<int, int> idx1(row, col);
n_index_t<int, aligned<16>> idx2(row, aligned<16>(col));
n_index_t<fixed<540>, fixed<960>> idx3(540_fixed, 960_fixed);
```

```
auto idx1 = n_index[row][col];
auto idx2 = n_index[row][aligned<16>(col)];
auto idx3 = n_index[540_fixed][960_fixed];
```

Class Hierarchy

It is expected that `n_index_generator` < … > not be directly used as a data member or parameter, instead only `n_index_t` <…> from which it is derived. The generator object `n_index` can be automatically downcast any place expecting an `n_index_t<…>` .

The following table provides the template arguments for `n_index_generator`

Template Argument	Description
`typename... TypeListT`	Comma separated list of types, where the number of types provided controls how many dimensions the generator currently represents. Each type in the list identifies how the size of the corresponding dimension is to be represented. The order of the dimensions is the same order as C++ subscripts declaring a multi-dimensional array – from leftmost to rightmost. **Requirements**: Type is `int`, `fixed<NumberT>`, or `aligned<AlignmentT>`.

The following table provides information on the types defined as members of `n_index_generator` in addition to those inherited from `n_index_t`.

Member Type	Description
`typedef n_index_t<TypeListT...> value_type`	Type value that the any chained [] operator calls have produced.

The following table provides information on the members of `n_index_generator` in addition to those inherited from `n_index_t`

Member	Description
`n_index_generator ()`	**Requirements:** TypeListT is empty. **Effects:** Construct generator with no indices specified.
`n_index_generator (const n_index_generator &a_other)`	**Effects:** Construct generator copying any index values from a_other
`n_index_generator<TypeListT..., int> operator [] (int a_index) const`	**Requirements:** a_size >= 0. **Returns:** n_index_generator<…> with additional rightmost integer based index.
`n_index_generator<TypeListT..., fixed<NumberT>> operator [] (fixed<NumberT> a_index) const`	**Requirements:** a_size >= 0. **Returns:** n_index_generator<…> with additional rightmost fixed<NumberT> index.
`n_index_generator<TypeListT..., aligned<AlignmentT>> operator [] (aligned<AlignmentT> a_index)`	**Requirements:** a_size >= 0 **Returns:** n_index_generator<…> with additional rightmost aligned<AlignmentT> based index.
`value_type value() const`	**Returns:** n_extent_t<…> with the correct types and values of the multi-dimensional extents aggregated by the generator.

index_d template function

Syntax

```
template<int DimensionT, typename ObjT>
auto index_d(const ObjT &a_obj)
```

Description

The template function offers a consistent way to determine the index of a dimension for a multi-dimensional object. It can avoid extracting an entire `n_index_t<…>` when only the extent of a single dimension is needed.

Template Argument	Description
`int DimensionT`	0 based index starting at the leftmost dimension indicating which n-dimensions to query the index of. **Requirements:** DimensionT >=0 and DimensionT < ObjT::rank
`typename ObjT`	The type of n-dimensional object from which to retrieve the extent. **Requirements:** ObtT is one of: n_index_t<…> n_index_generator<…>

Returns

The correctly typed index corresponding to the requested DimensionT of a_obj.

Example

```
template <typename IndicesT>
void foo(const IndicesT & a_pos)
{
    int z = index_d<0>(a_pos);
    int y = index_d<1>(a_pos);
    int x = index_d<2>(a_pos);
    /…
}
```

Convenience and Correctness

Users can include a single header file `sdlt.h` that includes all the supported public features, or users can include the individual headers of features they will be using (which might build faster). In other words,

```
#include <sdlt/sdlt.h>
```

instead of

```
#include <sdlt/primitive.h>
#include <sdlt/soa1d_container.h>
```

For convenience, SDLT provides a macro to encapsulate `#pragma forceinline recursive`.

```
SDLT_INLINE_BLOCK
```

SDLT reduces overhead by trusting the programmer to pass it valid values for template and function parameters. Adding conditional checks inside of a SIMD loop can cause unnecessary code generation and inhibit vectorization by creating multiple exit points in a loop. To assist in verifying that a program is indeed passing valid values to SDLT, the programmer can add a compilation flag to their build to define `SDLT_DEBUG=1`.

```
-DSDLT_DEBUG=1
```

If `_DEBUG` is defined and `SDLT_DEBUG` has not been defined to 0 or 1, then `SDLT_DEBUG` is automatically set to 1. When set to 1, every operator[] is bounds checked and all addresses are validated for correct alignment. It is very useful for tracking down any usage bugs.

The macro `__SDLT_VERSION` is predefined to be 2001. Programs could use it for conditional compilation if incompatibilities arise in future updates.

C++ implementations of `std::min` and `std::max` sometimes have a negative impact on performance. SDLT defines `min_val` and `max_val` that help avoid such performance penalties.

max_val

Return the right value if the right value is greater than left, otherwise returns the left value. *`#include <sdlt/min_max_val.h>`*

Syntax

```
template<typename T>
T max_val(const T left, const T right);
```

Arguments

`typename T`	The type of the left and right values

Description

C++ implementations of std::min and std::max create a conditional control flow that returns references to its parameters, which may cause inefficient vector code generation. max_val is a really simple template that returns by value instead of reference, allowing more efficient vector code to be generated. For most cases the algorithm didn't need a reference to the inputs and a copy by value should suffice. It should inline, adding no overhead. Inside of SIMD loops, we suggest using sdlt::max_val in place of std::max.

Requires < operator be defined for the type T.

min_val

Return the left value if the right value is greater than left, otherwise returns the right value. *`#include <sdlt/min_max_val.h>`*

Syntax

```
template<typename T>
T min_val(const T left, const T right);
```

Arguments

`typename T`	The type of the left and right values

Description

C++ implementations of std::min and std::max create a conditional control flow that returns references to its parameters, which may cause inefficient vector code generation. min_val is a really simple template that returns by value instead of reference, allowing more efficient vector code to be generated. For most cases the algorithm didn't need a reference to the inputs and a copy by value should suffice. It should inline, adding no overhead. Inside of SIMD loops, we suggest using sdlt::min_val in place of std::min.

Requires < operator be defined for the type T.

Examples

The example programs in this section demonstrate

- the efficiency of using SDLT and its Structure of Arrays approach rather than a typical Array of Structures
- construction of more complex SDLT primitives,
- performance improvement in case of a forward-dependency
- use of offsets and calling methods on the SDLT primitive, and
- RGB to YUV conversion.

Efficiency with Structure of Arrays Example

This example demonstrates the efficiency of using a Structure of Arrays (SoA) approach by comparing the assembly generated from a simple SIMD loop using an Array of Structures (AoS) approach with the assembly generated using the SoA approach of SDLT.

Array of Structures: Non-unit stride access version

Source:

```
#include <stdio.h>

#define N 1024

typedef struct RGBs {
    float r;
    float g;
    float b;
} RGBTy;


void main()
{
    RGBTy a[N];
    #pragma omp simd
    for (int k = 0; k<N; ++k) {
        a[k].r = k*1.5;  // non-unit stride access
        a[k].g = k*2.5;  // non-unit stride access
        a[k].b = k*3.5;  // non-unit stride access
    }
    std::cout << "k =" << 10 <<
        ", a[k].r =" << a[10].r <<
        ", a[k].g =" << a[10].g <<
        ", a[k].b =" << a[10].b << std::endl;
}
```

AVX2 assembly generated (69 instructions):

```
..TOP_OF_LOOP:
        vcvtdq2ps %ymm7, %ymm1
        lea       (%rax), %rcx
```

```
        vcvtdq2ps %ymm5, %ymm2
        vpaddd    %ymm3, %ymm7, %ymm7
        vpaddd    %ymm3, %ymm5, %ymm5
        vmulps    %ymm1, %ymm4, %ymm8
        vmulps    %ymm1, %ymm6, %ymm12
        vmulps    %ymm2, %ymm6, %ymm14
        vmulps    %ymm1, %ymm0, %ymm1
        vmulps    %ymm2, %ymm4, %ymm10
        addl      $16, %edx
        vextractf128 $1, %ymm8, %xmm9
        vmovss    %xmm8, (%rsp,%rcx)
        vmovss    %xmm9, 48(%rsp,%rcx)
        vextractps $1, %xmm8, 12(%rsp,%rcx)
        vextractps $2, %xmm8, 24(%rsp,%rcx)
        vextractps $3, %xmm8, 36(%rsp,%rcx)
        vmulps    %ymm2, %ymm0, %ymm8
        vextractps $1, %xmm9, 60(%rsp,%rcx)
        vextractps $2, %xmm9, 72(%rsp,%rcx)
        vextractps $3, %xmm9, 84(%rsp,%rcx)
        vextractf128 $1, %ymm12, %xmm13
        vextractf128 $1, %ymm14, %xmm15
        vextractf128 $1, %ymm1, %xmm2
        vextractf128 $1, %ymm8, %xmm9
        vmovss    %xmm12, 4(%rsp,%rax)
        vmovss    %xmm13, 52(%rsp,%rax)
        vextractps $1, %xmm12, 16(%rsp,%rax)
        vextractps $2, %xmm12, 28(%rsp,%rax)
        vextractps $3, %xmm12, 40(%rsp,%rax)
        vextractps $1, %xmm13, 64(%rsp,%rax)
        vextractps $2, %xmm13, 76(%rsp,%rax)
        vextractps $3, %xmm13, 88(%rsp,%rax)
        vmovss    %xmm14, 100(%rsp,%rax)
        vextractps $1, %xmm14, 112(%rsp,%rax)
        vextractps $2, %xmm14, 124(%rsp,%rax)
        vextractps $3, %xmm14, 136(%rsp,%rax)
        vmovss    %xmm15, 148(%rsp,%rax)
        vextractps $1, %xmm15, 160(%rsp,%rax)
        vextractps $2, %xmm15, 172(%rsp,%rax)
        vextractps $3, %xmm15, 184(%rsp,%rax)
        vmovss    %xmm1, 8(%rsp,%rax)
        vextractps $1, %xmm1, 20(%rsp,%rax)
        vextractps $2, %xmm1, 32(%rsp,%rax)
        vextractps $3, %xmm1, 44(%rsp,%rax)
        vmovss    %xmm2, 56(%rsp,%rax)
        vextractps $1, %xmm2, 68(%rsp,%rax)
        vextractps $2, %xmm2, 80(%rsp,%rax)
        vextractps $3, %xmm2, 92(%rsp,%rax)
        vmovss    %xmm8, 104(%rsp,%rax)
        vextractps $1, %xmm8, 116(%rsp,%rax)
        vextractps $2, %xmm8, 128(%rsp,%rax)
        vextractps $3, %xmm8, 140(%rsp,%rax)
        vmovss    %xmm9, 152(%rsp,%rax)
        vextractps $1, %xmm9, 164(%rsp,%rax)
        vextractps $2, %xmm9, 176(%rsp,%rax)
        vextractps $3, %xmm9, 188(%rsp,%rax)
        addq      $192, %rax
        vextractf128 $1, %ymm10, %xmm11
        vmovss    %xmm10, 96(%rsp,%rcx)
```

```
        vmovss    %xmm11, 144(%rsp,%rcx)
        vextractps $1, %xmm10, 108(%rsp,%rcx)
        vextractps $2, %xmm10, 120(%rsp,%rcx)
        vextractps $3, %xmm10, 132(%rsp,%rcx)
        vextractps $1, %xmm11, 156(%rsp,%rcx)
        vextractps $2, %xmm11, 168(%rsp,%rcx)
        vextractps $3, %xmm11, 180(%rsp,%rcx)
        cmpl      $1024, %edx
        jb        ..TOP_OF_LOOP
```

Structure of Arrays: Using SDLT for unit stride access

To introduce the use of SDLT, the code below will:

- declare a primitive,
- use an `soa1d_container` instead of an array
- use an accessor inside a SIMD loop to generate efficient code
- use a proxy object's data member interface to access individual data members of an element inside the container

Source:

```
#include <stdio.h>
#include <sdlt/sdlt.h>

#define N 1024

typedef struct RGBs {
    float r;
    float g;
    float b;
} RGBTy;

SDLT_PRIMITIVE(RGBTy, r, g, b)

void main()
{
    // Use SDLT to get SOA data layout
    sdlt::soa1d_container<RGBTy> aContainer(N);
    auto a = aContainer.access();

    // use SDLT Data Member Interface to access struct members r, g, and b.
    // achieve unit-stride access after vectorization
    #pragma omp simd
    for (int k = 0; k<N; k++) {
        a[k].r() = k*1.5;
        a[k].g() = k*2.5;
        a[k].b() = k*3.5;
    }
    std::cout << "k =" << 10 <<
        ", a[k].r =" << a[10].r() <<
        ", a[k].g =" << a[10].g() <<
        ", a[k].b =" << a[10].b() << std::endl;
}
```

AVX2 assemply generated (19 instructions):

```
..TOP_OF_LOOP:
        vpaddd    %ymm4, %ymm3, %ymm12
        vcvtdq2ps %ymm3, %ymm7
```

```
        vcvtdq2ps %ymm12, %ymm10
        vmulps    %ymm7, %ymm2, %ymm5
        vmulps    %ymm7, %ymm1, %ymm6
        vmulps    %ymm7, %ymm0, %ymm8
        vmulps    %ymm10, %ymm2, %ymm3
        vmulps    %ymm10, %ymm1, %ymm9
        vmulps    %ymm10, %ymm0, %ymm11
        vmovups   %ymm5, (%r13,%rax,4)
        vmovups   %ymm6, (%r15,%rax,4)
        vmovups   %ymm8, (%rbx,%rax,4)
        vmovups   %ymm3, 32(%r13,%rax,4)
        vmovups   %ymm9, 32(%r15,%rax,4)
        vmovups   %ymm11, 32(%rbx,%rax,4)
        vpaddd    %ymm4, %ymm12, %ymm3
        addq      $16, %rax
        cmpq      $1024, %rax
        jb        ..TOP_OF_LOOP
```

Both versions appear to have unrolled the loop twice. When examining the assembly generated for AVX2 instruction set, we can see a measurable reduction in the number of instructions (19 vs. 69) when we are able to perform unit stride access using SDLT. Also, at runtime, the `soa1d_container` aligned its data allocation and will gain any of the architectural advantages that come with using aligned instead of unaligned SIMD stores.

Complex SDLT Primitive Construction Example

This example demonstrates use of nested primitives and the use of an accessor inside a SIMD loop to generate efficient code.

```
#include <stdio.h>
#include <sdlt/sdlt.h>

#define N 1024

typedef struct XYZs {
    float x;
    float y;
    float z;
} XYZTy;

SDLT_PRIMITIVE(XYZTy, x, y, z)

typedef struct RGBs {
    float r;
    float g;
    float b;
    XYZTy w;
} RGBTy;

SDLT_PRIMITIVE(RGBs, r, g, b, w)

void main()
{
    sdlt::soa1d_container<RGBTy> aContainer(N);
    auto a = aContainer.access();

    #pragma omp simd
    for (int k = 0; k<N; k++) {
        RGBTy c;
```

```
        c.r = k*1.5f;
        c.g = k*2.5f;
        c.b = k*3.5f;
        c.w.x = k*4.5f;
        c.w.y = k*5.5f;
        c.w.z = k*6.5f;
        a[k] = c;
    }
    const RGBTy c = a[10];
    printf("k = %d, a[k].r = %f, a[k].g = %f, a[k].b = %f \n",
        10, c.r, c.g, c.b);

    printf("k = %d, a[k].w.x = %f, a[k].w.y = %f, a[k].w.z = %f \n",
        10, c.w.x, c.w.y, c.w.z);
```

Forward Dependency Example

This example demonstrates the declaration of a Structure of Arrays (SoA) interacting with a forward dependency.

```
#include <stdio.h>
#include <sdlt/primitive.h>
#include <sdlt/soa1d_container.h>

#define N 1024

typedef struct RGBs {
    float r;
    float g;
    float b;
} RGBTy;

SDLT_PRIMITIVE(RGBTy, r, g, b)

void main()
{
    // RGBTy a[N];  // AOS data layout

    sdlt::soa1d_container<RGBTy> aContainer(N);
    auto a = aContainer.access();    // SOA data layout

    // use SDLT access method to access struct members r, g, and b.
    // with unit-stride access  after vectorization
    #pragma omp simd
    for (int k = 0; k<N; k++) {
        a[k].r() = k*1.5;
        a[k].g() = k*2.5;
        a[k].b() = k*3.5;
    }

    // Test forward-dependency on SOA memory access
    #pragma omp simd
    for (int i = 0; i<N - 1; i++) {
        sdlt::linear_index k(i);
        a[k].r() = a[k + 1].r() + k*1.5;
        a[k].g() = a[k + 1].g() + k*2.5;
        a[k].b() = a[k + 1].b() + k*3.5;
    }
    std::cout << "k =" << 10 <<
```

```
        ", a[k].r =" << a[10].r() <<
        ", a[k].g =" << a[10].g() <<
        ", a[k].b =" << a[10].b() << std::endl;
}
```

Use of Offsets and Methods on a SDLT Primitive Example

This example demonstrates a linearized 2d stencil using embedded offsets and calling methods on the primitive.

```
#include <sdlt/sdlt.h>

// Typical C++ object to represent a pixel in an image
struct RGBs
{
    float red;
    float green;
    float blue;

    RGBs() {}
    RGBs(const RGBs &iOther)
        : red(iOther.red)
        , green(iOther.green)
        , blue(iOther.blue)
    {
    }

    RGBs & operator =(const RGBs &iOther)
    {
        red = iOther.red;
        green = iOther.green;
        blue = iOther.blue;
        return *this;
    }

    RGBs operator + (const RGBs &iOther) const
    {
        RGBs sum;
        sum.red = red + iOther.red;
        sum.green = green + iOther.green;
        sum.blue = blue + iOther.blue;
        return sum;
    }

    RGBs operator * (float iScalar) const
    {
        RGBs scaledColor;
        scaledColor.red = red * iScalar;
        scaledColor.green = green * iScalar;
        scaledColor.blue = blue * iScalar;
        return scaledColor;
    }
};

SDLT_PRIMITIVE(RGBs, red, green, blue)

const int StencilHaloSize = 1;
const int width = 1920;
const int height = 1080;
```

```
template<typename AccessorT> void loadImageStub(AccessorT) {}
template<typename AccessorT> void saveImageStub(AccessorT) {}


// performs average color filtering with neighbors left,right,above,below
void main(void)
{
    // We are padding +-1 so we can avoid boundary conditions
    const int paddedWidth = width + 2 * StencilHaloSize;
    const int paddedHeight = height + 2 * StencilHaloSize;
    int elementCount = paddedWidth*paddedHeight;
    sdlt::soa1d_container<RGBs> inputImage(elementCount);
    sdlt::soa1d_container<RGBs> outputImage(elementCount);

    loadImageStub(inputImage.access());

    SDLT_INLINE_BLOCK
    {
        const int endOfY = StencilHaloSize + height;
        const int endOfX = StencilHaloSize + width;
        for (int y = StencilHaloSize; y < endOfY; ++y)
        {
            // Embed offsets into Accessors to get the to correct row
            auto prevRow = inputImage.const_access((y - 1)*paddedWidth);
            auto curRow = inputImage.const_access(y*paddedWidth);
            auto nextRow = inputImage.const_access((y + 1)*paddedWidth);

            auto outputRow = outputImage.access(y*paddedWidth);

            #pragma omp simd
            for (int ix = StencilHaloSize; ix < endOfX; ++ix)
            {
                sdlt::linear_index x(ix);

                const RGBs color1 = curRow[x - 1];
                const RGBs color2 = curRow[x];
                const RGBs color3 = curRow[x + 1];
                const RGBs color4 = prevRow[x];
                const RGBs color5 = nextRow[x];
                // Despite looking like AOS code, compiler is able to create
                // privatized instances and call inlinable methods on the objects
                // keeping the algorithm at very high level
                const RGBs sumOfColors = color1 + color2 + color3 + color4 + color5;
                const RGBs averageColor = sumOfColors*(1.0f / 5.0f);
                outputRow[x] = averageColor;
            }
        }
    }
    saveImageStub(outputImage.access());
}
```

RGB to YUV Conversion Example

This example converts a 2D image from the RGB format to the YUV format. It demonstrates how storing both images in 2D SoA `n_container`s can improve performance.

```
#include <iostream>
#include <sdlt/sdlt.h>
using namespace sdlt;
#define WIDTH 1024
#define HEIGHT 1024

struct RGBs {
    float r;
    float g;
    float b;
};

struct YUVs {
    float y;
    float u;
    float v;

    YUVs(){ };

    YUVs& operator=(const RGBs &tmp){
        y = 0.229f * tmp.r + 0.587f * tmp.g + 0.114f * tmp.b;
        u = -0.147f * tmp.r - 0.289f * tmp.g + 0.436f * tmp.b;
        v = 0.615 * tmp.r - 0.515f * tmp.g - 0.100 * tmp.b;
        return *this;
    }
    YUVs(const RGBs &tmp){
        y = 0.229f * tmp.r + 0.587f * tmp.g + 0.114f * tmp.b;
        u = -0.147f * tmp.r - 0.289f * tmp.g + 0.436f * tmp.b;
        v = 0.615 * tmp.r - 0.515f * tmp.g - 0.100 * tmp.b;
   }
};

SDLT_PRIMITIVE(RGBs, r, g, b)
SDLT_PRIMITIVE(YUVs, y, u, v)

int main(){
    typedef layout::soa<> LayoutT;
    n_extent_t<int, int> extents(HEIGHT, WIDTH);

    /*  Creating a typedef for SoA N-dimensional container.
        RGBTy and YUVTy are user defined structures whose collection needs to be stored in SoA
format in memory.
        Layout in memory specified as layout::soa.
        In the below case N-dimensional SoA container is used in 2-D context
    */
    typedef sdlt::n_container< RGBs, LayoutT, decltype(extents) > ContainerRGB;
    typedef sdlt::n_container< YUVs, LayoutT, decltype(extents) > ContainerYUV;

    //Instantiate Input and Output Containers
    ContainerRGB inputRGB(extents);
    ContainerYUV outputYUV(extents);

    auto input = inputRGB.const_access();   //Get Constant Accessor object for inputRGB
```

```
    auto output = outputYUV.access();       //Get Accessor object for outputYUV

    //Select the iteration range in each dimension
    const auto iRGB1 = bounds_d<1>(input);  //bound_d<1>(input);
    const auto iRGB0 = bounds_d<0>(input);  //bound_d<0>(input);

    for(int y = iRGB0.lower(); y < iRGB0.upper(); y++)
    {
        #pragma simd
        for (int x = iRGB1.lower(); x < iRGB1.upper(); x++){
            const RGBs temp1 = input[y][x];
            YUVs temp2 = temp1;
            output[y][x] = temp2;
        }
    }
    return 0;
}
```

Intel® C++ Class Libraries

The Intel® C++ Class Libraries enable Single-Instruction, Multiple-Data (SIMD) operations. The principle of SIMD operations is to exploit microprocessor architecture through parallel processing. The effect of parallel processing is increased data throughput using fewer clock cycles. The objective is to improve application performance of complex and computation-intensive audio, video, and graphical data bit streams.

Hardware and Software Requirements

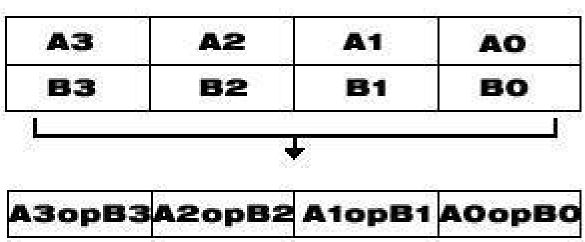
The Intel® C++ Class Libraries are functions abstracted from the instruction extensions available on Intel® processors.

Refer to http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/ for information on which Intel® processors use each instruction set.

Details About the Libraries

The Intel® C++ Class Libraries for SIMD Operations provide a convenient interface to access the underlying instructions for processors as specified above. These processor-instruction extensions enable parallel processing using the single instruction-multiple data (SIMD) technique as illustrated in the following figure.

SIMD Data Flow



Performing four operations with a single instruction improves efficiency by a factor of four for that particular instruction.

These new processor instructions can be implemented using assembly inlining, intrinsics, or the C++ SIMD classes. Compare the coding required to add four 32-bit floating-point values, using each of the available interfaces:

Comparison Between Inlining, Intrinsics and Class Libraries

Assembly Inlining	Intrinsics	SIMD Class Libraries
`... __m128 a,b,c; __asm{ movaps xmm0,b movaps xmm1,c addps xmm0,xmm1 movaps a, xmm0 } ...`	`#include <xmmintrin.h> ... __m128 a,b,c; a = _mm_add_ps(b,c); ...`	`#include <fvec.h> ... F32vec4 a,b,c; a = b +c; ...`

This table shows an addition of four single-precision floating-point values using assembly inlining, intrinsics, and the libraries. You can see how much easier it is to code with the Intel C++ SIMD Class Libraries. Besides using fewer keystrokes and fewer lines of code, the notation is like the standard notation in C++, making it much easier to implement over other methods.

C++ Classes and SIMD Operations

Use of C++ classes for SIMD operations allows for operating on arrays or vectors of data in a single operation. Consider the addition of two vectors, `A` and `B`, where each vector contains four elements. Using an integer vector class, the elements `A[i]` and `B[i]` from each array are summed as shown in the following example.

Typical Method of Adding Elements Using a Loop

```
int a[4], b[4], c[4];
for (i=0; i<4; i++) /* needs four iterations */
c[i] = a[i] + b[i]; /* computes c[0], c[1], c[2], c[3] */
```

The following example shows the same results using one operation with an integer class.

SIMD Method of Adding Elements Using Ivec Classes

```
Is16vec4 ivecA, ivecB, ivec C; /*needs one iteration*/
ivecC = ivecA + ivecB; /*computes ivecC0, ivecC1, ivecC2, ivecC3 */
```

Available Classes

The Intel® C++ SIMD classes provide parallelism, which is not easily implemented using typical mechanisms of C++. The following table shows how the Intel® C++ classes use the SIMD classes and libraries.

SIMD Vector Classes

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
MMX™ technology	`I64vec1`	unspecified	`__m64`	64	1	`ivec.h`
	`I32vec2`	unspecified	`int`	32	2	`ivec.h`
	`Is32vec2`	signed	`int`	32	2	`ivec.h`
	`Iu32vec2`	unsigned	`int`	32	2	`ivec.h`
	`I16vec4`	unspecified	`short`	16	4	`ivec.h`
	`Is16vec4`	signed	`short`	16	4	`ivec.h`
	`Iu16vec4`	unsigned	`short`	16	4	`ivec.h`

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
	I8vec8	unspecified	char	8	8	ivec.h
	Is8vec8	signed	char	8	8	ivec.h
	Iu8vec8	unsigned	char	8	8	ivec.h
Intel® SSE	F32vec4	unspecified	float	32	4	fvec.h
	F32vec1	unspecified	float	32	1	fvec.h
Intel® SSE2	F64vec2	unspecified	double	64	2	dvec.h
	I128vec1	unspecified	__m128i	128	1	dvec.h
	I64vec2	unspecified	long int	64	2	dvec.h
	I32vec4	unspecified	int	32	4	dvec.h
	Is32vec4	signed	int	32	4	dvec.h
	Iu32vec4	unsigned	int	32	4	dvec.h
	I16vec8	unspecified	int	16	8	dvec.h
	Is16vec8	signed	int	16	8	dvec.h
	Iu16vec8	unsigned	int	16	8	dvec.h
	I8vec16	unspecified	char	8	16	dvec.h
	Is8vec16	signed	char	8	16	dvec.h
	Iu8vec16	unsigned	char	8	16	dvec.h
Intel® AVX	F32vec8	unspecified	float	32	8	dvec.h
	F64vec4	unspecified	double	64	4	dvec.h
Intel® AVX-512 Foundation	F32vec16	unspecified	float	32	16	dvec.h
	F64vec8	unspecified	double	64	8	dvec.h
	M512vec	unspecified	__m512i	512	1	dvec.h
	I32vec16	unspecified	int	32	16	dvec.h
	Is32vec16	signed	int	32	16	dvec.h
	Iu32vec16	unsigned	int	32	16	dvec.h
	I64vec8	unspecified	long int	64	8	dvec.h
	Is64vec8	signed	long int	64	8	dvec.h
	Iu64vec8	unsigned	long int	64	8	dvec.h

Instruction Set	Class	Signedness	Data Type	Size	Elements	Header File
Intel® AVX-512 Byte and Word	I16vec32	unspecified	int	16	32	dvec.h
	Is16vec32	signed	int	16	32	dvec.h
	Iu16vec32	unsigned	int	16	32	dvec.h
	I8vec64	unspecified	int	8	64	dvec.h
	Is8vec64	signed	int	8	64	dvec.h
	Iu8vec64	unsigned	int	8	64	dvec.h

Most classes contain similar functionality for all data types and are represented by all available intrinsics. However, some capabilities do not translate from one data type to another without suffering from poor performance, and are therefore excluded from individual classes.

NOTE
Intrinsics that take immediate values and cannot be expressed easily in classes are not implemented. For example:

- `_mm_shuffle_ps`
- `_mm_shuffle_pi16`
- `_mm_shuffle_ps`
- `_mm_extract_pi16`
- `_mm_insert_pi16`

Access to Classes Using Header Files

The required class header files are installed in the include directory with the Intel® C++ Compiler. To enable the classes, use the `#include` directive in your program file as shown in the table that follows.

Include Directives for Enabling Classes

Instruction Set Extension	Include Directive
MMX™ Technology	`#include <ivec.h>`
Intel® SSE	`#include <fvec.h>`
Intel® SSE 2	`#include <dvec.h>`
Intel® SSE 3	`#include <dvec.h>`
Intel® SSE 4	`#include <dvec.h>`
Intel® AVX	`#include <dvec.h>`

Each succeeding file from the top down includes the preceding class. You only need to include `fvec.h` if you want to use both the `Ivec` and `Fvec` classes. Similarly, to use all the classes including those for Intel® Streaming SIMD Extensions 2, you only need to include the `dvec.h` file.

Usage Precautions

When using the C++ classes, you should follow some general guidelines. More detailed usage rules for each class are listed in Integer Vector Classes, and Floating-point Vector Classes.

Clear MMX Registers

If you use both the `Ivec` and `Fvec` classes at the same time, your program could mix Intel® MMX™ instructions, called by `Ivec` classes, with Intel® architecture floating-point instructions, called by `Fvec` classes. x87 floating-point instructions exist in the following `Fvec` functions:

- `fvec` constructors
- debug functions (`cout` and element access)
- `rsqrt_nr`

NOTE
Intel® MMX™ technology registers are aliased on the floating-point registers, so you should clear the MMX state with the EMMS instruction intrinsic before issuing an x87 floating-point instruction, as in the following example.

`ivecA = ivecA & ivecB;`	Ivec logical operation that uses MMX instructions
`empty ();`	clear state
`cout << f32vec4a;`	F32vec4 operation that uses x87 floating-point instructions

Caution
Failure to clear the Intel® MMX™ technology registers can result in incorrect execution or poor performance due to an incorrect register state.

Capabilities of C++ SIMD Classes

The fundamental capabilities of each C++ SIMD class include:

- Computation
- Horizontal data support
- Branch compression/elimination
- Caching hints

Understanding each of these capabilities and how they interact is crucial to achieving desired results.

Computation

The SIMD C++ classes contain vertical operator support for most arithmetic operations, including shifting and saturation.

Computation operations include: `+`, `-`, `*`, `/`, reciprocal (`rcp` and `rcp_nr`), square root (`sqrt`), and reciprocal square root (`rsqrt` and `rsqrt_nr`).

Operations `rcp` and `rsqrt` are approximating instructions with very short latencies that produce results with at least 12 bits of accuracy. You may get a different answer if used on non-Intel processors. Operations `rcp_nr` and `rsqrt_nr` use software refining techniques to enhance the accuracy of the approximations, with a minimal impact on performance. (The `nr` stands for Newton-Raphson, a mathematical technique for improving performance using an approximate result.)

Horizontal Data Support

The C++ SIMD classes provide horizontal support for some arithmetic operations. The term *horizontal* indicates computation across the elements of one vector, as opposed to the vertical, element-by-element operations on two different vectors.

The `add_horizontal`, `unpack_low` and `pack_sat` functions are examples of horizontal data support. This support enables certain algorithms that cannot exploit the full potential of SIMD instructions.

Shuffle intrinsics are another example of horizontal data flow. Shuffle intrinsics are not expressed in the C++ classes due to their immediate arguments. However, the C++ class implementation enables you to mix shuffle intrinsics with the other C++ functions. For example:

```
F32vec4 fveca, fvecb, fvecd;
fveca += fvecb;
fvecd = _mm_shuffle_ps(fveca,fvecb,0);
```

Branch Compression/Elimination

Branching in SIMD architectures can be complicated and expensive. The SIMD C++ classes provide functions to eliminate branches, using logical operations, max and min functions, conditional selects, and compares. Consider the following example:

```
short a[4], b[4], c[4];
for (i=0; i<4; i++)
c[i] = a[i] > b[i] ? a[i] : b[i];
```

This operation is independent of the value of `i`. For each `i`, the result could be either `A` or `B` depending on the actual values. A simple way of removing the branch altogether is to use the `select_gt` function, as follows:

```
Is16vec4 a, b, c
c = select_gt(a, b, a, b)
```

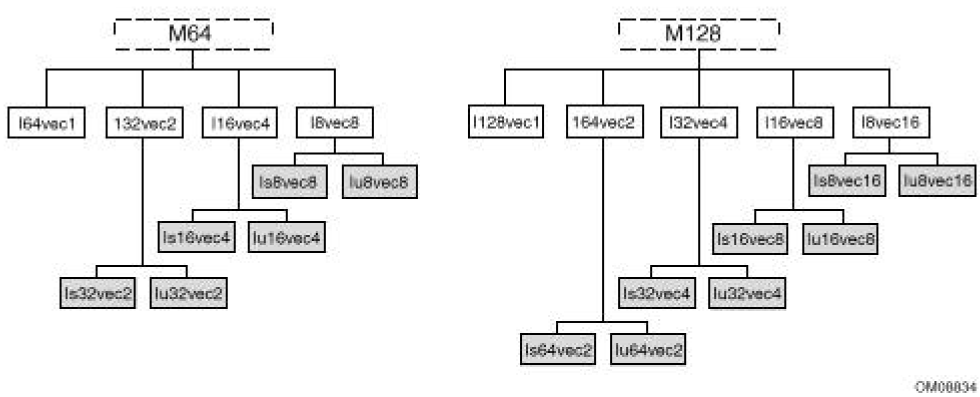
Caching Hints

Intel® Streaming SIMD Extensions provide prefetching and streaming hints. Prefetching data can minimize the effects of memory latency. Streaming hints allow you to indicate that certain data should not be cached.

Integer Vector Classes

The `Ivec` classes provide an interface to single instruction, multiple data (SIMD) processing using integer vectors of various sizes. The class hierarchy is represented in the following figure.

Ivec Class Hierarchy



The `M64` and `M128` classes define the `__m64` and `__m128i` data types from which the rest of the `Ivec` classes are derived. The first generation of child classes (the intermediate classes) are derived on element sizes of 128, 64, 32, 16, and 8 bits:

```
I128vec1, I64vec1, I64vec2, I32vec2, I32vec4, I16vec4, I16vec8, I8vec8, I8vec16
```

The second generation specify the signedness:

```
Is64vec2, Iu64vec2, Is32vec2, Iu32vec2, Is32vec4, Iu32vec4, Is16vec4, Iu16vec4,
Is16vec8, Iu16vec8, Is8vec8, Iu8vec8, Is8vec16, Iu8vec16
```

> **Caution**
> Intermixing the `M64` and `M128` data types will result in unexpected behavior.

Terms and Syntax

The following are special terms and syntax used in this chapter to describe functionality of the classes with respect to their associated operations.

Ivec Class Syntax Conventions

The name of each class denotes the data type, signedness, bit size, and number of elements using the following generic format:

```
<type><signedness><bits>vec<elements>
```

```
{ F | I } { s | u } { 128 | 64 | 32 | 16 | 8 } vec { 16 | 8 | 4 | 2 | 1 }
```

where

`type`	Indicates floating point (`F`) or integer (`I`).
`signedness`	Indicates signed (`s`) or unsigned (`u`). For the Ivec class, leaving this field blank indicates an intermediate class. For the Fvec classes, this field is blank because there are no unsigned Fvec classes.
`bits`	Specifies the number of bits per element.
`elements`	Specifies the number of elements.

Special Terms and Conventions

The following terms are used to define the functionality and characteristics of the classes and operations defined in this manual.

- **Nearest Common Ancestor**: This is the intermediate or parent class of two classes of the same size. For example, the nearest common ancestor of `Iu8vec8` and `Is8vec8` is `I8vec8`, and the nearest common ancestor between `Iu8vec8` and `I16vec4` is `M64`.
- **Casting:** Changes the data type from one class to another. When an operation uses different data types as operands, the return value of the operation must be assigned to a single data type, and one or more of the data types must be converted to a required data type. This conversion is known as a typecast. While typecasting is occasionally automatic, in cases where it is not automatic you must use special syntax to explicitly typecast it yourself.
- **Operator Overloading:** This is the ability to use various operators on the user-defined data type of a given class. In the case of the Ivec and Fvec classes, once you declare a variable, you can add, subtract, multiply, and perform a range of operations. Each family of classes accepts a specified range of operators, and must comply by rules and restrictions regarding typecasting and operator overloading as defined in the header files.

Rules for Operators

To use operators with the `Ivec` classes you must use one of the following three syntax conventions:

`[ Ivec_Class ] R = [ Ivec_Class ] A [ operator ][ Ivec_Class ] B`

Example 1:`I64vec1 R = I64vec1 A & I64vec1 B;`

`[ Ivec_Class ] R =[ operator ] ([ Ivec_Class ] A,[ Ivec_Class ] B)`

Example 2:`I64vec1 R = andnot(I64vec1 A, I64vec1 B);`

`[ Ivec_Class ] R [ operator ]= [ Ivec_Class ] A`

Example 3:`I64vec1 R &= I64vec1 A;`

[`operator`] represents an operator (for example, &, |, or ^)

[`Ivec_Class`] represents an `Ivec` class

`R`, `A`, `B` variables are declared using the pertinent `Ivec` classes

The table that follows shows automatic and explicit sign and size typecasting. "Explicit" means that it is illegal to mix different types without an explicit typecasting. "Automatic" means that you can mix types freely and the compiler will do the typecasting for you.

Summary of Rules Major Operators

Operators	Sign Typecasting	Size Typecasting	Other Typecasting Requirements
Assignment	N/A	N/A	N/A
Logical	Automatic	Automatic (to left)	Explicit typecasting is required for different types used in non-logical expressions on the right side of the assignment.
Addition and Subtraction	Automatic	Explicit	N/A
Multiplication	Automatic	Explicit	N/A
Shift	Automatic	Explicit	Casting Required to ensure arithmetic shift.
Compare	Automatic	Explicit	Explicit casting is required for signed classes for the less-than or greater-than operations.
Conditional Select	Automatic	Explicit	Explicit casting is required for signed classes for less-than or greater-than operations.

Data Declaration and Initialization

The following table shows literal examples of constructor declarations and data type initialization for all class sizes. All values are initialized with the most significant element on the left and the least significant to the right.

Declaration and Initialization Data Types for Ivec Classes

Operation	Class	Syntax
Declaration	`M128`	`I128vec1 A; Iu8vec16 A;`
Declaration	`M64`	`I64vec1 A; Iu8vec8 A;`
`__m128` Initialization	`M128`	`I128vec1 A(__m128 m);` `Iu16vec8(__m128 m);`
`__m64` Initialization	`M64`	`I64vec1 A(__m64 m);Iu8vec8 A(__m64 m);`
`__int64` Initialization	`M64`	`I64vec1 A = __int64 m;` `Iu8vec8 A =__int64 m;`
`int i` Initialization	`M64`	`I64vec1 A = int i; Iu8vec8 A = int i;`

`int` Initialization	`I32vec2`	`I32vec2 A(int A1, int A0);`
		`Is32vec2 A(signed int A1, signed int A0);`
		`Iu32vec2 A(unsigned int A1, unsigned int A0);`

`int` Initialization	`I32vec4`	`I32vec4 A(int A3, int A2, int A1, int A0);`
		`Is32vec4 A(signed int A3, ..., signed int A0);`
		`Iu32vec4 A(unsigned int A3, ..., unsigned int A0);`

`short int` Initialization	`I16vec4`	`I16vec4 A(short A3, short A2, short A1, short A0);`
		`Is16vec4 A(signed short A3, ..., signed short A0);`
		`Iu16vec4 A(unsigned short A3, ..., unsigned short A0);`

`short int` Initialization	`I16vec8`	`I16vec8 A(short A7, short A6, ..., short A1, short A0);`
		`Is16vec8 A(signed A7, ..., signed short A0);`
		`Iu16vec8 A(unsigned short A7, ..., unsigned short A0);`

`char` Initialization	`I8vec8`	`I8vec8 A(char A7, char A6, ..., char A1, char A0);`
		`Is8vec8 A(signed char A7, ..., signed char A0);`
		`Iu8vec8 A(unsigned char A7, ..., unsigned char A0);`

Operation	Class	Syntax
`char` Initialization	`I8vec16`	`I8vec16 A(char A15, ..., char A0);` `Is8vec16 A(signed char A15, ..., signed char A0);` `Iu8vec16 A(unsigned char A15, ..., unsigned char A0);`

Assignment Operator

Any `Ivec` object can be assigned to any other `Ivec` object; conversion on assignment from one `Ivec` object to another is automatic.

Assignment Operator Examples

```
Is16vec4 A;

Is8vec8 B;

I64vec1 C;

A = B; /* assign Is8vec8 to Is16vec4 */

B = C; /* assign I64vec1 to Is8vec8 */

B = A & C; /* assign M64 result of '&' to Is8vec8 */
```

Logical Operators

The logical operators use the symbols and intrinsics listed in the following table.

Bitwise Operation	Operator Symbols		Syntax Usage		Corresponding Intrinsic
	Standard	w/assign	Standard	w/assign	
AND	`&`	`&=`	`R = A & B`	`R &= A`	`_mm_and_si64` `_mm_and_si128`
OR	`|`	`|=`	`R = A | B`	`R |= A`	`_mm_and_si64` `_mm_and_si128`
XOR	^	^=	R = A^B	R ^= A	`_mm_and_si64` `_mm_and_si128`
ANDNOT	`andnot`	`N/A`	`R = A andnot B`	`N/A`	`_mm_and_si64` `_mm_and_si128`

Logical Operators and Miscellaneous Exceptions

A and B converted to M64. Result assigned to `Iu8vec8`.

```
I64vec1 A;

Is8vec8 B;

Iu8vec8 C;

C = A & B;
```

Same size and signedness operators return the nearest common ancestor.

```
I32vec2 R = Is32vec2 A ^ Iu32vec2 B;
```

A&B returns M64, which is cast to `Iu8vec8`.

```
C = Iu8vec8(A&B)+ C;
```

When `A` and `B` are of the same class, they return the same type. When `A` and `B` are of different classes, the return value is the return type of the nearest common ancestor.

The logical operator returns values for combinations of classes, listed in the following tables, apply when `A` and `B` are of different classes.

Ivec Logical Operator Overloading

Return (R)	AND	OR	XOR	NAND	A Operand	B Operand
`I64vec1 R`	`&`	`|`	`^`	`andnot`	`I[s|u]64vec2 A`	`I[s|u]64vec2 B`
`I64vec2 R`	`&`	`|`	`^`	`andnot`	`I[s|u]64vec2 A`	`I[s|u]64vec2 B`
`I32vec2 R`	`&`	`|`	`^`	`andnot`	`I[s|u]32vec2 A`	`I[s|u]32vec2 B`
`I32vec4 R`	`&`	`|`	`^`	`andnot`	`I[s|u]32vec4 A`	`I[s|u]32vec4 B`
`I16vec4 R`	`&`	`|`	`^`	`andnot`	`I[s|u]16vec4 A`	`I[s|u]16vec4 B`
`I16vec8 R`	`&`	`|`	`^`	`andnot`	`I[s|u]16vec8 A`	`I[s|u]16vec8 B`
`I8vec8 R`	`&`	`|`	`^`	`andnot`	`I[s|u]8vec8 A`	`I[s|u]8vec8 B`
`I8vec16 R`	`&`	`|`	`^`	`andnot`	`I[s|u]8vec16 A`	`I[s|u]8vec16 B`

For logical operators with assignment, the return value of `R` is always the same data type as the pre-declared value of R as listed in the table that follows.

Ivec Logical Operator Overloading with Assignment

Return Type	Left Side (R)	AND	OR	XOR	Right Side (Any Ivec Type)
`I128vec1`	`I128vec1 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`
`I64vec1`	`I64vec1 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`
`I64vec2`	`I64vec2 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`
`I[x]32vec4`	`I[x]32vec4 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`
`I[x]32vec2`	`I[x]32vec2 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`
`I[x]16vec8`	`I[x]16vec8 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`
`I[x]16vec4`	`I[x]16vec4 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`

Return Type	Left Side (R)	AND	OR	XOR	Right Side (Any Ivec Type)
`I[x]8vec16`	`I[x]8vec16 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`
`I[x]8vec8`	`I[x]8vec8 R`	`&=`	`|=`	`^=`	`I[s|u][N]vec[N] A;`

Addition and Subtraction Operators

The addition and subtraction operators return the class of the nearest common ancestor when the right-side operands are of different signs. The following code provides examples of usage and miscellaneous exceptions.

Syntax Usage for Addition and Subtraction Operators

Return nearest common ancestor type, `I16vec4`.

```
Is16vec4 A;

Iu16vec4 B;

I16vec4 C;

C = A + B;
```

Returns type left-hand operand type.

```
Is16vec4 A;

Iu16vec4 B;

A += B;

B -= A;
```

Explicitly convert `B` to `Is16vec4`.

```
Is16vec4 A,C;

Iu32vec24 B;

C = A + C;

C = A + (Is16vec4)B;
```

Addition and Subtraction Operators with Corresponding Intrinsics

Operation	Symbols	Syntax	Corresponding Intrinsics

Addition	`+`
	`+=`
		`R = A + B`
		`R += A`
			`_mm_add_epi64`
			`_mm_add_epi32`
			`_mm_add_epi16`
			`_mm_add_epi8`
			`_mm_add_pi32`
			`_mm_add_pi16`
			`_mm_add_pi8`

Subtraction	`-`
	`-=`
		`R = A - B`
		`R -= A`
			`_mm_sub_epi64`
			`_mm_sub_epi32`
			`_mm_sub_epi16`
			`_mm_sub_epi8`
			`_mm_sub_pi32`
			`_mm_sub_pi16`
			`_mm_sub_pi8`

The following table lists addition and subtraction return values for combinations of classes when the right side operands are of different signedness. The two operands must be the same size, otherwise you must explicitly indicate the typecasting.

Addition and Subtraction Operator Overloading

Return Value	Available Operators		Right Side Operands	
R	**Add**	**Sub**	**A**	**B**
`I64vec2 R`	`+`	`-`	`I[s|u]64vec2 A`	`I[s|u]64vec2 B`
`I32vec4 R`	`+`	`-`	`I[s|u]32vec4 A`	`I[s|u]32vec4 B`
`I32vec2 R`	`+`	`-`	`I[s|u]32vec2 A`	`I[s|u]32vec2 B`
`I16vec8 R`	`+`	`-`	`I[s|u]16vec8 A`	`I[s|u]16vec8 B`
`I16vec4 R`	`+`	`-`	`I[s|u]16vec4 A`	`I[s|u]16vec4 B`
`I8vec8 R`	`+`	`-`	`I[s|u]8vec8 A`	`I[s|u]8vec8 B`
`I8vec16 R`	`+`	`-`	`I[s|u]8vec2 A`	`I[s|u]8vec16 B`

The following table shows the return data type values for operands of the addition and subtraction operators with assignment. The left side operand determines the size and signedness of the return value. The right side operand must be the same size as the left operand; otherwise, you must use an explicit typecast.

Addition and Subtraction with Assignment

Return Value (R)	Left Side (R)	Add	Sub	Right Side (A)
`I[x]32vec4`	`I[x]32vec2 R`	`+=`	`-=`	`I[s|u]32vec4 A;`
`I[x]32vec2 R`	`I[x]32vec2 R`	`+=`	`-=`	`I[s|u]32vec2 A;`
`I[x]16vec8`	`I[x]16vec8`	`+=`	`-=`	`I[s|u]16vec8 A;`
`I[x]16vec4`	`I[x]16vec4`	`+=`	`-=`	`I[s|u]16vec4 A;`
`I[x]8vec16`	`I[x]8vec16`	`+=`	`-=`	`I[s|u]8vec16 A;`
`I[x]8vec8`	`I[x]8vec8`	`+=`	`-=`	`I[s|u]8vec8 A;`

Multiplication Operators

The multiplication operators can only accept and return data types from the `I[s|u]16vec4` or `I[s|u]16vec8` classes, as shown in the following example.

Syntax Usage for Multiplication Operators

Explicitly convert B to `Is16vec4`.

```
Is16vec4 A,C;

Iu32vec2 B;

C = A * C;

C = A * (Is16vec4)B;
```

Return nearest common ancestor type, `I16vec4`

```
Is16vec4 A;

Iu16vec4 B;

I16vec4 C;
```

```
C = A + B;
```

The `mul_high` and `mul_add` functions take `Is16vec4` data only.

```
Is16vec4 A,B,C,D;
C = mul_high(A,B);
D = mul_add(A,B);
```

Multiplication Operators with Corresponding Intrinsics

Symbols		Syntax Usage	Intrinsic
*	*=	R = A * B R *= A	`_mm_mullo_pi16` `_mm_mullo_epi16`
`mul_high`	N/A	`R = mul_high(A, B)`	`_mm_mulhi_pi16` `_mm_mulhi_epi16`
`mul_add`	N/A	`R = mul_high(A, B)`	`_mm_madd_pi16` `_mm_madd_epi16`

The multiplication return operators always return the nearest common ancestor as listed in the table that follows. The two operands must be 16 bits in size, otherwise you must explicitly indicate typecasting.

Multiplication Operator Overloading

R	Mul	A	B
`I16vec4 R`	`*`	`I[s|u]16vec4 A`	`I[s|u]16vec4 B`
`I16vec8 R`	`*`	`I[s|u]16vec8 A`	`I[s|u]16vec8 B`
`Is16vec4 R`	`mul_add`	`Is16vec4 A`	`Is16vec4 B`
`Is16vec8`	`mul_add`	`Is16vec8 A`	`Is16vec8 B`
`Is32vec2 R`	`mul_high`	`Is16vec4 A`	`Is16vec4 B`
`Is32vec4 R`	`mul_high`	`s16vec8 A`	`Is16vec8 B`

The following table shows the return values and data type assignments for operands of the multiplication operators with assignment. All operands must be 16 bytes in size. If the operands are not the right size, you must use an explicit typecast.

Multiplication with Assignment

Return Value (R)	Left Side (R)	Mul	Right Side (A)
`I[x]16vec8`	`I[x]16vec8`	`*=`	`I[s|u]16vec8 A;`
`I[x]16vec4`	`I[x]16vec4`	`*=`	`I[s|u]16vec4 A;`

Shift Operators

The right shift argument can be any integer or Ivec value, and is implicitly converted to a M64 data type. The first or left operand of a << can be of any type except `I[s|u]8vec[8|16]`.

Example Syntax Usage for Shift Operators

Automatic size and sign conversion.

```
Is16vec4 A,C;
Iu32vec2 B;
```

`C = A;`

A&B returns `I16vec4`, which must be cast to `Iu16vec4` to ensure logical shift, not arithmetic shift.

```
Is16vec4 A, C;
Iu16vec4 B, R;
R = (Iu16vec4)(A & B) C;
```

A&B returns `I16vec4`, which must be cast to `Is16vec4` to ensure arithmetic shift, not logical shift.

```
R = (Is16vec4)(A & B) C;
```

Shift Operators with Corresponding Intrinsics

Operation	Symbols	Syntax Usage	Intrinsic

Shift Left	`<<`
	`&=`
		`R = A << B`
		`R &= A`
			`_mm_sll_si64`
			`_mm_slli_si64`
			`_mm_sll_pi32`
			`_mm_slli_pi32`
			`_mm_sll_pi16`
			`_mm_slli_pi16`

Shift Right	`>>`	`R = A >> B`
		`R >>= A`
			`_mm_srl_si64`
			`_mm_srli_si64`
			`_mm_srl_pi32`
			`_mm_srli_pi32`
			`_mm_srl_pi16`
			`_mm_srli_pi16`
			`_mm_sra_pi32`
			`_mm_srai_pi32`
			`_mm_sra_pi16`
			`_mm_srai_pi16`

Right shift operations with signed data types use arithmetic shifts. All unsigned and intermediate classes correspond to logical shifts. The following table shows how the return type is determined by the first argument type.

Shift Operator Overloading

Option	R	Right Shift		Left Shift		A	B
Logical	`I64vec1`	>>	>>=	<<	<<=	`I64vec1 A;`	`I64vec1 B;`
Logical	`I32vec2`	>>	>>=	<<	<<=	`I32vec2 A`	`I32vec2 B;`
Arithmetic	`Is32vec2`	>>	>>=	<<	<<=	`Is32vec2 A`	`I[s|u][N]vec[N] B;`
Logical	`Iu32vec2`	>>	>>=	<<	<<=	`Iu32vec2 A`	`I[s|u][N]vec[N] B;`
Logical	`I16vec4`	>>	>>=	<<	<<=	`I16vec4 A`	`I16vec4 B`

Option	R	Right Shift		Left Shift		A	B
Arithmetic	Is16vec4	>>	>>=	<<	<<=	Is16vec4 A	I[s|u][N]vec[N] B;
Logical	Iu16vec4	>>	>>=	<<	<<=	Iu16vec4 A	I[s|u][N]vec[N] B;

Comparison Operators

The equality and inequality comparison operands can have mixed signedness, but they must be of the same size. The comparison operators for less-than and greater-than must be of the same sign and size.

Example of Syntax Usage for Comparison Operator

The nearest common ancestor is returned for compare for equal/not-equal operations.

```
Iu8vec8 A;
Is8vec8 B;
I8vec8 C;
C = cmpneq(A,B);
```

Type cast needed for different-sized elements for equal/not-equal comparisons.

```
Iu8vec8 A, C;
Is16vec4 B;
C = cmpeq(A,(Iu8vec8)B);
```

Type cast needed for sign or size differences for less-than and greater-than comparisons.

```
Iu16vec4 A;
Is16vec4 B, C;
C = cmpge((Is16vec4)A,B);
C = cmpgt(B,C);
```

Inequality Comparison Symbols and Corresponding Intrinsics

Compare For:	Operators	Syntax	Intrinsic
Equality	cmpeq	R = cmpeq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8
Inequality	cmpneq	R = cmpneq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8 _mm_andnot_si64
Greater Than	cmpgt	R = cmpgt(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8
Greater Than or Equal To	cmpge	R = cmpge(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8 _mm_andnot_si64

Compare For:	Operators	Syntax	Intrinsic

Less Than	`cmplt`	`R = cmplt(A, B)`	`_mm_cmpgt_pi32`
			`_mm_cmpgt_pi16`
			`_mm_cmpgt_pi8`

Less Than or Equal To	`cmple`	`R = cmple(A, B)`	`_mm_cmpgt_pi32`
			`_mm_cmpgt_pi16`
			`_mm_cmpgt_pi8`
			`_mm_andnot_si64`

Comparison operators have the restriction that the operands must be the size and sign as listed in the Compare Operator Overloading table.

Compare Operator Overloading

R	Comparison	A	B
`I32vec2 R`	`cmpeq` `cmpne`	`I[s|u]32vec2 B`	`I[s|u]32vec2 B`
`I16vec4 R`		`I[s|u]16vec4 B`	`I[s|u]16vec4 B`
`I8vec8 R`		`I[s|u]8vec8 B`	`I[s|u]8vec8 B`

`I32vec2 R`	`cmpgt`
	`cmpge`
	`cmplt`
	`cmple`
		`Is32vec2 B`	`Is32vec2 B`

`I16vec4 R`		`Is16vec4 B`	`Is16vec4 B`
`I8vec8 R`		`Is8vec8 B`	`Is8vec8 B`

Conditional Select Operators

For conditional select operands, the third and fourth operands determine the type returned. Third and fourth operands with same size, but different signedness, return the nearest common ancestor data type.

Conditional Select Syntax Usage

Return the nearest common ancestor data type if third and fourth operands are of the same size, but different signs.

```
I16vec4 R = select_neq(Is16vec4, Is16vec4, Is16vec4, Iu16vec4);
```

Conditional Select for Equality

```
R0 := (A0 == B0) ? C0 : D0;
R1 := (A1 == B1) ? C1 : D1;
R2 := (A2 == B2) ? C2 : D2;
R3 := (A3 == B3) ? C3 : D3;
```

Conditional Select for Inequality

```
R0 := (A0 != B0) ? C0 : D0;
R1 := (A1 != B1) ? C1 : D1;
R2 := (A2 != B2) ? C2 : D2;
R3 := (A3 != B3) ? C3 : D3;
```

Conditional Select Symbols and Corresponding Intrinsics

Conditional Select For:	Operators	Syntax	Corresponding Intrinsic	Additional Intrinsic (Applies to All)
Equality	`select_eq`	`R = select_eq(A, B, C, D)`	`_mm_cmpeq_pi32` `_mm_cmpeq_pi16` `_mm_cmpeq_pi8`	`_mm_and_si64` `_mm_or_si64` `_mm_andnot_si64`
Inequality	`select_neq`	`R = select_neq(A, B, C, D)`	`_mm_cmpeq_pi32` `_mm_cmpeq_pi16` `_mm_cmpeq_pi8`	
Greater Than	`select_gt`	`R = select_gt(A, B, C, D)`	`_mm_cmpgt_pi32` `_mm_cmpgt_pi16` `_mm_cmpgt_pi8`	
Greater Than or Equal To	`select_ge`	`R = select_gt(A, B, C, D)`	`_mm_cmpge_pi32` `_mm_cmpge_pi16` `_mm_cmpge_pi8`	
Less Than	`select_lt`	`R = select_lt(A, B, C, D)`	`_mm_cmplt_pi32` `_mm_cmplt_pi16` `_mm_cmplt_pi8`	
Less Than or Equal To	`select_le`	`R = select_le(A, B, C, D)`	`_mm_cmple_pi32` `_mm_cmple_pi16` `_mm_cmple_pi8`	

All conditional select operands must be of the same size. The return data type is the nearest common ancestor of operands `C` and `D`. For conditional select operations using greater-than or less-than operations, the first and second operands must be signed as listed in the table that follows.

Conditional Select Operator Overloading

R	Comparison	A and B	C	D
`I32vec2 R`	`select_eq` `select_ne`	`I[s|u]32vec2`	`I[s|u]32vec2`	`I[s|u]32vec2`
`I16vec4 R`			`I[s|u]16vec4`	`I[s|u]16vec4`
`I8vec8 R`			`I[s|u]8vec8`	`I[s|u]8vec8`
`I32vec2 R`	`select_gt` `select_ge` `select_lt` `select_le`	`Is32vec2`	`Is32vec2`	`Is32vec2`
`I16vec4 R`			`Is16vec4`	`Is16vec4`
`I8vec8 R`			`Is8vec8`	`Is8vec8`

The following table shows the mapping of return values from `R0` to `R7` for any number of elements. The same return value mappings also apply when there are fewer than four return values.

Conditional Select Operator Return Value Mapping

Return Value	A Operands	Available Operators						B Operands	C and D Operands
R0:=	A0	==	!=	>	>=	<	<=	B0	C0 : D0;

Return Value	A Operands	Available Operators						B Operands	C and D Operands
R1:=	A0	==	!=	>	>=	<	<=	B0	C1 : D1;
R2:=	A0	==	!=	>	>=	<	<=	B0	C2 : D2;
R3:=	A0	==	!=	>	>=	<	<=	B0	C3 : D3;
R4:=	A0	==	!=	>	>=	<	<=	B0	C4 : D4;
R5:=	A0	==	!=	>	>=	<	<=	B0	C5 : D5;
R6:=	A0	==	!=	>	>=	<	<=	B0	C6 : D6;
R7:=	A0	==	!=	>	>=	<	<=	B0	C7 : D7;

Debug Operations

The debug operations do not map to any compiler intrinsics for MMX™ instructions. They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output

The four 32-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec4 A;
```

```
cout << Iu32vec4 A;
```

```
cout << hex << Iu32vec4 A; /* print in hex format */
```

```
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The two 32-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is32vec2 A;
```

```
cout << Iu32vec2 A;
```

```
cout << hex << Iu32vec2 A; /* print in hex format */
```

```
"[1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The eight 16-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec8 A;
```

```
cout << Iu16vec8 A;
```

```
cout << hex << Iu16vec8 A; /* print in hex format */
```

```
"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The four 16-bit values of `A` are placed in the output buffer and printed in the following format (default in decimal):

```
cout << Is16vec4 A;
```

```
cout << Iu16vec4 A;
```

```
cout << hex << Iu16vec4 A; /* print in hex format */
```

```
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The sixteen 8-bit values of `A` are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec16 A; cout << Iu8vec16 A; cout << hex << Iu8vec8 A;
```

```
/* print in hex format instead of decimal*/
```

```
"[15]:A15 [14]:A14 [13]:A13 [12]:A12 [11]:A11 [10]:A10 [9]:A9 [8]:A8 [7]:A7 [6]:A6
[5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

The eight 8-bit values of `A` are placed in the output buffer and printed in the following format (default is decimal):

```
cout << Is8vec8 A; cout << Iu8vec8 A;cout << hex << Iu8vec8 A;
```

```
/* print in hex format instead of decimal*/
```

```
"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding Intrinsics: none

Element Access Operators

```
int R = Is64vec2 A[i];
```

```
unsigned int R = Iu64vec2 A[i];
```

```
int R = Is32vec4 A[i];
```

```
unsigned int R = Iu32vec4 A[i];
```

```
int R = Is32vec2 A[i];
```

```
unsigned int R = Iu32vec2 A[i];
```

```
short R = Is16vec8 A[i];
```

```
unsigned short R = Iu16vec8 A[i];
```

```
short R = Is16vec4 A[i];
```

```
unsigned short R = Iu16vec4 A[i];
```

```
signed char R = Is8vec16 A[i];
```

```
unsigned char R = Iu8vec16 A[i];
```

```
signed char R = Is8vec8 A[i];
```

```
unsigned char R = Iu8vec8 A[i];
```

Access and read element `i` of `A`. If DEBUG is enabled and the user tries to access an element outside of `A`, a diagnostic message is printed and the program aborts.

Corresponding Intrinsics: none

Element Assignment Operators

```
Is64vec2 A[i] = int R;
```

```
Is32vec4 A[i] = int R;
```

```
Iu32vec4 A[i] = unsigned int R;
```

```
Is32vec2 A[i] = int R;
```

```
Iu32vec2 A[i] = unsigned int R;
```

```
Is16vec8 A[i] = short R;
```

```
Iu16vec8 A[i] = unsigned short R;
```

```
Is16vec4 A[i] = short R;
```

```
Iu16vec4 A[i] = unsigned short R;
```

```
Is8vec16 A[i] = signed char R;
```

```
Iu8vec16 A[i] = unsigned char R;
```

```
Is8vec8 A[i] = signed char R;
```

```
Iu8vec8 A[i] = unsigned char R;
```

Assign `R` to element i of `A`. If `DEBUG` is enabled and the user tries to assign a value to an element outside of `A`, a diagnostic message is printed and the program aborts.

Corresponding Intrinsics: none

Unpack Operators

Interleave the 64-bit value from the high half of `A` with the 64-bit value from the high half of `B`.

```
I64vec2 unpack_high(I64vec2 A, I64vec2 B);
```

```
Is64vec2 unpack_high(Is64vec2 A, Is64vec2 B);
```

```
Iu64vec2 unpack_high(Iu64vec2 A, Iu64vec2 B);
```

```
R0 = A1;
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_epi64`

Interleave the two 32-bit values from the high half of `A` with the two 32-bit values from the high half of `B`.

```
I32vec4 unpack_high(I32vec4 A, I32vec4 B);
```

```
Is32vec4 unpack_high(Is32vec4 A, Is32vec4 B);
```

```
Iu32vec4 unpack_high(Iu32vec4 A, Iu32vec4 B);
```

```
R0 = A1;
R1 = B1;
R2 = A2;
R3 = B2;
```

Corresponding intrinsic: `_mm_unpackhi_epi32`

Interleave the 32-bit value from the high half of `A` with the 32-bit value from the high half of `B`.

```
I32vec2 unpack_high(I32vec2 A, I32vec2 B);
```

```
Is32vec2 unpack_high(Is32vec2 A, Is32vec2 B);
```

```
Iu32vec2 unpack_high(Iu32vec2 A, Iu32vec2 B);
```

```
R0 = A1;
R1 = B1;
```

Corresponding intrinsic: `_mm_unpackhi_pi32`

Interleave the four 16-bit values from the high half of `A` with the two 16-bit values from the high half of `B`.

```
I16vec8 unpack_high(I16vec8 A, I16vec8 B);
```

```
Is16vec8 unpack_high(Is16vec8 A, Is16vec8 B);
```

```
Iu16vec8 unpack_high(Iu16vec8 A, Iu16vec8 B);
```

```
R0 = A2;
R1 = B2;
R2 = A3;
R3 = B3;
```

Corresponding intrinsic: `_mm_unpackhi_epi16`

Interleave the two 16-bit values from the high half of `A` with the two 16-bit values from the high half of `B`.

```
I16vec4 unpack_high(I16vec4 A, I16vec4 B);
```

```
Is16vec4 unpack_high(Is16vec4 A, Is16vec4 B);
```

```
Iu16vec4 unpack_high(Iu16vec4 A, Iu16vec4 B);
```

```
R0 = A2;R1 = B2;
R2 = A3;R3 = B3;
```

Corresponding intrinsic: `_mm_unpackhi_pi16`

Interleave the four 8-bit values from the high half of `A` with the four 8-bit values from the high half of `B`.

```
I8vec8 unpack_high(I8vec8 A, I8vec8 B);
```

```
Is8vec8 unpack_high(Is8vec8 A, I8vec8 B);
```

```
Iu8vec8 unpack_high(Iu8vec8 A, I8vec8 B);
```

```
R0 = A4;
R1 = B4;
R2 = A5;
R3 = B5;
R4 = A6;
R5 = B6;
R6 = A7;
R7 = B7;
```

Corresponding intrinsic: `_mm_unpackhi_pi8`

Interleave the sixteen 8-bit values from the high half of `A` with the four 8-bit values from the high half of `B`.

```
I8vec16 unpack_high(I8vec16 A, I8vec16 B);
```

```
Is8vec16 unpack_high(Is8vec16 A, I8vec16 B);
```

```
Iu8vec16 unpack_high(Iu8vec16 A, I8vec16 B);
```

```
R0 = A8;
R1 = B8;
R2 = A9;
```

```
R3 = B9;
R4 = A10;
R5 = B10;
R6 = A11;
R7 = B11;
R8 = A12;
R8 = B12;
R2 = A13;
R3 = B13;
R4 = A14;
R5 = B14;
R6 = A15;
R7 = B15;
```

Corresponding intrinsic: `_mm_unpackhi_epi16`

Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`

```
R0 = A0;
R1 = B0;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the 64-bit value from the low half of `A` with the 64-bit values from the low half of `B`

```
I64vec2 unpack_low(I64vec2 A, I64vec2 B);
```

```
Is64vec2 unpack_low(Is64vec2 A, Is64vec2 B);
```

```
Iu64vec2 unpack_low(Iu64vec2 A, Iu64vec2 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the two 32-bit values from the low half of `A` with the two 32-bit values from the low half of `B`

```
I32vec4 unpack_low(I32vec4 A, I32vec4 B);
```

```
Is32vec4 unpack_low(Is32vec4 A, Is32vec4 B);
```

```
Iu32vec4 unpack_low(Iu32vec4 A, Iu32vec4 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_epi32`

Interleave the 32-bit value from the low half of `A` with the 32-bit value from the low half of `B`.

```
I32vec2 unpack_low(I32vec2 A, I32vec2 B);
```

```
Is32vec2 unpack_low(Is32vec2 A, Is32vec2 B);
```

```
Iu32vec2 unpack_low(Iu32vec2 A, Iu32vec2 B);
```

```
R0 = A0;
R1 = B0;
```

Corresponding intrinsic: `_mm_unpacklo_pi32`

Interleave the two 16-bit values from the low half of `A` with the two 16-bit values from the low half of `B`.

```
I16vec8 unpack_low(I16vec8 A, I16vec8 B);
```

```
Is16vec8 unpack_low(Is16vec8 A, Is16vec8 B);
```

```
Iu16vec8 unpack_low(Iu16vec8 A, Iu16vec8 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_epi16`

Interleave the two 16-bit values from the low half of `A` with the two 16-bit values from the low half of `B`.

```
I16vec4 unpack_low(I16vec4 A, I16vec4 B);
```

```
Is16vec4 unpack_low(Is16vec4 A, Is16vec4 B);
```

```
Iu16vec4 unpack_low(Iu16vec4 A, Iu16vec4 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

Corresponding intrinsic: `_mm_unpacklo_pi16`

Interleave the four 8-bit values from the high low of `A` with the four 8-bit values from the low half of `B`.

```
I8vec16 unpack_low(I8vec16 A, I8vec16 B);
```

```
Is8vec16 unpack_low(Is8vec16 A, Is8vec16 B);
```

```
Iu8vec16 unpack_low(Iu8vec16 A, Iu8vec16 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
R8 = A4;
R9 = B4;
R10 = A5;
R11 = B5;
R12 = A6;
R13 = B6;
R14 = A7;
R15 = B7;
```

Corresponding intrinsic: `_mm_unpacklo_epi8`

Interleave the four 8-bit values from the high low of `A` with the four 8-bit values from the low half of `B`.

```
I8vec8 unpack_low(I8vec8 A, I8vec8 B);
```

```
Is8vec8 unpack_low(Is8vec8 A, Is8vec8 B);
```

```
Iu8vec8 unpack_low(Iu8vec8 A, Iu8vec8 B);
```

```
R0 = A0;
R1 = B0;
```

```
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```

Corresponding intrinsic: `_mm_unpacklo_pi8`

Pack Operators

Pack the eight 32-bit values found in `A` and `B` into eight 16-bit values with signed saturation.

`Is16vec8 pack_sat(Is32vec2 A,Is32vec2 B);`
Corresponding intrinsic: `_mm_packs_epi32`

Pack the four 32-bit values found in `A` and `B` into eight 16-bit values with signed saturation.

`Is16vec4 pack_sat(Is32vec2 A,Is32vec2 B);`
Corresponding intrinsic: `_mm_packs_pi32`

Pack the sixteen 16-bit values found in `A` and `B` into sixteen 8-bit values with signed saturation.

`Is8vec16 pack_sat(Is16vec4 A,Is16vec4 B);`
Corresponding intrinsic: `_mm_packs_epi16`

Pack the eight 16-bit values found in `A` and `B` into eight 8-bit values with signed saturation.

`Is8vec8 pack_sat(Is16vec4 A,Is16vec4 B);`
Corresponding intrinsic: `_mm_packs_pi16`

Pack the sixteen 16-bit values found in `A` and `B` into sixteen 8-bit values with unsigned saturation.

`Iu8vec16 packu_sat(Is16vec4 A,Is16vec4 B);`
Corresponding intrinsic: `_mm_packus_epi16`

Pack the eight 16-bit values found in `A` and `B` into eight 8-bit values with unsigned saturation.

`Iu8vec8 packu_sat(Is16vec4 A,Is16vec4 B);`
Corresponding intrinsic: `_mm_packs_pu16`

Clear MMX™ State Operator

Empty the MMX™ registers and clear the MMX state. Read the guidelines for using the EMMS instruction intrinsic.

`void empty(void);`
Corresponding intrinsic: `_mm_empty`

Integer Functions for Streaming SIMD Extensions

NOTE
You must include `fvec.h` header file for the following functionality.

Compute the element-wise maximum of the respective signed integer words in `A` and `B`.

`Is16vec4 simd_max(Is16vec4 A, Is16vec4 B);`
Corresponding intrinsic: `_mm_max_pi16`

Compute the element-wise minimum of the respective signed integer words in `A` and `B`.

`Is16vec4 simd_min(Is16vec4 A, Is16vec4 B);`
Corresponding intrinsic: `_mm_min_pi16`

Compute the element-wise maximum of the respective unsigned bytes in `A` and `B`.

```
Iu8vec8 simd_max(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: `_mm_max_pu8`

Compute the element-wise minimum of the respective unsigned bytes in `A` and `B`.

```
Iu8vec8 simd_min(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: `_mm_min_pu8`

Create an 8-bit mask from the most significant bits of the bytes in `A`.

```
int move_mask(I8vec8 A);
```
Corresponding intrinsic: `_mm_movemask_pi8`

Conditionally store byte elements of `A` to address `p`. The high bit of each byte in the selector `B` determines whether the corresponding byte in `A` will be stored.

```
void mask_move(I8vec8 A, I8vec8 B, signed char *p);
```
Corresponding intrinsic: `_mm_maskmove_si64`

Store the data in `A` to the address `p` without polluting the caches. `A` can be any `Ivec` type.

```
void store_nta(__m64 *p, M64 A);
```
Corresponding intrinsic: `_mm_stream_pi`

Compute the element-wise average of the respective unsigned 8-bit integers in `A` and `B`.

```
Iu8vec8 simd_avg(Iu8vec8 A, Iu8vec8 B);
```
Corresponding intrinsic: `_mm_avg_pu8`

Compute the element-wise average of the respective unsigned 16-bit integers in `A` and `B`.

```
Iu16vec4 simd_avg(Iu16vec4 A, Iu16vec4 B);
```
Corresponding intrinsic: `_mm_avg_pu16`

Conversions between Fvec and Ivec

Convert the lower double-precision floating-point value of `A` to a 32-bit integer with truncation.

```
int F64vec2ToInt(F64vec42 A);
r := (int)A0;
```

Convert the four floating-point values of `A` to two the two least significant double-precision floating-point values.

```
F64vec2 F32vec4ToF64vec2(F32vec4 A);
r0 := (double)A0;
r1 := (double)A1;
```

Convert the two double-precision floating-point values of `A` to two single-precision floating-point values.

```
F32vec4 F64vec2ToF32vec4(F64vec2 A);
r0 := (float)A0;
r1 := (float)A1;
```

Convert the signed `int` in `B` to a double-precision floating-point value and pass the upper double-precision value from `A` through to the result.

```
F64vec2 InttoF64vec2(F64vec2 A, int B);
r0 := (double)B;
r1 := A1;
```

Convert the lower floating-point value of `A` to a 32-bit integer with truncation.

```
int F32vec4ToInt(F32vec4 A);
r := (int)A0;
```

Convert the two lower floating-point values of `A` to two 32-bit integer with truncation, returning the integers in packed form.

```
Is32vec2 F32vec4ToIs32vec2 (F32vec4 A);
r0 := (int)A0;
r1 := (int)A1;
```

Convert the 32-bit integer value `B` to a floating-point value; the upper three floating-point values are passed through from `A`.

```
F32vec4 IntToF32vec4(F32vec4 A, int B);
r0 := (float)B;
r1 := A1;
r2 := A2;
r3 := A3;
```

Convert the two 32-bit integer values in packed form in `B` to two floating-point values; the upper two floating-point values are passed through from `A`.

```
F32vec4 Is32vec2ToF32vec4(F32vec4 A, Is32vec2 B);
r0 := (float)B0;
r1 := (float)B1;
r2 := A2;
r3 := A3;
```

Floating-point Vector Classes

The floating-point vector classes, `F64vec2`, `F32vec4`, and `F32vec1`, provide an interface to SIMD operations. The class specifications are as follows:

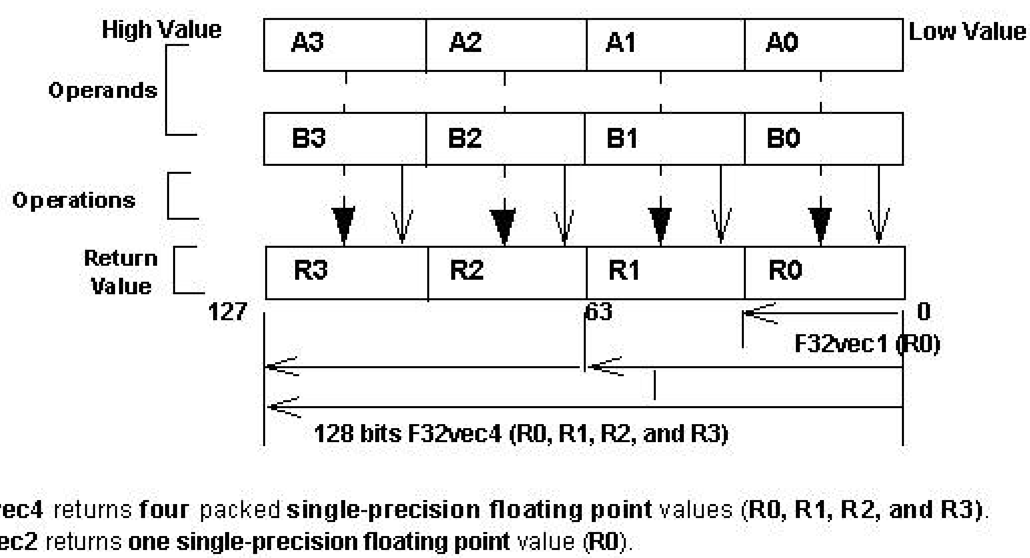
```
F64vec2 A(double x, double y);
```

```
F32vec4 A(float z, float y, float x, float w);
```

```
F32vec1 B(float w);
```

The packed floating-point input values are represented with the right-most value lowest as shown in the following table.

Single-Precision Floating-point Elements



Fvec Syntax and Notation

This reference uses the following conventions for syntax and return values.

Fvec Classes Syntax Notation

Fvec classes use the syntax conventions shown the following examples:

`[Fvec_Class] R = [Fvec_Class] A [operator][Ivec_Class] B;`

Example 1:`F64vec2 R = F64vec2 A & F64vec2 B;`

`[Fvec_Class] R = [operator]([Fvec_Class] A,[Fvec_Class] B);`

Example 2:`F64vec2 R = andnot(F64vec2 A, F64vec2 B);`

`[Fvec_Class] R [operator]= [Fvec_Class] A;`

Example 3:`F64vec2 R &= F64vec2 A;`

where

`[operator]` is an operator (for example, `&`, `|`, `or ^`)

`[Fvec_Class]` is any `Fvec` class (`F64vec2`, `F32vec4`, or `F32vec1`)

`R`, `A`, `B` are declared `Fvec` variables of the type indicated.

Return Value Notation

Because the `Fvec` classes have packed elements, the return values typically follow the conventions presented in the Return Value Convention Notation Mappings table. `F32vec4` returns four single-precision, floating-point values (R0, R1, R2, and R3); `F64vec2` returns two double-precision, floating-point values, and `F32vec1` returns the lowest single-precision floating-point value (R0).

Return Value Convention Notation Mappings

Example 1:	Example 2:	Example 3:	F32vec 4	F64vec 2	F32vec 1
`R0 := A0 & B0;`	`R0 := A0 andnot B0;`	`R0 &= A0;`	x	x	x
`R1 := A1 & B1;`	`R1 := A1 andnot B1;`	`R1 &= A1;`	x	x	N/A
`R2 := A2 & B2;`	`R2 := A2 andnot B2;`	`R2 &= A2;`	x	N/A	N/A
`R3 := A3 & B3`	`R3 := A3 andhot B3;`	`R3 &= A3;`	x	N/A	N/A

Data Alignment

Memory operations using the Intel® Streaming SIMD Extensions should be performed on 16-byte-aligned data whenever possible. Memory operations using the Intel® Advanced Vector Extensions should be performed on 32-byte-aligned data whenever possible.

`F32vec4` and `F64vec2` object variables are properly aligned by default. Note that floating point arrays are not automatically aligned. To get 16-byte alignment, you can use the alignment `__declspec`:

```
__declspec( align(16) ) float A[4];
```

Conversions

All `Fvec` object variables can be implicitly converted to `__m128` data types. For example, the results of computations performed on `F32vec4` or `F32vec1` object variables can be assigned to `__m128` data types.

```
__m128d mm = A & B; /* where A,B are F64vec2 object variables */
__m128 mm = A & B; /* where A,B are F32vec4 object variables */
__m128 mm = A & B; /* where A,B are F32vec1 object variables */
```

Constructors and Initialization

The following table shows how to create and initialize F32vec objects with the Fvec classes.

Constructors and Initialization for Fvec Classes

Example	Intrinsic	Returns
Constructor Declaration		

`F64vec2 A;`
`F32vec4 B;`
`F32vec1 C;`
	N/A	N/A

__m128 Object Initialization		

`F64vec2 A(__m128d mm);`
`F32vec4 B(__m128 mm);`
`F32vec1 C(__m128 mm);`
	N/A	N/A

Double Initialization		

`/* Initializes two doubles. */`
`F64vec2 A(double d0, double d1);`
`F64vec2 A = F64vec2(double d0, double d1);`
	`_mm_set_pd`	A0 := d0;
		A1 := d1;

`F64vec2 A(double d0);`
`/* Initializes both return values`
`with the same double precision value */.`
	`_mm_set1_pd`	A0 := d0;
		A1 := d0;

Float Initialization		

`F32vec4 A(float f3, float f2,`
`float f1, float f0);`
`F32vec4 A = F32vec4(float f3, float f2,`
`float f1, float f0);`
	`_mm_set_ps`	A0 := f0;
		A1 := f1;
		A2 := f2;
		A3 := f3;

`F32vec4 A(float f0);`
`/* Initializes all return values`
`with the same floating point value. */`
	`_mm_set1_ps`	A0 := f0;
		A1 := f0;
		A2 := f0;
		A3 := f0;

Float Initialization		
`F32vec4 A(double d0);` `/* Initialize all return values with the same double-precision value. */`	`_mm_set1_ps(d)`	A0 := d0; A1 := d0; A2 := d0; A3 := d0;
`F32vec1 A(double d0);` `/* Initializes the lowest value of A with d0 and the other values with 0.*/`	`_mm_set_ss(d)`	A0 := d0; A1 := 0; A2 := 0; A3 := 0;
`F32vec1 B(float f0);` `/* Initializes the lowest value of B with f0 and the other values with 0.*/`	`_mm_set_ss`	B0 := f0; B1 := 0; B2 := 0; B3 := 0;
`F32vec1 B(int I);` `/* Initializes the lowest value of B with f0, other values are undefined.*/`	`_mm_cvtsi32_ss`	B0 := f0; B1 := {} B2 := {} B3 := {}

Arithmetic Operators

The following table lists the arithmetic operators of the `Fvec` classes and generic syntax. The operators have been divided into standard and advanced operations, which are described in more detail later in this section.

Fvec Arithmetic Operators

Category	Operation	Operators	Generic Syntax
Standard	Addition	`+` `+=`	`R = A + B;` `R += A;`
	Subtraction	`-` `-=`	`R = A - B;` `R -= A;`
	Multiplication	`*` `*=`	`R = A * B;` `R *= A;`
	Division	`/` `/=`	`R = A / B;` `R /= A;`
Advanced	Square Root	`sqrt`	`R = sqrt(A);`
	Reciprocal (Newton-Raphson)	`rcp` `rcp_nr`	`R = rcp(A);` `R = rcp_nr(A);`
	Reciprocal Square Root (Newton-Raphson)	`rsqrt` `rsqrt_nr`	`R = rsqrt(A);` `R = rsqrt_nr(A);`

Standard Arithmetic Operator Usage

The following two tables show the return values for each class of the standard arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

Standard Arithmetic Return Value Mapping

R	A	Operators				B	F32vec 4	F64vec 2	F32vec 1
R0:=	A0	+	-	*	/	B0	X	X	X
R1:=	A1	+	-	*	/	B1	X	X	N/A
R2:=	A2	+	-	*	/	B2	X	N/A	N/A
R3:=	A3	+	-	*	/	B3	X	N/A	N/A

Arithmetic with Assignment Return Value Mapping

R	Operators				A	F32vec4	F64vec2	F32vec1
R0:=	+=	-=	*=	/=	A0	X	X	X
R1:=	+=	-=	*=	/=	A1	X	X	N/A
R2:=	+=	-=	*=	/=	A2	X	N/A	N/A
R3:=	+=	-=	*=	/=	A3	X	N/A	N/A

This table lists standard arithmetic operator syntax and intrinsics.

Standard Arithmetic Operations for Fvec Classes

Operation	Returns	Example Syntax Usage	Intrinsic
Addition	4 floats	`F32vec4 R = F32vec4 A + F32vec4 B;` `F32vec4 R += F32vec4 A;`	`_mm_add_ps`
	2 doubles	`F64vec2 R = F64vec2 A + F32vec2 B;` `F64vec2 R += F64vec2 A;`	`_mm_add_pd`
	1 float	`F32vec1 R = F32vec1 A + F32vec1 B;` `F32vec1 R += F32vec1 A;`	`_mm_add_ss`
Subtraction	4 floats	`F32vec4 R = F32vec4 A - F32vec4 B;` `F32vec4 R -= F32vec4 A;`	`_mm_sub_ps`

Operation	Returns	Example Syntax Usage	Intrinsic
	2 doubles	`F64vec2 R - F64vec2 A + F32vec2 B;` `F64vec2 R -= F64vec2 A;`	`_mm_sub_pd`
	1 float	`F32vec1 R = F32vec1 A - F32vec1 B;` `F32vec1 R -= F32vec1 A;`	`_mm_sub_ss`
Multiplication	4 floats	`F32vec4 R = F32vec4 A * F32vec4 B;` `F32vec4 R *= F32vec4 A;`	`_mm_mul_ps`
	2 doubles	`F64vec2 R = F64vec2 A * F364vec2 B;` `F64vec2 R *= F64vec2 A;`	`_mm_mul_pd`
	1 float	`F32vec1 R = F32vec1 A * F32vec1 B;` `F32vec1 R *= F32vec1 A;`	`_mm_mul_ss`
Division	4 floats	`F32vec4 R = F32vec4 A / F32vec4 B;` `F32vec4 R /= F32vec4 A;`	`_mm_div_ps`
	2 doubles	`F64vec2 R = F64vec2 A / F64vec2 B;` `F64vec2 R /= F64vec2 A;`	`_mm_div_pd`
	1 float	`F32vec1 R = F32vec1 A / F32vec1 B;` `F32vec1 R /= F32vec1 A;`	`_mm_div_ss`

Advanced Arithmetic Operator Usage

The following table shows the return values classes of the advanced arithmetic operators, which use the syntax styles described earlier in the Return Value Notation section.

Advanced Arithmetic Return Value Mapping

R	Operators					A	F32vec4	F64vec2	F32vec1
R0:=	`sqrt`	`rcp`	`rsqrt`	`rcp_nr`	`rsqrt_nr`	A0	X	X	X

R	Operators					A	F32vec4	F64vec2	F32vec1
R1:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A1	X	X	N/A
R2:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A2	X	N/A	N/A
R3:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A3	X	N/A	N/A
f :=	add_horizontal			(A0 + A1 + A2 + A3)			X	N/A	N/A
d :=	add_horizontal			(A0 + A1)			N/A	X	N/A

This table shows examples for advanced arithmetic operators.

Advanced Arithmetic Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Square Root		
4 floats	`F32vec4 R = sqrt(F32vec4 A);`	`_mm_sqrt_ps`
2 doubles	`F64vec2 R = sqrt(F64vec2 A);`	`_mm_sqrt_pd`
1 float	`F32vec1 R = sqrt(F32vec1 A);`	`_mm_sqrt_ss`
Reciprocal		
4 floats	`F32vec4 R = rcp(F32vec4 A);`	`_mm_rcp_ps`
2 doubles	`F64vec2 R = rcp(F64vec2 A);`	`_mm_rcp_pd`
1 float	`F32vec1 R = rcp(F32vec1 A);`	`_mm_rcp_ss`
Reciprocal Square Root		
4 floats	`F32vec4 R = rsqrt(F32vec4 A);`	`_mm_rsqrt_ps`
2 doubles	`F64vec2 R = rsqrt(F64vec2 A);`	`_mm_rsqrt_pd`
1 float	`F32vec1 R = rsqrt(F32vec1 A);`	`_mm_rsqrt_ss`

Reciprocal Newton Raphson		
4 floats	`F32vec4 R = rcp_nr(F32vec4 A);`	`_mm_sub_ps` `_mm_add_ps` `_mm_mul_ps` `_mm_rcp_ps`
2 doubles	`F64vec2 R = rcp_nr(F64vec2 A);`	`_mm_sub_pd` `_mm_add_pd` `_mm_mul_pd` `_mm_rcp_pd`
1 float	`F32vec1 R = rcp_nr(F32vec1 A);`	`_mm_sub_ss` `_mm_add_ss` `_mm_mul_ss` `_mm_rcp_ss`

Reciprocal Square Root Newton Raphson		
4 float	`F32vec4 R = rsqrt_nr(F32vec4 A);`	`_mm_sub_pd` `_mm_mul_pd` `_mm_rsqrt_ps`
2 doubles	`F64vec2 R = rsqrt_nr(F64vec2 A);`	`_mm_sub_pd` `_mm_mul_pd` `_mm_rsqrt_pd`
1 float	`F32vec1 R = rsqrt_nr(F32vec1 A);`	`_mm_sub_ss` `_mm_mul_ss` `_mm_rsqrt_ss`

Horizontal Add		
1 float	`float f = add_horizontal(F32vec4 A);`	`_mm_add_ss` `_mm_shuffle_ss`
1 double	`double d = add_horizontal(F64vec2 A);`	`_mm_add_sd` `_mm_shuffle_sd`

Minimum and Maximum Operators

Compute the minimums of the two double precision floating-point values of `A` and `B`.

```
F64vec2 R = simd_min(F64vec2 A, F64vec2 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
```

Corresponding intrinsic: `_mm_min_pd`

Compute the minimums of the four single precision floating-point values of `A` and `B`.

```
F32vec4 R = simd_min(F32vec4 A, F32vec4 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
R2 := min(A2,B2);
R3 := min(A3,B3);
```

Corresponding intrinsic: `_mm_min_ps`

Compute the minimum of the lowest single precision floating-point values of `A` and `B`.

```
F32vec1 R = simd_min(F32vec1 A, F32vec1 B)
R0 := min(A0,B0);
```
Corresponding intrinsic: `_mm_min_ss`

Compute the maximums of the two double precision floating-point values of `A` and `B`.

```
F64vec2 simd_max(F64vec2 A, F64vec2 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
```
Corresponding intrinsic: `_mm_max_pd`

Compute the maximums of the four single precision floating-point values of `A` and `B`.

```
F32vec4 R = simd_man(F32vec4 A, F32vec4 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
R2 := max(A2,B2);
R3 := max(A3,B3);
```
Corresponding intrinsic: `_mm_max_ps`

Compute the maximum of the lowest single precision floating-point values of `A` and `B`.

```
F32vec1 simd_max(F32vec1 A, F32vec1 B)
R0 := max(A0,B0);
```
Corresponding intrinsic: `_mm_max_ss`

Logical Operators

The following table lists the logical operators of the Fvec classes and generic syntax. The logical operators for `F32vec1` classes use only the lower 32 bits.

Fvec Logical Operators Return Value Mapping

Bitwise Operation	Operators	Generic Syntax
AND	`&` `&=`	`R = A & B;` `R &= A;`
OR	`|` `|=`	`R = A | B;` `R |= A;`
XOR	`^` `^=`	`R = A ^ B;` `R ^= A;`
andnot	`andnot`	`R = andnot(A);`

The following table lists standard logical operators syntax and corresponding intrinsics. Note that there is no corresponding scalar intrinsic for the `F32vec1` classes, which accesses the lower 32 bits of the packed vector intrinsics.

Logical Operations for Fvec Classes

Operation	Returns	Example Syntax Usage	Intrinsic
AND	4 floats	`F32vec4 & = F32vec4 A & F32vec4 B;` `F32vec4 & &= F32vec4 A;`	`_mm_and_ps`
	2 doubles	`F64vec2 R = F64vec2 A & F64vec2 B;`	`_mm_and_pd`

Operation	Returns	Example Syntax Usage	Intrinsic
		F64vec2 R &= F64vec2 A;	
	1 float	F32vec1 R = F32vec1 A & F32vec1 B; F32vec1 R &= F32vec1 A;	_mm_and_ps
OR	4 floats	F32vec4 R = F32vec4 A | F32vec4 B; F32vec4 R |= F32vec4 A;	_mm_or_ps
	2 doubles	F64vec2 R = F64vec2 A | F64vec2 B; F64vec2 R |= F64vec2 A;	_mm_or_pd
	1 float	F32vec1 R = F32vec1 A | F32vec1 B; F32vec1 R |= F32vec1 A;	_mm_or_ps
XOR	4 floats	F32vec4 R = F32vec4 A ^ F32vec4 B; F32vec4 R ^= F32vec4 A;	_mm_xor_ps
	2 doubles	F64vec2 R = F64vec2 A ^ F64vec2 B; F64vec2 R ^= F64vec2 A;	_mm_xor_pd
	1 float	F32vec1 R = F32vec1 A ^ F32vec1 B; F32vec1 R ^= F32vec1 A;	_mm_xor_ps
ANDNOT	2 doubles	F64vec2 R = andnot(F64vec2 A, F64vec2 B);	_mm_andnot_pd

Compare Operators

The operators described in this section compare the single precision floating-point values of A and B. Comparison between objects of any Fvec class return the same class being compared.

The following table lists the compare operators for the Fvec classes.

Compare Operators and Corresponding Intrinsics

Compare For:	Operators	Syntax
Equality	cmpeq	R = cmpeq(A, B)
Inequality	cmpneq	R = cmpneq(A, B)

Compare For:	Operators	Syntax
Greater Than	`cmpgt`	`R = cmpgt(A, B)`
Greater Than or Equal To	`cmpge`	`R = cmpge(A, B)`
Not Greater Than	`cmpngt`	`R = cmpngt(A, B)`
Not Greater Than or Equal To	`cmpnge`	`R = cmpnge(A, B)`
Less Than	`cmplt`	`R = cmplt(A, B)`
Less Than or Equal To	`cmple`	`R = cmple(A, B)`
Not Less Than	`cmpnlt`	`R = cmpnlt(A, B)`
Not Less Than or Equal To	`cmpnle`	`R = cmpnle(A, B)`

Compare Operators

The mask is set to `0xffffffff` for each floating-point value where the comparison is true and `0x00000000` where the comparison is false. The following table shows the return values for each class of the compare operators, which use the syntax described earlier in the Return Value Notation section.

Compare Operator Return Value Mapping

R	A0	For Any Operators	B	If True	If False	F32vec 4	F64vec 2	F32vec 1
R0 :=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B1) B1)	0xffffffff	0x0000000	X	X	X
R1 :=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B2) B2)	0xffffffff	0x0000000	X	X	N/A
R2 :=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A
R3 :=	A3	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A

The following table shows examples for arithmetic operators and intrinsics.

Compare Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Compare for Equality		
4 floats	`F32vec4 R = cmpeq(F32vec4 A);`	`_mm_cmpeq_ps`
2 doubles	`F64vec2 R = cmpeq(F64vec2 A);`	`_mm_cmpeq_pd`
1 float	`F32vec1 R = cmpeq(F32vec1 A);`	`_mm_cmpeq_ss`
Compare for Inequality		
4 floats	`F32vec4 R = cmpneq(F32vec4 A);`	`_mm_cmpneq_ps`
2 doubles	`F64vec2 R = cmpneq(F64vec2 A);`	`_mm_cmpneq_pd`
1 float	`F32vec1 R = cmpneq(F32vec1 A);`	`_mm_cmpneq_ss`
Compare for Less Than		
4 floats	`F32vec4 R = cmplt(F32vec4 A);`	`_mm_cmplt_ps`
2 doubles	`F64vec2 R = cmplt(F64vec2 A);`	`_mm_cmplt_pd`
1 float	`F32vec1 R = cmplt(F32vec1 A);`	`_mm_cmplt_ss`
Compare for Less Than or Equal		
4 floats	`F32vec4 R = cmple(F32vec4 A);`	`_mm_cmple_ps`
2 doubles	`F64vec2 R = cmple(F64vec2 A);`	`_mm_cmple_pd`
1 float	`F32vec1 R = cmple(F32vec1 A);`	`_mm_cmple_pd`
Compare for Greater Than		
4 floats	`F32vec4 R = cmpgt(F32vec4 A);`	`_mm_cmpgt_ps`
2 doubles	`F64vec2 R = cmpgt(F32vec42 A);`	`_mm_cmpgt_pd`
1 float	`F32vec1 R = cmpgt(F32vec1 A);`	`_mm_cmpgt_ss`

Compare for Greater Than or Equal To		
4 floats	`F32vec4 R = cmpge(F32vec4 A);`	`_mm_cmpge_ps`
2 doubles	`F64vec2 R = cmpge(F64vec2 A);`	`_mm_cmpge_pd`
1 float	`F32vec1 R = cmpge(F32vec1 A);`	`_mm_cmpge_ss`

Compare for Not Less Than		
4 floats	`F32vec4 R = cmpnlt(F32vec4 A);`	`_mm_cmpnlt_ps`
2 doubles	`F64vec2 R = cmpnlt(F64vec2 A);`	`_mm_cmpnlt_pd`
1 float	`F32vec1 R = cmpnlt(F32vec1 A);`	`_mm_cmpnlt_ss`

Compare for Not Less Than or Equal		
4 floats	`F32vec4 R = cmpnle(F32vec4 A);`	`_mm_cmpnle_ps`
2 doubles	`F64vec2 R = cmpnle(F64vec2 A);`	`_mm_cmpnle_pd`
1 float	`F32vec1 R = cmpnle(F32vec1 A);`	`_mm_cmpnle_ss`

Compare for Not Greater Than		
4 floats	`F32vec4 R = cmpngt(F32vec4 A);`	`_mm_cmpngt_ps`
2 doubles	`F64vec2 R = cmpngt(F64vec2 A);`	`_mm_cmpngt_pd`
1 float	`F32vec1 R = cmpngt(F32vec1 A);`	`_mm_cmpngt_ss`

Compare for Not Greater Than or Equal		
4 floats	`F32vec4 R = cmpnge(F32vec4 A);`	`_mm_cmpnge_ps`
2 doubles	`F64vec2 R = cmpnge(F64vec2 A);`	`_mm_cmpnge_pd`
1 float	`F32vec1 R = cmpnge(F32vec1 A);`	`_mm_cmpnge_ss`

Conditional Select Operators for Fvec Classes

Each conditional function compares single-precision floating-point values of A and B. The C and D parameters are used for return value. Comparison between objects of any Fvec class returns the same class.

Conditional Select Operators for Fvec Classes

Conditional Select for:	Operators	Syntax
Equality	`select_eq`	`R = select_eq(A, B)`
Inequality	`select_neq`	`R = select_neq(A, B)`
Greater Than	`select_gt`	`R = select_gt(A, B)`
Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Not Greater Than	`select_gt`	`R = select_gt(A, B)`
Not Greater Than or Equal To	`select_ge`	`R = select_ge(A, B)`
Less Than	`select_lt`	`R = select_lt(A, B)`
Less Than or Equal To	`select_le`	`R = select_le(A, B)`
Not Less Than	`select_nlt`	`R = select_nlt(A, B)`
Not Less Than or Equal To	`select_nle`	`R = select_nle(A, B)`

Conditional Select Operator Usage

For conditional select operators, the return value is stored in C if the comparison is true or in D if false. The following table shows the return values for each class of the conditional select operators, using the Return Value Notation described earlier.

Compare Operator Return Value Mapping

<table>
<tr><th>R</th><th>A0</th><th>Operators</th><th>B</th><th>C</th><th>D</th><th>F32vec4</th><th>F64vec2</th><th>F32vec1</th></tr>
<tr><td>R0:=</td><td>(A1
!(A1</td><td>select_[eq | lt | le | gt | ge]
select_[ne | nlt | nle | ngt | nge]</td><td>B0)
B0)</td><td>C0
C0</td><td>D0
D0</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>R1:=</td><td>(A2
!(A2</td><td>select_[eq | lt | le | gt | ge]
select_[ne | nlt | nle | ngt | nge]</td><td>B1)
B1)</td><td>C1
C1</td><td>D1
D1</td><td>X</td><td>X</td><td>N/A</td></tr>
<tr><td>R2:=</td><td>(A2
!(A2</td><td>select_[eq | lt | le | gt | ge]
select_[ne | nlt | nle | ngt | nge]</td><td>B2)
B2)</td><td>C2
C2</td><td>D2
D2</td><td>X</td><td>N/A</td><td>N/A</td></tr>
<tr><td>R3:=</td><td>(A3
!(A3</td><td>select_[eq | lt | le | gt | ge]</td><td>B3)
B3)</td><td>C3
C3</td><td>D3
D3</td><td>X</td><td>N/A</td><td>N/A</td></tr>
</table>

R	A0	Operators	B	C	D	F32vec4	F64vec2	F32vec1
		`select_[ne | nlt | nle | ngt | nge]`						

The following table shows examples for conditional select operations and corresponding intrinsics.

Conditional Select Operations for Fvec Classes

Returns	Example Syntax Usage	Intrinsic
Compare for Equality		
4 floats	`F32vec4 R = select_eq(F32vec4 A);`	`_mm_cmpeq_ps`
2 doubles	`F64vec2 R = select_eq(F64vec2 A);`	`_mm_cmpeq_pd`
1 float	`F32vec1 R = select_eq(F32vec1 A);`	`_mm_cmpeq_ss`
Compare for Inequality		
4 floats	`F32vec4 R = select_neq(F32vec4 A);`	`_mm_cmpneq_ps`
2 doubles	`F64vec2 R = select_neq(F64vec2 A);`	`_mm_cmpneq_pd`
1 float	`F32vec1 R = select_neq(F32vec1 A);`	`_mm_cmpneq_ss`
Compare for Less Than		
4 floats	`F32vec4 R = select_lt(F32vec4 A);`	`_mm_cmplt_ps`
2 doubles	`F64vec2 R = select_lt(F64vec2 A);`	`_mm_cmplt_pd`
1 float	`F32vec1 R = select_lt(F32vec1 A);`	`_mm_cmplt_ss`
Compare for Less Than or Equal		
4 floats	`F32vec4 R = select_le(F32vec4 A);`	`_mm_cmple_ps`
2 doubles	`F64vec2 R = select_le(F64vec2 A);`	`_mm_cmple_pd`
1 float	`F32vec1 R = select_le(F32vec1 A);`	`_mm_cmple_ps`

Compare for Greater Than		
4 floats	F32vec4 R = select_gt(F32vec4 A);	_mm_cmpgt_ps
2 doubles	F64vec2 R = select_gt(F64vec2 A);	_mm_cmpgt_pd
1 float	F32vec1 R = select_gt(F32vec1 A);	_mm_cmpgt_ss

Compare for Greater Than or Equal To		
4 floats	F32vec1 R = select_ge(F32vec4 A);	_mm_cmpge_ps
2 doubles	F64vec2 R = select_ge(F64vec2 A);	_mm_cmpge_pd
1 float	F32vec1 R = select_ge(F32vec1 A);	_mm_cmpge_ss

Compare for Not Less Than		
4 floats	F32vec1 R = select_nlt(F32vec4 A);	_mm_cmpnlt_ps
2 doubles	F64vec2 R = select_nlt(F64vec2 A);	_mm_cmpnlt_pd
1 float	F32vec1 R = select_nlt(F32vec1 A);	_mm_cmpnlt_ss

Compare for Not Less Than or Equal		
4 floats	F32vec1 R = select_nle(F32vec4 A);	_mm_cmpnle_ps
2 doubles	F64vec2 R = select_nle(F64vec2 A);	_mm_cmpnle_pd
1 float	F32vec1 R = select_nle(F32vec1 A);	_mm_cmpnle_ss

Compare for Not Greater Than		
4 floats	F32vec1 R = select_ngt(F32vec4 A);	_mm_cmpngt_ps
2 doubles	F64vec2 R = select_ngt(F64vec2 A);	_mm_cmpngt_pd
1 float	F32vec1 R = select_ngt(F32vec1 A);	_mm_cmpngt_ss

Compare for Not Greater Than or Equal		
4 floats	`F32vec1 R = select_nge(F32vec4 A);`	`_mm_cmpnge_ps`
2 doubles	`F64vec2 R = select_nge(F64vec2 A);`	`_mm_cmpnge_pd`
1 float	`F32vec1 R = select_nge(F32vec1 A);`	`_mm_cmpnge_ss`

Cacheability Support Operators

Stores (non-temporal) the two double-precision, floating-point values of `A`. Requires a 16-byte aligned address.

`void store_nta(double *p, F64vec2 A);`
Corresponding intrinsic: `_mm_stream_pd`

Stores (non-temporal) the four single-precision, floating-point values of `A`. Requires a 16-byte aligned address.

`void store_nta(float *p, F32vec4 A);`
Corresponding intrinsic: `_mm_stream_ps`

Debug Operations

The debug operations do not map to any compiler intrinsics for MMX™ technology or Intel® Streaming SIMD Extensions . They are provided for debugging programs only. Use of these operations may result in loss of performance, so you should not use them outside of debugging.

Output Operations

The two single, double-precision floating-point values of `A` are placed in the output buffer and printed in decimal format as follows:

```
cout << F64vec2 A;
"[1]:A1 [0]:A0"
```

Corresponding intrinsics: none

The four, single-precision floating-point values of `A` are placed in the output buffer and printed in decimal format as follows:

```
cout << F32vec4 A;
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```

Corresponding intrinsics: none

The lowest, single-precision floating-point value of `A` is placed in the output buffer and printed.

`cout << F32vec1 A;`
Corresponding intrinsics: none

Element Access Operations

```
double d = F64vec2 A[int i]
```

Read one of the two, double-precision floating-point values of `A` without modifying the corresponding floating-point value. Permitted values of `i` are 0 and 1. For example:

If DEBUG is enabled and `i` is not one of the permitted values (0 or 1), a diagnostic message is printed and the program aborts.

```
double d = F64vec2 A[1];
```
Corresponding intrinsics: none

Read one of the four, single-precision floating-point values of `A` without modifying the corresponding floating point value. Permitted values of `i` are 0, 1, 2, and 3. For example:

```
float f = F32vec4 A[int i]
```

If DEBUG is enabled and `i` is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.

```
float f = F32vec4 A[2];
```
Corresponding intrinsics: none

Element Assignment Operations

```
F64vec4 A[int i] = double d;
```

Modify one of the two, double-precision floating-point values of `A`. Permitted values of `int i` are 0 and 1. For example:

```
F32vec4 A[1] = double d;
F32vec4 A[int i] = float f;
```

Modify one of the four, single-precision floating-point values of `A`. Permitted values of `int i` are 0, 1, 2, and 3. For example:

If DEBUG is enabled and `int i` is not one of the permitted values (0-3), a diagnostic message is printed and the program aborts.

```
F32vec4 A[3] = float f;
```
Corresponding intrinsics: none.

Load and Store Operators

Loads two, double-precision floating-point values, copying them into the two, floating-point values of `A`. No assumption is made for alignment.

```
void loadu(F64vec2 A, double *p)
```
Corresponding intrinsic: `_mm_loadu_pd`

Stores the two, double-precision floating-point values of `A`. No assumption is made for alignment.

```
void storeu(float *p, F64vec2 A);
```
Corresponding intrinsic: `_mm_storeu_pd`

Loads four, single-precision floating-point values, copying them into the four floating-point values of `A`. No assumption is made for alignment.

```
void loadu(F32vec4 A, double *p)
```
Corresponding intrinsic: `_mm_loadu_ps`

Stores the four, single-precision floating-point values of `A`. No assumption is made for alignment.

```
void storeu(float *p, F32vec4 A);
```
Corresponding intrinsic: `_mm_storeu_ps`

Unpack Operators

Selects and interleaves the lower, double-precision floating-point values from `A` and `B`.

```
F64vec2 R = unpack_low(F64vec2 A, F64vec2 B);
```
Corresponding intrinsic: `_mm_unpacklo_pd(a, b)`

Selects and interleaves the higher, double-precision floating-point values from `A` and `B`.

`F64vec2 R = unpack_high(F64vec2 A, F64vec2 B);`
Corresponding intrinsic: `_mm_unpackhi_pd(a, b)`

Selects and interleaves the lower two, single-precision floating-point values from `A` and `B`.

`F32vec4 R = unpack_low(F32vec4 A, F32vec4 B);`
Corresponding intrinsic: `_mm_unpacklo_ps(a, b)`

Selects and interleaves the higher two, single-precision floating-point values from `A` and `B`.

`F32vec4 R = unpack_high(F32vec4 A F32vec4 B);`
Corresponding intrinsic: `_mm_unpackhi_ps(a, b)`

Move Mask Operators

Creates a 2-bit mask from the most significant bits of the two, double-precision floating-point values of `A`, as follows:

```
int i = move_mask(F64vec2 A)
i := sign(a1)<<1 | sign(a0)<<0
```
Corresponding intrinsic: `_mm_movemask_pd`

Creates a 4-bit mask from the most significant bits of the four, single-precision floating-point values of `A`, as follows:

```
int i = move_mask(F32vec4 A)
i := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)<<0
```
Corresponding intrinsic: `_mm_movemask_ps`

Classes Quick Reference

This appendix contains tables listing operators to perform various SIMD operations, corresponding intrinsics to perform those operations, and the classes that implement those operations. The classes listed here belong to the Intel® C++ Class Libraries for SIMD Operations.

In the following tables,

- N/A indicates that the operator is not implemented in that particular class. For example, in the Logical Operations table, the `Andnot` operator is not implemented in the `F32vec4` and `F32vec1` classes.
- All other entries under Classes indicate that those operators are implemented in those particular classes, and the entries under the Classes columns provide the suffix for the corresponding intrinsic. For example, consider the Arithmetic Operations: Part1 table, where the corresponding intrinsic is `_mm_add_[x]` and the entry `epi16` is under the `I16vec8` column. It means that the `I16vec8` class implements the addition operators and the corresponding intrinsic is `_mm_add_epi16`.

Logical Operations:

Operators	Corresponding Intrinsic	Classes				
		I128vec1, I64vec2, I32vec4, I16vec8, I8vec16	**I64vec1, I32vec2, I16vec4, I8vec8**	**F64vec 2**	**F32vec 4**	**F32vec 1**
`&, &=`	`_mm_and_[x]`	si128	si64	pd	ps	ps
`|, |=`	`_mm_or_[x]`	si128	si64	pd	ps	ps
`^, ^=`	`_mm_xor_[x]`	si128	si64	pd	ps	ps
`Andnot`	`_mm_andnot_[x]`	si128	si64	pd	N/A	N/A

Arithmetic Operations: Part 1

Operators	Corresponding Intrinsic	Classes			
		I64vec2	**I32vec4**	**I16vec8**	**I8vec16**
+, +=	_mm_add_[x]	epi64	epi32	epi16	epi8
-, -=	_mm_sub_[x]	epi64	epi32	epi16	epi8
*, *=	_mm_mullo_[x]	N/A	N/A	epi16	N/A
/, /=	_mm_div_[x]	N/A	N/A	N/A	N/A
mul_high	_mm_mulhi_[x]	N/A	N/A	epi16	N/A
mul_add	_mm_madd_[x]	N/A	N/A	epi16	N/A
sqrt	_mm_sqrt_[x]	N/A	N/A	N/A	N/A
rcp	_mm_rcp_[x]	N/A	N/A	N/A	N/A
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	N/A
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A

Arithmetic Operations: Part 2

Operators	Corresponding Intrinsic	Classes					
		I32vec2	**I16vec4**	**I8vec8**	**F64vec2**	**F32vec4**	**F32vec1**
+, +=	_mm_add_[x]	pi32	pi16	pi8	pd	ps	ss
-, -=	_mm_sub_[x]	pi32	pi16	pi8	pd	ps	ss
*, *=	_mm_mullo_[x]	N/A	pi16	N/A	pd	ps	ss
/, /=	_mm_div_[x]	N/A	N/A	N/A	pd	ps	ss
mul_high	_mm_mulhi_[x]	N/A	pi16	N/A	N/A	N/A	N/A
mul_add	_mm_madd_[x]	N/A	pi16	N/A	N/A	N/A	N/A
sqrt	_mm_sqrt_[x]	N/A	N/A	N/A	pd	ps	ss
rcp	_mm_rcp_[x]	N/A	N/A	N/A	pd	ps	ss
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	pd	ps	ss
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	pd	ps	ss

Operators	Corresponding Intrinsic	Classes					
		I32vec 2	I16vec 4	I8vec8	F64vec 2	F32vec 4	F32vec 1
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	pd	ps	ss

Shift Operations: Part 1

Operators	Corresponding Intrinsic	Classes				
		I128vec1	I64vec 2	I32vec 4	I16vec 8	I8vec1 6
>>,>>=	_mm_srl_[x] _mm_srli_[x] _mm_sra__[x] _mm_srai_[x]	N/A N/A N/A N/A	epi64 epi64 N/A N/A	epi32 epi32 epi32 epi32	epi16 epi16 epi16 epi16	N/A N/A N/A N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	N/A N/A	epi64 epi64	epi32 epi32	epi16 epi16	N/A N/A

Shift Operations: Part 2

Operators	Corresponding Intrinsic	Classes			
		I64vec 1	I32vec 2	I16vec 4	I8vec8
>>,>>=	_mm_srl_[x] _mm_srli_[x] _mm_sra__[x] _mm_srai_[x]	si64 si64 N/A N/A	pi32 pi32 pi32 pi32	pi16 pi16 pi16 pi16	N/A N/A N/A N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	si64 si64	pi32 pi32	pi16 pi16	N/A N/A

Comparison Operations: Part 1

Operators	Corresponding Intrinsic	Classes					
		I32vec 4	I16vec 8	I8vec1 6	I32vec 2	I16vec 4	I8vec8
cmpeq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpneq	_mm_cmpeq_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmpgt	_mm_cmpgt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64
cmplt	_mm_cmplt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmple	_mm_cmple_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64

Operators	Corresponding Intrinsic	Classes					
		I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8
cmpngt	_mm_cmpngt_[x]	epi32	epi16	epi8	pi32	pi16	pi8
cmpnge	_mm_cmpnge_[x]	N/A	N/A	N/A	N/A	N/A	N/A
cmpnlt	_mm_cmpnlt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
cmpnle	_mm_cmpnle_[x]	N/A	N/A	N/A	N/A	N/A	N/A

* Note that `_mm_andnot_[y]` intrinsics do not apply to the fvec classes.

Comparison Operations: Part 2

Operators	Corresponding Intrinsic	Classes		
		F64vec2	F32vec4	F32vec1
cmpeq	_mm_cmpeq_[x]	pd	ps	ss
cmpneq	_mm_cmpeq_[x] _mm_andnot_[y]*	pd	ps	ss
cmpgt	_mm_cmpgt_[x]	pd	ps	ss
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	pd	ps	ss
cmplt	_mm_cmplt_[x]	pd	ps	ss
cmple	_mm_cmple_[x] _mm_andnot_[y]*	pd	ps	ss
cmpngt	_mm_cmpngt_[x]	pd	ps	ss
cmpnge	_mm_cmpnge_[x]	pd	ps	ss
cmpnlt	_mm_cmpnlt_[x]	pd	ps	ss
cmpnle	_mm_cmpnle_[x]	pd	ps	ss

* Note that `_mm_andnot_[y]` intrinsics do not apply to the `fvec` classes.

Conditional Select Operations: Part 1

Operators	Corresponding Intrinsic	Classes					
		I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8
select_eq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_neq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64

Operators	Corresponding Intrinsic	Classes					
		I32vec4	**I16vec8**	**I8vec16**	**I32vec2**	**I16vec4**	**I8vec8**
select_gt	_mm_cmpgt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64
select_ngt	_mm_cmpgt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nge	_mm_cmpge_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nlt	_mm_cmplt_[x]	N/A	N/A	N/A	N/A	N/A	N/A
select_nle	_mm_cmple_[x]	N/A	N/A	N/A	N/A	N/A	N/A

* Note that `_mm_andnot_[y]` intrinsics do not apply to the `fvec` classes.

Conditional Select Operations: Part 2

Operators	Corresponding Intrinsic	Classes		
		F64vec2	**F32vec4**	**F32vec1**
select_eq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_neq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_gt	_mm_cmpgt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss

Operators	Corresponding Intrinsic	Classes		
		F64vec2	F32vec4	F32vec1
select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	pd	ps	ss
select_ngt	_mm_cmpgt_[x]	pd	ps	ss
select_nge	_mm_cmpge_[x]	pd	ps	ss
select_nlt	_mm_cmplt_[x]	pd	ps	ss
select_nle	_mm_cmple_[x]	pd	ps	ss

* Note that `_mm_andnot_[y]` intrinsics do not apply to the `fvec` classes.

Packing and Unpacking Operations: Part 1

Operators	Corresponding Intrinsic	Classes				
		I64vec2	I32vec4	I16vec8	I8vec16	I32vec2
unpack_high	_mm_unpackhi_[x]	epi64	epi32	epi16	epi8	pi32
unpack_low	_mm_unpacklo_[x]	epi64	epi32	epi16	epi8	pi32
pack_sat	_mm_packs_[x]	N/A	epi32	epi16	N/A	pi32
packu_sat	_mm_packus_[x]	N/A	N/A	epi16	N/A	N/A
sat_add	_mm_adds_[x]	N/A	N/A	epi16	epi8	N/A
sat_sub	_mm_subs_[x]	N/A	N/A	epi16	epi8	N/A

Packing and Unpacking Operations: Part 2

Operators	Corresponding Intrinsic	Classes				
		I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
unpack_high	_mm_unpackhi_[x]	pi16	pi8	pd	ps	N/A
unpack_low	_mm_unpacklo_[x]	pi16	pi8	pd	ps	N/A
pack_sat	_mm_packs_[x]	pi16	N/A	N/A	N/A	N/A
packu_sat	_mm_packus_[x]	pu16	N/A	N/A	N/A	N/A
sat_add	_mm_adds_[x]	pi16	pi8	pd	ps	ss
sat_sub	_mm_subs_[x]	pi16	pi8	pi16	pi8	pd

Conversions Operations:

Conversion operations can be performed using intrinsics only. There are no classes implemented to correspond to these intrinsics.

Operators	Corresponding Intrinsic
F64vec2ToInt	_mm_cvttsd_si32
F32vec4ToF64vec2	_mm_cvtps_pd
F64vec2ToF32vec4	_mm_cvtpd_ps
IntToF64vec2	_mm_cvtsi32_sd
F32vec4ToInt	_mm_cvtt_ss2si
F32vec4ToIs32vec2	_mm_cvttps_pi32
IntToF32vec4	_mm_cvtsi32_ss
Is32vec2ToF32vec4	_mm_cvtpi32_ps

Programming Example

This sample program uses the `F32vec4` class to average the elements of a twenty element floating point array.

```
//Include Intel® Streaming SIMD Extension (Intel® SSE) Class Definitions
 #include <fvec.h>

//Shuffle any two single precision floating point from a
//into low two SP FP and shuffle any two SP FP from b
//into high two SP FP of destination

#define SHUFFLE(a,b,i) (F32vec4)_mm_shuffle_ps(a,b,i)
#include <stdio.h>
#define SIZE 20

//Global variables
 float result;
_MM_ALIGN16 float array[SIZE];

//*****************************************************
// Function: Add20ArrayElements
// Add all the elements of a twenty element array
//*****************************************************
void Add20ArrayElements (F32vec4 *array, float *result) {
   F32vec4 vec0, vec1;
   vec0 = _mm_load_ps ((float *) array); // Load array's first four floats

   //*****************************************************
   // Add all elements of the array, four elements at a time
   //******************************************************
   vec0 += array[1]; // Add elements 5-8
   vec0 += array[2]; // Add elements 9-12
   vec0 += array[3]; // Add elements 13-16
   vec0 += array[4]; // Add elements 17-20

   //*****************************************************
   // There are now four partial sums.
```

```
   // Add the two lowers to the two raises,
   // then add those two results together
   //*****************************************************
   vec1 = SHUFFLE(vec1, vec0, 0x40);
   vec0 += vec1;
   vec1 = SHUFFLE(vec1, vec0, 0x30);
   vec0 += vec1;
   vec0 = SHUFFLE(vec0, vec0, 2);
   _mm_store_ss (result, vec0); // Store the final sum
}

void main(int argc, char *argv[]) {
   int i;

//Initialize the array
   for (i=0; i < SIZE; i++) { array[i] = (float) i; }

//Call function to add all array elements
   Add20ArrayElements (array, &result);

//Print average array element value
   printf ("Average of all array values = %f\n", result/20.);
   printf ("The correct answer is %f\n\n\n", 9.5);
}
```

Intel's valarray Implementation

The Intel® C++ Compiler Classic provides a high performance implementation of specialized one-dimensional valarray operations for the C++ standard STL valarray container.

The standard C++ valarray template consists of array/vector operations for high performance computing. These operations are designed to exploit high performance hardware features such as parallelism and achieve performance benefits.

Intel's valarray implementation uses the Intel® Integrated Performance Primitives (Intel® IPP), which is part of the product. Select IPP when you install the product.

The valarray implementation consists of a replacement header, `<valarray>`, that provides a specialized, high-performance implementation for the following operators and types:

Operator	Type
abs, acos, acosh, asin, asinh, atan, atan2, atanh, cbrt, cdfnorm, ceil, cos, cosh, erf, erfc, erfinv, exp, expm1, floor, hypot, inv, invcbrt, invsqrt, ln, log, log10, log1p, nearbyint, pow, pow2o3, pow3o2, powx, rint, round, sin, sinh, sqrt, tan, tanh, trunk	float, double
add, conj, div, mul, mulbyconj, mul, sub	Ipp32fc, Ipp64fc
addition, subtraction, division, multiplication	float, double
bitwise or, and, xor	(all unsigned) char, short, int
min, max, sum	signed or short/signed int, float, double

Use valarray in Source Code

Intel's valarray implementation allows you to declare huge arrays for parallel processing. Improved implementation of valarray is tied up with calling the IPP libraries that are part of Intel® Integrated Performance Primitives (Intel® IPP).

To use valarrays in your source code, include the valarray header file, `<valarray>`. The `<valarray>` header file is located in the path `<installdir>/perf_header`.

The following example shows a valarray addition operation (+) specialized through use of Intel's implementation of valarray:

```
#include <valarray>
void test( )
{
    std::valarray<float> vi(N), va(N);
    …
    vi = vi + va;  //array addition
    …
}
```

NOTE
To use the static merged library containing all CPU-specific optimized versions of the library code, you need to call the `ippStaticInit` function first, before any IPP calls. This ensures automatic dispatch to the appropriate version of the library code for Intel® processor and the generic version of the library code for non-Intel processors at runtime. If you do not call `ippStaticInit` first, the merged library will use the generic instance of the code. If you are using the dynamic version of the libraries, you do not need to call `ippStaticInit`.

Compiling valarray Source Code

To compile your valarray source code, the compiler option, `/Quse-intel-optimized-headers` (for Windows*) or `-use-intel-optimized-headers` (for Linux* and macOS), is used to include the required valarray header file and all the necessary IPP library files.

The following examples illustrate how to compile and link a program to include the Intel valarray replacement header file and link with the Intel® IPP libraries. Refer to the Intel® IPP documentation for details.

In the following examples, "merged" libraries refers to using a static library that contains all the CPU-specific variants of the library code.

Linux* OS Examples

The following command line performs a one-step compilation for a system based on Intel® 64 architecture, running Linux OS:

```
icpc -use-intel-optimized-headers source.cpp
```

The following command lines perform separate compile and link steps for a system based on Intel® 64 architecture, running Linux OS:

so (dynamic):

```
icpc -use-intel-optimized-headers -c source.cpp
```

```
icpc source.o -use-intel-optimized-headers -shared-intel
```

Merged (static):

```
icpc -use-intel-optimized-headers -c source.cpp
```

```
icpc source.o -use-intel-optimized-headers
```

Windows* OS Examples

The following command line performs a one-step compilation for a system based on IA-32 architecture, running Windows OS:

```
icc /Quse-intel-optimized-headers source.cpp
```

The following command lines perform separate compile and link steps for a system based on IA-32 architecture, running Windows OS:

DLL (dynamic):

```
icc /Quse-intel-optimized-headers /c source.cpp
```

```
icc source.obj /Quse-intel-optimized-headers
```

Merged (static):

```
icc /Quse-intel-optimized-headers /Qipp-link:static /c source.cpp
icx source.obj /Quse-intel-optimized-headers /Qipp-link:static
```

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Intel's C++ Asynchronous I/O Extensions for Windows

Intel's C/C++ asynchronous input/output (Intel's C/C++ AIO) extensions, like library functions or classes, can be used to improve the performance of C/C++ applications by executing I/O operations in asynchronous mode. The extensions initiate I/O operation and immediately resume normal tasks while the I/O operations are executed in parallel.

Intel's C/C++ asynchronous I/O extensions are supported on IA-32 architecture-based and Intel® 64 architecture-based Windows platforms.

Intel's C/C++ AIO library functions and template class are implemented in the `libicaio.lib` library. This library is supplied as part of the Intel® C++ Compiler Classic package and is installed into the common directory: `<install-dir>/lib`.

Types of Intel's C/C++ Asynchronous I/O Extensions

Intel's C/C++ asynchronous I/O extensions comprise the following:

- **Asynchronous I/O Library:** A set of POSIX-based asynchronous I/O library functions, supported on Windows operating systems, for applications written in C/C++ language. The interface file is `aio.h`.
- **Asynchronous I/O Template Class:** An `asych_class` template class, supported on Windows operating systems, for applications written in C++ language. This template class can be used to introduce asynchronous execution of I/O operations with the Standard Template Library's (STL's) streams classes. The interface file is `aiostream.h`.

See Also

Intel's C++ Asynchronous I/O Library for Windows
Intel's C++ Asynchronous I/O Class for Windows

Intel's C++ Asynchronous I/O Library for Windows

Intel's C/C++ asynchronous I/O (AIO) library implementation for the Windows operating system (on IA-32 and Intel® 64 platforms) is similar to the POSIX AIO library implementation for the Linux operating system.

The differences between Intel's C/C++ AIO Windows OS implementation and the standard POSIX AIO implementation are listed below:

- In `struct aiocb`,
  - The Windows OS compatible type `HANDLE` replaces the POSIX AIO type `unsigned int` for the file descriptor `aio_fildes`.
  - The type `intptr_t` replaces the POSIX AIO types `ssize_t` and `__off_t`.
- The structure specifying the signal event descriptor, `struct sigevent` is similar to the Linux operating system implementation of the POSIX AIO library. It differs from the Linux implementation in the following ways:
  - Signal notification and non-notification for thread call-back is supported
  - Signal notification on completion of the AIO operation is *not* supported

    This is true for programs that were already written for Linux/Unix and ported to Windows OS that wish to setup an AIO completion handler without the name of the handler set in the `aiocb struct`. Because of the way that signals are supported in Windows, this is impossible to implement. For new applications, or to port existing applications, the programmer should set the name of the handler before calling the `aio_read` or `aio_write` routines. For example:

```
static void aio_CompletionRoutine(sigval_t sigval)
{
    // … code …
}

… code …

my_aio.aio_sigevent.sigev_notify          = SIGEV_THREAD;
my_aio.aio_sigevent.sigev_notify_function = aio_CompletionRoutine;
```

> **NOTE**
> The POSIX AIO library and the Microsoft SDK provide similar AIO functions. The main difference between the POSIX AIO functions and the Windows operating system-based AIO functions is that while POSIX allows you to execute AIO operations with any file, the Windows operating system executes AIO operations only with files flagged with `FILE_FLAG_OVERLAPPED`.

Intel's asynchronous I/O library functions listed below are all based on POSIX AIO functions. They are defined in the `aio.h` file.

- `aio_read()`
- `aio_write()`
- `aio_suspend()`
- `aio_error()`
- `aio_return()`
- `aio_fsync()`
- `aio_cancel()`
- `lio_listio()`

aio_read

Performs an asynchronous read operation.

Syntax

`int aio_read(struct aiocb *aiocbp);`

Description

The `aio_read()` function requests an asynchronous read operation, calling the function,

```
"ReadFile(hFile, lpBuffer, nNumberOfBytesToRead, lpNumberOfBytesRead, NULL);"
```

where,

- `hFile` is given by *aiocbp->aio_fildes*
- `lpBuffer` is given by *aiocbp->aio_buf*
- `nNumberOfBytesToRead` is given by *aiocbp->aio_nbytes*

Use the function `aio_return()` to retrieve the actual bytes read in `lpNumberOfBytesRead`.

Use the extension *aiocb->aio_offset == (intptr_t)-1* to start the read operation after the last read record. This extension avoids extra file positioning and enhances performance.

Returns

0: On success

-1: On error

To get the correct error code, use `errno`. To get the error that occurred during asynchronous read operation, use `aio_error()` function.

See Also

Example Code for aio_read()

aio_write

Performs an asynchronous write operation.

Syntax

`int aio_write(struct aiocb *aiocbp);`

Description

The `aio_write()` function requests an asynchronous write operation, calling the function,

```
"WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, lpNumberOfBytesWritten, NULL);
```

where,

- `hFile` is given by *aiocbp->aio_fildes*
- `lpBuffer` is given by *aiocbp->aio_buf*
- `nNumberOfBytesToWrite` is given by *aiocbp->aio_nbytes*

Use the function `aio_return()` to retrieve the actual bytes written in `lpNumberOfBytesWritten`.

Use the extension *aiocb->aio_offset == (intptr_t)-1* to start the write operation after the last written record. This extension avoids extra file positioning and enhances performance.

Returns

0: On success

-1: On error

To get the correct error code, use `errno`. To get the error that occurred during asynchronous write operation, use `aio_error()` function.

See Also

Example Code for aio_write()

Example for aio_read and aio_write Functions

The example illustrates the performance gain of the asynchronous I/O usage in comparison with synchronous I/O usage. In the example, 5.6 MB of data is asynchronously written with the main program computation, which is the scalar multiplication of two vectors with some normalization.

C-source file executing a scalar multiplication:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double do_compute(double A, double B, int arr_len)
{
  int i;
  double   res = 0;
  double  *xA = malloc(arr_len * sizeof(double));
  double  *xB = malloc(arr_len * sizeof(double));
  if ( !xA || !xB )
   abort();
   for (i = 0; i < arr_len; i++) {
    xA[i] = sin(A);
    xB[i] = cos(B);
    res = res + xA[i]*xA[i];
   }
  free(xA);
  free(xB);
 return res;
}
```

C-main-source file using asynchronous I/O implementation:

```
#define DIM_X   123/*123*/
#define DIM_Y   70000
double  aio_dat[DIM_Y /*12MB*/] = {0};
double  aio_dat_tmp[DIM_Y /*12MB*/];

#include <stdio.h>
#include <aio.h>

typedef struct aiocb  aiocb_t;
   aiocb_t   my_aio;
   aiocb_t  *my_aio_list[1] = {&my_aio};

int main()
{
  double do_compute(double A, double B, int arr_len);
  int i, j;
  HANDLE fd = CreateFile("aio.dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL,
  NULL);
/* Do some complex computation */
for (i = 0; i < DIM_X; i++) {
```

```
 for ( j = 0; j < DIM_Y; j++ )
 aio_dat[j] = do_compute(i, j, DIM_X);

  if (i) aio_suspend(my_aio_list, 1, 0);
  my_aio.aio_fildes = fd;
  my_aio.aio_buf    = memcpy(aio_dat_tmp, aio_dat, sizeof(aio_dat_tmp));
  my_aio.aio_nbytes = sizeof(aio_dat_tmp);
  my_aio.aio_offset = (intptr_t)-1;
  my_aio.aio_sigevent.sigev_notify = SIGEV_NONE;

  if ( aio_write((void*)&my_aio) == -1 ) {
  printf("ERROR!!! %s\n", "aio_write()==-1");
  abort();}
  }
 aio_suspend(my_aio_list, 1, 0);
 return 0;
}
```

C-main-source file example 2 using asynchronous I/O implementation:

```
// icx (for C++) dpcpp (for DPC++) -c do_compute.c
// icx (for C++) dpcpp (for DPC++) aio_sample2.c do_compute.obj
// aio_sample2.exe

#define DIM_X   123
#define DIM_Y    70
double  aio_dat[DIM_Y] = {0};
double  aio_dat_tmp[DIM_Y];
static volatile int aio_flg = 1;

#include <aio.h>
typedef struct aiocb  aiocb_t;
aiocb_t               my_aio;
#define WAIT { while (!aio_flg); aio_flg = 0; }
#define aio_OPEN(_fname )\
CreateFile(_fname,                       \
           GENERIC_READ | GENERIC_WRITE, \
           FILE_SHARE_READ,              \
           NULL,                         \
           OPEN_ALWAYS,                  \
           FILE_ATTRIBUTE_NORMAL,        \
           NULL)

static void aio_CompletionRoutine(sigval_t sigval)
{
    aio_flg = 1;
}

int main()
{
    double do_compute(double A, double B, int arr_len);
    int      i, j, res;
    char    *fname = "aio_sample2.dat";
    HANDLE   aio_fildes = aio_OPEN(fname);

    my_aio.aio_fildes = aio_fildes;
    my_aio.aio_nbytes = sizeof(aio_dat_tmp);
    my_aio.aio_sigevent.sigev_notify          = SIGEV_THREAD;
    my_aio.aio_sigevent.sigev_notify_function = aio_CompletionRoutine;
```

```
    /*
    ** writing
    */
    my_aio.aio_offset = -1;
    printf("Writing\n");
    for (i = 0; i < DIM_X; i++) {
        for (j = 0; j < DIM_Y; j++)
            aio_dat[j] = do_compute(i, j, DIM_X);
        WAIT;
        my_aio.aio_buf = memcpy(aio_dat_tmp, aio_dat, sizeof(aio_dat_tmp));
        res = aio_write(&my_aio);
        if (res) {printf("res!=0\n");abort();}
    }

    //
    // flushing
    //
    printf("Flushing\n");
    WAIT;
    res = aio_fsync(O_SYNC, &my_aio);
    if (res) {printf("res!=0\n");abort();}
    WAIT;

    //
    // reading
    //
    printf("Reading\n");
    my_aio.aio_offset = 0;
    my_aio.aio_buf    = (volatile char*)aio_dat_tmp;
    for (i = 0; i < DIM_X; i++) {
        aio_read(&my_aio);
        for (j = 0; j < DIM_Y; j++)
            aio_dat[j] = do_compute(i, j, DIM_X);
        WAIT;
        res = aio_return(&my_aio);
        if (res != sizeof(aio_dat)) {
            printf("aio_read() did read %d bytes, expecting %d bytes\n", res, sizeof(aio_dat));
        }

        for (j = 0; j < DIM_Y; j++)
            if ( aio_dat[j] != aio_dat_tmp[j] )
                {printf("ERROR: aio_dat[j] != aio_dat_tmp[j]\n I=%d J=%d\n", i, j); abort();}
        my_aio.aio_offset += my_aio.aio_nbytes;
    }

    CloseHandle(aio_fildes);

    printf("\nDone\n");

return 0;
}
```

See Also

aio_read()

aio_write()

aio_suspend

Suspends the calling process until one of the asynchronous I/O operations completes.

Syntax

```
int aio_suspend(const struct aiocb * const cblist[], int n, const struct timespec
*timeout);
```

Arguments

cblist[]	Pointer to a control block on which I/O is initiated
n	Length of *cblist* list
**timeout*	Time interval to suspend the calling process

Description

The `aio_suspend()` function is like a wait operation. It suspends the calling process until,

- At least one of the asynchronous I/O requests in the list *cblist* of length *n* has completed
- A signal is delivered
- The time interval indicated in *timeout* is not `NULL` and has passed.

Each item in the *cblist* list must either be `NULL` (when it is ignored), or a pointer to a control block on which I/O was initiated using `aio_read()`, `aio_write()`, or `lio_listio()` functions.

Returns

0: On success

-1: On error

To get the correct error code, use `errno`.

See Also

Example Code for aio_suspend()

Example for aio_suspend Function

The following example illustrates a wait operation execution using the `aio_suspend()` function.

```
int aio_ex_2(HANDLE fd)
{
    static struct aiocb   aio[2];
    static struct aiocb  *aio_list[2] = {&aio[0], &aio[1]};
    int i, ret;

/* Data initialization */
IC_AIO_DATA_INIT(aio[0], fd, "rec#1\n", strlen("rec#1\n"), 0)
IC_AIO_DATA_INIT(aio[1], fd, "rec#2\n", strlen("rec#2\n"), aio[0].aio_nbytes)

/* Asynch-write */
if (aio_write(&aio[0]) == -1) return errno;
if (aio_write(&aio[1]) == -1) return errno;

/* Do some complex computation */
printf("do_compute(1000, 1.123)=%f", do_compute(1000, 1.123));

/* do the wait operation using sleep() */
```

```
ret = aio_suspend(aio_list, 2, 0);
if (ret == -1) return errno;

return 0;
}/* aio_ex_2 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
-bash-3.00$ cat dat
rec#1
rec#2
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
  {memset(&_aio, 0, sizeof(_aio)); \
  _aio.aio_fildes = _fd;          \
  _aio.aio_buf    = _dat;         \
  _aio.aio_nbytes = _len;         \
  _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd = CreateFile("dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
  NULL);
```

See Also

aio_suspend()

aio_error

Returns error status for asynchronous I/O requests.

Syntax

`int aio_error(const struct aiocb *aiocbp);`

Arguments

**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_error()` function returns the error status for the asynchronous I/O request in the control block, which is pointed to by *aiocbp*.

Returns

EINPROGRESS: When asynchronous I/O request is not completed

ECANCELED: When asynchronous I/O request is cancelled

0: On success

Error value: On error

To get the correct error value/code, use `errno`. This is the same error value returned when an error occurs during a `ReadFile()`, `WriteFile()`, or a `FlushFileBuffers()` operation.

See Also

Example Code for aio_error()

aio_return

Returns the final return status for the asynchronous I/O request.

Syntax

`ssize_t aio_return(struct aiocb *aiocbp);`

Arguments

**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_return` function returns the final return status for the asynchronous I/O request with control block pointed to by *aiocbp*.

Call this function only once for any given request, after `aio_error()` returns a value other than *EINPROGRESS*.

Returns

Return value for synchronous ReadFile()/WriteFile()/FlushFileBuffer() requests: When asynchronous I/O operation is completed

Undefined return value: When asynchronous I/O operation is not completed

Error value: When an error occurs

To get the correct error code/value, use `errno`.

See Also

Example Code for aio_return()

Example for aio_error and aio_return Functions

The following example illustrates how the `aio_error()` and `aio_return()` functions can be used.

```
int aio_ex_3(HANDLE fd)
{
 static struct aiocb aio;
 static struct aiocb *aio_list[] = {&aio};
 int    ret;
 char  *dat = "Hello from Ex-3\n";

/* Data initialization and asynchronously writing */

 IC_AIO_DATA_INIT(aio, fd, dat, strlen(dat), 0);
 if (aio_write(& aio) == -1) return errno;

 ret = aio_error(&aio);
 if ( ret == EINPROGRESS ) {
 fprintf(stderr, "ERRNO=%d STR=%s\n", ret, strerror(ret));
```

```
 ret = aio_suspend(aio_list, 1, NULL);
 if (ret == -1) return errno;}
 else if (ret)
 return ret;

 ret = aio_error(&aio);
 if (ret) return ret;

 ret = aio_return(&aio);
 printf("ret=%d\n", ret);

 return 0;
}/* aio_ex_3 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
ERRNO=115 STR=Operation now in progress
ret=16
-bash-3.00$ cat dat
Hello from Ex-3
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
 {memset(&_aio, 0, sizeof(_aio)); \
 _aio.aio_fildes = _fd;  \
 _aio.aio_buf    = _dat;  \
 _aio.aio_nbytes = _len;  \
 _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd  = CreateFile("dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
  NULL);
```

See Also

aio_error()

aio_return()

aio_fsync

Synchronizes all outstanding asynchronous I/O operations.

Syntax

`int aio_fsync(int op, struct aiocb *aiocbp);`

Arguments

op	Type of synchronization request operation

**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_fsync()` function performs a synchronization request operation on all outstanding asynchronous I/O operations associated with *aiocbp->aio_fildes*.

Returns

0: On successfully performing a synchronization request.

-1: On error; to get the correct error code, use `errno`.

aio_cancel

Cancels outstanding asynchronous I/O requests for the file descriptor fd.

Syntax

```
int aio_cancel(HANDLE fd, struct aiocb *aiocbp);
```

Arguments

fd	File descriptor
**aiocbp*	Pointer to control block from where asynchronous I/O request is generated

Description

The `aio_cancel()` function cancels outstanding asynchronous I/O requests for the file descriptor *fd*. If *aiocbp* is NULL, all outstanding asynchronous I/O requests are cancelled. If *aiocbp* is not NULL, only the requests described by the control block pointed to by *aiocbp* are cancelled.

Normal asynchronous notification occurs for cancelled requests. The request return status is set to -1, and the request error status is set to ECANCELED. The control block of requests that cannot be cancelled is not changed.

Unspecified results occur if *aiocbp* is not NULL and the *fd* differs from the file descriptor with which the asynchronous operation was initiated.

Returns

AIO_CANCELLED: When all specified requests are cancellled successfully.

AIO_NOTCANCELLED: When at least one of the specified requests is still in process of being cancelled; check the status of request using `aio_error`.

AIO_ALLDONE: When all specified requests were completed before cancel call was placed.

-1: When some error occurs. To get the correct error code, use `errno`.

See Also

Example Code for aio_cancel()

Example for aio_cancel Function

The following example illustrates how `aio_cancel()` function can be used.

```
int aio_ex_4(HANDLE fd)
{
  static struct aiocb   aio;
```

```
  static struct aiocb  *aio_list[] = {&aio};
  int    ret;
  char  *dat = "Hello from Ex-4\n";

 printf("AIO_CANCELED=%d AIO_NOTCANCELED=%d\n",
 AIO_CANCELED,   AIO_NOTCANCELED);

/* Data initialization and asynchronously writing */

 IC_AIO_DATA_INIT(aio, fd, dat, strlen(dat), 0);
 if (aio_write(&aio) == -1) return errno;

 ret = aio_cancel(fd, &aio);
 if ( ret == AIO_NOTCANCELED ) {
 fprintf(stderr, "ERRNO=%d STR=%s\n", ret, strerror(ret));
 ret = aio_suspend(aio_list, 1, NULL);
 if (ret == -1) return errno;}

 ret = aio_cancel(fd, &aio);
 if ( ret == AIO_CANCELED )
 fprintf(stderr, "ERRNO=%d STR=%s\n", ret, strerror(ret));
 else if (ret) return ret;

 return 0;
}/* aio_ex_4 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
AIO_CANCELED=0 AIO_NOTCANCELED=1
ERRNO=1 STR=Operation not permitted
-bash-3.00$ cat dat
Hello from Ex-4
-bash-3.00$
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
 {memset(&_aio, 0, sizeof(_aio)); \
 _aio.aio_fildes = _fd;          \
 _aio.aio_buf    = _dat;         \
 _aio.aio_nbytes = _len;         \
 _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd = CreateFile("dat",
  GENERIC_READ | GENERIC_WRITE,
  FILE_SHARE_READ,
  NULL,
  OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
  NULL);
```

See Also

aio_cancel()

lio_listio

Performs an asynchronous read operation.

Syntax

```
int lio_listio(int mode, struct aiocb *list[], int nent, struct sigevent *sig);
```

Arguments

mode

Takes following values declared in `<aio.h>` file:

- `LIO_WAIT`: Use when you want the function to return only after completing I/O operations (synchronous I/O operations)
- `LIO_NOWAIT`: Use when you want the function to return as soon as I/O operations are queued (asynchronous I/O requests)

**list[]*

Array of the `aiocb` pointers specifying the submitted I/O requests; NULL elements in the array are ignored

nent

Number of elements in the array

**sig*

Determines if asynchronous notification is sent after all I/O operations completes; takes following values:

- 0: Asynchronous notification occurs; a queued signal, with an application-defined value, is generated when an asynchronous I/O request occurs
- 1: Asynchronous notification does not occur even when asynchronous I/O requests are processed
- 2: Asynchronous notification occurs; a notification function is called to perform notification

Description

The `lio_listio()` function initiates a list of I/O requests with a single function call.

The *mode* argument determines whether the function returns when all the I/O operations are completed, or as soon as the operations are queued.

If the mode argument is `LIO_WAIT`, the function waits until all I/O operations are complete. The *sig* argument is ignored in this case.

If the *mode* argument is `LIO_NOWAIT`, the function returns immediately. Asynchronous notification occurs according to the *sig* argument after all the I/O operations complete.

Returns

When *mode*=`LIO_NOWAIT` the `lio_listio()` function returns:

- **0**: I/O operations are successfully queued
- **-1**: Error; I/O operations not queued; to get the proper error code, use `errno`.

When *mode*=`LIO_WAIT` the `lio_listio()` function returns:

- **0**: I/O operations specified completed successfully
- **-1**: Error; I/O operations not completed; to get the proper error code, use `errno`.

See Also

Example Code for lio_listio()

Example for lio_listio Function

The following example illustrates how the `lio_listio()` function can be used.

```
int aio_ex_5(HANDLE fd)
{
  static struct aiocb   aio[2];
  static struct aiocb  *aio_list[2] = {&aio[0], &aio[1]};
   int               i, ret;

/*
 ** Data initialization and Synchronously writing
*/
 IC_AIO_DATA_INIT(aio[0], fd, "rec#1\n", strlen("rec#1\n"), 0)
 IC_AIO_DATA_INIT(aio[1], fd, "rec#2\n", strlen("rec#2\n"),
 aio[0].aio_nbytes)
 aio[0].aio_lio_opcode = aio[1].aio_lio_opcode = LIO_WRITE;
 ret = lio_listio(LIO_WAIT, aio_list, 2, 0);
 if (ret) return ret;

 return 0;
}/* aio_ex_5 */
```

Result upon execution:

```
-bash-3.00$ ./a.out
-bash-3.00$ cat dat
rec#1
rec#2
-bash-3.00$
```

Remarks:

1. In the example, the `IC_AIO_DATA_INIT` is defined as follows:

```
#define IC_AIO_DATA_INIT(_aio, _fd, _dat, _len, _off)\
  {memset(&_aio, 0, sizeof(_aio)); \
   _aio.aio_fildes = _fd;          \
   _aio.aio_buf    = _dat;         \
   _aio.aio_nbytes = _len;         \
   _aio.aio_offset = _off;}
```

2. The file descriptor `fd` is obtained as:

```
HANDLE fd = CreateFile("dat",
   GENERIC_READ | GENERIC_WRITE,
   FILE_SHARE_READ,
   NULL,
   OPEN_ALWAYS,
   FILE_ATTRIBUTE_NORMAL/*|FILE_FLAG_OVERLAPPED*/,
   NULL);
```

3. The `aio_lio_opcode` refers to the field of each `aiocb` structure that specifies the operation to be performed. The supported operations are `LIO_READ` (do a 'read' operation), `LIO_WRITE` (do a 'write' operation), and `LIO_NOP` (do no operation); these symbols are defined in `<aio.h>`.

See Also

lio_listio()

Asynchronous I/O Function Errors

This topic only applies to Windows* OS.

The `errno` macro is used to obtain the errors that occur during asynchronous request functions such as `aio_read()`, `aio_write()`, `aio_fsync()`, and `lio_listio()` or asynchronous control functions, such as `aio_cancel()`, `aio_error()`, `aio_return()`, and `aio_suspend()`.

The following example illustrates how `errno` can be used.

```
#include <stdio.h>
#include <stdlib.h>
#include <aio.h>

struct aiocb    my_aio;
struct aiocb   *my_aio_list[1] = {&my_aio};

int main()
{
  int   res;
  double  arr[123456];
 timespec_t   my_t = {1, 0};

/* Data initialization */
 my_aio.aio_fildes = CreateFile("dat",
   GENERIC_READ | GENERIC_WRITE,
   FILE_SHARE_READ,
   NULL,
   OPEN_ALWAYS,
   FILE_ATTRIBUTE_NORMAL,
   NULL);
 my_aio.aio_buf    = (volatile char *)arr;
 my_aio.aio_nbytes = sizeof(arr);

/* Do asynchronous writing with computation overlapping */
 aio_write(&my_aio);
 do_compute(arr, 123456);

/* Suspend the asynchronous writing for 1 sec */
 res = aio_suspend(my_aio_list, 1, &my_t);
 if ( res ) {

/* The call was ended by timeout, before the indicated operations had completed. */
  if ( errno == EAGAIN ) {
  res = aio_suspend(my_aio_list, 1, 0);
  if ( res ) {
  printf("aio_suspend returned non-0\n"); return errno;}
  }
  else
  if ( res ) {
  printf("aio_suspend returned neither 0 nor EAGAIN\n");
  return errno;
  }
 }

 CloseHandle(my_aio.aio_fildes);
 printf("\nPass\n");
```

```
 return 0;
}
```

In the example, the program executes an asynchronous write operation, using `aio_write()`, overlapping with some computation, the `do_compute()` function execution. The pending write operation is suspended for one second using `aio_suspend()`.

On successful execution of the asynchronous write operation, zero is returned. `EAGAIN` or any other error value is returned when the call is ended by timeout before the indicated operation has completed.

You can check `EAGAIN` using the `errno` macro.

Intel's C++ Asynchronous I/O Class for Windows* Operating Systems

This topic only applies to Windows* OS.

Intel's C++ asynchronous I/O template class, `async_class`, is an implementation for the Windows* operating system on IA-32 and Intel® 64 architectures.

The `async_class` template class allows users to perform I/O operations asynchronously to the main program thread. In particular, the `async_class` template class can be used to introduce asynchronous execution of I/O operations with the STL streams classes. Users can quickly switch any of the I/O operations of the STL streams to asynchronous mode with minimal changes to the application code.

The template class `async_class` is defined in the `aiostream.h` file.

See Also

Details of template class async_class

Template Class async_class

This topic only applies to Windows* OS.

Intel's C++ asynchronous I/O class implementation contains two main classes within the `async` namespace: the `async_class` template class and the `thread_control` base class.

The header/typedef definitions are as follows:

```
namespace async {

template<class A>
class async_class:
public thread_control, public A
}
```

The template class `async_class` inherits support for asynchronous execution of I/O operations that are integrated within the base `thread_control` class.

All functionality to control asynchronous execution of a queue of STL stream operations is encapsulated in the base class `thread_control` and is inherited by template class `async_class`.

In most cases it is enough to add the header file `aiostream.h` to the source file and declare the file object as an instance of the new template class `async:async_class`. The initial stream class must be the parameter for the template class. Consequently, the defined output operator << and input operator >> are executed asynchronously.

NOTE
The header file `aiostream.h` includes all necessary declarations for the STL stream I/O operations to add asynchronous functionality of the `thread_control` class. It also contains the necessary declarations of extensions for the standard C++ STL streams I/O operations: output operator >> and input operator <<.

You can call synchronization method `wait()` to wait for completion of any I/O operations with the file object. If the `wait()` method is not called explicitly, it is called implicitly in the object destructor.

Public Interface of Template Class async_class

The following methods define the public interface of the template class `async_class`:

- `get_last_operation_id()`
- `wait()`
- `get_status()`
- `get_last_error()`
- `get_error_operation_id()`
- `stop_queue()`
- `resume_queue()`
- `clear_queue()`

Library Restrictions

Intel's C++ asynchronous I/O template class does not control the integrity or validity of the objects during asynchronous operation. Such control should be done by the user.

For application stability in the Visual Studio 2003 environment, link the C++ part of `libacaio.lib` library with multi-threaded `msvcrt` run-time library. Use `/MT` or `/MTd` compiler option.

See Also

Example of Using async_class Template Class

get_last_operation_id

Returns ID of the last added operation.

Syntax

```
void get_last_operation_id(void)
```

Description

This method returns the ID of the last added operation. Use this ID to get the status of operation or to wait for the operation to complete.

Return Values

Nothing

wait

Stops execution of current thread.

Syntax

```
int wait(void)
```

```
int wait(unsigned int operation_id)
```

Description

Method `wait(void)` stops execution of the current thread until all the asynchronous operations are completed.

Method `wait(operation_id)` stops execution of the current thread until the operation identified by *operation_id* is completed.

Return Values

-1 : On error during queue execution

Call the `get_last_error()` method to check the error code.

get_status

Returns status of specified operation.

Syntax

```
void get_status(unsigned int operation_id)
```

Description

This method returns the status of an operation, specified by *operation_id,* without stopping current thread execution.

Return Values

STATUS_WAIT: Operation is waiting for execution.

STATUS_COMPLETED: Operation finished execution.

STATUS_ERROR: An error occurred during operation execution.

STATUS_EXECUTE: Operation is executing.

STATUS_BLOCKED: Execution of the queue was blocked after some earlier errors.

get_last_error

Returns the error code of the last failed operation.

Syntax

```
unsigned int get_last_error()
```

Description

This method returns the error code of the last failed operation. If the error occurs during the execution of an asynchronous operation, the asynchronous thread stops executing the queue of asynchronous operations and waits for new user requests.

To obtain the error status, use the `wait()` and `get_status()` methods.

Return Values

Error code of last failed operation.

This error code is equal to the value returned by `GetLastError()` function on the Windows* platform. If the error occurs during the execution of an asynchronous operation, the asynchronous thread stops executing the queue of asynchronous operations and waits for new user requests.

get_error_operation_id

Returns the ID of the last failed operation.

Syntax

```
unsigned int get_error_operation_id()
```

Description

This method returns the ID of the last failed operation. If the error occurs during the execution of an asynchronous operation, the asynchronous thread stops executing the queue of the asynchronous operations and waits for new user requests.

To obtain the error status of the failed operation, use the `wait()` and `get_status()` methods.

Return Values

ID of last failed operation.

stop_queue

Stops queue execution.

Syntax

```
int stop_queue()
```

Description

This method allows you to control the asynchronous operations queue by stopping queue execution.

Return Values

0: On success

-1: On error

resume_queue

Resumes queue execution.

Syntax

```
int resume_queue()
```

Description

This method allows you to control the asynchronous operations queue by resuming queue execution.

Return Values

0: On success

-1: On error

clear_queue

Clears stopped or error-interrupted queues.

Syntax

```
void push_back_operation(class base_operation*)
```

Description

This method clears the content of stopped queues or queues interrupted by errors.

Return Values

0: On success

-1: On error

Example for Using async_class Template Class

The following example illustrates how Intel's C++ asynchronous I/O template class can be used. Consider the following code that writes arrays of floats to an external file.

```
// Data is array of floats
std::vector<float>  v(10000);

// User defines new operator << for std::vector<float> type
std::ofstream& operator << (std::ofstream & str, std::vector<float> & vec)
{
// User's output actions
...
 }
...
// Output file declaration - object of standard ofstream STL class
std::ofstream external_file("output.txt");
...
// Output operations
external_file << v;
```

The following code illustrates the changes to be made to the above code to execute the output operation asynchronously.

```
// Add new header to support STL asynchronous IO operations

#include <aiostream.h>
...

std::vector<float>  v(10000);

std::ofstream& operator << (std::ofstream & str, std::vector<float> & vec)
{... }
...
// Declare output file as the instance of new async::async_class template
// class.
// New inherited from STL ofstream type is declared
async::async_class<std::ofstream> external_file("output.txt");
...
external_file << v;
...
// Add stop operation, to wait the completion of all asynchronous IO //operations
external_file.wait();
…
```

Performance Recommendations

It is recommended not to use asynchronous mode for small objects. For example, do not use asynchronous mode when the output standard type value in a loop where execution of other loop operations takes less time than output of the same value to the STL stream.

However, if you can find the balance between output of small data and its previous calculation inside the loop, you still have some stable performance improvement.

For example, in the following code, the program reads two matrices from external files, calculates the elements of a third matrix, and prints out the elements inside the loop.

```
#define ARR_LEN 900
{
  std::ifstream fA("A.txt");
  fA >> A;
  std::ifstream fB("B.txt");
```

```
  fB >> B;
  std::ofstream fC(f);

  for(int i=0; i< ARR_LEN; i++)
   {
    for(int j=0; j< ARR_LEN; j++)
    {
     C[i][j] = 0;
     for(int k=0; k < ARR_LEN; k++)
     C[i][j]+ = A[i][k]*B[k][j]*sin((float)(k))*cos((float)(-k))*sin((float)(k+1)
     )*cos((float)(-k-1));
     fC << C[i][j] << std::endl;
    }
   }
}
```

By increasing matrix size, you can also achieve performance improvement during parallel data reading from two files.

IEEE 754-2008 Binary Floating-Point Conformance Library

The Intel® IEEE 754-2008 Binary Floating-Point Conformance Library provides all operations mandated by the IEEE 754-2008 standard for binary32 and binary64 binary floating-point interchange formats.

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Intel® IEEE 754-2008 Binary Floating-Point Conformance Library and Usage

The Intel® IEEE 754-2008 Binary Floating-Point Conformance Library provides all operations mandated by the IEEE 754-2008 standard for binary32 and binary64 binary floating-point interchange formats. The minimum requirements for correct operation of the library are an Intel® Pentium® 4 processor and an operating system supporting Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instructions.

The library supports all four rounding-direction attributes mandated by the IEEE 754-2008 standard for binary floating-point arithmetic: `roundTiesToEven`, `roundTowardPositive`, `roundTowardNegative`, `roundTowardZero`. The additional rounding-direction attribute, `roundTiesToAway`, is not required by the standard, hence, not fully supported in this library. The default rounding-direction attribute is set as `roundTiesToEven`.

The library also supports all mandated exceptions (invalid operation, division by zero, overflow, underflow, and inexact) and sets flags accordingly under default exception handling. Alternate exception handling, which is optional in the standard, is not supported.

The `bfp754.h` header file includes prototypes for the library functions. For a complete list of the functions available, refer to the Function List. The user also needs to specify linker option `-lbfp754` and floating-point semantics control option `-fp-model strict` in order to use the library.

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Operations

The IEEE standard 754-2008 defines four types of operations.

1. General-computational operations that produce correctly rounded floating-point or integer results. These operations might signal the floating-point exceptions.
2. Quiet-computational operations that produce floating-point results. These operations do not signal any floating-point exceptions.
3. Signaling-computational operations that produce no floating-point results. These operations might signal floating-point exceptions.
4. Non-computational operations that produce no floating-point results. These operations do not signal floating-point exceptions.

	Produce result	**Produce no result**
Might signal FP exception	General-computational	Signaling-computational
Do not signal FP exception	Quiet-computational	Non-computational

The standard also distinguishes among operations by their floating-point operand formats and result format for general-computational operations:

1. Homogenous general-computational operations whose floating-point operands and floating-point result are in the same format.
2. *formatOf* general-computational operations whose floating-point operands and floating-point result have different formats.

NOTE
The IEEE 754-2008 standard requires that all *formatOf* general-computational operations be computed without any loss of precision before converting to the destination format. This may differ from how these operations are implemented on most hardware and software.

For example, when all operands are in binary64 format and the destination format is binary32, most hardware and software implementations would first compute an intermediate result rounded in binary64 and then convert the intermediate result to binary32. This double rounding procedure may produce a result different from what is defined in the standard under certain rounding mode. For example: `x = 0x3ff0000010000000 = 1.000000000000000000000001_2`, `y = 0x3ca0000000000000 = 1.0_2*2^(-53) x+y = 1.00000000000000000000000100000000000000000000000000001_2`

When the rounding-direction attribute is set to `roundTiesToEven`, using double rounding procedure, the addition result rounds to `1.000000000000000000000001_2 (0x3ff0000010000000)` in binary64, which would then round to `1 (0x3f800000)` in binary32. On the other hand, according to the standard, the addition result should round to `1.00000000000000000000001_2 (0x3f800001)` in binary32.

Data Types

The following table correlates the names of the formats used in defining operations in the standard with their C99 types used in this library.

Format Name	**Definition**	**C99 Type**
`binary32`	IEEE 754-2008 binary32 interchange format	`float`

Format Name	Definition	C99 Type
`binary64`	IEEE 754-2008 binary64 interchange format	`double`
`int`	Integer operand formats	`int, unsigned int, long long int, unsigned long long int`
`int32`	Signed 32-bit integer	`int`
`uint32`	Unsigned 32-bit integer	`unsigned int`
`int64`	Signed 64-bit integer	`long long int`
`uint64`	Unsigned 64-bit integer	`unsigned long long int`
`boolean`	Boolean value represented by generic integer type	`int`
`enum`	Enumerated values of floating-point class	`int`
	Enumerated values of floating-point radix	`int`
`logBFormat`	Type for the destination of the `logB` operation and the scale exponent operand of the `scaleB` operation	`int`
`decimalCharacterSequence`	Decimal character sequence	`char*`
`hexCharacterSequence`	Hexadecimal-significand character sequence	
`exceptionGroup`	Set of exceptions as a set of booleans	`int`
`flags`	Set of status flags	`int`
`binaryRoundingDirection`	Rounding direction for binary	`int`
`modeGroup`	Dynamically-specifiable modes	`int`
`void`	No explicit operand or result	`void`

Use the Intel® IEEE 754-2008 Binary Floating-Point Conformance Library

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

To use the library, include the header file, `bfp754.h`, in your program.

Here is an example program illustrating the use of the library on Linux* OS.

```
//binary.c
#include <stdio.h>
#include <bfp754.h>
```

```
int main(){
  double a64, b64;
  float c32;
  a64 = 1.000000059604644775390625;
  b64 = 1.1102230246251565404236316680908203125e-16;
  c32 = __binary32_add_binary64_binary64(a64, b64);
  printf("The addition result using the libary: %8.8f\n", c32);
  c32 = a64 + b64;
  printf("The addition result without the libary: %8.8f\n", c32);
  return 0;
}
```

To compile `binary.c`, use the command:

```
icc -fp-model source -fp-model except binary.c -lbfp754
```

The output of `a.out` will look similar to the following:

```
The addition result using the libary: 1.00000012
The addition result without the libary: 1.00000000
```

See Also

Function List

Function List

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for homogeneous general-computational operations:

Function Group	Function	IEEE standard equivalent
Homogeneous General-Computational Operations Functions	`ilogb`	`logB`
	`maxnum`	`maxNum`
	`maxnum_mag`	`maxNumMag`
	`minnum`	`minNum`
	`minnum_mag`	`minNumMag`
	`next_down`	`nextDown`
	`next_up`	`nextUp`
	`rem`	`remainder`
	`round_integral_exact`	`roundToIntegralExact`
	`round_integral_nearest_away`	`roundToIntegralTiesToAway`
	`round_integral_nearest_even`	`roundToIntegralTiesToEven`
	`round_integral_negative`	`roundToIntegralTowardNegative`
	`round_integral_positive`	`roundToIntegralTowardPositive`
	`round_integral_zero`	`roundToIntegralTowardZero`
	`scalbn`	`scaleB`
General-Computational Operations Functions	`add`	`addition`

Function Group	Function	IEEE standard equivalent
	binary32_to_binary64 binary64_to_binary32	convertFormat
	div	division
	fma	fusedMultiplyAdd
	from_int32 from_uint32 from_int64 from_uint64	convert
	from_hexstring	convertFromHexCharacter
	from_string	convertFromDecimalCharacter
	mul	multiplication
	sqrt	squareRoot
	sub	subtraction
	to_hexstring	convertToHexCharacter
	to_int32_ceil to_uint32_ceil to_int64_ceil to_uint64_ceil	convertToIntegerTowardPositive
	to_int32_floor to_uint32_floor to_int64_floor to_uint64_floor	convertToIntegerTowardNegative
	to_int32_int to_uint32_int to_int64_int to_uint64_int	convertToIntegerTowardZero
	to_int32_rnint to_uint32_rnint to_int64_rnint to_uint64_rnint	convertToIntegerTiesToEven
	to_int32_xrnint to_uint32_xrnint to_int64_xrnint to_uint64_xrnint	convertToIntegerExactTiesToEven
	to_int32_rninta	convertToIntegerTiesToAway

Function Group	Function	IEEE standard equivalent
	to_uint32_rninta to_int64_rninta to_uint64_rninta	
	to_int32_xceil to_uint32_xceil to_int64_xceil to_uint64_xceil	convertToIntegerExactTowardPositive
	to_int32_xfloor to_uint32_xfloor to_int64_xfloor to_uint64_xfloor	convertToIntegerExactTowardNegative
	to_int32_xint to_uint32_xint to_int64_xint to_uint64_xint	convertToIntegerExactTowardZero
	to_int32_xrninta to_uint32_xrninta to_int64_xrninta to_uint64_xrninta	convertToIntegerExactTiesToAway
	to_string	convertToDecimalCharacter
Quiet-Computational Operations Functions	abs	abs
	copy	copy
	copysign	copySign
	negate	negate
Signaling-Computational Operations Functions	quiet_equal	compareQuietEqual
	quiet_greater	compareQuietGreater
	quiet_greater_equal	compareQuietGreaterEqual
	quiet_greater_unordered	compareQuietGreaterUnordered
	quiet_less	compareQuietLess
	quiet_less_equal	compareQuietLessEqual
	quiet_less_unordered	compareQuietLessUnordered
	quiet_not_equal	compareQuietNotEqual
	quiet_not_greater	compareQuietNotGreater
	quiet_not_less	compareQuietNotLess
	quiet_ordered	compareQuietOrdered
	quiet_unordered	compareQuietUnordered
	signaling_equal	compareSignalingEqual
	signaling_greater	compareSignalingGreater
	signaling_greater_equal	compareSignalingGreaterEqual

Function Group	Function	IEEE standard equivalent
	signaling_greater_unordered	compareSignalingGreaterUnordered
	signaling_less	compareSignalingLess
	signaling_less_equal	compareSignalingLessEqual
	signaling_less_unordered	compareSignalingLessUnordered
	signaling_not_equal	compareSignalingNotEqual
	signaling_not_greater	compareSignalingNotGreater
	signaling_not_less	compareSignalingNotLess
Non-Computational Operations Functions	class	class
	defaultMode	defaultModes
	getBinaryRoundingDirection	getBinaryRoundingDirection
	is754version1985	is754version1985
	is754version2008	is754version2008
	isCanonical	isCanonical
	isFinite	isFinite
	isInfinite	isInfinite
	isNaN	isNaN
	isNormal	isNormal
	isSignaling	isSignaling
	isSignMinus	isSignMinus
	isSubnormal	isSubnormal
	isZero	isZero
	lowerFlags	lowerFlags
	radix	radix
	raiseFlags	raiseFlags
	restoreFlags	restoreFlags
	restoreModes	restoreModes
	saveFlags	saveAllFlags
	saveModes	saveModes
	setBinaryRoundingDirection	setBinaryRoundingDirection
	testFlags	testFlags
	testSavedFlags	testSavedFlags
	totalOrder	totalOrder
	totalOrderMag	totalOrderMag

Homogeneous General-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for homogeneous general-computational operations:

round_integral_nearest_even

Description: The function rounds floating-point number `x` to its nearest integral value, with the halfway (tied) case rounding to even.

Calling interface:

```
float __binary32_round_integral_nearest_even(float x);
double __binary64_round_integral_nearest_even(double x);
```

round_integral_nearest_away

Description: The function rounds floating-point number `x` to its nearest integral value, with the halfway (tied) case rounding away from zero.

Calling interface:

```
float __binary32_round_integral_nearest_away(float x);
double __binary64_round_integral_nearest_away(double x);
```

round_integral_zero

Description: The function rounds floating-point number `x` to the closest integral value toward zero.

Calling interface:

```
float __binary32_round_integral_zero(float x);
double __binary64_round_integral_zero(double x);
```

round_integral_positive

Description: The function rounds floating-point number `x` to the closest integral value toward positive infinity.

Calling interface:

```
float __binary32_round_integral_positive(float x);
double __binary64_round_integral_positive(double x);
```

round_integral_negative

Description: The function rounds floating-point number `x` to the closest integral value toward negative infinity.

Calling interface:

```
float __binary32_round_integral_negative(float x);
double __binary64_round_integral_negative(double x);
```

round_integral_exact

Description: The function rounds floating-point number `x` to the closest integral value according to the rounding-direction applicable.

Calling interface:

```
float __binary32_round_integral_exact(float x);
double __binary64_round_integral_exact(double x);
```

next_up

Description: The function returns the least floating-point number in the same format as `x` that is greater than `x`.

Calling interface:

```
float __binary32_next_up(float x);
```

```
double __binary64_next_up(double x);
```

next_down

Description: The function returns the largest floating-point number in the same format as `x` that is less than `x`.

Calling interface:

```
float __binary32_next_down(float x);
double __binary64_next_down(double x);
```

rem

Description: The function returns the remainder of `x` and `y`.

Calling interface:

```
float __binary32_rem(float x, float y);
double __binary64_rem(double x, double y);
```

minnum

Description: The function returns the minimal value of `x` and `y`.

Calling interface:

```
float __binary32_minnum(float x, float y);
double __binary64_minnum(double x, double y);
```

maxnum

Description: The function returns the maximal value of `x` and `y`.

Calling interface:

```
float __binary32_maxnum(float x, float y);
double __binary64_maxnum(double x, double y);
```

minnum_mag

Description: The function returns the minimal absolute value of `x` and `y`.

Calling interface:

```
float __binary32_minnum_mag(float x, float y);
double __binary64_minnum_mag(double x, double y);
```

maxnum_mag

Description: The function returns the maximal absolute value of `x` and `y`.

Calling interface:

```
float __binary32_maxnum_mag(float x, float y);
double __binary64_maxnum_mag(double x, double y);
```

scalbn

Description: The function computes x X 2^n for integer value `n`.

Calling interface:

```
float __binary32_scalbn(float x, int n);
double __binary64_scalbn(double x, int n);
```

ilogb

Description: The function returns the exponent part of `x` as integer.

Calling interface:

```
int __binary32_ilogb(float x);
int __binary64_ilogb(double x);
```

formatOf General-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for *formatOf* general-computational operations:

add

Description: The function computes the addition of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_add_binary32_binary32(float x, float y);
float __binary32_add_binary32_binary64(float x, double y);
float __binary32_add_binary64_binary32(double x, float y);
float __binary32_add_binary64_binary64(double x, double y);
double __binary64_add_binary32_binary32(float x, float y);
double __binary64_add_binary32_binary64(float x, double y);
double __binary64_add_binary64_binary32(double x, float y);
double __binary64_add_binary64_binary64(double x, double y);
```

sub

Description: The function computes the subtraction of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_sub_binary32_binary32(float x, float y);
float __binary32_sub_binary32_binary64(float x, double y);
float __binary32_sub_binary64_binary32(double x, float y);
float __binary32_sub_binary64_binary64(double x, double y);
double __binary64_sub_binary32_binary32(float x, float y);
double __binary64_sub_binary32_binary64(float x, double y);
double __binary64_sub_binary64_binary32(double x, float y);
double __binary64_sub_binary64_binary64(double x, double y);
```

mul

Description: The function computes the multiplication of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_mul_binary32_binary32(float x, float y);
float __binary32_mul_binary32_binary64(float x, double y);
float __binary32_mul_binary64_binary32(double x, float y);
float __binary32_mul_binary64_binary64(double x, double y);
double __binary64_mul_binary32_binary32(float x, float y);
```

```
double __binary64_mul_binary32_binary64(float x, double y);
double __binary64_mul_binary64_binary32(double x, float y);
double __binary64_mul_binary64_binary64(double x, double y);
```

div

Description: The function computes the division of two floating-point numbers; the result is then converted to the destination format.

Calling interface:

```
float __binary32_div_binary32_binary32(float x, float y);
float __binary32_div_binary32_binary64(float x, double y);
float __binary32_div_binary64_binary32(double x, float y);
float __binary32_div_binary64_binary64(double x, double y);
double __binary64_div_binary32_binary32(float x, float y);
double __binary64_div_binary32_binary64(float x, double y);
double __binary64_div_binary64_binary32(double x, float y);
double __binary64_div_binary64_binary64(double x, double y);
```

sqrt

Description: The function computes the square root of floating-point number; the result is then converted to the destination format.

Calling interface:

```
float __binary32_sqrt_binary32(float x);
float __binary32_sqrt_binary64(double x);
double __binary32_sqrt_binary32(float x);
double __binary32_sqrt_binary64(double x);
```

fma

Description: The function computes the fused multiply and add of three floating-point numbers *x*, *y*, and *z* as (*x*×*y*) +*z*; the result is then converted to the destination format.

Calling interface:

```
float __binary32_fma_binary32_binary32_binary32(float x, float y, float z);
float __binary32_fma_binary32_binary32_binary64(float x, float y, double z);
float __binary32_fma_binary32_binary64_binary32(float x, double y, float z);
float __binary32_fma_binary32_binary64_binary64(float x, double y, double z);
float __binary32_fma_binary64_binary32_binary32(double x, float y, float z);
float __binary32_fma_binary64_binary32_binary64(double x, float y, double z);
float __binary32_fma_binary64_binary64_binary32(double x, double y, float z);
float __binary32_fma_binary64_binary64_binary64(double x, double y, double z);
double __binary64_fma_binary32_binary32_binary32(float x, float y, float z);
double __binary64_fma_binary32_binary32_binary64(float x, float y, double z);
double __binary64_fma_binary32_binary64_binary32(float x, double y, float z);
double __binary64_fma_binary32_binary64_binary64(float x, double y, double z);
double __binary64_fma_binary64_binary32_binary32(double x, float y, float z);
double __binary64_fma_binary64_binary32_binary64(double x, float y, double z);
double __binary64_fma_binary64_binary64_binary32(double x, double y, float z);
double __binary64_fma_binary64_binary64_binary64(double x, double y, double z);
```

from_int32 / from_uint32 / from_int64 / from_uint64

Description: This function converts integral values in the specified integer format to floating-point number.

Calling interface:

```
float __binary32_from_int32(int n);
double __binary64_from_int32(int n);
float __binary32_from_uint32(unsigned int n);
double __binary64_from_uint32(unsigned int n);
float __binary32_from_int64(long long int n);
double __binary64_from_int64(long long int n);
float __binary32_from_uint64(unsigned long long int n);
double __binary64_from_uint64(unsigned long long int n);
```

to_int32_rnint / to_uint32_rnint / to_int64_rnint / to_uint64_rnint

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded to even, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_rnint(float x);
int __binary64_to_int32_rnint(double x);
unsigned int __binary32_to_uint32_rnint(float x);
unsigned int __binary64_to_uint32_rnint(double x);
long long int __binary32_to_int64_rnint(float x);
long long int __binary64_to_int64_rnint(double x);
unsigned long long int __binary32_to_uint64_rnint(float x);
unsigned long long int __binary64_to_uint64_rnint(double x);
```

to_int32_int / to_uint32_int / to_int64_int / to_uint64_int

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward zero, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_int(float x);
int __binary64_to_int32_int(double x);
unsigned int __binary32_to_uint32_int(float x);
unsigned int __binary64_to_uint32_int(double x);
long long int __binary32_to_int64_int(float x);
long long int __binary64_to_int64_int(double x);
unsigned long long int __binary32_to_uint64_int(float x);
unsigned long long int __binary64_to_uint64_int(double x);
```

to_int32_ceil/ to_uint32_ceil / to_int64_ceil / to_uint64_ceil

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward positive infinity, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_ceil(float x);
int __binary64_to_int32_ceil(double x);
unsigned int __binary32_to_uint32_ceil(float x);
unsigned int __binary64_to_uint32_ceil(double x);
long long int __binary32_to_int64_ceil(float x);
long long int __binary64_to_int64_ceil(double x);
```

```
unsigned long long int __binary32_to_uint64_ceil(float x);
unsigned long long int __binary64_to_uint64_ceil(double x);
```

to_int32_floor/ to_uint32_floor / to_int64_floor / to_uint64_floor

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward negative infinity, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_floor(float x);
int __binary64_to_int32_floor(double x);
unsigned int __binary32_to_uint32_floor(float x);
unsigned int __binary64_to_uint32_floor(double x);
long long int __binary32_to_int64_floor(float x);
long long int __binary64_to_int64_floor(double x);
unsigned long long int __binary32_to_uint64_floor(float x);
unsigned long long int __binary64_to_uint64_floor(double x);
```

to_int32_rninta / to_uint32_rninta / to_int64_rninta / to_uint64_rninta

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded away from zero, without signaling the inexact exception.

Calling interface:

```
int __binary32_to_int32_rninta(float x);
int __binary64_to_int32_rninta(double x);
unsigned int __binary32_to_uint32_rninta(float x);
unsigned int __binary64_to_uint32_rninta(double x);
long long int __binary32_to_int64_rninta(float x);
long long int __binary64_to_int64_rninta(double x);
unsigned long long int __binary32_to_uint64_rninta(float x);
unsigned long long int __binary64_to_uint64_rninta(double x);
```

to_int32_xrnint / to_uint32_xrnint / to_int64_xrnint / to_uint64_xrnint

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded to even, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xrnint(float x);
int __binary64_to_int32_xrnint(double x);
unsigned int __binary32_to_uint32_xrnint(float x);
unsigned int __binary64_to_uint32_xrnint(double x);
long long int __binary32_to_int64_xrnint(float x);
long long int __binary64_to_int64_xrnint(double x);
unsigned long long int __binary32_to_uint64_xrnint(float x);
unsigned long long int __binary64_to_uint64_xrnint(double x);
```

to_int32_xint / to_uint32_xint / to_int64_xint / to_uint64_xint

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward zero, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xint(float x);
```

```
int __binary64_to_int32_xint(double x);
unsigned int __binary32_to_uint32_xint(float x);
unsigned int __binary64_to_uint32_xint(double x);
long long int __binary32_to_int64_xint(float x);
long long int __binary64_to_int64_xint(double x);
unsigned long long int __binary32_to_uint64_xint(float x);
unsigned long long int __binary64_to_uint64_xint(double x);
```

to_int32_xceil / to_uint32_xceil / to_int64_xceil / to_uint64_xceil

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward positive infinity, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xceil(float x);
int __binary64_to_int32_xceil(double x);
unsigned int __binary32_to_uint32_xceil(float x);
unsigned int __binary64_to_uint32_xceil(double x);
long long int __binary32_to_int64_xceil(float x);
long long int __binary64_to_int64_xceil(double x);
unsigned long long int __binary32_to_uint64_xceil(float x);
unsigned long long int __binary64_to_uint64_xceil(double x);
```

to_int32_xfloor / to_uint32_xfloor / to_int64_xfloor / to_uint64_xfloor

Description: This function rounds floating-point number to the nearest integral value in the specified integer format toward negative infinity, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xfloor(float x);
int __binary64_to_int32_xfloor(double x);
unsigned int __binary32_to_uint32_xfloor(float x);
unsigned int __binary64_to_uint32_xfloor(double x);
long long int __binary32_to_int64_xfloor(float x);
long long int __binary64_to_int64_xfloor(double x);
unsigned long long int __binary32_to_uint64_xfloor(float x);
unsigned long long int __binary64_to_uint64_xfloor(double x);
```

to_int32_xrninta / to_uint32_xrninta / to_int64_xrninta / to_uint64_xrninta

Description: This function rounds floating-point number to the nearest integral value in the specified integer format, with halfway cases rounded away from zero, signaling if inexact.

Calling interface:

```
int __binary32_to_int32_xrninta(float x);
int __binary64_to_int32_xrninta(double x);
unsigned int __binary32_to_uint32_xrninta(float x);
unsigned int __binary64_to_uint32_xrninta(double x);
long long int __binary32_to_int64_xrninta(float x);
long long int __binary64_to_int64_xrninta(double x);
unsigned long long int __binary32_to_uint64_xrninta(float x);
unsigned long long int __binary64_to_uint64_xrninta(double x);
```

binary32_to_binary64

Description: This function converts floating-point number in binary32 format to binary64 format.

Calling interface:

```
double __binary32_to_binary64(float x);
```

binary64_to_binary32

Description: This function rounds floating-point number in binary64 format to binary32 format.

Calling interface:

```
float __binary64_to_binary32(double x);
```

from_string

Description: This function converts decimal character sequence to floating-point number.

Calling interface:

```
float __binary32_from_string(char * s);
double __binary64_from_string(char * s);
```

to_string

Description: This function converts floating-point number to decimal character sequence.

Calling interface:

```
char *__binary32_to_string(float x);
char *__binary64_to_string(double x);
```

from_hexstring

Description: This function converts hexadecimal character sequence to floating-point number.

Calling interface:

```
float __binary32_from_hexstring(char * s);
double __binary64_from_hexstring(char * s);
```

to_hexstring

Description: This function converts floating-point number to hexadecimal character sequence.

Calling interface:

```
char *__binary32_to_hexstring(float x);
char *__binary64_to_hexstring(double x);
```

Quiet-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for quiet-computational operations:

copy

Description: The function copies input floating-point number `x` to output in the same floating-point format, without any change to the sign.

Calling interface:

```
float __binary32_copy(float x);
double __binary64_copy(double x);
```

NOTE
When the input is a signaling NaN, two different outcomes are allowed by the standard. The operation could either signal invalid exception with quieted signaling NaN as output, or deliver signaling NaN as output without signaling any exception.

negate

Description: The function copies input floating-point number x to output in the same floating-point format, reversing the sign.

Calling interface:
```
float __binary32_negate(float x);
double __binary64_negate(double x);
```

NOTE
When the input is a signaling NaN, two different outcomes are allowed by the standard. The operation could either signal invalid exception with quieted signaling NaN as output, or deliver signaling NaN as output without signaling any exception.

abs

Description: The function copies input floating-point number x to output in the same floating-point format, setting the sign to positive.

Calling interface:
```
float __binary32_abs(float x);
double __binary64_abs(double x);
```

NOTE
When the input is a signaling NaN, two different outcomes are allowed by the standard. The operation could either signal invalid exception with quieted signaling NaN as output, or deliver signaling NaN as output without signaling any exception.

copysign

Description: The function copies input floating-point number x to output in the same floating-point format, with the same sign as y.

Calling interface:
```
float __binary32_copysign(float x, float y);
double __binary64_copysign(double x, double y);
```

NOTE
When the first input is a signaling NaN, two different outcomes are allowed by the standard. The operation could either signal invalid exception with quieted signaling NaN as output, or deliver signaling NaN as output without signaling any exception.

Signaling-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for signaling-computational operations:

quiet_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is in the inputs.

Calling interface:

```
int __binary32_quiet_equal_binary32 (float x, float y);
int __binary32_quiet_equal_binary64(float x, double y);
int __binary64_quiet_equal_binary32(double x, float y);
int __binary64_quiet_equal_ binary64(double x, double y);
```

quiet_not_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_not_equal_binary32(float x, float y);
int __binary32_quiet_not_equal_binary64(float x, double y);
int __binary64_quiet_not_equal_binary32(double x, float y);
int __binary64_quiet_not_equal_binary64(double x, double y);
```

signaling_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_equal_binary32(float x, float y);
int __binary32_signaling_equal_binary64(float x, double y);
int __binary64_signaling_equal_binary32(double x, float y);
int __binary64_signaling_equal_binary64(double x, double y);
```

signaling_greater

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_greater_binary32(float x, float y);
int __binary32_signaling_greater_binary64(float x, double y);
int __binary64_signaling_greater_binary32(double x, float y);
int __binary64_signaling_greater_binary64(double x, double y);
```

signaling_greater_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater or equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_greater_equal_binary32(float x, float y);
int __binary32_signaling_greater_equal_binary64(float x, double y);
int __binary64_signaling_greater_equal_binary32(double x, float y);
int __binary64_signaling_greater_equal_binary64(double x, double y);
```

signaling_less

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_less_binary32(float x, float y);
int __binary32_signaling_less_binary64(float x, double y);
int __binary64_signaling_less_binary32(double x, float y);
int __binary64_signaling_less_binary64(double x, double y);
```

signaling_less_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less or equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_less_equal_binary32(float x, float y);
int __binary32_signaling_less_equal_binary64(float x, double y);
int __binary64_signaling_less_equal_binary32(double x, float y);
int __binary64_signaling_less_equal_binary64(double x, double y);
```

signaling_not_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not equal, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_not_equal_binary32(float x, float y);
int __binary32_signaling_not_equal_binary64(float x, double y);
int __binary64_signaling_not_equal_binary32(double x, float y);
int __binary64_signaling_not_equal_binary64(double x, double y);
```

signaling_not_greater

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not greater, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_not_greater_binary32(float x, float y);
int __binary32_signaling_not_greater_binary64(float x, double y);
int __binary64_signaling_not_greater_binary32(double x, float y);
int __binary64_signaling_not_greater_binary64(double x, double y);
```

signaling_less_unordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_less_unordered_binary32(float x, float y);
int __binary32_signaling_less_unordered_binary64(float x, double y);
int __binary64_signaling_less_unordered_binary32(double x, float y);
int __binary64_signaling_less_unordered_binary64(double x, double y);
```

signaling_not_less

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not less, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_not_less_ binary32(float x, float y);
int __binary32_signaling_not_less_binary64(float x, double y);
int __binary64_signaling_not_less_binary32(double x, float y);
int __binary64_signaling_not_less_binary64 (double x, double y);
```

signaling_greater_unordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when `NaN` is in the inputs.

Calling interface:

```
int __binary32_signaling_greater_unordered_binary32(float x, float y);
int __binary32_signaling_greater_unordered_binary64(float x, double y);
int __binary64_ signaling_greater_unordered_binary32(double x, float y);
int __binary64_signaling_greater_unordered_binary64(double x, double y);
```

quiet_greater

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_greater_binary32(float x, float y);
int __binary32_quiet_greater_binary64(float x, double y);
int __binary64_quiet_greater_binary32(double x, float y);
int __binary64_quiet_greater_binary64(double x, double y);
```

quiet_greater_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater or equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_greater_equal_binary32(float x, float y);
int __binary32_quiet_greater_equal_binary64(float x, double y);
int __binary64_quiet_greater_equal_binary32(double x, float y);
int __binary64_quiet_greater_equal_binary64(double x, double y);
```

quiet_less

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_less_binary32(float x, float y);
int __binary32_quiet_less_binary64(float x, double y);
int __binary64_quiet_less_binary32(double x, float y);
int __binary64_quiet_less_binary64(double x, double y);
```

quiet_less_equal

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less or equal, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_less_equal_binary32(float x, float y);
int __binary32_quiet_less_equal_binary64(float x, double y)
int __binary64_quiet_less_equal_binary32(double x, float y);
int __binary64_quiet_less_equal_binary64(double x, double y);
```

quiet_unordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is unordered, returns zero (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs

Calling interface:

```
int __binary32_quiet_unordered_binary32(float x, float y);
int __binary32_quiet_unordered_binary64(float x, double y);
int __binary64_quiet_unordered_binary32(double x, float y);
int __binary64_quiet_unordered_binary64(double x, double y);
```

quiet_not_greater

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not greater, returns zero (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_not_greater_binary32(float x, float y);
int __binary32_quiet_not_greater_binary64(float x, double y);
int __binary64_quiet_not_greater_binary32(double x, float y);
int __binary64_quiet_not_greater_binary64(double x, double y);
```

quiet_less_unordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is less or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_less_unordered_binary32(float x, float y);
int __binary32_quiet_less_unordered_binary64(float x, double y);
int __binary64_quiet_less_unordered_binary32(double x, float y);
int __binary64_quiet_less_unordered_binary64(double x, double y);
```

quiet_not_less

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is not less, returns zero (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_not_less_binary32(float x, float y);
int __binary32_quiet_not_less_binary64(float x, double y);
int __binary64_quiet_not_less_binary32(double x, float y);
int __binary64_quiet_not_less_binary64(double x, double y);
```

quiet_greater_unordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is greater or unordered, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_greater_unordered_binary32(float x, float y);
int __binary32_quiet_greater_unordered_binary64(float x, double y);
int __binary64_quiet_greater_unordered_binary32(double x, float y);
int __binary64_quiet_greater_unordered_binary64(double x, double y);
```

quiet_ordered

Description: The function returns 1 (true) if the relation between the two inputs `x` and `y` is ordered, returns 0 (false) otherwise. The function signals invalid operation exception when signaling `NaN` is one of the inputs.

Calling interface:

```
int __binary32_quiet_ordered_binary32(float x, float y);
int __binary32_quiet_ordered_binary64(float x, double y);
int __binary64_quiet_ordered_binary32(double x, float y);
int __binary64_quiet_ordered_binary64(double x, double y);
```

Non-Computational Operations Functions

Many routines in the *libbfp754* Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The Intel® IEEE 754-2008 Binary Conformance Library supports the following functions for non-computational operations:

is754version1985

Description: The function returns 1, if and only if this programming environment conforms to IEEE Std. 754-1985, otherwise returns 0.

Calling interface:

```
int __binary_is754version1985(void);
```

> **NOTE**
> This function in this library always returns 0.

is754version2008

Description: The function returns 1, if and only if this programming environment conforms to IEEE Std. 754-2008, otherwise returns 0.

Calling interface:

```
int __binary_is754version2008(void);
```

> **NOTE**
> This function in this library always returns 1.

class

Description: The function returns which class of the ten classes (`signalingNaN`, `quietNaN`, `negativeInfinity`, `negativeNormal`, `negativeSubnormal`, `negativeZero`, `positiveZero`, `positiveSubnormal`, `positiveNormal`, `positiveInfinity`) the input floating-point number `x` belongs.

Return value	Class
0	signalingNaN
1	quietNaN
2	negativeInfinity
3	negativeNormal
4	negativeSubnormal
5	negativeZero
6	positiveZero
7	positiveSubnormal
8	positiveNormal
9	positiveInfinity

Calling interface:

```
int __binary32_class(float x);
int __binary64_class(double x);
```

isSignMinus

Description: The function returns 1, if and only if its argument has negative sign.

Calling interface:

```
int __binary32_isSignMinus(float x);
int __binary64_isSignMinus(double x);
```

isNormal

Description: The function returns 1, if and only if its argument is normal (not `zero`, `subnormal`, `infinite`, or `NaN`).

Calling interface:

```
int __binary32_isNormal(float x);
```

```
int __binary64_isNormal(double x);
```

isFinite

Description: The function returns 1, if and only if its argument is `finite` (not `infinite` or `NaN`).

Calling interface:

isZero

Description: The function returns 1, if and only if its argument is ±0.

Calling interface:

```
int __binary32_isZero(float x);
int __binary64_isZero(double x);
```

isSubnormal

Description: The function returns 1, if and only if its argument is `subnormal`.

Calling interface:

```
int __binary32_isSubnormal(float x);
int __binary64_isSubnormal(double x);
```

isInfinite

Description: The function returns 1, if and only if its argument is `infinite`

Calling interface:

```
int __binary32_isInfinite(float x);
int __binary64_isInfinite(double x);
```

isNaN

Description: The function returns 1, if and only if its argument is a `NaN`.

Calling interface:

```
int __binary32_isNaN(float x);
int __binary64_isNaN(double x);
```

isSignaling

Description: The function returns 1, if and only if its argument is a signaling `NaN`.

Calling interface:

```
int __binary32_isSignaling(float x);
int __binary64_isSignaling(double x);
```

isCanonical

Description: The function returns 1, if and only if its argument is a finite number, `infinity`, or `NaN` that is canonical.

Calling interface:

```
int __binary32_isCanonical(float x);
int __binary64_isCanonical(double x);
```

NOTE
This function in this library always returns 1, as only canonical floating-point numbers are expected.

radix

Description: The function returns the radix of the format of the input floating-point number.

Calling interface:

```
int __binary32_radix(float x);
int __binary64_radix(double x);
```

NOTE
This function in this library always returns 2, as the library is intended for binary floating-point numbers.

totalOrder

Description: The function returns 1 if and only if two floating-point inputs `x` and `y` is total ordered and 0 otherwise.

Calling interface:

```
int _binary32_totalOrder(float x, float y);
int _binary64_totalOrder(double x, double y);
```

totalOrderMag

Description:`totalOrderMag(x, y)` is the same as `totalOrder(abs(x), abs(y))`.

Calling interface:

```
int _binary32_totalOrderMag(float x, float y);
int _binary64_totalOrderMag(double x, double y);
```

lowerFlags

Description: The function lowers the flags of the exception group specified by the input.

Value	Exception name
1	`__BFP754_INVALID`
2	`__BFP754_DIVBYZERO`
4	`__BFP754_OVERFLOW`
8	`__BFP754_UNDERFLOW`
16	`__BFP754_INEXACT`

Calling interface:

```
void __binary_lowerFlags(int x);
```

raiseFlags

Description: The function raises the flags of the exception group specified by the input.

Calling interface:

```
void __binary_raiseFlags(int x);
```

testFlags

Description: The function returns 1, if and only if any flag of the exception group specified by the input is raised, and 0 otherwise.

Calling interface:

```
int __binary_testFlags(int x);
```

testSavedFlags

Description: The function returns 1, if and only if any flag of the exception group specified by the input `y` is raised in `x`, and 0 otherwise.

Calling interface:

```
int __binary_testSavedFlags(int x, int y);
```

restoreFlags

Description: The function restores the flags to their states represented in `x`.

Calling interface:

```
void __binary_restoreFlags(int x);
```

saveFlags

Description: The function returns a representation of the state of all status flags.

Calling interface:

```
int __binary_saveFlags(void);
```

getBinaryRoundingDirection

Description: The function returns an integer representing the rounding direction in use.

Value	Exception name
0	`__BFP754_ROUND_TO_NEAREST_EVEN`
1	`__BFP754_ROUND_TOWARD_POSITIVE`
2	`__BFP754_ROUND_TOWARD_NEGATIVE`
3	`__BFP754_ROUND_TOWARD_ZERO`

Calling interface:

```
int __binary_getBinaryRoundingDirection(void);
```

setBinaryRoundingDirection

Description: The function sets the rounding direction based on input integer.

Calling interface:

```
void __binary_setBinaryRoundingDirection(int x);
```

saveModes

Description: The function saves the values of all dynamic-specifiable modes.

Calling interface:

```
int __binary_saveModes(void);
```

NOTE
`saveModes` behaves in the same way as `getBinaryRoundingDirection` does, as the rounding mode is the only dynamic-specifiable mode supported.

restoreModes

Description: The function restores the values of all dynamic-specifiable modes to the input.

Calling interface:
`int __binary_restoreModes(void);`

NOTE
`restoreModes` behaves in the same way as `setBinaryRoundingDirection` does, as the rounding mode is the only dynamic-specifiable mode supported.

defaultMode

Description: The function sets the values of all dynamic-specifiable modes to default.

Calling interface:
`void __binary_defaultMode(void);`

NOTE
`defaultMode` sets the rounding-direction attribute to `roundTiesToEven`, as the rounding mode is the only dynamic-specifiable mode supported.

Intel's Numeric String Conversion Library

Intel's Numeric String Conversion Library, `libistrconv`, provides a collection of routines for converting between ASCII strings and C data types, which are optimized for performance.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Use Intel's Numeric String Conversion Library

Intel's Numeric String Conversion Library, `libistrconv`, provides a collection of routines for converting between ASCII strings and C data types, which are optimized for performance. The `istrconv.h` header file declares prototypes for the library functions.

You can link the `libistrconv` library as a static or shared library on Linux* and macOS platforms.On Windows* platforms, you must link `libistrconv` as a static library only.

Using Intel's Numeric String Conversion Library

To use the `libistrconv` library, include the header file, `istrconv.h`, in your program.

Consider the following example `conv.c` file that illustrates how to use the library to convert between string and floating-point data type.

```
// conv.c
#include <stdio.h>
#include <istrconv.h>
#define LENGTH 20

int main() {
 const char pi[] = "3.14159265358979323";
 char s[LENGTH];
 int prec;
 float fx;
 double dx;
 printf("PI: %s\n", pi);
 printf("single-precision\n");
 fx = __IML_string_to_float(pi, NULL);
 prec = 6;
 __IML_float_to_string(s, LENGTH, prec, fx);
 printf("prec: %2d, val: %s\n", prec, s);
 printf("double-precision\n");
 dx = __IML_string_to_double(pi, NULL);
 prec = 15;
 __IML_double_to_string(s, LENGTH, prec, dx);
 printf("prec: %2d, val: %s\n", prec, s);
 return 0;
}
```

To compile the `conv.c` file with Intel's Numeric String Conversion Library (`libistrconv`) use one of the following commands. See Invoke the Compiler for information about all available drivers.

Linux and macOS

```
icpc conv.c -libistrconv
```

Windows

```
icl conv.c libistrconv.lib
```

After you compile this example and run the program, you should get the following results:

```
PI: 3.14159265358979323

single-precision
prec: 6, val: 3.14159

double-precision
prec: 15, val: 3.14159265358979
```

Integer Conversion Functions Optimized with SSE4.2 Instructions

The following integer conversion functions are optimized for better performance with SSE4.2 string processing instructions:

- `__IML_int_to_string`
- `__IML_uint_to_string`
- `__IML_int64_to_string`
- `__IML_uint64_to_string`
- `__IML_i_to_str`
- `__IML_u_to_str`
- `__IML_ll_to_str`

- `__IML_ull_to_str`
- `__IML_string_to_int`
- `__IML_string_to_uint`
- `__IML_string_to_int64`
- `__IML_string_to_uint64`
- `__IML_str_to_i`
- `__IML_str_to_u`
- `__IML_str_to_ll`
- `__IML_str_to_ull`

The SSE4.2 optimized versions of these functions can be deployed in the following situations:

- Used automatically on post-SSE4.2 processors through Intel run-time processor dispatching
- Called directly by defining the "__SSE4_2__" macro to the C preprocessor where <istrconv.h> is included.

The generic versions of these functions can be deployed in the following situations:

- Used automatically on pre-SSE4.2 processors through Intel run-time processor dispatching
- Called directly by adding _generic suffix to the function names

The SSE4.2 optimized versions of these functions moves strings from memory to XMM registers and vice versa directly to maximize performance. The functions would not overwrite the memory beyond the boundary; however, this may introduce memory access violation when the memory location immediately trailing the strings is not allocated or accessible. Users with concerns about potential memory access violation should use the generic versions instead.

Function List

Intel's Numeric String Conversion library (`libistrconv`) functions are listed in this topic.

Routines to convert floating-point numbers to ASCII strings

Intel's Numeric String Conversion Library supports the following functions to convert floating-point number *x* to string *s* in various formats, where *l* represents the length of the formatted string allowing for full conversion (not including the null terminator).

`__IML_float_to_string`, `__IML_double_to_string`

Description: These functions are similar to `snprintf(s, n, "%.*g", p, x)` in `stdio.h`, where *p* specifies the maximum number of significant digits in either fixed-point or exponential notation format. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the *(n-1)th* character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_float_to_string(char * s, size_t n, int p, float x);
```

```
int __IML_double_to_string(char * s, size_t n, int p, double x);
```

`__IML_float_to_string_f`, `__IML_double_to_string_f`

Description: These functions are similar to `snprintf(s, n, "%.*f", p, x)` in `stdio.h`, where *p* specifies the number of digits after the decimal point in the fixed-point notation format. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the *(n-1)th* character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_float_to_string_f(char * s, size_t n, int p, float x);
```

```
int __IML_double_to_string_f(char * s, size_t n, int p, double x);
```

`__IML_float_to_string_e`, `__IML_double_to_string_e`

Description: These functions are similar to `snprintf(s, n, "%.*e", p, x)` in `stdio.h`, where *p* specifies the number of digits after the decimal point in the exponential notation format. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the $(n\text{-}1)^{th}$ character are discarded and a null character is appended at the end. *l* is returned on success; otherwise, the result is undefined.

Calling interface:

```
int __IML_float_to_string_e(char * s, size_t n, int p, float x);
int __IML_double_to_string_e(char * s, size_t n, int p, double x);
```

`__IML_f_to_str`, `__IML_d_to_str`

Description: These functions are similar to `snprintf(s, n, "%.*g", p, x)` in `stdio.h`, where *p* specifies the maximum number of significant digits in either fixed-point or exponential notation format. If $l < n$, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written and *s* may be a null pointer. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_f_to_str(char * s, size_t n, int p, float x);
int __IML_d_to_str(char * s, size_t n, int p, double x);
```

`__IML_f_to_str_f`, `__IML_d_to_str_f`

Description: These functions are similar to `snprintf(s, n, "%.*f", p, x)` in `stdio.h`, where *p* specifies the number of digits after the decimal point in the fixed-point notation format. If $l < n$, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written and *s* may be a null pointer. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_f_to_str_f(char * s, size_t n, int p, float x);
int __IML_d_to_str_f(char * s, size_t n, int p, double x);
```

`__IML_f_to_str_e`, `__IML_d_to_str_e`

Description: These functions are similar to `snprintf(s, n, "%.*e", p, x)` in `stdio.h`, where *p*specifies the number of digits after the decimal point in the exponential notation format. If $l < n$, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written and *s* may be a null pointer. *l* is returned on success; otherwise the result is undefined.

Calling interface:

```
int __IML_f_to_str_e(char * s, size_t n, int p, float x);
int __IML_d_to_str_e(char * s, size_t n, int p, double x);
```

Routines to convert integers to ASCII strings

Intel's Numeric String Conversion Library supports the following functions to convert integer *x* to string *s*, where *l* represents the length of the formatted string allowing for full conversion (not including the null terminator).

`__IML_int_to_string`, `__IML_uint_to_string`, `__IML_int64_to_string`, `__IML_uint64_to_string`

Description: These functions are similar to `snprintf(s, n, "%[d|u|lld|llu]", x)` in `stdio.h`. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the *(n-1)th* character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

`int __IML_int_to_string(char * s, size_t n, int x);`

`int __IML_uint_to_string(char * s, size_t n, unsigned int x);`

`int __IML_int64_to_string(char * s, size_t n, long long x);`

`int __IML_uint64_to_string(char * s, size_t n, unsigned long long x);`

`__IML_int_to_oct_string`, `__IML_uint_to_oct_string`, `__IML_int64_to_oct_string`, `__IML_uint64_to_oct_string`

Description: These functions are similar to `snprintf(s, n, "%[o|llo]", x)` in `stdio.h`. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the *(n-1)th* character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

`int __IML_int_to_oct_string(char * s, size_t n, int x);`

`int __IML_uint_to_oct_string(char * s, size_t n, unsigned int x);`

`int __IML_int64_to_oct_string(char * s, size_t n, long long x);`

`int __IML_uint64_to_oct_string(char * s, size_t n, unsigned long long x);`

`__IML_int_to_hex_string`, `__IML_uint_to_hex_string`, `__IML_int64_to_hex_string`, `__IML_uint64_to_hex_string`

Description: These functions are similar to `snprintf(s, n, "%[x|llx]", x)` in `stdio.h`. If *n* is zero, nothing is written and *s* may be a null pointer. Output characters beyond the *(n-1)th* character are discarded and a null character is appended at the end. *l* is returned on success; otherwise the result is undefined.

Calling interface:

`int __IML_int_to_hex_string(char * s, size_t n, int x);`

`int __IML_uint_to_hex_string(char * s, size_t n, unsigned int x);`

`int __IML_int64_to_hex_string(char * s, size_t n, long long x);`

`int __IML_uint64_to_hex_string(char * s, size_t n, unsigned long long x);`

`__IML_i_to_str`, `__IML_u_to_str`, `__IML_ll_to_str`, `__IML_ull_to_str`

Description: These functions are similar to `snprintf(s, n, "%[d|u|lld|llu]", x)` in `stdio.h`. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the *nth* character are discarded and no null character is appended at the end. If *n* is zero, nothing is written, and *s* may be a null pointer. *l* is returned on success, otherwise the result is undefined.

Calling interface:

`int __IML_i_to_str(char * s, size_t n, int x);`

`int __IML_u_to_str(char * s, size_t n, unsigned int x);`

`int __IML_ll_to_str(char * s, size_t n, long long x);`

`int __IML_ull_to_str(char * s, size_t n, unsigned long long x);`

`__IML_i_to_oct_str`, `__IML_u_to_oct_str`, `__IML_ll_to_oct_str`, `__IML_ull_to_oct_str`

Description: These functions are similar to `snprintf(s, n, "%[o|llo]", x)` in `stdio.h`. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written, and *s* may be a null pointer. *l* is returned on success, otherwise the result is undefined.

Calling interface:

`int __IML_i_to_oct_str(char * s, size_t n, int x);`

`int __IML_u_to_oct_str(char * s, size_t n, unsigned int x);`

`int __IML_ll_to_oct_str(char * s, size_t n, long long x);`

`int __IML_ull_to_oct_str(char * s, size_t n, unsigned long long x);`

`__IML_i_to_hex_str`, `__IML_u_to_hex_str`, `__IML_ll_to_hex_str`, `__IML_ull_to_hex_str`

Description: These functions are similar to `snprintf(s, n, "%[x|llx]", x)` in `stdio.h`. If *l* < *n*, all output characters are stored in *s* with a null terminator at the end. Otherwise, output characters beyond the n^{th} character are discarded and no null character is appended at the end. If *n* is zero, nothing is written, and *s* may be a null pointer. *l* is returned on success, otherwise the result is undefined.

Calling interface:

`int __IML_i_to_hex_str(char * s, size_t n, int x);`

`int __IML_u_to_hex_str(char * s, size_t n, unsigned int x);`

`int __IML_ll_to_hex_str(char * s, size_t n, long long x);`

`int __IML_ull_to_hex_str(char * s, size_t n, unsigned long long x);`

Routines to convert ASCII strings to floating-point numbers

Intel's Numeric String Conversion Library supports the following functions to convert the initial portion of decimal string *s* to floating-point number *x*. If no conversion could be performed, zero is returned. If the correct value is outside the range of the return type, plus (+) or minus (-) `HUGE_VALF`, `HUGE_VAL`, or `HUGE_VALL` is returned, and the value of macro `ERANGE` is stored in `errno`.

`__IML_string_to_float`, `__IML_string_to_double`, `__IML_string_to_long_double`

Description: These functions are similar to `strtof(nptr, endptr)`, `strtod(nptr, endptr)`, and `strtold(nptr, endptr)` in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

`float __IML_string_to_float(const char * nptr, char ** endptr);`

`double __IML_string_to_double(const char * nptr, char ** endptr);`

`long double __IML_string_to_long_double(const char * nptr, char ** endptr);`

`__IML_str_to_f`, `__IML_str_to_d`, `__IML_str_to_ld`

Description: These functions convert the initial *n* decimal digits of the *significand* string multiplied by 10 raised to power of *exponent* to floating-point number as return. *endptr* points to the object that stores the final part of significand, provided that *endptr* is not a null pointer.

Calling interface:

`float __IML_str_to_f(const char * significand, size_t n, int exponent, char ** endptr);`

double __IML_str_to_d(*const char * significand*, *size_t n*, *int exponent*, *char ** endptr*);

long double __IML_str_to_ld(*const char * significand*, *size_t n*, *int exponent*, *char ** endptr*);

Routines to convert ASCII strings to integers

Intel's Numeric String Conversion Library supports the following functions to convert the initial portion of string *s* to integer *x*. If no conversion could be performed, zero is returned. If the correct value is outside the range of the return type, `INT_MIN`, `INT_MAX`, `UINT_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULLONG_MAX` is returned, and the value of macro `ERANGE` is stored in `errno`.

`__IML_string_to_int`, `__IML_string_to_uint`, `__IML_string_to_int64`, `__IML_string_to_uint64`

Description: These functions are similar to `([unsigned] int)strto[u]l(nptr, endptr, 10)` and `strto[u]ll(nptr, endptr, 10)` functions in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

int __IML_string_to_int(*const char * nptr*, *char ** endptr*);

unsigned int __IML_string_to_uint(*const char * nptr*, *char ** endptr*);

long long __IML_string_to_int64(*const char * nptr*, *char ** endptr*);

unsigned long long __IML_string_to_uint64(*const char * nptr*, *char ** endptr*);

`__IML_oct_string_to_int`, `__IML_oct_string_to_uint`, `__IML_oct_string_to_int64`, `__IML_oct_string_to_uint64`

Description: These functions are similar to `([unsigned] int)strto[u]l(nptr, endptr, 8)` and `strto[u]ll(nptr, endptr, 8)` functions in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

int __IML_oct_string_to_int(*const char * nptr*,*char ** endptr*);

unsigned int __IML_oct_string_to_uint(*const char * nptr*,*char ** endptr*);

long long __IML_oct_string_to_int64(*const char * nptr*,*char ** endptr*);

unsigned long long __IML_oct_string_to_uint64(*const char * nptr*,*char ** endptr*);

`__IML_hex_string_to_int`, `__IML_hex_string_to_uint`, `__IML_hex_string_to_int64`, `__IML_hex_string_to_uint64`

Description: These functions are similar to `([unsigned] int)strto[u]l(nptr, endptr, 16)` and `strto[u]ll(nptr, endptr, 16)` functions in `stdlib.h`, where *endptr* points to the object that stores the final part of *nptr* when *endptr* is not a null pointer.

Calling interface:

int __IML_hex_string_to_int(*const char * nptr*,*char ** endptr*);

unsigned int __IML_hex_string_to_uint(*const char * nptr*,*char ** endptr*);

long long __IML_hex_string_to_int64(*const char * nptr*,*char ** endptr*);

unsigned long long __IML_hex_string_to_uint64(*const char * nptr*,*char ** endptr*);

`__IML_str_to_i`, `__IML_str_to_u`, `__IML_str_to_ll`, `__IML_str_to_ull`

Description: These functions convert the initial *n* decimal digits (including an optional + or - sign) pointed to by *nptr* to integral values. When *endptr* is not a null pointer it points to the object that stores the final part of *nptr*. These functions treat any leading whitespace as invalid.

Calling interface:

```
int __IML_str_to_i(const char * nptr, size_t n, char ** endptr);
unsigned int __IML_str_to_u(const char * nptr, size_t n, char ** endptr);
long long __IML_str_to_ll(const char * nptr, size_t n, char ** endptr);
unsigned long long __IML_str_to_ull(const char * nptr, size_t n, char ** endptr);
```

`__IML_oct_str_to_i, __IML_oct_str_to_u, __IML_oct_str_to_ll, __IML_oct_str_to_ull`

Description: These functions convert the initial *n* octal digits (including an optional + or - sign) pointed to by *nptr* to integral values. When *endptr* is not a null pointer it points to the object that stores the final part of *nptr*. These functions treat any leading whitespace as invalid.

Calling interface:

```
int __IML_oct_str_to_i(const char * nptr,size_t n,char ** endptr);
unsigned int __IML_oct_str_to_u(const char * nptr,size_t n,char ** endptr);
long long __IML_oct_str_to_ll(const char * nptr,size_t n,char ** endptr);
unsigned long long __IML_oct_str_to_ull(const char * nptr,size_t n,char ** endptr);
```

`__IML_hex_str_to_i, __IML_hex_str_to_u, __IML_hex_str_to_ll, __IML_hex_str_to_ull`

Description: These functions convert the initial *n* hexadecimal digits (including an optional + or - sign) pointed to by *nptr* to integral values. When *endptr* is not a null pointer it points to the object that stores the final part of *nptr*. These functions treat any leading whitespace as invalid.

Calling interface:

```
int __IML_hex_str_to_i(const char * nptr,size_t n,char ** endptr);
unsigned int __IML_hex_str_to_u(const char * nptr,size_t n,char ** endptr);
long long __IML_hex_str_to_ll(const char * nptr,size_t n,char ** endptr);
unsigned long long __IML_hex_str_to_ull(const char * nptr,size_t n,char ** endptr);
```

Macros

The Intel® C++ Compiler Classic supports the ISO Standard predefined macros and additional predefined macros.

ISO Standard Predefined Macros

The ISO/ANSI standard for the C language requires that certain predefined macros be supplied with conforming compilers.

The compiler includes predefined macros in addition to those required by the standard. The default predefined macros differ among Windows*, Linux*, and macOS operating systems due to the default `/Za` compiler option on Windows*. Differences also exist on Linux* and macOS as a result of the `-std` compiler option.

The following table lists the macros that the Intel® C++ Compiler supplies in accordance with this standard:

Macro	Value
`__DATE__`	The date of compilation as an 11-character string literal in the form `mm dd yyyy`. If the day is less than 10 characters, a space is added before the day value.
`__FILE__`	A string literal representing the name of the file being compiled.
`__LINE__`	The current line number as a decimal constant.
`__STDC__`	Defined when compiling a C translation unit with the `Za` compiler option. (Windows*)
`__STDC_HOSTED__`	Defined and value is 1 only when compiling a C translation unit with `/Qstd=c99`.
`__STDC_VERSION__`	Defined and value is 199901L only when compiling a C translation unit with `/Qstd=c99`.
`__STDC_WANT_DEC_FP__`	Defined when compiling with option `D`. (Windows*, Linux*) Define this macro and include the respective header files to get the functions, macros, and types that comprise support for decimal floating-point functionality.
`__TIME__`	The time of compilation as a string literal in the form `hh:mm:ss`.

See Also

Additional Predefined Macros

Additional Predefined Macros

The compiler supports the predefined macros listed in the table below. The compiler also includes predefined macros specified by the ISO/ANSI standard.

Unless otherwise stated, the macros are supported on systems based on IA-32 and Intel® 64 architectures. IA-32 is not available on macOS*.

NOTE
The Intel® C++ Compiler defines the same target-architecture macros that GCC does. For `-m feature`, GCC defines `__FEATURE__`.

You can target specific processor architectures by using the `-x`, `-m`, and `-march` compiler options. Each of these options enables feature-specific macros in the compiler. These macros are used to guard a section of application code that uses target-specific feature. The following command emits the list of predefined macros enabled by targeting a specific processor architecture:

```
icpc -dM -E helloworld.cc -xarch
```

For example, you could do the following to determine which feature macros would help identify whether this is ICELAKE-SERVER:

```
icpc -dM -E helloworld.cc -xSKYLAKE-AVX512 > avx512.txt 2>&1
icpc -dM -E helloworld.cc -xICELAKE-SERVER > icelake.txt 2>&1
diff avx512.txt icelake.txt
 317a318
 > #define AVX512IFMA 1
 320a322,329
 > #define AVX512VBMI 1
 > #define AVX512VPOPCNTDQ 1
 > #define AVX512BITALG 1
 > #define AVX512VBMI2 1
 > #define GFNI 1
 > #define VAES 1
 > #define VPCLMUL 1
 > #define AVX512VNNI 1
 321a331,334
 > #define RDPID 1
 > #define SGX 1
 > #define WBNOINVD 1
 > #define PCONFIG 1
```

The result of the `diff` command is the list of feature macros that can be used to differentiate `icelake-server` from `skylake-avx512`.

Macro	Description
`__APPLE__` (macOS)	Defined as '1'.
`__APPLE_CC__` (macOS)	The gcc* build number
`__ARRAY_OPERATORS` (Linux*)	Defined as '1'.
`__AVX__` (Windows*, Linux, macOS)	On Windows*, defined as '1' when option `/arch:AVX`, `/QxAVX`, or higher processor targeting options are specified. On Linux*, defined as '1' when option `-march=corei7-avx`, `-mavx`, `-xAVX`, or higher processor targeting options are specified.

Macro	Description
	NOTE Available only for compilations targeting Intel® 64 architecture.
`__AVX2__` (Windows, Linux, macOS)	On Windows, defined as '1' when option `/arch:CORE-AVX2`, `/QxCORE-AVX2`, or higher processor targeting options are specified. On Linux, defined as '1' when option `-march=core-avx2`, `-xCORE-AVX2`, or higher processor targeting options are specified. **NOTE** Available only for compilations targeting Intel® 64 architecture.
`__AVX512BW__` (Windows*, Linux, macOS)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Byte and Word instructions. It is also defined as '1' when option `[Q]xCORE-AVX512` or higher processor-targeting options are specified.
`__AVX512CD__` (Windows*, Linux, macOS)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Conflict Detection instructions. It is also defined as '1' when option `[Q]xCORE-AVX512`, `[Q]xCOMMON-AVX512`, or higher processor-targeting options are specified.
`__AVX512DQ__` (Windows*, Linux, macOS)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Doubleword and Quadword instructions. It is also defined as '1' when option `[Q]xCORE-AVX512` or higher processor-targeting options are specified.
`__AVX512ER__` (Windows*, Linux, macOS)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Exponential and Reciprocal instructions.
`__AVX512F__` (Windows*, Linux, macOS)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions. It is also defined as '1' when option `[Q]xCORE-AVX512`, `[Q]xCOMMON-AVX512`, or higher processor-targeting options are specified.
`__AVX512PF__` (Windows*, Linux, macOS)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Prefetch instructions.
`__AVX512VL__` (Windows*, Linux, macOS)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Vector Length extensions.

Macro	Description
	It is also defined as '1' when option `[Q]xCORE-AVX512` or higher processor-targeting options are specified.
`__BASE_FILE__` (Linux)	Name of source file
`_BOOL` (Linux)	Defined as '1'.
`__COUNTER__` (Windows)	Defined as '0'.
`__cplusplus` (Linux)	Defined as '1' (for the Intel® C++ Compiler).
`__DEPRECATED` (Linux)	Defined as '1'.
`__DYNAMIC__` (macOS)	Defined as '1'.
`__EDG__` (Windows, Linux, macOS)	Defined as '1'.
`__EDG_VERSION__` (Windows, Linux, macOS)	EDG version
`__ELF__` (Linux)	Defined as '1' at the start of compilation.
`__EXCEPTIONS` (Linux)	Defined as '1' when option `fno-exceptions` is *not* used.
`__gnu_linux__` (Linux)	Defined as '1' at the start of compilation.
`__GNUC__` (Linux)	The major version number of gcc* installed on the system or explicitly specified via `-gcc-name/ -gxx-name`.
`__GNUC_MINOR__` (Linux)	The minor version number of gcc* or g++* installed on the system or explicitly specified via `-gcc-name/ -gxx-name`.
`__GNUC_PATCHLEVEL__` (Linux)	The patch level version number of gcc* or g++* installed on the system or explicitly specified via `-gcc-name/ -gxx-name`.
`__GNUG__` (Linux)	The major version number of g++* installed on the system or explicitly specified via `-gcc-name/ -gxx-name`.
`__GXX_ABI_VERSION` (Linux)	The value of this is dependent on the -fabi-version option in effect.

Macro	Description
	102: -fabi-version=1 || gcc version < 3.4 1008: gcc version >= 5.0 999999: -fabi-version=0 100? Where ? matches the -fabi-version passed: -fabi-version=2,3,4,5,6,7,8,9 1002: Gcc version > 3.5 and < 5.0
`__HONOR_STD` (Linux, macOS)	Defined as '1'.

`__i386__`
`__i386`
`i386`
(Linux, macOS)
	Defined as '1' for compilations targeting IA-32 architecture. IA-32 is not available on macOS*.

`__ICC` (Linux, macOS)	The version of the compiler. **NOTE** This macro may be affected by compiler options, such as `-no-icc`.
`__ICL` (Windows)	The version of the compiler. **NOTE** This macro may be affected by compiler options, such as `/Qicl-`.
`_INC_STDIO` (Windows)	Defined, no value.
`_INTEGRAL_MAX_BITS` (Windows)	64
`__INTEL_COMPILER` (Windows*, Linux, macOS)	The version of the compiler. **NOTE** This macro may be affected by compiler options, such as `-no-icc`.
`__INTEL_COMPILER_BUILD_DATE` (Windows*, Linux, macOS)	The compiler build date. It takes the form YYYYMMDD, where *YYYY* is the year, *MM* is the month, and *DD* is the day.
`__INTEL_COMPILER_UPDATE` (Windows, Linux, macOS)	Returns the current minor update number of the compiler, starting at 0. You can use this macro to differentiate between compiler updates when you have multiple updates of the Intel® C++ Compiler installed concurrently. **Example:** For Intel® C++ Compiler version *XX.0.2*, the macro would preprocess to "2".

Macro	Description
`__INTEL_CXX11_MODE__` (Windows, Linux)	Enables C++11 experimental support for C++ programs. Defined as '1' when option `[Q]std=c++11` is specified.
`__INTEL_MS_COMPAT_LEVEL` (Windows)	Defined as '1'. Equal to the same value *n* as specified by option `[Q]ms`*n*.
`__INTEL_RTTI__` (Linux, macOS)	Defined as '1' when option `-fno-rtti` is *not* specified.
`__INTEL_STRICT_ANSI__` (Linux, macOS)	Defined as '1' when option `-strict-ansi` is specified.

`__linux__`
`__linux`
`linux`
(Linux)
	Defined as '1' at the start of compilation.

`__LITTLE_ENDIAN__` (macOS)	Defined as '1'.
`__LONG_DOUBLE_SIZE__` (Windows*, Linux, macOS)	On Linux and macOS, defined as 80. On Windows, defined as 64; defined as 80 when option `/Qlong-double` is specified.
`__LONG_DOUBLE_64__` (Linux)	When this macro is defined, the long double type is 64-bits. It is defined when you specify option `-mlong-double-64`.
`__LONG_MAX__` (Linux)	9223372036854775807L **NOTE** Available only for compilations targeting Intel® 64 architecture.
`__LP64__` (Linux) `__LP64` (Linux)	Defined as '1'. **NOTE** Available only for compilations targeting Intel® 64 architecture.
`_M_AMD64` (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.
`_M_IX86` (Windows)	700

Macro	Description
_M_X64 (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.
__MACH__ (macOS)	Defined as '1'.
__MMX__ (Linux, macOS)	Defined as '1'. On Linux, it is available only on systems based on Intel® 64 architecture.
_MSC_EXTENSIONS (Windows)	Defined as '1'. This macro is defined when Microsoft extensions are enabled.
_MSC_FULL_VER (Windows)	The Visual C++* version being used. 190022609 for Visual C++* 2015 1800210051 for Visual C++* 2013
_MSC_VER (Windows)	The Visual C++* version being used. 1900 for Visual C++* 2015 1800 for Visual C++* 2013
_MT (Windows)	On Windows, defined as '1' when a multithreaded DLL or library is used (when option `/MD[d]` or `/MT[d]` is specified).
__NO_INLINE__ __NO_MATH_INLINES __NO_STRING_INLINES (Linux, macOS)	Defined as '1'.
_OPENMP (Windows, Linux, macOS)	201611 when you specify option `[Q]openmp`.
__OPTIMIZE__ (Linux, macOS)	Defined as '1'.
__pentium4 __pentium4__ (Linux, macOS)	Defined as '1'.
_PGO_INSTRUMENT (Windows, Linux)	Defined as '1' when option `[Q]cov-gen` or `[Q]prof-gen` is specified.
__PIC__ __pic__ (Linux, macOS)	On Linux, defined as '1' when option `fPIC` is specified. On macOS, defined as '1'. Only `__PIC__` is allowed on macOS.
_PLACEMENT_DELETE (Linux)	Defined as '1'.

Macro	Description
`__PTRDIFF_TYPE__` (Linux, macOS)	On Linux, defined as int on IA-32 architecture; defined as long on Intel® 64 architecture. On macOS, defined as int/long.
`__QMSPP_` (Windows, macOS)	Defined as '1'.
`__REGISTER_PREFIX__` (Linux, macOS)	
`__SIGNED_CHARS__` (Windows, Linux, macOS)	Defined as '1'.
`_SIZE_T_DEFINED` (Windows)	Defined, no value.
`__SIZE_TYPE__` (Linux, macOS)	On Linux, defined as unsigned on IA-32 architecture; defined as unsigned long on Intel® 64 architecture. On macOS, defined as unsigned long.
`__SSE__` (Windows, Linux, macOS)	On Linux and macOS, defined as '1' for processors that support SSE instructions. On Windows, defined as '1'. It is undefined when option `/arch:IA32` is specified.
`__SSE2__` (Windows, Linux, macOS)	On Linux and macOS, defined as '1' for processors that support Intel® SSE2 instructions. On Windows, defined as '1' by default or when option `/arch:SSE2`, `/QxSSE2`, `/QaxSSE2`, or higher processor targeting options are specified.
`__SSE3__` (Windows, Linux, macOS)	On Linux and macOS, defined as '1' for processors that support Intel® SSE3 instructions. On Windows, defined as '1' when option `/arch:SSE3`, `/QxSSE3`, or higher processor targeting options are specified.
`__SSE4_1__` (Windows, Linux)	On Linux, defined as '1' for processors that support Intel® SSE4 instructions. On Windows, defined as '1' when option `/arch:SSE4.1`, `/QxSSE4.1`, or higher processor targeting options are specified.
`__SSE4_2__` (Windows, Linux)	On Linux, defined as '1' for processors that support SSSE4 instructions. On Windows, defined as '1' when option `/arch:SSE4.2`, `/QxSSE4.2`, or higher processor targeting options are specified.
`__SSSE3__` (Windows, Linux, macOS)	On Linux and macOS, defined as '1' for processors that support SSSE3 instructions.

Macro	Description
	On Windows, defined as '1' when option `arch:SSSE3`, `QxSSSE3`, or higher processor targeting options are specified.
`__STDC__` (macOS)	Defined as '1'.
`__STDC_HOSTED__` (macOS)	Defined as '1'.

`unix`
`__unix`
`__unix__`
(Linux)
	Defined as '1'.

`__USER_LABEL_PREFIX__` (Linux, macOS)	
`_VA_LIST_DEFINED` (Windows)	Defined, no value.
`__VERSION__` (Linux, macOS)	The compiler version string
`__w64` (Windows)	Defined, no value.
`__WCHAR_MAX__` (macOS)	2147483647
`__WCHAR_T` (Linux)	Defined as '1'.
`_WCHAR_T_DEFINED` (Windows)	Defined when option `/Zc:wchar_t` is specified or "*wctype_t*" is defined in the header file.
`__WCHAR_TYPE__` (Linux, macOS)	On Linux, defined as long int on IA-32 architecture; defined as int on Intel® 64 architecture. On macOS, defined as long int.
`_WCTYPE_T_DEFINED` (Windows)	Defined when "*wctype_t*" is defined in the header file.
`_WIN32` (Windows)	Defined as '1' while building code targeting IA-32 or Intel® 64 architecture. IA-32 is not available on macOS*.
`_WIN64` (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.
`__WINT_TYPE__` (Linux, macOS)	Defined as unsigned int.

Macro	Description

`__x86_64`
`__x86_64__`
(Linux, macOS)
	Defined as '1' while building code targeting Intel® 64 architecture.

See Also

arch compiler option
march compiler option
m compiler option
intel-extensions, Qintel-extensions compiler option
D compiler option
U compiler option
qopenmp, Qopenmp compiler option
x, Qx compiler option
ISO Standard Predefined Macros

Use Predefined Macros to Specify Intel® Compilers

This topic shows how to use predefined macros to specify an Intel® compiler or version of an Intel compiler.

Predefined Macros to Specify Compiler and Version

When you install both the Intel® oneAPI Base Toolkit (Base Kit) and the Intel® oneAPI HPC Toolkit (HPC Kit), you will notice that there are three compilers installed:

- Intel® DPC++ Compiler
- Intel® C++ Compiler
- Intel® C++ Compiler Classic

NOTE This topic contains documentation for the Intel® C++ Compiler Classic. For information on predefined macros for the Intel® C++ Compiler and Intel® DPC++ Compiler, visit Predefined Macros for Intel® Compilers.

You can use the following predefined macros to invoke a specific compiler or version of a compiler:

Compiler	Predefined Macros to Differentiate from Other Compiler	Notes
Intel® C++ Compiler Classic	• `__INTEL_COMPILER` • `__INTEL_COMPILER_BUILD_DATE`	`__INTEL_COMPILER` is used to select the compiler. `__INTEL_COMPILER_BUILD_DATE` is used to select the compiler build.

Predefined Macros for Intel® C++ Compiler Classic

The following example uses `#if defined(__INTEL_COMPILER)` to define a code block specific to the Intel® C++ Compiler Classic:

```
// icc/icpc classic only
#if defined(__INTEL_COMPILER)
   // code specific for Intel C++ Compiler Classic below
```

```
   // ... ...

   // example only
   std::cout << "__INTEL_COMPILER_BUILD_DATE: " << __INTEL_COMPILER_BUILD_DATE << std::endl;
   std::cout << "__INTEL_COMPILER: " << __INTEL_COMPILER << std::endl;
   std::cout << "__VERSION__: " << __VERSION__ << std::endl;
#endif
```

Example output using the Intel® oneAPI Toolkit Gold release with an Intel C++ Compiler Classic patch release of 2021.1.2:

Linux

```
__INTEL_COMPILER_BUILD_DATE: 20201208

__INTEL_COMPILER: 2021

__VERSION__: Intel(R) C++ g++ 7.5 mode
```

Windows

```
__INTEL_COMPILER_BUILD_DATE: 20201208

__INTEL_COMPILER: 202110
```

Pragmas

Pragmas are directives that provide instructions to the compiler for use in specific cases. For example, you can use the `novector` pragma to specify that a loop should never be vectorized. The keyword `#pragma` is standard in the C++ language, but individual pragmas are machine-specific or operating system-specific, and vary by compiler.

Some pragmas provide the same functionality as compiler options. Pragmas override behavior specified by compiler options.

Some pragmas are available for both Intel® and non-Intel microprocessors but they may perform additional optimizations for Intel® microprocessors than they perform for non-Intel microprocessors. Refer to the individual pragma name for detailed description.

The Intel® C++ Compiler pragmas are categorized as follows:

- Intel-specific Pragmas - pragmas developed or modified by Intel to work specifically with the Intel® C++ Compiler
- Intel Supported Pragmas - pragmas developed by external sources that are supported by the Intel® C++ Compiler for compatibility reasons

Using Pragmas

You enter pragmas into your C++ source code using the following syntax:

```
#pragma <pragma name>
```

Individual Pragma Descriptions

Each pragma description has the following details:

Section	Description
Short Description	Contains a brief description of what the pragma does.

Section	Description
Syntax	Contains the pragma syntax.
Arguments	Contains a list of the arguments (parameters).
Description	Contains a detailed description of what the pragma does.
Example	Contains typical usage example/s.
See Also	Contains links or paths to other pragmas or related topics.

Intel-Specific Pragma Reference

Pragmas specific to the Intel® C++ Compiler Classic are listed in the following table.

Some pragmas are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

Pragma	Description
alloc_section	Allocates one or more variables in the specified section. Controls section attribute specification for variables.
block_loop/ noblock_loop	Enables or disables loop blocking for the immediately following nested loops. block_loop enables loop blocking for the nested loops. noblock_loop disables loop blocking for the nested loops.
code_align	Specifies the byte alignment for a loop
distribute_point	Instructs the compiler to prefer loop distribution at the location indicated.
inline/noinline/ forceinline	Specifies inlining of all calls in a statement. This also describes pragmas forceinline and noinline.
intel omp task	For Intel legacy tasking, specifies a unit of work, potentially executed by a different thread.
intel omp taskq	For Intel legacy tasking, specifies an environment for the while loop in which to queue the units of work specified by the enclosed `task` pragma.
ivdep	Instructs the compiler to ignore assumed vector dependencies.
loop_count	Specifies the iterations for a for loop.
nofusion	Prevents a loop from fusing with adjacent loops.
novector	Specifies that a particular loop should never be vectorized.
omp simd early_exit	Extends `#pragma omp simd`, allowing vectorization of multiple exit loops.
optimization_level	Controls optimization for one function or all functions after its first occurrence.
optimization_parameter	Passes certain information about a function to the optimizer.
optimize	Enables or disables optimizations for code after this pragma until another optimize pragma or end of the translation unit.
parallel/noparallel	Resolves dependencies to facilitate auto-parallelization of the immediately following loop (parallel) or prevents auto-parallelization of the immediately following loop (noparallel).

Pragma	Description
prefetch/noprefetch	Invites the compiler to issue or disable requests to prefetch data from memory. This pragma applies only to Intel® Advanced Vector Extensions 512 (Intel® AVX-512).
simd	Enforces vectorization of loops.
simdoff	Specifies a block of code in the SIMD loop or SIMD-enabled function that should be executed serially, in a logical order of SIMD lanes.
unroll/nounroll	Tells the compiler to unroll or not to unroll a counted loop.
unroll_and_jam/ nounroll_and_jam	Enables or disables loop unrolling and jamming. These pragmas can only be applied to iterative for loops.
unused	Describes variables that are unused (warnings not generated).
vector	Tells the compiler that the loop should be vectorized according to the argument keywords.

alloc_section

Allocates one or more variables in the specified section. Controls section attribute specification for variables.

Syntax

`#pragma alloc_section(`*`var1`*`,`*`var2`*`,...,` *`"r;attribute-list"`*`)`

Arguments

var	A variable that can be used to define a symbol in the section.
"r;attribute-list"	A comma-separated list of attributes; defined values are: '*short*' and '*long*'.

Description

The `alloc_section` pragma places the listed variables, *var1*, *var2*, etc., in the specified section. This pragma controls section attribute specification for variables. The compiler decides whether the variable, as defined by *var1*, *var2*, etc., should go to a "data", "bss", or "rdata" section.

The section name must be enclosed in double quotation marks. It should be previously introduced into the program using `#pragma section`. The list of comma-separated variable names follows the section name after a separating comma.

All listed variables must be defined before this pragma, in the same translation unit and in the same scope. The variables have static storage; their linkage does not matter in C modules, but in C++ modules they are defined with the extern "C" linkage specification.

Examples

This example shows how to use `#pragma alloc_section`:

```
#pragma alloc_section(var1, "r;short")
  int var1 = 20;
#pragma alloc_section(var2, "r;short")
  extern int var2;
```

block_loop/noblock_loop

Enables or disables loop blocking for the immediately following nested loops. block_loop enables loop blocking for the nested loops. noblock_loop disables loop blocking for the nested loops.

Syntax

`#pragma block_loop [`*`clause`*`[,`*`clause`*`]...]`

`#pragma noblock_loop`

Arguments

clause	Can be any of the following:	
	`factor` (*expr*)	*expr* is a positive scalar constant integer expression representing the blocking factor for the specified loops. This clause is optional. If the `factor` clause is not present, the blocking factor will be determined based on processor type and memory access patterns and will be applied to the specified levels in the nested loop following the pragma. At most only one *factor* clause can appear in a `block_loop` pragma.
	`level` (*level_expr*[, *level_expr*]...)	*level_expr* is specified in the form *const1* or *const1:const2* where *const1* is a positive integer constant *m*<= 8 representing the loop at level *m*, where the immediate following loop is level 1. The *const2* is a positive integer constant *n*<= 8 representing the loop at level *n*, where *n* > *m*. *const1:const2* represents the nested loops from level *const1* through *const2*.
	The clauses can be specified in any order. If you do not specify any clause, the compiler chooses the best blocking factor to apply to all levels of the immediately following nested loop.	

Description

The `block_loop` pragma lets you exert greater control over optimizations on a specific loop inside a nested loop.

Using a technique called loop blocking, the `block_loop` pragma separates large iteration counted loops into smaller iteration groups. Execution of these smaller groups can increase the efficiency of cache space use and augment performance.

If there is no `level` and `factor` clause, the blocking factor will be determined based on the processor's type and memory access patterns and it will apply to all the levels in the nested loops following this pragma.

You can use the `noblock_loop` pragma to tune the performance by disabling loop blocking for nested loops.

The loop-carried dependence is ignored during the processing of `block_loop` pragmas.

```
#pragma  block_loop factor(256) level(1)    /* applies blocking factor 256 to              */
#pragma  block_loop factor(512) level(2)    /* the top level loop in the following
                                               nested loop and blocking factor 512 to
                                               the 2nd level (1st nested) loop             */

#pragma  block_loop factor(256) level(2)
#pragma  block_loop factor(512) level(1)     /* levels can be specified in any order       */

#pragma  block_loop factor(256) level(1:2)   /* adjacent loops can be specified as a range  */

#pragma  block_loop factor(256)              /* the blocking factor applies to all levels
                                                of loop nest                                */

#pragma  block_loop               /* the blocking factor will be determined based on
                                     processor type and memory access patterns and will
                                     be applied to all the levels in the nested loop
                                     following the directive                                */

#pragma  noblock_loop             /* None of the levels in the nested loop following this
                                     directive will have a blocking factor applied          */
```

Consider the following:

```
#pragma block_loop factor(256) level(1:2)
for (j = 1 ; j<n ; j++){
  f  = 0 ;
  for (i =1 ;i<n  i++){
    f  =  f +   a[i]  *  b  [i] ;
  }
  c [j]  = c[j]  + f ;
}
```

The above code produces the following result after loop blocking:

```
for ( jj=1 ; jj<n/256+1 ; jj+){
  for ( ii = 1 ; ii<n/256+1 ;ii++){
    for ( j = (jj-1)*256+1 ;  min(jj*256, n) ;j++){
      f = 0 ;
      for ( i = (ii-1)*256+1 ;i<min(ii*256,n) ;i++){
        f = f + a[i] * b [i];
      }
      c[j]  = c[j] + f ;
    }
  }
}
```

code_align

Specifies the byte alignment for a loop

Syntax

`#pragma code_align`*(n)*

Arguments

n	Optional. A positive integer initialization expression indicating the number of bytes for the minimum alignment boundary. Its value must be a power of 2, between 1 and 4096, such as 1, 2, 4, 8, and so on. If you specify 1 for *n*, no alignment is performed. If you do not specify n, the default alignment is 16 bytes.

Description

This pragma must precede the loop to be aligned.

If the code is compiled with the `Qalign-loops:m` option, and a `code_align(n)` pragma precedes a loop, the loop is aligned on a max *(m, n)* byte boundary. If a procedure has the `code_align(k)` attribute and a `code_align(n)` pragma precedes a loop, then both the procedure and the loop are aligned on a max *(k, n)* byte boundary.

distribute_point

Instructs the compiler to prefer loop distribution at the location indicated.

Syntax

`#pragma distribute_point`

Arguments

None

Description

The `distribute_point` pragma is used to suggest to the compiler to split large loops into smaller ones; this is particularly useful in cases where optimizations like vectorization cannot take place due to excessive register usage.

The following rules apply to this pragma:

- When the pragma is placed inside a loop, the compiler distributes the loop at that point. All loop-carried dependencies are ignored.
- When inside the loop, pragmas cannot be placed within an `if` statement.
- When the pragma is placed outside the loop, the compiler distributes the loop based on an internal heuristic. The compiler determines where to distribute the loops and observes data dependency. If the pragmas are placed inside the loop, the compiler supports multiple instances of the pragma.

Examples

Use the `distribute_point` pragma outside the loop:

```
#define NUM 1024
void loop_distribution_pragma1(
       double a[NUM], double b[NUM], double c[NUM],
       double x[NUM], double y[NUM], double z[NUM] ) {
  int i;

  // Before distribution or splitting the loop
  #pragma distribute_point
  for (i=0; i< NUM; i++) {
    a[i] = a[i] + i;
    b[i] = b[i] + i;
```

```
    c[i] = c[i] + i;
    x[i] = x[i] + i;
    y[i] = y[i] + i;
    z[i] = z[i] + i;
  }
}
```

Use the `distribute_point` pragma inside the loop:

```
#define NUM 1024
void loop_distribution_pragma2(
       double a[NUM], double b[NUM], double c[NUM],
       double x[NUM], double y[NUM], double z[NUM] ) {
  int i;

  // After distribution or splitting the loop.
  for (i=0; i< NUM; i++) {
    a[i] = a[i] +i;
    b[i] = b[i] +i;
    c[i] = c[i] +i;
    #pragma distribute_point
    x[i] = x[i] +i;
    y[i] = y[i] +i;
    z[i] = z[i] +i;
  }
}
```

Use the `distribute_point` pragma inside and outside the loop:

```
void dist1(int a[], int b[], int c[], int d[]) {
  #pragma distribute_point
    // Compiler will automatically decide where to
    // distribute. Data dependency is observed.
  for (int i=1; i<1000; i++) {
    b[i] = a[i] + 1;
    c[i] = a[i] + b[i];
    d[i] = c[i] + 1;
  }
}

void dist2(int a[], int b[], int c[], int d[]) {
  for (int i=1; i<1000; i++) {
    b[i] = a[i] + 1;

    #pragma distribute_point
      // Distribution will start here,
      // ignoring all loop-carried dependency.
      c[i] = a[i] + b[i];
      d[i] = c[i] + 1;
  }
}
```

inline, noinline, forceinline

Specifies inlining of all calls in a statement. This also describes pragmas forceinline and noinline.

Syntax

```
#pragma inline [recursive]
```

```
#pragma forceinline [recursive]
```

```
#pragma noinline
```

Arguments

`recursive`	Indicates that the pragma applies to all of the calls that are called by these calls, recursively, down the call chain.

Description

`inline`, `forceinline`, and `noinline` are statement-specific inlining pragmas. Each can be placed before a C/C++ statement, and it will then apply to all of the calls within a statement and all calls within statements nested within that statement.

The `forceinline` pragma indicates that the calls in question should be inlined whenever the compiler is capable of doing so.

The `inline` pragma is a hint to the compiler that the user prefers that the calls in question be inlined, but expects the compiler not to inline them if its heuristics determine that the inlining would be overly aggressive and might slow down the compilation of the source code excessively, create too large of an executable, or degrade performance.

The `noinline` pragma indicates that the calls in question should not be inlined.

These statement-specific pragmas take precedence over the corresponding function-specific pragmas.

Examples

Use the `forceinline` recursive pragma:

```
#include <stdio.h>

static void fun(float a[100][100], float b[100][100]) {
  inti , j;
  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      a[i][j] = 2 * i;
      b[i][j] = 4 * j;
    }
  }
}

static void sun(float a[100][100], float b[100][100]) {
  int i, j;

  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      a[i][j] = 2 * i;
      b[i][j] = 4 * j;
    }
    fun(a, b);
  }
}
static float a[100][100];
static float b[100][100];

extern int main() {
  int i, j;
```

```
  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      a[i][j] = i + j;
      b[i][j] = i - j;
    }
  }
  for (i = 0; i < 99; i++) {
    fun(a, b);
#pragma forceinline recursive
    for (j = 0; j < 99; j++) {
      sun(a, b);
    }
  }
  fprintf(stderr, "%d %d\n", a[99][9], b[99][99]);
}
```

The `forceinline recursive` pragma applies to the call 'sun(a,b)' as well as the call 'fun(a,b)' called inside 'sun(a,b)'.

intel omp task

For Intel legacy tasking, specifies a unit of work, potentially executed by a different thread.

Syntax

`#pragma intel omp task [`*clause*`[[,]`*clause*`]...]`

`structured-block`

Arguments

clause	Can be any of the following:	
	`private` (*variable-list*)	Creates a private, default-constructed version for each object in *variable-list* for the `task`. The original object referenced by the variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.
	`captureprivate` (*variable-list*)	Creates a private, copy-constructed version for each object in *variable-list* for the `task` at the time the `task` is queued. The original object referenced by each variable retains its value but must not be modified within the dynamic extent of the `task` construct.

Description

The `intel omp task` pragma specifies a unit of work, potentially executed by a different thread.

NOTE
This pragma affects parallelization done using the `-qopenmp` option. Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® vs. non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

intel omp taskq

For Intel legacy tasking, specifies an environment for the while loop in which to queue the units of work specified by the enclosed `task` pragma.

Syntax

`#pragma intel omp taskq[`*`clause`*`[[,]`*`clause`*`]...]`

`structured-block`

Arguments

clause

Can be any of the following:

`private` (*variable-list*)	Creates a private, default-constructed version for each object in *variable-list* for the `taskq`. It also implies `captureprivate` on each enclosed task. The original object referenced by each variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.
`firstprivate` (*variable-list*)	Creates a private, copy-constructed version for each object in *variable-list* for the `taskq`. It also implies `captureprivate` on each enclosed task. The original object referenced by each variable must not be modified within the dynamic extent of the construct and has an indeterminate value upon exit from the construct.
`lastprivate` (*variable-list*)	Creates a private, default-constructed version for each object in *variable-list* for the `taskq`. It also implies `captureprivate` on each enclosed task. The original object referenced by each variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and is copy-

	assigned the value of the object from the last enclosed task after that task completes execution.
`reduction` (*operator* : *variable-list*)	Performs a reduction operation with the given operator in enclosed task constructs for each object in *variable-list*. *operator* and *variable-list* are defined the same as in the OpenMP* Specifications.
`ordered`	Organizes ordered constructs in enclosed `task` constructs in original sequential execution order. The `taskq` pragma, to which the `ordered` is bound, must have an `ordered` clause present.
`nowait`	Removes the implied barrier at the end of the `taskq`. Threads may exit the `taskq` construct before completing all the `task` constructs queued within it.

Description

The `intel omp taskq` pragma specifies the environment within which the enclosed units of work (tasks) are to be executed. From among all the threads that encounter a `intel omp taskq` pragma, one is chosen to execute it initially.

Conceptually, the `intel omp taskq` pragma causes an empty queue to be created by the chosen thread, and then the code inside the `taskq` block is executed as single-threaded. All the other threads wait for work to be queued on the conceptual queue.

The `intel omp taskq` pragma specifies a unit of work, potentially executed by a different thread. When a `task` pragma is encountered lexically within a `taskq` block, the code inside the `task` block is conceptually queued on the queue associated with the `taskq`. The conceptual queue is disbanded when all work queued on it finishes, and when the end of the `taskq` block is reached.

NOTE
This pragma affects parallelization done using the `Qopenmp` (Windows*) or `qopenmp` (Linux* or macOS) option. Options that use OpenMP* are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP* constructs and features that may perform differently on Intel® vs. non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

ivdep

Instructs the compiler to ignore assumed vector dependencies.

Syntax

```
#pragma ivdep
```

Arguments

None

Description

The `ivdep` pragma instructs the compiler to ignore assumed vector dependencies. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This pragma overrides that decision. Use this pragma only when you know that the assumed loop dependencies are safe to ignore.

In addition to the `ivdep` pragma, the `vector` pragma can be used to override the efficiency heuristics of the vectorizer.

> **NOTE**
> The proven dependencies that prevent vectorization are not ignored, only assumed dependencies are ignored.

Examples

The loop in this example will not vectorize without the `ivdep` pragma, since the value of *k* is not known; vectorization would be illegal if `k < 0`:

```
void ignore_vec_dep(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

The pragma binds only the `for` loop contained in current function. This includes a `for` loop contained in a sub-function called by the current function:

```
#pragma ivdep
  for (i=1; i<n; i++) {
    e[ix[2][i]] = e[ix[2][i]]+1.0;
    e[ix[3][i]] = e[ix[3][i]]+2.0;
}
```

This loop requires the parallel option in addition to the `ivdep` pragma to indicate there is no loop-carried dependencies:

```
#pragma ivdep
  for (j=0; j<n; j++) { a[b[j]] = a[b[j]] + 1; }
```

This loop requires the parallel option in addition to the `ivdep` pragma to ensure there is no loop-carried dependency for the store into `a()`.

See Also

Function Annotations and the SIMD Directive for Vectorization
novector pragma
vector pragma

loop_count

Specifies the iterations for a for loop.

Syntax

`#pragma loop_count(`*n*`)`

`#pragma loop_count=`*`n`*

or

`#pragma loop_count(`*`n1`*`[, `*`n2`*`]...)`

`#pragma loop_count=`*`n1`*`[, `*`n2`*`]...`

or

`#pragma loop_count `*`min`*`(`*`n`*`),`*`max`*`(`*`n`*`),`*`avg`*`(`*`n`*`)`

`#pragma loop_count `*`min`*`=`*`n`*`, `*`max`*`=`*`n`*`, `*`avg`*`=`*`n`*

Arguments

(n) or =n	A non-negative integer value. The compiler will attempt to iterate the next loop the number of times specified in *n*; however, the number of iterations is not guaranteed.
(n1[,n2]...) or = n1[,n2]...	Non-negative integer values. The compiler will attempt to iterate the next loop the number of time specified by *n1* or *n2*, or some other unspecified number of times. This behavior allows the compiler some flexibility in attempting to unroll the loop. The number of iterations is not guaranteed.
min(n), max(n), avg(n) or min=n, max=n, avg=n	Non-negative integer values. Specify one or more in any order without duplication. The compiler insures the next loop iterates for the specified maximum, minimum, or average number (*n1*) of times. The specified number of iterations is guaranteed for min and max.

Description

The `loop_count` pragma specifies the minimum, maximum, or average number of iterations for a `for` loop. In addition, a list of commonly occurring values can be specified to help the compiler generate multiple versions and perform complete unrolling.

You can specify more than one pragma for a single loop; however, do not duplicate the pragma.

Examples

Use the `loop_count` pragma to iterate through the loop a minimum of three, a maximum of ten, and average of five times:

```
#include <stdio.h>
int i;
int mysum(int start, int end, int a) {
  int iret=0;
  #pragma loop_count min(3), max(10), avg(5)
    for (i=start;i<=end;i++)
    iret += a;
    return iret;
}

int main() {
  int t;
  t = mysum(1, 10, 3);
  printf("t1=%d\r\n",t);
```

```
  t = mysum(2, 6, 2);
  printf("t2=%d\r\n",t);
  t = mysum(5, 12, 1);
  printf("t3=%d\r\n",t);
}
```

nofusion

Prevents a loop from fusing with adjacent loops.

Syntax

`#pragma nofusion`

Arguments

None

Description

The `nofusion` pragma lets you fine tune your program on a loop-by-loop basis. This pragma should be placed immediately before the loop that should not be fused.

Examples

```
#define SIZE 1024

int sub () {
int B[SIZE], A[SIZE];
  int i, j, k=0;
  for(j=0; j<SIZE; j++)
    A[j] = A[j] + B[j];

#pragma nofusion
  for (i=0; i<SIZE; i++)
    k += A[i] + 1;
  return k;
}
```

novector

Specifies that a particular loop should never be vectorized.

Syntax

`#pragma novector`

Arguments

None

Description

The `novector` pragma specifies that a particular loop should never be vectorized, even if it is legal to do so. When avoiding vectorization of a loop is desirable (when vectorization results in a performance regression rather than improvement), the `novector` pragma can be used in the source text to disable vectorization of a loop. This behavior is in contrast to the `vector always` pragma.

Examples

Use the novector pragma:

```
void foo(int lb, int ub) {
  #pragma novector
  for(j=lb; j<ub; j++) { a[j]=a[j]+b[j]; }
}
```

When the trip count (ub - lb) is too low to make vectorization worthwhile, you can use the novector pragma to tell the compiler not to vectorize, even if the loop is considered vectorizable.

See Also

Function Annotations and the SIMD Directive for Vectorization
vector pragma

omp simd early_exit

Extends #pragma omp simd, allowing vectorization of multiple exit loops.

Syntax

```
#pragma omp simd early_exit
```

Description

Extends #pragma omp simd allowing vectorization of multiple exit loops. When this clause is specified:

- Each operation before last lexical early exit of the loop may be executed as if early exit were not triggered within the SIMD chunk.
- After the last lexical early exit of the loop, all operations are executed as if the last iteration of the loop was found.
- Each list item specified in the linear clause is computed based on the last iteration number upon exiting the loop.
- The last value for linear clauses and conditional lastprivates clauses are preserved with respect to scalar execution.
- The last value for reductions clauses are computed as if the last iteration in the last SIMD chunk was executed up on exiting the loop.
- The shared memory state may not be preserved with regard to scalar execution.
- Exceptions are not allowed.

Examples

The following example demonstrates how to use this pragma.

In the following example, the pragma specifies that the vector execution of the for loop is safe even though the loop may exit before the loop upper bound condition j < ub becomes false. Suppose j1 is the smallest j, between lb and ub, such that j satisfies b[j] <= 0 . If j1 and j1+1, j1+2, ... are within the same (last) SIMD chunk, read of b[j1], b[j1+1], b[j1+2], ... and the subsequent evaluation of <= 0 will happen unconditionally, unlike the scalar execution of the same loop. Safety of such vector evaluation is programmer's responsibility. If necessary, simdlen() clause can be used to control the SIMD chunk size.

```
void foo(int lb, int ub) {
  float a = 0;
  #pragma omp simd early_exit reduction(+:a)
      for(j=lb; j<ub; j++) {
          if (b[j] <= 0 )
              break;
```

```
            a += b[j];
      }
}
```

optimize

Enables or disables optimizations for code after this pragma until another optimize pragma or end of the translation unit.

Syntax

`#pragma optimize("", on|off)`

Arguments

The compiler ignores first argument values. Valid second arguments for `optimize` are:

off	Disables optimization
on	Enables optimization

Description

The `optimize` pragma is used to enable or disable optimizations.

Specifying `#pragma optimize("", off)` disables optimization until either the compiler finds a matching `#pragma optimize("", on)` statement or until the compiler reaches the end of the translation unit.

Examples

In this example, optimizations are disabled for the `alpha()` function but not for the `omega()` function:

```
#pragma optimize("", off)
  alpha() { ... }

#pragma optimize("", on)
  omega() { ... }
```

In this example, optimizations are disabled for both the `alpha()` and `omega()` functions:

```
#pragma optimize("", off)
  alpha() { ... }
  omega() { ... }
```

optimization_level

Controls optimization for one function or all functions after its first occurrence.

Syntax

`#pragma [intel|GCC] optimization_level n`

Arguments

intel|GCC	Indicates the interpretation to use
n	An integer value specifying an optimization level; valid values are:

- 0: same optimizations as option `-O0` (Linux* and macOS) or `/Od` (Windows*)
- 1: same optimizations as option `O1`
- 2: same optimizations as option `O2`
- 3: same optimizations as option `O3`

Description

The `optimization_level` pragma is used to restrict optimization for a specific function while optimizing the remaining application using a different, higher optimization level. For example, if you specify option level `O3` for the application and specify `#pragma optimization_level 1`, the marked function will be optimized at option level `O1`, while the remaining application will be optimized at the higher level.

In general, this pragma optimizes the function at the level specified as *n*; however, certain compiler optimizations, like Inter-procedural Optimization (IPO), are not enabled or disabled during translation unit compilation. For example, if you enable IPO and a specific optimization level, IPO is enabled even for the function targeted by this pragma; however, IPO might not be fully implemented regardless of the optimization level specified at the command line. The reverse is also true.

Scope of optimization restriction

On Linux* and macOS systems, the scope of the optimization restriction can be affected by arguments passed to the `-pragma-optimization-level` compiler option as explained in the following table.

Syntax	Behavior
`#pragma intel optimization_level` *n*	Applies the pragma only to the next function, using the specified optimization level, regardless of the argument passed to the `-pragma-optimization-level` option.
`#pragma GCC optimization_level` *n* or `#pragma GCC optimization_level reset`	Applies the pragma to all subsequent functions, using the specified optimization level, regardless of the argument passed to the `-pragma-optimization-level` option. Specifying `reset` reverses the effect of the most recent `#pragma GCC optimization_level` statement, by returning to the optimization level previously specified.
`#pragma optimization_level` *n*	Applies either the Intel® C++ Compiler Classic implementation or the GCC* interpretation. Interpretation depends on the argument passed to the `-pragma-optimization-level` option.

NOTE
On Windows* systems, the pragma always uses the `intel` interpretation; the pragma is applied only to the next function.

Examples

Place the pragma immediately before the function being affected.

This example shows the `intel` interpretation of the `optimization_level` pragma:

```
#pragma intel optimization_level 1
  gamma() { ... }
```

This example shows GCC* interpretation of the optimization_level pragma:

```
#pragma GCC optimization_level 1
  gamma() { ... }
```

optimization_parameter

Passes certain information about a function to the optimizer.

Syntax

Linux and macOS:

`#pragma intel optimization_parameter target_arch=<CPU>`

`#pragma intel optimization_parameter inline-max-total-size=n`

`#pragma intel optimization_parameter inline-max-per-routine=n`

Windows:

`#pragma [intel] optimization_parameter target_arch=<CPU>`

`#pragma [intel] optimization_parameter inline-max-total-size=n`

`#pragma [intel] optimization_parameter inline-max-per-routine=n`

Arguments

target_arch=*<CPU>*

For the list of CPUs, see compiler options -m (or /arch) and [Q]x.

inline-max-per-routine=*n*

Specifies the maximum number of times the inliner may inline into the routine. *n* is one of the following:

- A non-negative integer constant that specifies the maximum number of times the inliner may inline into the function. If you specify zero, no inlining is done into the function.
- The keyword unlimited, which means that there is no limit to the number of times the inliner may inline into the function.

For more information, see option [Q]inline-max-per-routine.

inline-max-total-size=*n*

Specifies how much larger a function can normally grow when inline expansion is performed. *n* is one of the following:

- A non-negative integer constant that specifies the permitted increase in the function's size when inline expansion is performed. If you specify zero, no inlining is done into the function.
- The keyword unlimited, which means that there is no limit to the size a function may grow when inline expansion is performed.

For more information, see option [Q]inline-max-total-size.

Description

The intel optimization_parameter target_arch pragma controls the -m (or /arch) option settings at the function level, overriding the option values specified at the command-line.

Place #pragma intel optimization_parameter target_arch=*<CPU>* at the head of a function to get the compiler to target that function for a specified instruction set. The pragma works like the -m (or /arch) and [Q]x options, but applies only to the function before which it is placed.

The pragmas `intel optimization_parameter inline-max-total-size=n` and `intel optimization_parameter inline-max-per-routine=n` specify information used during inlining into a function.

Examples

This example shows targeting code for Intel® Advanced Vector Extensions (Intel® AVX) processors:

```
For C++:
icc -mAVX foo.c   // on Linux andmacOS
 -mAVX foo.c   // on Linux and macOS
```

This example code targets just the function `bar` for Intel® AVX processors, regardless of the command line options used:

```
#pragma intel optimization_parameter target_arch=AVX
void bar() { ... }
```

See Also

arch compiler option

m compiler option

Processor Targeting

ax, Qax
compiler option

inline-max-per-routine, Qinline-max-per-routine
compiler option

inline-max-total-size, Qinline-max-total-size
compiler option

parallel/noparallel

Resolves dependencies to facilitate auto-parallelization of the immediately following loop (parallel) or prevents auto-parallelization of the immediately following loop (noparallel).

Syntax

`#pragma parallel [`*clause*`[ [,]`*clause*`]...]`

`#pragma noparallel`

Arguments

clause	Can be any of the following:	
	`always [assert]`	Overrides compiler heuristics that estimate whether parallelizing a loop would increase performance. Using this clause on a loop that the compiler finds to be parallelizable tells the compiler to parallelize the loop even if doing so might not improve performance.

	If `assert` is added, the compiler will generate an error-level assertion test to display a message saying that the compiler efficiency heuristics indicate that the loop cannot be vectorized.
`firstprivate` (*var* [:*expr*] ...)text	Provides a superset of the functionality provided by the `private` clause. Variables that appear in a `firstprivate` list are subject to `private` clause semantics. In addition, its initial value is broadcast to all private instances upon entering the parallel loop.
`lastprivate` (*var* [:*expr*] ...)	Provides a superset of the functionality provided by the `private` clause. Variables that appear in a `lastprivate` list are subject to private clause semantics. In addition, when the parallel region is exited, each variable has the value that results from the sequentially last iteration of the loop up exiting the parallel loop.
`num_threads` (*n*)	Parallelizes the loop across *n* threads, where *n* is an integer.
`private` (*var* [:*expr*] ...)	Specifies a list of scalar and array variables (*var*) to privatize. An array or pointer variable can take an optional argument (*expr*) which is an int32 or int64 expression denoting the number of array elements to privatize.

Like the `private` clause, both the `firstprivate`, and the `lastprivate` clauses specify a list of scalar and array variables (*var*) to privatize. An array or pointer variable can take an optional argument (expr) which is an int32 or int64 expression denoting the number of array elements to privatize.

The same *var* is not allowed to appear in both the `private` and the `lastprivate` clauses for the same loop.

The same *var* is not allowed to appear in both the `private` and the `firstprivate` clauses for the same loop.

When *expr* is absent, the rules on *var* are the same as with OpenMP. The rules to be observed are as follows:

- *var* must not be part of another variable (as an array or structure element)
- *var* must not have a `const`-qualified type unless it is of class type with a mutable member
- *var* must not have an incomplete type or a reference type
- if *var* is of class type (or array thereof), then it requires an accessible, unambiguous default constructor for the class type. Furthermore, if this *var* is in a `lastprivate` clause, then it also requires an accessible, unambiguous copy assignment operator for the class type.

When *expr* is present, the same rules apply, but *var* must be an array or a pointer variable.

- If *`var`* is an array, then only its first *expr* elements are privatized. Without *expr*, the entire array is privatized.
- If *var* is a pointer, then the first *expr* elements are privatized (element size given by the pointer's target type). Without *expr*, only the pointer variable itself is privatized.
- Program behavior is undefined if *expr* evaluates to a non-positive value, or if it exceeds the array size.

Description

The `parallel` pragma instructs the compiler to ignore potential dependencies that it assumes could exist and which would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored.

The `noparallel` pragma prevents autoparallelization of the immediately following loop.

These pragmas take effect only if autoparallelization is enabled by the `[Q]parallel` compiler option. Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as the `arch`, `m`, or `[Q]x` compiler options.

Caution
Use this pragma with care. If a loop has cross-iteration dependencies, annotating it with this pragma can lead to incorrect program behavior.

Only use the `parallel` pragma if it is known that parallelizing the annotated loop will improve its performance.

Examples

This example shows how to use the `parallel` pragma:

```
void example(double *A, double *B, double *C, double *D) {
  int i;
  #pragma parallel
```

```
  for (i=0; i<10000; i++) {
    A[i] += B[i] + C[i];
    C[i] += A[i] + D[i];
  }
}
```

See Also

arch
m
parallel, Qparallel
x, Qx

prefetch/noprefetch

Invites the compiler to issue or disable requests to prefetch data from memory. This pragma applies only to Intel® Advanced Vector Extensions 512 (Intel® AVX-512).

Syntax

`#pragma prefetch`

`#pragma prefetch *:hint[:distance]`

`#pragma prefetch [var1 [: hint1 [: distance1]] [, var2 [: hint2 [: distance2]]]...]`

`#pragma noprefetch [var1 [, var2]...]`

Arguments

var	An optional memory reference (data to be prefetched)
hint	An optional hint to the compiler to specify the type of prefetch. Possible values: • `1`: For integer data that will be reused • 2: For integer and floating point data that will be reused from L2 cache • `3`: For data that will be reused from L3 cache • `4`: For data that will not be reused To use this argument, you must also specify *var*.
distance	An optional integer argument with a value greater than 0. It indicates the number of loop iterations ahead of which a prefetch is issued, before the corresponding load or store instruction. To use this argument, you must also specify *var* and *hint*.

Description

The `prefetch` pragma hints to the compiler to generate data prefetches for some memory references. These hints affect the heuristics used in the compiler. Prefetching data can minimize the effects of memory latency.

If you specify the `prefetch` pragma with no arguments, all arrays accessed in the immediately following loop are prefetched.

If the loop includes the expression `A(j)`, placing `#pragma prefetch A` in front of the loop instructs the compiler to insert prefetches for `A(j + d)` within the loop. Here, *d* is the number of iterations ahead of which to prefetch the data, and is determined by the compiler.

If you specify `#pragma prefetch *`, then *hint* and *distance* prefetches all array accesses in the loop.

To use these pragmas, compiler option `[Q]opt-prefetch` must be set (it is turned on by default if the compiler general optimization level is set at option `O2` or higher).

The `noprefetch` pragma hints to the compiler not to generate data prefetches for some memory references. This affects the heuristics used in the compiler.

Examples

Use the `prefetch` pragma:

```
#pragma prefetch htab_p:1:30
#pragma prefetch htab_p:0:6

// Issue vprefetch1 for htab_p with a distance of 30 vectorized iterations ahead
// Issue vprefetch0 for htab_p with a distance of 6 vectorized iterations ahead
// If pragmas are not present, compiler chooses both distance values

for (j=0; j<2*N; j++) { htab_p[i*m1 + j] = -1; }
```

Use `noprefetch` and `prefetch` pragmas together:

```
#pragma noprefetch b
#pragma prefetch a
for(i=0; i<m; i++) { a[i]=b[i]+1; }
```

Use `noprefetch` and `prefetch` pragmas together:

```
for (i=i0; i!=i1; i+=is) {

float sum = b[i];
int ip = srow[i];
int c = col[ip];

#pragma noprefetch col
#pragma prefetch value:1:80
#pragma prefetch x:1:40

for(; ip<srow[i+1]; c=col[++ip])
  sum -= value[ip] * x[c];
  y[i] = sum;
}
```

simd

Enforces vectorization of loops.

Syntax

`#pragma simd [`*clause*`[ [,] `*clause*`]...]`

Arguments

clause	Can be any of the following:

`vectorlength` (*n1*[, *n2*]...)	Where *n* is a vector length (VL). It must be an integer that is a power of 2; the value must be 2, 4, 8, or 16. If you specify more than one *n*, the vectorizor will choose the VL from the values specified. Causes each iteration in the vector loop to execute the computation equivalent to *n* iterations of scalar loop execution. Multiple `vectorlength` clauses are merged as a union.
`vectorlengthfor` (*data type*)	Where *data type* must be one of built-in integer types (8-, 16-, 32-, or 64-bit), pointer types (treated as pointer-sized integer), floating point types (32- or 64-bit), or complex types (64- or 128-bit). Otherwise, behavior is undefined. Causes each iteration in the vector loop to execute the computation equivalent to *n* iterations of scalar loop execution where *n* is computed from `size_of_vector_register`/`sizeof(`*`data type`*`)`. For example, `vectorlengthfor(float)` results in `n=4` for Intel® Streaming SIMD Extensions (Intel® SSE2) to Intel SSE4.2 targets (packed float operations available on 128bit XMM registers) and `n=8` for an Intel® Advanced Vector Extensions (Intel® AVX) target (packed float operations available on 256bit YMM registers). `vectorlengthfor(int)` results in `n=4` for Intel SSE2 to Intel AVX targets. `vectorlength()` and `vectorlengthfor()` clauses are mutually exclusive. In other words, the `vectorlengthfor()` clause may not be used with the `vectorlength()` clause, and vice versa. Behavior for multiple `vectorlengthfor` clauses is undefined.
`private` (*var1*[, *var2*]...)	Where *var* is a scalar variable. Causes each variable to be private to each iteration of a loop. Unless the variable appears in `firstprivate` clause, the initial value of the variable for the particular iteration is undefined. Unless the variable appears in `lastprivate` clause, the value of the variable upon exit of the loop is undefined. Multiple `private` clauses are merged as a union.

	NOTE Execution of the SIMD loop with `firtsprivate`/`lastprivate` clauses may be different from serial execution of the same code even if the loop fails to vectorize. A variable in a `private` clause cannot appear in a `linear`, `reduction`, `firstprivate`, or `lastprivate` clause.
`firstprivate` (*var1*[, *var2*]...)	Provides a superset of the functionality provided by the `private` clause. Variables that appear in a `firstprivate` list are subject to `private` clause semantics. In addition, its initial value is broadcast to all private instances for each iteration upon entering the SIMD loop. A variable in a `firstprivate` clause can appear in a `lastprivate` clause. A variable in a `firstprivate` clause cannot appear in a `linear`, `reduction`, or `private` clause.
`lastprivate` (*var1*[, *var2*]...)	Provides a superset of the functionality provided by the `private` clause. Variables that appear in a `lastprivate` list are subject to `private` clause semantics. In addition, when the SIMD loop is exited, each variable has the value that resulted from the sequentially last iteration of the SIMD loop (which may be undefined if the last iteration does not assign to the variable). A variable in a `lastprivate` clause can appear in a `firstprivate` clause. A variable in a `lastprivate` clause cannot appear in a `linear`, `reduction`, or `private` clause.
`linear` (*var1:step1* [,*var2:step2*]...)	Where *var* is a scalar variable and *step* is a compile-time positive, integer constant expression. For each iteration of a scalar loop, *var1* is incremented by *step1*, *var2* is incremented by *step2*, and so on. Therefore, every iteration of the vector loop increments the variables by VL*step1, VL*step2, …, to VL*stepN, respectively. If more than one

	step is specified for a *var*, a compile-time error occurs. Multiple linear clauses are merged as a union. A variable in a `linear` clause cannot appear in a `reduction`, `private`, `firstprivate`, or `lastprivate` clause.
`reduction` (*oper:var1* [,*var2*]...)	Where *oper* is a reduction operator and *var* is a scalar variable. Applies the vector reduction indicated by *oper* to *var1*, *var2*, ..., *varN*. The `simd` pragma may have multiple reduction clauses with the same or different operators. If more than one reduction operator is associated with a *var*, a compile-time error occurs. A variable in a `reduction` clause cannot appear in a `linear`, `private`, `firstprivate`, or `lastprivate` clause.
`[no]assert`	Directs the compiler to assert or not to assert when the vectorization fails. The default is `noassert`. If this clause is specified more than once, a compile-time error occurs.
`[no]vecremainder`	Instructs the compiler to vectorize or not to vectorize the remainder loop when the original loop is vectorized. See the description of the vector pragma for more information.

Description

The `simd` pragma is used to guide the compiler to vectorize more loops. Vectorization using the `simd` pragma complements (but does not replace) the fully automatic approach.

Without explicit `vectorlength()` and `vectorlengthfor()` clauses, the compiler will choose a `vectorlength` using its own cost model. Misclassification of variables into `private`, `firstprivate`, `lastprivate`, `linear`, and `reduction`, or lack of appropriate classification of variables may cause unintended consequences such as runtime failures and/or incorrect result.

You can only specify a particular variable in at most one instance of a `private`, `linear`, or `reduction` clause.

If the compiler is unable to vectorize a loop, a warning will be emitted (use the `assert` clause to make it an error).

If the vectorizer has to stop vectorizing a loop for some reason, the fast floating-point model is used for the SIMD loop.

The vectorization performed on this loop by the `simd` pragma overrides any setting you may specify for options `-fp-model` (Linux* and macOS) and `/fp` (Windows*) for this loop.

Note that the `simd` pragma may not affect all auto-vectorizable loops. Some of these loops do not have a way to describe the SIMD vector semantics.

The following restrictions apply to the `simd` pragma:

- The countable loop for the `simd` pragma has to conform to the for-loop style of an OpenMP worksharing loop construct. Additionally, the loop control variable must be a signed integer type.
- The vector values must be signed 8-, 16-, 32-, or 64-bit integers, single or double-precision floating point numbers, or single or double-precision complex numbers.
- A SIMD loop may contain another loop (`for`, `while`, `do-while`) in it. Goto out of such inner loops are not supported. Break and continue are supported.

NOTE Inlining can create such an inner loop, which may not be obvious at the source level.

- A SIMD loop performs memory references unconditionally. Therefore, all address computations must result in valid memory addresses, even though such locations may not be accessed if the loop is executed sequentially.

To disable transformations that enables more vectorization, specify the `-vec -no-simd` (Linux* and macOS) or `/Qvec /Qno-simd` (Windows*) options.

User-mandated vectorization, also called SIMD vectorization can assert or not assert an error if a `#pragma simd` annotated loop fails to vectorize. By default, the `simd` pragma is set to `noassert`, and the compiler will issue a warning if the loop fails to vectorize. To direct the compiler to assert an error when the `#pragma simd` annotated loop fails to vectorize, add the `assert` clause to the `simd` pragma. If a `simd` pragma annotated loop is not vectorized by the compiler, the loop holds its serial semantics.

Examples

This example shows how to use the `simd` pragma:

```
 void add_floats(float *a, float *b, float *c, float *d, float *e, int n){
  int i;
#pragma simd
  for (i=0; i<n; i++){
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
  }
}
```

In the example, the function `add_floats()` uses too many unknown pointers for the compiler's automatic runtime independence check optimization to kick-in. The programmer can enforce the vectorization of this loop by using the `simd` pragma to avoid the overhead of runtime check.

See Also

Function Annotations and the SIMD Directive for Vectorization
fp-model, fp compiler option
qsimd-honor-fp-model, Qsimd-honor-fp-model compiler option
qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction compiler option
vec, Qvec compiler option
vector pragma

simdoff

Specifies a block of code in the SIMD loop or SIMD-enabled function that should be executed serially, in a logical order of SIMD lanes.

Syntax

`#pragma simdoff`

structured-block

Arguments

None.

Description

The `simdoff` block will use a single SIMD lane to execute operations in the order of the loop iterations, or logical lanes of a SIMD-enabled function. This preserves ordering of operations in the block with respect to each other, and correlates with iteration space of the enclosing SIMD construct. The ordered `simd` block is executed in order, with respect to each SIMD lane or each loop iteration. The operations within the ordered `simd` or `simdoff` block can be re-ordered by optimizations, as long as the original execution semantics are preserved.

`simdoff` blocks allow the isolation and resolution of situations prohibited from SIMD execution. This includes cross-iteration data dependencies, function calls with side effects, such as OpenMP, oneTBB and native thread synchronization primitives.

Examples

`simdoff` sections are useful for resolving cross-iteration data dependencies in otherwise data-parallel computations. For example, the section may handle histogram updates as shown in this example code:

```
#pragma simd
for (int i = 0; i < N; i++)
{
    float amount = compute_amount(i);
    int   cluster = compute_cluster(i);
#pragma simdoff
    {
       totals[cluster] += amount; // Requires ordering to process multiple updates for the same
cluster
    }
}
```

unroll/nounroll

Tells the compiler to unroll or not to unroll a counted loop.

Syntax

`#pragma unroll`

`#pragma unroll(`*n*`)`

`#pragma nounroll`

Arguments

n	The unrolling factor representing the number of times to unroll a loop; it must be an integer constant from 0 through 255.

Description

The `unroll`*[n]* pragma tells the compiler how many times to unroll a counted loop.

The `unroll` pragma must precede the `for` statement for each `for` loop it affects. If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop.

This pragma is supported only when option `O3` is set. The `unroll` pragma overrides any setting of loop unrolling from the command line.

The pragma can be applied for the innermost loop nest as well as for the outer loop nest. If applied to outer loop nests, the current implementation supports complete outer loop unrolling. The loops inside the loop nest are either not unrolled at all or completely unrolled. The compiler generates correct code by comparing *n* and the loop count.

When unrolling a loop increases register pressure and code size it may be necessary to prevent unrolling of a loop. In such cases, use the `nounroll` pragma. The `nounroll` pragma instructs the compiler not to unroll a specified loop.

Examples

Use the `unroll` pragma for innermost loop unrolling:

```
void unroll(int a[], int b[], int c[], int d[]) {
  #pragma unroll(4)
  for (int i = 1; i < 100; i++) {
    b[i] = a[i] + 1;
    d[i] = c[i] + 1;
  }
}
```

Use the `unroll` pragma for outer loop unrolling:

```
int m = 0;
int dir[4]= {1,2,3,4};
int data[10];
#pragma unroll (4)  // outer loop unrolling
  for (int i = 0; i < 4; i++) {
     for (int j = dir[i]; data[j]==N ; j+=dir[i])
        m++;
  }
```

When you place the `unroll` pragma before the first `for` loop, it causes the compiler to unroll the outer loop completely. If an `unroll` pragma is placed before the inner `for` loop as well as before the outer `for` loop, the compiler ignores the inner `for` loop `unroll` pragma. If the `unroll` pragma is placed only for the innermost loop, the compiler unrolls the innermost loop according to some factor.

unroll_and_jam/nounroll_and_jam

Enables or disables loop unrolling and jamming. These pragmas can only be applied to iterative for loops.

Syntax

`#pragma unroll_and_jam`

`#pragma unroll_and_jam (`*n*`)`

`#pragma nounroll_and_jam`

Arguments

n	The unrolling factor representing the number of times to unroll a loop; it must be an integer constant from 0 through 255

Description

The `unroll_and_jam` pragma partially unrolls one or more loops higher in the nest than the innermost loop and fuses/jams the resulting loops back together. This transformation allows more reuses in the loop.

This pragma is not effective on innermost loops. Ensure that the immediately following loop is not the innermost loop after compiler-initiated interchanges are completed.

Specifying this pragma is a hint to the compiler that the unroll and jam sequence is legal and profitable. The compiler enables this transformation whenever possible.

The `unroll_and_jam` pragma must precede the `for` statement for each `for` loop it affects. If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop. The compiler generates correct code by comparing *n* and the loop count.

This pragma is supported only when compiler option `O3` is set. The `unroll_and_jam` pragma overrides any setting of loop unrolling from the command line.

When unrolling a loop increases register pressure and code size it may be necessary to prevent unrolling of a nested loop or an imperfect nested loop. In such cases, use the `nounroll_and_jam` pragma. The `nounroll_and_jam` pragma hints to the compiler not to unroll a specified loop.

Examples

Use the `unroll_and_jam` pragma:

```
int a[10][10];
int b[10][10];
int c[10][10];
int d[10][10];
void unroll(int n) {
    int i,j,k;
    #pragma unroll_and_jam (6)
    for (i = 1; i < n; i++) {
       #pragma unroll_and_jam (6)
       for (j = 1; j < n; j++) {
          for (k = 1; k < n; k++){
            a[i][j] += b[i][k]*c[k][j];
            }
       }
    }
}
```

unused

Describes variables that are unused (warnings not generated).

Syntax

`#pragma unused`

Arguments

None

Description

The `unused` pragma is implemented for compatibility with Apple* implementation of GCC.

vector

Tells the compiler that the loop should be vectorized according to the argument keywords.

Syntax

```
#pragma vector {always[assert]|aligned|unaligned|dynamic_align|nodynamic_align|[no]
multiple_gather_scatter_by_shuffles|temporal|nontemporal|[no]vecremainder|
[no]mask_readwrite|vectorlength(n1[, n2]...)}
```

```
#pragma vector nontemporal[(var1[, var2, ...])]
```

Arguments

`always`	Instructs the compiler to override any efficiency heuristic during the decision to vectorize or not, and vectorize non-unit strides or very unaligned memory accesses; controls the vectorization of the subsequent loop in the program; optionally takes the keyword assert.
`aligned`	Instructs the compiler to use aligned data movement instructions for all array references when vectorizing.
`unaligned`	Instructs the compiler to use unaligned data movement instructions for all array references when vectorizing.
`dynamic_align`[(*var*)]	Instructs the compiler to perform dynamic alignment optimization for the loop with an optionally specified variable to perform alignment on.
`nodynamic_align`	Disables dynamic alignment optimization for the loop.
`multiple_gather_scatter_by_shuffles`	Instructs the optimizer to disable the generation of gather/scatter and to transform gather/scatter into unit-strided loads/stores plus a set of shuffles wherever possible.
`nomultiple_gather_scatter_by_shuffles`	Instructs the optimizer to enable the generation of gather/scatter instructions and not to transform gather/scatter into unit-strided loads/stores.
`nontemporal`	Instructs the compiler to use non-temporal (that is, streaming) stores on systems based on all supported architectures, unless otherwise specified; optionally takes a comma-separated list of variables. When this pragma is specified, it is your responsibility to also insert any fences as required to ensure correct memory ordering within a thread or across threads. One typical way to do this is to insert a `_mm_sfence` intrinsic call just after the loops (such as the initialization loop) where the compiler may insert streaming store instructions.

`temporal`	Instructs the compiler to use temporal (that is, non-streaming) stores on systems based on all supported architectures, unless otherwise specified.
`vecremainder`	Instructs the compiler to vectorize the remainder loop when the original loop is vectorized.
`novecremainder`	Instructs the compiler not to vectorize the remainder loop when the original loop is vectorized.
`mask_readwrite`	Disables memory speculation, causing the generation of masked load and store operations within conditions.
`nomask_readwrite`	Enables memory speculation, causing the generation of non-masked loads and stores within conditions.
`vectorlength` (*n1*[, *n2*]...)	Instructs the vectorizer which vector length/factor to use when generating the main vector loop.

Description

The `vector` pragma indicates that the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability. The `vector` pragma takes several argument keywords to specify the kind of loop vectorization required. The compiler does not apply the `vector` pragma to nested loops, each nested loop needs a preceding pragma statement. Place the pragma before the loop control statement.

Using the `aligned/unaligned` keywords

When the `aligned/unaligned` argument keyword is used with this pragma, it indicates that the loop should be vectorized using aligned/unaligned data movement instructions for all array references. Specify only one argument keyword: `aligned` or `unaligned`.

> **Caution**
> If you specify `aligned` as an argument, you must be sure that the loop is vectorizable using this pragma. Otherwise, the compiler generates incorrect code.

Using the `always` keyword

When the `always` argument keyword is used, the pragma will ignore compiler efficiency heuristics for the subsequent loop. When `assert` is added, the compiler will generate a diagnostic message if the loop cannot be vectorized for any reason.

Using the `dynamic_align` and `nodynamic_align` keywords

Dynamic alignment is an optimization the compiler attempts to perform by default. It involves peeling iterations from the vector loop into a scalar loop before the vector loop so that the vector loop aligns with a particular memory reference. The `dynamic_align` (*var*) form of the directive allows the user to provide a scalar or array variable name to align on. Specifying `nodynamic_align` with or without *var* does not guarantee the optimization is performed; the compiler still uses heuristics to determine feasibility of the operation.

Using the `multiple_gather_scatter_by_shuffles` and `nomultiple_gather_scatter_by_shuffles` keywords

These clauses do not affect loops nested in the specified loop.

Using the `nontemporal` and `temporal` keywords

The nontemporal and temporal argument keywords are used to control how the "stores" of register contents to storage are performed (streaming versus non-streaming) on systems based on IA-32 and Intel® 64 architectures.

By default, the compiler automatically determines whether a streaming store should be used for each variable.

Streaming stores may cause significant performance improvements over non-streaming stores for large numbers on certain processors. However, the misuse of streaming stores can significantly degrade performance.

Using the `[no]vecremainder` keyword

If the vector always pragma and keyword are specified, the following occurs:

- If the vecremainder clause is specified, the compiler vectorizes both the main and remainder loops.
- If the novecremainder clause is specified, the compiler vectorizes the main loop, but it does not vectorize the remainder loop.

Using the `[no]mask_readwrite` keyword

If the vector pragma and mask_readwrite or nomask_readwrite keyword are specified, the following occurs:

- If the mask_readwrite clause is specified, the compiler generates masked loads and stores within all conditions in the loop.
- If the nomask_readwrite clause is specified, the compiler generates unmasked loads and stores for increased performance.

Using the `vectorlength` keyword

n is an integer power of 2; the value must be 2, 4, 6, 8, 16, 32, or 64. If more than one value is specified, the vectorizer will choose one of the specified vector lengths based on a cost model decision.

NOTE
The pragma vector should be used with care.

Overriding the efficiency heuristics of the compiler should only be done if the programmer is absolutely sure that vectorization will improve performance. Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a run-time exception in case some of the access patterns are actually unaligned.

Examples

Use the `vector aligned` pragma

In the following example, the aligned argument keyword is used to request that the loop be vectorized with aligned instructions.

Note that the arrays are declared in such a way that the compiler could not normally prove this would be safe to vectorize.

```
void vec_aligned(float *a, int m, int c) {
  int i;
  // Instruct compiler to ignore assumed vector dependencies.
  #pragma vector aligned
  for (i = 0; i < m; i++)
    a[i] = a[i] * c;
  // Alignment unknown but compiler can still align.
```

```
  for (i = 0; i < 100; i++)
    a[i] = a[i] + 1.0f;
}
```

Use the `vector always` pragma

```
void vec_always(int *a, int *b, int m) {
  #pragma vector always
  for(int i = 0; i <= m; i++)
    a[32*i] = b[99*i];
}
```

Use the `vector multiple gather type` pragma

```
float sum=0.0f;
#pragma omp simd reduction(+:sum)
for (i=0; i<N; i++){
   sum += A[3*i+0] + A[3*i+1] + A[3*i+2];
}
```

Use the `vector nontemporal` pragma

```
float a[1000];
void foo(int N){
  int i;
  #pragma vector nontemporal
  for (i = 0; i < N; i++) {
    a[i] = 1;
  }
}
```

Use ASM code for the loop body

A float-type loop together with the generated assembly is shown in the following example. For large *N*, significant performance improvements result on systems with Intel® Pentium® 4 processors over non-streaming implementations.

```
  .B1.2:
movntps XMMWORD PTR _a[eax], xmm0
movntps XMMWORD PTR _a[eax+16], xmm0
add eax, 32
cmp eax, 4096
jl .B1.2
```

Use pragma `vector nontemporal` with variables for implementing streaming stores

```
double A[1000];
double B[1000];
void foo(int n){
  int i;
#pragma vector nontemporal (A, B)
  for (i=0; i<n; i++){
    A[i] = 0;
    B[i] = i;
  }
}
```

See Also

Function Annotations and the SIMD Directive for Vectorization

Intel-supported Pragma Reference

The Intel® C++ Compiler Classic supports the following pragmas to ensure compatibility with other compilers.

Pragmas Compatible with the Microsoft* Compiler

The following pragmas are compatible with the Microsoft Compiler. For more information about these pragmas, go to the Microsoft Developer Network (http://msdn.microsoft.com).

Pragma	Description
`alloc_text`	Names the code section where the specified function definitions are to reside.
`auto_inline`	Excludes any function defined within the range where **`off`** is specified from being considered as candidates for automatic inline expansion.
`bss_seg`	Indicates to the compiler the segment where uninitialized variables are stored in the `.obj` file.
`check_stack`	The **`on`** argument indicates that stack checking should be enabled for functions that follow and the **`off`** argument indicates that stack checking should be disabled for functions that follow.
`code_seg`	Specifies a code section where functions are to be allocated.
`comment`	Places a comment record into an object file or executable file.
`component`	Controls collecting of browse information or dependency information from within source files.
`conform`	Specifies the run-time behavior of the `/Zc:`*`forScope`* compiler option.
`const_seg`	Specifies the segment where functions are stored in the `.obj` file.
`data_seg`	Specifies the default section for initialized data.
`deprecated`	Indicates that a function, type, or any other identifier may not be supported in a future release or indicates that a function, type, or any other identifier should not be used any more.
`fenv_access`	Informs an implementation that a program may test status flags or run under a non-default control mode.
`float_control`	Specifies floating-point behavior for a function.
`fp_contract`	Allows or disallows the implementation to contract expressions.
`loop`	Controls how the loop code will be considered or excluded from consideration by the auto-vectorizer.
`init_seg`	Specifies the section to contain C++ initialization code for the translation unit.
`message`	Displays the specified string literal to the standard output device (`stdout`).

Pragma	Description
`optimize`	Specifies optimizations to be performed on functions below the pragma or until the next optimize pragma; implemented to partly support the Microsoft implementation of same pragma; for the Intel® C++ Compiler Classic implementation, see the `optimize` reference page.
`pointers_to_members`	Specifies whether a pointer to a class member can be declared before its associated class definition and is used to control the pointer size and the code required to interpret the pointer.
`pop_macro`	Sets the value of the specified macro to the value on the top of the stack.
`push_macro`	Saves the value of the specified macro on the top of the stack.
`region/endregion`	Specifies a code segment in the Microsoft Visual Studio* Code Editor that expands and contracts by using the outlining feature.
`section`	Creates a section in an `.obj` file. Once a section is defined, it remains valid for the remainder of the compilation.
`vtordisp`	The **`on`** argument enables the generation of hidden `vtordisp` members and the **`off`** disables them. **`push`** argument pushes the current `vtordisp` setting to the internal compiler stack. **`pop`** argument removes the top record from the compiler stack and restores the removed value of `vtordisp`.
`warning`	Allows selective modification of the behavior of compiler warning messages.
`weak`	Declares symbol you enter to be weak.

OpenMP* Standard Pragmas

The Intel® C++ Compiler Classic currently supports OpenMP* 5.0 Version TR4, and some OpenMP Version 5.1 pragmas. Supported pragmas are isted below. For more information about these pragmas, reference the OpenMP* Version 5.1 specification.

Intel-specific clauses are noted in the affected pragma description.

Pragma	Description
omp atomic	Specifies a computation that must be executed atomically.
omp barrier	Specifies a point in the code where each thread must wait until all threads in the team arrive.
omp cancel	Requests cancellation of the innermost enclosing region of the type specified, and causes the encountering task to proceed to the end of the cancelled construct.
omp cancellation point	Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified. This construct does not implement a synchronization between threads or tasks.
omp critical	Specifies a code block that is restricted to access by only one thread at a time.

Pragma	Description
omp declare reduction	Declares User-Defined Reduction (UDR) functions (reduction identifiers) that can be used as reduction operators in a reduction clause.
omp declare simd	Creates a version of a function that can process multiple arguments using Single Instruction Multiple Data (SIMD) instructions from a single invocation from a SIMD loop.
omp distribute	Specifies that the iterations of one or more loops should be distributed among the initial threads of all thread teams in a league.
omp distribute parallel for	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
omp distribute parallel for simd	Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.
omp distribute simd	Specifies a loop that will be distributed across the primary threads of the teams region. It will be executed concurrently using SIMD instructions.
omp flush	Identifies a point at which a thread's temporary view of memory becomes consistent with the memory.
omp for	Specifies a work-sharing loop. Iterations of the loop are executed in parallel by the threads in the team.
omp for simd	Specifies that the iterations of the loop will be distributed across threads in the team. Iterations executed by each thread can also be executed concurrently using SIMD instructions.
omp master	Specifies a code block that must be executed only once by the primary thread of the team.
omp ordered	Specifies a block of code that the threads in a team must execute in the natural order of the loop iterations, or as a stand-alone directive, specifies cross-iteration dependences in a doacross loop-nest.
omp parallel	Specifies that a structured block should be run in parallel by a team of threads.
omp parallel for	Provides an abbreviated way to specify a parallel region containing only a FOR construct.
omp parallel for simd	Specifies a parallel construct that contains one for simd construct and no other statement.
omp parallel sections	Specifies a parallel construct that contains only a sections construct.
omp scan	Specifies a scan computation that updates each list item in each iteration of the loop.
omp sections	Defines a set of structured blocks that will be distributed among the threads in the team.
omp simd	Transforms the loop into a loop that will be executed concurrently using SIMD instructions.
omp single	Specifies that a block of code is to be executed by only one thread in the team.

Pragma	Description
omp target	Creates a device data environment and executes the construct on that device.
omp target data	Specifies that variables are mapped to a device data environment for the extent of the region.
omp target enter data	Specifies that variables are mapped to a device data environment.
omp target exit data	Specifies that variables are unmapped from a device data environment.
omp target teams	Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the primary thread in each team executing the structured block.
omp target teams distribute	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct.
omp target teams distribute parallel for	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct.
omp target teams distribute parallel for simd	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.
omp target teams distribute simd	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct. It will be executed concurrently using SIMD instructions.
omp target update	Makes the items listed in the device data environment consistent between the device and host, in accordance with the motion clauses on the pragma.
omp task	Specifies a code block whose execution may be deferred.
omp taskgroup	Causes the program to wait until the completion of all enclosed and descendant tasks.
omp taskwait	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
omp taskyield	Specifies that the current task can be suspended at this point in favor of execution of a different task.
omp teams	Creates a league of thread teams inside a target region to execute a structured block in the initial thread of each team.
omp teams distribute	Creates a league of thread teams and specifies that loop iterations will be distributed among the primary threads of all thread teams in the league.
omp teams distribute parallel for	Creates a league of thread teams and specifies that the associated loop can be executed in parallel by multiple threads that are members of multiple teams.

Pragma	Description
omp teams distribute parallel for simd	Creates a league of thread teams and specifies that the associated loop can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams.
omp teams distribute simd	Creates a league of thread teams and specifies that the associated loop will be distributed across the primary threads of the teams and executed concurrently using SIMD instructions.
omp threadprivate	Specifies a list of globally-visible variables that will be allocated private to each thread.

Pragmas Compatible with Other Compilers

The following pragmas are compatible with other compilers. For more information about these pragmas, see the documentation for the specified compiler.

Pragma	Description
`include_directory`	HP-compatible pragma. It appends the string argument to the list of places to search for `#include` files.
`poison`	GCC-compatible pragma. It labels the identifiers you want removed from your program; an error results when compiling a "`poisoned`" `identifier; #pragma POISON` is also supported.
`options`	GCC-compatible pragma; It sets the alignment of fields in structures.
`weak`	GCC-compatible pragma, it declares the symbol you enter to be weak.

See Also

Intel-specific Pragmas
optimize compiler option
Zc compiler option

Error Handling

This topic describes compiler remarks, warnings, and errors. The compiler sends these messages, along with the erroneous source line, to stderr.

Warnings

Warning messages report legal but questionable use of C or C++. The compiler displays warnings by default. You can suppress warning messages by specifying an appropriate compiler option. Warnings do not stop translation or linking. Warnings do not interfere with any output files.

The following are some representative warning messages:

```
declaration does not declare anything
```

```
pointless comparison of unsigned integer with zero
```

```
possible use of = where == was intended
```

Some warnings that start with `-W` can be disabled using the negative form of the option `-Wno-`. For example, option `-Wno-unknown-pragmas` disables option `-Wunknown-pragmas`.

Linux and macOS

Additional Warnings

The following Linux* and macOS options produce additional warnings:

Option	Result
`-W[no-]missing-prototypes`	Warn for missing prototypes.
`-W[no-]missing-declarations`	Warn for missing declarations.
`-W[no-]unused-variable`	Warn for unused variable.
`-W[no-]pointer-arith`	Warn for questionable pointer arithmetic.
`-W[no-]uninitialized`	Warn if a variable is used before being initialized.
`-W[no-]deprecated`	Display warnings related to deprecated features.
`-W[no-]abi`	Warn if generated code is not C++ ABI compliant.
`-W[no-]unused-function`	Warn if declared function is not used.
`-W[no-]unknown-pragmas`	Warn if an unknown `#pragma` directive is used.
`-W[no-]main`	Warn if return type of main is not expected.
`-W[no-]comment[s]`	Warn when /* appears in the middle of a /* */ comment.
`-W[no-]return-type`	Warn when a function uses the default `int` return type Warn when a return statement is used in a void function.

Errors

These messages report syntactic or semantic misuse of C or C++. The compiler always displays error messages. Errors suppress object code for the module containing the error and prevent linking, but they allow parsing to continue to detect other possible errors.

The following are some representative error messages:

```
missing closing quote
```

```
expression must have arithmetic type
```

```
expected a ";"
```

Remarks

Remark messages report common but sometimes unconventional use of C or C++. Remarks do not stop translation or linking. Remarks do not interfere with any output files.

The following are some representative remark messages:

```
function declared implicitly
```

```
type qualifiers are meaningless in this declaration
```

```
controlling expression is constant
```

Some remarks, warnings, and errors are numbered and can be disabled using option `-diag-disable=list` or `/Qdiag-disable:list`.

Example

Linux and macOS

```
-diag-disable=117,230,450
```

Windows

```
/Qdiag-disable:117,230,450
```

Option Summary

You can use the following compiler options to control remarks, warnings, and errors.

Linux and macOS

Option	Result
`-w`	Enables diagnostics for errors; disables diagnostics for warnings.
`-w1`	Enables diagnostics for warnings and errors.
`-w2`	Enables diagnostics for verbose warnings, warnings, and errors.
`-w3`	Enables diagnostics for remarks, warnings, and errors. Additional warnings are also enabled above level 2 warnings.
`-Wremarks`	Display remarks and comments.
`-Wbrief`	Display brief one-line diagnostics.
`-Wcheck`	Enable more strict diagnostics.
`-Werror-all`	Change all warnings and remarks to errors.
`-Werror`	Change all warnings to errors.

Windows

Option	Result
`/W0`	Enables diagnostics for errors; disables diagnostics for warnings.
`/W1`, `/W2`	Enables diagnostics for warnings and errors.
`/W3`	Enables diagnostics for remarks, warnings, and errors. Additional warnings are also enabled above level 2 warnings.
`/W4, /Wall`	Enables diagnostics for all level 3 warnings, plus informational warnings and remarks, which, in most cases, can be safely ignored.
`/W5`	Enables diagnostics for all remarks, warnings, and errors. This option produces the most diagnostic messages.
`/WL`	Display brief one-line diagnostics.
`/WX`	Change all warnings to errors.
`/Wp64`	Display diagnostics for 64-bit porting.

You can also control the display of diagnostic information with variations of the `[Q]diag` compiler option. This compiler option accepts numerous arguments and values, allowing you wide control over displayed diagnostic messages and reports.

Some of the most common variations include the following:

<table>
<tr><th>Option</th><th>Result</th></tr>
<tr><td>[Q]diag-enable<:|=>list</td><td>Enables a diagnostic message or a group of messages.</td></tr>
<tr><td>[Q]diag-disable<:|=>list</td><td>Disables a diagnostic message or a group of messages.</td></tr>
<tr><td>[Q]diag-warning<:|=>list</td><td>Tells the compiler to change diagnostics to warnings.</td></tr>
<tr><td>[Q]diag-error<:|=>list</td><td>Tells the compiler to change diagnostics to errors.</td></tr>
<tr><td>[Q]diag-remark<:|=>list</td><td>Tells the compiler to change diagnostics to remarks (comments).</td></tr>
</table>

The *list* items can be specific diagnostic IDs, one of the keywords `warn`, `remark`, or `error`, or a keyword specifying a certain group (`par`, `vec`, `driver`, `thread`, `port-linux` (available on Windows* systems), `port-win` (available on Linux* and macOS systems), or *`openmp`*). For more information, see `[Q]diag`.

Other diagnostic-related options include the following:

<table>
<tr><th>Option</th><th>Result</th></tr>
<tr><td>[Q]diag-dump</td><td>Tells the compiler to print all enabled diagnostic messages and stop compilation.</td></tr>
<tr><td>[Q]diag-file[<:|=>file]</td><td>Causes the results of diagnostic analysis to be output to a file.</td></tr>
<tr><td>[Q]diag-file-append[<:|=>file]</td><td>Causes the results of diagnostic analysis to be appended to a file.</td></tr>
<tr><td>[Q]diag-error-limit<:|=>n</td><td>Specifies the maximum number of errors allowed before compilation stops.</td></tr>
</table>

Part IV Compilation

This section contains information about features that can affect compilation, such as environment variables, and using configuration files.

Compilation Overview

Compilation Environment

You can customize the environment used during compilation using a combination of

- Configuration Files
- Environment variables
- Response Files

You can also modify the compilation by adding additional include directories for the compiler to search during compilation. See Specify Compiler Files for more information.

Default Compiler Behavior

The Intel® C++ Compiler Classic processes C/C++ language source files. Compilation can be divided into these major phases: :

- Preprocessing
- Semantic parsing
- Optimization
- Code generation
- Linking

By default, the compiler performs the first four phases of compilation and then invokes the linker to perform the linking phase. The default linkers are `ld` for Linux and `link` for Windows.

Default settings for the compiler include:

- Optimization level O2 (`-O2`)
- Floating point model = fast (`-fp-model=fast`)
- `icpc` C++ language standard: Gnu C++14 (`-std=gnu++14`)
- C++ runtime:
  - Linux: libstdc++, using headers and libraries installed on the system
  - Windows: Microsoft Visual C++ (MSVC) provided headers and libraries
- SVML and specific interfaces enabled to call into the Intel libirc library

Customize the Compilation Process

The Intel® C++ Compiler Classic provides multiple options to customize compilation.

Preprocessing

Several options are available to customize preprocessing. For example, you can:

- Specify the location of system and user header files
- Specify macros
- Stop the compilation process after preprocessing
- Send preprocessed output to stdout

You can optionally use your own preprocessor to generate a preprocessed file which can then be passed to the compiler.

For a detailed list of preprocessing options, see Preprocessor Options.

Compiling

Compiler options are not required to compile your program, but can be used to control different aspects of your application, such as:

- Code generation
- Optimization
- Output file (type, name, location)
- Linking properties
- Size of the executable
- Speed of the executable

For a detailed list of all compiler options, see Compiler Options.

Linking

You can perform the linking phase using the Intel compiler to invoke the linker (default) or by calling the linker directly.

> **NOTE** On Linux, calling the linker directly requires explicit understanding of which specific system and Intel libraries need to be linked in, as they will need to be passed directly to the linker.

To prevent default linking at compilation time, use the `-c` option. You must then explicitly pass along the generated object on the compilation command line and the compiler will create the final binary.

You can pass options to the linker for additional control of the linking phase. See Pass Options to the Linker for additional information.

See Also

Compiler Options
Specify Compiler Files
Preprocessor Options
Pass Options to the Linker

Supported Environment Variables

You can customize your system environment by specifying paths where the compiler searches for certain files such as libraries, include files, configuration files, and certain settings.

Compiler Compile-Time Environment Variables

The following table shows the compile-time environment variables that affect the compiler:

Compile-Time Environment Variable	Description
`CL` (Windows) `_CL_` (Windows)	Define the files and options you use most often with the CL variable. Note: You cannot set the CL environment variable to a string that contains an equal sign. You can use the pound sign instead. In the following example, the pound sign (#) is used as a substitute for an equal sign in the assigned string: `SET CL=/Dtest#100`
`COV_DIR` (Windows)	Same as `PROF_DIR`.
`COV_DPI` (Windows)	Same as `PROF_DPI`.
`IA32ROOT` (IA-32 architecture and Intel® 64 architecture)	Points to the directories containing the include and library files for a non-standard installation structure. **NOTE** IA-32 architecture is no longer supported on macOS*.
`ICCCFG`	Specifies the configuration file for customizing compilations when invoking the compiler using `icc`. Used instead of the default configuration file.
`ICPCCFG`	Specifies the configuration file for customizing compilations when invoking the compiler using `icpc`. Used instead of the default configuration file.
`ICLCFG` (Windows)	Specifies a configuration file, which the compiler should use instead of the default configuration file.
`INTEL_LICENSE_FILE`	Specifies the location for the Intel license file. **NOTE** On Windows, this environment variable cannot be set from Visual Studio.
`__INTEL_PRE_CFLAGS` `__INTEL_POST_CFLAGS`	Specifies a set of compiler options to add to the compile line. This is an extension to the facility already provided in the compiler configuration file `icl.cfg`. **NOTE** By default, a configuration file named `icl.cfg` (Windows), `icc.cfg` (Linux, macOS), or `icpc.cfg` (Linux, macOS) is used. This file is in the same directory as the compiler executable. To use another configuration file in another location, you can use the `ICLCFG` (Windows), `ICCCFG` (Linux, macOS), or `ICPCCFG` (Linux, macOS) environment variable to assign the directory and file name for the configuration file. You can insert command line options in the prefix position using `__INTEL_PRE_CFLAGS` or in the suffix position using `__INTEL_POST_CFLAGS`. The command line is built as follows:

Compile-Time Environment Variable	Description
	Syntax: (On Windows, use `icl`. On Linux or macOS, use `icc`) `icl/icc` *<PRE flags> <flags from configuration file> <flags from the compiler invocation> <POST flags>* **NOTE** The driver issues a warning that the compiler is overriding an option because of an environment variable, but only when you include the option `/W5` (Windows) or `-w3` (Linux and macOS).
`INTEL_TARGET_ARCH_IA32` (Linux and Windows)	Set this environment variable to target 32-bit compilations for all associated tools (this includes the compiler and Intel-specific linker tools). Without this environment variable, you will be required to use the explicit command line options, `/Qm32` on Windows and `-m32` on Linux, for each compiler invocation. **NOTE** IA-32 architecture is no longer supported on macOS. IA-32 is deprecated for other operating systems and will be removed in a future release.
`PATH`	Specifies the directories the system searches for binary executable files. **NOTE** On Windows, this also affects the search for Dynamic Link Libraries (DLLs).

`TMP`
`TMPDIR`
`TEMP`
	Specifies the location for temporary files. If none of these are specified, or writeable, or found, the compiler stores temporary files in `/tmp` (Linux, macOS) or the current directory (Windows).
	The compiler searches for these variables in the following order: `TMP`, `TMPDIR`, and `TEMP`.
	
	NOTE
	On Windows, these environment variables cannot be set from Visual Studio.

`LD_LIBRARY_PATH` (Linux)	Specifies the location for shared objects (.so files).
`DYLD_LIBRARY_PATH` (macOS)	Specifies the path for dynamic libraries.
`INCLUDE` (Windows)	Specifies the directories for the source header files (include files).
`LIB` (Windows)	Specifies the directories for all libraries used by the compiler and linker.
GNU Environment Variables and Extensions	
`CPATH` (Linux and macOS)	Specifies the path to include directory for C/C++ compilations.

Compile-Time Environment Variable	Description
C_INCLUDE_PATH (Linux and macOS)	Specifies path to include directory for C compilations.
CPLUS_INCLUDE_PATH (Linux and macOS)	Specifies path to include directory for C++ compilations.
DEPENDENCIES_OUTPUT (Linux and macOS)	Specifies how to output dependencies for make based on the non-system header files processed by the compiler. System header files are ignored in the dependency output.
GCC_EXEC_PREFIX (Linux)	Specifies alternative names for the linker (ld) and assembler (as).
GCCROOT (Linux)	Specifies the location of the GCC* binaries. Set this variable only when the compiler cannot locate the GCC binaries when using the -gcc-name option.
GXX_INCLUDE (Linux)	Specifies the location of the GCC headers. Set this variable to specify the locations of the GCC installed files when the compiler does not find the needed values as specified by the use of -gcc-name=*directory-name*/gcc or -gxx-name=*directory-name*/g++.
GXX_ROOT (Linux)	Specifies the location of the GCC binaries. Set this variable to specify the locations of the GCC installed files when the compiler does not find the needed values as specified by the use of -gcc-name=*directory-name*/gcc or -gxx-name=*directory-name*/g++.
LIBRARY_PATH (Linux and macOS)	Specifies the path for libraries to be used during the link phase.
SUNPRO_DEPENDENCIES (Linux)	This variable is the same as DEPENDENCIES_OUTPUT, except that system header files are not ignored.

NOTE INTEL_ROOT is an environment variable that is reserved for the Intel® Compiler. Its use is not supported.

Compiler Run-Time Environment Variables

The following table summarizes compiler environment variables that are recognized at run time.

Run-Time Environment Variable	Description
INTEL_CHKP_REPORT_MODE (Linux)	Changes the pointer checker reporting mode at run time. See Finding and Reporting Out-of-Bounds Errors.

Run-Time Environment Variable	Description
INTEL_ISA_DISABLE	Causes named features (in a comma-separated list) not to be visible on the host even if the CPUID reports that it has them onboard. See CPU Feature Targeting.
GNU extensions (recognized by the Intel OpenMP* compatibility library)	
GOMP_CPU_AFFINITY (Linux)	GNU extension recognized by the Intel OpenMP compatibility library. Specifies a list of OS processor IDs. You must set this environment variable before the first parallel region or before certain API calls including omp_get_max_threads(), omp_get_num_procs() and any affinity API calls. For detailed information on this environment variable, see *Thread Affinity Interface*. **Default:** Affinity is disabled
GOMP_STACKSIZE (Linux)	GNU extension recognized by the Intel OpenMP compatibility library. Same as OMP_STACKSIZE.KMP_STACKSIZE overrides GOMP_STACKSIZE, which overrides OMP_STACKSIZE. **Default:** See the description for OMP_STACKSIZE.
OpenMP Environment Variables (OMP_) and Extensions (KMP_)	
OMP_CANCELLATION	Activates cancellation of the innermost enclosing region of the type specified. If set to TRUE, the effects of the cancel construct and of cancellation points are enabled and cancellation is activated. If set to FALSE, cancellation is disabled and the cancel construct and cancellation points are effectively ignored. **NOTE** Internal barrier code will work differently depending on whether the cancellation is enabled. Barrier code should repeatedly check the global flag to figure out if the cancellation had been triggered. If a thread observes the cancellation it should leave the barrier prematurely with the return value 1 (may wake up other threads). Otherwise, it should leave the barrier with the return value 0. Enables (TRUE) or disables (FALSE) cancellation of the innermost enclosing region of the type specified. **Default:** FALSE **Example:** OMP_CANCELLATION=TRUE

Run-Time Environment Variable	Description
`OMP_DISPLAY_ENV`	Enables (`TRUE`) or disables (`FALSE`) the printing to stderr of the OpenMP version number and the values associated with the OpenMP environment variable. Possible values are: TRUE, FALSE, or VERBOSE. **Default:** FALSE **Example:** `OMP_DISPLAY_ENV=TRUE`
`OMP_DEFAULT_DEVICE`	Sets the device that will be used in a target region. The OpenMP routine `omp_set_default_device` or a `device` clause in a target pragma can override this variable. If no device with the specified device number exists, the code is executed on the host. If this environment variable is not set, device number 0 is used.
`OMP_DYNAMIC`	Enables (`TRUE`) or disables (`FALSE`) the dynamic adjustment of the number of threads. **Default:** FALSE **Example:** `OMP_DYNAMIC=TRUE`
`OMP_MAX_ACTIVE_LEVELS`	The maximum number of levels of parallel nesting for the program. Possible values: Non-negative integer. **Default:** 1
`OMP_NESTED`	Deprecated; use `OMP_MAX_ACTIVE_LEVELS` instead.
`OMP_NUM_THREADS`	Sets the maximum number of threads to use for OpenMP parallel regions if no other value is specified in the application. The value can be a single integer, in which case it specifies the number of threads for all parallel regions. The value can also be a comma-separated list of integers, in which case each integer specifies the number of threads for a parallel region at a nesting level. The first position in the list represents the outer-most parallel nesting level, the second position represents the next-inner parallel nesting level, and so on. At any level, the integer can be left out of the list. If the first integer in a list is left out, it implies the normal default value for threads is used at the outer-most level. If the integer is left out of any other level, the number of threads for that level is inherited from the previous level.

Run-Time Environment Variable	Description
	This environment variable applies to the options `Qopenmp` (Windows) or `qopenmp` (Linux and macOS), and `Qparallel` (Windows) or `parallel` (Linux and macOS). **Default:** The number of processors visible to the operating system on which the program is executed. **Syntax:** `OMP_NUM_THREADS=value[,value]*`

`OMP_PLACES`	Specifies an explicit ordered list of places, either as an abstract name describing a set of places or as an explicit list of places described by nonnegative numbers. An exclusion operator "!" can also be used to exclude the number or place immediately following the operator.
	
	For **explicit lists**, the meaning of the numbers and how the numbering is done for a list of nonnegative numbers are implementation defined. Generally, the numbers represent the smallest unit of execution exposed by the execution environment, typically a hardware thread.
	
	Intervals can be specified using the `<lower-bound>` : `<length>` : `<stride>` notation to represent the following list of numbers:
	
	`"<lower-bound>, <lower-bound> + <stride>, ..., <lower-bound> +(<length>-1)*<stride>."`
	
	When `<stride>` is omitted, a unit stride is assumed. Intervals can specify numbers within a place as well as sequences of places.
	
	`# EXPLICIT LIST EXAMPLE`
	`setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"`
	`setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"`
	`setenv OMP_PLACES "{0:4}:4:4"`
	
	The **abstract names** listed below should be understood by the execution and run-time environment:
	
	• `threads`: Each place corresponds to a single hardware thread on the target machine.
	• `cores`: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
	• `ll_caches`: Each place corresponds to a set of cores that share the last level cache on the device.

<table>
<tr><th>Run-Time Environment Variable</th><th>Description</th></tr>
<tr><td></td><td>

<code>numa_domains</code>: Each place corresponds to a set of cores for which their closest memory on the device is 1) the same memory and 2) at a similar distance from the cores.
<code>sockets</code>: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

When requesting fewer places or more resources than available on the system, the determination of which resources of type <code>abstract_name</code> are to be included in the place list is implementation-defined. The precise definitions of the abstract names are implementation defined. An implementation may also add abstract names as appropriate for the target platform. The abstract name may be appended by a positive number in parentheses to denote the length of the place list to be created, that is <code>abstract_name(num-places)</code>.
<pre><code># ABSTRACT NAMES EXAMPLE
  setenv OMP_PLACES threads
  setenv OMP_PLACES threads(4)</code></pre>
NOTE
If any numerical values cannot be mapped to a processor on the target platform the behavior is implementation-defined. The behavior is also implementation-defined when the <code>OMP_PLACES</code> environment variable is defined using an abstract name.
</td></tr>
<tr><td><code>OMP_PROC_BIND</code> (Windows, Linux)</td><td>
Sets the thread affinity policy to be used for parallel regions at the corresponding nested level. Enables (<code>TRUE</code>) or disables (<code>FALSE</code>) the binding of threads to processor contexts. If enabled, this is the same as specifying <code>KMP_AFFINITY=scatter</code>. If disabled, this is the same as specifying <code>KMP_AFFINITY=none</code>.
Acceptable values: <code>TRUE</code>, <code>FALSE</code>, or a comma separated list, each element of which is one of the following values: <code>PRIMARY</code>, <code>MASTER</code> (deprecated), <code>CLOSE</code>, <code>SPREAD</code>.
Default: <code>FALSE</code>
If set to <code>FALSE</code>, the execution environment may move OpenMP threads between OpenMP places, thread affinity is disabled, and proc_bind clauses on parallel constructs are ignored. Otherwise, the execution environment should not move OpenMP
</td></tr>
</table>

Run-Time Environment Variable	Description
	threads between OpenMP places, thread affinity is enabled, and the initial thread is bound to the first place in the OpenMP place list. If set to PRIMARY, all threads are bound to the same place as the primary thread. If set to CLOSE, threads are bound to successive places, close to where the primary thread is bound. If set to SPREAD, the primary thread's partition is subdivided and threads are bound to single place successive sub-partitions. **NOTE** KMP_AFFINITY takes precedence over GOMP_CPU_AFFINITY and OMP_PROC_BIND. GOMP_CPU_AFFINITY takes precedence over OMP_PROC_BIND.
OMP_SCHEDULE	Sets the run-time schedule type and an optional chunk size. **Default:** static, no chunk size specified **Example syntax:** OMP_SCHEDULE="[modifier:]kind[,chunk_size]" where • modifier is one of monotonic or nonmonotonic • kind is one of static, dynamic, guided, or auto • chunk_size is a positive integer **NOTE** Some environment variables are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.
OMP_STACKSIZE	Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread. Recommended size is 16M. Use the optional suffixes to specify byte units: B (bytes), K (Kilobytes), M (Megabytes), G (Gigabytes), or T (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit is assumed to be K (Kilobytes).

Run-Time Environment Variable	Description
	This variable does not affect the native operating system threads created by the user program, or the thread executing the sequential part of an OpenMP program or parallel programs created using the option `Qparallel` (Windows) or `parallel` (Linux and macOS) . The `kmp_{set,get}_stacksize_s()` routines set/retrieve the value. The `kmp_set_stacksize_s()` routine must be called from sequential part, before first parallel region is created. Otherwise, calling `kmp_set_stacksize_s()` has no effect. **Default (IA-32 architecture):** 2M **Default (Intel® 64 architecture):** 4M **NOTE** IA-32 architecture is no longer supported on macOS. IA-32 is deprecated for other operating systems and will be removed in a future release. **Related environment variables:** `KMP_STACKSIZE` (overrides `OMP_STACKSIZE`). **Syntax:** `OMP_STACKSIZE=value`
`OMP_THREAD_LIMIT`	Limits the number of simultaneously-executing threads in an OpenMP program. If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue. This environment variable is only used for programs compiled with the following options: `Qopenmp` (Windows) or `qopenmp` (Linux and macOS), or `Qparallel` (Windows) or `parallel` (Linux and macOS) . The `omp_get_thread_limit()` routine returns the value of the limit. **Default:** No enforced limit **Related environment variable:** `KMP_ALL_THREADS` (overrides `OMP_THREAD_LIMIT`). **Example syntax:** `OMP_THREAD_LIMIT=value`
`OMP_WAIT_POLICY`	Decides whether threads spin (active) or yield (passive) while they are waiting.

Run-Time Environment Variable	Description
	OMP_WAIT_POLICY=ACTIVE is an alias for KMP_LIBRARY=turnaround, and OMP_WAIT_POLICY=PASSIVE is an alias for KMP_LIBRARY=throughput. **Default:** Passive **Syntax:** OMP_WAIT_POLICY=value
OMP_DISPLAY_AFFINITY	Instructs the runtime to display formatted affinity information for all OpenMP threads in the parallel region upon entering the first parallel region and when any change occurs in the information accessible by the format specifiers listed in the OMP_AFFINITY_FORMAT entry. Possible values: TRUE or FALSE **Default:** FALSE
OMP_AFFINITY_FORMAT	Defines the format when displaying OpenMP thread affinity information. Possible values are any string with the following format field available: • %t or %{team_num}: Value returned by omp_get_team_num() • %T or %{num_teams}: Value returned by omp_get_num_teams() • %L or %{nesting_level}: Value returned by omp_get_level() • %n or %{thread_num}: Value returned by omp_get_thread_num() • %a or %{ancestor_tnum}: Value returned by omp_get_ancestor_thread_num(omp_get_level() - 1) • %H or %{host}: Name of host device • %P or %{process_id}: Process ID • %i or %{native_thread_id}: Native thread ID on the platform • %A or %{thread_affinity}: List of processor ID on which a thread may execute **Default:** 'OMP: pid %P tid %i thread %n bound to OS proc set {%A}'
OMP_MAX_TASK_PRIORITY	Controls the use of task priorities by setting the initial value. Possible values: Non-negative integer. **Default:** 0
OMP_TOOL	Controls whether the OpenMP runtime will try to register a first party tool that uses OMPT interface. Possible values: ENABLED or DISABLED.

Run-Time Environment Variable	Description
	Default: ENABLED **NOTE** Only the host OpenMP runtime is supported.
OMP_TOOL_LIBRARIES	Sets a list of first-party tool locations that use the OMPT interface. The list enumerates names of dynamically-loadable libraries with OS-specific path separator. **Default:** Empty **NOTE** Only the host OpenMP runtime is supported.
OMP_TOOL_VERBOSE_INIT	Controls whether the OpenMP runtime will verbosely log the registration of a tool that uses the OMPT interface. Possible values: • DISABLED: Do not log the registration. • STDOUT: Log the registration to stdout. • STDERR: Log the registration to stderr. • File_Name: Log the registration to the location specified by File_Name. **Default:**DISABLED **NOTE** Only the host OpenMP runtime is supported.
OMP_DEBUG	Controls whether the OpenMP runtime collects information that an OMPD library may need to support a tool. Possible values: ENABLED or DISABLED. **Default:** DISABLED **NOTE** Only the host OpenMP runtime is supported.
OMP_ALLOCATOR	Specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. **Default:** omp_default_mem_alloc **Syntax:** <PredefinedMemAllocator> | <PredefinedMemSpace> | <PredefinedMemSpace>:<Traits>

Run-Time Environment Variable	Description
	Currently supported values for <PredefinedMemAllocator> and <PredefinedMemSpace> : • `omp_default_mem_alloc` and `omp_default_mem_space` Additional values are supported if `libmemkind` is available and there is system support for it: • `omp_high_bw_mem_alloc` and `omp_high_bw_mem_space` • `omp_large_cap_mem_alloc` and `omp_large_cap_mem_space` Refer to the OpenMP specification for more information.
`OMP_NUM_TEAMS`	Sets the maximum number of teams created by a teams construct by setting nteams-var ICV. Possible values: Positive integer. **Default:** 1
`OMP_TEAMS_THREAD_LIMIT`	Sets the maximum number of OpenMP threads to use in each team created by a teams construct. Possible values: Positive integer. **Default:** <NumberOfProcessors> / <nteams-var ICV>
`KMP_AFFINITY` (Linux, Windows)	Enables run-time library to bind threads to physical processing units. You must set this environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls. For detailed information on this environment variable, see *Thread Affinity Interface*. **Default:** noverbose,warnings,noreset,respect,granularity=core,none Default (Windows with multiple processor groups): noverbose,warnings,noreset,norespect,granularity=group,compact,0,0 **NOTE** On Windows with multiple processor groups, the norespect affinity modifier is assumed when the process affinity mask equals a single processor group (which is default on Windows). Otherwise, the respect affinity modifier is used.

Run-Time Environment Variable	Description
KMP_ALL_THREADS	Limits the number of simultaneously-executing threads in an OpenMP program. If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, then the program may abort with an error message. If this limit is reached at the time an OpenMP parallel region begins, a one-time warning message may be generated indicating that the number of threads in the team was reduced, but the program will continue execution. This environment variable is only used for programs compiled with the `Qopenmp`(Windows) or `qopenmp` (Linux and macOS) option. **Default:** No enforced limit.
KMP_BLOCKTIME	Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. Use the optional character suffixes: `s` (seconds), `m` (minutes), `h` (hours), or `d` (days) to specify the units. Specify `infinite` for an unlimited wait time. **Default:** 200 milliseconds **Related Environment Variable:** `KMP_LIBRARY` environment variable.
KMP_CPUINFO_FILE	Specifies an alternate file name for a file containing the machine topology description. The file must be in the same format as `/proc/cpuinfo`. **Default:** None
KMP_DETERMINISTIC_REDUCTION	Enables (`TRUE`) or disables (`FALSE`) the use of a specific ordering of the reduction operations for implementing the reduction clause for an OpenMP parallel region. This has the effect that, for a given number of threads, in a given parallel region, for a given data set and reduction operation, a floating point reduction done for an OpenMP reduction clause has a consistent floating point result from run to run, since round-off errors are identical. **NOTE** When compiling, you must set the following flag to ensure correct behavior: • `-fp-model precise` (Linux) • `-fp:precise` (Windows) **Default:** `FALSE`

Run-Time Environment Variable	Description
`KMP_DYNAMIC_MODE`	Selects the method used to determine the number of threads to use for a parallel region when `OMP_DYNAMIC=TRUE`. Possible values: (`asat` | `load_balance` | `thread_limit`), where, • `asat`: estimates number of threads based on parallel start time; **NOTE** Support for `asat` (automatic self-allocating threads) is now deprecated and will be removed in a future release. • `load_balance`: tries to avoid using more threads than available execution units on the machine; • `thread_limit`: tries to avoid using more threads than total execution units on the machine. **Default (IA-32 architecture):** `load_balance` (on all supported OSes) **Default (Intel® 64 architecture):** `load_balance` (on all supported OSes) **NOTE** IA-32 architecture is no longer supported on macOS. IA-32 is deprecated for other operating systems and will be removed in a future release.
`KMP_HOT_TEAMS_MAX_LEVEL`	Sets the maximum nested level to which teams of threads will be hot. **NOTE** A *hot* team is a team of threads optimized for faster reuse by subsequent parallel regions. In a hot team, threads are kept ready for execution of the next parallel region, in contrast to the cold team, which is freed after each parallel region, with its threads going into a common pool of threads. For values of 2 and above, nested parallelism should be enabled. **Default:** 1
`KMP_HOT_TEAMS_MODE`	Specifies the run-time behavior when the number of threads in a hot team is reduced. Possible values:

Run-Time Environment Variable	Description
	• 0: Extra threads are freed and put into a common pool of threads. • 1: Extra threads are kept in the team in reserve, for faster reuse in subsequent parallel regions. **Default:** 0
`KMP_HW_SUBSET`	Specifies the subset of available hardware resources for the hardware topology hierarchy. The subset is specified in terms of number of units per upper layer unit starting from top layer downwards. For example, it can specify the number of sockets (top layer units), cores per socket, and the threads per core, to use with an OpenMP application. It is a convenient alternative to writing complicated explicit affinity settings or a limiting process affinity mask. You can also specify an offset value to set which resources to use. When available, you can specify attributes to select different subsets of resources. An extended syntax is available when `KMP_TOPOLOGY_METHOD=hwloc`. Depending on what resources are detected, you may be able to specify additional resources, such as NUMA nodes and groups of hardware resources that share certain cache levels. **Basic syntax:** `[num_units]ID[@`*`offset`*`][:`*`attribute`*`] [, [num_units]ID[@`*`offset`*`][:`*`attribute`*`]...]` where • `num_units` is either a positive integer, which requests an exact number of resources, or an asterisk (*), which means using all available resources at that layer (for example, using all cores per socket). If `num_units` is not specified, the asterisk (*) semantics are assumed. • `ID` is a supported ID: *`S - socket`* — *num_units* specifies the requested number of sockets. *`D - die`* — *num_units* specifies the requested number of dies per socket. *`C - core`* — *num_units* specifies the requested number of cores per die - if any - otherwise, per socket. *`T - thread`* — *num_units* specifies the requested number of HW threads per core.

Run-Time Environment Variable	Description
	Supported unit IDs are not case-sensitive. • `offset` is the number of units to skip (optional). • `attribute` is an attribute differentiating resources at a particular layer (optional). This is only available for the core layer on machines with Intel® Hybrid Technology. The attributes available to users are: • Core type: Either *intel_atom* or *intel_core* • Core efficiency: Specified as `effnum` where *num* is a number from 0 to the number of core efficiencies detected in the machine topology minus one. For example: `eff0`. The greater the efficiency number, the more performant the core. There may be more core efficiencies than core types, which can be viewed by setting `KMP_AFFINITY=verbose`. **NOTE** The hardware cache can be specified as a unit, for example L2 for L2 cache, or LL for last level cache. **Extended syntax when `KMP_TOPOLOGY_METHOD=hwloc`:** Additional IDs can be specified if detected. For example: *`N - numa`*: *num_units* specifies the requested number of NUMA nodes per upper layer unit, e.g. per socket. *`TI - tile`*: *num_units* specifies the requested number of tiles to use per upper layer unit, e.g. per NUMA node. When any *numa* or *tile* units are specified in `KMP_HW_SUBSET`, the `KMP_TOPOLOGY_METHOD` will be automatically set to `hwloc`,so there is no need to set it explicitly. If you don't specify one or more types of resource, such as socket or thread, all available resources of that type are used. The run-time library prints a warning, and the setting of `KMP_HW_SUBSET` is ignored if: • a resource is specified, but detection of that resource is not supported by the chosen topology detection method and/or • a resource is specified twice. An exception to this condition is if attributes differentiate the resource.

Run-Time Environment Variable	Description
	• attributes are used when unavailable, not detected in the machine topology, or conflict with each other. This variable does not work if the OpenMP affinity is set to `disabled`. **Default:** If omitted, the default value is to use all the available hardware resources. **Examples:** • `2s,4c,2t`: Use the first 2 sockets (s0 and s1), the first 4 cores on each socket (c0 - c3), and 2 threads per core. • `2s@2,4c@8,2t`: Skip the first 2 sockets (s0 and s1) and use 2 sockets (s2-s3), skip the first 8 cores (c0-c7) and use 4 cores on each socket (c8-c11), and use 2 threads per core. • `5C@1,3T`: Use all available sockets, skip the first core and use 5 cores, and use 3 threads per core. • `1T`: Use all cores on all sockets, 1 thread per core. • `1s, 1d, 1n, 1c, 1t`: Use 1 socket, 1 die, 1 NUMA node, 1 core, 1 thread - use HW thread as a result. • `4c:intel_atom,5c:intel_core`: Use all available sockets and use 4 Intel Atom® processor cores and 5 Intel® Core™ processor cores per socket. • `2c:eff0,3c:eff1`: Use all available sockets and use 2 cores with efficiency 0 and 3 cores with efficiency 1 per socket. • `1s, 1c, 1t`: Use 1 socket, 1 core, 1 thread. This may result in using single thread on a 3-layer topology architecture, or multiple threads on 4-layer or 5-layer architecture. Result may even be different on the same architecture, depending on `KMP_TOPOLOGY_METHOD` specified, as hwloc can often detect more topology layers than default method used by the OpenMP run-time library. To see the result of the setting, you can specify the `verbose` modifier in for the `KMP_AFFINITY` environment variable. The OpenMP run-time library will output to stderr stream the information about discovered HW topology before and after the `KMP_HW_SUBSET` setting was applied. For example, on Intel® Xeon Phi™ 7210 CPU in SNC-4 Clustering Mode, the setting `KMP_AFFINITY=verbose KMP_HW_SUBSET=1N,1L2,1L1,1T` outputs various

Run-Time Environment Variable	Description
	verbose information to stderr, including the following lines about discovered HW topology before and after KMP_HW_SUBSET was applied: • Info #191: KMP_AFFINITY: 1 socket x 4 NUMA domains/socket x 8 tiles/NUMA domain x 2 cores/tile x 4 threads/core. (64 total cores) • Info #191: KMP_HW_SUBSET 1 socket x 1 NUMA domain/socket x 1 tile/NUMA domain x 1 core/tile x 1 thread/core (1 total cores)
KMP_INHERIT_FP_CONTROL	Enables (TRUE) or disables (FALSE) the copying of the floating-point control settings of the primary thread to the floating-point control settings of the OpenMP worker threads at the start of each parallel region. **Default:** TRUE
KMP_LIBRARY	Selects the OpenMP run-time library execution mode. The values for this variable are serial, turnaround, or throughput. **Default:** throughput
KMP_PLACE_THREADS	Deprecated; use KMP_HW_SUBSET instead.
KMP_SETTINGS	Enables (TRUE) or disables (FALSE) the printing of OpenMP run-time library environment variables during program execution. Two lists of variables are printed: user-defined environment variables settings and effective values of variables used by OpenMP run-time library. **Default:** FALSE
KMP_STACKSIZE	Sets the number of bytes to allocate for each OpenMP thread to use as its private stack. Recommended size is 16m. Use the optional suffixes to specify byte units: B (bytes), K (Kilobytes), M (Megabytes), G (Gigabytes), or T (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit is assumed to be K (Kilobytes). This variable does not affect the native operating system threads created by the user program nor the thread executing the sequential part of an OpenMP program or parallel programs created using the option Qparallel (Windows) or parallel (Linux and macOS) . KMP_STACKSIZE overrides GOMP_STACKSIZE, which overrides OMP_STACKSIZE. **Default (IA-32 architecture):** 2m

<table>
<tr><th>Run-Time Environment Variable</th><th>Description</th></tr>
<tr><td></td><td>Default (Intel® 64 architecture): 4m

NOTE IA-32 architecture is no longer supported on macOS. IA-32 is deprecated for other operating systems and will be removed in a future release.</td></tr>
<tr><td><code>KMP_TOPOLOGY_METHOD</code></td><td>Forces OpenMP to use a particular machine topology modeling method.

Possible values are:
<code>all</code>: Lets OpenMP choose which topology method is most appropriate based on the platform and possibly other environment variable settings.
<code>cpuid_leaf11</code>: Decodes the APIC identifiers as specified by leaf 11 of the cpuid instruction.
<code>cpuid_leaf4</code>: Decodes the APIC identifiers as specified in leaf 4 of the cpuid instruction.
<code>cpuinfo</code>: If <code>KMP_CPUINFO_FILE</code> is not specified, forces OpenMP to parse <code>/proc/cpuinfo</code> to determine the topology (Linux only). If <code>KMP_CPUINFO_FILE</code> is specified as described above, uses it (Windows or Linux).
<code>group</code>: Models the machine as a 2-level map, with level 0 specifying the different processors in a group, and level 1 specifying the different groups (Windows 64-bit only) .
NOTE
Support for <code>group</code> is now deprecated and will be removed in a future release. Use <code>all</code> instead.
<code>flat</code>: Models the machine as a flat (linear) list of processors.
<code>hwloc</code>: Models the machine as the Portable Hardware Locality* (hwloc) library does. This model is the most detailed and includes, but is not limited to: numa nodes, packages, cores, hardware threads, caches, and Windows processor groups.
Default: <code>all</code></td></tr>
<tr><td><code>KMP_USER_LEVEL_MWAIT</code></td><td>Enables (<code>TRUE</code>) or disables (<code>FALSE</code>) the use of user-level mwait as alternative to putting waiting threads to sleep, if available, either from ring3 or <code>WAITPKG</code>.

Default: <code>FALSE</code></td></tr>
</table>

Run-Time Environment Variable	Description
`KMP_VERSION`	Enables (`TRUE`) or disables (`FALSE`) the printing of OpenMP run-time library version information during program execution. **Default:** `FALSE`
`KMP_WARNINGS`	Enables (`TRUE`) or disables (`FALSE`) displaying warnings from the OpenMP run-time library during program execution. **Default:** `TRUE`
Profile Guided Optimization (PGO_) Environment Variables	
`INTEL_PROF_DUMP_CUMULATIVE`	When using interval profile dumping (initiated by `INTEL_PROF_DUMP_INTERVAL` or the function `_PGOPTI_Set_Interval_Prof_Dump`) during the execution of an instrumented user application, allows creation of a single `.dyn` file to contain profiling information instead of multiple `.dyn` files. If not set, executing an instrumented user application creates a new `.dyn` file for each interval. Setting this environment variable is useful for applications that do not terminate or those that terminate abnormally (bypass the normal exit code).
`INTEL_PROF_DUMP_INTERVAL`	Initiates interval profile dumping in an instrumented user application. This environment variable may be used to initiate Interval Profile Dumping in an instrumented application. See Interval Profile Dumping for more information
`INTEL_PROF_DYN_PREFIX`	Specifies the prefix to be used for the `.dyn` filename to distinguish it from the other `.dyn` files dumped by other PGO runs. Executing the instrumented application generates a .dyn filename as follows: <prefix>_<timestamp>_<pid>.dyn, where <prefix> is the identifier that you have specified. **NOTE** The value specified in this environment variable must not contain < > : " / \ | ? * characters. The default naming scheme is used if an invalid prefix is specified.
`PROF_DIR`	Specifies the directory where profiling files (files with extensions `.dyn`, `.dpi`, `.spi` and so on) are stored. The default is to store the `.dyn` files in the

Run-Time Environment Variable	Description
	source directory of the file containing the first executed instrumented routine in the binary compiled with [Q]prof-gen option. This variable applies to all three phases of the profiling process: • Instrumentation compilation and linking • Instrumented execution • Feedback compilation
PROF_DPI	Name for the .dpi file. **Default:** pgopti.dpi
PROF_DUMP_INTERVAL	Deprecated; use INTEL_PROF_DUMP_INTERVAL instead.
PROF_NO_CLOBBER	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges data from all dynamic information files and creates a new pgopti.dpi file if the .dyn files are newer than an existing pgopti.dpi file. When this variable is set, the compiler does not overwrite the existing pgopti.dpi file. Instead, the compiler issues a warning. You must remove the pgopti.dpi file if you want to use additional dynamic information files.

The following table summarizes CPU environment variables that are recognized at run time.

Runtime configuration	Default value	Description
CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE	32KB	Forces CL_DEVICE_PRIVATE_MEM_SIZE for the CPU device to be the given value. The value must include the unit; for example: 8MB, 8192KB, 8388608B. **NOTE** You must compile your host application with sufficient stack size.
CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE	32KB	Forces CL_DEVICE_LOCAL_MEM_SIZE for CPU device to be the given value. The value needs to be set with size including units, examples: 8MB, 8192KB, 8388608B.

Runtime configuration	Default value	Description
		NOTE You must compile your host application with sufficient stack size. Our recommendation is to set the stack size equal to twice the local memory size to cover possible application and OpenCL Runtime overheads.
`CL_CONFIG_CPU_EXPENSIVE_MEM_OPT`	0	A bitmap indicating enabled expensive memory optimizations. These optimizations may lead to more JIT compilation time, but give some performance benefit. **NOTE** Currently, only the least significant bit is available. Available bits: • 0: OpenCL address space alias analysis
`CL_CONFIG_CPU_STREAMING_ALWAYS`	False	Controls whether non-temporal instructions are used.

See Also

Qopenmp compiler option
parallel, Qparallel compiler option
prof-gen, Qprof-gen compiler option
Thread Affinity Interface

Pass Options to the Linker

Specify Linker Options

This topic describes the options that let you control and customize linking with tools and libraries and define the output of the linker.

Linux and macOS

This section describes options specified at compile-time that take effect at link-time to define the output of the `ld` linker. See the `ld` man page for more information on the linker.

Option	Description
`-L`*directory*	Instruct the linker to search *directory* for libraries.

Option	Description
`-Qoption,`*tool,list*	Passes an argument list to another program in the compilation sequence, such as the assembler or linker.
`-shared`	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable.
`-shared-libgcc`	`-shared-libgcc` has the opposite effect of `-static-libgcc` . When it is used, the GNU standard libraries are linked in dynamically, allowing the user to override the static linking behavior when the `-static` option is used. **NOTE** Note: By default, all C++ standard and support libraries are linked dynamically.
`-shared-intel`	Specifies that all Intel-provided libraries should be linked dynamically.
`-static`	Causes the executable to link all libraries statically, as opposed to dynamically. When `-static` is not used: • `/lib/ld-linux.so.2` is linked in • all other libs are linked dynamically When `-static` is used: • `/lib/ld-linux.so.2` is not linked in • all other libs are linked statically
`-static-libgcc`	This option causes the GNU standard libraries to be linked in statically.
`-Bstatic` `-Bdynamic`	Either option is placed in the linker command line corresponding to its location on the user command line to control the linking behavior of any library being passed in via the command line.
`-static-intel`	This option causes Intel-provided libraries to be linked in statically. It is the opposite of `-shared-intel`.
`-Wl,`*optlist*	This option passes a comma-separated list (*optlist*) of options to the linker.
`-Xlinker` *val*	This option passes a value (*val*), such as a linker option, an object, or a library, directly to the linker.

Windows

This section describes options specified at compile-time that take effect at link-time.

You can use the `link` option to pass options specifically to the linker at compile time. For example:

```
icl a.cpp libfoo.lib /link -delayload:comct132.dll
```

In this example, the compiler recognizes that `libfoo.lib` is a library that should be linked with `a.cpp`, so it does not need to follow the `link` option on the command line. The compiler does not recognize `-delayload:comct132.dll`, so the `link` option is used to direct the option to the linking phase. On C++, you can use the `Qoption` option to pass options to various tools, including the linker. You can also use `#pragma comment` on C++ to pass options to the linker. For example:

```
#pragma comment(linker, "/defaultlib:mylib.lib")
```

OR

```
#pragma comment(lib, "mylib.lib")
```

Both examples instruct the compiler to link `mylib.lib` at link time.

Linking Tools and Options

This topic describes how to use the Intel® linking tools, `xild` (Linux* and macOS) and `xilink` (Windows*).

The Intel® linking tools behave differently on different platforms. The following sections summarize the primary differences between linking behavior.

Linux* and macOS Linking Behavior Summary

The linking tool invokes the Intel® C++ Compiler to perform IPO if objects containing `IR` (intermediate representation) are found. These are mock objects. The tool invokes GNU `ld` to link the application.

The command-line syntax for `xild` is the same as that of the GNU linker:

```
xild [<options>] <normal command-line>
```

where:

- `[<options>]`: One or more options supported only by `xild` (optional).
- `<normal command-line>`: Linker command line containing a set of valid arguments for `ld`.

To create the file `app` using IPO, use the option `o`*[filename]* as shown in the following example:

```
xild -qipo-fas-oapp a.o b.o c.o
```

The linking tool calls the compiler to perform IPO for objects containing IR and creates a new list of object(s) to be linked. The linker then calls `ld` to link the object files that are specified in the new list and produce the application with the name specified by the `o` option. The linker supports the `ipo`[*n*] option and `ipo-separate` option.

To display a list of the supported link options from `xild`, use the following command:

```
$ xild -qhelp
```

Windows* Linking Behavior Summary

The linking tool invokes the Intel® C++ Compiler to perform multi-file IPO if objects containing `IR` (intermediate representation) is found. These are mock objects. It invokes the Microsoft linker `link.exe` to link the application.

The command-line syntax for the Intel® linker is the same as that of the Microsoft linker:

```
xilink [<options>] <normal command-line>
```

where:

Windows* Linking Behavior Summary

• `[<options>]`: One or more options supported only by `xilink` (optional).
• `<normal command-line>`: Linker command line containing a set of valid arguments for the Microsoft linker.
 	
To place the multifile IPO executable in `ipo_file.exe`, use the linker option `out`:*[filename]* , for example:
 	
`xilink -qipo-fas/out:ipo_file.exe a.obj b.obj c.obj`
 	
The linker calls the compiler to perform IPO for objects containing `IR` and creates a new list of object(s) to be linked. The linker calls Microsoft `link.exe` to link the object files that are specified in the new list and produce the application with the name specified by the `out`:*[filename]* linker option.
 	
To display a list of support link options from `xilink` , use the following command:
 	
`>> xilink /qhelp`
 	
`xilink.exe` accepts all the options of `link.exe` and will pass them on to `link.exe` at the final linking stage.

Using the Linking Tools

You must use the Intel® linking tools to link your application if the following conditions apply:

- Your source files were compiled with multi-file IPO enabled. Multi-file IPO is enabled by specifying compiler option `[Q]ipo`.
- You would normally invoke the GNU linker (`ld`) to link your application.
- You would normally invoke the Microsoft linker (`link.exe`) to link your application.

Linker Options

The following table provides information on linking options.

Linking Tools Option	Description
`qdiag-`[*type*]=[*diag-list*]	Controls the display of diagnostic information. The *type* is an action to perform on diagnostics. Possible values are: • **Enable:** Enables a diagnostic message or a group of messages. • **Disable:** Disables a diagnostic message or a group of messages. The *diag-list* is a diagnostic group or ID value. Possible values are: • **thread:** Specifies diagnostic messages that help in thread-enabling a program. • **vec:** Specifies diagnostic messages issued by the vectorizer. • **par:** Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer). • **openmp:** Specifies diagnostic messages issued by the OpenMP* parallelizer. • **warn:** Specifies diagnostic messages that have a "warning" severity level. • **error:** Specifies diagnostic messages that have an "error" severity level. • **remark:** Specifies diagnostic messages that are remarks or comments. • **cpu-dispatch:** Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default. • **id[,id,...]:** Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each "id".

Linking Tools Option	Description
	• **tag[,tag,...]:** Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each "tag". **NOTE** Diagnostic messages generated by this option can be affected by other options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.
`m32` (Linux* only), `m64` (Linux* and macOS) `Qm32`, `Qm64` (Windows*)	`[Q]m32` generates code for IA-32 architecture. Option `-m32` is only available on Linux* systems. `[Q]m64` generates code for Intel® 64 architecture. For example, when your compilation environment is configured for Intel® 64 architecture, and you use [Q]m32 with the compiler, you also need to use `qm32` on the linker command line to make sure the proper compilation target is set up for any IPO compilations or the final link.

See Also

Using IPO from the command line

Specify Alternate Tools and Paths

Use the `Qlocation` option to specify an alternate path for a tool. This option accepts two arguments using the following syntax:

Linux and macOS

```
-Qlocation,tool,path
```

Windows

```
/Qlocation,tool,path
```

where *tool* designates which compilation tool is associated with the alternate *path*.

tool	Description
`cpp`	Specifies the preprocessor for the compiler.
`c`	Specifies the Intel® C++ Compiler Classic .
`asm`	Specifies the assembler.
`link`	Specifies the linker.

Use the `Qoption` option to pass an option specified by *optlist* to a *tool*, where *optlist* is a comma-separated list of options. The syntax for this command is:

Linux and macOS

```
-Qoption,tool,optlist
```

Windows

```
/Qoption,tool,optlist
```

where

- *tool* designates which compilation tool receives the *optlist*
- *optlist* indicates one or more valid argument strings for the designated program. If the argument is a command-line option, you must include the hyphen. If the argument contains a space or tab character, the entire argument must be enclosed in quotation characters (""). Separate multiple arguments with commas.

Use Configuration Files

You can decrease the time you spend entering command-line options by using the configuration file to automate command-line entries. Configuration files are automatically processed every time you run the Intel® C++ Compiler Classic. You can insert any valid command-line options into the configuration file. The compiler processes options in the configuration file in the order in which they appear, followed by the specified command-line options when the compiler is invoked.

NOTE
Options in the configuration file are executed every time you run the compiler. If you have varying option requirements for different projects, use Using Response Files .

Sample Configuration Files

The default configuration files `icc.cfg` and `icpc.cfg` (Linux* and macOS) or or `icl.cfg` (Windows*), are located in the same directory as the compiler executable file. If you want to use a different configuration file than the default, you can use the `ICCCFG/ICPCCFG` (for Linux* and macOS) or `ICLCFG` (for Windows) environment variables to specify the location of another configuration file.

NOTE
Anytime you instruct the compiler to use a different configuration file, the default configuration file(s) are ignored.

The following examples illustrate basic configuration files.

Linux

```
## Sample icpc.cfg file
  -I/my_headers
```

Windows

```
## Sample icl.cfg file
  /Ic:\my_headers
```

In the Windows examples, the compiler reads the configuration file and invokes the `I` option every time you run the compiler, along with any options specified on the command line.

See Also

Supported Environment Variables
Using Response Files

Use Response Files

You can use response files to:

- Specify options used during particular compilations or projects.
- Save this information in individual files.

Response files are invoked as options on the command line. Options in response files are inserted in the command line at the point where the response file is invoked. Unlike configuration files, which are automatically processed every time you run the compiler, response files must be invoked as an option on the command line. If you create a response file without specifying it on the command line, it will not be invoked.

Sample Response Files

Linux and macOS

```
# response file: response1.txt
# compile with these options
  -w0
# end of response1 file
```

```
# response file: response2.txt
# compile with these options
  -O0
# end of response2 file
```

Windows

```
# response file: response1.txt
# compile with these options
  /W0
# end of response1 file
```

```
# response file: response2.txt
# compile with these options
  /Od
# end of response2 file
```

Use response files to decrease the time spent entering command-line options and to ensure consistency by automating command-line entries. Use individual response files to maintain options for specific projects.

Any number of options or file names can be placed on a line in a response file. Several response files can be referenced in the same command line. The following example shows how to specify a response file on the command line:

macOS

```
icpx @response1.txt prog1.cpp @response2.txt prog2.cpp
```

NOTE
An "at" sign (`@`) must precede the name of the response file on the command line.

See Also

Using Configuration Files

Global Symbols and Visibility Attributes for Linux* and macOS

This topic applies to C/C++ applications for Linux* and macOS only.

A global symbol is one that is visible outside the compilation unit (single source file and its include files) in which it is declared. In C/C++, this means anything declared at file level without the `static` keyword. For example:

```
int x = 5;          // global data definition
extern int y;       // global data reference
int five()          // global function definition
  { return 5; }
extern int four(); // global function reference
```

A complete program consists of a main program file and possibly one or more shareable object (`.so`) files that contain the definitions for data or functions referenced by the main program. Similarly, shareable objects might reference data or functions defined in other shareable objects. Shareable objects are so called because if more than one simultaneously executing process has the shareable object mapped into its virtual memory, there is only one copy of the read-only portion of the object resident in physical memory. The main program file and any shareable objects that it references are collectively called the components of the program.

Each global symbol definition or reference in a compilation unit has a visibility attribute that controls how (or if) it may be referenced from outside the component in which it is defined. There are five possible values for visibility:

- **`EXTERNAL`** – The compiler must treat the symbol as though it is defined in another component. For a definition, this means that the compiler must assume that the symbol will be overridden (preempted) by a definition of the same name in another component. See Symbol Preemption. If a function symbol has external visibility, the compiler knows that it must be called indirectly and can inline the indirect call stub.
- **`DEFAULT`** – Other components can reference the symbol. Furthermore, the symbol definition may be overridden (preempted) by a definition of the same name in another component.
- **`PROTECTED`** – Other components can reference the symbol, but it cannot be preempted by a definition of the same name in another component.
- **`HIDDEN`** – Other components cannot directly reference the symbol. However, its address might be passed to other components indirectly (for example, as an argument to a call to a function in another component, or by having its address stored in a data item reference by a function in another component).
- **`INTERNAL`** – The symbol cannot be referenced outside its defining component, either directly or indirectly.

Static local symbols (in C/C++, declared at file scope or elsewhere with the keyword static) usually have `HIDDEN` visibility— they cannot be referenced directly by other components (or, for that matter, other compilation units within the same component), but they might be referenced indirectly.

NOTE
Visibility applies to references as well as definitions. A symbol reference's visibility attribute is an assertion that the corresponding definition will have that visibility.

Specify Symbol Visibility Explicitly

You can explicitly set the visibility of an individual symbol using the `visibility` attribute on a data or function declaration. For example:

```
int i __attribute__ ((visibility("default")));
void __attribute__ ((visibility("hidden"))) x () {...}
extern void y() __attribute__ ((visibility("protected")));
```

The `visibility` declaration attribute accepts one of the five keywords:

- `external`
- `default`
- `protected`
- `hidden`
- `internal`

The value of the `visibility` declaration attribute overrides the default set by the options `-fpic`, `-fvisibility`, or `-fno-common` .

If you have a number of symbols for which you wish to specify the same `visibility` attribute, you can set the visibility using one of the five command line options:

- `-fvisibility-external=`*file*
- `-fvisibility-default=`*file*
- `-fvisibility-protected=`*file*
- `-fvisibility-hidden=`*file*
- `-fvisibility-internal=`*file*

where *file* is the pathname of a file containing a list of the symbol names whose visibility you wish to set.

The symbol names in the file are separated by white space (blanks, TAB characters, or newlines). For example, the command line option: `-fvisibility-protected=prot.txt`, where file `prot.txt` contains:

```
a
    bcd
  e
```

This sets protected visibility for symbols *a*, *b*, *c*, *d*, and *e*.

This has the same effect as `__attribute__ ((visibility=("protected")))` on the declaration for each of the symbols.

NOTE
These two ways to explicitly set visibility are mutually exclusive– you may use `__attribute((visibility()))` on the declaration or specify the symbol name in a file, but not both.

You can set the default visibility for symbols using one of the command line options:

- `-fvisibility=external`
- `-fvisibility=default`
- `-fvisibility=protected`
- `-fvisibility=hidden`
- `-fvisibility=internal`

This option sets the visibility for symbols not specified in a visibility list file and that do not have `__attribute__ ((visibility=()))` in their declaration. For example, the command line options: `-fvisibility=protected -fvisibility-default=prot.txt`, where file `prot.txt` is as previously described, will cause all global symbols except *a*, *b*, *c*, *d*, and *e* to have protected visibility. Those five symbols, however, will have default visibility, and thus will be preemptable.

Save Compiler Information in Your Executable

If you want to save information about the compiler in your executable, use the `[Q]sox` option to save:

- Compiler version number and options used to produce the executable.
- Profile data and inlining information (if optional arguments were specified).

Linux

To view the information stored in the object file, use the `objdump` command. For example:

```
objdump -sj comment a.out
strings -a a.out | grep comment:
```

Windows

To view the linker directives stored in string format in the object file, use the `link` command. For example:

```
link /dump /directives filename.obj
```

In the output, the ?-comment linker directive displays the compiler version information. To search your executable for compiler information, use the `findstr` command. For example, to search for any strings that contain the substring "Compiler":

```
findstr "Compiler" filename.exe
```

Link Debug Information

Linux*

Use option `g` at compile time to tell the compiler to generate symbolic debugging information in the object file.

Use option `gsplit-dwarf` to create a separate object file containing DWARF debug information. Because the DWARF object file is not used by the linker, this reduces the amount of debug information the linker must process and it results in a smaller executable file. See `gsplit-dwarf` for detailed information.

macOS

You can link the DWARF debug information from the object files for an executable using `dsymutil`, a utility included with Xcode*. By linking the debug information in an executable, you eliminate the need to retain object files specifically for debugging purposes.

The utility runs automatically in the following cases:

- When you use the Intel® C++ Compiler to compile directly from source to executable using the command line with option `g`. For example:

Example

```
icc -g myprogram.c
```

- When you compile using Xcode*.

In other cases, you must explicitly run `dsymutil`, such as when you compile using a make file that builds `.o` files and subsequently links the program.

Windows*

Use option `Z7` at compile time or option `debug` at link time to tell the compiler to generate symbolic debugging information in the object file. Alternately, use option `Zi` at link time to generate executables with debug information in the `.pdb` file.

Part V

Optimization and Programming

This section contains information about features related to code optimization and program performance improvement.

OpenMP* Support

The Intel® C++ Compiler Classic supports most of the OpenMP* Application Programming Interface versions 5.0 and 5.1. For the complete OpenMP specification, read the specifications available from the OpenMP web site (http://www.openmp.org; see OpenMP Specifications on that site). The descriptions of OpenMP language characteristics in this documentation often use terms defined in that specification.

The OpenMP API provides symmetric multiprocessing (SMP) with the following major features:

- Relieves you from implementing the low-level details of iteration space partitioning, data sharing, thread creation, scheduling, or synchronization.
- Provides the benefit of performance available from shared memory multiprocessor and multi-core processor systems on all supported Intel architectures, including those processors with Intel® Hyper-Threading Technology (Intel® HT Technology).

The compiler performs transformations to generate multithreaded code based on your placement of OpenMP pragmas in the source program, making it simple to add threading to existing software. The compiler compiles parallel programs and supports the industry-standard OpenMP pragmas.

The compiler provides Intel®-specific extensions to the OpenMP specification including run-time library routines and environment variables. A summary of the compiler options appear in the OpenMP Options Quick Reference.

Parallel Processing with OpenMP

To compile with the OpenMP API, add the pragmas to your code. The compiler processes the code and internally produces a multithreaded version which is then compiled into an executable with the parallelism implemented by threads that execute parallel regions or constructs.

Using Other Compilers

The OpenMP specification does not define interoperability of multiple implementations, so the OpenMP implementation supported by other compilers and OpenMP support in the Intel® C++ Compiler Classic might not be interoperable. Even if you compile and build the entire application with one compiler, be aware that different compilers might not provide OpenMP source compatibility that enable you to compile and link the same set of application sources with a different compiler and get the expected parallel execution results.

Add OpenMP* Support

To add OpenMP* support to your application, do the following:

1. Add the appropriate OpenMP pragmas to your source code.
2. Compile the application with the `/Qopenmp` (Windows*) or `-qopenmp` (Linux* and macOS) option.
3. For applications with large local or temporary arrays, you may need to increase the stack space available at runtime. In addition, you may need to increase the stack allocated to individual threads by using the `OMP_STACKSIZE` environment variable or by setting the corresponding library routines.

You can set other environment variables to control multi-threaded code execution.

OpenMP Pragma Syntax

To add OpenMP support to your application, first declare the OpenMP header and then add appropriate OpenMP pragmas to your source code.

To declare the OpenMP header, add the following in your code:

```
#include <omp.h>
```

OpenMP pragmas use a specific format and syntax. Intel Extension Routines to OpenMP describes the OpenMP extensions to the specification that have been added to the Intel® C++ Compiler Classic.

To use pragmas in your source, use this syntax:

```
<prefix> <pragma> [<clause>, ...] <newline>
```

where:

- *<prefix>* - Required for all OpenMP pragmas. The prefix must be `#pragma omp`.
- *<pragma>* - A valid OpenMP pragma. Must immediately follow the prefix.
- [*<clause>*] - Optional. Clauses can be in any order and repeated as necessary, unless otherwise restricted.
- *<newline>* - A required component of pragma syntax. It precedes the structured block that is enclosed by this pragma.

The pragmas are interpreted as comments if you omit the `/Qopenmp` (Windows) or `-qopenmp` (Linux and macOS) option.

The following example demonstrates one way of using an OpenMP pragma to parallelize a loop:

```
#include <omp.h>
void simple_omp(int *a){
  int i;
  #pragma omp parallel for
  for (i=0; i<1024; i++)
    a[i] = i*2;
}
```

Compile the Application

The `/Qopenmp` (Windows) or `-qopenmp` (Linux and macOS) option enables the parallelizer to generate multi-threaded code based on the OpenMP pragmas in the source. The code can be executed in parallel on single processor, multi-processor, or multi-core processor systems.

The `/Qopenmp` (Windows) or `-qopenmp` (Linux and macOS) option works with both `-O0` (Linux and macOS) and `/Od` (Windows*) and with any optimization level of `O1`, `O2` and `O3`.

Specifying `-O0` (Linux and macOS) or `/Od` (Windows) with the `/Qopenmp` (Windows) or `-qopenmp` (Linux and macOS) option helps to debug OpenMP applications.

Compile your application using a command similar to one of the following:

Linux

```
icpc -qopenmp source_file
```

macOS

```
icpc -qopenmp source_file
```

Windows

```
icl /Qopenmp source_file
```

For example, to compile the previous code example without generating an executable, use the `c` option:

Linux

```
icpc -qopenmp -c parallel.cpp
```

macOS

```
icpc -qopenmp -c parallel.cpp
```

Windows

```
icl /Qopenmp /c parallel.cpp
```

Configure the OpenMP Environment

Before you run the multi-threaded code, you can set the number of desired threads using the OpenMP environment variable, `OMP_NUM_THREADS`.

See Also

c compiler option
O compiler option
OpenMP* Examples
qopenmp, Qopenmp compiler option
Supported Environment Variables

Parallel Processing Model

A program containing OpenMP* pragmas begins execution as a single thread, called the initial thread of execution. The initial thread executes sequentially until the first parallel construct is encountered.

The `omp parallel` pragma defines the extent of the parallel construct. When the initial thread encounters a parallel construct, it creates a team of threads, with the initial thread becoming the primary thread of the team. All program statements enclosed by the parallel construct are executed in parallel by each thread in the team, including all routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes all statements encountered during the execution of a construct by a thread, including all called routines.

When a thread encounters the end of a structured block enclosed by a parallel construct, the thread waits until all threads in the team have arrived. When that happens the team is dissolved, and only the primary thread continues execution of the code following the parallel construct. The other threads in the team enter a wait state until they are needed to form another team. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

The following example illustrates, from a high level, the execution model for the OpenMP constructs. The comments in the code explain the structure of each construct or section.

```
main() {                      // Begin serial execution.
  ...                         // Only the initial thread executes
  #pragma omp parallel        // Begin a parallel construct and form a team.
  {
    #pragma omp sections      // Begin a worksharing construct.
    {
       #pragma omp section    // One unit of work.
       {...}
       #pragma omp section    // Another unit of work.
       {...}
    }                         // Wait until both units of work complete.
    ...                       // This code is executed by each team member.
```

```
    #pragma omp for nowait   // Begin a worksharing Construct
    for(...) {               // Each iteration chunk is unit of work.
      ...                    // Work is distributed among the team members.
    }                        // End of worksharing construct.
                             // nowait was specified so threads proceed.
    #pragma omp critical     // Begin a critical section.
    {...}                    // Only one thread executes at a time.
    ...                      // This code is executed by each team member.
    #pragma omp barrier      // Wait for all team members to arrive.
    ...                      // This code is executed by each team member.
  }                          // End of Parallel Construct
                             // Disband team and continue serial execution.
  ...                        // Possibly more parallel constructs.
}                            // End serial execution.
```

Use Orphaned Pragmas

In routines called from within parallel constructs, you can also use pragmas. Pragmas that are not in the static extent of the parallel construct, but are in the dynamic extent, are called orphaned pragmas. Orphaned pragmas allow you to execute portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

```
int main(void) {
  #pragma omp parallel {
     phase1();
  }
}

void phase1(void) {
  #pragma omp for // This is an orphaned pragma.
  for(i=0; i < n; i++) { some_work(i); }
}
```

This is an orphaned `omp for` loop pragma since the parallel region is not lexically present in routine `phase1`.

Data Environment

You can control the data environment of OpenMP constructs by using data environment clauses supported by the construct. You can also privatize named global-lifetime objects by using the `threadprivate` pragma.

Refer to the OpenMP specification for the full list of data environment clauses. Some commonly used ones include:

- `default`
- `shared`
- `private`
- `firstprivate`
- `lastprivate`
- `reduction`
- `linear`
- `map`

You can use several pragma clauses to control the data scope attributes of variables for the duration of the construct in which you specify them; however, if you do not specify a data scope attribute clause on a pragma, the behavior for the variable is determined by the default scoping rules, which are described in the OpenMP specification, for the variables affected by the directive.

Determine How Many Threads to Use

For applications where the workload depends on application input that can vary widely, delay the decision about the number of threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree-based structures, and size of list-based structures. Similarly, for applications designed to run on systems where the processor count can vary widely, defer choosing the number of threads to employ until application runtime when the machine size can be examined.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, the calibration results can be made persistent by storing the results in a permanent place like the file system.

Avoid simultaneously using more threads than the number of processing units on the system. This situation causes the operating system to multiplex threads on the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by the library, because it is possible that the user has outer-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Use the `num_threads` clause on parallel regions to control the number of threads employed and use the `if` clause on parallel regions to decide whether to employ multiple threads at all. The `omp_set_num_threads()` routine can also be used, but it also affects parallel regions created by the calling thread. The `num_threads` clause is local in its effect, so it does not impact other parallel regions. The disadvantages of explicitly setting the number of threads are:

1. In a system with a large number of processors, your application will use some but not all of the processors.
2. In a system with a small number of processors, your application may force over subscription that results in poor performance.

The Intel OpenMP runtime will create the same number of threads as the available number of logical processors unless you use the omp_set_num_threads() routine. To determine the actual limits, use omp_get_thread_limit() and omp_get_max_active_levels(). Developers should carefully consider their thread usage and nesting of parallelism to avoid overloading the system. The `OMP_THREAD_LIMIT` environment variable limits the number of OpenMP threads to use for the whole OpenMP program. The `OMP_MAX_ACTIVE_LEVELS` environment variable limits the number of active nested parallel regions.

Binding Sets and Binding Regions

The binding task set for an OpenMP construct is the set of tasks that are affected by, or provide the context for, the execution of its region. It can be all tasks, the current team tasks, all tasks of the current team that are generated in the region, the binding implicit task, or the generating task.

The binding thread set for an OpenMP construct is the set of threads that are affected by, or provide the context for, the execution of its region. It can be all threads on a device, all threads in a contention group, all primary threads executing an enclosing teams region, the current team, or the encountering thread.

The binding region for an OpenMP construct is the enclosing region that determines the execution context and the scope of the effects of the directive:

- The binding region for an omp ordered construct is the innermost enclosing omp for loop region.
- The binding region for a omp taskwait construct is the innermost enclosing omp task region.
- For all other constructs for which the binding thread set is the current team or the binding task set is the current team tasks, the binding region is the innermost enclosing region.
- For constructs for which the binding task set is the generating task, the binding region is the region of the generating task.

- A omp parallel construct need not be active to be a binding region.
- A construct need not be explicit to be a binding region.
- A region never binds to any region outside of the innermost enclosing parallel region.

Worksharing Using OpenMP*

To get the maximum performance benefit from a processor with multi-core and Intel® Hyper-Threading Technology (Intel® HT Technology), an application needs to be executed in parallel. Parallel execution requires threads, and threading an application is not a simple thing to do; using OpenMP* can make the process a lot easier. Using the OpenMP pragmas, most loops with no loop-carried dependencies can be threaded with one simple statement. This topic explains how to start using OpenMP to parallelize loops, which is also called worksharing.

Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Most loops can be threaded by inserting one pragma immediately prior to the loop. Further, by leaving the details to the Intel® C++ Compiler Classic and OpenMP, you can spend more time determining which loops should be threaded and how to best restructure the algorithms for maximum performance. The maximum performance of OpenMP is realized when it is used to thread hotspots, the most time-consuming loops in your application.

The power and simplicity of OpenMP is demonstrated by looking at an example. The following loop converts a 32-bit RGB (red, green, blue) pixel to an 8-bit gray-scale pixel. One pragma, which has been inserted immediately before the loop, is all that is needed for parallel execution.

```
#pragma omp parallel for
for (i=0; i < numPixels; i++) {
  pGrayScaleBitmap[i] = (unsigned BYTE)
     (pRGBBitmap[i].red * 0.299 +
      pRGBBitmap[i].green * 0.587 +
      pRGBBitmap[i].blue * 0.114);
}
```

First, the example uses worksharing, which is the general term used in OpenMP to describe distribution of work across threads. When worksharing is used with the `for` construct, as shown in the example, the iterations of the loop are distributed among multiple threads so that each loop iteration is executed exactly once with different iterations executing if there is more than one available threads. The `for` construct on its own only distributes the loop iterations among existing threads. The example uses a `parallel for` construct, which combines `parallel` and `for` constructs to first create a team of threads and then distribute the loop iterations among the threads. Since there is no explicit `num_threads` clause, OpenMP determines the number of threads to create and how to best create, synchronize, and destroy them. OpenMP places the following five restrictions on which loops can be threaded:

- The loop variable must be of type signed or unsigned integer, random access iterator, or pointer.
- The comparison operation must be in the form `loop_variable <, <=, >, >=, or != loop_invariant_expression` of a compatible type.
- The third expression or increment portion of the `for` loop must be either addition or subtraction by a loop invariant value.
- If the comparison operation is < or <=, the loop variable must increment on every iteration; conversely, if the comparison operation is > or >=, the loop variable must decrement on every iteration.
- The loop body must be single-entry-single-exit, meaning no jumps are permitted from inside to outside the loop, with the exception of the `exit` statement that terminates the whole application. If the statements `goto` or `break` are used, the statements must jump within the loop, not outside it. Similarly, for exception handling, exceptions must be caught within the loop.

Although these restrictions might sound somewhat limiting, non-conforming loops can frequently be rewritten to follow these restrictions.

Basics of Compilation

Using the OpenMP pragmas requires an OpenMP-compatible compiler and thread-safe libraries. Adding the `/Qopenmp` (Windows*) or `-qopenmp` (Linux* and macOS) option to the compiler instructs the compiler to pay attention to the OpenMP pragmas and to generate multi-threaded code. If you omit the `/Qopenmp` (Windows) or `-qopenmp` (Linux and macOS) option, the compiler will ignore OpenMP pragmas, which provides a very simple way to generate a single-threaded version without changing any source code. To compile programs containing target and related constructs for offloading to a GPU, the `-fopenmp-targets=spir64` and `/Qopenmp-targets:spir64` flags are needed on Linux and Windows respectively.

For conditional compilation, the compiler defines the `_OPENMP` macro. If needed, the macro can be tested as shown in the following example.

```
#ifdef _OPENMP
   fn();
#endif
```

A Few Simple Examples

The following examples illustrate how simple OpenMP is to use. In common practice, additional issues need to be addressed, but these examples illustrate a good starting point.

In the first example, the loop clips an array to the range from 0 to 255.

```
// clip an array to 0 <= x <= 255
for (i=0; i < numElements; i++) {
  if (array[i] < 0)
  array[i] = 0;
  else if (array[i] > 255)
    array[i] = 255;
}
```

You can thread it using a single OpenMP pragma; insert the pragma immediately prior to the loop:

```
#pragma omp parallel for
for (i=0; i < numElements; i++) {
  if (array[i] < 0)
  array[i] = 0;
  else if (array[i] > 255)
    array[i] = 255;
}
```

In the second example, the loop generates a table of square roots for the numbers from 0 to 100.

```
double value;
double roots[100];
for (value = 0.0; value < 100.0; value ++) { roots[(int)value] = sqrt(value); }
```

Thread the loop by changing the loop variable to a signed integer or unsigned integer and inserting a `#pragma omp parallel for` pragma.

```
int value;
double roots[100];
#pragma omp parallel for
for (value = 0; value < 100; value ++) { roots[value] = sqrt((double)value); }
```

Avoid Data Dependencies and Race Conditions

When a loop meets all five loop restrictions (listed above) and the compiler threads the loop, the loop still might not work correctly due to the existence of data dependencies.

Data dependencies exist when different iterations of a loop (more specifically a loop iteration that is executed on a different thread) read or write the same location in shared memory. Consider the following example that calculates factorials.

```
// Each loop iteration writes a value that a different iteration reads.
#pragma omp parallel for
for (i=2; i < 10; i++) { factorial[i] = i * factorial[i-1]; }
```

The compiler will thread this loop, but the threading will fail because at least one of the loop iterations is data-dependent upon a different iteration. This situation is referred to as a race condition. Race conditions can only occur when using shared resources (like memory) and parallel execution. To address this problem either rewrite the loop or pick a different algorithm, one that does not contain the race condition.

Race conditions are difficult to detect because, for a given case or system, the threads might win the race in the order that happens to make the program function correctly. Because a program works once does not mean that the program will work under all conditions. Testing your program on various machines, some with Intel® Hyper-Threading Technology and some with multiple physical processors, is a good starting point to help identify race conditions.

Traditional debuggers are useless for detecting race conditions because they cause one thread to stop the race while the other threads continue to significantly change the runtime behavior; however, thread checking tools can help.

Manage Shared and Private Data

Nearly every loop (in real applications) reads from or writes to memory; it's your responsibility, as the developer, to instruct the compiler what memory should be shared among the threads and what memory should be kept private. When memory is identified as shared, all threads access the same memory location. When memory is identified as private, however, a separate copy of the variable is made for each thread to access in private. When the loop ends, the private copies are destroyed. By default, all variables are shared except for the loop variable, which is private.

Memory can be declared as private in two ways:

- Declare the variable inside the loop-really inside the parallel OpenMP pragma-without the static keyword.
- Specify the private clause on an OpenMP pragma.

The following loop fails to function correctly because the variable *temp* is shared. It should be private.

```
// Variable temp is shared among all threads, so while one thread
// is reading variable temp another thread might be writing to it
#pragma omp parallel for
for (i=0; i < 100; i++) {
  temp = array[i];
  array[i] = do_something(temp);
}
```

The following two examples both declare the variable *temp* as private memory, which solves the problem.

```
#pragma omp parallel for
for (i=0; i < 100; i++) {
  int temp; // variables declared within a parallel construct
            // are, by definition, private
  temp = array[i];
  array[i] = do_something(temp);
}
```

The *temp* variable can also be made private in the following way:

```
#pragma omp parallel for private(temp)
for (i=0; i < 100; i++) {
  temp = array[i];
  array[i] = do_something(temp);
}
```

Every time you use OpenMP to parallelize a loop, you should carefully examine all memory references, including the references made by called functions. Variables declared within a parallel construct are defined as private except when they are declared with the `static` declarator, because static variables are not allocated on the stack.

Reductions

Loops that accumulate a value are fairly common, and OpenMP has a specific clause to accommodate them. Consider the following loop that calculates the sum of an array of integers.

```
sum = 0;
for (i=0; i < 100; i++) {
  sum += array[i]; // this variable needs to be shared to generate
                   // the correct results, but private to avoid
                   // race conditions from parallel execution
}
```

The variable sum in the previous loop must be shared to generate the correct result, but it also must be private to permit access by multiple threads. OpenMP provides the `reduction` clause that is used to efficiently combine the mathematical reduction of one or more variables in a loop. The following example demonstrates how the loop can use the `reduction` clause to generate the correct results.

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 100; i++) { sum += array[i]; }
```

In the case of the example listed above, the reduction provides private copies of the variable *sum* for each thread, and when the threads exit, it adds the values together and places the result in the one global copy of the variable.

The following table lists the possible reduction operations, along with their initial values (mathematical identity values).

Operation	***private*** **Variable Initialization Value**
+ (addition)	0
- (subtraction)	0
* (multiplication)	1
& (bitwise and)	~0
| (bitwise or)	0
^ (bitwise exclusive or)	0
&& (conditional and)	1
|| (conditional or)	0

Multiple reductions in a loop are possible by specifying comma-separated variables and operations on a given `parallel` construct. Reduction variables must meet the following requirements:

- can be listed in just one reduction.
- cannot be declared constant.
- cannot be declared private in the `parallel` construct.

Load Balancing and Loop Scheduling

Load balancing, the equal division of work among threads, is among the most important attributes for parallel application performance. Load balancing is extremely important, because it ensures that the processors are busy most, if not all, of the time. Without a balanced load, some threads may finish significantly before others, leaving processor resources idle and wasting performance opportunities.

Within loop constructs, poor load balancing is often caused by variations in compute time among loop iterations. It is usually easy to determine the variability of loop iteration compute time by examining the source code. In most cases, you will see that loop iterations consume a uniform amount of time. When that is not true, it may be possible to find a set of iterations that consume similar amounts of time. For example, sometimes the set of all even iterations consumes about as much time as the set of all odd iterations. Similarly, it might be the case that the set of the first half of the loop consumes about as much time as the second half. In contrast, it might be impossible to find sets of loop iterations that have a uniform execution time. Regardless of the case, you should provide this extra loop scheduling information to OpenMP so it can better distribute the iterations of the loop across the threads (and therefore processors) for optimum load balancing.

If you know that all loop iterations consume roughly the same amount of time, the OpenMP `schedule` clause should be used to distribute the iterations of the loop among the threads in roughly equal amounts via the scheduling policy. In addition, you need to minimize the chances of memory conflicts that may arise because of false sharing due to using large chunks. This behavior is possible because loops generally touch memory sequentially, so splitting up the loop in large chunks— like the first half and second half when using two threads— will result in the least chance for overlapping memory. While this may be the best choice for memory issues, it may be bad for load balancing. Unfortunately, the reverse is also true; what might be best for load balancing may be bad for memory performance. You must strike a balance between optimal memory usage and optimal load balancing by measuring the performance to see what method produces the best results.

Use the following general form on the `parallel` construct to schedule an OpenMP loop:

```
#pragma omp parallel for schedule(kind [, chunk size])
```

Four different loop scheduling types (kinds) can be provided to OpenMP, as shown in the following table. The optional parameter (chunk), when specified, must be a positive integer.

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is *loop_count*/*number_of_threads*. Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately *loop_count*/*number_of_threads*.

Kind	Description
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
runtime	Uses the `OMP_SCHEDULE` environment variable to specify which one of the three loop-scheduling types should be used. `OMP_SCHEDULE` is a string formatted exactly the same as would appear on the parallel construct.

Assume that you want to parallelize the following loop.

```
for (i=0; i < NumElements; i++) {
   array[i] = StartVal;
   StartVal++;
}
```

As written, the loop contains a data dependency, making it impossible to parallelize without a change. The new loop, shown below, fills the array in the same manner, but without data dependencies. The new loop benefits from using the SIMD instructions generated by the compiler.

```
#pragma omp parallel for
for (i=0; i < NumElements; i++)
{
     array[i] = StartVal + i;
}
```

Observe that the code is not 100% identical because the value of variable *StartVal* is not incremented. As a result, when the parallel loop is finished, the variable will have a value different from the one produced by the serial version. If the value of `StartVal` is needed after the loop, the additional statement, shown below, is needed.

```
// This works and is identical to the serial version.
#pragma omp parallel for
for (i=0; i < NumElements; i++)
{
     array[i] = StartVal + i;
}
StartVal += NumElements;
```

OpenMP Tasking Model

The OpenMP tasking model enables parallelization of a large range of applications. A task is an instance of executable code and its data environment that can be scheduled for execution by threads.

The `task` Construct

The `task` construct defines an explicit `task` region as shown in the following example:

```
void test1(LIST *head) {
  #pragma omp parallel shared(head)
  {
    #pragma omp single
     {
      LIST *p = head;
      while (p != NULL) {
        #pragma omp task firstprivate(p)
        {
          do_work1(p);
```

```
      }
      p = p->next;
    }
  }
}
```

The binding thread set of the `task` region is the current parallel team. A `task` region binds to the innermost enclosing `parallel` region. When a thread encounters a `task` construct, a task is generated from the structured block enclosed in the construct. The encountering thread may immediately execute the task, or defer its execution. A `task` construct may be nested inside an outer task, but the `task` region of the inner task is not a part of the `task` region of the outer task.

Use Clauses with the **task** Construct

The `task` construct can take optional clauses. The data environment of the task is created according to the data-sharing attribute clauses on the `task` construct and any defaults that apply. The example below shows a way to generate `N` tasks with one thread and execute the generated tasks with the threads in the parallel team:

```
double data[N];
int i;
#pragma omp parallel shared(data)
{
  #pragma omp single private(i)
  {
    for (i=0, i<N; i++)
    {
      #pragma omp task firstprivate(i) shared(data))
      {
        do_work(data, i);
      }
    }
  }
}
```

Task Scheduling

When a thread reaches a task scheduling point, it may perform a task switch, suspending the current task and beginning or resuming execution of a different task bound to the current team. Refer to the OpenMP 5.1 specifications for the full list of task scheduling point locations. Some examples include:

- the point where a task is explicitly generated.
- the point immediately following the generation of an explicit task.
- after the last instruction of a `task` region.
- in a `taskwait` region.
- in a `taskyield` region.
- in implicit and explicit barrier regions.

NOTE
Task scheduling points dynamically divide `task` regions into parts. Each part is executed from start to finish without interruption. Different parts of the same `task` region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different schedulable tasks is unspecified. A correct program must behave correctly and consistently with all conceivable scheduling sequences.

The **taskwait** Construct

The `taskwait` construct specifies a wait on the completion of child tasks generated since the beginning of the current task. A `taskwait` region binds to the current `task` region. The binding thread set of the `taskwait` region is the current team.

The `taskwait` region includes an implicit task scheduling point in the current `task` region. The current `task` region is suspended at the task scheduling point until execution of all its child tasks generated before the `taskwait` region is completed.

```
#pragma omp task // TASK1
{
  ...
  #pragma omp task // TASK 2 (child of TASK1)
  {
    do_work1();
  }
  #pragma omp task // TASK3 (child of TASK 1)
  {
    ...
    #pragma omp task // TASK4 (child of TASK3, not TASK1)
    {
      do_work2();
    }
    ...
  }
  #pragma omp taskwait // suspend TASK1; wait for TASK2 and TASK3 to complete
  ...
}
```

The `taskyield` Construct

The `taskyield` construct specifies that the current task can be suspended at that point and the thread may switch to the execution of a different task. You can use this construct to provide an explicit task scheduling point at a particular point in the task.

See Also

OMP_SCHEDULE
qopenmp, Qopenmp
Supported Environment Variables

Control Thread Allocation

The `KMP_HW_SUBSET` and `KMP_AFFINITY` environment variables allow you to control how the OpenMP* runtime uses the hardware threads on the processors. These environment variables allow you to try different thread distributions on the cores of the processors and determine how these threads are bound to the cores. You can use the environment variables to work out what is optimal for your application.

The `KMP_HW_SUBSET` variable controls the allocation of hardware resources and the `KMP_AFFINITY` variable controls how the OpenMP threads are bound to those resources.

Control Thread Distribution

The `KMP_HW_SUBSET` variable controls the hardware resources that will be used by the program. This variable often specifies three layers of machine topology: the number of sockets to use, how many cores to use per socket, and how many threads to use per core. For example, `KMP_HW_SUBSET=2s,12c,2t` means to use two sockets, 12 cores per socket, and two threads per core, giving a total of 48 available hardware threads.

When more layers exist (NUMA domain, tile, etc.) in the machine topology, you can specify those layers as well. For example, `KMP_HW_SUBSET=2s,2n,8c,2t` means to use two sockets, two NUMA domains per socket, eight cores per NUMA domain, and two threads per core, giving a total of 64 available hardware threads. For historical reasons, when a layer is not explicitly specified in KMP_HW_SUBSET, it is assumed you want all the resources in that unspecified layer. You can use `KMP_AFFINITY=verbose` to see all the different detected layers in the machine. For example, `KMP_HW_SUBSET=2s,2t` is interpreted to mean use two sockets, all cores per socket (and possibly all resources of other detected layers as well), and two threads per layer.

When available, you can specify core attributes to choose different sets of cores. The core attributes are appended to the regular core layer specification with a colon (:) and attribute. There are two attributes to help filter types of cores:

1. Core type, specified as `intel_core`, or `intel_atom`.
2. Core efficiency, specified as `eff`*num* where *num* is a non-negative integer from zero to the number of core efficiencies detected minus one. The larger the efficiency the more performant the core. For example, `KMP_HW_SUBSET=4c:eff0,5c:eff1` will select all sockets, four cores of efficiency 0, five cores of efficiency 1, and all threads per those cores.

There is also a special syntax to explicitly request all resources at a specific layer. Instead of specifying a positive integer, you can use an optional asterisk (*). For example, `KMP_HW_SUBSET='*c:eff0'` or `KMP_HW_SUBSET=c:eff0` will request all the cores of efficiency 0.

Consider a system with 24 cores and four hardware threads per core. While specifying two threads per core often yields better performance than one thread per core, specifying three or four threads per core may or may not improve the performance. This variable enables you to conveniently measure the performance of up to four threads per core.

For example, you can determine the effects of assigning 24, 48, 72, or the maximum 96 OpenMP threads in a system with 24 cores by specifying the following variable settings:

To Assign This Number of Threads ...	... Use This Setting
24	`KMP_HW_SUBSET=24c,1t`
48	`KMP_HW_SUBSET=24c,2t`
72	`KMP_HW_SUBSET=24c,3t`
96	`KMP_HW_SUBSET=24c,4t`

Caution
Take care when using the `OMP_NUM_THREADS` variable along with this variable. Using the `OMP_NUM_THREADS` variable can result in over or under subscription.

NOTE
If you use `KMP_HW_SUBSET` to specify more resources than the system has, the runtime will issue a warning and ignore the setting. For example, setting `KMP_HW_SUBSET=24c,5t` will be ignored on a system where each core has four hardware threads.

Control Thread Bindings

The `KMP_AFFINITY` variable controls how the OpenMP threads are bound to the hardware resources allocated by the `KMP_HW_SUBSET` variable. While this variable can be set to several binding or affinity types, the following are the recommended affinity types to use to run your OpenMP threads on the processor:

- *compact*: Distribute the threads sequentially among the cores.
- *scatter*: Distribute the threads among the cores in a round robin manner. Distribution is one thread per core initially, followed by repeat distribution among the cores.

The following table shows how the threads are bound to the cores when you want to use three threads per core on two cores by specifying `KMP_HW_SUBSET=2c,3t`:

Affinity	OpenMP Threads on Core 0	OpenMP Threads on Core 1
`KMP_AFFINITY=compact`	0, 1, 2	3, 4, 5
`KMP_AFFINITY=scatter`	0, 2, 4	1, 3, 5

Determine the Best Setting

To determine the best thread distribution and bindings using these variables, use the following:

1. Ensure that your OpenMP code is working properly before using these environment variables.
2. Establish a baseline with your current OpenMP code to compare to the performance when you allocate the threads to a processor.
3. Measure the performance of distributing one, two, three, or four threads per core by use the `KMP_HW_SUBSET` variable.
4. Measure the performance of binding the threads to the cores by using the `KMP_AFFINITY` variable.

See Also

Thread Affinity Interface
Supported Environment Variables

OpenMP* Pragmas

This is a summary of the OpenMP* pragmas supported in the Intel® C++ Compiler Classic. For detailed information about the OpenMP API, see the *OpenMP Application Program Interface* Version 5.1 specification, which is available from the OpenMP web site.

PARALLEL Pragma

Use this pragma to form a team of threads and execute those threads in parallel.

Pragma	Description
omp parallel	Specifies that a structured block should be run in parallel by a team of threads.

TASKING Pragma

Use these pragmas for deferring execution.

Pragma	Description
omp task	Specifies a code block whose execution may be deferred.

Pragma	Description
omp taskloop	Specifies that the iterations of one or more associated for loops should be executed using OpenMP tasks. The iterations are distributed across tasks that are created by the construct and scheduled to be executed in parallel by the current team.

WORKSHARING Pragmas

Use these pragmas to share work among a team of threads.

Pragma	Description
omp for	Specifies a work-sharing loop. Iterations of the loop are executed in parallel by the threads in the team.
omp sections	Defines a set of structured blocks that will be distributed among the threads in the team.
omp single	Specifies that a block of code is to be executed by only one thread in the team.

SYNCHRONIZATION Pragmas

Use these pragmas to synchronize between threads.

Pragma	Description
omp atomic	Specifies a computation that must be executed atomically.
omp barrier	Specifies a point in the code where each thread must wait until all threads in the team arrive.
omp critical	Specifies a code block that is restricted to access by only one thread at a time.
omp flush	Identifies a point at which a thread's temporary view of memory becomes consistent with the memory.
omp master	Specifies a code block that must be executed only once by the primary thread of the team.
omp ordered	Specifies a block of code that the threads in a team must execute in the natural order of the loop iterations, or as a stand-alone directive, specifies cross-iteration dependences in a doacross loop-nest. The following clauses are available as Intel-specific extensions of the OpenMP* specification: `monotonic` Specifies a block of code in which the value of the new list item on each iteration of the associated SIMD loop(s) corresponds to the value of the original list item before entering the associated loop, plus the number of the iterations for which the conditional update happens prior to the current iteration, times linear-step. The value corresponding to the sequentially last iteration of the associated loop(s) is assigned to the original list item. Use with the `simd` clause.

Pragma	Description
	`overlap` Specifies a block of code that has to be executed scalar for overlapping `inx` values and parallel for different `inx` values within SIMD loop. Use with the `simd` clause.
omp taskgroup	Causes the program to wait until the completion of all enclosed and descendant tasks.
omp taskwait	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
omp taskyield	Specifies that the current task can be suspended at this point in favor of execution of a different task.

Data Environment Pragmas

Use these pragmas to affect the data environment.

Pragma	Description
omp scan	Specifies a scan computation that updates each list item in each iteration of the loop.
omp threadprivate	Specifies a list of globally-visible variables that will be allocated private to each thread.

Offload Target Control Pragmas

Use these pragmas to control execution on one or more offload targets.

Pragma	Description
omp distribute	Specifies that the iterations of one or more loops should be distributed among the initial threads of all thread teams in a league.
omp target enter data	
omp target exit data	
omp teams	Creates a league of thread teams inside a target region to execute a structured block in the initial thread of each team.

Vectorization Pragmas

Use these pragmas to control execution on vector hardware.

Pragma	Description
omp simd	Transforms the loop into a loop that will be executed concurrently using SIMD instructions. The `early_exit` clause is an Intel-specific extension of the OpenMP* specification. `early_exit`

Pragma	Description
	Allows vectorization of multiple exit loops. When this clause is specified: • Each operation before last lexical early exit of the loop may be executed as if early exit were not triggered within the SIMD chunk. • After the last lexical early exit of the loop, all operations are executed as if the last iteration of the loop was found. • Each list item specified in the `linear` clause is computed based on the last iteration number upon exiting the loop. • The last value for `linear` clauses and conditional `lastprivates` clauses are preserved with respect to scalar execution. • The last value for `reductions` clauses are computed as if the last iteration in the last SIMD chunk was executed up on exiting the loop. • The shared memory state may not be preserved with regard to scalar execution. • Exceptions are not allowed.
omp declare simd	Creates a version of a function that can process multiple arguments using Single Instruction Multiple Data (SIMD) instructions from a single invocation from a SIMD loop.

Cancellation Constructs

Pragma	Description
omp cancel	Requests cancellation of the innermost enclosing region of the type specified, and causes the encountering task to proceed to the end of the cancelled construct.
omp cancellation point	Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified. This construct does not implement a synchronization between threads or tasks.

User-Defined Reduction Pragma

Use this pragma to define reduction identifiers that can be used as reduction operators in a reduction clause.

Pragma	Description
omp declare reduction	Declares User-Defined Reduction (UDR) functions (reduction identifiers) that can be used as reduction operators in a reduction clause.

Combined and Composite Pragmas

Use these pragmas as shortcuts for multiple pragmas in sequence. A combined construct is a shortcut for specifying one construct immediately nested inside another construct. A combined construct is semantically identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

A composite construct is composed of two constructs but does not have identical semantics to specifying one of the constructs immediately nested inside the other. A composite construct either adds semantics not included in the constructs from which it is composed or the nesting of the one construct inside the other is not conforming.

Pragma	Description
omp distribute parallel for [1]	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
omp distribute parallel for simd[1]	Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.
omp distribute simd [1]	Specifies a loop that will be distributed across the primary threads of the teams region. It will be executed concurrently using SIMD instructions.
omp for simd[1]	Specifies that the iterations of the loop will be distributed across threads in the team. Iterations executed by each thread can also be executed concurrently using SIMD instructions.
omp parallel for	Provides an abbreviated way to specify a parallel region containing only a FOR construct.
omp parallel for simd	Specifies a parallel construct that contains one for simd construct and no other statement.
omp parallel sections	Specifies a parallel construct that contains only a sections construct.
	Creates a device data environment and executes the parallel region on that device.
	Provides an abbreviated way to specify a target construct that contains an omp target parallel for construct and no other statement between them.
	Specifies a target construct that contains an omp target parallel for simd construct and no other statement between them.
	Specifies a target construct that contains an omp simd construct and no other statement between them.
omp target teams	Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the primary thread in each team executing the structured block.
omp target teams distribute	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct.
omp target teams distribute parallel for	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct.

Pragma	Description
omp target teams distribute parallel for simd	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a teams construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.
omp target teams distribute simd	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the primary threads of all thread teams in a league created by a teams construct. It will be executed concurrently using SIMD instructions.
omp taskloop simd [1]	Specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using OpenMP* tasks.
omp teams distribute	Creates a league of thread teams and specifies that loop iterations will be distributed among the primary threads of all thread teams in the league.
omp teams distribute parallel for	Creates a league of thread teams and specifies that the associated loop can be executed in parallel by multiple threads that are members of multiple teams.
omp teams distribute parallel for simd	Creates a league of thread teams and specifies that the associated loop can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams.
omp teams distribute simd	Creates a league of thread teams and specifies that the associated loop will be distributed across the primary threads of the teams and executed concurrently using SIMD instructions.

Footnotes:

[1] This directive specifies a composite construct.

OpenMP* Library Support

This section provides information about OpenMP* run-time library routines, Intel® compiler extension routines to OpenMP, OpenMP support libraries and how to use them, and the thread affinity interface.

OpenMP* Run-time Library Routines

OpenMP* provides run-time library routines to help you manage your program in parallel mode. Many of these run-time library routines have corresponding environment variables that can be set as defaults. The run-time library routines let you dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library routine overrides any corresponding environment variable.

> **Caution**
> Running OpenMP runtime library routines may initialize the OpenMP runtime environment, which might cause a situation where subsequent programmatic setting of OpenMP environment variables has no effect. To avoid this situation, you can use the Intel extension routine `kmp_set_defaults()` to set OpenMP environment variables.

The compiler supports all the OpenMP run-time library routines. Refer to the OpenMP API specification for detailed information about using these routines.

Include the appropriate declarations of the routines in your source code by adding a statement similar to the following:

```
#include <omp.h>
```

The header files are provided in the `../include` (Linux* and macOS) or `..\include` (Windows*) directory of your compiler installation.

Thread Team Routines

Routines that affect and monitor thread teams in the current contention group.

Routine	Description
`void omp_set_num_threads(int nthreads)`	Sets the number of threads to use for subsequent parallel regions created by the calling thread.
`int omp_get_num_threads(void)`	Returns the number of threads that are being used in the current parallel region. This function does not necessarily return the value inherited by the calling thread from the `omp_set_num_threads()` function.
`int omp_get_max_threads(void)`	Returns the number of threads available to subsequent parallel regions created by the calling thread.
`int omp_get_thread_num(void)`	Returns the thread number of the calling thread, within the context of the current parallel region.
`int omp_in_parallel(void)`	Returns `TRUE` if called within the dynamic extent of a parallel region executing in parallel; otherwise returns `FALSE`.
`void omp_set_dynamic(int dynamic_threads)`	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If `dynamic_threads` is `TRUE`, dynamic threads are enabled. If `dynamic_threads` is `FALSE`, dynamic threads are disabled. Dynamic threads are disabled by default.
`int omp_get_dynamic(void)`	Returns `TRUE` if dynamic thread adjustment is enabled, otherwise returns `FALSE`.
`int omp_get_cancellation(void)`	Returns `TRUE` if cancellation is enabled, otherwise returns `FALSE`. This routine can be affected by the setting for environment variable `OMP_CANCELLATION`.

Routine	Description
`void omp_set_nested(int nested)` **NOTE** This has been deprecated.	Enables or disables nested parallelism. If `nested` is `TRUE`, nested parallelism is enabled. If `nested` is `FALSE`, nested parallelism is disabled. Nested parallelism is disabled by default.
`int omp_get_nested(void)` **NOTE** This has been deprecated.	Returns `TRUE` if nested parallelism is enabled, otherwise returns `FALSE`.
`void omp_set_schedule(omp_sched_t kind,int chunk_size)`	Determines the schedule of a worksharing loop that is applied when '`runtime`' is used as the schedule kind.
`void omp_get_schedule(omp_sched_kind *kind,int *chunk_size)`	Returns the schedule of a worksharing loop that is applied when the '`runtime`' schedule is used.
`int omp_get_thread_limit(void)`	Returns the maximum number of simultaneously executing threads in an OpenMP program.
`int omp_get_supported_active_levels(void)`	Returns the number of active levels of parallelism supported by the implementation.
`void omp_set_max_active_levels(int max_active_levels)`	Limits the number of nested active parallel regions. The value of `max_active_levels` must evaluate to a non-negative integer.
`int omp_get_max_active_levels(void)`	Returns the maximum number of nested active parallel regions.
`int omp_get_level(void)`	Returns the number of nested parallel regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region.
`int omp_get_ancestor_thread_num(int level)`	Returns the thread number of the ancestor at a given nest level of the current thread.
`int omp_get_team_size(int level)`	Returns the size of the thread team to which the ancestor or the current thread belongs for a given nested level.
`int omp_get_active_level(void)`	Returns the number of nested, active parallel regions enclosing the task that contains the call.

Thread Affinity Routines

Routines that affect and access thread affinity policies that are in effect.

Function	Description
omp_proc_bind_t omp_get_proc_bind(void)	Returns the currently active thread affinity policy, which can be initialized by the environment variable OMP_PROC_BIND. This policy is used for subsequent nested parallel regions.
int omp_get_num_places(void)	Returns the number of places available to the execution environment in the place list of the initial task, usually threads, cores, or sockets.
int omp_get_place_num_procs(int place_num)	Returns the number of processors associated with the place numbered place_num. The routine returns zero when place_num is negative or is greater than or equal to omp_get_num_places().
void omp_get_place_proc_ids(int place_num, int *ids)	Returns the numerical identifiers of each processor associated with the place numbered place_num. The numerical identifiers are non-negative and their meaning is implementation defined. The numerical identifiers are returned in the array ids and their order in the array is implementation defined.The array ids must be sufficiently large to contain omp_get_place_num_procs(place_num) elements. The routine has no effect when place_num is negative or greater than or equal to omp_get_num_places().
int omp_get_place_num(void)	Returns the place number of the place to which the encountering thread is bound. The returned value is between 0 and omp_get_num_places() - 1, inclusive. When the encountering thread is not bound to a place, the routine returns -1.
int omp_get_partition_num_places(void)	Returns the number of places in the place partition of the innermost implicit task.
void omp_get_partition_place_nums(int *place_nums)	Returns the list of place numbers corresponding to the places in the place-partition-var ICV of the innermost implicit task. The array place_nums must be sufficiently large to contain omp_get_partition_num_places() elements.
void omp_set_affinity_format(const char *format)	Sets the affinity format to be used on the device by setting the value of the affinity-format-var ICV.
size_t omp_get_affinity_format(char *buffer, size_t size)	Returns the value of the affinity-format-var ICV on the device.
void omp_display_affinity(const char *format)	Prints the OpenMP thread affinity information using the format specification provided.
size_t omp_capture_affinity(char *buffer, size_t size, const char *format)	Prints the OpenMP thread affinity information into a buffer using the format specification provided.

Teams Region Routines

Routines that affect and monitor the league of teams that may execute a teams region.

Function	Description
`int omp_get_num_teams(void)`	Returns the number of initial teams in the current teams region.
`int omp_get_team_num(void)`	Returns the initial team number of the calling thread.
`void omp_set_num_teams(int num_teams)`	Affects the number of threads to be used for subsequent teams regions that do not specify a `num_teams` clause.
`int omp_get_max_teams(void)`	Returns an upper bound on the number of teams that could be created by a teams construct without a `num_teams` clause that is encountered after execution returns from this routine.
`void omp_set_teams_thread_limit(int thread_limit)`	Defines the maximum number of OpenMP threads that can participate in each contention group created by a teams construct.
`int omp_get_teams_thread_limit(void)`	Returns the maximum number of OpenMP threads available to participate in each contention group created by a teams construct.

Tasking Routines

Routines that pertain to OpenMP explicit tasks.

Function	Description
`int omp_get_max_task_priority(void)`	Returns the maximum value that can be specified in the `priority` clause.
`int omp_in_explicit_task(void)`	Returns `TRUE` if called within an explicit task region; otherwise returns `FALSE`.
`int omp_in_final(void)`	Returns `TRUE` if called within a final task region; otherwise returns `FALSE`.

Resource Relinquishing Routines

Routines that relinquish resources used by the OpenMP runtime. These routines are only effective on the host device.

Function	Description
`int omp_pause_resource(omp_pause_resource_t kind, int device_num)`	Allows the runtime to relinquish resources used by OpenMP on the specified device. The routine returns zero in case of success, and non-zero otherwise.
`int omp_pause_resource_all(omp_pause_resource_t kind)`	Allows the runtime to relinquish resources used by OpenMP on all devices. The routine returns zero in case of success, and non-zero otherwise.

Device Information Routines

Routines that pertain to the set of devices that are accessible to an OpenMP program.

Function	Description
`int omp_get_num_procs(void)`	Returns the number of processors available to the program.
`void omp_set_default_device(int device_number)`	Sets the default device number.
`int omp_get_default_device(void)`	Returns the default device number.
`int omp_get_num_devices(void)`	Returns the number of target devices.
`int omp_get_device_num(void)`	Returns the device number of the device on which the calling thread is executing.
`int omp_is_initial_device(void)`	Returns `TRUE` if the current task is running on the host device; otherwise, `FALSE`.
`int omp_get_initial_device(void)`	Returns the device number of the host device. The value of the device number is implementation defined. If it is between 0 and `omp_get_num_devices()`-1, then it is valid in all device constructs and routines; if it is outside that range, then it is only valid in the device memory routines and not in the `device` clause.

Lock Routines

Use these routines to affect OpenMP locks.

Function	Description
`void omp_init_lock(omp_lock_t *lock)`	Initializes the lock to the unlocked state.
`void omp_init_nest_lock(omp_nest_lock_t *lock)`	Initializes the nested lock to the unlocked state. The nesting count for the nested lock is set to zero.
`void omp_init_lock_with_hint(omp_lock_t *lock, omp_sync_hint_t hint)`	Initializes the lock to the unlocked state, optionally choosing a specific lock implementation based on `hint`. See the OpenMP specification for the available hints.
`void omp_init_nest_lock_with_hint(omp_nest_lock_t *lock, omp_sync_hint_t hint)`	Initializes the nested lock to the unlocked state, optionally choosing a specific lock implementation based on `hint`. The nesting count for the nested lock is set to zero. See the OpenMP specification for the available hints.
`void omp_destroy_lock(omp_lock_t *lock)`	Changes the state of the lock to uninitialized.
`void omp_destroy_nest_lock(omp_nest_lock_t *lock)`	Changes the state of the nested lock to uninitialized.

Function	Description
`void omp_set_lock(omp_lock_t *lock)`	Forces the executing thread to wait until the lock is available. The thread is granted ownership of the lock when it becomes available.
`void omp_set_nest_lock(omp_nest_lock_t *lock)`	Forces the executing thread to wait until the nested lock is available. If the thread already owns the lock, then the lock nesting count is incremented.
`void omp_unset_lock(omp_lock_t *lock)`	Releases the executing thread from ownership of the lock. The behavior is undefined if the executing thread does not own the lock.
`void omp_unset_nest_lock(omp_nest_lock_t *lock)`	Decrements the nesting count for the nested lock and releases the executing thread from ownership of the nested lock if the resulting nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock.
`int omp_test_lock(omp_lock_t *lock)`	Attempts to set the lock. If successful, returns `TRUE`, otherwise returns `FALSE`.
`int omp_test_nest_lock(omp_nest_lock_t *lock)`	Attempts to set the nested lock. If successful, returns the nesting count, otherwise returns zero.

Timing Routines

Function	Description
`double omp_get_wtime(void)`	Returns a double precision value equal to the elapsed wall clock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.
`double omp_get_wtick(void)`	Returns a double precision value equal to the number of seconds between successive clock ticks.

Event Routines

Function	Description
`void omp_fulfill_event(omp_event_handle_t event)`	Fulfills the event associated with the event handle `event` and destroys the event.

Memory Management Routines

Function	Description
`omp_allocator_handle_t omp_init_allocator(omp_memspace_handle_t memspace, int ntraits, const omp_alloctrait_t traits[])`	Creates a new allocator that is associated with the `memspace` memory space and returns a handle to it.

Function	Description
void omp_destroy_allocator(omp_allocator_handle_t allocator)	Releases all resources used to implement the allocator handle.
void omp_set_default_allocator(omp_allocator_handle_t allocator)	Sets the default memory allocator to be used by allocation calls, allocate directives and allocate clauses that do not specify an allocator.
omp_allocator_handle_t omp_get_default_allocator(void)	Returns a handle to the memory allocator to be used by allocation calls, allocate directives and allocate clauses that do not specify an allocator.
void *omp_alloc(size_t size, omp_allocator_handle_t allocator)	Requests a memory allocation of size bytes from the specified memory allocator.
void *omp_aligned_alloc(size_t alignment, size_t size, omp_allocator_handle_t allocator)	Requests a memory allocation of `size` bytes from the specified memory allocator. Memory allocated by `omp_aligned_alloc` will be byte-aligned to at least the maximum of the alignment required by malloc, the alignment trait of the allocator and the alignment argument value.
void omp_free(void *ptr, omp_allocator_handle_t allocator)	Deallocates the memory to which `ptr` points. The `ptr` argument must have been returned by an OpenMP allocation routine.
void *omp_calloc(size_t nmemb, size_t size, omp_allocator_handle_t allocator)	Requests a memory allocation from the specified memory allocator for an array of `nmemb` elements each of which has a size of `size` bytes.
void *omp_aligned_calloc(size_t alignment, size_t nmemb, size_t size, omp_allocator_handle_t allocator)	Requests a memory allocation from the specified memory allocator for an array of `nmemb` elements each of which has a size of `size` bytes. Memory allocated by `omp_aligned_calloc` will be byte-aligned to at least the maximum of the alignment required by malloc, the alignment trait of the allocator and the alignment argument value.
void *omp_realloc(void *ptr, size_t size, omp_allocator_handle_t allocator, omp_allocator_handle_t free_allocator)	Deallocates the memory to which `ptr` points and requests a new memory allocation of `size` bytes from the specified memory allocator. Upon success it returns a pointer to the allocated memory and the contents of the new object shall be the same as that of the old object prior to deallocation up to the minimum size of old allocated size and `size` argument.

Tool Control Routines

Function	Description
int omp_control_tool(int command, int modifier, void *arg)	Enables a program to pass commands to an active tool.

Environment Display Routines

Function	Description
`void omp_display_env(int verbose)`	Displays the OpenMP version number and the initial values of ICVs associated with the environment variables.

See Also

Intel Extension Routines to OpenMP*

Intel® Compiler Extension Routines to OpenMP*

The Intel® compiler implements the following group of routines as extensions to the OpenMP* run-time library:

- Get and set the execution environment
- Get and set the stack size for parallel threads
- Memory allocation
- Get and set the thread sleep time for the throughput execution mode
- Target memory allocation

The Intel® extension routines described in this section can be used for low-level tuning to verify that the library code and application are functioning as intended. These routines are generally not recognized by other OpenMP-compliant compilers, which may cause the link stage to fail in the other compiler. To execute these OpenMP routines, use the `/Qopenmp-stubs` (Windows*) or `-qopenmp-stubs` (Linux* and macOS) option.

In most cases, environment variables can be used in place of the extension library routines. For example, the stack size of the parallel threads may be set using the `OMP_STACKSIZE` environment variable rather than the `kmp_set_stacksize_s()` library routine.

NOTE
A run-time call to an Intel extension routine takes precedence over the corresponding environment variable setting.

Execution Environment

Function	Description
`void kmp_set_defaults(char const *)`	Sets OpenMP environment variables defined as a list of variables separated by "**|**" in the argument.
`void kmp_set_library_throughput(void)`	Sets execution mode to throughput, which is the default. Allows the application to determine the runtime environment. Use in multi-user environments.
`void kmp_set_library_turnaround(void)`	Sets execution mode to turnaround. Use in dedicated parallel (single user) environments.
`void kmp_set_library_serial(void)`	Sets execution mode to serial.
`void kmp_set_library(int)`	Sets execution mode indicated by the value passed to the function. Valid values are: • **1** - serial mode • **2** - turnaround mode

Function	Description
	• **3** - throughput mode Call this routine before the first parallel region is executed.
`int kmp_get_library(void)`	Returns a value corresponding to the current execution mode: • **1** - serial • **2** - turnaround • **3** - throughput

Stack Size

Function	Description
`size_t kmp_get_stacksize_s(void)`	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed with `kmp_set_stacksize_s()` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable.
`int kmp_get_stacksize(void)`	Provided for backwards compatibility only. Use `kmp_get_stacksize_s()` routine for compatibility across different families of Intel processors.
`void kmp_set_stacksize_s(size_t`*`size`*`)`	Sets to *size* the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the `KMP_STACKSIZE` environment variable. In order for `kmp_set_stacksize_s()` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.
`void kmp_set_stacksize(int `*`size`*`)`	Provided for backward compatibility only. Use `kmp_set_stacksize_s()` for compatibility across different families of Intel® processors.

Memory Allocation

The Intel® compiler implements a group of memory allocation routines as an extension to the OpenMP run-time library to enable threads to allocate memory from a heap local to each thread. These routines are: `kmp_malloc()`, `kmp_calloc()`, and `kmp_realloc()`.

The memory allocated by these routines must also be freed by the `kmp_free()` routine. While you can allocate memory in one thread and then free that memory in a different thread, this mode of operation incurs a slight performance penalty.

Function	Description
`void* kmp_malloc(size_t `*`size`*`)`	Allocate memory block of *size* bytes from thread-local heap.

Function	Description
`void* kmp_calloc(size_t nelem, size_t elsize)`	Allocate array of *nelem* elements of size *elsize* from thread-local heap.
`void* kmp_realloc(void* ptr, size_t size)`	Reallocate memory block at address *ptr* and *size* bytes from thread-local heap.
`void* kmp_free(void* ptr)`	Free memory block at address *ptr* from thread-local heap. Memory must have been previously allocated with `kmp_malloc()`, `kmp_calloc()`, or `kmp_realloc()`.

Thread Sleep Time

In the throughput OpenMP* Support Libraries, threads wait for new parallel work at the ends of parallel regions, and then sleep, after a specified period of time. This time interval can be set by the `KMP_BLOCKTIME` environment variable or by the `kmp_set_blocktime()` function.

Function	Description
`int kmp_get_blocktime(void)`	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping, as set either by the `KMP_BLOCKTIME` environment variable or by `kmp_set_blocktime()`.
`void kmp_set_blocktime(int msec)`	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping. This routine affects the block time setting for the calling thread and any OpenMP team threads formed by the calling thread. The routine does not affect the block time for any other threads.

See Also

openmp-stubs, Qopenmp-stubs compiler option
OpenMP* Run-time Library Routines
OpenMP* Support Libraries

OpenMP* Support Libraries

The Intel® C++ Compiler Classic provides support libraries for OpenMP*. There are several kinds of libraries:

- **Performance:** supports parallel OpenMP execution.
- **Stubs:** supports serial execution of OpenMP applications.

Each kind of library is available for both dynamic and static linking on Linux* and macOS operating systems. Only dynamic linking is supported on Windows* operating systems.

Performance Libraries

To use these libraries, specify the `/Qopenmp` (Windows*) or `-qopenmp` (Linux* and macOS) option.

Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Operating System	Dynamic Link	Static Link
Linux	`libiomp5.so`	`libiomp5.a`
macOS	`libiomp5.dylib`	`libiomp5.a`
Windows	`libiomp5md.lib` `libiomp5md.dll`	None

Many routines in the OpenMP support libraries are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Stubs Libraries

To use these libraries, specify `/Qopenmp-stubs` (Windows*) or `-qopenmp-stubs` (Linux* and macOS) option. These allow you to compile OpenMP applications in serial mode and provide stubs for OpenMP routines and extended Intel-specific routines.

Operating System	Dynamic Link	Static Link
Linux	`libiompstubs5.so`	`libiompstubs5.a`
macOS	`libiompstubs5.dylib`	`libiompstubs5.a`
Windows	`libiompstubs5md.lib` `libiompstubs5md.dll`	None

Execution Modes

The compiler enables you to run an application under different execution modes specified at run time; the libraries support the turnaround, throughput, and serial modes. Use the `KMP_LIBRARY` environment variable to select the modes at run time.

Mode	Description
throughput (default)	The throughput mode allows the program to yield to other running programs and adjust resource usage to produce efficient execution in a dynamic environment. In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads yield to other threads while waiting for more parallel work. After completing the execution of a parallel region, threads wait for new parallel work to become available. After a certain period of time has elapsed, they stop waiting and sleep. Until more parallel work becomes available, sleeping allows processor and resources to be used for other work by non-OpenMP threaded code that may execute between parallel regions, or by other applications.

Mode	Description
turnaround	The amount of time to wait before sleeping is set either by the `KMP_BLOCKTIME` environment variable or by the `kmp_set_blocktime()` function. A small blocktime value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions. A larger blocktime value may be more appropriate if threads are to be reserved solely for use for OpenMP execution, but may penalize other concurrently-running OpenMP or threaded applications. The turnaround mode is designed to keep active all processors involved in the parallel computation, which minimizes execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads (although they are still subject to `KMP_BLOCKTIME` control). In a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively use all processors all of the time. **NOTE** Avoid over-allocating system resources. The condition can occur if either too many threads have been specified, or if too few processors are available at run time. If system resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.
serial	The serial mode forces parallel applications to run as a single thread.

See Also

openmp, Qopenmp compiler option
openmp-stubs, Qopenmp-stubs compiler option

Use the OpenMP Libraries

This section describes the steps needed to set up and use the OpenMP Libraries from the command line. On Windows systems, you can also build applications compiled with the OpenMP libraries in the Microsoft Visual Studio development environment.

For a list of the options and libraries used by the OpenMP libraries, see OpenMP Support Libraries.

Set Up Environment

Set up your environment for access to the compiler to ensure that the appropriate OpenMP library is available during linking.

Linux and macOS

On Linux and macOS systems you can source the appropriate script file (`compilervars` file).

Windows

On Windows systems you can either execute the appropriate batch (`.bat`) file or use the command-line window supplied in the compiler program folder that already has the environment set up.

During compilation, ensure that the version of `omp.h` used when compiling is the version provided by that compiler. For example, use the `omp.h` provided with GCC when you compile with GCC.

Caution
Be aware that when using the GCC or Microsoft Compiler, you may inadvertently use inappropriate header or module files. To avoid this, copy the header or module file(s) to a separate directory and put it in the appropriate `include` path using the `-I` option.

If a program uses data structures or classes that contain members with data types defined in the `omp.h` file, then source files that use those data structures should all be compiled with the same `omp.h` file.

Linux Examples

This section shows several examples of using OpenMP with the Intel C++ Compiler Classic from the command line on Linux.

Compile and Link OpenMP Libraries

You can compile an application and link the Intel OpenMP libraries with a single command using the `-qopenmp` option. For example:

```
icpc -qopenmp hello.cpp
```

By default, the Intel C++ Compiler Classic performs a dynamic link of the OpenMP libraries. To perform a static link (not recommended), add the option `-qopenmp-link=static`. The option `-qopenmp-link` controls whether the linker uses static or dynamic OpenMP libraries on Linux and macOS systems (default is `-qopenmp-link=dynamic`). See OpenMP Support Libraries for more information about dynamic and static OpenMP libraries.

Link OpenMP Object Files Compiled with GCC or Intel C++ Compiler Classic

You can use the `icc`/`icpc` compilers with the `gcc`/`g++` compilers to compile parts of an application and create object files that can then be linked (object-level interoperability).

When using `gcc` or the `g++` compiler to link the application with the Intel C++ Compiler Classic OpenMP compatibility library, you need to specify the following:

- The Intel OpenMP library name using the `-l` option
- The Linux `pthread` library using the `-l` option
- The path to the Intel libraries where the Intel C++ Compiler Classic is installed using the `-L` option

For example:

1. Compile `foo.c` and `bar.c` with `gcc`, using the `-fopenmp` option to enable OpenMP support:

```
gcc -fopenmp -c foo.c bar.c
```

   The `-c` prevents linking at this step.
2. Use the `gcc` compiler to link the application object code with the Intel OpenMP library:

```
gcc foo.o bar.o -liomp5 -lpthread -L<install_dir>/lib
```

   where <install_dir> is the location of the installed Intel OpenMP library.

Alternately, you can use the Intel C++ Compiler Classic to link the application so that you don't need to specify the `gcc-l` option, `-L` option, and the `-lpthread` options.

For example:

1. Compile `foo.c` with `gcc`, using the `gcc -fopenmp` option to enable OpenMP:

```
gcc -fopenmp -c foo.c
```

2. Compile `bar.c` with icc, using the `-qopenmp` option to enable OpenMP:

```
icc -qopenmp -c bar.c
```

3. Use the icc compiler to link the resulting application object code with the Intel OpenMP library:

```
icc -qopenmp foo.o bar.o
```

Link Mixed C/C++ and Fortran Object Files

You can mix C/C++ and Fortran object files and link the Intel OpenMP libraries using GNU, GCC, or Intel C++ Compiler Classic compilers.

This example shows mixed C and Fortran sources, linked using the Intel C++ Compiler Classic. Consider the mixed source files `ibar.c`, `gbar.c`, and `foo.f`, where the main program is contained in ibar.c:

1. Compile `ibar.c` using the icc compiler:

```
icc -qopenmp -c ibar.c
```

2. Compile `gbar.c` using the `gcc` compiler:

```
gcc -fopenmp -c gbar.c
```

3. Compile `foo.f` using the `ifort` compiler:

```
ifort -qopenmp -c foo.f
```

4. Use the icc compiler to link the resulting object files:

```
icc -qopenmp foo.o ibar.o gbar.o
```

If the main program were contained in the Fortran file foo.f, the linking step must be performed by the ifort compiler.

> **NOTE**
> Do not mix objects created by the Intel Fortran Compiler Classic and Intel Fortran Compiler with the GNU Fortran Compiler (`gfortran`); instead, recompile all Fortran sources with either `ifort` or `ifx`, or recompile all Fortran sources with the GNU Fortran Compiler . The GNU Fortran Compiler is only available on Linux operating systems.

When using the GNU `gfortran` Compiler to link the application with the Intel C++ Compiler Classic OpenMP compatibility library, you need to specify the following:

- The Intel® OpenMP compatibility library name and the Intel®`irc` libraries using the `-l` option
- The Linux `pthread` library using the `-l` option
- The path to the Intel® libraries where the Intel C++ Compiler Classic Classic is installed using the `-L` option

You do not need to specify the `-fopenmp` option on the link line.

For example, consider the mixed source files `ibar.c`, `gbar.c`, and `foo.f`:

1. Compile `ibar.c` using the icc compiler:

```
icc -qopenmp -c ibar.c
```

2. Compile `gbar.c` using the GCC compiler:

```
gcc -fopenmp -c gbar.c
```

3. Compile `foo.f` using the `gfortran` compiler:

```
gfortran -fopenmp -c foo.f
```

4. Use the `gfortran` compiler to link the application object code with the Intel OpenMP library. You do not need to specify the `-fopenmp` option in the link command:

```
gfortran foo.o ibar.o gbar.o -lirc -liomp5 -lpthread -lc -L<install_dir>/lib
```

where <install_dir> is the location of the installed Intel OpenMP library.

Alternately, you can use the Intel C++ Compiler Classic . to link the application object code but need to pass multiple `gfortran` libraries using the `-l` options at the link step.

This example shows mixed C and GNU Fortran sources linked using the icc compiler. Consider the mixed source files `ibar.c` and `foo.f`:

1. Compile the C source with the icc compiler:

```
icc -qopenmp -c ibar.c
```

2. Compile the GNU Fortran source with `gfortran`:

```
gfortran -fopenmp -c foo.f
```

3. Use icc to link the resulting object files with the `-l` option to pass the needed `gfortran` libraries:

```
icc -qopenmp foo.o ibar.o -lgfortran
```

macOS Examples

This section shows several examples of using OpenMP with the Intel C++ Compiler Classic from the command line on macOS.

Compile and Link OpenMP Libraries

You can compile an application and link the Intel OpenMP libraries with a single command using the `-qopenmp` option. For example:

```
icpc -qopenmp hello.cpp
```

By default, the the Intel C++ Compiler Classic performs a dynamic link of the OpenMP libraries. To perform a static link (not recommended), add the option `-qopenmp-link=static`. The option `-qopenmp-link` controls whether the linker uses static or dynamic OpenMP libraries on Linux and macOS systems (default is `-qopenmp-link=dynamic`).

Link Mixed OpenMP Object Files

You can mix OpenMP object files compiled by the Intel C++ Compiler Classic and Intel Fortran Compiler Classic.

This example shows mixed C and Fortran sources, compiled by the `icc` and `ifort` drivers, linked using the the Intel C++ Compiler Classic . Consider the mixed source files `ibar.c` and `foo.f90`:

1. Compile `ibar.c` using `icc`:

```
icc -qopenmp -c ibar.c
```

   The -c prevents linking at this step.

2. Compile `foo.f90` using `ifort`:

```
ifort -qopenmp -c foo.f90
```

3. Use `icc` to link the resulting object files:

```
icc -qopenmp foo.o ibar.o
```

Windows Examples

This section shows several examples of using OpenMP with the Intel C++ Compiler Classic from the command line on Windows.

Compile and Link OpenMP Libraries

You can compile an application and link the Compatibility libraries with a single command using the `/Qopenmp` option. By default, the Intel C++ Compiler Classic performs a dynamic link of the OpenMP libraries.

For example, to compile source file `hello.cpp` and link Compatibility libraries using the Intel C++ Compiler Classic:

```
icl /MD /Qopenmp hello.cpp
```

When using the Microsoft Visual C++ Compiler, you should link with the Intel® OpenMP compatibility library. You need to avoid linking the Microsoft OpenMP runtime library (`vcomp`) and explicitly pass the name of the Intel® OpenMP compatibility library as linker options using the `/link` option. For example:

```
cl /MD /openmp hello.cpp /link /nodefaultlib:vcomp libiomp5md.lib
```

Mix OpenMP Object Files Compiled with Visual C++ Compiler or Intel C++ Compiler Classic

You can use the Intel C++ Compiler Classic with the Visual C++ Compiler to compile parts of an application and create object files that can then be linked (object-level interoperability).

For example:

1. Compile `f1.c` and `f2.c` with the Visual C++ Compiler, using the `/openmp` option to enable OpenMP support:

```
cl /MD /openmp /c f1.c f2.c
```

   The `/c` prevents linking at this step.
2. Compile `f3.c` and `f4.c` with the icl compiler, using the `/Qopenmp` option to enable OpenMP support:

```
icl /MD /Qopenmp /c f3.c f4.c
```

3. Use the icl compiler to link the resulting application object code with the Intel C++ Compiler OpenMP library:

```
icl /MD /Qopenmp f1.obj f2.obj f3.obj f4.obj /Feapp /link /nodefaultlib:vcomp
```

   The `/Fe` specifies the generated executable file name.

   Alternatively, use the Visual C++ linker to link the application object code with the Compatibility library `libiomp5md.lib`:

```
link f1.obj f2.obj f3.obj f4.obj /out:app.exe /nodefaultlib:vcomp libiomp5md.lib
```

Use Intel OpenMP Libraries from Visual Studio

When running Windows, you can make certain changes in the Visual C++ Visual Studio development environment to use the Intel C++ Compiler Classic and Visual C++ to create applications that use the Intel OpenMP libraries.

Set the project **Property Pages** to indicate the Intel OpenMP runtime library location:

1. Open the project's property pages in from the main menu: **Project** > **Properties** (or right-click the Project name and select **Properties**) .
2. Select **Configuration Properties** > **Linker** > **General** > **Additional Library Directories**.
3. Enter the path to the Intel®-provided compiler libraries. For example, for an IA-32 architecture system (C/C++ only), enter:

```
<Intel_compiler_installation_path>\windows\compiler\lib\ia32 win
```

Make the Intel OpenMP dynamic runtime library accessible at runtime; you must specify the corresponding path:

1. Open the project's property pages in from the main menu: **Project** > **Properties** (or right-click the Project name and select **Properties**).
2. Select **Configuration Properties** > **Debugging** > **Environment**.

3. Enter the path to the Intel®-provided compiler libraries. For example, for an IA-32 architecture system (C/C++ only), enter:

```
PATH=%PATH%;<Intel_compiler_installation_path>\windows\redist\ia32_win\compiler
```

Add the Intel OpenMP runtime library name to the linker options and exclude the default Microsoft OpenMP runtime library:

1. Open the project's property pages in from the main menu: **Project** > **Properties** (or right-click the Project name and select **Properties**).
2. Select **Configuration Properties** > **Linker** > **Command Line** > **Additional Options**.
3. Enter the OpenMP library name and the Visual C++ linker option, `/nodefaultlib:vcomp libiomp5md.lib`.

See Also

qopenmp, Qopenmp compiler option
Using IPO
OpenMP Support Libraries
qopenmp-link, Qopenmp-link compiler option

Thread Affinity Interface

The Intel® runtime library has the ability to bind OpenMP* threads to physical processing units. The interface is controlled using the `KMP_AFFINITY` environment variable. Depending on the system (machine) topology, application, and operating system, thread affinity can have a dramatic effect on the application speed.

Thread affinity restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor computer. Depending upon the topology of the machine, thread affinity can have a dramatic effect on the execution speed of a program.

Thread affinity is supported on Windows* systems and versions of Linux* systems that have kernel support for thread affinity, but is not supported by macOS.

The Intel OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. There are three types of interfaces you can use to specify this binding, which are collectively referred to as the Intel OpenMP Thread Affinity Interface:

- The high-level affinity interface uses an environment variable to determine the machine topology and assigns OpenMP threads to the processors based upon their physical location in the machine. This interface is controlled entirely by the `KMP_AFFINITY` environment variable.
- The mid-level affinity interface uses an environment variable to explicitly specifies which processors (labeled with integer IDs) are bound to OpenMP threads. This interface provides compatibility with the GCC* `GOMP_AFFINITY` environment variable, but you can also invoke it by using the `KMP_AFFINITY` environment variable. The `GOMP_AFFINITY` environment variable is supported on Linux systems only, but users on Windows or Linux systems can use the similar functionality provided by the `KMP_AFFINITY` environment variable.
- The low-level affinity interface uses APIs to enable OpenMP threads to make calls into the OpenMP runtime library to explicitly specify the set of processors on which they are to be run. This interface is similar in nature to `sched_setaffinity` and related functions on Linux systems or to `SetThreadAffinityMask` and related functions on Windows systems. In addition, you can specify certain options of the `KMP_AFFINITY` environment variable to affect the behavior of the low-level API interface. For example, you can set the affinity type `KMP_AFFINITY` to disabled, which disables the low-level affinity interface, or you could use the `KMP_AFFINITY` or `GOMP_AFFINITY` environment variables to set the initial affinity mask, and then retrieve the mask with the low-level API interface.

The following terms are used in this section:

- The total number of processing elements on the machine is referred to as the number of *OS thread contexts.*
- Each processing element is referred to as an Operating System processor, or *OS proc*.
- Each OS processor has a unique integer identifier associated with it, called an *OS proc ID*.

- The term *package* refers to a single or multi-core processor chip.
- The term *OpenMP Global Thread ID* (GTID) refers to an integer which uniquely identifies all threads known to the Intel OpenMP runtime library. The thread that first initializes the library is given GTID 0. In the normal case where all other threads are created by the library and when there is no nested parallelism, then *n-threads-var* - 1 new threads are created with GTIDs ranging from 1 to *ntheads-var* - 1, and each thread's GTID is equal to the OpenMP thread number returned by function `omp_get_thread_num()`. The high-level and mid-level interfaces rely heavily on this concept. Hence, their usefulness is limited in programs containing nested parallelism. The low-level interface does not make use of the concept of a GTID and can be used by programs containing arbitrarily many levels of parallelism.

Some environment variables are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

The KMP_AFFINITY Environment Variable

NOTE
You must set the `KMP_AFFINITY` environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls, as described in Low Level Affinity API, below.

The `KMP_AFFINITY` environment variable uses the following general syntax:

Syntax
KMP_AFFINITY=[<*modifier*>,...]<*type*>[,<*permute*>][,<*offset*>]

For example, to list a machine topology map, specify `KMP_AFFINITY=verbose,none` to use a *modifier* of `verbose` and a *type* of `none`.

The following table describes the supported specific arguments.

Argument	**Default**	**Description**

`modifier`	`noverbose`
	`respect`
	`granularity=core`
		Optional. String consisting of keyword and specifier.
		• `granularity=<specifier>` takes the following specifiers: fine, thread, core, tile, die, node, group, and socket
		• `norespect`
		• `noverbose`
		• `nowarnings`
		• `noreset`
		• `proclist={<proc-list>}`
		• `respect`
		• `verbose`
		• `warnings`
		• `reset`
		The syntax for `<proc-list>` is explained in mid-level affinity interface.

Argument	Default	Description
		NOTE On Windows with multiple processor groups, the norespect affinity modifier is assumed when the process affinity mask equals a single processor group (which is default on Windows). Otherwise, the respect affinity modifier is used.
`type`	`none`	Required string. Indicates the thread affinity to use. • `balanced` • `compact` • `disabled` • `explicit` • `none` • `scatter` • `logical` (deprecated; instead use `compact`, but omit any `permute value`) • `physical` (deprecated; instead use `scatter`, possibly with an `offset value`) The `logical` and `physical` types are deprecated but supported for backward compatibility.
`permute`	`0`	Optional. Positive integer value. Not valid with type values of `explicit`, `none`, or `disabled`.
`offset`	`0`	Optional. Positive integer value. Not valid with type values of `explicit`, `none`, or `disabled`.

Affinity Types

Type is the only required argument.

type = none (default)

Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology. Specify `KMP_AFFINITY=verbose,none` to list a machine topology map.

type = balanced

Places threads on separate cores until all cores have at least one thread, similar to the `scatter` type. However, when the runtime must use multiple hardware thread contexts on the same core, the `balanced` type ensures that the OpenMP thread numbers are close to each other, which `scatter` does not do. This affinity type is supported on the CPU only for single socket systems.

NOTE
The OpenMP* environment variable `OMP_PROC_BIND=spread` is similar to `KMP_AFFINITY=balanced` and is available on all platforms, including multi-socket CPU systems.

type = compact

Specifying `compact` assigns the OpenMP thread <*n*>+1 to a free thread context as close as possible to the thread context where the <*n*> OpenMP thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads.

type = disabled

Specifying `disabled` completely disables the thread affinity interfaces. This forces the OpenMP run-time library to behave as if the affinity interface was not supported by the operating system. This includes the low-level API interfaces such as `kmp_set_affinity` and `kmp_get_affinity`, which have no effect and will return a nonzero error code.

type = explicit

Specifying `explicit` assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the proclist= modifier, which is required for this affinity type. See Explicitly Specifying OS Proc IDs (GOMP_CPU_AFFINITY).

type = scatter

Specifying `scatter` distributes the threads as evenly as possible across the entire system. `scatter` is the opposite of `compact`; so the leaves of the node are most significant when sorting through the machine topology map.

Deprecated Types: logical and physical

Types `logical` and `physical` are deprecated and may become unsupported in a future release. Both are supported for backward compatibility.

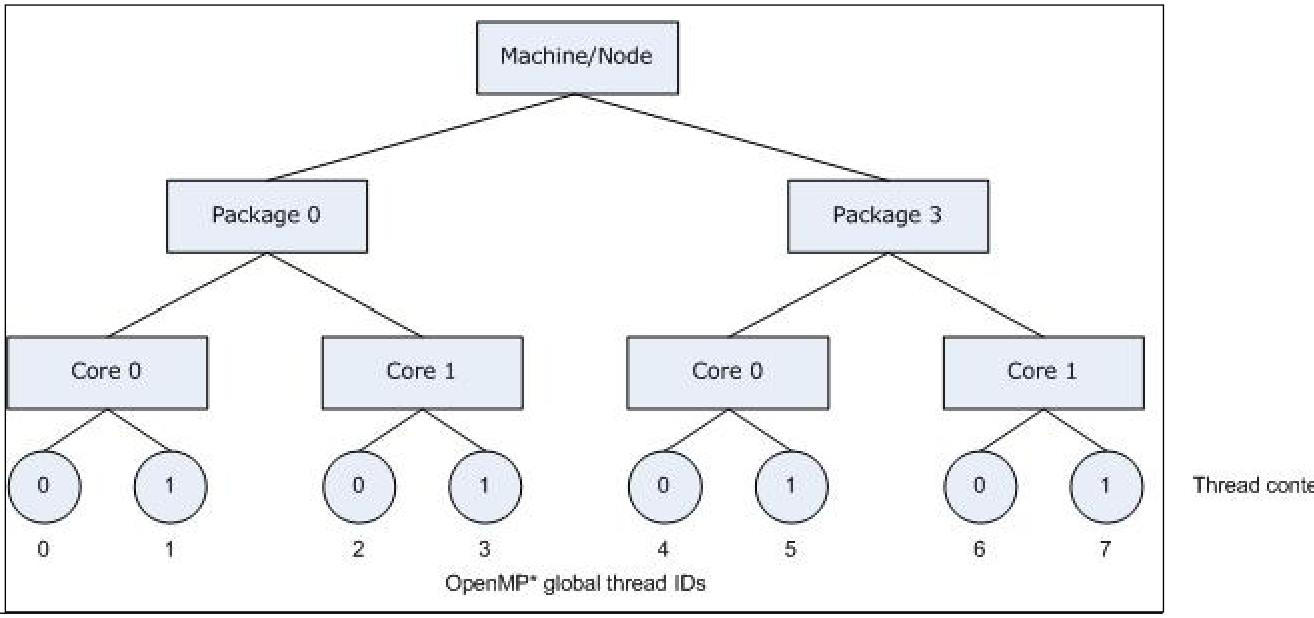
For `logical` and `physical` affinity types, a single trailing integer is interpreted as an `offset` specifier instead of a `permute` specifier. In contrast, with `compact` and `scatter` types, a single trailing integer is interpreted as a `permute` specifier.

- Specifying `logical` assigns OpenMP threads to consecutive logical processors, which are also called hardware thread contexts. The type is equivalent to `compact`, except that the `permute` specifier is not allowed. Thus, `KMP_AFFINITY=logical,n` is equivalent to `KMP_AFFINITY=compact,0,n` (this equivalence is true regardless of the whether or not a `granularity=fine` modifier is present).
- Specifying `physical` assigns threads to consecutive physical processors (cores). For systems where there is only a single thread context per core, the type is equivalent to `logical`. For systems where multiple thread contexts exist per core, `physical` is equivalent to `compact` with a `permute` specifier of 1; that is, `KMP_AFFINITY=physical,n` is equivalent to `KMP_AFFINITY=compact,1,n` (regardless of the whether or not a `granularity=fine` modifier is present). This equivalence means that when the compiler sorts the map it should permute the innermost level of the machine topology map to the outermost, presumably the thread context level. This type does not support the `permute` specifier.

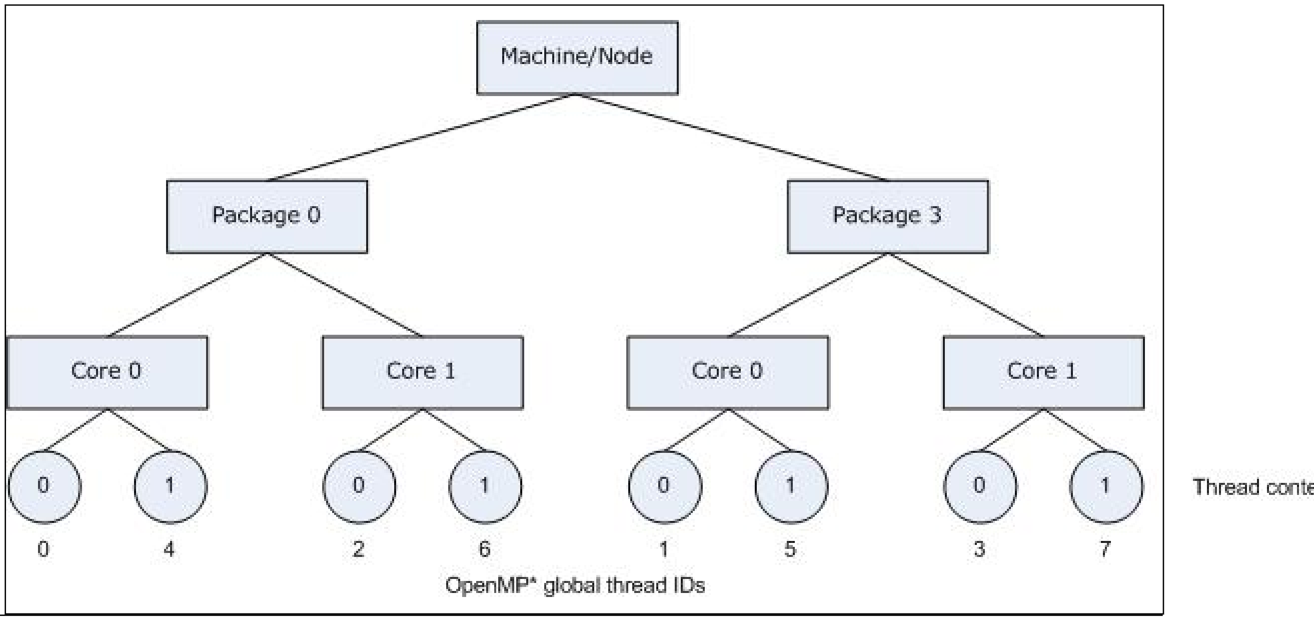
Examples of Types `compact` and `scatter`

The following figure illustrates the topology for a machine with two processors, and each processor has two cores; further, each core has Intel® Hyper-Threading Technology (Intel® HT Technology) enabled.

The following figure also illustrates the binding of OpenMP thread to hardware thread contexts when specifying `KMP_AFFINITY=granularity=fine,compact`.



Specifying `scatter` on the same system as shown in the figure above, the OpenMP threads would be assigned the thread contexts as shown in the following figure, which shows the result of specifying `KMP_AFFINITY=granularity=fine,scatter`.



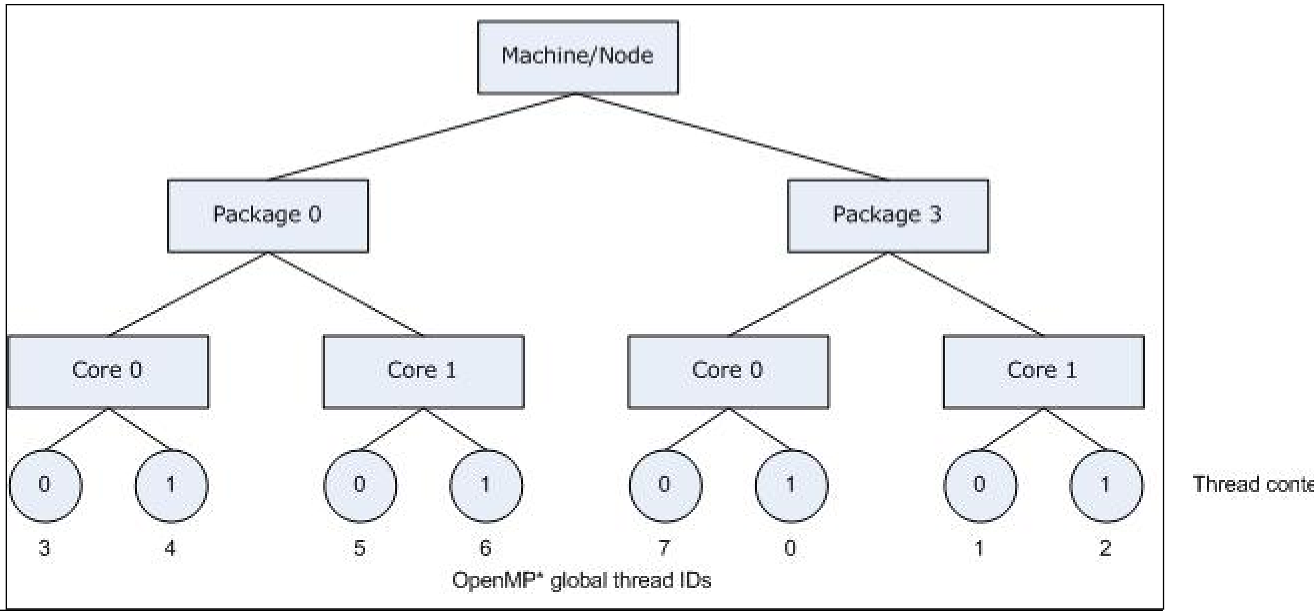
permute and offset combinations

For both `compact` and `scatter`, `permute` and `offset` are allowed; however, if you specify only one integer, the compiler interprets the value as a permute specifier. Both `permute` and `offset` default to 0.

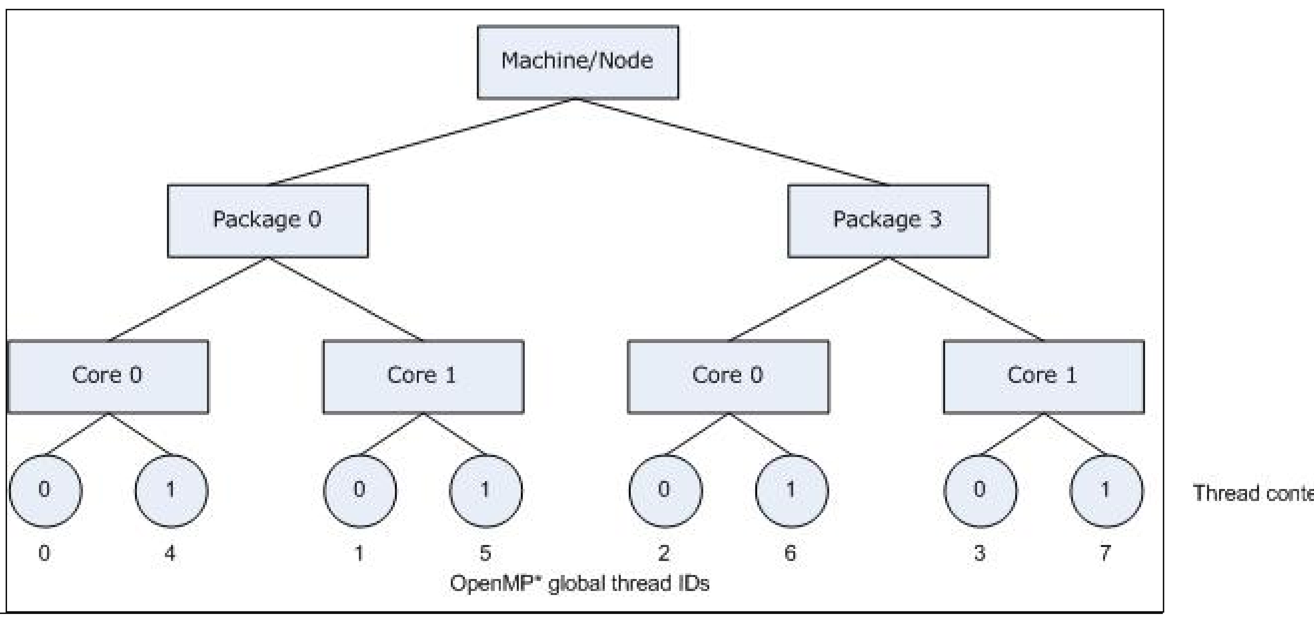
The `permute` specifier controls which levels are most significant when sorting the machine topology map. A value for permute forces the mappings to make the specified number of most significant levels of the sort the least significant, and it inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.

The `offset` specifier indicates the starting position for thread assignment.

The following figure illustrates the result of specifying `KMP_AFFINITY=granularity=fine,compact,0,5`.



Consider the hardware configuration from the previous example, running an OpenMP application which exhibits data sharing between consecutive iterations of loops. We would therefore like consecutive threads to be bound close together, as is done with `KMP_AFFINITY=compact`, so that communication overhead, cache line invalidation overhead, and page thrashing are minimized. Now, suppose the application also had a number of parallel regions which did not utilize all of the available OpenMP threads. It is desirable to avoid binding multiple threads to the same core and leaving other cores not utilized, since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core. Since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core, you might want to avoid binding multiple threads to the same core while leaving other cores unused. The following figure illustrates this strategy of using `KMP_AFFINITY=granularity=fine,compact,1,0` as a setting.



The OpenMP thread *n*+1 is bound to a thread context as close as possible to OpenMP thread *n*, but on a different core. Once each core has been assigned one OpenMP thread, the subsequent OpenMP threads are assigned to the available cores in the same order, but they are assigned on different thread contexts.

Modifier Values for Affinity Types

Modifiers are optional arguments that precede type. If you do not specify a modifier, the `noverbose`, `respect`, and `granularity=core` modifiers are used automatically.

Modifiers are interpreted in order from left to right, and they may conflict. Following conflicting modifier is ignored. For example, specifying `KMP_AFFINITY=verbose,noverbose,scatter` is therefore equivalent to setting `KMP_AFFINITY=verbose,scatter`.

modifier = noverbose (default)

Does not print verbose messages.

modifier = verbose

Prints messages concerning the supported affinity. The messages include information about the number of packages, number of cores in each package, number of thread contexts for each core, and OpenMP thread bindings to physical thread contexts.

Information about binding OpenMP threads to physical thread contexts is indirectly shown in the form of the mappings between hardware thread contexts and the operating system (OS) processor (proc) IDs. The affinity mask for each OpenMP thread is printed as a set of OS processor IDs.

For example, specifying `KMP_AFFINITY=verbose,scatter` on a dual core system with two processors, with Intel® Hyper-Threading Technology (Intel® HT Technology) disabled, results in a message listing similar to the following when then program is executed:

Verbose, scatter message

```
...
KMP_AFFINITY: Initial OS proc set respected: 0,1,2,3
KMP_AFFINITY: affinity capable, using hwloc.
KMP_AFFINITY: 4 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
KMP_AFFINITY: pid 79739 tid 79739 thread 0 bound to OS proc set 0
KMP_AFFINITY: pid 79739 tid 79740 thread 2 bound to OS proc set 2
KMP_AFFINITY: pid 79739 tid 79741 thread 3 bound to OS proc set 3
KMP_AFFINITY: pid 79739 tid 79742 thread 1 bound to OS proc set 1
```

The verbose modifier generates several standard, general messages. The following table summarizes how to read the messages.

Message String	Description
"affinity capable"	Indicates that all components (compiler, operating system, and hardware) support affinity, so thread binding is possible.
"decoding x2APIC ids"	Indicates that the machine topology was discovered by binding a thread to each operating system processor and decoding the output of the `cpuid` instruction.

Message String	Description
"using hwloc"	Indicates that the Portable Hardware Locality* (`hwloc`) library used to determine machine topology.
"using /proc/cpuinfo"	Linux only. Indicates that `cpuinfo` is being used to determine machine topology.
"using flat"	Operating system processor ID is assumed to be equivalent to physical package ID. This method of determining machine topology is used if none of the other methods will work, but may not accurately detect the actual machine topology.
"uniform topology"	The machine topology map is a full tree with no missing leaves at any level.

The mapping from the operating system processors to thread context ID is printed next. The binding of OpenMP thread context ID is printed next unless the affinity type is none. For more information, see Determining Machine Topology.

modifier = granularity

Binding OpenMP threads to particular packages and cores will often result in a performance gain on systems with Intel processors with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled; however, it is usually not beneficial to bind each OpenMP thread to a particular thread context on a specific core. Granularity describes the lowest levels that OpenMP threads are allowed to float within a topology map.

This modifier supports the following additional specifiers.

Specifier	Description
`core`	Default. Allows all the OpenMP threads bound to a core to float between the different thread contexts.
`fine` or `thread`	The finest granularity level. Causes each OpenMP thread to be bound to a single thread context. The two specifiers are functionally equivalent.
`tile`, or `die`, or `node`, or `group`, or `socket`	Allows all the OpenMP threads bound to a tile, or die, or NUMA node, or group, or socket to float between the different thread contexts of cores the tile, or die, or NUMA node, or group, or socket consists of.

Specifying `KMP_AFFINITY=verbose,granularity=core,compact` on the same dual core system with two processors as in the previous section, but with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, results in a message listing similar to the following when the program is executed:

Verbose, granularity=core,compact message

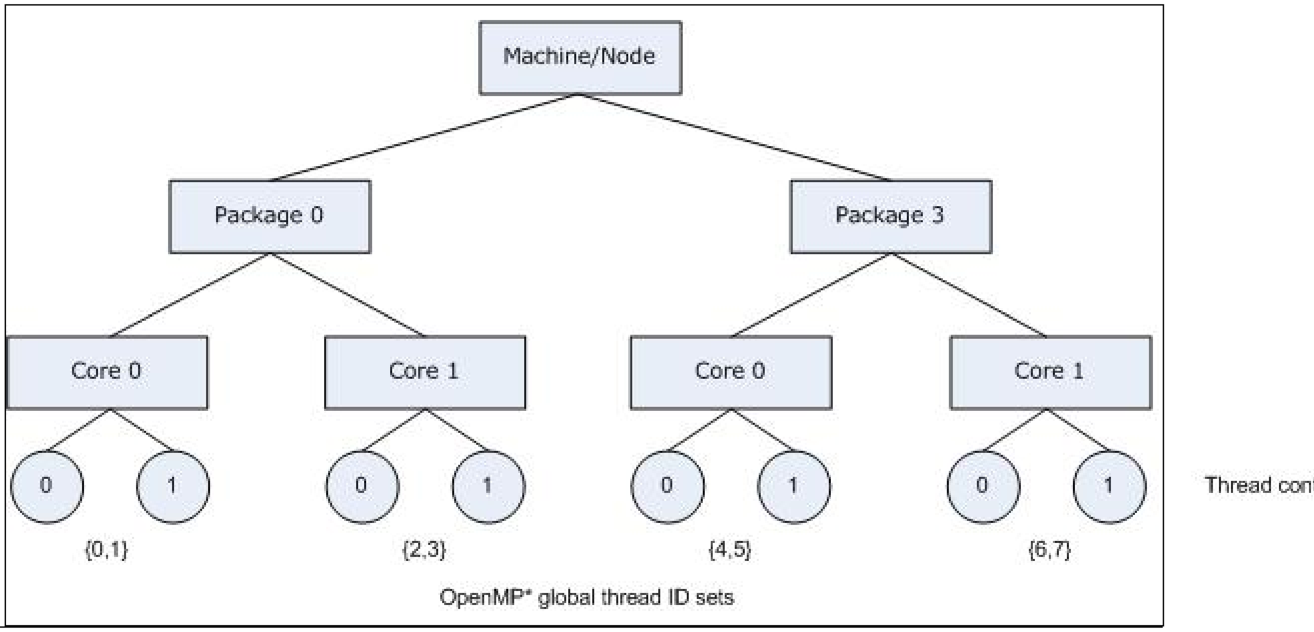
```
KMP_AFFINITY: Initial OS proc set respected: 0-7
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 8 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockects x 2 cores/socket x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to socket 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to socket 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to socket 3 core 0 thread 1
```

Verbose, granularity=core,compact message

```
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to socket 3 core 1 thread 1
KMP_AFFINITY: pid 40880 tid 40880 thread 0 bound to OS proc set 0,4
KMP_AFFINITY: pid 40880 tid 40881 thread 1 bound to OS proc set 0,4
KMP_AFFINITY: pid 40880 tid 40882 thread 2 bound to OS proc set 2,6
KMP_AFFINITY: pid 40880 tid 40883 thread 3 bound to OS proc set 2,6
KMP_AFFINITY: pid 40880 tid 40884 thread 4 bound to OS proc set 1,5
KMP_AFFINITY: pid 40880 tid 40885 thread 5 bound to OS proc set 1,5
KMP_AFFINITY: pid 40880 tid 40886 thread 6 bound to OS proc set 3,7
KMP_AFFINITY: pid 40880 tid 40887 thread 7 bound to OS proc set 3,7
```

The affinity mask for each OpenMP thread is shown in the listing (above) as the set of operating system processor to which the OpenMP thread is bound.

The following figure illustrates the machine topology map, for the above listing, with OpenMP thread bindings.



In contrast, specifying `KMP_AFFINITY=verbose,granularity=fine,compact` or `KMP_AFFINITY=verbose,granularity=thread,compact` binds each OpenMP thread to a single hardware thread context when the program is executed:

Verbose, granularity=fine,compact message

```
KMP_AFFINITY: Initial OS proc set respected: 0-7
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 8 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to socket 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to socket 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
```

Verbose, granularity=fine,compact message

```
KMP_AFFINITY: OS proc 5 maps to socket 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to socket 3 core 1 thread 1
KMP_AFFINITY: pid 40895 tid 40895 thread 0 bound to OS proc set 0
KMP_AFFINITY: pid 40895 tid 40896 thread 1 bound to OS proc set 4
KMP_AFFINITY: pid 40895 tid 40897 thread 2 bound to OS proc set 2
KMP_AFFINITY: pid 40895 tid 40898 thread 3 bound to OS proc set 6
KMP_AFFINITY: pid 40895 tid 40899 thread 4 bound to OS proc set 1
KMP_AFFINITY: pid 40895 tid 40900 thread 5 bound to OS proc set 5
KMP_AFFINITY: pid 40895 tid 40901 thread 6 bound to OS proc set 3
KMP_AFFINITY: pid 40895 tid 40902 thread 7 bound to OS proc set 7
```

The OpenMP to hardware context binding for this example was illustrated in the first example.

Specifying `granularity=fine` will always cause each OpenMP thread to be bound to a single OS processor. This is equivalent to `granularity=thread`, currently the finest granularity level.

modifier = respect (default)

Respect the process' original affinity mask, or more specifically, the affinity mask in place for the thread that initializes the OpenMP run-time library. The behavior differs between Linux and Windows:

- On Windows: Respect original affinity mask for the process.
- On Linux: Respect the affinity mask for the thread that initializes the OpenMP run-time library.

> **NOTE** On Windows with multiple processor groups, the `norespect` affinity modifier is the default when the process affinity mask equals a single processor group (which is default on Windows). Otherwise, the respect affinity modifier is the default.

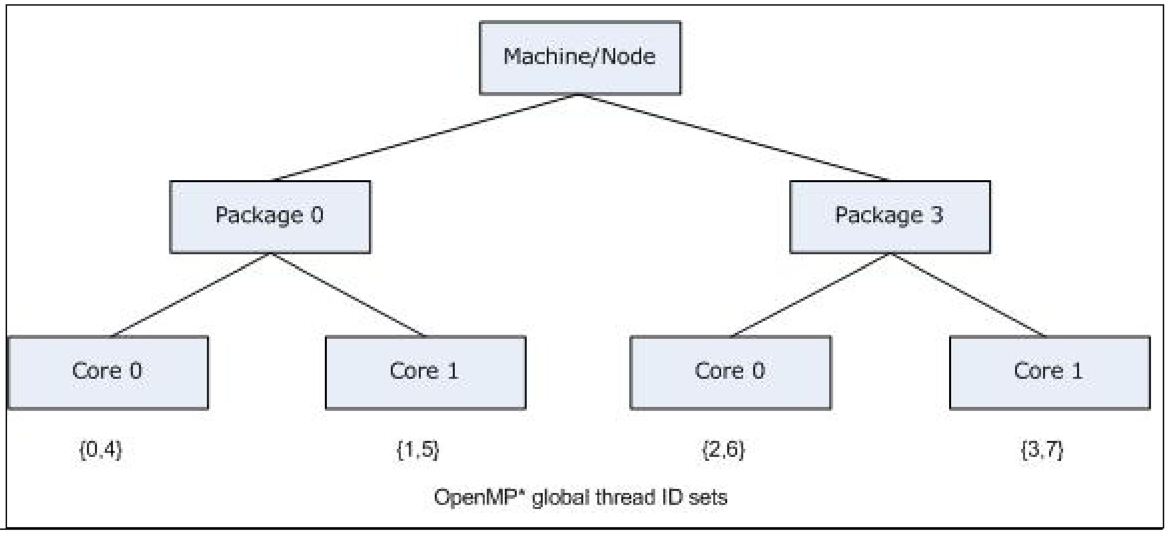
Specifying `KMP_AFFINITY=verbose,compact` for the same system used in the previous example, with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, and invoking the library with an initial affinity mask of {4,5,6,7} (thread context 1 on every core) causes the compiler to model the machine as a dual core, two-processor system with Intel® HT Technology disabled.

Verbose,compact message

```
KMP_AFFINITY: Initial OS proc set respected: 4-7
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 4 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 4 maps to socket 0 core 0 thread 1
KMP_AFFINITY: OS proc 6 maps to socket 0 core 1 thread 1
KMP_AFFINITY: OS proc 5 maps to socket 3 core 0 thread 1
KMP_AFFINITY: OS proc 7 maps to socket 3 core 1 thread 1
KMP_AFFINITY: pid 41032 tid 41032 thread 0 bound to OS proc set 4
KMP_AFFINITY: pid 41032 tid 41033 thread 1 bound to OS proc set 6
KMP_AFFINITY: pid 41032 tid 41034 thread 2 bound to OS proc set 5
KMP_AFFINITY: pid 41032 tid 41035 thread 3 bound to OS proc set 7
```

Because there are four thread contexts accessible on the machine, by default the compiler created four threads for an OpenMP `parallel` construct.

The following figure illustrates the corresponding machine topology map and threads placement in case eight OpenMP threads requested via `OMP_NUM_THREADS=8`



When using the local `cpuid` information to determine the machine topology, it is not always possible to distinguish between a machine that does not support Intel® Hyper-Threading Technology (Intel® HT Technology) and a machine that supports it, but has it disabled. Therefore, the compiler does not include a level in the map if the elements (nodes) at that level had no siblings, with the exception that the package level is always modeled. As mentioned earlier, the package level will always appear in the topology map, even if there only a single package in the machine.

modifier = norespect

Do not respect original affinity mask for the process. Binds OpenMP threads to all operating system processors.

In early versions of the OpenMP run-time library that supported only the `physical` and `logical` affinity types, `norespect` was the default and was not recognized as a modifier.

The default was changed to respect when types `compact` and `scatter` were added; therefore, thread bindings may have changed with the newer compilers in situations where the application specified a partial initial thread affinity mask.

modifier = nowarnings

Do not print warning messages from the affinity interface.

modifier = warnings (default)

Print warning messages from the affinity interface (default).

modifier = noreset (default)

Do not reset the primary thread's affinity after each outermost parallel region is complete. This setting preserves the primary thread's OpenMP affinity setting between parallel regions. For example, if `KMP_AFFINITY=compact,granularity=core`, then the primary thread's affinity is set to the first core for the first parallel region and kept that way for the thread's lifetime, even during serial regions.

modifier = reset

Reset the primary thread's affinity after each outermost parallel region is complete. This setting will reset the primary thread's affinity back to the initial affinity before OpenMP was initialized after each outermost parallel region is complete.

Determining Machine Topology

On IA-32 and Intel® 64 architecture systems, if the package has an APIC (Advanced Programmable Interrupt Controller), the compiler will use the `cpuid` instruction to obtain the `package id`, `core id`, and `thread context id`. Under normal conditions, each thread context on the system is assigned a unique APIC ID at boot time. The compiler obtains other pieces of information obtained by using the *cpuid* instruction, which together with the number of OS thread contexts (total number of processing elements on the machine), determine how to break the APIC ID down into the `package ID`, `core ID`, and `thread context ID`.

There are several ways to specify the APIC ID in the *cpuid* instruction - the legacy method in leaf 4, and the more modern method in leaf 11 and leaf 31. Only 256 unique APIC IDs are available in leaf 4. Leaf 11 and leaf 31 have no such limitation.

Normally, all `core ids` on a package and all `thread context ids` on a core are contiguous; however, numbering assignment gaps are common for `package ids`, as shown in the figure above.

If the compiler cannot determine the machine topology using any other method, but the operating system supports affinity, a warning message is printed, and the topology is assumed to be `flat`. For example, a flat topology assumes the operating system process *N* maps to package *N*, and there exists only one thread context per core and only one core for each package.

If the machine topology cannot be accurately determined as described above, the user can manually copy `/proc/cpuinfo` to a temporary file, correct any errors, and specify the machine topology to the OpenMP runtime library via the environment variable `KMP_CPUINFO_FILE=<temp_filename>`, as described in the section `KMP_CPUINFO_FILE` and `/proc/cpuinfo`.

Regardless of the method used in determining the machine topology, if there is only one thread context per core for every core on the machine, the thread context level will not appear in the topology map. If there is only one core per package for every package in the machine, the core level will not appear in the machine topology map. The topology map need not be a full tree, because different packages may contain a different number of cores, and different cores may support a different number of thread contexts.

The package level will always appear in the topology map, even if there only a single package in the machine.

KMP_CPUINFO_FILE and /proc/cpuinfo

One of the methods the Intel® C++ Compiler Classic OpenMP runtime library can use to detect the machine topology on Linux systems is to parse the contents of `/proc/cpuinfo`. If the contents of this file (or a device mapped into the Linux file system) are insufficient or erroneous, you can consider copying its contents to a writable temporary file `<temp_file>`, correct it or extend it with the necessary information, and set `KMP_CPUINFO_FILE=<temp_file>`.

If you do this, the OpenMP runtime library will read the `<temp_file>` location pointed to by `KMP_CPUINFO_FILE` instead of the information contained in `/proc/cpuinfo` or attempting to detect the machine topology by decoding the APIC IDs. That is, the information contained in the <*temp_file*> overrides these other methods. You can use the KMP_CPUINFO_FILE interface on Windows systems, where `/proc/cpuinfo` does not exist.

The content of `/proc/cpuinfo` or `<temp_file>` should contain a list of entries for each processing element on the machine. Each processor element contains a list of entries (descriptive name and value on each line). A blank line separates the entries for each processor element. Only the following fields are used to determine the machine topology from each entry, either in <*temp_file*> or `/proc/cpuinfo`:

Field	Description
processor :	Specifies the OS ID for the processing element. The OS ID must be unique. The `processor` and `physical id` fields are the only ones that are required to use the interface.
physical id :	Specifies the package ID, which is a physical chip ID. Each package may contain multiple cores. The package level always exists in the compiler's OpenMP run-time library model of the machine topology.
core id :	Specifies the core ID. If it does not exist, it defaults to 0. If every package on the machine contains only a single core, the core level will not exist in the machine topology map (even if some of the core ID fields are non-zero).
apicid :	Specifies the thread ID. If it does not exist, it defaults to 0. If every core on the machine contains only a single thread, the thread level will not exist in the machine topology map (even if some thread ID fields are non-zero).
node_*n* id :	This is a extension to the normal contents of `/proc/cpuinfo` that can be used to specify the nodes at different levels of the memory interconnect on Non-Uniform Memory Access (NUMA) systems. Arbitrarily many levels *n* are supported. The node_0 level is closest to the package level; multiple packages comprise a node at level 0. Multiple nodes at level 0 comprise a node at level 1, and so on.

Each entry must be spelled exactly as shown, in lowercase, followed by optional whitespace, a colon (:), more optional whitespace, then the integer ID. Fields other than those listed are simply ignored.

NOTE
It is common for the `thread id` field to be missing from `/proc/cpuinfo` on many Linux variants, and for a field labeled `siblings` to specify the number of threads per node or number of nodes per package. However, the Intel OpenMP runtime library ignores fields labeled `siblings` so it can distinguish between the `thread id` and `siblings` fields. When this situation arises, the warning message `Physical node/pkg/core/thread ids not unique` appears (unless the `type specified is nowarnings`).

Windows Processor Groups

On a 64-bit Windows operating system, it is possible for multiple processor groups to accommodate more than 64 processors. Each group is limited in size, up to a maximum value of sixty-four (64) processors.

If multiple processor groups are detected, the default is to model the machine as a 2-level tree, where level 0 are for the processors in a group, and level 1 are for the different groups. Threads are assigned to a group until there are as many OpenMP threads bound to the groups as there are processors in the group. Subsequent threads are assigned to the next group, and so on.

By default, threads are allowed to float among all processors in a group, that is to say, granularity equals the group [granularity=group]. You can override this binding and explicitly use another affinity type like compact, scatter, and so on. If you do so, the granularity must be sufficiently fine to prevent a thread from being bound to multiple processors in different groups.

Using a Specific Machine Topology Modeling Method (KMP_TOPOLOGY_METHOD)

You can set the `KMP_TOPOLOGY_METHOD` environment variable to force OpenMP to use a particular machine topology modeling method.

Value	Description
`cpuid_leaf11`	Decodes the APIC identifiers as specified by leaf 11 of the *cpuid* instruction.
`cpuid_leaf4`	Decodes the APIC identifiers as specified in leaf 4 of the *cpuid* instruction.
`cpuinfo`	If `KMP_CPUINFO_FILE` is not specified, forces OpenMP to parse `/proc/cpuinfo` to determine the topology (Linux only). If `KMP_CPUINFO_FILE` is specified as described above, uses it (Windows or Linux).
`group`	Models the machine as a 2-level map, with level 0 specifying the different processors in a group, and level 1 specifying the different groups (Windows 64-bit only) .
`flat`	Models the machine as a flat (linear) list of processors.
`hwloc`	Models the machine as the Portable Hardware Locality* (hwloc) library does. This model is the most detailed and includes, but is not limited to: numa nodes, packages, cores, hardware threads, caches, and Windows processor groups.

Explicitly Specifying OS Processor IDs (GOMP_CPU_AFFINITY)

NOTE
You must set the `GOMP_CPU_AFFINITY` environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls, as described in Low Level Affinity API, below.

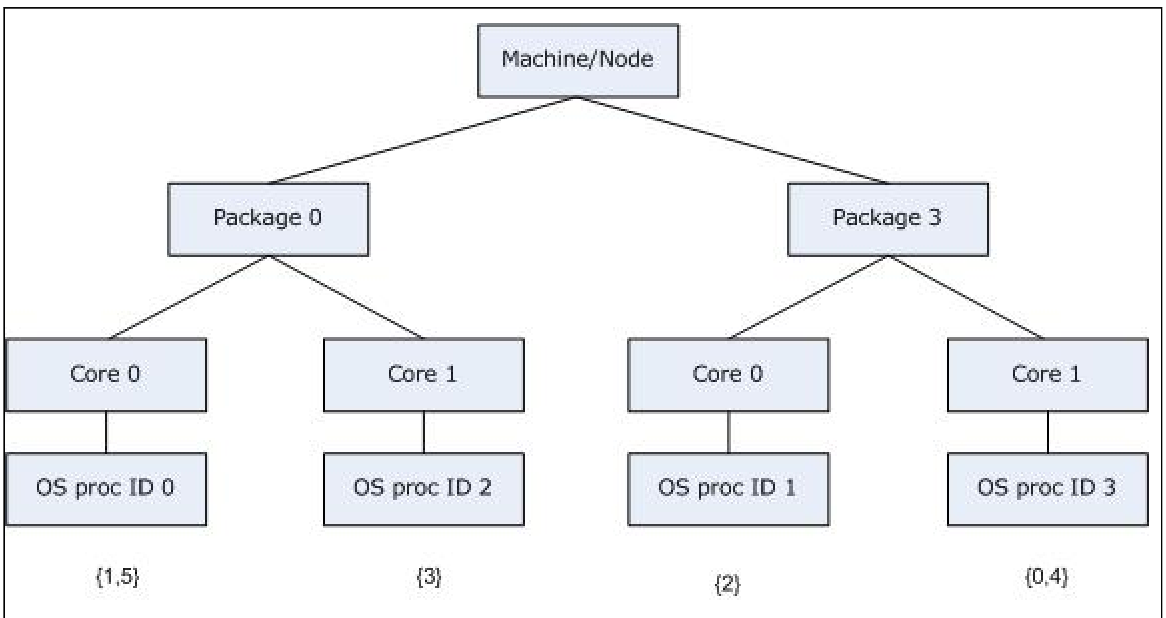
Instead of allowing the library to detect the hardware topology and automatically assign OpenMP threads to processing elements, the user may explicitly specify the assignment by using a list of operating system (OS) processor (proc) IDs. However, this requires knowledge of which processing elements the OS proc IDs represent.

On Linux systems, when using the Intel OpenMP compatibility libraries enabled by the compiler option `-qopenmp-lib compat`, you can use the `GOMP_AFFINITY` environment variable to specify a list of OS processor IDs. Its syntax is identical to that accepted by `libgomp` (assume that *<proc_list>* produces the entire `GOMP_AFFINITY` environment string):

Value	**Description**
<proc_list> :=	<entry> | *<elem>* , <list> | <elem> <whitespace> <list>
<elem> :=	*<proc_spec>* | *<range>*
<proc_spec> :=	*<proc_id>*
<range> :=	*<proc_id>* - *<proc_id>* | *<proc_id>* - *<proc_id>* : <int>
<proc_id> :=	<positive_int>

OS processors specified in this list are then assigned to OpenMP threads, in order of OpenMP Global Thread IDs. If more OpenMP threads are created than there are elements in the list, then the assignment occurs modulo the size of the list. That is, OpenMP Global Thread ID *n* is bound to list element *n* mod *<list_size>*.

Consider the machine previously mentioned: a dual core, dual-package machine without Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, where the OS proc IDs are assigned in the same manner as the example in a previous figure. Suppose that the application creates six OpenMP threads instead of 4 (the default), oversubscribing the machine. If `GOMP_AFFINITY=3,0-2`, then OpenMP threads are bound as shown in the figure below, just as should happen when compiling with `gcc` and linking with `libgomp`:



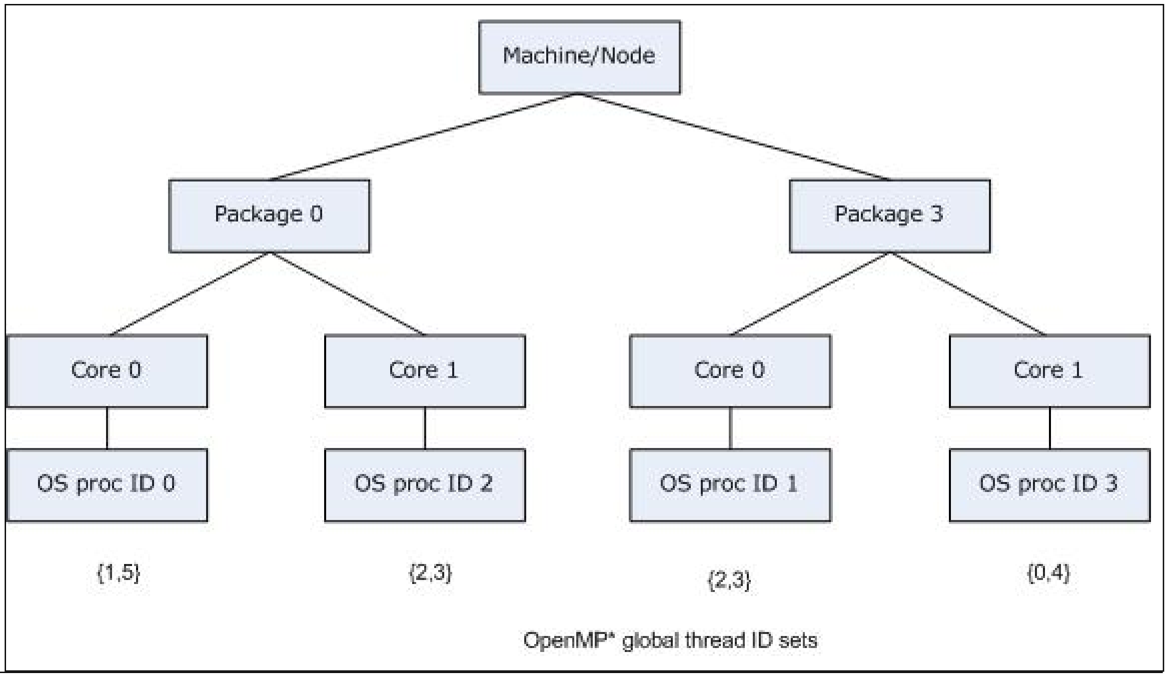
OpenMP* global thread ID sets

The same syntax can be used to specify the OS proc ID list in the proclist=[<proc_list>] modifier in the `KMP_AFFINITY` environment variable string. There is a slight difference: in order to have strictly the same semantics as in the `gcc` OpenMP runtime library `libgomp`: the `GOMP_AFFINITY` environment variable implies `granularity=fine`. If you specify the OS proc list in the `KMP_AFFINITY` environment variable without a `granularity=` specifier, then the default `granularity` is not changed. That is, OpenMP threads are allowed to float between the different thread contexts on a single core. Thus `GOMP_AFFINITY=<proc_list>` is an alias for `KMP_AFFINITY="granularity=fine,proclist=[<proc_list>],explicit"`.

In the `KMP_AFFINITY` environment variable string, the syntax is extended to handle operating system processor ID sets. The user may specify a set of operating system processor IDs among which an OpenMP thread may execute ("float") enclosed in brackets:

Value	Description
<proc_list> :=	<proc_id> | { <float_list> }
<float_list> :=	<proc_id> | <proc_id> , <float_list>

This allows functionality similarity to the `granularity= specifier`, but it is more flexible. The OS processors on which an OpenMP thread executes may exclude other OS processors nearby in the machine topology, but include other distant OS processors. Building upon the previous example, we may allow OpenMP threads 2 and 3 to "float" between OS processor 1 and OS processor 2 by using `KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"`, as shown in the figure below:



If `verbose` were also specified, the output when the application is executed would include:

KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"

```
KMP_AFFINITY: Initial OS proc set respected: 0,1,2,3
KMP_AFFINITY: decoding x2APIC ids.
KMP_AFFINITY: 4 available OS procs
KMP_AFFINITY: Uniform topology
KMP_AFFINITY: 2 sockets x 2 cores/socket x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map:
KMP_AFFINITY: OS proc 0 maps to socket 0 core 0 thread 0
KMP_AFFINITY: OS proc 2 maps to socket 0 core 1 thread 0
KMP_AFFINITY: OS proc 1 maps to socket 3 core 0 thread 0
KMP_AFFINITY: OS proc 3 maps to socket 3 core 1 thread 0
```

KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"

```
KMP_AFFINITY: pid 41464 tid 41464 thread 0 bound to OS proc set 3
KMP_AFFINITY: pid 41464 tid 41465 thread 1 bound to OS proc set 0
KMP_AFFINITY: pid 41464 tid 41466 thread 2 bound to OS proc set 1,2
KMP_AFFINITY: pid 41464 tid 41467 thread 3 bound to OS proc set 1,2
KMP_AFFINITY: pid 41464 tid 41468 thread 4 bound to OS proc set 3
KMP_AFFINITY: pid 41464 tid 41469 thread 5 bound to OS proc set 0
```

Low Level Affinity API

Instead of relying on the user to specify the OpenMP thread to OS proc binding by setting an environment variable before program execution starts (or by using the `kmp_settings` interface before the first parallel region is reached), each OpenMP thread can determine the desired set of OS procs on which it is to execute and bind to them with the `kmp_set_affinity` API call.

> **Caution**
> When you use this affinity interface you take complete control of the hardware resources on which your threads run. To do that sensibly you need to understand in detail how the logical CPUs, the enumeration of hardware threads controlled by the OS, map to the physical hardware of the specific machine on which you are running. That mapping can be, and likely is, different on different machines, so you risk binding machine-specific information into your code, which can result in explicitly forcing bad affinities when your code runs on a different machine. And if you are concerned with optimization at this level of detail, your code is probably valuable, and therefore will probably move to another machine.
>
> This interface may also allow you to ignore the resource limitations that were set by the program startup mechanism, such as Message Passing Interface (MPI), specifically to prevent multiple OpenMP processes on the same node from using the same hardware threads. Again, this can result in explicitly forcing affinities that cause bad performance, and the OpenMP runtime will neither prevent this from happening, nor warn you when it does. These are expert interfaces and you must use them with caution.
>
> It is recommended, therefore, to use the higher level affinity settings if you possibly can, because they are more portable and do not require this low level knowledge.

The C/C++ API interfaces follow, where the type name `kmp_affinity_mask_t` is defined in `omp.h`:

Syntax	Description
`int kmp_set_affinity (kmp_affinity_mask_t *mask)`	Sets the affinity mask for the current OpenMP thread to *`mask`, where *`mask` is a set of OS proc IDs that has been created using the API calls listed below, and the thread will only execute on OS procs in the set. Returns either a zero (0) upon success or a nonzero error code.
`int kmp_get_affinity (kmp_affinity_mask_t *mask)`	Retrieves the affinity mask for the current OpenMP thread, and stores it in *`mask`, which must have previously been initialized with a call to `kmp_create_affinity_mask()`. Returns either a zero (0) upon success or a nonzero error code.

Syntax	Description
`int kmp_get_affinity_max_proc (void)`	Returns the maximum OS proc ID that is on the machine, plus 1. All OS proc IDs are guaranteed to be between 0 (inclusive) and `kmp_get_affinity_max_proc()` (exclusive).
`void kmp_create_affinity_mask (kmp_affinity_mask_t *mask)`	Allocates a new OpenMP thread affinity mask, and initializes *`mask` to the empty set of OS procs. The implementation is free to use an object of `kmp_affinity_mask_t` either as the set itself, a pointer to the actual set, or an index into a table describing the set. Do not make any assumption as to what the actual representation is.
`void kmp_destroy_affinity_mask (kmp_affinity_mask_t *mask)`	Deallocates the OpenMP thread affinity mask. For each call to `kmp_create_affinity_mask()`, there should be a corresponding call to `kmp_destroy_affinity_mask()`.
`int kmp_set_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)`	Adds the OS proc ID proc to the set *`mask`, if it is not already. Returns either a zero (0) upon success or a nonzero error code.
`int kmp_unset_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)`	If the OS proc ID proc is in the set *`mask`, it removes it. Returns either a zero (0) upon success or a nonzero error code.
`int kmp_get_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)`	Returns 1 if the OS proc ID proc is in the set *`mask`; if not, it returns 0.

Once an OpenMP thread has set its own affinity mask via a successful call to `kmp_set_affinity()`, then that thread remains bound to the corresponding OS proc set until at least the end of the parallel region, unless reset via a subsequent call to `kmp_set_affinity()`.

Between parallel regions, the affinity mask (and the corresponding OpenMP thread to OS proc bindings) can be considered thread private data objects, and have the same persistence as described in the OpenMP Application Program Interface. For more information, see the OpenMP API specification (http://www.openmp.org), some relevant parts of which are provided below:

In order for the affinity mask and thread binding to persist between two consecutive active parallel regions, all three of the following conditions must hold:

- Neither parallel region is nested inside another explicit parallel region.
- The number of threads used to execute both parallel regions is the same.
- The value of the dyn-var internal control variable in the enclosing task region is false at entry to both parallel regions."

Therefore, by creating a parallel region at the start of the program whose sole purpose is to set the affinity mask for each thread, you can mimic the behavior of the `KMP_AFFINITY` environment variable with low-level affinity API calls, if program execution obeys the three aforementioned rules from the OpenMP specification.

The following example shows how these low-level interfaces can be used. This code binds the executing thread to the specified logical CPU:

Example

```
// Force the executing thread to execute on logical CPU i
// Returns 1 on success, 0 on failure.
int forceAffinity(int i)
{
kmp_affinity_mask_t mask;

kmp_create_affinity_mask(&mask);
kmp_set_affinity_mask_proc(i, &mask);

return (kmp_set_affinity(&mask) == 0);
}
```

This program fragment was written with knowledge about the mapping of the OS proc IDs to the physical processing elements of the target machine. On another machine, or on the same machine with a different OS installed, the program would still run, but the OpenMP thread to physical processing element bindings could differ and you might be explicitly force a bad distribution.

OpenMP* Memory Spaces and Allocators

For storage and retrieval variables, OpenMP* provides memory known as memory spaces. Different memory spaces have different traits. Depending on how a variable is to be used and accessed determines which memory space is appropriate for allocation of the variable.

Each memory space has a unique allocator that is used to allocate and deallocate memory in that space. The allocators allocate variables in contiguous space that does not overlap any other allocation in the memory space. Multiple memory spaces with different traits may map to a single memory resource.

The behavior of the allocator is affected by the allocator traits that you specify. The allocator traits, their possible values, and their default values are shown in the following table:

Allocator Trait	Values That Can Be Specified	Default Value
`access`	• `all` • `cgroup` • `pteam` • `thread`	All
`alignment`	A positive integer value that is a power of 2 specifying number of bytes	1 byte
`fallback`	• `abort_fb` • `allocator_fb` • `default_mem_fb` • `null_fb`	`default_mem_fb`
`fb_data`	An allocator handle	None
`partition`	• `blocked` • `environment` • `interleaved` • `nearest`	`environment`
`pinned`	• `true` • `false`	`false`

Allocator Trait	Values That Can Be Specified	Default Value
pool_size	a positive integer value	Implementation defined
sync_hint	• contended • uncontended • private • serialized	contended

The access trait specifies the accessibility of the allocated memory. The following are values you can specify for access:

- all

  This value indicates that the allocated memory must be accessible by all threads in the device where the memory allocation occurs.

  This is the default setting.
- cgroup

  This value indicates that the allocated memory must be accessible by all threads of the same contention group as the thread that requested the allocation. Accessing the allocated memory thread that is not part of the same contention group results in undefined behavior.
- pteam

  This value indicates that the allocated memory is accessible by all threads that bind to the same parallel region as the thread that requests the allocations. Access to the memory by a thread that does not bind to the same parallel region as the thread that allocated the memory results in undefined behavior.
- thread

  This value indicates that the memory allocated is accessible only by the thread that allocated it. Attempts to allocate the memory by another thread result in undefined behavior.

The alignment trait specifies how allocated variables will be aligned. Variables will be byte-aligned to at least the value specified for this trait. The default setting is 1 byte. Alignment can also be affected by directives and OpenMP runtime allocator routines that specify alignment requirements.

The fallback trait indicates how an allocator behaves if it is unable to satisfy an allocation request. The following are values you can specify for fallback:

- abort_fb

  This value indicates that the program terminates if the allocation request fails.
- allocator_fb

  If this value is specified and the allocation request fails, the allocation will be tried by the allocator specified by the fb_data trait.
- default_mem_fb

  This value indicates that a failed allocation request will be retried in the omp_default_mem_space memory space. All traits for the omp_default_mem_space allocator should be set to the default trait values, except the fallback trait should be set to null_fb. This is the default setting.
- null_fb

  This value indicates the allocator returns a zero value when an allocation request fails.

The fb_data trait lets you specify a fall back allocator to be used if the requested allocator fails to satisfy the allocation request. The fallback trait of the failing allocator must be set to allocator_fb in order for the allocator specified by the fb_data trait to be used.

The partition trait describes the partitioning of allocated memory over the storage resources represented by the memory space of the allocator. The following are values you can specify for partition:

- blocked

This value indicates the allocated memory is partitioned into blocks of memory of approximately equal size with one block per storage resource.

- `environment`

  This value indicates the allocated memory placement is determined by the runtime execution environment. This is the default setting.
- `interleaved`

  This value indicates the allocated memory is distributed in a round-robin fashion across the storage resources.
- `nearest`

  This value indicates that the allocated memory will be placed in the storage resource nearest to the thread that requested the allocation.

If the `pinned` trait has the value `true`, the allocator ensures each allocation made by the allocator will remain in the storage resource at the same location where it was allocated until it is deallocated. The default setting is `false`.

The value of `pool_size` is the total number of bytes of storage available to an allocator when there have been no allocations. The following affect `pool_size`:

- If the `access` trait has the value all, the value of `pool_size` is the limit for all allocations for all threads having access to the allocator.
- If the `access` trait of the allocator has the value `cgroup`, the value of `pool_size` is the limit for allocations made from the threads within the same contention group.
- For allocators with the `access` access trait value of `pteam`, the value of `pool_size` is the limit for allocations made within the same parallel team.
- If the `access` trait has the value `thread`, the value of `pool_size` is the limit for allocations made from each thread using the allocator.
- An allocation request for more space than the value of `pool_size` results in the allocator not fulfilling the allocation request.

The `sync_hint` trait describes the way that multiple threads can access an allocator. The following are values you can specify for `sync_hint`:

- `contended` or `uncontended`

  Value `contended` indicates that many threads are anticipated to make simultaneous allocation requests while the value `uncontended` indicates that few threads are anticipated to make simultaneous allocation. The default setting is `contended`.
- `private`

  This value indicates that all allocation requests will come from the same thread. Specifying `private` when this is not the case and two or more threads make allocation requests by the same allocator results in undefined behavior.
- `serialized`

  This value indicates that only one thread will request an allocation at a given time. The behavior is undefined if two threads request an allocation simultaneously by an allocator whose `sync_hint` value is serialized.

There are five predefined memory spaces in OpenMP:

- The system default memory is referred to as `omp_default_mem_space`.
- Large capacity memory is referred to as `omp_large_cap_mem_space`.
- High bandwidth memory is referred to as `omp_high_bw_mem_space`.
- Low latency memory is referred to as `omp_low_lat_mem_space`.
- Memory designed for optimal storage of constant values is referred to as `omp_const_mem_space`.

  It can be initialized with compile-time constant expressions or by using a firstprivate clause.

  Writing to variables in `omp_const_mem_space` results in undefined behavior.

The following table shows the predefined memory allocators, the memory space they are associated with, and the non-default memory trait values they possess.

Allocator Name	Associated Memory Space	Non-Default Trait Values
omp_default_mem_alloc	omp_default_mem_space	fallback=null_fb
omp_large_cap_mem_alloc	omp_large_cap_mem_space	none
omp_low_lat_mem_alloc	omp_low_lat_mem_space	none
omp_high_bw_mem_alloc	omp_high_bw_mem_space	none
omp_const_mem_alloc	omp_const_mem_space	none
omp_cgroup_mem_alloc	implementation/system defined	access=cgroup
omp_pteam_mem_alloc	implementation/system defined	access=pteam
omp_thread_mem_alloc	implementation/system defined	access=thread

See Also

OpenMP* Run-time Library Routines

OpenMP* Advanced Issues

This topic discusses how to use the OpenMP* library functions and environment variables and discusses some guidelines for enhancing performance with OpenMP.

OpenMP provides specific function calls, and environment variables. See the following topics to refresh your memory about the primary functions and environment variable used in this topic:

- OpenMP Run-time Library Routines
- OpenMP Environment Variables

To use the function calls, include the `omp.h` header file. This file is installed in the INCLUDE directory during the compiler installation and compile the application using the `/Qopenmp` (Windows*) or `-qopenmp` (Linux* and macOS) option.

The following example demonstrates how to use the OpenMP functions to print the alphabet and illustrates several important concepts:

1. When using functions instead of pragmas, your code must be rewritten; rewrites can mean extra debugging, testing, and maintenance efforts.
2. It becomes difficult to compile without OpenMP support.
3. It is very easy to introduce simple bugs, as in the loop (shown in example) that fails to print all the letters of the alphabet when the number of threads is not a multiple of 26.
4. You lose the ability to adjust loop scheduling without creating your own work-queue algorithm, which is a lot of extra effort. You are limited by your own scheduling, which is mostly likely static scheduling as shown in the example.

```
#include <stdio.h>
#include <omp.h>

int main(void) {
    int i;
    omp_set_num_threads(4);

    #pragma omp parallel private(i)
    {
        // OMP_NUM_THREADS is not a multiple of 26,
```

```
        // which can be considered a bug in this code.
        int LettersPerThread = 26 / omp_get_num_threads();
        int ThisThreadNum = omp_get_thread_num();
        int StartLetter = 'a'+ThisThreadNum*LettersPerThread;
        int EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread;

        for (i=StartLetter; i<EndLetter; i++) { printf("%c", i); }
    }
    printf("\n");
    return 0;
}
```

Debugging threaded applications is a complex process because debuggers change the run-time performance, which can mask race conditions. Even print statements can mask issues, because they use synchronization and operating system functions. OpenMP itself also adds some complications, because it introduces additional structure by distinguishing private variables and shared variables and inserts additional code. A debugger that supports OpenMP can help you to examine variables and step through threaded code. You can use Intel® Inspector to detect many hard-to-find threading errors analytically. Sometimes, a process of elimination can help identify problems without resorting to sophisticated debugging tools.

Remember that most mistakes are race conditions. Most race conditions are caused by shared variables that really should have been declared private. Start by looking at the variables inside the parallel regions and make sure that the variables are declared private when necessary. Next, check functions called within parallel constructs. By default, variables declared on the stack are private, but the C/C++ keyword `static` changes the variable to be placed on the global heap and therefore shared for OpenMP loops.

The `default(none)` clause can be used to help find those hard-to-spot variables. If you specify `default(none)`, then every variable must be declared with a data-sharing attribute clause. For example:

```
#pragma omp parallel for default(none) private(x,y) shared(a,b)
```

Another common mistake is using uninitialized variables. Remember that private variables do not have initial values upon entering a parallel construct. Use the `firstprivate` and `lastprivate` clauses to initialize them only when necessary, because doing so adds extra overhead.

If you still can't find the bug, then consider the possibility of reducing the scope. Try a binary-hunt. Force parallel sections to be serial again with `if(0)` on the parallel construct or commenting out the pragma altogether. Another method is to force large chunks of a parallel region to be critical sections. Pick a region of the code that you think contains the bug and place it within a critical section. Try to find the section of code that suddenly works when it is within a critical section and fails when it is not. Now look at the variables, and see if the bug is apparent. If that still doesn't work, try setting the entire program to run in serial by setting the compiler-specific environment variable `KMP_LIBRARY=serial`.

If the code is still not working, and you are not using any OpenMP API function calls, compile it without the `/Qopenmp` (Windows) or `-qopenmp` (Linux and macOS) option to make sure the serial version works. If you are using OpenMP API function calls, use the `/Qopenmp-stubs` (Windows) or `-qopenmp-stubs` (Linux and macOS) option.

Performance

OpenMP threaded application performance is largely dependent upon the following things:

- The underlying performance of the single-threaded code.
- CPU utilization, idle threads, and load balancing.
- The percentage of the application that is executed in parallel by multiple threads.
- The amount of synchronization and communication among the threads.
- The overhead needed to create, manage, destroy, and synchronize the threads, made worse by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions.
- Performance limitations of shared resources such as memory, bus bandwidth, and CPU execution units.
- Memory conflicts caused by shared memory or falsely shared memory.

Performance always begins with a properly constructed parallel algorithm or application. For example, parallelizing a bubble-sort, even one written in hand-optimized assembly language, is not a good place to start. Keep scalability in mind; creating a program that runs well on two CPUs is not as efficient as creating one that runs well on *n* CPUs. With OpenMP, the number of threads is chosen by the compiler, so programs that work well regardless of the number of threads are highly desirable. Producer/consumer architectures are rarely efficient, because they are made specifically for two threads.

Once the algorithm is in place, make sure that the code runs efficiently on the targeted Intel® architecture; a single-threaded version can be a big help. Turn off the `/Qopenmp` (Windows) or `-qopenmp` (Linux and macOS) option to generate a single-threaded version, or build with the `/Qopenmp-stubs` (Windows) or `-qopenmp-stubs` (Linux and macOS) option, and run the single-threaded version through the usual set of optimizations.

Once you have gotten the single-threaded performance, it is time to generate the multi-threaded version and start doing some analysis.

Optimizations are really a combination of patience, experimentation, and practice. Make little test programs that mimic the way your application uses the computer resources to get a feel for what things are faster than others. Be sure to try the different scheduling clauses for the parallel sections of code. If the overhead of a parallel region is large compared to the compute time, you may want to use an `if` clause to execute the section serially.

See Also

OpenMP* Run-time Library Routines
Worksharing Using OpenMP*
qopenmp, Qopenmp
qopenmp-stubs, Qopenmp-stubs

OpenMP* Implementation-Defined Behaviors

This topic summarizes the behaviors that are described as implementation defined in the OpenMP* API specification.

NOTE
Internal Control Variables (ICVs) mentioned below are discussed in the OpenMP API specification.

Name	Description
`single` construct	The first thread that encounters the single construct executes the structured block.
`teams` construct	The number of teams that are created is equal to 1 if you don't specify the `num_teams` clause.
`dist_schedule` clause, `distribute` construct	If you don't specify the `dist_schedule` clause, then the schedule for the `distribute` construct is `static`.
`omp_set_num_threads` routine	If the argument is not a positive integer, then Intel's OpenMP implementation sets the value of the first element of the `nthreads-var` ICV of the current task to 1.
`omp_set_max_active_levels` routine	If the argument is a negative integer this call is ignored and the last valid setting is used.

Name	Description
`omp_get_max_active_levels` routine	When called from within any explicit parallel region the binding thread set, and binding region, if required, for the `omp_get_max_active_levels` region is the current task region.
`OMP_SCHEDULE` environment variable	If the value of the variable does not conform to the specified format then the value of the `run-sched-var` ICV is set to `static`.
`OMP_NUM_THREADS` environment variable	If any value of the list specified in the environment variable is negative then the whole list is ignored. If any value of the list is zero then this value is set to 1.
`OMP_PROC_BIND` environment variable	If the value is not `true`, `false`, or a comma separated list of `master` (deprecated), `primary`, `close`, or `spread`, then Intel's OpenMP implementation sets the value of `bind-var` ICV to false.
`OMP_DYNAMIC` environment variable	If the value is neither `true` nor `false`, then the implementation sets the value of `dyn-var` ICV to false.
`OMP_NESTED` environment variable	If the value is neither `true` nor `false`, then the implementation sets the value of `nest-var` ICV to false.
`OMP_STACKSIZE` environment variable	If the value does not conform to the specified format or the implementation cannot provide a stack of the specified size, then Intel's OpenMP implementation sets the value of `stacksize-var` ICV to the default size, which is specified as being from 1MB to 4MB depending on the architecture. On Linux* or macOS*, the implementation can set the value of `stacksize-var` ICV up to 256MB, respecting the operating system's stack size limit.
`OMP_MAX_ACTIVE_LEVELS` environment variable	If the value is a negative integer or is greater than the number of parallel levels an implementation can support, then Intel's OpenMP implementation sets the value of the `max-active-levels-var` ICV to 1.
`OMP_THREAD_LIMIT` environment variable	If the requested value is greater than the number of threads an implementation can support, or if the value is a negative integer, then Intel's OpenMP implementation sets the value of the `thread-limit-var` ICV to the maximum number of threads supported on a particular platform. If the requested value is zero then the implementation sets the value of the `thread-limit-var` ICV to 1.
Runtime library definitions	Intel's OpenMP implementation provides both the include file `omp.h` and `omp-tools.h` .

OpenMP* Examples

The following examples show how to use several OpenMP* features.

A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. Dynamic scheduling is used to improve load balancing.

The `for` pragma has a `nowait` clause because there is an implicit barrier at the end of the parallel region. Therefore it is not necessary to also have a barrier at the end of the `for` region.

```
void for1(float a[], float b[], int n) {
  int i, j;
  #pragma omp parallel shared(a,b,n) {
   #pragma omp for schedule(dynamic,1) private (i,j) nowait
    for (i = 1; i < n; i++)
       for (j = 0; j < i; j++)
         b[j + n*i] = (a[j + n*i] + a[j + n*(i-1)]) / 2.0;
  }
}
```

Two Difference Operators: for Loop Version

This example uses two parallel loops fused to reduce fork/join overhead. The first `for` pragma has a `nowait` clause because all the data used in the second loop is different than all the data used in the first loop.

```
void for2(float a[], float b[], float c[], float d[], int n, int m) {
  int i, j;
  #pragma omp parallel shared(a,b,c,d,n,m) private(i,j) {
    #pragma omp for schedule(dynamic,1) nowait
    for (i = 1; i < n; i++)
      for (j = 0; j < i; j++)
        b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
    #pragma omp for schedule(dynamic,1) nowait
    for (i = 1; i < m; i++)
      for (j = 0; j < i; j++)
        d[j + m*i] = ( c[j + m*i] + c[j + m*(i-1)] )/2.0;
  }
}
```

Two Difference Operators: sections Version

This example demonstrates the use of the `sections` pragma . The logic is identical to the preceding `for` pragma example, but uses a `sections` pragma instead of a `for` pragma . Here the speedup is limited to two because there are only two units of work whereas in the example above there are `(n-1) + (m-1)` units of work.

```
void sections1(float a[], float b[], float c[], float d[], int n, int m) {
  int i, j;
  #pragma omp parallel shared(a,b,c,d,n,m) private(i,j) {
    #pragma omp sections nowait {
      #pragma omp section
       for (i = 1; i < n; i++)
         for (j = 0; j < i; j++)
           b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
      #pragma omp section
       for (i = 1; i < m; i++)
```

```
        for (j = 0; j < i; j++)
          d[j + m*i] = ( c[j + m*i] + c[j + m*(i-1)] )/2.0;
    }
  }
}
```

Update a Shared Scalar

This example demonstrates how to use a `single` construct to update an element of the shared array `a`. The optional `nowait` clause after the first loop is omitted because it is necessary to wait at the end of the loop before proceeding into the `single` construct to avoid a race condition.

```
void sp_1a(float a[], float b[], int n) {
  int i;
  #pragma omp parallel shared(a,b,n) private(i) {
    #pragma omp for
      for (i = 0; i < n; i++)
        a[i] = 1.0 / a[i];
      #pragma omp single
        a[0] = MIN( a[0], 1.0 );
      #pragma omp for nowait
      for (i = 0; i < n; i++)
      b[i] = b[i] / a[i];
  }
}
```

Automatic Parallelization

The auto-parallelization feature of the Intel®C++ Compiler automatically translates serial portions of the input program into equivalent multithreaded code. Automatic parallelization determines the loops that are good worksharing candidates, performs the dataflow analysis to verify correct parallel execution, and partitions the data for threaded code generation as needed in programming with OpenMP* directives. The OpenMP* and auto-parallelization functionality provides the performance gains from shared memory on multiprocessor and dual core systems.

The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multithreaded code for those loops which can safely and efficiently be executed in parallel.

This behavior enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

The guided auto-parallelization feature of the Intel®C++ Compiler helps you locate portions in your serial code that can be parallelized further. You can invoke guidance for parallelization, vectorization, or data transformation using specified compiler options of the `[Q]guide` series.

Automatic parallelization frees developers from having to:

- Find loops that are good worksharing candidates.
- Perform the dataflow analysis to verify correct parallel execution.
- Partition the data for threaded code generation as is needed in programming with OpenMP* directives.

Although OpenMP* directives enable serial applications to transform into parallel applications quickly, you must explicitly identify specific portions of your application code that contain parallelism and add the appropriate compiler directives. Auto-parallelization, which is triggered by the `[Q]parallel` option, automatically identifies those loop structures that contain parallelism. During compilation, the compiler automatically attempts to deconstruct the code sequences into separate threads for parallel processing. No other effort is needed.

NOTE In order to execute a program that uses auto-parallelization on Linux* or macOS systems, you must include the `-parallel` compiler option when you compile and link your program.

NOTE
Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

Serial code can be divided so that the code can execute concurrently on multiple threads. For example, consider the following serial code example.

Example 1: Original Serial Code

```
void ser(int *a, int *b, int *c) {
  for (int i=0; i<100; i++)
    a[i] = a[i] + b[i] * c[i];
}
```

The following example illustrates one method showing how the loop iteration space, shown in the previous example, might be divided to execute on two threads.

Example 2: Transformed Parallel Code

```
void par(int *a, int *b, int *c) {
  int i;
  // Thread 1
  for (i=0; i<50; i++)
    a[i] = a[i] + b[i] * c[i];
  // Thread 2
  for (i=50; i<100; i++)
    a[i] = a[i] + b[i] * c[i];
}
```

Auto-Vectorization and Parallelization

Auto-vectorization detects low-level operations in the program that can be done in parallel, and then converts the sequential program to process 2, 4, 8, or (up to) 16 elements in one operation, depending on the data type. In some cases, auto-parallelization and vectorization can be combined for better performance results.

The following example demonstrates how code can be designed to explicitly benefit from parallelization and vectorization. Assuming you compile the code shown below using the `[Q]parallel` option, the compiler will parallelize the outer loop and vectorize the innermost loop.

Example

```
#include <stdio.h>
#define ARR_SIZE 500 //Define array
int main() {
  int matrix[ARR_SIZE][ARR_SIZE];
  int arrA[ARR_SIZE]={10};
  int arrB[ARR_SIZE]={30};
  int i, j;
  for(i=0;i<ARR_SIZE;i++) {
```

Example

```
     for(j=0;j<ARR_SIZE;j++) { matrix[i][j] = arrB[i]*(arrA[i]%2+10); }
  } printf("%d\n",matrix[0][0]);
}
```

Compiling the example code with the correct options, the compiler should report results similar to the following:

```
vectorization.c(18) : (col. 6) remark: LOOP WAS VECTORIZED.
     vectorization.c(16) : (col. 3) remark: LOOP WAS AUTO-PARALLELIZED.
```

With the relatively small effort of adding OpenMP* directives to existing code you can transform a sequential program into a parallel program. The [Q]openmp option must be specifed to enable the OpenMP directives.

The following example demonstrates one method of using the OpenMP* pragmas within code.

Example

```
#include <stdio.h>
#define ARR_SIZE 100 //Define array
void foo(int ma[][ARR_SIZE], int mb[][ARR_SIZE], int *a, int *b, int *c);
int main() {
  int arr_a[ARR_SIZE];
  int arr_b[ARR_SIZE];
  int arr_c[ARR_SIZE];
  int i,j;
  int matrix_a[ARR_SIZE][ARR_SIZE];
  int matrix_b[ARR_SIZE][ARR_SIZE];
  #pragma omp parallel for
// Initialize the arrays and matrices.
  for(i=0;i<ARR_SIZE; i++) {
    arr_a[i]= i;
    arr_b[i]= i;
    arr_c[i]= ARR_SIZE-i;
    for(j=0; j<ARR_SIZE;j++) {
       matrix_a[i][j]= j;
       matrix_b[i][j]= i;
    }
  }
  foo(matrix_a, matrix_b, arr_a, arr_b, arr_c);
}
void foo(int ma[][ARR_SIZE], int mb[][ARR_SIZE], int *a, int *b, int *c)
{
  int i, num, arr_x[ARR_SIZE];
  #pragma omp parallel for private(num)
// Expresses the parallelism using the OpenMP pragma: parallel for.
// The pragma guides the compiler generating multithreaded code.
// Array arr_X, mb, b, and c are shared among threads based on OpenMP
// data sharing rules. Scalar num si specifed as private
// for each thread.
  for(i=0;i<ARR_SIZE;i++) {
     num = ma[b[i]][c[i]];
     arr_x[i]= mb[a[i]][num];
     printf("Values: %d\n", arr_x[i]); //prints values 0-ARR_SIZE-1
    }
}
```

NOTE
Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Using Parallelism Reports

To generate a parallelism report, use the `-opt-report-phase=`*par* (Linux* and macOS) or the `/Qopt-report-phase:`*par* option along with the `-opt-report=`*n* or `/Qopt-report:`*n* option. By default the auto-parallelism report generates a medium level of detail, where *n*`=2`. You can use `[Q]opt-report` option along with the `[Q]opt-report-phase` option if you want a greater or lesser level of detail. Specifying a value of '5' generates the maximum diagnostic details.

Run the report by entering commands similar to the following:

Operating System	Command
Linux*	`icpc -c -parallel -opt-report-phase=par -opt-report:5 sample.cpp`
macOS	`icl++ -c -parallel -opt-report-phase=par -opt-report:5 sample.cpp`
Windows*	`icl /c /Qparallel /Qopt-report-phase:par /Qopt-report:5 sample.cpp`

NOTE The `-c` (Linux* and macOS) or `/c` (Windows*) prevents linking and instructs the compiler to stop compilation after the object file is generated. The example is compiled without generating an executable.

The output, by default, produces a file with the same name as the object file, with `.optrpt` extension, and is written into the same directory as the object file. Using the above command-line entries, you will obtain an output file called `sample.optrpt`. Use the `[Q]opt-report-file` option to specify any other name for the output file that captures the report results. Use the arguments *stdout* or *stderr* to send the optimization report to stdout or stderr.

For example, assume you want a full diagnostic report on the following example code:

Example

```
void no_par(void) {
  int i;
  int a[1000];
  for (i=1; i<1000; i++) {
    a[i] = (i * 2) % i * 1 + sqrt(i);
    a[i] = a[i-1] + i;
  }
}
```

The following example output illustrates the diagnostic report generated by the compiler for the example code shown above. In most cases, the comment listed next to the line is self-explanatory.

Example Parallelism Report

```
procedure: no_par
sample.c(13):(3) remark #15048: DISTRIBUTED LOOP WAS AUTO-PARALLELIZED
sample.c(13):(3) remark #15050: loop was not parallelized: existence of parallel dependence
sample.c(19):(5) remark #15051: parallel dependence: proven FLOW dependence between a line 19,
and a line 19
```

For more information on options to generate reports see the Optimization Report Options topic.

See Also

Guided Auto-Parallelization

`parallel`, `Qparallel`
compiler option

`par-runtime-control`, `Qpar-runtime-control`
compiler option

`par-threshold`, `Qpar-threshold`
compiler option

`guide`, `Qguide`
compiler option

`qopt-report-phase`, `Qopt-report-phase`
compiler option

`qopt-report`, `Qopt-report`
compiler option

Enabling Auto-parallelization

To enable the auto-parallelizer, use the `[Q]parallel` option. This option detects parallel loops capable of being executed safely in parallel, and automatically generates multi-threaded code for these loops.

> **NOTE** You may need to set the `KMP_STACKSIZE` environment variable to an appropriately large size to enable parallelization with this option.

> **NOTE**
> Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

An example of the command using auto-parallelization is as follows:

Commanding auto-parallelization in Linux*

```
icc -c -parallel prog.cpp
```

Commanding auto-parallelization in Windows*

```
icl /c /Qparallel prog.cpp
```

Commanding auto-parallelization in macOS
`icc -c -parallel prog.cpp`

Auto-parallelization uses two specific pragmas: `#pragma parallel` and `#pragma noparallel`.

The format of an auto-parallelization compiler pragma is below:

Syntax
`<prefix> <pragma>`

where `<prefix>` indicates `#pragma`, the `<prefix>` is followed by the pragma name, as in:

Syntax
`#pragma parallel`

The `#pragma parallel` pragma instructs the compiler to ignore dependencies that it assumes may exist and that would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored. In addition, `parallel [always]` overrides the compiler heuristics that estimate the likelihood that parallelization of a loop increases performance. It allows a loop to be parallelized even if the compiler thinks parallelization may not improve performance. If the `ASSERT` keyword is added, as in `#pragma parallel [always [assert]]`, the compiler generates an error-level assertion message saying that the compiler analysis and cost model indicate that the loop cannot be parallelized.

The `#pragma noparallel` pragma disables auto-parallelization.

See Also

`parallel`, `Qparallel`
compiler option

Programming with Auto-parallelization

The auto-parallelization feature implements some concepts of OpenMP*, such as the worksharing construct (with the `PARALLEL for` directive). This section provides details on auto-parallelization.

Guidelines for Effective Auto-parallelization Usage

A loop can be parallelized if it meets the following criteria:

- The loop is countable at compile time: This means that an expression representing how many times the loop will execute (loop trip count) can be generated just before entering the loop.
- There are no `FLOW` (*READ* after *WRITE*), `OUTPUT` (*WRITE* after *WRITE*) or `ANTI` (*WRITE* after *READ*) loop-carried data dependencies. A loop-carried data dependency occurs when the same memory location is referenced in different iterations of the loop. At the compiler's discretion, a loop may be parallelized if any assumed inhibiting loop-carried dependencies can be resolved by run-time dependency testing.

The compiler may generate a run-time test for the profitability of executing in parallel for loop, with loop parameters that are not compile-time constants.

Coding Guidelines

Enhance the power and effectiveness of the auto-parallelizer by following these coding guidelines:

- Expose the trip count of loops whenever possible; use constants where the trip count is known and save loop parameters in local variables.
- Avoid placing structures inside loop bodies that the compiler may assume to carry dependent data, for example, procedure calls, ambiguous indirect references or global references.

Auto-parallelization Data Flow

For auto-parallelization processing, the compiler performs the following steps:

1. **Data flow analysis:** Computing the flow of data through the program.
2. **Loop classification:** Determining loop candidates for parallelization based on correctness and efficiency, as shown by Enabling Auto-parallelization.
3. **Dependency analysis:** Computing the dependency analysis for references in each loop nest.
4. **High-level parallelization:** Analyzing the dependency graph to determine loops that can execute in parallel, and computing run-time dependency.
5. **Data partitioning:** Examining data reference and partition based on the following types of access: *SHARED*, *PRIVATE*, and *FIRSTPRIVATE*.
6. **Multithreaded code generation:** Modifying loop parameters, generating entry/exit per threaded task, and generating calls to parallel run-time routines for thread creation and synchronization.

NOTE
Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

See Also

Enabling Auto-parallelization

Enabling Further Loop Parallelization for Multicore Platforms

Parallelizing loops for multicore platforms is subject to certain conditions. Three requirements must be met for the compiler to parallelize a loop:

- The number of iterations must be known before entry into a loop to insure that the work can be divided in advance. A `do while` loop, for example, usually cannot be made parallel.
- There can be no jumps into or out of the loop.
- The loop iterations must be independent (no cross-iteration dependencies).

Correct results must not logically depend on the order in which the iterations are executed. There may be slight variations in the accumulated rounding error, for example, when the same quantities are added in a different order. In some cases, such as summing an array or other uses of temporary scalars, the compiler may be able to remove an apparent dependency by a simple transformation.

Potential aliasing of pointers or array references is another common impediment to safe parallelization. Two pointers are aliased if both point to the same memory location. The compiler may not be able to determine whether two pointers or array references point to the same memory location, for example, if they depend on function arguments, run-time data, or the results of complex calculations.

If the compiler cannot prove that pointers or array references are safe, it will not parallelize the loop, except in limited cases when it is deemed worthwhile to generate alternative code paths to test explicitly for aliasing at run-time.

An alternative way in C to assert that a pointer is not aliased is to use the `restrict` keyword in the pointer declaration, along with the `[Q]restrict` command-line option. The compiler will never parallelize a loop that it can prove to be unsafe.

If you know parallelizing a particular loop is safe and that potential aliases can be ignored, you can instruct the compiler to parallelize the loop using the `#pragma parallel` pragma.

Parallelizing Loops with Cross-iteration Dependencies

Before the compiler can auto-parallelize a loop, it must prove that the loop does not have potential cross-iteration dependencies that prevent parallelization. A cross-iteration dependency exists if a memory location is written to in an iteration of a loop and accessed (read from or written to) in another iteration of the loop. Cross-iteration dependencies often occur in loops that access overlapping array ranges, such as a loop that reads from `a(1:100)` and writes to `a(0:99)`.

Sometimes, even though a loop does not have cross-iteration dependencies, the compiler does not have enough information to prove it and does not parallelize the loop. In such cases, you can assist the compiler by providing additional information about the loop using the `#pragma parallel` pragma. Adding the `#pragma parallel` pragma before a `for` loop informs the compiler that the loop does not have cross-iteration dependencies. Auto-parallelization analysis ignores potential dependencies that it assumes could exist; however, the compiler still may not parallelize the loop if heuristics estimate parallelization is unlikely to increase performance of the loop.

The `#pragma parallel always` pragma has the same effect to ignore potential dependencies as the `#pragma parallel` pragma, but it also overrides the compiler heuristics that estimate the likelihood that parallelization of a loop would increase performance. It allows a loop to be parallelized even when the compiler estimates that parallelization might not improve performance.

The `#pragma noparallel` pragma prevents auto-parallelization of the immediately following `for` loop. Unlike `#pragma parallel`, which is a hint, the `noparallel` pragma is guaranteed to prevent parallelization of the following loop.

These pragmas take effect only if auto-parallelization is enabled by the option `[Q]parallel`.

Parallelizing Loops with Private Clauses

When you use the Guided Auto Parallelism feature, the compiler's auto-parallelizer gives you advice on where to alter your program to enhance parallelization. For instance, you may get advice to check if a condition (that the compiler could not prove) is true, and if true, to insert `#pragma parallel` in your source code so that the associated loop is parallelized when you recompile.

To specify that it is legal for each thread to create a new, private copy (not visible by other threads) of a variable, and replace the original variable in the loop with the new private variable, use the `#pragma parallel` pragma with the *private* clause. The *private* clause allows you to list scalar and array type variables and specify the number of array elements to privatize.

Use the *firstprivate* clause to specify private variables that need to be initialized with the original value before entering the parallel loop.

Use the *lastprivate* clause to specify those variables with a value you want to reuse after it exits a parallelized loop. When you use the *lastprivate* clause to handle a particular privatized variable, the value is copied to the original variable when it exits from the parallelized loop.

> **NOTE**
> Do not use the same variable in both *private* and *lastprivate* clauses for the same loop. You will get an error message.

Parallelizing Loops with External Function Calls

The compiler can only effectively analyze loops with a relatively simple structure. For example, the compiler cannot determine the thread safety of a loop containing external function calls because it does not know whether the function call might have side effects that introduce dependencies. You can invoke interprocedural optimization with the `[Q]ipo` option. Using this option gives the compiler the opportunity to analyze the called function for side effects.

Parallelizing Loops with OpenMP*

When the compiler is unable to automatically parallelize loops you know to be parallel, use OpenMP*. OpenMP* is the preferred solution because you understand the code better than the compiler and can express parallelism at a coarser granularity. Alternatively, automatic parallelization can be effective for nested loops, such as those in a matrix multiply. Moderately coarse-grained parallelism results from threading of the outer loop, allowing the inner loops to be optimized for fine-grained parallelism using vectorization or software pipelining.

Threshold Parameter to Parallelize Loops

If a loop can be parallelized, it does not necessarily mean that it should be parallelized. The compiler uses a threshold parameter to decide whether to parallelize a loop. The `[Q]par-threshold` compiler option adjusts this behavior. The threshold ranges from *0* to *100* , where *0* instructs the compiler to always parallelize a safe loop and *100* instructs the compiler to only parallelize those loops for which a performance gain is highly probable. Use the `[Q]par-report` option to determine which loops were parallelized. The compiler will also report which loops could not be parallelized and indicate probable reason(s) why. See OpenMP* and Parallel Processing Options for more information on the using these compiler options.

The following example illustrates using the options in combination.

Example code

```
void add (int k, float *a, float *b) {
  for (int i = 1; i < 10000; i++) {
   a[i] = a[i+k] + b[i];
  }
}
```

Entering a command-line compiler command similar to the following will result in the compiler issuing parallelization messages:

```
//Linux* and macOS
icpc -c -parallel -opt-report-phase=par -opt-report=3 add.cpp
```

The compiler might report results similar to those listed below:

Sample results

```
add.cpp
procedure:
add serial loop: line 2
anti data dependence assumed from line 2 to line 2, due to "a"
flow data dependence assumed from line 2 to line 2, due to "a"
flow data dependence assumed from line 2 to line 2, due to "a"
```

Because the compiler does not know the value of *k*, the compiler assumes the iterations depend on each other, for example if *k* equals -1, even if the actual case is otherwise. You can override the compiler by inserting the `#pragma parallel` pragma.

Example

```
void add(int k, float *a, float *b) {
  #pragma parallel
  for (int i = 0; i < 10000; i++) {
    a[i] = a[i+k] + b[i];
  }  }
```

Caution
Do not call this function with a value of *k* that is less than 10000; passing a value less than 10000 could lead to incorrect results.

See Also

parallel
pragma

OpenMP* and Parallel Processing Options

`qopt-report-phase, Qopt-report-phase` compiler option

`opt-report, Qopt-report`
compiler option

`par-threshold, Qpar-threshold`
compiler option

`restrict, Qrestrict`
compiler option

`ipo, Qipo`
compiler option

Language Support for Auto-parallelization

This topic addresses specific C++ language features that better help to parallelize code.

Annotating Functions with Declarations

Annotating functions with the declaration:

```
//  (Windows* OS)
__declspec(concurrency_safe(cost(cycles) | profitable)) -OR-//  (Linux* OS)
__attribute__(concurrency_safe(cost(cycles) | profitable))
```

guides the compiler to parallelize more loops and straight-line code.

Using the `concurrency_safe` attribute indicates to the compiler that there are no unaffected side-effects and no illegal (or improperly synchronized) memory access interfences among multiple invocations of the annotated function or between an invocation of this annotated function and other statements in the program, if they are executed concurrently.

NOTE
For every function that is annotated with the `concurrency_safe` attribute, it is your responsibility to ensure that its side effects (if any) are acceptable (or expected), and the memory access interferences are properly synchronized.

The *cost* clause specifies the execution cycles of the annotated function for the compiler to perform parallelization profitability analysis while compiling its enclosing loops or blocks. The *profitable* clause indicates that the loops or blocks that contain calls to the annotated function are profitable to parallelize.

NOTE
The value of *cycles* is a 2-byte unsigned integer (unsigned short), its maximal value is 2^^{16-1}. If the cycle count is greater than 2^^{16-1}, the user should use *profitable* clause.

The following example illustrates the use of this declaration.

Example using __declspec(concurrency_safe(cost(cycles) | profitable))

```
#define N 10
#define M 40
#define NValue N

#if defined(COSTLOW)

// The function cost is ~5 cycles, the loop calling "foo" will not be parallellized
__declspec(concurrency_safe(cost(5)))
#elif defined(COSTHIGH)

// The function cost is ~100 cycles, so the loop calling "foo" will be paralleized
__declspec(concurrency_safe(cost(200)))
#elif defined(PROFITABLE)

// The function is profitable to be executed in parallel, so the loop calling "foo"
// should be paralleized.
__declspec(concurrency_safe(profitable))
#endif

__declspec(noinline)
int foo(float A[], float B[]) {
   for (int i = 0; i < N; i++) {
   B[i] = A[i];
   }
   return N;
}

int testp(float A[], float B[], float* In[], float* Out[]) {
   int i, j;
   for (i = 0; i < M; i++) {
     foo (A, B);
     for (j = 0; j < N; j++) {
       Out[i][j] = In[i][j] + (NValue*j);
     }
   }
   return N;
}

[C:/temp] icl -c -DCOSTLOW -Qparallel -Qpar-report2 -Qansi-alias v.cpp
C:\temp\v.cpp(28): (col. 3) remark: loop was not parallelized: insufficient computational work.

[C:/temp] icl -c -DCOSTHIGH -Qparallel -Qpar-report -Qansi-alias v.cpp
C:\temp\v.cpp(28): (col. 3) remark: LOOP WAS AUTO-PARALLELIZED.

[C:/temp] icl -c -DPROFITABLE -Qparallel -Qpar-report -Qansi-alias v.cpp
C:\temp\v.cpp(28): (col. 3) remark: LOOP WAS AUTO-PARALLELIZED.
```

See Also

`__declspec(concurrency_safe)` declaration

Vectorization

Vectorization is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector (a series of adjacent values).

Automatic Vectorization

The automatic vectorizer (also called the auto-vectorizer) is a component of the compiler that automatically uses SIMD instructions in the Intel® Streaming SIMD Extensions (Intel® SSE, Intel® SSE2, Intel® SSE3 and Intel® SSE4), Supplemental Streaming SIMD Extensions (SSSE3) instruction sets, Intel® Advanced Vector Extensions (Intel® AVX, Intel® AVX2) instruction sets, and Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instruction set. The vectorizer detects operations in the program that can be done in parallel and converts the sequential operations to parallel; for example, the vectorizer converts the sequential SIMD instruction that processes up to 16 elements into a parallel operation, depending on the data type.

Automatic vectorization occurs when the compiler generates packed SIMD instructions to unroll a loop. Because the packed instructions operate on more than one data element at a time, the loop executes more efficiently. This process is referred to as auto-vectorization only to emphasize that the compiler identifies and optimizes suitable loops on its own, without external input. However, it is useful to note that in some cases, certain keywords or directives may be applied in the code for auto-vectorization to occur.

The compiler supports a variety of auto-vectorizing hints that can help the compiler to generate effective vector instructions. Automatic vectorization is supported on IA-32 (for C++ only) and Intel® 64 architectures. Intel® Advisor, a separate tool included in the Intel® oneAPI Base Toolkit, provides a Vectorization Advisor feature that can analyze the compiler's optimization reports and make recommendations for enhancing vectorization.

> **NOTE**
> This option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

Vectorization Programming Guidelines

The goal of including the vectorizer component in the Intel® C++ Compiler Classic is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help by supplying the compiler with additional information; for example, by using auto-vectorizer hints or pragmas.

> **NOTE**
> This option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

Guidelines to Vectorize Innermost Loops

Follow these guidelines to vectorize innermost loops bodies.

Use:

- Straight-line code (a single basic block).

- Vector data only (arrays and invariant expressions on the right hand side of assignments). Array references can appear on the left hand side of assignments.
- Only assignment statements.

Avoid:

- Function calls (other than math library calls).
- Non-vectorizable operations (either because the loop cannot be vectorized, or because an operation is emulated through a number of instructions).
- Mixing vectorizable types in the same loop (leads to lower resource utilization).
- Data-dependent loop exit conditions (leads to loss of vectorization).

To make your code vectorizable, you need to edit your loops. You should only make changes that enable vectorization, and avoid these common changes:

- Loop unrolling, which the compiler performs automatically.
- Decomposing one loop with several statements in the body into several single-statement loops.

Restrictions

There are a number of restrictions that you should consider. Vectorization depends on two major factors: hardware and style of source code.

Factor	Description
Hardware	The compiler is limited by restrictions imposed by the underlying hardware. Intel® Streaming SIMD Extensions (Intel® SSE) has vector memory operations that are limited to stride-1 accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture.
Style of source code	The style in which you write source code can inhibit vectorization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, pointer arithmetic, and memory operations within the loop bodies.

By understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorization.

Guidelines for Writing Vectorizable Code

Follow these guidelines to write vectorizable code:

- Use simple `for` loops. Avoid complex loop termination conditions – the upper iteration limit must be invariant within the loop. For the innermost loop in a nest of loops, you could set the upper limit iteration to be a function of the outer loop indices.
- Write straight-line code. Avoid branches such as `switch`, `goto`, or `return` statements; most function calls; or `if` constructs that cannot be treated as masked assignments.
- Avoid dependencies between loop iterations or at the least, avoid read-after-write dependencies.
- Try to use array notations instead of the use of pointers. C programs in particular impose very few restrictions on the use of pointers; aliased pointers may lead to unexpected dependencies. Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Wherever possible, use the loop index directly in array subscripts instead of incrementing a separate counter for use as an array address.
- Access memory efficiently:

  - Favor inner loops with unit stride.
  - Minimize indirect addressing.
  - Align your data to 16-byte boundaries (for Intel® SSE instructions).
- Choose a suitable data layout with care. Most multimedia extension instruction sets are rather sensitive to alignment. The data movement instructions of Intel® SSE, for example, operate much more efficiently on data that is aligned at a 16-byte boundary in memory. Therefore, the success of a vectorizing compiler also depends on its ability to select an appropriate data layout which, in combination with code restructuring (like loop peeling), results in aligned memory accesses throughout the program.
- Use aligned data structures: Data structure alignment is the adjustment of any data object in relation with other objects.

  You can use the declaration `__declspec(align)`.

  > **Caution** Use this hint with care. Incorrect usage of aligned data movements result in an exception when using Intel® SSE.

- Use structure of arrays (SoA) instead of array of structures (AoS): An array is the most common type of data structure that contains a contiguous collection of data items that can be accessed by an ordinal index. You can organize this data as an array of structures (AoS) or as a structure of arrays (SoA). While AoS organization is excellent for encapsulation, it can be a hindrance for use of vector processing. To make vectorization of the resulting code more effective, you can also select appropriate data structures.

Dynamic Alignment Optimizations

Dynamic alignment optimizations can improve the performance of vectorized code, especially for long trip count loops. Disabling such optimizations can decrease performance, but it may improve bitwise reproducibility of results, factoring out data location from possible sources of discrepancy.

To enable or disable dynamic data alignment optimizations, specify the option `Qopt-dynamic-align[-]` (Windows) or `[no-]qopt-dynamic-align[-]` (Linux).

Use Aligned Data Structures

Data structure alignment is the adjustment of any data object with relation to other objects. The Intel® C++ Compiler Classic may align individual variables to start at certain addresses to speed up memory access. Misaligned memory accesses can incur large performance losses on certain target processors that do not support them in hardware.

Alignment is a property of a memory address, expressed as the numeric address modulo of powers of two. In addition to its address, a single datum also has a size. A datum is called 'naturally aligned' if its address is aligned to its size, otherwise it is called 'misaligned'. For example, an 8-byte floating-point datum is naturally aligned if the address used to identify it is aligned to eight (8).

A data structure is a way of storing data in a computer so that it can be used efficiently. Often, a carefully chosen data structure allows a more efficient algorithm to be used. A well-designed data structure allows a variety of critical operations to be performed, using as little resources (execution time and memory space) as possible.

```
struct MyData{
   short   Data1;
   short   Data2;
   short   Data3;
};
```

In the example data structure above, if the type `short` is stored in two bytes of memory then each member of the data structure is aligned to a boundary of two bytes. `Data1` would be at offset `0`, `Data2` at offset `2` and `Data3` at offset `4`. The size of this structure is six bytes. The type of each member of the structure usually has a required alignment, meaning that it is aligned on a pre-determined boundary, unless you request otherwise. In cases where the compiler has taken sub-optimal alignment decisions, you can use the declaration `declspec(align(base,offset))`, where `0 <= offset < base` and `base` is a power of two, to allocate a data structure at offset from a certain base.

Consider as an example, that most of the execution time of an application is spent in a loop of the following form:

```
double a[N], b[N];
  ...
for (i = 0; i < N; i++){ a[i+1] = b[i] * 3; }
```

If the first element of both arrays is aligned at a 16-byte boundary, then either an unaligned load of elements from `b` or an unaligned store of elements into `a` must be used after vectorization.

In this instance, peeling off an iteration does not help but you can enforce the alignment shown below. This alignment results in two aligned access patterns after vectorization (assuming an 8-byte size for doubles):

```
__declspec(align(16, 8))  double a[N];
__declspec(align(16, 0))  double b[N];
/* or simply "align(16)"  */
```

If pointer variables are used, the compiler is usually not able to determine the alignment of access patterns at compile time. Consider the following simple `fill()` function:

```
void fill(char *x) {
  int i;
  for (i = 0; i < 1024; i++){ x[i] = 1; }
}
```

Without more information, the compiler cannot make any assumption on the alignment of the memory region accessed by the above loop. At this point, the compiler may decide to vectorize this loop using unaligned data movement instructions or, generate the runtime alignment optimization shown here:

```
peel = x & 0x0f;
if (peel != 0) {
  peel = 16 - peel;
  /* runtime peeling loop */
  for (i = 0; i < peel; i++) { x[i] = 1; }
}

/* aligned access */
for (i = peel; i < 1024; i++) { x[i] = 1; }
```

Runtime optimization provides a generally effective way to obtain aligned access patterns at the expense of a slight increase in code size and testing. If incoming access patterns are aligned at a 16-byte boundary, you can avoid this overhead with the hint `__assume_aligned(x, 16);` in the function to convey this information to the compiler.

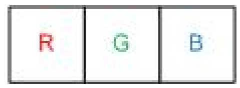
For example, suppose you can introduce an optimization in the case where a block of memory with address `n2` is aligned on a 16-byte boundary. You could use `_assume(n2%16==0)`.

Caution Incorrect use of aligned data movements result in an exception for Intel® SSE.

Use Structure of Arrays Versus Array of Structures

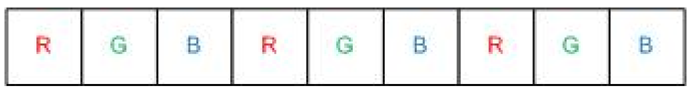
The most common and well-known data structure is the array that contains a contiguous collection of data items, which can be accessed by an ordinal index. This data can be organized as an array of structures (AoS) or as a structure of arrays (SoA). While AoS organization works excellently for encapsulation, for vector processing it works poorly.

You can select appropriate data structures to make vectorization of the resulting code more effective. To illustrate this point, compare the traditional array of structures (AoS) arrangement for storing the `r`, `g`, `b` components of a set of three-dimensional points with the alternative structure of arrays (SoA) arrangement for storing this set.



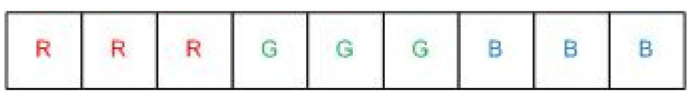
For example, a point structure with data in an AoS arrangement:

```
struct Point{
   float r;
   float g;
   float b;
}
```



For example, a points structure with data in a SoA arrangement:

```
struct Points{
   float* x;
   float* y;
   float* z;
}
```



With the AoS arrangement, a loop that visits all components of an RGB point before moving to the next point exhibits a good locality of reference. This is because all elements in the fetched cache lines are used. The disadvantage of the AoS arrangement is that each individual memory reference in such a loop exhibits a non-unit stride, which, in general, adversely affects vector performance. Furthermore, a loop that visits only one component of all points exhibits less satisfactory locality of reference because many of the elements in the fetched cache lines remain unused.

With the SoA arrangement, the unit-stride memory references are more amenable to effective vectorization and still exhibit good locality of reference within each of the three data streams. Consequently, an application that uses the SoA arrangement may outperform an application based on the AoS arrangement when compiled with a vectorizing compiler. This performance difference may not be obviously apparent during the early implementation phase.

Before you start vectorization, try out some simple rules:

- Make your data structures vector-friendly.
- Make sure that inner loop indices correspond to the outermost (last) array index in your data (row-major order).
- Use structure of arrays over array of structures.

For instance when dealing with three-dimensional coordinates, use three separate arrays for each component (SoA), instead of using one array of three-component structures (AoS). To avoid dependencies between loops that will eventually prevent vectorization, use three separate arrays for each component (SoA), instead of one array of three-component structures (AoS). When you use the AoS arrangement, each iteration produces one result by computing XYZ, but it can at best use only 75% of the SSE unit because the fourth component is not used. Sometimes, the compiler may use only one component (25%). When you use the SoA arrangement, each iteration produces four results by computing XXXX, YYYY and ZZZZ, using 100% of the SSE unit. A drawback for the SoA arrangement is that your code will likely be three times as long.

If your original data layout is in AoS format, you may want to consider a conversion to SoA before the critical loop:

- Use the smallest data types that give the needed precision to maximize potential SIMD width. (If only 16-bits are needed, using a `short` rather than an `int` can make the difference between 8-way or four-way SIMD parallelism.)
- Avoid mixing data types to minimize type conversions.
- Avoid operations not supported in SIMD hardware.
- Use all the instruction sets available for your processor. Use the appropriate command line option for your processor type, or select the appropriate IDE option (Windows only):
  - **Project** > **Properties** > **C/C++** > **Code Generation** > **Intel Processor-Specific Optimization**, if your application runs only on Intel® processors.
  - **Project** > **Properties** > **C/C++** > **Code Generation** > **Enable Enhanced Instruction Set**, if your application runs on compatible, non-Intel processors.
- Vectorizing compilers usually have some built-in efficiency heuristics to decide whether vectorization is likely to improve performance. The Intel® C++ Compiler Classic disables vectorization of loops with many unaligned or non-unit stride data access patterns. If experimentation reveals that vectorization improves performance, you can override this behavior using the `#pragma vector always` hint before the loop. The compiler vectorizes any loop regardless of the outcome of the efficiency analysis (provided that vectorization is safe).

See Also

__declspec(align)

Vectorization and Loops

Loop Constructs

qopt-dynamic-align, Qopt-dynamic-align
compiler option

Use Automatic Vectorization

Automatic vectorization is supported on IA-32and Intel® 64 architectures. The information below will guide you in setting up the auto-vectorizer.

Vectorization Speedup

Where does the vectorization speedup come from? Consider the following sample code, where `a`, `b`, and `c` are integer arrays:

```
for (i=0;i<=MAX;i++)
   c[i]=a[i]+b[i];
```

If vectorization is not enabled, and you compile using the `O1`, `-no-vec-` (Linux), or `/Qvec-` (Windows) option, the compiler processes the code with unused space in the SIMD registers, even though each register can hold three additional integers. If vectorization is enabled (compiled using `O2` or higher options), the compiler may use the additional registers to perform four additions in a single instruction. The compiler looks for vectorization opportunities whenever you compile at default optimization (`O2`) or higher.

NOTE
This option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

Tip
This tip is only for the Intel®C++ (icc) Classic Compiler. To allow comparisons between vectorized and non-vectorized code, disable vectorization using the `-no-vec` (Linux or macOS) or `/Qvec-` (Windows) option; enable vectorization using the `O2` option.

To learn if a loop was vectorized or not, enable generation of the optimization report using the options `qopt-report=1 qopt-report-phase=vec` (Linux and macOS) or `Qopt-report:1 Qopt-report-phase:vec` (Windows) options. These options generate a separate report in an `*.optrpt` file that includes optimization messages. In Microsoft Visual Studio, the program source is annotated with the report's messages, or you can read the resulting `.optrpt` file using a text editor. A message appears for every loop that is vectorized, for example:

```
icl /Qopt-report:1 /Qopt-report-phase:vec  Multiply.c
Multiply.c(92): (col. 5) remark: LOOP WAS VECTORIZED.
```

The source line number (92 in the above example) refers to either the beginning or the end of the loop.

To get details about the type of loop transformations and optimizations that took place, use the `[Q]opt-report-phase` option by itself or along with the `[Q]opt-report` option.

To get information on if the loop was vectorized using the Microsoft Visual Studio IDE, select **Project** > **Properties** > **C/C++** > **Diagnostics** > **Optimization Diagnostic Level** as **Level 1 (/Qopt-report:1)** and **Optimization Diagnostic Phase** as **Loop Nest Optimization (/Qopt-report-phase:loop)**. To get a diagnostic message for every loop that was not vectorized, with a brief explanation of why the loop was not vectorized, select `/Qopt-report-phase:vec`.

Linux

To evaluate performance enhancement, run `vec_samples`:

1. Source an environment script such as `compilervars.sh` or the `compilervars.csh` in the `<installdir>/bin` directory and use the attribute appropriate for the architecture.
2. Navigate to the `<install-dir>\Samples\<locale>\C++\` directory. This application multiplies a vector by a matrix using the following loop:

```
for (j = 0;j < size2; j++) { b[i] += a[i][j] * x[j]; }
```

3. Build and run the application, first without enabling auto-vectorization. The default `O2` optimization enables vectorization, so you need to disable it with a separate option.

```
icc -O2 -no-vec  Multiply.c -o NoVectMult
./NoVectMult
```

4. Build and run the application, this time with auto-vectorization.

```
icx -O2 -qopt-report=3 -vec Multiply.c -o VectMult
./VectMult
```

Windows

To evaluate performance enhancement, run `vec_samples`:

1. Under the **Start** menu item for your product, select an icon under **Compiler and Performance Libraries** > **Command Prompt with Intel Compiler** for Classic Compilers or

2. Navigate to the `<install-dir>\Samples\<locale>\C++\`directory. On Windows, unzip the sample project `vec_samples.zip` to a writable directory. This small application multiplies a vector by a matrix using the following loop:

```
for (j = 0;j < size2; j++) { b[i] += a[i][j] * x[j]; }
```

3. Build and run the application, first without enabling auto-vectorization. The default `O2` optimization enables vectorization, so you need to disable it with a separate option.

```
icl /O2 /Qvec- Multiply.c /FeNoVectMult
NoVectMult
```

4. Build and run the application, this time with auto-vectorization.

```
icl /O2 /Qopt-report:3 /Qvec Multiply.c /FeVectMult
VectMult
```

When you compare the timing of the two runs, you may see that the vectorized version runs faster. The time for the non-vectorized version is only slightly faster than would be obtained by compiling with the `O1` option.

Obstacles to Vectorization

The following issues do not always prevent vectorization, but frequently cause the compiler to decide that vectorization would not be worthwhile.

- **Non-contiguous memory access:** Four consecutive integers or floating-point values, or two consecutive doubles, may be loaded directly from memory in a single SSE instruction. But if the four integers are not adjacent, they must be loaded separately using multiple instructions, which is considerably less efficient. The most common examples of non-contiguous memory access are loops with non-unit stride or with indirect addressing, shown in the examples below. The compiler rarely vectorizes these loops, unless the amount of computational work is larger compared to the overhead from non-contiguous memory access.

```
// arrays accessed with stride 2
for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

// inner loop accesses a with stride SIZE
for (int j=0; j<SIZE; j++)  {
  for (int i=0; i<SIZE; I++)   b[i] += a[i][j] * x[j];
}

// indirect addressing of x using index array
  for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[index[i]];
```

  The typical message from the vectorization report is: `vectorization possible but seems inefficient`, although indirect addressing may also result in the following report: `existence of vector dependence`.
- **Data dependencies:** Vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation.
  - The simplest case is when data elements that are written (stored to) do not appear in any other iteration of the individual loop. In this case, all the iterations of the original loop are independent of each other, and can be executed in any order, without changing the result. The loop may be safely executed using any parallel method, including vectorization.
  - When a variable is written in one iteration and read in a subsequent iteration, there is a read-after-write dependency, also known as a flow dependency, for example:

```
A[0]=0;
for (j=1; j<MAX; j++)  A[j]=A[j-1]+1;
          // this is equivalent to:
   A[1]=A[0]+1;
```

```
   A[2]=A[1]+1;
   A[3]=A[2]+1;
   A[4]=A[3]+1;
```

The value of `j` is propagated to all `A[j]`. This cannot safely be vectorized: if the first two iterations are executed simultaneously by a SIMD instruction, the value of `A[1]` is used by the second iteration before it has been calculated by the first iteration.

- When a variable is read in one iteration and written in a subsequent iteration, this is a write-after-read dependency, also known as an anti-dependency, for example:

```
for (j=1; j<MAX; j++)  A[j-1]=A[j]+1;
          // this is equivalent to:
   A[0]=A[1]+1;
   A[1]=A[2]+1;
   A[2]=A[3]+1;
   A[3]=A[4]+1;
```

This write-after-read dependency is not safe for general parallel execution, since the iteration with the write may execute before the iteration with the read. No iteration with a higher value of `j` can complete before an iteration with a lower value of `j`, and so vectorization is safe (it gives the same result as non-vectorized code).

The following example may not be safe, since vectorization might cause some elements of `A` to be overwritten by the first SIMD instruction before being used for the second SIMD instruction.

```
for (j=1; j<MAX; j++)  {
  A[j-1]=A[j]+1;
}

   // this is equivalent to:
   A[0]=A[1]+1;
   A[1]=A[2]+1;
   A[2]=A[3]+1;
   A[3]=A[4]+1;
```

- Read-after-read situations are not really dependencies, and do not prevent vectorization or parallel execution. If a variable is unwritten, it does not matter how often it is read.
- Write-after-write, or output dependencies, where the same variable is written to in more than one iteration, are generally unsafe for parallel execution, including vectorization.
- One important exception that contains all of the above types of dependency is:

```
sum=0;
for (j=1; j<MAX; j++)  sum = sum + A[j]*B[j]
```

Although `sum` is both read and written in every iteration, the compiler recognizes such reduction idioms, and is able to vectorize them safely. The loop in the first example was another example of a reduction, with a loop-invariant array element in place of a scalar.

These types of dependencies between loop iterations are sometimes known as loop-carried dependencies.

The above examples are of proven dependencies. The compiler cannot safely vectorize a loop if there is even a potential dependency. For example:

```
for (i = 0; i < size; i++) { c[i] = a[i] * b[i]; }
```

In the above example, the compiler needs to determine whether, for some iteration `i`, `c[i]` might refer to the same memory location as `a[i]` or `b[i]` for a different iteration. Such memory locations are sometimes said to be aliased. For example, if `a[i]` pointed to the same memory location as `c[i-1]`, there would be a read-after-write dependency. If the compiler cannot exclude this possibility, it will not vectorize the loop unless you provide the compiler with hints.

Help the Compiler Vectorize

Sometimes the compiler has insufficient information to decide to vectorize a loop. There are several ways to provide additional information to the compiler:

- **Pragmas:**
  - `#pragma ivdep`: may be used to tell the compiler that it may safely ignore any potential data dependencies. (The compiler will not ignore proven dependencies). Use of this pragma when there are dependencies may lead to incorrect results.

    There are cases where the compiler cannot tell by a static dependency analysis that it is safe to vectorize. Consider the following loop:

```
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) { cp_a[i] = cp_b[i]; }
}
```

Without more information, a vectorizing compiler must conservatively assume that the memory regions accessed by the pointer variables `cp_a` and `cp_b` may (partially) overlap, which can cause potential data dependencies that prohibit straightforward conversion of this loop into SIMD instructions. At this point, the compiler may decide to keep the loop serial or generate a runtime test for overlap, where the loop in the true-branch can be converted into SIMD instructions:

```
if (cp_a + n < cp_b || cp_b + n < cp_a)
          /* vector loop */
  for (int i = 0; i < n; i++) cp_a[i] = cp_b [I];
  else
          /* serial loop */
  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
```

Runtime data-dependency testing provides a way to exploit implicit parallelism in C or C++ code at the expense of a slight increase in code size and testing overhead. If the function copy is only used in specific ways, you can help the compiler:

- If the function is mainly used for small values of `n` or for overlapping memory regions, you can prevent vectorization and the corresponding runtime overhead by inserting a `#pragma novector` hint before the loop.
- Conversely, if the loop is guaranteed to operate on non-overlapping memory regions, you can provide this information to the compiler by means of a `#pragma ivdep` hint before the loop. This tells the compiler that conservatively assumed data dependencies that prevent vectorization can be ignored and results in vectorization of the loop without runtime data-dependency testing.

```
#pragma ivdep
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) { cp_a[i] = cp_b[i]; }
}
```

NOTE You can also use the `restrict` keyword.

- `#pragma loop count (n)`: gives the typical trip count of the loop. This helps the compiler decide if vectorization is worthwhile, or if it should generate alternative code paths for the loop.
- `#pragma vector always`: asks the compiler to vectorize the loop.
- `#pragma vector align`: asserts that data within the following loop is aligned (to a 16-byte boundary, for Intel® SSE instruction sets).
- `#pragma novector`: asks the compiler not to vectorize a particular loop.
- `#pragma vector nontemporal`: gives a hint to the compiler that data will not be reused, and to use streaming stores that bypass cache.

- **Keywords:** The `restrict` keyword is used to assert that the memory referenced by a pointer is not aliased. The keyword requires the use of the `[Q]restrict` or `[Q]std=c99` compiler option. The example under `#pragma ivdep` above can also be handled using the `restrict` keyword.

  You may use the `restrict` keyword in the declarations of `cp_a` and `cp_b`, as shown below, to inform the compiler that each pointer variable provides exclusive access to a certain memory region. The `restrict` qualifier in the argument list lets the compiler know that there are no other aliases to the memory where the pointers point. The pointer where it is used provides the only means of accessing the memory in the scope where the pointers live. Even if the code gets vectorized without the `restrict` keyword, the compiler checks for aliasing at runtime, if the `restrict` keyword was used. You may have to use an extra compiler option, such as `[Q]restrict` option for the Intel® C++ Classic Compiler.

```
void copy(char * __restrict cp_a, char * __restrict cp_b, int n) {
  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
}
```

  This method is best used when the exclusive access property holds for the pointer variables in your code with many loops, because it avoids annotating each of the vectorizable loops individually. Both the loop-specific `#pragma ivdep` hint, and the pointer variable-specific `restrict` hint must be used with care because incorrect usage may change the semantics intended in the original program.

  Another example is the following loop that may also not get vectorized because of a potential aliasing problem between pointers `a`, `b`, and `c`:

```
void add(float *a, float *b, float *c) {
  for (int i=0; i<SIZE; i++) { c[i] += a[i] + b[i]; }
}
```

  If the `restrict` keyword is added to the parameters, the compiler assumes that you will not access the memory in question with any other pointer and vectorize the code properly:

```
// let the compiler know, the pointers are safe with restrict
void add(float * __restrict a, float * __restrict b, float * __restrict c) {
  for (int i=0; i<SIZE; i++) { c[i] += a[i] + b[i]; }
}
```

  The down-side of using `restrict` is that not all compilers support this keyword, so your source code may lose portability. If you care about source code portability you may want to consider using the `[Q]ansi-alias` compiler option instead. Compiler options work globally, so you must make sure they do not cause harm to other code fragments.
- **Options/switches:** You can use options to enable different levels of optimizations to achieve automatic vectorization:
  - **Interprocedural optimization (IPO):** Enable IPO using the `[Q]ip` option within a single source file, or using `[Q]ipo` option across source files. You provide the compiler with additional information (trip counts, alignment, or data dependencies) about a loop. Enabling IPO may also allow inlining of function calls.
  - **Disambiguation of pointers and arrays:** Use the options `-fno-alias` (Linux or macOS) or `/Oa` (Windows) to assert there is no aliasing of memory references, that is, the same memory location is not accessed via different arrays or pointers. Other options make more limited assertions, for example, `-fargument-noalias` (Linux or macOS) or `/Qalias-args-` (Windows) asserts that function arguments cannot alias each other (they cannot overlap).

    The `/Qansi-alias` (`-fargument-alias`) options allow the compiler to assume strict adherence to the aliasing rules in the ISO C standard. Use these options responsibly; if you use these options when memory is aliased it may lead to incorrect results.

NOTE When you specify the `[Q]ansi-alias` option, the ansi-alias checker is enabled by default. To disable the ansi-alias checker, you must specify `-no-ansi-alias-check` (Linux and macOS) or `/Qansi-alias-check` (Windows).

Use the `[Q]ansi-alias-check` option to enable the ansi-alias checker. The ansi-alias checker checks the source code for potential violations of ANSI aliasing rules and disables unsafe optimizations related to the code for those statements that are identified as potential violations.

- **High-level optimizations (HLO):** Enable HLO with option `O3`. This enables additional loop optimizations that make it easier for the compiler to vectorize the transformed loops. The HLO report, obtained using the `[Q]opt-report-phase[:]loop` option or the corresponding IDE selection, tells you if some of these additional transformations occurred.

See Also

ansi-alias, Qansi-alias compiler option
ansi-alias-check, Qansi-alias-check compiler option
qopt-report, Qopt-report compiler option
qopt-report-phase, Qopt-report-phase compiler option

Vectorization and Loops

This topic provides more information on the interaction between the auto-vectorizer and loops.

Interactions with Loop Parallelization

Combine the [Q]parallel and [Q]x options to instruct the Intel® C++ Compiler to attempt both Automatic Parallelization and automatic loop vectorization in the same compilation.

NOTE
Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

NOTE
This option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

In most cases, the compiler will consider outermost loops for parallelization and innermost loops for vectorization. If deemed profitable, however, the compiler may even apply loop parallelization and vectorization to the same loop.

See Programming with Auto-parallelization and Programming Guidelines for Vectorization.

In some rare cases, a successful loop parallelization (either automatically or by means of OpenMP directives) may affect the messages reported by the compiler for a non-vectorizable loop in a non-intuitive way; for example, in the cases where the `/Qopt-report:2 /Qopt-report-phase:vec` (Windows) or `-qopt-report=2 -qopt-report-phase=vec` (Linux and macOS) options indicate that loops were not successfully vectorized.

Types of Vectorized Loops

For integer loops, the 128-bit Intel® Streaming SIMD Extensions (Intel® SSE) and the Intel® Advanced Vector Extensions (Intel® AVX) provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types, with limited support for the 64-bit integer data type.

Vectorization may proceed if the final precision of integer wrap-around arithmetic is preserved. A 32-bit shift-right operator, for instance, is not vectorized in 16-bit mode if the final stored value is a 16-bit integer. Also, note that because the Intel® SSE and the Intel® AVX instruction sets are not fully orthogonal (shifts on byte operands, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, Intel® SSE provides SIMD instructions for the following arithmetic operators:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

Additionally, Intel® SSE provide SIMD instructions for the binary `MIN` and `MAX` and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions `SIN`, `COS`, and `TAN`) are supported in software in a vector mathematical run-time library that is provided with the compiler.

To be vectorizable, loops must be:

- **Countable:** The loop trip count must be known at entry to the loop at runtime, though it need not be known at compile time (that is, the trip count can be a variable but the variable must remain constant for the duration of the loop). This implies that exit from the loop must not be data-dependent.
- **Single entry and single exit:** as is implied by stating that the loop must be countable. Consider the following example of a loop that is not vectorizable, due to a second, data-dependent exit:

```
void no_vec(float a[], float b[], float c[]){
  int i = 0.;
  while (i < 100) {
    a[i] = b[i] * c[i];
    //  this is a data-dependent exit condition:
    if (a[i] < 0.0)
    break;
    ++i;
  }
}
```

  Which results in the following message when the code is compiled:

```
> icc -c -O2 -qopt-report=2 -qopt-report-phase=vec two_exits.cpp
two_exits.cpp(4) (col. 9): remark: loop was not vectorized: nonstandard loop is not a
vectorization candidate.
```

- **Contain straight-line code:** SIMD instruction perform the same operation on data elements from multiple iterations of the original loop, therefore, it is not possible for different iterations to have different control flow; that is, they must not branch. It follows that `switch` statements are not allowed. However, `if` statements are allowed if they can be implemented as masked assignments, which is usually the case. The calculation is performed for all data elements but the result is stored only for those elements for which the mask evaluates to true. Consider the following example that may be vectorized:

```
#include <math.h>
void quad(int length, float *a, float *b, float *c, float *restrict x1, float *restrict x2) {
  for (int i=0; i<length; i++) {
    float s = b[i]*b[i] - 4*a[i]*c[i];
    if ( s >= 0 ) {
      s = sqrt(s) ;
      x2[i] = (-b[i]+s)/(2.*a[i]);
```

```
      x1[i] = (-b[i]-s)/(2.*a[i]);
    } else {
      x2[i] = 0.;
      x1[i] = 0.;
    }
  }
}
```

Which results in the following message when the code is compiled:

```
> icc -c -restrict -qopt-report=2 -qopt-report-phase=vec quad.cpp
quad5.cpp(5) (col. 3): remark: LOOP WAS VECTORIZED.
```

- **Innermost loop of a nest:** The only exception is if an original outer loop is transformed into an inner loop as a result of some other prior optimization phase, such as unrolling, loop collapsing or interchange, or an original outermost loop is transformed to an innermost loop due to loop materialization.
- **Without function calls:** Even a `print` statement is sufficient to prevent a loop from getting vectorized. The vectorization report message is typically: non-standard loop is not a vectorization candidate. The two major exceptions are for intrinsic math functions and for functions that may be inlined.

Intrinsic math functions are allowed, because the compiler runtime library contains vectorized versions of these functions. See below for a list of these functions; most exist in both float and double versions:

- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atan2`
- `atanh`
- `cbrt`
- `ceil`
- `cos`
- `cosh`
- `erf`
- `erfc`
- `erfinv`
- `exp`
- `exp2`
- `fabs`
- `floor`
- `fmax`
- `fmin`
- `log`
- `log2`
- `log10`
- `pow`
- `round`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`
- `trunc`

The loop in the following example may be vectorized because `sqrt()` is vectorizable and `func()` gets inlined. Inlining is enabled at default optimization for functions in the same source file. An inlining report may be obtained by setting the options `/Qopt-report:2 /Qopt-report-phase:ipo` (Windows) or `-qopt-report=2 -qopt-report-phase=ipo` (Linux).

```
float func(float x, float y, float xp, float yp) {
  float denom;
  denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
  denom = 1./sqrtf(denom);
  return denom;
}

float trap_int(float y, float x0, float xn, int nx, float xp, float yp) {
  float x, h, sumx;
  int i;
  h = (xn-x0) / nx;
  sumx = 0.5*( func(x0,y,xp,yp) + func(xn,y,xp,yp) );
  for (i=1;i<nx;i++) {
    x = x0 + i*h;
    sumx = sumx + func(x,y,xp,yp);
  }
  sumx = sumx * h;
  return sumx;
}
```

Which results in the following message when the code is compiled:

```
> icc -c -qopt-report=2 -qopt-report-phase=vec trap_integ.c
trap_int.c(16) (col. 3): remark: LOOP WAS VECTORIZED.
```

Statements in the Loop Body

The vectorizable operations are different for floating-point and integer data.

Integer Array Operations

The statements within the loop body may contain *char*, *unsigned char*, *short*, *unsigned short*, *int*, and *unsigned int*. Calls to functions such as `sqrt` and `fabs` are also supported. Arithmetic operations are limited to addition, subtraction, bitwise `AND`, `OR`, and `XOR` operators, division (via run-time library call), multiplication, `min`, and `max`. You can mix data types but this may potentially cost you in terms of lowering efficiency. Some example operators where you can mix data types are multiplication, shift, or unary operators.

Other Operations

No statements other than the preceding floating-point and integer operations are allowed. In particular, note that the special `__m64__m128`, and `__m256` data types are not vectorizable. The loop body cannot contain any function calls. Use of Intel® SSE intrinsics (for example, `_mm_add_ps`) or Intel® AVX intrinsics (for example, `_mm256_add_ps`) are not allowed.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

Automatic Parallelization
Programming with Auto-parallelization

Programming Guidelines for Vectorization
qopt-report-phase, Qopt-report-phase
compiler option
x, Qx compiler option
parallel, Qparallel compiler option

Loop Constructs

Loops can be formed with the usual `for` and `while` constructs. Loops must have a single entry and a single exit to be vectorized. The following examples illustrate loop constructs that can and cannot be vectorized. The non-vectorizable structure example shows a loop that cannot be vectorized because of the inherent potential for an early exit from the loop.

Vectorizable structure:

```
void vec(float a[], float b[], float c[]) {
  int i = 0;
  while (i < 100) {
// The if branch is inside body of loop.
    a[i] = b[i] * c[i];
    if (a[i] < 0.0)
        a[i] = 0.0;
        i++;
  }
}
```

Non-vectorizable structure:

```
void no_vec(float a[], float b[], float c[]) {
  int i = 0;
  while (i < 100) {
    if (a[i] < 50)
// The next statement is a second exit
// that allows an early exit from the loop.
      break;
    ++i;
  }
}
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable and the number of iterations must be expressed as one of the following:

- A constant.
- A loop invariant term.
- A linear function of outermost loop indices.

In the case where a loops exit depends on computation, the loops are not countable. The examples below show loop constructs that are countable and non-countable. The non-countable loop example demonstrates a loop construct that is non-countable due to dependency loop variant count value.

Countable loop, example one:

```
void cnt1(float a[], float b[], float c[],
          int n, int lb) {
// Exit condition specified by "N-1b+1"
  int cnt=n, i=0;
  while (cnt >= lb) {
```

```
// lb is not affected within loop.
    a[i] = b[i] * c[i];
    cnt--;
    i++;
  }
}
```

Countable loop, example two:

```
void vec(float a[], float b[], float c[]) {
  int i = 0;
  while (i < 100) {
// The if branch is inside body of loop.
    a[i] = b[i] * c[i];
    if (a[i] < 0.0)
        a[i] = 0.0;
        i++;
  }
}
```

Non-countable loop:

```
void no_cnt(float a[], float b[], float c[]) {
  int i=0;
// Iterations dependent on a[i].
  while (a[i]>0.0) {
    a[i] = b[i] * c[i];
    i++;
  }
}
```

Strip-mining and Cleanup

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encoding of loops, as well as a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- By increasing the temporal and spatial locality in the data cache if the data is reusable in different passes of an algorithm.
- By reducing the number of iterations of the loop by a factor of the length of each vector, or number of operations being performed per SIMD operation. With the Intel® Streaming SIMD Extensions (Intel® SSE), the vector or strip-length is reduced by four times: four floating-point data items per single Intel® SSE single-precision floating-point SIMD operation are processed.

First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine.

The compiler automatically strip-mines your loop and generates a cleanup loop. For example, assume the compiler attempts to strip-mine the loop before vectorization. After vectorization, the compiler might handle the strip mining and loop cleaning by restructuring the loop.

Before vectorization:

```
i=0;
while(i<n) {
  // Original loop code
  a[i]=b[i]+c[i];
  ++i;
}
```

After vectorization:

```
// The vectorizer generates the following two loops
i=0;
while(i<(n-n%4)) {
  // Vector strip-mined loop
  // Subscript [i:i+3] denotes SIMD execution
  a[i:i+3]=b[i:i+3]+c[i:i+3];
  i=i+4;
}
while(i<n) {
  // Scalar clean-up loop
  a[i]=b[i]+c[i];
  ++i;
}
```

Loop Blocking

It is possible to treat loop blocking as strip-mining in two or more dimensions. Loop blocking is a useful technique for memory performance optimization. The main purpose of loop blocking is to eliminate as many cache misses as possible. This technique transforms the memory domain into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, maximizing data reuse.

Consider the following example, loop blocking allows arrays *A* and *B* to be blocked into smaller rectangular chunks so that the total combined size of two blocked (*A* and *B*) chunks is smaller than cache size, which can improve data reuse.

The transformed loop after blocking example illustrates loop blocking the `add` function (from the original loop example). In order to benefit from this optimization, you might have to increase the cache size.

Original loop:

```
#include <time.h>
#include <stdio.h>
#define MAX 7000

void add(int a[][MAX], int b[][MAX]);
int main() {
int i, j;
int A[MAX][MAX];
int B[MAX][MAX];
time_t start, elaspe;
int sec;

//Initialize array
for(i=0;i<MAX;i++) {
  for(j=0;j<MAX; j++) {
    A[i][j]=j;
    B[i][j]=j;
  }
}

 start= time(NULL);
 add(A, B);
 elaspe=time(NULL);
 sec = elaspe - start;
 printf("Time %d",sec); //List time taken to complete add function
}
```

```
void add(int a[][MAX], int b[][MAX]) {
 int i, j;
 for(i=0;i<MAX;i++) {
  for(j=0; j<MAX;j++ {
     a[i][j] = a[i][j] + b[j][i]; //Adds two matrices
    }
  }
}
```

Transformed loop after blocking:

```
#include <stdio.h>
#include <time.h>
#define MAX 7000
void add(int a[][MAX], int b[][MAX]);

int main() {
  #define BS 8  //Block size is selected as the loop-blocking factor.
  int i, j;
  int A[MAX][MAX];
  int B[MAX][MAX];
  time_t start, elaspe;
  int sec;

//initialize array
for(i=0;i<MAX;i++) {
  for(j=0;j<MAX;j++) {
    A[i][j]=j;
    B[i][j]=j;
  }
}
start= time(NULL);

add(A, B);
elapse=time(NULL);
sec = elapse - start;
printf("Time %d",sec); //Display time taken to complete loopBlocking function
}

void add(int a[][MAX], int b[][MAX]) {
  int i, j, ii, jj;
  for(i=0;i<MAX;i+=BS) {
   for(j=0; j<MAX;j+=BS) {
     for(ii=i; ii<i+BS; ii++) {   //outer loop
       for(jj=j;jj<j+BS; jj++) {  //Array B experiences one cache miss
                                  //for every iteration of outer loop
         a[ii][jj] = a[ii][jj] + b[jj][ii]; //Add the two arrays
        }
      }
    }
  }
}
```

Loop Interchange and Subscripts with Matrix Multiply

Loop interchange is often used for improving memory access patterns. Matrix multiplication is commonly written as shown in the typical matrix multiplication example.

The use of *B*(*K*,*J*) is not a stride-1 reference and therefore will not be vectorized efficiently.

If the loops are interchanged, all the references become stride-1 as shown in the matrix multiplication with stride-1 example.

Typical matrix multiplication:

```
void matmul_slow(float *a[], float *b[], float *c[]) {
  int N = 100;
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Matrix multiplication with stride-1:

```
void matmul_fast(float *a[], float *b[], float *c[]) {
  int N = 100;
  for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++)
      for (int j = 0; j < N; j++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Interchanging is not always possible because of dependencies, which can lead to different results.

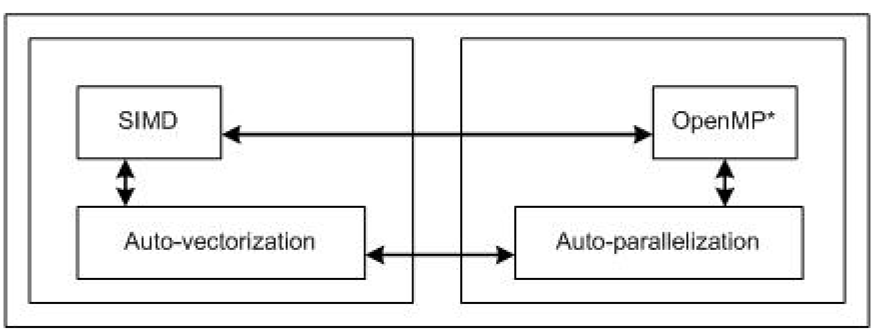
Explicit Vector Programming

This section contains information about explicit vector programming.

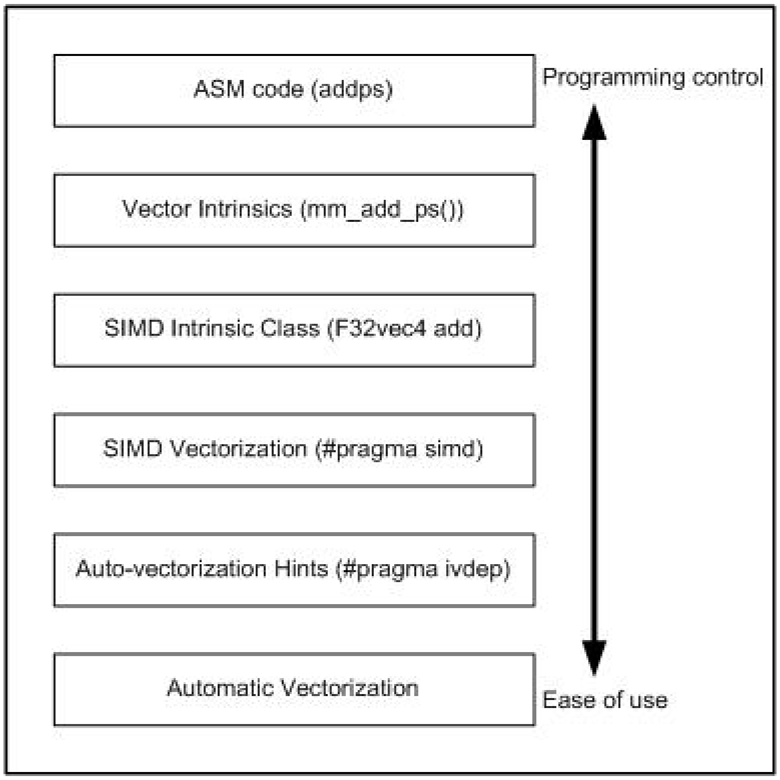
User-mandated or SIMD Vectorization

User-mandated or SIMD vectorization supplements automatic vectorization just like OpenMP parallelization supplements automatic parallelization. The following figure illustrates this relationship. User-mandated vectorization is implemented as a single-instruction-multiple-data (SIMD) feature and is referred to as SIMD vectorization.

NOTE
The SIMD vectorization feature is available for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.



The following figure illustrates how SIMD vectorization is positioned among various approaches that you can take to generate vector code that exploits vector hardware capabilities. The programs written with SIMD vectorization are very similar to those written using auto-vectorization hints. You can use SIMD vectorization to minimize the number of code changes that you may have to go through in order to obtain vectorized code.



SIMD vectorization uses the `#pragma omp simd` pragma to effect loop vectorization. You must add this pragma to a loop and recompile to vectorize the loop using the option `-qopenmp-simd` (Linux and macOS) or `Qopenmp-simd` (Windows).

Consider an example in C++ where the function `add_floats()` uses too many unknown pointers for the compiler's automatic runtime independence check optimization to kick in. You can give a data dependence assertion using the auto-vectorization hint via `#pragma ivdep` and let the compiler decide whether the auto-vectorization optimization should be applied to the loop. Or you can now enforce vectorization of this loop by using `#pragma omp simd` .

Vectorization without #pragma omp simd:

```
[D:/simd] cat example1.c
void add_floats(float *a, float *b, float *c, float *d, float *e, int n) {
 int i;
 for (i=0; i<n; i++){
  a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
 }
}
```

```
[D:/simd] icl -nologo -c -Qopt-report2 -Qopt-report-file=stderr -Qopt-report-phase=vec -Qopenmp-simd example1.c
example1.c

Begin optimization report for: add_floats(float *, float *, float *, float *, float *, int)

    Report from: Vector optimizations [vec]

LOOP BEGIN at C:\Users\test\run\example1.c(3,2)
```

```
   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First
dependence is shown below. Use level 5 report for details
   remark #15346: vector dependence: assumed FLOW dependence between a[i] (4:3) and b[i] (4:3)
LOOP END

LOOP BEGIN at C:\Users\test\run\example1.c(3,2)
<Remainder>
LOOP END
===========================================================================
```

Vectorization with #pragma omp simd:

```
[D:/simd] cat example1.c
void add_floats(float *a, float *b, float *c, float *d, float *e, int n) {
    int i;
    #pragma omp simd
    for (i=0; i<n; i++){
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
        }
}
```

```
[D:/simd] icl -nologo -c -Qopt-report2 -Qopt-report-file=stderr -Qopt-report-phase=vec -Qopenmp-
simd example1.c
example1.c

Begin optimization report for: add_floats(float *, float *, float *, float *, float *, int)

    Report from: Vector optimizations [vec]

LOOP BEGIN at C:\iUsers\test\run\example1.c(4,2)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at C:\iUsers\test\run\example1.c(4,2)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at C:\iUsers\test\run\example1.c(4,2)
<Alternate Alignment Vectorized Loop>
LOOP END

LOOP BEGIN at C:\iUsers\test\run\example1.c(4,2)
<Remainder loop for vectorization>
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at C:\iUsers\test\run\example1.c(4,2)
<Remainder loop for vectorization>
LOOP END
===========================================================================
```

The difference between using `#pragma omp simd` and auto-vectorization hints is that with `#pragma omp simd`, the compiler generates a warning when it is unable to vectorize the loop. With auto-vectorization hints, actual vectorization is still under the discretion of the compiler, even when you use the `#pragma vector always` hint.

`#pragma omp simd` has optional clauses to guide the compiler on how vectorization must proceed. Use these clauses appropriately so that the compiler obtains enough information to generate correct vector code. For more information on the clauses, see the `#pragma omp simd` description.

Additional Semantics

Note the following points when using the omp simd pragma.

- A variable may belong to zero or one of the following: private, linear, or reduction.
- Within the vector loop, an expression is evaluated as a vector value if it is private, linear, reduction, or it has a sub-expression that is evaluated to a vector value. Otherwise, it is evaluated as a scalar value (that is, broadcast the same value to all iterations). Scalar value does not necessarily mean loop invariant, although that is the most frequently seen usage pattern of scalar value.
- A vector value may not be assigned to a scalar L-value. It is an error.
- A scalar L-value may not be assigned under a vector condition. It is an error.
- The `switch` statement is not supported.

NOTE
You may find it difficult to describe vector semantics using the SIMD pragma for some auto-vectorizable loops. One example is `MIN`/`MAX` reduction in C since the language does not have `MIN`/`MAX` operators.

Using `vector` Declaration

Consider the following C++ example code with a loop containing the math function, `sinf()`.

NOTE All code examples in this section are applicable for C/C++ on Windows only.

Loop Where the Math Function Is Auto-vectorized

```
[D:/simd] cat example2.c
void vsin(float *restrict a, float *restrict b, int n) {
int i;
for (i=0; i<n; i++) {
  a[i] = sinf(b[i]);
  }
}
```

```
[D:/simd] icl -nologo -c -Qrestrict -Qopt-report2 -Qopt-report-file=stderr -Qopt-report-
phase=vec example2.c
example2.c

Begin optimization report for: vsin(float *restrict, float *restrict, int)

    Report from: Vector optimizations [vec]


LOOP BEGIN at C:\Users\test\run\example2.c(3,1)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at C:\Users\test\run\example2.c(3,1)
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at C:\Users\test\run\example2.c(3,1)
<Alternate Alignment Vectorized Loop>
LOOP END

LOOP BEGIN at C:\Users\test\run\example2.c(3,1)
```

```
<Remainder loop for vectorization>
LOOP END
===========================================================================
```

When you compile the above code, the loop with `sinf()` function is auto-vectorized using the appropriate Short Vector Math Library (SVML) library function provided by the Intel® C++ Compiler. The auto-vectorizer identifies the entry points, matches up the scalar math library function to the SVML function, and invokes it.

However, within this loop if you have a call to your function, `foo()`, that has the same prototype as `sinf()`, the auto-vectorizer fails to vectorize the loop because it does not know what `foo()` does unless it is inlined to this call site.

Loop With User-Defined Function Is Not Auto-vectorized

```
[D:/simd] cat example2.c
float foo(float);
void vfoo(float *restrict a, float *restrict b, int n){
    int i;
    for (i=0; i<n; i++){
        a[i] = foo(b[i]);
        }
}
```

```
[D:/simd] icl -nologo -c -Qrestrict -Qopt-report2 -Qopt-report-file=stderr -Qopt-report-
phase=vec example2.c
example2.c

Begin optimization report for: vsin(float *restrict, float *restrict, int)

    Report from: Vector optimizations [vec]

Non-optimizable loops:


LOOP BEGIN at C:\Users\test\run\example2.c(3,1)
   remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.
LOOP END
```

In such cases, you can use the `__declspec(vector)` (Windows) or `__attribute__((vector))` (Linux) declaration to vectorize the loop. All you need to do is add the `vector` declaration to the function declaration, and recompile both the caller and callee code, and the loop and function are vectorized.

Loop with User-Defined Function with SIMD Declaration Is Vectorized

```
[D:/simd] cat example3.c
// foo() and vfoo() do not have to be in the same compilation unit as long
// as both see the same "#pragma omp declare simd" lines.
#pragma omp declare simd
float foo(float);
void vfoo(float *restrict a, float *restrict b, int n){
    int i;
    for (i=0; i<n; i++) { a[i] = foo(b[i]); }
}

float foo(float x) { ... }
```

```
[D:/simd] bash-3.2$ icl -nologo -c -Qopenmp-simd -Qrestrict -Qopt-report1 -Qopt-report-
file=stderr -Qopt-report-phase=vec example3.c
example3.c
```

```
Begin optimization report for: vfoo(float *restrict, float *restrict, int)

    Report from: Vector optimizations [vec]


LOOP BEGIN at C:\Users\test\run\example3.c(7,5)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at C:\Users\test\run\example3.c(7,5)
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at C:\Users\test\run\example3.c(7,5)
<Alternate Alignment Vectorized Loop>
LOOP END

LOOP BEGIN at C:\Users\test\run\example3.c(7,5)
<Remainder loop for vectorization>
LOOP END
===========================================================================

Begin optimization report for: foo.._simdsimd3__xmm4nv(float)

    Report from: Vector optimizations [vec]

remark #15347: FUNCTION WAS VECTORIZED with xmm, simdlen=4, unmasked, formal parameter types:
(vector)
===========================================================================

Begin optimization report for: foo.._simdsimd3__xmm4mv(float)

    Report from: Vector optimizations [vec]

remark #15347: FUNCTION WAS VECTORIZED with xmm, simdlen=4, masked, formal parameter types:
(vector)
===========================================================================
```

Restrictions on Using a #pragma omp declare simd Declaration

Vectorization depends on two major factors: hardware and the style of source code. When using the vector declaration, the following features are not allowed:

- Thread creation and joining through , OpenMP `parallel/for/sections/task/target/teams`, and explicit threading API calls.
- Locks, barriers, atomic construct, critical sections (These are allowed inside `#pragma omp ordered simd` blocks).
- Inline ASM code, VM, and Vector Intrinsics (for example, SVML intrinsics).
- Using setjmp, longjmp, SHE and computed `GOTO`.
- EH is not allowed and all vector functions are considered noexcept.
- The `switch` statement (in some cases this may be supported and converted to `if` statements, but this is not reliable).
- The `exit()/abort()` calls.

Non-vector function calls are generally allowed within vector functions but calls to such functions are serialized lane-by-lane and so might perform poorly. Also for SIMD-enabled functions it is not allowed to have side effects except writes by their arguments. This rule can be violated by non-vector function calls, so be careful executing such calls in SIMD-enabled functions.

Formal parameters must be of the following data types:

- (un)signed 8, 16, 32, or 64-bit integer
- 32- or 64-bit floating point
- 64- or 128-bit complex
- A pointer (C++ reference is considered a pointer data type)

See Also

__declspec(vector) declaration

Function Annotations and the SIMD Directive for Vectorization

SIMD-enabled Functions

SIMD-enabled functions (formerly called elemental functions) are a general language construct to express a data parallel algorithm. A SIMD-enabled function is written as a regular C/C++ function, and the algorithm describes the operation on one element, using scalar syntax. The function can then be called as a regular C/C++ function to operate on a single element or it can be called in a data parallel context to operate on many elements.

If you are using SIMD-enabled functions and need to link a compiler object file with an object file from a previous version of the compiler (for example, 13.1), you need to use the `[Q]vecabi` compiler option, specifying the `legacy` keyword.

How SIMD-enabled Functions Work

When you write a SIMD-enabled function, the compiler generates short vector variants of the function that you requested, which can perform your function's operation on multiple arguments in a single invocation. The short vector variant may be able to perform multiple operations as fast as the regular implementation performs a single one by using the vector instruction set architecture (ISA) in the CPU. When a call to a SIMD-enabled function occurs in a SIMD loop or another SIMD-enabled function, the compiler replaces the scalar call with the best fit from the available short-vector variants of the function.

In addition, when invoked from a `pragma omp` construct, the compiler may assign different copies of the SIMD-enabled functions to different threads (or workers), executing them concurrently. The result is that your data parallel operation executes on the CPU using both the parallelism available in the multiple cores and the parallelism available in the vector ISA. In other words, if the short vector function is called inside a parallel loop, (a vectorized auto-parallelized loop) you can achieve both vector-level and thread-level parallelism.

Declare a SIMD-enabled Function

You need to use the appropriate syntax from below in your code for the compiler to generate the short vector function:

Linux and macOS

Use the `__attribute__((vector (clauses)))` declaration:

```
__attribute__((vector (clauses))) return_typesimd_enabled_function_name(parameters)
```

Alternately, you can use the following OpenMP pragma, which requires the `[q or Q]openmp` or `[q or Q]openmp-simd` compiler option:

```
#pragma omp declare simd clauses
```

Windows

Use the `__declspec(vector (clauses))` declaration:

```
__declspec(vector (clauses)) return_type simd_enabled_function_name(parameters)
```

The clauses in the vector declaration may be used for achieving better performance by overriding defaults. These clauses at SIMD-enabled function definition declare one or several short vector variants for a SIMD-enabled function. Multiple vector declarations with different set of clauses may be attached to one function in order to declare multiple different short vector variants available for a SIMD-enabled function.

The clauses are defined as follows:

Clause	Definition
processor(cpuid)	Tells the compiler to generate a vector variant using the instructions, the caller/callee interface, and the default vector length selection scheme suitable to the specified processor. Use of this clause is highly recommended, especially for processors with wider vector register support (example: `core_2nd_gen_avx` and newer). `cpuid` takes one of the following values: • `core_4th_gen_avx_tsx` • `core_4th_gen_avx` • `core_3rd_gen_avx` • `core_2nd_gen_avx` • `core_aes_pclmulqdq` • `core_i7_sse4_2` • `atom` • `core_2_duo_sse4_1` • `core_2_duo_ssse3` • `pentium_4_sse3` • `pentium_m` • `pentium_4` • `haswell` • `broadwell` • `skylake` • `skylake_avx512` • `knl` • `knm`
vectorlength(n) / *simdlen(n)* (for `omp declare simd`)	Where `n` is a vector length that is a power of 2, no greater than 32. The *simdlen* clause tells the compiler that each routine invocation at the call site should execute the computation equivalent to `n` times the scalar function execution. When omitted the compiler selects the vector length automatically depending on the routine return value, parameters, and/or the processor clause. When multiple vector variants are called from one vectorization context (for example,

<table>
<tr><th>Clause</th><th>Definition</th></tr>
<tr><td></td><td>two different functions called from the same vector loop), explicit use of identical simdlen values are advised to achieve good performance.</td></tr>
<tr><td>linear(list_item[, list_item...]), where list_item is one of:
param[:step]
val(param[:step])
ref(param[:step])
uval(param[:step])</td><td> The linear clause tells the compiler that for each consecutive invocation of the routine in a serial execution, the value of param is incremented by step, where param is a formal parameter of the specified function or the C++ keyword this. The linear clause can be used on parameters that are either scalar (non-arrays and of non-structured types), pointers, or C++ references. step is a compile-time integer constant expression, which defaults to 1 if omitted. If more than one step is specified for a particular parameter, a compile-time error occurs. Multiple linear clauses will be merged as a union. The meaning of each variant of the clause is as follows: linear(param[:step]): For parameters that are not C++ references: the clause tells the compiler that on each iteration of the loop from which the routine is called the value of the parameter will be incremented by step. The clause can also be used for C++ references for backward compatibility, but it is not recommended.
linear(val(param[:step])): For parameters that are C++ references: the clause tells the compiler that on each iteration of the loop from which the routine is called the referenced value of the parameter will be incremented by step.
linear(uval(param[:step])): For C++ references: means the same as linear(val()). It differs from linear(val()) so if linear(val()) a vector of references is passed to vector variant of the routine but in case of linear(uval()) only one reference is passed (and thus linear(uval()) is better to use in terms of performance).
linear(ref(param[:step])) :For C++ references: means that the reference itself is linear, i.e. the referenced values (that form a vector for calculations) are located sequentially, like in array with the distance between elements equal to step.</td></tr>
<tr><td>uniform(param [, param,]…)</td><td>Where param is a formal parameter of the specified function or the C++ keyword this.</td></tr>
</table>

Clause	Definition
	The *uniform* clause tells the compiler that the values of the specified arguments can be broadcast to all iterations as a performance optimization. It is often useful in generating more favorable vector memory references. An acknowledgment of a *uniform* clause may allow broadcast operations to be hoisted out of the caller loop. Evaluate carefully the performance implications. Multiple uniform clauses are merged as a union.
mask / nomask	The *mask* and *nomask* clauses tell the compiler to generate only masked or unmasked (respectively) vector variants of the routine. When omitted, both masked and unmasked variants are generated. The masked variant is used when the routine is called conditionally.
inbranch / notinbranch	The *inbranch* and *notinbranch* clauses are used with `#pragma omp declare simd`. The *inbranch* clause works the same as the *mask* clause above and the *notinbranch* clause works the same as the *nomask* clause above.

Write the code inside your function using existing C/C++ syntax and relevant built-in functions (see the section on `__intel_simd_lane()` below).

Usage of Vector Function Specifications

You may define several vector variants for one routine with each variant reflecting a possible usage of the routine. Encountering a call, the compiler matches vector variants with actual parameter kinds and chooses the best match. Matching is done by priorities. In other words, if an actual parameter is the loop invariant and the `uniform` clause was specified for the corresponding formal parameter, then the variant with the `uniform` clause has a higher priority. Linear specifications have the following order, from high priority to low: `linear(uval())`, `linear()`, `linear(val())`, `linear(ref())`. Consider the following example loops with the calls to the same routine.

```
// routine prototype
#pragma omp declare simd                           // universal but slowest definition matches
the use in all three loops
#pragma omp declare simd linear(in1) linear(ref(in2)) uniform(mul) // matches the use in the
first loop
#pragma omp declare simd linear(ref(in2))                             // matches the use in the
second and the third loops
#pragma omp declare simd linear(ref(in2)) linear(mul)              // matches the use in the
second loop
#pragma omp declare simd linear(val(in2:2))                           // matches the use in the
third loop
extern int func(int* in1, int& in2, int mul);

int *a, *b, mul, *c;
int *ndx, nn;
...
// loop examples
   for (int i = 0; i < nn; i++) {
       c[i] = func(a + i, *(b + i), mul); // in the loop, the first parameter is changed
```

```
linearly,
                                          // the second reference is changed linearly too
                                          // the third parameter is not changed
   }

   for (int i = 0; i < nn; i++) {
       c[i] = func(&a[ndx[i]], b[i], i + 1); // the value of the first parameter is
unpredictable,
                                             // the second reference is changed linearly
                                             // the third parameter is changed linearly
   }

   #pragma omp simd
   for (int i = 0; i < nn; i++) {
       int k = i * 2;  // during vectorization, private variables are transformed into arrays: k-
>k_vec[vector_length]
       c[i] = func(&a[ndx[i]], k, b[i]); // the value of the first parameter is unpredictable,
                                         // the second reference and value can be considered
linear
                                         // the third parameter has unpredictable value
                                         // (the #pragma simd linear(val(in2:2))) will be chosen
from the two matching variants)
  }
```

SIMD-enabled Functions and C++

You should use SIMD-enabled functions in modern C++ with caution: C++ imposes strict requirements on compilation and execution environments that may not compose well with semantically-rich language extensions such as SIMD-enabled functions. There are three key aspects of C++ that interrelate with SIMD-enabled functions concept: exception handling, dynamic polymorphism, and the C++ type system.

SIMD-enabled Functions and Exception Handling

Exceptions are currently not supported in SIMD contexts: exceptions cannot be thrown and/or caught in SIMD loops and SIMD-enabled functions. Therefore, all SIMD-enabled functions are considered `noexcept` in C++11 terms. This affects not only short vector variants of a function, but its original scalar routine as well. This is enforced when the function is compiled: it is checked against throw construct and against function calls throwing exceptions. It is also enforced when the SIMD-enabled function call is compiled.

SIMD-enabled Functions and Dynamic Polymorphism

Vector attributes can be applied to virtual functions of classes with some limitations during polymorphic virtual function calls. The syntax of vector declarations is the same as for regular SIMD-enabled class methods: attach vector declarations as described above to the method declarations inside the class declaration.

Vector function attributes for virtual methods are inherited. If a vector attribute is specified for an overriding virtual function, it must match that of the overridden function. Even if the virtual method implementation is overridden in a derived class the vector declarations are inherited and applied. A set of vector variants is produced for the override according to vector variants set on parent. This rule also applies when the parent does not have any vector variants. If a virtual method is introduced as non-SIMD-enabled (no vector declarations supplied), it cannot become SIMD-enabled in the derived class even if the derived class contains its own implementation of the virtual method.

Matching vector variants for a virtual method is done by the declared (static) type of an object for which the method is called. The actual (dynamic) type of an object may either coincide with the static type or be inherited from it.

Unlike regular function calls which transfer control to one target function, the call target of a virtual function depends on the dynamic type of the object for which the method is called and accomplished indirectly via the virtual function table of a class. In a single SIMD chunk, the virtual method may be invoked for objects of multiple classes, for example, elements of a polymorphic collection. This requires multiple calls to different targets within a single SIMD chunk. This works as follows:

1. If a SIMD-enabled virtual function call is matched to a variant with a uniform *this* parameter, multiple calls are not needed. The compiler makes an indirect call to the matched vector variant of a virtual method of the object's dynamic class.
2. If a SIMD-enabled virtual function call is matched to a variant with a non-uniform *this* parameter, all objects in a SIMD chunk may still share virtual method implementation. This is checked and a single, indirect call to the matched vector variant of the target virtual method implementation is invoked.
3. Otherwise, lanes sharing virtual call targets are masked-in and a masked vector variant corresponding to the match is invoked in a loop for each unique virtual call target. If a masked variant is not provided for matching a vector variant and a *this* parameter is not declared uniform, the match will be rejected.

The following example illustrates SIMD-enabled virtual functions:

```
struct Base {
#pragma omp declare simd
#pragma omp declare simd uniform(this)
   virtual int process(int);
};

struct Child1 : Base {
   // int process(int); is inherited
};

struct Child11 : Child1 {
  int process(int); // Overrides implementation, inherits vector declarations
};

struct Child2 : Base {
  int process(int); // Overrides implementation, inherits vector declarations
};

int main() {
    int arr[100];
    Base* c2 = new Child2();
    Base* objs[100];
    int res = 0;

// SIMD-enabled virtual function call for uniform object
#pragma omp simd reduction(+:res)
    for (int i = 0; i < 100; i++) {
      res += c2->process(arr[i]);  // Variant with uniform this is matched
                                   // call to vector variant of
                                   // Child2::process() is invoked
   }

// Initialize polymorphic array of objects
    for (int i = 0; i < 100; i++) {
       if (i % 16 < 4) objs[i] = new Base();
       else if (i % 16 < 8) objs[i] = new Child1();
       else if (i % 12 < 12) objs[i] = new Child11();
       else objs[i] = new Child2();
   }

// SIMD-enabled virtual function call for non-uniform objects
```

```
#pragma omp simd reduction(+:res) simdlen(8)
    for (int i = 0; i < 100; i++) {
       res += objs[i]->process(arr[i]);  // Variant with non-uniform this is
                                         // matched
       // Base and Child1 share the same 'process' implementation, so call
       // targets for each even chunk [i*16:i*16+7] are the same even though
       // this pointers are different for all elements of objs[] array.

       // Odd chunks [i*16+8:i*16+15] consist of objects of classes Child11
       // and Child2 and so require calls to their respective implementations
       // of process() virtual functions. Masked vector variant for
       // Child11::process() is called with mask 0b00001111 (lower lanes of a
       // chunk) and masked vector variant for Child2::process() is called
       // with mask 0b11110000 (upper lanes of a chunk).
   }

   return res;
}
```

The following are limitations to SIMD-enabled virtual function support:

- Multiple inheritance, including virtual inheritance, is not supported for classes having SIMD-enabled virtual methods. This is because calls to virtual functions in multiple inheritance cases may be done through special functions called thunks, which adjust the 'this' pointer and/or virtual function table pointer. The current implementation doesn't support thunks for SIMD-enabled virtual calls because in this case thunks should themselves become SIMD-enabled functions that are not implemented.
- It is not possible to get the address of a SIMD-enabled virtual method. Support of SIMD-enabled virtual functions would require additional information, so their binary representation is different. Such cases will not be handled properly by code expecting a regular pointer to the virtual member.

SIMD-enabled Functions and the C++ Type System

Vector attributes are attributes in the C++11 sense and so are not part of a functional type of SIMD-enabled functions. Vector attributes are bound to the function itself, an instance of a functional type. This has the following implications:

- Template instantiations having SIMD-enabled functions as template parameters won't catch vector attributes, so it is impossible to preserve vector attributes in function wrapper templates like `std::bind` which add indirection. This indirection may sometimes be optimized away by compiler and the resulting direct call will have all vector attributes associated.
- There is no way to overload or specialize templates by vector attributes.
- There is no way to write functional traits to capture vector attributes for the sake of template metaprogramming.

The example below depicts various situations where this situation may be observed:

```
template <int f(int)>   // Function value template - captures exact function
                        // not a function type
int caller1(int x[100]) {
   int res = 0;
#pragma omp simd reduction(+:res)
   for (int i = 0; i < 100; i++) {
      res += f(x[i]);   // Exact function put here upon instantiation
   }
   return res;
}

template <typename F>  // Generic functional type template - captures
                       // object type for functors or entire functional type
```

```
                       // for functions. If vector attributes were part of
                       // a functional type they might be captured and applied
                       // but currently they are not.
int caller2(F f, int x[100]) {
   int res = 0;
#pragma omp simd reduction(+:res)
   for (int i = 0; i < 100; i++) {
      res += f(x[i]);  // Will call matching function 'f' indirectly
                       // Will call matching f.operator() directly
   }
   return res;
}

template <typename RET, typename ARG>  // Type-decomposing template
                                       // captures argument and return types.
                                       // Vector attributes would be lost
                                       // even if they were part of a
                                       // functional type.
int caller3(RET (*f)(ARG), int x[100]) {
   int res = 0;
#pragma omp simd reduction(+:res)
   for (int i = 0; i < 100; i++) {
      res += f(x[i]);  // Will call matching function 'f' indirectly
   }
   return res;
}


#pragma omp declare simd
int function(int x); // SIMD-enabled function
int nv_function(int x);                 // Regular scalar function

struct functor {                        // Functor class with
#pragma omp declare simd                      // SIMD-enabled operator()
   int operator()(int x);
};

int arr[100];

int main() {
   int res;
#pragma noinline
   res = caller1<function>(arr); // This will be instantiated for
                                 // function() and call short vector variant
#pragma noinline
   res += caller1<nv_function>(arr); // This will be separately instantiated
                                     // for nv_function()
#pragma noinline
   res += caller2(function, arr); // This will be instantiated for
                                  // int(*)(int) type and will call scalar
                                  // function() indirectly
#pragma noinline
   res += caller2(nv_function, arr); // This will call the same
                                     // instantiation as above on nv_function

#pragma noinline
   res += caller2(functor(), arr); // This will be instantiated for
                                   // functor type and will call short vector
```

```
                                   // variant of functor::operator()
#pragma noinline
   res += caller3(function, arr); // This will be instantiated for
                                  // <int, int> types and will call scalar
                                  // function() indirectly
#pragma noinline
   res += caller3(nv_function, arr); // This will call the same
                                     // instantiation as above on nv_function
   return res;
}
```

NOTE If calls to `caller1`, `caller2` and `caller3` are inlined, the compiler is able to replace indirect calls by direct calls in all cases. In this case `caller2(function, arr)` and `caller3(function, arr)` both call short vector variants of a function as result of the usual replacement of direct calls to function() by matching short vector variants in the SIMD loop.

Invoke a SIMD-enabled Function with Parallel Context

Typically, the invocation of a SIMD-enabled function provides arrays wherever scalar arguments are specified as formal parameters.

The following two invocations will give instruction-level parallelism by having the compiler issue special vector instructions.

```
a[:] = ef_add(b[:],c[:]);    //operates on the whole extent of the arrays a, b, c
```

```
a[0:n:s] = ef_add(b[0:n:s],c[0:n:s]); //use the full array notation construct to also specify n
as an extend and s as a stride
```

NOTE The array notation syntax, as well as calling the SIMD-enabled function from the regular `for` loop, results in invoking the short vector function in each iteration and using the vector parallelism but the invocation is done in a serial loop, without using multiple cores.
Use of array notation syntax and SIMD-enabled functions in a regular `for` loop results in invoking the short vector function in each iteration and using the vector parallelism, but the invocation is done in a serial loop without using multiple cores.

Use the __intel_simd_lane() Built-in Function

When called from within a vectorized loop, the `__intel_simd_lane()` built-in function will return a number between 0 and *vectorlength* - 1 that reflects the current "lane id" within the SIMD vector. `__intel_simd_lane()` will return zero if the loop is not vectorized. Calling `__intel_simd_lane()` outside of an explicit vector programming construct is discouraged. It may prevent auto-vectorization and such a call often results in the function returning zero instead of a value between 0 and *vectorlength*-1.

To see how `__intel_simd_lane()` can be used, consider the following example:

```
void accumulate(float *a, float *b, float *c, d){
  *a+=sin(d);
  *b+=cos(d);
  *c+=log(d);
}
```

```
for (i=low; i<high; i++){
    accumulate(&suma, &sumb, &sumc, d[i]);
}
```

The gather-scatter type memory addressing caused by the references to arrays A, B, and C in the SIMD-enabled function `accumulate()` will significantly hurt performance making the whole conversion useless. To avoid this penalty, you may use the `__intel_simd_lane()` built-in function as follows:

```
#pragma omp declare simd uniform(a,b,c) aligned(a,b,c)
void accumulate(float *a, float *b, float *c, float d){
// No need to take "loop index". No need to know VL.
  a[__intel_simd_lane()]+=sin(d);
  b[__intel_simd_lane()]+=cos(d);
  c[__intel_simd_lane()]+=log(d);
}

#define VL 16 // actual SIMD code may use vectorlength of 4 but it's okay.
float a[VL] = {0.0f};
float b[VL] = {0.0f};
float c[VL] = {0.0f};
#pragma omp simd for simdlen(VL)
for (i=low; i<high; i++){
    // If low is known to be zero at compile time, "i & (VL-1)"
    // would accomplish what __intel_simd_lane() is intended for,
    // but only on the caller side.
    accumulate(a, b, c, d[i]);
}
for(i=0;i<VL;i++){
    suma += a[i];
    sumb += b[i];
    sumc += c[i];
}
```

With the use of `__intel_simd_lane()`, the references to the arrays in `accumulate()` will have unit-stride.

Limitations

The following language constructs are not allowed within SIMD-enabled functions:

- The `GOTO` statement.
- The `switch` statement with 16 or more `case` statements.
- Operations on `classes` and `structs` (other than member selection).
- Any OpenMP construct.

See Also

`vector` attribute
User-Mandated or SIMD Vectorization
Function Annotations and the SIMD Directive for Vectorization
SIMD-Enabled Function Pointers

SIMD-enabled Function Pointers

SIMD-enabled functions (formerly called elemental functions) are a general language construct to express a data parallel algorithm. A SIMD-enabled function is written as a regular C/C++ function, and the algorithm within describes the operation on one element, using scalar syntax. The function can then be called as a regular C/C++ function to operate on a single element or it can be called in a data parallel context to operate on many elements.

In some cases it is desirable to have a pointer for SIMD-enabled functions, but without special effort, the vector nature of a function will be lost: function pointers will point to the scalar function and there will be no way to call the short vector variants existing for this scalar function.

In order to support indirect calls to vector variants of SIMD-enabled functions, SIMD-enabled function pointers were introduced. A SIMD-enabled function pointer is a special kind of pointer incompatible with a regular function pointer. They refer to an entire set of short vector variants as well as the scalar function. This incompatibility incurs the risk of inappropriate misuse, especially in C++ code. Therefore vector function pointer support is disabled by default.

To enable support of SIMD-enabled function pointers use the following compiler switches: `Qsimd-function-pointers` on Windows or `simd-function-pointers` on Linux/macOS.

Such pointers may hold the address of a SIMD-enabled function in a way that enables indirect calls to the appropriate vector variants of the function from a SIMD loop or another SIMD-enabled function.

When disabled with `Qsimd-function-pointers-` on Windows or `no-simd-function-pointers` on Linux/macOS vector attributes (`__declspec(vector)` or `__attribute__((vector))` or `#pragma omp declare simd`) can only be placed on function declarations and definitions. Other placements will result in a warning message and then be ignored.

How SIMD-enabled Function Pointers Work

When you write a SIMD-enabled function, the compiler generates short vector variants of the function that you requested, which can perform your function's operation on multiple arguments in a single invocation. The short vector variants may be able to perform multiple operations as fast as the regular implementation performs just one such operation by utilizing the vector instruction set architecture (ISA) in the CPU. When a call to SIMD-enabled function occurs in a SIMD loop or another SIMD-enabled function, the compiler replaces the scalar call with the best fit short vector variant of the function among those available.

Indirect SIMD-enabled function calls are handled similarly, but the set of available variants should be associated with the function pointer variable, not the target function, because actual call targets are unknown at the indirect call. That means all SIMD-enabled functions to be referenced by a SIMD-enabled function pointer should have a set of variants that match the set of variants declared for the pointer.

Declare a SIMD-enabled Function Pointer Variable

In order for the compiler to generate a pointer to a SIMD-enabled function, you need to provide an indication in your code.

Linux and macOS

Use the `__attribute__((vector (clauses)))` attribute, as follows:

```
__attribute__((vector (clauses))) return_type (*function_pointer_name) (parameters)
```

Alternately, you can use OpenMP `#pragma omp declare simd`, which requires the `[q or Q]openmp` or `[q or Q]openmp-simd` compiler option.

Windows

Use the `__declspec(vector (clauses))` attribute, as follows:

```
__declspec(vector (clauses)) return_type (*function_pointer_name) (parameters)
```

The clauses are described in the previous topic on SIMD-enabled functions.

Usage of Vector Function Attributes on Pointers

You may associate several vector attributes with one SIMD-enabled function pointer which reflects all the variants available for the target functions to be called through the pointer. The attributes usually reflect a possible use of the function pointer in the loops. Encountering an indirect call, the compiler matches the vector variants declared on the function pointer with the actual parameter kinds and chooses the best match.

Matching is done exactly the same way as with direct calls (see the previous topic on SIMD-enabled functions). Consider the following example of the declaration of vector function pointers and loops with indirect calls.

```
// pointer declaration
#pragma omp declare simd                           // universal but slowest definition matches
the use in all three loops
#pragma omp declare simd linear(in1) linear(ref(in2)) uniform(mul)   // matches the use in the
first loop
#pragma omp declare simd linear(ref(in2))                            // matches the use in the
second and the third loops
#pragma omp declare simd linear(ref(in2)) linear(mul)                // matches the use in the
second loop
#pragma omp declare simd linear(val(in2:2))                          // matches the use in the
third loop
int (*func)(int* in1, int& in2, int mul);

int *a, *b, mul, *c;
int *ndx, nn;
...
// loop examples
   for (int i = 0; i < nn; i++) {
       c[i] = func(a + i, *(b + i), mul); // in the loop, the first parameter is changed
linearly,
                                          // the second reference is changed linearly too
                                          // the third parameter is not changed
   }

   for (int i = 0; i < nn; i++) {
       c[i] = func(&a[ndx[i]], b[i], i + 1); // the value of the first parameter is
unpredictable,
                                             // the second reference is changed linearly
                                             // the third parameter is changed linearly
   }

   #pragma omp simd
   for (int i = 0; i < nn; i++) {
       int k = i * 2;  // during vectorization, private variables are transformed into arrays: k-
>k_vec[vector_length]
       c[i] = func(&a[ndx[i]], k, b[i]); // the value of the first parameter is unpredictable,
                                         // the second reference and value can be considered
linear
                                         // the third parameter has unpredictable value
                                         // (the __declspec(vector(linear(val(in2:2)))) will be
chosen from the two matching variants)
   }
```

Before any use in a call, the function pointer should be assigned either the address of a function or another function pointer. Just as with function pointers, vector function pointers should be compatible at assignment and initialization. The compatibility rules are described below.

Vector Function Pointer Compatibility

Pointer assignment compatibility is defined as following:

1. If a SIMD-enabled function pointer is assigned the address of a function, the function should be compatible with the pointer in the usual C/C++ sense, it should be SIMD-enabled, and the set of vector variants declared for the function should be a superset of those declared for the pointer. This includes initializations and passing addresses of SIMD-enabled functions as parameters.

2. If a SIMD-enabled function pointer is assigned another function pointer, the source pointer should be compatible with the destination function pointer in the general C/C++ sense, it should be SIMD-enabled, and the set of vector variants declared for the source pointer should be exactly the same as those declared for destination pointer. This includes initializations and passing SIMD-enabled function pointers as parameters.
3. If a regular (non-SIMD-enabled) function pointer is assigned the address of a SIMD-enabled function, the address of a scalar function is assigned. Vector variants cannot be called through the pointer and it cannot be reinterpreted as or converted into a SIMD-enabled function pointer as discussed in rule 2.
4. If a regular (non-SIMD-enabled) function pointer is assigned a SIMD-enabled function pointer matching in the C/C++ sense, the implicit dynamic casting of the right-hand side of the assignment (RHS) is performed by extracting the address of a scalar function and this address is assigned. Vector variants cannot be called through these pointers and it cannot be reinterpreted as or converted into a SIMD-enabled function pointer as discussed in rule 2.

NOTE
SIMD-enabled function pointers and regular function pointers are binary-incompatible and handled differently. Mixing them may lead to severe unpredictable results. The compiler does its best to check compatibility where it is allowed by C/C++ language standards, but in certain cases it cannot check, such as passing function pointers to undeclared functions or as variable arguments. It is best to refrain from using SIMD-enabled function pointers in these contexts. Additional complexities with respect to the C++ type system are described in the *SIMD-enabled Function Pointers and the C++ Type System* section below.

A SIMD-enabled function pointer may be assigned to a scalar function pointer with a cast as described in rule 4 above, but a SIMD-enabled function pointer cannot refer to a scalar function pointer.

```
// pointer declarations
#pragma omp declare simd
int (*ptr1)(int*, int);
#pragma omp declare simd
int (*ptr1a)(int*, int);

#pragma omp declare simd
#pragma omp declare simd linear(a)
typedef int (*fptr_t2)(int* a, int b);

typedef int (*fptr_t3)(int*, int);

fptr_t2 ptr2, ptr2a;
fptr_t3 ptr3;

// function declarations
#pragma omp declare simd
int func1(int* x, int b);

#pragma omp declare simd
#pragma omp declare simd linear(x)
int func2(int* x, int b);

#pragma omp declare simd
#pragma omp declare simd linear(x)
int func3(float* x, int b);

//----------------------------------------
  // allowed assignments
  ptr1 = func1;  // same prototype and vector spec
```

```
  ptr2 = func2;  // same prototype and vector spec
  ptr1a = ptr1;  // same prototype and vector spec
  ptr1a = func2; // same prototype vector spec on function includes all vector spec on pointer

  ptr3 = func1; // scalar pointer with same prototype - use scalar func1
  ptr3 = func2; // scalar pointer with same prototype - use scalar func2
  ptr3 = ptr1;  // scalar pointer with same prototype - implicit conversion from vector to
scalar pointer
  ptr3 = ptr2;  // scalar pointer with same prototype - implicit conversion from vector to
scalar pointer

  // disallowed assignments
  ptr2 = func1; // vector spec on function does not have all specs on pointer
  ptr2 = func3; // prototype mismatch although vector spec matched
  ptr1 = func3; // prototype mismatch although vector spec matched
  ptr3 = func3; // prototype mismatch
  ptr1 = ptr2;  // pointers should have the same vector spec
  ptr2 = ptr3;  // pointers should have the same vector spec
```

Call Sequence

Unlike regular function calls, which transfer control to a target function, the call target of an indirect call depends on the dynamic content of the function pointer. In a loop, call targets may be different on different iterations of a vectorized loop or on different lanes of a SIMD-enabled function executing the call. When vectorized, such an indirect call may involve multiple calls to different targets within a single SIMD chunk. This works as follows:

1. If the vector function pointer is uniform (refer to the OpenMP specification) or if it can be determined to be uniform by the compiler, then multiple calls are not needed. The compiler makes a single indirect call to a matched vector variant accessible by the pointer.
2. If the vector function pointer is not known to be uniform at compile time, all values of the pointer in a SIMD chunk may still be the same. This is checked at run time and a single indirect call to a matched vector variant is invoked.
3. Otherwise, lanes sharing the same function pointer value (call target) are masked-in and a masked vector variant corresponding to the matched one is invoked in the loop for each unique call target. If the masked variant is not provided for the matching vector variant and the function pointer is not proven to be uniform by compiler the match will be rejected and the compiler may serialize the call, or in other words, generate several scalar calls.

```
// pointer typedefs
#pragma omp declare simd
typedef int (*fptr_t1)(int*, int);

// function declarations
#pragma omp declare simd
int func1(int* x, int b);

// uses of vector function pointers
fptr_t1 *fptr_array;   // array of vector function pointers
void foo(int N, int *x, int y){
  fptr_t1 ptr1 = func1;
#pragma omp simd
  for (int i = 0; i < N; i++) {
    ptr1(x+i, y);  // ptr1 is uniform by OpenMP rule.
    fptr_t1 ptr1a = ptr1;
    ptr1a(x+i, y); // compiler can prove ptr1a is uniform.
    fptr_t1 ptr1b = fptr_array[i];
```

```
    ptr1b(x+i,y);  // ptr1b may or may not be uniform.
  }
}
```

SIMD-enabled Function Pointers and the C++ Type System

Use caution when using SIMD-enabled function pointers in modern C++: C++ imposes strict requirements on compilation and execution environments which may not compose well with semantically-rich language extensions such as SIMD-enabled function pointers. Vector specifications on SIMD-enabled function pointers are attributes in C++11 sense and so are not part of a pointer type even though they make that pointer binary incompatible with another pointer of the same type but without the attribute. Vector specifications are not bound to a pointer type, but instead are bound to the variable or function argument (which is an instance of a pointer type) itself. For a given function pointer, the type of the pointer is the same with or without SIMD-enabled function pointer decoration. This has the following important implications:

- Vector attributes put on a function argument are not reflected in C++ name mangling, so the functions differ only in the vector attributes of a functional pointer argument (or lack thereof) will have the same name and will be treated the same by the C++ linker. This may result in a parameter of incorrect vectorness (having the vector attribute or not) being passed into the function. In some cases there is no way for the compiler to detect this situation, so you're strongly encouraged to distinctly name functions having SIMD-enabled function pointers as parameters.
- The incorrect interpretation of function pointers is extremely dangerous because it may lead to the execution of unwanted code or non-code. To identify these situations the compiler issues the following warning if a vector function pointer is used as a C++ function parameter: Warning #3757: this use of a vector function type is not fully supported. If you are sure that no ambiguity is possible—for example, the function accepting the vector function pointer has a distinct name and is fully declared before all uses—you may ignore this warning. Otherwise, ensure that no ambiguity is possible. To prevent such situations the warning can be converted to an error using the command line switch `-diag-error=3757`.
- Template instantiations having SIMD-enabled pointer types as template parameters won't catch vector attributes. The template will be instantiated a parameter matching the non-SIMD-enabled pointer type. All variables, class members, and function arguments bound to the template argument type will be regular function pointers. The use of such templates with a SIMD-enabled function pointer as a template function parameter, template class method parameter, or RHS of template class member assignment will lead to a dynamic cast to the non-SIMD-enabled function pointer and loss of vectorness.
- There is no way to overload or achieve template specialization by the vector attributes of a functional pointer
- There is no way to write functional traits to capture vector attributes for the sake of template metaprogramming.

```
// pointer typedefs and pointer declarations
typedef int
(*fptr_t)(int*, int);

#pragma omp declare simd
typedef int (*fptr_t1)(int*, int);

#pragma omp declare simd
#pragma omp declare simd linear(x)
typedef int (*fptr_t2)(int* a, int b);

fptr_t ptr
fptr_t1 ptr1
fptr_t2 ptr2

// function prototype that only differs in SIMD-enabled function decoration
// All these will have identical mangled names.
void foo(fptr_t);
```

```
void foo(fptr_t1);
void foo(fptr_t2);

// template instantiation
template <typename T>
void bar(T);
…
  bar(fptr);          // bar<fptr_t>
  bar(fptr1);         // bar<fptr_t>
  bar(fptr2);         // bar<fptr_t>
```

Indirect Invocation of a SIMD-enabled Function with Parallel Context

Typically, the invocation of a SIMD-enabled function directly or indirectly provides arrays wherever scalar arguments are specified as formal parameters.

The following invocations will give instruction-level parallelism by having the compiler issue special vector instructions.

```
#pragma omp declare simd
float (**vf_ptr)(float, float);

//operates on the whole extent of the arrays a, b, c
a[:] = vf_ptr[:] (b[:],c[:]);

// use the full array notation construct to also specify n
// as an extend and s as a stride
a[0:n:s] = vf_ptr[0:n:s] (b[0:n:s],c[0:n:s]);
```

NOTE The array notation syntax, as well as calling the SIMD-enabled function from the regular `for` loop, results in invoking the short vector variant in each iteration and utilizing the vector parallelism but the invocation is done in a serial loop, without utilizing multiple cores.

See Also

`vector` attribute
User-mandated or SIMD Vectorization
Function Annotations and the SIMD Directive for Vectorization
SIMD-enabled functions

Vectorize a Loop Using the _Simd Keyword

In this section we introduce the `_Simd` keyword, which provides an alternative to the `simd` pragma. Just like the `simd` pragma, the `_Simd` keyword modifies a serial `for` loop for vectorization. The syntax is as follows:

```
_Simd [_Safelen(constant-expression)][_Reduction (reduction-identifier : list)]
```

The `_Simd` keyword and any clauses should come after the `for` keyword as in this example:

```
for _Simd (int i=0; i<10; i++){
  // loop body
}
```

Differences between the `simd` pragma and `_Simd` keyword:

- Omission of the `private` and `lastprivate` clauses of the `simd` pragma construct because C and C++ already have variable-scoping rules that allow a programmer to cleanly declare a private variable within the scope of a loop iteration

- The `linear` clause is omitted because the ability to increment multiple variables makes it unnecessary. See the following example:

```
float add_floats(float *a, float *b, int n){
  int i=0;
  int j=0;
  float sum=0;

  for _Simd _Reduction(+:sum) (i=0; i<n; i++, j+=2){
    a[i] = a[i] + b[j];
    sum += a[i];
  }
  return sum;
}
```

To ensure that your loop is vectorized keep the following in mind:

- The countable loop for the _Simd keyword has to conform to the for-loop style of an OpenMP* canonical loop form except that multiple variables may be incremented in the incr-expr (See the OpenMP* specification at www.openmp.org).
- The loop control variable must be a signed integer type.
- The vector values should be signed 8-, 16-, 32-, or 64-bit integers, single or double-precision floating point numbers, or single or double-precision complex numbers.
- You cannot use any control constructs to jump into or out of a SIMD loop. That includes the `break`, `return`, `goto`, and `throw` constructs.
- A SIMD loop may contain another loop (`for`, `while`, `do-while`) in it, but `goto` out of such inner loops is not supported. You may use `break` and `continue` with the inner loop.
- A SIMD loop performs memory references unconditionally. Therefore, all address computations must result in valid memory addresses, even though such locations may not be accessed if the loop is executed sequentially

See Also

User-mandated or SIMD Vectorization

simd Enforces vectorization of loops.

Function Annotations and the SIMD Directive for Vectorization

This topic presents specific C++ language features that better help to vectorize code.

> **NOTE**
> The SIMD vectorization feature is available for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x`.

The `__declspec(align(`*n*`))` declaration enables you to overcome hardware alignment constraints. The auto-vectorization hints address the stylistic issues due to lexical scope, data dependency, and ambiguity resolution. The SIMD feature's pragma allows you to enforce vectorization of loops.

You can use the `__declspec(vector)` `__attribute__(vector)` and the `__declspec(vector[clauses])` `__attribute__(vector(`*clauses*`))` declarations to vectorize user-defined functions and loops. For SIMD usage, the `vector` function is called from a loop that is being vectorized.

The C/C++ extensions for Array Notations `map` operations can be defined to provide general data parallel semantics, where you do not express the implementation strategy. You can write the same operation regardless of the size of the problem. The implementation uses the construct by combining SIMD, loops and tasking to implement the operation. With these semantics, you can choose more elaborate programming and express a single dimensional operation at two levels. You can use both task constructs and array operations to force a preferred parallel and vector execution.

The usage model of the `vector` declaration takes a small section of code generated for the function (*vectorlength*) of the array and exploits SIMD parallelism. The implementation of task parallelism is done at the call site.

The following table summarizes the language features that help vectorize code.

Language Feature	Description
`__declspec(align(`*n*`))`	Directs the compiler to align the variable to an *n*-byte boundary. Address of the variable is *address* `mod n=0`.
`__declspec(align(`*n*`,off))`	Directs the compiler to align the variable to an *n*-byte boundary with offset off within each *n*-byte boundary. Address of the variable is *address* `mod n=off`.
`__declspec(vector)` (Windows*) `__attribute__(vector)` (Linux* and macOS)	Combines with the `map` operation at the call site to provide the data parallel semantics. When multiple instances of the vector declaration are invoked in a parallel context, the execution order among them is not sequenced.
`__declspec(vector[`*clauses*`])` (Windows*) `__attribute__(vector(`*clauses*`))` (Linux* and macOS)	Combines with the `map` operation at the call site to provide the data parallel semantics with the following values for *clauses*: • processor clause: *processor(cpuid)* • vector length clause: *vectorlength(n)* • linear clause: *linear(param1:step1 [, param2:step2]…)* • uniform clause: *uniform(param [, param,]…)* • mask clause: *[no]mask* When multiple instances of the vector declaration are invoked in a parallel context, the execution order among them is not sequenced.
`restrict`	Permits the disambiguator flexibility in alias assumptions, which enables more vectorization.
`__declspec(vector_variant(`*clauses*`))` (Windows*) `__attribute__(vector_variant(`*clauses*`))` (Linux* and macOS)	Provides the ability to vectorize user-defined functions and loops. The *clauses* are as follows: • implements clause (required): *implements (function declarator) [, simd-clauses])* • simd-clauses (optional): one or more of the clauses allowed for the vector attribute
`__assume_aligned(`*a*`,`*n*`)`	Instructs the compiler to assume that array *a* is aligned on an *n*-byte boundary; used in cases where the compiler has failed to obtain alignment information.
`__assume(`*cond*`)`	Instructs the compiler to assume that the represented condition is true where the keyword appears. Typically used for conveying

Language Feature	Description
	properties that the compiler can take advantage of for generating more efficient code, such as alignment information.
Auto-vectorization Hints	
`#pragma ivdep`	Instructs the compiler to ignore assumed vector dependencies.
`#pragma vector {aligned|unaligned|always|temporal|nontemporal}`	Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored. Using the `assert` keyword with the `vector {always}` pragma generates an error-level assertion message if the compiler efficiency heuristics indicate that the loop cannot be vectorized. Use `#pragma ivdep!` to ignore the assumed dependencies.
`#pragma novector`	Specifies that the loop should never be vectorized.

NOTE
Some pragmas are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

User-Mandated Pragma	
`#pragma simd`	Enforces vectorization of loops.
omp simd	Transforms the loop into a loop that will be executed concurrently using SIMD instructions.

See Also

__declspec(align) declaration

__declspec(vector) declaration

__declspec(vector_variant) declaration

ivdep pragma
simd pragma
vector pragma
SIMD-enabled functions
User-mandated or SIMD Vectorization

Guided Auto Parallelism

NOTE This feature has been deprecated.

The Guided Auto Parallelism (GAP) feature of the Intel®C++ Compiler is a tool that offers selective advice to improve the performance of serially-coded applications by suggesting changes that take advantage of the compiler's ability to automatically vectorize and parallelize code and improve the efficiency of data operations. Despite having the words “auto parallelism” in the name, this tool does not require a threaded code implementation to improve the execution performance of the code, or require that the code is already threaded or parallel.

Advanced optimization techniques, such as inter-procedural analysis or profile-guided feedback, are not needed to use this feature. Using the `[Q]guide` set of options in addition to the compiler options normally used is sufficient to enable the GAP feature, with the requirement that you must compile with `O2` or higher optimization levels. The compiler does not generate any object files or executables during the GAP run.

In debug mode (`/Zi` on Windows*, `-g` on Linux*), the compiler's optimization level defaults to `/Od` (on Windows*) or `-O0` (on Linux* and macOS); thus `O2` (or a higher level optimization) must be specified explicitly on the command-line.

NOTE Use the `[Q]diag-disable` option along with the `[Q]guide` option to direct the compiler to suppress specific diagnostic messages.
For example, the options: `//` **`(Windows*)`**`/Qguide, /Qdiag-disable:30534` and `//` **`(Linux* and macOS)`** `-guide, -diag-disable:30534` tell the compiler not to emit the 30534 diagnostic. The `[Q]diag-disable` mechanism works the same way as it does for compiler-warnings.

If you decide to follow the advice offered by the GAP tool by making the suggested code changes and/or using the suggested compiler options, you must then recompile the program without the `[Q]guide` option.

Any advice generated by the compiler when using the GAP tool is optional; it can be implemented or rejected. The advice typically falls under three broad categories:

- **Advice for source modifications:** The compiler advises you to make source changes that are localized to a loop-nest or a routine. For example, the tool may recommend that you use a local-variable for the upper-bound of a loop, (instead of a class member) or that you should initialize a local variable unconditionally at the top of the loop-body, or you may be told to add the `restrict` keyword to pointer-arguments of a function definition (if appropriate).
- **Advice to apply pragmas:** The compiler advises you to apply a new pragma on a certain loop-level if the pragma semantics can be satisfied (you must verify this). In many cases, you may be able to apply the pragma (thus implicitly asserting new program/loop properties) that the compiler can take advantage of to perform enhanced optimizations.
- **Advice to add compiler options:** The compiler advises you to add command-line options that assert new properties; for example, you may be asked to use the `[Q]ansi-alias` option or `/Qalias-args` (Windows*) or `-fargument-alias` (on Linux*) compiler options.

NOTE These suggested compiler options apply to the entire file. It is your responsibility to check that the properties asserted by these options are valid for the entire file, and not just the loop in question.

If you use GAP options along with option `[Q]parallel`, the compiler may suggest options to further parallelize your application. The compiler may also offer advice on enabling other optimizations of your application, including vectorization.

If you use the GAP options without enabling auto parallelism (without using the `[Q]parallel` option), the compiler may only suggest enabling optimizations such as vectorization for your application. This approach is recommended when you wish to improve the performance of a single-threaded code without the use of parallelization or when you want to improve the performance of threaded applications that do not rely on the compiler for auto parallelism.

See Also

Using Guided Auto Parallelism

`diag`
compiler option

`ansi-alias, Qansi-alias`
compiler option

`fargument-alias, Qalias-args`

compiler option

Zi

compiler option

g

compiler option

Using Guided Auto Parallelism

The Guided Auto Parallelism feature of the Intel® C++ Compiler is a tool offering selective advice to improve the performance of serially-coded applications. The tool suggests changes that take advantage of the compiler's ability to automatically vectorize and parallelize code as well as improve the efficiency of data operations. The tool does not require that you implement threaded code to improve the execution performance of your code, nor does it require that your code is already threaded or parallel code.

To invoke this tool, use the compiler option `[Q]guide`*[=n]*. Using this option causes the compiler to generate messages suggesting ways to optimize the performance of your application. You can also use more specific compiler options such as `[Q]guide-vec`, `[Q]guide-par`, and `[Q]guide-data-trans` , to perform individual guided optimizations for vectorizing, parallelizing, and data transformation of your application.

When any guided auto parallelism option is used, the compiler provides only diagnostic advice. Object files or executables are not created in this mode. See the table below for descriptions of the options.

Syntax	Description
`[Q]guide`	Allows you to set a level of guidance for auto-vectorization and data transformation analysis. To obtain guidance for auto parallelism, you must use the `[Q]parallel` option along with the `[Q]guide` option.
`[Q]guide-par`	Allows you to set a level of guidance only for auto parallelism analysis. **NOTE** You must use the `[Q]parallel` option along with the `[Q]guide-par` option to get this advice.
`[Q]guide-vec`	Allows you to set a level of guidance for auto-vectorization analysis only.
`[Q]guide-data-trans`	Allows you to set a level of guidance for data transformation analysis only.

For all of the above options, the optional argument *n* specifies the level of guidance. The argument *n* takes the values `1-4`. When *n* is not specified, the default is `4`. If you specify *n*=`1` or `2`, a standard level of guidance is provided.

When you use *n*=`3` or *n*=`4`, you may get advanced messages. For example, you may get messages about how to optimize a particular loop-nest or get a message on how exception-handling inside a loop-nest affects optimizations for that loop-nest. Or you may get a message on how to provide extra information to the compiler on cost-modeling (expected values of trip-counts, and so on).

If you simultaneously specify a level of guidance for the general `[Q]guide` option and also for one or more of the other specific guide options, the level of guidance (*n*) for the specific guide option overrides the general `[Q]guide` option setting.

If you do not specify a level of guidance for the general `[Q]guide` option, but do set a level of guidance for one or more of the specific guide options, the `[Q]guide` option is set equal to the greatest value passed to the specific guide options.

Capturing Guidance Messages

The guided auto parallelism tool analyzes all of your serial code or individual parts of your code and generates advisory messages. By default, messages that are generated by the guided auto parallelism tool are output to `stderr`.

To capture messages in a file, use the options listed in the following table.

NOTE
The options listed in the following table must be used with the `[Q]guide`, `[Q]guide-par`, `[Q]guide-vec`, or `[Q]guide-data-trans` options. If not, they are ignored.

Syntax	Description
`[Q]guide-file`	Gathers all messages generated during a guided auto-parallelization run into the specified file.
`[Q]guide-file-append`	Allows you to specify the file into which all messages generated during a guided auto parallelism run should be appended.

For the above options, the *file_name* argument can also include a path. If a path is not specified, the file is created in the current working directory. If there is already a file named *file_name*, it is overwritten when you use the `[Q]guide-file` option. If you do not include an extension as part of the *file_name*, the extension `.guide` is appended.

Configuring Code Regions for GAP Messages

To limit guided auto parallelism analysis to specific regions (hotspots) in your application, use the option mentioned in the table below.

Syntax	Description
`[Q]guide-opts`	Allows you to analyze specified code elements, identified by *string*.

You must use the `[Q]guide-opts` option along with one of the guided auto parallelism options, such as `[Q]guide`, `[Q]guide-vec`, `[Q]guide-par`, and `[Q]guide-data-trans` . Use the *string* parameter to provide information about known areas of interest (hotspots). The *string* parameter takes one or more of the following variables: *filename*, *routine*, *range*. The compiler parses the *string* parameter and generates syntax errors if there are any.

Windows* Syntax

`/Qguide-opts:`*`string`*

Linux* and macOS Syntax

`-guide-opts=`*`string`*

Follow these guidelines when using the *string* parameter:

- Use only valid file names, routine names, and line numbers. The guided auto parallelism tool ignores invalid values and issues a diagnostic message stating what was ignored.
- Enclose routine names within single quotation marks. Specify original source names (demangled names) as routine names. A routine name alone may not always be sufficient to uniquely identify a routine. You may need to specify additional parameter information to uniquely identify the routine. For example:

Linux* and macOS:

```
-guide-
opts="foo.cpp,'CLHEP::StaticRandomSta::restore(std::basic_istream<char,std::char_traits<char>>&)'
"
-guide-opts="bar.f90,'module_1::routine_name`"
-guide-opts="baz.c,'c_routine_name'"
```

Windows*:

```
/Qguide-
opts:"foo.cpp,'CLHEP::StaticRandomSta::restore(std::basic_istream<char,std::char_traits<char>>&)'
"
/Qguide-opts:"bar.f90,'module_1::routine_name`"
/Qguide-opts:"baz.c,'c_routine_name'"
```

For any specified routine name, the GAP tool first tries to uniquely identify the routine using specified routine information. If that is not possible, then it selects all routines with the specified routine name. The GAP tool uses the parameter information, if specified, to narrow the selection.

- When inlining is involved, use the callee line numbers. The generated messages also use the callee line numbers.

See Also

`guide, Qguide` compiler option

`guide-par, Qguide-par` compiler option

`guide-vec, Qguide-vec` compiler option

`guide-data-trans, Qguide-data-trans` compiler option

`guide-file, Qguide-file` compiler option

`guide-file-append, Qguide-file-append` compiler option

`guide-opts, Qguide-opts` compiler option

See Also

Using Guided Auto Parallelism in the Microsoft Visual Studio* IDE

Guided Auto Parallelism Messages

The Guided Auto Parallelism (GAP) messages provide advice that should improve optimizations.

The messages provide suggestions for:

- Automatic parallelization of loop nests
- Automatic vectorization of inner loops
- Data transformation

You must decide whether to follow a particular suggestion. For example, if the advice is to apply a particular pragma, you must understand the semantics of the pragma and carefully consider whether it can be safely applied to the loop (or loop nest) in question.

If you apply the pragma improperly, the compiler may generate incorrect code, causing the application to execute incorrectly.

If you do not fully understand the suggested advice, please refer to the relevant topics in the compiler documentation before applying that advice.

Once you apply the suggested advice, the compiler assumes that it is correct and it does not perform any checks or issue any warnings.

In general, messages that relate to loops tend to target vectorization and/or parallelization of loops. If you are not familiar with loop optimizations, please refer to the compiler documentation on this kind of optimization.

See Also

Using Guided Auto Parallelism

Guided Auto Parallelism
Enabling Auto-parallelization
Enabling Further Loop Parallelization for Multicore Platforms
Loop Constructs

GAP Message (Diagnostic ID 30506)

Message

If the following operations(s) can be safely performed unconditionally, the loop at line %d will be vectorized by adding a "%s ivdep" statement right before the loop: %s.

Advice

Add "#pragma ivdep" before the specified loop.

This pragma enables the vectorization of the loop at the specified line. Insure that any conditional divide, sqrt, and inverse sqrt operations will not alter the exception semantics expected by the program when they are performed unconditionally.

Example

Consider the following:

```
void foo(float *a, int n) {
  int i;
  for (i=0; i<n; i++) {
    if (a[i] > 0) {
      a[i] = 1 / a[i];
    }
  }
  return;
}
```

In this case, the compiler is unable to vectorize the loop if compiled with floating-point exception semantics (such as `/fp:except option`) because the condition "a[i] > 0" may be guarding floating-point exceptions for the divide.

If you determine it is safe to do so, you can add the pragma as follows:

```
void foo(float *a, int n) {
  int i;
#pragma ivdep
  for (i=0; i<n; i++) {
    if (a[i] > 0) {
      a[i] = 1 / a[i];
    }
  }
  return;
}
```

Verify

Confirm that the program operands have safe values for all iterations.

GAP Message (Diagnostic ID 30513)

Message

Insert a "%s ivdep" statement right before the loop at line %d to vectorize the loop.

Advice

Add "#pragma ivdep" before the specified loop. This pragma enables the vectorization of the loop at the specified line by ignoring some of the assumed cross-iteration data dependencies.

Example

Consider the following:

```
void foo(float *a, int x, int n) {
  int i;
  for (i=0; i<n; i++) {
    a[i] = a[i+x]+1;
  }
  return;
}
```

In this case, the compiler is unable to vectorize the loop because x could be -1, where each iteration is dependent on the previous iteration. If x is known to be positive, you can vectorize this loop.

If you determine it is safe to do so, you can add the pragma as follows:

```
void foo(float *a, int x, int n) {
  int i;
#pragma ivdep
  for (i=0; i<n; i++) {
    a[i] = a[i+x]+1;
  }
  return;
}
```

Verify

Confirm that any arrays in the loop do not have unsafe cross-iteration dependencies. A cross-iteration dependency exists if a memory location is modified in an iteration of the loop and accessed by a read or a write statement in another iteration of the loop. Make sure that there are no such dependencies, or that any cross-iteration dependencies can be safely ignored.

GAP Message (Diagnostic ID 30515)

Message

Assign a value to the variable(s) "%s" at the beginning of the body of the loop in line %d. This will allow the loop to be vectorized.

Advice

You should unconditionally initialize the scalar variables at the beginning of the specified loop. This allows the vectorizer to privatize those variables for each iteration and vectorize the loop. You must ensure that all the uses of those variables see the same values before and after the source code change.

Example

Consider the following:

```
void foo(float *a, int n) {
  int i;
  float b;
  for (i=0; i<n; i++) {
    if (a[i] > 0) {
      b = a[i];
      a[i] = 1 / a[i];
    }
    if (a[i] > 1) {
      a[i] += b;
    }
  }
  return;
}
```

In this case, the compiler is unable to vectorize the loop because it failed to privatize the variable *b*. Vectorization is assisted when assignment to *b* occurs in each iteration where the value of *b* is used. One of the ways to do this is to assign the value in every iteration.

If you determine it is safe to do so, you can modify the program code as follows:

```
void foo(float *a, int n) {
  int i;
  float b;
  for (i=0; i<n; i++) {
    b = a[i];
    if (a[i] > 0) {
      a[i] = 1 / a[i];
    }
    if (a[i] > 1) {
      a[i] += b;
    }
  }
  return;
}
```

Verify

Confirm that in the original program, any variables read in any iteration of the loop have been defined earlier in the same iteration.

GAP Message (Diagnostic ID 30519)

Message

Insert a "%s parallel" statement right before the loop at line %d to parallelize the loop.

Advice

Add "#pragma parallel" before the specified loop. This pragma enables the parallelization of the loop at the specified line by ignoring assumed cross-iteration data dependencies.

Example

Consider the following:

```
void foo(float *a, int m, int n) {
  int i;
  for (i=0; i<n; i++) {
    a[i] = a[i+m]+1;
  }
  return;
}
```

In this case, the compiler is unable to parallelize the loop without further information about *m*. For example, if *m* is negative, then each iteration will be dependent on the previous iteration. However, if *m* is known to be greater than *n*, you can parallelize the loop.

If you determine it is safe to do so, you can add the pragma as follows:

```
void foo(float *a, int m, int n) {
  int i;
#pragma parallel
  for (i=0; i<n; i++) {
    a[i] = a[i+m]+1;
  }
  return;
}
```

Verify

Confirm that any arrays in the loop do not have cross-iteration dependencies. A cross-iteration dependency exists if a memory location is modified in an iteration of the loop and accessed by a read or a write statement in another iteration of the loop.

GAP Message (Diagnostic ID 30521)

Message

Assign a value to the variable(s) "%s" at the beginning of the body of the loop in line %d. This will allow the loop to be parallelized.

Advice

Check to see if you can unconditionally initialize the scalar variables at the beginning of the specified loop. If so, do the code change for such initialization (standard), or list the variables in the private clause of a parallel pragma (advanced). This allows the parallelizer to privatize those variables for each iteration and to parallelize the loop.

Example

Consider the following:

```
#define N 100000
double A[N], B[N];

void foo(int cond1, int cond2){
  int i, t=7;
  for (i=0; i<N; i++){
    if (cond1) {
      t = i+1;
    }
```

```
    if (cond2) {
      t = i-1;
    }
    A[i] = t;
  }
}
```

In this case, the compiler does not parallelize the loop because it cannot privatize the variable *t* without further information. If you know that cond1 or cond2 is true, or both cond1 and cond2 are true, then you can assist the parallelizer by ensuring that any iteration that uses *t* also writes to *t* before its use in the same iteration. One of the ways to do this is to assign a value to *t* at the top of each iteration.

If you determine it is safe to do so, you can modify the program code as follows:

```
#define N 100000
double A[N], B[N];

void foo(int cond1, int cond2){
  int i, t=7;
  for (i=0; i<N; i++){
    t=0;
    if (cond1) {
      t = i+1;
    }
    if (cond2) {
      t = i-1;
    }
    A[i] = t;
  }
}
```

Verify

Confirm that in the original program, any variables read in any iteration of the loop have been defined earlier in the same iteration.

See Also

GAP Message (Diagnostic ID 30523)

GAP Message (Diagnostic ID 30522)

Message

Insert a "%s parallel private(%s)" statement right before the loop at line %d to parallelize the loop.

Advice

Add "#pragma parallel private" before the specified loop. This pragma enables the parallelization of the loop at the specified line.

Example

Consider the following:

```
float A[10][10000];
float B[10][10000];
float C[10][10000];
void foo(
 int n,
 int m1,
```

```
 int m2
)
{
  int i,j;
  float W[10000];
  for (i =0; i < n; i++) {
    for (j =0; j < m1; j++)
      W[j]  =  A[i][j] * B[i][j];
    for (j =0; j < m2; j++)
      C[i][j] += W[j] + 1.0;
  }
}
```

In this case, the compiler does not parallelize the loop since it cannot determine whether m1 >= m2.

If you know that this property is true, and that no element of W is fetched before it is written to after the loop, then you can use the recommended pragma.

If you determine it is safe to do so, you can add the pragma as follows:

```
float A[10][10000];
float B[10][10000];
float C[10][10000];
void foo(
 int n,
 int m1,
 int m2
)
{
  int i,j;
  float W[10000];
#pragma parallel private (W)
  for (i =0; i < n; i++) {
    for (j =0; j < m1; j++)
      W[j]  =  A[i][j] * B[i][j];
    for (j =0; j < m2; j++)
      C[i][j] += W[j] + 1.0;
  }
}
```

Verify

Before an element of an array can be read in the loop, there must have been a previous write to it during the same loop iteration. In addition, if an element is read after the loop, there must have been a previous write to it before the read after the loop.

GAP Message (Diagnostic ID 30523)

Message

Assign a value to the variable(s) "%s" at the beginning of the body of the loop in line %d. This will allow the loop to be parallelized.

Advice

Check to see if you can unconditionally initialize the scalar variables at the beginning of the specified loop. If so, do the code change for such initialization (standard), or list the variables in the private clause of a parallel pragma (advanced). This allows the parallelizer to privatize those variables for each iteration and to parallelize the loop.

Example

Consider the following:

```
#define N 100000
double A[N], B[N];
```

```
void foo(int cond1, int cond2){
  int i, t=7;
  for (i=0; i<N; i++){
    if (cond1) {
      t = i;
    }
    if (cond2) {
      A[i] = t;
    }
  }
}
```

In this case, the compiler does not parallelize the loop because it cannot privatize the variable *t* without further information. If you know that cond2 always implies cond1, then you can assist the parallelizer by ensuring that any iteration that uses *t*, also writes to *t* before it is used in the same iteration. One of the ways to do this is to assign a value to *t* at the top of every iteration. Another way is to list the variables to be privatized in the private clause of a parallel pragma.

If you determine it is safe to do so, you can add the pragma as follows:

```
#define N 100000
double A[N], B[N];

void foo(int cond1, int cond2){
  int i, t=7;
#pragma private (t)
  for (i=0; i<N; i++){
    if (cond1) {
      t = i;
    }
    if (cond2) {
      A[i] = t;
    }
  }
}
```

Verify

Confirm that in the original program, any variables read in any iteration of the loop have been defined earlier in the same iteration or have been privatized by means of the private clause of a parallel pragma.

See Also

GAP Message (Diagnostic ID 30521)

GAP Message (Diagnostic ID 30525)

Message

Insert a "%s loop count min(%d)" statement right before the loop at line %d to parallelize the loop.

Advice

Add "#pragma loop count" before the specified loop. This pragma indicates the minimum trip count (number of iterations) of the loop that enables the parallelization of the loop.

The minimum trip count required to parallelize the loop may differ depending on the target architecture, and this will be reflected in the message generated.

Example

Consider the following:

```
#define N 10000
float A[N], B[N];

void foo(int n) {
  int i;
  for (i =0; i < n; i++) {
      A[i] = A[i] + B[i] * B[i] + 1.5;
  }
}
```

In this case, the compiler may not parallelize the loop because it is not sure that n is large enough for the parallelization to be beneficial.

If you determine it is safe to do so, you can add the pragma as follows:

```
#define N 10000
float A[N], B[N];

void foo(int n) {
  int i;
#pragma loop count min(128)
  for (i =0; i < n; i++) {
      A[i] = A[i] + B[i] * B[i] + 1.5;
  }
}
```

Verify

Confirm that the loop has the minimum number of iterations, as specified in the diagnostic message.

See Also

Guided Auto Parallelism Messages provides advice for improving optimizations

GAP Message (Diagnostic ID 30526)

Message

To parallelize the loop at line %d, annotate the routine %s with %s.

Advice

If the loop contains a call to a function, the compiler cannot parallelize the loop without more information about the function being called.

However, if the function being called in the loop is a const function or a concurrency-safe function, then the call does not inhibit parallelization of the loop.

Example

Consider the following:

```
#define N 10000
double A[N], B[N];
int bar(int);
void foo(){
  int i;
  for (i=0;i<N;i++){
    A[i] = B[i] * bar(i);
  }
}
```

In this case, the compiler does not parallelize the loop because it is not safe to do so without further information about routine `bar`, which is being called.

If you determine it is safe to do so, you can modify the program code as follows:

```
#define N 10000
double A[N], B[N];
__declspec(const) int bar(int);
void foo(){
  int i;
  for (i=0;i<N;i++){
    A[i] = B[i] * bar(i);
  }
}
```

If you determine it is safe to do so, an alternative way you can modify the program code is as follows:

```
#define N 10000
double A[N], B[N];
__declspec(concurrency_safe(profitable)) int bar(int);
void foo(){
  int i;
  for (i=0;i<N;i++){
    A[i] = B[i] * bar(i);
  }
}
```

Verify

Confirm the routine satisfies the semantics of this annotation. A weaker annotation able to achieve a similar effect is `__declspec(concurrency_safe(profitable)`.

See Also

GAP Message (Diagnostic ID 30528)

`__declspec(concurrency_safe)` declaration

`__declspec(const)` declaration

GAP Message (Diagnostic ID 30528)

Message

Add "%s" to the declaration of routine "%s" in order to parallelize the loop at line %d. Adding "%s" achieves a similar effect.

Advice

Confirm that the routine specified is indeed a const function or a concurrency-safe function before following the advice to add the annotation.

If the routine is not one of these kinds of functions, try to inline it with #pragma forceinline recursive. This action may or may not be beneficial.

Example

Consider the following:

```
#define N 10000
double A[N], B[N];
int bar(int);
void foo(){
  int i;
  for (i=0;i<N;i++){
    A[i] = B[i] * bar(i);
  }
}
```

In this case, the compiler does not parallelize the loop because it is not safe to do so without further information about routine `bar`, which is being called.

If you determine it is safe to do so, you can add the pragma as follows:

```
#define N 10000
double A[N], B[N];
void foo(){
  int i;
#pragma forceinline recursive
  for (i=0;i<N;i++){
    A[i] = B[i] * bar(i);
  }
}
```

Verify

Confirm that the routine satisfies the semantics of this declaration. Another way to help the loop being parallelized is to inline the routine with the forceinline recursive pragma, but this method does not guarantee parallelization.

See Also

GAP Message (Diagnostic ID 30526)

GAP Message (Diagnostic ID 30531)

Message

Store the value of the upper-bound expression of the loop at line %d into a temporary local variable, and use this variable as the new upper-bound expression of the loop. To do this, insert a statement of the form "temp = upper-bound of loop" right before the loop, where "temp" is the newly created local variable. Choose a variable name that is unique, then replace the loop's original upper-bound expression with "temp".

Advice

Use a local-variable for the loop upper-bound if the upper-bound does not change during the execution of the loop. This enables the compiler to recognize the loop as a proper counted do loop, which enables various loop optimizations including vectorization and parallelization.

This message appears when the compiler cannot output the exact upper-bound variable to be replaced.

Example

Consider the following:

```
class FooClass  {
public:
    const int getValue() { return m_numTimeSteps;}
        void Foo2(double*  vec);
private:
        int m_numTimeSteps;
};
void FooClass::Foo2(double*  vec)
{
        // this will not vectorize
        for (int k=0; k < m_numTimeSteps; k++)
                vec[k] = 0.0;

        // this will not vectorize
        for (int k=0; k < getValue(); k++)
                vec[k] = 0.0;

        // this will vectorize
        int ub1 = m_numTimeSteps;
        for (int k=0; k < ub1; k++)
                vec[k] = 0.0;

        // this will vectorize
        int ub2 = getValue();
        for (int k=0; k < ub2; k++)
                vec[k] = 0.0;
}
```

Verify

Confirm that the value of the upper-bound expression does not change throughout the entire execution of the loop.

See Also

GAP Message (Diagnostic ID 30532)

GAP Message (Diagnostic ID 30532)

Message

Store the value of the upper-bound expression (%s) of the loop at line %d into a temporary local variable, and use this variable as the new upper-bound expression of the loop. To do this, insert a statement of the form "temp = %s" right before the loop, where "temp" is the newly created local variable. Choose a variable name that is unique, then replace the loop's original upper-bound expression with "temp".

Advice

Use a local-variable for the loop upper-bound if the upper-bound does not change during the execution of the loop. This enables the compiler to recognize the loop as a proper counted do loop, which in turn enables various loop optimizations including vectorization and parallelization.

This message appears when the compiler can output the exact upper-bound variable to be replaced.

Example

Consider the following:

```
typedef struct  {
    float*      data;
} Vector;

typedef struct {
    int         len;
} NetEnv;
```

```
// This one does not vectorize
void
mul(
    NetEnv*     ne,
    Vector*     rslt,
    Vector*     den,
    Vector*     num)
{
    float*      r;
    float*      d;
    float*      n;
    int i;

    r = rslt->data;
    d = den->data;
    n = num->data;

    for (i = 0; i < ne->len; ++i) {
        r[i] = n[i] * d[i];
    }
    return;
}
```

In this case, the compiler is unable to vectorize the loop at setting `O2`, the default.

If you determine it is safe to do so, you can modify the program code as follows:

```
typedef struct {
    float*      data;
} Vector;

typedef struct {
    int         len;
} NetEnv;

// This one vectorizes
void
mul(
    NetEnv*     ne,
    Vector*     rslt,
    Vector*     den,
    Vector*     num)
{
    float*      r;
    float*      d;
    float*      n;
    int i, local_len;
```

```
    r = rslt->data;
    d = den->data;
    n = num->data;

    local_len = ne->len;
    for (i = 0; i < local_len; ++i) {
        r[i] = n[i] * d[i];
    }
    return;
}
```

Verify

Confirm that the value of the upper-bound expression does not change throughout the entire execution of the loop.

See Also

GAP Message (Diagnostic ID 30531)

GAP Message (Diagnostic ID 30533)

Message

Compile with the %s option to vectorize and/or parallelize the loop at line %d.

Advice

Use the `[q or Q]opt-subscript-in-range` option for the specified file during compilation.

This option helps the compiler vectorize and parallelize the loop at the specified line. You must verify that the loops in the file do not contain very large integers and are not likely to generate very large integers in intermediate computations. A very large integer is loosely defined as follows: On an *n*-bit machine, a very large integer is typically >= 2^{n-2}. For example, on a 32-bit machine, a very large integer would be >= 2^{30}.

Example

Consider the following:

```
int f(int* A, int upper1, int upper2){
  long extra = 100.0;
  int return_val = 0;
  int val;
  for(int j=0; j < upper1; j++){
    for(int i = 0; i < upper2; i++){
      val = A[i*extra];
      return_val += val;
    }
  }
  return return_val;
}
```

If you determine it is safe to do so, compiling this example with the `[q or Q]opt-subscript-in-range` option results in vectorization of the innermost loop.

Verify

Confirm that no loop in the program contains or generates very large integers (typically very large integers are >= 2^{30}).

GAP Message (Diagnostic ID 30534)

Message

Add %s option for better type-based disambiguation analysis by the compiler if appropriate (the option will apply for the entire compilation). This will improve optimizations for the loop at line %d.

Advice

Use option `[Q]ansi-alias` for the specified file. This option will help the compiler to optimize the loop at the specified line. You must verify that the ANSI rules are followed for the entire file. gcc assumes this property by default in default setting `O2`; the Intel compiler does not. This option is particularly useful for C++ programs since it enables type-based disambiguation between pointers and other elemental datatypes (that in turn enables optimizations such as vectorization and parallelization).

Option `[Q]ansi-alias` enables or disables use of ANSI aliasing rules optimizations, and assert that the program adheres to these rules.

ANSI-aliasing rules are described in the Standards documentation:

- C: ISO/IEC 9899, chapter 6.5 paragraph 7
- C++: ISO/IEC 14882, chapter 3.10, paragraph 15

Example

Consider the following:

```
#include <stddef.h>

template<typename T>
class blocked_range {
    T my_begin;
    T my_end;
public:
    blocked_range();
    T begin() const {return my_begin;}
    T end() const {return my_end;}
};

class ApplyMatAdd {
    double *const A, *const B, *const C;
    const size_t size;
public:
    ApplyMatAdd(double *A_, double *B_, double *C_, size_t size_) : A(A_), B(B_), C(C_),
size(size_) {}
    void operator()( const blocked_range<size_t>& range ) const;
};

void ApplyMatAdd::operator()( const blocked_range<size_t>& range ) const {
    for (size_t i=range.begin(); i<range.end(); ++i) {
        for (size_t j=0; j<size; ++j) {
            C[i*size + j] = A[i*size + j] + B[i*size + j];
        }
    }
}
```

In this case, the compiler is unable to vectorize the innermost loop at setting `O2`, the default.

If you determine it is safe to do so, you can compile the above example with the `[Q]ansi-alias` option, which enables vectorization of the innermost loop.

Verify

Make sure that the semantics of this option are obeyed for the entire compilation.

GAP Message (Diagnostic ID 30535)

Message

Removing Exception-Handling code associated with the loop-body may enable more optimizations for the loop at line %d.

Advice

Loop optimizations could not be performed because of exception-handling code inside the loop body. You can remove the exception-handling code or use different libraries, etc.

Example

Consider the following:

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>

namespace ublas = boost::numeric::ublas;

int main () {
  unsigned size = 1000;
  ublas::vector<double> dest(size), src(size), arg(src);

  for (int i = 0; i < size; ++ i) {
    src(i) = i * 1.2;
    arg(i) = i * 2.3;
  }

  // Loop to be vectorized
  dest = src + 1.5 * arg;

  return 0;
};
```

In this case, the compiler is unable to vectorize the loop at setting `O2`, the default. Remove the exception-handling code or recode using different libraries.

Verify

Make sure that the restructured code without exception-handling code inside the loop-body follows original program semantics.

GAP Message (Diagnostic ID 30536)

Message

Add %s option for better type-based disambiguation analysis by the compiler, if appropriate (the option will apply for the entire compilation). This will improve optimizations such as vectorization for the loop at line %d.

Advice

Use option `-fnoargument-alias` (Linux* OS and macOS) or `/Qno-alias-args` (Windows* OS) for the specified file. This option will help the compiler to optimize the loop at the specified line. The user has to verify that there is no argument-aliasing for routines in this file before applying this option for the current file. This option is particularly useful for C++ programs since it enables type-based disambiguation between pointers that are passed in as arguments, which in turn enables optimizations such as vectorization and parallelization.

Option `-fargument-alias` (Linux* OS and macOS) and `/Qalias-args` (Windows* OS) enable or disable the C/C++ rule that function arguments may be aliased. When disabling the rule, you assert that this is safe.

Example

Consider the following example that demonstrates a violation of `-fnoargument-alias` or `/Qno-alias-args`:

```
void f(double *p, double *q, double *r) {
  int i;
  for (i = 0; i < n; i++)
    p[i] = q[i] + r[i];
}

int n, m;
double A[100], B[100];
...
f(&A[n], &A[m], &B[0]);
```

Since both pointers *p* and *q* will be pointing to the same array *A*, there may be overlap depending on the values of *n* and *m*.

Also, you cannot use the restrict keyword for parameters *p* and *q* in the function f for this test case.

You must analyze all the callers of function *f* in the current file and make sure that such overlap does not exist before applying `-fnoargument-alias` or `/Qno-alias-args` or the restrict qualifier. Note that such call sites may occur in other files as well.

Verify

Make sure that the semantics of this option is obeyed for the entire compilation.

Another way to get the same effect is to add the "restrict" keyword to each pointer-typed formal parameter of the routine "%s". This allows optimizations such as vectorization to be applied to the loop at line %d. Make sure that semantics of the "restrict" pointer qualifier is satisfied; in the routine, all data accessed through the pointer must not be accessed through any other pointer.

See Also

GAP Message (Diagnostic ID 30537)

GAP Message (Diagnostic ID 30537)

Message

Add the "restrict" keyword to each pointer-typed formal parameter of the routine "%s". This allows optimizations such as parallelization and vectorization to be applied to the loop at line %d.

Advice

Rather than using option -fnoargument-alias (Linux* OS and macOS) or /Qno-alias-args (Windows* OS), which affects the entire file, you can add the restrict qualifier to the pointer arguments to this routine. This change is more localized since it affects only the routines where the keyword is applied.

The restrict qualifier is part of C standard C99. This qualifier can be applied to a data pointer to indicate that during the scope of that pointer declaration, all data accessed through it will be accessed only through that pointer but not through any other pointer. So, the restrict keyword enables the compiler to perform certain optimizations based on the premise that a given object cannot be changed through another pointer. You must ensure that restrict-qualified pointers are used as they are intended to be used. Otherwise, undefined behavior may result.

The Intel® Compiler requires that you also specify option `[Q]restrict` when compiling non-C99 programs.

Example

Consider the following:

```
void matrix_mul_matrix(int  N, float * C,  float  *A,  float  *B) {
  int  i,j,k;

  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      C[i*N+j]=0;
      for(k=0;k<N;k++) {
        C[i*N+j]+=A[i*N+k] * B[k*N+j];
      }
    }
  }
}
```

In this case, the compiler is unable to apply loop optimizations such as loop-interchange and vectorization at default setting `O2`.

If you determine it is safe to do so, you can modify the program code as follows:

```
void matrix_mul_matrix(int  N, float * restrict C,  float  * restrict A,  float
 * restrict B) {
  int  i,j,k;

  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      C[i*N+j]=0;
      for(k=0;k<N;k++) {
        C[i*N+j]+=A[i*N+k] * B[k*N+j];
      }
    }
  }
}
```

Note that instead of using the restrict qualifier, you could have specified `-fnoargument-alias` or `/Qno-alias-args` before compiling the code.

Verify

Make sure that semantics of the "restrict" pointer qualifier is satisfied: in the routine, all data accessed through the pointer must not be accessed through any other pointer.

See Also

GAP Message (Diagnostic ID 30536)

GAP Message (Diagnostic ID 30538)

Message

Moving the block of code that consists of a function-call (line %d), if-condition (line %d), and an early return (line %d) to outside the loop may enable parallelization of the loop at line %d.

Advice

Move the function call and an associated return from inside the loop (perhaps by inserting them before the loop) to help parallelize the loop.

This kind of function-leading-to-return inside a loop usually handles some error-condition inside the loop. If this error check can be done before starting the execution of the loop without changing the program semantics, the compiler may be able to parallelize the loop thus improving performance.

Example

Consider the following:

```
extern int num_nodes;
typedef struct TEST_STRUCT {
    // Coordinates of city1
    float latitude1;
    float longitude1;

    // Coordinates of city2
    float latitude2;
    float longitude2;
} test_struct;

extern int *mark_larger;
extern float *distances, **matrix;
extern test_struct** nodes;
extern test_struct ***files;
extern void init_node(test_struct *node, int i);
extern void process_nodes(void);
float compute_max_distance(void);

extern int check_error_condition(int width);

#include <math.h>
#include <stdio.h>

void process_nodes(int width)
{
  float const R = 3964.0;
  float temp, lat1, lat2, long1, long2, result, pat2;
  int m, j, temp1 = num_nodes;

      nodes = files[0];
      m = 1;

#pragma loop count min(4)
#pragma parallel
      for (int k=0; k < temp1; k++) {

            if (check_error_condition(width)) {
```

```
                return;
            }

         lat1 = nodes[k]->latitude1;
            lat2 = nodes[k]->latitude2;

            long1 = nodes[k]->longitude1;
            long2 = nodes[k]->longitude2;

            // Compute the distance between the two cities
            temp = sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) *
                                              cos(long1-long2);
            result = 2.0 * R * atan(sqrt((1.0-temp)/(1.0+temp)));

            pat2 = 0;
            for(j=0; j<width; j++) {
              pat2 += distances[j];
              matrix[k][j] = distances[k]+j;
            }
            // Store the distance computed in the distances array
            if (result > distances[k]) {
                distances[k] = result + pat2;
            }
      }
}
```

In this case, the compiler is unable to parallelize the loop at line 38.

If you determine it is safe to do so, you can modify the above code as follows:

```
extern int num_nodes;
typedef struct TEST_STRUCT {
    // Coordinates of city1
    float latitude1;
    float longitude1;

    // Coordinates of city2
    float latitude2;
    float longitude2;
} test_struct;

extern int *mark_larger;
extern float *distances, **matrix;
extern test_struct** nodes;
extern test_struct ***files;
extern void init_node(test_struct *node, int i);
extern void process_nodes(void);
float compute_max_distance(void);

extern int check_error_condition(int width);

#include <math.h>
#include <stdio.h>

void process_nodes(int width) {
  float const R = 3964.0;
  float temp, lat1, lat2, long1, long2, result, pat2;
  int m, j, temp1 = num_nodes;
```

```
      nodes = files[0];
      m = 1;

      if (check_error_condition(width)) {
      return;
      }

#pragma loop count min(4)
#pragma parallel
      for (int k=0; k < temp1; k++) {

         lat1 = nodes[k]->latitude1;
            lat2 = nodes[k]->latitude2;

            long1 = nodes[k]->longitude1;
            long2 = nodes[k]->longitude2;

            // Compute the distance between the two cities
            temp = sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) *
                                               cos(long1-long2);
            result = 2.0 * R * atan(sqrt((1.0-temp)/(1.0+temp)));

            pat2 = 0;
            for(j=0; j<width; j++) {
              pat2 += distances[j];
              matrix[k][j] = distances[k]+j;
            }
            // Store the distance computed in the distances array
            if (result > distances[k]) {
                distances[k] = result + pat2;
            }
      }
}
```

Verify

Confirm that the function call does not rely on any computation inside the loop and that restructuring the code as suggested above, retains the original program semantics.

GAP Message (Diagnostic ID 30753)

Message

Convert array of struct "%s" into a new struct whose fields are arrays of the corresponding fields in the original struct. This improves performance due to better data locality.

Advice

You should apply full peeling to a class or structure. This is done by splitting a class or structure into separate fields. This should improve performance by better utilizing the processor cache. This message is generated only when the entire application is built with Interprocedural Optimization (IPO). This transformation requires that you change all access to any peeled structure and its fields in the entire application. In some cases, it may not be easy to change source code to apply full peeling.

Example

Consider the following:

```
// peel.c
#include <stdio.h>
#include <stdlib.h>

#define N 100000
int a[N];
double b[N];
struct S3 {
    int *pi;
    double d;
    int j;
};

struct S3 *sp;

void init_hot_s3_i() {

    int ii = 0;

    for (ii = 0; ii < N; ii++) {
        sp[ii].pi = &a[ii];
    }
}
void init_hot_s3_d() {
    int ii = 0;

    for (ii = 0; ii < N; ii++) {
        sp[ii].d = b[ii];
    }
}
void init_hot_s3_j() {
    int ii = 0;

    for (ii = 0; ii < N; ii++) {
        sp[ii].j = 0;
    }
}
void dump_s3() {
    int ii;

    for (ii = 0; ii < N; ii++) {
        printf("i= %d ", *(sp[ii].pi));
        printf("d= %g \n", sp[ii].d);
        printf("j= %g \n", sp[ii].j);
    }
}

main() {

   sp = (struct S3 *)calloc(N, sizeof(struct S3));
   init_hot_s3_i();
   init_hot_s3_d();
   init_hot_s3_j();
   dump_s3();
}
```

In this case, the compiler tells you to convert struct "S3".

If you determine it is safe to do so, you can modify the program code as follows:

```
#include <stdio.h>
#include <stdlib.h>

#define N 100000
int a[N];
double b[N];
struct S3 {
    int *pi;
};
struct new_d {
    double d;
};
struct new_j {
    int j;
};

struct S3 *sp;
struct new_d *sp_d;
struct new_j *sp_j;

void init_hot_s3_i() {

    int ii = 0;

    for (ii = 0; ii < N; ii++) {
        sp[ii].pi = &a[ii];
    }
}
void init_hot_s3_d() {
    int ii = 0;

    for (ii = 0; ii < N; ii++) {
        sp[ii].d = b[ii];
    }
}
void init_hot_s3_j() {
    int ii = 0;

    for (ii = 0; ii < N; ii++) {
        sp[ii].j = 0;
    }
}
void dump_s3() {
    int ii;

    for (ii = 0; ii < N; ii++) {
        printf("i= %d ", *(sp[ii].pi));
        printf("d= %g \n", sp[ii].d);
        printf("j= %g \n", sp[ii].j);
    }
}

main() {

   sp = (struct S3 *)calloc(N, sizeof(struct S3));
```

```
   init_hot_s3_i();
   init_hot_s3_d();
   init_hot_s3_j();
   dump_s3();
}
```

Verify

Make sure that the restructured code satisfies the original program semantics.

GAP Message (Diagnostic ID 30754)

Message

Aligning the fields '%s' in the structure '%s' on an 8-byte boundary may improve performance. Default alignment of double precision floating point data is 4-byte on the Linux IA32 platform. [ALTERNATIVE] Reordering fields of the structure may help to align double precision floating point data on an 8-byte boundary. [ALTERNATIVE] Another way is to use __attribute__((aligned(8))) for the fields '%s' in the structure '%s' to allocate the fields on an 8-byte boundary.

This messsage is only available on Linux* systems.

Advice

You must reorder the fields of a class or structure type to make "double" fields 8-byte aligned. On Linux* systems on IA-32 architecture, "double" fields are not required to be 8-byte aligned. This should enable optimizations like vectorization to generate better code. You must verify that the application code does not rely on the structure fields to be laid out in a specific order.

Example

Consider the following:

```
//alignment.c
#include <stdlib.h>
#include <stdio.h>

#define N 1000

struct S {
    int i;
    double d1;
    double d2;
    double d3;
};

struct S *sp;

static struct S*
alloc_s(int num) {
    struct S * temp;

    temp = calloc(num, sizeof(struct S));
    return temp;
}

struct S temp;
```

```
static void
swap_s(int i, int j) {
    memcpy(&temp, sp + i, sizeof(struct S));
    memcpy(sp + i, sp + j, sizeof(struct S));
    memcpy(sp+ j, &temp, sizeof(struct S));
}

static void
init_s(int num) {
    int ii;

    for (ii = 0; ii < num; ii++) {
        sp[ii].i = ii;
        sp[ii].d1 = (double) ii + 1;
        sp[ii].d2 = (double) ii + 2;
        sp[ii].d3 = (double) ii + 3;
    }
}

main() {
    int ii;
    double d = 0.0;

    sp = alloc_s(N);

    for(ii = 0; ii < N -1; ii += 2) {
        swap_s(ii, ii+1);
    }

    for (ii = 0; ii < N ; ii++) {
        sp[ii].d1 = sp[ii].d1 * sp[ii].d2 * sp[ii].d3;
        d += sp[ii].d1;
    }

    for (ii = 0; ii < N ; ii++) {
        printf(" %d:  %g   %g  %g  \n", sp[ii].i, sp[ii].d1, sp[ii].d2, sp[ii].d3);
    }
}
```

In this case, when the program is compiled, the compiler generates a message saying that aligning the fields 'd1, d2, d3' in the structure 'S' on an 8-byte boundary may improve performance.

Alternatively, '__attribute__((aligned(8)))' can be used to align 'd1, d2, d3' on an 8-byte boundary. One possible way to do this is shown below:

```
struct S {
    int i;
    __attribute__((aligned(8)))  double d1;
    double d2;
    double d3;
};
```

Verify

Make sure that the restructured code satisfies the original program semantics. Note that size of the structure may change due to the alignment changes. Make sure that the change in the structure layout satisfies the original program semantics.

GAP Message (Diagnostic ID 30755)

Message

Reordering the fields of the structure '%s' will improve data locality. Suggested field order: '%s'.

Advice

You should reorder the fields of the class or structure type in the specified order. This should improve performance by better utilizing the processor cache.

You must verify that the application code does not rely on the structure fields to be laid out in a specific order. For example, if the application code uses the address of a field to access other fields, it may stop working once the field reordering is applied. Note also that such code is not considered valid.

Example

Consider the following:

```
//field_reord.c
struct str {
    int a1, b1, carr[100], c1, d1, e1;
};

extern struct str sp[];

int hot_func1() {
    int i, ret = 0;

    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret += sp[i].c1;
    }
    return ret;
}

int hot_func2() {
    int ret = 0, i;
    for (i = 0; i < 100000; i++) {
        ret += sp[i].a1;
        ret -= sp[i].e1;
    }
    return ret;
}

int hot_func3() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].carr[10];
    }
    return ret + sp[0].b1 + sp[0].d1;
}
```

In this case, when the program is compiled, the compiler generates a message saying that reordering the fields of the structure 'str' will improve data locality and that the suggested field order is 'a1, c1, e1, carr, b1, d1'.

For the above example, the only changes in `field_reord.c` to reorder fields of the structure 'str' as advised are the following:

```
//field_reord.c
struct str {
    int a1, c1, e1, carr[100], b1, d1;
};
...
```

Verify

The suggestion is based on the field references in the current compilation. Please make sure that the restructured code satisfies the original program semantics.

GAP Message (Diagnostic ID 30756)

Message

Split the structure '%s' into two parts to improve data locality. Frequently accessed fields are '%s'; performance may improve by putting these fields into one structure and the remaining fields into another structure. Alternatively, performance may also improve by reordering the fields of the structure. Suggested field order: '%s'.

Advice

This message is issued when both structure splitting and field reordering transformations are applicable. Structure splitting transformation is expected to lead to higher performance gains if the transformation can be successfully applied. However, field reordering transformation is usually simple enough to apply, but the downside is that the performance gain seen may be lower.

You must verify that the structure meets the requirements for applying the splitting or reordering transformation. Some of these requirements are described in the description of these individual transformations.

Example

Consider the following:

```
//str_split_reord.c
struct str {
    int a1, b1, carr[100], c1, e1;
};

#define N 1000000

struct str *sp;

void allocate_str_mem() {
    sp = malloc(N * sizeof(struct str));
}

int hot_func1() {
    int i, ret = 0;

    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret += sp[i].c1;
    }
    sp->carr[0] = ret;
```

```
    return ret;
}

int hot_func2() {
    int ret = 0, i;
    for (i = 0; i < 100000; i++) {
        ret += sp[i].a1;
        ret -= sp[i].e1;
    }
    return ret;
}

int hot_func3() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].b1;
    }
    return ret;
}
```

In this case, a message is displayed that is similar to the following:

drive: program-name: remark #30756: (DTRANS) Split the structure 'str' into two parts to improve data locality. Frequently accessed fields are 'a1, b1, c1'; performance may improve by putting these fields into one structure and the remaining fields into another structure. Alternatively, performance may also improve by reordering the fields of the structure. Suggested field order: 'a1, c1, e1, b1, carr'. ...(etc.)

If you determine it is safe to do so, you can modify the program code as shown below to split the structure 'str'. Other references to structure 'str' that are not in the current module should also be similarly modified.

```
struct str_cold {
    int carr[100], e1;
};

struct str {
    int a1, b1, c1; struct str_cold *cold_ptr;
};

#define N 1000000

struct str *sp;

void allocate_str_mem()
{
    struct str *temp;
    struct str_cold *cold_begin;
    int index;

    temp = malloc(N * sizeof(struct str) + N * sizeof(struct str_cold));
    sp = temp;
    cold_begin = (struct str_cold *) (temp + N);
    for(index = 0; index < N; index++) {
       temp[index].cold_ptr = cold_begin + index;
    }
}

int hot_func1() {
    int i, ret = 0;

    for (i = 0; i < 1000000; i++) {
```

```
        ret += sp[i].a1;
        ret += sp[i].c1;
    }
    sp->cold_ptr->carr[0] = ret;
    return ret;
}

int hot_func2() {
    int ret = 0, i;
    for (i = 0; i < 100000; i++) {
        ret += sp[i].a1
        ret -= sp[i].cold_ptr->e1;
    }
    return ret;
}

int hot_func3() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].b1;
    }
    return ret;
}
```

For the above example, the only source change required to reorder fields in structure 'str' as alternatively suggested are the following:

```
//str_split_reord.c
struct str {
    int a1, c1, e1, b1, carr[100];
};
...
```

Verify

The suggestion is based on the field references in the current compilation. Please make sure that the restructuring is applied to field references in all source files of the application, and that the restructured code satisfies the original program semantics.

GAP Message (Diagnostic ID 30757)

Message

Remove unused field(s) '%s' from the struct '%s'.

This message is emitted only with whole-program recognition.

Advice

Some unused fields were seen in a class or structure type. If the unused fields can be removed from the structure definition, it will lead to reduced memory usage and better cache utilization since the cache will no longer be filled with unused data.

The advice is based on the analysis of the source code that is seen. You must verify that the fields that are reported as unused are not accessed elsewhere in the application. You also need to be careful when removing unused fields if the code relies on the structure fields to be laid out in a specific order. As an example, if the application code uses the address of a field to access other fields, it may stop working once unused fields are removed. Note that such code is not considered valid in the first place.

Example

Consider the following:

```
//unused_field_1.c
struct str {
    int a1, b1, c1, d1, e1;
};

struct str sp[1000000];
 int hot_func1() {
    int i, ret = 0;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret += sp[i].b1;
    }
    return ret;
}

int hot_func2() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret -= sp[i].c1;
    }
    return ret;
}

int hot_func3() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].d1;
    }
    return ret;
}

main() {
    hot_func1();
    hot_func2();
    hot_func3();
}
```

In this case, if the unused fields can be removed, the only source change needed would be the following:

```
//unused_field_1.c
struct str {
    int a1, b1, c1, d1;
};
...
```

Verify

Make sure that the restructured code satisfies the original program semantics.

See Also

GAP Message (Diagnostic ID 30758)

GAP Message (Diagnostic ID 30758)

Message

Remove unused field(s) "%s" from the struct "%s".

This message is emitted even without whole-program recognition in advanced mode.

Advice

Some unused fields were seen in a class or structure type. If the unused fields can be removed from the structure definition, it will lead to reduced memory usage and better cache utilization since the cache will no longer be filled with unused data.

The advice is based on the analysis of the source code that is seen. You must verify that the fields that are reported as unused are not accessed elsewhere in the application. You also need to be careful when removing unused fields if the code relies on the structure fields to be laid out in a specific order. As an example, if the application code uses the address of a field to access other fields, it may stop working once unused fields are removed. Note that such code is not considered valid in the first place.

Example

Consider the following:

```
//unused_field_2.c
struct str {
    int a1, b1, c1, d1, e1;
};

extern struct str sp[];

int hot_func1() {
    int i, ret = 0;

    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret += sp[i].b1;
    }
    return ret;
}

int hot_func2() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret -= sp[i].c1;
    }
    return ret;
}

int hot_func3() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].d1;
    }
    return ret;
}
```

In this case, if the unused fields can be removed, the only source change needed would be the following:

```
//unused_field_2.c
struct str {
    int a1, b1, c1, d1;
};
...
```

Verify

The suggestion is based on the field references in the current compilation. Please make sure that there are no references to these fields across the entire application.

See Also

GAP Message (Diagnostic ID 30757)

GAP Message (Diagnostic ID 30759)

Message

Remove unused field(s) '%s' from the struct '%s'. The fields: '%s' were conservatively assumed by the compiler as referenced since their address is taken.

This message is emitted only with whole-program recognition.

Advice

Some unused fields were seen in a class or structure type. If the unused fields can be removed from the structure definition, it will lead to reduced memory usage and better cache utilization since the cache will no longer be filled with unused data.

You must verify that the fields that are reported as unused are not accessed elsewhere in the application. You also need to be careful when removing unused fields if the code relies on the structure fields to be laid out in a specific order. For example, if the application code uses the address of a field to access other fields, it may stop working once unused fields are removed. Note that such code is not considered valid in the first place.

The unused field analysis considers address taken fields as used. It will report address taken fields also when reporting any unused fields.

Example

Consider the following:

```
//unused_field_3.c
struct str {
    int a1, b1, c1, d1, e1, f1;
};

struct str sp[1000000];

int hot_func1() {
    int i, ret = 0;

    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret += sp[i].b1;
    }
    return ret;
}
```

```
int hot_func2() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret -= sp[i].c1;
    }
    return ret;
}

int *gip;

int hot_func3() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].d1;
    }
    gip = &(sp->f1);
    return ret;
}

int main() {
    hot_func1();
    hot_func2();
    hot_func3();
}
```

The above code will cause a message to be displayed that is similar to the following:

program-name: remark #30759: (DTRANS) Remove unused field(s) 'e1' from the struct 'str'. The fields: 'f1' were conservatively assumed by the compiler as referenced since their address is taken... (etc.)

In this case, if the unused fields can be removed, the only source change needed would be the following:

```
//unused_field_3.c
struct str {
    int a1, b1, c1, d1, f1;
};
...
```

Verify

Make sure that the restructured code satisfies the original program semantics.

See Also

GAP Message (Diagnostic ID 30760)

GAP Message (Diagnostic ID 30760)

Message

Remove unused field(s) '%s' from the struct '%s'. The fields: '%s' were conservatively assumed by the compiler as referenced since their address is taken.

This message is emitted even without whole-program recognition in advanced mode.

Advice

Some unused fields were seen in a class or structure type. If the unused fields can be removed from the structure definition, it will lead to reduced memory usage and better cache utilization since the cache will no longer be filled with unused data.

You must verify that the fields that are reported as unused are not accessed elsewhere in the application. You also need to be careful when removing unused fields if the code relies on the structure fields to be laid out in a specific order.

For example, if the application code uses the address of a field to access other fields, it may stop working once unused fields are removed. Note that such code is not considered valid in the first place.

The unused field analysis considers address taken fields as used. It will report address taken fields also when reporting any unused fields.

Example

Consider the following:

```
//unused_field_4.c
struct str {
    int a1, b1, c1, d1, e1, f1;
};

extern struct str sp[];

int hot_func1() {
    int i, ret = 0;

    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret += sp[i].b1;
    }
    return ret;
}

int hot_func2() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].a1;
        ret -= sp[i].c1;
    }
    return ret;
}

int *gip;

int hot_func3() {
    int ret = 0, i;
    for (i = 0; i < 1000000; i++) {
        ret += sp[i].d1;
    }

    gip = &(sp->f1);
    return ret;
}
```

The above code will cause a message to be displayed that is similar to the following:

drive: program-name: remark #30760: (DTRANS) Remove unused field(s) 'e1' from the struct 'str'. The fields: 'f1' were conservatively assumed by the compiler as referenced since their address is taken. ...(etc.)

In this case, the if the unused fields can be removed, the only source change needed would be the following:

```
//unused_field_4.c
struct str {
    int a1, b1, c1, d1, f1;
};
...
```

Verify

The suggestion is based on the field references in the current compilation. Please make sure that there are no references to these fields across the entire application.

See Also

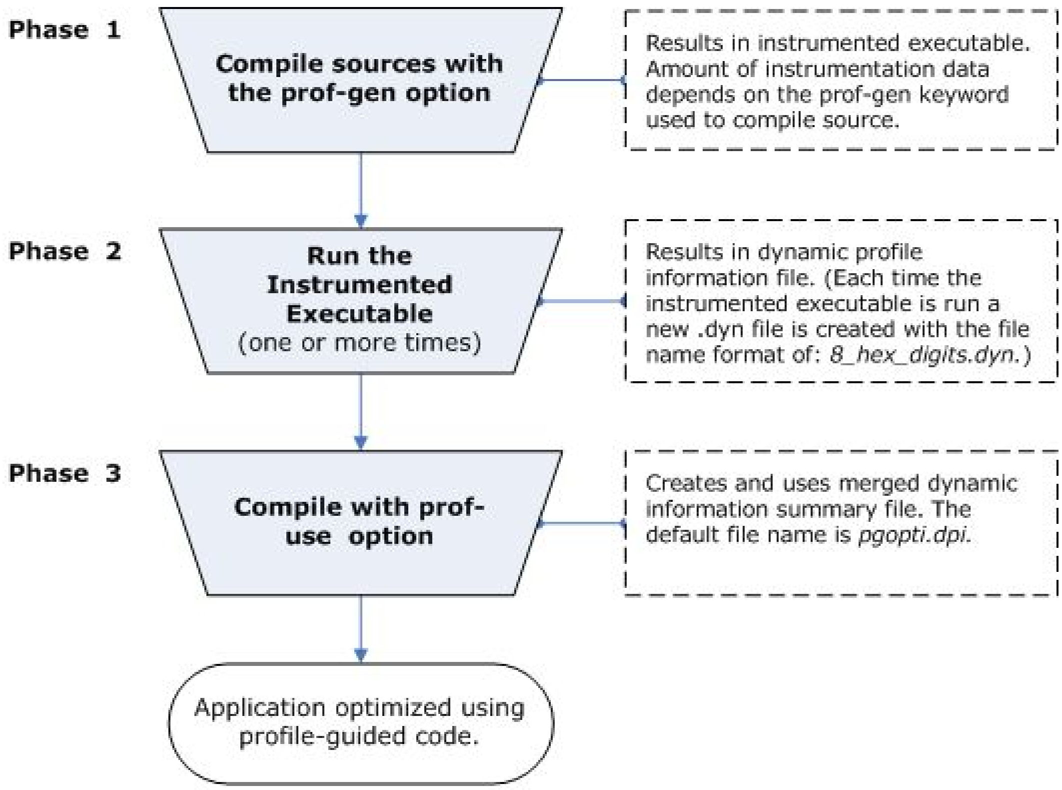
GAP Message (Diagnostic ID 30759)

Profile-Guided Optimization (PGO)

Profile-guided Optimization (PGO) improves application performance by shrinking code size, reducing branch mispredictions, and reorganizing code layout to reduce instruction-cache problems. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

PGO consists of three phases or steps.

1. Instrument the program. The compiler creates and links an instrumented program from your source code and special code from the compiler.
2. Run the instrumented executable. Each time you execute the instrumented code, the instrumented program generates a dynamic information file, which is used in the final compilation.
3. Final compilation. When you compile a second time, the dynamic information files are merged into a summary file. Using the summary of the profile information in this file, the compiler attempts to optimize the execution of the most heavily traveled paths in the program.



See Profile-guided Optimization Options for information about the supported options and Profile an Application for specific details about using PGO from the command line.

PGO provides the following benefits:

- Use profile information for register allocation to optimize the location of spill code.
- Improve branch prediction for indirect function calls by identifying the most likely targets. Some processors have longer pipelines, which improves branch prediction and translates into high performance gains.
- Detect and do not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

Interprocedural optimization (IPO) and PGO can affect each other; using PGO can often enable the compiler to make better decisions about inline function expansion, which increases the effectiveness of interprocedural optimizations. Unlike other optimizations, such as those strictly for size or speed, the results of IPO and PGO vary. This variability is due to the unique characteristics of each program, which often include different profiles and different opportunities for optimizations.

Performance Improvements with PGO

PGO works best for code with many frequently executed branches that are difficult to predict at compile time. An example is the code with intensive error-checking in which the error conditions are false most of the time. The infrequently executed (cold) error-handling code can be relocated so the branch is rarely predicted incorrectly. Minimizing cold code interleaved into the frequently executed (hot) code improves instruction cache behavior.

When you use PGO, consider the following guidelines:

- Minimize changes to your program after you execute the instrumented code and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated. If you modify your program, the compiler can issue a warning that the dynamic information does not correspond to a modified function when PGO remarks are enabled or found in the optimization report.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.
- Know the sections of your code that are the most heavily used. If the data set provided to your program is very consistent and displays similar behavior on every execution, then PGO can probably help optimize your program execution.
- Different data sets can result in different algorithms being called. The difference can cause the behavior of your program to vary for each execution. In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits. If it takes multiple data sets to accurately characterize application performance, execute the application with all data sets then merge the dynamic profiles; this technique should result in an optimized application.

You must insure that the benefit of the profiled information is worth the effort required to maintain up-to-date profiles.

Profile-Guided Optimization via Hardware Counters

A lightweight profiling mechanism can be used to achieve many of the benefits of instrumentation-based profiling, but without the overhead of inserting instrumentation into the application binary. This mode of operation can be beneficial in cases where increased code/data size or changes in runtime due to instrumentation may make regular Performance-Guided Optimization (PGO) infeasible. This approach requires the use of Intel® VTune™ Profiler to collect information from the hardware counters. The information is collected with minimal overhead, and combined with debug information produced by the compiler to identify the primary code path for optimizations.

Follow these steps to use this method:

1. Compile the application with the option `prof-gen-sampling`.

   This option instructs the compiler to generate additional debug information for the application, which is used to map the information collected by the hardware counters to a specific source code. Using this option does not affect the generated instruction sequence in the way that instrumented PGO does. Optimizations may be enabled during this build, but it is recommended that you disable function inlining.
2. Run the generated executable on one or more representative workloads with the Intel VTune Profiler tool:

```
<installation-root>/bin64/amplxe-pgo-report.sh <your application and command line>
```

   Additional information regarding options for data collection can be found in the Intel VTune Profiler documentation. This step generates files in the form: `r`*`NNN`*`pgo_icc.pgo` (where *`NNN`* is a three digit number). These files are used as input in the next steps.
3. Merge the report files produced during step 2.

   The tool `profmergesampling` can be used to produce an indexed file of results that speeds up processing data during the next step.

```
profmergesampling -file <input-file[:input_file]*> -out <output_name>
```

4. Compile the application with the option `prof-use-sampling:input-file[:input_file]*`

   In this step, one or more result files produced during step 2 (or an indexed file from step 3) can be fed into the compiler to direct the optimizations.

See Also

prof-gen-sampling
  compiler option

prof-use-sampling
  compiler option

Profile an Application with Instrumentation

Profiling an application includes the following three phases:

- Instrumentation compilation and linking
- Instrumented execution
- Feedback compilation

This topic provides detailed information on how to profile an application by providing sample commands for each of the three phases (or steps).

1. Instrumentation compilation and linking

Use `[Q]prof-gen` to produce an executable with instrumented information included. Use `/Qcov-gen` (Windows) option to obtain minimum instrumentation only for code coverage.

Operating System	Commands
Linux and macOS	`icpc -prof-gen -prof-dir/usr/profiled a1.cpp a2.cpp a3.cpp` `icpc a1.o a2.o a3.o`
Windows	`icl /Qprof-gen /Qprof-dirc:\profiled a1.cpp a2.cpp a3.cpp` `icl a1.obj a2.obj a3.obj`
Windows	`icl /Qcov-gen /Qcov-dirc:/cov_data a1.cpp a2.cpp a3.cpp` `icl a1.obj a2.obj a3.obj`

Use the `[Q]prof-dir` or `/Qcov-dir` (Windows) option if the application includes the source files located in multiple directories; using the option insures the profile information is generated in one consistent place. The example commands demonstrate how to combine these options on multiple sources.

The compiler gathers extra information when you use the `-prof-gen=srcpos` (Linux and macOS) or `/Qprof-gen:srcpos` (Windows) option; however, the extra information is collected to provide support for specific Intel tools, including the code coverage Tool. If you do not expect to use such tools, do not specify `-prof-gen=srcpos` (Linux and macOS) or `/Qprof-gen:srcpos` (Windows); the extended option does not provide better optimization and could slow parallel compile times. If you are interested in using the instrumentation only for code coverage, use the `/Qcov-gen` (Windows) option, instead of the `/Qprof-gen:srcpos` (Windows) option, to minimize instrumentation overhead.

PGO data collection is optimized for collecting data on serial applications at the expense of some loss of precision on areas of high parallelism. However, you can specify the `threadsafe` keyword with the `-prof-gen` (Linux* and macOS) or the `/Qprof-gen` (Windows) compiler option for files or applications that contain parallel constructs using OpenMP* features, for example. Using the `threadsafe` keyword produces instrumented object files that support the collection of PGO data on applications that use a high level of parallelism but may increase the overhead for data collection.

NOTE
Unlike serial programs, parallel programs using OpenMP* may involve dynamic scheduling of code paths, and counts collected may not be perfectly reproducible for the same training data set.

2. Instrumented execution

Run your instrumented program with a representative set of data to create one or more dynamic information files.

Operating System	Command
Linux and macOS	`./a1.out`
Windows	`a1.exe`

Executing the instrumented applications generates a dynamic information file that has a unique name and .dyn suffix. A new dynamic information file is created every time you execute the instrumented program.

You can run the program more than once with different input data.

By default, the `.dyn` filename follows this naming convention: `<timestamp>_<pid>.dyn`. The `.dyn` file is either placed into a directory specified by an environment variable, a compile-time specified directory, or the current directory.

To make it easy to distinguish files from different runs, you can specify a prefix for the `.dyn` filename in the environment variable, `INTEL_PROF_DYN_PREFIX`. In such a case, executing the instrumented application generates a `.dyn` filename as follows: `<prefix>_<timestamp>_<pid>.dyn`, where `<prefix>` is the identifier that you have specified. Be sure to set the `INTEL_PROF_DYN_PREFIX` environment variable prior to starting your instrumented application.

NOTE
The value specified in `INTEL_PROF_DYN_PREFIX` environment variable must not contain `< > : " / \ | ? *` characters. The default naming scheme will be used if an invalid prefix is specified.

3. Feedback compilation

Before this step, copy all .dyn and .dpi files into the same directory. Compile and link the source files with `[Q]prof-use`; the option instructs the compiler to use the generated dynamic information to guide the optimization:

Operating System	Examples
Linux and macOS	`icpc -prof-use -ipo -prof-dir/usr/profiled a1.cpp a2.cpp a3.cpp`
Windows	`icl /Qprof-use /Qipo /Qprof-dir:c:\profiled a1.cpp a2.cpp a3.cpp`

This final phase compiles and links the sources files using the data from the dynamic information files generated during instrumented execution (phase 2).

In addition to the optimized executable, the compiler produces a pgopti.dpi file.

Most of the time, you should specify the default optimizations,`O2`, for phase 1, and specify more advanced optimizations, `[Q]ipo`, during the phase 3 (final) compilation. The example shown above used`O2` in step 1 and`[Q]ipo` in step 3.

NOTE
The compiler ignores the `[Q]ipo` or `[Q]ip` option during phase 1 with `[Q]prof-gen`.

Profile-Guided Optimization Report

The PGO report can help identify where and how the compiler used profile information to optimize the source code. The PGO report can also identify where profile information was discarded due to source code changes made between the time of instrumentation and feedback steps. The PGO report is most useful when combined with the PGO compilation steps outlined in the topic, Profile an Application with Instrumentation. Without the profiling data generated during the application profiling process the report will generally not provide useful information.

Combine the final PGO step with the reporting options by including `-prof-use` (Linux* and macOS) or `/Qprof-use` (Windows*). The following syntax examples demonstrate how to run the report using the combined options.

Operating System	Syntax Examples
Linux*	`icpc -prof-use -qopt-report-phase=pgo pgotools_sample.c`
macOS	`icpc -prof-use -qopt-report-phase=pgo pgotools_sample.c`
Windows*	`icl /Qprof-use /Qopt-report-phase:pgo pgotools_sample.c`

By default the PGO report generates a medium level of detail (where the `[q or Q]opt-report` argument *n*=2). You can use the `-qopt-report=`*n* (Linux and macOS) or `/Qopt-report:`*n* option along with the `[q or Q]opt-report-phase` option if you want a greater or lesser level of diagnostic detail.

The output, by default, comes out to a file with the same name as the object file but with an `.optrpt` extension and is written into the same directory as the object file. Using the entries in the example above, the output file will be `pgotools_sample.optrpt`. Use the `-qopt-report-file` (Linux and macOS) or the `/Qopt-report-file` (Windows) option to specify any other name for the output file that captures the report results, or to specify that the output should go to `stdout` or `stderr`.

See Also

`qopt-report-phase`, `Qopt-report-phase`
 compiler option

`qopt-report`, `Qopt-report`
 compiler option

`qopt-report-file`, `Qopt-report-file`
 compiler option

`prof-use`, `Qprof-use`
 compiler option

Profile an Application

PGO API Support

The Profile-Guided Optimizations (PGO) API lets you control the generation of profile information during the instrumented execution phase of profile-guided optimizations.

A set of functions and an environment variable comprise the PGO API. The remaining topics in this section describe the associated functions and environment variables.

The compiler sets a `define` for the `_PGO_INSTRUMENT` pre-processor macro when you compile with `[Q]prof-gen` options, to instrument your code. Without instrumentation, the PGO API functions cannot provide PGO API support.

Normally, profile information is generated by an instrumented application when it terminates by calling the standard `exit()` function.

To ensure that profile information is generated, the functions described in this section may be necessary or useful in the following situations:

- The instrumented application exits using a non-standard exit routine.
- The instrumented application is a non-terminating application: `exit()` is never called.
- The application requires control of when the profile information is generated.

You can use the PGO API functions in your application by including the `pgouser.h` header file at the top of any source file where the functions may be used.

Example

```
#include <pgouser.h>
```

NOTE
You do not need to remove the PGO API functions from your code when you have completed the instrumentation step. Changes to the source code at this stage could inhibit obtaining profile data feedback on the routines that were modified. For the instrumentation step (using `-[Q]prof-gen` or `-[Q]prof-gen:<aug>` option, where <aug> can be srcpos, globdata, or default), the definition for the `_PGO_INSTRUMENT` macro is automatically set, allowing instrumentation library routines to be used. For the production step (using `-[Q]prof-use` option), the definition for the `_PGO_INSTRUMENT` macro is removed, allowing profile data to be fed back.

The Profile IGS Environment Variables

The environment variable for PGO API is `INTEL_PROF_DUMP_INTERVAL`. This environment variable may be used to initiate Interval Profile Dumping in an instrumented user application.

The environment variable `INTEL_PROF_DUMP_CUMULATIVE` can be used to provide additional control over the internal profiling dumping behavior.

The environment variable, `INTEL_PROF_DYN_PREFIX`, allows specifying a prefix string that is used for naming the `.dyn` files. If this variable is defined then the `.dyn` files will be named as `<prefix>_<timestamp>_<pid>.dyn`, instead of the default naming convention of `<timestamp>_<pid>.dyn`. This can be useful for identifying `.dyn` files produced by specific training sets.

See Also

Supported Environment Variables
Interval Profile Dumping
[Q]prof-gen

Resetting Profile Information

The `_PGOPTI_Prof_Reset_All()` function clears the profile information collected by the instrumented application. The prototype of the function call is listed below.

During a run of an instrumented executable, a segment is maintained for each routine executed during that run, storing the following information:

- Number of times each basic block in the routine is executed
- Specific values observed at each data point undergoing value-profiling
- Number of times each of these values are observed

When the _PGOPTI_Prof_Reset() function is called, the basic block execution count for each basic block is set to 0, and the value-profiling information is reset to indicate that no values were observed at any data point.

Syntax
`void _PGOPTI_Prof_Reset_All(void);`

The older version of this function, `_PGOPTI_Prof_Reset()`, is deprecated.

When `_PGOPTI_Prof_Reset_All()` function is called, it insures that all the counters within the main application and all the instrumented shared libraries are cleared. All the counters for block execution counts and value profiling information is reset to 0.

NOTE
For routines that were in progress when the reset call was made, the counts for portions of the routine that executed following the call to the reset function may have higher counts than portions of the routine that executed prior to the reset call.

Dumping Profile Information

The `_PGOPTI_Prof_Dump_All()` function dumps the profile information collected by the instrumented application. The prototype of the function call is listed below.

Syntax
`void _PGOPTI_Prof_Dump_All(void);`

An older version of this function, `_PGOPTI_Prof_Dump()`, is deprecated and no longer used.

The new function, `_PGOPTI_Prof_Dump_All()`, operates much like the deprecated function, except on Linux* operating systems, when it is used in connection with shared libraries (.so). When `_PGOPTI_Prof_Dump_All()` is called, it insures that a .dyn file is created for all shared libraries needing to create a .dyn file. Use `_PGOPTI_Prof_Dump_All()` on Linux OS to insure portability and correct functionality.

The profile information is generated in a `.dyn` file (generated in phase 2 of PGO).

The environment variables that affect the `_PGOPTI_Prof_Dump_All()` function are `PROF_DIR`, `COV_DIR`, `PROF_DPI`, `and COV_DPI`. Set the `PROF_DIR or COV_DIR` environment variable to specify the directory to which the .dyn file must be stored. Alternately, you can use the `-[Q]prof_dir` compiler option, when building with `-[Q]prof-gen`, to specify this directory without setting the `PROF_DIR or COV_DIR` variable. Set the `PROF_DPI` or `COV_DPI` environment variables to specify an alternative .dpi filename. The default filename is `pgopti.dpi`. You can also use the `-prof_dpi` profmerge tool option, when merging the .dyn files, to specify the filename for the summary .dpi file.

Recommended Usage

If your application does not use a standard exit() call, insert a single call to this function in the body of the function that terminates the user application. Normally, `_PGOPTI_Prof_Dump_All()` should be called just once. It is also possible to use this function in conjunction with _PGOPTI_Prof_Reset_All() function to generate multiple .dyn files (presumably from multiple sets of input data).

> **NOTE**
> If the data is not reset between the dumps with a call to the _PGOPTI_Prof_Reset_All() function, the counters will continue accumulating data, resulting in the subsequent .dyn file containing data that was previously dumped.

Example

```
#include <pgouser.h>
void process_data(int foo) {}
int  get_input_data() { return 1; }
int  main(void)
{
// Selectively collect profile information for the portion
// of the application involved in processing input data.
  int input_data = get_input_data();
  while (input_data) {
    _PGOPTI_Prof_Reset_All();
    process_data(input_data);
    _PGOPTI_Prof_Dump_All();
    input_data = get_input_data();
```

Example

```
  }
  return 0;
}
```

Interval Profile Dumping

The `_PGOPTI_Set_Interval_Prof_Dump()` function activates Interval Profile Dumping and sets the approximate frequency at which dumps occur. This function is used in non-terminating applications.

The prototype of the function call is listed below.

Syntax

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

This function is used in non-terminating applications.

The `interval` parameter specifies the time interval at which profile dumping occurs and is measured in milliseconds. For example, if interval is set to 5000, then a profile dump and reset will occur approximately every 5 seconds. The interval is approximate because the time-check controlling the dump and reset is only performed upon entry to any instrumented function in your application.

Setting the interval to zero or a negative number will disable interval profile dumping, and setting a very small value for the interval may cause the instrumented application to spend nearly all of its time dumping profile information. Be sure to set interval to a large enough value so that the application can perform actual work and substantial profile information is collected.

The following example demonstrates one way of using interval profile dumping in non-terminating code.

Example

```
#include <stdio.h>
// The next include is to access
// _PGOPTI_Set_Interval_Prof_Dump_All
#include <pgouser.h>
int returnValue() { return 100; }
int main() {
  int ans;
  printf("CTRL-C to quit.\n");
  _PGOPTI_Set_Interval_Prof_Dump(5000);
  while (1)
    ans = returnValue();
}
```

You can compile the code shown above by entering commands similar to the following:

Operating System	Example
Linux*	`icc -prof-gen -o intrumented_number number.c`
macOS	`icc -prof-gen -o intrumented_number number.c`
Windows*	`icl /Qprof-gen /Feinstrumented_number number.c`

When compiled, the code shown above will dump profile information a `.dyn` file about every five seconds until the program is ended.

You can use the profmerge and proforder Tools tool to merge the `.dyn` files.

Recommended usage

Call this function at the start of a non-terminating user application to initiate interval profile dumping. Note that an alternative method of initiating interval profile dumping is by setting the environment variable INTEL_PROF_DUMP_INTERVAL to the desired interval value prior to starting the application.

Using interval profile dumping, you should be able to profile a non-terminating application with minimal changes to the application source code.

To control the behavior during dumping, you may use the environment variable INTEL_PROF_DUMP_CUMULATIVE. When `INTEL_PROF_DUMP_INTERVAL` or the API routine `_PGOPTI_SET_INTERVAL_PROF_DUMP` are used without `INTEL_PROF_DUMP_CUMULATIVE`, the counters are reset after each dump, and a new `.dyn` file will be created containing the data collected during each interval. This may result in a potentially large number of `.dyn` files that need to be stored and merged together by profmerge.

When `INTEL_PROF_DUMP_CUMULATIVE` is set (to any value), the `.dyn` file dump will be created at a specified interval as before, but the data will not be reset following the dump, and only the most recent `.dyn` file will be kept on the file system. This allows for the ability to create a `.dyn` file for a non-terminating application that contains all the accumulated counts to that point, but without the need for storage and merging of a large set of .dyn files.

Resetting the Dynamic Profile Counters

The `_PGOPTI_Prof_Reset()` function resets the dynamic profile counters. The prototype of the function call is listed below.

Syntax
`void _PGOPTI_Prof_Reset(void);`

This function is now deprecated. See _PGOPTI_Prof_Reset_All() function, which can be used instead.

One of the activities performed under profile-guided optimization is value-profiling. With value-profiling, the compiler inserts instrumentation to obtain a sample of the variable values in the program. Based upon this sampling, the compiler optimizes the user's code using frequently observed values.

In effect, during a run of an instrumented executable, the segment maintained for each routine executed during that run stores the following information:

- Number of times each basic block in the routine is executed
- Specific values observed at each data point undergoing value-profiling
- Number of times each of these values are observed

When the `_PGOPTI_Prof_Reset()` function is called, the basic block execution count for each basic block is set to 0, and the value-profiling information is reset to indicate that no values were observed at any data point.

Recommended Usage

Use this function to clear the profile counters prior to collecting profile information on a section of the instrumented application. See the example under Dumping Profile Information.

Dumping and Resetting Profile Information

The `_PGOPTI_Prof_Dump_And_Reset()` function dumps the profile information to a new .dyn file and then resets the dynamic profile counters. Then the execution of the instrumented application continues.

The prototype of the function call is listed below.

Syntax
`void _PGOPTI_Prof_Dump_And_Reset(void);`

This function is used in non-terminating applications and may be called more than once. Each call will dump the profile information to a new .dyn file.

Recommended Usage

Periodic calls to this function enables a non-terminating application to generate one or more profile information files (`.dyn` files). These files are merged during the feedback phase of profile-guided optimizations. The direct use of this function enables your application to control precisely when the profile information is generated.

Getting Coverage Summary Information on Demand

This API is supported only on Linux* OS for C/C++ applications.

The `_PGOPTI_Get_Coverage_Info()` function gets the basic-block and line coverage percentage for each instrumented file while the application is running.

The prototype of the function call is given below.

Syntax
`int _PGOPTI_Get_Coverage_Info(PGOPTI_COVERAGE_SUMMARY *coverage_array);`

This API provides on-demand coverage information for all the files that get profiled. The coverage information is stored in a dynamically allocated array of structures, a pointer to which is returned in the argument *coverage_array*. The return value of the call is the number of elements in this array.

You can use the coverage information as needed but you are responsible for freeing up the dynamically allocated coverage array.

You can also choose to print the on-demand coverage information onto the terminal screen as shown in the example below.

```
#include <pgouser.h>
void Coverage_Summary(void)
{
    int index, num_files;
    PGOPTI_COVERAGE_SUMMARY coverage_array, curp;
    // Get coverage summary information and print it out
    num_files =  _PGOPTI_Get_Coverage_Info(&coverage_array);
    for (index = 0; index < num_files; index++) {
        curp = &coverage_array[index];
        printf( "%s ", curp->file_name);
        printf( "Block coverage percent: %u, ", curp->coverage_percent);
        printf( "Line coverage percent: %u\n", curp->line_coverage_percent);
        free(curp->file_name);
    }
    if (num_files > 0) {
       free(coverage_array);
    }}
```

High-Level Optimization (HLO)

High-level Optimizations (HLO) exploit the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages. While the default optimization level, option `O2` , performs some high-level optimizations, specifying the `O3` option provides the best chance for performing loop transformations to optimize memory accesses.

> **NOTE**
> Loop optimizations may result in calls to library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The optimizations performed can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS), or `[Q]x` options. Additional HLO transformations may be performed for Intel® microprocessors than for non-Intel microprocessors.

Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization
- Predicate Optimization
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining, Memory Layout Change
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loopnests
- Multiversioning: Checks include Dependency of Memory References, and Trip Counts
- Loop Collapsing

Interprocedural Optimization

Interprocedural Optimization (IPO) is an automatic, multi-step process that allows the compiler to analyze your code to determine where you can benefit from specific optimizations.

The compiler may apply the following optimizations:

- Address-taken analysis
- Array dimension padding
- Alias analysis
- Automatic array transposition
- Automatic memory pool formation
- C++ class hierarchy analysis
- Common block variable coalescing
- Common block splitting

- Constant propagation
- Dead call deletion
- Dead formal argument elimination
- Dead function elimination
- Formal parameter alignment analysis
- Forward substitution
- Indirect call conversion
- Inlining
- Mod/ref analysis
- Partial dead call elimination
- Passing arguments in registers to optimize calls and register usage
- Points-to analysis
- Routine key-attribute propagation
- Specialization
- Stack frame alignment
- Structure splitting and field reordering
- Symbol table data promotion
- Un-referenced variable removal
- Whole program analysis

IPO Compilation Models

IPO supports two compilation models - single-file compilation and multi-file compilation.

Single-file compilation uses the `[Q]ip` compiler option, and results in one, real object file for each source file being compiled. During single-file compilation the compiler performs inline function expansion for calls to procedures defined within the current source file.

The compiler performs some single-file interprocedural optimization at the `O2` default optimization level; additionally the compiler may perform some inlining for the `O1` optimization level, such as inlining functions marked with inlining pragmas or attributes (GNU C and C++) and C++ class member functions with bodies included in the class declaration.

Multi-file compilation uses the `[Q]ipo` option, and results in one or more mock object files rather than normal object files. (See the *Compilation* section below for information about mock object files.) Additionally, the compiler collects information from the individual source files that make up the program. Using this information, the compiler performs optimizations across functions and procedures in different source files.

> **NOTE**
> Inlining and other optimizations are improved by profile information. For a description of how to use IPO with profile information for further optimization, see Profile an Application.

Compiling with IPO

As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in a mock object file. The mock object files contain the IR instead of the normal object code. Mock object files can be ten times or more larger than the size of normal object files.

During the IPO compilation phase only the mock object files are visible.

Linking with IPO

When you link with the `[Q]ipo` compiler option the compiler is invoked a final time. The compiler performs IPO across all mock object files. The mock objects must be linked with the compiler or by using the Intel® linking tools. While linking with IPO, the compiler and other linking tools compile mock object files as well as invoke the real/true object files linkers provided on the user's platform.

Link-time optimization using the `-ffat-lto-objects` compiler option is provided for GCC compatibility. During IPO compilation, you can specify `-ffat-lto-objects` option, for the compiler to generate a fat link-time optimization (LTO) object that has both a real/true object and a discardable intermediate language section. This enables both link-time optimization (LTO) linking and normal linking.

You can specify the `-fno-fat-lto-objects` option for the compiler to generate a link-time optimization (LTO) object that only has a discardable intermediate language section; no real/true object is generated. These files are inserted into archives in the form in which they were created. Using this option may improve compilation time and save space for objects.

If you use `ld` rather than `xild` to link objects or `ar` instead of `xiar` to create an archive, the real/true object, generated during fat link-time optimization guarantees that there will be no impediment to linking/building the archive. However, cross-file optimizations are lost in this case. The extra true object also takes additional space and takes compile time to generate it, so using `-fno-fat-lto-objects` compiler option is an advantage provided that you link the IPO mock object files with `xild` and archive them with `xiar`.

Whole Program Analysis

The compiler supports a large number of IPO optimizations that can be applied or have its effectiveness greatly increased when the whole program condition is satisfied.

During the analysis process, the compiler reads all Intermediate Representation (IR) in the mock file, object files, and library files to determine if all references are resolved and whether or not a given symbol is defined in a mock object file. Symbols that are included in the IR in a mock object file for both data and functions are candidates for manipulation based on the results of whole program analysis.

There are two types of whole program analysis - object reader method and table method. Most optimizations can be applied if either type of whole program analysis determines that the whole program conditions exists; however, some optimizations require the results of the object reader method, and some optimizations require the results of table method.

Object reader method

In the object reader method, the object reader emulates the behavior of the native linker and attempts to resolve the symbols in the application. If all symbols are resolved, the whole program condition is satisfied. This type of whole program analysis is more likely to detect the whole program condition.

Table method

In the table method the compiler analyzes the mock object files and generates a call-graph.

The compiler contains detailed tables about all of the functions for all important language-specific libraries, like `libc`. In this second method, the compiler constructs a call-graph for the application. The compiler then compares the function table and application call-graph. For each unresolved function in the call-graph, the compiler attempts to resolve the calls by finding an entry for each unresolved function in the compiler tables. If the compiler can resolve the functions call, the whole program condition exists.

See Also

ax, Qax

Inline Expansion of Functions

Interprocedural Optimization Options

ip, Qip

ipo, Qipo

ipo-c, Qipo-c

Linking Tools and Options

O

x, Qx

Use Interprocedural Optimization

Use Interprocedural Optimization

This topic discusses how to use IPO from the command line.

Compiling and Linking Using IPO

To enable IPO, you first compile each source file, then link the resulting source files.

Linux and macOS

1. Compile your source files with the `ipo` compiler option:

```
icpc -ipo -c a.cpp b.cpp c.cpp
```

   The command produces `a.o`, `b.o`, and `c.o` object files.

   Use the `c` compiler option to stop compilation after generating `.o` object files. The output files contain compiler intermediate representation (IR) corresponding to the compiled source files.
2. Link the resulting files. The following example command will produce an executable named `app`:

```
icpc -ipo -o app a.o b.o c.o
```

   The command invokes the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xild` tool, with the appropriate linking options.

The separate compile and link commands from the previous steps can be combined into a single command, for example:

```
icpc -ipo -o app a.cpp b.cpp c.cpp
```

The icpc command, shown in the examples, calls `GCC ld` to link the specified object files and produce the executable application, which is specified by the `-o` option.

Windows

1. Compile your source files with the `/Qipo` compiler option:

```
icl /Qipo /c a.cpp b.cpp c.cpp
```

   The command produces `a.obj`, `b.obj`, and `c.obj` object files.

   Use the `c` compiler option to stop compilation after generating `.obj` files. The output files contain compiler intermediate representation (IR) corresponding to the compiled source files.
2. Link the resulting files. The following example command will produce an executable named `app`:

```
icl /Qipo /Feapp a.obj b.obj c.obj
```

   The command invokes the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xilink` tool, with the appropriate linking options.

The separate compile and link commands from the previous steps can be combined into a single command, for example:

```
icl /Qipo /Feapp a.cpp b.cpp c.cpp
```

The icl command, shown in the examples, calls `link.exe` to link the specified object files and produce the executable application, which is specified by the `/Fe` option.

NOTE
Linux: Using icpc allows the compiler to use standard C++ libraries automatically; icc will not use the standard C++ libraries automatically.

macOS: Using icc/icpc commands allows the compiler to use libc++ libraries, by default. You can switch to using the GNU implementation of the standard C++ library using the `-stdlib=`*libstdc++* compiler option.

The Intel linking tools emulate the behavior of compiling at `-O0` (Linux and macOS) and `/Od` (Windows) option.

If multiple file IPO is applied to a series of object files, no one which are mock object files, no multi-file IPO is performed. The object files are simply linked with the linker.

Capturing Intermediate IPO Output

The `[Q]ipo-c` and `[Q]ipo-S` compiler options are useful for analyzing the effects of multi-file IPO, or when experimenting with multi-file IPO between modules that do not make up a complete program.

- Use the `[Q]ipo-c` compiler option to optimize across files and produce an object file. The option performs optimizations as described for the `[Q]ipo` option but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.o` (Linux and macOS) or `ipo_out.obj` (Windows).
- Use the `[Q]ipo-S` compiler option to optimize across files and produce an assembly file. The option performs optimizations as described for `[Q]ipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.s` (Linux) or `ipo_out.asm` (Windows).

For both options, you can use the `-o` (Linux and macOS) or `/Fe` (Windows) option to specify a different name.

These options generate multiple outputs if multi-object IPO is being used. The name of the first file is taken from the value of the `-o` (Linux and macOS) or `/Fe` (Windows) option.

The names of subsequent files are derived from the first file with an appended numeric value to the file name. For example, if the first object file is named `foo.o` (Linux and macOS) or `foo.obj` (Windows), the second object file will be named `foo1.o` or `foo1.obj`.

You can use the object file generated with the `[Q]ipo-c` option, but you will not get the full benefit of whole program optimizations if you use this option.

The object file created using the `[Q]ipo-c` option is a real object file, in contrast to the mock file normally generated using IPO; however, the generated object file is significantly different than the mock object file. Whole program optimizations, which require a knowledge of how the real object file will be linked in with other files to produce and object, are not applied.

The compiler generates a message indicating the name of each object or assembly file it generates. These files can be added to the real link step to build the final application.

Using -auto-ilp32 (Linux OS) or /Qauto-ilp32 (Windows OS) Option

On Linux systems based on Intel® 64 architecture, the `auto-ilp32` option has no effect unless you specify SSE3 or a higher suffix for the `x` option.

See Also

auto-ilp32, Qauto-ilp32

compiler option
c compiler option
o compiler option
Fe compiler option
ipo, Qipo compiler option
ipo-c, Qipo-c compiler option
ipo-S, Qipo-S compiler option
O compiler option

Performance and Large Program Considerations

IPO-related Performance Issues

There are some general optimization guidelines for using IPO that you should keep in mind:

- Using IPO on very large programs might trigger internal limits of other compiler optimization phases.
- Combining IPO and PGO can be a key technique for optimizing C++ applications. The following compiler options may result in performance gains: `O3`, `[Q]ipo`, and `[Q]prof-use`
- Applications where the compiler does not have sufficient intermediate representation (IR) coverage to do whole program analysis might not perform as well as those where IR information is complete.

In addition to these general guidelines, there are some practices to avoid while using IPO. The following list summarizes the activities to avoid:

- Do not use the link phase of an IPO compilation using mock object files produced for your application by a different compiler. Intel® compilers cannot inspect mock object files generated by other compilers for optimization opportunities.
- Do not link mock files with the `[Q]prof-use` compiler option unless the mock files were also compiled with the `[Q]prof-use` compiler option.
- Update make files to call the appropriate Intel linkers when using IPO from scripts. For Linux and macOS, replace all instances of `ld` with `xild`; for Windows, replace all instances of `link` with `xilink`.

IPO for Large Programs

In most cases, IPO generates a single true object file for the link-time compilation. This behavior is not optimal for very large programs, perhaps even making it impossible to use `[Q]ipo` compiler option on the application.

The compiler provides two methods to avoid this problem. The first method is an automatic size-based heuristic, which causes the compiler to generate multiple true object files for large link-time compilations. The second method is to manually instruct the compiler to perform multi-object IPO.

- Use the `[Q]ipo`*N* compiler option and pass an integer value in the place of *N*.
- Use the `[Q]ipo-separate` compiler option.

  The number of true object files generated by the link-time compilation is invisible to you unless the `[Q]ipo-c` or `[Q]ipo-S` compiler option is used.

Regardless of the method used, it is best to use the compiler defaults first and examine the results. If the defaults do not provide the desired results then experiment with generating a different number of object files.

You can use the `[Q]ipo-jobs` compiler option to control the number of processes, or jobs, executed during parallel IPO builds.

Using [Q]ipo*N* to Create Multiple Object Files

If you specify `[Q]ipo0`, which is the same as not specifying a value, the compiler uses heuristics to determine whether to create one or more object files based on the expected size of the application. The compiler generates one object file for small applications, and two or more object files for large applications. If you specify any value greater than 0, the compiler generates that number of object files, unless the value you pass a value that exceeds the number of source files. In that case, the compiler creates one object file for each source file then stops generating object files. The generated object files follow OS-specific naming conventions.

The following example commands demonstrate how to use `[Q]ipo2` option to compile large programs.

Linux and macOS

```
icpc -ipo2 -c a.cpp b.cpp
```

The resulting object files are `ipo_out.o`, `ipo_out1.o`, and `ipo_out2.o`.

Windows

```
icl /Qipo2 /c a.cpp b.cpp
```

The resulting object files are `ipo_out.obj`, `ipo_out1.obj`, and `ipo_out2.obj`.

Link the resulting object files as shown in Use Interprocedural Optimization or Linking Tools and Options..

Creating the Maximum Number of Object Files

Using `[Q]ipo-separate` allows you to force the compiler to generate the maximum number of true object files that the compiler will support during multiple object compilation. The maximum number of true object files is the equal to the number of mock object files passed on the link line.

For example, you can pass example commands similar to the following:

Linux and macOS

```
icpc -ipo-separate -ipo-c a.o b.o c.o
```

Windows

```
icl /Qipo-separate /Qipo-c a.obj b.obj c.obj
```

The compiler generates multiple object files that use the same naming convention discussed above.

Link the resulting object files as shown in Using IPO or Linking Tools and Options.

Understanding Code Layout and Multi-Object IPO

One of the optimizations performed during an IPO compilation is code layout. The analysis performed by the compiler during multi-file IPO determines a layout order for all of the routines for which it has intermediate representation (IR) information. For a multi-object IPO compilation, the compiler must tell the linker about the desired order.

The compiler first puts each routine in a named text section that varies depending on the operating system:

Linux

The first routine is placed in `.text00001`, the second is placed in `.text00002`, and so on.

Windows

The first routine is placed in `.text$00001`, the second is placed in `.text$00002`, and so on.

See Also

O compiler option

prof-use, Qprof-use compiler option

ipo, Qipo compiler option
ipo-c, Qipo-c
compiler option
ipo-jobs, Qipo-jobs
compiler option
ipo-S, Qipo-S
compiler option
ipo-separate,Qipo-separate
compiler option

Create a Library from IPO Objects

Libraries are often created using a library manager such as `xiar` for Linux/macOS or `xilib` for Windows. Given a list of objects, the library manager will insert the objects into a named library to be used in subsequent link steps.

Linux and macOS

Use `xiar` to create a library from a list of objects. For example the following command creates a library named `user.a` containing the `a.o` and `b.o` objects:

```
xiar cru user.a a.o b.o
```

If the objects have been created using `[Q]ipo -c` then the archive will not only contain a valid object, but the archive will also contain intermediate representation (IR) for that object file. For example, the following example will produce `a.o` and `b.o` that may be archived to produce a library containing both object code and IR for each source file. For example:

```
icc -ipo -c a.cpp b.cpp
```

The commands generate mock object files, which when placed in archive will also be accompanied by a true object file.

Using `xiar` is the same as specifying `xild -lib`.

macOS

When using `xilibtool`, specify `-static` to generate static libraries, or specify `dynamic` to create dynamic libraries. For example, the following command will create a static library named `mylib.a` that includes the `a.o`, `b.o`, and `c.o` objects:

```
xilibtool -static -o mylib.a a.o b.o c.o
```

Alternately, the following example command will create a dynamic library named `mylib.dylib` that includes the `a.o`, `b.o`, and `c.o` objects.

```
xilibtool -dynamic -o mylib.dylib a.o b.o c.o
```

Specifying `xilibtool` is the same as specifying `xild -libtool`.

Windows

Use `xilib` or `xilink -lib` to create libraries of IPO mock object files and link them on the command line.

For example:

1. Assume that you create three mock object files using a command similar to:

```
icl /c /Qipo a.cpp b.cpp c.cpp
```

2. Further assume `a.obj` contains the main subprogram. Create a library with a command similar to:

```
xilib -out:main.lib b.obj c.obj
```

or

```
xilink -lib -out:main.lib b.obj c.obj
```

3. Link the library and the main program object file with a command similar to:

```
xilink -out:result.exe a.obj main.lib
```

See Also

dynamiclib
compiler option

ipo-c, Qipo-c ,
compiler option

static compiler option

Request Compiler Reports with the xi* Tools

The compiler options `qopt-report` (Linux* and macOS) and `[Q]opt-report` (Windows*) generate optimization reports with different levels of detail. Related compiler options, listed under Optimization Report Options, allow you to specify the phase, direct output to a file (instead of `stderr`), and request reports from all routines with names containing a specific string as part of their name.

The xi* tools are used with inter-procedural optimization (IPO) during the final stage of IPO compilation. You can request compiler reports to be generated during the final IPO compilation by using certain options. The supported xi* tools are:

- Linker tools: `xilink` (Windows*) and `xild` (Linux* and macOS)
- Library tools: `xilib` (Windows*), `xiar` (Linux* and macOS), `xilibtool` (macOS)

The following tables lists the compiler report options that can be used with the xi* tools during the final IPO compilation.

Optimization Report Option	Description
`-qopt-report`[=*n*] (Linux* and macOS) `/Qopt-report`[:*n*] (Windows*)	Enables optimization report generation with different levels of detail. Valid values for *n* are 0 through 5. By default, when you specify this option without passing a value the compiler will generate a report with a medium level of detail. Higher numbers give greater levels of detail.
`-qopt-report-file`=*filename* (Linux* and macOS) `/Qopt-report-file`:*filename* (Windows*)	Generates an optimization report and directs the report output to the specified *file* name. If you omit the path, the file is created in the current directory. To create the file in a different directory, specify the full path to the output file and its file name.
`-qopt-report-phase`[=*list*] (Linux* and macOS) `/Qopt-report-phase`[:*list*] (Windows*)	Specifies a comma separated *list* of optimization phases to use when generating reports. If you do not specify a phase the compiler defaults to all. You can request a list of all available phases by using the `[Q]opt-report-help` option. To generate a report for the IPO phase, use the `-qopt-report-phase=`*ipo* (Linux* and macOS) or `/Qopt-report-phase:`*ipo* (Windows) option.

Optimization Report Option	Description
`-qopt-report-routine=`*substring* (Linux* and macOS) `/Qopt-report-routine:`*substring* (Windows*)	Generates reports for all routines with names containing *substring* as part of their name. You can also specify a sequence of substrings separated by commas. If you do this, the compiler generates an optimization report for each of the routines whose name contains one or more of these substrings. If *substring* is not specified, the compiler generates reports on all routines.
`-qopt-report-filter=`*string* (Linux* and macOS) `/Qopt-report-filter:`*string* (Windows*)	Tells the compiler to find the indicated parts of your application specified by *string*, and generate optimization reports for them. If both `-qopt-report-routines=`*string1* and `qopt-report-filter=`*string2* are specified, it is treated as `-qopt-report-filter=`*string1;string2*.
`-qopt-report-help` (Linux* and macOS) `/Qopt-report-help` (Windows*)	Displays the optimization phases available to use when using the `-qopt-report-phase` (Linux* and macOS) or `[q or Q]opt-report-phase` (Windows*) or option.
`-qopt-report-names` (Linux* and macOS) `/Qopt-report-names` (Windows*)	Specifies whether mangled or unmangled names appear in the optimization report. If this option is not specified, unmangled names are used by default. If you specify mangled, encoding (also known as decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the assembly listing. If you specify unmangled, no encoding (or decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the source listing. If you use this option, you do not have to specify option `-qopt-report` (Linux* OS and macOS) or `/Qopt-report` (Windows* OS).

See Also

qopt-report, Qopt-report
compiler option

qopt-report-file, Qopt-report-file
compiler option

qopt-report-help,Qopt-report-help
compiler option

qopt-report-phase, Qopt-report-phase
compiler option

qopt-report-routine, Qopt-report-routine
compiler option

qopt-report-filter, Qopt-report-filter
compiler option

Inline Expansion of Functions

Inline function expansion does not require that the applications meet the criteria for whole program analysis normally required by IPO; so this optimization is one of the most important optimizations done in Interprocedural Optimization (IPO). For function calls that the compiler believes are frequently executed, the compiler often decides to replace the instructions of the call with code for the function itself.

In the compiler, inline function expansion is performed on relatively small user functions more often than on functions that are relatively large. This optimization improves application performance by performing the following:

- Removing the need to set up parameters for a function call
- Eliminating the function call branch
- Propagating constants

Function inlining can improve execution time by removing the runtime overhead of function calls; however, function inlining can increase code size, code complexity, and compile times. In general, when you instruct the compiler to perform function inlining, the compiler can examine the source code in a much larger context, and the compiler can find more opportunities to apply optimizations.

Specifying the `[Q]ip` compiler option, single-file IPO, causes the compiler to perform inline function expansion for calls to procedures defined within the current source file; in contrast, specifying the `[Q]ipo` compiler option, multi-file IPO, causes the compiler to perform inline function expansion for calls to procedures defined in other files.

Caution
Using the `[Q]ip` and `[Q]ipo` (Windows*) options can, in some cases, significantly increase compile time and code size.

The compiler does a certain amount of inlining at the default level. Although such inlining is similar to what is done when you use the [Q]ip option, the amount of inlining done is generally less than when you use the option.

Selecting Routines for Inlining

The compiler attempts to select the routines whose inline expansions provide the greatest benefit to program performance. The selection is done using default heuristics. The inlining heuristics used by the compiler differ based on whether or not you use options for Profile-Guided Optimizations (PGO): `[Q]prof-use` compiler option.

When you use PGO with `[Q]ip` or `[Q]ipo`, the compiler uses the following guidelines for applying heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.

Using IPO with PGO

Combining IPO and PGO typically produces better results than using IPO alone. PGO produces dynamic profiling information that can usually provide better optimization opportunities than the static profiling information used in IPO.

The compiler uses characteristics of the source code to estimate which function calls are executed most frequently. It applies these estimates to the PGO-based guidelines described above. The estimation of frequency, based on static characteristics of the source, is not always accurate.

Inline Expansion of Library Functions

By default, the compiler automatically inlines (expands) a number of standard and math library functions at the point of the call to that function, which usually results in faster computation.

Many routines in the `libirc`, `libm`, or the `svml` library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.

The `-fno-builtin` (Linux* and macOS) or the `/Qno-builtin-<name>` and `/Oi-` (Windows*) options disable inlining for intrinsic functions and disable the by-name recognition support of intrinsic functions and the resulting optimizations. The `/Qno-builtin-<name>` option provides the ability to disable inlining for intrinsic functions, fine-tuning the functionality of the `/Oi-` option, which disables almost all intrinsic functions when used. Use these options if you redefine standard library routines with your own version and your version of the routine has the same name as the standard library routine.

Inlining and Function Preemption (Linux)

You must specify `fpic` to use function preemption. By default the compiler does not generate the position-independent code needed for preemption.

Compiler Directed Inline Expansion of Functions

Without directions from the user, the compiler attempts to estimate what functions should be inlined to optimize application performance.

The following options are useful in situations where an application can benefit from user function inlining but does not need specific direction about inlining limits.

Option	Effect
`inline-level`(Linux* and macOS) or `Ob` (Windows*)	Specifies the level of inline function expansion. Note that the option /Ob2 on Windows* is equivalent to -inline-level=2 on Linux* and macOS. Allowed values are 0, 1, and 2.
`[Q]ip-no-inlining`	Disables only inlining enabled by the `[Q]ip`, `[Q]ipo`, or `Ob2` options.
`[Q]ip-no-pinlining`	Disables partial inlining enabled by the `[Q]ip` or `[Q]ipo` options. No other IPO optimizations are disabled.
`fno-builtin` (Linux* and macOS) or `Oi-` (Windows)	Disables inlining for intrinsic functions. Disables the by-name recognition support of intrinsic functions and the resulting optimizations. Use this option if you redefine standard library routines with your own version and your version of the routine has the same name as the standard library routine. By default, the compiler automatically inlines (expands) a number of standard and math library functions at the point of the call to that function, which usually results in faster computation. Many routines in the `libirc`, `libm`, or `svml` library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.
setting `inline-debug-info` for the `debug` option	Indicates that the source position information for an inlined function should be retained, rather than replaced, by that of the call which is being inlined.

Developer Directed Inline Expansion of User Functions

In addition to the options that support compiler directed inline expansion of user functions, the compiler also provides compiler options and pragmas that allow you to more precisely direct when and if inline function expansion should occur.

The compiler measures the relative size of a routine in an abstract value of intermediate language units, which is approximately equivalent to the number of instructions that will be generated. The compiler uses the intermediate language unit estimates to classify routines and functions as relatively small, medium, or large functions. The compiler then uses the estimates to determine when to inline a function; if the minimum criteria for inlining is met and all other things are equal, the compiler has an affinity for inlining relatively small functions and not inlining relative large functions.

Typically, the compiler targets functions that have been marked for inlining based on the following:

- **Inlining keywords:** Tells the compiler to inline the specified function. For example, `__inline`, `__forceinline`.
- **Procedure-specific inlining pragmas:** Tells the compiler to inline calls within the targeted procedure if it is legal to do so. For example,`#pragma inline` or `#pragma forceinline` .
- **GCC function attributes for inlining:** Tells the compiler to inline the function even when no optimization level is specified. For example, `__attribute__((always_inline))`.

The following developer directed inlining options and pragmas provide the ability to change the boundaries used by the inliner to distinguish between small and large functions.

In general, you should use the `[Q]inline-factor` option before using the individual inlining options listed below; this single option effectively controls several other upper-limit options.

If your code hits an inlining limit, the compiler issues a warning at the highest warning level. The warning specifies which of the inlining limits have been hit, and the compiler option and/or pragmas needed to get a full report. For example, you could get a message as follows:

```
Inlining inhibited by limit max-total-size.  Use -qopt-report -qopt-report-phase=ipo for full
report.
```

Messages in the report refer directly to the command line options or pragmas that can be used to overcome the limits.

The following table lists the options you can use to fine-tune inline expansion of functions. The pragmas associated with the options are documented in the **Effect** column.

Option	Effect
`[Q]inline-factor`	Controls the multiplier applied to all inlining options that define upper limits: `inline-max-size`, `inline-max-total-size`, `inline-max-per-routine`, and `inline-max-per-compile`. While you can specify an individual increase in any of the upper-limit options, this single option provides an efficient means of controlling all of the upper-limit options with a single command. By default, this option uses a multiplier of 100, which corresponds to a factor of 1. Specifying 200 implies a factor of 2, and so on. Experiment with the multiplier carefully. You could increase the upper limits to allow too much inlining, which might result in your system running out of memory.
`[Q]inline-force-inline`	Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.

Option	Effect
	Without this option, the compiler treats functions declared with the `__inline` keyword as merely being recommended for inlining. When this option is used, it is as if they were declared with the `__forceinline` keyword.
`[Q]inline-min-size`	Redefines the maximum size of small routines; routines that are equal to or smaller than the value specified are more likely to be inlined.
`[Q]inline-max-size`	Redefines the minimum size of large routines; routines that are equal to or larger than the value specified are less likely to be inlined.
`[Q]inline-max-total-size`	Limits the expanded size of inlined functions. You can also use #pragma optimization_parameter inline-max-total-size=N to control the size an individual routine can grow through inlining.
`[Q]inline-max-per-routine`	Limits the number of times inlining can be applied within a routine. You can also use #pragma optimization_parameter inline-max-per-routine to control the number of times inlining may be applied to a routine.
`[Q]inline-max-per-compile`	Limits the number of times inlining can be applied within a compilation unit. The compilation unit limit depends on the whether or not you specify the `[Q]ipo` compiler option. If you enable IPO, all source files that are part of the compilation are considered one compilation unit. For compilations not involving IPO each source file is considered an individual compilation unit.

See Also

fbuiltin, Oi compiler option
fpic compiler option
ip, Qip compiler option
ipo, Qipo compiler option
prof-use, Qprof-use compiler option
debug (Linux* OS) compiler option
debug (Windows* OS) compiler option
Zi, Z7, ZI compiler option
inline-level, Ob compiler option
ip-no-pinlining, Qip-no-pinlining compiler option
inline-factor, Qinline-factor compiler option
inline-forceinline, Qinline-forceinline compiler option
inline-max-per-compile, Qinline-max-per-compile compiler option
inline-max-per-routine, Qinline-max-per-routine compiler option
inline-max-total-size, Qinline-max-total-size compiler option

inline-max-size, Qinline-max-size compiler option
inline-min-size, Qinline-min-size compiler option

Inlining Report

Function inlining can improve execution time by removing the runtime overhead of function calls; however, function inlining can increase code size, code complexity, and compile times. In general, when you instruct the compiler to perform function inlining, the compiler examines the source code in a much larger context, and the compiler can find more opportunities to apply optimizations.

The Inlining Report is part of the Opt Report. The compiler options `-qopt-report` (Linux* and macOS) and `/Qopt-report` (Windows*) generate optimization reports with different levels of detail. Related compiler options, listed under Optimization Report Options, allow you to specify the phase, direct output to a specific file, `stdout` or `stderr`, and request reports from all routines with names containing a specific string as part of their name.

The inlining report is a description of the inlining choices that were made for each routine that is compiled in the program. It is produced as part of the opt report. To restrict the opt report to contain ONLY the inlining report, use the option `-qopt-report-phase=ipo` (Linux* and macOS) or `/Qopt-report-phase:ipo` (Windows*).

The user can control the amount of information by specifying a level for the inlining report. The level is shown by a number from 1 to 5. Level 1 contains the smallest amount of information, and each level adds information to the report. Level 2 is the default report.

Level	**Summary**
Level 1	Shows each call that was inlined
Level 2 (default report)	Shows the values of the key inlining options
Level 3	Shows the calls to routines with external linkage
Level 4	Shows: • Whole program information • Size (sz) of the each routine inlined and the increase in application size (isz) due to each instance of inlining • Routine percentages • Calls that are not inlined
Level 5	Shows inlining footnotes, which contain advice on how to change the inlining to potentially improve application performance

The inlining report gives you an in-depth overview of the compiler's inlining decisions, which occur within five levels of granularity. You can specify levels with `-qopt-report=1`, `-qopt-report=2`, etc., (Linux* and macOS) or `/Qopt-report=1`, `/Qopt-report=2`, etc. (Windows*). See below for specific level details.

Level 1

The Inlining Report is activated when you run the Optimization Report, using `[q or Q]qopt-report`.

For each routine you compile, you get one report with the title INLINE REPORT that shows the calls inlined into that routine.

Example: Inlining Report Level 1 - Typical Routine

```
INLINE REPORT: (APPLU)
  -> INLINE: (295,12) SETBV
    -> INLINE: (398,18) EXACT
    -> INLINE: (399,18) EXACT
    -> INLINE: (409,18) EXACT
    -> INLINE: (410,18) EXACT
```

Example: Inlining Report Level 1 - Typical Routine

```
    -> INLINE: (420,18) EXACT
    -> INLINE: (421,18) EXACT
  -> INLINE: (299,12) SETIV
  -> (303,12) ERHS
  -> (307,12) SSOR
  -> INLINE: (311,12) ERROR
    -> INLINE: (1518,24) EXACT
    -> INLINE: (1552,24) EXACT
  -> INLINE: (315,12) PINTGR
  -> (319,12) VERIFY
```

The report gives the name of the compiled routine (APPLU), and contains one line for each call that the compiler decided to inline or not inline. In the above report, the compiler made 15 inlining decisions for calls in the routine APPLU. It decided to inline 12 of the calls. These decisions are indicated by the lines which start with `-> INLINE`. It decided not to inline three of the calls. These decisions are indicated by the lines without the word INLINE.

On each line, the position of the call in the source code is given in parentheses, followed by the name of the routine being called. For example:

```
-> INLINE: (398,18) EXACT
```

This refers to a call at line 398 column 18 to a routine called EXACT.

Level 2

Level 2 includes the values of important compiler options related to inlining. Unless the user specifies one of these values by using the option on the command line, the default value of the option in shown. You can read more about the meaning of the individual inlining options in the Inlining Options section.

Example: Inlining Report Level 2 - Values of Inlining Options

```
INLINING OPTION VALUES:
  -inline-factor: 100
  -inline-min-size: 30
  -inline-max-size: 230
  -inline-max-total-size: 2000
  -inline-max-per-routine: 10000
  -inline-max-per-compile: 500000
```

Level 3

Level 3 contains one additional line for each call to an external routine made in the application. Such calls are not candidates for inlining, because the code for these routines is not present in the file or files being compiled.

Example: Inlining Report Level 3 - External Linkage

```
Begin optimization report for: APPLU
    Report from: Interprocedural optimizations [ipo]
INLINE REPORT: (APPLU) [1] applu.f (1,16)
   -> EXTERN: (1,16) for_set_reentrancy
   -> EXTERN: (80,7) for_read_seq_lis
```

Level 4

Level 4 adds four additional pieces of information. The specifics are shown below:

- Whole Program values:

Example: Whole Program

```
WHOLE PROGRAM [SAFE] [EITHER METHOD]: false
WHOLE PROGRAM [SEEN] [TABLE METHOD]: true
WHOLE PROGRAM [READ] [OBJECT READER METHOD]: false
```

An application for which whole program is determined is subject to a higher degree of optimization than one which is not. The Intel compiler uses two methods of determining whole program, a TABLE METHOD and an OBJECT READER METHOD.

- The size of the routine [sz] and the inlined size of the routine [isz]. Usually isz is less than sz:

Example: Size of the Routine (sz) vs. Inlined Size of the Routine (isz)

```
 -> INLINE: (295,12) SETBV (isz = 752) (sz = 755)
   -> INLINE: (398,18) EXACT (isz = 98) (sz = 109)
```

In the above example, the routine SETBV was inlined into the routine that called it. The size of SETBV, before inlining, was 755 units. After inlining, the calling routine was increased by 752 units. The increase in the size of the calling routine is slightly less than the size of SETBV, because some of the overhead of calling SETBV was removed when SETBV was inlined.

- The percentage of time that has passed in the process of compiling the file:

Example: Percentage of Time Passed During Compilation

```
INLINE REPORT: (APPLU) [1/16=6.2%] applu.f (1,16)
```

For example, on the line above, [1/16 = 6.2%] indicates that APPLU is the first routine out of 16 to be compiled, and when this routine is done being compiled, 6.2% of the compilation is finished. You can use these numbers to estimate how long the compilation is going to take.

- The calls that did not get inlined and the reason why they did not get inlined. The reason is shown in double brackets [[ ]].

Example: Calls That Are Not Inlined

```
-> (303,12) ERHS (isz = 2125) (sz = 2128)
   [[ Inlining would exceed -inline-max-size value (2128>253)]]
```

In the above example, the routine ERHS is not inlined, because the size of the routine (2128 units) is larger than the allowable size (253 units). If you wish to inline routines that are this large, you can use the option `-inline-max-size=2128` (or larger).

Level 5

Level 5 adds the inlining footnotes.

Example: Use of Footnote

```
-> (303,12) ERHS (isz = 2058) (sz = 2061)
   [[ Inlining would exceed -inline-max-size-value (2061>230) <1>]]
```

The inlining footnotes explain the text found in the double brackets [[ ]]. They include a description for why an inlining call did not happen, and what you can do to make the inlining of this call happen.

The footnote annotation <1> refers to the first footnote in the INLINING FOOTNOTES section at the bottom of the inlining report, which is produced when the user selects Level 5. For example, the footnote produced for annotation <1> above is:

Example: Footnote Text

```
<1> The subprogram is larger than the inliner would normally inline. Use the
option -inline-max-size to increase the size of any subprogram that would
normally be inlined, add “!DIR$ATTRIBUTES FORCELINE” to the
declaration of the called function, or add “!DIR$ FORCELINE” before
the call site.
```

Processor Targeting

The manual processor dispatch feature allows you to target processors manually. You can control processor dispatching in a number of ways, including:

- Use the `cpu_specific` and `cpu_dispatch` keywords (attributes in Linux* or `__declspecs` in Windows*) to write one or more versions of a function that executes only on specified types of Intel® processors. You can also write a generic version that executes on other Intel or non-Intel processors. The Intel processor type is detected at runtime, and the corresponding function version is executed. This feature is available only for Intel processors based on IA-32 or Intel® 64 architecture. The feature not available for non-Intel processors. Applications built using the manual processor dispatch feature may be more highly optimized for Intel processors than for non-Intel processors.

  For more information, see below.
- Use the `optimization_parameter` pragma.

  For more information, see below.
- On Linux, in addition to the Intel-defined attributes `cpu_specific` and `cpu_dispatch`, C++ compilations with GNU Compiler Collection (GCC*) compatibility 4.8 or higher support creation of multiple function versions using the `target` attribute.

  For more information, see the GCC documentation on Function Multiversioning.

Using `cpu_dispatch` for Manual Processor Dispatch Programming

Use the `__declspec(cpu_dispatch(cpuid, cpuid,...))` syntax in your code to provide a list of targeted processors along with an empty function body/function stub. Use the `__declspec(cpu_specific(cpuid))` in your code to declare each function version targeted at particular type[s] of processors.

For a list of the values for `cpuid`, see the list on cpu_dispatch, cpu_specific.

NOTE
If no other matching Intel processor type is detected, the *generic* version of the function is executed. If you want the program to execute on non-Intel processors, a *generic* function version must be provided. You can control the degree of optimization of the *generic* function version and the processor features that it assumes.

The `cpuid` attributes are not case sensitive. The body of a function declared with `__declspec(cpu_dispatch)` must be empty, and is referred to as a stub (an empty-bodied function).

The following example illustrates how the `cpu_dispatch` and `cpu_specific` keywords can be used to create function versions for the 2nd generation Intel® Core™ processor family with support of Intel® Advanced Vector Extensions (Intel® AVX), for the Intel® Core™ processor family, for the Intel® Core™2 Duo processor family, and for other Intel and compatible, non-Intel processors. Each processor-specific function body might contain processor-specific intrinsic functions, or it might be placed in a separate source file and compiled with a processor-specific compiler option.

Example

```
#include <stdio.h>
// need to create specific function versions for the following processors:
__declspec(cpu_dispatch(core_2nd_gen_avx, core_i7_sse4_2, core_2_duo_ssse3, generic ))
void dispatch_func() {};      //  stub that will call the appropriate specific function
version

__declspec(cpu_specific(core_2nd_gen_avx))
void dispatch_func() {
  printf("\nCode for 2nd generation Intel Core processors with support for Intel AVX goes here
\n");
}

__declspec(cpu_specific(core_i7_sse4_2))
void dispatch_func() {
  printf("\nCode for Intel Core processors with support for SSE4.2 goes here\n");
}

__declspec(cpu_specific(core_2_duo_ssse3))
void dispatch_func() {
  printf("\nCode for Intel Core 2 Duo processors with support for SSSE3 goes here\n");
}

__declspec(cpu_specific(generic))
void dispatch_func() {
  printf("\nCode for non-Intel processors and generic Intel processors goes here\n");
}

int main() {
  dispatch_func();
  printf("Return from dispatch_func\n");
  return 0;
}
```

Considerations

Before using manual dispatch, consider whether the benefits outweigh the additional effort and possible performance issues. You may encounter any one or all of the following issues when using manual processor dispatch in your code:

- code and executable sizes increase considerably
- additional performance overhead may be introduced because of additional function calls

Test your application on all targeted platforms before release.

Using Pragmas to Target Processors Manually

You can use `#pragma intel optimization_parameter target_arch` to flag those routines in your code that you want to execute on specified types of processors. This pragma controls the `-m`or `/arch` option at a routine level, overriding the option values specified at the command-line, using the same values as the `-m` or `/arch` option to target processors. The following example illustrates how to use the pragma to target a routine `bar()` to execute only on Intel® AVX supported processors regardless of what the command-line has specified.

Example

```
include <immintrin.h>
#define N 1024

double x[N], y[N], z[N];

#pragma
intel optimization_parameter target_arch=AVX

void bar()
{
    int i;
    if (allow_avx) {
        _allow_cpu_features(_FEATURE_AVX);
        for (i = 0; i < N; i++) {
            z[i] = x[i] * y[i];
        }
    }
    else {
        for (i = 0; i < N; i++) {
            z[i] = x[i] * y[i];
        }
    }
}
```

You can also use the `_allow_cpu_features` intrinsic to tell the compiler that the code region may be targeted for processors with specified features, and the `_may_i_use_cpu_feature` to query the processor dynamically at the source level to determine if processor-specific features are available.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

See Also

`_allow_cpu_features`

`_may_i_use_cpu_feature`

`arch`

`m`

CPU Feature Targeting

The compiler provides a CPU-dispatching feature that enables users to provide different implementations of their functionality. The CPU-dispatching mechanism checks the target architecture and then selects the best implementation at runtime. However, the mechanism has two related potential limitations:

- Users cannot control the selected code path when they have a particular reason to do so; this can be decided only by the CPU-dispatching mechanism at runtime according to the target platform.
- Users cannot test their different implementations on the same machine.

CPU feature targeting addresses these limitations.

Usage

CPU feature targeting does not add any command-line options. Instead, the user must set a new environment variable, `INTEL_ISA_DISABLE`, before running their application.

The user can set the environment variable as follows (Linux example):

```
export INTEL_ISA_DISABLE=features
```

where *features* is a comma-separated list of features such as `sse2,avx`.

NOTE The feature names are those used with the `-m` option.

Setting the environment variable causes the named features not to be visible on the host even if the `CPUID` reports that it has them onboard. This has the following implications:

- If the user disables a CPU feature (for example, `_FEATURE_SSE2`) using `export INTEL_ISA_DISABLE=sse2`, then `_may_i_use_cpu_feature(_FEATURE_SSE2)` will return false; however, there will be no impact on other features for `_may_i_use_cpu_feature`.
- The CPU-dispatching mechanism will be affected; that is, dispatching will not take paths that require features disabled via `INTEL_ISA_DISABLE`.
- Libraries that use `libirc` for their CPU dispatching (such as `mkl` and `libimf`/`libsvml`) will be affected by `INTEL_ISA_DISABLE` in the same way.

Additional Information

- CPU feature targeting has no architecture restrictions. Users can set the environment variable effectively on all our current architectures.
- CPU feature targeting has no default setting (such as OFF or ON). The feature is triggered by the `INTEL_ISA_DISABLE` environment variable, so if users do not set that variable before running their application, everything works normally (with no CPU feature targeting). Also, if users specify invalid feature names within the environment variable's value, those names will be ignored.
- There is no IDE equivalent for the CPU feature targeting feature.

Important points to remember

- The value of environment variable `INTEL_ISA_DISABLE` is a feature list string comprising feature names separated by commas. The feature names are those used with the `-m` option.
- Users must set `INTEL_ISA_DISABLE` before running their application.
- Users must not disable any feature that is requested by the `-x` *`target`* option. For example, if you compile with `-xcore-avx2` and then disable `fma` (which is required by avx2) via the `INTEL_ISA_DISABLE` environment variable, a runtime error will occur indicating that the CPU is not supported.

Example

`hide_avx.c`:

```
#include "immintrin.h"
#define CHECK(feature) \
printf("%3s: %s\n", _may_i_use_cpu_feature(feature) ? "yes" : "no", #feature);

int main() {
```

```
    CHECK(_FEATURE_GENERIC_IA32);
    CHECK(_FEATURE_SSE4_2);
    CHECK(_FEATURE_AVX);
    CHECK(_FEATURE_AVX2);
    return 0;
}
```

Build `hide_avx.c` using `icc`:

```
icc hide_avx.c -o hide_avx.exe
```

Run `hide_avx.exe` on a machine with avx2, producing the following output:

```
yes: _FEATURE_GENERIC_IA32
yes: _FEATURE_SSE4_2
yes: _FEATURE_AVX
yes: _FEATURE_AVX2
```

Then set the environment variable on the command line:

```
export INTEL_ISA_DISABLE=avx2,avx
```

And then run `hide_avx.exe` again, producing the following output:

```
yes: _FEATURE_GENERIC_IA32
yes: _FEATURE_SSE4_2
no: _FEATURE_AVX
no: _FEATURE_AVX2
```

See Also

cpu_dispatch, cpu_specific

Methods to Optimize Code Size

This section provides some guidance on how to achieve smaller object and smaller executable size when using the optimizing features of Intel compilers.

There are two compiler options that are designed to prioritize code size over performance:

Option	Result	Notes
`Os`	Favors size over speed	This option enables optimizations that do not increase code size; it produces smaller code size than option `O2`. Option `Os` disables some optimizations that may increase code size for a small speed benefit.
`O1`	Minimizes code size	Compared to option `Os`, option `O1` disables even more optimizations that are generally known to increase code size. Specifying option `O1` implies option `Os`.

Option	Result	Notes
		As an intermediate step in reducing code size, you can replace option `O3` with option `O2` before specifying option `O1`. Option `O1` may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.

For more information about compiler options mentioned in this topic, see their full descriptions in the Compiler Reference.

The rest of this topic briefly discusses other methods that may help you further improve code size even when compared to the default behaviors of options `Os` and `O1`.

Things to remember:

- Some of these methods may already be applied by default when options `Os` and `O1` are specified. All the methods mentioned in this topic can be applied at higher optimization levels.
- Some of the options referred to in this topic will not necessarily cause code size reduction, and they may provide varying results (good, bad, or neutral) based on the characteristics of the target code. Still, these are the recommended things to try to see if they cause your binaries to become smaller while maintaining acceptable performance.

Disable or Decrease the Amount of Inlining

Inlining replaces a call to a function with the body of the function. This lets the compiler optimize the code for the inlined function in the context of its caller, usually yielding more specialized and better performing code. This also removes the overhead of calling the function at runtime.

However, replacing a call to a function by the code for that function usually increases code size. The code size increase can be substantial. To eliminate this code size increase, at the cost of the potential performance improvement, inlining can be disabled.

As an alternative to completely disabling inlining, the default amount of inlining can be decreased by using an inline factor less than the default value of 100. It corresponds to scaling the default values of the main inlining parameters by *n*%.

- Advantage: Disabling or reducing this optimization can reduce code size.
- Disadvantage: Performance is likely to be sacrificed by disabling or reducing inlining especially for applications with many small functions.

Use options to disable inlining:

Linux and macOS

`fno-inline`

Windows

`Ob0`

Use options to reduce inlining and factor the main inlining parameters:

Linux and macOS

`inline-factor=`*n*

Windows

`Qinline-factor:`*n*

Use options to fine tune the main inlining parameters:

Linux and macOS

- `inline-factor`
- `inline-max-per-compile`
- `inline-max-per-routine`
- `inline-max-size`
- `inline-max-total-size`
- `inline-min-size`

Windows

- `Qinline-factor`
- `Qinline-max-per-compile`
- `Qinline-max-per-routine`
- `Qinline-max-size`
- `Qinline-max-total-size`
- `Qinline-min-size`

Strip Symbols from Your Binaries

You can specify a compiler option to omit debugging and symbol information from the executable without sacrificing its operability.

- Advantage: This method noticeably reduces the size of the binary.
- Disadvantage: It may be very difficult to debug a stripped application.

Use options:

Linux

`Wl,--strip-all`

Windows

None

Dynamically Link Intel-provided Libraries

By default, some of the Intel support and performance libraries are linked statically into an executable. As a result, the library codes are linked into every executable being built. This means that codes are duplicated.

It may be more profitable to link them dynamically.

- Advantage: Performance of the resulting executable is normally not significantly affected. Library codes that are otherwise linked in statically into every executable will not contribute to the code size of each executable with this option. These codes will be shared between all executables using them, and they will be available independent of those executables.
- Disadvantage: The libraries on which the resulting executable depends must be re-distributed with the executable for it to work properly. When libraries are linked statically, only library content that is actually used is linked into the executable. Dynamic libraries contain all the library content. Therefore, it may not be beneficial to use this option if you only need to build and/or distribute a single executable. The executable itself may be much smaller when linked dynamically, compared to a statically linked executable. However, the total size of the executable plus shared libraries or DLLs may be much larger than the size of the statically linked executable.

Use Options:

Linux and macOS

`shared-intel`

Windows

```
MD
```

NOTE Option MD affects all libraries, not only the Intel-provided ones.

Exclude Unused Code and Data from the Executable

Programs often contain dead code or data that is not used during their execution. Even if no expensive whole-program inter-procedural analysis is made at compile time to identify dead code, there are compiler options you can specify to eliminate unused functions and data at link time.

This method is often referred to as function-level or data-level linking.

- Advantage: Only the code that is referenced remains in an executable. Dead functions and data are stripped from the executable. For the options passed to the linker, they also enable the linker to reorder the sections for other possible optimization.
- Disadvantage: The object codes may become slightly larger because each function or datum is put into a separate section. The overhead is eliminated at the linking stage. This method requires linker support to strip unused sections and may increase linking time.

Use Options:

Linux and macOS

```
-fdata-sections -ffunction-sections -Wl,--gc-sections
```

Windows

```
/Gw /Gy /link /OPT:REF
```

NOTE Option MD affects all libraries, not only the Intel-provided ones.

These options (from the use options example above) are passed to the linker:

Linux and macOS

```
Wl,--gc-sections
```

Windows

```
link /OPT:REF
```

Disable Recognition and Expansion of Intrinsic Functions

When recognized, intrinsic functions can get expanded inline or their faster implementation in a library may be assumed and linked in. By default, Inline expansion of intrinsic functions is enabled.

In some cases, disabling this behavior may noticeably improve the size of the produced object or binary.

- Advantage: Both the size of the object files and the size of library codes brought into an executable can be reduced.
- Disadvantage: This method can prevent various performance optimizations from happening. Slower standard library implementation will be used. The size of the final executable can be increased in cases when code pulled in statically from a library for an otherwise inlined intrinsic is large.

Use Options:

Linux and macOS

```
fno-builtin
```

Windows

`Oi-`

Additional information:

- This option is already the default if you specify option `O1`.
- For C++, you can specify Linux option `nolib-inline` to disable inline expansion of standard library or intrinsic functions.
- Depending on code characteristics, this option can sometimes increase binary size.

Optimize Exception Handling Data

If a program requires support for exception handling, the compiler creates a special section containing DWARF directives that are used by the Linux and macOSruntime to unwind and catch an exception.

This information is found in the `.eh_frame` section and may be shrunk using the compiler options listed below.

- Advantage:

  These options may shrink the size of the object or binary file by up to 15%, though the amount of the reduction depends on the target platform. These options control whether unwind information is precise at an instruction boundary or at a call boundary. For example, option `fno-asynchronous-unwind-tables` can be used for programs that may *only* throw or catch exceptions.
- Disadvantage: Both options may change the program's behavior. Do not use option `fno-exceptions` for programs that require standard C++ handling for objects of classes with destructors. Do not use option `fno-asynchronous-unwind-tables` for functions compiled with option `-fexceptions` or option `traceback` that contain calls to other functions that might throw exceptions or for C++ functions that declare objects with destructors.

Use Options:

Linux and macOS

`fno-exceptions` or `fno-asynchronous-unwind-tables`

Windows

None

Read the compiler option descriptions, which explain what the defaults and behavior are for each target platform.

Disable Passing Arguments in Registers Instead of on the Stack

You can specify an option that causes the compiler to pass arguments in registers rather than on the stack. This can yield faster code.

However, doing this may require the compiler to create an additional entry point for any function that can be called outside the code being compiled.

In many cases, this will lead to an increase in code size. To prevent this increase in code size, you can disable this optimization.

- Advantage: Disabling this optimization can reduce code size.
- Disadvantage: The amount of code size saved may be small when compared to the corresponding performance loss of disabling the optimization.

Use Options:

Linux and macOS

`qopt-args-in-regs=none`

Windows

`Qopt-args-in-regs:none`

Additional information:

- Specify **none** for option `[q or Q]opt-args-in-regs`. The default behavior for the option is that parameters are passed in registers when they are passed to routines whose definition is seen in the same compilation unit.
- Depending on code characteristics, this option can sometimes increase binary size.

Disable Loop Unrolling

Unrolling a loop increases the size of the loop proportionally to the unroll factor.

Disabling (or limiting) this optimization may help reduce code size at the expense of performance.

- Advantage: Code size is reduced.
- Disadvantage: Performance of otherwise unrolled loops may noticeably degrade because this limits other possible loop optimizations.

Use Options:

Linux and macOS

`unroll=0`

Windows

`Qunroll:0`

Additional information:

This option is already the default if you specify option `Os` or option `O1`.

Disable Automatic Vectorization

The compiler finds possibilities to use SIMD (Intel® Streaming SIMD Extensions (Intel® SSE)/Intel® Advanced Vector Extensions (Intel® AVX)) instructions to improve performance of applications. This optimization is called automatic vectorization.

In most cases, this optimization involves transformation of loops and increases code size, in some cases significantly.

Disabling this optimization may help reduce code size at the expense of performance.

- Advantage: Compile-time is also improved significantly.
- Disadvantage: Performance of otherwise vectorized loops may suffer significantly. If you care about the performance of your application, you should use this option selectively to suppress vectorization on everything except performance-critical parts.

Use Options:

Linux and macOS

`no-vec`

Windows

`Qvec-`

Additional information:

Depending on code characteristics, this option can sometimes increase binary size.

Avoid References to Compiler-specific Libraries

While compiler-specific libraries are intended to improve the performance of your application, they increase the size of your binaries.

Certain compiler options may improve the code size.

- Advantage: The compiler will not assume the presence of compiler-specific libraries. It will generate only calls that appear in the source code.
- Disadvantage: This method may sacrifice performance if the library codes were in hotspots. Also, because we cannot assume any libraries, some compiler optimizations will be suppressed.

Use Options:

Linux and macOS

`ffreestanding`

Windows

`Qfreestanding-`

Additional information:

- This option implies option `fno-builtin`. You can override that default by explicitly specifying option `fbuiltin`.
- Depending on code characteristics, this option can sometimes increase binary size.

Avoid Unnecessary 16-Byte Alignment

This topic only applies to Linux systems on IA-32 architecture.

This method should only be used in certain situations that are well understood. It can potentially cause correctness issues when linking with other objects or libraries that aren't built with this option.

The 32-bit Linux ABI states that stacks need only maintain 4-byte alignment. However, for performance reasons in modern architectures, GCC and ICC maintain an alignment of 16-bytes on the stack. Maintaining 16-byte alignment may require additional instructions to adjust the stack on function entries where no stack adjustment would otherwise be needed. This can impact code size, especially in code that consists of many small routines.

You can specify a compiler option that will revert ICC back to maintaining 4-byte alignment, which can eliminate the need for extra stack adjust instructions in some cases.

Use this option *only* if one of the following is true:

- Your code does not call any other object or library that can be built without this option and, therefore, may rely on the stack being aligned to 16-bytes when called.
- Your code is targeted for architectures that do not have or support SSE instructions; therefore, it would never need 16-byte alignment for correctness reasons.

- Advantage: Code size can be smaller because you do not need extra instructions to maintain 16-byte alignment when not needed. This method can improve performance in some cases because of this reduction of instructions.
- Disadvantage: This method can cause incompatibility when linked with other objects or libraries that rely on the stack being 16-byte aligned across the calls.

Use Options:

Linux

`falign-stack=assume-4-byte`

macOS

None

Windows

None

Additional information:

Depending on code characteristics, this option can sometimes increase binary size.

Use Interprocedural Optimization

Using interprocedural optimization (IPO) may reduce code size. It enables dead code elimination and suppresses generation of code for functions that are always inlined or proven that they are never to be called during execution.

- Advantage: Depending on the code characteristics, this optimization can reduce executable size and improve performance.
- Disadvantage: Binary size can increase depending on code/application..

Use Options:

Linux and macOS

`ipo`

Windows

`Qipo`

> **NOTE** This method is not recommended if you plan to ship object files as part of a final product.

Intel® C++ Compiler Classic Math Library

The Intel® C++ Compiler Classic includes a mathematical software library containing highly optimized and very accurate mathematical functions. These functions are commonly used in scientific or graphic applications, as well as other programs that rely heavily on floating-point computations. To include support for C99 `_Complex` data types, use the `[Q]std=c99` compiler option.

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

> **NOTE**
> Intel's `math.h` header file is compatible with the GCC Math Library `libm`, but it does not cause the GCC Math Library to be linked. The source can be built with `gcc`, `icc`, or `icl`. The header file for the math library, `mathimf.h`, contains additional functions that are found only in the math library. The source can only be built using the compiler and libraries.
>
> The long double functions, such as `expl` or `logl`, in the math library are ABI incompatible with the Microsoft libraries. The Intel compiler and libraries support the 80-bit long double data type (see the description of the Qlong-double option). For maximum compatibility, use `math.h` or `mathimf.h` header files along with the math library.

Compiler Math Libraries for Linux and macOS

The math library linked to an application depends on the compilation or linkage options specified.

Library	**Description**
`libimf.a`	Default static math library.

Library	Description
`libimf.so`	Default shared math library.

NOTE The math libraries contain performance-optimized implementations for various Intel platforms. By default, the best implementation for the underlying hardware is selected at runtime. The library dispatch of multi-threaded code may lead to apparent data races, which may be detected by certain software analysis tools. However, as long as the threads are running on cores with the same CPUID, these data races are harmless and not a cause for concern.

Compiler Math Libraries for Windows

The math library linked to an application depends on the compilation or linkage options specified.

Library	Option	Description
`libm.lib`		Default static math library.
`libmmt.lib`	`/MT`	Multi-threaded static math library.
`libmmd.lib`	`/MD`	Dynamically linked math library.
`libmmdd.lib`	`/MDd`	Dynamically linked debug math library.
`libmmds.lib`		Static version compiled with `/MD` option.

oneAPI and OpenCL™ Considerations

Currently, oneAPI uses the OpenCL Specification to determine the ULP accuracy for OpenCL mathematical functions. Details about their precision and accuracy, including tables for single and double precision functions, are available from the Khronos OpenCL Specification's section, Relative Error as ULPs.

Mathematical functions have different accuracy levels on different devices. The OpenCL specification sets a limit on the maximum ULP error (where applicable), but individual devices may provide a more accurate implementation. If the OpenCL implementation is optimized for CPU usage, using the same code may not work on a GPU device.

See Also

Math Function List
Qlong-double compiler option
MD compiler option
MT compiler option
std, Qstd compiler option

Use the Intel® C++ Compiler Classic Math Library

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

To use the Intel® C++ Compiler Classic math library, include the header file, `mathimf.h`, in your program. If the compiler is used for linking, then the math library is used by default.

Use Real Functions

The following examples demonstrate how to use the math library with the compiler. After you compile this example and run the program, the program will display the sine value of `x`.

Linux and macOS

```
// real_math.c
#include <stdio.h>
#include <mathimf.h>

int main() {
 float fp32bits;
 double fp64bits;
 long double fp80bits;
 long double pi_by_four = 3.141592653589793238/4.0;

// pi/4 radians is about 45 degrees
 fp32bits = (float) pi_by_four; // float approximation to pi/4
 fp64bits = (double) pi_by_four; // double approximation to pi/4
 fp80bits = pi_by_four; // long double (extended) approximation to pi/4

// The sin(pi/4) is known to be 1/sqrt(2) or approximately .7071067
 printf("When x = %8.8f, sinf(x) = %8.8f \n", fp32bits, sinf(fp32bits));
 printf("When x = %16.16f, sin(x) = %16.16f \n", fp64bits, sin(fp64bits));
 printf("When x = %20.20Lf, sinl(x) = %20.20Lf \n", fp80bits, sinl(fp80bits));

 return 0;
}
```

Use the following command to compile the example code on Linux platforms:

```
icc real_math.c
```

Windows

```
// real_math.c
#include <stdio.h>
#include <mathimf.h>

int main() {
  float fp32bits;
  double fp64bits;

// /Qlong-double compiler option required because, without it,
// long double types are mapped to doubles.
  long double fp80bits;
  long double pi_by_four = 3.141592653589793238/4.0;

// pi/4 radians is about 45 degrees
  fp32bits = (float) pi_by_four;

// float approximation to pi/4
  fp64bits = (double) pi_by_four;

// double approximation to pi/4
  fp80bits = pi_by_four;

// long double (extended) approximation to pi/4
// The sin(pi/4) is known to be 1/sqrt(2) or approximately .7071067
  printf("When x = %8.8f, sinf(x) = %8.8f \n",
```

```
  fp32bits, sinf(fp32bits));

  printf("When x = %16.16f, sin(x) = %16.16f \n",
  fp64bits, sin(fp64bits));

  printf("When x = %20.20f, sinl(x) = %20.20f \n",
  (double) fp80bits, (double) sinl(fp80bits));

// printf() does not support the printing of long doubles
// on Microsoft Windows, so fp80bits is cast to double in this example.
  return 0;
}
```

Since the `real_math.c` program includes the `long double` data type, use the `/Qlong-double` and `/Qpc80` compiler options in the command line:

Use the following command to compile the example code on Windows platforms:

```
icl /Qlong-double /Qpc80 real_math.c
```

Use Complex Functions

After you compile this example and run the program, you should get the following results:

`When z = 1.0000000 + 0.7853982 i, cexpf(z) = 1.9221154 + 1.9221156 i`

`When z = 1.000000000000 + 0.785398163397 i, cexp(z) = 1.922115514080 + 1.922115514080 i`

Linux, macOS, and Windows

```
// complex_math.c
#include <stdio.h>
#include <complex.h>

int main() {
  float _Complex c32in,c32out;
  double _Complex c64in,c64out;
  double pi_by_four= 3.141592653589793238/4.0;
  c64in = 1.0 + I pi_by_four;

// Create the double precision complex number 1 + (pi/4)  i
// where I is the imaginary unit.
  c32in = (float _Complex) c64in;

// Create the float complex value from the double complex value.
  c64out = cexp(c64in);
  c32out = cexpf(c32in);

// Call the complex exponential,
// cexp(z) = cexp(x+iy) = e^ (x + i y) = e^x  (cos(y) + i sin(y))
 printf("When z = %7.7f + %7.7f i, cexpf(z) = %7.7f + %7.7f i \n"
 ,crealf(c32in),cimagf(c32in),crealf(c32out),cimagf(c32out));
 printf("When z = %12.12f + %12.12f i, cexp(z) = %12.12f + %12.12f i \n"
 ,creal(c64in),cimag(c64in),creal(c64out),cimagf(c64out));

  return 0;
}
```

Since this example program includes the `_Complex` data type, be sure to include the `[Q]std=c99` compiler option in the command line. For example:

Linux or macOS

```
icc -std=c99 complex_math.c
```

Windows

```
icl Qstd=c99 complex_math.c
```

NOTE `_Complex` data types are supported in C but not in C++ programs.

Exception Conditions

If you call a math function using argument(s) that may produce undefined results, an error number is assigned to the system variable `errno`. Math function errors are usually domain errors or range errors.

Domain errors result from arguments that are outside the domain of the function. For example, `acos` is defined only for arguments between -1 and +1 inclusive. Attempting to evaluate `acos(-2)` or `acos(3)` results in a domain error, where the return value is `QNaN`.

Range errors occur when a mathematically valid argument results in a function value that exceeds the range of representable values for the floating-point data type. Attempting to evaluate `exp(1000)` results in a range error, where the return value is `INF`.

When domain or range error occurs, the following values are assigned to `errno`:

- domain error (`EDOM`): `errno = 33`
- range error (`ERANGE`): `errno = 34`

The following example shows how to read the `errno` value for an `EDOM` and `ERANGE` error.

```
// errno.c
#include <errno.h>
#include <mathimf.h>
#include <stdio.h>

int main(void) {
  double neg_one=-1.0;
  double zero=0.0;

// The natural log of a negative number is considered a domain error - EDOM
  printf("log(%e) = %e and errno(EDOM) = %d \n",neg_one,log(neg_one),errno);

// The natural log of zero is considered a range error - ERANGE
  printf("log(%e) = %e and errno(ERANGE) = %d \n",zero,log(zero),errno);
}
```

The output of `errno.c` will look like this:

`log(-1.000000e+00) = nan and errno(EDOM) = 33`

`log(0.000000e+00) = -inf and errno(ERANGE) = 34`

For the math functions in this section, a corresponding value for `errno` is listed when applicable.

Other Considerations

Some math functions are inlined automatically by the compiler. The functions actually inlined may vary and may depend on any vectorization or processor-specific compilation options used. You can disable automatic inline expansion of all functions by compiling your program with the `-fno-builtin` option (Linux and macOS) or the `/Oi-` option (Windows).

It is strongly recommended to use the default rounding mode (round-to-nearest-even) when calling math library transcendental functions and compiling with default optimization or higher. Faster implementations—in terms of latency and/or throughput— of these functions are validated under the default round-to-nearest-even mode. Using other rounding modes may make results generated by these faster implementations less accurate, or set unexpected floating-point status flags. This behavior may be avoided by using the `-no-fast-transcendentals` option (Linux and macOS) or `/Qfast-transcendentals-` option (Windows), which disables calls to the faster implementations of math functions, or by using the `-fp-model strict` option (Linux and macOS) or `/fp: strict` option (Windows). This option warns the compiler not to assume default settings for the floating-point environment.

> **NOTE** 64-bit decimal transcendental functions rely on binary double extended precision arithmetic. To obtain accurate results, user applications that call 64-bit decimal transcendentals should ensure that the x87 unit is operating in 80-bit precision (64-bit binary significands). In an environment where the default x87 precision is not 80 bits, such as Windows, it can be set to 80 bits by compiling the application source files with the `/Qpc80` option.

A change of the default precision control or rounding mode may affect the results returned by some of the mathematical functions.

The following are important compiler options when using certain data types in IA-32 and Intel® 64 architectures running Windows operating systems:

- `/Qlong-double`: Use this option when compiling programs that require support for the `long double` data type (80-bit floating-point). Without this option, compilation will be successful, but `long double` data types will be mapped to `double` data types.
- `/Qstd=c99`: Use this option when compiling programs that require support for `_Complex` data types.

See Also

fbuiltin, Oi compiler option
Overview: Tuning Performance
pc, Qpc compiler option
Qlong-double compiler option
std, Qstd compiler option

Math Function List

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math functions are listed here by function type.

Function Type	Name
Trigonometric Functions	`acos`
	`acosd`
	`acospi`
	`asin`
	`asind`
	`asinpi`

Function Type	Name
	atan
	atan2
	atan2pi
	atand
	atan2d
	atand2
	atanpi
	cos
	cosd
	cospi
	cot
	cotd
	sin
	sincos
	sincosd
	sind
	sinpi
	tan
	tand
	tanpi
Hyperbolic Functions	acosh
	asinh
	atanh
	cosh
	sinh
	sinhcosh
	tanh
Exponential Functions	cbrt
	exp
	exp10
	exp2

Function Type	Name
	expm1
	frexp
	hypot
	invsqrt
	ilogb
	ldexp
	log
	log10
	log1p
	log2
	logb
	pow
	pow2o3
	pow3o2
	powr
	scalb
	scalbln
	scalbn
	sqrt
Special Functions	annuity
	cdfnorm
	cdfnorminv
	compound
	erf
	erfcx
	erfc
	erfcinv
	erfinv
	gamma
	gamma_r
	j0

Function Type	Name
	j1
	jn
	lgamma
	lgamma_r
	tgamma
	y0
	y1
	yn
Nearest Integer Functions	ceil
	floor
	llrint
	llround
	lrint
	lround
	modf
	nearbyint
	rint
	round
	trunc
Remainder Functions	fmod
	remainder
	remquo
Miscellaneous Functions	copysign
	fabs
	fdim
	finite
	fma
	fmax
	fmin
	fpclassify
	isfinite

Function Type	Name
	isgreater
	isgreaterequal
	isinf
	isless
	islessequal
	islessgreater
	isnan
	isnormal
	isunordered
	maxmag
	minmag
	nan
	nextafter
	nexttoward
	signbit
	significand
Complex Functions	cabs
	cacos
	cacosh
	carg
	casin
	casinh
	catan
	catanh
	ccos
	cexp
	cexp2
	cimag
	cis
	clog
	clog10

Function Type	Name
	conj
	ccosh
	cpow
	cproj
	creal
	csin
	csinh
	csqrt
	ctan
	ctanh

Trigonometric Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following trigonometric functions:

acos

Description: The `acos` function returns the principal value of the inverse cosine of `x` in the range [0, pi] radians for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:

```
double acos(double x);
long double acosl(long double x);
float acosf(float x);
```

acosd

Description: The `acosd` function returns the principal value of the inverse cosine of `x` in the range [0,180] degrees for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:

```
double acosd(double x);
long double acosdl(long double x);
float acosdf(float x);
```

acospi

Description: The `acospi` function returns the principal value of the inverse cosine of `x`, divided by pi, in the range [0,1] for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:
```
double acospi(double x);
float acospif(float x);
```

asin

Description: The `asin` function returns the principal value of the inverse sine of `x` in the range [-pi/2, +pi/2] radians for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:
```
double asin(double x);
long double asinl(long double x);
float asinf(float x);
```

asind

Description: The `asind` function returns the principal value of the inverse sine of `x` in the range [-90,90] degrees for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1

Calling interface:
```
double asind(double x);
long double asindl(long double x);
float asindf(float x);
```

asinpi

Description: The `asinpi` function returns the principal value of the inverse sine of `x`, divided by pi, in the range [-1/2,1/2] degrees for `x` in the interval [-1,1].

errno: EDOM, for |x| > 1 divided by pi

Calling interface:
```
double asinpi(double x);
float asinpif(float x);
```

atan

Description: The `atan` function returns the principal value of the inverse tangent of `x` in the range [-pi/2, +pi/2] radians.

Calling interface:
```
double atan(double x);
long double atanl(long double x);
float atanf(float x);
```

atan2

Description: The `atan2` function returns the principal value of the inverse tangent of `y/x` in the range [-pi, +pi] radians.

errno: EDOM, for x = 0 and y = 0

Calling interface:
```
double atan2(double y, double x);
long double atan2l(long double y, long double x);
float atan2f(float y, float x);
```

atan2pi

Description: The `atan2pi` function returns the principal value of the inverse tangent of `y/x`, divided by pi, in the range [-1, +1].

errno: EDOM, for x = 0 and y = 0

Calling interface:

```
double atan2pi(double y, double x);
float atan2pif(float y, float x);
```

atand

Description: The `atand` function returns the principal value of the inverse tangent of `x` in the range [-90,90] degrees.

Calling interface:

```
double atand(double x);
long double atandl(long double x);
float atandf(float x);
```

atan2d

Description: The `atan2d` function returns the principal value of the inverse tangent of `y/x` in the range [-180, +180] degrees.

errno: EDOM, for x = 0 and y = 0.

Calling interface:

```
double atan2d(double x, double y);
long double atan2dl(long double x, long double y);
float atan2df(float x, float y);
```

atand2

Description: The `atand2` function returns the principal value of the inverse tangent of `y/x` in the range [-180, +180] degrees.

errno: EDOM, for x = 0 and y = 0.

Calling interface:

```
double atand2(double x, double y);
long double atand2l(long double x, long double y);
float atand2f(float x, float y);
```

atanpi

Description: The `atanpi` function returns the principal value of the inverse tangent of `x`, divided by pi, in the range [-1/2, +1/2].

Calling interface:

```
double atanpi(double x);
float atanpif(float x);
```

cos

Description: The `cos` function returns the cosine of `x` measured in radians.

Calling interface:

```
double cos(double x);
long double cosl(long double x);
```

```
float cosf(float x);
```

cosd

Description: The `cosd` function returns the cosine of `x` measured in degrees.

Calling interface:

```
double cosd(double x);
long double cosdl(long double x);
float cosdf(float x);
```

cospi

Description: The `cospi` function returns the cosine of `x` multiplied by pi, `cos(x*pi)`.

Calling interface:

```
double cospi(double x);
float cospif(float x);
```

cot

Description: The `cot` function returns the cotangent of `x` measured in radians.

errno: ERANGE, for overflow conditions at x = 0.

Calling interface:

```
double cot(double x);
long double cotl(long double x);
float cotf(float x);
```

cotd

Description: The `cotd` function returns the cotangent of `x` measured in degrees.

errno: ERANGE, for overflow conditions at x = 0.

Calling interface:

```
double cotd(double x);
long double cotdl(long double x);
float cotdf(float x);
```

sin

Description: The `sin` function returns the sine of `x` measured in radians.

Calling interface:

```
double sin(double x);
long double sinl(long double x);
float sinf(float x);
```

sincos

Description: The `sincos` function returns both the sine and cosine of `x` measured in radians.

Calling interface:

```
void sincos(double x, double *sinval, double *cosval);
void sincosl(long double x, long double *sinval, long double *cosval);
void sincosf(float x, float *sinval, float *cosval);
```

sincosd

Description: The `sincosd` function returns both the sine and cosine of `x` measured in degrees.

Calling interface:
```
void sincosd(double x, double *sinval, double *cosval);
void sincosdl(long double x, long double *sinval, long double *cosval);
void sincosdf(float x, float *sinval, float *cosval);
```

sind

Description: The `sind` function computes the sine of `x` measured in degrees.

Calling interface:
```
double sind(double x);
long double sindl(long double x);
float sindf(float x);
```

sinpi

Description: The `sinpi` function returns the sine of `x` multiplied by pi, `sin(x*pi)`.

Calling interface:
```
double sinpi(double x);
float sinpif(float x);
```

tan

Description: The `tan` function returns the tangent of `x` measured in radians.

Calling interface:
```
double tan(double x);
long double tanl(long double x);
float tanf(float x);
```

tand

Description: The `tand` function returns the tangent of `x` measured in degrees.

errno: ERANGE, for overflow conditions

Calling interface:
```
double tand(double x);
long double tandl(long double x);
float tandf(float x);
```

tanpi

Description: The `tanpi` function returns the tangent of `x` multiplied by pi, `tan(x*pi)`.

Calling interface:
```
double tanpi(double x);
float tanpif(float x);
```

Hyperbolic Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following `hyperbolic` functions:

acosh

Description: The `acosh` function returns the inverse hyperbolic cosine of `x`.

errno: EDOM, for x < 1

Calling interface:
```
double acosh(double x);
long double acoshl(long double x);
float acoshf(float x);
```

asinh

Description: The `asinh` function returns the inverse hyperbolic sine of `x`.

Calling interface:
```
double asinh(double x);
long double asinhl(long double x);
float asinhf(float x);
```

atanh

Description: The `atanh` function returns the inverse hyperbolic tangent of `x`.

errno:

EDOM, for |x| > 1

ERANGE, for x = 1

Calling interface:
```
double atanh(double x);
long double atanhl(long double x);
float atanhf(float x);
```

cosh

Description: The `cosh` function returns the hyperbolic cosine of `x`, $(e^{x} + e^{-x})/2$.

errno: ERANGE, for overflow conditions

Calling interface:
```
double cosh(double x);
long double coshl(long double x);
float coshf(float x);
```

sinh

Description: The `sinh` function returns the hyperbolic sine of `x`, $(e^{x} - e^{-x})/2$.

errno: ERANGE, for overflow conditions

Calling interface:
```
double sinh(double x);
long double sinhl(long double x);
float sinhf(float x);
```

sinhcosh

Description: The `sinhcosh` function returns both the hyperbolic sine and hyperbolic cosine of `x`.

errno: ERANGE, for overflow conditions

Calling interface:
```
void sinhcosh(double x, double *sinval, double *cosval);
void sinhcoshl(long double x, long double *sinval, long double *cosval);
void sinhcoshf(float x, float *sinval, float *cosval);
```

tanh

Description: The `tanh` function returns the hyperbolic tangent of `x`, $(e^x - e^{-x}) / (e^x + e^{-x})$.

Calling interface:
```
double tanh(double x);
long double tanhl(long double x);
float tanhf(float x);
```

Exponential Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following exponential functions:

cbrt

Description: The `cbrt` function returns the cube root of `x`.

Calling interface:
```
double cbrt(double x);
long double cbrtl(long double x);
float cbrtf(float x);
```

exp

Description: The `exp` function returns `e` raised to the `x` power, e^x.

errno: ERANGE, for underflow and overflow conditions

Calling interface:
```
double exp(double x);
long double expl(long double x);
float expf(float x);
```

exp10

Description: The `exp10` function returns 10 raised to the `x` power, 10^x.

errno: ERANGE, for underflow and overflow conditions

Calling interface:
```
double exp10(double x);
long double exp10l(long double x);
float exp10f(float x);
```

exp2

Description: The `exp2` function returns 2 raised to the `x` power, 2^x.

errno: ERANGE, for underflow and overflow conditions

Calling interface:
```
double exp2(double x);
long double exp2l(long double x);
float exp2f(float x);
```

expm1

Description: The `expm1` function returns `e` raised to the `x` power, minus 1, $e^x - 1$.

errno: ERANGE, for overflow conditions

Calling interface:
```
double expm1(double x);
long double expm1l(long double x);
float expm1f(float x);
```

frexp

Description: The `frexp` function converts a floating-point number `x` into signed normalized fraction in [1/2, 1) multiplied by an integral power of two. The signed normalized fraction is returned, and the integer exponent stored at location `exp`.

Calling interface:
```
double frexp(double x, int *exp);
long double frexpl(long double x, int *exp);
float frexpf(float x, int *exp);
```

hypot

Description: The `hypot` function returns the square root of $(x^2 + y^2)$.

errno: ERANGE, for overflow conditions

Calling interface:
```
double hypot(double x, double y);
long double hypotl(long double x, long double y);
float hypotf(float x, float y);
```

ilogb

Description: The `ilogb` function returns the exponent of `x` base two as a `signed int` value.

errno: ERANGE, for x = 0

Calling interface:
```
int ilogb(double x);
int ilogbl(long double x);
int ilogbf(float x);
```

invsqrt

Description: The `invsqrt` function returns the inverse square root.

Calling interface:
```
double invsqrt(double x);
long double invsqrtl(long double x);
float invsqrtf(float x);
```

ldexp

Description: The `ldexp` function returns $x*2^{exp}$, where `exp` is an integer value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);
float ldexpf(float x, int exp);
```

log

Description: The `log` function returns the natural log of `x`, ln(`x`).

errno: EDOM, for x < 0
errno: ERANGE, for x = 0

Calling interface:

```
double log(double x);
long double logl(long double x);
float logf(float x);
```

log10

Description: The `log10` function returns the base-10 log of `x`, $\log_{10}(x)$.

errno: EDOM, for x < 0
errno: ERANGE, for x = 0

Calling interface:

```
double log10(double x);
long double log10l(long double x);
float log10f(float x);
```

log1p

Description: The `log1p` function returns the natural log of `(x+1), ln(x + 1).`

errno: EDOM, for x < -1
errno: ERANGE, for x = -1

Calling interface:

```
double log1p(double x);
long double log1pl(long double x);
float log1pf(float x);
```

log2

Description: The `log2` function returns the base-2 log of `x`, $\log_2(x)$.

errno: EDOM, for x < 0
errno: ERANGE, for x = 0

Calling interface:

```
double log2(double x);
long double log2l(long double x);
float log2f(float x);
```

logb

Description: The `logb` function returns the signed exponent of `x`.

errno: EDOM, for x = 0

Calling interface:

```
double logb(double x);
long double logbl(long double x);
float logbf(float x);
```

pow

Description: The `pow` function returns `x` raised to the power of `y`, x^y.

errno: EDOM, for x = 0 and y < 0
errno: EDOM, for x < 0 and y is a non-integer
errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double pow(double x, double y);
long double powl(double x, double y);
float powf(float x, float y);
```

pow2o3

Description: The `pow2o3` function returns the cube root of `x` squared, `cbrt(`x^2`)`.

Calling interface:

```
double pow2o3(double x);
float pow2o3f(float x);
```

pow3o2

Description: The `pow3o2` function returns the square root of the cube of `x`, `sqrt(`x^3`)`.

errno: EDOM, for x < 0
errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double pow3o2(double x);
float pow3o2f(float x);
```

powr

Description: The `powr` function returns `x` raised to the power of `y`, x^y, where x ≥ 0.

errno: EDOM, for x < 0
errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double powr(double x, double y);
float powrf(float x, float y);
```

scalb

Description: The `scalb` function returns $x*2^y$, where `y` is a floating-point value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double scalb(double x, double y);
```

```
long double scalbl(long double x, long double y);
float scalbf(float x, float y);
```

scalbn

Description: The `scalbn` function returns $x*2^n$, where `n` is an integer value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double scalbn(double x, int n);
long double scalbnl (long double x, int n);
float scalbnf(float x, int n);
```

scalbln

Description: The `scalbln` function returns $x*2^n$, where `n` is a long integer value.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double scalbln(double x, long int n);
long double scalblnl (long double x, long int n);
float scalblnf(float x, long int n);
```

sqrt

Description: The `sqrt` function returns the correctly rounded square root.

errno: EDOM, for $x < 0$

Calling interface:

```
double sqrt(double x);
long double sqrtl(long double x);
float sqrtf(float x);
```

Special Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following special functions:

annuity

Description: The `annuity` function computes the present value factor for an annuity, $(1 - (1+x)^{(-y)}) / x$, where `x` is a rate and `y` is a period.

errno: ERANGE, for underflow and overflow conditions

Calling interface:

```
double annuity(double x, double y);
long double annuityl(long double x, long double y);
float annuityf(float x, float y);
```

cdfnorm

Description: The `cdfnorm` function returns the cumulative normal distribution function value.

Calling interface:
```
double cdfnorm(double x);
float cdfnormf(float x);
```

cdfnorminv

Description: The `cdfnorminv` function returns the inverse cumulative normal distribution function value.

errno:
EDOM, for finite or infinite `(x > 1) || (x < 0)`
ERANGE, for `x = 0 or x = 1`

Calling interface:
```
double cdfnorminv(double x);
float cdfnorminvf (float x);
```

compound

Description: The `compound` function computes the compound interest factor, $(1+x)^{y}$, where `x` is a rate and `y` is a period.

errno: ERANGE, for underflow and overflow conditions

Calling interface:
```
double compound(double x, double y);
long double compoundl(long double x, long double y);
float compoundf(float x, float y);
```

erf

Description: The `erf` function returns the error function value.

Calling interface:
```
double erf(double x);
long double erfl(long double x);
float erff(float x);
```

erfc

Description: The `erfc` function returns the complementary error function value.

errno: ERANGE, for underflow conditions

Calling interface:
```
double erfc(double x);
long double erfcl(long double x);
float erfcf(float x);
```

erfcx

Description: The `erfcx` function returns the scaled complementary error function value.

errno: ERANGE, for overflow conditions

Calling interface:
```
double erfcx(double x);
float erfcxf(float x);
```

erfcinv

Description: The `erfcinv` function returns the value of the inverse complementary error function of `x`.

errno: EDOM, for finite or infinite (x > 2) || (x < 0)

Calling interface:

```
double erfcinv(double x);
float erfcinvf(float x);
```

erfinv

Description: The `erfinv` function returns the value of the inverse error function of `x`.

errno: EDOM, for finite or infinite |x| > 1

Calling interface:

```
double erfinv(double x);
long double erfinvl(long double x);
float erfinvf(float x);
```

gamma

Description: The `gamma` function returns the value of the logarithm of the absolute value of gamma.

errno: ERANGE, for overflow conditions when x is a negative integer.

Calling interface:

```
double gamma(double x);
long double gammal(long double x);
float gammaf(float x);
```

gamma_r

Description: The `gamma_r` function returns the value of the logarithm of the absolute value of gamma. The sign of the gamma function is returned in the integer `signgam`.

Calling interface:

```
double gamma_r(double x, int *signgam);
long double gammal_r(long double x, int *signgam);
float gammaf_r(float x, int *signgam);
```

j0

Description: Computes the Bessel function (of the first kind) of `x` with order 0.

Calling interface:

```
double j0(double x);
long double j0l(long double x);
float j0f(float x);
```

j1

Description: Computes the Bessel function (of the first kind) of `x` with order 1.

Calling interface:

```
double j1(double x);
long double j1l(long double x);
float j1f(float x);
```

jn

Description: Computes the Bessel function (of the first kind) of `x` with order `n`.

Calling interface:

```
double jn(int n, double x);
long double jnl(int n, long double x);
float jnf(int n, float x);
```

lgamma

Description: The `lgamma` function returns the value of the logarithm of the absolute value of gamma.

errno: ERANGE, for overflow conditions, x=0 or negative integers.

Calling interface:

```
double lgamma(double x);
long double lgammal(long double x);
float lgammaf(float x);
```

lgamma_r

Description: The `lgamma_r` function returns the value of the logarithm of the absolute value of gamma. The sign of the gamma function is returned in the integer `signgam`.

errno: ERANGE, for overflow conditions, x=0 or negative integers.

Calling interface:

```
double lgamma_r(double x, int *signgam);
long double lgammal_r(long double x, int *signgam);
float lgammaf_r(float x, int *signgam);
```

tgamma

Description: The `tgamma` function computes the gamma function of `x`.

errno:

EDOM, for x=0 or negative integers.

ERANGE, for overflow conditions.

Calling interface:

```
double tgamma(double x);
long double tgammal(long double x);
float tgammaf(float x);
```

y0

Description: Computes the Bessel function (of the second kind) of `x` with order 0.

errno: EDOM, for x <= 0

Calling interface:

```
double y0(double x);
long double y0l(long double x);
float y0f(float x);
```

y1

Description: Computes the Bessel function (of the second kind) of `x` with order 1.

errno: EDOM, for x <= 0

Calling interface:

```
double y1(double x);
long double y1l(long double x);
float y1f(float x);
```

yn

Description: Computes the Bessel function (of the second kind) of `x` with order `n`.

errno: EDOM, for x <= 0

Calling interface:

```
double yn(int n, double x);
long double ynl(int n, long double x);
float ynf(int n, float x);
```

Nearest Integer Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following nearest integer functions:

ceil

Description: The `ceil` function returns the smallest integral value not less than `x` as a floating-point number.

Calling interface:

```
double ceil(double x);
long double ceill(long double x);
float ceilf(float x);
```

floor

Description: The `floor` function returns the largest integral value not greater than `x` as a floating-point value.

Calling interface:

```
double floor(double x);
long double floorl(long double x);
float floorf(float x);
```

llrint

Description: The `llrint` function returns the rounded integer value (according to the current rounding direction) as a `long long int`.

errno: ERANGE, for values too large

Calling interface:

```
long long int llrint(double x);
long long int llrintl(long double x);
long long int llrintf(float x);
```

llround

Description: The `llround` function returns the rounded integer value as a `long long int`.

errno: ERANGE, for values too large

Calling interface:

```
long long int llround(double x);
long long int llroundl(long double x);
long long int llroundf(float x);
```

lrint

Description: The `lrint` function returns the rounded integer value (according to the current rounding direction) as a `long int`.

errno: ERANGE, for values too large

Calling interface:

```
long int lrint(double x);
long int lrintl(long double x);
long int lrintf(float x);
```

lround

Description: The `lround` function returns the rounded integer value as a `long int`. Halfway cases are rounded away from zero.

errno: ERANGE, for values too large

Calling interface:

```
long int lround(double x);
long int lroundl(long double x);
long int lroundf(float x);
```

modf

Description: The `modf` function returns the value of the signed fractional part of `x` and stores the integral part at `*iptr` as a floating-point number.

Calling interface:

```
double modf(double x, double *iptr);
long double modfl(long double x, long double *iptr);
float modff(float x, float *iptr);
```

nearbyint

Description: The `nearbyint` function returns the rounded integral value as a floating-point number, using the current rounding direction.

Calling interface:

```
double nearbyint(double x);
long double nearbyintl(long double x);
float nearbyintf(float x);
```

rint

Description: The `rint` function returns the rounded integral value as a floating-point number, using the current rounding direction.

Calling interface:

```
double rint(double x);
long double rintl(long double x);
float rintf(float x);
```

round

Description: The `round` function returns the nearest integral value as a floating-point number. Halfway cases are rounded away from zero.

Calling interface:

```
double round(double x);
long double roundl(long double x);
float roundf(float x);
```

trunc

Description: The `trunc` function returns the truncated integral value as a floating-point number.

Calling interface:

```
double trunc(double x);
long double truncl(long double x);
float truncf(float x);
```

Remainder Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following remainder functions:

fmod

Description: The `fmod` function returns the value `x-n*y` for integer `n` such that if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`.

errno: EDOM, for y = 0

Calling interface:

```
double fmod(double x, double y);
long double fmodl(long double x, long double y);
float fmodf(float x, float y);
```

remainder

Description: The `remainder` function returns the value of `x REM y` as required by the IEEE standard.

errno: EDOM, for y = 0

Calling interface:

```
double remainder(double x, double y);
long double remainderl(long double x, long double y);
float remainderf(float x, float y);
```

remquo

Description: The `remquo` function returns the value of `x REM y`. In the object pointed to by `quo` the function stores a value whose sign is the sign of `x/y` and whose magnitude is congruent modulo 2^n of the integral quotient of `x/y`. `N` is an implementation-defined integer. For all systems, `N` is equal to 31.

errno: EDOM, for y = 0

Calling interface:

```
double remquo(double x, double y, int *quo);
long double remquol(long double x, long double y, int *quo);
float remquof(float x, float y, int *quo);
```

Miscellaneous Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following miscellaneous functions:

copysign

Description: The `copysign` function returns the value with the magnitude of `x` and the sign of `y`.

Calling interface:

```
double copysign(double x, double y);
long double copysignl(long double x, long double y);
float copysignf(float x, float y);
```

fabs

Description: The `fabs` function returns the absolute value of `x`.

Calling interface:

```
double fabs(double x);
long double fabsl(long double x);
float fabsf(float x);
```

fdim

Description: The `fdim` function returns the positive difference value, `x-y` (for `x > y`) or +0 (for `x <= to y`).

errno: ERANGE, for overflow conditions

Calling interface:

```
double fdim(double x, double y);
long double fdiml(long double x, long double y);
float fdimf(float x, float y);
```

finite

Description: The `finite` function returns 1 if `x` is not a `NaN` or +/- infinity. Otherwise 0 is returned.

Calling interface:

```
int finite(double x);
int finitel(long double x);
int finitef(float x);
```

fma

Description: The `fma` functions return `(x*y)+z`.

Calling interface:

```
double fma(double x, double y, double z);
long double fmal(long double x, long double y, long double z);
float fmaf(float x, float y, float z);
```

fmax

Description: The `fmax` function returns the maximum numeric value of its arguments.

Calling interface:

```
double fmax(double x, double y);
long double fmaxl(long double x, long double y);
float fmaxf(float x, float y);
```

fmin

Description: The `fmin` function returns the minimum numeric value of its arguments.

Calling interface:

```
double fmin(double x, double y);
long double fminl(long double x, long double y);
float fminf(float x, float y);
```

fpclassify

Description: The `fpclassify` function returns the value of the number classification macro appropriate to the value of its argument.

Return Value
0 (NaN)
1 (Infinity)
2 (Zero)
3 (Subnormal)
4 (Finite)

Calling interface:

```
int fpclassify(double x);
int fpclassifyl(long double x);
int fpclassifyf(float x);
```

isfinite

Description: The `isfinite` function returns 1 if `x` is not a NaN or +/- infinity. Otherwise 0 is returned.

Calling interface:

```
int isfinite(double x);
int isfinitel(long double x);
int isfinitef(float x);
```

isgreater

Description: The `isgreater` function returns 1 if `x` is greater than `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isgreater(double x, double y);
int isgreaterl(long double x, long double y);
int isgreaterf(float x, float y);
```

isgreaterequal

Description: The `isgreaterequal` function returns 1 if `x` is greater than or equal to `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isgreaterequal(double x, double y);
int isgreaterequall(long double x, long double y);
int isgreaterequalf(float x, float y);
```

isinf

Description: The `isinf` function returns a non-zero value if and only if its argument has an infinite value.

Calling interface:

```
int isinf(double x);
int isinfl(long double x);
int isinff(float x);
```

isless

Description: The `isless` function returns 1 if `x` is less than `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isless(double x, double y);
int islessl(long double x, long double y);
int islessf(float x, float y);
```

islessequal

Description: The `islessequal` function returns 1 if `x` is less than or equal to `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int islessequal(double x, double y);
int islessequall(long double x, long double y);
int islessequalf(float x, float y);
```

islessgreater

Description: The `islessgreater` function returns 1 if `x` is less than or greater than `y`. This function does not raise the invalid floating-point exception.

Calling interface:

```
int islessgreater(double x, double y);
int islessgreaterl(long double x, long double y);
int islessgreaterf(float x, float y);
```

isnan

Description: The `isnan` function returns a non-zero value, if and only if `x` has a NaN value.

Calling interface:

```
int isnan(double x);
int isnanl(long double x);
int isnanf(float x);
```

isnormal

Description: The `isnormal` function returns a non-zero value, if and only if `x` is normal.

Calling interface:

```
int isnormal(double x);
int isnormall(long double x);
int isnormalf(float x);
```

isunordered

Description: The `isunordered` function returns 1 if either `x` or `y` is a NaN. This function does not raise the invalid floating-point exception.

Calling interface:

```
int isunordered(double x, double y);
int isunorderedl(long double x, long double y);
int isunorderedf(float x, float y);
```

maxmag

Description: The `maxmag` function returns the value of larger magnitude from among its two arguments, `x` and `y`. If `|x| > |y|` it returns `x`; if `|y| > |x|` it returns `y`; otherwise it behaves like `fmax(x,y)`.

Calling interface:

```
double maxmag(double x, double y);
float maxmagf(float x, float y);
```

minmag

Description: The `minmag` function returns the value of smaller magnitude from among its two arguments, `x` and `y`. If `|x| < |y|` it returns `x`; if `|y| < |x|` it returns `y`; otherwise it behaves like `fmin(x,y)`.

Calling interface:

```
double minmag(double x, double y);
float minmagf(float x, float y);
```

nan

Description: The `nan` function returns a quiet NaN, with content indicated through `tagp`.

Calling interface:

```
double nan(const char *tagp);
long double nanl(const char *tagp);
float nanf(const char *tagp);
```

nextafter

Description: The `nextafter` function returns the next representable value in the specified format after `x` in the direction of `y`.

errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double nextafter(double x, double y);
```

```
long double nextafterl(long double x, long double y);
float nextafterf(float x, float y);
```

nexttoward

Description: The `nexttoward` function returns the next representable value in the specified format after `x` in the direction of `y`. If `x` equals `y`, then the function returns `y` converted to the type of the function. Use the `Qlong-double` option (for C++ only) on Windows* operating systems for accurate results.

errno: ERANGE, for overflow and underflow conditions

Calling interface:

```
double nexttoward(double x, long double y);
long double nexttowardl(long double x, long double y);
float nexttowardf(float x, long double y);
```

signbit

Description: The `signbit` function returns a non-zero value, if and only if the sign of `x` is negative.

Calling interface:

```
int signbit(double x);
int signbitl(long double x);
int signbitf(float x);
```

significand

Description: The `significand` function returns the significand of `x` in the interval [1,2). For `x` equal to zero, NaN, or +/- infinity, the original `x` is returned.

Calling interface:

```
double significand(double x);
long double significandl(long double x);
float significandf(float x);
```

Complex Functions

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library supports the following complex functions:

cabs

Description: The `cabs` function returns the complex absolute value of `z`.

Calling interface:

```
double cabs(double _Complex z);
long double cabsl(long double _Complex z);
float cabsf(float _Complex z);
```

cacos

Description: The `cacos` function returns the complex inverse cosine of `z`.

Calling interface:

```
double _Complex cacos(double _Complex z);
```

```
long double _Complex cacosl(long double _Complex z);
float _Complex cacosf(float _Complex z);
```

cacosh

Description: The `cacosh` function returns the complex inverse hyperbolic cosine of `z`.

Calling interface:

```
double _Complex cacosh(double _Complex z);
long double _Complex cacoshl(long double _Complex z);
float _Complex cacoshf(float _Complex z);
```

carg

Description: The `carg` function returns the value of the argument in the interval [-pi, +pi].

Calling interface:

```
double carg(double _Complex z);
long double cargl(long double _Complex z);
float cargf(float _Complex z);
```

casin

Description: The `casin` function returns the complex inverse sine of `z`.

Calling interface:

```
double _Complex casin(double _Complex z);
long double _Complex casinl(long double _Complex z);
float _Complex casinf(float _Complex z);
```

casinh

Description: The `casinh` function returns the complex inverse hyperbolic sine of `z`.

Calling interface:

```
double _Complex casinh(double _Complex z);
long double _Complex casinhl(long double _Complex z);
float _Complex casinhf(float _Complex z);
```

catan

Description: The `catan` function returns the complex inverse tangent of `z`.

Calling interface:

```
double _Complex catan(double _Complex z);
long double _Complex catanl(long double _Complex z);
float _Complex catanf(float _Complex z);
```

catanh

Description: The `catanh` function returns the complex inverse hyperbolic tangent of `z`.

Calling interface:

```
double _Complex catanh(double _Complex z);
long double _Complex catanhl(long double _Complex z);
float _Complex catanhf(float _Complex z);
```

ccos

Description: The `ccos` function returns the complex cosine of `z`.

Calling interface:

```
double _Complex ccos(double _Complex z);
long double _Complex ccosl(long double _Complex z);
float _Complex ccosf(float _Complex z);
```

ccosh

Description: The `ccosh` function returns the complex hyperbolic cosine of `z`.

Calling interface:

```
double _Complex ccosh(double _Complex z);
long double _Complex ccoshl(long double _Complex z);
float _Complex ccoshf(float _Complex z);
```

cexp

Description: The `cexp` function returns e^z (`e` raised to the power `z`).

Calling interface:

```
double _Complex cexp(double _Complex z);
long double _Complex cexpl(long double _Complex z);
float _Complex cexpf(float _Complex z);
```

cexp2

Description: The `cexp` function returns 2^z (2 raised to the power `z`).

Calling interface:

```
double _Complex cexp2(double _Complex z);
long double _Complex cexp2l(long double _Complex z);
float _Complex cexp2f(float _Complex z);
```

cexp10

Description: The `cexp10` function returns 10^z (10 raised to the power `z`).

Calling interface:

```
double _Complex cexp10(double _Complex z);
long double _Complex cexp10l(long double _Complex z);
float _Complex cexp10f(float _Complex z);
```

cimag

Description: The `cimag` function returns the imaginary part value of `z`.

Calling interface:

```
double cimag(double _Complex z);
long double cimagl(long double _Complex z);
float cimagf(float _Complex z);
```

cis

Description: The `cis` function returns the cosine and sine (as a complex value) of `z` measured in radians.

Calling interface:

```
double _Complex cis(double x);
```

```
long double _Complex cisl(long double z);
float _Complex cisf(float z);
```

cisd

Description: The `cisd` function returns the cosine and sine (as a complex value) of `z` measured in degrees.

Calling interface:

```
double _Complex cisd(double x);
long double _Complex cisdl(long double z);
float _Complex cisdf(float z);
```

clog

Description: The `clog` function returns the complex natural logarithm of `z`.

Calling interface:

```
double _Complex clog(double _Complex z);
long double _Complex clogl(long double _Complex z);
float _Complex clogf(float _Complex z);
```

clog2

Description: The `clog2` function returns the complex logarithm base 2 of `z`.

Calling interface:

```
double _Complex clog2(double _Complex z);
long double _Complex clog2l(long double _Complex z);
float _Complex clog2f(float _Complex z);
```

clog10

Description: The `clog10` function returns the complex logarithm base 10 of `z`.

Calling interface:

```
double _Complex clog10(double _Complex z);
long double _Complex clog10l(long double _Complex z);
float _Complex clog10f(float _Complex z);
```

conj

Description: The `conj` function returns the complex conjugate of `z` by reversing the sign of its imaginary part.

Calling interface:

```
double _Complex conj(double _Complex z);
long double _Complex conjl(long double _Complex z);
float _Complex conjf(float _Complex z);
```

cpow

Description: The `cpow` function returns the complex power function, x^y.

Calling interface:

```
double _Complex cpow(double _Complex x, double _Complex y);
long double _Complex cpowl(long double _Complex x, long double _Complex y);
float _Complex cpowf(float _Complex x, float _Complex y);
```

cproj

Description: The `cproj` function returns a projection of `z` onto the Riemann sphere.

Calling interface:

```
double _Complex cproj(double _Complex z);
long double _Complex cprojl(long double _Complex z);
float _Complex cprojf(float _Complex z);
```

creal

Description: The `creal` function returns the real part of `z`.

Calling interface:

```
double creal(double _Complex z);
long double creall(long double _Complex z);
float crealf(float _Complex z);
```

csin

Description: The `csin` function returns the complex sine of `z`.

Calling interface:

```
double _Complex csin(double _Complex z);
long double _Complex csinl(long double _Complex z);
float _Complex csinf(float _Complex z);
```

csinh

Description: The `csinh` function returns the complex hyperbolic sine of `z`.

Calling interface:

```
double _Complex csinh(double _Complex z);
long double _Complex csinhl(long double _Complex z);
float _Complex csinhf(float _Complex z);
```

csqrt

Description: The `csqrt` function returns the complex square root of `z`.

Calling interface:

```
double _Complex csqrt(double _Complex z);
long double _Complex csqrtl(long double _Complex z);
float _Complex csqrtf(float _Complex z);
```

ctan

Description: The `ctan` function returns the complex tangent of `z`.

Calling interface:

```
double _Complex ctan(double _Complex z);
long double _Complex ctanl(long double _Complex z);
float _Complex ctanf(float _Complex z);
```

ctanh

Description: The `ctanh` function returns the complex hyperbolic tangent of `z`.

Calling interface:

```
double _Complex ctanh(double _Complex z);
```

```
long double _Complex ctanhl(long double _Complex z);
float _Complex ctanhf(float _Complex z);
```

C99 Macros

Many routines in the Intel® C++ Compiler Classic Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

The `mathimf.h` header file includes prototypes for Intel® C++ Compiler Classic Math Library functions.

The math library and `mathimf.h` header file support the following C99 macros:

```
int fpclassify(x);
int isfinite(x);
int isgreater(x, y);
int isgreaterequal(x, y);
int isinf(x);
int isless(x, y);
int islessequal(x, y);
int islessgreater(x, y);
int isnan(x);
int isnormal(x);
int isunordered(x, y);
int signbit(x);
```

See Also

Miscellaneous Functions

Automatically-Aligned Dynamic Allocation

Automatically-Aligned Dynamic Allocation

Background

It is possible to tell the compiler that a data structure has a greater alignment requirement than its individual elements require. For example:

C++ standard syntax

```
class alignas(64) X {
    double elem[8];
};
```

GNU-compatible syntax

```
class __attribute__((aligned(64))) X {
    double elem[8];
};
```

Microsoft-compatible syntax

```
class __declspec(align(64)) X {
    double elem[8];
};
```

This is especially important for a structure that will be used with SIMD instructions, which typically require greater alignment than the individual data elements. The compiler will ensure that variables declared with such a type, either statically or on the stack, will be allocated with the appropriate alignment.

However, if an object of such a type is allocated dynamically, with a `new`-expression, the compiler was not previously able to do anything to ensure the appropriate alignment. That is because the C++ language requires that only very specific allocation methods be used, over which the programmer can take control if necessary, and none of those allocation methods are able to support specific alignment. They all assume that some alignment value is enough for everything, and guarantee that (and nothing more).

In the past, to ensure a greater alignment for a given type, a programmer had to take control of its allocation. One way to do that is by always allocating the memory separately with the appropriate alignment, and using a non-allocating placement `new`-expression. For example:

Incorrect alignment

```
new X
```

Correct alignment

```
new (_mm_malloc(sizeof(X), alignof(X))) X
```

However, this method is verbose, tedious, and error-prone.

Another way is to write class-specific allocation and deallocation functions—`operator new` and `operator delete`. For example:

```
class alignas(64) X {
  double elem[8];

public:
  void *operator new(size_t size){
    return _mm_malloc(size, alignof(X));
  }

  void operator delete(void *p){
    return _mm_free(p);
  }
};
```

This method is easier, because the changes are centralized in the class, instead of being distributed over the uses of the class. But to get it right in general is still fairly involved, because it requires defining several more functions, in case arrays of the class are dynamically allocated or `nothrow` allocation is used.

Automatically-Aligned Dynamic Allocation

In this release of the compiler, all that is necessary in order to get correct dynamic allocation for aligned data is to include a new header:

```
#include <aligned_new>
```

After this header is included, a `new`-expression for any aligned type will automatically allocate memory with the alignment of that type.

On Windows*, it is possible to direct the compiler to include a file at the beginning of the primary source file, without modifying the source, using the `/FI` command-line option.

Implementation Details

This section explains the language rules for the new feature. If a program needs to take control of dynamic allocation and deallocation of aligned data for some reason other than alignment, this section explains how it can be done.

Header <aligned_new> defines several new alignment-aware allocation and deallocation functions, each of which takes an alignment argument:

```
void *operator new    (size_t, align_val_t);
void *operator new    (size_t, align_val_t, nothrow_t const &);
void operator delete  (void *, align_val_t);
void operator delete  (void *, align_val_t, nothrow_t const &);
void *operator new[]  (size_t, align_val_t);
void *operator new[]  (size_t, align_val_t, nothrow_t const &);
void operator delete[](void *, align_val_t);
void operator delete[](void *, align_val_t, nothrow_t const &);
```

The type `align_val_t` is declared internally by the compiler as if by a declaration like this:

```
namespace std {
  enum class align_val_t: size_t;
};
```

In other words, `std::align_val_t` is a scoped enumeration type, which can not be implicitly converted to an integer type, but has the same range and representation as `std::size_t`.

When the compiler processes a `new`-expression for a type whose alignment is greater than `(2 * sizeof(void *))`, it builds an argument list according to the normal C++ rules, but with an additional alignment argument of type `align_val_t` following the size argument (followed by the placement arguments from the `new`-expression, if any). It then uses overload resolution to try to find an alignment-aware `operator new` or `operator new[]` function that can be called with those arguments. If no alignment-aware function is found, the alignment argument is removed from the argument list, and overload resolution is attempted again. An error is reported if this second attempt fails.

Class-specific Allocation and Deallocation Functions

If a program already provides class-specific allocation and deallocation functions for an aligned class, including <aligned_new> will not change the behavior, because class-specific functions take precedence over global functions, and <aligned_new> defines only global functions.

Unless class-specific allocation and deallocation functions are written for a base class of a class hierarchy containing classes with different alignments, it is probably not necessary to write alignment-aware allocation and deallocation functions that take an alignment argument; the appropriate alignment can instead be built into the class-specific allocation and deallocation functions.

Replacing Global Allocation and Deallocation Functions

> **NOTE** If a program defines its own global allocation and deallocation functions, replacing the ones from the standard library, and uses a non-placement `new`-expression to allocate aligned data, and <aligned_new> is included before the point of such a `new`-expression, **the behavior of the program will change**. The allocation will no longer use the program's replacement allocation functions, but instead Intel's provided alignment-aware allocation functions. **In a program that replaces the global allocation and deallocation functions, care must be used to decide whether to include <aligned_new>.**

If a program wants to replace the global allocation and deallocation functions, and also wants to take advantage of the compiler's ability to provide an alignment argument to such functions, <aligned_new> should not be included, because it provides inline definitions of the alignment-aware functions, which will conflict with or take precedence over the program's definitions. Instead, <aligned_new> should be used as a guide to write program-specific declarations and definitions of the alignment-aware functions that need to be replaced.

Pointer Checker

Pointer Checker Overview

The pointer checker is not supported on macOS systems.

This feature requires installation of another product. For more information, see Feature Requirements.

The pointer checker is a debugging feature that helps you find buffer overruns in applications. The feature performs bounds checking for memory accesses through pointers and identifies any out-of-bounds access in pointer-checker enabled code. The pointer checker can also detect dangling pointers, that is, pointers that point to memory that has been freed. When this detection is enabled, using a dangling pointer in an indirect access will also cause an out-of-bounds error.

The C and C++ languages define semantics for memory access for pointers. However, many applications still make out-of-bounds memory accesses and these accesses can go undetected, risking data corruption and increasing vulnerability to malicious attacks. The pointer checker provides full checking of all memory accesses through pointers and catches out-of-bounds memory accesses before memory corruption occurs. When you compile your code with the pointer checker enabled, it identifies and reports out-of-bounds memory accesses.

The pointer checker is designed for use during application testing and debugging. Because it adds overhead in terms of the size and execution time of a program, you will want to deploy programs with the pointer checker disabled.

Your application can contain both pointer checker enabled code as well as code that is not enabled. The pointer checker allows this co-existence because it does not change the data structure layout of functions during its checking.

See Also

Pointer Checker Feature Summary
Feature Requirements

Pointer Checker Feature Summary

The pointer checker is not supported on macOS systems.

The pointer checker provides a number of related elements, summarized in the following table.

Element	Description
Compiler Options:	
`[Q]check-pointers`	Enables the pointer checker and adds the associated libraries. This compiler option enables checking of all indirect accesses through pointers and accesses to arrays. The possible option keywords are [`none` | `write` | `rw`], where: • **none**: Disables the pointer checker (default). • **write**: Checks bounds for only writes through pointers. • **rw**: Checks bounds for reads and writes through pointers. If the compiler determines that an access is safe during optimization, then the compiler removes the pointer checking code. See Checking Bounds.
`[Q]check-pointers-dangling`	Enables checking for dangling pointer references.

Element	Description
	The possible option keywords are [none | heap | stack | all], where: • **none**: Disables checking for dangling pointer references (default). • **heap**: Checks for dangling pointer references on the heap. • **stack**: Checks for dangling pointer references on the stack. • **all**: Checks for dangling pointer references on both the heap and the stack. **NOTE** To use this option, you must also use the [Q]check-pointers compiler option. This option cannot be used with [Q]check-pointers-mpx. See Checking for Dangling Pointers.
[Q]check-pointers-undimensioned	Enables the checking of bounds for arrays without dimensions. **NOTE** To use this option, you must also use the [Q]check-pointers compiler option. See Checking Arrays.
[Q]check-pointers-narrowing	Determines whether the compiler enables or disables the narrowing of pointers to structure fields. Narrowing restricts a field pointer so that it can only legally point to that field. Enables or disables the compiler narrowing of pointers to structure fields. The default is enabled for narrowing pointer references. Disabling this feature can improve Pointer Checker compatibility with non-ANSI compliant code. To disable the narrowing of pointers to structure fields, specify the negative form of the option. **NOTE** To use this option, you must also use the [Q]check-pointers compiler option.
check-pointers-mpx, Qcheck-pointers-mpx	Determines whether the compiler checks bounds for memory access through pointers on processors supporting Intel® Memory Protection Extensions (Intel® MPX). It enables checking of all indirect accesses through pointers, and all array accesses. Specifies what type of bounds checking occurs. The possible option keywords are [none | write | rw], where: • **none**: Disables the pointer checker (default). • **write**: Checks bounds for only writes through pointers. • **rw**: Checks bounds for reads and writes through pointers. The compiler may optimize these checks away when it can determine that an access is safe.

Element	Description
	This option cannot be used with `[Q]check-pointers-dangling`. If you specify option `[Q]check-pointers` along with option `[Q]check-pointers-mpx`, `[Q]check-pointers-mpx` takes precedence. On supported Windows* target platforms, MPX instructions can also be accessed using MPX intrinsic functions and the `__declspec(mpx)` feature. For more details, please see the Intel Memory Protection Extensions Enabling Guide (https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf).
Intrinsics:	
`void * __chkp_lower_bound(void **)`	Returns the lower bound associated with the pointer. See Writing a Wrapper.
`void * __chkp_upper_bound(void **)`	Returns the upper bound associated with the pointer. See Writing a Wrapper.
`void * __chkp_kill_bounds(void *p)`	Removes the bounds information to allow the pointer specified in the argument to access all memory. Use this function for a pointer from a non-enabled module that will be used in an enabled module where you cannot determine the bounds of the pointer. The function ensures that the pointer created from a non-enabled module does not inherit the bounds from another pointer that was in the same memory address. The return value is a pointer without bounds information. See Working with Enabled and Non-Enabled Modules.

`void * __chkp_make_bounds(void *p, size_t size)`	Creates new bounds information within the allocated memory address for the pointer in the argument, replacing any previously associated bounds information. The new bounds are:
	`p = __chkp_make_bounds(q, size)`
	`    // lower_bound(p) = (char *)q`
	`    // upper_bound(p) = lower_bound(p) + size`
	See Checking Custom Memory Allocators.

Reporting Function:	
`void __chkp_report_control(__chkp_report_option_t option, __chkp_callback_t callback)`	Determines how errors are reported. See Finding and Reporting Out-of-Bounds Errors.
Enumeration:	
`__chkp_report_option_t`	Controls how out-of-bounds error are reported. This enumeration is declared in the header file `chkp.h`. See Finding and Reporting Out-of-Bounds Errors.
Environment Variable:	
`INTEL_CHKP_REPORT_MODE`	Changes the pointer checker reporting mode at runtime. See Finding and Reporting Out-of-Bounds Errors.
Header file:	

Element	Description
chkp.h	Defines intrinsic and reporting functions. The header file is located in the <install-dir>\include directory.

See Also

check-pointers, Qcheck-pointers
check-pointers-dangling, Qcheck-pointers-dangling
check-pointers-undimensioned, Qcheck-pointers-undimensioned
check-pointers-narrowing, Qcheck-pointers-narrowing
check-pointers-mpx, Qcheck-pointers-mpx
__declspec(mpx)
Finding and Reporting Out-of-Bounds Errors
Working with Enabled and Non-Enabled Modules
Checking Custom Memory Allocators
Writing a Wrapper
Checking Arrays
Checking for Dangling Pointers
Checking Bounds

Using the Pointer Checker

Checking Bounds

The pointer checker is not supported on macOS systems.

The pointer checker checks indirect accesses through pointers for accesses that are out of bounds.

Checking Bounds on Read/Write Operations

To check the bounds of pointers, compile your module with [Q]check-pointers compiler option, specifying the rw argument.

You can also check bounds by specifying the write argument. This also checks the bounds of pointers, but only for pointer write operations.

Consider the case where you create an array with ten elements using the malloc() function and then you write a character to each array element:

Example: Writing to Each Array Element

```
char *buf = malloc(10);
for (int i=0; i<=10; i++) { buf[i] = ‘A’ + i; }
```

The array has ten elements, but the loop iterates eleven times. On the eleventh iteration, the function writes a character to the eleventh element of the array, which is outside of the allocated memory. Regardless of whether you specify bounds checking for read and write operations or only write operations, the pointer checker will report an out-of-bounds error. Even in the case of a statically allocated buffer, the pointer checker will still report an error. Consider this case:

Example: Out-of-bounds Error with a Statically Allocated Buffer

```
fprintf(stderr, "buf[%d]=%d\n",i,buf[i]);
```

Here, the reference to buf[i] is a read (or load) operation. Therefore, an out-of-bounds error will not be reported if you specified pointer checking only for write operations.

Pointer Arithmetic and Pointer Checking

Pointer arithmetic does not affect the pointer checker. A pointer can go out of range as long as the pointer does not make an indirect reference to an out of range address.

In the case where you create an array with 100 elements, the following applies:

Example: Pointer Arithmetic with Pointer Checking

```
char *p = malloc(100);
  p += 200;       // pointer is out of range, but no error
  p[-101] = 0; // access is still in range, it is the original p[99]
  p[0] = 0;    // out-of-bounds error occurs here, because it is original p[200]
```

See Also

check-pointers, Qcheck-pointers compiler option

Checking for Dangling Pointers

The pointer checker is not supported on macOS systems.

When dangling pointer checking or heap is enabled, the compiler uses a wrapper for the C runtime function free() and the C++ delete operator. These wrappers find all pointers that point to the block being freed, and change their bounds so that any access through the pointer will cause a bound violation. The bounds of these dangling pointers are actually set to:

- lower_bound(*p*) = 2;
- upper_bound(*p*) = 0;

If your program gets a bound violation with these bounds, it is the result of a reference through a dangling pointer.

When dangling pointer checking is enabled for stack, the compiler finds all pointers that point to the locals of the function and changes their bounds in the same way as heap pointers above, just before the function exits.

If you have a custom memory allocator, you can enable it to do dangling pointer checking. The free() function of your custom memory allocator should call this function in the pointer checker runtime code:

```
void __chkp_invalidate_dangling(void *ptr, size_t size);
```

This function is declared in the chkp.h file. You must include that header file to use this function because it uses a custom call interface.

Example

```
#include <chkp.h>
     void my_free(void *ptr) {
      size_t size = my_get_size(ptr);
     // do the free
 __chkp_invalidate_dangling(ptr, size);
     }
```

You can also enabled dangling pointer checking in any function you use to override the C++ delete operator.

See Also

check-pointers-dangling, Qcheck-pointers-dangling compiler option

Checking Arrays

The pointer checker is not supported on macOS systems.

The C and C++ language allows you to define arrays in another module with the `extern` keyword. These arrays can be defined without specifying the dimensions.

Example: Creating an Undimensioned Array

```
extern char an_undimensioned_array [];
```

The compiler allows more than one definition for externally defined arrays. During link time, the compiler uses the array definition with the largest bounds.

To check these arrays, the compiler defines a global symbol that marks the end of the array. However, checking undimensioned arrays can lead to a multiple defined linker error. To fix this linker error, do one of the following:

- Use only one array definition.
- Use the negative form of the `[Q]check-pointers-undimensioned` compiler option to disable checking arrays without bounds.

> **NOTE**
> This compiler option suppresses checking in the module that declares an array without bounds. The pointer checker will still check the arrays in modules that actually define the arrays with bounds.

See Also

check-pointers-undimensioned, Qcheck-pointers-undimensioned compiler option

Working with Enabled and Non-Enabled Modules

The pointer checker is not supported on macOS systems.

An enabled module is a module compiled with the pointer checker option enabled, while a non-enabled module is a module compiled with this compiler option disabled.

If you write a pointer to memory or return a pointer from a non-enabled module, the pointer may get incorrect bounds information. If you use this pointer with the incorrect bounds information in an enabled module, the pointer checker will report an incorrect out-of-bounds error because the bounds do not correspond to the pointer.

To minimize this issue, the pointer checker stores a copy of the pointer along with the bounds information. When the pointer is loaded into memory, the value of the pointer is compared with the value of the pointer copy. If these two values match, the bounds information is assumed to be correct and is then used. However, if the two values do not match, the bounds are set to allow access to any memory.

The pointer checker can still report an out-of-bounds error if a pointer from a non-enabled module matches the pointer copy stored with the bounds information.

For example, consider the case where you create the following pointer by using a run-time library function from a non-enabled module:

Example: Pointer Created with RTL Function

```
p = my_realloc(p, old_size + 100);
```

If the memory allocator can simply extend the memory allocated to `p`, and then returns the same pointer, an enabled module could use this pointer with the old bounds information. The pointer checker then reports an out-of-bounds error because this feature does not know about the extension created by the `realloc()` function.

To prevent incorrect out-of-bounds errors when you have both enabled and non-enabled modules, do one of the following:

- Remove the bounds information from the pointer by using the `__chkp_kill_bounds()` intrinsic function
- Set the correct bounds information by using the `__chkp_make_bounds()` intrinsic function in an enabled module.

Removing the Bounds Information

When you remove the bounds information, you disable pointer checking on this pointer. You can remove the bounds information by using the `__chkp_kill_bounds()` intrinsic function.

Example: Removing Bounds Information with __chkp_kill_bounds()

```
void * unknown_pointer_returning_function() {
   ...
   // Use the intrinsic function in the return pointer
   return __chkp_kill_bounds(the_ptr);
}
```

Setting the correct bounds information

You can use the `__chkp_make_bounds()` intrinsic function to set the correct bounds information for a pointer.

For example, you use the Windows* `HeapAlloc()` function to create a pointer. Since this operating system function is from a non-enabled module, the pointer from this function will not have the correct bounds information.

To get a pointer with the correct bounds information, use the `__chkp_make_bounds()` intrinsic function in the return value:

Example: Obtaining a Pointer with __chkp_make_bounds()

```
void * myalloc(size_t size){ return __chkp_make_bounds(HeapAlloc(MyHeap, flags, size), size); }
```

Storing Bounds Information

The pointer checker is not supported on macOS systems.

The pointer checker stores bounds information in a bounds table located in a memory address that is not adjacent to the memory address of the pointer. The pointer checker calculates this address by using the address of the pointer.

Because the bounds information is being stored in a separate memory address, use of the pointer checker in this module does not affect the data structure layouts and stack frames. You can check the bounds of pointers in enabled modules. Non-enabled modules will still work properly although this feature will not check the pointers in these modules.

When a pointer is loaded in a register, the compiler also loads the bounds from the bounds table. When a pointer is stored from a register, the compiler stores the bounds information in the bounds table.

Passing and Returning Bounds

The pointer checker is not supported on macOS systems.

When you pass a pointer to a function, the pointer checker also passes the bounds information associated with the pointer. The feature uses the following methods to pass and return arguments:

- If you pass a pointer on the stack, the pointer is in a memory location, so the pointer checker stores the bounds information when you compile your enabled module. The bounds information is stored in the bounds table entry associated with the address of the pointer.
- If you pass a pointer on a register, the compiler uses a location in thread local storage to pass the bounds. There is one such location associated with each register in which a pointer can be passed. This same location is used to return the bounds when a pointer is returned by a function.

Checking Run-Time Library Functions

The pointer checker is not supported on macOS systems.

The pointer checker provides checking on C run-time library functions that manipulate memory through pointers. It uses a library of functions that either replace the run-time library function, or wrap them with the appropriate pointer checking mechanisms.

For functions that allocate memory, such as the `malloc()` function or various C++ new functions, the wrapper function create bounds information for the pointers returned by the memory allocator.

For functions that copy memory, such as the `memcpy()` function, the memory address may contain the pointers along with their associated bounds information. The wrapper functions check for out-of-bounds accesses and ensure that any bounds associated with the copied memory are also copied.

The point checker C run-time function wrappers are located in the `libchkpwrap` library. To determine which C run-time routines are wrapped, you can examine the entry points in the library. For example, the following will yield a list of entry points:

Example

```
// Linux*
  % nm libchkpwrap.a | egrep 'T __chkp_'

// Windows* (x86)
  dumpbin /symbols libchkpwrap.lib | egrep 'SECT.*External.*
  [_]*__chkp_'
```

The returned list will include entry points that signify wrappers. For example, `__chkp_strcpy` is the wrapper for `strcpy`.

Writing a Wrapper

The pointer checker is not supported on macOS systems.

You can write your own wrappers for run-time library functions. Typically, you would use one or more of the pointer checker intrinsics.

Example: Allocation Wrapping with __chkp_make_bounds

```
extern void *wrap_malloc(size_t bytes) {
    void* ppp;
    ppp = malloc(bytes);
    if (ppp) { ppp = (void*)__chkp_make_bounds(ppp, bytes);
    } else { ppp = (void*)0;}
    return ppp;
}
```

The next example shows a wrapper that checks the validity of the pointer passed by performing writes to the first and last addresses that the C run-time routine will write. This will cause out of bounds events if necessary, while still allowing optimized handling of the C run-time library call.

Example: Checking without using Pointer Checker Intrinsics

```
extern void *wrap_memset(void *dst, int c, size_t size) {
    if (size > 0) {
      *(char *)dst = c;          // write to first address
      *((char*)dst+size-1) = c;  // write to last address
      (void)memset(dst, c, size);
    }
    return dst;
}
```

Alternatively, you can perform the checking directly by comparing to the bounds associated with the pointer. In this case, you must first make sure that the bounds are meaningful. You can use the `__chkp_upper_bound` and `__chkp_lower_bound` intrinsics for this purpose.

Example: Upper and Lower Bound Intrinsics

```
extern void *wrap_memset(void *dst, int c, size_t size) {
  if (size > 0) {
    char *ub = __chkp_upper_bound(&dst);
    if ((intptr_t)ub != (intptr_t)-1) {
      char *lb = __chkp_lower_bound(&dst);
      char *max = (char*)dst+size-1;
        if (dst < lb)
           *(char*)dst = c;  // cause bounds violation
        if (max > ub)
           *(char*)max = c;  // cause bounds violation
      }
      (void)memset(dst, c, size);
    }
    return dst;
}
```

Checking Custom Memory Allocators

The pointer checker is not supported on macOS systems.

Many C and C++ applications use standard memory allocation functions to allocate large chunks of memory and then define their own custom memory allocation functions to allocate these large chunks of memory into smaller chunks. If you use the pointer checker on a module that contains custom memory allocation functions, every memory allocation from these custom functions will have the bounds information from the large chunk of memory.

To create the correct bounds information for a pointers in a custom memory allocator function, use the `__chkp_make_bounds()` intrinsic function.

For example, consider the case where you create a custom memory allocator function that returns a pointer. To add the exact bounds information to the return pointer, use the `__chkp_make_bounds()` intrinsic function in the return value:

Adding exact bounds information to a return pointer

```
void *myalloc(size_t size) {
      // Code to do allocate the large chunk of memory into small chunks.
      // Add bounds information to the pointer
   return __chkp_make_bounds(p, size);
}
```

NOTE
If you override the `new` operator in C++, you can use the same technique to give bounds information to the return pointer.

Checking Multi-Threaded Code

The pointer checker is not supported on macOS systems.

A common assumption is that reading or writing a pointer is an atomic operation that cannot be interrupted by starting another thread. This is not the case with using the pointer checker to check pointers in multi-threaded modules.

When you read or write a pointer from memory, the bounds information associated with the pointer must also be read or written. Reading and writing bounds information takes multiple instructions. While a thread is in the process of writing a pointer and its bounds, it could be swapped out for another thread. If that thread then writes to the same pointer, you can end up with a pointer and bounds information that are not synchronized—the pointer is from one thread and the bounds information is from another thread.

To synchronize the pointer and bounds information in multi-threaded code, use a locking mechanism, such as a mutex or critical section when reading or writing a pointer in memory locations shared by more than one thread. Typically, accesses to shared memory are already protected this way.

If your application relies on a pointer read or a pointer write that is atomic and performs reads or writes to shared pointers without such locking, you can get extraneous bounds violations unless you protect these accesses.

How the Compiler Defines Bounds Information for Pointers

The pointer checker is not supported on macOS systems.

The following defines how the compiler determines the bound information for pointers.

NOTE
In each section, `lower_bound(`*p*`)` refers to the lower bound associated with *p* and `upper_bound(`*p*`)` refers to the upper bound associated with *p*.

Pointers created by the alloca() function

```
p = alloca(size);
    // lower_bound (p) is (char *)p
    // upper_bound (p) is lower_bound(p) + size - 1
```

Pointers created by the calloc() function

```
p = calloc(num, size);
    // lower_bound(p) is (char *)p
    // upper_bound(p) is lower_bound(p) + size * num - 1
```

Pointers created by the `malloc()` function

`p = malloc(size);`
`    // lower_bound(p) is (char *)p`
`    // upper_bound(p) is lower_bound(p) + size - 1`

Pointers created by casting

`p = (T *)q;`
`    // lower_bound(p) is lower_bound(q)`
`    // upper_bound(p) is upper_bound(q)`

Casting a pointer does not affect the bounds of a pointer. If you cast a pointer to a new type that is larger than the bounds associated with the original pointer, you will get an out-of-bounds error when you try to access any member or element outside the original bounds. If you cast a pointer to a smaller type than the original pointer, you can still access the original data.

Pointers created for a variable length array in a structure

```
typedef struct {
    int num;
    int a[];
} T;

q = malloc(sizeof(T) + sizeof(int) * num);
p = &q->a;
    // lower_bound(p) is (char *)&q->a
    // upper_bound(p) is upper_bound(q)
```

When you define an array as the last member of a structure, the upper bound is not narrowed and is allowed to access all of the array elements allocated by the `malloc()` function.

Pointers defined by the address (&) operator

```
p = &v;
    // lower_bound(p) is (char *)&v
    // upper_bound(p) is (char *)&v + sizeof(v) - 1

p = &v.m;
    // lower_bound(p) is (char *)&v + offsetof(typeof(v),  m)
    // upper_bound(p) is lower_bound(p) + sizeof(v.m) - 1

p = &q->m;
    // lower_bound(p) is (char *)q + offsetof(typeof(*q),  m)
    // upper_bound(p) is lower_bound(p) + sizeof(q->m) - 1
```

NOTE
The bounds information is narrowed to the size of the member when you point to a member of a structure, union, or class.

Pointers defined by the new operator

```
p = new T;
    // lower_bound(p) is (char *)p
    // upper_bound(p) is lower_bound(p) + sizeof(T) - 1
```

Pointers defined by the addresses in an array

```
T a[X][Y];

p = a;
p = &a[x];
p = &a[x][y];
    // lower_bound(p) is (char *)a
    // upper_bound(p) is lower_bound(p) + sizeof(a) - 1
```

When you take the address of an element of an array or the address of a single row of a multi-dimensioned array, the bounds are not narrowed to the size of the element. You can increment or decrement the pointer throughout the array.

Incrementing and Decrementing Pointers

```
p = &a[x][y].m;
    // lower_bound(p) is (char *)&a[n][m] + offsetof(T, m)
    // upper_bound(p) is lower_bound(p) + sizeof(T.m) - 1
```

When you take the address of a member of an element, the bounds are narrowed to the size of the member.

Pointers defined by pointer copies

```
p = q;
p = q + expr;
p = q - expr;
    // lower_bound(p) is lower_bound(q)
    // upper_bound(p) is upper_bound(q)
```

The bounds are copied from *q*. Offsetting the pointer on the right does not affect the bounds.

Pointers defined by incrementing or decrementing a pointer

```
p++;
p--;
++p;
--p;
p += expr;
p -= expr;
```

The bounds do not change when you increment or decrement a pointer.

Finding and Reporting Out-of-Bounds Errors

The pointer checker is not supported on macOS systems.

The pointer checker includes the `__chkp_report_control()` library function and the `__chkp_report_option_t` enumeration to allow you to control how errors are reported. The function and enumeration are declared in the header file `chkp.h`.

The report control enumeration has one of the following values:

Enum Value	Action
`__CHKP_REPORT_NONE`	Do nothing.
`__CHKP_REPORT_BPT`	Execute a breakpoint interrupt. If you specify this value, the pointer checker will issue a breakpoint for any out-of-bounds error that it finds. If you are using a debugger, the breakpoint will trap into the debugger so that you can determine where the error occurred. You can then use the features of the debugger to determine the cause of the error.
`__CHKP_REPORT_LOG`	Log the error and continue; the compiler will report each out-of-bounds pointer it finds.
`__CHKP_REPORT_TERM`	Log the error and exit the program; the compiler will only report the first bounds violation and then terminate.
`__CHKP_REPORT_CALLBACK`	Call a user defined function; the compiler will invoke a user-defined function to deal with a bounds error.
`__CHKP_REPORT_TRACEBACK _BPT`	Print a traceback including source file and line number for the instruction where the out-of-bounds error occurred, then execute a breakpoint interrupt. **NOTE** Specify the `traceback` compiler option to obtain better traceback information, including routine names.
`__CHKP_REPORT_TRACE_LOG`	Log the error and continue; the log will include traceback information for each out-of-bounds error. This is the default reporting mode. **NOTE** Specify the `traceback` compiler option to obtain better traceback information, including routine names.
`__CHKP_REPORT_TRACE_TERM`	Log the error and terminate; the log will include traceback information for each out-of-bounds error. Only the first bounds error will be reported. **NOTE** Specify the `traceback` compiler option to obtain better traceback information, including routine names.
`__CHKP_REPORT_TRACE_CALLBACK`	Log the error and call a user-defined routine; the log will include traceback information for each out-of-bounds error. **NOTE** Specify the `traceback` compiler option to obtain better traceback information, including routine names.
`__CHKP_REPORT_OOB_STATS`	Emit statistics for the bounds violation; Currently, this is a count of the out-of-bounds errors.

Enum Value	Action
__CHKP_REPORT_USE_ENV_VAR	Use the environment variable INTEL_CHKP_REPORT_MODE to specify the reporting mode. If the environment variable is not set, the default reporting mode is used.

Changing the Reporting Mode

To change the reporting mode from the default __CHKP_REPORT_TRACE_LOG:

1. Include chkp.h in your program source.
2. Add a call to the report control routine __chkp_report_control() (before any pointer references are made), specifying one of the enum values.

   For example, to report all bounds errors, specify the following:

```
__chkp_report_control(__CHKP_REPORT_LOG, 0);
```

In the above, the first parameter to the routine is the enum value and the second parameter is 0, except in the case of the __CHKP_REPORT_CALLBACK enum value, which requires the name of a user-defined callback routine as the second parameter.

You can also change the reporting mode using the environment variable INTEL_CHKP_REPORT_MODE. This allows you to change the reporting mode without recompiling your code. To use the environment variable, do the following:

1. Add an include of chkp.h in your program source.
2. Add a call to the report control routine __chkp_report_control() (before any pointer references are made), specifying __CHKP_REPORT_USE_ENV_VAR.
3. Set the INTEL_CHKP_REPORT_MODE environment variable to the desired report mode. For example:

```
export INTEL_CHKP_REPORT_MODE=__CHKP_REPORT_OOB_STATS
```

NOTE
The INTEL_CHKP_REPORT_MODE environment variable is valid only if a call to __chkp_report_control has been made with the report mode set to __CHKP_REPORT_USE_ENV_VAR. Otherwise, it is ignored.

If you specify the report mode to be __CHKP_REPORT_USE_ENV_VAR and the INTEL_CHKP_REPORT_MODE environment variable is not set, the default report mode (__CHKP_REPORT_TRACE_LOG) is used.

See Also

/Zi compiler option

traceback, notraceback compiler option

Tools

PGO Tools

PGO Tools Overview

This section describes the tools that take advantage of or support the Profile-guided Optimizations (PGO) available in the compiler.

- Code coverage Tool
- Test prioritization Tool
- Profmerge and proforder Tools

Code Coverage Tool

The code coverage tool provides software developers with a view of how much application code is exercised when a specific workload is applied to the application. To determine which code is used, the code coverage tool uses Profile-guided Optimization (PGO) options and optimizations. The major features of the code coverage tool are listed below:

- Visually presenting code coverage information for an application with a customizable code coverage coloring scheme
- Displaying dynamic execution counts of each basic block of the application
- Providing differential coverage, or comparison, profile data for two runs of an application

The information about using the code coverage tool is separated into the following sections:

- Code coverage tool Requirements
- Visually Presenting Code Coverage for an Application
- Excluding Code from Coverage Analysis
- Exporting Coverage Data

The tool analyzes static profile information generated by the compiler, as well as dynamic profile information generated by running an instrumented form of the application binaries on the workload. The tool can generate an HTML-formatted report and export data in both text-, and XML-formatted files. The reports can be further customized to show color-coded, annotated, source-code listings that distinguish between used and unused code.

The code coverage tool is available on all supported Intel architectures on Linux*, Windows*, and macOS operating systems.

You can use the tool in a number of ways to improve development efficiency, reduce defects, and increase application performance:

- During the project testing phase, the tool can measure the overall quality of testing by showing how much code is actually tested.
- When applied to the profile of a performance workload, the code coverage tool can reveal how well the workload exercises the critical code in an application. High coverage of performance-critical modules is essential to taking full advantage of the Profile-Guided Optimizations that Intel Compilers offer.
- The tool provides an option useful for both coverage and performance tuning, enabling developers to display the dynamic execution count for each basic block of the application.
- The code coverage tool can compare the profile of two different application runs. This feature can help locate portions of the code in an application that are unrevealed during testing but are exercised when the application is used outside the test space, for example, when used by a customer.

Code Coverage Tool Requirements

To run the code coverage tool on an application, you must have following items:

- The application sources.
- The .spi file generated by the Intel® compiler when compiling the application for the instrumented binaries using the `-prof-gen=srcpos` (Linux and macOS) or `/Qprof-gen:srcpos` (Windows) option.

NOTE
Use the `-[Q]prof-gen:srcpos` option if you intend to use the collected data for code coverage and profile feedback. If you are only interested in using the instrumentation for code coverage, use the `/Qcov-gen` option. Using the `/Qcov-gen` option saves time and improves performance. This option can be used only on Windows platform for all architectures.

- A `pgopti.dpi` file that contains the results of merging the dynamic profile information (.dyn) files, which is most easily generated by the profmerge tool. This file is also generated implicitly by the Intel® compilers when compiling an application with `[Q]prof-use` options with available .dyn and .dpi files.

Using the Tool

The tool uses the following syntax:

Tool Syntax
`codecov [-codecov_option]`

where `-codecov_option` is one or more optional parameters specifying the tool option passed to the tool. The available tool options are listed in the code coverage tools Options section. If you do not use any additional tool options, the tool will provide the top-level code coverage for the entire application.

In general, you must perform the following steps to use the code coverage tool:

1. Compile the application using `-prof-gen=srcpos` (Linux and macOS) or `/Qprof-gen:srcpos` (Windows), and/or `/Qcov-gen` (Windows) option.

This step generates an instrumented executable and a corresponding static profile information (`pgopti.spi`) file when the `[Q]prof-gen=srcpos` option is used. When the `/Qcov-gen` option is used, minimum instrumentation only for code coverage and generation of .spi file is enabled.

> **NOTE**
> You can specify both the `/Qprof-gen=srcpos` and `/Qcov-gen` options on the command line. The higher level of instrumentation needed for profile feedback is enabled along with the profile option for generating the .spi file, regardless of the order the options are specified on the command line.

2. Run the instrumented application.

This step creates the dynamic profile information (.dyn) file. Each time you run the instrumented application, the compiler generates a unique .dyn file either in the current directory or the directory specified in by the -prof-dir (Linux or macOS) or /Qprof-dir (Windows) option, or PROF_DIR environment variable. On Windows, you can use the /Qcov-dir or COV_DIR environment variable. These have the same meaning as /Qprof-dir and PROF_DIR.

3. Use the profmerge tool to merge all the .dyn files into one .dpi (`pgopti.dpi`) file.

This step consolidates results from all runs and represents the total profile information for the application, generates an optimized binary, and creates the dpi file needed by the code coverage tool.

You can use the profmerge tool to merge the .dyn files into a .dpi file without recompiling the application. The profmerge tool can also merge multiple .dpi files into one .dpi file using the `profmerge -a` option. Select an alternate name for the output .dpi file using the `profmerge -prof_dpi` option.

> **Caution**
> The profmerge tool merges all .dyn files that exist in the given directory. Confirm that unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code.

4. Run the code coverage tool. The valid syntax and tool options are shown below.

This step creates a report or exported data as specified. If no other options are specified, the code coverage tool creates a single HTML file (CODE_COVERAGE.HTML) and a sub-directory (CodeCoverage) in the current directory. Open the file in a web browser to view the reports.

NOTE
Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

The code coverage tool allows you to name the project and specify paths to specific, necessary files. The following example demonstrates how to name a project and specify .dpi and .spi files to use:

Example: specify .dpi and .spi files

```
codecov -prj myProject -spi pgopti.spi -dpi pgopti.dpi
```

The tool can add a contact name and generate an email link for that contact at the bottom of each HTML page. This provides a way to send an electronic message to the named contact. The following example demonstrates how to add specify a contact and the email links:

Example: add contact information

```
codecov -prj myProject -mname JoeSmith -maddr js@company.com
```

This following example demonstrates how to use the tool to specify the project name, specify the dynamic profile information file, and specify the output format and file name.

Example: export data to text

```
codecov -prj test1 -dpi test1.dpi -txtbcvrg test1_bcvrg.txt
```

Code Coverage Tool Options

Option	Default	Description
`-bcolor`*color*	#FFFF99	Specifies the HTML color name for code in the uncovered blocks.
`-beginblkdsbl`*string*		Specifies the comment that marks the beginning of the code fragment to be ignored by the coverage tool.
`-blockcounts`		When used with `-txtlcov`, reports individual bloc counts for lines that involved multiple blocks.
`-ccolor`*color*	#FFFFFF	Specifies the HTML color name or code of the covered code.
`-comp`*file*		Specifies the file name that contains the list of files being (or not) displayed.
`-counts`		Generates dynamic execution counts.
`-demang`		Demangles both function names and their arguments.
`-dpi`*file*	`pgopti.dpi`	Specifies the file name of the dynamic profile information file (.dpi).
`-endblkdsbl`*string*		Specifies the comment that marks the end of the code fragment to be ignored by the coverage tool.
`-fcolor`*color*	#FFCCCC	Specifies the HTML color name for code of the uncovered functions.
`-help`, `-h`		Prints tool option descriptions.

Option	Default	Description
-icolor*color*	#FFFFFF	Specifies the HTML color name or code of the information lines, such as basic-block markers and dynamic counts.
-include-nonexec		Block details will also be listed for functions that did not execute, when used with –xmlbcvrg[full] or –txtbcvrg[full] options.
-maddr*string*	Nobody	Sets the email address of the web-page owner
-mname*string*	Nobody	Sets the name of the web-page owner.
-nopartial		Treats partially covered code as fully covered code.
-nopmeter		Turns off the progress meter. The meter is enabled by default.
-nounwind		Ignores compiler-generated unwind handlers for exception handling cleanup when computing and displaying basic block coverage.
-onelinedsblstring		Specifies the comment that marks individual lines of code or the whole functions to be ignored by the coverage tool.
-pcolor*color*	#FAFAD2	Specifies the HTML color name or code of the partially covered code.
-prj*string*		Sets the project name.
-ref		Finds the differential coverage with respect to ref_dpi_file.
-showdirnames		Displays the full path name for source files in the HTML report, instead of just the base filename.
-spi*file*	pgopti.spi	Specifies the file name of the static profile information file (.spi).
-srcroot*dir*		Specifies a different top level project directory than was used during compiler instrumentation run to use for relative paths to source files in place of absolute paths.

Option	Default	Description

		NOTE
		In order for the substitution to take place, the sources need to be compiled with one of the following options: `[Q]prof-src-root`, `[Q]prof-src-root-cwd`. This option specifies the base directory that is to be treated as the project root directory.
		An example of use is:
		`C:> ifort -Qprof-gen:srcpos -Qprof-src-root c:\workspaces\orig_project_dir test1.f90 test2.f90`
		`C:> test1.exe`
		`C:> profmerge`
		`C:> cd \workspaces\`
		`C:> mv orig_project_dir new_project_dir`
		`C:> cd new_project_dir\src`
		`C:> codecov -srcroot C:\workspaces\new_project_dir`
		Now, "C:\workspaces\new_project_dir" will be substituted for "c:\workspaces\orig_project_dir" when looking for the source files.
		For use of `[Q]prof-src-root`, `[Q]prof-src-root-cwd` options, refer to prof-src-root/Qprof-src-root, prof-src-root-cwd/Qprof-src-root-cwd

`-txtbcvrg`*file*		Export block-coverage for covered functions as text format. The file parameter must be in the form of a valid file name.
`-txtbcvrgfull`*file*		Export block-coverage for entire application in text and HTML formats. The file parameter must be in the form of a valid file name.
`-txtdcg`*file*		Generates the dynamic call-graph information in text format. The file parameter must be in the form of a valid file name.
`-txtfcvrg`*file*		Export function coverage for covered function in text format. The file parameter must by in the form of a valid file name.
`-txtlcov`*file*		Generates line coverage in text format output files, instead of block coverage in HTML output files.
`-ucolor`*color*	#FFFFFF	Specifies the HTML color name or code of the unknown code.
`-xcolor`*color*	#90EE90	Specifies the HTML color of the code ignored.
`-xmlbcvrg`*file*		Export the block-coverage for the covered function in XML format. The file parameter must by in the form of a valid file name.
`-xmlbcvrgfull`*file*		Export function coverage for entire application in XML format in addition to HTML output. The file parameter must be in the form of a valid file name.

Option	Default	Description
`-xmlfcvrg`*file*		Export function coverage for covered function in XML format. The file parameter must be in the form of a valid file name.

Visually Presenting Code Coverage for an Application

Based on the profile information collected from running the instrumented binaries when testing an application, the Intel® compiler will create HTML-formatted reports using the code coverage tool. These reports indicate portions of the source code that were or were not exercised by the tests. When applied to the profile of the performance workloads, the code coverage information shows how well the training workload covers the application's critical code. High coverage of performance-critical modules is essential to taking full advantage of the profile-guided optimizations.

The code coverage tool can create two levels of coverage:

- Top level (for a group of selected modules)
- Individual module source views

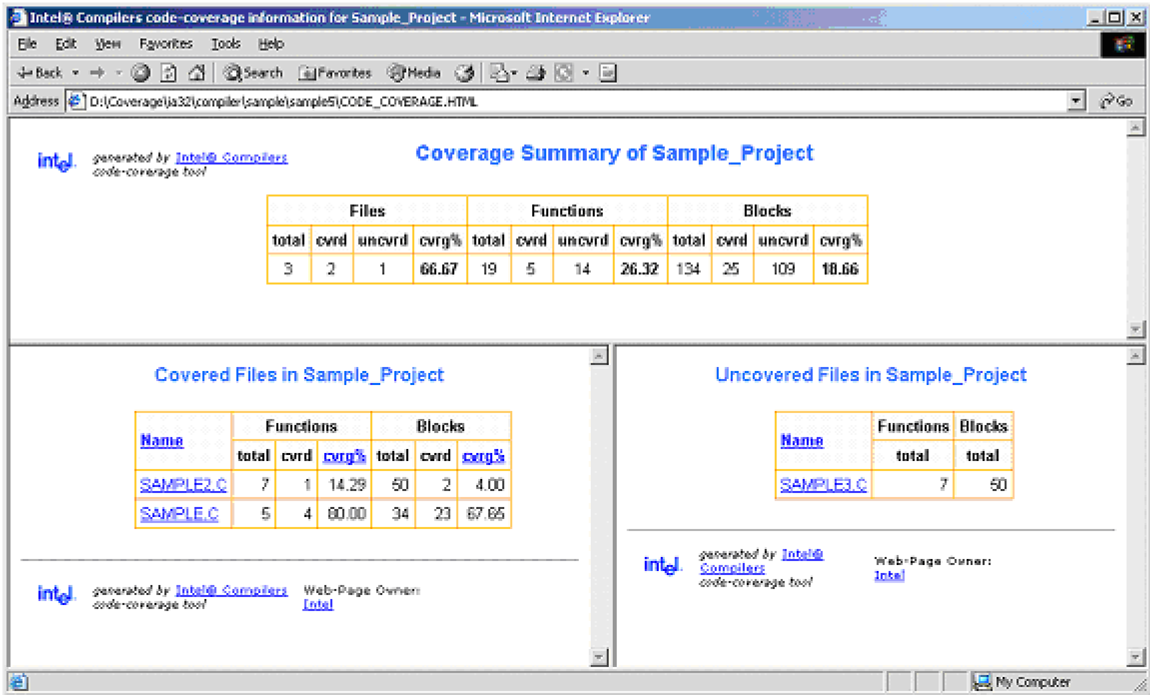
Top-Level Coverage

The top-level coverage reports the overall code coverage of the modules that were selected. The following options are provided:

- Select the modules of interest.
- For the selected modules, the tool generates a list with their coverage information. The information includes the total number of functions and blocks in a module and the portions that were covered.
- By clicking on the title of columns in the reported tables, the lists may be sorted in ascending or descending order based on:
  - Basic-block coverage
  - Function coverage
  - Function name

By default, the code coverage tool generates a single HTML file (CODE_COVERAGE.HTML) and a subdirectory (CodeCoverage) in the current directory. The HTML file defines a frameset to display all of the other generated reports. Open the HTML file in a web-browser. The tool places all other generated report files in a CodeCoverage subdirectory.

If you choose to generate the html-formatted version of the report, you can view coverage source of that particular module directly from a browser. The following figure shows the top-level coverage report.



The coverage tool creates a frame set that allows quick browsing through the code to identify uncovered code. The top frame displays the list of uncovered functions while the bottom frame displays the list of covered functions. For uncovered functions, the total number of basic blocks of each function is also displayed. For covered functions, both the total number of blocks and the number of covered blocks as well as their ratio (that is, the coverage rate) are displayed.

For example, 66.67(4/6) indicates that four out of the six blocks of the corresponding function were covered. The block coverage rate of that function is thus 66.67%. These lists can be sorted based on the coverage rate, number of blocks, or function names. Function names are linked to the position in source view where the function body starts. With one click you can see the least-covered function in the list, and with another click you can see the body of the function. You can scroll down in the source view and browse through the function body.

Individual Module Source View

Within the individual module source views, the tool provides the list of uncovered functions as well as the list of covered functions. The lists are reported in two distinct frames that provide easy navigation of the source code. The lists can be sorted based on:

- Number of blocks within uncovered functions
- Block coverage in the case of covered functions
- Function names

Setting the Coloring Scheme for the Code Coverage

The tool provides a visible coloring distinction of the following coverage categories: covered code, uncovered basic blocks, uncovered functions, partially covered code, and unknown code. The default colors that the tool uses for presenting the coverage information are shown in the tables that follows:

Category	Default	Description
Covered code	#FFFFFF	Indicates code was exercised by the tests. You can override the default color with the `-ccolor` tool option.

Category	Default	Description
Uncovered basic block	#FFFF99	Indicates the basic blocks that were not exercised by any of the tests. However, these blocks were within functions that were executed during the tests. You can override the default color with the `-bcolor` tool option.
Uncovered function	#FFCCCC	Indicates functions that were never called during the tests. You can override the default color with the `-fcolor` tool option.
Partially covered code	#FAFAD2	Indicates that more than one basic block was generated for the code at this position. Some of the blocks were covered and some were not. You can override the default color with the `-pcolor` tool option.
Ignored code	#90EE90	Indicates code that was specifically marked to be ignored. You can override this default color using the `-xcolor` tool option.
Information lines	#FFFFFF	Indicates basic-block markers and dynamic counts. You can override the default color with the `-icolor` tool option.
Unknown	#FFFFFF	Indicates that no code was generated for this source line. Most probably, the source at this position is a comment, a header-file inclusion, or a variable declaration. You can override the default color with the `-ucolor` tool option.

The default colors can be customized to be any valid HTML color name or hexadecimal value using the options mentioned for each coverage category in the table above.

For code coverage colored presentation, the coverage tool uses the following heuristic: source characters are scanned until reaching a position in the source that is indicated by the profile information as the beginning of a basic block. If the profile information for that basic block indicates that a coverage category changes, then the tool changes the color corresponding to the coverage condition of that portion of the code, and the coverage tool inserts the appropriate color change in the HTML-formatted report files.

NOTE
You need to interpret the colors in the context of the code. For example, comment lines that follow a basic block that was never executed would be colored in the same color as the uncovered blocks. Another example is the closing brackets in C/C++ applications.

Dynamic Counters

The coverage tool can be configured to generate the information about the dynamic execution counts. This ability can display the dynamic execution count of each basic block of the application and is useful for both coverage and performance tuning.

The custom configuration requires using the `-counts` option. The counts information is displayed under the code after a "^" sign precisely under the source position where the corresponding basic block begins.

If more than one basic block is generated for the code at a source position (for example, for macros), then the total number of such blocks and the number of the blocks that were executed are also displayed in front of the execution count. For example, line 11 in the code is an `if` statement:

Example

```
11  if ((N == 1).OR. (N == 0))
     ^ 10 (1/2)
12      printf("%d\n", N)
        ^
```

The coverage lines under code lines 11 and 12 contain the following information:

- The `IF` statement in line 11 was executed 10 times.
- Two basic blocks were generated for the `IF` statement in line 11.
- Only one of the two blocks was executed, resulting in the partial coverage color.
- Only seven out of the ten times variable n had a value of 0 or 1.

In certain situations, it may be desirable to consider all the blocks generated for a single source position as one entity. In such cases, it is necessary to assume that all blocks generated for one source position are covered when at least one of the blocks is covered. This assumption can be configured with the `-nopartial` option. When this option is specified, decision coverage is disabled, and the related statistics are adjusted accordingly. The code lines 11 and 12 indicate that the `print` statement in line 12 was covered. However, only one of the conditions in line 11 was ever true. With the `-nopartial` option, the tool treats the partially covered code (like the code on line 11) as covered.

Differential Coverage

Using the code coverage tool, you can compare the profiles from two runs of an application: a reference run, and a new run identifying the code that is covered by the new run but not covered by the reference run. Use this feature to find the portion of the applications code that is not covered by the applications tests but is executed when the application is run by a customer. It can also be used to find the incremental coverage impact of newly added tests to an applications test space.

Generating Reference Data

Create the dynamic profile information for the reference data, which can be used in differential coverage reporting later, by using the `-ref` option. The following command demonstrate a typical command for generating the reference data:

Example: generating reference data

```
codecov -prj Project_Name -dpi customer.dpi -ref appTests.dpi
```

The coverage statistics of a differential-coverage run shows the percentage of the code exercised on a new run but missed in the reference run. In such cases, the tool shows only the modules that included the code that was not covered. Keep this in mind when viewing the coloring scheme in the source views.

The code with the same coverage property (covered or not covered) on both runs is considered covered code. Otherwise, if the new run indicates that the code was executed, while in the reference run the code was not executed, then the code is treated as uncovered. Alternately, if the code is covered in the reference run but not covered in the new run, the differential-coverage source view shows the code as covered.

Running Differential Coverage

To run the code coverage tool for differential coverage, you must have the application sources, the .spi file, and the .dpi file, as described in the code coverage tool Requirements section (above).

Once the required files are available, enter a command similar to the following begin the process of differential coverage analysis:

Example

```
codecov -prj Project_Name -spi pgopti.spi -dpi pgopti.dpi
```

Specify the .dpi and .spi files using the `-spi` and `-dpi` options.

Excluding Code from Coverage Analysis

The code coverage tool allows you to exclude portions of your code from coverage analysis. This ability can be useful during development; for example, certain portions of code might include functions used for debugging only. The test case should not include tests for functionality that will be unavailable in the final application.

Another example of code that can be excluded is code that might be designed to deal with internal errors unlikely to occur in the application. In such cases, lack of a test case is preferred. You may want to ignore infeasible (dead) code in the coverage analysis. The code coverage tool provides several options for marking portions of the code infeasible and ignoring the code at the file level, function level, line level, and arbitrary code boundaries indicated by user-specific comments. The following sections explain how to exclude code at different levels.

Including and Excluding Coverage at the File Level

The code coverage tool provides the ability to selectively include or exclude files for analysis. Create a component file and add the appropriate string values that indicate the file and directory name for code you want included or excluded. Pass the file name as a parameter of the `-comp` option. The following example shows the general command:

Example: specifying a component file

```
codecov -comp file
```

where *file* is the name of a text file containing strings that act as file/directory name masks for including and excluding file-level analysis. For example, assume the following:

- You want to include all files with the string "source" in the file name or directory name.
- You create a component text file named `myComp.txt` with the selective inclusion string "source".

Once you have a component file, enter a command similar to the following:

Example

```
codecov -comp myComp.txt
```

In this example, filenames with string "source" (like source1.c and source2.c) and all files within directories where the directory name contains the string "source" (like source/file1.c and source2\file2.c) are included in the analysis.

To exclude files or directories, add the tilde (~) prefix to the string. You can specify inclusion and exclusion in the same component file. For example, assume you want to analyze all individual files or files in a directory where the file/directory name includes the string "source", and you want to exclude all individual files and files in directories where the file/directory name includes the string "skip". You add content similar to the following to the component file (`myComp.txt`) and pass it to the `-comp` option:

Example: inclusion and exclusion strings in the myComp.txt file

```
source

~skip
```

Entering the `codecov -comp myComp.txt` command with both instructions in the component file, `myComp.txt`, instructs the tool to:

- Include files with filename containing "source" (like source1.c and source2.c)
- Include all files in directories with the directory name containing "source" (like source/file1.c and source2\file2.c)

- Exclude all files with filename containing "skip" (like skipthis1.c and skipthis2.c)
- Exclude all files in directories with the directory name containing "skip" (like skipthese1\debug1.c and skipthese2\debug2.c)

Excluding Coverage at the Line and Function Level

You can mark individual lines for exclusion my passing string values to the `-onelinedsbl` option. For example, assume that you have some code similar to the following:

Sample code

```
printf ("internal error 123 - please report!\n");  // NO_COVER
printf ("internal error 456 - please report!\n");  /* INF IA-32 architecture */
```

If you wanted to exclude all functions marked with the comments NO_COVER or INF IA-32 architecture, you would enter a command similar to the following.

Example

```
codecov -onelinedsbl NO_COVER -onelinedsbl "INF IA-32 architecture"
```

You can specify multiple exclusion strings simultaneously, and you can specify any string values for the markers; however, you must remember the following guidelines when using this option:

- Inline comments must occur at the end of the statement.
- The string must be a part of an inline comment.

An entire function can be excluded from coverage analysis using the same methods. For example, the following function will be ignored from the coverage analysis when you issue example command shown above.

Sample code

```
void dumpInfo (int n)
{      // NO_COVER
...
}
```

Additionally, you can use the code coverage tool to color the infeasible code with any valid HTML color code by combining the `-onelinedsbl` and `-xcolor` options. The following example commands demonstrate the combination:

Example: combining tool options

```
codecov -onelinedsbl INF -xcolor lightgreen

codecov -onelinedsbl INF -xcolor #CCFFCC
```

Excluding Code by Defining Arbitrary Boundaries

The code coverage tool provides the ability to arbitrarily exclude code from coverage analysis. This feature is most useful where the excluded code either occur inside of a function or spans several functions.

Use the `-beginblkdsbl` and `-endblkdsbl` options to mark the beginning and end (respectively) of any arbitrarily defined boundary to exclude code from analysis. Remember the following guidelines when using these options:

- Inline comments must occur at the end of the statement.
- The string must be a part of an inline comment.

For example assume that you have the following code:

Sample code

```
void div (int m, int n) {
if (n == 0)
/* BEGIN_INF */
{ printf (internal error 314 please report\n);
recover (); }
/* END_INF */
else { ... }
}
...
// BINF
Void recover () { ... }
// EINF
```

The following example commands demonstrate how to use the `-beginblkdsbl` option to mark the beginning and the `-endblkdsbl` option to mark the end of code to exclude from the sample shown above.

Example: arbitrary code marker commands

```
codecov -xcolor #ccFFCC -beginblkdsbl BINF -endblkdsbl EINF

codecov -xcolor #ccFFCC -beginblkdsbl "BEGIN_INF" -endblkdsbl "END_INF"
```

Notice that you can combine these options in combination with the `-xcolor` option.

Exporting Coverage Data

The code coverage tool provides specific options to extract coverage data from the dynamic profile information (.dpi files) that result from running instrumented application binaries under various workloads. The tool can export the coverage data in various formats for post-processing and direct loading into databases: the default HTML, text, and XML. You can choose to export data at the function and basic block levels.

There are two basic methods for exporting the data: quick export and combined export. Each method has associated options supported by the tool

- **Quick export:** The first method is to export the data coverage to text- or XML-formatted files without generating the default HTML report. The application sources need not be present for this method. The code coverage tool creates reports and provides statistics only about the portions of the application executed. The resulting analysis and reporting occurs quickly, which makes it practical to apply the coverage tool to the dynamic profile information (the .dpi file) for every test case in a given test space instead of applying the tool to the profile of individual test suites or the merge of all test suites. The `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg` and `-txtbcvrg` options support the first method.
- **Combined export:** The second method is to generate the default HTML and simultaneously export the data to text- and XML-formatted files. This process is slower than first method since the application sources are parsed and reports generated. The `-xmlbcvrgfull` and `-txtbcvrgfull` options support the second method.

These export methods provide the means to quickly extend the code coverage reporting capabilities by supplying consistently formatted output from the code coverage tool. You can extend these by creating additional reporting tools on top of these report files.

Quick Export

The profile of covered functions of an application can be exported quickly using the `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg`, and `-txtbcvrg` options. When using any of these options, specify the output file that will contain the coverage report. For example, enter a command similar to the following to generate a report of covered functions in XML formatted output:

Example: quick export of function data

```
codecov -prj test1 -dpi test1.dpi -xmlfcvrg test1_fcvrg.xml
```

The resulting report will show how many times each function was executed and the total number of blocks of each function, together with the number of covered blocks and the block coverage of each function. The following example shows some of the content of a typical XML report.

XML-formatted report example

```
<PROJECT name = "test1">
   <MODULE name = "D:\SAMPLE.C">
      <FUNCTION name="f0" freq="2">
         <BLOCKS total="6" covered="5" coverage="83.33%"></BLOCKS>
      </FUNCTION>
      ...
   </MODULE>
   <MODULE name = "D:\SAMPLE2.C">
      ...
   </MODULE>
</PROJECT>
```

In the above example, we note that function f0, which is defined in file sample.c, has been executed twice. It has a total number of six basic blocks, five of which are executed, resulting in an 83.33% basic block coverage.

You can also export the data in text format using the `-txtfcvrg` option. The generated text report, using this option, for the above example would be similar to the following example:

Text-formatted report example

```
Covered Functions in File: "D:\SAMPLE.C"
"f0"    2       6       5        83.33
"f1"    1       6       4        66.67
"f2"    1       6       3        50.00
...
```

In the text formatted version of the report, the each line of the report should be read in the following manner:

Column 1	Column 2	Column 3	Column 4	Column 5
Function name	Execution frequency	Line number of the start of the function definition	Column number of the start of the function definition	Percentage of basic-block coverage of the function

Additionally, the tool supports exporting the block level coverage data using the `-xmlbcvrg` option. For example, enter a command similar to the following to generate a report of covered blocks in XML formatted output:

Example: quick export of block data to XML

```
codecov -prj test1 -dpi test1.dpi -xmlbcvrg test1_bcvrg.xml
```

The example command shown above would generate XML-formatted results similar to the following:

XML-formatted report example

```
<PROJECT name = "test1">
  <MODULE name = "D:\SAMPLE.cpp">
    <FUNCTION name="f0" freq="2">
    ...
      <BLOCK line="11" col="2">
        <INSTANCE id="1" freq="1"> </INSTANCE>
      </BLOCK>
      <BLOCK line="12" col="3">
        <INSTANCE id="1" freq="2"> </INSTANCE>
        <INSTANCE id="2" freq="1"> </INSTANCE>
      </BLOCK>
```

In the sample report, notice that one basic block is generated for the code in function f0 at the line 11, column 2 of the file sample.cpp. This particular block has been executed only once. Also notice that there are two basic blocks generated for the code that starts at line 12, column 3 of file. One of these blocks, which has id = 1, has been executed two times, while the other block has been executed only once. A similar report in text format can be generated through the `-txtbcvrg` option.

Combined Exports

The code coverage tool has also the capability of exporting coverage data in the default HTML format while simultaneously generating the text- and XML-formatted reports.

Use the `-xmlbcvrgfull` and `-txtbcvrgfull` options to generate reports in all supported formats in a single run. These options export the basic-block level coverage data while simultaneously generating the HTML reports. These options generate more complete reports since they include analysis on functions that were not executed at all. However, exporting the coverage data using these options requires access to application source files and take much longer to run.

Dynamic Call Graphs

Using the `-txtdcg` option the tool can provide detailed information about the dynamic call graphs in an application. Specify an output file for the dynamic call-graph report. The resulting call graph report contains information about the percentage of static and dynamic calls (direct, indirect, and virtual) at the application, module, and function levels.

Test Prioritization Tool

The test prioritization tool, also known as the tselect tool, enables the profile-guided optimizations on all supported Intel® architectures, on Linux*, Windows*, and macOS operating systems, to select and prioritize tests for an application based on prior execution profiles.

The tool offers a potential of significant time saving in testing and developing large-scale applications where testing is the major bottleneck.

Development often requires changing applications modules. As applications change, developers can have a difficult time retaining the quality of their functional and performance tests so they are current and on-target. The test prioritization tool lets software developers select and prioritize application tests as application profiles change.

The information about the tool is separated into the following sections:

- Features and benefits
- Requirements and syntax
- Usage model
- Tool options
- Running the tool

Features and Benefits

The test prioritization tool provides an effective testing hierarchy based on the code coverage for an application. The following list summarizes the advantages of using the tool:

- Minimizing the number of tests that are required to achieve a given overall coverage for any subset of the application. The tool defines the smallest subset of the application tests that achieve exactly the same code coverage as the entire set of tests.
- Reducing the turn-around time of testing. Instead of spending a long time on finding a possibly large number of failures, the tool enables the users to quickly find a small number of tests that expose the defects associated with the regressions caused by a change set.
- Selecting and prioritizing the tests to achieve certain level of code coverage in a minimal time based on the data of the tests' execution time.

See Understanding Profile-guided Optimization and Profile an Application topics for general information on creating the files needed to run this tool.

Test Prioritization Tool Requirements

The test prioritization tool needs the following items to work:

- The .spi file generated by Intel® compilers when compiling the application for the instrumented binaries with the `-prof-gen=srcpos` (Linux* and macOS) or `/Qprof-gen:srcpos` (Windows*) option.
- The .dpi files generated by the profmerge tool as a result of merging the dynamic profile information .dyn files of each of the application tests. Run the profmerge tool on all .dyn files that are generated for each individual test and name the resulting .dpi in a fashion that uniquely identifies the test.

- User-generated file containing the list of tests to be prioritized. For successful instrumented code run, you should:

- Name each test .dpi file so the file names uniquely identify each test.
- Create a .dpi list file, which is a text file that contains the names of all .dpi test files.

  Each line of the .dpi list file should include one, and only one .dpi file name. The name can optionally be followed by the duration of the execution time for a corresponding test in the dd:hh:mm:ss format.

  For example: `Test1.dpi 00:00:60:35` states that Test1 lasted 0 days, 0 hours, 60 minutes and 35 seconds.

  The execution time is optional. However, if it is not provided, then the tool will not prioritize the test for minimizing execution time. It will prioritize to minimize the number of tests only.

> **Caution**
> The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code. The tool uses the following general syntax:

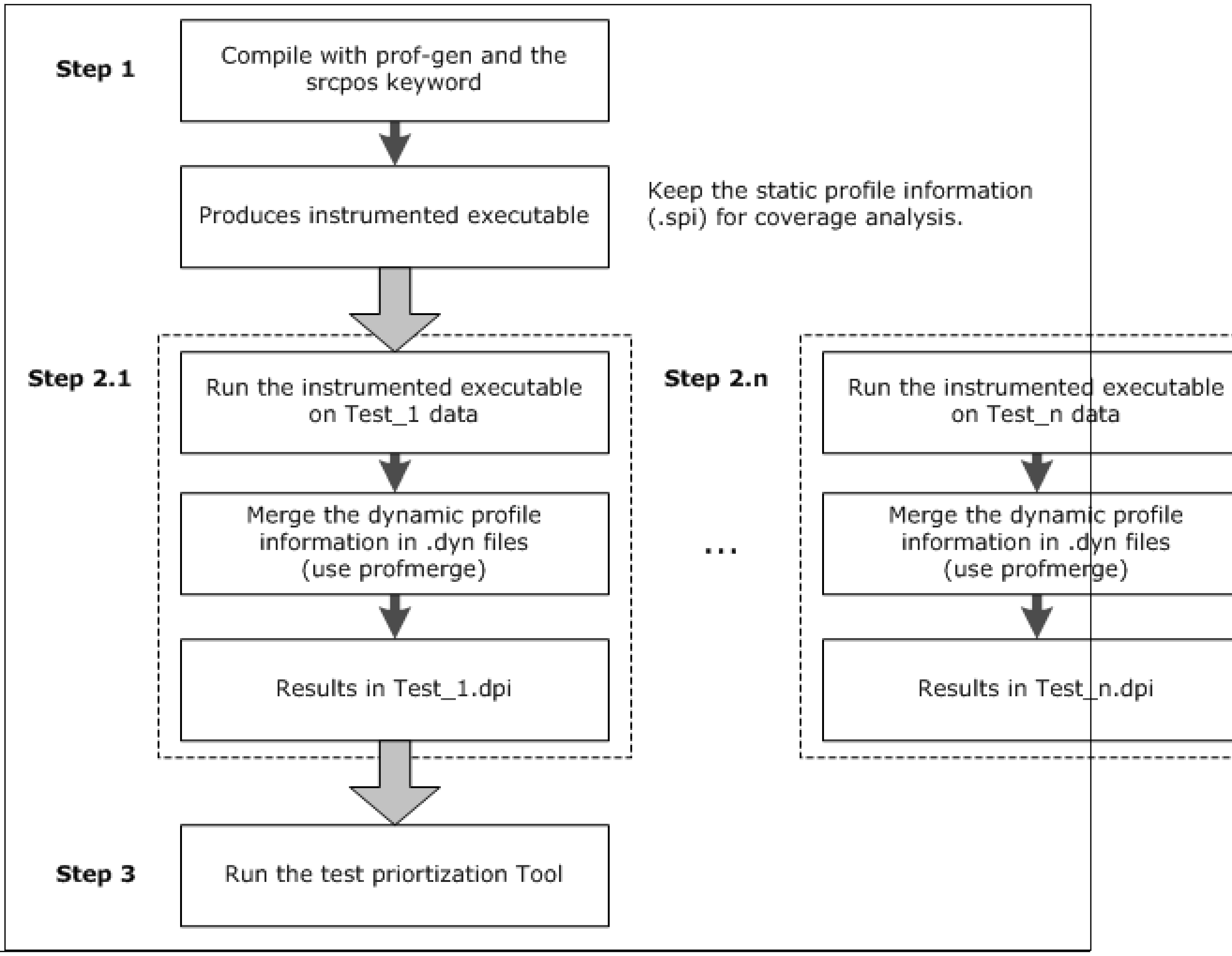
Tool Syntax
`tselect -dpi_list`*file*

`-dpi_list` is a required tool option that sets the path to the list file containing the list of the all .dpi files. All other tool commands are optional.

> **NOTE**
> Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

Usage Model

The following figure illustrates a typical test prioritization tool usage model.



Test Prioritization Tool Options

The tool uses the options that are listed in the following table:

Option	Description
`-help`	Prints tool option descriptions.
`-dpi_list`*file*	Required. Specifies the name of the file that contains the names of the dynamic profile information (`.dpi`) files. Each line of the file must contain only one .dpi file name, which can be followed by its execution time (optional). The name must uniquely identify the test.
`-spi`*file*	Specifies the file name of the static profile information file (.SPI). Default is `pgopti.spi`
`-o`*file*	Specifies the file name of the output report file.

Option	Description
`-comp`*file*	Specifies the file name that contains the list of files of interest.
`-cutoff`*value*	Instructs the tool to terminate when the cumulative block coverage reaches a preset percentage, as specified by `value`, of pre-computed total coverage. `value` must be greater than 0.0 (for example, 99.00) but not greater than 100. `value` can be set to 100.
`-nototal`	Instructs the tool to ignore the pre-compute total coverage process.
`-mintime`	Instructs the tool to minimize testing execution time. The execution time of each test must be provided on the same line of `dpi_list` file, after the test name in dd:hh:mm:ss format.
`-srcbasedir`*dir*	Specifies a different top-level project directory than was used during compiler instrumentation run with the `prof-src-root` compiler option to support relative paths to source files in place of absolute paths.
`-verbose`	Instructs the tool to generate more logging information about program progress.

Running the Tool

The following steps demonstrate one simple example for running the tool on IA-32 architectures.

1. Specify the directory by entering a command similar to the following:

   Example

   ```
   set PROF_DIR=c:\myApp\prof-dir
   ```
2. Compile the program and generate instrumented binary by issuing commands similar to the following:

Operating System	Command
Linux and macOS	`icpc -prof-gen=srcpos myApp.cpp`
Windows	`icl /Qprof-gen:srcpos myApp.cpp`

   The commands shown above compile the program and generate instrumented binary `myApp`, as well as the corresponding static profile information `pgopti.spi`.
3. Confirm that unrelated `.dyn` files are not present by issuing a command similar to the following:

   Example

   ```
   rm prof-dir \*.dyn
   ```
4. Run the instrumented files by issuing a command similar to the following:

Example

```
myApp < data1
```

The command runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by the `-prof-dir` step above.

5. Merge all `.dyn` file into a single file by issuing a command similar to the following:

Example

```
profmerge -prof_dpi Test1.dpi
```

The profmerge tool merges all the .dyn files into one file (`Test1.dpi`) that represents the total profile information of the application on Test1.

6. Confirm again there are no unrelated .dyn files present a second time by issuing a command similar to the following:

Example

```
rm prof-dir \*.dyn
```

7. Run the instrumented application, and generate one or more new dynamic profile information files that have an extension .dyn in the directory specified in the `prof-dir` step above by issuing a command similar to the following:

Example

```
myApp < data2
```

8. Merge all .dyn files into a single file by issuing a command similar to the following:

Example

```
profmerge -prof_dpi Test2.dpi
```

At this step, the profmerge tool merges all the .dyn files into one file (`Test2.dpi`) that represents the total profile information of the application on Test2.

9. Confirm that there are no unrelated .dyn files present for the final time by issuing a command similar to the following:

Example

```
rm prof-dir \*.dyn
```

10. Run the instrumented application and generate one or more new dynamic profile information files that have an extension .dyn in the directory specified by `-prof-dir` by issuing a command similar to the following:

Example

```
myApp < data3
```

11. Merge all .dyn file into a single file, by issuing a command similar to the following:

Example

```
profmerge -prof_dpi Test3.dpi
```

At this step, the profmerge tool merges all the .dyn files into one file (`Test3.dpi`) that represents the total profile information of the application on Test3.

12. Create a file named `tests_list` with three lines. The first line contains `Test1.dpi`, the second line contains `Test2.dpi`, and the third line contains `Test3.dpi`.

Tool Usage Examples

When these items are available, the test prioritization tool may be launched from the command line in the `prof-dir` directory as described in the following examples.

Example 1: Minimizing the Number of Tests

The following example describes how minimize the number of test runs.

Example Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi
```

where the `-spi` option specifies the path to the .spi file.

The following sample output shows typical results.

Sample Output

```
Total number of tests   =  3
Total block coverage    ~  52.17
Total function coverage ~  50.00
 num  %RatCvrg  %BlkCvrg  %FncCvrg  Test Name @ Options
 ---  --------  --------  --------  -------------------
   1   87.50     45.65     37.50    Test3.dpi
   2   100.50    52.17     50.00    Test2.dpi
```

In this example, the results provide the following information:

- By running all three tests, you achieve 52.17% block coverage and 50.00% function coverage.
- Test3 alonecovers 45.65% of the basic blocks of the application, which is 87.50% of the total block coverage that can be achieved from all three tests.
- By adding Test2, you achieve a cumulative block coverage of 52.17% or 100% of the total block coverage of Test1, Test2, and Test3.
- Elimination of Test1 has no negative impact on the total block coverage.

Example 2: Minimizing Execution Time

Assume you have the following execution time of each test in the `tests_list` file:

Sample Output

```
Test1.dpi 00:00:60:35
Test2.dpi 00:00:10:15
Test3.dpi 00:00:30:45
```

The following command minimizes the execution time by passing the `-mintime` option:

Sample Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi -mintime
```

The following sample output shows possible results:

Sample Output

```
Total number of tests   =  3
Total block coverage    ~  52.17
Total function coverage ~  50.00
Total execution time    =  1:41:35
 num  elaspedTime  %RatCvrg  %BlkCvrg  %FncCvrg  Test Name @ Options
 ---  -----------  --------  --------  --------  -------------------
   1   10:15         75.00     39.13     25.00    Test2.dpi
   2   41:00         100.00    52.17     50.00    Test3.dpi
```

In this case, the results indicate that running all tests sequentially would require one hour, 45 minutes, and 35 seconds, while the selected tests would achieve the same total block coverage in only 41 minutes.

The order of tests when based on minimizing time (first Test2, then Test3) may be different than when prioritization is done based on minimizing the number of tests. See Example 1 shown above: first Test3, then Test2. In Example 2, Test2 is the test that gives the highest coverage per execution time, so Test2 is picked as the first test to run.

Using Other Options

The `-cutoff` enables the tool to exit when it reaches a given level of basic block coverage. The following example demonstrates how to use the option:

Example

```
tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00
```

If the tool is run with the cutoff value of 85.00, as in the above example, only Test3 will be selected, as it achieves 45.65% block coverage, which corresponds to 87.50% of the total block coverage that is reached from all three tests.

The tool does an initial merging of all the profile information to figure out the total coverage that is obtained by running all the tests. The `-cutoff` enables you to skip this step. In such a case, only the absolute coverage information will be reported, as the overall coverage remains unknown.

Profmerge and Proforder Tools

Profmerge Tool

Use the profmerge tool to merge dynamic profile information (`.dyn`) files and any specified summary files (`.dpi`). The compiler executes profmerge automatically during the feedback compilation phase when you specify the `[Q]prof-use` option.

The command-line usage for profmerge is as follows:

Syntax

```
profmerge [-prof_dir dir_name]
```

The tool merges all `.dyn` files in the current directory, or the directory specified by `-prof_dir`, and produces a summary file: `pgopti.dpi`.

NOTE
The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example -prof_dir) instead of the hyphen used by compiler options (for example `[Q]prof-dir`) to join words. Also, on Windows* systems, the tool options are preceded by a hyphen ("-") unlike Windows* compiler options, which are preceded by a forward slash ("/").

You can use profmerge tool to merge .dyn files into a `.dpi` file without recompiling the application. You can run the instrumented executable file on multiple systems to generate `.dyn` files, and optionally use profmerge with the `-prof_dpi` option to name each summary `.dpi` file created from the multiple `.dyn` files.

Because the profmerge tool merges all the `.dyn` files that exist in the given directory, confirm that unrelated `.dyn` files are not present; otherwise, profile information will be based on invalid profile data, which can negatively impact the performance of optimized code.

Profmerge Options

The profmerge tool supports the following options:

Tool Option	Description
`-dump`	Displays profile information.
`-help`	Lists supported options.
`-nologo`	Disables version information. This option is supported on Windows* only.
`-exclude_files`*`files`*	Excludes functions from the profile if the function comes from one of the listed files. The list items must be separated by a comma (","); you can use a period (".") as a wild card character in function names.
`-exclude_funcs`*`functions`*	Excludes functions from the profile. The list items must be separated by a comma (","); you can use a period (".") as a wild card character in function names.
`-prof_dir`*`dir`*	Specifies the directory from which to read `.dyn` and `.dpi` files, and write the `.dpi` file. Alternatively, you can set the environment variable`PROF_DIR`.
`-prof_dpi`*`file`*	Specifies the name of the `.dpi` file being generated.
`-prof_file`*`file`*	Merges information from file matching: dpi_file_and_dyn_tag.
`-src_old`*`dir`*`-src_new`*`dir`*	Changes the directory path stored within the `.dpi` file.
`-no_src_dir`	Uses only the file name and not the directory name when reading dyn/dpi records. If you specify `-no_src_dir`, the directory name of the source file

Tool Option	Description
	will be ignored when deciding which profile data records correspond to a specific application routine, and the `-src-root` option is ignored.
`-src-root`*`dir`*	Specifies a directory path prefix for the root directory where the user's application files are stored. This option is ignored if you specify `-no_src_dir`.
`-a`*`file1.dpi`*`...`*`fileN.dpi`*	Specifies and merges available `.dpi` files.
`-verbose`	Instructs the tool to display full information during merge.
`-weighted`	Instructs the tool to apply an equal weighting (regardless of execution times) to the `.dyn` file values to normalize the data counts. This keyword is useful when the execution runs have different time durations and you want them to be treated equally.
`-gen_weight_spec` *file*	Instructs the tool to generate a text file containing a list of the `.dyn` and `.dpi` file that were merged with default weight=1/run_count. The text file is created in the directory specified by the `prof_dir` option.
`-weight_spec` *weight_spec.txt*	Instructs the profmerge tool to generate and use the text file, *weight_spec.txt,* listing individual `.dyn`/`.dpi` files or directory names along with weight values for them. When the `-weight_spec` option is used: • A new `.dpi` file is always created • Only files called out by the specification file are merged • `.dyn` timestamps are ignored and merge always takes place The `prof_dir` option controls where the input/ output *weight_spec.txt* is located, and the destination of the .dpi file. The `-weight_spec` option overrides: • Any values of `-a` option • Any computation from using `-weighted` option

Weighting the Runs

Using the `-weight_spec` option results in a new `.dpi` file. Only the files listed in the text file are merged. No files in the current directory are used unless they are included in the text file.

Relocating source files using profmerge

The Intel® C++ Compiler uses the full path to the source file for each routine to look up the profile summary information associated with that routine. By default, this prevents you from:

- Using the profile summary file (`.dpi`) if you move your application sources.
- Sharing the profile summary file with another user who is building identical application sources that are located in a different directory.

You can disable the use of directory names when reading `.dyn`/`.dpi` file records by specifying the `profmerge` option `-no_scr_dir`. This `profmerge` option is the same as the compiler option `-no-prof-src-dir` (Linux* and macOS) and `/Qprof-src-dir-` (Windows*).

To enable the movement of application sources, as well as the sharing of profile summary files, you can use the `profmerge` option `-src-root` to specify a directory path prefix for the root directory where the application files are stored. Alternatively, you can specify the option pair `-src_old-src_new` to modify the data in an existing summary dpi file. For example:

Example: relocation command syntax

```
profmerge -prof_dir <dir1> -src_old <dir2> -src_new <dir3>
```

where *<dir1>* is the full path to dynamic information file (.dpi), *<dir2>* is the old full path to source files, and *<dir3>* is the new full path to source files. The example command (above) reads the `pgopti.dpi` file, in the location specified in *<dir1>*. For each function represented in the `pgopti.dpi` file, whose source path begins with the *<dir2>* prefix, profmerge replaces that prefix with *<dir3>*. The `pgopti.dpi` file is updated with the new source path information.

You can run profmerge more than once on a given `pgopti.dpi` file. For example, you may need to do this if the source files are located in multiple directories:

Operating System	Command Examples
Linux* and macOS	`profmerge -prof_dir -src_old /src/prog_1 -src_new /src/prog_2` `profmerge -prof_dir -src_old /proj_1 -src_new /proj_2`
Windows*	`profmerge -src_old "c:/program files" -src_new "e:/program files"` `profmerge -src_old c:/proj/application -src_new d:/app`

In the values specified for `-src_old` and `-src_new`, uppercase and lowercase characters are treated as identical in Windows. Likewise, forward slash (`/`) and backward slash (`\`) characters are treated as identical.

NOTE
Because the source relocation feature of profmerge modifies the `pgopti.dpi` file, consider making a backup copy of the file before performing the source relocation.

Proforder Tool

The proforder tool is used as part of the feedback compilation phase, to improve program performance. Use proforder to generate a function order list for use with the `/ORDER` linker option in Windows. The tool uses the following syntax:

Syntax

```
proforder [-prof_dir dir] [-o file]
```

where *dir* is the directory containing the profile files (`.dpi` and `.spi`), and *file* is the optional name of the function order list file. The default name is `proford.txt` .

NOTE
The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example `-prof_dir`) instead of the hyphen used by compiler options (for example `[Q]prof-dir`) to join words. Also, on Windows* systems, the tool options are preceded by a hyphen ("-") unlike Windows* compiler options, which are preceded by a forward slash ("/").

Proforder Options

The proforder tool supports the following options:

Tool Option	Default	Description
`-help`		Lists supported options.
`-nologo`		Disables version information. This option is supported on Windows* only.
`-omit_static`		Instructs the tool to omit static functions from function ordering.
`-prof_dir`*dir*		Specifies the directory where the `.spi` and `.dpi` file reside.
`-prof_dpi`*file*		Specifies the name of the `.dpi` file.
`-prof_file`*string*		Selects the `.dpi` and `.spi` files that include the substring value in the file name matching the values passed as string.
`-prof_spi`*file*		Specifies the name of the `.spi` file.
`-o`*file*	`proford.txt`	Specifies an alternate name for the output file.

See Also

Supported Environment Variables

Using Function Order Lists, Function Grouping, Function Ordering, and Data Ordering Optimizations

Instead of doing a full multi-file interprocedural build of your application by using the compiler option `[Q]ipo`, you can obtain some of the benefits by having the compiler and linker work together to make global decisions about where to place the functions and data in your application. These optimizations are not supported on macOS systems.

The following table lists each optimization, the type of functions or global data it applies to, and the operating systems and architectures that it is supported on.

Optimization	Type of Function or Data	Supported OS and Architectures
Function Order Lists: Specifies the order in which the linker should link the non-static routines (functions) of your program. This optimization can improve application performance	`extern`functions procedures and library functions only (not static functions).	Windows: all Intel architectures Linux: not supported

Optimization	Type of Function or Data	Supported OS and Architectures
by improving code locality and reduce paging. Also see Comparison of Function Order Lists and IPO Code Layout. **Function Grouping**: Specifies that the linker should place the extern and static routines (functions) of your program into hot or cold program sections. This optimization can improve application performance by improving code locality and reduce paging. **NOTE** This option will cause functions to be placed into the linker sections named ".text.hot" and ".text.unlikely." If you are using a custom linker script, you will need to specify memory placement for these sections.	`extern`functions and static functions only (not library functions).	Linux: IA-32 and Intel 64 architectures Windows: not supported
Function Ordering: Enables ordering of static and extern routines using profile information. Specifies the order in which the linker should link the routines (functions) of your program. This optimization can improve application performance by improving code locality and reduce paging.	`extern`functions and static functions only (not library functions)	Linux and Windows: all Intel architectures
Data Ordering: Enables ordering of static global data items based on profiling information. Specifies the order in which the linker should link global data of your program. This optimization can improve application performance by improving the locality of static global data, reduce paging of large data sets, and improve data cache use.	Static global data only	Linux and Windows: all Intel architectures

You can only use one of the function-related ordering optimizations listed above on each application. However, you can use the Data Ordering optimization with any one of the function-related ordering optimizations listed above, such as Data Ordering with Function Ordering, or Data Ordering with Function Grouping. In this case, specify the `prof-gen` option keyword `globdata` (needed for Data Ordering) instead of `srcpos` (needed for function-related ordering).

The following sections show the commands needed to implement each of these optimizations: function order list, function grouping, function ordering, and data ordering. For all of these optimizations, omit the `[Q]ipo` or equivalent compiler option.

Generating a Function Order List (Windows)

This section provides an example of the process for generating a function order list. Assume you have a C++ program that consists of the following files: `file1.cpp` and `file2.cpp`. Additionally, assume you have created a directory for the profile data files called `c:\profdata`. You would enter commands similar to the following to generate and use a function order list for your Windows application.

1. Compile your program using the `/Qprof-gen:srcpos` option. Use the `/Qprof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

Example commands

```
icl /Femyprog /Qprof-gen=srcpos /Qprof-dir c:\profdata file1.cpp file2.cpp
```

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Example commands

```
myprog.exe
```

3. Before this step, copy all .dyn and .dpi files into the same directory. Merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file. Use the `/prof_dir` option to specify the directory location of the .dyn files.

Example commands

```
profmerge /prof_dir c:\profdata
```

4. Generate the function order list using the proforder tool. By default, the function order list is produced in the file `proford.txt`.

Example commands

```
proforder /prof_dir c:\profdata /o myprog.txt
```

5. Compile the application with the generated profile feedback by specifying the `ORDER` option to the linker. Use the `/Qprof-dir` option to specify the directory location of the profile files.

Example commands

```
icl /Femyprog /Qprof-use /Qprof-dir c:\profdata file1.cpp file2.cpp /link -
ORDER:@myprog.txt
```

Using Function Grouping (Linux)

This section provides a general example of the process for using the function grouping optimization. Assume you have a C++ program that consists of the following files: `file1.cpp` and `file2.cpp`. Additionally, assume you have created a directory for the profile data files called `profdata`. You would enter commands similar to the following to use a function grouping for your Linux application.

1. Compile your program using the `-prof-gen` option. Use the `-prof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

Example commands
`icc -o myprog -prof-gen -prof-dir ./profdata file1.cpp file2.cpp`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Example commands
`./myprog`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tools to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `-prof-func-group` option to request the function grouping as well as the `-prof-use` option to request feedback compilation. Again, use the `-prof-dir` option to specify the location of the profile files.

Example commands
`icl /Femyprog file1.cpp file2.cpp -prof-func-group -prof-use -prof-dir ./profdata`

NOTE On Linux, the –prof-func-group option is on by default when –prof-use is selected.

Finer grain control over the number of functions placed into the hot region can be controlled with the `-prof-hotness-threshold` compiler option, see the command line reference for more details.

Using Function Ordering

This section provides an example of the process for using the function ordering optimization. Assume you have a C++ program that consists of the following files: `file1.cpp` and `file2.cpp`, and that you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to generate and use function ordering for your application.

1. Compile your program using the `-prof-gen=srcpos` (Linux) or `/Qprof-gen:srcpos` (Windows) option. Use the `[Q]prof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`icc -o myprog -prof-gen=srcpos -prof-dir ./profdata file1.cpp file2.cpp`
Windows	`icl /Femyprog /Qprof-gen:srcpos /Qprof-dir c:\profdata file1.cpp file2.cpp`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tools to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `[Q]prof-func-order` option to request the function ordering, as well as the `[Q]prof-use` option to request feedback compilation. Again, use the `[Q]prof-dir` option to specify the location of the profile files.

Operating System	Example commands
Linux	`icpc -o myprog -prof-dir ./profdata file1.cpp file2.cpp -prof-func-order-prof-use`
Windows	`icl /Femyprog /Qprof-dir c:\profdata file1.cpp file2.cpp /Qprof-func-order /Qprof-use`

Using Data Ordering

This section provides an example of the process for using the data order optimization. Assume you have a C ++ program that consists of the following files: `file1.cpp` and `file2.cpp`, and that you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to use data ordering for your application.

1. Compile your program using the `-prof-gen=globdata` (Linux) or `/Qprof-gen:globdata` (Windows) option. Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`icc -o myprog -prof-gen=globdata -prof-dir ./profdata file1.cpp file2.cpp`
Windows	`icl /Femyprog /Qprof-gen=globdata /Qprof-dir c:\profdata file1.cpp file2.cpp`

2. Run the instrumented program with one or more sets of input data. If you specified a location other than the current directory, change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tools to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `[Q]prof-data-order` option to request the data ordering as well as the `[Q]prof-use` option to request feedback compilation. Again, use the `[Q]prof-dir` option to specify the location of the profile files.

Operating System	Example commands
Linux	`icpc -o myprog -prof-dir ./profdata file1.cpp file2.cpp -prof-data-order-prof-use`

Operating System	Example commands
Windows	`icl /Femyprog /Qprof-dir c:\profdata file1.cpp file2.cpp /Qprof-data-order/Qprof-use`

Comparison of Function Order Lists and IPO Code Layout

The Intel® compiler provides two methods of optimizing the layout of functions in the executable:

- Using a function order list
- Using the `/Qipo` (Windows) compiler option

Each method has advantages. A function order list, created with proforder, lets you optimize the layout of non-static functions (external and library functions whose names are exposed to the linker).

The linker cannot directly affect the layout order for static functions because the names of these functions are not available in the object files.

The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Alternately, using the `/Qipo` (Windows) option allows you to optimize the layout of all `static` or `extern` functions compiled with the Intel® C++ Compiler. The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Function Order List Effects

Function Type	IPO Code Layout	Function Ordering with proforder
Static	X	No effect
Extern	X	X
Library	No effect	X

Function Order List Usage Guidelines (Windows*)

Use the following guidelines to create a function order list:

- The order list only affects the order of non-static functions.
- You must compile with `/Gy` to enable function-level linking. (This option is active if you specify either option `/O1` or `/O2`.)

Compiler Option Mapping Tool

The Intel compiler's Option Mapping Tool provides an easy method to derive equivalent options between Windows* and Linux*. If you are a Windows-based application developer who is developing an application for Linux, you may want to know, for example, the Linux equivalent for the `/Oy-` option. Likewise, the Option Mapping Tool provides Windows equivalents for Intel compiler options supported on Linux.

NOTE
The Option Mapping Tool is not supported on macOS.

Using the Compiler Option Mapping Tool

You can start the Option Mapping Tool from the command line by:

- invoking the compiler and using the `[Q]map-opts` option
- executing the tool directly

> **NOTE**
> Compiler options are mapped to their equivalent on the architecture you are using.

Calling the Option Mapping Tool with the Compiler

If you use the compiler to execute the Option Mapping Tool, the following syntax applies:

```
<compiler command> <map-opts option> <compiler option(s)>
```

Example: Finding the Linux equivalent for `/Oy-`

```
icl /Qmap-opts /Oy-
Intel(R) Compiler option mapping tool
mapping Windows options to Linux for C++
'-Qmap-opts' Windows option maps to
  --> '-map-opts' option on Linux
  --> '-map_opts' option on Linux
'-Oy-' Windows option maps to
  --> '-fomit-frame-pointer-' option on Linux
  --> '-fno-omit-frame-pointer' option on Linux
  --> '-fp' option on Linux
```

Example: Finding the Windows equivalent for `-fp`

```
icpc -map-opts -fp
Intel(R) Compiler option mapping tool
mapping Linux options to Windows for C++
'-map-opts' Linux option maps to
  --> '-Qmap-opts' option on Windows
  --> '-Qmap_opts' option on Windows
'-fp' Linux option maps to
  --> '-Oy-' option on Windows
```

Output from the Option Mapping Tool also includes:

- option mapping information (not shown here) for options included in the compiler configuration file
- alternate forms of the option that are supported but may not be documented

When you call the Option Mapping Tool with the compiler, your source file is not compiled.

Calling the Option Mapping Tool Directly

Use the following syntax to execute the Option Mapping Tool directly from a command line environment where the full path to the `map_opts` executable is known (compiler `bin` directory):

```
map_opts [-nologo] -t<target OS> -l<language> -opts <compiler option(s)>
```

where values for:

- `<target OS> = {l|linux|w|windows}`
- `<language> = {f|fortran|c}`

Example: Finding the Linux equivalent for `/Oy-`

```
map_opts -tl -lc -opts /Oy-
Intel(R) Compiler option mapping tool
mapping Windows options to Linux for C++
'-Oy-' Windows option maps to
```

```
  --> '-fomit-frame-pointer-' option on Linux
  --> '-fno-omit-frame-pointer' option on Linux
  --> '-fp' option on Linux
```

Example: Finding the Windows equivalent for `-fp`

```
map_opts -tw -lc -opts -fp
Intel(R) Compiler option mapping tool
mapping Linux options to Windows for C++
'-fp' Linux option maps to
  --> '-Oy-' option on Windows
```

Part VI

Compatibility and Portability

This section contains information about conformance to language standards, language compatibility, and portability.

Conformance to the C/C++ Standards

The Intel® C++ Compiler conforms to the following ANSI/ISO standards:

- **C++** ISO/IEC 14882:1998
- **C** ISO/IEC 9899:1990

provides conformance to the ANSI/ISO standard for C language compilation (ISO/IEC 9899:1990). This standard requires that conforming C compilers accept minimum translation limits. This compiler exceeds all of the ANSI/ISO requirements for minimum translation limits.

C++ Support

The Intel® C++ Compiler supports many features in C++11. For a list of support features, see *C++ Features Supported by Intel® C++ compiler* at https://software.intel.com/content/www/us/en/develop/articles/c0x-features-supported-by-intel-c-compiler.html.

Template Instatiation

The Intel® C++ Compiler supports `extern` template, which lets you specify that a template in a specific translation unit will not be instantiated because it will be instantiated in a different translation unit or different library. The compiler now includes additional support for:

- **inline template** – instantiates the compiler support data for the class (example: *vtable*) for a class without instantiating its members.
- **static template** – instantiates the static data members of the template, but not the virtual tables or member functions.

You can now use the following options to gain more control over the point of template instantiation:

Option	Description
`-fno-implicit-templates`	Never emit code for non-inline templates which are instantiated implicitly (i.e. by use). only emit code for explicit instantiations.
`-fno-implicit-inline-templates`	Do not emit code for implicit instantiations of inline templates either. The default is to handle inlines differently so that compilations, with and without optimization, will need the same set of explicit instantiations.

C99 Support

The following C99 features are supported in this version of the Intel® C++ Compiler:

- restricted pointers (`restrict` keyword).
- variable-length Arrays
- flexible array members
- complex number support (`_Complex` keyword)
- hexadecimal floating-point constants
- compound literals
- designated initializers
- mixed declarations and code
- macros with a variable number of arguments
- inline functions (`inline` keyword)
- boolean type (`_Bool` keyword)

These `long double` (128-bit representations) feature is not supported:

See Also

`-fno-implicit-templates` compiler option

`-fno-implicit-inline-templates` compiler option

GCC Compatibility and Interoperability

GCC Compatibility

The Intel® C++ Compiler Classic is compatible with most versions of the GNU Compiler Collection (GCC). The release notes contains a list of compatible versions.

C language object files created with the compiler are binary compatible with the GCC and C/C++ language library. You can use the Intel® C++ Compiler Classic or the GCC compiler to pass object files to the linker.To pass IPO mock object files or libraries of IPO mock object files produced by theIntel® C++ Compiler Classic to the linker, use the linking tools provided with the compiler. Specifically:

Use `icc`, `icpc`, `xild`, and `xiar`.

NOTE When using an Intel software development product that includes a compiler with a Clang front-end, you can also use `icl`.

Link-time optimization using the `-ffat-lto-objects` compiler option is provided for GCC compatibility. This implies that `ld` and `ar` can be used to link and archive object files, but by doing so you will lose cross-file optimizations. You can use the `-fno-fat-lto-objects` compiler option when linking using IPO mock object files, provided that you link the IPO mock object files with `xild` and archive them with `xiar`.

The Intel® C++ Compiler Classic supports many of the language extensions provided by the GNU compilers.

Statement expressions are supported, except that the following are prohibited inside them:

- Dynamically-initialized local static variables
- Local non-POD class definitions
- Try/catch
- Variable length arrays

Branching out of a statement expression and statement expressions in constructor initializers are not allowed. Variable-length arrays are no longer allowed in statement expressions.

The Intel® C++ Compiler Classic supports GCC-style inline ASM if the assembler code uses AT&T* System V/386 syntax.

GCC Interoperability

C++ compilers are interoperable if they can link object files and libraries generated by one compiler with object files and libraries generated by the second compiler, and the resulting executable runs successfully. The Intel® C++ Compiler Classic is highly compatible with the GNU compilers.

The Intel® C++ Compiler Classic and GCC support the following predefined macros:

- `__GNUC__`
- `__GNUG__`
- `__GNUC_MINOR__`
- `__GNUC_PATCHLEVEL__`

> **Caution** Not defining these macros results in different paths through system header files. These alternate paths may be poorly tested or otherwise incompatible.

How the Compiler Uses GCC

The Intel® C++ Compiler Classic uses the GNU tools on the system, such as the GNU header files, including `stdio.h`, and the GNU linker and libraries. So the compiler has to be compatible with the version of GCC or G++* you have on your system. For example, if you have GCC version 4.6 on your system, `icc` behaves like GCC 4.6, with the compatible features and behaviors.

By default, the compiler determines which version of GCC or G++ you have installed from the `PATH` environment variable.

If you want use a version of GCC or G++ other than the default version on your system, you need to use the `--gcc-toolchain` compiler option to specify the location of the base toolchain. For example:

- You want to build something that cannot be compiled by the default version of the system compiler, so you need to use a legacy version for compatibility, such as if you want to use third party libraries that are not compatible with the default version of the system compiler.
- You want to use a later version of GCC or G++ than the default system compiler.

The Intel® C++ Compiler Classic driver uses the default version of GCC/G++, or the version you specify, to extract the location of the headers and libraries.

Compatibility with Open Source Tools

The Intel® C++ Compiler Classic includes improved support for the following open source tools:

- **GNU Libtool**: A script that allows package developers to provide generic shared library support.
- **Valgrind**: A flexible system for debugging and profiling executables running on x86 processors.
- **GNU Automake**: A tool for automatically generating `Makefile.ins` from files called `Makefile.am`.

See Also

ffat-lto-objects

Microsoft Compatibility

The Intel® C++ Compiler Classic is fully source- and binary-compatible (native code only) with Microsoft Visual C++ (MSVC). You can debug binaries built with the Intel® C++ Compiler Classic from within the Microsoft Visual Studio environment.

The compiler supports security checks with the /GS option. You can control this option in the Microsoft Visual Studio IDE by using **C/C++** > **Code Generation** > **Buffer Security Check**.

The compiler also includes support for safe exception handling features with the /Qsafeseh option for 32-bit binaries. This option is on by default. You can control this option in the Microsoft Visual Studio IDE by using **C/C++** > **Command Line** > **Additional options**.

> **Important**
> The compiler is a *hosted* compiler, not a *standalone* compiler. The compiler requires that standard development tools for the host system (linker, librarian, and so forth), and standard libraries and headers, are installed and available in your Path, Library Path, and Include environment variables. The host compiler provides access to I/O facilities through, for example, `<stdio.h>` and the C runtime library, as well as providing the implementation for the C++ standard template (for example, `<vector>`). When you build your application with the compiler, the `stdio.h` file is found in the host compiler's library. Likewise when you link your application, the link step uses the host OS linker to bind the application, and the host C runtime library provides the implementation for the runtime support routines.
>
> On Windows, the standard compiler is Microsoft Visual C++. On Linux, the standard compiler is GCC. The standard compiler must be installed and available in your environment before you run the compiler.

Microsoft Visual Studio Integration

The compiler is compatible with Microsoft Visual Studio 2017, 2019, and 2022 projects.

> **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

Unsupported Features

Unsupported project types:

- .NET-based CLR C++ project types are not supported by the Intel® C++ Compiler Classic. The specific project types will vary depending on your version of Visual Studio, for example:
  - CLR Class Library
  - CLR Console App
  - CLR Empty Project

Unsupported major features:

- COM Attributes
- C++ Accelerated Massive Parallelism (C++ AMP)
- Managed extensions for C++ (new pragmas, keywords, and command-line options)
- Event handling (new keywords)
- Select keywords:
  - `__abstract`

- `__box`
- `__delegate`
- `__gc`
- `__identifier`
- `__nogc`
- `__pin`
- `__property`
- `__sealed`
- `__try_cast`
- `__w64`

Unsupported preprocessor features:

- `#import` directive changes for attributed code
- `#using` directive
- `managed`, `unmanaged` pragmas
- `_MANAGED` macro
- `runtime_checks` pragma

Mixing Managed and Unmanaged Code

If you use the managed extensions to the C++ language in Microsoft Visual Studio .NET, you can use the compiler for your non-managed code for better application performance. Make sure managed keywords do not appear in your non-managed code.

For information on how to mix managed and unmanaged code, refer to the article, An Overview of Managed/Unmanaged Code Interoperability, on the Microsoft Web site.

Precompiled Header Support

There are some differences in how precompiled header (PCH) files are supported between the Intel® C++ Compiler Classic and the Microsoft Visual C++ Compiler:

- The PCH information generated by the Intel® C++ Compiler Classic is not compatible with the PCH information generated by the Microsoft Visual Studio Compiler.
- The Intel® C++ Compiler Classic does not support PCH generation and use in the same translation unit.
- The Intel® C++ Compiler does not generate PCH information beyond a point where a declaration is seen in the primary translation unit. When the `/Yu` option is specified, the Microsoft Visual C++ compiler ignores all text, including declarations preceding the `#include` statement of the specified file.
- The Microsoft Visual C++ Compiler will not emit an error if a function or variable definition occurs in a PCH file, which is included in two different source files and is not referenced. The Intel® C++ Compiler will always give a multiple definition link error under these circumstances.

Compilation and Execution Differences

While the Intel® C++ Compiler Classic is compatible with the Microsoft Visual C++ Compiler, some differences can prevent successful compilation. There can also be some incompatible generated-code behavior of some source files with the Intel® C++ Compiler Classic. In most cases, a modification of the user source file enables successful compilation with both the Intel® C++ Compiler Classic and the Microsoft Visual C++ Compiler. The differences between the compilers are:

- **Evaluation of Left Shift Operations**

  The Intel® C++ Compiler differs from the Microsoft Visual C++ Compiler in the evaluation of left shift operations where the right operand, or shift count, is equal to or greater than the size of the left operand expressed in bits. The ANSI C standard states that the behavior of such left-shift operations is undefined, meaning a program should not expect a certain behavior from these operations. This difference is only evident when both operands of the shift operation are constants. The following example illustrates this difference between the Intel® C++ Compiler and the Microsoft Visual C++ Compiler:

```
int x;
int y = 1; //set y=1
void func() {
  x = 1 << 32;
  // Intel C++ Compiler generates code to set x=1
  // Visual C++ Compiler generates code to set x=0

  y = y << 32;
  // Intel C++ Compiler generates code to set y=1
  // Visual C++ Compiler generates code to set y=1
}
```

- **Inline Assembly Target Labels (IA-32 Architecture Only)**

  For compilations targeted for IA-32 architecture, inline assembly target labels of `goto` statements are case sensitive. The Microsoft Visual C++ compiler treats these labels in a case insensitive manner. For example, the Intel® C++ Compiler Classic issues an error when compiling the following code:

```
int func(int x) {
   goto LAB2;
     // error: label "LAB2" was referenced but not defined
   __asm lab2: mov x, 1
   return x;
}
```

  However, the Microsoft Visual C++ Compiler accepts the preceding code. As a work-around for the Intel® C++ Compiler Classic, when a `goto` statement refers to a label defined in inline assembly, you must match the label reference with the label definition in both name and case.

- **Inlining Functions Marked for dllimport**

  The Intel® C++ Compiler Classic will attempt to inline any functions that are marked dllimport but Microsoft will not. Therefore, any calls or variables used inside a dllimport routine need to be available at link time or the result will be an unresolved symbol.

  The following example contains two files: `header.h` and `bug.cpp`.

  header.h:

```
#ifndef _HEADER_H
#define _HEADER_H
namespace Foo_NS {

        class Foo2 {
        public:
                Foo2(){};
                ~Foo2();
                static int test(int m_i);
        };
}
#endif
```

  bug.cpp:

```
#include "header.h"
struct Foo2 {
  static void test();
};
```

```
struct __declspec(dllimport) Foo
{
   void getI() { Foo2::test(); };
};

struct C  {
  virtual void test();
};

void C::test() { Foo p;  p->getI(); }

int main() {
   return 0;
}
```

Declaration in Scope of Function Defined in a Namespace

In accordance with the C++ language specification, if a function declaration is encountered within a function definition, the function referenced is taken to be another member of the namespace of the containing function. This is regardless of whether the containing function definition is lexically within a namespace definition. The Microsoft Visual C++ compiler takes the referenced function to be a global function (not in any namespace).

Functions declared in global or namespace scopes are interpreted the same way by both the Intel® C++ Compiler and the Microsoft Visual C++ compiler.

Enum Bit-Field Signedness

The Intel® C++ Compiler Classic and Microsoft Visual C++ differ in how they attribute signedness to bit fields declared with an `enum` type. Microsoft Visual C++ always considers `enum` bit fields to be signed, even if not all values of the `enum` type can be represented by the bit field.

The Intel® C++ Compiler Classic considers an `enum` bit field to be unsigned, unless the `enum` type has at least one `enum` constant with a negative value. In any case, the Intel® C++ Compiler Classic produces a warning if the bit field is declared with too few bits to represent all the values of the `enum` type.

See Also

/GS compiler option
/Qsafeseh

Port from Microsoft Visual C++ to the Intel® C++ Compiler Classic*

This section describes a basic approach to porting applications from Microsoft Visual C++* for Windows* to the Intel® C++ Compiler Classic for Windows.

If you build your applications from the Windows command line, you can port applications from Microsoft Visual C++ to the Intel® C++ Compiler Classic by modifying your makefile to invoke the Intel® C++ Compiler Classic instead of Microsoft Visual C++.

The Intel® C++ Compiler Classic integration with Microsoft Visual Studio provides a conversion path to the Intel® C++ Compiler Classic that allows you to build your Visual C++ projects with the Intel® C++ Compiler Classic. This version of the Intel® C++ Compiler Classic supports:

- Microsoft Visual Studio 2022

- Microsoft Visual Studio 2019
- Microsoft Visual Studio 2017

> **NOTE** Support for Microsoft Visual Studio 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

See the appropriate section in this documentation for details on using the Intel® C++ Compiler Classic with Microsoft Visual Studio.

The Intel® C++ Compiler Classic also supports many of the same compiler options, macros, and environment variables you already use in your Microsoft work.

One challenge in porting applications from one compiler to another is making sure there is support for the compiler options you use to build your application. The *Compiler Options* reference lists compiler options that are supported by both the Intel® C++ Compiler Classic and Microsoft C++.

See Also

Other Considerations

Modify Your Makefile

Modify Your makefile

If you use makefiles to build your Microsoft* application, you need to change the value for the compiler variable to use the Intel® C++ Compiler Classic. You may also want to review the options specified by `CPPFLAGS`. For example, a sample Microsoft makefile:

```
# name of the program
  PROGRAM = area.exe

# names of source files
  CPPSOURCES = area_main.cpp area_functions.cpp

# names of object files
  CPPOBJECTS = area_main.obj area_functions.obj

# Microsoft(R) compiler options
  CPPFLAGS = /RTC1 /EHsc

# Use Microsoft C++(R)
  CPP = cl

# link objects
  $(PROGRAM): $(CPPOBJECTS)
     link.exe /out:$@ $(CPPOBJECTS)

# build objects
  area_main.obj: area_main.cpp area_headers.h
  area_functions.obj: area_functions.cpp area_headers.h

# clean
  clean:  del  $(CPPOBJECTS) $(PROGRAM)
```

Modified makefile for the Intel® C++ Compiler Classic

Before you can run `nmake` with the Intel® C++ Compiler Classic, you need to set the proper environment. In this example, only the name of the compiler changed to use icc:

```
# name of the program
  PROGRAM = area.exe

# names of source files
  CPPSOURCES = area_main.cpp area_functions.cpp

# names of object files
  CPPOBJECTS = area_main.obj area_functions.obj

# Intel(R) C/C++ Compiler options
  CPPFLAGS = /RTC1 /EHsc

# Use the Intel C/C++ Compiler
  CPP = icc

# link objects
  $(PROGRAM): $(CPPOBJECTS)
     link.exe /out:$@ $(CPPOBJECTS)

# build objects
  area_main.obj: area_main.cpp area_headers.h
  area_functions.obj: area_functions.cpp area_headers.h

# clean
  clean:  del  $(CPPOBJECTS) $(PROGRAM)
```

With the modified makefile, the output of `nmake` is similar to the following:

```
Microsoft (R) Program Maintenance Utility Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

        icl /RTC1 /EHsc  /c area_main.cpp area_functions.cpp

Intel(R) Compiler for applications running on IA-32 or IA-64
Copyright (C) 1985-2006 Intel Corporation. All rights reserved.

area_main.cpp
area_functions.cpp
        link.exe /out:area.exe area_main.obj area_functions.obj

Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.
```

Use IPO in makefiles

By default, IPO generates dummy object files containing interprocedural information used by the compiler. To link or create static libraries with these object files requires specific Intel-provided tools. To use them in your makefile, replace references to `link` with `xilink` and references to `lib` with `xilib`. For example:

```
 # name of the program
  PROGRAM = area.exe

# names of source files
  CPPSOURCES = area_main.cpp area_functions.cpp
```

```
# names of object files
  CPPOBJECTS = area_main.obj area_functions.obj

# Intel C/C++ Compiler options
  CPPFLAGS = /RTC1 /EHsc /Qipo

# Use the Intel C/C++ Compiler
  CPP = icc

# link objects
  $(PROGRAM): $(CPPOBJECTS)
     xilink.exe /out:$@ $(CPPOBJECTS)

# build objects
  area_main.obj: area_main.cpp area_headers.h
  area_functions.obj: area_functions.cpp area_headers.h

# clean
  clean:  del  $(CPPOBJECTS) $(PROGRAM)
```

Other Considerations

There are some notable differences between the Intel® C++ Compiler Classic and the Microsoft* Compiler. Consider the following as you begin compiling your code with the Intel® C++ Compiler Classic.

Set the Environment

The compiler installation provides a batch file, `setvars.bat`, that sets the proper environment for the Intel® C++ Compiler Classic. For information on running `setvars.bat`, see Specifying the Location of Compiler Components.

Use Optimization

The Intel® C++ Compiler Classic is an optimizing compiler that begins with the assumption that you want improved performance from your application when it is executed on Intel® architecture. Consequently, certain optimizations, such as option `O2`, are part of the default invocation of the compiler. By default, Microsoft turns off optimization, which is the equivalent of compiling with options `Od` or `O0`. Other forms of the `O`[*n*] option compare as follows:

Option	Intel® C++ Compiler Classic	Microsoft Compiler
`/Od`	Turns off all optimization. Same as `O0`.	Default. Turns off all optimization.
`/O1`	Decreases code size with some increase in speed.	Optimizes code for minimum size.
`/O2`	Default. Favors speed optimization with some increase in code size. Intrinsics, loop unrolling, and inlining are performed.	Optimizes code for maximum speed.
`/O3`	Enables `-O2` optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations.	Not supported.

Target Specific Intel® Processors

While many of the same options that target specific processors are supported with both compilers, the Intel® C++ Compiler includes options that utilize processor-specific instructions to target the latest Intel® architecture processors. Consider using the Intel® C++ Compiler `[Q]x` , `/arch` , or `[Q]ax` options for applications that run on IA-32 architecture or Intel® 64 architecture. Refer to the descriptions of these compiler options for more specific information.

Modify Your Configuration

The Intel® C++ Compiler Classic lets you maintain configuration and response files that are part of compilation. Options stored in the configuration file apply to every compilation, while options stored in response files apply only where they are added on the command line. If you have several options in your makefile that apply to every build, you may find it easier to move these options to the configuration file (`..\bin\icc.cfg`).

In a multi-user, networked environment, options listed in the `icc.cfg` file are generally intended for everyone who uses the compiler. If you need a separate configuration, you can use the `ICCCFG` environment variable to specify the name and location of your own `.cfg` file, such as `\my_code\my_config.cfg`. Anytime you instruct the compiler to use a different configuration file, the `icc.cfg` system configuration file is ignored.

Use the Intel Libraries

The Intel® C++ Compiler Classic supplies additional libraries that contain optimized implementations of many commonly used functions. Some of these functions are implemented using CPU dispatch. This means that different code may be executed when run on different processors.

Supplied libraries include the Intel® C++ Compiler Classic (*libm*), the Short Vector Math Library (*svml_disp*), *libirc*, as well as others. These libraries are linked in by default when the compiler sees that references to them have been generated. Some library functions, such as `sin` or `memset`, may not require a call to the library, since the compiler may inline the code for the function.

Intel® C++ Compiler Classic Math Library (*libm*)

With the Intel® C++ Compiler Classic, the math library, *libm*, is linked by default when calling math functions that require the library. Some functions, such as `sin`, may not require a call to the library, since the compiler already knows how to compute the `sin` function. The math library also includes some functions not found in the standard math library.

> **NOTE**
> You cannot make calls to the math library with the Microsoft Compiler.

Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Short Vector Math Library (*svml_disp*)

When vectorization is in progress, the compiler may translate some calls to the *libm* math library functions into calls to *svml_disp* functions. These functions implement the same basic operations as the math library, but operate on short vectors of operands. This results in greater efficiency. In some cases, the *svml_disp* functions are slightly less precise than the equivalent *libm* functions.

Many routines in the Short Vector Math Library (SVML) are more optimized for Intel® microprocessors than for non-Intel microprocessors.

libirc

libirc contains optimized implementations of some commonly used string and memory functions. For example, it contains functions that are optimized versions of `memcpy` and `memset`. The compiler will automatically generate calls to these functions when it sees calls to `memcpy` and `memset`. The compiler may also transform loops that are equivalent to `memcpy` or `memset` into calls to these functions.

Many routines in the *libirc* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

O compiler option
Using Configuration Files
Using Response Files
Specifying the Location of Compiler Components

Port from GCC* to the Intel® C++ Compiler Classic

This section describes a basic approach to porting applications from the (GNU Compiler Collection*) GCC C/C++ compilers to the Intel® C++ Compiler Classic. These compilers correspond to each other as follows:

Language	Intel® Compiler	GCC Compiler
C	`icc`	`gcc`
C++	`icpc`	`g++`

NOTE Unless otherwise indicated, the term "`gcc`" refers to both GCC and G++* compilers from the GCC.

Advantages to Using the Intel® C++ Compiler Classic

In many cases, porting applications from `gcc` to the Intel® C++ Compiler Classic can be as easy as modifying your makefile to invoke the Intel® C++ Compiler Classic (`icc`) instead of `gcc`. Using the Intel® C++ Compiler Classic typically improves the performance of your application, especially for those that run on Intel processors. In many cases, your application's performance may also show improvement when running on non-Intel processors. When you compile your application with the Intel® C++ Compiler Classic, you have access to:

- Compiler options that optimize your code for the latest Intel® architecture processors.
- Advanced profiling tools (PGO) similar to the GNU profiler `gprof`.
- High-level optimizations (HLO).
- Interprocedural optimization (IPO).
- Intel intrinsic functions that the compiler uses to inline instructions, including various versions of Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions.

- Highly-optimized Intel® C++ Compiler Classic Math Library for improved accuracy.

Because the Intel® C++ Compiler Classic is compatible and interoperable with `gcc`, porting your `gcc` application to the Intel® C++ Compiler Classic includes the benefits of binary compatibility. As a result, you should not have to re-build libraries from your `gcc` applications. The Intel® C++ Compiler Classic also supports many of the same compiler options, macros, and environment variables you already use in your `gcc` work.

Equivalent Macros

The Intel® C++ Compiler Classic is compatible with the predefined GNU* macros.

See http://gcc.gnu.org for a list of compatible predefined macros.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

Modify Your makefile
Supported Environment Variables
Additional Predefined Macros

Modify Your makefile

If you use makefiles to build your GCC* application, you need to change the value for the GCC compiler variable to use the Intel® C++ Compiler Classic. You may also want to review the options specified by `CFLAGS`. For example, a sample GCC makefile:

```
# Use gcc compiler
  CC = gcc

# Compile-time flags
  CFLAGS = -O2 -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c -fno-asm $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

Modified makefile for the Intel® C++ Compiler Classic

In this example, the name of the compiler is changed to use icc

```
# Use Intel C/C++ Compiler
  CC = icc

# Compile-time flags
```

```
  CFLAGS = -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c -fno-asm $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

If your GCC code includes features that are not supported with the Intel® C++ Compiler Classic (compiler options, language extensions, macros, pragmas, and so on), you can compile those sources separately with GCC if necessary.

In the above makefile, `area_functions.c` is an example of a source file that includes features unique to GCC. Because the Intel® C++ Compiler Classic uses the `O2` option by default and GCC uses option `O0` as the default, we instruct GCC to compile at option `O2`. We also include the `-fno-asm` switch from the original makefile because this switch is not supported with the Intel® C++ Compiler Classic. The following sample makefile is modified for using the Intel® C++ Compiler Classic and GCC together:

```
# Use Intel C/C++ Compiler
  CC = icc
# Use gcc for files that cannot be compiled by icc
  GCC = gcc
# Compile-time flags
  CFLAGS = -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(GCC) -c -O2 -fno-asm $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

Output of `make` using a modified makefile:

```
icc -c -std=c99 area_main.c
gcc -c -O2 -fno-asm -std=c99 area_functions.c
icc area_main.o area_functions.o -o area
```

Use IPO in Makefiles

By default, IPO generates "dummy" object files containing Interprocedural information used by the compiler. To link or create static libraries with these object files requires special Intel-provided tools. To use them in your makefile, simply replace references to "ld" with "xild" and references to "ar" with "xiar", or use `icc` to link as shown in the example:

```
# Use Intel C/C++ Compiler
  CC = icc
# Compile-time flags
  CFLAGS = -std=c99 -ipo
```

```
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

Other Considerations

There are some notable differences between the Intel® C++ Compiler Classic and GCC*. Consider the following as you begin compiling your source code with the Intel® C++ Compiler Classic.

Set the Environment

The Intel® C++ Compiler Classic relies on environment variables for the location of compiler binaries, libraries, man pages, and license files. In some cases these are different from the environment variables that GCC uses. Another difference is that these variables are not set by default after installing the Intel® C++ Compiler Classic. The following environment variables need to be set prior to running the Intel® C++ Compiler Classic:

- PATH: Add the location of the compiler binaries to PATH.
- LD_LIBRARY_PATH: Sets the location where the generated executable picks up the runtime libraries (*.so files).
- MANPATH – add the location of the compiler man pages (`icc or icpc`) to MANPATH.
- INTEL_LICENSE_FILE – sets the location of the Intel® C++ Compiler license file.

To set these environment variables, run the `compilervars.sh` script.

NOTE
Setting these environment variables with `compilervars.sh` does not impose a conflict with GCC. You should be able to use both compilers in the same shell.

Use Optimization

The Intel® C++ Compiler Classic is an optimizing compiler that begins with the assumption that you want improved performance from your application when it is executed on Intel® architecture. Consequently, certain optimizations, such as option `O2`, are part of the default invocation of the Intel® C++ Compiler Classic. Optimization is turned off in GCC by default, the equivalent of compiling with option `O0`. Other forms of the `O`<*n*> option compare as follows:

Option	Intel® C++ Compiler Classic	GCC
`-O0`	Turns off optimization.	Default. Turns off optimization.
`-O1`	Decreases code size with some increase in speed.	Decreases code size with some increase in speed.

Option	Intel® C++ Compiler Classic	GCC
`-O2`	**Default.** Favors speed optimization with some increase in code size. Same as option `O`. Intrinsics, loop unrolling, and inlining are performed.	Optimizes for speed as long as there is not an increase in code size. Loop unrolling and function inlining, for example, are not performed.
`-O3`	Enables option `O2` optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations.	Optimizes for speed while generating larger code size. Includes option `O2` optimizations plus loop unrolling and inlining. Similar to option `O2 -ip` on the Intel® C++ Compiler.

Target Intel® Processors

While many of the same options that target specific processors are supported with both compilers, Intel includes options that utilize processor-specific instruction scheduling to target the latest Intel® processors. If you compile your GCC application with the `march` or `mtune` option, consider using the Intel® C++ Compiler options `x` or `ax` options for applications that run on IA-32 architecture or Intel® 64 architecture.

Modify Your Configuration

The Intel® C++ Compiler Classic lets you maintain configuration and response files that are part of compilation. Options stored in the configuration file apply to every compilation, while options stored in response files apply only where they are added on the command line. If you have several options in your makefile that apply to every build, you may find it easier to move these options to the configuration file (`icc.cfg or icpc.cfg`).

In a multi-user, networked environment, options listed in the `icc.cfg or icpc.cfg` files are generally intended for everyone who uses the compiler. If you need a separate configuration, you can use the `ICCCFG or ICPCCFG`environment variable to specify the name and location of your own `.cfg` file, such as `/my_code/my_config.cfg`. Anytime you instruct the compiler to use a different configuration file, the system configuration files (`icc.cfg or icpc.cfg`) are ignored.

Use the Intel Libraries

The Intel® C++ Compiler Classic supplies additional libraries that contain optimized implementations of many commonly used functions. Some of these functions are implemented using CPU dispatch. This means that different code may be executed when run on different processors.

Supplied libraries include the Intel® C++ Compiler Classic Math Library (*libimf*), the Short Vector Math Library (*libsvml*), *libirc*, as well as others. These libraries are linked in by default. Some library functions, such as `sin` or `memset`, may not require a call to the library, since the compiler may inline the code for the function.

NOTE The Intel Compiler Math Libraries contain performance-optimized implementations for various Intel platforms. By default, the best implementation for the underlying hardware is selected at runtime. The library dispatch of multi-threaded code may lead to apparent data races, which may be detected by certain software analysis tools. However, as long as the threads are running on cores with the same CPUID, these data races are harmless and are not a cause for concern.

Intel® C++ Compiler Classic Math Library (*libimf*)

With the Intel® Compiler, the math library, *libimf*, is linked by default. Some functions, such as sin, may not require a call to the library, since the compiler already knows how to compute the `sin` function. The math library also includes some functions not found in the standard math library.

NOTE
You cannot make calls to the math library with GCC.

Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Short Vector Math Library (*libsvml*)

When vectorization is being done, the compiler may translate some calls to the *libimf* math library functions into calls to *libsvml* functions. These functions implement the same basic operations as the math library, but operate on short vectors of operands. This results in greater efficiency. In some cases, the *libsvml* functions are slightly less precise than the equivalent *libimf* functions.

Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

libirc

libirc contains optimized implementations of some commonly used string and memory functions. For example, it contains functions that are optimized versions of `memcpy` and `memset`. The compiler will automatically generate calls to these functions when it sees calls to `memcpy` and `memset`. The compiler may also transform loops that are equivalent to `memcpy` or `memset` into calls to these functions.

Many routines in the *libirc* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

See Also

ax, Qax
Invoke the Compiler

march
mtune
`-O` compiler option
Using Configuration Files
Using Response Files
x, Qx

Index

A

D

E

M

N

O

P

R

S

T

U

V

W

X

Z