



Accelerating Nios II Systems with the C2H Compiler Tutorial

August 2008, Version 8.0

Tutorial

Introduction

The Nios® II C2H Compiler is a powerful tool that generates hardware accelerators for software functions. The C2H Compiler enhances design productivity by allowing you to use a compiler to accelerate software algorithms in hardware. You can quickly prototype hardware functional changes in C, and explore hardware-software design tradeoffs in an efficient, iterative process. The C2H Compiler is well suited to improving computational bandwidth as well as memory throughput. It is possible to achieve substantial performance gains with minimal engineering effort.

This tutorial teaches you how to use the C2H Compiler to accelerate a fast Fourier transform (FFT), yielding a large performance gain over a purely software based approach.

Table of Contents

Introduction	1
Prerequisites.....	2
Hardware & Software Requirements	2
Getting the Hardware and Software Files	2
FFT Background	3
Analyzing the FFT Code	4
Creating a Build Report	4
Performance Metrics.....	5
Unoptimized Accelerator	7
Optimizing the FFT	8
Adding On-Chip Buffers	8
Building the Accelerator.....	9
Optimized Performance Metrics.....	11
Downloading and Running the Accelerated System.....	11
Bottlenecks	12
Computational Bottlenecks.....	12
Memory Bottlenecks.....	12
Narrow Memory Access	13
Random Access to SDRAM	13
Multiple Master Port Memory Stalls	13
Restructuring Code to Optimize the Accelerator	14
Data Buffering Optimizations.....	14
Fast Memory.....	14
Double Buffering	14
Master Port Minimization.....	17
Sine and Cosine Data Buffering.....	17
Bit Reversal Buffering	17
SDRAM Memory Access Optimizations.....	17
Calculation Stage Optimizations.....	18
Conclusion	19

Prerequisites

To make effective use of this tutorial, you should be familiar with the following topics:

- ANSI C syntax and usage
- Defining and generating Nios II hardware systems with SOPC Builder
- Compiling Nios II hardware systems with the Altera® Quartus® II development software
- Creating, compiling, and running Nios II software projects
- Nios II C2H Compiler theory of operation

 To familiarize yourself with the basics of the C2H Compiler, refer to the *Nios II C2H Compiler User Guide*, especially chapters *Introduction to the C2H Compiler*, and *Getting Started Tutorial*. To learn about defining, generating and compiling Nios II systems, refer to the *Nios II Hardware Development Tutorial*. To learn about Nios II software projects, refer to the *Nios II Software Development Tutorial*, available in the Nios II IDE help system.

Hardware & Software Requirements

This tutorial requires you to have the following software and hardware:

- Quartus II development software version 7.2 or later, installed on a Windows or Linux computer.
- Nios II Embedded Design Suite (EDS) version 7.2 or later
- One of the following Nios II development boards:
 - Stratix® II Edition
 - Cyclone™ II Edition
- A JTAG download cable compatible with your target hardware, such as a USB-Blaster™ cable.

Getting the Hardware and Software Files

The tutorial software files are available on the Nios II literature page. A link to the software files appears next to *Accelerating Nios II Systems with the C2H Compiler Tutorial* (this document), at www.altera.com/literature/lit-nio2.jsp. The hardware and software files are distributed in a zip file.

Extract the design files included in the file **c2h_tutorial.zip** to a new directory on your host computer. Be sure to recreate the directory structure on your local file system (for example, turn on **Use folder names** in the WinZip application).

If you are targeting a Cyclone II development board, the files in the **c2h_fft_cyclone_ii** subdirectory are relevant to you. If you are targeting a Stratix II development board, the files in the **c2h_fft_stratix_ii** subdirectory are relevant. The remainder of this document refers to the relevant directory as *<tutorial install dir>*.

The *<tutorial install dir>* folder contains a Quartus II project and a software folder. The software folder contains two subdirectories: **c2h_fft** and **c2h_fft_syslib**. **c2h_fft** is the main software project and contains the following files:

- **sw_only_fft.c**, **sw_only_fft.h** — These files implement a 256-point radix-two FFT that can run on a Nios II processor without any hardware acceleration.

- **accelerator_optimized_fft.c, accelerator_optimized_fft.h** — These files implement a 256-point radix-two FFT that is functionally equivalent to the FFT defined in **sw_only_fft.c**. These files have been modified to run optimally in a hardware accelerator based system. For details about the optimizations, see the [Restructuring Code to Optimize the Accelerator](#) section.
- **pound_defines.h** — This file contains macros used by the FFT as well as pragma directives passed to the C2H Compiler.
- **the_top_file.c** — This file contains function `main()`. This is the top-level benchmark which runs both the software only and accelerated FFT functions and compares the performance of the two approaches.
- **testdata.dat, results.dat** — These files contain the input data to the FFT and the expected output results.
- **twiddles.dat** — This data file contains precalculated sine and cosine terms used in the software FFT calculation.

To import the software projects, perform the following steps:

1. Launch the Nios II IDE, and import the application and system library projects.
 - a. Click **Import** in the Nios II IDE File menu.
 - b. Select **Existing Altera Nios II Project into Workspace**, and click **Next**.
 - c. Browse to the `<tutorial install dir>/software` directory, select the **c2h_fft** folder, and click **OK**.
 - d. Click **Finish**.
 - e. Repeat the above steps for the **c2h_fft_syslib** project.



If the import dialog box does not automatically locate the SOPC Builder system file, browse to the `<tutorial install dir>` directory, select **FFT_system.ptf**, and click **OK**.

2. The C2H Compiler ignores the **Configuration** setting.

FFT Background

The fast Fourier transform (FFT) is a highly efficient method for calculating the discrete Fourier transform (DFT). The DFT is used in signal processing applications for a range of purposes, such as analyzing the frequency components of signals and data compression. The DFT is a computationally intensive function. A naïve (non-FFT) implementation of an n -point DFT requires n^2 complex multiplications.

The FFT algorithm achieves its efficiency gains by decomposing the DFT into a number of smaller DFTs and exploiting the symmetry and periodicity of the sub stages to reduce the number of calculations. An n -point FFT only requires $n \times \log_2 n$ complex multiplications. Cutting down the number of complex multiplications improves the FFT performance, often by several orders of magnitude, depending on the order of the transform.

A full description of the FFT algorithm is beyond the scope of this tutorial. Here are some basic facts about the FFT algorithm to be aware of:

- The FFT operates on complex data. It performs calculations simultaneously on real and imaginary components of the data. The algorithm implements complex multiplication as four multiplications, one addition and one subtraction.

- One of the fundamental operations in the FFT algorithm is the *butterfly calculation*. The butterfly calculation either breaks a larger DFT into smaller DFTs, or recombines smaller DFTs into a larger. The name *butterfly* comes from the shape of the dataflow diagram describing the operation.

 You can find more information at www.wikipedia.org, under "Butterfly (FFT algorithm)".

- The FFT function uses a technique called bit reversal to rearrange the input points so that the outputs are in the correct order.
- Conventional software FFT implementations obtain some of their speed by pre-calculating sine and cosine terms used in the butterfly calculations. These sine and cosine terms are called twiddle factors.

The example design included with this tutorial is based on a radix-two implementation of the FFT function. This a decimation-in-time FFT, which decomposes the original 256-point DFT into two 128-point DFTs, which it then breaks down to four 64-point DFTs, and so on, until ultimately it evaluates 128 two-point DFTs.

Analyzing the FFT Code

A typical first step with the C2H Compiler is to accelerate the C function without restructuring the code. This approach rarely yields optimal performance. However, it provides performance metrics which allow you to identify the system bottlenecks.

Creating a Build Report

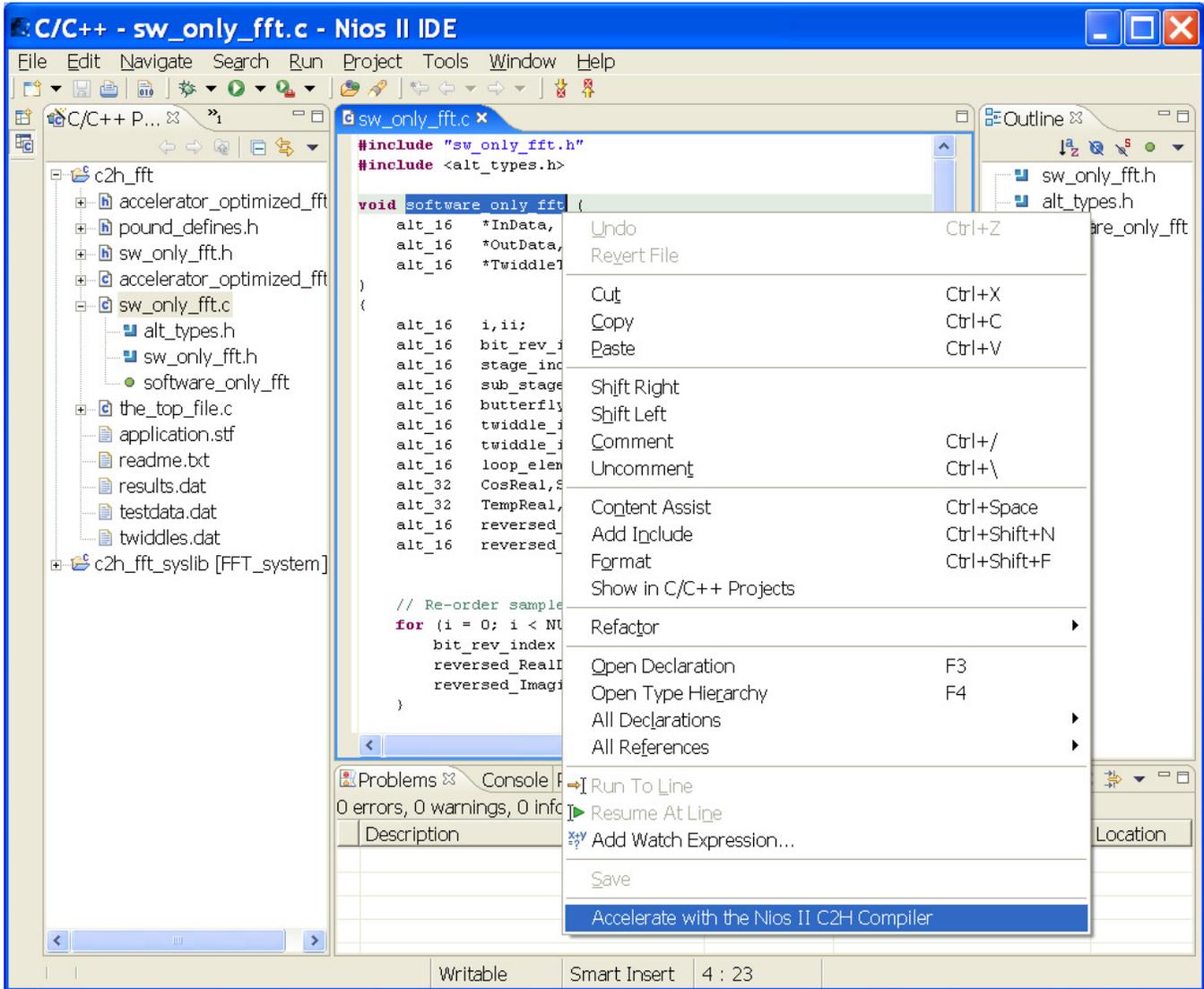
The C2H Compiler provides tools to help you analyze your C code. Carry out the following steps to see an analysis of the FFT function.

1. Launch the Nios II IDE if it is not already running.
2. Open the file `sw_only_fft.c` in the application project.
3. Highlight the function name `software_only_fft`, right-click, and click **Accelerate with the Nios II C2H Compiler**, as shown in [Figure 1](#).
4. The Nios II IDE displays the C2H view. Expand the folders labeled `c2h_fft` and `sw_only_fft()`, and select **Use Software implementation** and **Analyze all accelerators**. Make sure the settings are as follows:
 - Use software implementation for all accelerators
 - Use hardware accelerator in place of software implementation. Flush data cache before each call

The message **Build report cannot be displayed** is normal at this point.

5. Right click the `c2h_fft` application project, and then click **Build Project**. As part of the build process, the Nios II IDE performs the following steps:
 - Analyzes the function to be accelerated, determining the mapping from C constructs to hardware, and computing performance metrics.
 - Compiles the software executable, using the software implementation of `software_only_fft()`. Depending on the speed of your platform, this can take ten to twenty minutes. While you wait for the build to complete, you might wish to read ahead in the [Unoptimized Accelerator](#) section. This section describes how the C2H Compiler accelerates `software_only_fft()` without optimizations.

Figure 1. Accelerating the Function



 The Nios II compiler displays the following message: ignoring `#pragma altera_accelerate connect_variable`. You can disregard this warning. The C2H Compiler uses the `altera_accelerate` pragma to limit the number of master ports, as described in the [Restructuring Code to Optimize the Accelerator](#) section. It has no meaning for the Nios II compiler.

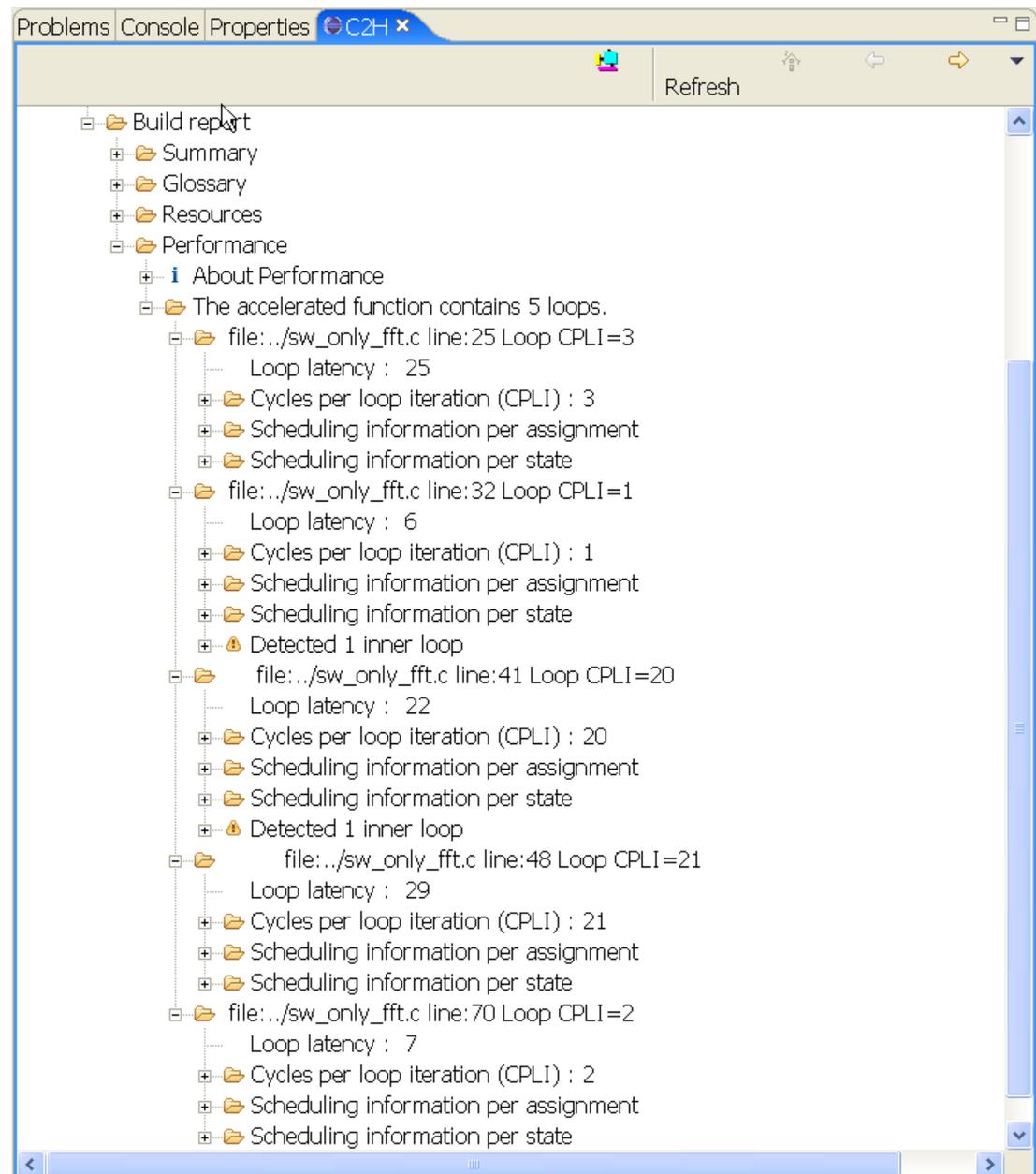
Performance Metrics

Although the C2H Compiler can accelerate unmodified ANSI C code, you typically need to modify the code to build a fully optimized hardware design. To gain insight into which portions of the function might need optimization, look at the build report generated by the C2H Compiler. The build report appears in the C2H view after you build the project. To see the performance metrics, expand the folders labeled **Build Report**, **Performance**, and **The accelerated function contains 5 loops**, and then expand each folder labeled `file:./sw_only_fft.c line:n Loop CPLI=m`, as shown in [Figure 2](#) on page 6.

The most important performance metrics are:

- Cycles per loop iteration (CPLI) — the number of clock cycles per loop iteration in the best case. The lowest possible value of CPLI is 1. This means that an iteration of the loop occurs every clock cycle, assuming no stalling for inner loops or memory access. A loop with $CPLI = 1$ has the maximum throughput possible without fundamental restructuring beyond the scope of this tutorial.
- Loop latency — the initial overhead when the accelerator enters the state machine implementing a C loop. The accelerator must fill its pipeline before the first result is ready, and the loop latency is the number of clock cycles it requires to do so.

Figure 2. Performance Metrics for Non-Optimized FFT



The results in [Figure 2](#) show that most of the values of loop latency and CPI are high. The next section provides an overview of how the C2H Compiler generates the unoptimized accelerator.

Unoptimized Accelerator

The C2H Compiler translates C constructs to their hardware equivalents in a straightforward way. C code is usually designed assuming serial execution on a CPU, and therefore is not optimal for parallel execution on hardware.

The FFT function uses the following operations:

- Memory Access
- Multiplication
- Division
- Addition and subtraction
- Bit Shifting
- Iteration with counter

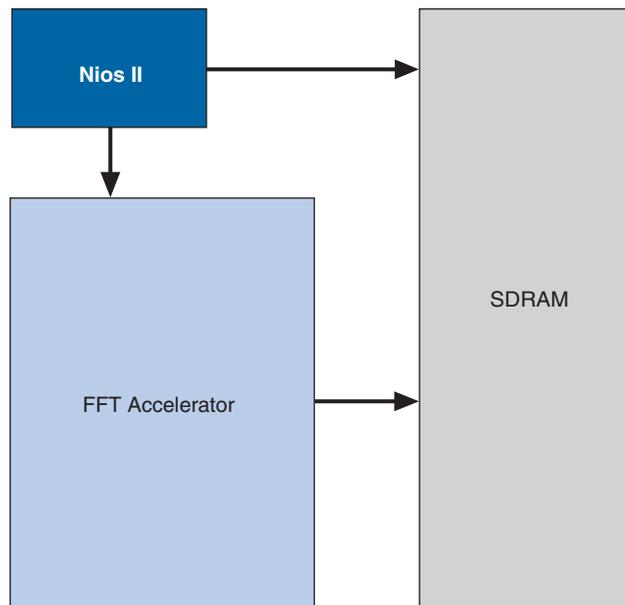
All of the operations listed above translate to hardware constructs.



For further information about C2H hardware transforms, refer to the chapter *C-to-Hardware Mappings Reference* in the *Nios II C2H Compiler User Guide*.

[Figure 3](#) illustrates the system that the C2H Compiler builds when you accelerate the unoptimized FFT function.

Figure 3. Hardware Accelerated System



In the next section, you accelerate the FFT algorithm with optimizations for the C2H Compiler.

Optimizing the FFT

This tutorial includes a modified version of the FFT algorithm, `accelerator_optimized_fft()`. It is optimized according to techniques described later in this tutorial. In this section, you perform the following steps:

- Accelerate the function `accelerator_optimized_fft()`
- Build the software project
- Compile the hardware design, which includes the accelerator
- Run the accelerated FFT, comparing its performance with the software implementation

The following sections guide you through the process of creating the optimized accelerator.

Adding On-Chip Buffers

Perform the following steps to prepare the hardware project for optimized acceleration:

1. Start the Quartus II development software, and open the Quartus II project (**2s60_fft_acceleration.qpf** or **2c35_fft_acceleration.qpf**).
2. Start SOPC Builder.
3. Add four on-chip memories to the SOPC Builder system with the following properties:
 - **Memory Type = RAM**
 - **Dual-Port Access** enabled
 - **Memory Width = 16 bits**
 - **Total Memory Size = 512 bytes**
 - **Read Latency = 1** (this applies to both slave ports)

The accelerator uses these on-chip memories to buffer the input and output data for the FFT. The [Restructuring Code to Optimize the Accelerator](#) section discusses the reasons for the memory settings.

4. Give the memories the following names:
 - **BufferRAM1**
 - **BufferRAM2**
 - **BufferRAM3**
 - **BufferRAM4**
5. Add two on-chip memories to the system with the following properties.
 - **Memory Type = RAM**
 - **Dual Port Access** disabled
 - **Memory Width = 16 bits**
 - **Total Memory Size = 512 bytes**
 - **Read Latency = 1**

The accelerated code uses these on-chip buffers to store sine and cosine terms. The [Restructuring Code to Optimize the Accelerator](#) section discusses the reasons for the memory settings.

6. Give these memories the following names:
 - **CosRAM**
 - **SinRAM**
7. Disconnect all on-chip memory slave ports. By default, SOPC Builder connects the on-chip memories to the Nios II processor's instruction and data master ports. The C2H Compiler connects the memory's slave ports to the accelerator at build time.
8. Set each memory's base address as shown in [Table 1](#). In the case of dual-port memories, set both slave ports to the same base address. Be sure to lock the base address of all on-chip memories in the system.

Table 1. Memory Addresses

Memory Name	Base Address
BufferRAM1	0x00000000
BufferRAM2	0x00000200
BufferRAM3	0x00000400
BufferRAM4	0x00000600
CosRAM	0x00000800
SinRAM	0x00000A00



It is important to use the exact names shown in Table 1, because the FFT source code refers to them explicitly.

9. Verify that your system resembles the system depicted in [Figure 4](#).

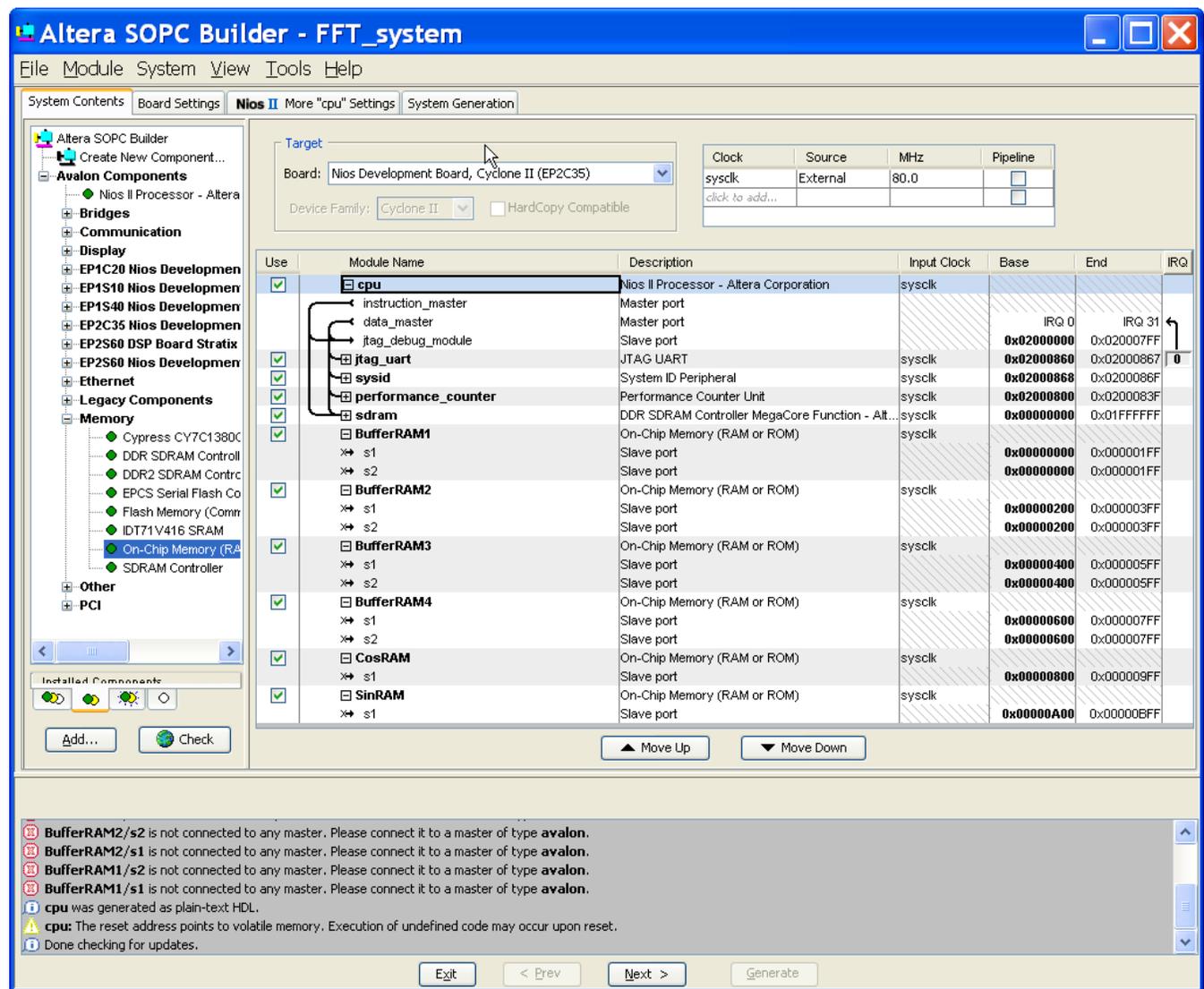
Disregard the messages stating that the slave ports are not connected to any master port. The C2H Compiler connects them later, when it builds the accelerator.
10. Exit SOPC Builder, making sure to save the system when prompted. You do not need to generate the SOPC Builder system at this point. The C2H Compiler generates it for you after it builds the accelerator.

Building the Accelerator

Perform the following steps to build the optimized accelerator.

1. Launch the Nios II IDE, if it is not already running.
2. Remove the accelerator from `software_only_fft()`.
 - a. In the C2H view, select the function name, right-click, and click **Remove C2H**. The Nios II IDE prompts you: **Do you really want to remove the function "software_only_fft()" from the list of functions to accelerate?**
 - b. Click **Yes**. A message appears saying **The accelerator has been removed. Rebuild the project to update the SOPC Builder system.**
 - c. Click **OK**.

Figure 4. SOPC Builder System



- In the file `accelerator_optimized_fft.c`, accelerate the function `accelerator_optimized_fft()`, as you accelerated `software_only_fft()` in the [Creating a Build Report](#) section on page 4. This time, leave **Build software, generate SOPC Builder system, and run Quartus II compilation** selected (the default).
- Expand the `c2h_fft` and `accelerator_optimized_fft()` folder icons in the C2H view, and make sure the following settings are turned on:
 - Build software, generate SOPC Builder system, and run Quartus II compilation
 - Use hardware accelerator in place of software implementation. Flush data cache before each call
- Build the application project again. As part of the build process, the Nios II IDE performs the following steps:
 - Analyzes the function to be accelerated, determining the mapping from C constructs to hardware, and computing performance metrics

- Integrates the accelerator into the SOPC Builder system
- Generates the HDL
- Compiles the system in Quartus II
- Creates a wrapper function to invoke the accelerator
- Compiles the software executable

The build process might take 20 to 40 minutes, depending on the speed of your platform. While you wait, you might wish to read ahead in the [Bottlenecks](#) and [Restructuring Code to Optimize the Accelerator](#) sections. These sections describe how `accelerator_optimized_fft()` is optimized to improve the accelerator performance.



The Nios II compiler displays the following message: `ignoring #pragma altera_accelerate_connect_variable`. You can disregard this warning. The C2H Compiler uses the `altera_accelerate` pragma to limit the number of master ports, as described in the [Restructuring Code to Optimize the Accelerator](#) section. It has no meaning for the Nios II compiler.

Optimized Performance Metrics

After you accelerate the function, the Nios II IDE displays a new set of performance metrics in the C2H view.

The new performance metrics show that the latency of most loops is lower, and `CPLI=1` for each for loop in the design. Even though the calculation stage consists of three nested loops, each loop has `CPLI=1`, minimizing stalling of the outer loops.

To review the accelerator that the C2H Compiler has added to the SOPC Builder System, open the design in SOPC Builder. The system connections and on-chip memory base addresses appear.

You might notice that the on-chip RAM slave ports are not visibly connected. This is normal. SOPC Builder hides accelerator master ports, because they are often so numerous that the connection grid is unreadable. You cannot use SOPC Builder to edit master-slave connections inserted by the C2H Compiler.

Downloading and Running the Accelerated System

Perform the following steps to download and run the accelerated system.

1. After the compilation has finished, download the resulting FPGA configuration file (`.sof`) to the development board using the Quartus II programmer.
2. Return to the Nios II IDE, right-click the `c2h_fft` application project, point to **Run As**, and click **Nios II Hardware** to run the software project on the development board.

The example executes 1000 iterations of the unaccelerated and accelerated FFT functions, and verifies that the output data from the last run of each is valid. After the software is finished, the results of the FFT benchmark appear in the Console view of the Nios II IDE. The console output for the Nios II Cyclone II development board resembles [Figure 5](#).

Figure 5. FFT Benchmark Sample Output

```

FFT Benchmark Starting (this will take up to 20 seconds)
- Running 1000 iterations for both software and hardware.
- Each iteration runs a 256 point radix 2 FFT transformation.

--Performance Counter Report--
Total Time: 0.930541 seconds (93054055 clock-cycles)
+-----+-----+-----+-----+-----+
| Section      | %   | Time (sec)| Time (clocks)|Occurrences|
+-----+-----+-----+-----+-----+
|Software Only | 94.3| 0.87767| 87767457| 1|
+-----+-----+-----+-----+-----+
|HW Accelerated| 5.67| 0.05272| 5271886| 1|
+-----+-----+-----+-----+-----+

The software only output data is correct
The hardware accelerated output data is correct

```

The report details the performance of the FFT on the Nios II processor using unaccelerated software, followed by the performance results from the FFT accelerated with the C2H Compiler. In the example shown in [Figure 5](#), the C2H Compiler has improved the performance of the FFT calculation by a factor of approximately 16.

The remainder of this tutorial describes the techniques used to achieve this performance improvement.

Bottlenecks

This tutorial shows three common types of performance bottlenecks which can occur in unoptimized accelerators, discussed in the following sections:

- [Computational Bottlenecks](#)
- [Memory Bottlenecks](#)
- [Multiple Master Port Memory Stalls](#)

Computational Bottlenecks

The FFT is subject to a common type of computational bottleneck caused by mismatched data widths. `software_only_fft()`, designed for a Nios II system with plenty of memory and a 32-bit data path, uses 32-bit signed data types to avoid overflow and underflow. However, when you implement a design in hardware, wide data paths consume more logic than narrow ones, which can reduce f_{MAX} for the entire design. When accelerating software functions in hardware, it is best to tailor the data width to your exact data range requirements.

Selection of signed or unsigned data types also plays a role in the performance of the design. When possible, use unsigned values. Converting an unsigned value to signed is trivial, but the opposite conversion requires extra logic.

Memory Bottlenecks

Memory bottlenecks cause the largest performance penalty in the unoptimized hardware accelerator. This tutorial exemplifies two common types of memory bottleneck:

- [Narrow Memory Access](#)
- [Random Access to SDRAM](#)

The following sections discuss these types of memory bottleneck.

Narrow Memory Access

The SDRAM used in the example design has a 32 bit data width. However, the FFT software function uses 16 bit data types. Therefore each time the accelerator fetches a variable from memory, half of the memory bandwidth is wasted, because the function only uses 16 bits. For the best performance, access high latency memory devices with bus transfers of the same width as the memory device.

Random Access to SDRAM

The SDRAM device used in the example design suffers from long latency times. SDRAM devices achieve their highest bandwidth when they are accessed sequentially. By contrast, SRAM based memories typically have no performance penalty for random (non-sequential) access.

There are two situations in which the FFT algorithm accesses the SDRAM non-sequentially:

- [Single Master Port Random Access](#)
- [Multiple Master Port Random Access](#)

The following sections discuss each of these situations.

Single Master Port Random Access

The FFT algorithm uses a technique called bit reversal to rearrange the input points so that the outputs are in the correct order. The bit reversal values form a pattern of array indices beginning with the following values: 0, 128, 64, 192, and 32. The algorithm uses these values as array indices for each input point read from SDRAM. This causes poor memory performance, because the indices are not sequential.

Multiple Master Port Random Access

The hardware accelerator contains multiple Avalon-MM master ports, all of which compete for access to the SDRAM. Each master port accesses independent locations within memory. This results in non-sequential memory accesses when the Avalon interconnect fabric arbitrates between master ports.

This type of random access is typical of any system with multiple master ports.

Multiple Master Port Memory Stalls

Another memory bottleneck results from the physical limitations of memory interfaced with multiple master ports. Only one master port can access the memory at a time. If a second master port tries to access memory when the first is in control, the second master port must wait. This stalls the pipeline.

The FFT hardware accelerator must read data, sine and cosine terms from memory, and also write results back to memory. These multiple types of memory access cause memory stalls. One state in the pipelined transform can be starved of input data when master ports belonging to other states gain access to the SDRAM.

Restructuring Code to Optimize the Accelerator

To improve accelerator performance, optimize your code to allow independent reading and writing tasks to occur in parallel. This optimization technique requires fast data buffering, which you can implement using on-chip memory available in the FPGA. This is why you add the on-chip memories in the [Adding On-Chip Buffers](#) section.

Data Buffering Optimizations

This example illustrates several types of data buffering optimizations:

- [Fast Memory](#)
- [Double Buffering](#)
- [Master Port Minimization](#)
- [Sine and Cosine Data Buffering](#)
- [Bit Reversal Buffering](#)

Fast Memory

The accelerated FFT stores data in fast memory with a low, fixed latency. The on-chip memories have a latency of 1, the lowest available. Low latency lets the calculation stage access the data rapidly, because it reduces the number of states the C2H Compiler must create for the state machines that schedule the memory access. Fixed latency means that the accelerator need not access the memory sequentially to achieve the highest throughput.

Double Buffering

The software FFT implementation performs in-place data calculations. The software implementation uses the same memory buffer to load input values, save intermediate calculation results, and store output values. In-place calculations save memory resources, but often slow performance when translated to hardware. Full concurrency is difficult to achieve with a single buffer, because of the memory stalls described in [Multiple Master Port Memory Stalls](#) on page 13.

The optimized FFT accelerator uses on-chip memory to buffer the input and output data, so that the calculation portion of the accelerator can operate independently of SDRAM. To achieve full concurrency, the transformation phase of the FFT must be able to read and write at the same time.

The optimized FFT achieves this with double buffering. Double buffering, also known as ping-pong buffering, allows one master port in the FFT accelerator to read input data from one buffer while another master port writes results into another buffer. This type of data buffering avoids memory stalls.

[Figure 6](#) illustrates a simplified form of the double buffering scheme used in the optimized FFT function. In this FFT implementation, double buffering requires a total of 4 buffers. This is because the FFT performs calculations on real and imaginary input data simultaneously. The FFT uses one pair of buffers for real data, and one pair for imaginary data. The real and imaginary calculations are independent of one another, so the accelerator can perform them concurrently. This section describes the buffering method for the real input data. The algorithm handles imaginary data exactly the same.

Figure 6. Hardware Accelerated Data Buffering Scheme

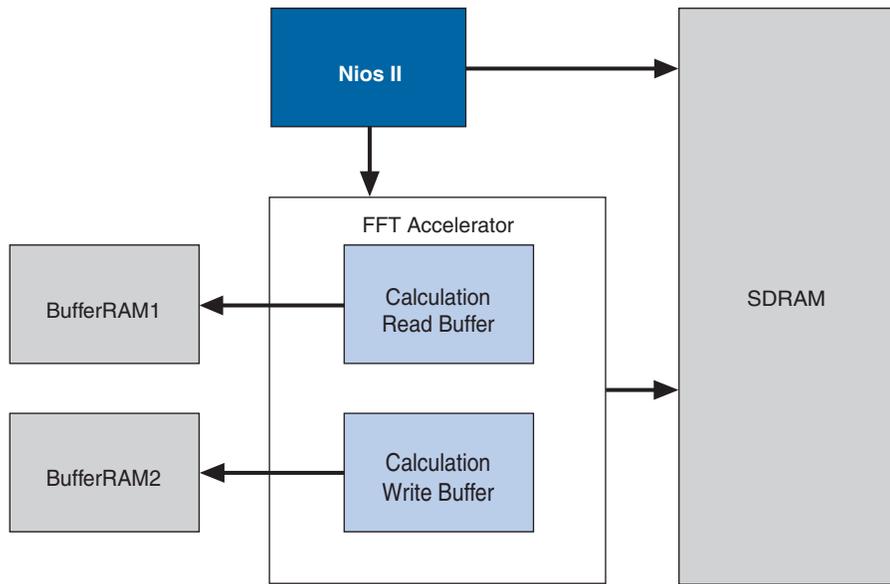
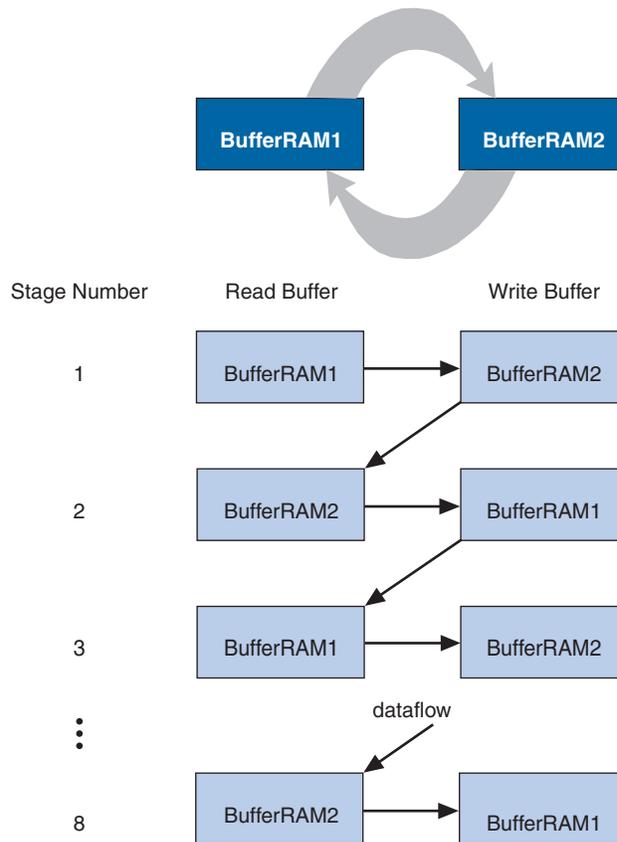


Figure 7 illustrates the double buffering scheme used in the optimized code.

Figure 7. Double Buffering



The 256-point FFT in this example decomposes into eight stages. During the first stage the read buffer is **BufferRAM1** and the write buffer is **BufferRAM2**, as shown in [Example 1](#).

Example 1. accelerator_optimized_fft.c: Initial Buffer State

```

52      /* Assign the ping pong buffers default address locations */
53      BufferedRealCalcDataRead = BufferRAM1;
54      BufferedRealCalcDataReadPort2 = BufferRAM1;
55      BufferedRealCalcDataWrite = BufferRAM2;
56      BufferedRealCalcDataWritePort2 = BufferRAM2;

```

The algorithm swaps the buffers after each stage, as shown in [Example 2](#).

Example 2. accelerator_optimized_fft.c: Swapping Buffers

```

144      BufferedRealCalcDataRead = BufferedRealCalcDataWrite;
145      BufferedRealCalcDataWrite = BufferedRealCalcDataReadPort2;
146      BufferedRealCalcDataReadPort2 = BufferedRealCalcDataRead;
147      BufferedRealCalcDataWritePort2 = BufferedRealCalcDataWrite;

```

When all eight stages are complete, the results of the FFT are in **BufferRAM1**. The function copies the results back to SDRAM, as shown in [Example 3](#).

Example 3. accelerator_optimized_fft.c: Results Stored in SDRAM

```

155      /* returning the interleaved results to sdram
156      * Since the data is 16 bit and interleaved we'll stick the real and
157      * imaginary parts together and send them off to sdram */
158      for(outputCounter = 0; outputCounter < NUM_POINTS; outputCounter++) {
159          tempOutputPtr[outputCounter] =
160              (((alt_u32)(BufferedImagCalcDataRead[outputCounter]) & 0x0000FFFF)<<16) |
161              ((alt_u32)BufferedRealCalcDataRead[outputCounter] & 0x0000FFFF);

```

The accelerator maintains data integrity, because it always reads from memory written to on the previous loop iteration. Because there is an even number of stages, the finished data ends up in **BufferRAM1**.

This buffering scheme improves performance, because when the accelerator is performing butterfly calculations, it can read and write data concurrently. Also, the on-chip buffers generate no excess wait states which could cause pipeline stalls.

The input consists of 256 data points, each 16 bits wide. This means that each of the two on-chip memory buffers is 512 bytes long. As the memory block type is set to **Automatic** in SOPC Builder, the Quartus II development software allocates it to a single M4K block if available. In the tutorial, the real data flows through buffers **BufferRAM1** and **BufferRAM2**, while the imaginary data flows through buffers **BufferRAM3** and **BufferRAM4**.

The double buffer scheme described above connects two master ports to each of the buffers (the calculation read master port and the calculation write master port).

Master Port Minimization

When designing buffer schemes for the C2H Compiler, it is important to consider how many master ports are connected to each memory. The more master ports that are connected to a memory the higher the likelihood of degrading system f_{MAX} , and causing memory stalls as master ports contend for access to the memory. There is also a higher likelihood of master ports competing for the same resources when large numbers of master ports are used. Therefore it is important to balance the number of master ports connected to each memory with the throughput needs of the system.

When multiple master ports are connected to a slave port, SOPC Builder must generate logic to arbitrate among them. This logic, if too complex, causes a reduction in f_{MAX} .

In this example, only the FFT accelerator needs to access the on-chip memories, so the Nios II processor is not connected to them. The accelerated code uses pragma directives to connect only the master ports that are required.

Sine and Cosine Data Buffering

The software FFT implementation in this tutorial stores the sine and cosine terms (twiddle factors) in SDRAM. Storing these values in low latency on-chip memory increases performance.

In this tutorial, the sine terms are in **SinRAM**, and the cosine terms are in **CosRAM**, as shown in [Example 4](#). The Quartus II development software initializes the memories from files **SinRAM.hex** and **CosRAM.hex**, which contain the precalculated sine and cosine terms.

Example 4. `accelerator_optimized_fft.c`: Sine and Cosine Data

```
63      /* Point the Cosine and Sine Tables to the CosRAM and SinRAM on-chip memory
64      * buffers. These memories are local to the accelerator and are not shared
65      * with the Nios II processor. */
66      CosineTable = CosRAM;
67      SineTable = SinRAM;
```

Bit Reversal Buffering

The optimized FFT algorithm reads input data sequentially from SDRAM and stores it in the read buffer. It then accesses the read buffer non-sequentially, using the bit reversal indexes. The read buffer is implemented in on-chip memory, which has no latency penalty for random access. The bit reversal pattern is the only part of the FFT function that accesses memory in a non-sequential order. This means that all of the accelerator's SDRAM accesses are sequential, taking advantage of the SDRAM's optimal bandwidth.

SDRAM Memory Access Optimizations

It is important to use the full data width of the available memory interface. In the FFT, each real and imaginary data point is 16 bits wide. However, the SDRAM device used in this example has a 32 bit interface. For example, the unoptimized code makes two separate accesses to SDRAM when reading the data samples, as shown in [Example 5](#). This wastes half the bandwidth of the SDRAM interface.

Example 5. *sw_only_fft.c: Unoptimized SDRAM Access*

```

24     // Re-order samples using bit reversal
25     for (i = 0; i < NUM_POINTS; i++) {
26         bit_rev_index = bitrev(i);
27         reversed_RealData[bit_rev_index] = InData[2*i];
28         reversed_ImaginaryData[bit_rev_index] = InData[2*i+1];
29     }

```

Each data point is a complex number, stored as a pair (real and imaginary). Thus a single 32 bit read from SDRAM can access the entire pair. The optimized hardware accelerator uses this fact in the input stage, reading data pairs and storing the real and imaginary components concurrently into separate buffers. The accelerator uses the same optimization in the output phase, storing the real and imaginary components into SDRAM in a single 32 bit write.

This code is optimized for the C2H Compiler by reading from the SDRAM into a temporary variable and then writing each half of the temporary variable into the appropriate buffer, as shown in [Example 6](#).

Example 6. *accelerator_optimized_fft.c: Optimizing SDRAM Access*

```

71     /* Calculate the bitreversal index and read
72     * 32 bits of data from the input buffer in SDRAM (real and imaginav pair).
73     * Split the data read into half and write them into real and imaginary
74     * buffers concurrently */
75     for (inputCounter = 0; inputCounter < NUM_POINTS; inputCounter++) {
76         bit_rev_index = bitrev(inputCounter);
77
78         tempInput = tempInputPtr[inputCounter];
79         BufferedRealCalcDataRead[bit_rev_index] = (alt_16)(tempInput & 0x0000FFFF);
80         BufferedImagCalcDataRead[bit_rev_index] = (alt_16)((tempInput & 0xFFFF0000)>>16);
81     }

```

Calculation Stage Optimizations

The FFT function uses complex multiplications to calculate the outputs from each butterfly calculation. The algorithm implements complex multiplication as four multiplications, one addition and one subtraction.

To improve the throughput of the FFT butterfly calculation, the accelerator uses four separate hardware multipliers, so that all the mathematical operations occur on a single clock cycle, as shown in [Example 7](#). This improves the computational bandwidth of the accelerator. However, all the inputs to the computation stage come from on-chip memory buffers. To maximize the computational throughput, the memory buffers must be able to match the throughput.

Example 7. *accelerator_optimized_fft.c: Using Parallel Multipliers*

```

113     /* Scale twiddle products to accomodate 16 bit storage */
114     /* CosReal, SinReal, temp1, and temp2 are all registers so no
115     * waiting occurs here (this happens concurrently) */
116     tRealData = (( CosReal * temp1 ) + ( SinReal * temp2 ))>> PRESCALE;
117     tImagData = (( CosReal * temp2 ) - ( SinReal * temp1 ))>> PRESCALE;

```

To improve the throughput of the memory buffers, the read and write buffers are implemented as dual-port memories. Dual-port buffers are helpful because the butterfly calculation uses two inputs for every output. The

Conclusion

two inputs always come from different memory locations. Therefore the accelerator can retrieve them simultaneously without collision. [Example 8](#) shows the optimized code.

Example 8. accelerator_optimized_fft.c: Using Dual-Port Memory

```
104      /* using temps (regs) to allow this to happen concurrently since
105      * these are DP RAM accesses that do not overlap. We are using read
106      * pointers here so that the write pointers at the bottom can work in
107      * parallel */
108      temp1 = BufferedRealCalcDataRead[1];
109      temp2 = BufferedImagCalcDataRead[1];
110      temp3 = BufferedRealCalcDataReadPort2[butterfly_index];
111      temp4 = BufferedImagCalcDataReadPort2[butterfly_index];
```

Conclusion

With a few straightforward code optimizations, the Nios II C2H Compiler can sharply improve the computational bandwidth and memory throughput of a software algorithm.

In the case of an FFT, we apply the following optimizations:

- Use 16-bit integers in place of 32-bit integers
- Use unsigned integers in place of signed integers
- Fetch data from SDRAM 32 bits at a time
- Avoid non-sequential SDRAM access by buffering data in SRAM
- Avoid multi-master-port memory stalls by buffering data and constants in multiple memories
- Facilitate pipelining with double buffering and dual-port RAM
- Reduce wait states by using low-latency on-chip RAM

The optimized accelerator is up to 50 times faster than a software-only implementation.



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Technical support:
www.altera.com/support
Product literature:
www.altera.com/literature

© 2008 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

