Developer Zone

Introduction to GEN Assembly

Submitted by Robert Ioffe (Intel) on January 22, 2016

Contents

Introduction

In order to better optimize and debug OpenCL kernels, sometimes it is very helpful to look at the underlying assembly. This article shows you the tools available in the Intel® SDK for OpenCL™ Applications (https://software.intel.com/en-us/intel-opencl) that allow you to view assembly generated by the offline compiler for individual kernels, highlight the regions of the assembly code that correspond to OpenCL C code, as well as attempts at a high level explain different portions of the generated assembly. We also give you a brief overview of the register region syntax and semantics, show different types of registers, and summarize available assembly instructions and data types that these instructions can manipulate on. We hope to give you enough ammunition to get started. In the upcoming articles we will cover assembly debugging as well as assembly profiling with Intel® VTune™ Amplifier (https://software.intel.com/en-us/intel-vtune-amplifier-xe/).

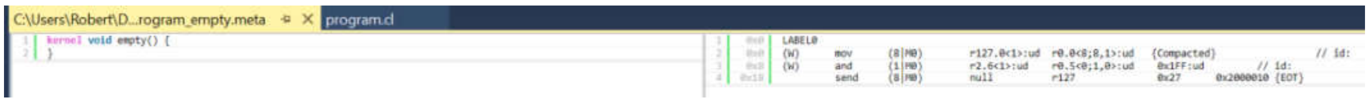
Assembly for Simple OpenCL Kernels

Let us start with a simple kernel:

```
kernel void empty() {
}
```

Figure 1. The Simplest OpenCL Kernel.

This is as simple as kernels get. We are going to build this kernel in a Code Builder Session Explorer. Go ahead and create a new session by going to CODE-BUILDER/OpenCL Kernel Development/New Session, copying the kernel above to an empty program.cl file and then building it. If you have a 5th generation Intel processor (Broadwell) or a 6th generation Intel processor (Skylake), you will notice that one of the artifacts being generated is program_empty.gen file. Go ahead and double-click on it. What you will see is something like this:



```
kernel void empty() {
}
```

```
0x0   LABEL0
0x0   (W)   mov    (8|M0)   r127.0<1>:ud   r0.0<8;8,1>:ud   {Compacted}              // id:
0x8   (W)   and    (1|M0)   r2.6<1>:ud     r0.5<0;1,0>:ud   0x1FF:ud         // id:
0x10        send   (8|M0)   null           r127             0x27      0x2000010 {EOT}
```

Figure 2. The Empty Kernel and Corresponding Assembly.

The assembly for the kernel is on the right: let me annotate it for you:

```
// Start of Thread
LABEL0
(W)      and      (1|M0)        r2.6<1>:ud    r0.5<0;1,0>:ud     0x1FF:ud          // id:

// End of thread
(W)      mov      (8|M0)        r127.0<1>:ud  r0.0<8;8,1>:ud    {Compacted}                  // id:
         send     (8|M0)        null          r127               0x27      0x2000010 {EOT}  // id:
```

Figure 3. Annotated Assembly of the Empty Kernel.

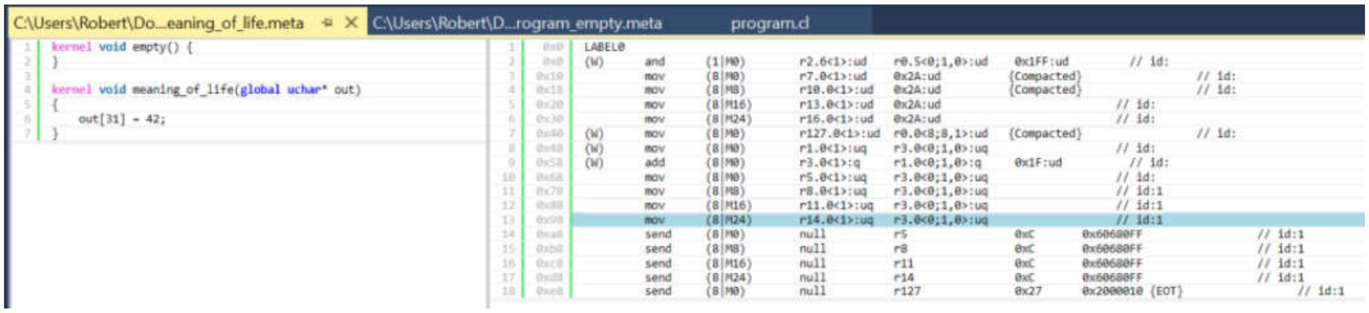
Not much, but it is a start.

Now, let's complicate life a little. Copy the following into program.cl:

```
kernel void meaning_of_life(global uchar* out)
{
 out[31] = 42;
}
```

Figure 4. meaning_of_life Kernel.

After rebuilding the file you will notice program_meaning_of_life.gen file. After double clicking on it you will see something more complex:



C:\Users\Robert\Do...eaning_of_life.meta   C:\Users\Robert\D...rogram_empty.meta   program.cl

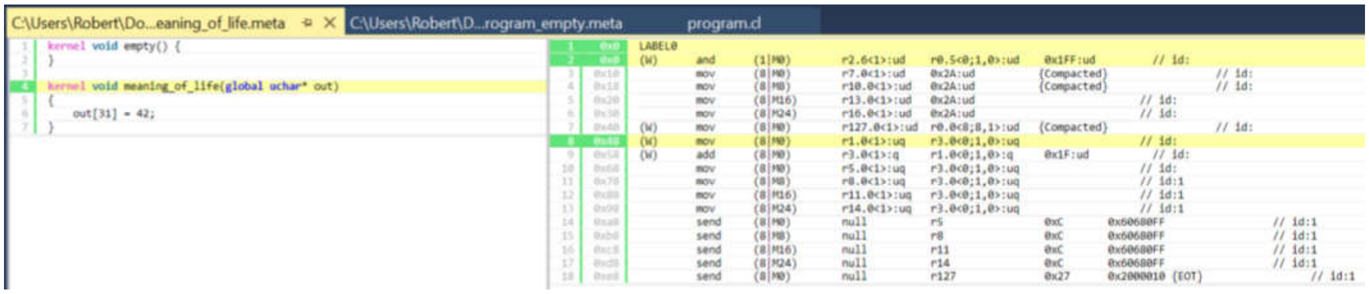
```
kernel void empty() {
}

kernel void meaning_of_life(global uchar* out)
{
    out[31] = 42;
}
```

```
0x0   LABEL0
0x0   (W)   and    (1|M0)    r2.6<1>:ud     r0.5<0;1,0>:ud   0x1FF:ud        // id:
0x10        mov    (8|M0)    r7.0<1>:ud     0x2A:ud          {Compacted}               // id:
0x18        mov    (8|M8)    r10.0<1>:ud    0x2A:ud          {Compacted}               // id:
0x20        mov    (8|M16)   r13.0<1>:ud    0x2A:ud                          // id:
0x30        mov    (8|M24)   r16.0<1>:ud    0x2A:ud                          // id:
0x40  (W)   mov    (8|M0)    r127.0<1>:ud   r0.0<8;8,1>:ud   {Compacted}               // id:
0x48  (W)   mov    (8|M0)    r1.0<1>:uq     r3.0<0;1,0>:uq                   // id:
0x58  (W)   add    (8|M0)    r3.0<1>:q      r1.0<0;1,0>:q    0x1F:ud         // id:
0x68        mov    (8|M0)    r5.0<1>:uq     r3.0<0;1,0>:uq                   // id:
0x78        mov    (8|M8)    r8.0<1>:uq     r3.0<0;1,0>:uq                   // id:1
0x88        mov    (8|M16)   r11.0<1>:uq    r3.0<0;1,0>:uq                   // id:1
0x98        mov    (8|M24)   r14.0<1>:uq    r3.0<0;1,0>:uq                   // id:1
0xa8        send   (8|M0)    null           r5               0xC     0x60680FF          // id:1
0xb8        send   (8|M8)    null           r8               0xC     0x60680FF          // id:1
0xc8        send   (8|M16)   null           r11              0xC     0x60680FF          // id:1
0xd8        send   (8|M24)   null           r14              0xC     0x60680FF          // id:1
0xe8        send   (8|M0)    null           r127             0x27    0x2000010 {EOT}      // id:1
```

Figure 5. meaning_of_life Kernel and Corresponding Assembly.

What you can do now is to click on different parts of the kernel on the left, and see different parts of assembly being highlighted:

Here are instructions corresponding to the beginning of the kernel:



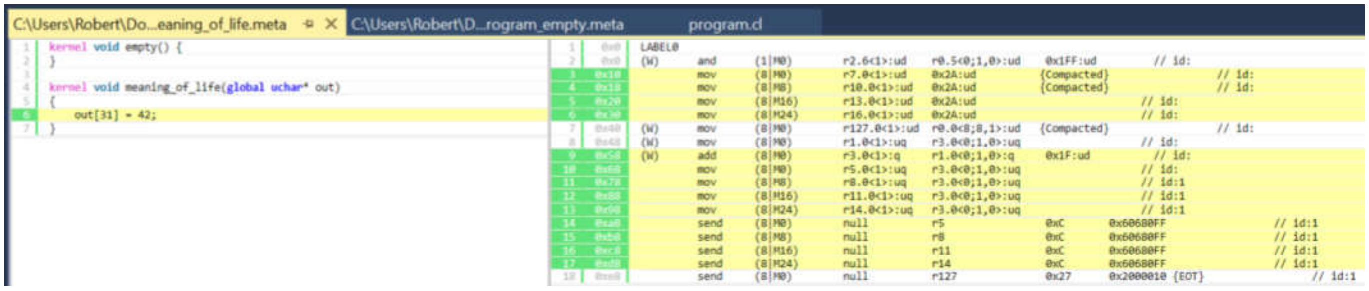
C:\Users\Robert\Do...eaning_of_life.meta | C:\Users\Robert\D...rogram_empty.meta | program.cl

```
kernel void empty() {
}

kernel void meaning_of_life(global uchar* out)
{
    out[31] = 42;
}
```

```
0x0          LABEL0
0x8    (W)   and     (1|M0)     r2.6<1>:ud     r0.5<0;1,0>:ud   0x1FF:ud            // id:
0x10         mov     (8|M0)     r7.0<1>:ud     0x2A:ud          {Compacted}                 // id:
0x18         mov     (8|M8)     r10.0<1>:ud    0x2A:ud          {Compacted}                 // id:
0x20         mov     (8|M16)    r13.0<1>:ud    0x2A:ud                              // id:
0x30         mov     (8|M24)    r16.0<1>:ud    0x2A:ud                              // id:
0x40   (W)   mov     (8|M0)     r127.0<1>:ud   r0.0<8;8,1>:ud   {Compacted}                 // id:
0x48   (W)   mov     (8|M0)     r1.0<1>:uq     r3.0<0;1,0>:uq                       // id:
0x58   (W)   add     (8|M0)     r3.0<1>:q      r1.0<0;1,0>:q    0x1F:ud             // id:
0x68         mov     (8|M0)     r5.0<1>:uq     r3.0<0;1,0>:uq                       // id:
0x78         mov     (8|M8)     r8.0<1>:uq     r3.0<0;1,0>:uq                       // id:1
0x88         mov     (8|M16)    r11.0<1>:uq    r3.0<0;1,0>:uq                       // id:1
0x98         mov     (8|M24)    r14.0<1>:uq    r3.0<0;1,0>:uq                       // id:1
0xa8         send    (8|M0)     null           r5               0xC    0x6060680FF           // id:1
0xb8         send    (8|M8)     null           r8               0xC    0x6060680FF           // id:1
0xc8         send    (8|M16)    null           r11              0xC    0x6060680FF           // id:1
0xd8         send    (8|M24)    null           r14              0xC    0x6060680FF           // id:1
0xe8         send    (8|M0)     null           r127             0x27   0x2000010 {EOT}         // id:1
```

Figure 6. Beginning of meaning_of_life kernel.

The body of the kernel:



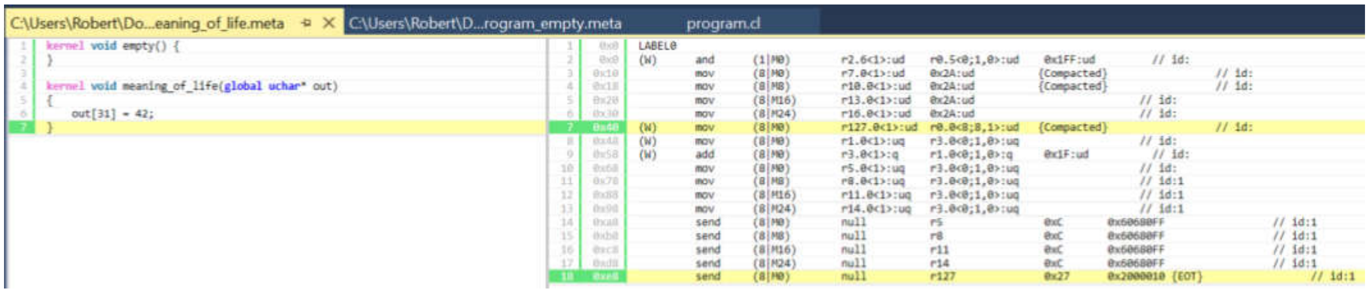
C:\Users\Robert\Do...eaning_of_life.meta | C:\Users\Robert\D...rogram_empty.meta | program.cl

```
kernel void empty() {
}

kernel void meaning_of_life(global uchar* out)
{
    out[31] = 42;
}
```

```
0x0          LABEL0
0x8    (W)   and     (1|M0)     r2.6<1>:ud     r0.5<0;1,0>:ud   0x1FF:ud            // id:
0x10         mov     (8|M0)     r7.0<1>:ud     0x2A:ud          {Compacted}                 // id:
0x18         mov     (8|M8)     r10.0<1>:ud    0x2A:ud          {Compacted}                 // id:
0x20         mov     (8|M16)    r13.0<1>:ud    0x2A:ud                              // id:
0x30         mov     (8|M24)    r16.0<1>:ud    0x2A:ud                              // id:
0x40   (W)   mov     (8|M0)     r127.0<1>:ud   r0.0<8;8,1>:ud   {Compacted}                 // id:
0x48   (W)   mov     (8|M0)     r1.0<1>:uq     r3.0<0;1,0>:uq                       // id:
0x58   (W)   add     (8|M0)     r3.0<1>:q      r1.0<0;1,0>:q    0x1F:ud             // id:
0x68         mov     (8|M0)     r5.0<1>:uq     r3.0<0;1,0>:uq                       // id:
0x78         mov     (8|M8)     r8.0<1>:uq     r3.0<0;1,0>:uq                       // id:1
0x88         mov     (8|M16)    r11.0<1>:uq    r3.0<0;1,0>:uq                       // id:1
0x98         mov     (8|M24)    r14.0<1>:uq    r3.0<0;1,0>:uq                       // id:1
0xa8         send    (8|M0)     null           r5               0xC    0x6060680FF           // id:1
0xb8         send    (8|M8)     null           r8               0xC    0x6060680FF           // id:1
0xc8         send    (8|M16)    null           r11              0xC    0x6060680FF           // id:1
0xd8         send    (8|M24)    null           r14              0xC    0x6060680FF           // id:1
0xe8         send    (8|M0)     null           r127             0x27   0x2000010 {EOT}         // id:1
```

Figure 7. Body of meaning_of_life kernel

And the end of the kernel:



C:\Users\Robert\Do...eaning_of_life.meta | C:\Users\Robert\D...rogram_empty.meta | program.cl

```
kernel void empty() {
}

kernel void meaning_of_life(global uchar* out)
{
    out[31] = 42;
}
```

```
0x0          LABEL0
0x8    (W)   and     (1|M0)     r2.6<1>:ud     r0.5<0;1,0>:ud   0x1FF:ud            // id:
0x10         mov     (8|M0)     r7.0<1>:ud     0x2A:ud          {Compacted}                 // id:
0x18         mov     (8|M8)     r10.0<1>:ud    0x2A:ud          {Compacted}                 // id:
0x20         mov     (8|M16)    r13.0<1>:ud    0x2A:ud                              // id:
0x30         mov     (8|M24)    r16.0<1>:ud    0x2A:ud                              // id:
0x40   (W)   mov     (8|M0)     r127.0<1>:ud   r0.0<8;8,1>:ud   {Compacted}                 // id:
0x48   (W)   mov     (8|M0)     r1.0<1>:uq     r3.0<0;1,0>:uq                       // id:
0x58   (W)   add     (8|M0)     r3.0<1>:q      r1.0<0;1,0>:q    0x1F:ud             // id:
0x68         mov     (8|M0)     r5.0<1>:uq     r3.0<0;1,0>:uq                       // id:
0x78         mov     (8|M8)     r8.0<1>:uq     r3.0<0;1,0>:uq                       // id:1
0x88         mov     (8|M16)    r11.0<1>:uq    r3.0<0;1,0>:uq                       // id:1
0x98         mov     (8|M24)    r14.0<1>:uq    r3.0<0;1,0>:uq                       // id:1
0xa8         send    (8|M0)     null           r5               0xC    0x6060680FF           // id:1
0xb8         send    (8|M8)     null           r8               0xC    0x6060680FF           // id:1
0xc8         send    (8|M16)    null           r11              0xC    0x6060680FF           // id:1
0xd8         send    (8|M24)    null           r14              0xC    0x6060680FF           // id:1
0xe8         send    (8|M0)     null           r127             0x27   0x2000010 {EOT}         // id:1
```

Figure 8. Ending of meaning_of_life kernel.

We are going to rearrange the assembly to make it a little bit more understandable:

```
// Start of Thread
LABEL0
(W)     and     (1|M0)        r2.6<1>:ud    r0.5<0;1,0>:ud    0x1FF:ud         // id:
// r3 and r4 contain the address of out variable (8 unsigned quadwords – uq)
// we are going to place them in r1 and r2
(W)     mov     (8|M0)        r1.0<1>:uq    r3.0<0;1,0>:uq                   // id:


// Move 42 (0x2A:ud – ud is unsigned dword) into 32 slots (our kernel is compiled SIMD32)
// We are going to use registers r7, r10, r13 and r16, each register fitting 8 values
        mov     (8|M0)        r7.0<1>:ud    0x2A:ud          {Compacted}             // id:
        mov     (8|M8)        r10.0<1>:ud   0x2A:ud          {Compacted}             // id:
        mov     (8|M16)       r13.0<1>:ud   0x2A:ud                          // id:
        mov     (8|M24)       r16.0<1>:ud   0x2A:ud                          // id:

// Add 31 (0x1F:ud) to eight quadwords in r1 and r2 and place the results in r3 and r4
// Essentially, we get &out[31]
 (W)      add      (8|M0)        r3.0<1>:q     r1.0<0;1,0>:q     0x1F:ud          // id:

// Now we spread &out[31] into r5,r6, r8,r9, r11, r12, and r14, r15 – 32 values in all.
        mov      (8|M0)        r5.0<1>:uq    r3.0<0;1,0>:uq                     // id:
        mov      (8|M8)        r8.0<1>:uq    r3.0<0;1,0>:uq                     // id:1
        mov      (8|M16)       r11.0<1>:uq   r3.0<0;1,0>:uq                     // id:1
        mov      (8|M24)       r14.0<1>:uq   r3.0<0;1,0>:uq                     // id:1

// Write to values in r7 into addresses in r5, r6, etc.
        send     (8|M0)        null          r5                0xC       0x60680FF
        send     (8|M8)        null          r8                0xC       0x60680FF
        send     (8|M16)       null          r11               0xC       0x60680FF
        send     (8|M24)       null          r14               0xC       0x60680FF

// End of thread
(W)      mov      (8|M0)        r127.0<1>:ud  r0.0<8;8,1>:ud   {Compacted}                // id:
        send     (8|M0)        null          r127              0x27      0x2000010 {EOT}
```

Figure 9. Annotated assembly of meaning_of_life kernel.

Now, we are going to complicate life ever so slightly, by using get_global_id(0) instead of a fixed index to write things out

```
kernel void meaning_of_life2(global uchar* out)
{
 int i = get_global_id(0);
 out[i] = 42;
}
```

Figure 10. meaning_of_life2 kernel.

Note, that the addition of get_global_id(0) increases the size of our kernel by 9 assembly instructions. This mainly has to do with the fact that we will need to calculate increasing addresses for each subsequent workitem in a thread (there 32 work items there):

```
// Start of Thread
LABEL0
(W)      and      (1|M0)        r7.6<1>:ud    r0.5<0;1,0>:ud    0x1FF:ud          // id:

// Move 42 (0x2A:ud – ud is unsigned dword) into 32 slots (our kernel is compiled SIMD32)
// We are going to use registers r17, r20, r23 and r26, each register fitting 8 values
         mov      (8|M0)        r17.0<1>:ud   0x2A:ud          {Compacted}                // id:
         mov      (8|M8)        r20.0<1>:ud   0x2A:ud          {Compacted}                // id:
         mov      (8|M16)       r23.0<1>:ud   0x2A:ud                            // id:
         mov      (8|M24)       r26.0<1>:ud   0x2A:ud                            // id:
// get_global_id(0) calculation, r0.1, r7.0 and r7.3 will contain the necessary starting values
(W)      mul      (1|M0)        r3.0<1>:ud    r0.1<0;1,0>:ud    r7.3<0;1,0>:ud    // id:
(W)      mul      (1|M0)        r5.0<1>:ud    r0.1<0;1,0>:ud    r7.3<0;1,0>:ud    // id:
(W)      add      (1|M0)        r3.0<1>:ud    r3.0<0;1,0>:ud    r7.0<0;1,0>:ud    {Compacted} // id:
(W)      add      (1|M0)        r5.0<1>:ud    r5.0<0;1,0>:ud    r7.0<0;1,0>:ud    {Compacted} // id:
// r3 thru r6 will contain the get_global_id(0) offsets; r1 and r2 contain 32 increasing values
         add      (16|M0)       r3.0<1>:ud    r3.0<0;1,0>:ud    r1.0<8;8,1>:uw    // id:1
         add      (16|M16)      r5.0<1>:ud    r5.0<0;1,0>:ud    r2.0<8;8,1>:uw    // id:1
// r8 and r9 contain the address of out variable (8 unsigned quadwords – uq)
// we are going to place these addresses in r1 and r2
 (W)      mov      (8|M0)        r1.0<1>:uq    r8.0<0;1,0>:uq                     // id:1

// Move the offsets in r3 thru r6 to r7, r8, r9, r10, r11, r12, r13, r14
         mov      (8|M0)        r7.0<1>:q     r3.0<8;8,1>:d                      // id:1
         mov      (8|M8)        r9.0<1>:q     r4.0<8;8,1>:d                      // id:1
         mov      (8|M16)       r11.0<1>:q    r5.0<8;8,1>:d                      // id:1
         mov      (8|M24)       r13.0<1>:q    r6.0<8;8,1>:d                      // id:1

// Add the offsets to address of out in r1 and place them in r15, r16, r18, r19, r21, r22, r24, r25
         add      (8|M0)        r15.0<1>:q    r1.0<0;1,0>:q     r7.0<4;4,1>:q     // id:1
         add      (8|M8)        r18.0<1>:q    r1.0<0;1,0>:q     r9.0<4;4,1>:q     // id:1
         add      (8|M16)       r21.0<1>:q    r1.0<0;1,0>:q     r11.0<4;4,1>:q    // id:2
         add      (8|M24)       r24.0<1>:q    r1.0<0;1,0>:q     r13.0<4;4,1>:q    // id:2

// write into addresses in r15, r16, values in r17, etc.
         send     (8|M0)        null          r15               0xC       0x60680FF
         send     (8|M8)        null          r18                0xC       0x60680FF
         send     (8|M16)       null          r21                0xC       0x60680FF
         send     (8|M24)       null          r24                0xC       0x60680FF

// End of thread
(W)      mov      (8|M0)        r127.0<1>:ud  r0.0<8;8,1>:ud    {Compacted}                  // id:
         send     (8|M0)        null          r127               0x27      0x2000010 {EOT}
```

Figure 11. Annotated assembly of meaning_of_life2 kernel.

And finally, let's look at a kernel that does, reading, writing and some math:

```
kernel void modulate(global float* in, global float* out) {
 int i = get_global_id(0);

 float f = in[i];
 float temp = 0.5f * f;
 out[i] = temp;
}
```

Figure 12. A simple kernel that does floating point math.

It will be translated to the following (note, that I rearranged some assembly instructions for better understanding):

```
// Start of Thread
LABEL0
(W)      and      (1|M0)        r7.6<1>:ud    r0.5<0;1,0>:ud    0x1FF:ud          // id:

// r3 and r4 will contain the address of out buffer
(W)      mov      (8|M0)        r3.0<1>:uq    r8.1<0;1,0>:uq                      // id:
// int i = get_global_id(0);
(W)      mul      (1|M0)        r5.0<1>:ud    r0.1<0;1,0>:ud    r7.3<0;1,0>:ud    // id:
(W)      mul      (1|M0)        r9.0<1>:ud    r0.1<0;1,0>:ud    r7.3<0;1,0>:ud    // id:
(W)      add      (1|M0)        r5.0<1>:ud    r5.0<0;1,0>:ud    r7.0<0;1,0>:ud    {Compacted} // id:
(W)      add      (1|M0)        r9.0<1>:ud    r9.0<0;1,0>:ud    r7.0<0;1,0>:ud    {Compacted} // id:
         add      (16|M0)       r5.0<1>:ud    r5.0<0;1,0>:ud    r1.0<8;8,1>:uw    // id:
         add      (16|M16)      r9.0<1>:ud    r9.0<0;1,0>:ud    r2.0<8;8,1>:uw    // id:

// r1 and r2 will contain the address of in buffer
(W)      mov      (8|M0)        r1.0<1>:uq    r8.0<0;1,0>:uq                    // id:1
// r11, r12, r13, r14, r15, r16, r17 and r18 will contain 32 qword offsets
         mov      (8|M0)        r11.0<1>:q    r5.0<8;8,1>:d                     // id:1
         mov      (8|M8)        r13.0<1>:q    r6.0<8;8,1>:d                     // id:1
         mov      (8|M16)       r15.0<1>:q    r9.0<8;8,1>:d                     // id:1
         mov      (8|M24)       r17.0<1>:q    r10.0<8;8,1>:d                    // id:1

//  float f = in[i];
         shl      (8|M0)        r31.0<1>:uq   r11.0<4;4,1>:uq   0x2:ud            // id:1
         shl      (8|M8)        r33.0<1>:uq   r13.0<4;4,1>:uq   0x2:ud            // id:1
         shl      (8|M16)       r35.0<1>:uq   r15.0<4;4,1>:uq   0x2:ud            // id:1
         shl      (8|M24)       r37.0<1>:uq   r17.0<4;4,1>:uq   0x2:ud            // id:1
         add      (8|M0)        r19.0<1>:q    r1.0<0;1,0>:q     r31.0<4;4,1>:q    // id:1
         add      (8|M8)        r21.0<1>:q    r1.0<0;1,0>:q     r33.0<4;4,1>:q    // id:2
         add      (8|M16)       r23.0<1>:q    r1.0<0;1,0>:q     r35.0<4;4,1>:q    // id:2
         add      (8|M24)       r25.0<1>:q    r1.0<0;1,0>:q     r37.0<4;4,1>:q    // id:2
// read in f values at addresses in r19, r20, r21, r22, r23, r24, r25, r26 into r27, r28, r29, r30
         send     (8|M0)        r27           r19               0xC       0x4146EFF
         send     (8|M8)        r28           r21               0xC       0x4146EFF
         send     (8|M16)       r29           r23               0xC       0x4146EFF
         send     (8|M24)       r30           r25               0xC       0x4146EFF

// float temp = 0.5f * f; - 0.5f is 0x3F000000:f
//     We multiply 16 values in r27, r28 by 0.5f and place them in r39, r40
//     We multiple 16 values in r29, r30 by 0.5f and place them in r47, r48
         mul      (16|M0)       r39.0<1>:f    r27.0<8;8,1>:f    0x3F000000:f      // id:3
         mul      (16|M16)      r47.0<1>:f    r29.0<8;8,1>:f    0x3F000000:f      // id:3

//     out[i] = temp;
         add      (8|M0)        r41.0<1>:q    r3.0<0;1,0>:q     r31.0<4;4,1>:q    // id:2
         add      (8|M8)        r44.0<1>:q    r3.0<0;1,0>:q     r33.0<4;4,1>:q    // id:2
         add      (8|M16)       r49.0<1>:q    r3.0<0;1,0>:q     r35.0<4;4,1>:q    // id:2
         add       (8|M24)       r52.0<1>:q     r3.0<0;1,0>:q     r37.0<4;4,1>:q   // id:3

         mov       (8|M0)        r43.0<1>:ud    r39.0<8;8,1>:ud  {Compacted}                // id:3
         mov       (8|M8)        r46.0<1>:ud    r40.0<8;8,1>:ud  {Compacted}                // id:3
         mov       (8|M16)       r51.0<1>:ud    r47.0<8;8,1>:ud                       // id:3
         mov       (8|M24)       r54.0<1>:ud    r48.0<8;8,1>:ud                       // id:3

// write into addresses r41, r42 the values in r43, etc.
         send      (8|M0)        null           r41               0xC       0x6066EFF
         send      (8|M8)        null           r44               0xC       0x6066EFF
         send      (8|M16)       null           r49               0xC       0x6066EFF
         send      (8|M24)       null           r52               0xC       0x6066EFF

// End of thread
(W)      mov       (8|M0)        r127.0<1>:ud   r0.0<8;8,1>:ud   {Compacted}                // id:
         send      (8|M0)        null           r127              0x27      0x2000010 {EOT}
```

Figure 13. Annotated assembly of a simple floating point math kernel.

How to Read an Assembly Instruction

Typically, all instructions have the following form:

```
[(pred)] opcode (exec-size|exec-offset) dst src0 [src1] [src2]
```

(pred) is the optional predicate. We are going to skip it for now.

opcode is the symbol of the instruction, like add or mov (we have a full table of opcodes below.

exec-size is the SIMD width of the instruction, which of our architecture could be 1, 2, 4, 8, or 16. In SIMD32 compilation, typically two instructions of execution size 8 or 16 are grouped into one.

exec-offset is the part that's telling the EU, which part of the ARF registers to read or write from, e.g. (8|M24) consults the bits 24-31 of the execution mask. When emitting SIMD16 or SIMD32 code like the following:

```
mov   (8|M0)    r11.0<1>:q   r5.0<8;8,1>:d    // id:1
mov   (8|M8)    r13.0<1>:q   r6.0<8;8,1>:d    // id:1
mov   (8|M16)   r15.0<1>:q   r9.0<8;8,1>:d    // id:1
mov   (8|M24)   r17.0<1>:q   r10.0<8;8,1>:d   // id:1
```

Figure 14. mov instructions of SIMD32 assembly.

the compiler has to emit four 8-wide operations due to a limitation of how many bytes can be accessed per operand in the GRF.

dst is a destination register

src0 is a source register

src1 is an optional source register. Note, that it could also be an immediate value, like 0x3F000000:f (0.5) or 0x2A:ud (42).

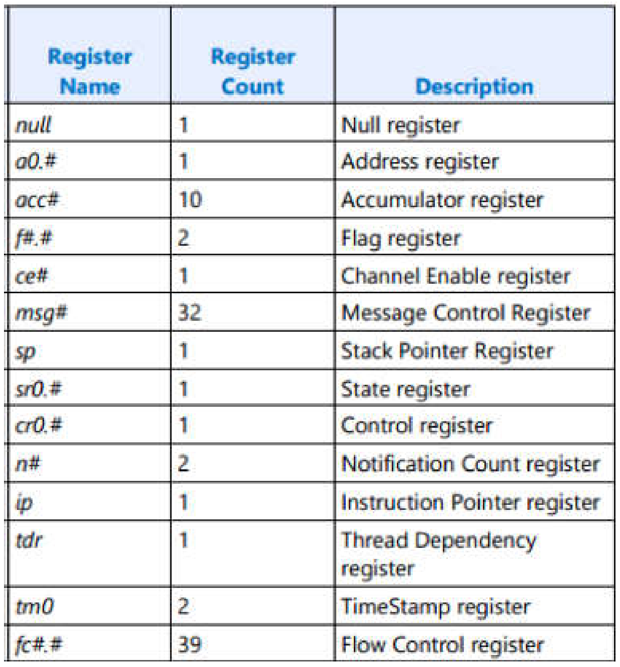
src2 is an optional source register.

General Register File (GRF) Registers

Each thread has a dedicated space of 128 registers, r0 through r127. Each register is 256 bits or 32 bytes.

Architecture Register File (ARF) Registers

In the assembly code above, we only saw one of these special registers, the null register, which is typically used as a destination for send instructions used for writing and indicating end of thread. Here is a full table of other architecture registers:



Register Name	Register Count	Description
null	1	Null register
a0.#	1	Address register
acc#	10	Accumulator register
f#.#	2	Flag register
ce#	1	Channel Enable register
msg#	32	Message Control Register
sp	1	Stack Pointer Register
sr0.#	1	State register
cr0.#	1	Control register
n#	2	Notification Count register
ip	1	Instruction Pointer register
tdr	1	Thread Dependency register
tm0	2	TimeStamp register
fc#.#	39	Flow Control register

Figure 15. Architecture Register File (ARF) Registers.

Since our registers are 32 bytes wide and are byte addressable, our assembly has a register region syntax, to be able to access values stored in these registers.

Below, we have a series of diagrams explaining how register region syntax works.

Here we have a register region r4.1<16;8,2>:w. The w at the end of the region indicates that we are talking about word (or two bytes) values. The full table of allowable integer and floating datatypes is below. The origin is at r4.1, which means that we are starting with the second word of register r4. The vertical stride is 16, which means that we need to skip 16 elements to start the second row. Width parameter is 8 and refers to the number of elements in a row; Horizontal stride of 2 means that we are taking every second element. Note, that we refer here to the content of both r4 and r5. The picture below summarizes the result:

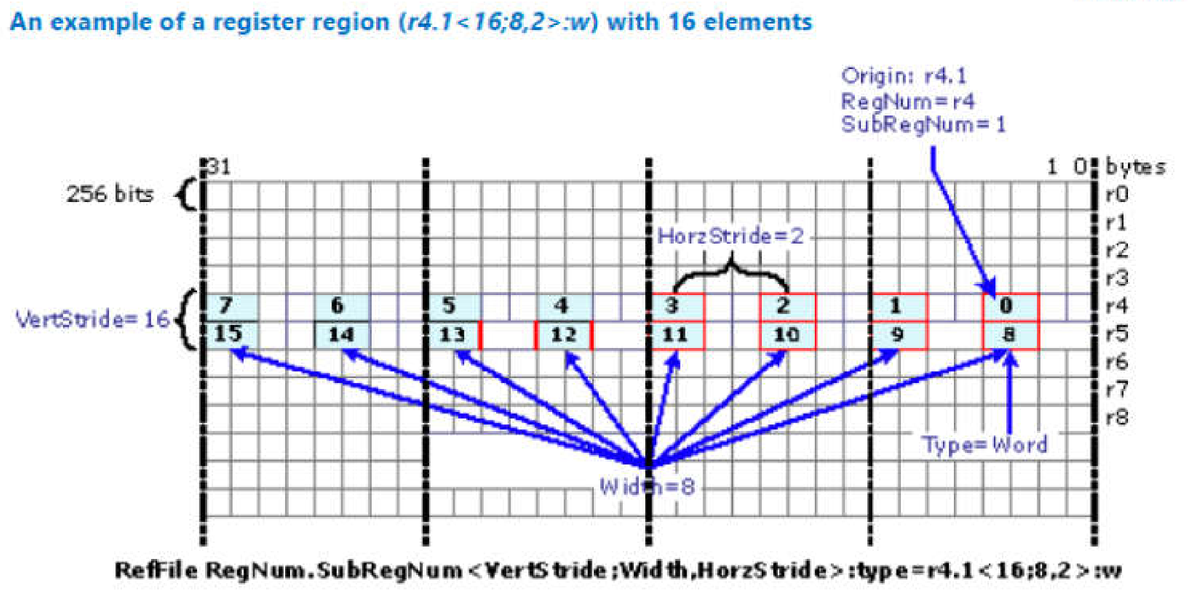


Figure 16. An example of a register region (r4.1<16;8,2>:w> with 16 elements.

In this example, let's consider a register region r5.0<1;8,2>:w. The region starts at a first element of r5. We have 8 elements in a row, row containing every second element, so the first row is {0, 1, 2, 3, 4, 5, 6, 7}. The second row starts at offset of 1 word, or at r5.2 and so it contains {8, 9, 10, 11, 12, 13, 14, 15}. The picture below summarizes the result:

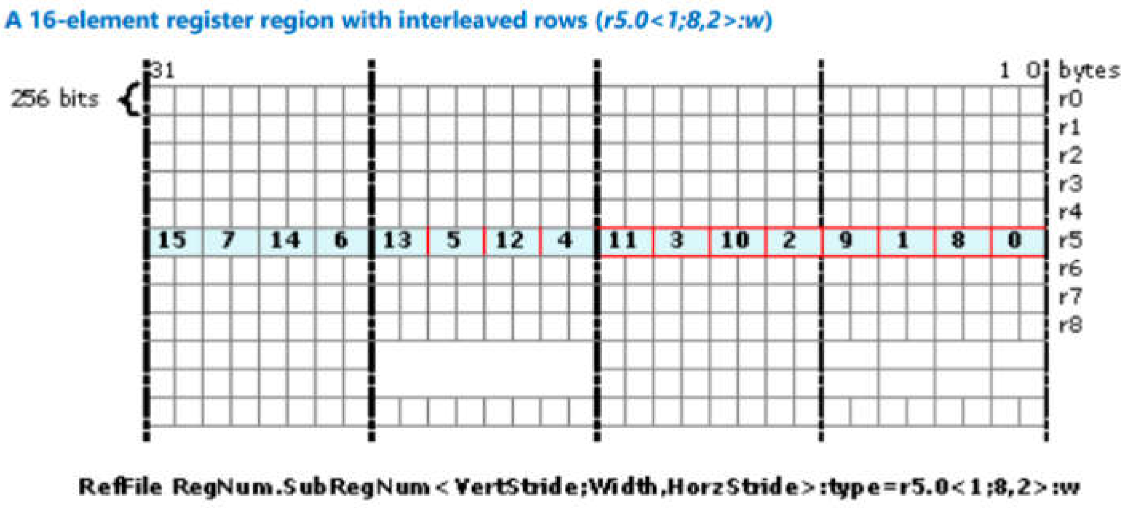

A 16-element register region with interleaved rows (r5.0<1;8,2>:w)
256 bits
31
1 0
bytes
r0
r1
r2
r3
r4
r5
r6
r7
r8
15 7 14 6 13 5 12 4 11 3 10 2 9 1 8 0
RefFile RegNum.SubRegNum<VertStride;Width,HorzStride>:type=r5.0<1;8,2>:w

Figure 17. A 16-element register region with interleaved rows (r5.0<1;8,2>:w).

Consider the following assembly instruction

```
add(16|M0) r6.0<1>:w r1.7<16;8,1>:b r2.1<16;8,1>:b
```

The src0 starts at r1.7 and has 8 consecutive bytes in the first row, followed by the second row of 8 bytes, which starts at r1.23.

The src1 starts at r2.1 and has 8 consecutive bytes in the first row, followed by the second row of 8 bytes, which starts at r2.17.

The dst starts at r6.0, stores the values as words, and since the instruction Add(16) will operate on 16 values, stores 16 consecutive words into r6.

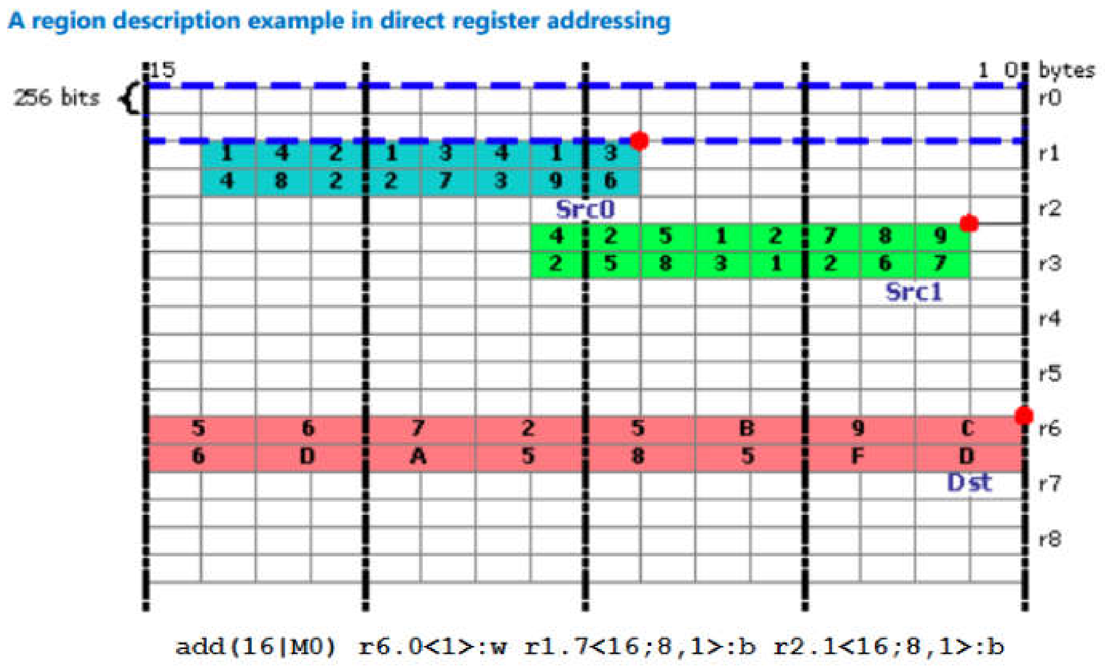

A region description example in direct register addressing
256 bits
15
1 0
bytes
r0
r1
r2
r3
r4
r5
r6
r7
r8
Src0
Src1
Dst
add(16|M0) r6.0<1>:w r1.7<16;8,1>:b r2.1<16;8,1>:b

Figure 18. A region description example in direct register addressing.

Let’s consider the following assembly instruction:

```
add(16|M0) r6.0<1>:w r1.14<16;8,0>:b r2.17<16;8,1>:b
```

Src0 is r1.14<16;8,0>:b, which means the we have the first byte sized value at r1.14, 0 in the stride value means that we are going to repeat the value for the width of the region, which is 8, and the region continues at r1.30, and we are going to repeat the value stored there 8 times as well, so we are talking about the following value {1,1,1,1,1,1,1,1, 4, 4, 4, 4, 4, 4, 4, 4}.

Src1 is r2.17<16;8,1>:b, so we actually start with 8 bytes starting from r2.17 and end up with the second row of 8 bytes starting from r3.1.

A region description example in direct register addressing with src0 as a vector of replicated scalars

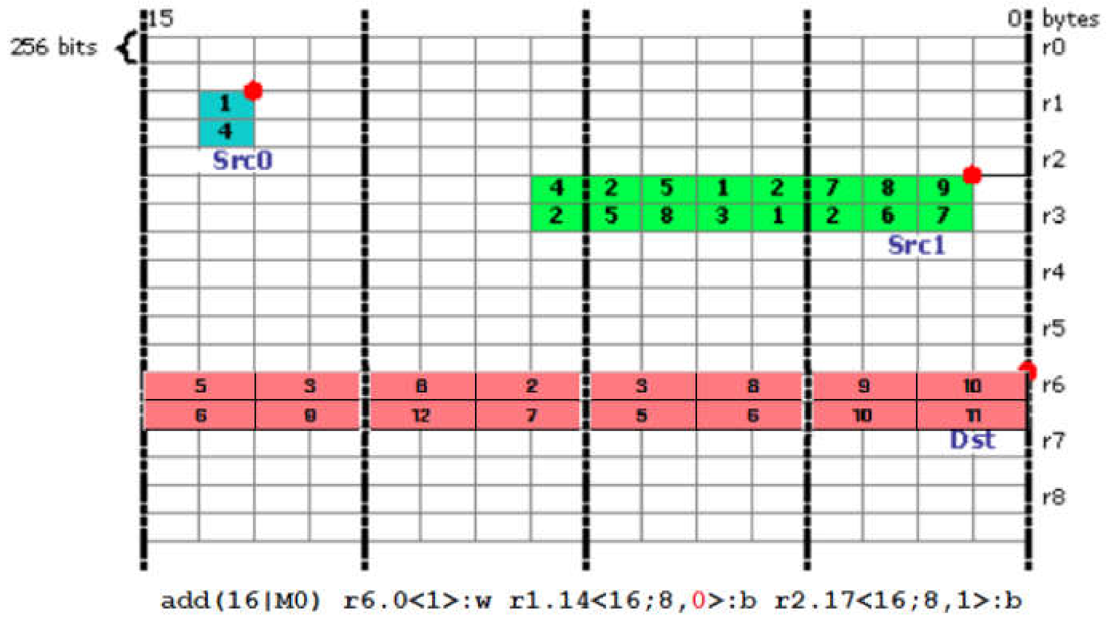
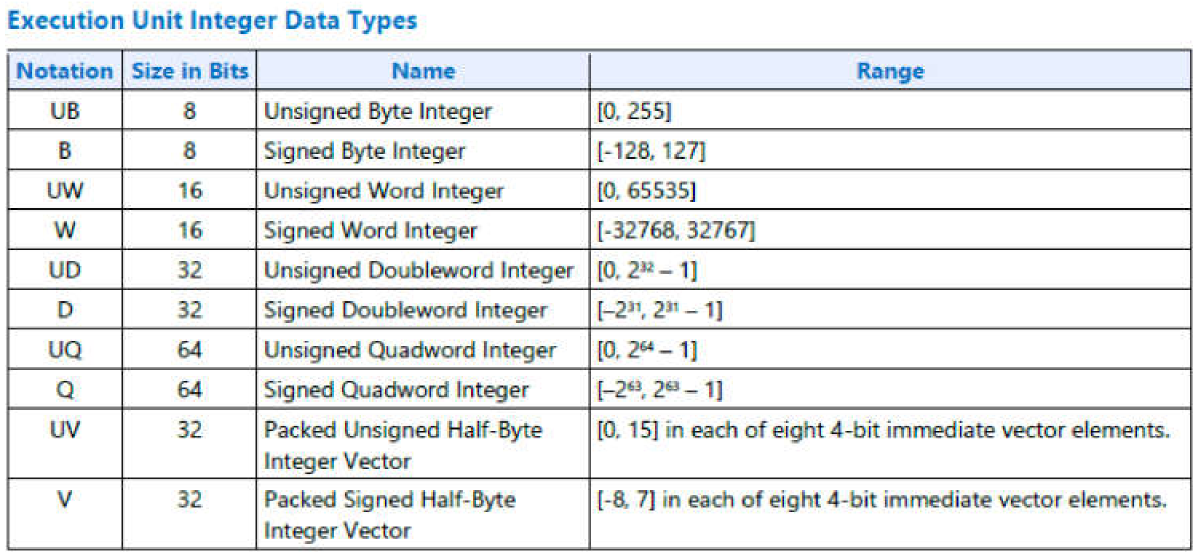


Figure 19. A region description example in direct register addressing with src0 as a vector of replicated scalars.

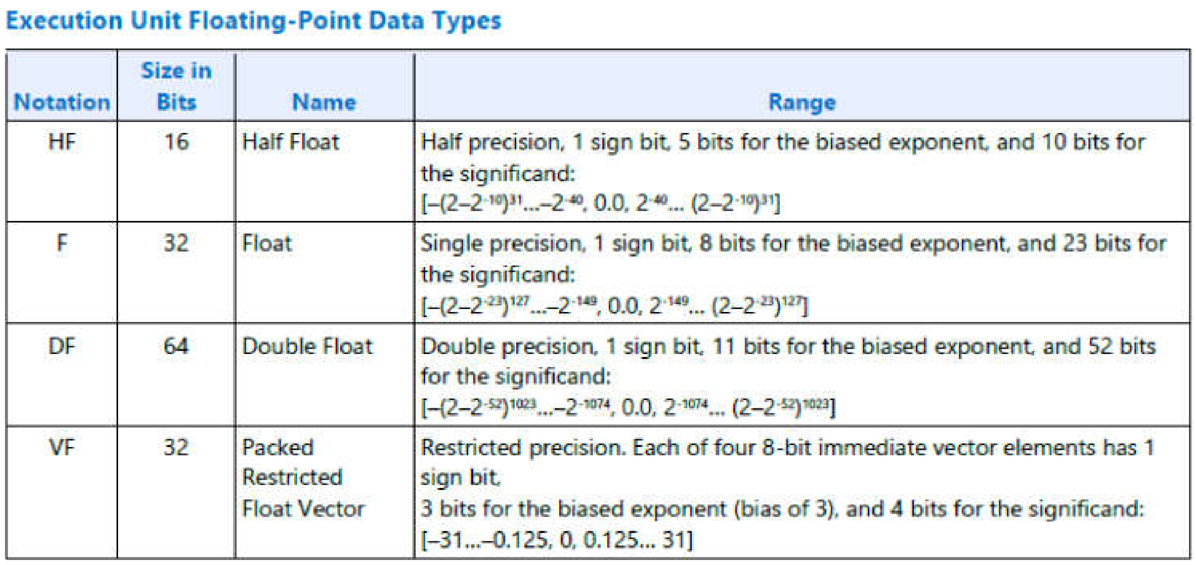
The letter after : in the register region signifies the data type stored there. Here are two tables summarizing the available integer and floating point types:



Execution Unit Integer Data Types

Notation	Size in Bits	Name	Range
UB	8	Unsigned Byte Integer	[0, 255]
B	8	Signed Byte Integer	[-128, 127]
UW	16	Unsigned Word Integer	[0, 65535]
W	16	Signed Word Integer	[-32768, 32767]
UD	32	Unsigned Doubleword Integer	$[0, 2^{32} – 1]$
D	32	Signed Doubleword Integer	$[–2^{31}, 2^{31} – 1]$
UQ	64	Unsigned Quadword Integer	$[0, 2^{64} – 1]$
Q	64	Signed Quadword Integer	$[–2^{63}, 2^{63} – 1]$
UV	32	Packed Unsigned Half-Byte Integer Vector	[0, 15] in each of eight 4-bit immediate vector elements.
V	32	Packed Signed Half-Byte Integer Vector	[-8, 7] in each of eight 4-bit immediate vector elements.

Figure 20. Execution Unit Integer Data Types.

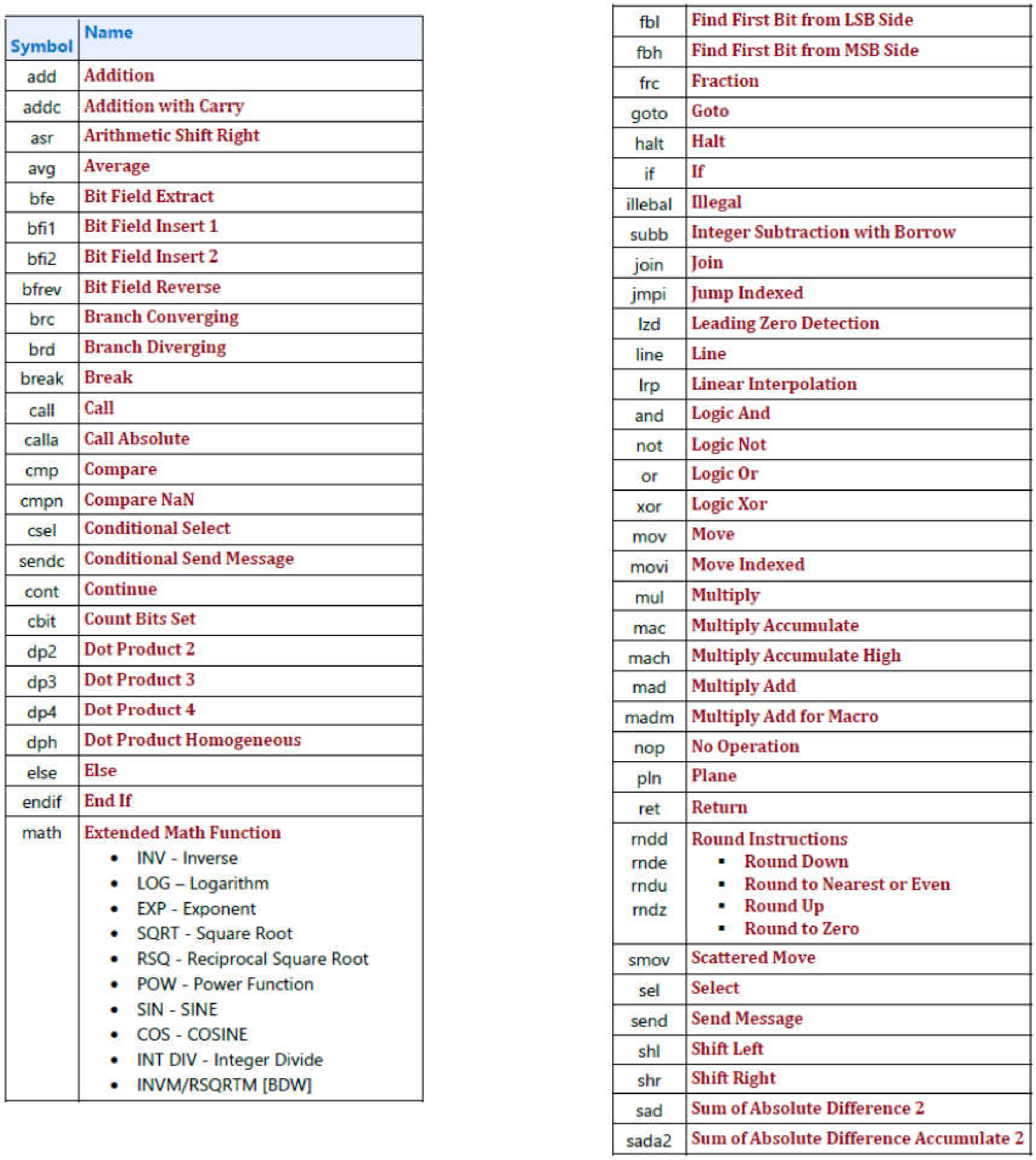


Execution Unit Floating-Point Data Types

Notation	Size in Bits	Name	Range
HF	16	Half Float	Half precision, 1 sign bit, 5 bits for the biased exponent, and 10 bits for the significand: $[–(2–2^{-10})^{31}…–2^{-40}, 0.0, 2^{-40}… (2–2^{-10})^{31}]$
F	32	Float	Single precision, 1 sign bit, 8 bits for the biased exponent, and 23 bits for the significand: $[–(2–2^{-23})^{127}…–2^{-149}, 0.0, 2^{-149}… (2–2^{-23})^{127}]$
DF	64	Double Float	Double precision, 1 sign bit, 11 bits for the biased exponent, and 52 bits for the significand: $[–(2–2^{-52})^{1023}…–2^{-1074}, 0.0, 2^{-1074}… (2–2^{-52})^{1023}]$
VF	32	Packed Restricted Float Vector	Restricted precision. Each of four 8-bit immediate vector elements has 1 sign bit, 3 bits for the biased exponent (bias of 3), and 4 bits for the significand: [–31…–0.125, 0, 0.125… 31]

Figure 21. Execution Unit Floating-Point Data Types.

The following tables summarize available assembly instructions:



Symbol	Name
add	Addition
addc	Addition with Carry
asr	Arithmetic Shift Right
avg	Average
bfe	Bit Field Extract
bfi1	Bit Field Insert 1
bfi2	Bit Field Insert 2
bfrev	Bit Field Reverse
brc	Branch Converging
brd	Branch Diverging
break	Break
call	Call
calla	Call Absolute
cmp	Compare
cmpn	Compare NaN
csel	Conditional Select
sendc	Conditional Send Message
cont	Continue
cbit	Count Bits Set
dp2	Dot Product 2
dp3	Dot Product 3
dp4	Dot Product 4
dph	Dot Product Homogeneous
else	Else
endif	End If
math	Extended Math Function • INV - Inverse • LOG – Logarithm • EXP - Exponent • SQRT - Square Root • RSQ - Reciprocal Square Root • POW - Power Function • SIN - SINE • COS - COSINE • INT DIV - Integer Divide • INVM/RSQRTM [BDW]
fbl	Find First Bit from LSB Side
fbh	Find First Bit from MSB Side
frc	Fraction
goto	Goto
halt	Halt
if	If
illebal	Illegal
subb	Integer Subtraction with Borrow
join	Join
jmpi	Jump Indexed
lzd	Leading Zero Detection
line	Line
lrp	Linear Interpolation
and	Logic And
not	Logic Not
or	Logic Or
xor	Logic Xor
mov	Move
movi	Move Indexed
mul	Multiply
mac	Multiply Accumulate
mach	Multiply Accumulate High
mad	Multiply Add
madm	Multiply Add for Macro
nop	No Operation
pln	Plane
ret	Return
rndd rnde rndu rndz	Round Instructions ▪ Round Down ▪ Round to Nearest or Even ▪ Round Up ▪ Round to Zero
smov	Scattered Move
sel	Select
send	Send Message
shl	Shift Left
shr	Shift Right
sad	Sum of Absolute Difference 2
sada2	Sum of Absolute Difference Accumulate 2

Figure 22. Available GEN Assembly Instructions.

References:

Volume 7 of Intel Graphics documentation is available here:

- Volume 7: 3D-Media-GPGPU (https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-bdw-vol07-3d_media_gpgpu_3.pdf)

Full set of Intel Graphics Documentation is available here:

https://01.org/linuxgraphics/documentation/hardware-specification-prms (https://01.org/linuxgraphics/documentation/hardware-specification-prms)

About the Author

Robert Ioffe is a Technical Consulting Engineer at Intel's Software and Solutions Group. He is an expert in OpenCL programming and OpenCL workload optimization on Intel Iris and Intel Iris Pro Graphics with deep knowledge of Intel Graphics Hardware. He was heavily involved in Khronos standards work, focusing on prototyping the latest features and making sure they can run well on Intel architecture. Most recently he has been working on prototyping Nested Parallelism (enqueue_kernel functions) feature of OpenCL 2.0 and wrote a number of samples that demonstrate Nested Parallelism functionality, including GPU-Quicksort for OpenCL 2.0. He also recorded and released two Optimizing Simple OpenCL Kernels videos and is in the process of recording a third video on Nested Parallelism.

You might also be interested in the following:

GPU-Quicksort in OpenCL 2.0: Nested Parallelism and Work-Group Scan Functions (https://software.intel.com/en-us/articles/gpu-quicksort-in-opencl-20-using-nested-parallelism-and-work-group-scan-functions)

Sierpiński Carpet in OpenCL 2.0 (https://software.intel.com/en-us/articles/sierpinski-carpet-in-opencl-20)

Optimizing Simple OpenCL Kernels: Modulate Kernel Optimization (https://software.intel.com/en-us/videos/optimizing-simple-opencl-kernels-modulate-kernel-optimization)

Optimizing Simple OpenCL Kernels: Sobel Kernel Optimization (https://software.intel.com/en-us/videos/optimizing-simple-opencl-kernels-sobel-kernel-optimization)