# GEN アセンブリーの概要

この記事は、インテル® デベロッパー・ゾーンに公開されている「Introduction to GEN Assembly」の日本語参考訳です。

---

## 目次



## はじめに

OpenCL* カーネルの最適化とデバッグには、アセンブリー・コードが非常に役立つことがあります。この記事では、インテル® SDK for OpenCL* Applications で利用可能なツールについて説明します。個々のカーネル用にオフライン・コンパイラーにより生成されたアセンブリーの表示、OpenCL* C コードに対応するアセンブリー・コードの領域のハイライト、生成されたアセンブリーの異なる部分の高レベルの説明などを行います。レジスター領域の構文の概要と意味を説明した後、異なる種類のレジスター、利用可能なアセンブリー命令とこれらの命令で操作できるデータ型を紹介します。この記事をお読みになったら、あとは作業を開始するだけです。この記事の続編では、インテル® VTune™ Amplifier を使用したアセンブリーのプロファイルおよびアセンブリーのデバッグを説明する予定です。

## 単純な OpenCL* カーネルのアセンブリー

まず単純なカーネルで始めましょう。

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

**図 1.** 単純な OpenCL* カーネル

これは最も単純なカーネルと言えます。このカーネルを Code Builder Session Explorer でビルドしましょう。CODE-BUILDER/OpenCL Kernel Development/New Session を選択して新しいセッションを作成し、上記のカーネルを空の program.cl ファイルにコピーして、このファイルをビルドします。第 5 世代インテル® プロセッサー (Broadwell) または第 6 世代インテル® プロセッサー (Skylake) を利用していれば、program_empty.gen ファイルが生成されているはずです。このファイルをダブルクリックします。次のような画面が表示されます。

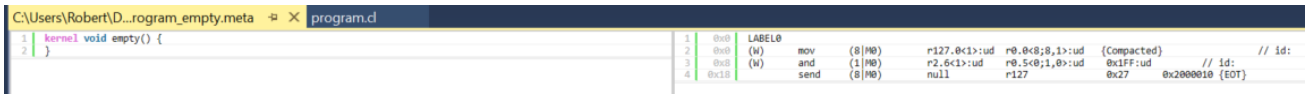


**図 2.** empty カーネルと対応するアセンブリー

カーネルのアセンブリーは右に表示されています。いくつか注釈を追加します。

```
// 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:
```

**図 3.** empty カーネルの注釈付きアセンブリー

ほとんど何も含まれていませんが、これはあくまでもスタートです。

では、少しだけ複雑にしてみましょう。次のコードを program.cl にコピーしてください。

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

**図 4.** meaning_of_life カーネル

ファイルをリビルドすると、program_meaning_of_life.gen ファイルが生成されます。このファイルをダブルクリックすると、少しだけ複雑になったカーネルが表示されます。

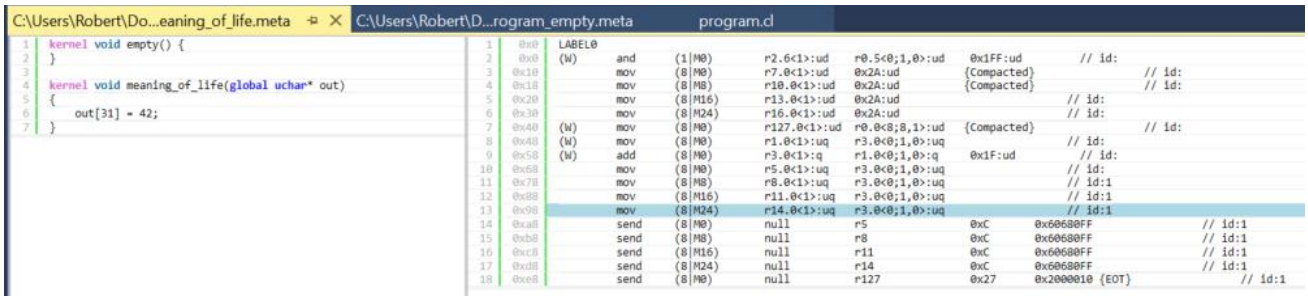


**図 5.** meaning_of_life カーネルと対応するアセンブリー

左のカーネルの異なる部分をクリックすると、アセンブリーの異なる部分がハイライトされることを確認します。

次はカーネルの最初と対応する命令です。

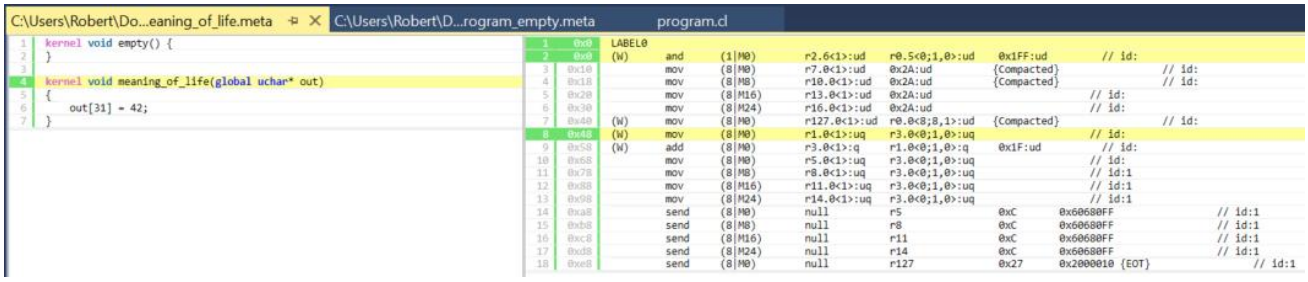


**図 6.** meaning_of_life カーネルの最初

カーネルの本体と対応する命令。

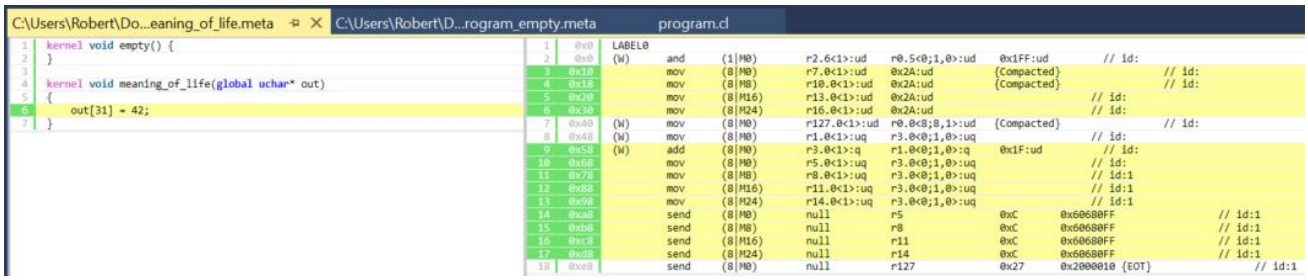


**図 7.** meaning_of_life カーネルの本体

カーネルの最後と対応する命令。

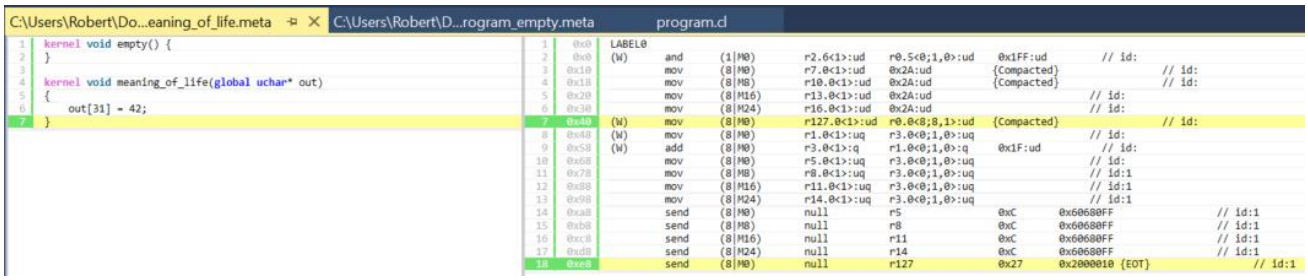


**図 8.** meaning_of_life カーネルの最後

分かりやすくなるように、アセンブリーを再配置しましょう。

```
// 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                  // id:1
         send     (8|M8)         null          r8                0xC
0x60680FF                  // id:1
         send     (8|M16)        null          r11               0xC
0x60680FF                  // id:1
         send     (8|M24)        null          r14               0xC
0x60680FF                  // id:1

// 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:1
```

**図 9.** meaning_of_life カーネルの注釈付きアセンブリー

固定インデックスの代わりに get_global_id(0) を使用して、カーネルをもう少し複雑にしてみましょう。

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

**図 10.** meaning_of_life2 カーネル

get_global_id(0) を追加したことでカーネルのサイズが増加している (アセンブリー命令が 9 つ増えている) ことに注意してください。スレッドの各ワークアイテムの増加アドレスを計算する必要があります (32 のワークアイテムがあります)。

```
// 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:1
// 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                 // id:2
         send     (8|M8)        null          r18               0xC
0x60680FF                 // id:2
         send     (8|M16)       null          r21               0xC
0x60680FF                 // id:2
         send     (8|M24)       null          r24               0xC
0x60680FF                 // id:2

// 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:2
```

**図 11.** meaning_of_life2 カーネルの注釈付きアセンブリー

最後に、読み書きと計算を行うカーネルを見てみましょう。

```
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;
}
```

**図 12.** 浮動小数点演算を行う単純なカーネル

コードは次のようになります (分かりやすくなるように一部のアセンブリー命令を再配置していることに注意してください)。

```
// 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                  // id:2
         send     (8|M8)        r28           r21               0xC
0x4146EFF                  // id:2
         send     (8|M16)       r29           r23               0xC
0x4146EFF                  // id:2
         send     (8|M24)       r30           r25               0xC
0x4146EFF                  // id:2

// 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                  // id:3
         send     (8|M8)        null          r44               0xC
0x6066EFF                  // id:3
         send     (8|M16)       null          r49               0xC
0x6066EFF                  // id:3
         send     (8|M24)       null          r52               0xC
0x6066EFF                  // id:4

// 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:4
```

**図 13.** 単純な浮動小数点演算カーネルの注釈付きアセンブリー

## アセンブリー命令の読み方

すべての命令は次のような形式になります。

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

**(pred)** は、オプションのプレディケートです。ここでは説明しません。

**opcode** は、add や mov のような命令のシンボルです (opcode のリストは最後の表を参照)。

**exec-size** は、命令の SIMD 幅です。アーキテクチャーに応じて 1、2、4、8 または 16 になります。SIMD32 コンパイルでは、通常、実行サイズ 8 または 16 の 2 つの命令が 1 つのグループになります。

**exec-offset** は、ARF レジスターのどの部分を読み書きするか、実行ユニットに知らせる部分です。例えば、(8|M24) は実行マスクのビット 24 から 31 を参照します。SIMD16 または SIMD32 のコードは次のようになります。

```
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
```

**図 14.** SIMD32 アセンブリーの mov 命令

GRF のオペランドごとにアクセスできるバイト数の制限により、コンパイラーは 4 つの 8 要素幅の演算を行う必要があります。

**dst** は、デスティネーション・レジスターです。

**src0** は、ソースレジスターです。

**src1** は、オプションのソースレジスターです。0x3F000000:f (0.5) や 0x2A:ud (42) のように、即値の場合もあることに注意してください。

**src2** は、オプションのソースレジスターです。

**ジェネラル・レジスター・ファイル (GRF) レジスター**

各スレッドには 128 個のレジスターの専用スペース (r0 から r127) があります。各レジスターは 256 ビットまたは 32 バイトです。

**アーキテクチャー・レジスター・ファイル (ARF) レジスター**

上記のアセンブリー・コードには、これらの特殊なレジスターの 1 つである null レジスターのみ含まれています。null レジスターは、send 命令のデスティネーションとして使用され、スレッドの最後を記述および示すために使用されます。その他のアーキテクチャー・レジスターを次に示します。

| 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 |

**図 15.** アーキテクチャー・レジスター・ファイル (ARF) レジスター

レジスターは 32 バイト幅でバイトアドレス指定可能であるため、アセンブリーにはこれらのレジスターに格納された値にアクセスできるレジスター領域構文が用意されています。

この後、一連の図でレジスター領域構文について説明します。

例として、レジスター領域 r4.1<16;8,2>:w を見てみましょう。領域の最後の w はワード (または 2 バイト) 値であることを示しています。利用可能な整数データ型と浮動小数点データ型の一覧は、後の表を参照してください。起点は r4.1 で、レジスター r4 の 2 つ目のワードから始まることを意味します。縦ストライドは 16 で、2 つ目の行から始まるため 16 の要素をスキップする必要があることを意味します。幅パラメーターは 8 で、行の要素数を指します。横ストライドは 2 で、2 つおきに要素を処理することを意味します。ここでは r4 と r5 の両方の内容を参照していることに注意してください。次の図は結果をまとめたものです。

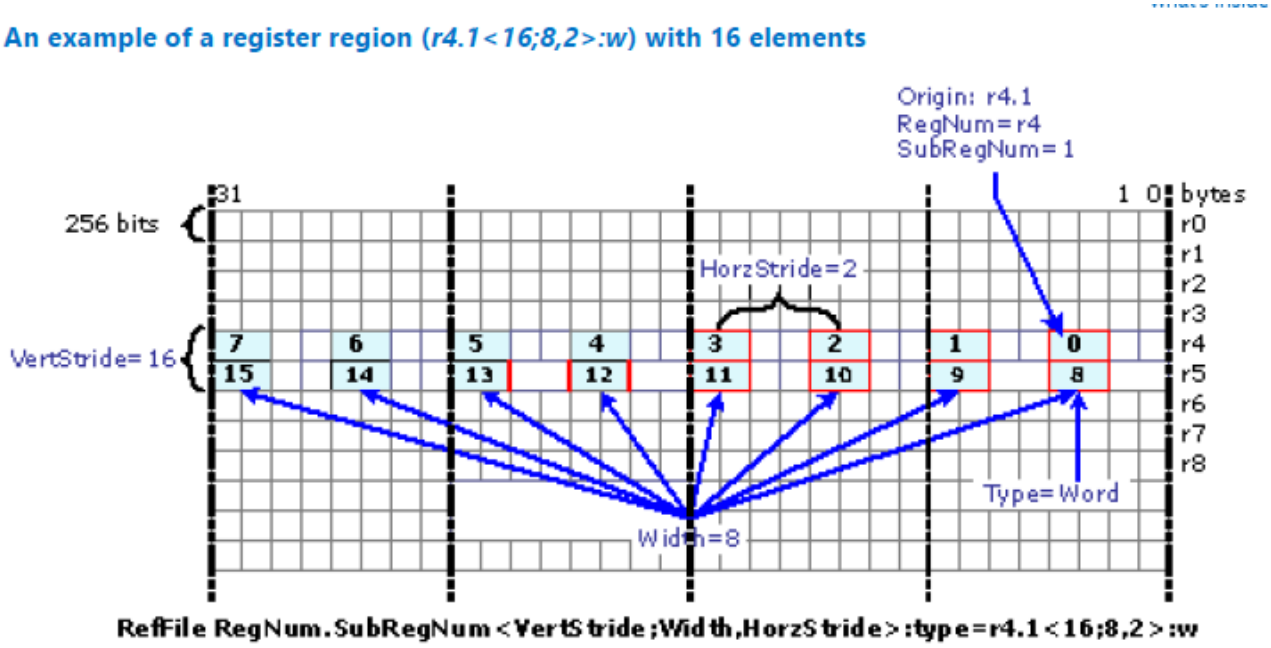


**図 16.** 16 要素のレジスター領域 (r4.1<16;8,2>:w) の例

この例の、レジスター領域 r5.0<1;8,2>:w について考えてみましょう。領域は r5 の最初の要素から始まります。1 行に 8 つの要素があり、2 つおきの要素を含みます。つまり、最初の行は {0, 1, 2, 3, 4, 5, 6, 7} です。2 つ目の行は 1 ワードのオフセットまたは r5.2 で始まり、{8, 9, 10, 11, 12, 13, 14, 15} を含みます。次の図は結果をまとめたものです。

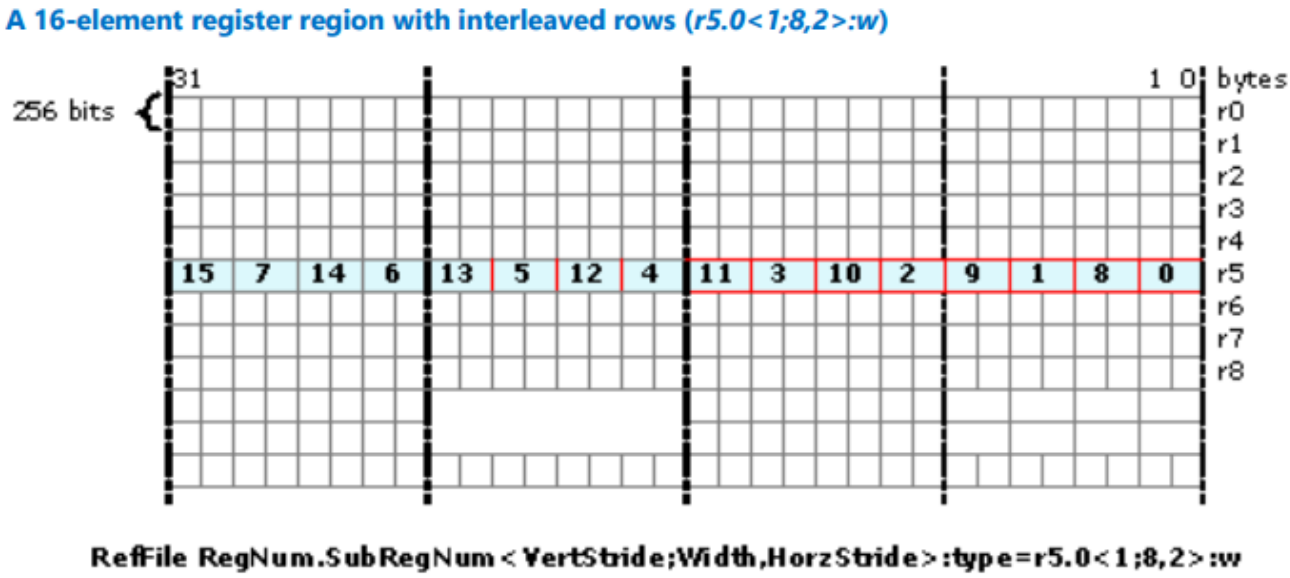


**図 17.** 行がインターリーブされた 16 要素のレジスター領域 (r5.0<1;8,2>:w)

次のアセンブリー命令について考えてみましょう。

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

src0 は、r1.7 から始まり 8 つの連続するバイトの最初の行、r1.23 から始まる 2 つ目の行が続きます。

src1 は、r2.1 から始まり 8 つの連続するバイトの最初の行、r2.17 から始まる 2 つ目の行が続きます。

dst は、r6.0 から始まり、値をワードで格納します。命令 Add(16) は 16 の値を操作するため、16 の連続するワードを r6 に格納します。

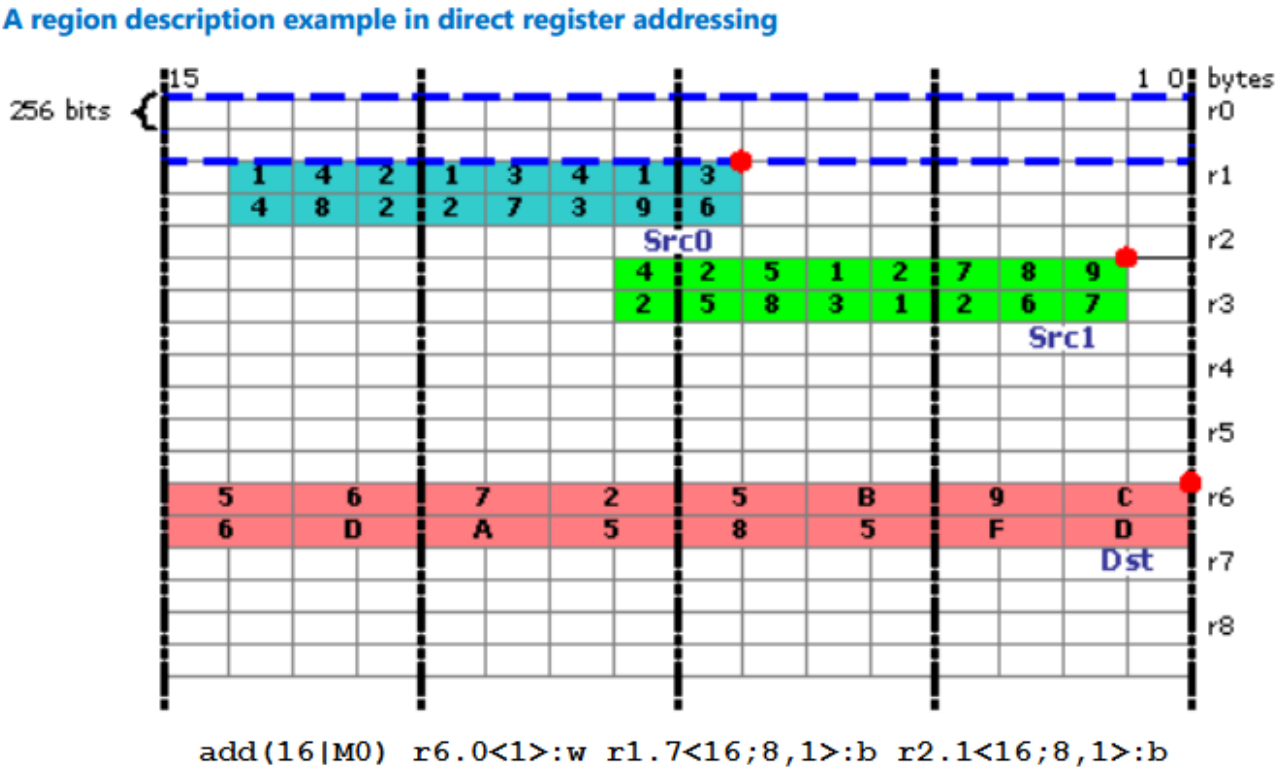


**図 18.** 直接レジスター・アドレッシングの領域説明の例

次のアセンブリー命令について考えてみましょう。

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

src0 は r1.14<16;8,0>:b です。最初のバイトサイズの値は r1.14 から始まります。ストライド値 0 は領域の幅 (8) で値を繰り返します。次に、r1.30 から始まる領域が続き、そこに 8 回格納された値を繰り返します。つまり、値 {1,1,1,1,1,1,1,1, 4, 4, 4, 4, 4, 4, 4, 4} を操作します。

src1 は r2.17<16;8,1>:b で、最初の行は r2.17 から始まる 8 バイト、2 つ目の行は r3.1 から始まる 8 バイトです。

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

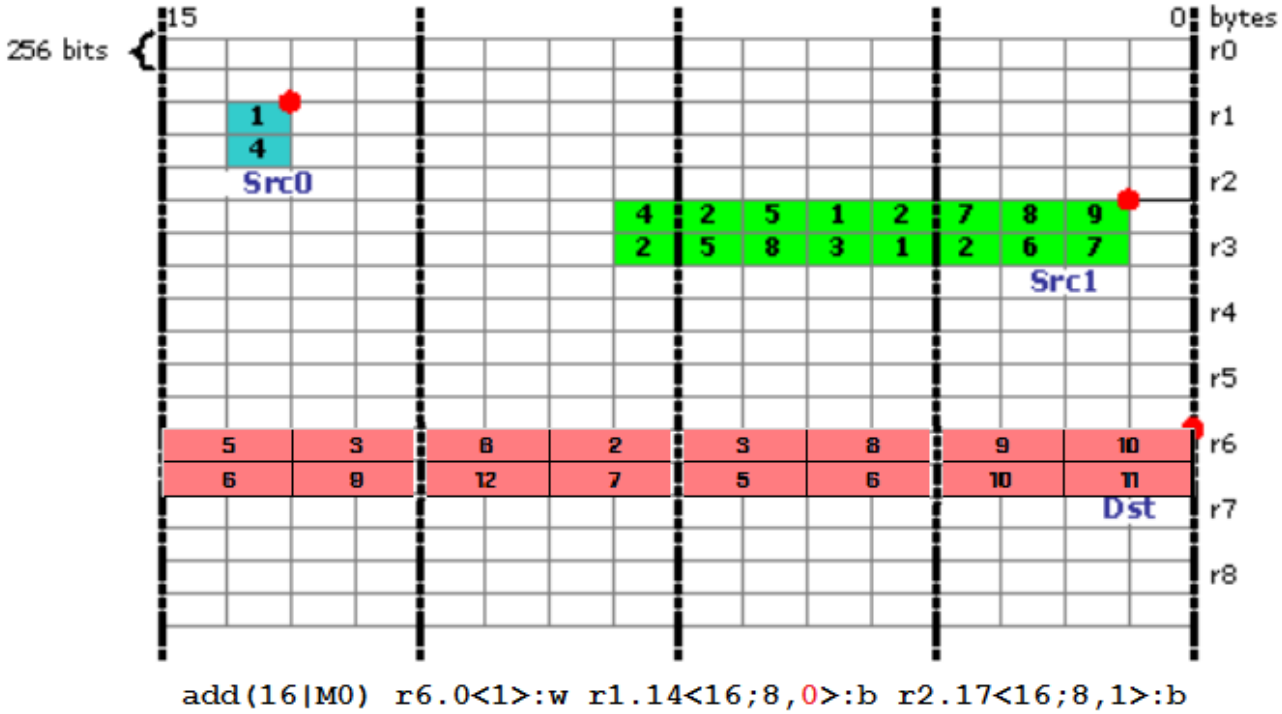


**図 19.** src0 を複製スカラーのベクトルとして含む直接レジスター・アドレッシングの領域説明の例

レジスター領域の：の後の文字は、格納されるデータ型を示します。利用可能な整数データ型と浮動小数点データ型の 2 つの表を次に示します。

**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. |

**図 20.** 実行ユニット (整数データ型)

**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:<br>$[-(2-2^{-10})^{31} \ldots -2^{-40}, 0.0, 2^{-40} \ldots (2-2^{-10})^{31}]$ |
| F | 32 | Float | Single precision, 1 sign bit, 8 bits for the biased exponent, and 23 bits for the significand:<br>$[-(2-2^{-23})^{127} \ldots -2^{-149}, 0.0, 2^{-149} \ldots (2-2^{-23})^{127}]$ |
| DF | 64 | Double Float | Double precision, 1 sign bit, 11 bits for the biased exponent, and 52 bits for the significand:<br>$[-(2-2^{-52})^{1023} \ldots -2^{-1074}, 0.0, 2^{-1074} \ldots (2-2^{-52})^{1023}]$ |
| VF | 32 | Packed Restricted Float Vector | Restricted precision. Each of four 8-bit immediate vector elements has 1 sign bit,<br>3 bits for the biased exponent (bias of 3), and 4 bits for the significand:<br>[–31...–0.125, 0, 0.125... 31] |

**図 21.** 実行ユニット (浮動小数点データ型)

次の表は、利用可能なアセンブリー命令をまとめたものです。

| 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**<br>• INV - Inverse<br>• LOG – Logarithm<br>• EXP - Exponent<br>• SQRT - Square Root<br>• RSQ - Reciprocal Square Root<br>• POW - Power Function<br>• SIN - SINE<br>• COS - COSINE<br>• INT DIV - Integer Divide<br>• 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<br>rnde<br>rndu<br>rndz | Round Instructions<br>▪ Round Down<br>▪ Round to Nearest or Even<br>▪ Round Up<br>▪ 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 |

**図 22.** 利用可能な GEN アセンブリー命令

## 参考文献 (英語)


インテル® グラフィックス・ドキュメントの第 7 巻:

- Volume 7: 3D-Media-GPGPU

インテル® グラフィックス・ドキュメントのセット:

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


## 著者紹介


Robert Ioffe は、インテル コーポレーションのソフトウェア & ソリューション・グループのテクニカル・コンサルティング・エンジニアです。OpenCL* プログラミングとインテル® Iris™ グラフィックスまたはインテル® Iris™ Pro グラフィックスにおける OpenCL* ワークロードの最適化のエキスパートで、インテル® グラフィックス・ハードウェアを熟知しています。Khronos* の標準化作業に深くかかわっており、これまでに最新機能のプロトタイプの作成やインテル® アーキテクチャーでの動作検証を行ってきました。最近では、OpenCL* 2.0 の入れ子構造の並列処理 (enqueue_kernel functions) 機能のプロトタイプの作成に取り組み、OpenCL* 2.0 用の GPU クイックソートを含む、いくつかの入れ子構造の並列処理サンプルコードを作成しました。また、以下の単純な OpenCL* カーネルの最適化に関する動画 2 つを公開しています。現在、入れ子の並列処理に関する 3 つ目の動画を作成中です。


以下の記事も参照してください。

GPU-Quicksort in OpenCL 2.0: Nested Parallelism and Work-Group Scan Functions

Sierpiński Carpet in OpenCL 2.0

Optimizing Simple OpenCL Kernels: Modulate Kernel Optimization

Optimizing Simple OpenCL Kernels: Sobel Kernel Optimization

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。