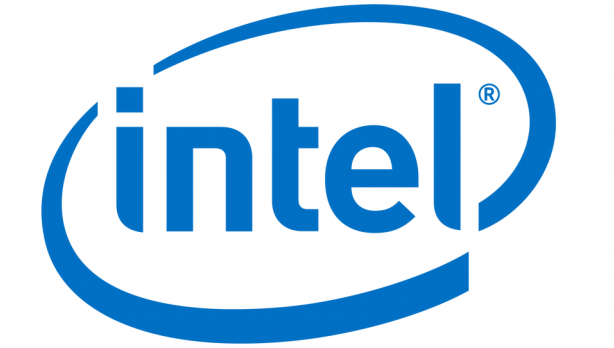

Movidius™

Myriad 2 SHAVE Assembly

Programming Guide

v1.01 / February 2018

Intel® Movidius™ Confidential

Copyright and Proprietary Information Notice

Intel Movidius
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Table of Contents

1 Introduction

This document is intended to be a programmer's guide on working with assembly language.

2 General structure of an .asm file

- `.version <versionNumber>` ; – set, macro, include, other preprocessor directives.
- `.data section` – data elements that are declared with an initial value (`.data .data.example`) .
- `.code section` – the instruction codes (`.code .text.example`).
- `.end`

3 Calling convention

The main purpose of these calling conventions is to allow programs compiled with `moviCompile` to call functions written in SHAVE assembly, and vice-versa. The programmer must ensure that the Stack pointer register, Link register and all of the preserved registers have the same values at the return of the handwritten assembly function as they had at the entry point.

Registers:

- *Stack pointer register* – IRF register I19 is reserved as a dedicated stack pointer and should never be modified except to allocate and free space for stack variables.
- *Link Register* – IRF register I30 is used both to store the address to which control flow should jump when a function is called, and the return address to which control flow should jump after a function has completed execution.
- *Preserved IRF registers* (i20,...,i29,i31) and *VRF registers* (v24,...,v31).

Up to 8 scalar parameters may be passed via the IRF, using registers i11, i12, ..., i18. Register i18 is also used to store the return value for a method call.

Up to 8 vector parameters may be passed via the VRF, using registers v16, v17, ..., v23. Register v23 is also used to store the return value for a method call if the return value is one of the above types.

If a method accepts more than 8 parameters, then parameters from 9 and onwards must be passed via the stack.

Example:

```
int Example(int a, int b, int c, int d, int e, int f,int g,int h,int
i,int j)

i18 = a;
i17 = b;...
i11 = h;
read i from i19
read j from i19 + sizeof(i)
```

At the end of the function, i18 should contain the value returned by Example.

4 Looping

You may use `BRU.RPL` (a variable number of times) or `BRU.RPS` (a constant number of times) to create a loop. Or you may use different jump instructions (`BRU.BRA` , `BRU.JMP`) with a condition or counter to create a conditional loop.

Example 1:

```
lsu0.ldil i0 Loop_end     || lsu1.ldih i0 Loop_end
lsu0.ldil i1 0x3          || lsu1.ldih i1 0x0
lsu0.ldil i2 0x0          || lsu1.ldih i2 0x0 // Initial value of i2
is zero
// this loop will execute i1 times;
Loop_start:
bru.rpl i0 i1 Loop_end    || iau.add i2,i2,1 // final value of i2 will
be 3
Loop_end:
nop 7
```

Example 2:

```
lsu0.ldil i0 0x0    || lsu1.ldih i0 0x0
lsu0.ldil i1 0x3    || lsu1.ldih i1 0x0
lsu0.ldil i2 0x0    || lsu1.ldih i2 0x0
// Initial value of i2 is zero; final value of i2 will be 3
L_for:
cmu.cmii.i32 i0,i1
peu.pc1c NEQ        || bru.bra L_for || iau.add i2,i2,1
iau.incs i0,1
nop 5
```

5 Branching

The basic implementation of an IF..THEN..ELSE statement will look like this:

```
lsu0.ldil i0 0x0     || lsu1.ldih i0 0x0
lsu0.ldil i1 0x3     || lsu1.ldih i1 0x0
lsu0.ldil i2 0x0     || lsu1.ldih i2 0x0
cmu.cmii.i32 i0,i1
peu.pc1c   EQ        || bru.bra L_branch1
peu.pc1c NEQ         || bru.bra L_branch2
nop 6
L_branch1:
// do branch 1 operations
iau.add i2,i2,3
iau.add i3,i2,i2
bru.bra L_end // make sure you jump over branch 2
nop 6
L_branch2:
// do branch 2 operations
iau.add i2,i2,1
iau.add i3,i2,i2
L_end:
```

If your branches are not so complicated, better avoid those `BRU.BRA` and those `nops` and do something like this:

```
lsu0.ldil i0 0x0     || lsu1.ldih i0 0x0
lsu0.ldil i1 0x3     || lsu1.ldih i1 0x0
lsu0.ldil i2 0x0     || lsu1.ldih i2 0x0
cmu.cmii.i32 i0,i1
peu.pc1c EQ          || iau.add i2,i2,3
peu.pc1c EQ          || iau.add i3,i2,i2
peu.pc1c NEQ         || iau.add i2,i2,1
peu.pc1c NEQ         || iau.add i3,i2,i2
```

or:

```
lsu0.ldil i0 0x0     || lsu1.ldih i0 0x0
lsu0.ldil i1 0x3     || lsu1.ldih i1 0x0
lsu0.ldil i2 0x0     || lsu1.ldih i2 0x0
iau.add i2,i2,1
iau.add i3,i2,i2
cmu.cmii.i32 i0,i1
peu.pc1c EQ          || iau.add i2,i2,3
peu.pc1c EQ          || iau.add i3,i2,i2
```

6 Predicate execution of instructions

PEU instructions operate on the condition code (CC) registers. The CC register has dedicated fields for condition codes generated by the CMU, SAU and IAU.

Example:

```
iau.sub i0 i1 i2
peu.pc1i GT                || predicted operations (IAU CC)

cmu.cmii.i32 i3,i4
peu.pc1c EQ                || predicted operations (CMU CC)


iau.sub i0 i1 i2           || cmu.cmii.i32 i3,i4
peu.pcc0i.AND  EQ GT       || predicted operations (combination between
CMU CC and IAU CC)
```

7 Predicate execution of instructions

You may predicate individual units, by using PEU.PCCX or PEU.PCIX. Each unit has a corresponding slot, which must be set to one if we want to predicate that unit.

```
// if i1== i2 all the operations will be executed
// else only the lsu1 operation will be executed
iau.sub i0 i1 i2
peu.pcix.EQ 0x9    ||  iau.operation || vau.operation ||
lsu1.operation
// this 0x9 is 1001 în binary, IAU and VAU slots are set to 1
```

Or even predicate Vector Elements to generate a VRF write enable for the current VAU/SAU/LSU0/LSU1 instruction.

```
// v0 = 1 2 1 3  // only if v0.x > v1.x v2.x will be increased with 5
// v1 = 1 1 1 1
// v2 = 0 0 0 0

cmu.cmvv.i32 v0 v1
peu.pvv32 gt        || vau.add.i32 v2 v2 5

// v2 = 0 5 0 5
```

8 Accumulator

Accumulators are used within the VAU and SAU and allow for power-efficient accumulation of values.

Suppose we have two matrices- int A[4][4] and int B[4][4] and we want to:

```
compute C[4] where
C[i]= A[0][i]*B[0][i]+A[1][i]*B[1][i]-A[2][i]*B[2][i]-A[3][i]*B[3][i]
```

Use accumulator on VAU unit:

```
vau.macpz.i32  v0   v4
vau.macp.i32   v1   v5
vau.macn.i32   v2   v6
vau.macnw.i32  v14  v3  v7
```

instead of this:

```
vau.mul.i32  v8   v0  v4
vau.mul.i32  v9   v1  v5
vau.mul.i32  v10  v2  v6
vau.mul.i32  v11  v3  v7
vau.add.i32  v12  v8  v9
nop
vau.add.i32  v13  v10  v11
nop 2
vau.sub.i32  v14  v12  v13
```

9 SIMD

Use SIMD (Single Instruction Multiple Data) to increase the performance of your code.

Suppose we want to add two arrays: int A[4] and int B[4].

We can do this by using IRF registers and 4 IAU.ADD instructions or we can do this by using VRF registers and a single VAU.ADD instruction.

Do this:

```
lsu0.ldi.64.l v0 srcA     || lsu1.ldi.64.l v1 srcB
lsu0.ld.64.h v0 srcA      || lsu1.ld.64.h  v1 srcB
nop 5
vau.add.i32 v0,v0,v1
```

instead of this:

```
lsu0.ldi.32 i0 srcA       || lsu1.ldi.32 i4 srcB
lsu0.ldi.32 i1 srcA       || lsu1.ldi.32 i5 srcB
lsu0.ldi.32 i2 srcA       || lsu1.ldi.32 i6 srcB
lsu0.ld.32 i3 srcA        || lsu1.ld.32  i7 srcB
nop 3
iau.add i0 i0 i4
iau.add i1 i1 i5
iau.add i2 i2 i6
iau.add i3 i3 i7
```

10 Deinterleave / Swizzle

Let's say we have two `VRF` registers and we want to save their elements from odd positions into a certain location and their elements from even positions into another location. This is how we can do this easily:

```
// input v0,v1
cmu.vdilv.x32 v3 v2 v0 v1
// output v2 - even positions
              v3 - odd positions
```

If you want to multiply all elements of a `VRF` with the first element of another `VRF`, do this:

```
cmu.vszm.word v0,v0,[0000] // fill v0 with v0.0   vau.mul.i32 v1,v1,v0
```

or better this:

```
vau.mul.i32  v1 v1 v0 || lsu1.swzmv4.word  [0000]
```

11 Alignvec / Shiv

Let's say we have a VRF register, an IRF register and we want to output a combination of these two. `Vout = IRF Vin[0] Vin[1] Vin[2]`

```
cmu.shliv.x32 Vout Vin IRF      // Shift IRF left into VRF.
```

or

```
cmu.cpivr v0,IRF                // fill a vrf with IRF value
cmu.alignvec Vout v0 Vin 12     // vector align with byte offset
```

12 Tips and tricks

- Minimize the number of memory accesses. We need to have a very good reason to read the same memory location multiple times. This has very big impact in the number of cycles, if the kernel will be used in a bigger project.

- Use preserved registers without saving them on the stack. If you need more IRF registers but you have lots of spare V registers, just use VRF registers as temporary storage if your application permits it.

- Don't forget to move the registers back to their proper places before exiting your handcrafted function!

- Avoid many small data transfers. Instead, use a single large data transfer. If possible, use one VRF register instead of four 32-bit IRF registers.

- Avoid floating point use if it's not absolutely necessary. Latency for integer types is lower and precision is higher, and the power consumption is probably lower too (operations in floating point incorporate significant more HW logic).

- If you have a full 32-bit number and you need to divide by x, you can simply do a multiply with the precalculated 1/x value. This is faster because multiplication is faster than division.

- Use `BRU.RPI` or `BRU.RPIM` instead of doing a loop if you have only one instruction to repeat several times. This will keep your code size smaller.

13 Assembly Optimization Guide

- **Step 0 – Starting from c implementation**

For the purpose of this guide we will take in consideration example 011 from `mdk/examples/Progressive`.

When writing assembly code, we need to take in consideration the initial `.c` code.

The main purpose of writing a simple and efficient code is meeting the need to process as many values we can at a time.

- **Step 1 – Translation**

The initial goal for writing assembly code is to write it in simple steps.

The first written code in assembly should include all the latency of the instructions used, and it should have a linear implementation. (no parallelization)

This type of writing is for the need of debugging first. If we have a very optimized code, it is very hard to debug the problems that appear at implementing `asm` code.

First step in the assembly implementation will include also many comments related to the functionality.

- **Step 2 – Removing “nop's” and parallelizing the asm code**

In the second step we want to remove unnecessary “nop's” that are present in the code from the first step. Unnecessary in this case means that the functionality will be the same even without some of the “nop's”. We start parallelizing the instruction by moving some `LSU0` instructions on the `LSU1` unit and put them in parallel.

Removing the nop's from the delay slot of the loop is another thing that is needed to be done. You will see that we don't have nop 7 at the end of the loop anymore. Instead of them we will have functional code from part of the loop. This means that we only moved the label of the main loop up a few instructions, but maintaining the 7 delay slot of the `bru.rpl` instruction.

- **Step 3 – Loop parallelization: changing instructions in order to have less instructions in a loop**

To achieve the best performance of the written `asm` code, the loop dimension should be given based on the most used unit.

In our simple code it is mandatory to take in consideration the VAU unit, that is used in 7 instructions plus another 9 instructions that are used to write out in a `vrf` register the result of our operations.

The main goal is to have the loop consisting of as little as possible number of instructions, so in our case the code from STEP 2 has 7+9 instructions that will give the loop a dimension consisting of 16 instructions.

For the purpose of this guide we will consider optimizing it at full capacity, and not take the precision in consideration.

The change of `VAU.MAC` with `VAU.MUL` and `VAU.ADD` will have the end result of reducing the number of instructions in the loop to 13 instructions.

- **Step 4 – Loop parallelization: making the loop based only on the principal unit used**

In order to achieve this end-result code, it is very useful to have different registers used for the results of the instructions, the registers used for inputs, should be different, then the ones used for the results, in this way we can parallelize the code better. It should work like a chain: the output from one instruction, should be the input for other instructions.

It is necessary to have in mind, that a function consists from a LOAD part, a PROCESS part and a STORE part. The structure of parallelizing should be the following:

- LOAD
- LOAD + PROCESSING
- LOAD + PROCESSING + STORE in the main loop parallelizing

This structure of optimizing an `asm` code can be achieved by parallelizing blocks, and removing them from the loop. In this situation, an overhead will be created, but the number of instructions from the loop will get reduced.

Depending on the code, this action of parallelizing and removing from the loop can consist of fewer steps.

In the STEP4 `asm` code, all the steps are marked to be easy to understand.