

# C152 Laboratory Exercise 4

Professor: Krste Asanovic

TA: Christopher Celio

Department of Electrical Engineering & Computer Science

University of California, Berkeley

March 22, 2012

## 1 Introduction and goals

The goal of this laboratory assignment is to allow you to explore the *vector-thread* architecture using the **Chisel** simulation environment.

You will be provided a complete implementation of a *vector-thread* processor, called *Hwacha*. Students will write *vector-thread* assembly code targeting *Hwacha*, to gain a better understanding of how data-level parallel code maps to vector-style processors, and to practice optimizing vector code for a given implementation. For the “open-ended” section, students will optimize a vector implementation of matrix-matrix multiply.

The lab has two sections, a directed portion and an open-ended portion. Everyone will do the directed portion the same way, and grades will be assigned based on correctness. The open-ended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task.

Students are encouraged to discuss solutions to the lab assignments with other students, but must run through the directed portion of the lab by themselves and turn in their own lab report. For the open-ended portion of each lab, students can work individually or in groups of two or three. Any open-ended lab assignment completed as a group should be written up and handed in separately. Students are free to take part in different groups for different lab assignments.

For this lab, there will only be one open-ended assignment. If you would prefer to do something else, you must contact your TA or professor with an alternate proposal of significant rigor.

## 2 Background

### 2.1 The *Vector-thread* Architecture

The *vector-thread* architecture is new style of a data-level parallel (DLP) accelerator that combines the efficiencies of traditional vector processors with the programability of general purpose GPU processors.[1][2]

Perhaps the easiest way to explain *vector-thread* is to first discuss *traditional* vector processors (i.e., the type of “vector processor” discussed in CS152 Lecture 15).

#### The *Traditional Vector* Architecture

Figure 1 shows a diagram of the programmer’s view of a *traditional* vector processor. The vector processor is composed of a *control processor* and a vector of *micro-threads*. The *control processor* fetches, decodes, and executes regular scalar code. It also fetches and decodes vector instructions, translating and sending the appropriate *vector commands* to an attached vector unit, which is conceptually composed on a vector of *micro-threads*.

A typical sequence of *traditional* vector assembly code is shown on the right half of Figure 1.

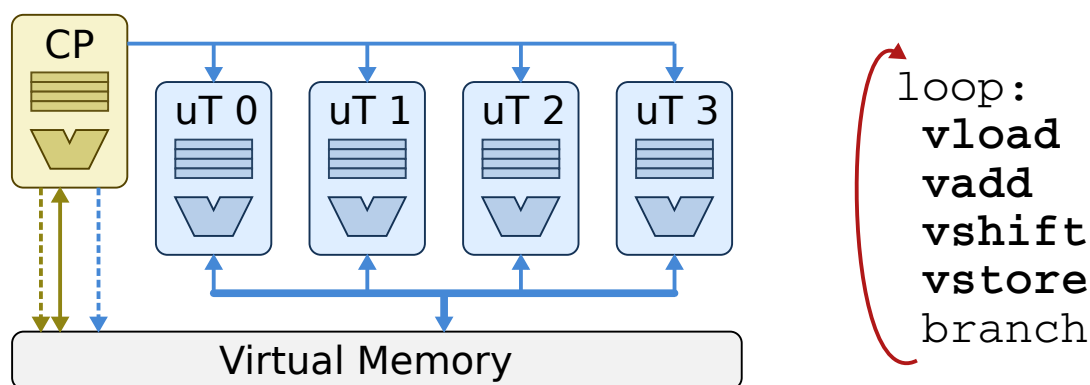


Figure 1: The programmer’s view of a *traditional* vector processor.

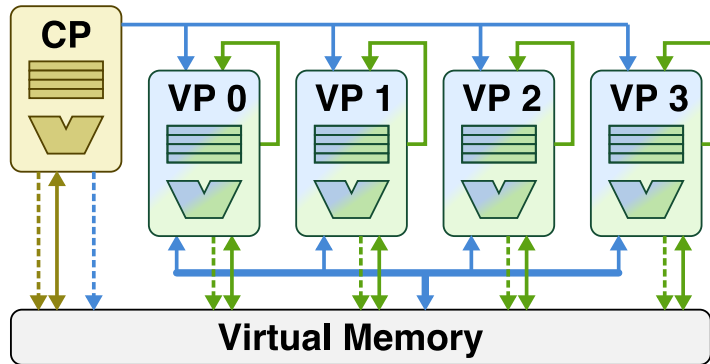
#### The Programmer’s View of *Vector-thread*

Figure 2 shows a diagram of the programmer’s view of a *vector-thread* processor. The *vector-thread* processor still has a *control processor*, which fetches and executes scalar code as usual. However, the *control processor* is connected to a vector of “virtual processors”, or *micro-threads*, which are **capable of fetching, decoding, and executing their own scalar code**. Thus, a programmer can write regular scalar code in a function `foo()`, which every *micro-thread* will execute (an “element function” in GPU-speak). The scalar core can then send the PC of the “vector-thread function” to each *micro-thread*.

**Assembly  
Programmer's  
View**

**Control  
Processor**

**Vector of Virtual  
Processors**



```

loop:
  vload
  vfetch foo
  vstore
  branch
foo:
  add
  shift
  vp.stop
  
```

Figure 2: The programmer’s view of a *vector-thread* processor. The *control processor* can send each “virtual processor”, or “micro-thread”, a PC which points to some piece of code `foo()`. Then, each “micro-thread” can begin fetching and executing the code found within the `foo()` function.

## 2.2 The *Rocket/Hwacha* Vector-thread Processor

A Chisel implementation of a full *vector-thread* processor is provided. The provided *vector-thread* processor comes with two big pieces: the *control processor*, known as *Rocket*, and the vector unit, known as *Hwacha*.

*Rocket* is a RV64S 6-stage, fully bypassed in-order core. It has full supervisor support (including virtual memory). It also supports sub-word memory accesses and floating point. In short, *Rocket* supports the entire 64-bit RISC-V ISA (however, no OS will be used in this lab, so code will still run “bare metal” as in previous labs).

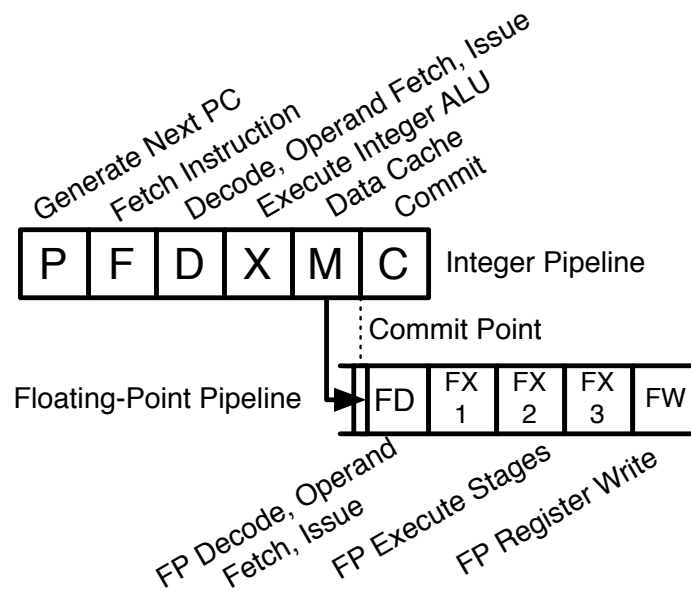


Figure 3: The *Rocket* control processor pipeline.

As the Control Processor, *Rocket* executes scalar code. However, when it encounters a *vector fetch* instruction, it will send the instruction (and the corresponding PC operand) to the *Hwacha* vector-unit, which will begin fetching and executing instructions starting at the given PC.

*Rocket* also handles vector memory operations. In the case of vector loads, *Rocket* calculates the base address of the load and sends it out to memory itself, while sending a “writeback” command to *Hwacha* (thus, *Hwacha* will know to expect load data to come back from memory). For vector stores, *Rocket* calculates the base address of the store and sends the instruction and address to *Hwacha*, where the store data can then be sent out to memory.

*Rocket* and *Hwacha* share L1 data and L1 instruction caches with each other. These caches are then backed up by a large magic “memory” that is a fixed number of cycles away.

As of 2012 March 27, both *Rocket* and *Hwacha* are being rapidly developed and debugged for a tape-out of a joint Berkeley/MIT research chip (Berkeley is developing the cores and L1 caches, while MIT is focusing on novel memory designs that are far beyond the scope of this lab). The upside of this is that you are playing with an actual, realistic processor design that is being used for real computer architecture research. The downside is that many of the tools and features are not yet mature, and it can be harder to grasp all of the moving parts of these very real processors!

## 2.3 Graded Items

You will turn in a hard copy of your results to the professor or TA. Please label each section of the results clearly. The following items need to be turned in for evaluation:

First, the end-goal of this lab is to fill out Chart 1, which compares the floating point performance of the *vector-thread* code running on the *Hwacha* vector-unit against the reference C code running on the scalar *Rocket* core. Each problem will guide you through the steps to accomplish this task. The performance results of *Rocket* have already been filled in for you.

Table 1: Performance of floating point benchmarks, measured by floating point operations per second (GFLOPs), cycles per element (CPE), and cycles per retired control processor instruction (CPI).

	vvadd	cmplxmult	matmul
Rocket (scalar)	0.143 GFLOPs 7.00 CPE 1.55 CPI	0.146 GFLOPs 41.05 CPE 2.73 CPI	0.279 GFLOPs 7.18 CPE 1.46 CPI
Hwacha (vt)			

1. Problem 3.3: Vvadd performance statistics and answers
2. Problem 3.4: Cmplxmult code, statistics, and answers
3. Problem 4.1: Matmul code, statistics, and answers
4. Problem 5: Feedback on this lab

## 3 Directed Portion

### 3.1 General Methodology

This lab will focus on writing *vector-thread* assembly code. This will be done in two steps: step 1) write assembly code and test it for correctness using the very fast *RISC-V ISA simulator*, and Step 2) measure the performance of your correct code on a Chisel-generated cycle-accurate simulator of the *Rocket/Hwacha* processor.

### 3.2 Setting Up Your Chisel Workspace

To complete this lab you will log in to an instructional server ( `t7400-{1,2,3,...,12}.eecs` ), which is where you will use **Chisel** and the RISC-V tool-chain.

First, download the lab materials:<sup>1</sup>

```
inst$ cp -R ~cs152/Lab4 ./Lab4
```

```
inst$ cd ./Lab4
```

```
inst$ export LAB4ROOT=$PWD
```

We will refer to `./Lab4` as `${LAB4ROOT}` in the rest of the handout to denote the location of the Lab 4 directory.

Some of the directory structure is shown below:

- `${LAB4ROOT}/`
  - `test/` Source code for benchmarks and tests.
    - \* **riscv-bmarks/** Benchmarks (mostly) written in C. This is where you will spend nearly all of your time.
      - `Makefile` Makefile for testing benchmarks for correctness on the ISA simulator.
      - `vec_vvadd` C and assembly code for the vector-vector add benchmark.
      - `vec_cmplmult` C and assembly code for the complex-multiply benchmark.
      - `vec_matmul` C and assembly code for the matrix multiply benchmark.
    - \* `riscv-tests/` Tests written in assembly.
  - `emulator/` C++ simulation tools and output files.
    - \* **Makefile/** Makefile for driving cycle-accurate C++ simulations.
  - `chisel` The **Chisel** source code.
  - `hwacha` The *Hwacha* source code.
  - `hardfloat` The floating point unit source code.
  - `src/` The *Rocket* source code.
  - `sbt/` **Chisel/Scala** voodoo. You can safely ignore this directory.

---

<sup>1</sup>The capital “R” in “cp -R” is critical, as the -R option maintains the symbolic links used.

The following command will set up your bash environment, giving you access to the entire CS152 lab tool-chain. Run it before each session:<sup>2</sup>

```
inst$ source ~cs152/tools/cs152.bashrc
```

For this lab, we will play with the benchmarks `vec_vvadd`, `vec_cmplxmult`, and `vec_matmul`. To compile and run these benchmarks on the RISC-V ISA simulator, execute the following commands:

```
inst$ cd ${LAB4ROOT}/test/riscv-bmarks/  
inst$ make clean; make; make run-riscv
```

This quickly tests the benchmarks for correctness using the ISA simulator. The `vec_cmplxmult` and `vec_matmul` benchmarks should FAIL, because you have not written the code for them yet!

To run the benchmarks on the cycle-accurate C++ simulator of *Hwacha/Rocket*, execute the following commands:

```
inst$ cd ${LAB4ROOT}/emulator  
inst$ make clean; make; make run
```

A number of scary error messages may pop up: “Aborted”, “Error 134 (ignored)”, etc. These can all be safely ignored, so long as each test outputs a PASS or a FAIL.

Unlike previous labs, the Chiseled processor and its C++ simulator have been compiled for you. Instead, you will only be compiling and running the benchmarks. Once you add your working complex multiply and matrix multiply code, the total simulation time should be about five to ten minutes.

### 3.3 Measuring the Performance of Vector-Vector Add (`vec_vvadd`)

To acclimate ourselves to the Lab 4 infrastructure and *vector-thread* coding in general, we will first look at the provided Vector-Vector Add (`vec_vvadd`) benchmark and measure its performance on *Hwacha*.

First, navigate to the `vec_vvadd` directory, found in `${LAB4ROOT}/test/riscv-bmarks/vec_vvadd/`. In the `vec_vvadd` directory, there are a few files of interest. First, the `dataset.h` file holds a static copy of the input vectors and results vector.<sup>3</sup> Second, `vec_vvadd_main.c` holds the main driving C code for the benchmark, which includes initializing the state of the program, calling the `vvadd` function itself, and verifying the correct results of the function. An example scalar implementation of `vvadd`, written in C, is provided in `vec_vvadd_main.c` as well. The assembly implementations are found in `vec_vvadd_asm.S`. Two versions are provided: first, a scalar assembly version of `vvadd`, and second, a *vector-thread* version of `vvadd`.

---

<sup>2</sup>Or better yet, add this command to your bash profile.

<sup>3</sup>There is a smaller input set, found in the file `dataset_test.h`, which is more manageable when testing out your code. Simply go in to the `*_main.c` file and change out which `dataset*.h` is included to change which input vectors to test your code on.

Now let's run the vector version of `vec_vvadd` on the ISA simulator:

```
inst$ cd ${LAB4ROOT}/test/riscv-bmarks/  
inst$ make clean; make; make run-riscv
```

This will delete out any old copies of the benchmarks, build new copies of the benchmarks, generate obj-dump files, generate hex file copies, and run the resulting RISC-V binaries on the RISC-V ISA simulator.<sup>4</sup> You should see a PASS for `vec_vvadd`, denoting that the output vector of our *vector-thread* implementation matches the reference results provided by the `dataset.h` file. For now, you should see FAIL for both the `vec_cmplmult` and `vec_matmul` benchmarks, since we have not yet written the code for them yet!

Now, we will run `vec_vvadd` on the cycle-accurate simulator of *Hwacha*.

```
inst$ cd ${LAB4ROOT}/emulator  
inst$ make clean; make; make run
```

You should see the following output, which corresponds to `vec_vvadd`:

```
fesvr -c -nopr -m2500000 +loadmem=../test/riscv-bmarks/vec_vvadd.riscv.hex none 2  
> vec_vvadd.riscv.out  
*** PASSED *** (num_cycles = 0x000000000000014d5, num_inst_retired = 0x000000000000001b)
```

The first line calls the RISC-V front-end server, in C++ simulator mode, and loads the `vec_vvadd` benchmark into the simulator's memory, and stores any log information into `vec_vvadd.risc.out`.

The second line is the output from the `vec_vvadd` program itself. In this example, we are provided information about the *number of cycles* executed by the critical function, and the *number of instructions retired* by the critical function in hexadecimal form (5333 cycles and 27 instructions respectively, in decimal).

Use this information to calculate the CPI of the control processor (retired instructions is measured from the scalar control processor's point of view), and to calculate the FLOPs ("floating point operations / second") achieved by *Hwacha*.

To calculate the FLOPs achieved, we need to know two things: how many floating point operations were performed, and how many seconds elapsed. To calculate the former, we need to look at the `vec_vvadd` code (`vec_vvadd_main.c` and `dataset.h`): we can see that every iteration performs one floating point add operation, and that `vec_vvadd` runs for 1024 iterations. To calculate seconds, we need to know the number of cycles that elapsed (provided by the above printout), as well as the clock rate of the processor. Both *Rocket* and *Hwacha* run at 1 GHz. Thus, since *Rocket* is a single-issue machine, its absolute maximum theoretical floating point performance is 1 GFLOPs.<sup>5,6</sup>

---

<sup>4</sup>Notice that the shown command (`make run-riscv`) runs the RISC-V binary. It is also possible to build the code and run it on the "host" x86 platform, using `make run-host`. The advantage is that you get full printf support (and a full OS), but the disadvantage is that you can not use any RISC-V assembly in your code.

<sup>5</sup>This is actually a bit of a lie, since *Rocket* and *Hwacha* support "fused multiply add" instructions, which perform a  $d = c + (a \times b)$  operation. Thus, with the `fmadd` and `fmsub` instructions, the processor can actually issue *two* floating point operations in a single cycle!

<sup>6</sup>Also, as discussed in Appendix C, *Hwacha* is also a "single-issue" machine, and thus has the same peak perfor-

### 3.4 Implementing Complex Multiply (cmlxmult) in *Vector-thread*

Now that you understand the infrastructure, how to run benchmarks, and how to collect results, you can write your own benchmark and measure its performance on the *Hwacha vector-thread* core.

The first benchmark will be Complex Multiply (cmlxmult). Complex multiply involves multiplying two vectors of complex numbers together element-wise. The pseudo-code is shown below:

```
1 // pseudo code
2 for ( i = 0; i < n; i++ )
3 {
4     e = (a*b) - (c*d);
5     f = (c*b) + (a*d);
6 }
7 }
```

In terms of calculating FLOPs, each iteration involves four FP multiplies and two FP adds, for a total of six FLOP per iteration. The actual C code is shown here:

```
1 struct Complex
2 {
3     float real;
4     float imag;
5 };
6
7 // scalar C implementation
8 void cmlxmult( int n, struct Complex a[], struct Complex b[], struct Complex c[] )
9 {
10     int i;
11     for ( i = 0; i < n; i++ )
12     {
13         c[i].real = (a[i].real * b[i].real) - (a[i].imag * b[i].imag);
14         c[i].imag = (a[i].imag * b[i].real) + (a[i].real * b[i].imag);
15     }
16 }
```

Add your *vector-thread* code to `test/riscv-bmarks/vec_cmlxmult/vec_cmlxmult_asm.S`. You will find an example scalar implementation written in RISC-V assembly in that same file, as well as a brief description of the RISC-V ABI calling convention (which provides suggestions on which registers to use).

When you are ready to test your *vector-thread* code, first test for correctness on the ISA simulator:

```
inst$ cd ${LAB4ROOT}/test/riscv-bmarks/
inst$ make clean; make; make run/
```

Once your code passes the correctness test, you can then gather performance results on cycle-accurate simulator of *Hwacha*:

```
inst$ cd ${LAB4ROOT}/emulator
inst$ make clean; make; make run
```

---

mance has *Rocket*, 2 GFLOPs (when using the FMA).



Collect your results and fill out the corresponding entries in Chart 1. Also, attach your *vector-thread* code to the appendix in your lab writeup.

**Hints:** You will almost certainly want to work with *strided* vector memory operations for this problem. For strided loads, the instruction is `vflstw vf1, rBaseAddr, rStride`, or *vector floating point load strided (word version)*. The argument `vf1` is the vector floating point register # 1 (you may use any number from 0 to 31. In RISC-V, the floating point register # 0 is *not* hard-wired to zero). The operand register `rBaseAddr` holds the starting memory address for the vector strided load to begin loading from, and `rStride` is a register that holds the size of the stride. Because this problem involves vectors of structs, and each complex number struct is 8 bytes in size, trying to load a vector of the *real* part of the complex numbers will involve a stride value of 8 (bytes). The corresponding store version is `vfsstw`.

Although not necessary, you may also get higher performance by using “fused multiply add” instructions, which are supported by *Hwacha* and *Rocket* ( $d = c + (axb)$ ). These instructions (`fmadd` and `fmsub`) allow *two* floating point operations to be issued in a single cycle, doubling floating point performance! See the provided `riscv-spec.pdf` for more information about the provided floating point instructions.

## 4 Open-ended Portion

For this lab, there will only be one open-ended portion that all students can do. As will all labs, you can work together in groups of two or three.

### 4.1 Contest: Vectorizing and Optimizing Matrix-Matrix Multiply

For this problem, you will implement a *vector-thread* implementation of matrix-matrix multiply. A scalar implementation written in C can be found in `test/riscv-bmarks/vec_matmul/vec_matmul_main.c`, and a scalar implementation written in RISC-V assembly can be found in `test/riscv-bmarks/vec_matmul/vec_matmul_asm.S`. Add your own *vector-thread* implementation in `test/riscv-bmarks/vec_matmul/vec_matmul_asm.S`.

Once your code passes the correctness test, do your best to optimize `matmul` for *Hwacha*. This will be a contest, with the best team, as measured by the achieved FLOPs (i.e., the lowest number of cycles to correctly execute), will receive a bonus +2 points on the lab. You are only allowed to write code in the `vt_vvadd_asm` function (i.e., do *not* change any code in the `vec_matmul_main.c` file).

Attach your `matmul` *vector-thread* assembly code in an appendix of your lab report. Describe what your code does, and some of the strategies that you tried.

### Matrix Multiply Hints

A number of strategies can be used to optimize your code for this problem. First, the problem size is for square matrices 64 elements on a side, with a total memory footprint of 48 kB (the L1 data cache is only 32 kB). Common techniques that generally work well are loop unrolling, lifting loads out of inner loops and scheduling them earlier, blocking the code to utilize the full register file, transposing matrices to achieve unit-stride accesses to make full use of the L1 cache lines, and loop interchange.

More specific to *vector-thread*, try and have all element loads be re-factored into vector loads performed by the control processor. Use fused multiply-add instructions as often as possible. Also, carefully choose *which* loop(s) you decide to vectorize for this problem: not all loops can be safely vectorized!

Finally, be mindful about the use of the `fence.l.v` instruction: it is expensive and can hurt performance, but you *must* use it when you need the results of stores visible from a given *micro-thread* to the *control processor* or to other *micro-threads*.

## 5 The Third Portion: Feedback

This is a brand new lab, and as such, your TA would like your feedback again! How many hours did the directed portion take you? How many hours did you spend on the open-ended portion? Was this lab boring? Did you learn anything? Is there anything you would change?

Feel free to write as little or as much as you want (a point will be taken off only if left completely empty).

## 6 Acknowledgments

This lab was made possible through the work of Yunsup Lee and Andrew Waterman (among others) in developing the *Rocket* and *Hwacha* processors, and in helping make the RISC-V tool-chain available to users at large.

## A Appendix: Debugging

Debugging your *vector-thread* code can be difficult. To make matters worse, you do not have an OS to call upon, `gdb`, or `printf`.

However, there are a couple of strategies that will help.

First, some simple printing functions are provided: `printstr()` and `printhex()`. These functions, found in `test/riscv-bmark/stuff/syscalls.cc`, allow you to print out a static string and an integer value respectively. This can allow you to check conditions and print out the appropriate strings from your code.

Second, the ISA simulator can be run in a debug mode that prints out an instruction trace. For example, the basic command for running `vec_vvadd` in the ISA simulator is:

```
fesvr -nopk vec_vvadd.riscv
```

However, adding “-d” will provide a log of the instructions executed.

```
fesvr -d -nopk vec_vvadd.riscv
```

The only down-side is that this only shows the instruction trace from the point of view of the *control processor*: the vector unit is effectively invisible.

You can also look at the instruction trace outputted by the cycle-accurate simulator. For `vec_vvadd`, that would be found in `emulator/vec_vvadd.riscv.out`.

The objdump of the RISC-V binaries can be found in `test/riscv-bmarks/*.riscv.dump`, which can be very useful for comparing with the instruction traces and verifying that the code you wrote was correctly translated by the compiler.

If you are confused about vector-thread, I recommend that you look at the CS152 Section 9 slides, look through Yunsup Lee’s ISCA 2011 slides on *vector-thread* (<http://www.eecs.berkeley.edu/~yunsup/papers/maven-isca2011-talk.pdf>)[2], and look through the provided `vec_vvadd` code. Once you’ve done that, feel free to begin hammering Piazza for more guidance!

## B Appendix: The Vector Configure Instruction

Before executing vector code, the *control processor* must execute the `vvcfgivl` instruction to “configure” the vector unit (e.g., `vvcfgivl rVLen, rN, 32, 32`). The `vvcfgivl` instruction takes in four operands: the *second* operand is a register source signifying the *requested* application vector length; the *first* operand is a register destination in which the vector unit tells what the *given* vector length is<sup>7</sup>; the first immediate field signifies how many fixed point registers each *micro-thread* needs; and the second immediate field signifies how many floating point registers each *micro-thread* needs.

For example, `vec_vvadd` only needs two floating point registers to add two elements together; therefore, *Hwacha* is capable of providing a much larger *vector length* by giving each element fewer registers to work with.

However, the programmer must be careful to only use the lower order registers. In the case of `vec_vvadd`, since only two registers are requested, the code must only touch the `f0` and `f1` registers. **WARNING:** there is a bug in *Hwacha* if you set `vector configure` to use *zero* registers of either type ... so don’t do that.

## C Appendix: The Hardware View of *Hwacha* and other Data-Parallel Accelerators

Section 2.1 describes the programmer’s view of the *vector-thread* and *traditional vector* paradigms. However, it may be instructive to understand how the programmer’s view gets mapped to actual hardware.

Figure 4 show a number of different styles of machine for accelerating data-level-parallel code (DLP). As reference, machine (a) is a simple scalar core, which has a single set of registers and a single set of ALUs/FP units. Multi-threaded cores (b) “time-plex” multiple *threads* onto a single set of functional units, by adding additional copies of the register file (and other thread state). A *control processor* (CP) and SIMD unit (c) is used by x86 processors to provide some acceleration to DLP code. A *traditional vector* core, in the style of the Cray 1 (d), uses a scalar CP to fetch and send vector commands to the attached vector unit. As shown in Figure 4 (d), a single-lane *temporally* maps a set of *micro-threads* to a single set of functional units. Figure 4 (e) shows that you can also map *micro-threads* spatially, as well as temporally, to multiple lanes for increased performance (at a cost to power, energy, and area efficiency). A quasi GPU style is shown in Figure 4 (f), aka Single Instruction Multiple Threads (SIMT), which lacks a control processor to run ahead and start vector memory operations early for the *micro-threads*.

Finally, Figure 4 (g) shows a *vector-thread* core that is mapping *micro-threads* both spatially and temporally to multiple lanes. Notice that the Vector Issue Unit can directly access the Instruction

---

<sup>7</sup>As an example, the application may request a vector length of 512 elements, but the maximum vector length of the processor may only be 32, so 32 is what will be returned in the *rd* field.

Memory, as *vector-thread* units fetch and decode their own instructions.

An excellent source of understanding the taxonomy of DLP accelerators in general, and *vector-thread* cores in particular, is the paper “Exploring the Tradeoffs in Programmability and Efficiency in Data-Parallel Accelerators” [<http://www.eecs.berkeley.edu/~krste/papers/maven-isca2011.pdf>][3].

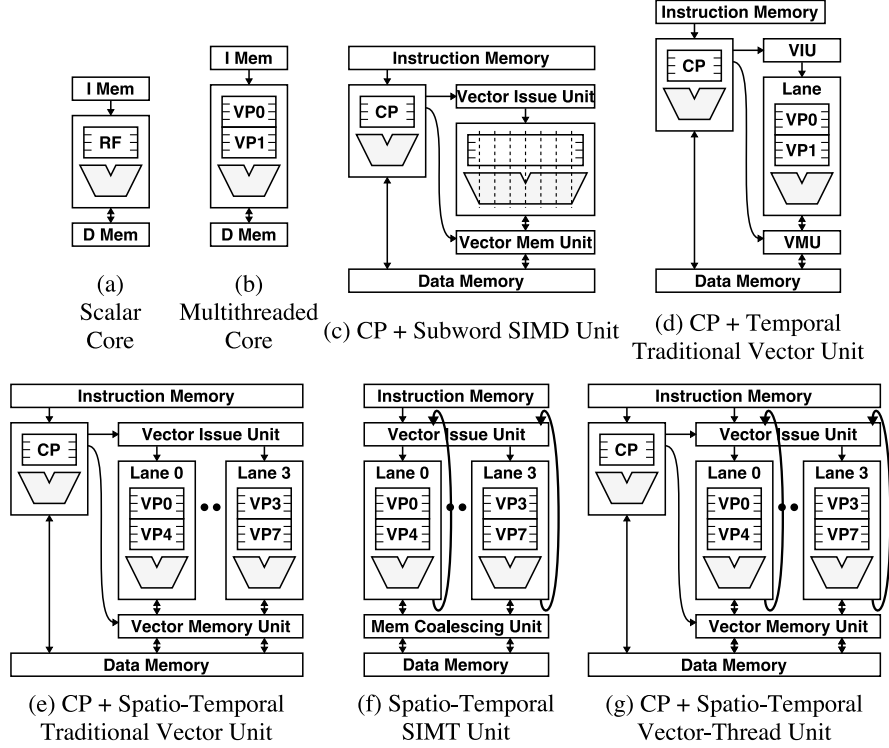


Figure 4: A comparison of different styles of machines that are all designed to exploit data-level parallelism. It is possible to map *micro-threads* to hardware both *spatially* and *temporally*.

## The *Hwacha* Implementation

*Hwacha*, much like the Cray 1 discussed in class, is a single-lane implementation of a *vector-thread* core. In other words, *Hwacha* has only a single set of functional units, which means that every cycle it is only issuing a single *micro-thread* to the floating point unit at a time! Thus, in terms of raw performance, *Rocket* and *Hwacha* should be closely matched!

In reality, *Hwacha* should out-perform *Rocket*. For one, *Hwacha* uses a CP to run-ahead and “prefetch” data into *Hwacha*’s vector registers. *Hwacha* also has a lot more registers than *Rocket*, which effectively act as a zero-level cache. The CP can also perform things like “book-keeping” in parallel with the execution of the micro-threads, removing activities like array index math off the critical path of the computation.

*Hwacha* is also able to chain incoming loads to the floating point unit. For `vvadd`, *Rocket* requires at least four cycles per element (`load`, `load`, `fadd`, `store`). However for *Hwacha* the last `load` can be chained into the `fadd`, allowing for a minimum of three cycles per element.

Finally, while this lab asks students to measure the raw performance in FLOPs, it completely ignores the significant gains in energy and power efficiency that *vector-thread* provides (for example, fetching and decoding far fewer instructions).

## References

- [1] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. *IEEE Micro*, 24(6):84–90, 2004. [<http://groups.csail.mit.edu/cag/scale/papers/vta-isca2004.pdf>].
- [2] Y. Lee. *ISCA Talk Slides*, 2012. [<http://www.eecs.berkeley.edu/~yunsup/papers/maven-isca2011-talk.pdf>].
- [3] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanovic. Exploring the Tradeoffs in Programmability and Efficiency in Data-Parallel Accelerators. *ISCA*, 2011. [<http://www.eecs.berkeley.edu/~krste/papers/maven-isca2011.pdf>].