CS 152 Laboratory Exercise 5

Professor: John Wawrzynek
TA: Martin Maas
Department of Electrical Engineering & Computer Science
University of California, Berkeley

November 19, 2016

1 Introduction and goals

The goal of this laboratory assignment is to allow you to explore a multi-core, shared memory environment using the `Chisel` simulation environment.

You will be provided a complete implementation of a multi-core *Rocket* processor (the scalar processor from Lab 4). Students will write C code targeting *Rocket*, to gain a better understanding of how data-level parallel (DLP) code maps to multi-core processors, and to practice optimizing code for different cache coherence protocols.

For the "open-ended" section, students will write and optimize a multi-threaded implementation of spare-matrix vector multiply for two different cache coherence protocols.

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 will work individually or in groups of two or three. Each group will turn in a single report for the open-ended portion of the lab. 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 Multi-core *Rocket* Processor

A `Chisel` implementation of a full multi-core processor is provided.

The *Rocket* Processor

Rocket will return from lab 4, but this time, there are multiple *Rocket* cores. Each core has its own private L1 instruction and data caches. The data caches are kept "coherent" with one another.

Rocket is a RV64G 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.). Both the user-level ISA manual and the supervisor-level ISA manual can be found on the RISC-V website.

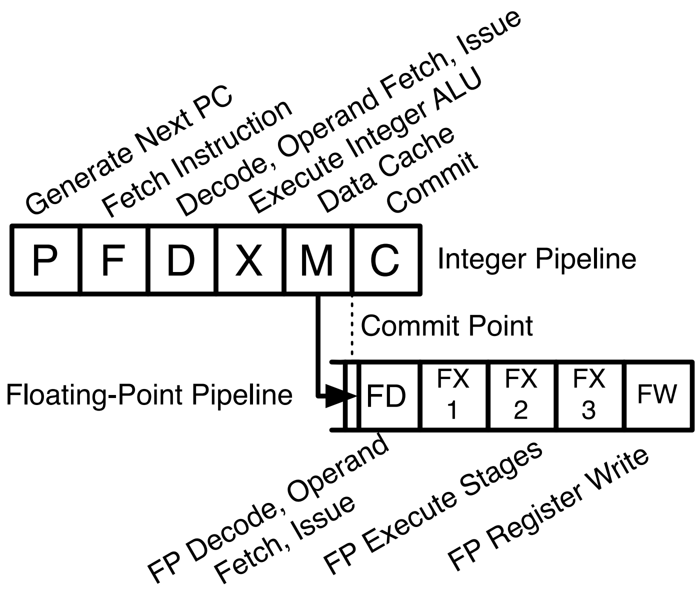

Figure 1: The *Rocket* control processor pipeline.

In this lab, the *Hwacha* vector-fetch unit has been removed; we will only be writing multi-threaded code.

2.2 The Memory System

In this lab, you are provided a multi-core processor that utilizes a directory-based cache coherence protocol. Figure 2 shows the high-level schematic of this system.

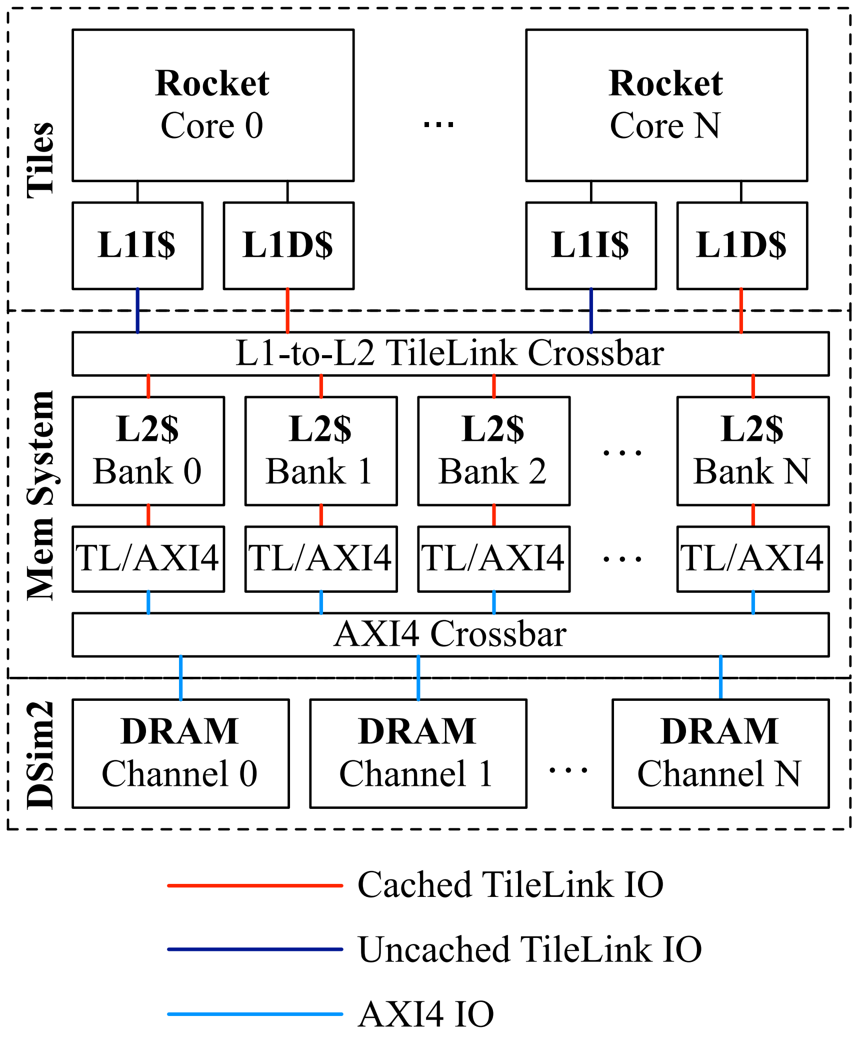

Figure 2: The multi-core *Rocket* system. A crossbar connects the private caches to the shared cache and the shared cache to main memory. The shared cache contains the directory and manages the cache coherence protocol.

Each *Rocket* core has its own private L1 instruction and write-back data caches. The cores share a single multi-banked level 2 cache. An off-chip memory provides the last level in the memory hierarchy. Both cores are connected via a crossbar to main memory. The crossbar and L2 cache can handle multiple inflight transactions. The L2 is connected to memory with another crossbar that has configurable number of memory channels on the backside.

Conceptually, Cache coherence is maintained by having caches send their intentions to the L2 cache and having it check for with other caches to acheieve coherence.

2.3 The Lab 5 Multi-threaded Programming Environment

In most multi-threaded programming systems, one thread begins execution at `main()`, who must then call some sort of `spawn()` or `thread_create()` function to create more threads with help from the operating system.

However, we will not be using an operating system in this lab. Instead, *ALL* thread begins execution at a function called `thread_entry()` (there is *no* `main()` function in this lab). Each thread is provided a `coreid` (its inique core id number, either 0 or 1), and `ncores` (number of cores, which will vary from 2 to 8 for this lab).

You will need to be careful where you allocate memory in your code. As there is no OS, you cannot use `malloc` to dynamically allocate more memory. By default, your code will allocate space on the stack, *however* each thread is provided only a very small amount of stack space. You will want to use the `static` keyword to allocate memory *statically* in the binary, where it is visible to both threads. There is also the `__thread` modifier, which denotes a variable that should be located in "thread-local storage" memory. Each thread is provided a very small amount of "thread-local storage", where variables visible only to the thread can be located.

2.4 Memory Fences & other Synchronization Primitives

A `barrier()` function is provided to synchronize all threads. Once a thread hits the `barrier()` function, it stalls until all threads in the system have hit the `barrier()`. Implicit in the `barrier` is a memory `fence`. The `barrier()` function should probably be enough to implement any algorithms necessary in this lab.

For more information on the RISC-V memory model, consult Section 2.7 of the user-level ISA manual in the `specifications` section of the RISC-V website. The RISC-V `fence` instruction can be executed by calling `__sync_synchronize()` gcc built-in function (i.e., saving you the hassle of inlining assembly). The gcc compiler provides more built-in functions, such as `__sync_fetch_and_add()`, which correspond to RISC-V atomic operations.

The `fence` instruction performs as follows: it is sent to the L1 data cache. If the cache is not busy, the `fence` instruction returns immediately and the pipeline continues executing. If the cache is busy servicing oustanding memory requests (i.e., cache misses), the `fence` stalls the processor pipeline until the cache is no longer servicing any outstanding memory requests. In this manner, the `fence` instruction ensures that any memory operations *before* the fence has completed before any memory operations *after* the fence has started.

2.5 WARNINGS and Pitfalls

Here are a few warnings and pitfalls that may cause errors in your code.

The stack space provided to each thread is only 8KB. There is no virtual memory protecting your stack, so there is no warning if you overrun your stack (try to allocate arrays and other large structures statically).

The thread-local storage is also very small, and also has no warning if you overrun it. Also, no matter what your code says, all memory is initialized to *zero* in thread-local storage.

You may use `printf` to debug your code, however, only thread 0 may execute it. Also, the printf provided with this lab does not support outputting floating point numbers; you will have to cast them to integers first. However, you will note that the auto-generated input vectors are actually using whole numbers.

2.6 Graded Items

You will turn in a copy of your results via Gradescope. Please label each section of the results clearly. The directed items need to be turned in for evaluation. Your group only needs to turn in *one* of the problems found in the open-ended portion. Some of the open-ended questions also request source code – please turn those in via e-mail to the TA. This should include the files you have modified such that they can be replaced with the current versions to replicate your results.

The following items need to be turned in for evaluation: First, the end-goal of this lab is to fill out Table 1. Some of the values have been filled in for you. Each problem will guide you through the steps to accomplish this task.

1. Problem 3.3: `Vvadd` performance statistics and answers
2. Problem 3.4: `Vvadd-Optimized` code, performance statistics and answers
3. Problem 4.1: `Matmul` or SPMV code, statistics, and answers
4. Problem 5: Feedback on this lab, and all labs

Submit your `vvadd` and `matmul` or `spmv` code via email.

3 Directed Portion (50% of lab grade)

3.1 General Methodology

This lab will focus on writing multi-threaded C code. This will be done in two steps: step 1) build the C++ cycle-accurate emulator of the multi-core processor (if the cache coherence needs to be changed, or the number of cores), and Step 2) verify the correctness and measure the performance of your code on the cycle-accurate emulator.

3.2 Setting Up Your `Chisel` Workspace

The tools for this lab were set up to run on any of the 5 instructional Linux servers `icluster5.eecs`, `icluster6.eecs`, `...`, `icluster9.eecs`. (see `http://inst.eecs.berkeley.edu/cgi-bin/clients.cgi?choice=servers` for more information about available machines).

First, download the lab materials. This lab is now managed as a git repository which means you can also use git to fetch updates from the published version. To copy the repo you will need to clone it:

```
inst$ cd ~
inst$ git clone ~cs152/fa16/lab5-git lab5
inst$ cd lab5
inst$ export LAB5ROOT=$PWD
```

If any updates are released you can then pull in the new updates using

```
inst$ cd ${LAB5ROOT}
inst$ git pull
```

If you encounter problems using git feel free to post a question on Piazza or consult the git documentation (see `https://git-scm.com/doc`)

We will refer to `./lab5` as `${LAB5ROOT}` in the rest of the handout to denote the location of the Lab 5 directory.

- ${LAB5ROOT}/rocket-chip/
  - riscv-tools/riscv-tests/ Source code for assembly tests and benchmarks.
    - **benchmarks/** Benchmarks (mostly) written in C. This is where you will spend nearly all of your time.
      - mt-vvadd C and assembly code for the vector-vector add benchmark.
      - mt-matmul C and assembly code for the matrix multiply benchmark.
      - mt-spmv C and assembly code for the sparse matrix vector multiply benchmark.
    - isa/ Assembly code for the individual instruction tests.
  - **emulator/** C++ simulation tools and output files.
  - csrc/ C++ test bench source code.
  - dramsim2/ DRAMSim2 source code that is used to emulate DRAM.
  - chisel/ The `Chisel` source code.
  - rocket/ The *Rocket* processor.
  - hardfloat/ The floating point unit source code.
  - uncore/ The uncore source code.
  - src/ Top-level source code.
  - sbt/ `Chisel`/Scala voodoo. You can safely ignore this directory.

The following command will set up your bash environment, giving you access to the entire CS152 lab tool-chain. Run it before each session:[1]

```
inst$ source ~cs152/fa16/cs152.lab5.bashrc
```

To compile the cycle-accurate dual-core *Rocket*C++ simulator, execute the following commands:

```
inst$ cd ${LAB5ROOT}/rocket-chip/emulator
inst$ make clean; make
```

This will build the default configuration which is two cores with a 256KB L2 cache, with an MI coherence directory protocol.

For this lab, we will play with the benchmarks `mt-vvadd`, and either `mt-matmul` or `mt-spmv`. To compile these benchmarks, and execute the binary on the ISA simulator, run the following commands:

```
inst$ cd ${LAB5ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks
inst$ make clean; make; make run-riscv
```

If they run correctly you should see output like the following at the end

```
spike mt-vvadd.riscv > mt-vvadd.riscv.out
spike mt-matmul.riscv > mt-matmul.riscv.out
echo; perl -ne 'print "  [$1] $ARGV \t$2\n" if /\*{3}(.{8})\*{3}(.*)/' \
       mt-vvadd.riscv.out mt-matmul.riscv.out; echo;
```

[1]Or better yet, add this command to your bash profile.

Now, we run the compiled benchmarks on the C++ emulator:

```
inst$ cd ${LAB5ROOT}/rocket-chip/emulator
inst$ make run-bmark-tests
```

The naive versions of `vvadd` and `matmul` benchmarks should PASS, but the optimized versions should FAIL, because you have not written the code for them yet!

This tests the benchmarks for correctness and outputs the performance metrics running on the emulator. It should take about one to two minutes to run both `vvadd` and `matmul` on the emulator.

3.3 Measuring the Performance of Vector-Vector Add (`vvadd`)

First, to acclimate ourselves to the Lab 5 infrastructure, we will gather the results of a poorly written implementation of `vvadd`. Navigate to the `mt-vvadd` directory, found in

```
${LAB5ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/
```

In the `mt-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.[2] Second, `mt-vvadd.c` and `vvadd.c` hold the 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 very poor implementation of `vvadd` can be found in the function `vvadd()`. Run the `vvadd` benchmark and gather the performance results of this unoptimized implementation:

```
inst$ cd ${LAB5ROOT}/emulator
inst$ make run-bmark-tests
```

`make` will run both `vvadd` and `matmul` benchmarks on the emulator, when changes are detected. You should get something similar to the following output, which corresponds to `vvadd`:

```
vvadd(cid, nc, 1000, results_data, input2_data, results_data);
barrier(nc): 17800 cycles, 17.8 cycles/iter, 1.9 CPI
```

This is the output from the `stats()` macro, which times a section of code and outputs the resulting performance statistics. The `vvadd()` function is a non-optimal implementation of a multi-threaded `vvadd` function. For now, ignore the `matmul` statistics.

Record the non-optimal `vvadd()` *function results.* By default, the *Rocket* processor is using the MI cache coherence protocol. Now we will change the cache coherence protocol and record the new performance metrics.

[2] You can generate your own input arrays with a smaller size for rapid testing. See `vvadd_gendata.pl` for details.

Changing the Cache Coherence Protocol

The different protocols are already set in different configurations. For this lab there are configs from 2 to 8 cores with MI, MSI, and MESI protocols. To rebuild the emulator using the MSI protocol for example:

```
inst$ cd ${LAB5ROOT}/rocket-chip/emulator
inst$ make clean; make CONFIG=DualCoreMSIConfig run-bmark-tests
```

This will take about five minutes to build and an additional two minutes to run the benchmarks. *Record and report the results of* `vvadd()` using the MSI protocol.

Analyze the `vvadd()` code in `${LAB5ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/vvadd/vvadd.c`. *Taking into consideration that the code is written for a multi-core cache-coherent system, what is sub-optimal about the provided implementation?*

3.4 Optimize VVADD

Now that you know how to run benchmarks, record results, and change the cache coherence protocol, you can now optimize `vvadd` for the dual-core *Rocket* processor. You should write your code in the provided `vvadd_opt` function, found in `${LAB5ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/vvadd/vvadd.c`. Make sure to remove the multi-line comments in `mt-vvadd.c` on lines 72 and 93 to begin running the optimized version.

Collect results of your `vvadd` *implementation, for the MI, MSI, and MESI protocols. What did you do differently to get better performance over the provided* `vvadd` *function?* You should be able to at least double the performance over the provided `vvadd` function (or thereabouts). Does your performance scale as you add more cores? Record the statistics for the 4 and 8 core with different protocols. You can run all of the different configurations with:

```
inst$ cd ${LAB5ROOT}/rocket-chip/emulator
inst$ make clean; ./cs152-run-all.sh
```

If you're implementation doesn't scale well what do you think you could do to fix it? Feel free to optimize further if you would like but it is not required for this lab.

VVADD Hints

You can now use `printf` to test your code, however it can only be executed from core 0, and does not handle floating point numbers. If you want to see what each core is doing cycle by cycle, look at the `*.out` log file in the emulator directory.[3]

You may also want to use the command:

```
inst$ cd ${LAB5ROOT}/rocket-chip/emulator
inst$ make output/mt-vvadd.riscv.out
```

This will allow you to only run `vvadd` which will speed up your development time.

[3]Core 0 output is prefixed with `C0`, and core 1 data is prefixed with `C1`. You can parse `stderr` by using `2> output.txt` to pipe `stderr` to a file.

4 Open-ended Portion (50% of lab grade + 10% bonus)

For this lab, there will only be two open-ended portions that all students can do. As will all labs, you can work individually or together in groups of two. If you would like try something else, talk to your TA about other applications you can implement instead that are of sufficient merit.

4.1 Contest: Parallelizing and Optimizing Matrix-Matrix Multiply or Sparse Matrix-Vector Multiply

For this problem, you will implement a multi-threaded implementation of matrix-matrix multiply, or sparse matrix-vector multiply. The matrix-matrix version is slightly simpler and more commonly solved, for a challenge and more interesting grading for your TA the sparse matrix-vector multiplication is recommended.

You will write three versions: one version that targets each of the cache three cache coherence protocols MI, MSI, MESI. If you do not feel any special targetting is necessary please explain carefully why this is the case or you risk losing points.

A naive, single-threaded implementation can be found in `${LAB5ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/mt-[matmul,spmv]/[matmul,spmv].c`. Feel free to comment it out to save yourself simulation time.

You will fill in your own in the provided function `[matmul,spmv]_opt()`. You may add additional helper functions, so long as any additional code you add is within the `stats()` function.

Once your code passes the correctness test, do your best to optimize `matmul` or `spmv`. This will be a contest, with the best team, as measured by the fewest cycles, will receive a bonus 10% points on the lab. There will be separate contests for each of the two benchmarks and you can only submit to one contest. Your results from the MI, MSI, and MESI versions will be averaged together. Go crazy!

Include your code in your email submission. *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 32 elements on a side, with a total memory footprint of 12 KB (the L1 data cache is only 4 KB, 4-way set-associative). 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.

You will also want to minimize sharing between cores; in particular, you will want to have each core responsible for writing its own pieces of the arrays (you do not want write permissions to ping pong between caches).

5 The Third Portion: Feedback

This is a newly refreshed lab, and as such, we 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 hard work of Andrew Waterman and Henry Cook (among others) in developing the *Rocket* processor, memory system, cache coherence protocols, and multi-threading software environment. This lab was originally developed for CS152 at UC Berkeley by Christopher Celio.

Table 1: Performance of the Lab 5 benchmarks, measured by *total cycles*, *cycles per iteration*, and *cycles per instruction* (CPI). Single thread performance is compared against dual thread implementations running on MI and MSI cache coherence protocols.

	vvadd	vvadd (opt)	matmul	spmv
one thread	23.7k cycles 23.7 cycles/iter 2.6 CPI	n/a	494k cycles 15.0 cycles/iter 1.6 CPI	
two threads (MI)				
two threads (MSI)				
two threads (MESI)				
four threads (MI)				
four threads (MSI)				
four threads (MESI)				
eight threads (MI)				
eight threads (MSI)				
eight threads (MESI)				