CS152 Laboratory Exercise 5, Version 1.1

Professor: Krste Asanović
TA: Albert Magyar
Department of Electrical Engineering & Computer Science
University of California, Berkeley

April 22, 2019

1   Introduction and goals

The goal of this laboratory assignment is to allow you to explore a dual-core, shared memory environment using the Chisel simulation environment. You will be provided a complete implementation of a dual-core Rocket processor supporing the RV64G ISA. You 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.

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 and appropriate formatting of the report. 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. For this lab, there is only one open-ended assignment, in which you will write and optimize a multi-threaded implementation of matrix-matrix multiply for two different cache coherence protocols.

You are encouraged to discuss problem-solving approaches with other students, but each student 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, you may 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. You are free to take part in different groups for different lab assignments.

2 Background

2.1 The Dual-Core Rocket Processor

Rocket will be returning from lab 2, but this time, there are two Rocket cores. Each core has its own private L1 instruction and data caches. The data caches are kept *coherent* with one another.

Rocket is an 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 CS 152 course website, under the handouts section.
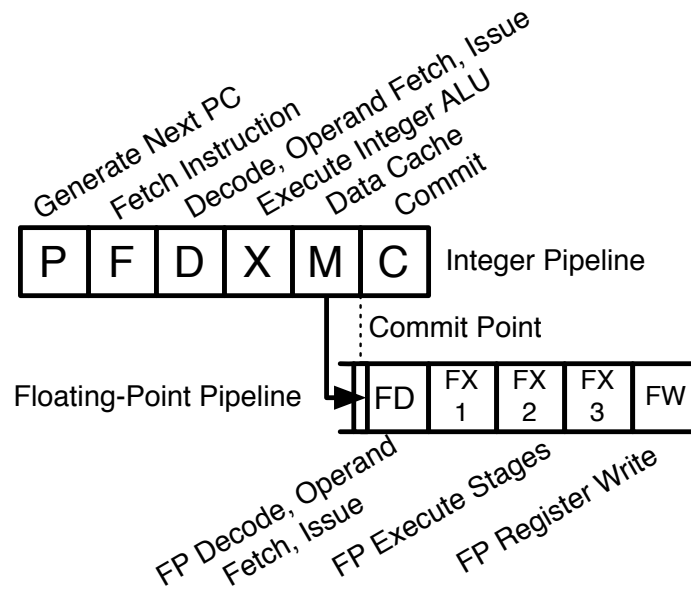

Figure 1: The Rocket control processor pipeline.

2.2   The Memory System

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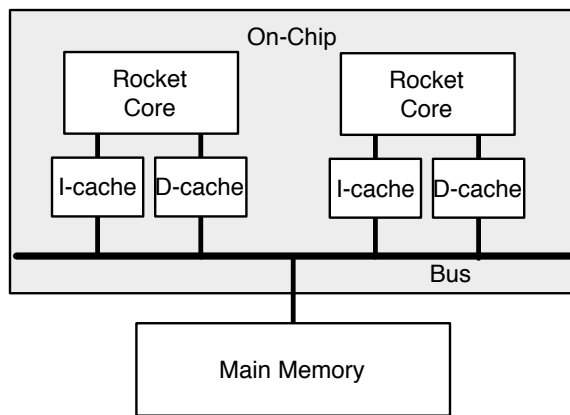

Figure 2: The dual-core Rocket system. A *logical* bus connects the caches and main memory to one another. In practice, the bus is implemented as a cross-bar with a coherence hub that arbitrates access to the "bus," initiates coherence "probe" traffic across the bus, and handles the cache coherence protocol.

Each Rocket core has its own private L1 instruction and write-back data caches. An off-chip memory provides the last level in the memory hierarchy. Both cores are connected via a bus to main memory. Only one agent may talk on the bus at a time.

Conceptually, Cache coherence is maintained by having caches broadcast their intentions across the bus, and to "snoop", or listen in on, the actions of the other caches. Consult "Lecture 18: Cache Coherence" for more information on snoopy caches.

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 OS.

In contrast with a typical multi-threaded environment, 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 unique core id number, either 0 or 1), and `ncores` (number of cores, which will always be 2 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 both 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 `resources` section of the CS 152 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()`.

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 outstanding 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

- 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 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:

1. Problem 3.3: `Vvadd` performance statistics and answers

2. Problem 3.5: `Vvadd-Optimized` code, performance statistics and answers

3. Problem 4: `Matmul` code, statistics, and answers

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. You will also submit your `vvadd` and `matmul` implementations via Gradescope. Submission instructions will be posted to Piazza.

Table 1: Performance of the Lab 5 benchmarks, measured by *total cycles* and *cycles per iteration*. Single-threaded performance is compared against dual thread implementations running on MI and MSI cache coherence protocols. For the purposes of this table, single-threaded performance is measured using the naive code on the single-core configuration (i.e. no `CONFIG=...`)

	vvadd	vvadd (opt)	matmul
one thread	36.7k cycles 36.7 cycles/iter	N/A	
two threads (MI)			
two threads (MSI)			

3 Directed Portion

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 Verilog cycle-accurate emulator of the dual-core processor (if the cache coherence needs to be changed), 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

To complete this lab you will log in to an instructional server (`icluster{6,7,8,9}.eecs`), which is where you will use Chisel and the RISC-V tool-chain.

First, clone the lab materials:

```
inst$ git clone ~cs152/sp19/lab5.git
inst$ cd lab5
inst$ ./init-submodules.sh
inst$ export LAB5ROOT=$PWD
```

We will refer to `~/lab5` as `${LAB5ROOT}` in the rest of the handout to denote the location of the Lab 5 directory. Some of the directory structure is shown below:

- `${LAB5ROOT}/`
  - test/riscv-bmarks Source code for benchmarks.
    - * common C code for common infrastructure.
    - * vvadd C code for the vector-vector add benchmark.
    - * matmul C code for the matrix multiply benchmark.
  - **verisim/** Verilator simulation tools and output files.
  - csrc/ Verilator test bench C++ source code.
  - vsrc/ Verilator test bench Verilog source code.
  - rocket-chip Rocket-Chip SoC generator.
    - * chisel/ The `Chisel` source code.
    - * rocket/ The Rocket processor.
    - * hardfloat/ The floating point unit source code.
  - src/ Top-level source code.

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/sp19/cs152.lab5.bashrc
```

To compile the cycle-accurate dual-core Rocket Verilog simulator, execute the following commands:

```
inst$ cd ${LAB5ROOT}/verisim
inst$ make clean; make
```

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

```
inst$ cd ${LAB5ROOT}/test/riscv-bmarks
inst$ make clean; make; make run
```

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

```
inst$ cd ${LAB5ROOT}/verisim
inst$ make run-benchmarks
```

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

---

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

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 vvadd directory, found in ${LAB5ROOT}/test/riscv-bmarks/. In the 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, vvadd.c holds 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.

A 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 on a dual-core CPU with MSI coherence policy:

```
inst$ cd ${LAB5ROOT}/verisim
inst$ make CONFIG=Lab5MSIConfig run-benchmarks
```

make will run both vvadd and matmul benchmarks on the emulator when changes are detected. The CONFIG= option tells the generator to use the configuration with two cores and MSI coherence.

You should get something similar to the following output, which corresponds to vvadd:

```
vvadd(...); barrier(nc): 37481 cycles, 37.4 cycles/iter, 8.2 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. You should also see performance statistics for a vvadd_opt() function, which will be followed by a FAILED message. This is because you have not implemented the optimized vvadd_opt() function yet!

For now, ignore the matmul statistics. *Record the non-optimal* vvadd() *function results.*

3.4 Recording non-optimal results

Delete the .riscv.out files generated by run-benchmarks, then run the benchmarks again, but using the MIDualCoreConfg. This will take about five minutes to build and about two minutes to run the benchmarks.

```
inst$ cd ${LAB5ROOT}/verisim
rm output/*.riscv.out
make CONFIG=Lab5MIConfig run-benchmarks
```

---

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

Record and report the results of `vvadd()` *using the MI protocol. Taking into account that the code is written for a dual-core, cache-coherent system, analyze the naive implementation and describe why it is sub-optimal.*

3.5   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}/test/riscv-bmarks/ vvadd/vvadd.c`.

Collect results of your `vvadd` *implementation for both the MI and MSI protocols. What did you do differently to get better performance over the provided* `vvadd` *function? You should be able to reduce the number of cycles per iteration to about 60% of the naive implementation under MSI and to less than half of the naive case under MI. Since Matrix Multiply has the potential for added data reuse over the naive implementation, potential performance benefits are much higher.*

VVADD Hints

You can now use `printf` to test your code, however it can only be executed from core 0. The current vvadd code prints the contents of `results_data` and `verify_data` if there is an error in `vvadd_opt`. You can compare these to each other to figure out what went wrong. 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 go into the `${LAB5ROOT}/Makefrag` and remove `matmul.riscv` from the `bmarks` variable. This will allow you to only run `vvadd` every time you call `make run`, which will speed up your development time.

4   Open-Ended Portion: Optimizing Multi-Threaded Matrix Multiply

For this problem, you will implement a multi-threaded implementation of matrix-matrix multiply. A naive implementation can be found in `${LAB5ROOT}/test/ riscv-bmarks/matmul/matmul.c`. Feel free to comment it out to save yourself simulation time. You will fill in your own in the provided functions `matmul_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`. Your results from the MI and MSI versions will be averaged together. Go crazy!

Collect results for your `matmul` *implementation for both the MI and MSI protocols. What did you do differently to get better performance over the naive* `matmul` *implementation?*

---

[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.

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 false sharing to make lines ping-pong between caches). Under the MI coherence protocol, it is also useful to avoid having both cores access the same portions of the input arrays at any given time, as there is no "S" state to accommodate shared lines.

5 Submitting Code

You will submit your `vvadd.c` and `matmul.c` files as part of the closed-ended and open-ended Gradescope assignments, respectively. Please do not modify any other files from the lab code for either portion. Instructions for submitting code will be posted to Piazza.

6 The Third Portion: Feedback

Please include any feedback you may have as part of your lab report.

7 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.