CS 152 Laboratory Exercise 5

Professor: Krste Asanović
TAs: Albert Ou and Jerry Zhao
Department of Electrical Engineering & Computer Sciences
University of California, Berkeley

April 15, 2021

Revision History

Revision	Date	Author(s)	Description
1.0	2021-04-15	Jerry Zhao	Initial release

1 Introduction and Goals

The goal of this laboratory assignment is to allow you to explore shared memory multiprocessor systems using the Chisel simulation environment. You will be provided a complete implementation of a dual-core Rocket processor supporting the RV64GC ISA. You will write multi-threaded C programs 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.

While students are encouraged to discuss solutions to the lab assignments with each other, you must complete the directed portion of the lab yourself and submit your own work for these problems. For the open-ended portion of each lab, students can either 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. You are free to participate in different groups for different lab assignments.

1.1 Graded Items

All code and reports are to be submitted through **Gradescope**. Please label each section of the results clearly. All directed items need to be turned in for evaluation.

- (Directed) Problem 3.3: Naive `vvadd` results for MSI
- (Directed) Problem 3.4: Naive `vvadd` results for MI and analysis
- (Directed) Problem 3.5: Optimized `vvadd` code, results, and analysis
- (Open-ended) Problem 4: Optimized `matmul` code, results, and analysis

- (Directed) Problem 5: Feedback

! → Lab reports must be written in *readable* English; avoid raw dumps of logfiles. **Your lab report must be typed, and the open-ended portion must not exceed ten (10) pages.** Charts, tables, and figures – where appropriate – are excellent ways to succinctly summarize your data.

2 Background

2.1 Dual-Core Rocket Processor

Rocket will be returning from Lab 2, but this time, there are two Rocket cores.

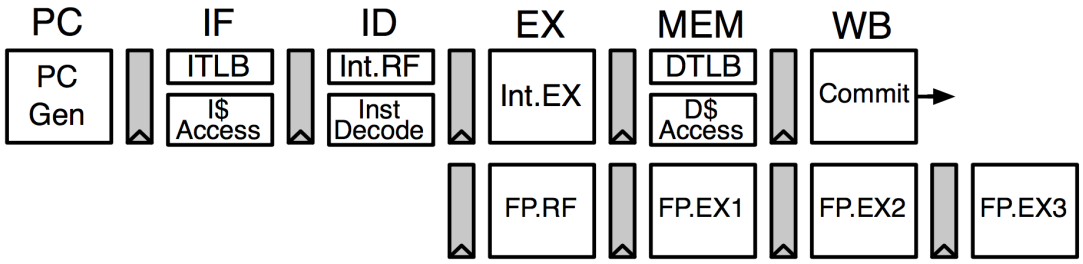

Figure 1: Rocket pipeline

Rocket is a 5-stage, single-issue, fully-bypassed, in-order RISC-V core. The configuations used in this lab implement the RV64IMAFDC instruction set variant[1], which refers to the 64-bit RISC-V base ISA (RV64I) along with a set of useful extensions [1]: M for integer multiply/divide, A for atomic memory operations, F and D for single- and double-precision floating-point, and C for 16-bit compressed representations of common instructions

Rocket also supports the RISC-V privileged architecture [2] with machine, supervisor, and user modes. It has an MMU that implements the Sv39 virtual memory scheme, which provides 39-bit virtual address spaces with 4 KiB pages. These processors are fully capable of booting mainstream operating systems such as Linux; however, no OS will be used in this lab, so code will run "bare metal" in M-mode.

2.2 Memory System

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

Each Rocket core has its own private L1 caches:

- 16 KiB 4-way set-associative L1 instruction cache
- 4 KiB 4-way set-associative L1 data cache

The data caches are kept *coherent* with one another.

[1] Also known as RV64GC, with G ("general-purpose") being the canonical shorthand for "IMAFD"

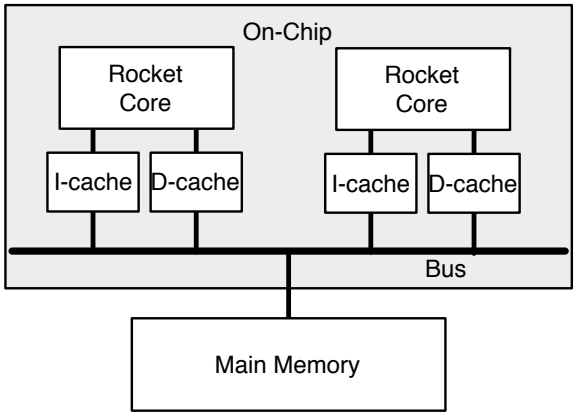

Figure 2: The dual-core Rocket system. A *logical* bus connects the caches and main memory. In practice, the bus is implemented as a crossbar with a coherence hub that arbitrates access to the "bus," initiates coherence "probe" traffic across the bus, and orchestrates the cache coherence protocol.

An off-chip memory provides the last level of the memory hierarchy. Both cores are connected via a bus to main memory, which is backed by a DRAM model that simulates the functional and timing behaviors of a DDR3 memory system. Only one agent may access the bus at a time.

Conceptually, cache coherence is maintained by having caches broadcast their intentions across the bus and "snooping", or monitoring, the actions of the other caches.

2.3 Multi-threaded Programming Environment

In most conventional multi-threaded programming environments, one thread begins execution at `main()`, which must then call some sort of spawn or `clone()` function to create more threads with assistance from the operating system.

In contrast, we will not be using an OS in this lab. Instead, *all* threads enter `main()` roughly simutaneously after a designated "boot" thread finishes initializing the C runtime environment. Each thread is provided with `ncores` (the number of cores in the system) and a `coreid` (a unique numerical identifier from 0 to `ncores` $-$ 1, inclusive).

2.3.1 Memory Allocation

You will need to be careful how you allocate memory in your code. Local variables can be allocated on the stack as usual; however, each thread is reserved only a limited amount of stack space. You will want to use the `static` keyword to allocate variables statically in the executable image, where it is visible to all threads.

There is also the `__thread` storage class keyword, which denotes a thread-specific variable that should be located in *thread-local storage* (TLS). TLS is a mechanism by which each thread is given its own private instance of the variable. It requires significant orchestration

between the linker and system libraries to work, but this complexity is largely transparent to user code.[2]

2.3.2 Synchronization Primitives

In the software framework, a `barrier()` function is provided to synchronize threads. Once a thread reaches the `barrier()` function, it waits until all threads in the system have reached the same `barrier()`. Implicit in the barrier is a memory fence. The `barrier()` function should probably be sufficient to implement any algorithm necessary in this lab.

For more information on the RISC-V memory ordering instructions, consult Section 2.7 of the user-level ISA manual [1]. Section 14 defines the RVWMO (RISC-V weak memory ordering) memory consistency model. Appendix A offers a more in-depth explanation of the rationale behind RVWMO.

The RISC-V `fence` instruction can be inserted in C code using the `__sync_synchronize()` GCC built-in function (saving you the hassle of inlining assembly). The GCC compiler provides more built-in functions for atomic memory accesses, such as `__sync_fetch_and_add()`.[3]

The `fence` instruction behaves as follows: If the data cache is not busy, the `fence` immediately retires, the pipeline continues execution. If the cache is busy servicing outstanding memory requests (i.e., cache misses), the `fence` stalls the pipeline until the cache is no longer busy. In this manner, the `fence` instruction ensures that all memory operations *before* the fence have completed before any memory operations *after* the fence are issued.[4]

2.3.3 Warnings and Pitfalls

- The stack space provided to each thread is only 24 KiB. As there is no virtual memory protection, there will be no warning if you overrun your stack. Try to allocate arrays and other large data structures statically.
- `printf()` can be used to debug your code. However, it is up to you to ensure that it is called by only one thread at a time; otherwise, the output may be incomprehensibly interleaved. Also, the `printf` implementation in this lab (provided by a stripped-down version of newlib, an embedded C library) does not support formatting floating-point types. You will have to cast them to integer types first. However, you will note that the randomly generated test vectors are actually using whole numbers for convenience.

[2] `http://people.redhat.com/drepper/tls.pdf`

[3] `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html`

[4] Finer-grained fences can be performed by setting the *predecessor* and *successor* fields in the instruction, which define which types of accesses (memory reads, memory writes, device reads, device write) should be ordered.

3 Directed Portion (25%)

3.1 General Methodology

This lab will focus on writing multi-threaded C code. This will be done in two steps:

1. Build the Verilog cycle-accurate emulator of the dual-core processor (if the cache coherence protocol needs to be changed)
2. Verify the correctness and measure the performance of your code on the cycle-accurate emulator

3.2 Setup

To complete this lab, `ssh` into an instructional server with the instructional computing account provided to you. The lab infrastructure has been set up to run on the `eda{1..8}.eecs.berkeley.edu` machines (`eda-1.eecs`, `eda-2.eecs`, etc.).

Once logged in, source the following script to initialize your shell environment so as to be able to access to the tools for this lab. Run it before each session.

```
eecs$ source ~cs152/sp21/cs152.lab5.bashrc
```

First, clone the lab materials into an appropriate workspace and initialize the submodules.

```
eecs$ git clone ~cs152/sp21/lab5.git
eecs$ cd lab5
eecs$ LAB5ROOT="$(pwd)"
eecs$ ./scripts/init-submodules-no-riscv-tools.sh
```

The remainder of this lab will use `${LAB5ROOT}` to denote the path of the `lab5` working tree. Its directory structure is outlined below:

```
${LAB5ROOT}
    lab/                                  Benchmark source code
        mt-vvadd-naive/                   Naive vvadd code
        mt-vvadd-opt/                     Optimized vvadd code
        mt-matmul-naive/                  Naive matmul code
        mt-matmul-opt/                    Optimized matmul code
    generators/                           Library of RTL generators
        chipyard/                         SoC configurations
        rocket-chip/                      Rocket Chip generator
        testchipip/                       RTL blocks for interfacing with test chips
        ...
    sims/
```

```
        verilator/                          Verilator simulation flow
        vcs/                                Synopsys VCS simulation flow
        ...
    tools/
        chisel3/                            Chisel hardware description library
        firrtl/                             RTL intermediate representation library
        barstools/                          Collection of common FIRRTL transformations
        ...
```

3.3 Measuring Vector-Vector Add with MSI Coherence

First, to acclimate ourselves to the Lab 5 infrastructure, we will gather the results of a poorly written implementation of `vvadd`, which performs a simple vector-vector addition.

Navigate to the `${LAB5ROOT}/lab/mt-vvadd-naive` directory, which has a few files of interest. First, `dataset.h` holds a static copy of the input vectors and expected results vector.[5] Second, `mt-vvadd_main.c` contains code for managing the benchmark, which includes initializing the state of the program, calling the `vvadd` function itself, and verifying the output of the function. Lastly, a very poor implementation of multi-threaded `vvadd` can be found in `mt-vvadd_naive.c`.

Build the simulator and run the `mt-vvadd-naive` benchmark on a dual-core configuration with an MSI coherence policy:

```
eecs$ cd ${LAB5ROOT}/sims/verilator
eecs$ make CONFIG=Lab5MSIDualRocketConfig run-mt-vvadd-naive
```

`make` will automatically rebuild the simulator and benchmarks programs if changes to the sources are detected. The `CONFIG` variable instructs the generator to use the configuration with two cores and MSI coherence.

You should see something similar to the following output for `mt-vvadd-naive`, which comes from timing a section of code that calls the `vvadd_naive()` function:

```
vvadd_naive: 37481 cycles, 37.4 cycles/iter, 8.2 CPI
```

! → *Record and report the suboptimal `mt-vvadd-naive` results with the MSI coherence protocol.*

3.4 Measuring Vector-Vector Add with MI Coherence

Run the `mt-vvadd-naive` benchmark again but using an MI coherence policy.

```
eecs$ cd ${LAB5ROOT}/sims/verilator
eecs$ make CONFIG=Lab5MIDualRocketConfig run-mt-vvadd-naive
```

[5] For rapid testing, you can generate your own input arrays that are a smaller size using `mt-vvadd_gendata.py`.

! → *Record and report the `mt-vvadd-naive` results with the MI coherence protocol. Taking into account that the code is executed on a multi-core cache-coherent system, analyze the naive implementation in `vvadd_naive()` and describe why it is suboptimal on MI and MSI.*

3.5 Optimizing Multi-Threaded Vector-Vector Add

Now that you know how to run benchmarks, gather performance results, and change the cache coherence protocol, you can now optimize `vvadd` for the dual Rocket cores.

Write your code in the `vvadd_opt()` function found in `${LAB5ROOT}/lab/mt-vvadd-opt/mt-vvadd_opt.c`, and then run the `mt-vv-addopt` benchmark as follows.

```
eecs$ cd ${LAB5ROOT}/sims/verilator
eecs$ make CONFIG=Lab5MIDualRocketConfig run-mt-vvadd-opt
eecs$ make CONFIG=Lab5MSIDualRocketConfig run-mt-vvadd-opt
```

! → *Collect results for your optimized implementation with both the MI and MSI protocols. What did you do differently to achieve better performance over the naive `vvadd`? You should be able to reduce the number of cycles per iteration to about 60% of the naive implementation under MSI.*

3.5.1 Submission

Use the following command to prepare your code for submission, and upload the resulting `mt-vvadd.zip` file to the Gradescope autograder.

```
eecs$ cd ${LAB5ROOT}/lab
eecs$ make zip-vvadd
```

Note that code outside of `mt-vvadd_opt.c` will be ignored.

3.5.2 Vector-Vector Add Tips

Refer back to Section 2.3.3 for potential pitfalls with programming in this bare-metal environment. Remember that you can use `printf()` for debugging, with caveats: Floating-point values are not supported, and make sure only one thread calls `printf()` at a time.

The benchmark prints the contents of the `output_data` and `verify_data` arrays if a mismatch is found.

To see what each core is doing cycle by cycle, look at the trace in `${LAB5ROOT}/sims/verilator/output/chipyard.TestHarness.*/mt-vvadd-opt.riscv.out`, where `*` is the chosen `CONFIG`. The output from core 0 and core 1 is prefixed with `C0:` and `C1:`, respectively. The disassembly in `${LAB5ROOT}/lab/mt-vvadd-opt.riscv.dump` may be useful for understanding the trace.

You can force `make` to repeat the simulation by manually removing the `mt-vvadd-opt.riscv.out` file from the simulator output directory.

If you encounter an error, you can first debug your code in the ISA-level simulator.

```
eecs$ cd ${LAB5ROOT}/lab
eecs$ make
eecs$ spike -p2 mt-vvadd-opt.riscv
```

The `-p` option sets the number of simulated hardware threads.

Note that `spike` is **not** a cycle-accurate processor model, and the "performance" numbers can be distorted since the hardware threads do not execute concurrently (unlike our actual system) but are switched after 5000 instructions. Also, this coarse-grained interleaving does not expose every race condition that would be possible on hardware.

4 Open-Ended Portion: Optimizing Multi-Threaded Matrix Multiply (75%)

For this problem, you will implement and optimize a multi-threaded version of matrix-matrix multiply.

A naive implementation can be found in `${LAB5ROOT}/lab/mt-matmul-naive/mt-matmul_naive.c`. First build and run the `mt-matmul-naive` benchmark to measure the baseline performance.

```
eecs$ cd ${LAB5ROOT}/sims/verilator
eecs$ make CONFIG=Lab5MIDualRocketConfig run-mt-matmul-naive
eecs$ make CONFIG=Lab5MSIDualRocketConfig run-mt-matmul-naive
```

Write your code in the `matmul_opt()` function found in `${LAB5ROOT}/lab/mt-matmul-opt/mt-matmul_opt.c`, and then run the `mt-matmul-opt` benchmark as follows.

```
eecs$ make CONFIG=Lab5MIDualRocketConfig run-mt-matmul-opt
eecs$ make CONFIG=Lab5MSIDualRocketConfig run-mt-matmul-opt
```

Once your code passes the correctness test, do your best to optimize its performance. Your results from the MI and MSI runs will be averaged together. Go crazy!

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

4.0.1 Submission

Use the following command to prepare your code for submission, and upload the resulting `mt-matmul.zip` file to the Gradescope autograder.

```
eecs$ cd ${LAB5ROOT}/lab
eecs$ make zip-matmul
```

Note that code outside of `mt-matmul_opt.c` will be ignored.

4.0.2 Matrix Multiply Tips

A number of strategies can be used to optimize your code for this problem. First, the problem size is for 32×32 matrices of `int` elements, with a total memory footprint of 12 KiB (the L1 data cache is only 4 KiB, 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 (to avoid false sharing that causes lines to 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 Feedback Portion

In order to improve the labs for the next offering of this course, we would like your feedback. Please append your feedback to your individual report for the directed portion.

- How many hours did you spend on the directed and open-ended portions?
- What did you dislike most about the lab?
- What did you like most about the lab?
- Is there anything that you would change?
- Is there something else you would like to explore in the open-ended portion?
- Are you interested in modifying hardware designs as part of the lab?

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

5.1 Team Feedback

In addition to feedback on the lab itself, please answer a few questions about your team:

- In a few sentences, describe your contributions to the project.
- Describe the contribution of each of your team members.
- Do you think that every member of the team contributed fairly? If not, why?

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.

References

[1] A. Waterman and K. Asanović, Eds., *The RISC-V instruction set manual, volume I: User-level ISA*, Version 20191213, RISC-V Foundation, Dec. 2019. [Online]. Available: `https://riscv.org/specifications/`.

[2] A. Waterman and K. Asanović, Eds., *The RISC-V instruction set manual, volume II: Privileged architecture*, Version 20190608-Priv-MSU-Ratified, RISC-V Foundation, Jun. 2019. [Online]. Available: `https://riscv.org/specifications/privileged-isa/`.