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 (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 matrix-matrix 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 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 Dual-core Rocket Processor

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

The Rocket Processor

Rocket will be returning from lab 4, 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 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).

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.

![Diagram of Rocket control processor pipeline](image)

Figure 1: The Rocket control processor pipeline.

In this lab, the Hwacha vector-thread unit has been removed; we will only be writing multi-threaded code.
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.](image)

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: Snoopy Caches” 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.

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 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 2.8 of the user-level ISA manual in the handouts section of the CS 152 website. The RISC-V `FENCE` instruction can be executed by calling `__synch_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

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 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 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` code, statistics, and answers
4. Problem 5: Feedback on this lab (emailed this time)

Attach your `vvadd` and `matmul` code at the end of your lab. You only need to provide code for their functions you modified. Feedback will be provided via email to your TA.
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.

<table>
<thead>
<tr>
<th></th>
<th>vvadd</th>
<th>vvadd (opt)</th>
<th>matmul</th>
</tr>
</thead>
<tbody>
<tr>
<td>one thread</td>
<td>18.7k cycles</td>
<td>n/a</td>
<td>886k cycles</td>
</tr>
<tr>
<td></td>
<td>18.6 cycles/iter</td>
<td></td>
<td>27.0 cycles/iter</td>
</tr>
<tr>
<td></td>
<td>4.0 CPI</td>
<td></td>
<td>4.0 CPI</td>
</tr>
<tr>
<td>two threads (MI)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>two threads (MSI)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 C++ 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 (t7400-{1,2,3,...,12}.eecs), which is where you will use Chisel and the RISC-V tool-chain.

First, download the lab materials:

```
inst$ cp -R ~cs152/Lab5 ./Lab5
inst$ cd ./Lab5
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.

---

1The capital “R” in “cp -R” is critical, as the -R option maintains the symbolic links used.
Some of the directory structure is shown below:

- `{LAB5ROOT}/
  - mt-bmarks Source code and execution environment for Lab 5 multi-threaded benchmarks.
  - riscv-rocket Directory for the dual-core Rocket processor.
    - test/ Source code for benchmarks and tests.
      - riscv-bmarks/ Benchmarks written in C.
      - riscv-tests/ Tests written in assembly.
    - emulator/ C++ simulation tools and output files.
      - Makefile/ Makefile for building the cycle-accurate C++ emulator.
    - chisel The Chisel source code.
    - hwacha The Hwacha source code.
    - hardfloat The floating point unit source code.
    - src/ The Rocket source code, including the settings for the cache coherence behavior.
    - 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:

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

For this lab, we will play with the benchmarks `vvadd` and `matmul`, found in the directory `{LAB5ROOT}/mt-bmarks/` (multi-threaded benchmarks).

To compile and run these benchmarks on the Chisel cycle-accurate emulator, execute the following commands:

```
inst$ cd `{LAB5ROOT}/mt-bmarks/
inst$ make clean; make
```

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 `vvadd` directory, found in `{LAB5ROOT}/mt-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.

---

2Or better yet, add this command to your bash profile.
and results vector. 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.

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 ${LAB4ROOT}/mt-bmarks/
inst$ make clean; make;
```

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 cycle-accurate, dual-core Rocket emulator. This will run both vvadd and matmul benchmarks.

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

```
vvadd(40, results_data, input1_data, input2_data); barrier(): 1088 cycles, 27.2 cycles/elt, 4.4 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 yet implemented the optimized vvadd-opt() function yet!

For now, ignore the matmul statistics.

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

**Changing the Cache Coherence Protocol**

Navigate to ${LAB5ROOT}/riscv-rocket/src/ and open the consts.scala file. Find the parameter ENABLE_SHARING and set it to false to set the cache coherence protocol to MI.

Now rebuild the emulator.

```
inst$ cd ${LAB5ROOT}/riscv-rocket/emulator/
inst$ make
```

This will take about five minutes. When finished, navigate back to ${LAB5ROOT}/mt-bmarks/ and rerun the benchmarks on the new emulator using the MI protocol. Record and report the results of vvadd().

Analyze the vvadd() code in ${LAB5ROOT}/mt-bmarks/vvadd/vvadd.c. Taking into consideration that the code is written for a dual-core cache-coherent system, what is sub-optimal about the provided implementation?

---

3You can generate your own input arrays that are a smaller size for rapid testing. See vvadd_gendata.pl for details.
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 mt-bmark/vvadd/vadd.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 at least double the performance over the provided vvadd function (or thereabouts).

VVADD Hints

You can now use printf to test your code, however it can only be executed from core 0. You can also use the provided printArray() function to look at the results data and compare it to the verify_data array. If you want to see what each core is doing cycle by cycle, you can go into the Makefile in the mt-bmark directory, and add the “-l” flag to the fesvr binary, which tells it to log data to stderr.\(^4\)

You may also want to go into the Makefile and remove matmul from the bmarks variable. This will allow you to only run vvadd every time you call make, which will speed up your development time.

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. 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

For this problem, you will implement a multi-threaded implementation of matrix-matrix multiply. You will write two versions: one version that targets the MI cache coherence protocol, and another version that targets the MSI cache coherence protocol.

A naive, single-threaded implementation can be found in mt-bmarks/matmul/matmul.c. Feel free to comment it out to save yourself simulation time. You will fill in your own in the provided function matmul(). 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. This will be a contest, with the best team, as measured by the fewest cycles, will receive a bonus +2 points on the lab. Your results from the MI and MSI versions will be averaged together. Go crazy!

Attach your matmul 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 32 elements on a side, with a total memory footprint of 12 kB (the L1

\(^4\)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.
data cache is only 4 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.

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 brand new lab, and as such, your TA would like your feedback one last time! Was this lab boring? Did you learn anything? Is there anything you would change? Do you prefer optimizing code for different coherence protocols, or would you rather implement coherence protocols in hardware?

Now that the semester is almost over, look back on the labs you have done this semester. Which was your favorite lab? Which was your least favorite? Which lab did you learn the most from? Are there any subjects covered in class that you wish was covered in lab?

Feel free to write as little or as much as you want (two points will be taken off only if left completely empty). Please email your feedback to your TA at celio at eecs (turn in the rest of the lab in paper format like usual).

6 Acknowledgments

This lab was made possible through the very 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.