

C152 Laboratory Exercise 3

Professor: John Wawrynek

TA: Martin Maas

Department of Electrical Engineering & Computer Science
University of California, Berkeley

October 14, 2016

1 Introduction and goals

The goal of this laboratory assignment is to allow you to conduct a variety of experiments in the `Chisel` simulation environment.

You will be provided a complete implementation of a speculative superscalar out-of-order processor. Students will run experiments on it, analyze the design, and make recommendations for future development. You can also choose to improve the design as part of the open-ended portion.

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 or the arguments you provide in support of your ideas.

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 those problems. 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.

You are only required to do one of the open-ended assignments. These assignments are in general starting points or suggestions. Alternatively, you can propose and complete your own open-ended project as long as it is sufficiently rigorous. If you feel uncertain about the rigor of a proposal, feel free to consult the TA or the professor.

1.1 Chisel Rocket-Chip, & The Berkeley Out-of-Order Machine

The `Chisel` infrastructure is much more advanced than what we saw in Lab 1 and 2. For this lab, we will use the same infrastructure that the Berkeley Architecture group (UCB-BAR) uses for their research. The infrastructure is composed of several components that are each maintained as separate git repositories and bundled (as git submodules) in a large main repository called Rocket-Chip (you don't have to worry too much about this setup: once you check out Rocket-Chip, all the submodules simply appear as folders).

Rocket-Chip is a system-on-a-chip (SoC) generator implemented in `Chisel`. It can generate processors, caches, accelerators, and connections off chip including memory channels. The default processor in Rocket-Chip is a single issue 5-stage pipeline processor called Rocket, the namesake of

the repository. In this lab, however, we will be using the RISC-V Berkeley Out-of-Order Machine, or “BOOM”. BOOM is heavily inspired by the MIPS R10k and the Alpha 21264 out-of-order processors [1, 3]. Like the R10k and the 21264, BOOM is a unified physical register file design (also known as “explicit register renaming”). BOOM can be configured to be a multi-issue processor (1,2,3,4 are the most commonly used widths).

In this lab, we will be using a Rocket Chip configuration that features a single BOOM core which is directly connected to an instruction cache and a non-blocking data cache, both of configurable size. These caches are backed by an L2 cache, which is connected to a large amount of simulated DRAM (located “off-chip”). We simulate DRAM using [2], an open-source simulator to calculate accurate DRAM timing numbers.

2 The BOOM Pipeline

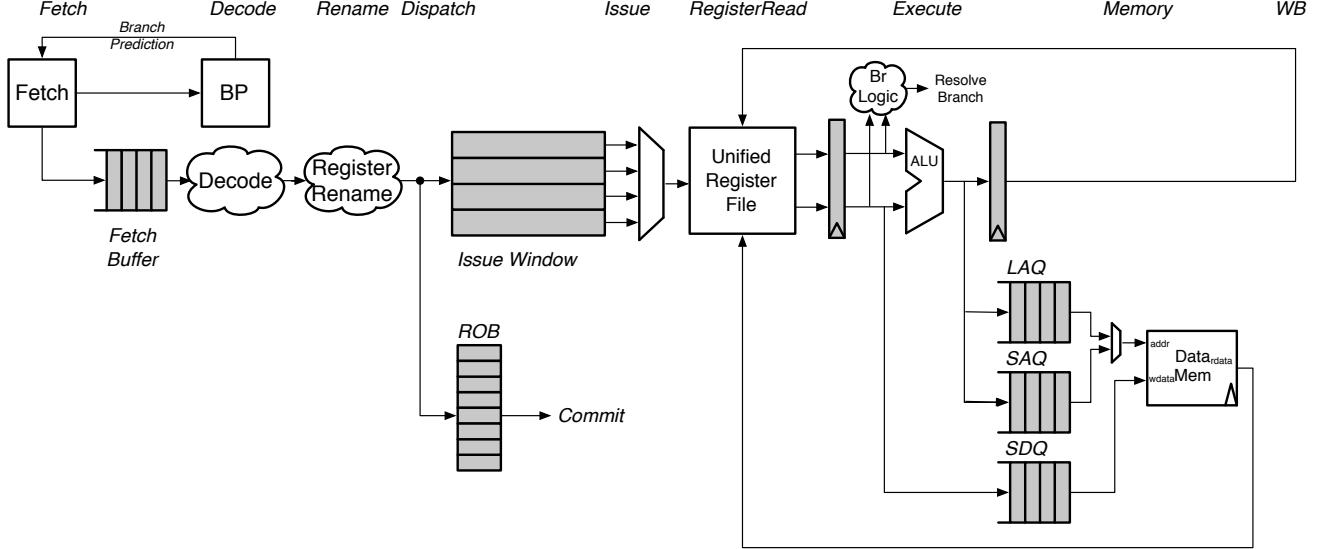


Figure 1: The Berkeley Out of Order Machine Processor.

Conceptually, BOOM is broken up into 10 stages: *Fetch*, *Decode*, *Register Rename*, *Dispatch*, *Issue*, *Register Read*, *Execute*, *Memory*, *Writeback*, and *Commit*. However, many of those stages are combined in the current implementation, yielding six stages: *Fetch*, *Decode/Rename/Dispatch*, *Issue/RegisterRead*, *Execute*, *Memory*, and *Writeback* (*Commit* occurs asynchronously, so I’m not counting that as part of the “pipeline”).

Here is a brief description of each of the BOOM Pipeline stages:

Fetch Instructions are *fetched* from the Instruction Memory and pushed into a FIFO queue, known as the *fetch buffer*.¹

Decode *Decode* pulls instructions out of the *fetch buffer* and generates the appropriate “micro-op” to place into the pipeline.

Rename The ISA, or “logical”, register specifiers are then *renamed* into “physical” register specifiers.

Dispatch The micro-op is then *dispatched*, or written, into the *Issue Window*.

Issue Micro-ops sitting in the *Issue Window* wait until all of their operands are ready, and are then *issued*.² This is the beginning of the out-of-order piece of the pipeline.

RF Read Issued micro-ops first *read* their operands from the unified physical register file (or from the bypass network)...

Execute ... and then enter the *Execute* stage where the functional units reside. Issued memory operations perform their address calculations in the *Execute* stage, and then store the calculated addresses in the Load/Store Unit which resides in the *Memory* stage.

Memory The Load/Store Unit consists of three queues: a Load Address Queue (LAQ), a Store Address Queue (SAQ), and a Store Data Queue (SDQ). Loads are fired to memory when their address is present in the LAQ. Stores are fired to memory at *Commit* time (and naturally, stores cannot be *committed* until both their address and data have been placed in the SAQ and SDQ).

Writeback ALU operations and load operations are *written* back to the physical register file.

Commit The Reorder Buffer, or ROB, tracks the status of each instruction in the pipeline. When the head of the ROB is not-busy, the ROB *commits* the instruction. For stores, the ROB signals to the store at the head of the Store Queue that it can now write its data to memory.

BOOM supports full branch speculation and branch prediction. Each instruction, no matter where it is in the pipeline, is accompanied by a branch tag that marks which branches the instruction is “speculated under”. A mispredicted branch requires killing all instructions that depended on that branch. When a branch instructions passes through *Rename*, copies of the *Register Rename Table* and the *Free List* are made. On a mispredict, the saved processor state is restored.

Although Figure 1 shows a simplified pipeline, BOOM implements the RV64G and privileged ISAs, which includes single and double-precision floating point, atomic memory support, and page-based virtual memory (so note that you will need to use the `riscv64-` compiler and tools).

Additional information on BOOM can be found in the online documents at <http://cclio.github.io/riscv-boom-doc/>.

¹While the fetch buffer is N entries deep, it can instantly read out the first instruction on the front of the FIFO. Put another way, instructions don’t need to spend N cycles moving their way through the *fetch buffer* if there are no instructions in front of them.

²More precisely, micro-ops that are ready assert their request, and the issue scheduler chooses which micro-ops to issue that cycle.

2.1 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:

1. Problem 3.2: Baseline CPI, branch predictor statistics, and answers
2. Problem 3.3: Issue window and width statistics and answers
3. Problem 4.1/4.2/4.3 modifications and evaluations
4. Problem 5: Feedback on this lab

3 Directed Portion (50% of lab grade)

The questions in the directed portion of the lab use **Chisel**. A tutorial (and other documentation) on the **Chisel** language can be found at (<http://chisel.eecs.berkeley.edu>). Although students will not be required to write **Chisel** code as part of this lab, students will need to write instrumentation code in C++ code which probes the state of a **Chisel** processor.

WARNING: **Chisel** is an ongoing project at Berkeley and continues to undergo rapid development. Any documentation on **Chisel** may be out of date, especially regarding syntax. Feel free to consult with your TA with any questions you may have, and report any bugs you encounter. Likewise, BOOM will pass all tests and benchmarks for the default parameters, however, changing parameters or adding new branch predictors will create new instruction interleavings which may expose bugs in the processor itself.

3.1 Setting Up Your Chisel Workspace

To complete this lab you will log in to an instructional server to run the RISC-V ISA simulator and compiler tool chain. We will provide you with an instructional computing account for this purpose.

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 need to 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/lab3-git lab3  
inst$ cd lab3  
inst$ export LAB3ROOT=$PWD
```

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

```
inst$ cd ${LAB3ROOT}
inst$ 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>)

The following command will set up your bash environment, giving you access to the entire CS152 lab tool-chain. Run it before each session:³

```
inst$ source ~cs152/fa16/cs152.lab3.bashrc
```

We will refer to `./lab3` as `${LAB3ROOT}` in the rest of the handout to denote the location of the Lab 3 directory.

The directory structure is shown below:

- `${LAB3ROOT} /`
 - `rocket-chip /` Top level rocket-chip directory.
 - * `boom /` `Chisel` source code for the BOOM processor.
 - * `chisel /` The source code of `Chisel` itself.
 - * `context-dependent-environments /` Library for the parameter system used in rocket-chip.
 - * `csrc /` Miscellaneous C code for rocket-chip.
 - * `dramsim2 /` A DRAM simulator that the `Chisel` emulator hooks into.
 - * `emulator /` C++ simulation makefile and generated source.
 - * `hardfloat /` `Chisel` code implementing various floating point functional units.
 - * `junctions /` `Chisel` code implementing converters for different interfaces used throughout rocket-chip.
 - * `LICENSE` Open source license for the code in rocket-chip.
 - * `Makefrag` High-level portion of a makefile that holds many basic options for other deeper Makefiles.
 - * `project /` Various scala, sbt magic configuration files.
 - * `README.md` Readme based on the full rocket-chip repository.
 - * `riscv-tools /` Toolchain for RISC-V. Only used for the tests in this lab.
 - * `rocket /` `Chisel` code implementing a well-tuned 5-stage in-order RISC-V processor.
 - * `sbt-launch.jar` SBT jar used to manage scala projects.
 - * `src /` `Chisel` code stitching together all of the other components into an entire chip.
 - * `uncore /` `Chisel` code implementing things outside the core, including L2 cache, and interface to the outside world.
 - `coremark /` Coremark benchmark suite binary and executable script.

³Or better yet, add this command to your bash profile.

To compile the `Chisel` source code for BOOM, compile the resulting C++ simulator, and run all tests and benchmarks, run the following Bash script:

```
inst$ cd ${LAB3ROOT}/rocket-chip/emulator
inst$ make run
```

To “clean” everything, simply use the “clean target of the Makefile:

```
inst$ make clean
```

The entire build and test process should take around 10 to 15 minutes on the icluster machines.⁴ In this lab, you will be experimenting with different configurations of BOOM. Each configuration is given a name in `${LAB3ROOT}/rocket-chip/src/main/scala/PrivateConfigs.scala`. In order to build a different configuration, you can use the `CONFIG` environmental variable. For example, you can build a smaller BOOM configuration through the following command:

```
inst$ make run CONFIG=SmallBOOMConfig
```

3.2 Gathering the CPI and Branch Prediction Accuracy of BOOM

For this problem, you will learn how to measure and collect the **CPI** and **branch predictor accuracy** of BOOM, and report the results for the benchmarks *dhrystone*, *median*, *multiply*, *qsort*, *towers*, *mm*, *spmv*, and *vvadd* (some of which you might remember from Lab 1). To execute these benchmarks on the default BOOM configuration (summarized in Table 1), run the following:

```
inst$ cd ${LAB3ROOT}/rocket-chip/emulator
inst$ make run
inst$ make stats
```

This configuration is designed to match a Cortex-A9 class processor.

Table 1: The BOOM Parameters for Problem 3.2.

	Default
Fetch Width	2
Issue Width	3
Register File	110 physical registers
ROB	48 entries
Inst Window	20 entries
Load/Store Queue	16 entries
Max Branches	8 branches
Branch Prediction	4KB GShare Predictor
BTB	on

⁴The generated C++ source code is \sim 10MB in size, so some patience is required while it compiles.

The Makefile is similar to the one you interacted with in Lab 1, which compiles the Chisel code into C++ code, then compiles that C++ code into a cycle-accurate simulator, and finally calls the RISC-V front-end server, which starts the simulator and runs a suite of benchmarks on the target processor. The `stats` target is generating `*.run` files which contain the number of cycles taken, the number of instructions completed and several microarchitectural counters that denote interesting events in BOOM. The microarchitectural counters are generated in `${LAB3ROOT}/rocket-chip/boom/src/main/scala/core.scala`. Take a look at this file and think about how you would calculate branch prediction accuracy and CPI from these numbers.

Branch Prediction in BOOM When looking at the code, you will see three kinds of updates that are propagated back to the branch predictor: `btb_update`, `bht_update` and `bpd_update`. This can be a bit confusing – we have a BTB and a small BHT that is used in conjunction with it and applies only to entries in the BTB (`btb_update`, `bht_update`). In addition, we also have a GShare predictor that represents the main (backing) predictor, or BPD (which is updated through `bpd_update`). Note that the BPD executes in parallel to the BTB but takes longer to complete. If the BTB hits and BTB and BPD disagree, the result from the BTB is used and overrides the PBD.

When calculating branch prediction accuracy, we are interested in the fraction of branches (not jumps, etc.) that turn out mispredicted once they are resolved in the execute stage. You should think carefully about which of the microarchitectural counters you need and how you can use them to calculate the prediction accuracy. You might need to carefully look through the BOOM code to understand what the different signals mean (and make sure to take both the BTB and BPD into account). **Write out the exact formula for calculating the branch prediction accuracy as a function of the different microarchitectural counters. Explain your thinking!**

Hint: To check whether your branch prediction accuracy numbers are correct, you can add additional uarch counters to measure the numbers of mispredicted branches using the `br_unit.brinfo` structure, which gives you detailed information about every branch that gets resolved in the execute stage. Your calculated numbers and the numbers given by `brinfo` should be similar to the ones you calculated from the given counters (but may be slightly off due to speculation). However, note that your final answer should not use any uarch counters other than the ones given.

Now that we know how to compute CPI and branch prediction accuracy, we are interested in how BOOM’s CPI compares with the in-order, 5-stage processor from Lab 1. Notice that BOOM is a 6-stage processor (with no bypassing enabled for this problem), so it can be most closely compared to the in-order, 5-stage with no bypassing (i.e., interlocked). Perform this comparison by completing Table 2. Explain the results you gathered and how you used them to compute the CPI. Are they what you expected? Was out-of-order issue an improvement on the CPI for these benchmarks? Explain what you think the reason is for why this was or wasn’t the case. Next, we will look at the impact of branch prediction on the CPI and measure the branch prediction accuracy of BOOM’s default branch predictor. Using the data collected from your experiments, calculate the branch prediction accuracy and start to complete Table 3. Next, to understand the impact that the branch predictor has on CPI and branch prediction accuracy, we will use a config that will turn off branch prediction entirely. The following configuration has no BHT or BTB:

```
inst$ make stats CONFIG=NoBPCOnfig
```

Table 2: CPI for the in-order 5-stage pipeline and the out-of-order “6-stage” BOOM pipeline, with and without branch prediction and/or branch target buffer. Fill in the rest of the table.

	dhry	mm	median	multiply	qsort	spmv	towers	vvadd
5-stage (interlocking)	1.98	N/A	1.71	1.90	2.78	N/A	1.36	1.69
5-stage (bypassing)	1.31	N/A	1.33	1.58	1.55	N/A	1.15	1.24
BOOM (PC+4)								
BOOM (BHT)								
BOOM (BTB+BHT)								

What happened to your computed branch prediction accuracy? Is this what you expected? What if we still had a BHT but no BTB (`CONFIG=NoBTBConfig`)? What happened to your computed branch prediction accuracy? Is this what you expected? Complete Table 2 and 3 with your results.

Table 3: Branch prediction accuracy for *predict PC+4* and a simple 2-bit BHT prediction scheme. Fill in the rest of the table.

	dhry	mm	median	multiply	qsort	spmv	towers	vvadd
BOOM (PC+4)								
BOOM (BHT)								
BOOM (BTB+BHT)								

Additional Notes: For the purpose of this lab, do not count jumps towards the branch predictor accuracy. Note that the CPI is calculated at the *Commit* stage. Finally, the branch predictor accuracy is calculated based on the signals in the *Execute* stage, which means that the reported accuracy is also including *misspeculated* instructions (the branch predictor itself is updated in the *Commit* stage).

3.3 Issue width limitations

Building an out-of-order processor is hard. Building an out-of-order processor that is well balanced and high performance is *really hard*. Any one piece of the processor can bottleneck the machine and lead to poor performance. For this problem, we will investigate how the machines issues instructions and some possible limitations. In order to gain insight into these issues, we will use a different set of counters than the previous question.

Open up `${LAB3ROOT}/rocket-chip/boom/src/main/scala/core.scala` again and change the commented lines to the second set. Look at the list of counters and their descriptions and make sure you understand what they signify.

First, we will compare the values of the counters for our baseline machine from the previous question, and a similar machine with a smaller and wider issue width.

```
inst$ make stats CONFIG=OneWideConfig
inst$ make stats CONFIG=FourWideConfig
```

Once we have a wider issue processor, we may be bottlenecked by other features, so we experiment with a larger reorder buffer (ROB):

```
inst$ make stats CONFIG=FourWideSmallROBConfig
inst$ make stats CONFIG=FourWideBigROBConfig
```

Another important factor for performance is how we determine which operation to issue next. BOOM support two methods: an unordered version that selects instructions simply based on their location in the issue window, and an age-based policy that selects the oldest instructions first.

```
inst$ make stats CONFIG=StaticIssueConfig
```

Finally, we look at the difference a larger or smaller issue window has with the age-based policy.

```
inst$ make stats CONFIG=SmallIssueConfig
inst$ make stats CONFIG=BigIssueConfig
```

Table 4: CPI for the in-order 5-stage pipeline and the out-of-order “6-stage” pipeline. Gradually turn on additional features as you move down the table. Fill in the rest of the table.

	dhry	mm	median	multiply	qsort	spmv	towers	vvadd
BOOM (default)								
BOOM (OneWideConfig)								
BOOM (FourWideConfig)								
BOOM (FourWideSmallROBConfig)								
BOOM (FourWideBigROBConfig)								
BOOM (StaticIssueConfig)								
BOOM (SmallIssueConfig)								
BOOM (BigIssueConfig)								
BOOM (SmallCustomConfig)								
BOOM (BigCustomConfig)								

Calculate CPIs from all these experiments in order to complete Table 4. Looking at the collected performance counters across these different configurations, do they change as you would expect? Why or why not? What other parameters in configs.scala that we did not change in this question do you think would have the largest impact on CPI, positive and negative? (pick two)

Finally, select one of `LSU_ENTRIES`, `PHYS_REGISTERS`, and `MAX_BR_COUNT` to increase or decrease in size. Make your modifications in two configurations, `SmallCustomConfig` and `BigCustomConfig` and list them. Look at `${LAB3ROOT}/rocket-chip/boom/src/main/scala/configs.scala` and `${LAB3ROOT}/rocket-chip/src/main/scala/PrivateConfigs.scala` for examples. How does the impact of these parameters compare to the impact of the other parameters we changed earlier?

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

4.1 Branch predictor contest: The Chisel Edition!

Currently, BOOM uses a GShare branch predictor by default. A version of this code with slightly simplified interfaces can be found in:

```
 ${LAB3ROOT}/rocket-chip/boom/src/main/scala/simplegshare.scala
```

For this problem, your goal is to implement a better branch predictor for BOOM. It is recommended to look at the following papers:

<https://www.cis.upenn.edu/~milom/cis501-Fall09/papers/Alpha21264.pdf>

<http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf>

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=491460

Consider implementing, a local predictor, a tournament predictor, or something else. The code has been commented with FIXME's in places you should easily be able to modify the code. Feel free to modify other places as well, but be aware that you may have less support doing so. You can also add print statements (an example is given) to investigate what the state of your predictor is during program execution.

Before you get started hacking, you should create a block diagram, at a similar level of detail to slides 21 and 24 from lecture 5. Include this block diagram with your write-up.

Describe the branch predictor(s) you decided to implement and fill in their entries in the table below. In this case, we are only interested in the accuracy of the BPD predictor and accuracy can therefore simply be calculated by $1 - (\text{uarch5} / \text{uarch1})$.

Use this approach to compute the numbers for the default configuration from question 3.2, and add them to the table below. To give you a comparison, the table also contains numbers collected for the yet to be released TAGE predictor. Compare your implementation(s) with these predictors. How well did you do? How much more state does your predictor require compared to the default GShare? Do you think your predictor would affect cycle time?

Table 5: Branch prediciton accuracy. Fill in the rest of the table.

	dhry	mm	multiply	qsort	towers	vvadd
BOOM (default)						
BOOM (TAGE)	99.7	65.6	90.6	65.4	91.2	89.5
BOOM (CustomBranchPredictorConfig)						

Please send your modified code as a .zip file in an email to the TA. We will be checking that it runs and generates the numbers you claim.

4.2 Branch predictor contest: The C++ Edition!

For this open-ended project, you will design your own branch predictor and test it on some realistic benchmarks. Changing the operation of branch prediction in hardware would be arduous, but luckily a completely separate framework for such an exploration already exists. It was created

for a branch predictor contest run by the MICRO conference and the Journal of Instruction-Level Parallelism. The contest provided entrants with C++ framework for implementing and testing their submissions, which is what you will use for our in-class study. Information and code can be found at: <http://www.jilp.org/cbp/>

A description of the available framework can be found in the README. The framework has been included in `${LAB3ROOT}/cbp/cbp-framework-version-3`. You can compile and run this framework on essentially any machine with a decently modern version of `gcc/g++`. So, while the TA will not be able to help you with setup problems on your personal machine, you may choose to compile and experiment there to avoid server contention.

In the interests of time, you can pick 3-5 benchmarks from the many included with the framework to test iterations of your predictor design on. A final rule: you can browse textbooks/technical literature for ideas for branch predictor designs, but don't get code from the internet.

For the lab report: Submit the source code for your predictor, an overall description of its functionality, and a summary of its performance on 3-5 of the benchmarks provided with the framework. Report which benchmarks you tested your predictor out on.

For the contest: We will take the code you submit with the lab, and test its performance on a set of benchmarks chosen by us. Please email your code in a .zip file to the TA.

4.3 BOOM Parameter Introspection With Software

The goal of this open-ended assignment is to purposefully design a set of benchmarks which stress different parts of BOOM. This problem is broken down into two parts:

- Write two benchmarks to stress the Load/Store Unit
- Write a benchmark (or benchmarks) to introspect a parameter within BOOM

4.3.1 Part 1: Load/Store Unit Micro-benchmarks

You may have noticed that many of the benchmarks do not use all of the (very complicated) features in the Load/Store Unit. For example, few benchmarks perform any store data forwarding. For this part, you will implement two (small) benchmarks, each attempting to exercise a different characteristic.

- Maximize store data forwarding
- Maximize memory ordering failures

As a reminder, “store data forwarding” is when a load is able to use the data waiting in the store data queue (SDQ) before the store has committed (there is a `store->load` dependence in the program). A memory ordering failure is when a load that depends on a store (a `store->load` dependence) is issued to memory before the store has been issued to memory - the load has received the wrong data. There is a set of uarch counters that you can enable to count these events.

There is no line limit for the code used in this problem. Each benchmark must run for at least twenty thousand cycles (as provided by the `SetStats()` printout).

Two skeleton benchmarks are provided for you in:

`${LAB3ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/lsu_forwarding/` and
 `${LAB3ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/lsu_failures/`

To build and test them under the RISC-V ISA simulator:

```
inst$ cd ${LAB3ROOT}/riscv-tools/riscv-tests/benchmarks/  
inst$ make run-riscv
```

Once you are satisfied with your code and would like to run it on BOOM, type:

```
inst$ cd ${LAB3ROOT}/rocket-chip/emulator/  
inst$ make run-bmark-tests
```

Finally, you can run a single benchmark with:

```
inst$ cd ${LAB3ROOT}/rocket-chip/emulator  
inst$ make output/lsu_forwarding.riscv.MediumBOOMConfig.run
```

Be creative! When you are finished, submit your code via zip attached to your email submission. In your report, discuss some of the ideas you considered, and describe how your final benchmarks work. Finally, it is possible that you may uncover bugs in BOOM through your stress testing: if you do, consider your benchmarking efforts a success! (save a copy of any offending code and let your TA know about any bugs you find).

4.3.2 Part 2: Parameter Introspection

Now the *real* challenge! Pick a non-binary parameter in BOOM's design and try to discover its value via a benchmark you design and implement yourself!

The basic strategy is as follows:

- Step 1) implement a micro-benchmark that stresses a certain parameter of the machine and measure the machine's performance.
- Step 2) go into `${LAB3ROOT}/rocket-chip/boom/src/main/scala/configs.scala` to change the parameter you are studying. The default config can be changed by changing `WithMediumBOOMs`. Then rerun your benchmark.
- Step 3) Repeat to gather more results.
- Step 4) Build a model to describe how performance is affected by modifying your parameter.

Your model should be good enough that the TA can take your model and benchmark, run it on a machine and discover the value of the parameter in question without knowing its value a priori (make sure that the TA can change other parameters of the machine, so your model is not simply a lookup table).

Here are a set of parameters to choose from:⁵

- ROB size
- Number of physical registers
- Maximum number of branches
- Number of issue slots
- Number of entries in the load and store queues
- Number of entries in the fetch buffer
- Number of entries in the BHT
- Data cache associativity

A skeleton benchmark is provided for you in:

```
 ${LAB3ROOT}/rocket-chip/riscv-tests/benchmarks/param_introspect/=
```

To build and test the benchmark under the RISC-V ISA simulator, use the same steps as in the first part. Submit your code, describe how it works, and what ideas you explored. Also submit your data and your model showing how well it works on BOOM.

Naturally, this is a challenging task. The goal of this project is to make you think very carefully about out-of-order micro-architecture and write code to defeat the processor. There may not necessarily be a “clean” answer here.

Warning: not all parameters are created equal. Some will be harder challenges than others, and we cannot guarantee that all parameters will be doable. But with a dose of cleverness, you might be surprised what you can discover! (especially when you can white-box test your ideas).

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 originally developed for CS152 at UC Berkeley by Christopher Celio, and partially inspired by the previous set of CS152 labs written by Henry Cook.

References

- [1] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, 1999.
- [2] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-june 2011.
- [3] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, 1996.

⁵You may not use cache size(number of sets) as a parameter, as that is too easy.