# C152 Laboratory Exercise 3

Professor: Krste Asanovic
TA: Christopher Celio
Department of Electrical Engineering & Computer Science
University of California, Berkeley

March 7, 2012

## 1 Introduction and goals

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

Using the Simics Microarchitectural Interface and an out–of–order execution processor model, you will collect statistics and make some architectural recommendations based on the results.

With regards to `Chisel`, you will be provided a complete implementation of a speculative 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 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.

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 professor.

This lab assumes you have completed the earlier laboratory assignments. However, we will re-include all the relevant files from past labs in this lab's distribution bundle for your convenience. Furthermore, we will assume that you remember all the commands used in earlier labs for controlling Simics simulation. If you feel any confusion about these points, feel free to consult the earlier labs or the Simics User Guide.

### 1.1 Simics MAI Overview

The Simics simulator by default assumes that every instruction completes instantaneously in a single cycle. This allows for speedy simulations that are useful for software/firmware correctness

testing. As we saw in previous labs, Simics can be extended with memory hierarchy timing modules to model realistic performance effects of a user-defined memory hierarchy. These extensions increase simulation realism at the expense of simulation speed.

In this lab, we will make use of further extensions which allow an instruction's execution to be delayed according to a microarchitectural model. This model can be programmed to simulate the timing behavior of the instructions as if they were being run on an in-order or out-of-order execution (OoO) processor. Microarchitectural models interact with Simics execution via the Micro-Architectural Interface.

Microarchitecturally accurate models are coded using C or the Simics Device Modeling Language. For this lab, we will use MAI modules included with Simics, rather than code our own. Specifically, we will use the MAI extension for the Sunfire UltraSPARC II processor (the Bagle machine). This extension allows for out-of-order execution, branch target speculation, and includes a memory hierarchy as well.

Several new variables are exposed to the user when working with MA models. The user can configure the width of the pipeline (the number of instructions allowed to fetch, execute, retire or commit in a single cycle), the size of the reorder buffer, whether instructions must retire in-order, and several memory hierarchy parameters related to OoO. The variables will be discussed as they are needed in the following lab sections.

When running in MA mode, Simics tracks dependencies of several varieties (register, control and memory) that exist in the instruction stream. It then places the instructions in a structure called an instruction tree — for the machine we are simulating, the instruction tree tracks the same state as the reorder buffer and associated structures would in an actual processor. This tree can be examined with the command `print-instruction-queue`. A load–store queue is also simulated.

The Simics microarchitecture simulator we will use in this lab speculates on branches by filling the instruction tree with instructions from both branch paths. Speculated instructions may only commit when their preceding branches have resolved. This behavior is notably different from the actual execution of many real out-of-order processors, but produces similar performance effects. The simulator we are using speculatively predicts branch target addresses.

The execution of instructions is divided into 6 stages (init, fetch, decode, execute, retire, commit) and instructions are only allowed to advance to the next stage when all applicable dependencies have been satisfied. In this way, instructions are allowed to execute out of order but may accumulate a multi-cycle delay appropriate to the underlying microarchitecture of the simulated processor.

A point worthy of note is that `steps` and `cycles` are no longer necessarily equivalent. A `step` occurs whenever an instruction commits. Advancing the simulation by one cycle may mean that multiple steps occur, or that none do. Similarly, advancing the simulation by one instruction may pause the simulation in the middle of a cycle. For this reason, it is advisable to use the `step-cycle` or `run-cycles` command to advance simulation, and then simply measure the number of steps that have occurred.

See the Simics MAI User Guide included with this lab for more information about any of these topics.

### 1.2 `Chisel` & The Berkeley Out–of–Order Machine

In addition to Simics, we will also be re-introducing `Chisel`. The infrastructure is nearly identical to Lab 1, with the addition of a new processor, 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, 2]. Like the R10k and the 21264, BOOM is a unified physical register file design (also known as "explicit register renaming"). BOOM is (currently) a single-issue processor.

## The BOOM Pipeline

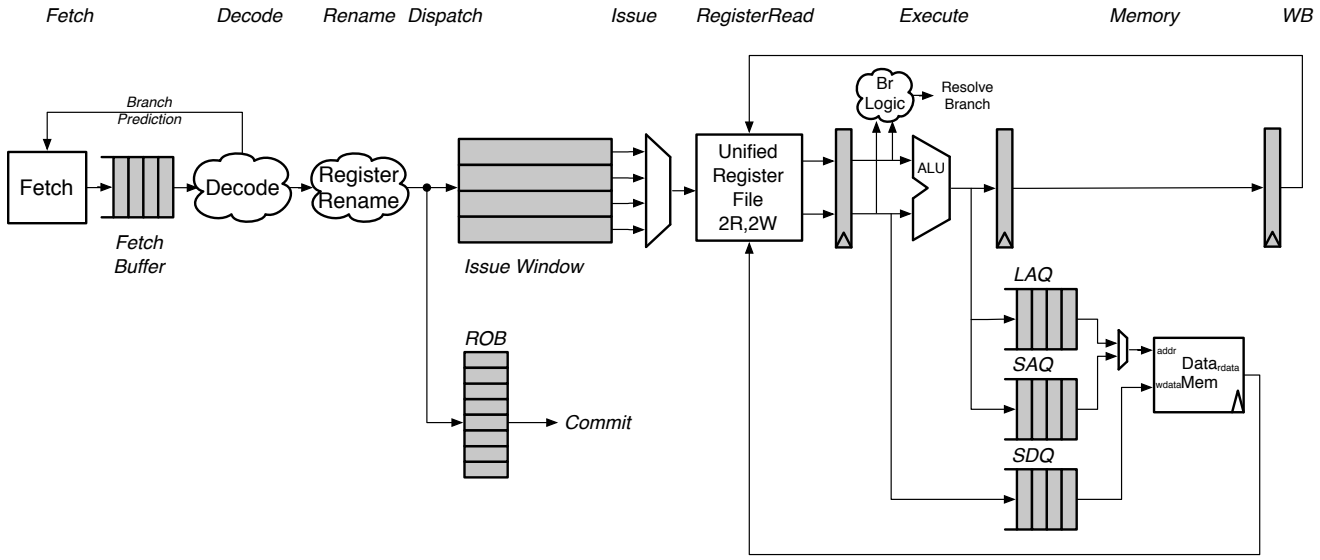


Figure 1: The Berkeley Out of Order Machine Processor.

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").

**Fetch** Instructions are *fetched* from the Instruction Memory and placed into a four-entry deep FIFO, 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.[2]

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

**Dispatch** The instruction is then *dispatched*, or written, into the *Issue Window*.

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

[1] While the fetch buffer is four-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 four cycles moving their way through the *fetch buffer* if there are no instructions in front of them.

[2] Because RISC-V is a RISC ISA, nearly all instructions generate only a single micro-op, with the exception of store instructions, which generate a "store address generation" micro-op and a "store data generation" micro-op.

**RF Read** Issued instructions first *read* their operands from the unified physical register file...

**Execute** and then enter the *Execute* stage where the integer ALU resides. 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 queue and does not conflict with any of the store addresses that the load depends on.[3] Stores are fired to memory at commit time, when both its address and its data are present.

**Writeback** ALU operations and load operations are *written* back to the physical register file.[4]

**Commit** The Reorder Buffer, or ROB, tracks the status of each instruction in the pipeline. When the head of the ROB is not-busy, it *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.

The *Decode* stage contains a Branch History Table, composed of simple $n$-bit history counters indexed by PC. On a predicted branch, the *Decode* stage kills the *fetch buffer* and redirects the *Fetch* stage. Otherwise, the *Fetch* stages fetches along PC+4.[5]

In this lab, BOOM implements a basic set of instructions from RV32. BOOM does not support sub-word memory accesses, floating point, or exceptions. Also, just like the simple pipelines from Lab 1, BOOM is connected to a magic, single-cycle memory (there are no caches or memory hierarchy). However, bypasses have been removed, meaning that the use-delay on back-to-back instructions is three cycles. Therefore, out–of–order issue is still an important component to good performance.

### 1.3 Graded Items

You will turn in a hard copy of your results to the professor or TA. Some of the open-ended questions also request emailing source code to the TA. Please label each section of the results clearly. The following items need to be turned in for evaluation:

[3] Technically, the load could bypass the data it needs out of the SDQ if it found a match in the SAQ. However, BOOM at this time does not support the bypassing of load data out of the SDQ. Problem 3.1 covers this issue in more detail.

[4] While BOOM is a single-issue processor, it does provide ALU operations and memory operations each their own write port, meaning the register file is a two-read, two-write register file (two different destinations can be written simultaneously).

[5] As of the writing of this handout, BOOM does *not* use a Branch Target Buffer, which would allow BOOM to redirect the PC in the *Fetch* stage.

1. Problem 2.3: Simics: CPI statistics for each benchmark and answers
2. Problem 2.4: Simics: CPI statistics for all configurations and answers
3. Problem 2.6: Chisel: CPI and branch predictor statistics and answers
4. Problem 2.7: Chisel: Issue Window statistics, and answers
5. Problem 2.8: Chisel: Issue Window statistics, instrumentation code, and answers
6. Problem 3.1/3.2/3.3/3.4/3.5 modifications and evaluations (include source code if required)
7. Problem 4: Feedback on this lab

# 2 Directed Portion

### Simics: Directed Portion

The first two questions of this lab will cover Simics. Utilize checkpoints and the scripts you wrote in Lab 2 to facilitate your data collection.

## 2.1 Simics: General Methodology

While you must ensure you are capturing a representative portion of the program's execution, you can measure instruction execution statistics whenever you like with `ptime` or logging.

For maximum efficiency, you should make sure that you are making use of all three operation modes of Simics. You only want to run in the slow, highly detailed modes when it is necessary to do so in order to collect accurate data.

The general methodology of this lab is:

1. Start Simics in fast mode, but use an MA-extended machine
2. Mount the host file system and load the appropriate files
3. Checkpoint the system
4. Restart Simics in stall mode and begin executing benchmark code to warm the caches
5. Checkpoint the system
6. Restart Simics in MA mode and collect OoO data

During this process Simics may report errors depending on which mode you are using and whether or not you are starting from a checkpointed simulation. If your simulation still runs after an error is reported, then simply disregard it.

You can use any of the `t7400-{1,2,3,...,12}.eecs` instructional servers to complete this lab assignment. Do not wait until the night before the assignment is due, because you will face resource contention that may significantly increase the time it takes to complete the assignment.

## 2.2 Setup

Start Simics in `-fast` mode, using the `targets/sunfire/bagle-ma-common.simics` script. Make sure the host filesystem or workspace is mounted and that the benchmark binaries and input files are copied into the target machine. To save time in the following sections, you should create a checkpoint that has all the files loaded, and probably a checkpoint at each of the initial magic breakpoints in the benchmark programs. Unfortunately, while the target machine being simulated is the same as in some previous labs, the underlying Simics modules used in the simulation are different (they use the MAI), so you will **not** be able to use checkpoints created in past labs.

## 2.3 Simics: Collecting CPI statistics using the MAI

In this section you will collect information on the instruction level parallelism inherent to the benchmark programs. You will do this by running the benchmarks on a simulated superscalar out-of-order processor and measuring the average number of instructions executed per cycle.

Remember to enable magic breakpoints. When you start from a checkpoint you may see an error relating to the `'last_cache'` component. Disregard this error.

For each benchmark:

- Start Simics in stall mode, and if you don't already have a checkpoint saved, run the benchmark program (`bzip_sparc, mcf_sparc, soplex_sparc`).
  ```
  host$ ./simics -stall -c bzip_bagle_files_loaded.conf
  simics> c
  target# ./<benchmark>_sparc input.<jpg/in/mps>
  ```
- It will reach a magic breakpoint and the simulation will pause.
- Run for at least 100,000,000 instructions to warm the cache. You can check its statistics the same way we did in Lab 2. By default for this lab, instruction accesses are instantaneous and not cacheable, and only data accesses are stored in the cache:
  ```
  simics> c 100_000_000
  simics> cache_cpu0.statistics
  ```
- Create a checkpoint. This will be useful to you for the remaining sections of the lab assignment.
  ```
  write-configuration bzip_bagle_warmed_caches.conf
  ```
- Start Simics in -ma mode. When you start from a checkpoint you may see an error relating to the `'last_cache'` component. Disregard this error. Set the OoO parameters to the desired values:
  ```
  host$ ./simics -ma -c bzip_bagle_warmed_caches.conf
  simics> ma_cpu0->fetches_per_cycle = 4
  simics> ma_cpu0->execute_per_cycle = 4
  simics> ma_cpu0->retires_per_cycle = 4
  simics> ma_cpu0->commits_per_cycle = 4
  simics> cpu0->reorder_buffer_size = 32
  ```
- Run for at least 10,000,000 **cycles** and count the number of instructions that commit in this time frame. The MAI-enabled processor automatically reports the number of steps that have

occurred every million cycles (this count is cumulative). The step count is incremented every time an instruction commits.
`simics> run-cycles 11_000_000`

- When you have collected enough data, halt Simics and proceed to the next benchmark.

For each benchmark, record the number of cumulative instructions executed in the span of cycles that you measured. What is the recorded CPI for each benchmark? Which benchmark had the best CPI, and which had the worst?

### 2.4 Collecting data about the effect of superscalar pipeline width on CPI

In this section, you will use the *bzip* benchmark and examine the effects of superscalar issue width on CPI for *bzip*. To do this, vary the parameters of the `ma_cpu0` object.

```
simics> ma_cpu0->fetches_per_cycle = <width>
simics> ma_cpu0->execute_per_cycle = <width>
simics> ma_cpu0->retires_per_cycle = <width>
simics> ma_cpu0->commits_per_cycle = <width>
simics> cpu0->reorder_buffer_size = 32
```

Vary all widths together though {1, 2, 4, 8, 16}, while keeping the reorder buffer size at 32. Then repeat with a reorder buffer size of 64. Are there diminishing returns on increasing pipeline width? How does reorder buffer size affect this performance? What factors might limit the effectiveness of increasing pipeline width?

### Chisel: Directed Portion

The next three questions in the directed portion of the lab use `Chisel`. A tutorial on the `Chisel` language can be found at (`http://www-inst.eecs.berkeley.edu/~cs152/sp12/handouts/chisel-tutorial.pdf`). 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.

### 2.5 Setting Up Your `Chisel` Workspace

To complete this lab you will log in to an instructional server, which is where you will use `Chisel` and the RISC-V tool-chain.

The tools for this lab were set up to run on any of the twelve instructional Linux servers `t7400-1.eecs`, `t7400-2.eecs`, ..., `t7400-12.eecs`. However, it *is* possible to download the `Chisel` lab directory to your Mac or Linux machine. The only requirement is that you install

the Scala Build Tool (SBT) locally on your machine (the instructions to do so are found in the `${LAB3ROOT}/README`). There are only two drawbacks: 1) you cannot compile your own RISC-V binaries[6], and 2) the compilation of the generated C++ code of BOOM can take significant amounts of memory.[7]

First, download the lab materials:[8,9]

```
inst$ cp -R ~cs152/Lab3 ./Lab3

inst$ cd ./Lab3
inst$ export LAB3ROOT=$PWD
```

We will refer to ./Lab3 as `${LAB3ROOT}` in the rest of the handout to denote the location of the Lab 3 directory (note: this only holds the `Chisel` part of this lab, not the Simics part!).

The directory structure is shown below:

- `${LAB3ROOT}`/
  - doc/ Useful documentation and related materials.
  - runall.sh Run this script to build BOOM and run all tests on it.
  - test/ Source code for benchmarks and tests.
    - riscv-bmarks/ Benchmarks written in C.
    - riscv-tests/ Tests written in assembly.
  - chisel The `Chisel` source code.
  - Makefile The high-level Makefile
  - src/
    - rv32_boom/ `Chisel` source code for the BOOM processor.
  - emulator/
    - common/Common emulation infrastructure shared between all processors.
    - rv32_boom/ C++ simulation tools and output files.
  - sbt/ `Chisel`/Scala voodoo. You can safely ignore this directory.

[6] Contact your TA if you would *really* like to compile your own RISC-V binaries locally. This requires access to the riscv-gcc source code and an enterprising individual who can install the riscv-gcc tool-chain without TA supervision.

[7] Regarding the computation and memory required in compiling the C++ simulator of BOOM, the entire build and test process should take no more than five minutes. However, if you are hitting the swap space on your local machine, it could last easily over an hour. On the t7400 machines, your TA has timed the entire build and test time as taking two and a half minutes. However, this requires using at least gcc 4.4, and keeping -debug off. Compiling under -O0 can also dramatically speed up compile time, at a huge hit to run-time performance.

[8] The capital "R" in "cp -R" is critical, as the -R option maintains the symbolic links used.

[9] The actual name of the Lab3 directory might have letters appended to it to denote different *versions*. Newer versions will be necessary as bugs are ironed out.

The following command will set up your bash environment, giving you access to the entire CS152 lab tool-chain. Run it before each session:[10]

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

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 ${LAB3ROOT}/
inst$ ./runall.sh
```

To "clean" everything, simply run the same script with an additional parameter:

```
inst$ ./runall.sh clean
```

### 2.6 Chisel: Gathering the CPI and Branch Prediction Accuracy of BOOM

For this problem, collect and report the CPI and branch predictor accuracy for the benchmarks *median, mix_manufacturing, multiply, qsort, towers,* and *vvadd*. You will do this twice for BOOM: with and without branch prediction turned on. First, turn off branch prediction as follows:

```
inst$ vim ${LAB3ROOT}/src/rv32_boom/consts.scala
```

Change the line `USE_BRANCH_PREDICTION` to be set to "false". Then compile the resulting simulator and run it through the benchmarks as follows:

```
inst$ cd ${LAB3ROOT}/
inst$ ./runall.sh
inst$ cd ${LAB3ROOT}/emulator/rv32_boom/
inst$ grep \# *.riscv.out
```

The script `runall.sh` drives the Makefile 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 tests and benchmarks on the target processor. The "grep" command is reading the *.out files and pulling out the Tracer statistics.

Do this again, but with branch prediction turned on (keep the branch predictor on for the rest of the lab).[11]

The default parameters for BOOM are summarized in Table 1. While some of these parameters (instruction window, ROB, LD/ST unit) are on the small side, the machine is generally well fed

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

[11] The branch predictor provided with BOOM is a branch history table made up of 128 two-bit counters, indexed by PC.

because it only fetches and dispatches one instruction at a time, memory is always one cycle away, and the pipeline is only six cycles long.[12]

Table 1: Default BOOM Parameters.

| | |
|---|---|
| Register File | 64 physical registers |
| Inst Window | 4 entries |
| ROB | 8 entries |
| LD Queue | 4 entries |
| ST Queue | 4 entries |
| Max Branches | 4 branches |

Table 2: CPI for the in-order 5-stage pipeline and the out-of-order "6-stage" pipeline. Fill in the rest of the table.

| | median | mix | multiply | qsort | towers | vvadd |
|---|---|---|---|---|---|---|
| 5-stage (bypassed) | 1.49 | 1.66 | 1.52 | 1.41 | 1.11 | 1.22 |
| 5-stage (interlocked) | 1.75 | 3.32 | 1.81 | 1.88 | 1.44 | 1.89 |
| BOOM (PC+4) | | | | | | |
| BOOM (BHT) | | | | | | |

Compare your collected results with the in-order, 5-stage processor. Notice that BOOM is a 6-stage processor (with no bypassing), so it can be most closely compared to the in-order, 5-stage with no bypassing (i.e., interlocked). Explain the results you gathered. Are they what you expected? Was out-of-order issue an improvement on the CPI for these benchmarks? Was using a BHT in the *Decode* Stage always a win for BOOM? Why or why not? (Don't forget to include the accuracy numbers of the branch predictor!).

**Additional Notes:** Jump and Jump-and-Link are predicted as *always taken*. 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 *mispeculated* instructions.[13]

**Warning:** the generated *.out files can become very large (a total of 650MB in size), and the Instructional machines may silently decide to stop writing the printout information to them. Although this lab should fit within your 800MB quota, you can also create a directory on /home/tmp, which has no quota but is also not backed up to tape. The program `/share/b/bin/mkhometmpdir` creates a directory for you at `/home/tmp/` (visit `http://inst.eecs.berkeley.edu/share/b/pub/disk.quotas` for details).

[12] Also, by keeping many of BOOM's data structures small, it keeps compile time fast and allows us to easily visualize the entire state on the machine when viewing the *.out files generated by simulation.

[13] Also, the branch predictor itself is updated in the *Execute* stage. This is an interesting design choice; one could easily have chosen to only update the branch predictor at *Commit* so the predictor is only learning "true" branches.

## 2.7 Chisel: Analyzing the Issue Window, Part I

BOOM currently only supports single-issue: all stages of the pipeline handle only a single instruction at a time. However, it is more than possible to implement an out-of-order processor that allows different stages to handle different amounts of instructions at a time (for example, committing two instructions at a time for a single-issue machine makes a considerable amount of sense. Why?[14]).

In fact, for this problem, your TA is wondering "Just how much performance is being left on the table by only allowing one instruction to be issued out of the *Issue Window* at a time?"

Your job is to quantify this, and answer your TA's question.

You will solve this question by writing C++ code in the "Out–of–Order Tracer" object that probes the state of BOOM every cycle. The OOOTracer object is found in `emulator/rv32_boom/oootracer.cpp/.h`. It is the same Tracer object you saw in Lab 1, with a few modifications. For this question, you will add any counters you need in the appropriate locations.[15] The main piece of your logic will go into the `Tracer_t::monitor_issue_window()` function. Read the instructions provided in `oootracer.cpp` for additional information. See Appendix A for details on how the *Issue Window* works.

To answer this question, **count the number of cycles in which at least two issue slots are requesting to be issued**. Make sure you are only counting cycles in which the StatsEnable co-processor register is asserted.

Some sample code is provided in `Tracer_t::monitor_issue_window()` to show how to detect that issue slot #0 is "valid".

For all six benchmarks, report how many cycles contain two instructions requesting to be issued. Do you think it would be beneficial to issue up to two instructions every cycle out of the issue window?

## 2.8 Chisel: Analyzing the Issue Window, Part II

Issuing two instructions simultaneously could be very expensive: it would require adding two more read ports and a *third* write port to the register file to handle the worst case of two ALU operations being issued and writing back in the same cycle that a load from memory comes back.

Instead, your TA proposes to issue two instructions simultaneously *if and only if* one instruction is an ALU operation and the second instruction is a memory operation. This will require adding a second ALU to perform address calculations, and an additional read port to read out the base address or store data required for load and store micro-ops.

To answer this question, augment your previous C++ probing code by checking the micro-op code, or "uopc" tag, on each issue slot (See Figure 2): **count the number of cycles in which at least one ALU micro-op and one memory micro-op requests to be issued**. The values of each "uopc" can be found in `src/rv32_boom/consts.scala` (roughly lines 172-210).

Consider any non-Load and non-Store to be an ALU operation, for the purposes of this question.

Report your results for the benchmarks, and attach your C++ code in an appendix of your lab report. Having collected data for Sections 2.7 and 2.8, what is your final recommendation on supporting multiple issue in BOOM? Is single-issue out of the *Issue Window* good enough, or would ALU/Mem dual-issue or even full dual-issue be worth the added costs?

[14] Solution to thought question: because waiting on hazards to resolve can back the machine up, potentially making the ROB commit the bottleneck.

[15] Grep for "Step".

# 3 Open-ended Portion

## 3.1 Analyzing the BOOM Load/Store Unit Design

You are a new employee at Processors-R-Us charged with analyzing the Load/Store Unit design of your company's latest offering. Under heavy pressure to make the looming tape-out deadline (contracts with your customers are pretty strict), the lead processor architect and your boss, Chris, decided to cut corners on the Load/Store Unit to make the shipping date: **the current design does *not* bypass load values out of the Store Data Queue (SDQ)**. Concerned with the potentially enormous performance loss, you decide to investigate.[16]

Probe the Load/Store Unit, in a manner similar to Question 2.7, to analyze how much performance is being left on the table by not bypassing load values from dependent stores. Monitor the Load Address Queue and track how often a load has a valid address waiting to be fired to memory, but is held up by a matching store address with valid data held in the SDQ. *Remember:* you do not know when to bypass store data to a load until the load address is valid, the store address is valid, and the store data is valid. Also, make sure you are only comparing against stores the load depends on (use the *store mask*).

Attach your code in an appendix of your lab report. Describe exactly *how* you are quantifying the performance lost, and make a case for your final recommendation: is it worth missing the shipping deadline for the improved performance? Try your best to quantify how much CPI is being lost.

See Appendix D and Figure 3 for more information on the Load/Store Unit.

## 3.2 Branch predictor contest: The Chisel Edition!

Currently, BOOM uses a simple Branch History Table of 128 two-bit counters; the same design used by the MIPS R10k (except the R10k used 512 entries). For this problem, your goal is to implement a better branch predictor for BOOM.

A good design to try is the Alpha 21264's "tournament" branch predictor[1]. It consists of three sets of n-bit counters; a "global" history predictor indexes a set of 2-bit counters using a global history register; a "local" history predictor that uses the PC to index a table of local history registers which are then used to index a set of 3-bit counters; and an "arbiter" predictor which indexes a table of 2-bit counters using the PC to predict whether the global predictor or the local predictor is more accurate.

The current branch predictor used by BOOM can be found in `src/rv32_boom/brpredictor.scala`. Feel free to modify the code in here, or better yet, make a copy of the file so you can compare your branch predictor with the default predictor.

Submit the resulting CPI of your predictor on all six benchmarks, a description of its overall design, and an explanation that summarizes its performance (i.e., when did it do well, when did it perform poorly, and why? What codes do you expect it to do well on? Etc.).

Also, attach the source code in your report *and* email your final `Chisel` code to your TA. The best team will get an extra 2 points on this lab!

**Note:** the nice thing about branch predictors is their correctness is only a secondary concern: their job is to output a single True/False signal, and the pipeline will handle cleaning up the mess! Corollary: if you see any tests or benchmarks fail, this is a bug in BOOM that is being uncovered

[16] This hypothetical is in no way auto-biographical.

by new instruction interleavings created by your branch predictor. Contact your TA if this occurs and carry on.

### 3.3 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 Simics 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://cava.cs.utsa.edu/camino/cbp2`

A description of the available framework can be found at:
`http://cava.cs.utsa.edu/camino/cbp2/cbp2-infrastructure-v2/doc/index.html`

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. You will only have to modify one .h file to complete the assignment! Just follow the directions at the above link.

Just like the original contest, we will allow your submissions to be in one of two categories (or both). The categories are realistic predictors (the size of the data structures used by your predictor are capped) or idealistic predictors (no limits on the resources used by your predictor). Even for realistic predictors, we are only concerned about the memory used by the simulated branch predictor structures, not the memory used by the simulator itself. Follow the original contest guidelines.

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. If you want to submit to the contest, make sure you leave **at least one** benchmark from the whole set that you **do not** test the predictor 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.

### 3.4 Create code that performs no better on an OoO machine

The goal of this open-ended assignment is to purposefully design code which does **not** perform any better when run on a superscalar out-of-order processor. Such code will demonstrate poor ILP, as shown by the measurable CPI. The goal is to have CPIs of the code on the out-of-order processor be as close as possible to the CPI of the code on the in-order processor.

You should compare the code when run on a 4–width OoO core (i.e. using the `targets/sunfire/bagle-ma-common.simics` machine) with the code when run on a single-issue in-order core (i.e. using the `targets/sunfire/bagle-gcache-common.simics` machine). However, make sure the parameters and configuration of the memory hierarchies are identical for both

machines!

There is no line limit for the code used in this lab. Your code must run for at least one million cycles, and it does not have to terminate. Remember to use the full Simics path on `/share/instsww/...` when compiling on the target machine.

Submit your source code, an explanation of how it operates and how it restricts ILP, and the record you made of the code's CPI on the in-order and out-of-order cores.

### 3.5 Collecting data about the limits of ILP

The goal of this open-ended assignment is to test the limits of ILP achievable for the three benchmarks included in the lab. As suggested in the directed portion of the lab, there are diminishing returns provided by increasing the width of the processor and the size of the reorder buffer. Your job for this project is to determine what these limits are using the same procedures applied in the directed portion of the lab. Use the OoO Bagle machine with a cache access delay of 1 and a memory access delay of 10. For each benchmark, make a recommendation of processor width and reorder buffer size, and provide as evidence data which demonstrate that your choice maximizes CPI while minimizing on-chip overhead.

## 4 The Third Portion: Feedback

This is a brand new lab, and as such, your TA would like your feedback again! This time though, I'm requesting answers for **both feedback sections**! Each section of feedback will be worth one point a piece (your response to each section should be at least one word. Obviously, the feedback is more valuable when more words are provided).

### 4.1 Feedback Part 1

Would you prefer more questions in the vein of Section 2.8 and Section 3.1, in which you analyze pieces of BOOM via the C++ test harness? Particularly if Simics is removed from the lab?

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

### 4.2 Feedback Part 2

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).

## 5 Acknowledgments

The Simics portion of this lab comes from a previous CS 152 lab written by Henry Cook.

# A Appendix: The Issue Window

Figure 2 shows a single issue slot from the *Issue Window*.[17]

Instructions (actually they are "micro-ops" by this stage) are *dispatched* into the *Issue Window*. From here, they wait for all of their operands to be ready ("p" stands for *presence* bit, which marks when an operand is *present* in the register file).

Once ready, the *issue slot* will assert its "request" signal, and wait to be *issued*. Currently, BOOM only issues a single micro-op every cycle, and has a fixed priority encoding to give the lower ID entries priority.

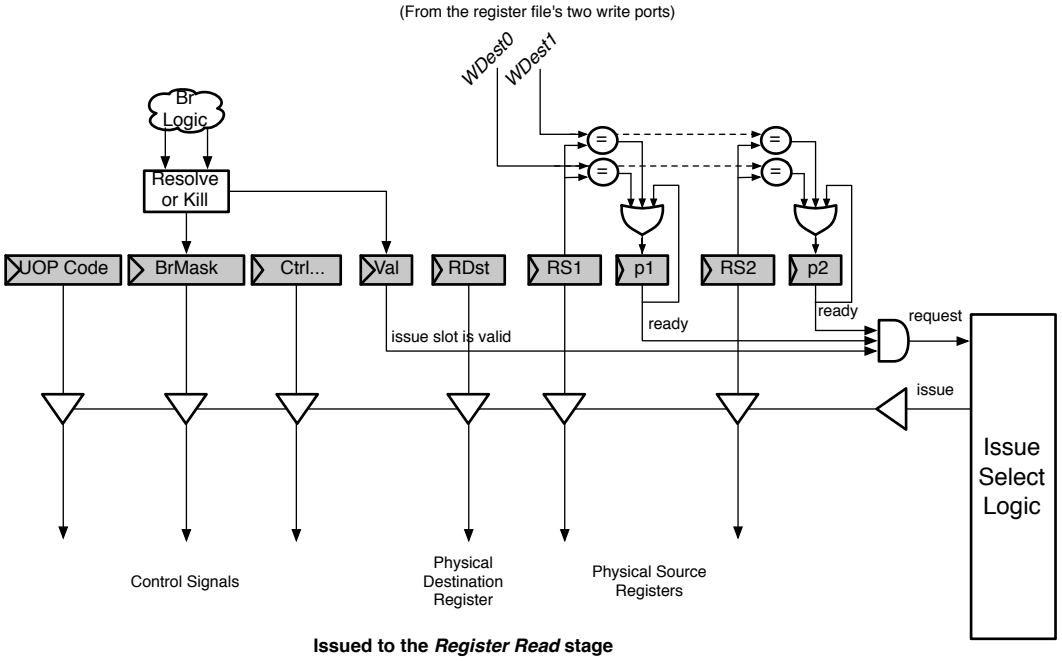


Figure 2: A single issue slot from the Issue Window.

[17] Conceptually, a bus is shown for implementing the driving of the signals sent to the *Register Read* Stage. In reality, for now anyways, BOOM actually uses muxes.

# B Appendix: The BOOM Source Code

The BOOM source code can be found in `{LAB3ROOT}/src/rv32_boom`.

The code structure is shown below:

- `rv32_boom/`
  - consts.scala All constants and adjustable parameters.
  - tile.scala The top-level module, instantiates memory and the CPU.
  - cpu.scala The top-level of the processor component.
  - mem.scala Single-cycle, single-port memory.
  - datapath.scala Main chunk of the BOOM datapath and control code.
  - brpredictor.scala Branch predictor. Uses a table of n-bit history counters.
  - rob.scala Re-order Buffer.
  - lsu.scala Load/Store Unit.
  - fifo.scala A FIFO queue. Used for the *Fetch Buffer*.
  - htif.scala The Host-Target Interface. Tells the outside world when a program finished successfully.
  - util.scala Utility code.
  - oracle*.scala Unused. Holds code for an "oracle" 1-stage processor.
  - cpath.scala Unused. All of this code got pushed to dpath.scala (because I'm a horrible person).
  - instructions.scala All RISC-V instruction definitions.

# C Appendix: How to Read Chisel Signals in the C++ Test-Harness Code

In this lab, we will be exercising the C++ tool-flow of `Chisel`(`Chisel` can also emit a Verilog version of a design). Often, whether for debugging purposes or for instrumentation, we will often want to probe the state of a Chisel design from the C++ test-harness.

As an example, let's probe the "micro-op opcode" signal ("uop code") that is stored in the *issue slot* of the *Issue Window* (see Figure 2). If we look through the `Chisel` code of BOOM, we see that the `IntegerIssueSlot` component is what describes the *issue slot*, and that it contains the variable `slot_uopc`. The `slot_uopc` signal is a `Reg` type, or *register*, and is written to on the positive-edge of the clock signal when the *issue slot*'s *write-enable* signal is asserted.

Each `IntegerIssueSlot` component is instantiated inside the `DatPath` component, which itself is instantiated inside the `Cpu` component, which in turn is instantiated inside the `Tile` component. When `Chisel` generates the resulting C++ code, the signal `slot_uopc` contains its entire parentage in its name-mangled C++ name.

## C.1 Finding the C++ Variable

The best way to find the C++ variable name for `slot_uopc` is to look through the generated C++ code in `${LAB3ROOT}/emulator/rv32_boom/generated-src/Tile.h`, which holds *all* Chisel signals. Grepping for `slot_uopc` we find the variables:

```
dat_t<8> Tile_cpu_d_IntegerIssueSlot_1__slot_uopc;
dat_t<8> Tile_cpu_d_IntegerIssueSlot_1__slot_uopc_shadow;
dat_t<8> Tile_cpu_d_IntegerIssueSlot_1__slot_uopc_shadow_out;
dat_t<8> Tile_cpu_d_IntegerIssueSlot_1__slot_uopc__prev;

dat_t<8> Tile_cpu_d_IntegerIssueSlot__slot_uopc;
dat_t<8> Tile_cpu_d_IntegerIssueSlot__slot_uopc_shadow;
dat_t<8> Tile_cpu_d_IntegerIssueSlot__slot_uopc_shadow_out;
dat_t<8> Tile_cpu_d_IntegerIssueSlot__slot_uopc__prev;
```

etc....

First, since there are four issue slots in BOOM by default, we will find 4 chunks of "slot_uopc" signals. `Chisel` will automatically add 1,2,3... to the component's name when it finds multiple instantiations of it.

Second, we see the full path name to `slot_uopc`: the top-level module is "Tile", followed by "cpu", "d" (for datapath), and finally "IntegerIssueSlot."

Third, we see additional versions of the `slot_uopc` variable: a `_shadow`, a `_shadow_out`, and a `_prev` version. You can safely ignore these variables.[18]

## C.2 Reading out the value from the C++ Variable

Although we have now found the variable we are interested in (`Tile_cpu_d_IntegerIssueSlot__slot_uopc`, `Tile_cpu_d_IntegerIssueSlot_1__slot_uopc`, etc.), we can see that it is of type `dat_t<8>`. This is a special templated class type that encapsulates all `Chisel` variables. In this case, it is describing an 8-bit wide value. The problem is we may occasionally want to describe variables of over 128 bits in our `Chisel` design, but natively C and C++ can only handle double the size of the native host machine's register. Thus, `Chisel` uses its own data-type class which maps to an array of `uint64_t` variables under the hood.

The important thing to know is that we can use the function `.lo_word()` to pull out the lowest 64-bits from a `dat_t<>` variable.

```
Tile_t *tile = new Tile_t(); // instantiate our Chisel design
uint64_t slot0_uopc = tile->Tile_cpu_d_IntegerIssueSlot__slot_uopc.lo_word();
uint64_t slot1_uopc = tile->Tile_cpu_d_IntegerIssueSlot_1__slot_uopc.lo_word();
...
etc.
```

[18] They exist because `slot_uopc` is a *register*. For example, on `clock_lo` the *shadow* version is updated, and on `clock_hi` the actual `slot_uopc` is updated from the `shadow` copy.

# D Appendix: The Load/Store Unit

The Load/Store Unit is responsible for deciding when to fire memory operations to the memory system. There are three queues: the Load Address Queue (LAQ), the Store Address Queue (SAQ), and the Store Data Queue (SDQ). Load instructions generate a "uopLD" micro-op. When issued, "uopLD" calculates the load address and places its result in the LAQ. Store instructions generate *two* micro-ops, "uopSTA" (Store Address Generation) and "uopSTD" (Store Data Generation). The STA micro-op calculates the store address and places its result in the SAQ queue. The STD micro-op moves the store data from the register file to the SDQ. Each of these micro-ops will issue out of the *Issue Window* as soon their operands are ready.

## D.1 Store Instructions

Entries in the Store Queue[19] are allocated in the *Decode* stage (the appropriate bit in the `stq_entry_val` vector is set). A "valid" bit denotes when an entry in the SAQ or SDQ holds a valid address or data (`saq_val` and `sdq_val` respectively). Store instructions are fired to the memory system at *Commit*; the ROB notifies the Store Queue when it can fire the next store. By design, stores are fired to the memory in program order.

## D.2 Load Instructions

Entries in the Load Queue (LAQ) are allocated in the *Decode* stage (`laq_entry_val`). In *Decode*, each load entry is also given a *store mask* (`laq_st_mask`), which marks which stores in the Store Queue the given load depends on. When a store is fired to memory and leaves the Store Queue, the appropriate bit in the *store mask* is cleared.

Once a load address has been computed and placed in the LAQ, the corresponding *valid* bit is set (`laq_val`). Once set, the load instruction will attempt to fire as soon as possible (getting loads fired early is a huge benefit of out–of–order pipelines). The load instruction compares its address with all of the store addresses that it depends on. The following scenarios can occur:

1. One of the dependent store addresses is not valid: The load must wait.
2. One of the dependent store addresses matches: The load must wait.
3. All dependent store addresses are valid, do not match: The load can fire.

This is the current BOOM load behavior. However, there are two sub-optimal decisions here: 1) for Situation # 2, loads *should* pull their data out of the corresponding Store Data Queue entry (this is what Question 3.1 is about) and 2) for Situation # 1, it can often be advantageous to speculate that there will be no conflicts between load and store addresses, and handle resetting the appropriate pipeline state on a mispeculate ("memory dependence speculation").

# References

[1] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, 1999.
[2] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, 1996.

[19] When I refer to the *Store Queue*, I really mean both the SAQ and SDQ.

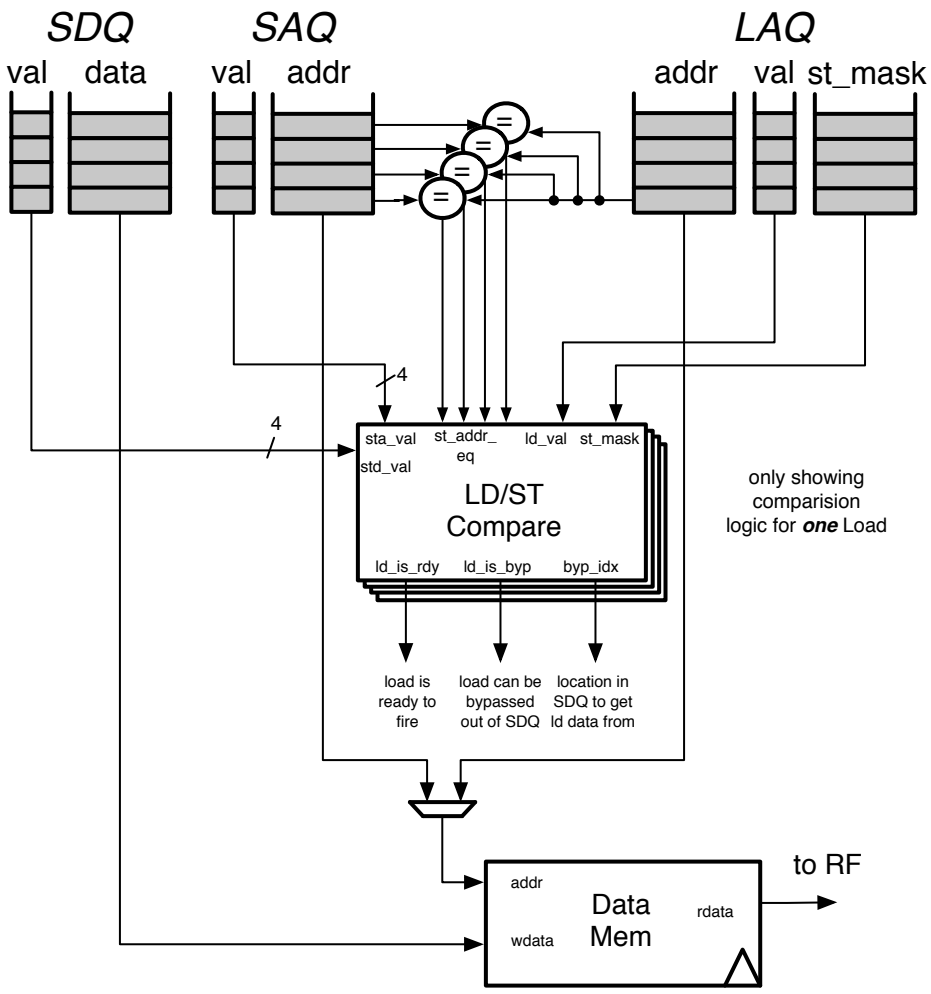


Figure 3: The Load/Store Unit. Shown is the comparison logic for *one* load. Notice that each load must compare itself against all other stores.