C152 Laboratory Exercise 3

Instructor: John Lazzaro
TA: Eric Love
Author: Christopher Celio
Department of Electrical Engineering & Computer Science
University of California, Berkeley

Apr 9, 2014

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 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. 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 the professor.

1.1 `Chisel` & The Berkeley Out–of–Order Machine

The `Chisel` infrastructure is very similar 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[**?**, **?**]. Like the R10k and the 21264, BOOM is a unified physical register file design (also known as "explicit register renaming"). BOOM is a superscalar processor; the width of its various stages is parameterizable.

The Chip

For this lab, you will be given an entire functioning processor, implemented in `Chisel`. The `Chisel` source code describes an entire "chip" with an interface to the outside world via a DRAM memory link. On-chip is an out–of–order core, which is where the focus of this lab will be. The core, in this case the BOOM processor, is directly connected to an instruction cache (16kB) and a non-blocking data cache (32kB). Any miss in either cache will require a trip to DRAM[**?**] (located "off-chip").

The BOOM Pipeline

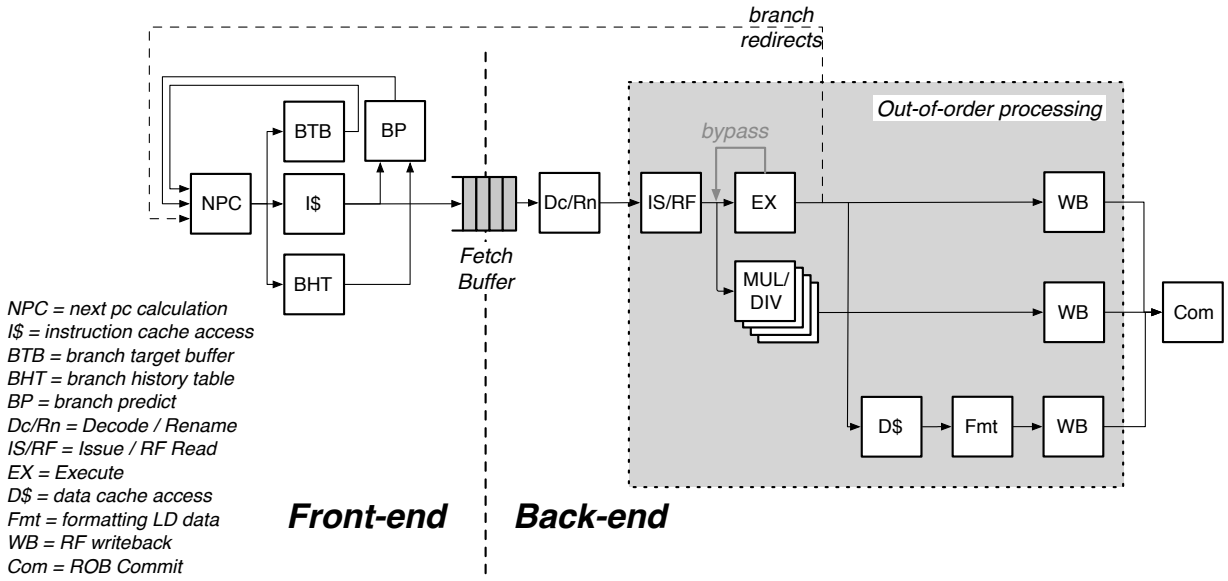

Figure 1: The Berkeley Out of Order Machine Processor. Shown is a single-issue version.

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).

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]

[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. In fact, BOOM's fetch buffer is a "flow-through queue" meaning if the queue is empty, the Decode stage can immediately operate on the instructions returning from the instruction cache without placing them in the queue first.

[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.

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.

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 the address is sent to the data cache (if it is a load) in which the data is accessed during the *Memory* stage. The calculated addresses are also written into the Load/Store Unit at the end of the *Execute* 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) (see Figure 4). Loads are optimistically fired to memory when their address is added to the LAQ during the *Execute* stage. In parallel, the incoming load compares its address against the SAQ to find if there are any store addresses that the load depends on. If the store data is present, the load receives the data from the store (*store data forwarding*) and the memory request is killed. If the data is not present, the load is put to sleep. Loads that are put to sleep are reissued to memory at a later time. The current policy in BOOM is to retry the load at the head of the LAQ .[3]

Stores are fired to memory at *commit*, 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. For loads, the ROB signals the Load/Store Unit to verify that the load did not fail a memory ordering dependence (i.e., a load issued before a store it depended on committed). If the load did fail, the entire pipeline must be killed and restarted. Exceptions are also taken at this point, which requires slowly unwinding the ROB to return the rename map tables to their proper state.

BOOM supports full branch speculation and branch prediction. Each instruction, no matter where it is in the pipeline, is accompanied by a branch mask 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

[3] A higher performance processor would allow loads to track *why* they were put to sleep, and to wake and retry loads once the issue has been resolved. This issue is explored further in Problem **??**.

[4] Even the single-issue version of BOOM provides 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). Also notice that in a *unified physical register file design*, speculative instructions are being written back.

Register Rename Table and the *Free List* are made. On a mispredict, the saved processor state is restored.

The "front-end" contains a Branch History Table, composed of simple n-bit history counters indexed by the PC. The BHT is read in parallel with instruction cache access. As an instruction is returned from the cache and inserted into the *fetch buffer*, the instruction is quickly checked to see if it is a branch. If the instruction is a branch, the prediction is used to redirect the *front-end* on a *TAKE BRANCH* prediction. BOOM also has a Branch Target Buffer (BTB) that directs the *next pc*.

BOOM implements the RISC-V variant RV64IMS[5]. RV64IMS is the 64-bit variant which provides the basic integer instructions, multiply/divide instructions, and the supervisor-level ISA. BOOM does *not* provide hardware support for floating point.

See Figure 4 for a more detailed diagram of the pipeline. Additional information on BOOM can be found in the appendices and the CS152 BOOM Section notes; in particular, the issue window, the load/store unit, and the execution pipeline are covered in greater detail.

[5] Atomic memory operations and floating point instructions are the main categories of instructions that BOOM does not implement.

1.2 Graded Items

Submit your lab report as a file `report.pdf` in the top level of the `lab3` directory in your private repo on github. Some of the open-ended questions also request source code - please also submit this to your git repo. Please label each section of the results clearly. The following items need to be turned in for evaluation:

1. Problem 2.2: CPI, branch predictor statistics, and answers
2. Problem 2.3: CPI statistics and answers
3. Problem 2.4: Issue Window statistics and answers
4. Problem 2.5: Issue Window statistics, instrumentation code, and answers
5. Problem `??/??/??/??/??/??` modifications and evaluations (open-ended portion, coming soon)

2 Directed Portion

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`).[6] 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.

2.1 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`.

[6] Chisel documentation can also be found within the lab itself. Look under `$Lab3Root/chisel/doc/` for more information.

First, download the lab materials into your private repo directory:[7,8]

```
inst$ cp -R ~cs152/sp14/lab3 ./lab3

inst$ cd ./lab3
inst$ export LAB3ROOT=$PWD
```

You should make sure to `git add` this directory structure and commit it to your repo now, before you generate lots of extra stuff while compiling the emulator

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}`/
  - doc/ Useful documentation and related materials.
  - csrc/ C++ simulation source code. The out-of-order tracer lives here, as does emulator.cc
  - bsrc/ Chisel source code for BOOM.
    - **consts.scala** BOOM configuration file. Parameters such as machine width, ROB size, bypassing-enable, etc.
  - emulator/Compiled emulator binaries and object files.
    - **Makefile** The Makefile you'll need to run experiments using the benchmarks
    - generated-src/ C++ code generated by Chisel for the emulator
      - **Top.h** Declarations of all Chisel signals in C++
    - **output/** results of `make run` and `make run-bmarks-test` (CPI etc.)
  - riscv-test/ Source code for benchmarks and tests.
    - benchmarks/ Benchmarks written in C.
      - lsu_forwarding/ A benchmark you can write for open-ended problem **??**.
      - lsu_failures/ A benchmark you can write for open-ended problem **??**.
    - riscv-tests/ Tests written in assembly.
  - install/ Install directory for tests and benchmarks that are visible to BOOM.
  - chisel/ The `Chisel` source code.
  - src/ Chisel code for interfacing BOOM with the surrounding simulator infrastructure
  - uncore/ `Chisel` source code to the uncore (i.e., outside of the "tile").
  - riscv-pk/ Repository of the RISC-V proxy kernel. Serves as the OS for BOOM.
  - dramsim2/ A DRAM simulator that the `Chisel` emulator hooks into.
  - sbt/ `Chisel`/Scala voodoo.
  - Makefile The high-level Makefile.

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

[8] 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 most interesting items have been bolded:**emulator/Makefile** to build and test the processor, the `Chisel` source code found in `bsrc`, and the output files found in `emulator/output/`.

The following command will set up your bash environment, giving you access to the entire CS152 lab tool-chain. Run it before each session:[9] [10]

```
inst$ source ~cs152/sp14/.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 ${LAB3ROOT}/emulator
inst$ make run
```

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

```
inst$ make clean
```

The entire build and test process should take around ten to fifteen minutes on the t7400 machines.[11] The first time you `make run`, you may have to wait a little longer while the Scala Build Tool downloads the `Chisel` compiler.

2.2 Gathering the CPI and Branch Prediction Accuracy of BOOM

For this problem, collect and report the **CPI** and **branch predictor accuracy** for the benchmarks *bubble, dhrystone, 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}/bsrc/consts.scala
```

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

```
inst$ cd ${LAB3ROOT}/emulator
inst$ make run-bmarks-test
inst$ make stats
```

The `make run` and `make run-bmarks-test` commands compile `Chisel` code into C++ code, and then compile theat C++ code into a cycle-accurate simulator. Finally, they call the RISC-V front-end server to start the simulator and run a suite of tests and/or benchmarks, respectively, on the target processor. The `make stats` command greps for `*` in the generated *.riscv.out files in `output/` to display the tracer's statistics. Try running `make run-bmarks-test` now.

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

[10] If you see errors about "htif_pthread.h", then you probably have an improperly set environment.

[11] The generated C++ source code is ∼5MB in size, so some patience is required while it compiles.

Then do it again, but with branch prediction turned on.[12]

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 because it only fetches and dispatches one instruction at a time, and the pipeline is not very long.[13]

Table 1: The BOOM Parameters for Problem 2.2.

	Default
Register File	64 physical registers
ROB	32 entries
Inst Window	12 entries
LD Queue	8 entries
ST Queue	8 entries
Max Branches	8 branches
Branch Prediction	128 two-bit counters
Issue loads ASAP	on
ALU Bypassing	**off**
BTB	**off**

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.

	bubble	dhry	median	mix	multiply	qsort	towers	vvadd
5-stage (interlocking)	1.90	n/a	2.42	3.95	2.00	2.24	1.56	2.75
5-stage (bypassing)	1.37	n/a	2.19	2.32	1.68	1.47	1.37	2.31
BOOM (PC+4)								
BOOM (BHT)								

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

	bubble	dhry	median	mix	multiply	qsort	towers	vvadd
BOOM (PC+4)								
BOOM (BHT)								

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

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

Compare your collected results with the in-order, 5-stage processor. 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). 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 always a win for BOOM? Why or why not? (Don't forget to include the accuracy numbers of the branch predictor!).[14]

Additional Notes: Jump-and-Link-Register (JALR) is included in the branch accuracy statistics. Jump-and-Link (JAL) is handled perfectly by the front-end[15], while JALR is always predicted as *not taken*.[16] 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.[17]

2.3 Bottlenecks to performance

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 of the processor can bottleneck the machine and lead to poor performance.

For this problem you will set the parameters of the machine to a low-featured "worst-case" baseline (see Table 4).

Table 4: BOOM Parameters: worst-case baseline versus "default" for the rest of the lab questions.

	Worst-case	Default
Register File	33 physical registers	64 physical registers
Branch Prediction	off	128 two-bit counters
Issue loads ASAP	off	on
ALU Bypassing	off	**on**

Begin by setting BOOM to the values in the "worst-case" column from Table 4. All of the necessary parameters can be found in `src/riscv-boom/consts.scala`.[18]

Run the benchmarks (`make run-bmarks-test; make stats`) to collect the data for the first row in Table 5. The performance should be dreadful. And when you see an error, don't panic.[19]

[14] Hint: when a branch is misspredicted for BOOM, what is the branch penalty?

[15] Once the JAL instruction returns from the instruction cache, the front-end can quickly decode it, calculate the target address, and redirect the PC. The JAL continues towards the backend where it is treated as a NOP, its execution already handled by the front-end!

[16] This is a sign that yours truly needs to build a *Return Address Stack* to properly predict JALR instructions.

[17] The branch predictor itself is updated in the *Commit* stage.

[18] The exact name of the variables, in order, are "PHYS_REG_COUNT", "USE_BRANCH_PREDICTOR", "ENABLE_SPECULATE_LOADS", and "ENABLE_ALU_BYPASSING".

[19] You will probably see `Dhrystone` throw an error. Upon inspecting `emulator/output/dhrystone.riscv.out`, you should see that `Dhrystone` timed out. This is normal (it just ran that slow!). The CPI measurements are still valid and you can move on with the rest of the experiment.

Table 5: 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.

	bubble	dhry	median	mix	multiply	qsort	towers	vvadd
5-stage (interlocking)	1.90	n/a	2.42	3.95	2.00	2.24	1.56	2.75
5-stage (bypassing)	1.37	n/a	2.19	2.32	1.68	1.47	1.37	2.31
BOOM (worst case baseline)								
BOOM (64 regs)								
BOOM (BHT)								
BOOM (fast loads)								
BOOM (bypassing)								

Now we will slowly add back the features we took away. For the 2nd row, return the physical register count to 64 registers (from 33), and rerun the benchmarks (thought problem: why is 33 registers the smallest allowed amount?)[20]

For the 3rd row, add back branch prediction. Then for the 4th row add back load speculation (without load speculation, the loads wait until *commit* to execute). And for the last row, enable ALU bypassing (the last row in the table should have all of the "default" values set).

Collecting this data is pretty straight-forward but admittedly time consuming (~10 minutes per row in the table), so *do* walk away from the computer, go outside, get coffee, or watch *Arrested Development* while your computer hums away. The idea here is to get a feel for the performance numbers when certain features are missing. But not to worry, the lab picks up very quickly in the next section!

2.4 Chisel: Analyzing the Issue Window, Part I

So far, you have been running BOOM in single-issue mode; 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?[21]).

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 `${LAB3ROOT}/csrc/ootracer.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.[22] The main piece

[20] Answer: the ISA has 32 registers, and you need one additional register to act as a temporary once you have allocated all of the ISA registers.

[21] Answer: because waiting on hazards to resolve can back the machine up, potentially making the ROB commit the bottleneck.

[22] Grep for "Step".

of your logic will go into the `Tracer_t::monitor_issue_window()` function. Read the instructions provided in `$LAB3ROOT/csrc/ootracer.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 "Tracer.paused" is not asserted. Some sample code is provided in `Tracer_t::monitor_issue_window()` to show how to detect that issue slot #0 is "valid".

Also, make sure to edit `consts.scala` and set `FETCH_WIDTH` equal to 2. The reason for doing this is that the limited fetch bandwidth of a one-wide machine will make the apparent benefit of dual-issue artificially small. Imagine, for example, a program containing only `addi` instructions with no dependencies. Each instruction will be removed from the issue window as soon as it is put there, and so there will never be more than one requesting instruction at any time. However, if instructions were fetched in pairs, it would become obvious that two instructions could be issued every cycle.

For all 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?

2.5 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/riscv-boom/consts.scala` (roughly lines 192-64).

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 submit your code via Github. Having collected data for Sections 2.4 and 2.5, 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?

2.6 Chisel: Analyzing the Issue Window, Part III

Now that you've analyzed the theoretical benefit of wider issue, you can see how much actual performance the current BOOM implementation manages to gain from going two-wide. Open up `consts.scala` once more and set `ISSUE_WIDTH` equal to two. Now run `make run` and look at the new CPIs. How much did performance actually improve?

3 Open-ended Portion

Coming soon!

4 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.

A Appendix: The Issue Window

Figure 2 shows a single issue slot from the *Issue Window*.[23]

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 uses a fixed priority encoding to give the lower ID entries priority.

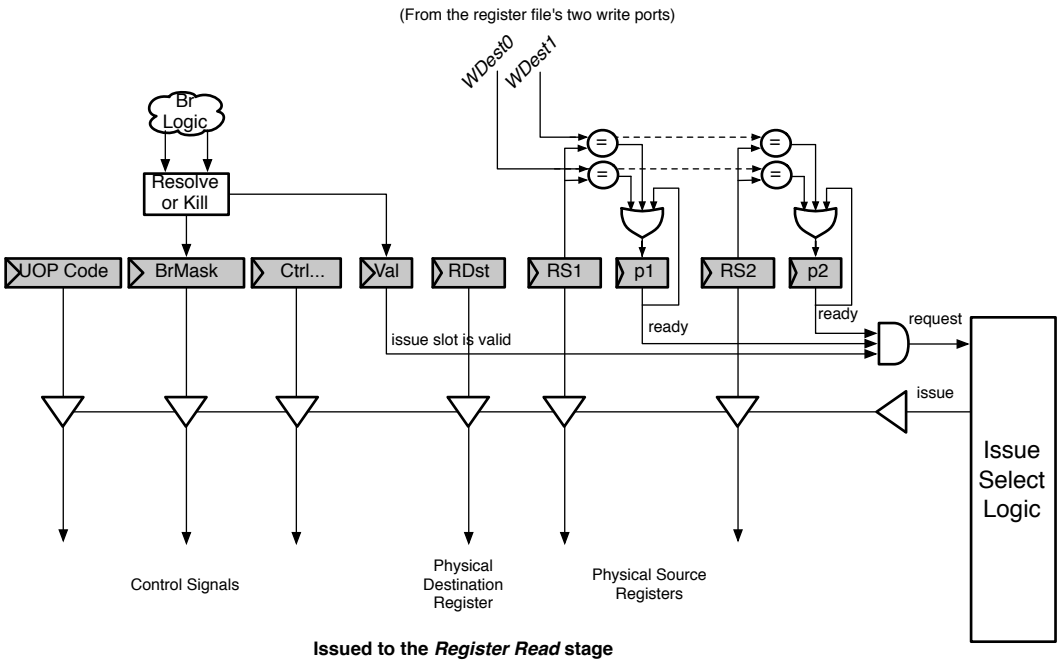

Figure 2: A single issue slot from the Issue Window.

[23] 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/riscv-boom`.

The code structure is shown below:

- `riscv-boom`/
  - consts.scala All constants and adjustable parameters.
  - top.scala The top-level module, instantiates the tile and links to the outside world.
  - tile.scala The tile, instantiates memory and the core.
  - core.scala The top-level of the processor core component.
  - icache.scala Instruction cache.
  - nbdcache.scala Non-blocking data cache.
  - datapath.scala Main chunk of the BOOM datapath and control code.
  - brpredictor.scala Branch predictor. Uses a table of n-bit history counters.
  - prefetcher.scala Data prefetcher.
  - decode.scala Decode table.
  - rename.scala Register renaming logic.
  - rob.scala Re-order Buffer.
  - lsu.scala Load/Store Unit.
  - dcachewrapper.scala Instantiates the DC and translates into OoO-speak.
- `common` /
  - util.scala Utility code.
  - instructions.scala All RISC-V instruction definitions.
  - htif.scala The Host-Target Interface. Tells the outside world when a program finished successfully.

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", or "uopc") 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 `slotUop`.[24] The `slotUop` is a `Chisel` "Bundle", or group of signals (think "struct" in C). We can see the definition of the "MicroOp" bundle in `dpath.scala`, near lines 50-100. One of the field variables within the bundle is `uopc`: this is the "micro-op opcode". Thus, the "uopc" within the *issue slot* would be "slotUop.uopc" in `Chisel`, or `slotUop_uopc` once it has been generated into `C++` code. The `slotUop` 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.

[24] The code for the issue slot can be found in `src/riscv-boom/dpath.scala`, near lines 200-300.

Each `IntegerIssueSlot` component is instantiated inside the `DatPath` component, which itself is instantiated inside the `Core` component, which in turn is instantiated inside the `SodorTile` component, which in turn is instantiated inside the `Top` component (phew!). When `Chisel` generates the resulting C++ code, the signal `slotUop_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 `slotUop_uopc` is to look through the generated C++ code in `${LAB3ROOT}/emulator/riscv-boom/generated-src/Top.h`, which holds *all* Chisel signals. Grepping for `slotUop_uopc` we find the variables:

```
dat_t<9> Top_SodorTile_core_d_IntegerIssueSlot_1__slotUop_uopc;
dat_t<9> Top_SodorTile_core_d_IntegerIssueSlot_1__slotUop_uopc_shadow;

dat_t<9> Top_SodorTile_core_d_IntegerIssueSlot__slotUop_uopc;
dat_t<9> Top_SodorTile_core_d_IntegerIssueSlot__slotUop_uopc_shadow;
```

etc....

First, since there are four issue slots in BOOM by default, we will find 4 chunks of "slotUop_uopc" signals. `Chisel` will automatically add 1,2,3... to the component's name when it finds multiple instantiations of it (but sadly not "0" to the first one[25]).

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

Third, we see an additional version of the `slotUop_uopc` variable: a `_shadow` version. You can safely ignore these variables.[26]

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

Although we have now found the variable we are interested in (`Top_SodorTile_core_d_IntegerIssueSlot__slotUop_uopc`, `Top_SodorTile_core_d_IntegerIssueSlot_1__slotUop_uopc`, etc.), we can see that it is of type `dat_t<9>`. This is a special templated class type that encapsulates all `Chisel` variables. In this case, it is describing an 9-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.

```
Top_t *tile = new Top_t(); // instantiate our Chisel design
uint64_t slot0_uopc = tile->Top_SodorTile_core_d_IntegerIssueSlot__slotUop_uopc.lo_word();
uint64_t slot1_uopc = tile->Top_SodorTile_core_d_IntegerIssueSlot_1__slotUop_uopc.lo_word();
```

etc...

[25] Yes, I've complained to them about this.

[26] They exist because `slot_uopc` is a *register*. On `clock_lo` the *shadow* version is updated, and on `clock_hi` the regular version is updated to the *shadow* version's value.

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[27] 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`).

Loads are optimistically fired to memory on arrival to the LSU (getting loads fired early is a huge benefit of out–of–order pipelines). Simultaneously, the load instruction compares its address with all of the store addresses that it depends on. If there is a match, the memory request is killed. If the corresponding store data is present, then the store data is *forwarded* to the load and the load marks itself as having *succeeded*. If the store data is not present, then the load goes to *sleep*. Loads that have been put to sleep are woken up later. Once the sleeping load is at the head of the LAQ, it is retried.[28]

D.3 Memory Ordering Failures

The Load/Store Unit has to be careful regarding `store->load` dependences. For the best performance, loads need to be fired to memory as soon as possible.

```
sw x1 -> 0(x2)
ld x3 <- 0(x4)
```

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

[28] Higher performance processors will track *why* a load was put to sleep and wake it up once the blocking cause has been alleviated.

However, if `x2` and `x4` reference the same memory address, then the load in our example *depends* on the earlier store. If the load issues to memory before the store has been issued, the load will read the wrong value from memory, and a *memory ordering failure* has occurred. On an ordering failure, the pipeline must be flushed and the rename map tables reset. This is an incredibly expensive operation.

To discover ordering failures, when a store address is computed and enters the SAQ, it checks the entire LAQ for any address matches. If there is a match, the store checks to see if the load has *executed*, and if it got its data from memory or if the data was forwarded from an older store. In either case, a memory ordering failure has occurred. The ordering failure information is sent to the ROB and the load is marked as having thrown an exception (a memory ordering failure is treated as an exception, as it requires clearing the pipeline, resetting the map tables, and retrying the load).

See Figure 3 for more information about the Load/Store Unit.

E Appendix: Adding a New Benchmark

For some of these questions, you will either need to create new benchmarks, or add existing benchmarks to the build system.

E.1 Creating a new benchmark

The source code for all benchmarks can be found in `test/riscv-bmarks/`. Each benchmark is given its own directory. To create a new benchmark, it is easiest to copy an existing benchmark directory.

```
inst$ cp -r vvadd my_new_bench
inst$ cd my_new_bench; ls
bmark.mk dataset1.h vvadd_gendata.pl vvadd_main.c
```

First, open `bmark.mk`. You will want to find and replace all instances of "vvadd" with "my_new_bench". If your benchmark requires multiple C files, add them to `vvadd_c_src`. Likewise, any assembly files can be added to `vvadd_riscv_src`. The build system should be able to take care of actually building and linking your different source files.

The `vvadd` benchmark has a Perl script `vvadd_gendata.pl` to generate a random input set of arrays, which are stored in `dataset1.h`. This removes the processor from having to generate and test its own input vectors. You can safely ignore these files (or repurpose them for your own use).

The `vvadd_main.c` file holds the main code for `vvadd`. Rename this file to fit the file name declared in `bmark.mk` (probably `my_new_bench_main.c`). Now you can add your own code in here. You can delete pretty much everything except two vital functions: `setStats(int)` and `finishTest(int)`. The `setStats(int)` turns on and off the statistic tracking uses by `oootracer.*`. *Note: setStats() should only be called twice in your benchmark, as only the last set of data will be stored by oootracer.**. The `finishTest(int)` function will notify the proxy-kernel we are finished and stop the emulation. Pass in "1" if the test was a success, and ">1" for the error code.

E.2 Adding a benchmark to the build system

Once you are happy with your new benchmark, you need to modify two `Makefile`s. First, open `test/riscv-bmarks/Makefile`, and find the `bmarks` variable. Add "my_new_bench" to the listing. You can now build your benchmark and test it on the ISA simulator.

```
inst$ make; make run;
```

Once you are satisfied, you must "install" the benchmark to `install/riscv-bmarks`. This is where BOOM looks for benchmarks to run.

```
inst$ make install
```

One final `Makefile` modification is required. Open `emulator/common/Makefile.include` and find the variable `global_bmarks`. Add your benchmark to this variable as well. Now running the top-level "runall.sh" script will run your new benchmark on BOOM!

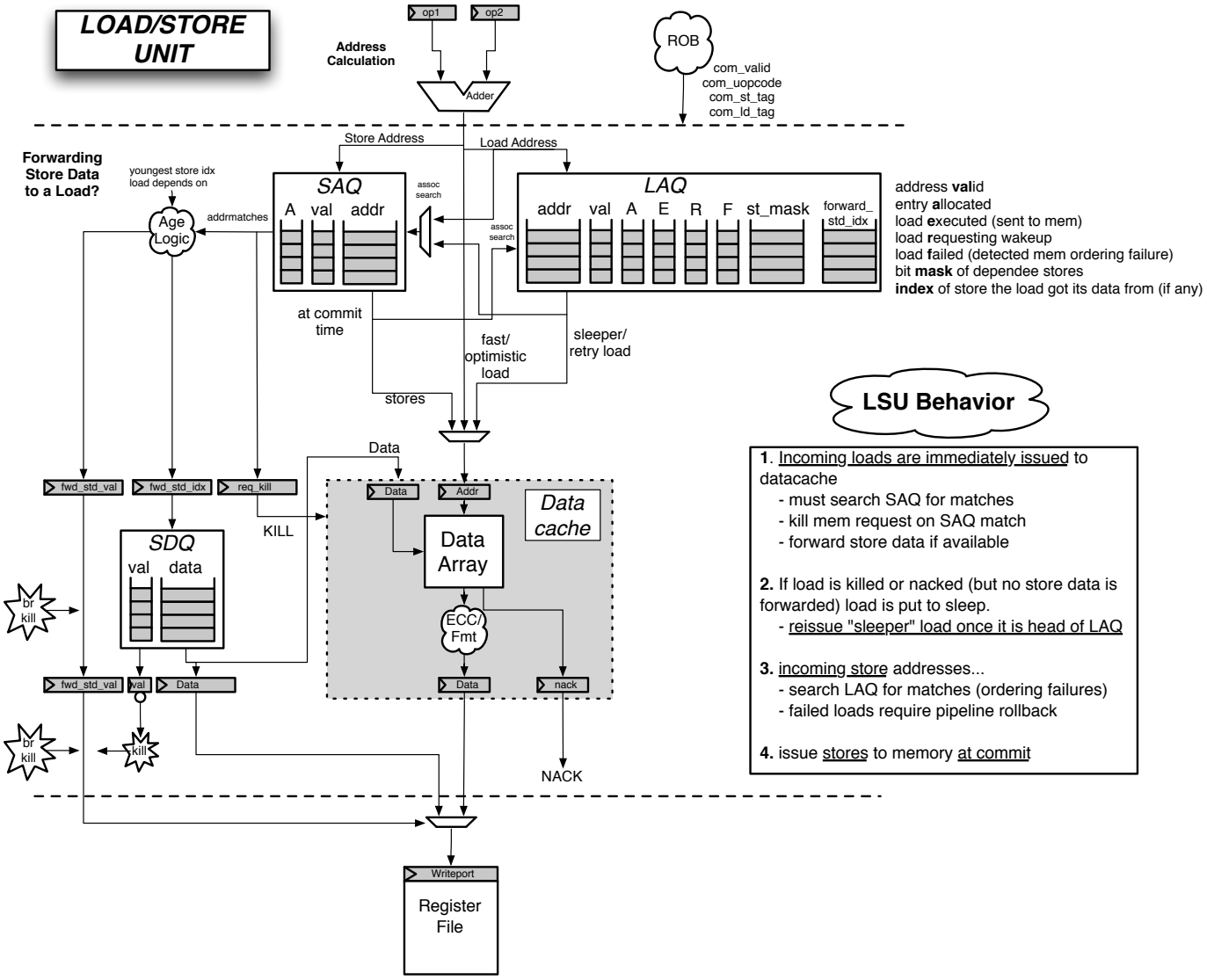

Figure 3: The Load/Store Unit.

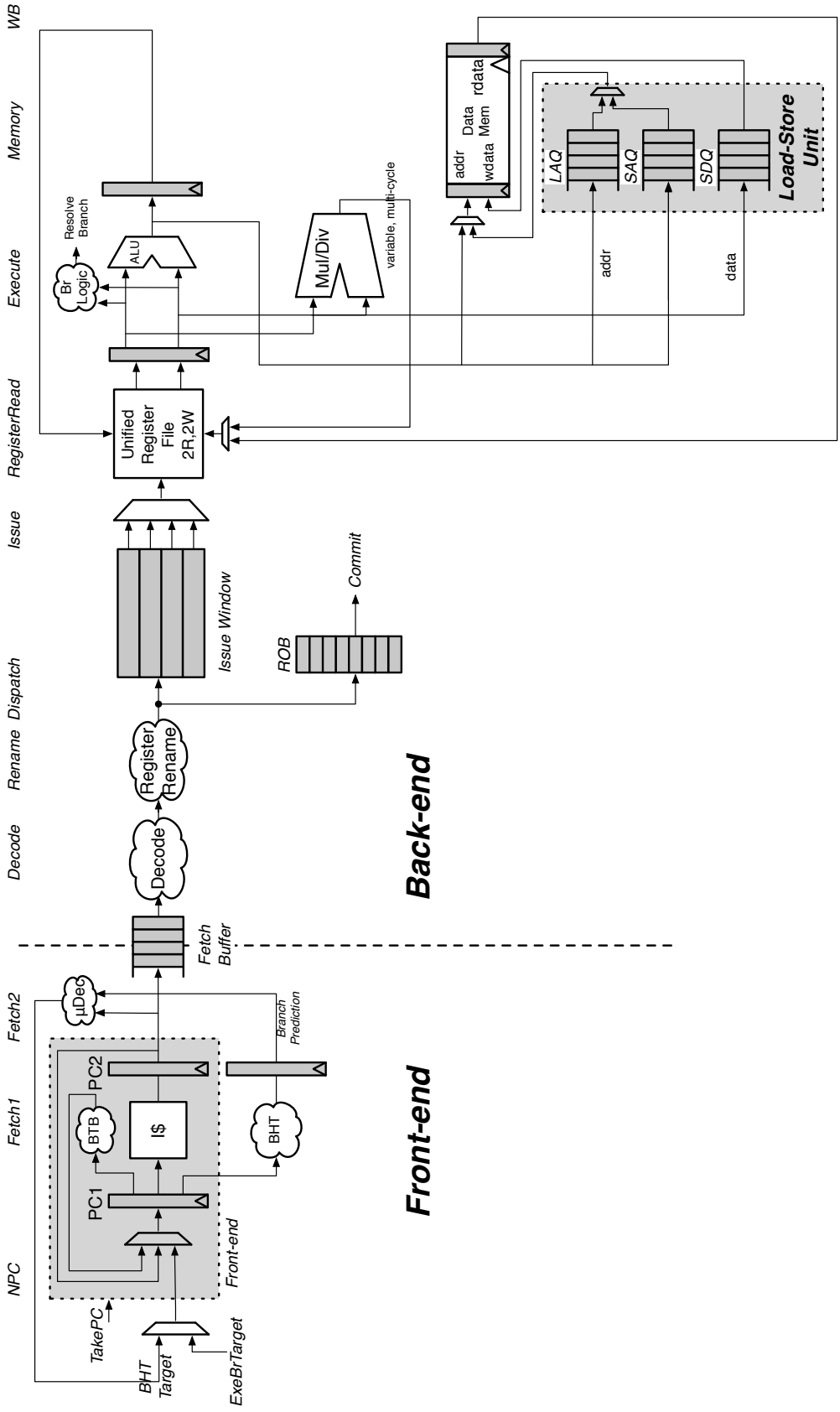

Figure 4: A more detailed diagram of BOOM.

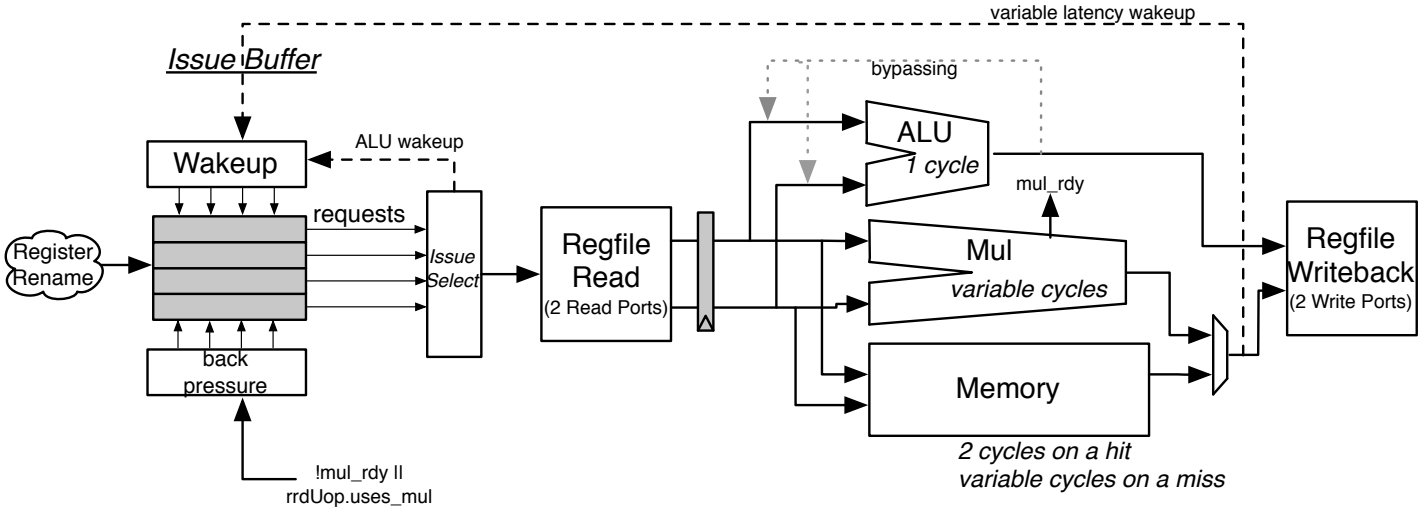

Figure 5: The issue logic and execution pipeline for a single-issue pipeline.

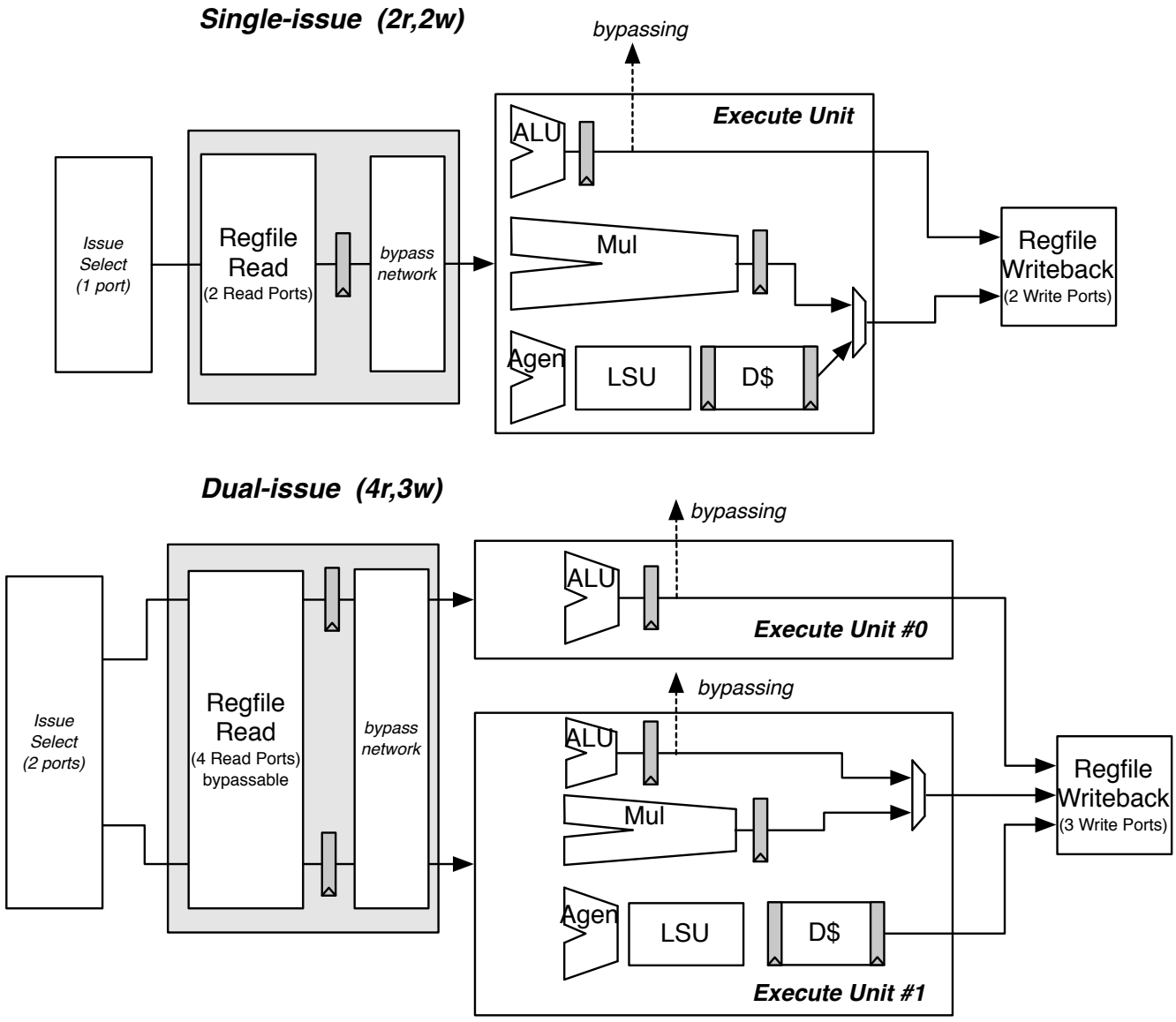

Figure 6: The issue logic and execution pipeline for single-issue and dual-issue pipelines. The issue logic can only issue one micro-op to each execution unit, but within each execution unit are multiple functional units. For example, the dual-issue pipeline can handle two ALU micro-ops per cycle, or one ALU micro-op and one memory micro-op per cycle, but not two memory micro-ops. Also notice that only the ALU functional units allow for the bypassing of back-to-back instructions.