CS 152 Laboratory Exercise 1

Professor: John Wawrzynek
TAs: Hasan Genc and Josh Kang
Department of Electrical Engineering & Computer Sciences
University of California, Berkeley

January 18, 2022

Revision History

Revision	Date	Author(s)	Description
1.0	2022-01-18	hngenc	Initial release

1 Introduction and Goals

The goal of this laboratory assignment is to familiarize yourself with the Chipyard simulation environment while also allowing you to conduct some simple experiments. By modifying an existing instruction tracer script, you will collect instruction mix statistics and make some architectural recommendations based on the results. You will be conducting cycle-accurate simulations of the "Sodor" instructional cores. These cores were designed to demonstrate basic principles of core design.

This lab consists of 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.

While students are encouraged to discuss solutions to the lab assignments with each other, you must complete the directed portion of the lab yourself and submit your own lab report for these problems. For the open-ended portion of each lab, students can either work individually or in groups of two or three. Each group will turn in a single report for the open-ended portion of the lab. You are free to participate in different groups for different lab assignments.

1.1 Graded Items

All reports are to be submitted through **Gradescope**. Please label each section of the results clearly. All directed items need to be turned in for evaluation. Your group only needs to submit *one* of the problems in the Open-Ended Portion.

- (Directed) Problem 3.4: recorded instruction mixes for each benchmark and answers
- (Directed) Problem 3.5: 1-stage CPI analysis answers
- (Directed) Problem 3.6: 5-stage CPI analysis answers

- (Directed) Problem 3.7: design problem answers
- (Open-ended) Problem 4.1: recorded ratio, answers, and source code
- (Open-ended) Problem 4.2: data and the modified section of Chisel source code
- (Open-ended) Problem 4.3: instruction definition, test code, worksheet, modified section of Chisel source code
- (Open-ended) Problem 4.4: design proposal and supporting data
- (Directed) Problem 5: feedback on this lab

! → Lab reports must be written in *readable* English; avoid raw dumps of logfiles. **Your lab report must be typed, and the open-ended portion must not exceed six (6) pages.** Charts, tables, and figures – where appropriate – are excellent ways to succinctly summarize your data.

2 Background

2.1 The RISC-V Instruction Set Architecture

The processor cores featured in this lab implement the RISC-V ISA, developed at UC Berkeley for use in education, research, and industry [1].

! → The RISC-V ISA manual is available under the "Resources" section of the CS 152 webpage or directly at `https://riscv.org/specifications/`. For Lab 1, all processors conform to the 32-bit base ISA, known as RV32I.

A complete software toolchain is pre-installed on the lab machines. Note that the GNU utilities are prefixed with the target triplet[1] (`riscv32-unknown-elf`) but otherwise function similarly as their native binutils and gcc counterparts that may be familiar to you. The components most relevant to this lab are:

- `riscv32-unknown-elf-gcc`: GNU cross-compiler for C
- `riscv32-unknown-elf-objdump`: GNU disassembler for RISC-V machine code
- `spike`: Functional ISA simulator which serves as the de-facto golden reference for the RISC-V ISA. Since it is not a cycle-accurate model, it cannot be relied on for performance measurements but can execute software much more quickly than an RTL simulator to verify correctness.

2.2 Chipyard

This lab, as well as subsequent CS 152 labs, is based on the **Chipyard** framework being actively developed UC Berkeley.

Chipyard is an integrated design, simulation, and implementation framework for agile development of systems-on-chip (SoCs). It combines Chisel, the Rocket Chip generator, and other Berkeley projects to produce a full-featured RISC-V SoC from a rich library of processor cores, accelerators, memory system components, and I/O peripherals. Chipyard supports several hardware development flows, including software RTL simulation, FPGA-accelerated simulation (FireSim), and automated VLSI methodologies (Hammer).

! → Chipyard documentation: `https://chipyard.readthedocs.io/en/latest/`

2.3 Chisel

Chisel is a *hardware design language* developed at UC Berkeley that facilitates advanced circuit generation and design reuse for digital logic designs.

[1] A canonical name for the system type that follows the nomenclature *cpu-vendor-os*

Chisel adds hardware construction primitives to the Scala programming language, providing designers with higher-level features such as object orientation, functional programming, parameterized types, and type inference to write complex, parameterizable hardware generators that produce synthesizable Verilog. This generator methodology enables the creation of re-usable components and libraries, raising the level of abstraction in design while retaining fine-grained control. A Chisel design is essentially a legal Scala program whose execution emits low-level RTL code, which can then be mapped to ASICs, FPGAs, or cycle-accurate software simulators such as VCS and Verilator.

! → Documentation about the Chisel language, along with an interactive bootcamp tutorial, can be found at `https://www.chisel-lang.org/`.

2.3.1 Chisel in This Lab

The "Sodor" instructional cores in this lab are implemented using the Chisel HDL according to the generator design methodology. In this lab, you will compile these Chisel-based processors into software simulators using Verilator and run cycle-accurate experiments on instruction mixes and pipeline hazards.

Students will not be required to write Chisel code as part of this lab, beyond adding and modifying parameters as directed.

3 Directed Portion (30% of lab grade)

3.1 Terminology and Conventions

Throughout this course, the term *host* refers to the machine on which the simulation runs, while *target* refers to the machine being simulated. For this lab, an instructional server will act as the host, and the RISC-V processors will be the target machines.

UNIX shell commands to be run on the host are prefixed with the prompt "`eecs$`".

3.2 Setup

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

Once logged in, source the following script to initialize your shell environment so as to be able to access to the tools for this lab. Run it before each session.[3]

```
eecs$ source ~cs152/sp22/cs152.lab1.bashrc
```

First, clone the lab materials into an appropriate workspace.[4]

[2] Create a CS152-specific instructional account through the WebAcct service: `http://inst.eecs.berkeley.edu/webacct/`

[3] Or add it to your `bash` profile.

[4] Since NFS homedirs can be slow, local disk space is available on the `eda` servers under the `/scratch` partition (`mkdir -p -m 700 /scratch/$USER`), but remember that it is *not* backed up automatically.

```
eecs$ mkdir -m 0700 -p /scratch/$USER
eecs$ cd /scratch/$USER
eecs$ git clone ~cs152/sp22/lab1.git
eecs$ cd lab1
eecs$ LAB1ROOT=$PWD
eecs$ BMARKS=$LAB1ROOT/generators/riscv-sodor/riscv-bmarks
eecs$ SCRIPTS=$LAB1ROOT/generators/riscv-sodor/scripts
eecs$ ./scripts/init-submodules-no-riscv-tools.sh
```

The `init-submodules-no-riscv-tools.sh` script clones all the git submodules of the various Chipyard components. This step is expected to take several minutes.

! → **It is highly recommended** to work in the local `/scratch` partition to avoid issues with filesystem performance and quotas. Even simulations of modest length (few hundred thousand cycles) can produce a few gigabytes of logs and waveform dumps. Do not use your NFS home directory to avoid slowing down the simulation. Remember that `/scratch` is *not* backed up automatically.

The remainder of this exercise will use `${LAB1ROOT}` to denote the path of the `lab1` working tree. Its directory structure is outlined below:

```
${LAB1ROOT}
    generators/                 Chisel source code for cores/caches/peripherals/etc.
        riscv-sodor/                                  Sodor sources and utilities
            src/main/scala/
              common/           Common source code shared between all Sodor cores
              rv32_1stage/                  Source code for the 1-stage core
              rv32_2stage/                  Source code for the 2-stage core
              rv32_3stage/                  Source code for the 3-stage core
              rv32_5stage/                  Source code for the 5-stage core
              rv32_ucode/                   Source code for the microcoded core
        riscv-bmarks/                       Pre-compiled benchmark binaries
        scripts/                            Python scripts for analyzing Sodor traces
        test/
            custom-tests/                   Stub for open-ended question 4.3
            custom-bmarks/                  Stub for open-ended question 4.1
    scripts/                                Contains repo initialization script
    sims/
        verilator/                          Verilator simulation directory
            generated-src/                  Generated Verilog after Chisel elaboration
            output/                         Simulation traces are logged here
```

Of particular note is that the Chisel source code for the processors can be found in `${LAB1ROOT}/generators/riscv-sodor/src/main/scala`. While you do not need understand the code to do this assignment, it may be interesting to examine the internals of a processor. Although it is not recommended that you alter any of the processors while collecting data from them in the

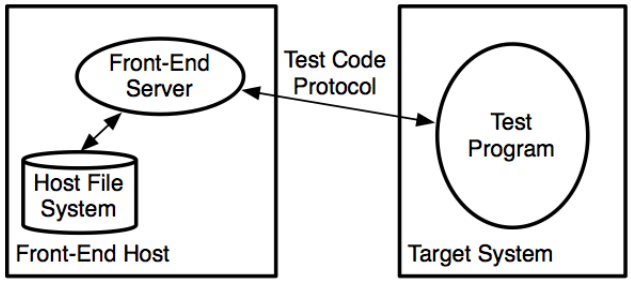

Figure 1: The simulation environment. The front-end server (`fesvr`) reads a RISC-V ELF binary from the *host* filesystem, starts the *target* system simulator, and populates the *target* system memory with the given ELF program segments. Once `fesvr` finishes loading the binary, it releases the *target* system from reset, and the simulated processor then begins execution at the reset vector PC. Here, the test protocol is the standard RISC-V debug module interface [2].

directed lab portion (except as instructed), feel free in your own time (or perhaps as part of the open-ended portion) to modify the processors as you see fit.

3.3 First Steps: Building and Simulating the 1-Stage Processor

The lab repository contains five different cores: 1/2/3/5-stage pipelines and a microcoded processor.

3.3.1 Building the 1-stage Processor

Run the following commands to build the 1-stage processor:

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ make CONFIG=Sodor1StageConfig
```

The first run of `sbt` may take some time since it must fetch various Scala dependencies. We recommend you run this step in `tmux` or `screen`, and find something else to do as the simulator builds. [5]

! → **It is expected** that the first invocation of this `make` command will take > 10 minutes to complete, as the framework must compile the Chisel and FIRRTL compilers, all Scala dependencies, and Verilator.

The `make` command orchestrates the following steps:

1. Start `sbt` (the Scala Build Tool), select the `Sodor1StageConfig` config, and compile and run the Chisel code which generates a Verilog RTL description of the processor. The generated Verilog code can be found in `${LAB1ROOT}/sims/verilator/generated-src`.
2. Run `verilator`, an open-source tool that converts Verilog into a C++ cycle-accurate simulation model.
3. Compile the Verilator-generated C++ code into an x86 executable.

[5] Should you encounter a `java.lang.OutOfMemoryError` exception, repeat the `make` command.

3.3.2 Simulating the 1-stage Processor

Run the following commands to run a simulation of the Sodor 1-stage processor running the Towers of Hanoi benchmark.

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ make CONFIG=Sodor1StageConfig run-binary BINARY=${BMARKS}/towers.riscv
```

The simulation should print the cycle count (`mcycle`) and instruction count (`minstret`) upon completion. You may want to try running the other benchmarks in `riscv-bmarks` as well. If any benchmarks fail to complete and print `mcycle` and `minstret`, verify that you are running on a recommended instructional machine. Otherwise, contact your TA.

3.3.3 Building Other Processors

To select a different processor design point, simply change the `CONFIG=` key of the `make` command. Valid options are listed in Table 2.

Sodor1StageConfig
Sodor2StageConfig
Sodor3StageConfig
Sodor5StageConfig
SodorUCodeConfig

Table 2: The configs available in this lab.

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ make CONFIG=Sodor3StageConfig run-binary BINARY=${BMARKS}/towers.riscv
```

3.3.4 Dumping Waveforms for Debugging

! → (This information is provided for completeness but is not necessary to complete the lab.)

In the very unlikely scenario that you need to debug what you suspect to be an RTL bug, VCD-formatted waveforms can be obtained by running `make run-binary-debug` instead of the usual `make run-binary` command. Open the resulting `output/*.vcd` files in a waveform viewer such as GTKWave (`http://gtkwave.sourceforge.net/`).

3.4 Tracing Instruction Mixes Using the 1-Stage Processor

For this section of the lab, you will look at the instruction mixes of several RISC-V benchmark programs provided to you.

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ make CONFIG=Sodor1StageConfig run-binary BINARY=${BMARKS}/vvadd.riscv
eecs$ less output/chipyard.TestHarness.Sodor1StageConfig/vvadd.out
```

We have provided a set of benchmarks for you to gather results from: `dhrystone`, `median`, `multiply`, `qsort`, `rsort`, `towers`, and `vvadd`. Using your editor of choice, inspect the output files generated by `make run-binary` after running each of these benchmarks.

The processor commit state is logged to the output trace file on every cycle. We have provided a

Python script which analyzes the contents of the omitted trace file and generates basic statistics. Run the following command to view the statistics.

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ ${SCRIPTS}/tracer.py output/chipyard.TestHarness.Sodor1StageConfig/vvadd.out
```

```
Stats:

CPI          : 1.000
IPC          : 1.000
Cycles       : 12413
Instructions : 12414
Bubbles      : 0

Instruction Breakdown:
% Arithmetic   : 46.858 %
% Ld/St        : 29.861 %
% Branch/Jump  : 22.152 %
% Misc.        : 1.128 %
```

Note how the mix of different types of instructions vary between benchmarks. Record the mix for each benchmark. (Remember: Do not provide raw dumps. A good way to visualize this kind of data would be a bar graph.) Which benchmark has the highest arithmetic intensity? Which benchmark seems most likely to be memory bound? Which benchmark seems most likely to be dependent on branch predictor performance?[6]

3.5 CPI Analysis Using the 1-Stage Processor

Consider the results gathered from the RV32 1-stage processor. Suppose you were to design a new machine such that the average CPI of loads and stores is 2 cycles, integer arithmetic instructions take 1 cycle, and other instructions take 1.5 cycles on average. What is the overall CPI of the machine for each benchmark?

What is the relative performance for each benchmark if loads/stores are sped up to have an average CPI of 1? Is this still a worthwhile modification if it means that the cycle time increases 30%? Is it worthwhile for all benchmarks or only a subset? Explain.

3.6 CPI Analysis Using the 5-Stage Processor

For this section, we will analyze the effects of branching and bypassing in a 5-stage processor.[7]

The 5-stage processor has been parameterized to support both full-bypassed (but must still stall for load-use hazards) and fully-interlocked configurations. The fully-interlocked variant performs *no* bypassing and instead must stall (interlock) the instruction fetch and decode stages until all hazards have been resolved.

First, we verify that full bypassing is enabled in the design. Navigate to the Chisel source code:

```
eecs$ cd ${LAB1ROOT}/generators/riscv-sodor/src/main/scala/rv32_5stage
eecs$ vim consts.scala  # Use any editor of your choice
```

[6] The disassembly for all benchmarks is available at `${LAB1ROOT}/${BMARKS}/*.dump`.

[7] The 2-stage and 3-stage processors will not be explicitly used in this lab, but they exist to demonstrate how pipelining in a relatively simple microarchitecture is implemented.

The `consts.scala` file defines constants and compile-time parameters for the processor. Observe the parameter on line 21 is `val USE_FULL_BYPASSING = true`. You can see how this parameter changes the pipeline by referring to the data path in `dpath.scala` (lines 269-301) and the control path in `cpath.scala` (lines 226-245). The data path instantiates the bypass muxes when full bypassing is activated. The control path contains the stall logic, which must account for more situations when no bypassing is selected.

Like we did for the 1-stage processor, build and run the processor on all provided benchmarks, with the default behavior of bypassing enabled.

```
eecs$ make CONFIG=Sodor5StageConfig run-binary BINARY=${BMARKS}/vvadd.riscv
```

Record the CPI values for all benchmarks. Are they what you expected?

Now disable full bypassing in `consts.scala`, and re-run the build (check that your Chisel code recompiles).

Record the new CPI values for all benchmarks. How does full bypassing perform compared to full interlocking? If adding full bypassing would hurt the cycle time of the processor by 25%, would it be worth it? Argue your case quantitatively.

3.7 Design Problem Using the 5-Stage Processor

Imagine that you are being asked by your employer to evaluate a potential modification to the design of a 5-stage RISC-V pipeline. The proposed modification is that the Execute / Address Calculation stage and the Memory Access stage be merged into a single pipeline stage. In this combined stage, the ALU and Memory will operate in parallel. Data access instructions will use memory while leaving the ALU idle, and arithmetic instructions will use the ALU while leaving memory idle. These changes are beneficial in terms of area and power efficiency. Think to yourself why this is the case, and if you are still unsure, ask about it in discussion section or office hours.

In RISC-V, the effective address of a load or store is calculated by summing the contents of one register ($rs1$) with an immediate value (imm).

The problem with the new design is that there is is now no way to perform any address calculation in the middle of a load or store instruction, since loads and stores do not get to access the ALU. Proponents of the new design advocate changing the ISA to allow only one addressing mode: register direct addressing. Only one source register is used, and the value it contains is the memory address to be accessed. No offset can be specified.

In RISC-V, the only way to perform register direct addressing register-immediate address calculation with $imm = 0$.

With the proposed design, any load or store instruction which uses register-immediate addressing with $imm \neq 0$ will take two instructions. First, the register and immediate values must be summed with an add instruction, and then this calculated address can be loaded from or stored to in the next instruction. Load and store instructions which currently use an offset of zero will not require extra instructions on the new design.

Your job is to determine the percentage increase in the total number of instructions that would have to be executed under the new design. This will require a more detailed analysis of the different types of loads and stores executed by our benchmark codes.

In order to track more specific statistics about the instructions being executed, you will need to modify the Python script at `${LAB1ROOT}/generators/ricsv-sodor/scripts/tracer.py`.

Modify the tracer to detect the percentage of instructions that are loads and stores with non-zero

offsets. Follow the existing framework in `tracer.py` to accomplish this task. There is existing code which you can adapt for your modifications.

Consult the RISC-V unprivileged ISA specification (Volume I, found under "Resources" on the CS 152 webpage) to determine which instruction bits correspond towhich fields.

After modifying `tracer.py`, re-run the tracer on the output files to gather results.

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ ${SCRIPTS}/tracer.py output/chipyard.TestHarness.Sodor1StageConfig/vvadd.out
```

What percentages of the instruction mix do the various types of load and store instructions make up? Evaluate the new design in terms of the percentage increase in the number of instructions that will have to be executed. Which design would you advise your employer to adopt? Justify your position quantitatively.

4 Open-ended Portion (70% of lab grade)

Select *one* of the following questions per team. The open-ended portion is worth a large fraction of the grade of the lab, and the grade depends on how complex and interesting a project you complete, so spend the appropriate amount of time and energy on it. Also, have fun with it!

4.1 Mix Manufacturing

The goal of this problem is to investigate how effectively (or ineffectively) the compiler might handle complicated C code of your creation.

Using no more than 15 lines of C code, attempt to produce RISC-V machine code with the maximum ratio of branch to non-branch instructions when run on the 5-stage processor (fully bypassed).[8] In other words, try to produce as many branch instructions as possible. You can use code that emits jumps, but unconditional jump instructions do not count as branches. Your C code can contain as many poor coding practices as you like but must adhere to the following criteria:

- Limit to one statement per line.[9] Selection (`if`, `else`, `switch`) and iteration (`for`, `while`, `do`) statements each count as one statement in addition to the body.
- Do not call functions or execute code not contained within the 15-line block.
- Do not use inline assembly or comma operators.
- Limit to one ternary operator (`?:`) per expression.
- The code must always terminate.

Write your code in `${LAB1ROOT}/generators/riscv-sodor/test/custom-bmarks/mix.c`. To test for correctness, compile and run it on the functional ISA simulator:

```
eecs$ cd ${LAB1ROOT}/generators/riscv-sodor/test/custom-bmarks
eecs$ make
eecs$ make run
```

To produce a disassembly of the code as `mix.dump`:

```
eecs$ make dump
```

However, to obtain a cycle-accurate trace to determine the actual effect of your program on CPI, you must run the code on the RV32 5-stage processor (fully bypassed):

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ CUSTOM_BMARKS=${LAB1ROOT}/generators/riscv-sodor/test/custom-bmarks
eecs$ make CONFIG=Sodor5StageConfig run-binary BINARY=${CUSTOM_BMARKS}/mix.riscv
```

Analyze `output/chipyard.TestHarness.Sodor5StageConfig/mix.out` with the `tracer.py` script and report the ratio of branch to non-branch instructions achieved with your code. What is the resulting CPI? As more branches were added, did the CPI increase or decrease? Explain why the CPI changed in the direction that it did. In your report, summarize some of the ideas that you tried. Submit this write-up, your lines of C code, and the excerpt of the disassembly that corresponds to your C code.

[8] Most compiler optimizations are disabled (`-O0`) to make this exercise easier.
[9] As defined in ISO/IEC 9899 6.8: `http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf`

4.2 Bypass Path Analysis

As an engineer working for a new start-up processor design company, you find yourself 3% over budget area-wise on your company's latest 5-stage processor (your company makes very small processors, and every bit of area counts!). However, if you remove one bypass path you can meet the budget and ship on time!

With the Chisel source code in `${LAB1ROOT}/generators/riscv-sodor/src/main/scala/rv32_5stage`, analyze the impact on CPI when different bypass paths are removed from the design. The files `dpath.scala` and `cpath.scala` contain the relevant code for modifying the bypass and stall logic. Ensure that your modified pipeline passes all of the assembly tests!

Use your data to support your conclusion about which bypass path could be eliminated with the least impact on CPI. Include snippets of your modified Chisel code in an appendix in your report.

Feel free to email your TA or attend office hours if you need help understanding Chisel, the processor, or anything else regarding this problem.

4.3 Define and Implement Your Favorite Complex Instruction

In Problem Set 1, we have asked you to implement two complex instructions (`ADDm` and `STRLEN`) in the microcoded processor. Imagine that you are adding a new instruction to the RISC-V ISA. Propose a new complex instruction (other than `MOVN`/`MOVZ`) that involves an EZ/NZ µBr and at least one memory operand.

First devise an encoding for your new instruction. Consult the RISC-V unprivileged ISA specification to select an appropriate instruction format (see §2.2 "Base Instruction Formats"), and then find an unused opcode space (see the base opcode map in Table 24.1). Note that the *custom-0/1/2/3* and *reserved* spaces are currently available.

Define your instruction in `${LAB1ROOT}/generators/riscv-sodor/src/main/scala/common/instructions.scala` (search for a TODO comment in the file). We refer to the definition for `MOVN` as an example:

```
def MOVN = BitPat("b?????????????????????????1110111")
```

The bit pattern specifies which bits should match a fixed value for decoding (e.g. an opcode). Note that the `?` character denotes a "don't-care" bit location that may take any value (e.g., register specifiers). Underscore `_` characters are ignored. The variable identifier is used as a label for the microcode dispatcher.

Once you have assigned an instruction encoding, you will have to write an assembly test to test your instruction. As an example, an assembly test for the `MOVN` instruction is provided in `${LAB1ROOT}/generators/riscv-sodor/test/custom-tests/movn.S`. Since the assembler is not directly aware of our custom instructions, we must numerically encode the instruction with a `.word` directive.[10] We also write some assembly code to load values into registers and memory. Finally, the code checks the correctness of the result.

We have provided you with an empty assembly template to complete at `${LAB1ROOT}/generators/riscv-sodor/test/custom-tests/yourinst.S` (search for a TODO comment in the file). Compile your assembly test:

```
eecs$ cd ${LAB1ROOT}/generators/riscv-sodor/test/custom-tests
eecs$ make
```

[10] Recent versions of the GNU assembler support the more user-friendly `.insn` directive: `https://sourceware.org/binutils/docs/as/RISC_002dV_002dFormats.html`

Next, work out the microcode implementation on a worksheet that you have used in Problem Set 1 (worksheet 2.A or 2.B). Once you have figured out all the states and control signals, add your microcode to `${LAB1ROOT}/generators/riscv-sodor/src/main/scala/rv32_ucode/microcode.scala` (search for a TODO comment in the file). Again, as an example, the `MOVN` instruction has already been implemented in `microcode.scala`. Once you are done, build the processor and run the assembly test:

```
eecs$ cd ${LAB1ROOT}/sims/verilator
eecs$ CUSTOM_TESTS=${LAB1ROOT}/generators/riscv-sodor/test/custom-tests
eecs$ make CONFIG=SodorUCodeConfig run-binary BINARY=${CUSTOM_TESTS}/rv32ui-p-yourinst
```

Look at the cycle-by-cycle trace written to `${LAB1ROOT}/output/chipyard.TestHarness.SodorUCodeConfig/rv32ui-p-yourinst.out` to examine the microarchitectural state. Verify that the processor has executed your microcoded instruction correctly. Revise your implementation if necessary.

Feel free to email your TA or attend office hours if you need help understanding Chisel, the processor, or anything else regarding this problem.

4.4 Processor Design

Propose a microarchitectural modification of your own to a 3-stage or 5-stage pipeline. Justify the motivation, cost, and overhead of your design modification by explaining which instructions are affected by the changes you propose and in what way.

You may have to draw a block diagram to clarify your proposed changes, and you will very likely have to modify the `tracer.py` script to track specific types of instructions not previously traced. A further tactic might be to show that while some instructions are impacted negatively, these instructions are not a significant portion of certain benchmarks. Feel free to be creative. Try to quantitatively justify your case, but you do *not* need to implement your proposed processor design.

4.5 Your Own Idea

We are also open to your own ideas. Particularly enterprising individuals can even modify the provided Chisel processors as part of a study of one's own design. However, you must first consult with the professor and/or TAs to ensure that your idea is of sufficient merit and of manageable complexity.

5 Feedback Portion

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

- How many hours did the directed portion take you?
- How many hours did you spend on the open-ended portion?
- Was this lab boring?
- What did you learn?
- Is there anything that you would change?

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

5.1 Team Feedback

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

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

6 Acknowledgments

Many people have contributed to versions of this lab over the years. This lab is based off of the work by Yunsup Lee and was originally developed for CS 152 at UC Berkeley by Christopher Celio, and heavily inspired by the previous set of CS 152 labs (which targeted the Simics emulators) written by Henry Cook. This lab was made possible through the work of Jonathan Bachrach, who lead the development of Chisel, and through the work of Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović who developed the RISC-V ISA.

References

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

[2] T. Newsome and M. Wachs, Eds., *RISC-V external debug support*, Version 0.13.2, RISC-V Foundation, Mar. 2019. [Online]. Available: `https://riscv.org/specifications/debug-specification/`.

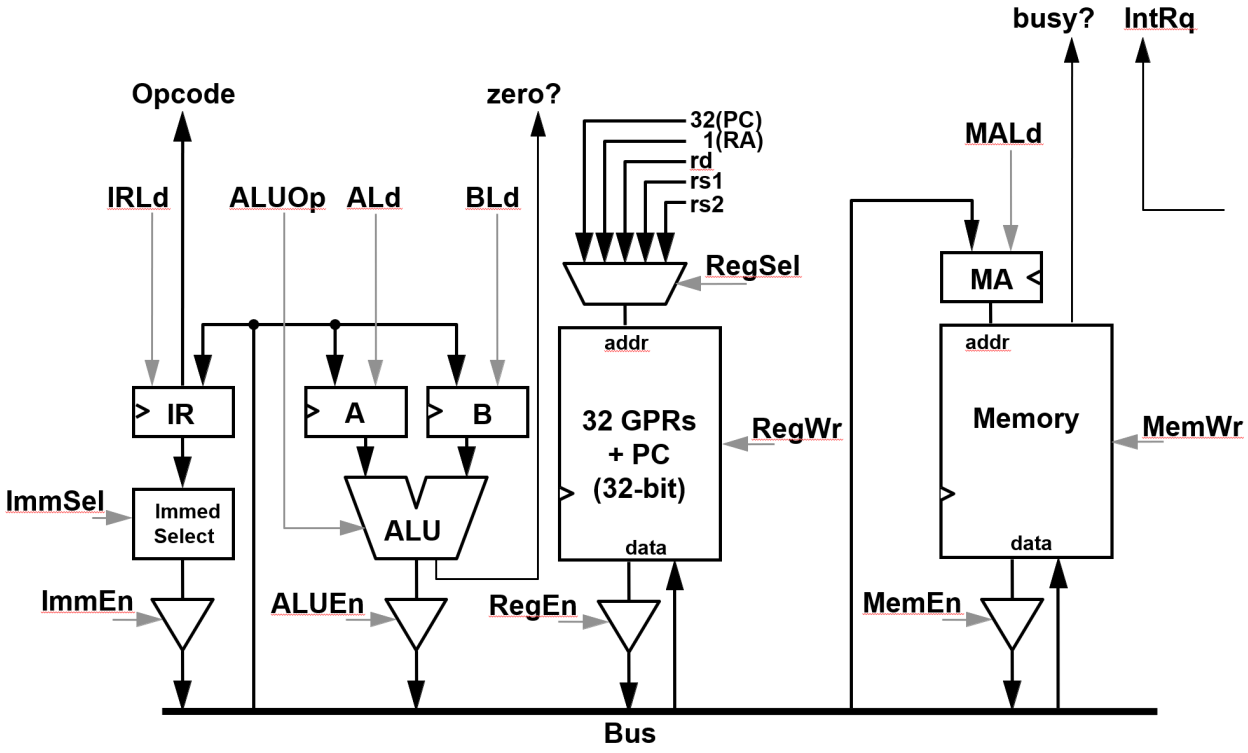

Figure 2: RV32 bus-based microcoded core

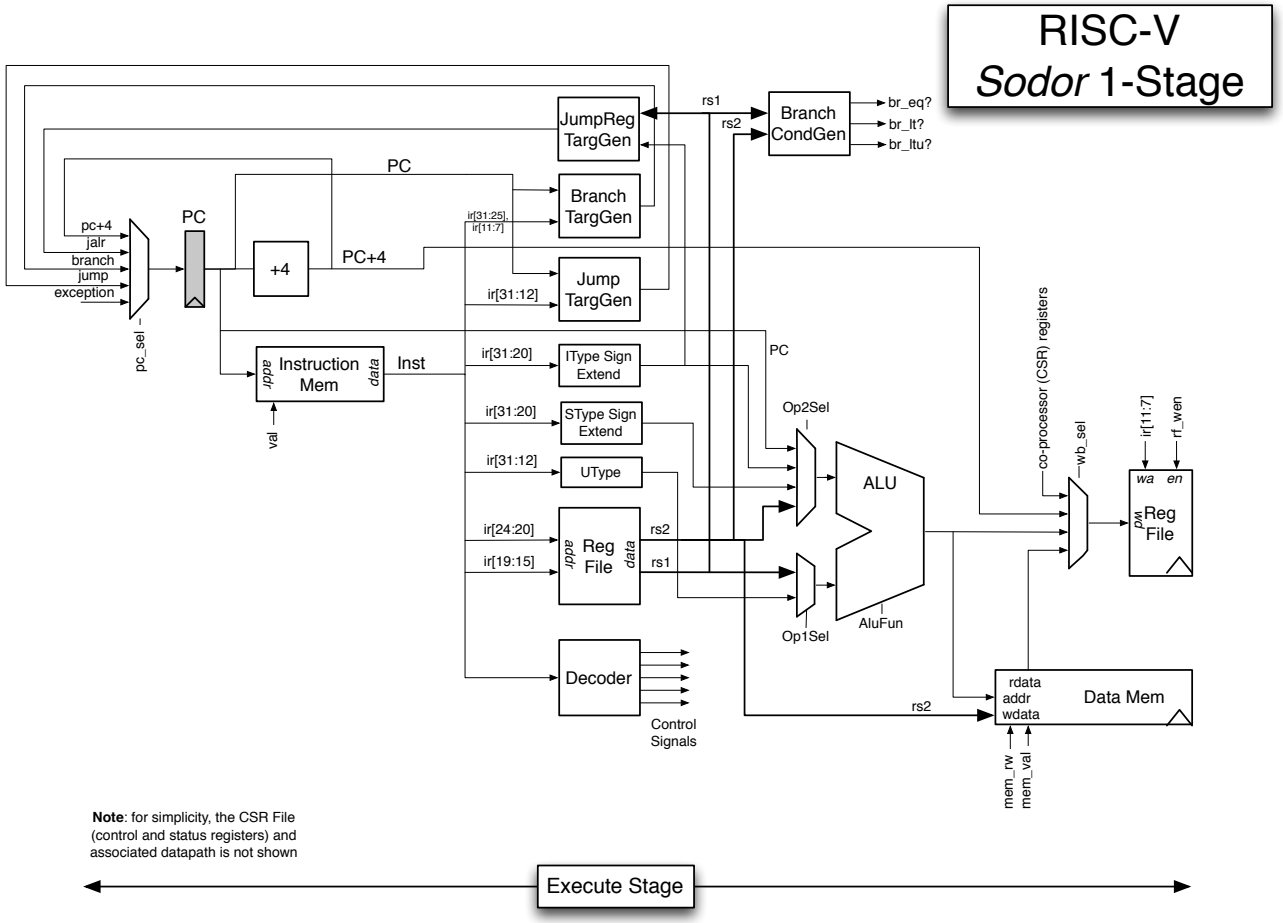

Figure 3: RV32 1-stage pipeline

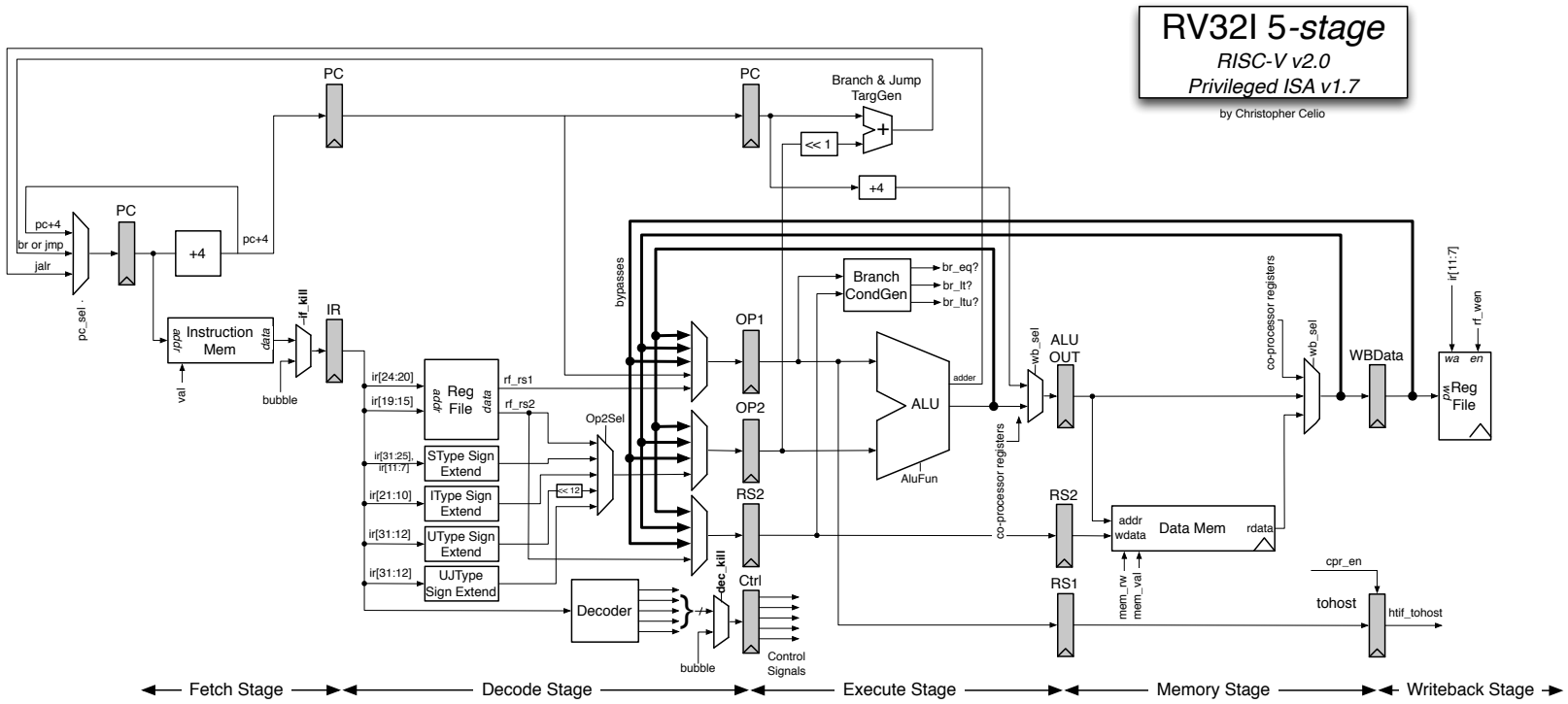

Figure 4: RV32 5-stage pipeline