C152 Laboratory Exercise 1

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

January 24, 2012

1 Introduction and goals

The goal of this laboratory assignment is to familiarize you with the `Chisel` simulation environment while also allowing you to conduct some simple experiments. By modifying an existing instruction tracer module, you will collect instruction mix statistics and make some architectural recommendations based on the results.

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.

1.1 Graded Items

You will turn in a hard copy of your results to the professor or TA. Please label each section of the results clearly. The following items need to be turned in for evaluation:

1. Problem 4.4: recorded instruction mixes for each benchmark and answers
2. Problem 4.5: thought problem answers
3. Problem 4.6: CPI analysis answers
4. Problem 4.7: design problem answers
5. Problem 5.1: source code and recorded ratio
6. Problem 5.2 data and the modified section of `Chisel` source code
7. Problem 5.3 and Problem 5.4 design proposals and supporting data

8. Problem 6: Feedback on this lab

You only need to turn in *one* of the problems found in the Open-Ended Portion (Section 5).

Lab reports must be in *readable* English and not raw dumps of log-files. It is *highly* recommended that your lab report be typed. Charts, tables, and figures - when appropriate - are great ways to succinctly summarize your data.

2 The RISC-V Instruction Set Architecture

The processors in this lab that you will be studying implement the RISC-V ISA, recently developed at UC Berkeley for use in education and research.

An entire tool-chain is provided. The `riscv-gcc` cross-compiler builds new binaries from RISC-V assembly and `C` source code. The `riscv-objdump` tool can disassembly existing RISC-V binaries to show the exact sequence of instructions being executed.

The ISA simulator `riscv-isa-sim` can execute RISC-V binaries (as an ISA simulator, it is not cycle-accurate, but it executes very quickly). The front-end server, `fesvr`, loads a RISC-V binary and connects to either the ISA simulator or a `Chisel`-created simulator.

The RISC-V ISA manual can be found in the "extra course materials" section of the CS 152 website (`http://www-inst.eecs.berkeley.edu/~cs152/sp12/handouts/riscv-spec.pdf`). For Lab 1, all processors implement the 32-bit variant, known as RV32.

3 Chisel

`Chisel` is a new *hardware construction language* developed at UC Berkeley for the rapid design and development of hardware. `Chisel` raises the level of abstraction by allowing designers to utilize concepts such as object orientation, functional programming, parameterized types, and type inference. Unlike HDL languages such as Verilog which were designed first to be *simulation* languages, `Chisel` is designed *specifically* for constructing actual hardware.

`Chisel` can generate both low-level Verilog code, for mapping designs to FPGA and ASICs, and high-speed C++-based cycle-accurate software simulators.

`Chisel`, an acronym for Constructing Hardware In a Scala Embedded Language, is a domain-specific language embedded inside of Scala. `Chisel` code describing a processor is actually a legal Scala program whose execution outputs either Verilog code or C++ code. Figure 1 shows the design flow of `Chisel`.

3.1 Chisel in This Lab

Provided with this lab are four different processors: a 1-stage pipeline, a 2-stage pipeline, a 5-stage pipeline, and a micro-coded pipeline. All are implemented in `Chisel`.

In this lab, you will compile the provided `Chisel` processors into C++ software simulators, and use the C++ simulators to quickly run cycle-accurate experiments regarding instruction mixes and pipeline hazards.

A tutorial on the `Chisel` language can be found at (`http://www-inst.eecs.berkeley.edu/~cs152/sp12/handouts/chisel-tutorial.pdf`). Students will not be required to write `Chisel` code as part of this lab, beyond changing parameters as directed.

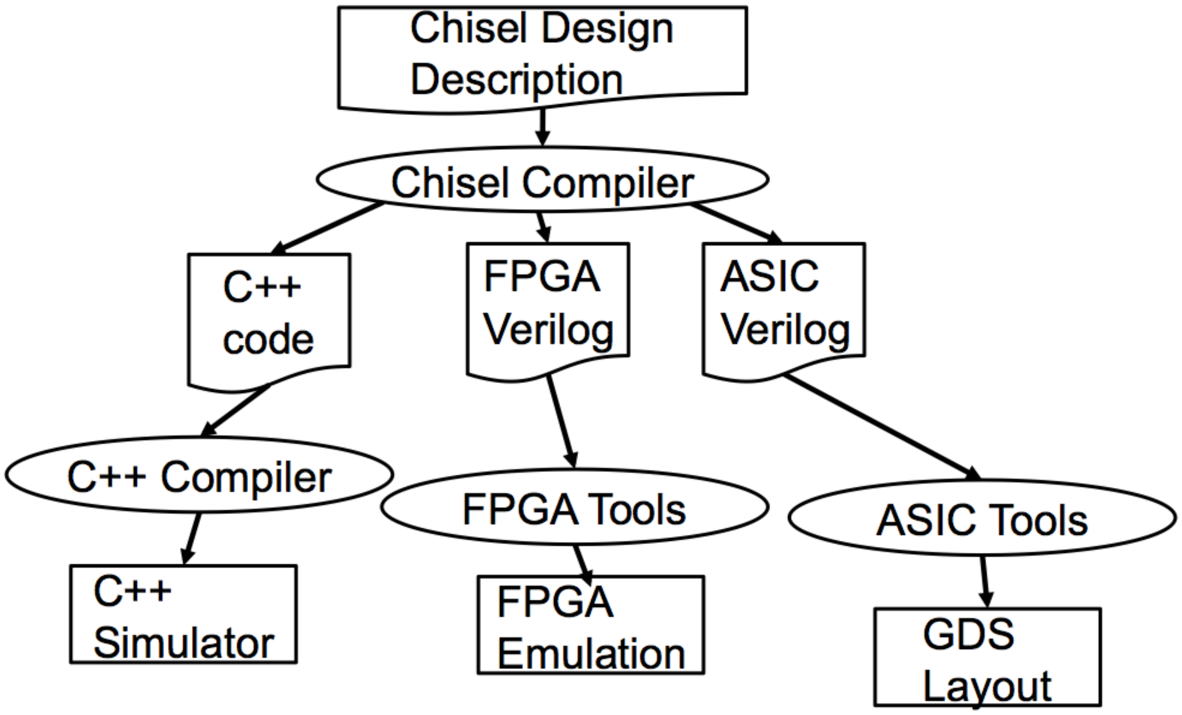

Figure 1: Chisel Design Flow. Only the C++ Simulator path will be exercised for this lab.

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.

4 Directed Portion

4.1 Terminology and conventions

Host machine refers to the machine you are using to run simulators on. For this lab, the *Host machine* will be an Instructional machine (`inst$`). *Target machine* refers to the simulated machine, in this case the provided RISC-V processors will be your target machines.

4.2 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. We will provide you with an instructional computing account for this purpose.

The tools for this lab were set up to run on any of the twelve instructional Linux servers `t7400-1.eecs`, `t7400-2.eecs`, `...`, `t7400-12.eecs`. (see `http://inst.eecs.berkeley.edu/cgi-bin/clients.cgi?choice=servers` for more information about available machines).

First, download the lab materials[1]:

```
inst$ cp -R ~cs152/Lab1 ./Lab1

inst$ cd ./Lab1
inst$ export LAB1ROOT=$PWD
```

We will refer to ./Lab1 as `${LAB1ROOT}` in the rest of the handout to denote the location of the Lab 1 directory.

The directory structure is shown below:

- `${LAB1ROOT}/`
  - doc/ Useful documentation and related materials.
  - test/ Local source code for benchmarks and tests.[2]
    - riscv-bmarks/ Local benchmarks written in C.
    - riscv-tests/ Local tests written in assembly.
  - chisel
    - Makefile
    - src/ `Chisel` source code for each processor.
      - rv32_1stage/ Symbolic link to src directory found deep inside sbt/ directory.
      - rv32_2stage/ ...
      - rv32_5stage/ ...
      - rv32_ucode/ ...
    - emulator/
      - common/Common emulation infrastructure shared between all processors.
      - rv32_1stage/ C++ simulation tools and output files.
      - rv32_2stage/ ...
      - rv32_5stage/ ...
      - rv32_ucode/ ...
    - sbt/ `Chisel`/Scala voodoo. You can safely ignore this directory.

Of particular note is that the source code for the `Chisel` processors can be found in `${LAB1ROOT}/chisel/src/`. While you do not have to understand the code to do this assignment, it may be interesting to see the entire workings of a processor. While it is not recommended that you modify any of the processors while collecting data for them in the directed lab portion (except as directed), feel free in your own time (or perhaps as part of the open-ended portion) to change and tweak the processors as you see fit.

[1]The capital "R" in "cp -R" is critical, as the -R option maintains the symbolic links found in the ${LAB1ROOT}/chisel/src/ directory.

[2]Most benchmarks and assembly tests that you will be running are globally installed in ~cs152/install/, however, you can add your own benchmarks and assembly tests in riscv-bmarks/ and riscv-tests/ directories and leverage the existing Makefiles to build and test them.

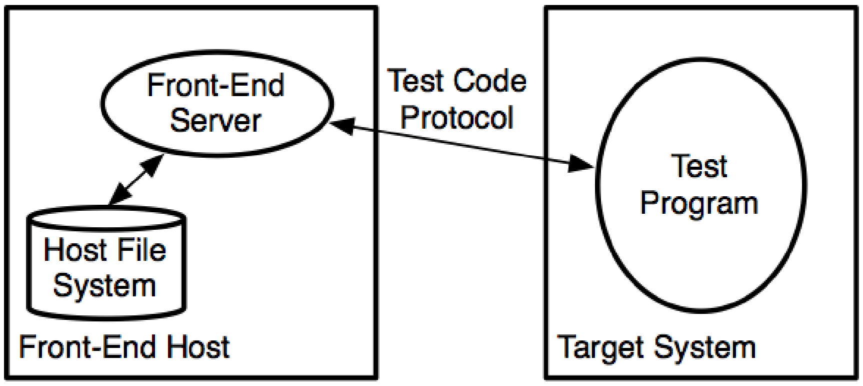

Figure 2: The Testing Environment. The front-end server (`fesvr`) loads the RISC-V binary from the `Host` file system, starts the `Target` system simulator, and sends the RISC-V binary code to the `Target` simulator to populate the simulated processor's instruction memory with the program. Once the `fesvr` finishes sending the test code, the `fesvr` sends the "Start" command and the `Target` processor begins execution.

4.3 First Steps: Building the 1-Stage Processor

In this lab, four different processors are provided: a 1-stage processor, a 2-stage processor, a 5-stage processor, and a micro-coded processor. The 5-stage processor implements both a fully-bypassed pipeline and a no-bypassing/fully interlocked pipeline.

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

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

Navigate to the `chisel/` directory and execute the following command:

```
inst$ cd ${LAB1ROOT}/chisel/
inst$ make run-emulator
```

If this is your first time running `sbt`, this command may take a while. The command `make run-emulator` does the following:

- opens `sbt`, the Scala Built Tool, selects the rv32_1stage project, and runs the `Chisel` code which generates a C++ cycle-accurate description of the processor. The generated C++ code can be found in `${LAB1ROOT}/chisel/emulator/rv32_1stage/generated-src/`.
- compiles the generated C++ code into a binary called `emulator`.
- calls the RISC-V front-end server (called `fesvr`), which opens a socket to your C++ binary `emulator`, and sends it a RISC-V binary for the target processor to execute (See Figure 2). All of the RISC-V tests and benchmarks will be executed when calling "`make run-emulator`".[4]

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

[4]Which tests and benchmarks are executed can be found in the ${LAB1ROOT}/chisel/emulator/common/Makefile.include.

A `PASS` should be generated by each program. If you see any `FAIL`s, verify you are running on a recommended instructional machine. Otherwise, contact your TA.

Building Other Processors

By default, `make run-emulator` builds the 1-stage processor (`rv32_1stage`). To build the other processors:

```
inst$ cd ${LAB1ROOT}/chisel/
inst$ export MK_TARGET_PROC=rv32_2stage
inst$ make run-emulator
```

The bash variable `${MK_TARGET_PROC}` tells the `Makefile` which processor you want to build. The valid options are:

```
inst$ export MK_TARGET_PROC=rv32_1stage
inst$ export MK_TARGET_PROC=rv32_2stage
inst$ export MK_TARGET_PROC=rv32_5stage
inst$ export MK_TARGET_PROC=rv32_ucode
```

4.4 Instruction Mix Tracing Using the 1-Stage Processor

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

```
inst$ cd ${LAB1ROOT}/chisel/
inst$ export MK_TARGET_PROC=rv32_1stage
inst$ make run-emulator
inst$ cd emulator/rv32_1stage/
inst$ ls
inst$ emacs vvadd.riscv.out
```

We have provided a set of benchmarks for you to gather results from: `median`, `multiply`, `qsort`, `towers`, and `vvadd`. Using your editor of choice, look at the output files generated from `make run-emulator`.[5] The processor state is written to the output file on every cycle. At the end of the file, statistics from the "Trace" object can be found:

[5] To speed up parsing data out of all of the benchmark output files, type "grep \# *.riscv.out" to dump all trace information to stdout.

```
#---------- Tracer Data ----------
#
#      CPI   : 1.00
#      IPC   : 1.00
#      cycles: 2704
#
#      Bubbles     : 0.000 %
#      Nop instr   : 0.000 %
#      Arith instr : 55.621 %
#      Ld/St instr : 33.284 %
#      branch instr: 11.095 %
#      misc instr  : 0.000 %
#---------------------------------
```

A few things to note: software compiler generated NOPs *do* count towards the instruction count but machine-inserted "bubbles"[6] do *not*, and "Tracer" only counts data when the processor's "Stats Enable" register is set (co-processor register "cr10"). Also, the denominator used for calculating the percentages is "cycles."

Note how the mix of different types of instructions vary between benchmarks. Record the mix you observed for each benchmark (remember: don't provide raw dumps!). 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?[7]

4.5 Thought problem

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 cycle? Is this still a worthwhile modification if it means that the cycle time is increased 30%? Is it worthwhile for all benchmarks, or only some? Explain.

4.6 CPI Analysis Using the 5-Stage Processor

For this section we will analyze the effects of branching and bypassing in a 5-stage processor.[8]

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

[6] A "bubble" is inserted, for example, when the 2-stage processor takes a branch and must kill the Instruction Fetch stage.

[7] If you would like to see the disassembly of any benchmark, you can visit ~cs152/install/riscv-bmarks/, and view the *.riscv.dump files. You can also use `riscv-objdump` to create your own disassembly files.

[8] The 2-stage processor will not be explicitly used in this lab, but it is provided to show how pipelining in a very simple processor is implemented. Likewise, the micro-coded processor is also not explicitly used in this lab.

First, we must set the pipeline to "Full-Bypassing". Navigate to the `Chisel` source code:

```
inst$ cd ${LAB1ROOT}/chisel/src/rv32_5stage
inst$ emacs consts.scala
```

The file `consts.scala` provides all constants and machine parameters for the processor. Change the parameter on line 18 to `"val USE_FULL_BYPASSING=true;"`. You can see how this parameter changes the pipeline by looking at the data path in `dpath.scala` (lines 156-175) and the control path in `cpath.scala` (lines 168-185). The data path holds the bypass muxes used when full bypassing is activated. The control path holds the `stall` logic, which must account for more situations when no bypassing is supported.

After turning "full bypassing" on, compile and run the processor as follows:

```
inst$ cd ${LAB1ROOT}/chisel/
inst$ export MK_TARGET_PROC=rv32_5stage
inst$ make run-emulator
inst$ cd emulator/rv32_5stage/
inst$ ls
inst$ emacs vvadd.riscv.out
```

Record the CPI value for all benchmarks. Is it what you expected?

Now turn "full bypassing" off in `consts.scala`, and re-run the results (make sure it recompiled your `Chisel` code).

Record the new CPI values for all benchmarks. How does full bypassing versus full interlocking perform? If adding full-bypassing hurt the cycle time of the processor by 50%, would it be worth it? Argue your case. Be quantitative.

4.7 Design problem

Imagine that you are being asked by your employer to evaluate a potential modification to the design of a 5–stage RISC-V. 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 Section or OH.

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 "Tracer" class found in the source files `tracer.cpp` and `tracer.h` (located in the `${LAB1ROOT}/chisel/emulator/common/` directory).

Modify "Tracer" to detect the percentage of instructions that are loads and stores with non–zero offsets.

Follow the steps laid out in the `tracer.h` file to accomplish this task. There is existing code provided in "Tracer" which you can follow to implement your modifications.

Use the provided RISC-V ISA specification (found in "extra course materials" on the CS 152 webpage) to determine which bits of the instruction correspond to which fields.

After modifying `tracer.h` and `tracer.cpp`, you can re-compile and re-run your data with "`make run`" in the `${LAB1ROOT}/chisel/emulator/rv32_1stage/` directory.[9]

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.)

5 Open-ended Portion

Pick *one* of the following questions. The open-ended portion is worth a little more than half the grade of the lab, so spend the appropriate amount of time and energy on it. Also, have fun with it!

5.1 Mix Manufacturing

The goal of this section is to investigate how effectively (or ineffectively) the compiler will handle complicated C code created by you.

Using no more than 15 lines of C code, attempt to produce RISC-V assembly code with the maximum ratio of branch to non–branch instructions. In other words, try to produce as many branch instructions as possible. Your goal is to create the worst possible CPI for the 5–stage processor. Your C code can contain as many poor coding practices as you like, but limit yourself to one statement per line and do not cheat by calling functions or executing any code not contained within the 15 line block. Your code must terminate. You can use code that creates jumps, but jump instructions do not count; only conditional branches count.

Write your code into the file `${LAB1ROOT}/test/test-bmarks/mix_manufacturing/mix_manufacturing_main.c`. Modify it to add your custom code.

[9]You do not need to run "make run-emulator" from the ${LAB1ROOT}/chisel/ directory if you do not need to recompile the Chisel code for your target processor. Instead you can stay in the emulator ${LAB1ROOT}/chisel/emulator/rv32_1stage/ directory, and only rebuild the C++ simulator.

To test for correctness you can just compile and run it as follows:

```
inst$ cd ${LAB1ROOT}/test/test-bmarks/
inst$ make; make run
```

This invokes the RISC–V ISA simulator, which quickly tests the correctness of the code. You may also view the disassembly by opening the file `mix_manufacturing.riscv.dump`.

However, to get a cycle-accurate trace of the code to determine the effect your program has on CPI, you will have to run the code on the RV32 5-stage processor:

```
inst$ cd ${LAB1ROOT}/test/test-bmarks/
inst$ make
inst$ cd ${LAB1ROOT}/chisel/emulator/
inst$ export MK_TARGET_PROC=rv32_5stage
inst$ make run-emulator
```

Report the ratio of branches to non–branches that you achieved in your code. You will submit this mix report, the achieved CPI of the 5-stage processor, your lines of C code, and the disassembly of your C code.

5.2 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 make the budget and ship on time!

Using the `Chisel` source code found in `${LAB1ROOT}/chisel/src/rv32_5stage/`, analyze the impact on CPI when different bypass paths are removed from the design. The files `dpath.scala` and `cpath.scala` hold the relevant code for modifying the bypass paths and stall logic. Make sure that your modified pipeline passes all of the assembly tests!

Show your data to support the bypass path you feel can be removed with the least impact on CPI. Also, include in an appendix snippets of your modified `Chisel` code.

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

5.3 Processor Design

Propose a processor modification of your own to a 5–stage pipeline. Justify your design modification's overhead, cost, or motivation by explaining which instructions are affected by the changes you propose and in what way. You may have to draw a diagram explaining your proposed changes for clarity, and you will very likely have to modify the "Trace" object 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 make your case, but you do *not* need to implement your proposed processor design.

5.4 ISA Design

Imagine you are modifying the RISC-V ISA to allow for two sizes of instructions (let's pretend that it can't already, and that you are creating a competing proposal). With this modification, you will be able reduce the total code size of programs compiled for your machine by allowing some instructions to be encoded in a more compact form. Pick an instruction format (they are listed in the RISC-V handout) and make a proposal for why it should be the one which is allowed to be reduced in size. What is your proposal for reducing the size of this format, and how long is this smaller encoding? By how much does including the variable length instruction reduce code size for the benchmarks we looked at earlier? You can use the full RISC-V manual (`http://www-inst.eecs.berkeley.edu/~cs152/sp12/handouts/riscv-spec.pdf`)) if you want to find additional instruction types which use your proposed format. Be quantitative, and use the provided testing infrastructure to do experiments and make informed decisions. For example, if you propose shorter immediates, run traces on the provided benchmarks to show *exactly* how many instructions you can shorten (i.e., how many instructions can utilize the smaller immediate). Consult with your TA if you need further information on probing the state of the emulator.

5.5 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 your own design. However, you must first consult with the professor and TA to ensure your idea is of sufficient merit.

6 The Third Portion: Feedback

This is a brand new lab, and as such, your TA would like your feedback! 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).

7 Acknowledgments

This lab is 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 Asanovic who developed the RISC-V ISA.

8 Appendix: Processor Diagrams

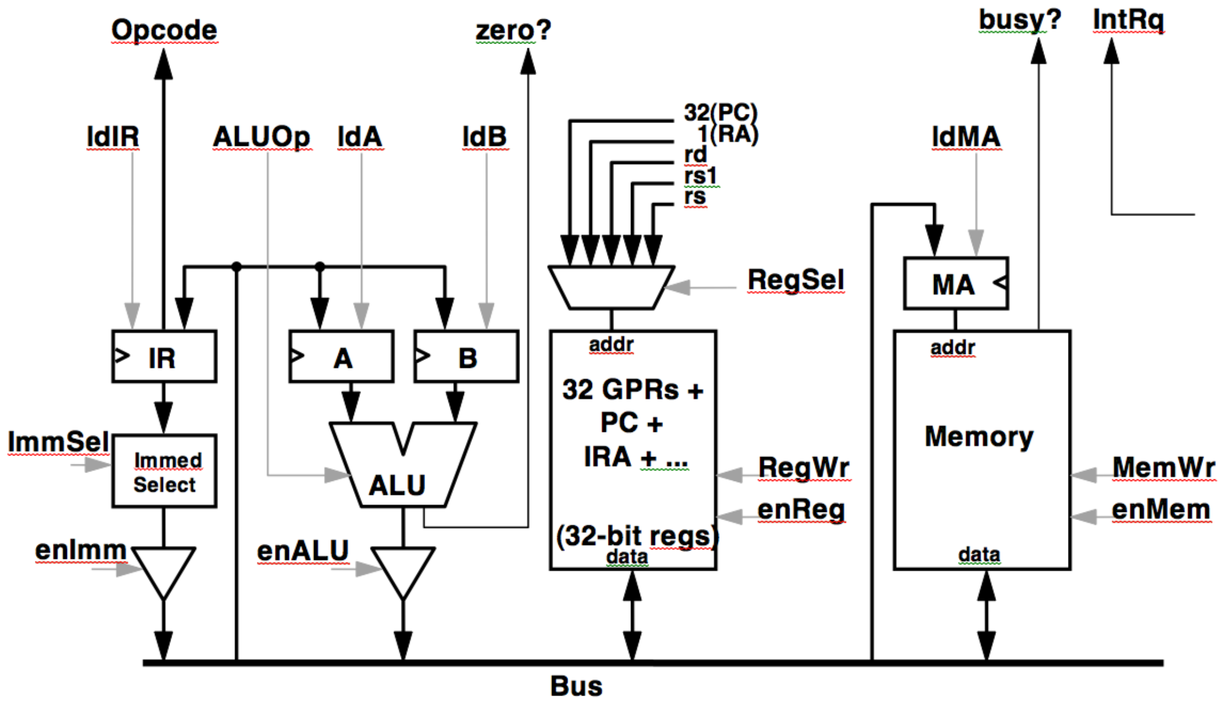

Figure 3: The Bus-Based RISC-V Implementation.

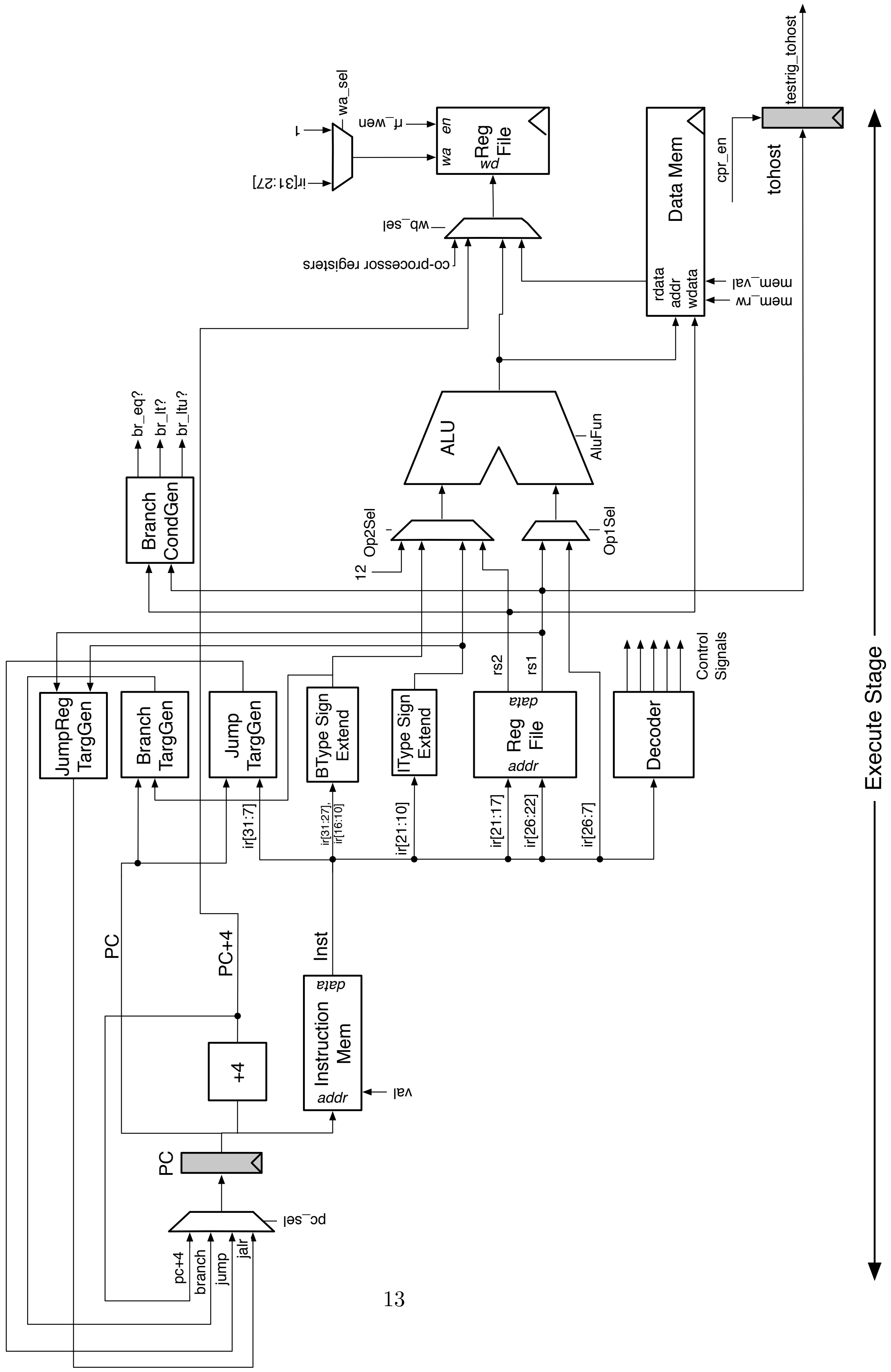

Figure 4: The RV32 1-Stage Processor.

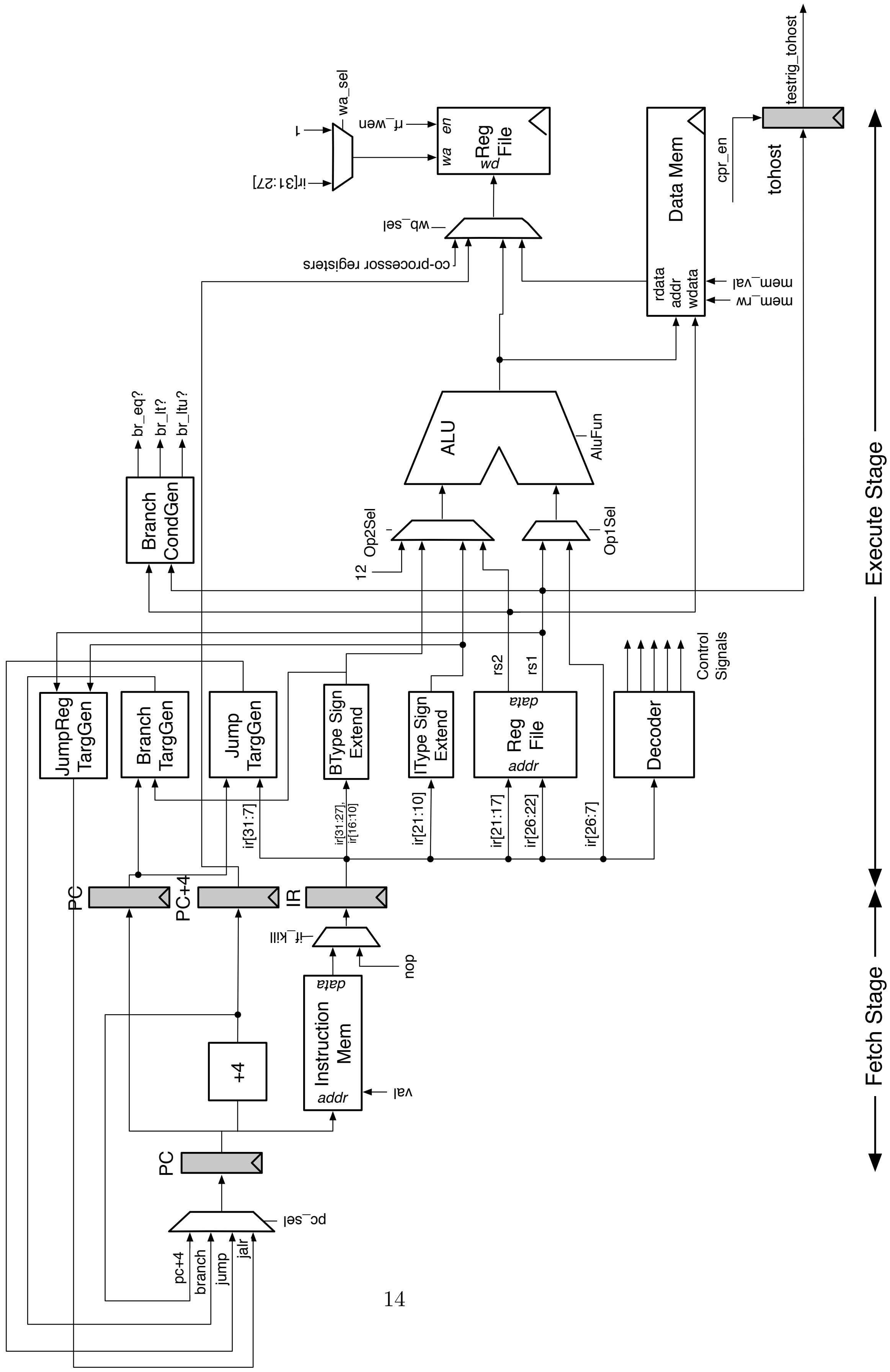

Figure 5: The RV32 2-Stage Processor.