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 complete the questions on their own. That is, all submitted code, as well as written reports, must be students’ own work.

1.1 Graded Items

You will turn in a lab report as part of your lab repo as a file report.pdf. We will add instructions for submitting the completed lab soon. Please label each section of the results clearly.

1. (Directed) Problem 4.6: recorded instruction mixes for each benchmark and answers
2. (Directed) Problem 4.7: thought problem answers
3. (Directed) Problem 4.8: CPI analysis answers
4. (Directed) Problem 4.9: design problem answers

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, riscv-g++ cross-compilers build new binaries from RISC-V assembly, C, and C++ source codes. The riscv-objdump tool can disassemble existing RISC-V binaries to show the exact sequence of instructions being executed.

The ISA simulator riscv-isa-sim can execute RISC-V binaries. The ISA simulator serves as the golden reference for the ISA. 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 online at riscv.org, and in the doc/ subdirectory of the lab1 repo.

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. For more details, please visit http://chisel.eecs.berkeley.edu.

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://chisel.eecs.berkeley.edu. Students will not be required to write Chisel code as part of this lab, beyond changing and adding parameters as directed.

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 on which you run the simulators. 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 are your target machines.
Figure 1: Chisel Design Flow. Only the C++ Simulator path will be exercised for this lab.

4.2 Setting Up Your Lab1 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).

4.2.1 Obtaining Lab1 from Github

We will be using github (http://github.com) to distribute the labs, and for collecting student submissions. To that end, all students should create a github userid and send it to the GSI (cs152sp14ta@gmail.com) as soon as possible. All userids will be added to the ucberkeley-cs152-sp14 organization, and students will have access to the ucberkeley-cs152-sp14 / lab1 repo.

To download the lab, first visit the repo site on github. You should see a page like that shown in figure 2.

You will need to click the “fork” button shown in the upper right hand corner. Follow the pop-up box to create a copy of the repo local to your personal github account. When you have done this, navigate to the github page for your forked copy of lab1 (the top of the page should look like Figure 3). Copy the repo URL as shown in Figure 4 and then clone it on the instructional machine into your home directory:
Figure 2: Lab1 Repository on Github. You will need to fork a copy of this repo once you have access to it.

Figure 3: The Github page for your forked version of the repo should look like this (where “ericlove” is replaced by your username)

Figure 4: Copy this link when you run git clone. If you’ve uploaded you SSH key to github, using the SSH link will be more convenient, as you won’t have to log in each time you push or pull.
4.2.2 Getting Updates to Lab1

One motivation for our use of git in this class is that it allows us to continue improving the lab repo after it has been released. When a change is made to the lab repo, you can fetch those changes into your fork.

To setup the forwarding of these changes, follow these directions (taken from https://help.github.com/articles/fork-a-repo) from within lab1/:

```
inst$ git remote add upstream git@github.com:ucberkeley-cs152-sp14/lab1.git
```

Execute the above command once to add the lab1 base repo as a remote source to your fork. Then, each time an update is made to the lab, run git fetch to retrieve it:

```
inst$ git fetch upstream
```

This will add the revisions as a new branch in your local repo. You will then want to merge these changes into your master branch with this command:

```
inst$ git merge upstream/master
```

4.3 Setting Up Your Chisel Workspace

You’ll need to set up your local environment on the EECS instructional machines in order to use Chisel and the RISCV toolchain. We’ve done most of the work for you by installing this software on the instructional cluster, but you’ll need to add a line to your .bash_profile script (which runs each time you log in using SSH):

```
source ~/cs152/sp14/.bashrc
```

Copy the line below and add it to the end of ~/.bash_profile using your favorite editor. Then, make sure you log out and log back in, or type

```
inst$ source ~/.bash_profile
```

before you continue. You’ll know you’ve succeeded if you see a message “CS152 Login Script...” You may also want to set some environment variables of your own. For example:

```
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}/`
  - Makefile
  - src/ Chisel source code for each processor.
    - common/ Common source code shared between all processors.
    - rv32_1stage/ Source code for the RISC-V 1-stage processor
    - rv32_2stage/ ...
    - rv32_5stage/ ...
  - emulator/
    - common/ Common emulation infrastructure shared between all processors.
    - rv32_1stage/ C++ simulation tools and output files.
    - rv32_2stage/ ...
    - rv32_5stage/ ...
  - test/ Local source code for benchmarks and tests.
    - riscv-bmarks/ Local benchmarks written in C.
    - riscv-is/ Local tests written in assembly.
  - install/ Compiled assembly tests and benchmarks.
  - chisel/ Chisel voodoo. You can safely ignore this directory.
  - sbt/ 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}/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.

### 4.4 First Steps: Building the Sodor Processors

In the directed portion lab, three different processors are provided: a 1-stage processor, a 2-stage processor, and a 5-stage processor. The 5-stage processor implements both a fully-bypassed pipeline and a no-bypassing/fully interlocked pipeline.

The first time you build the processors, you’ll need to run the configure script as follows:

```sh
inst$ cd ${LAB1ROOT}/
inst$ ./configure
```

Now, to compile and run the processors, type

```sh
inst$ make run-emulator
```

---

1 A micro-coded processor and a 3-stage will be available during the next portion of the lab. You can ignore those directories of the repo for now, however.
Figure 5: 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.

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

- runs sbt, the Scala Built Tool, selects each processor as the “project”, and runs the Chisel code which generates a C++ cycle-accurate description of the processor. The generated C++ code for the 1-stage can be found in ${LAB1ROOT}/emulator/rv32_1stage/generated-src/, with the code for other processors in analogously named directories.
- compiles the generated C++ code into a binary called emulator.
- launches the compiled binary, which calls the RISC-V front-end server (called fesvr, linked into the emulator as libfesvr), sends it a RISC-V binary for the target processor to execute (See Figure 5). All of the RISC-V tests and benchmarks will be executed when calling “make run-emulator”.3

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

4.5 Working with Individual Processors

The commands you ran in the previous section will build an run all the Sodor processors. Usually you’ll just want to build one processor at a time. Suppose you just want to build the 1-stage. Go to its emulator directory and do the following:

    inst$ cd ${LAB1ROOT}/emulator/rv32_1stage/
    inst$ make run

If you’re ever uncertain what these commands are doing, or need to know how to do something different, examine the makefiles. Note that the Makefile in each emulator directory is just a processor-specific stub that actually includes a larger makefile in ../common/Makefile.include. Look there for more information on the various run commands.

2If you get a java.lang.OutOfMemoryError exception, run make run-emulator again.
3Which tests and benchmarks are executed can be found in the ${LAB1ROOT}/emulator/common/Makefile.include.
4.6 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.

```bash
inst$ cd ${LAB1ROOT}/emulator/rv32_1stage
inst$ make run
inst$ ls output
inst$ vim output/vvadd.riscv.out
```

We have provided a set of benchmarks for you to gather results from: median, dhrystone, qsort, towers, and vvadd. Using your editor of choice, look at the output files generated from make run.\(^4\) 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:

\(^4\)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: 3029
#
# Bubbles : 0.000 %
# Nop instr : 0.000 %
# Arith instr : 59.756 %
# Ld/St instr : 30.175 %
# branch instr: 9.937 %
# misc instr : 0.132 %
#-----------------------------------

A few things to note: software compiler-generated NOPs do count towards the instruction count but machine-inserted “bubbles” do not. 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?

4.7 CPI analysis 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.

4.8 CPI Analysis Using the 5-Stage Processor

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

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.

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

---

5 A “bubble” is inserted, for example, when the 2-stage processor takes a branch and must kill the Instruction Fetch stage.

6 If you would like to see the disassembly of any benchmark, you can visit lab1/install/riscv-bmarks/, and view the *.riscv.dump files. You can also use riscv-objdump to create your own disassembly files.

7 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.
The file consts.scala provides all constants and machine parameters for the processor. Change the parameter on line 19 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 186-220) and the control path in cpath.scala (lines 193-212). 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 ${LAB1ROOT}/emulator/rv32_5stage
inst$ make run
inst$ ls output
inst$ vim output/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 25%, would it be worth it? Argue your case. Be quantitative.

4.9 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 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 ≠ 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}/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 the doc/ subdir of the lab1 repo, and on riscv.org) 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}/emulator/rv32_5stage/ directory.

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

We will post a new version of the lab document when this section is ready. It will have a separate due date, which we will announce when it is released. Doing the directed portion now will prepare you to complete this part in the future.

6 Acknowledgments

Many people have contributed to versions of this lab over the years. This lab was originally developed for CS152 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.
7 Appendix: Processor Diagrams
Figure 6: The RV32 1-Stage Processor.
Figure 7: The RV32 2-Stage Processor.
Figure 8: The RV32 5-Stage Processor.