本实验室作业的目标是让您熟悉 Chisel 模拟环境，
同时还允许您进行一些简单的实验。 通过修改现有的指令跟踪器脚本，
您将收集指令混合统计数据并根据结果提出一些架构建议。
实验室有两个部分，一个定向部分和一个开放式部分。
每个人都会以相同的方式完成指导部分，并根据正确性分配等级。
开放式部分将使您能够进行更具创造性的调查，
您的成绩将基于完成任务的努力或您为支持您的想法而提供的论据。
鼓励学生与其他学生讨论实验室作业的解决方案，
但必须自己完成实验室的指导部分，并针对这些问题提交自己的实验室报告。
对于每个实验室的开放式部分，学生将单独或以二人或三人为一组进行工作。
每个小组将为实验室的开放式部分提交一份报告。
学生可以自由参加不同的小组进行不同的实验室作业

Project 1

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 script, 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 those problems. For the open-ended portion of each lab, students will 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. 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 at the beginning of class on the due date. Please label each section of the results clearly. The directed items need to be turned in for evaluation. Your group only needs to turn in *one* of the problems found in the Open-Ended Portion.

1. (Directed) Problem 4.4: recorded instruction mixes for each benchmark and answers
2. (Directed) Problem 4.5: thought problem answers
3. (Directed) Problem 4.6: CPI analysis answers
4. (Directed) Problem 4.7: design problem answers
5. (Open-ended) Problem 5.1: source code and recorded ratio
6. (Open-ended) Problem 5.2: data and the modified section of Chisel source code
7. (Open-ended) Problem 5.3: instruction definition, test code, worksheet, modified section of Chisel source code
8. (Open-ended) Problem 5.4: design proposal and supporting data
9. (Directed) Problem 6: Feedback on this lab

Lab reports must be in *readable* English and not raw dumps of log-files. Your lab reports must be typed and the open-ended portion **must not exceed 6 pages**. 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, developed at UC Berkeley for use in education, research, and industry.

An entire toolchain is provided. The riscv64-unknown-elf-gcc, riscv64-unknown-elf-g++ cross-compilers build new binaries from RISC-V assembly, C, and C++ source codes. The riscv64-unknown-elf-objdump tool can disassemble existing RISC-V binaries to show the exact sequence of instructions being executed.

The ISA simulator riscv-isa-sim also known as spike 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 in the "resources" section of the CS 152 website or directly at http://riscv.org/spec/riscv-spec-v2.0.pdf. For Lab 1, all processors implement the 32-bit variant, known as RV32.

ISA 模拟器 riscv-isa-sim 也称为 spike，可以执行 RISC-V 二进制文件。ISA 模拟器是 ISA 的黄金参考。 它不是周期精确的，但它执行得非常快。前端服务器fesvr 加载RISC-V 二进制文件并连接到ISA 模拟器或Chisel 创建的模拟器。

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 was designed to construct actual hardware.

Chisel can generate low-level Verilog code for mapping designs to FPGA, ASICs, or cycle-accurate software simulators like VCS or Verilator.

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 Verilog code.

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 Verilog software simulators, and use the 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.

4 Directed Portion (40% of lab grade)

4.1 Terminology and conventions

Throughout this course, the term *host* refers to the machine on which you run the simulators, while *target* refers to the simulated machine. For this lab, the host will be an instructional machine (inst$), while the provided RISC-V processors are your target machines.

4.2 Setting Up Your Chisel Workspace

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 _rst to be simulation languages, Chisel is designed speci_cally for constructing actual hardware. Chisel can generate low-level Verilog code for mapping designs to FPGA, ASICs, or high-speed cycle-accurate software simulators like VCS or Verilator. 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 Verilog code.

Chisel in This Lab

Provided with this lab are five different processors: a 1-stage pipeline, a 2-stage pipeline, a 3-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 Verilog software simulators, and use the 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.

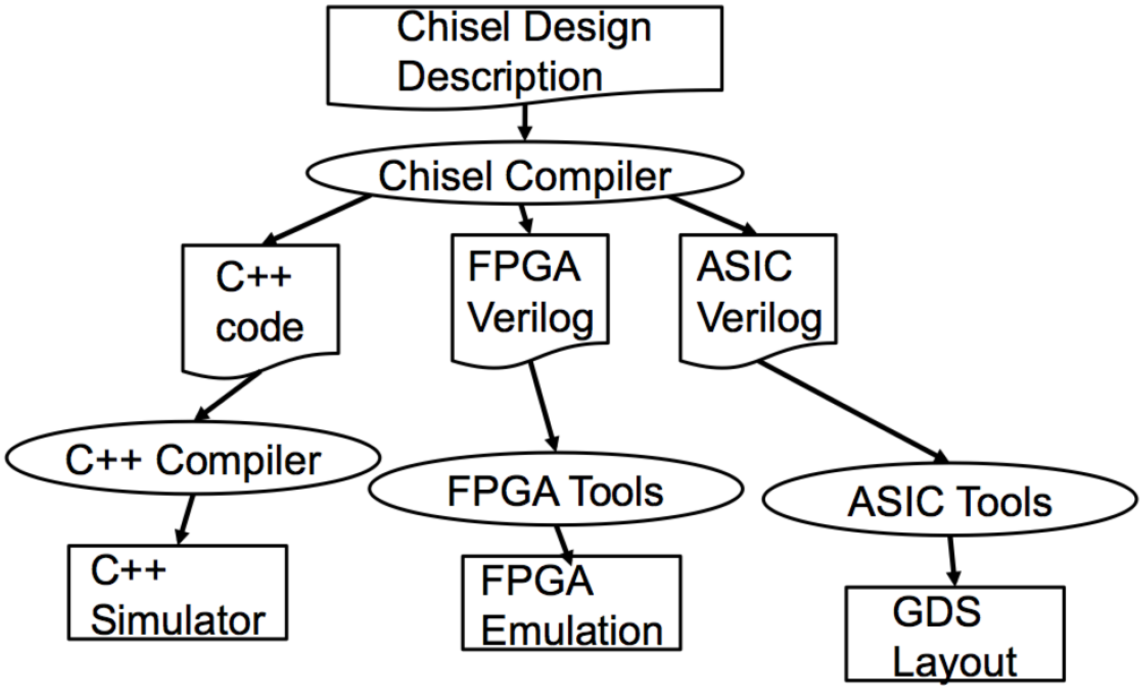

Figure 1: Chisel Design Flow. Only the C++ Simulator path will be exercised for this lab.

Getting the repo

First, clone the lab materials. The --recursive flag is necessary in order to checkout the submodules in the git directory.

```
git clone https://github.com/ucb-bar/riscv-sodor.git --recursive
```

Building the processor emulators

Install verilator

For Ubuntu 17.04

```
sudo apt install pkg-config verilator
#optionally gtkwave to view waveform dumps
```

For Ubuntu 16.10 and lower

```
sudo apt install pkg-config
wget http://mirrors.kernel.org/ubuntu/pool/universe/v/verilator/verilator_3.900-1_amd64.deb
sudo dpkg -i verilator_3.900-1_amd64.deb
```

Install the RISC-V front-end server

```
cd riscv-fesvr
mkdir build; cd build
../configure --prefix=/usr/local
make install
```

Build the sodor emulators

```
./configure --with-riscv=/usr/local
make
# To run the all the stages with the given tests available in ./install
make run-emulator
# To install the executables on the local system
make install
# Clean all generated files
make clean
```

Gathering the results

```
(all)   $ make reports
(cpi)   $ make report-cpi
(bp)    $ make report-bp
(stats) $ make report-stats
```

(Optional) Running debug version to produce signal traces

```
make run-emulator-debug
```

When run in debug mode, all processors will generate .vcd information (viewable by your favorite waveform viewer). All processors can also spit out cycle-by-cycle log information. Although already done for you by the build system, to generate .vcd files, see emulator/common/Makefile.include to add the "-v${vcdfilename}" flag to the emulator-debug binary.

RISC-V fesvr allows you to use elf as input to sodor cores so no need to generate the hex files

The directory structure is shown below:

- 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/ ...
  - 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/ ...
- test/ Local source code for benchmarks and tests.
  - custom-bmarks/ Local benchmarks written in C.
  - custom-tests/ Local tests written in assembly.
- install/ Compiled assembly tests and benchmarks.
- doc/ Various documentation.
- project/ Scala 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 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.

4.3 First Steps: Building the 1-Stage Processor

In this lab, five different processors are provided: a 1-stage processor, a 2-stage processor, a 3-stage processor, a 5-stage processor, and a microcoded processor. The 5-stage processor implements both a fully-bypassed pipeline and a no-bypassing/fully interlocked pipeline.

Navigate to the riscv-sodor directory and execute the following command:

```
cd emulator/rv32_1stage
make run
```

If this is your first time running sbt, this command may take a while.[2] The command make run does the following:

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

- runs sbt, the Scala Build Tool, selects the rv32 1stage project, and runs the Chisel code which generates a Verilog RTL description of the processor. The generated Verilog code can be found in emulator/rv32_1stage/generated-src/.
- runs verilator, an open-source tools that compiles Verilog to cycle-accurate C++ simulation code
- compiles the generated C++ code into a binary called emulator.
- runs the emulator binary, loading the provided RISC-V binaries into the simulated memory. All of the RISC-V tests and benchmarks will be executed when calling \make run".

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.

Building Other Processors

Depending on which directory you run commands in some environmental variables will be set selecting the type of processor to build. If you are in ${LABROOT}/ the make run command will build all processors by default. To build specific other processors:

```
cd emulator/rv32_2stage
make run
```

This will build the 2 stage variant. The valid processor variants are rv32_1stage, rv32_2stage, rv32_3stage, rv32_5stage, rv32_ucode.

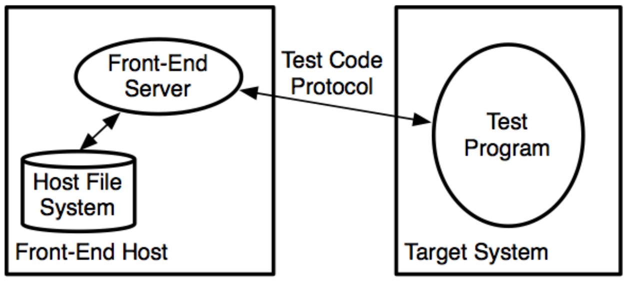

Figure 2: The Testing Environment. The front-end server (fesvr) loads the RISC-V binary from the Host file system, starts the Target system simulator, and sends the RISC-V binary code to the Target simulator to populate the simulated processor's instruction memory with the program. Once the fesvr finishes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

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.

```
cd  emulator/rv32_1stage
make run
vim output/vvadd.riscv.out
```

We have provided a set of benchmarks for you to gather results from: median, multiply, qsort, rsort, towers, dhrystone, and vvadd. Using your editor of choice, look at the output files generated from make run.[1] The processor state is written to the output file on every cycle. At the end of the file, statistics from the “tracer” script can be found:

```
#----------- 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 %
#-----------------------------------
```

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

A few things to note: software compiler-generated NOPs *do* count towards the instruction count but machine-inserted "bubbles"[5] 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 – one 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?

4.5 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.

4.6 CPI Analysis Using the 5-Stage Processor

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

```
cd src/rv32_5stage
vim consts.scala
```

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 209-241) and the control path in cpath.scala (lines 220-239). 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:

```
cd emulator/rv32_5stage/
make run
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.

4.7 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 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 Section or OH.

In RISC-V, the effective address of a load or store is calculated by summing the contents of one register (*rs*1) 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* /= 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 python script tracer.py (located in the

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.py 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 to determine which bits of the instruction correspond to which fields.

After modifying tracer.py, you can re-run your data with "make run" in the 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. Be quantitative.)

5 Open-ended Portion (60% of lab grade)

Pick *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!

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 (and no inline assembly or comma operators), attempt to produce RISC-V assembly code with the maximum ratio of branch to non-branch instructions when run on the 5-stage processor (fully bypassed). In other words, try to produce as many branch instructions as possible. 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.

Finally, run your code on the 5-stage processor (fully bypassed). What is the resulting CPI? As you added more branches, did the CPI get higher or lower? Explain why the CPI went in the direction it did.

Write your code into the file test/custom-bmarks/mix.c[2]. Modify it to add your

custom code. In your write-up, summarize some of the ideas you tried.

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

[2]Note: the provided processors only support load word and store word; therefore, you should avoid char variables which synthesize load byte unsigned and store byte instructions.

```
cd test/custom-bmarks/
make; make run-riscv
```

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.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:

```
cd test/custom-bmarks/
make
cd ../../emulator/rv32_5stage
export local_bmarks=mix
make run
```

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 Define and Implement Your Favorite Complex Instruction

In the problem set, we have asked you to implement two complex instructions (ADDm and STRLEN) in the micro-coded processor. Imagine you are adding a new instruction to the RISC-V ISA. Propose a new complex instruction (other than MOVN/MOVZ) that needs an EZ/NZ uBr in its implementation, and has at least one memory operand. Write an assembly test for the proposed instruction. Implement the instruction in the micro-coded processor, and test your new instruction. To define an instruction, you first need to come up with an encoding. Consult the RISC-V ISA document to first come up with an instruction format (see section 2.2 of the RISC-V base ISA specification), and then spot an opcode space that is empty (look at Table 19.1 of the RISC-V base ISA specification). Note that the custom-0/1/2/3 and reserved spaces are currently available. You will have to add your instruction definition to src/common/instructions.scala
(please search for FIXME in the file). Let’s take a look at the definition for MOVN:
def MOVN = BitPat("b?????????????????????????1110111")

Note that the ? character is used for bit locations that can change (e.g., register specifiers). Bit locations that are fixed have the actual bits written down. _ characters are ignored. The name of the variable is used as a label for the dispatcher in the microcode.

Once you come up with an instruction encoding, you’ll have to write an assembly test to test your instruction. To help you out, we wrote an assembly test for the MOVN instruction. Please take a look at ${LAB1ROOT}/test/custom-tests/movn.S. Since the assembler doesn’t know about the instruction, we manually write down the instruction with a .word assembly construct. We also write some assembly code to load values to registers and memory. Finally, the code checks whether it computes the right value. We have provided you with an empty assembly file that you can use at

${LAB1ROOT}/test/custom-tests/yourinst.S (please search for FIXME in the file). Compile your assembly test:

```
cd test/custom-tests/
make rv32ui-p-yourinst
```

Next, work out the implementation in a worksheet you have used in the problem set (worksheet 2.A or 2.B). Once you have figured out all the states and control signals, start adding your microcode to src/rv32_ucode/microcode.scala (please search for FIXME in the file). You will be able to see the microcode instructions for all RISC-V instructions implemented in the micro- coded machine. Again, to provide you an example, we have implemented the MOVN instruction in microcode.scala. Once you are done, build the processor and run the assembly test:

```
cd emulator/rv32_ucode
export local_asm_tests=rv32ui-p-yourinst
make run
```

Look at the log output at emulator/rv32_ucode/output/rv32ui-p-yourinst.out to observe the machine state. See if the micro-coded processor has executed your instruction cor- rectly. Change your implementation if necessary.

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.

6 The Third Portion: Feedback

In order to improve the labs for the next offering of this course we would like your feedback. Please submit your feedback via an online form (the domain will be provided separately).

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 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).

6.1 Team Feedback

In addition to feedback on the lab, we would like you to answer a few questions about your team:

1. In one short paragraph, describe your contributions to the project.
2. Describe the contribution of each of your team mates.
3. Do you think that every member of the team contributed fairly? If no, why?

7 Acknowledgments

This lab is modified from University of California, Berkeley, CS 152 Laboratory Exercise 1.

8 Appendix: Processor Diagrams

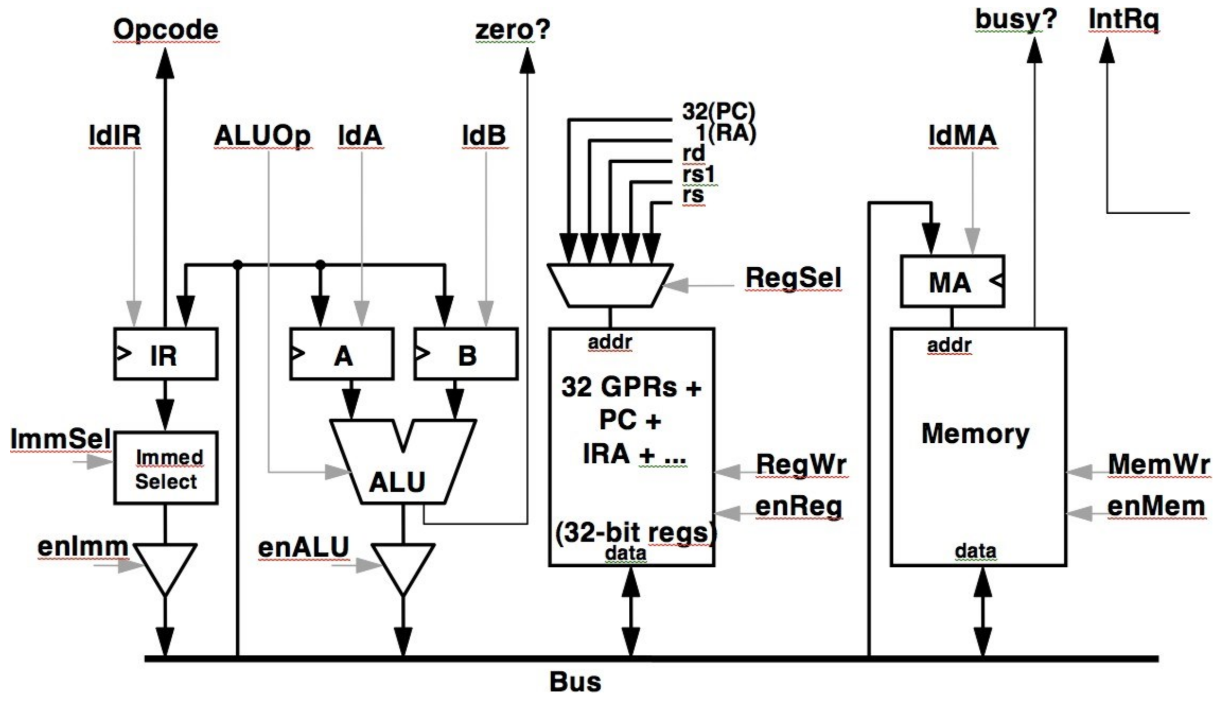

Figure 3: The Bus-Based RISC-V Implementation.

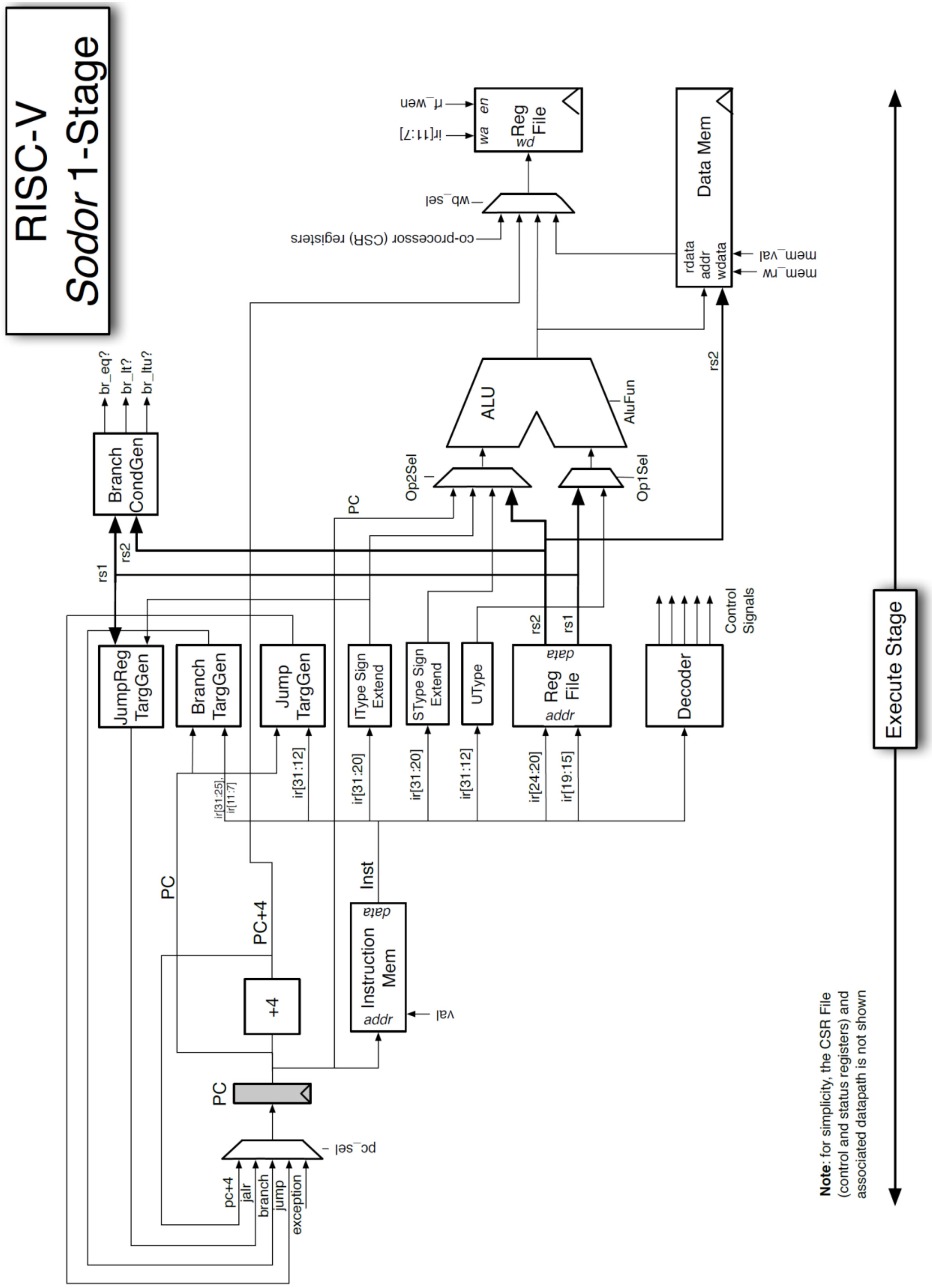

Figure 4: The RV32 1-Stage Processor.

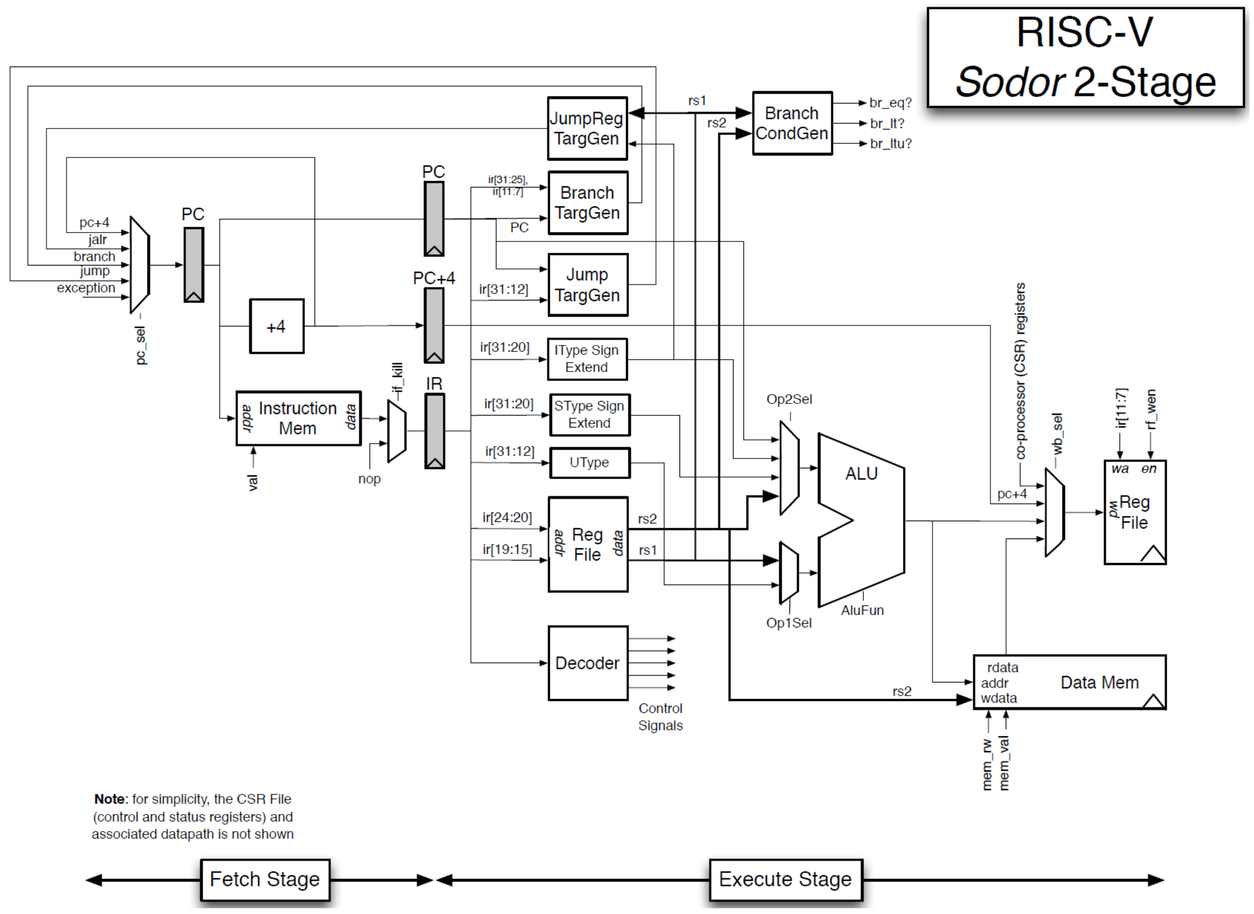

Figure 5: The RV32 2-Stage Processor.

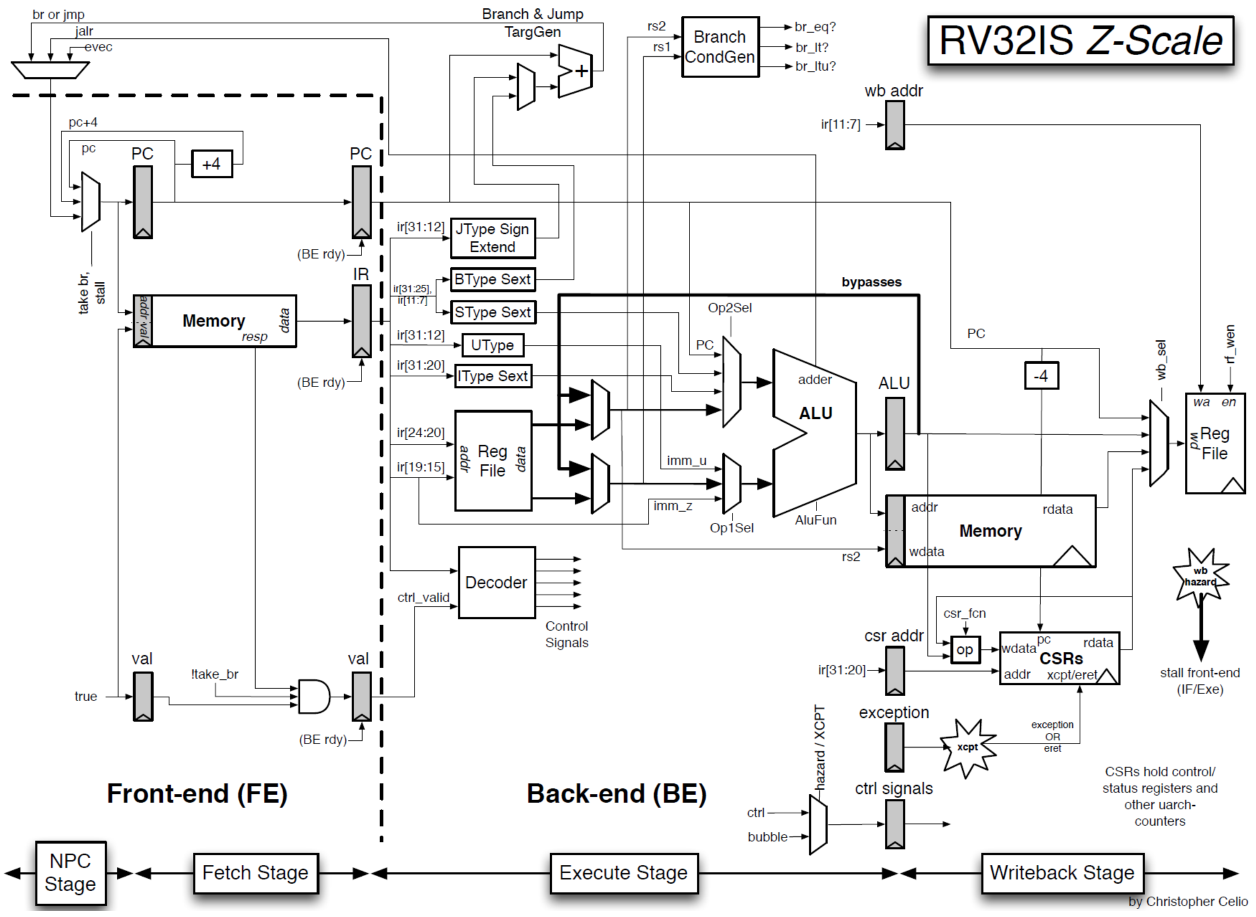

Figure 6: The RV32 3-Stage Processor.

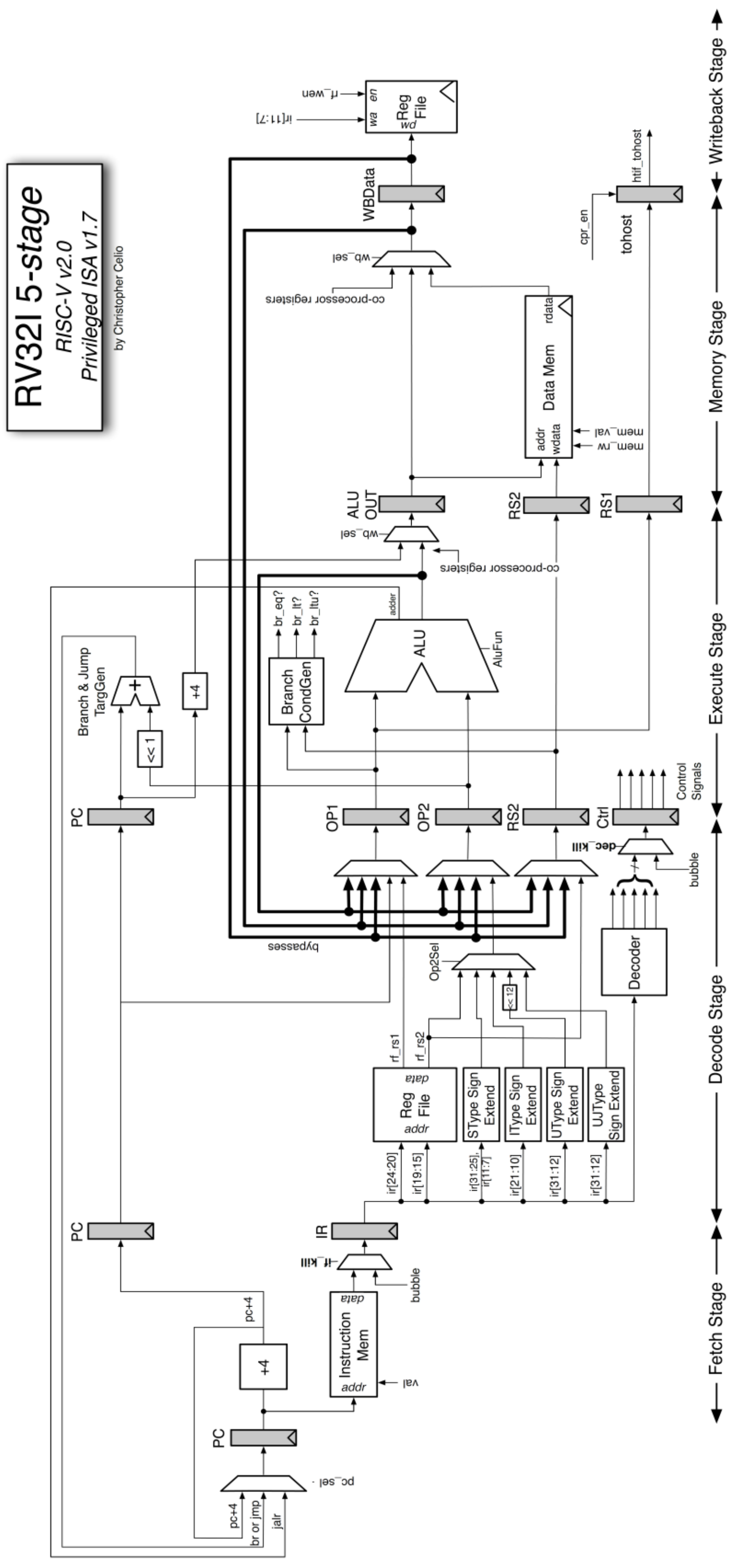

Figure 7: The RV32 5-Stage Processor.