CSC3050 Project 3: RISC-V Simulator

Deadline: April 7, 2025 at 11:59 PM

You may work individually or in a team of up to two members. Please add your team member name and student ID in your report.

1 Background

Efficient execution of instructions in a RISC-V pipeline relies on avoiding data hazards, where an instruction depends on the result of a previous instruction that has not yet completed. Data hazards can cause stalls, reducing the efficiency of the processor. To mitigate these hazards, instruction reordering and specialized fused operations like fmadd (fused multiply-add) can be utilized.

This assignment has **two** parts:

- **Implementing the fmadd Instruction** In this part, you will implement the fused multiply-add (fmadd) instruction, which performs a multiplication followed by an addition in a single step. This reduces the number of instructions executed and can eliminate certain data hazards, leading to more efficient computation.
- **Reordering Instructions to Avoid Data Hazards** You will be given a sequence of RISC-V instructions (add/mul) that suffer from data hazards. Your task will be to rearrange them while maintaining correctness. This exercise will help you understand the importance of instruction scheduling in hazard mitigation and performance optimization.

By completing this assignment, you will gain some basic hands-on experience with hazard avoidance strategies in RISC-V, while learning how fmadd can be used to optimize multiplication-addition sequences and how instruction reordering can improve pipeline execution efficiency.

2 Docker Image

You can either using the docker image, or set up your own local environment by following the next section.

You can pull the image via:

```
docker pull enderturtle/csc3050-project3:latest
```

You can also download the docker image from: https://oj.cuhk.edu.cn/project3.tar, and use

```
docker load -i project3.tar
```

Refer to tutorial 2 for details if you are not familiar with docker. The username is student, and the password is csc3050.

3 RISC-V GNU Toolchain

Notice: By using Docker, you can skip this part as it is already set up in the Docker image.

RISC-V GNU Toolchain is already in your Docker, so you do not need to download it from the official link. But we highly suggest that you open the official link and read the README.

To set up the RISC-V development environment, you need to compile and install the RISC-V GNU toolchain. This toolchain supports the RISC-V 32I instruction set with M extension (integer multiplication and division), based on the RISC-V Specification 2.2. Follow these steps to configure and compile the toolchain:

Create a build directory, configure the toolchain, and compile it with the following commands:

```
mkdir build; cd build
../configure --with-arch=rv32im --enable-multilib  --prefix=/path/to/riscv32i
make -j$(nproc)
```

4 A Simple RISC-V64I Simulator

We use a modified version of Hao He's simulator. You can find the modified repository:

https://github.com/EnderturtleOrz/CSC3050-2025-Spring-Project-3.

It is a simple RISC-V Emulator suppprting user mode RV64I instruction set, from PKU Computer Architecture Labs, Spring 2019.

4.1 Compile

Standard **C++** compile procedure with **cmake**:

```
mkdir build
cd build
cmake ..
make
```

4.2 Usage

Use this command to run the compiled files:

```
./Simulator riscv-elf-file-name [-v] [-s] [-d] [-x] [-b strategy]
```

4.3 Parameters

- `-v` for verbose output, can redirect output to file for further analysis.
- `-s` for single step execution, often used in combination with `-v`.
- `-d` for creating memory and register history dump in `dump.txt`.
- `-b` for branch prediction strategy (default BTFNT), accepted parameters are `AT`, `NT`, `BTFNT`, and `BPB`. **You can ignore this one in this assignment**
- `-x` for disabling data forwarding. **You need to implement this one**

5 Part I: RISC-V32I Simulator

The first task in this assignment is to change the RISC-V64I simulator to be RISC-V32I simulator. This is an easy job, but we suggest that you carefully read the code and know the logical structure of the simulator.

To start your project, please first clone the GitHub repository. Please read the README and this PDF carefully before starting your project. We provide many useful scripts to simply your job.

```
git clone https://github.com/EnderturtleOrz/CSC3050-2025-Spring-Project-3.git
```

You can re-compile the sample test cases to test your RISC-V32I simulator. Take `quicksort` as an example:

```
riscv32-unknown-elf-gcc -march=rv32i \
    test-basic/quicksort.c test-basic/lib.c -o riscv-elf/quicksort.riscv
```

Then, run your simulator to see the results.

```
cd build
./Simulator ../riscv-elf/quicksort.riscv
```

You can change `-march=rv32i` to `-march=rv32imf` for the remain part of the assignment. Also, we provided some useful scripts for convenience. Please check README.

6 Part I: Fused Instructions

The fused instruction is part of the RISC-V ISA's F (single-precision floating-point) and D (double-precision floating-point) extensions. These extensions provide support for floating-point arithmetic operations. In this project, you only need to implement the integer version.

Take **fmadd.s** instruction as an example.

This instruction performs a fused multiply-add operation for floating-point numbers, which means it computes the product of two floating-point numbers and then adds a third floating-point number to the result, all in a single instruction. Obviously, this operation is beneficial for both performance and precision, as it reduces the number of rounding errors compared to performing the multiplication and addition separately.

In this assignment, you are required to implement the fused instruction for **integer type**. Also, you need to support data forwarding for these instructions. We used the same format as the standard RISCV **R4** instruction. We used the reserved custom opcode **0x0B** as our opcode.

Inst	Name	funct2	funct3	Description
fmadd.i	Fused Mul-Add	0x0	0x0	rd = rs1 * rs2 + rs3
fmadd.u	Unsigned Fused Mul-Add	0x1	0x0	rd = rs1 * rs2 + rs3
fmsub.i	Fused Mul-Sub	0x2	0x0	rd = rs1 * rs2 - rs3
fmsub.u	Unsigned Fused Mul-Sub	0x3	0x0	rd = rs1 * rs2 - rs3
fmnadd.i	Fused Neg Mul-Add	0x0	0x1	rd = -rs1 * rs2 + rs3
fnmsub.i	Fused Neg Mul-Sub	0x1	0x1	rd = -rs1 * rs2 - rs3

6.1 R4 Instruction

R4 instructions, as in Figure 1, involve four registers (rs1, rs2, rs3, rd), which is different from those you are familiar with. To use standard R4 format, you need to add F-extension when compiling. In other words you

should use `-march=rv32if` but NOT change the compiling commands of RISC-V GNU Toolchain. (**Again, we are not using floating-points.**)

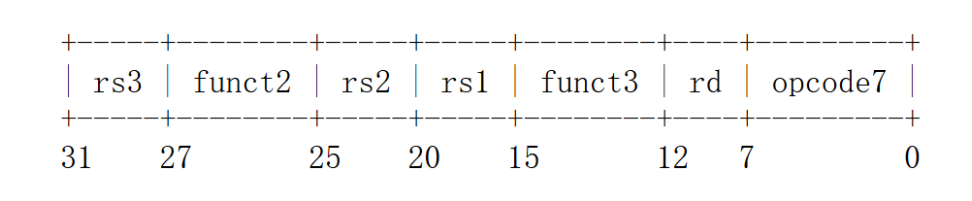

Figure 1: R4 format

6.2 Cycle counts

The fused instruction needs more cycles to process, so we define that our fused instruction needs 3 more cycles to execute. Specifically, the number of cycles required to complete this instruction is $+3$ compared to standard instructions. The **mul** instruction also incurs an additional 3 cycles, making **fmadd** more efficient in terms of cycle count.

Suppose **add** instruction takes 5 cycles to complete, then we have:

$$\text{add}(1) + \text{mul}(3) = 4 > \text{fmadd}(3)$$

6.3 Other Important Information

We also provide some basic test cases for reference. Please refer to the README and `/test-fused` under the root of the project.

Also, you can try to compare the number of cycles between fused instructions and basic mul and add instructions.

7 Part I: Disable Data Forwarding

Add an option `-x` to disable data forwarding. Note that it is **only** an option, which means that you need to make sure that your simulator can run with and without data forwarding

You can add a passing `-x` argument in the method `main` in `MainCPU.cpp`.

You need to change the logic of the simulator, i.e., we need to see your bubble/stall in single-step mode. Adding cycles alone will only earn you half the points.

8 Part II: Introduction

In this part of the assignment, you will analyze a given sequence of RISC-V instructions that suffer from data hazards. (With forwarding turned off) Your task is to rearrange these instructions while maintaining correctness, ensuring that the processor pipeline executes efficiently. Then, you should be able to further optimize it by substituting **add/mul** operations with **fmadd** operations. By strategically reordering instructions, you will learn how to reduce stalls, improve instruction throughput, and optimize execution flow in a pipelined RISC-V architecture.

8.1 Part II:Pipeline

Since this part will be covered in class, we only provide information sufficient for this project. There are totally 5 stages:

- IF(Fetch): fetch the instruction
- ID(Decode): decode the instruction for execution
- EX(Execute): execute the instruction based on the obtained information
- MEM(Memory): write/read memory
- WB(Writeback): writeback data to registers

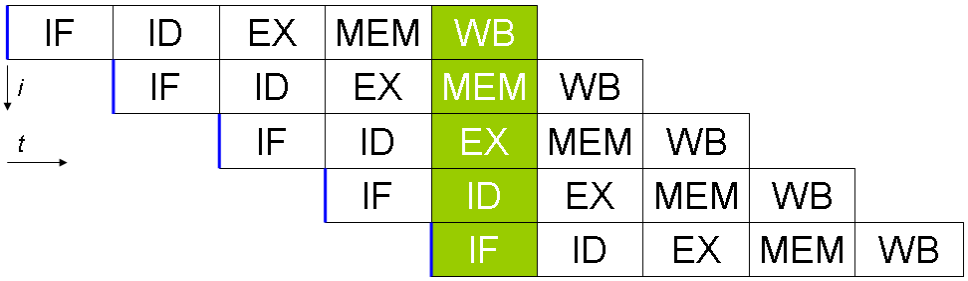

Figure 2: Pipeline

A simple example of instructions with hazard is:

- ADD x1, x2, x3
- SUB x4, x1, x5
- MUL x6, x7, x8

When **SUB** is at **DECODE** stage, it requires the value of **x1**. However, **ADD** is still at **EXECUTE** stage, where the value of **x1** is not yet written back to the register file. Hence, need to delay **SUB x4, x1, x5** for 2 cycles. If we reorder to the following order, we will have no delay.

- ADD x1, x2, x3
- MUL x6, x7, x8
- nop
- SUB x4, x1, x5

9 Part II: Rearrange

In the **part2.s** file, you will find a RISC-V program that contains several data hazards affecting pipeline efficiency. Your task is to rearrange the instructions to minimize stalls while ensuring the program produces the same output as the original. You should start by reviewing and running **part2.s** in the simulator to understand its functionality and identify potential improvements. Your optimized version should preserve correctness while reducing the number of stalled cycles. Grading will be based on both correctness and execution efficiency (fewer cycles due to reduced hazards). Name your result with **part2_p2.s**

10 Part II: Using fmadd.i

After optimizing **part2.s**, you may identify opportunities to replace certain instruction sequences with the more efficient **fmadd.i** instruction (based on either **part2.s** or **part2_p2.s**). The final optimized file, **part2_p3.s**, should produce the same output as both **part2.s** and **part2_p2.s** while improving execution efficiency. Since **fmadd.i** combines multiplication and addition into a single operation, the total cycle count should be further reduced. Grading will be based on both correctness and execution efficiency. Name your optimized file **part2_p3.s** before submission.

10.1 Part II: useful commands

Compile **.s** files into elf:

```
riscv32-unknown-elf-as test.s -o test.o
riscv32-unknown-elf-ld test.o -o test.elf
```

You can also use useful scripts on GitHub.

11 Grading Criteria

The maximum score you can get for this lab is 100 points, and it is composed by the following components:

- **Part 1** correctness of supporting RISCV32I 10 pts
- **Part 1** correctness of **fused instructions** with **data forwarding** 25 pts
- **Part 1** correctness of **all instructions without data forwarding** 20 pts
- **Part 2** correctness of **part2_p2.s** 20 pts
- **Part 2** efficiency of **part2_p2.s** 20 pts
- A short **report** about anything you have learn in this project 5 pts
- **Part 2** correctness of **part2_p3.s** Extra Credit 1 pts
- **Part 2** efficiency of **part2_p3.s** Extra Credit 1 pts
- Improve the logic of Hao He's simulator. Please describe it in your **report**. Extra Credit based on your implementation

For Part 1, the expected number of cycles is shown on GitHub README. In addition, you need to ensure that the output of test cases is correct.

For Part 2, you should aim to reduce the cycle count by at least 30 cycles compared to the original version of **part2.s**, which runs in approximately 700 cycles.

Late Policy: For each day after the deadline, 10 points will be deducted from your final score up to 30 points, after which you will get 0 point.

12 Submission

The submission structure should be as follows:

```
StudentID-project3.zip (e.g. 121090000-project3.zip)
  |---src/
  |    |---... (same file structure as GitHub repo/src)
  |---part2_p2.s
  |---part2_p3.s
  |---report.pdf
  |---...  (any other necessary files or documents you want to show)
```