Final Project Report
ECE 473 RISC-V Processor

Martin Guarnieri
Isaac Violette

December 14,2022

Abstract

A 5-stage RISC-V pipleine processor was designed to run nine assembly programs. The instruction set architecture (ISA) was followed by the use of the RISC-V instruction set core instruction formats which helped implement the basic RV32I base integer instructions. Progress was made by completing one milestone at a time, together, which was then heavily debugged to make sure each instruction worked correctly. Once the processor was designed, eight of the nine programs ran successfully which was well over the required five programs.

Contents

List of Figures

List of Tables

1 Introduction

The following report describes the collaboration, project obstacles and design skills that were used for the successful completion of the 5 stage RISC-V pipeline processor. To begin the project, two milestones were given which laid out the order to complete the required instructions. Milestone one was completed in two weeks of time and then was diligently debugged to make sure that we would not have to return to previous instructions. Both the forwarding unit and hazard detector were implemented in these two weeks as well. Milestone two was completed within a week since the only major instruction types left were branches and jumps. Once the test programs were released, they were all tested and eight of the nine programs passed. In this process many obstacles occurred which challenged us tremendously but in the end helped us both takeaway very rewarding design skills of which will be used for many engineering projects in the future.

2 Collaboration

Our strategy was to work on the project together at all times. Therefore, we did not divide up the workload at all and instead collaborated on all steps of the process. This includes lab periods and multiple nights a week for the 5 weeks it took us to complete the project. In this way, we estimate 50 hours were spent together on the project, but this estimate is on the lower side. This method also had the benefit of not needing to share files back and forth, because the entire CPU was built on one laptop, and the other laptop was generally kept for RARS simulations.

TABLE I
Collaboration

Instruction	Martin	Isaac
ALU	✓	✓
Pipeline Registers	✓	✓
Forwarding Unit	✓	✓
Hazard Detection	✓	✓
PC	✓	✓
Control	✓	✓
Immediate Generator	✓	✓
BONUS: Static Branch Prediction	✓	✓

3 Project Obstacles

The majority of time spent on the project was time spent on debugging. Sometimes these bugs were small typos that resulted in hours that felt wasted. Such as miss-naming a bus line in Quartus to the pipeline register input instead of the pipeline register output, or having a typo inside a pipeline

register so a control signal was arriving one clock cycle too early. Obstacles for each specific type of instruction are listed in the subsequent subsections.

3.1 Immediate R-Type and I-Type

The SLT, SLTU, SLTI, and SLTIU gave the most trouble for these types of instructions. Since they only are calculated to be 1 or 0, it was easy to think they were working when they actually were not. This was a problem in the ALU code, which was corrected by looking at the example ALU on the ULTA-EMBEDDED.com code, which first compared the most significant bit to determine if both rs1 and rs2 were both signed, both unsigned, or one of each.

3.2 I-Type Loads and S-Type Stores

The main complication with the load and store type instructions arose because the instruction and data memory were word addressable and not byte addressable. This was combated by dividing the inputs to the memories by 4. For the instruction memory, this was done was bit-shifting the input address by 2. For the data memory, the target address and immediate number offset were divided by 4 in the ALU code.

3.3 B-Type and J-Type

For the branch instructions, once we realized this would essentially always give a data hazard, we decided to implement static branch prediction. This always predicted the branch would not be taken, so the following instructions would continue normally. If the branch was taken, the branch signal was input into the first 3 pipeline registers. If went high, this signal would set all the data inside to 0, essentially flushing out the previous instructions that would no longer be taken. For JALR instructions, we implemented an extra controller output which controlled a MUX before the PC adder, so that PC=rs1+imm instead of PC+=imm. For the ”JAL zero” statements in the test programs, we needed to hard-code the zero register to be zero in the register file. This was the only change between milestone 2 working and 8 test programs working.

3.4 U-Type

For the U-type instructions, we had one complication when both the immediate generator and the ALU were shifting up by 12 bits. Once this was realized, it was an easy fix by removing the bit shifting in the ALU.

3.5 Data Hazards and Forwarding

Once we realized that the first milestone had data hazards, we needed to implement the forwarding unit and hazard detection unit, using logic given in the textbook which is show in Appendix B. This proved to be less of an obstacle then we had initially imagined.

3.6 Hardware Issues

Occasionally when we would push the button too fast, the CPU would behave strangely. This never happened when the project clock was an automatic clock. We think this was due to a debouncing error which went unfixed. We also had a switch that controlled if the clock input was an automatic clock or a button, and flipping this switch while the CPU was running resulted in strange functions. It worked best when the switch was in the correct position before the board was programmed.

4 Design Skills and Philosophies

This project taught us the importance of team accountability. By always working on the project together, we were much more motivated then we would have been otherwise. We also had the benefit of two minds to think through problems and propose solutions, which prevented us from being stuck on any one problem for too long. We also had no trouble implementing separate modules together, because each of us was present for the creation of every module, and knew the required inputs and outputs. Speaking of modules, this project also showed the importance of modular design. By separating tasks into smaller components, it was much easier to find errors in the data paths. For example, our first debugging step when the overall CPU didn't behave correctly was to run a simulation on just the ALU, to make sure the correct calculation was being performed for the assignment statement.

This project also showed the importance of using all your resources to their fullest extent. We always had the textbook open to a CPU diagram as a reference. Using the given general purpose registers on the VGA module was extremely helpful in tracking control signals. We used both the Waveform simulations and the RARS CPU simulator to ensure correct design was instrumental through the entire debugging process. Another design skill we felt was extremely helpful was simply clock management. The PC and the pipeline registers each ran on the positive edge of the project clock (either a button for debugging or a fast clock for the final design) to keep the pipeline working continuously. Every other always block in other modules were run whenever their inputs changed. In this way, we bypassed worrying about the negative edge of the clock signal, and ensured data was always available to be picked up or dropped off by the pipeline.

5 Future Class Improvements

From our standpoint of the project, we believe the class was a great learning experience of how to take on a huge task and complete it in modules with another person. It made sure that we knew the data paths of a CPU and how every part of a processor comes together to run programs. We believe the following list would make the class work better for future years:

- Have the python script which turns assembly files into MIF files much sooner to test milestones.
- Make sure there is a working decoder going into the final project that displays each instruction for every PC increment.
- Make Lab 5 and 6 a partner lab.
- Make sure students understand the importance of the textbook earlier on in the class.

The python script which turns assembly to MIF files was given with the test programs which we didn't utilize at all. Since the test programs were already released there was no need at that point to check our own programs. The process up to that point was making MIF files by hand which took a lot of time which could have been saved otherwise.

Lab 6 asked us to create a working decoder which was very important to use for the CPU design but it also asked us to display the instruction on the VGA. We were able to get the decoder to work and it simulated fine on the Waveforms simulator but it never was able to display on the VGA. We were able to get what type of instruction it was but never the entire instruction. It would have been very beneficial to get it working fully as it would help out a lot while debugging to know what instruction it was on. Also, we believe that lab 5 and 6 should be partner labs since that work is the foundation for the project. This would also let the students see if they work well with their partner and if not possibly change before the final project starts.

Lastly, a very important element of the class was the textbook. It was required but we didn't start looking at it until the project was released. After we started looking through the pages, we realized that a lot of the class lecture slides matched the work in the textbook and we believe that would have helped us going through the preliminary work up to the project.

6 Conclusion

We feel that this project rewarded us for the time we put into it. By working on the project early, we were able to achieve our goals in a reasonable time. Eight out of nine test programs were passed, and the ninth was extremely close, as is detailed in the table below:

TABLE II
Test Program Completion

Test	Status	Program
1	✓	multiply
2	✓	sum_array
3	✓	square_array
4	✓	string_len
5	✓	isPrime
6	✓	palindrome
7	✓	bubble_sort
8	✓	fibanacci
9	x*	binary_search

*The Binary search program ran to completion, but the stack pointer and return address registers were altered once after the program should have entered a dead loop.

Appendices

Appendix A Helpful Textbook Schematics [1]

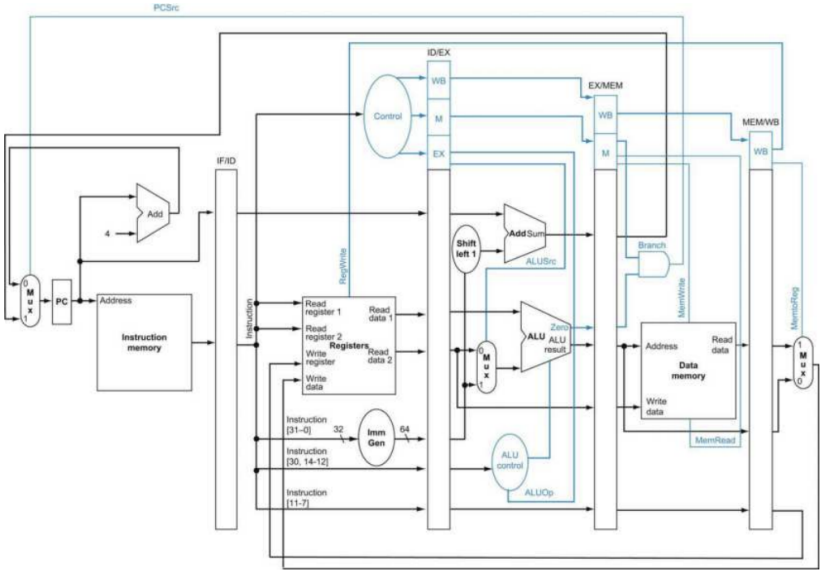

Fig. 1. Very useful overview of the datapath and control connections on Pg. 314.

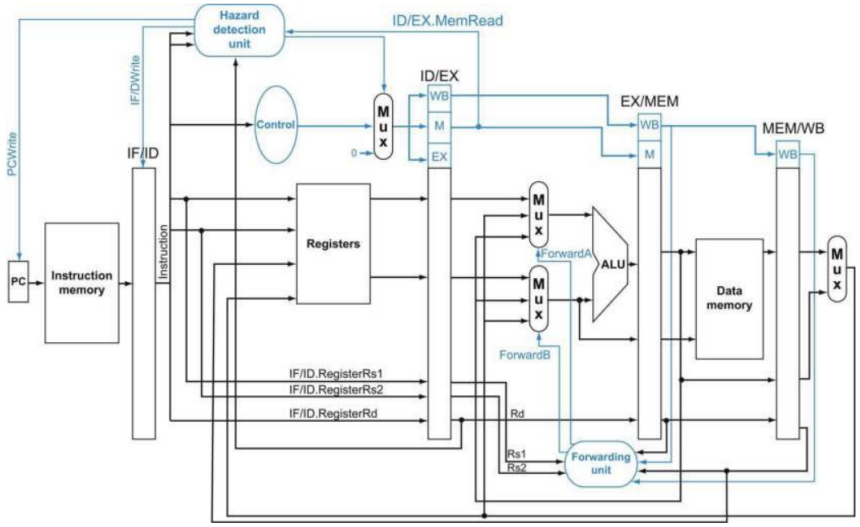

Fig. 2. Very useful overview connecting hazard detection and forwarding unit on Page 325.

Appendix B Helpful Textbook Logic [1]

EX *hazard* in Forwarding Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

MEM *hazard* in Forwarding Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
 and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
 and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Control for Hazard Detection Unit:

```
if (ID/EX.MemRead and
 ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
 (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
 stall the pipeline
```

References

[1] D. A. P. . J. L. Hennessy, “Computer Organization and Design RISC-V Edition: The Hardware Software Interface Second Edition,” 2021.