Computer Architecture Lab 3

Oklahoma State University

Pipelined RISC-V Implementation

Author
Garrett Page

Instructor
Rose Thompson

May 2, 2024

Contents

1 Introduction

In this lab, we designed a pipelined RISC-V microprocessor on an FPGA. The processor supports 37 basic instructions within the RISC-V architecture. The project ties together much of the information being taught in lecture. The project was all compiled into its own github repository with past projects to use a showcase for FPGA development. Please note the extra credit section at the end of this report.

1.1 RISC-V

At its core, RISC-V stands for Reduced Instruction Set Computing, Version 5. It's an open standard, offering a breath of fresh air in an industry often dominated by proprietary designs. RISC-V is built on a few fundamental principles that make it both powerful and adaptable.

First and foremost, RISC-V embraces simplicity. Its instruction set is lean and efficient, comprising a small number of basic operations that form the building blocks of computing. This simplicity not only makes RISC-V easy to understand but also allows for streamlined hardware implementation.

Another key feature of RISC-V is its modular structure. The architecture defines a base instruction set, which forms the foundation for all RISC-V implementations. This base set includes essential operations like arithmetic, logic, and memory access. Beyond the base set, RISC-V also supports optional extensions, allowing designers to tailor the architecture to specific application domains. This structure allowed us to create relatively simple methods to read different types of instructions from only a few instruction layouts, making the design and development of the simulator much easier than other instruction sets may have been.

2 Baseline Design

The baseline design for the project required 24 instructions to be implemented on a pipelined RISC-V architecture. Much of the processor was written for us, including a basic 5-stage pipeline architecture as well as several basic example instructions. The lab required us to implement the remaining instructions into the processor using the various resources provided to us in the class.

The difference between the pipelined processor and the single cycle arthictecure from lab 2 are the various pipeline stages that the processor has to take to execute a single instruction. The instructions take more than one cycle to complete, however multiple instructions can be in flight at a single time, making the overall performance for the machine much higher. The machine can complete more instructions over a shorter period of time.

Single-Cycle RISC-V Machine: In a single-cycle RISC-V machine, each instruction executes in a single clock cycle. The entire instruction fetch, decode, execution, memory access, and write-back stages occur within this single cycle. While this design is straightforward and predictable, it suffers from limitations. The longest instruction determines the cycle time, leading to inefficiency. Additionally, all hardware resources (such as the ALU and memory) must be available for every instruction, which restricts performance. Achieving high clock frequencies is challenging due to the long cycle time.

Pipelined RISC-V Machine: Pipelining breaks down each instruction into a series of shorter steps, with one step per clock cycle. Multiple instructions can overlap in time, allowing for better utilization of hardware resources. The pipeline stages include instruction fetch (F), decode (D), execute (E), memory access (M), and write-back (W). Pipelined machines achieve improved throughput by overlapping instruction execution. However, they introduce complexity. Hazards (such as data hazards and control hazards) can cause stalls or incorrect results. Handling dependencies between instructions becomes crucial. Branch instructions may also introduce pipeline bubbles (stalls) when their target address is not known early.

Trade-offs: Pipelining improves throughput by allowing multiple instructions to progress simultaneously. It utilizes hardware resources more efficiently. However, it requires careful management of dependencies and hazards. Single-cycle machines are simpler but sacrifice performance due to their fixed cycle time. Real-world processors often use pipelining or more advanced techniques to strike a balance between simplicity and performance

2.1 Changes

A few changes were made to the design outlined in the lab manual. Most of these changes were the same as the changes made in lab 2, however the main difference is that adding

new instructions required saving and transmitting states across multiple pipeline stages. Each of the following additions to the pipeline required a new value to be preserved across multiple stages depending on their position within the pipeline.

- A MSBExtend was added to the ALU to account for the behavioral difference in arithmetic and binary instructions msb extending.
- A ResultMask was added to the data memory loading and saving architecture to allow data to be saved in 2-byte and single byte segments. This allowed the implementation of sh, sb, lh, lb, lhu, and lbu with proper alignment and bitmasking.
- A memdec (memory decoder) module was designed to specify the alignment and write mask requirements for the different save and load functions.
- ResultSrc was extended for PCPlusImm and Imm passthrough values
- ImmSrc was extended to handle U-type instructions.
- PCSrc was updated to handle different types of branching operations.

3 Design Details

The final design closely resembled that of the template code given, however a few changes were made that should be noted.

3.1 MSBExtend

The ALU controller was extended by adding a MSBExtend logic input. This allowed the decoder to specify how an instruction would handle bit extensions. A value of 1 would require the processor to preserve the most significant bit to perform an arithmetic shift, whereas a value of 0 would zero-extend the given input. This allowed the arithmetic instructions to be implemeneted alongside their logical counterparts.

3.2 ResultMask and Alignment

The ResultMask was added to ensure that different sizes of data could be properly read from and stored to without overwriting nearby data. If the machine was instructed to write a single byte to a specific address, the whole 32-bit register cannot be used. Only the bottom byte of the register should be used to write the bit. Furthermore, an alignment specifier was used to ensure proper alignment between data values of different sizes. Half words and full words must be aligned to their 2-byte and 4-byte positions respectively within memory.

3.3 Memory Decoder

To aid in the process of data storage and alignment, a memory decoder was designed to create a MemControl value from opcodes and funct3 values of a given instruction. This MemControl specified both how the data was to be aligned and masked, as well as how the highest bit in the mask should be extended if a smaller data type was to be placed in a larger memory cell. The memory decoder was implemented on the memory stage of the pipeline.

3.4 PCPlusImm and Imm Passthrough

To handle J-type and U-type instructions, PCPlusImm and immediate passthrough result sources were needed. These were added by extending the ResultSrc width by one bit and adding two more options. The mux3 that defined the ResultSrc was expanded into a

mux5 to handle the new options. Both of these were captured on the data and execute stage, and were transmitted to the write stage for use with the resultmux.

3.5 ImmSrc

To handle U-type instructions, ImmSrc was expanded by one bit to allow for the U-type immediate layout specifier. This was the fifth and final immediate layout that was yet to be implemented.

3.6 PCSrc

PCSrc was also updated to handle different type of branching instructions. Branching instructions use one of two ALU operations: subtract, or set less than. The ALU outputs either a zero or non-zero value for each branch instruction, and the following figure contains the logic for branching given the funct3 of the branch instruction and the output of the ALU. This was handled on the execute stage of the pipeline.

Figure 3.1: PCSrc change

```
//  assign PCSrcE = (BranchE & ZeroE) | JumpE;
assign PCSrcE = BranchE & (funct3E[2] ? (~ZeroE ^ funct3E[0]) : (ZeroE ^ funct3E[0])) | JumpE;
```

3.7 Python Instruction Converter (From Lab 2)

The Python-based instruction converter contains a formatted list of all accepted instructions with exact formats and layouts. The compiler then defines logic to read each of the instructions in their respective assembly syntax and converts them to opcodes using the format table as shown below. All resulting opcodes are then written to *tests/test.opcodes*. The result can then be used directly as a memfile for testing the RISC-V machine.

4 Testing Strategy

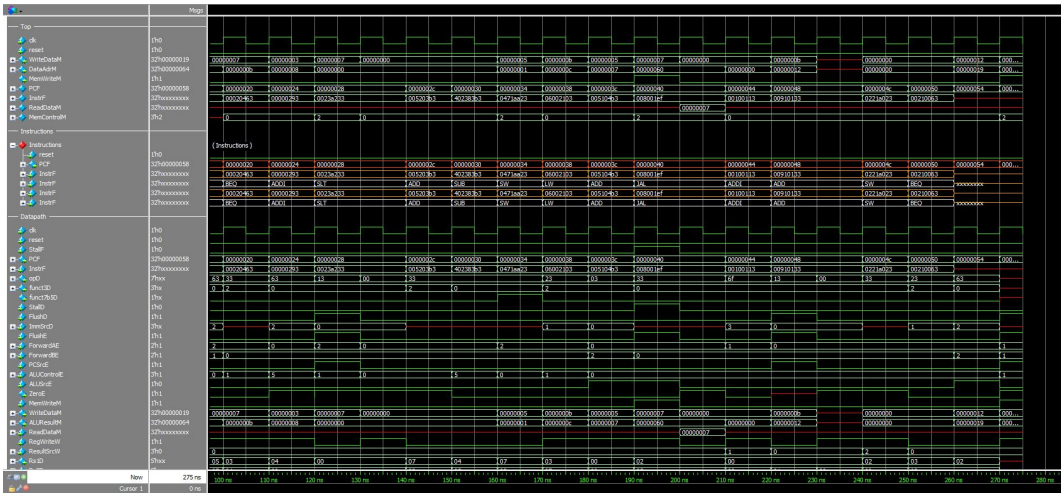
The testing strategy for the RISC-V machine was certainly the most notable and difficult part of the project. While some tests were provided, they required a lot of manual work to ensure that they were operating. When a problem goes wrong within the machine that is not at the instruction level, such as a problem reading and decoding an instruction, there is much room for error when handling large changes to the code base. Therefore, it was much easier to commit small changes to individual instructions one at a time to get the system working.

4.1 Custom Tests

Testing the machine during development and before all instructions were implemented was more tricky. I opted to use my simple Python converter from lab 2 for instructions that ensured complete and correct bytecode, stored as hexidecimal 32-bit numbers. These bytecodes could then be read by the machine directly as a memfile. The result of running this memfile was compared to the result of the simulator from lab 1.

Instructions were added one at a time in groups depending on their type. R-type instructions were implemented and tested first because they were the easiest to start with considering the given code template. I-type instructions were implemented and tested alongside R-type instructions because of their similar nature. Then, the load and store functionality was extended to handle the different load and store sizes for halves and bytes for load/store instructions. Next, branch instructions were implemented by updating the PCSrc logic, then J-type and U-type instructions were implemented last due to their complexity.

Figure 4.1: Local waveform



4.2 Moving to Vivado

Once all tests passed on the local machine and systemverilog simulation was successful, the top and testbench modules were disabled. The remaining code was moved to Vivado to connect into the FPGA and DDR3 memory. The MemStrobe and PReady signals were added to stall the device whenever a realtime memory transfer was taking place. This was unnecessary during the local testing phase because the load and store instructions always responded in a single clock cycle. The MemStrobe and PReady implementations were very similar to lab 2. The pipeline must stall during memory transfers.

5 Evaluation

5.1 Test Results

The tests on Vivado were successful in demonstrating the capabilities of the pipelined RISC-V simulator. The code synthesized and the implementation ran with no issues. The FPGA device was able to be programmed to behave as a pipelined RISC-V microprocessor.

5.2 Performance Analysis

There are many points to note on area and energy of the machine. There was minimal wasting of bits within the decoder, however there were many values that went unused within the different submodules. For example, the immediate source used three bits and had a maximum possible eight values, howver it only used five of these values and the others were wasted. It may be possible to combine some of these signals in a more comprehensive way to use space more efficiently.

There were also multiple levels of case/switch statements. These levels can lead to a complicated array of switches within the device designed to choose between every possible outcome at once. This could be optimized by reducing the number of cases and replacing them with more concise bit usage.

5.3 Performance Difference from Single Cycle Architecture

In computer architecture, the performance difference between single-cycle and pipelined architectures is significant, reflecting varying approaches to executing instructions. In a single-cycle architecture, each instruction takes exactly one clock cycle to complete, ensuring simplicity and predictability. However, this simplicity comes at a cost of efficiency, as some instructions might require longer to execute than others, leading to idle time between cycles.

On the other hand, pipelined architectures break down the instruction execution process into several stages, with each stage handling a specific task. This allows multiple instructions to be processed simultaneously, overlapping different stages of execution. As a result, pipelined architectures often achieve higher throughput and better resource utilization compared to single-cycle architectures. However, pipelining introduces additional complexity, such as the need for hazard detection and resolution mechanisms to handle dependencies between instructions and ensure correct execution order.

While single-cycle architectures offer simplicity and predictable execution times, pipelined architectures provide higher throughput and better resource utilization at

the expense of increased complexity and potential hazards. The choice between the two depends on the specific requirements of the application and the trade-offs that need to be made between performance, complexity, and predictability.

5.4 Summary

This project was a challenging continuation of the world of RISC-V. It showed us how computers work at the lowest level and the instructions and operations of the architecture have implications in many other aspects of computing. Modern computers running much more complex instruction sets still have versions of the extremely basic instructions that we simulated in this lab.

5.5 Extra Credit

My extra credit attempt for this lab was to create a Github repository for this lab as well as lab 1 and 2 for future use. This repository will allow me to use the labs as resume projects or as a reference for other projects. I hope this can count as extra credit because I had a difficult time coming up with other ideas that wouldn't be incredibly complex to implement or haven't been done in lab 2.

6 References

[1] GitHub Repository

https://github.com/stineje/ecen4243S24/