Computer Architecture Lab 2

Oklahoma State University

Single Cycle RISC-V Implementation

Author
Garrett Page

Instructor
Rose Thompson

May 2, 2024

Contents

1 Introduction

In this lab, we designed a single cycle 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 also includes a Python converter for testing out different sets of instructions. 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 single cycle architecture. Much of the processor was written for us, including a basic single cycle flip flop 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 processor works at a high level by combine a set of modules that define each part of the processor's schematic. The most important part is the flip-flop that adds four to the PC value to go to the next instruction. Each cycle, the instruction is sent to the controller, decoded, then sent through the datapath. The datapath splits the instruction into multiple parts depending on what the controller specifies, then the results of the instruction are all combined and executed within the same cycle.

2.1 Changes

A few changes were made to the design outlined in the lab manual.

- 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.

2.2 Simplified Compiler

The custom Python RISC-V converter will convert a RISC-V assembly syntax into a proper memfile that can be used for testing. Only the 38 instructions supported by the simulator are compiled properly.

Figure 2.1: Example of *tests/test.riscv*

```
addi x0, x0, 10          # Add 10 to register 0
addi x1, x1, 64          # Add 64 to register 1
sw x0, 8(x1)             # Use register 1 and 0 to store word
ecall x0, x0, 0          # Halt the simulation
```

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.

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.

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.

Figure 3.1: PCSrc change

```
//assign PCSrc = Branch & (Zero ^ funct3[0]) | Jump;
assign PCSrc = Branch & (funct3[2] ? (~Zero ^ funct3[0]) : (Zero ^ funct3[0])) | Jump;
```

3.7 Python Instruction Converter

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.

Figure 3.2: Custom Python bytecode format table

```
# R-type opcodes
'add':      '0000000 rs2 rs1 000 rd 0110011',
'sub':      '0100000 rs2 rs1 000 rd 0110011',
'xor':      '0000000 rs2 rs1 100 rd 0110011',
'or':       '0000000 rs2 rs1 110 rd 0110011',
'and':      '0000000 rs2 rs1 111 rd 0110011',
'sll':      '0000000 rs2 rs1 001 rd 0110011',
'srl':      '0000000 rs2 rs1 101 rd 0110011',
'sra':      '0100000 rs2 rs1 101 rd 0110011',
'slt':      '0000000 rs2 rs1 010 rd 0110011',
'sltu':     '0000000 rs2 rs1 011 rd 0110011',
# I-type opcodes (0010011)
'addi':     'imm[11:0] rs1 000 rd 0010011',
'xori':     'imm[11:0] rs1 100 rd 0010011',
'ori':      'imm[11:0] rs1 110 rd 0010011',
'andi':     'imm[11:0] rs1 111 rd 0010011',
'slli':     '0000000 imm[4:0] rs1 001 rd 0010011',
'slri':     '0000000 imm[4:0] rs1 101 rd 0010011',
'srai':     '0100000 imm[4:0] rs1 101 rd 0010011',
'slti':     'imm[11:0] rs1 010 rd 0010011',
'sltiu':    'imm[11:0] rs1 011 rd 0010011',
'jalr':    'imm[11:0] rs1 000 rd 1100111',
'ecall':    '000000000000 rs1 000 rd 1110011',
'ebreak':    '000000000001 rs1 000 rd 1110011',
# I-type opcodes (0000011)
'lb':       'imm[11:0] rs1 000 rd 0000011',
'lh':       'imm[11:0] rs1 001 rd 0000011',
'lw':       'imm[11:0] rs1 010 rd 0000011',
'lbu':      'imm[11:0] rs1 100 rd 0000011',
'lhu':      'imm[11:0] rs1 101 rd 0000011',
# S-type opcodes
'sb':       'imm[11:5] rs2 rs1 000 imm[4:0] 0100011',
'sh':       'imm[11:5] rs2 rs1 001 imm[4:0] 0100011',
'sw':       'imm[11:5] rs2 rs1 010 imm[4:0] 0100011',
# B-type opcodes
'beq':      'imm[12|10:5] rs2 rs1 000 imm[4:1|11] 1100011',
'bne':      'imm[12|10:5] rs2 rs1 000 imm[4:1|11] 1100011',
'blt':      'imm[12|10:5] rs2 rs1 000 imm[4:1|11] 1100011',
'bge':      'imm[12|10:5] rs2 rs1 000 imm[4:1|11] 1100011',
'bltu':     'imm[12|10:5] rs2 rs1 000 imm[4:1|11] 1100011',
'bgeu':     'imm[12|10:5] rs2 rs1 000 imm[4:1|11] 1100011',
# U-type opcodes
'lui':      'imm[31:12] rd 0110111',
'auipc':    'imm[31:12] rd 0010111',
# J-type opcodes
'jal':      'imm[20|10:1|11|19:12] rd 1101111',
```

4 Testing Strategy

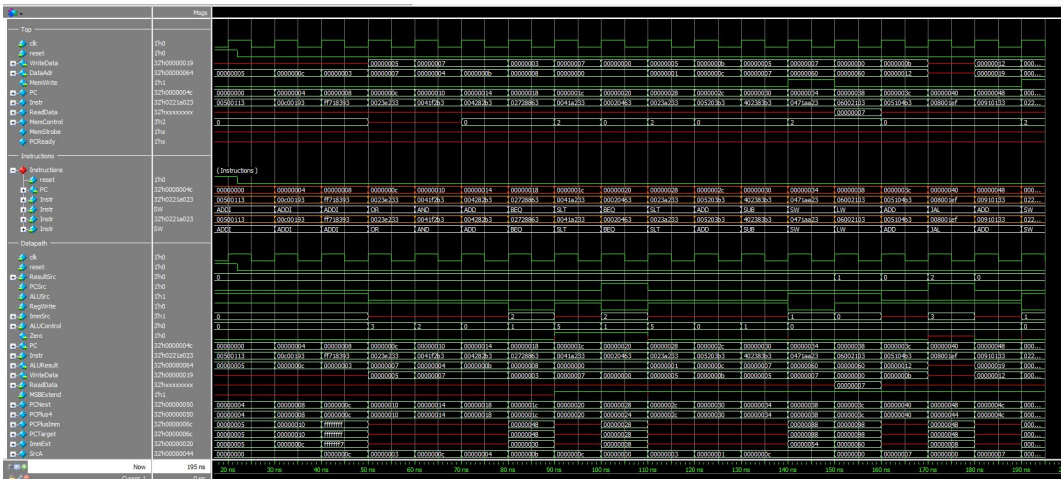
The testing strategy for the RISC-V machine was certainly the most notable and difficult part of the project. While some tests were provided, they required a lot of manual work to ensure that they were operating. When a problem goes wrong within the machine that is not at the instruction level, such 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 create a simple Python converter 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.

5 Evaluation

5.1 Test Results

The tests on Vivado were successful in demonstrating the capabilities of the single-cycle 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 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 Summary

This project was a challenging introduction into 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.

5.4 Extra Credit

I attempted some extra credit for this lab which can be found within the tests folder. The compile.py script is designed to convert RISC-V assembly code directly into memfiles that can be used for testing custom code quickly and efficiently. This greatly improved the testing strategy for the RISC-V machine.

6 References

[1] GitHub Repository

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