Computer Architecture Lab 1

Oklahoma State University

RISC-V Simulator Implementation

Author
Garrett Page

Instructor
Rose Thompson

February 22, 2024

Contents

1 Introduction

In this lab, we designed a RISC-V instruction set simulator. The simulator supports 38 basic instructions for the RISC-V architecture. The project ties together much of the information being taught in lecture. Also included is a custom instruction compiler written in Python.

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 36 instructions to be implemented on a RISC-V instruction set simulator. Much of the simulator was written for us, such as the basic instruction processing, PC, simulated computer state, and a few basic examples. The lab required us to implement the remaining instructions into the simulator.

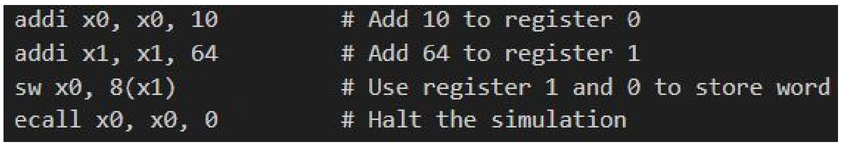
2.1 Changes

A few changes were made to the design outlined in the lab manual. First, a simple RISC-V compiler was constructed using Python called *tests/compile.py*. This compiler takes a simplified form of RISC-V assembly defined in *tests/test.riscv* and compiles it into valid RISC-V assmebly that can be read by the simulator. The resulting bytecode is placed into a file called *tests/test.opcodes*. These tests were used to more easily debug individual operations quickly while developing the simulator, as no manual conversion was required, reducing the risk of human error.

2.2 Simplified Compiler

The compiler compiles a simplified custom syntax for assembly. The syntax closely follows that of regular RISC-V assembly, however no data block is included. 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 Files sim.c and isa.h

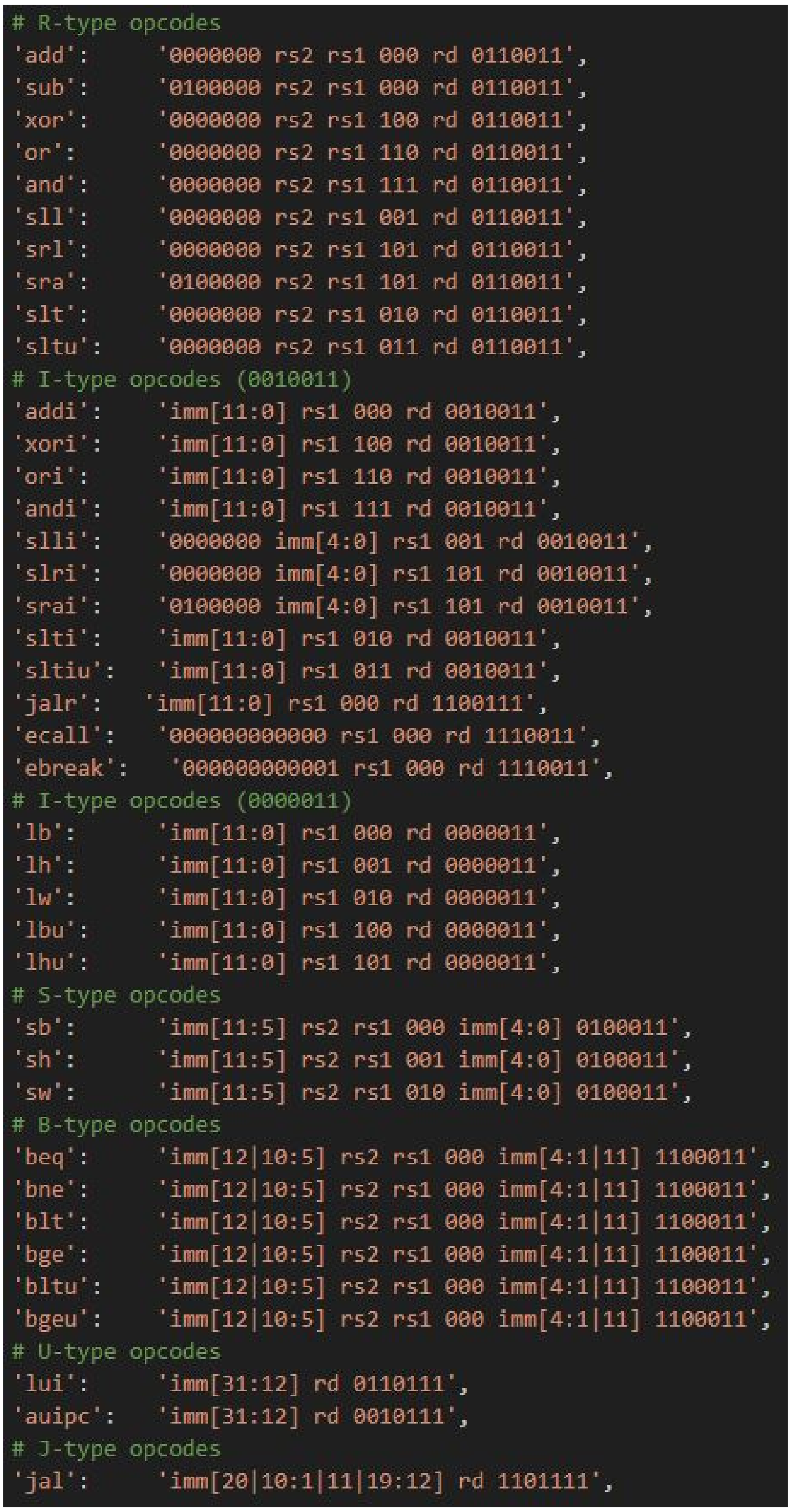
Within sim.c, the original design never considered Funct7 and passed Funct3 to isa.h. The new design properly considers Funct7, while Funct3 is handled directly within the sim.c file, differentiating between different instructions. Each individual instruction, as differentiated by Funct3 and Funct7 is handled separately in isa.h.

Within isa.h, each instruction is handled as a separate function with a matching name. Depending on the category of instruction being handled, the functions take differing inputs. There are 38 total instruction modeled in isa.h, each called separately from sim.c.

3.2 Python Compiler

The Python-based compiler 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*.

Figure 3.1: 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 simulator 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 simulator that is not at the instruction level, such as a problem reading and decoding an instruction, there is much room for human error both in the simulator code itself as well as the opcode and bytecode conversion process when manually testing instructions.

4.1 Custom Tests

Testing the simulator during development and before all instructions were implemented was more tricky. I opted to create a simple Python compiler for instructions that ensured complete and correct bytecode, stored as hexidecimal 32-bit numbers. These bytecodes could then be read by the simulator, which printed the immediate, funct3, funct7, and opcode values to the output when reading an instruction. These values were then compared to the original simplified assembly code written for the test compiler. Only once everything matched and the instruction was read and formatted properly were the isa implementations written. This system also made it easier to test indidvidual instructions quickly.

4.2 Provided Tests

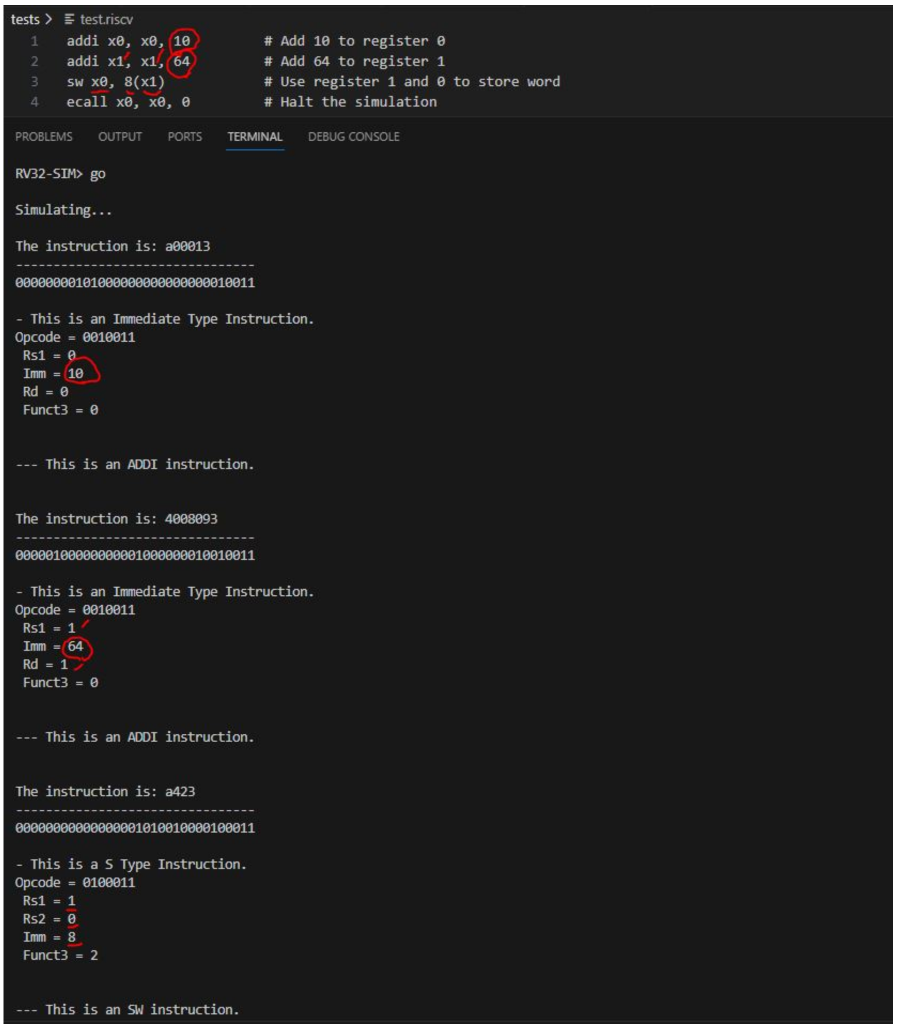
Once all testing had been performed on the layout of instructions and all instruction implementations finished, the provided tests were run using the *go* or *r 1* commands. Once the desired instructions for each test finished, the *rd* command dumped the register values to the output to ensure that each instruction was being executed properly.

5 Evaluation

5.1 Test Results

The basic tests performed well on the simulator. An example test is shown below, and the order in which the instructions were written, the opcodes, as well as each part of the bytecode matches from the original assembly syntax. Each instruction executes properly from the provided tests as well.

Figure 5.1: Sample Results from Tests



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
```

```
RV32-SIM> go

Simulating...

The instruction is: a00013
--------------------------------
00000000101000000000000000010011

- This is an Immediate Type Instruction.
Opcode = 0010011
 Rs1 = 0
 Imm = 10
 Rd = 0
 Funct3 = 0


--- This is an ADDI instruction.


The instruction is: 4008093
--------------------------------
00000100000000001000000010010011

- This is an Immediate Type Instruction.
Opcode = 0010011
 Rs1 = 1
 Imm = 64
 Rd = 1
 Funct3 = 0


--- This is an ADDI instruction.


The instruction is: a423
--------------------------------
00000000000000001010010000100011

- This is a S Type Instruction.
Opcode = 0100011
 Rs1 = 1
 Rs2 = 0
 Imm = 8
 Funct3 = 2


--- This is an SW instruction.
```

5.2 Performance Analysis

There are many points of performance issues within the simulator. The most notable issues are the concerning number of if branches that each must run to determine the type of an instruction. This was the fastest way to design and develop the simulator, however with the decreased development overhead came an increase in the performance demands on the machine.

A solution to this problem would be switch statements that operate independently on the opcode first, then funct3, then funct7 to determine instruction type and behavior. Currently, there is one if statement for every instruction

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.

6 References

[1] GitHub Repository

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