

Lab 0

Revisiting Digital Logic and SystemVerilog Simulation

Thomas Kidd CWID: 20275230

Zach Wilson CWID: 20267002

Electrical and Computer Engineering

Oklahoma State University

Stillwater, Oklahoma - United states of America

thomas.kidd@okstate.edu & stealth.wilson@okstate.edu

1 Section 1: Introduction

RISK-V is a compelling, relatively new, open-source, Instruction Set Architecture (ISA) that is quickly taking over computer chips across the globe. This lab is important to us in order to keep up with the trends of modern architectures. Not innovating and keeping step with the norm is the biggest set back in any tech related industry. Further, learning the basics of this ISA. not only propels us into difference career opportunities, but also sets us up for success in future labs and in class. The in-class material covers the fundamentals of each type of instruction, their philosophy, structure, and importance, while the lab helps us put these paradigms into practice and appreciate the innovators clever ideas. In future labs we will be working with the RISK-V ISA on a hardware level, and having a software simulator to check our wanted outputs before any hardware implementation is very helpful. Throughout the lab-time we were able to complete the baseline design of the simulator through verifying each kind of instruction type. This was done through first our own small test files that we created to quickly check the validity of the instructions. Afterwards we used the longer, more detailed and rigorous test files given by the instructor. Since this was an entirely software related lab, we were unable to collect any information on the number of executed clock cycles. We made a few changes to the simulator in modifying how some of the sign extension works to better accommodate our preference on how we interpret the the counting of bits. All in all, its a user preference. Throughout the lab experiment we were able to implement all the RISK-V RV32I instructions that we were given. Through creating our own testing files to quickly check the legitimacy of our work, and using the given test files we believe we have successfully completed the given assignment.

2 Section 2: Baseline Design

In the baseline design we were tasked with implementing the RV32I instruction set of the RISK-V ISA. We were given two main files that needed editing. One contained all the methods for understanding what the instructions are by parsing the given binary code, while the other was a library of our implementation of the RV32I instructions. To clearly explain the workings of the simulation please see the following bullet points

- Read the opcode of the incoming instruction.
- Based on the opcode call the correct process function.
- Decode the rest of the instruction into funct3, 7, rd, etc...
- Detect the instruction wanted and call its function from the library.
- Increase the PC counter to have the process restarted for the next instruction.

Simply the bulk of the work as seen is done through recognizing the correct function needed to be called. The design is very modular as the way of reading every function code, r1 or r2, rd, or even the immediate is quite similar. In all the simulator is composed of a handful of for loops to help decode the information. But this simplicity should really be credited to the architects themselves, as without them the implementation of the simulation would be quite different. The one thing that the simulator really helps with is the easy of understanding what the program will do next. This is the beauty of C code or any high level programming language for that manner, and it really helped drive the point home.

3 Section 3: Detailed Design

As mentioned in the previous sections bullet points there is a certain order that the simulator follows to ensure that the simulation feels real. While the bulk of the work comes from implementing how to recognize the different instructions and actually executing those instructions, many basic functions already came pre-coded on the simulator. This mainly shines in the memory read and write functions of the simulator. Many functions deal with writing and reading from memory, as RISK-V is a load/store computer architecture. For this, being able to simulate the memory of a computer is a key component in creating a successful model. Luckily this component was already given to us. But integrating this functionality was important and daunting at first into our simulated RV32I ISA. Below you will see a flow chart that better describes the operation of the simulator. We will dive deeper into each section.

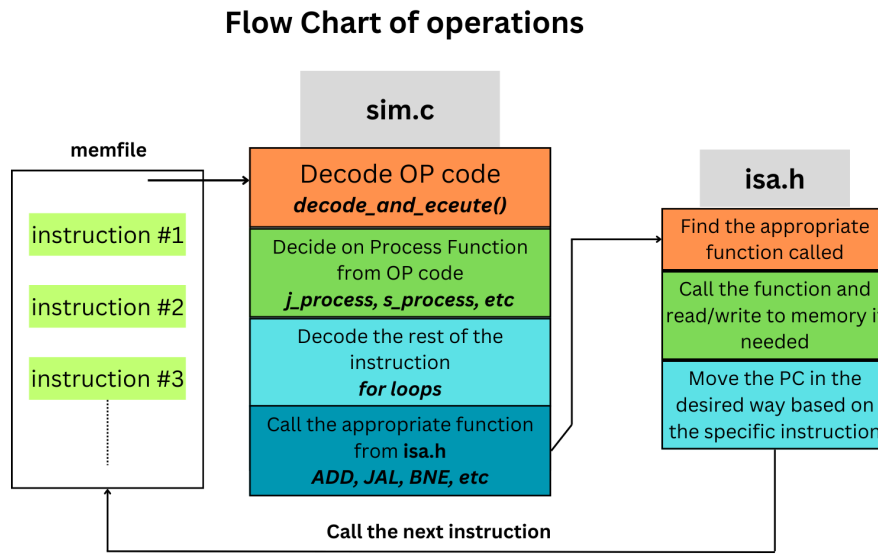


Figure 1: Flow chart of the order of operations in the simulator

3.1 sim.c - the brains of the simulator

When taking a closer look at how sim.c works, we quickly notice a growing pattern. Read the OP-code using the *decode_and_execute* function, call the process function associated with each OP-code *s_process()*, *b_process()*, *etc*, decode the rest of the functions to get all the important information out using *for loops*, and finally call the wanted function from the ISA.h and pass all the deciphered information along with it (rd, imm, r1, etc) *ADD, BNE, JAL, ADDI, LW, etc*. This process was essential to get right, as without decoding the instruction bit by bit, we would end up with the wrong destination register value, or the incorrect immediate given by the instruction. For many of the instructions, reading the values was very straight forward and easy to do with a for loop. The *R-type*, *I-type*, *S-type* and *U-type* instructions all have their values in order. Reading

the rd register, for example, was repetitive across all of the instructions, and so using a simple for loop to get this value was never a question. The cavities mostly came when wanting to decode the immediate values of the *J-type* and *B-type* instructions. The order of the immediate are scattered and presented it to be difficult to read. Please see the figure below for a better explanation.

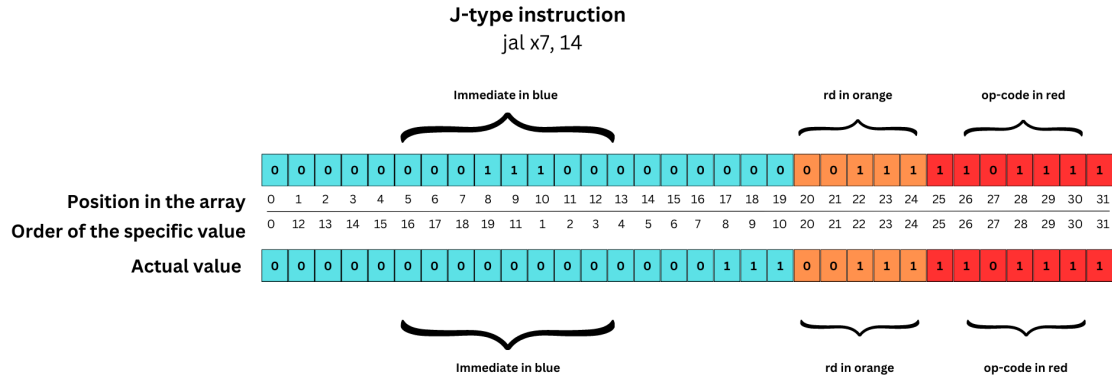


Figure 2: J-type instruction. The order of the immediate is the key highlight in this figure. It is also important to notice that if we directly convert the immediate value as seen in the image, we would result with 7. But the correct value is 14. This is because there is a shift of 1 bit in the immediate of a J type instruction and this was not illustrated in the figure. Therefore if we shift all the bits by one to the left in the immediate section, we can see that $1110_{2c} = 14_{10}$

The process of decoding the specific values was the main difficulty when working with the `sim.c` file. While other problems popped up, such as the op-code not being recognized in the `decode_and_execute` function, after adding them, our problems went away.

3.2 isa.h the library for the RV32I ISA

3.2.1 Overview

This part of the simulator was difficult to implement for some instructions, while easier for others. Understanding how to load and store into the simulated memory was a challenge. The `isa.h` library came with 2 essential functions that almost all of the other functions, what we can call instruction functions like *ADD*, *BNE*, *JAL*, *ADDI*, *LW*, *etc*, used. These were the *SIGNEXT* and *ZEROEXT*. Both of these functions are vital to ensure that the execution of the instruction goes as desired. The *SIGNEXT* was given to us, but we made a change to the original function, as we found it difficult to implement in our code.

Given code:

```
#define SIGNEXT(v, sb) ((v) | ( (0 < ( (v) & (1 << (sb) ) ) ) ? ~( (1 << (sb) )-1 ) : 0))
```

Modified code by doing $sb-1$:

```
#define SIGNEXT(v, sb) ((v) | ( (0 < ( (v) & (1 << (sb-1) ) ) ) ? ~( (1 << (sb-1) )-1 ) : 0))
```

This subtraction of $sb - 1$ proved to be vital. What we noticed is that the ADDI function given to us was not working properly with the original **SIGNEXT** function. This was because if $sb = 12$ it will move the 12 index location which is the 13Th bit. If $sb = 12$ and we do $sb - 1 = 11$ then we get the 11Th index location of the array and therefore we signextended the 12Th bit. Which is what is desired for the ADDI function. Further we created a **ZEROEXTEND** function as well to accommodate the unsigned numbers as well as any operations that needed to use the **ZEROEXTEND** function. In the making of the *isa.h* library we categorized each type of instruction, and used the Greencard provided in the GitHub to better understand what the instructions want us to do. Some of the difficulties arose when needing to implement either store or load functions such as **SW**, **LW**, **SB**, **LB**. These instructions posed a challenge due to having to access the memory. Running through the test files instructions one by one allowed us to understand what the test-file wanted to do, and helped us debug where our code is making a mistake. Overall many instructions were very similar to each other and the repetitive nature and using a higher level programming language made it easier to understand what was going on.

R-type Since all arithmetic operations are handled in the registers all of the R-type instructions used a very similar method of taking the input in and giving the output to the specified register. It all came down to simple math. Taking the input and giving an output.

S-type There was a similar motion when it came to the S-type instructions as well. The only difference between the three different instructions **SB**, **SH**, **SW** was which but are we going to signextend. In the case of **SB** we would signextend the 8Th bit, the 16Th bit for **SH** and no signextension for **SW** as its 32 bit. These three functions also all had similarities in that $rs1 + Imm$ (The Imm serves as an offset. This is especially useful when storing data in an array) is always the memory address we want to load the data into.

B-type The B-type instructions also were quite repetitive. Based on a condition, there is a branch operation made to a specific location in the program counter (PC). An essential task of the computer and PC, the branch instructions are fairly straight forward.

I-type I-type instructions use the immediate value to carry out instructions from arithmetic to load instructions. Instead of adding two numbers together, we can add a number with a constant stored in the immediate. We can see that these arithmetic instructions are nearly identical to the R-type ones. Further for load instructions, the immediate is important as it allows us to provide the offset. This can be important similarly to the S-type instructions where the immediate can provide an offset in case of an array.

4 Section 4: Testing Strategy

4.1 Our own tests

When it comes to testing our simulator, we implemented both the given test files, as well as our own test-files for quick verification. We noticed that some of the test files rely on many different instructions to be completed. In the beginning, when we had only a few instructions given, testing with the original test files resulted in constant errors. Instead, we decided to create our own test-files, containing three to five instructions. With these files we were able to test most of the arithmetic instructions, as well as many of the I-type and B-type instructions. Further, writing these smaller

test files, really helped us understand how RISK-V works and what it takes for a computer to run correctly.

4.2 Given tests

Once we felt that the instructions we wrote in *isa.h* were working, we went ahead and tested the simulator with the given test files. Each of these test files had several different tests within them, to measure the resiliency of the simulator. We went through several of these tests and checked the registers and the memory of the simulator if needed for conformation of the results. This helped us further debug our simulator with testing the ultimate conditions of each instruction.

4.3 Conclusion of tests

Starting with our own test files, we feel confident in each instruction works correctly within our testing domain. Furthermore all the instructions passed several tests from each given test-file. After going through the given test-files and running all of them through completion, we can confidentially say that all of our instructions work as designed.

5 Section 5: Evaluation

In the evaluation stage of our testing we can show that each instruction works by using the given test-files, as well as our own files we created, to determine the state of each register and its value after running the simulation or specific instruction. With the simulation file set up to run and show the information of the registers, as well as the PC counter, we insured each instruction was called properly by the opcode, implemented the intended instructions and input intended data to the correct register.

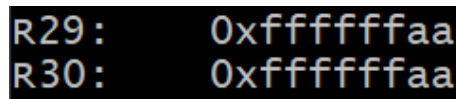
A screenshot showing two lines of text on a dark background. The first line is 'R29: 0xffffffffaa' and the second line is 'R30: 0xffffffffaa'. The text is in a light blue or cyan color.

Figure 3: Screenshot of the register R29 and R30. Our program compares these two registers and if they are equal, it moves onto the next test. As seen here, they are equal meaning our program will then move onto the next test.

Furthermore if the program halts and the 10Th register was filled with 10, it means that our test was successful. We found that this was not working because the 0Th register was being over written in some of the test files. This was not supposed to happen, as the 0Th bit is always supposed to be zero. We believe this was a mistake, or we were supposed to implement that the 0Th register always stays zero.

When it comes to qualitative analysis of our simulator, we cannot say one piece of code ran faster than another as these measurements are hard to achieve. Instead we can say that some pieces of instructions look more understandable and shorten the code base used. All of our instructions try to use the least amount of lines while maintaining clarity for the user to understand what each instruction is doing. Further in the *sim.c* file, we tried using as many for loops as possible to shorten the code base.