ECEC 355 – Computer Architecture

Project One – Single-cycle RISC-V Simulation

Instructor: Dr. Anup Das

Distributed, Intelligent, and Scalable COmputing (DISCO) Lab

ECE Department

Drexel University

June 24, 2019

1. Objective and Requirement

1.1 Objective

This project is intended to be a comprehensive introduction to single-cycle RISC-V simulation and RISC-V assembly programming. Please submit your work by July 26th, 11:59 pm, via Bblearn. You may work on this project in teams of up to two people.

1.2 Required Reading

Chapter 2. Instructions: Language of the Computer, Sections 2.1 – 2.10, Sections 2.12 – 2.14; Chapter 4. The Processor, Sections 4.1 – 4.4.

2. System and Software

Software: A cycle-level RISC-V simulator designed by Drexel DISCO Lab

Source codes: https://github.com/Shihao-Song/DREXEL-DISCO-RISC-V-Simulator-C-Impl

Instruction: please clone or download this repository to a Linux machine (any Linux flavor with GCC installed will work).

3. Design your simulator step by step

3.1 Simulator Overview

Your final simulator should be able to simulate the behavior of a single-cycle RISC-V CPU shown in Figure 1. We will achieve this goal through the following steps.

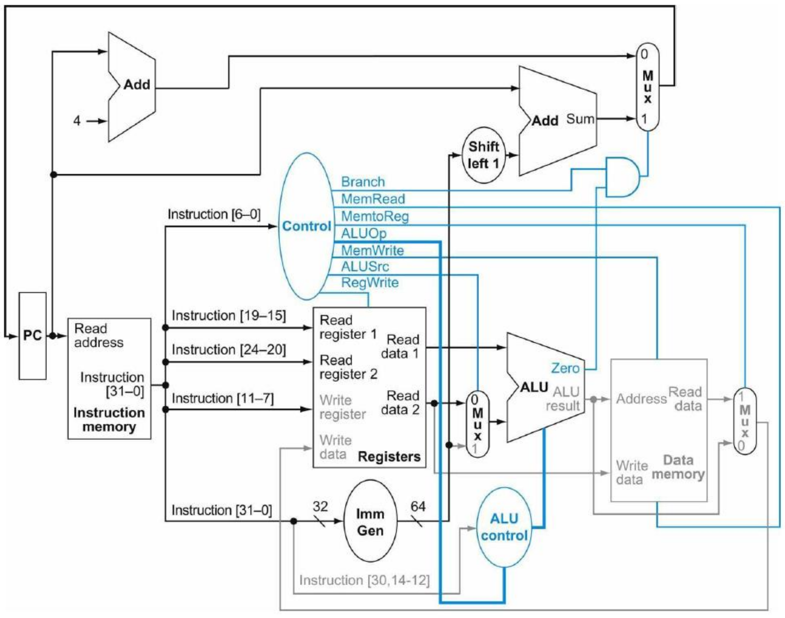


Figure 1 A Single-cycle RISC-V CPU

3.2 First step, what a CPU understands?

As we learned in the lecture, a CPU can only understand instruction in binary format or so-called machine language. For example, if you code something like "x = 0" (assign a variable x with the value of 0), the compiler will first translate it into assembly language, in RISC-V, it can be "add x5, x0, x0" (compiler puts variable x into register x5, then how to initialize x5 to 0? Simply by doing x5 = x0 + x0 where x0 is a special register that has value of 0). Then, another program called assembler translates this assembly instruction into machine language, which is simply a string of 1s and 0s that a CPU can understand, in this case, the translated machine instruction is "00000000000000000000001010110011". (Please read through chapter 2.5 for more details on how to translate an assembly instruction into machine language.)

To execute this machine instruction, a program called loader loads it into instruction memory (can you find it in Figure 1?). Finally, the CPU fetches it from the instruction memory, decodes it, and completes execution.

So, what is your first task? Complete "Parser.c". "Parser.c" takes every assembly instruction of the trace file, translates it into binary format, and loads it to instruction memory (We have already given an example of how to translate R-type instructions into machine codes).

Requirement:

Your parser should be able to handle the following operations

(1) I-type: ld, addi, slli, xori, srli, ori, andi, jarl;
(2) S-type: sd, add, sub, sll, srl, xor, or, and;
(3) SB-type: beq, bne, blt, bge;

(4) UJ-type: jal

3.3 Second step, what a CPU needs?

At this point, we have had an instruction memory loaded with machine codes. But what else does a CPU need? Let's revisit Figure 1, to have a fully functioning CPU, one register file is certainly required, but how do you represent it in your C code, an array of uint64_t seems to be a pretty good choice; CPU also needs a data memory right? How to represent it in your code? Maybe an array of uint8_t? And what else you should consider in designing data memory? Hint: Endianness.

Oh, you also need an ALU! I have to stop you right here before you march into the digital logic world. Remember, our simulator is a functional simulator, so we do not model any gate-level structures. To simulate an ALU, you can write a function such as "int ALU (Read Data 1, Read Data 2, ALU Control Signal)". For example, if the value of ALU Control Signal represents addition, the function returns Read Data 1 + Read Data 2, as simple as that!

So, what is your second task? Complete Core.h and Core.c.

Requirement:

(1) Correct representations of a register file and data memory;
(2) Functions to simulate ALU, Control, MUXes…

3.3 Final step, connect everything!

Your final task is to complete tickFunc; a function describes the entire execution flow from fetching an instruction, decoding, and execution.

Requirement:

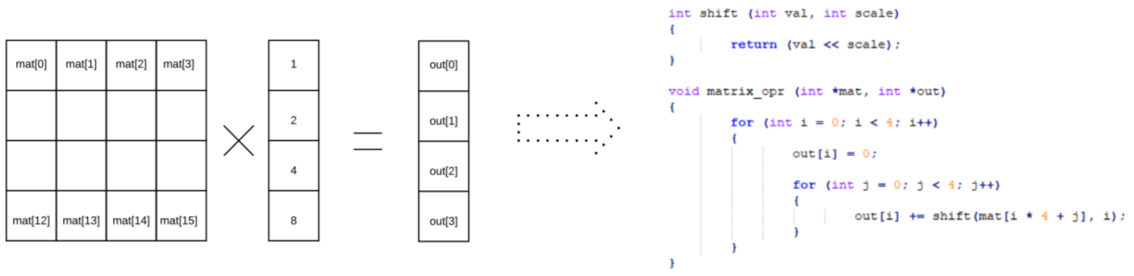
Test your simulator with cpu_traces/example_cpu_trace, also with following configurations:

(1) Set x25 to 4;
(2) Set x10 to 4;
(3) Set data memory from 0th location to uint64_t arr[] = {16, 128, 8, 4}
(4) Set x22 to 1;

Report the followings in your final report:

(1) Value of x9;
(2) Value of x11.

4. Simulating complex programs



```c
int shift (int val, int scale)
{
    return (val << scale);
}

void matrix_opr (int *mat, int *out)
{
    for (int i = 0; i < 4; i++)
    {
        out[i] = 0;

        for (int j = 0; j < 4; j++)
        {
            out[i] += shift(mat[i * 4 + j], i);
        }
    }
}
```

Figure 2 C to RISC-V Assembly Translation

Figure 2 shows a special matrix operation written in C. Please translate it into RISC-V assembly language (assume ***int64_t mat[16] = {0, 1, 2, … 15}***). Your parser doesn't need to resolve any symbols, so when you write something like *jal x1, shift* or *bne x5, x7, exit*, use the corresponding absolute address or relative address instead.

5. Submissions

Coding-wise:

- All your simulator source codes;
- The matrix operation program written in RISC-V assembly language;

Report:

- How do you complete the Core structure? Can you explain in pseudocode?
- How do you test the matrix operation program? Can you print out and explain the final layout of data memory?

Zip above and submit through Bblearn.