ECEC 355 – Computer Architecture

Single-cycle RISC-V Simulation with Exploration to Multi-core System

Instructor: Dr. Anup Das

Distributed, Intelligent, and Scalable COmputing (DISCO) Lab

ECE Department

Drexel University

January 14, 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. There are two parts to this project:

1. In part one, you will design a complete single-cycle RISC-V CPU simulator to be able to simulate complex program.
2. In part two, you will write RISC-V assembly program for single-core and multi-core system then verify the speedup achieved by using multi-core system through simulations.

Please submit your work by February 10, 2019, 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

System: Linux

We highly recommend the latest Ubuntu, however, you can use any flavor of Linux you prefer. If you are using Windows or Mac, please download VirtualBox and install Ubuntu on it.

Software: A simple but cycle-accurate RISC-V full system simulator designed by Drexel DISCO Lab

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

3. Tutorials on Single-cycle RISC-V Simulator Code Template

3.1 Code Template

The code template for a single-cycle RISC-V simulator (under directory *src/single_cycle)* has been given. Figure 1 demonstrates the system architecture of the simulator; an external configuration file provided by the user specifies the number of cores per socket and the operating frequency for all the cores.

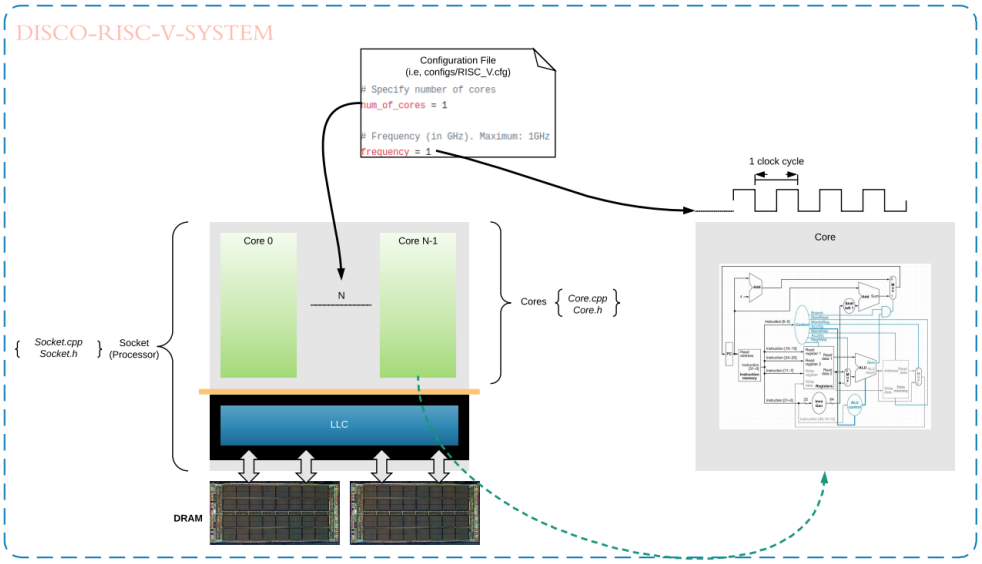

Figure 1 System Architecture of the Simulator

You are encouraged to study the existing source codes in order to have a better understanding on the entire simulation eco-system though we will cover most of it in the later sections. The code hierarchy looks like this.

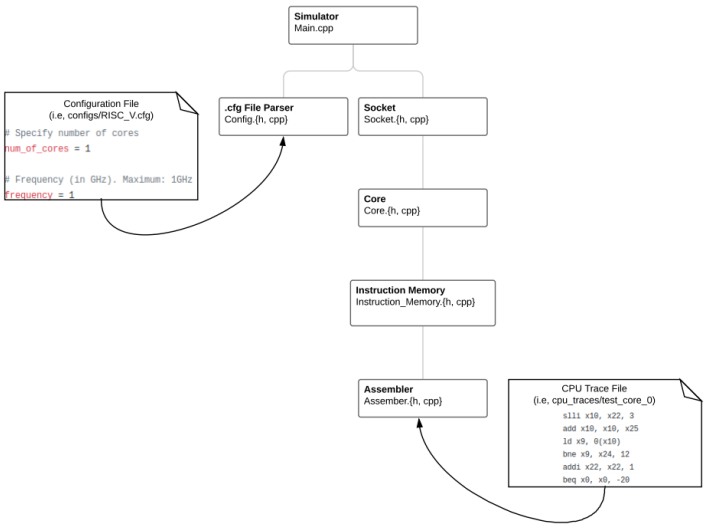

Figure 2 Source Code Hierarchy

Assembler{.h, .cpp} defines a primitive assembler which simply converts RISC-V assembly language into machine language (binary format) and stores the binary format into instruction memory defined by Instruction_Memory{.h, .cpp}. As one of the major components in the single-cycle circuit diagram, the instruction memory, combined with other components that you will design form the Core defined by Core{.h, .cpp}. Socket{.h, .cpp} defines the socket which is simply the container of number of cores specified by an external configuration.

3.2 How it works?

To compile the simulator, navigate to the root directory of DREXEL-DISCO-RISC-V-Simulator and type *bash compile.bash single-cycle*. To run the simulator, one must provide (1) an external configuration file (a sample .cfg file has already been provided under *configs/*), (2) an output file where the simulation results written to, (3) a trace file for each core.

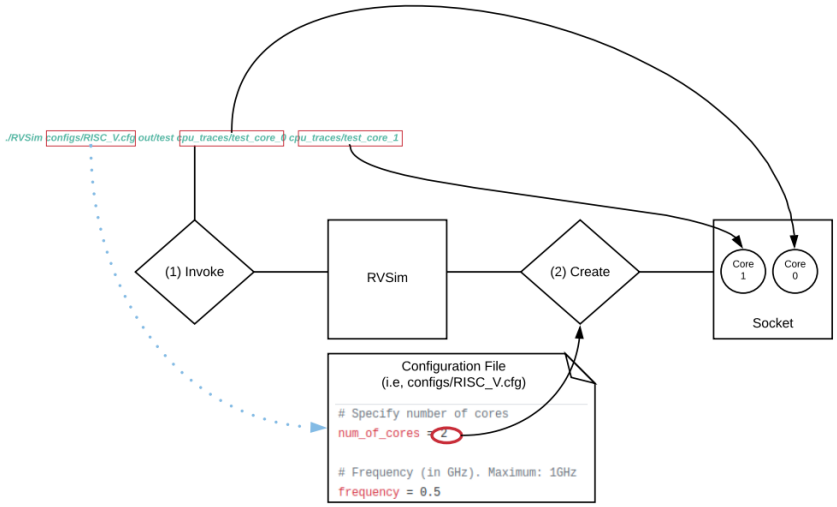

Figure 3 Simulation Stage One - Object Creation

Figure 3 shows the object creation process (Main.cpp, line 24 – 44), the number of cores to be created is specified by the configuration file and each core takes care of one trace file.

Now, let's examine two important classes: Core and Instruction_Memory. As mentioned before, a Core is the place that the single-cycle circuit diagram should be implemented, so it will be a good practice to place all the necessary components under the private section of Core class, an instruction memory has already been given (Core.h, line 42).

Under the private section of Instruction_Memory class (defined in Instruction_Memory.h), each "instruction" is a mapping between an address (multiple of 4) and an Instruction object defined in Instruction.h. We will see the benefit of doing it in the later section.

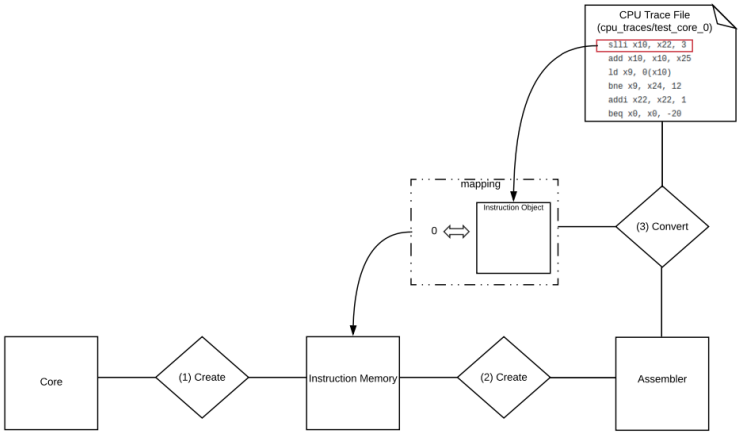

Figure 4 Instruction Memory Creation

As shown in Figure 4, when Instruction_Memory (object) is created, an Assembler (object) will be created along with it. For each instruction in the CPU trace file, the assembler creates an Instruction object and stores the translated machine code into it. The assembler then maps the Instruction object with an address which is always the multiple of 4 since the length of instruction in RISC-V is always 4 bytes and we assume memory is byte-addressable. Finally, all the mappings will be stored in Instruction_Memory. In Core.cpp, line 30, shows how to get an Instruction object with an address.

3.3 Cycle-accurate simulation

A single-cycle CPU means the CPI of all the instructions is 1. As demonstrated in Figure 5, at the beginning of every clock cycle except clock cycle one, there are two events happened: (1) the instruction that gets issued in the previous clock cycle has finished; (2) issues new instruction.

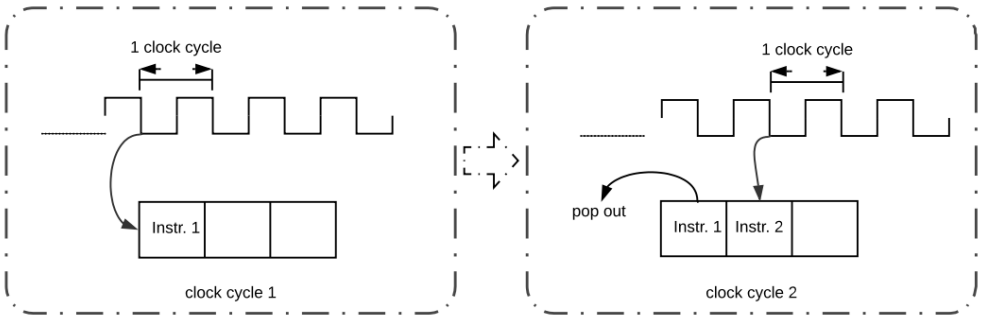

Figure 5 Cycle-accurate Simulation

The tick () function in Core.cpp demonstrates the implementation of such cycle-accurate behavior.

4. What to do

Implement a complete single-cycle RISC-V CPU shown in Figure 4.21 of your textbook. You should have all of the components mentioned in the Figure for your simulator. Here are some important specifications for your design:

- The input of control unit, register file, Imm Gen must be extracted from the binary format machine language.
- The data memory must be byte-addressable. Hints: you can implement data memory as an array of uint8_t.

5. Debugging, Testing and Output File

A sample CPU trace file (cpu_traces/cpu_traces_core_0) has been provided for debugging and testing purpose. Every time an old instruction finishes execution, a new entry for output file will be generated (serve_pending_instrs() in Core.cpp). In order to track the hardware status more comprehensively, you should add more fields (you need to modify function printStats() in Core.cpp) for each entry to be printed, for example, for cpu_traces_core_0 trace file, register x9, x10, x22, x24, x25, memory location 0(x10) are used, so you should have all of them for each entry.

6. Simulating Complex Program

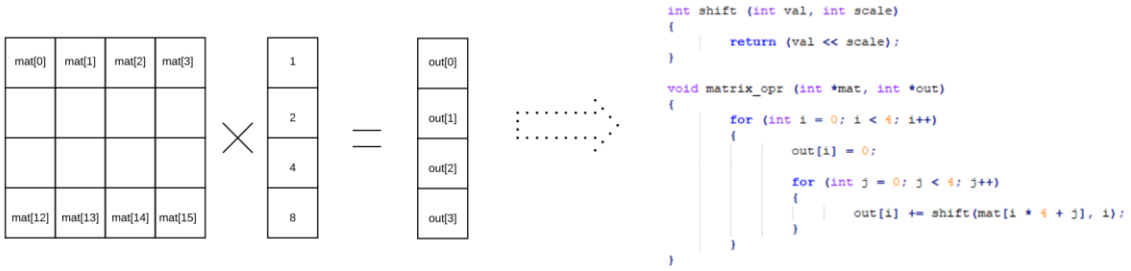

Figure 6 C to RISC-V Assembly Translation

Figure 6 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}***). Please pay attention that (1) our assembler doesn't resolve any symbols, so when you write something lik *jal x1, shift* or *bne x5, x7, exit*, please use the corresponding absolute address or relative address instead; (2) there is always a SPACE after the comma.

7. Exploration to Multi-core System

The simulator calculates execution time in nanosecond every time program finishes execution. Simulate the matrix operation program with two different configurations:

- Configuration (1): num_of_cores = 1; frequency = 1
- Configuration (2): num_of_cores = 1; frequency = 0.5

Summarize what you find in your report. Your final task is to run the simulation with frequency = 0.25, HOWEVER, getting similar or even lower execution time than Configuration (1).

Hints: Only let Core x calculates out[x], for example, core 0 calculates out[0], core 1 calculates out[1]… Does Figure 3 give you any ideas?

8. Submissions

Zip the followings and submit through Bblearn.

- Report on how you complete the Core Class
- The matrix operation program written in RISC-V assembly language
- src/single_cycle/
- Discussions on the experiment of section 7 – Exploration to Multi-core System