HW3

# Q1 Writing a simulator for single-cycle RISC-V datapath.

100 Points

**Grading comment:**

The goal of this assignment is to understand how RISC-V datapath is designed and performed. To achieve this, we will write a cycle-accurate instruction-level simulator that supports a subset of the RV32I ISA (32-bit integer). This instruction-level simulator will model the behavior of each instruction and will allow the user to run RISC-V programs and see their outputs. In subsequent assignments, you will use the result of this assignment as a reference to verify that your later assignments execute code correctly.

**What You Should Do**

Your job is to implement the process\_instruction() function in sim.c. The sim.c file is part of the code template you must use; details about the template will be explained later. The process\_instruction() function should be able to simulate the instruction-level execution of the following subset of entire RV32I instructions:

|  |  |
| --- | --- |
| Instruction type | instruction |
| R | add, slta |
| I | addi, slli |
| S | sw |
| SB | bne |
| U | auipc |
| UJ | jal |

When there is no instruction to process, the process\_instruction() function that you write to set the global variable RUN BIT to 0 so that the program terminates.

Your simulator should precisely simulate each instruction’s behavior by updating the proper register and memory location after executing each instruction. TA/grader will evaluate your simulator using several input cases covering the instruction sets you should implement.

To test the correctness of your simulator, you should run the input programs we provide you with and write one or more programs using all the required RISC-V instructions that are listed in the table above and execute them one at a time (run 1). You can use the rdump command to verify that the state of the machine is updated correctly after the execution of each instruction.

Each RISC-V instruction you are supposed to implement belongs to one of the RISC-V instruction types: R, I, S, SB, U, and UJ. Refer to the RISC-V reference data or lecture slides for details of each instruction type. While the table has many instructions, there are only a few unique instruction behaviors with several minor variations.

Finally, note that your simulator does not have to handle instructions we do not include in the table above or any other invalid instructions. We will only test your simulator with a valid code that uses the instructions listed above.

The simulator will take an input file that contains a RISC-V instruction or program. Each input file line corresponds to a single RISC-V instruction written in a hexadecimal string. For example, add t1, t2, t3 or (add x6, x7, x28) corresponds to 01c38333 in a hexadecimal representation. We will provide several input files. However, you should also create additional input files to test your simulator comprehensively.

The simulator will execute the input program one instruction at a time. After each instruction, the simulator will update the RISC-V architectural state: values stored in registers and memory. The simulator is partitioned into two main sections: the (1) shell and the (2) simulation routine. Your job is to implement the simulation routine.

The code template for this assignment is provided, containing 4 files: Makefile, shell.c, sim.c, and shell.h. The shell.c (including shell.sh) implements an interactive shell for running the simulator. DO NOT modify both files unless you want to debug a shell feature. In the src/ directory, we provide two files (shell.c and shell.h) that already implement the shell. There is a third file (sim.c) where you will implement the simulator routine - this is the only file you can change.

# Interactive shell

The provided shell.c implements several command lines to control the execution of the simulator. You can think of it as a command line version of the RISC-V simulator in Ripes (<https://github.com/mortbopet/Ripes>), like loading and running a program, examining register and memory values, etc. The shell accepts one program file as a command line argument and loads it into the program memory. The shell supports the following commands:

* g | G | go: simulate the program until it indicates that the simulator should halt.
* r | R | run <n>: simulate the execution of the machine for n instructions.
* mdump <low> <high>: dump the contents of memory from location low to location high to the screen. <low> and <high> addresses should be provided as hexadecimal numbers.
* rdump: dump the current instruction count, the contents of x0- x31, FLAG N, Z, C, V, and the PC to the screen.
* i | I | input reg\_num reg\_val: set register (denoted reg\_num) to value (reg\_val).
* ? | help: print out a list of all shell commands.
* q | Q| quit: quit the shell.