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.

# The Simulation Routine

The simulation routine carries out the instruction-level simulation of the input RISC-V program in machine code. During the execution of an instruction, the simulator should take the current architectural state and modify it according to the ISA description of the instruction in <https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf>. The architectural state includes the PC, the registers, FLAGS, instruction formats, and the memory. The state is contained in the following global variables:

#define RISCV\_REGS 32

typedef struct CPU\_State {

uint32\_t PC; /\*program counter\*/

uint32\_t REGS[MIPS\_REGS]; /\*register file.\*/

int FLAG\_NV; /\*invalid\*/

int FLAG\_DZ; /\*divide by zero\*/

int FLAG\_OF; /\*overflow\*/

int FLAG\_UF; /\*underflow\*/

int FLAG\_NX; /\*inexact\*/

} CPU\_State;

/\* STATE\_CURRENT is the current arch. state \*/

/\* STATE\_NEXT is the resulting arch. State

after the current instruction is processed \*/

CPU\_State STATE\_CURRENT, STATE\_NEXT;

int RUN\_BIT; /\*initialized to 1; need to be changed to 0 if the HLT instruction is encountered\*/

Furthermore, the simulator models the simulated system's memory. You need to use the following functions, which we provide, to access the simulated memory:

uint32\_t mem\_read\_32(uint64\_t address);

void mem\_write\_32(uint64\_t address, uint32\_t value);

As mentioned in the lecture, memory is byte-addressable whereas registers are word-addressable. Furthermore, we will implement a little-endian architecture where bytes are ordered from the big end (most significant bit).

The provided simulator template, sim.c includes an empty function named process\_instruction(), which is called by the shell and simulates one machine instruction at a time. You have to write the code for process\_instruction() to simulate the execution of instructions. You can also write additional functions to make the simulation modular. (Keep in mind that you will be using the code that you write in the later assignment in order to validate your work.)

# Template Files

The template contains all the code for this assignment. You can compile the simulator with the provided Makefile. You will be provided with the same input files for testing your implementation. To test whether your implemented simulator supports all ISAs you are supposed to implement, you should create your own test files, which should be in hexadecimal formats. The simulator won’t accept files in other formats.

We assume your programming environment should be in Linux or similar. If you have access to a Linux machine (either a native or a virtual machine), ensure the system includes the GCC compiler. The provided template should work with any version of GCC. If your code uses a different compiler than GCC, you should provide enough information (how to compile and run) in your submission. If you use Macintosh, you should also be able to compile and run C code using Terminal. Lastly, if your system is Windows, Cygwin (<https://www.cygwin.com/>) is strongly recommended. After installing Cygwin, you will have a Linux-like prompt where you can compile and run C programs using GCC.

# Test Cases

Use the following cases to test individual instruction. Put the machine code portion in a text file (e.g., auipc.mem) and use it as an input to the compiled simulator. For example, ./sim auipc.mem.

|  |  |
| --- | --- |
| Machine code | Corresponding RISC-V instruction |
| 10000917 | auipc x18 0x10000 |
| 00090913 | addi x18 x18 0 |
| 015ea023 | sw x21 0 x29 |
| 002e1e93 | slli x29 x28 2 |
| 015a0b33 | add x22 x20 x21 |
| 013e2f33 | slt x30 x28 x19 |
| fe0f10e3 | bne x30 x0 -32 |

The following is the sequence of machine codes that perform iterative Fibonacci.

10000917

00090913

00a00993

00200e13

00000a13

00490e93

00100a93

015ea023

002e1e93

015a0b33

000a8a13

000b0a93

01d90bb3

016ba023

001e0e13

013e2f33

fe0f10e3

00000013

# Submission Guideline

* All program submissions should be made to Gradescope.
* Write down your name at the top of the sim.c file.
* Document your code well, so it is easily readable.
* You need to upload any additional documents or test files/cases that are used in your implementation.

# Useful Tips

* Please refer to RISC-V green sheet. <https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf>.
* Use the Ripes simulator (<https://github.com/mortbopet/Ripes>) to generate the corresponding machine code for an individual instruction or a RISC-V program.
* Use the Ripes simulator also to convert decimal to binary or hexadecimal, or vice versa.
* Again, you are allowed to modify sim.c only and leave the other files as they are.

Note that you MUST use the provided mem\_read\_32 and mem\_write\_32 functions to implement loading and storing 8-bit (byte). Note that you need to call mem\_read\_32 and mem\_write\_32 with only 32-bit-aligned addresses (i.e., the bottom two bits of the address should be zero). Note also that you need to modify only the appropriate part of a 32-bit word.

# Additional Tips

* sim.c is the only file you need to update and upload.
* Keep the structure in process\_instruction() as it is and modify the body of fetch(), decode(), and execute() only.
* You must define global variables to pass around the information processed in each stage (function). For example, there should be a variable for the fetched instruction in fetch() so decode() can decode the instruction and read registers (e.g., rs1)
* You must update the PC within fetch(). This can be done using NEXT\_STATE.PC = CURRENT\_STATE.PC + 4;
* In decode(), you must decode all the fields in each instruction type, beginning with opcode.
* In execute(), you must implement each instruction’s functionality based on the decoded values from decode(). For example, if the fetched instruction is add x3, x2, x1, you need to perform add using the values in x2 and x1 (which are read during decode() and save the outcome to x3.

# Grading Guidelines

* Your simulator compiles without errors. (and it appears to be correct on the surface): 10 pts.
  + The TA/grader will use the make command to compile your program.
  + Submit only your implemented sim.c file, including all your comments within it.
* The program executes correctly: 80 pts (10 pts/instr)
  + Should pass rigorous test cases.
* Documentation and comment on the program: 10 pts
  + Have a commented program header with your name and assignment number.
  + The program should be well commented so much so the TA/grader can understand what you are doing without you there to explain it.