ENSC 254 Lab Assignment 3

Important Logistics:

- Lab 3 weighs 9% of the final marks.
- Lab 3 will be done and graded per 2-student group, as detailed in the lab logistics.
- Please make sure your code can compile and run correctly on the lab computers (i.e., FAS-RLA Linux computers). If your code cannot compile on the lab computers, then you get 0 mark for lab 2. If your code can compile, but cannot run correctly, then you only get the points where your code runs correctly on the lab computers.

Introduction of Lab 3:

The purpose of this assignment is to enhance your understanding of the RISC-V instruction set design and the functionality implemented by a CPU. Specifically, you will develop an emulator in lab 3, which is built on top of your lab 2. An emulator is a software model that simulates the functionality of a target CPU (e.g., RISC-V CPU in lab 3) on a host CPU (e.g., X86 CPU in lab 3). As a result, even though we don't have a real RISC-V computer, we can run RISC-V programs inside the emulator, which is running on the real X86 computer.

Emulators are commonly used, and popular examples include Venus, QEMU, and other virtual machines that you may have used in the past. In fact, if you use a Macbook Pro/Air laptop that is equipped with an ARM CPU, some of the legacy software—which only runs on an X86 CPU—is also running inside an emulator; this emulator simulates an X86 CPU on a real ARM CPU.

Take the challenge and have fun :-)

Instructions That You Need to Support

The instructions that your emulator must handle are the same as the list in lab 2. For more details, please refer to lab 2 description and the RISC-V green card.

Framework Code

Your lab 2 code has included all the necessary framework code. **Here, we assume your lab 3 folder is called 'lab3', which is the same code from your lab 2 (change your folder name to lab3), except that we have updated the *WorkDistForGrading.csv* file**: ***please download it from Canvas and replace the old file in the lab3 folder.***

First, it performs the following emulator setup:

- It reads the program's machine code from an input file (passed as a command line parameter) into the simulated memory (starting at address 0x01000). Each program is given 1 MiB of memory and is byte-addressable. It uses little-endian.
- It initializes all 32 RISC-V registers to 0 and sets the program counter (PC) to 0x01000. The only exceptions to the register initializations are the stack pointer (set to 0xEFFFF) and the global pointer (set to 0x03000). In the context of our emulator, the global pointer will refer to the static portion of our memory. The registers and PC are managed by the ***Processor*** struct defined in types.h.
- It sets flags that govern how the program interacts with the user. Depending on the options specified on the command line, the simulator will either show a disassembly dump (-d) of the program on the screen (i.e., lab 2), or it will execute the program (i.e., lab 3). More information on the command line options is below.

Then, it enters the main simulation loop, which simply executes a single instruction repeatedly until the simulation is complete. Executing an instruction performs the following tasks:

- [Framework code] It fetches an instruction from the simulated memory, using the PC as the address.
- **[Your lab 3 task]** It examines the opcode/funct3/funct7 to determine what instruction was fetched.
- **[Your lab 3 task]** It executes the instruction and updates the PC.

The framework supports a handful of command-line options:

- -i runs the simulator in interactive mode, in which the simulator executes an instruction each time the "Enter" key is pressed. The disassembly of each executed instruction is printed. This option could be very useful for debugging purpose.
- -t runs the simulator in tracing mode, in which each instruction executed is printed.
- -r instructs the simulator to print the contents of all 32 registers after each instruction is executed.
- -v initialize the register values to a nonzero value. Only used to test R-type instructions.
- -d instructs the simulator to disassemble the entire program, then quit before executing.
- -e instructs the simulator to simulate till the end of the program (the end of the program indicated by the "ecall" instruction. It's helpful when programs have loops).

In lab 2, you should have already developed the code to parse and disassemble an instruction, which will be reused in lab 3; if you didn't pass all the lab 2 tests, please review lab 2 solution tutorial by the TA and revise your lab 2 code.

In lab 3, you will focus on the implementation to execute each parsed RISC-V instruction, update CPU registers, memory values, and the PC. We have also provided example code to walk you through the process of how to execute a few R-type instructions such as add, sub, and mul.

Your TODO list:

In lab 3, you will only need to work on one file: *emulator.c*; it's ok for you to revise utils.c and disasm.c from lab 2 based on the TA tutorial. Please do not change any other files. To

implement the emulator in lab 3, you will implement the following functions. The function prototypes have been given; don't change them. Please change the function bodies, which are indicated by the comment "/* YOUR CODE HERE */". If you want to add extra helper functions, it's ok; but don't change existing function prototypes.

- **execute_*() in emulator.c**: These functions handle the majority of the execution of the RISC-V instructions that we implement in lab 3. You should update the appropriate register values, interact with memory when applicable, and update the PC. You will find the RISC-V green card and the example code for R-type instructions particularly helpful here.
- **store() in emulator.c**: This function takes an address, a size, and a value and stores the first -size- bytes of the given value at the given address. You will find the example code for the ***load*** function particularly helpful here.
- **PC updates in each execute_*() function**: The program counter (PC) register in the RISC V processor is special. It is a pointer to the memory location of the next instruction that will be run by the processor.
    - PC itself is just a pointer. Every step the processor has to read the memory location pointed to by the PC. We have provided the example ***load*** function to read memory.
    - To make progress this has to be incremented or changed after running the current instruction. If it appears that your program is not making any progress, e.g., it is stuck running the same instruction repeatedly then check whether you are updating the PC in ***execute_*()*** functions.
    - The program counter is updated explicitly by branches (SB type), JAL (J type), and JALR (I type). These instructions include immediates that need to be extracted and sign extended (helper function that you finished in lab 2).
    - You may also need to do some arithmetic on the offset before setting the PC. This arithmetic may need you to read other registers (jalr) or even the PC itself (e.g., jal). Sometimes instructions may update the PC and other registers (e.g., JAL). Double-check the calculation of these immediates.

Basic Commands and Testing:

- To build your emulator, type the following command on a lab computer (all commands are typed when you are inside the lab3 folder):

  make riscv

  The built executable is called "riscv". Note: typing "make" does the same thing.

- To run your emulator with a test input, type the following command:

  ./riscv -r code/input/R/add.input

  -r option tells the riscv executable to print the contents of all 32 registers after each instruction is executed. The input is the "code/input/R/add.input" file.

  To redirect this output into an output file ("code/out/R/add.trace"), instead of printing it onto the screen, type the following command:

  ./riscv -r code/input/R/add.input > code/out/R/add.trace

- To validate the correctness of the above test, type the following command:

  diff code/out/R/add.trace code/ref/R/add.trace

  For every input file (except blind tests), we have also provided the golden reference file (.trace files in the corresponding code/ref/ folder) to compare against. In fact, we have provided a Python script to do a more detailed comparison and give more user friendly hints if your solution doesn't match with the golden reference; simply type the following command:

  python3 emulator_tester.py R/add

  You can use gdb or cgdb to help the debugging and code fix. Note we already provided the example code to execute add, sub, and mul.

More Testing and Grading:

- **[65 points] Test 1: test for each type of instruction,** including
  - [7 points] R-type instructions in code/input/Ri/Ri.input. 10 instruction in total in this input file and we have provided the code to execute 3 of them. For the remaining 7 instructions, passing each instruction gives you 1 point for lab 3. The golden reference for this input is in code/ref/Ri/Ri.trace. To perform this test, type the following commands

    ./riscv -v -r ./code/input/Ri/Ri.input > ./code/out/Ri/Ri.trace

    python3 emulator_tester.py Ri/Ri

    Note R-type instructions are special: we can't initialize the CPU registers to 0; otherwise, it will always calculate a result of 0 or fail (e.g., divide by 0). So we need to provide an extra -v option to initialize the CPU registers to nonzero values. That's why we use the folder name Ri (i for initialization) instead of R in this test.

  - [19 points] I-type instructions in code/input/I/I.input (I-type except load instructions) and code/input/I/L.input (load instructions). 19 instruction in total (6 load instructions and 13 non-load instructions) and each counts for 1 point. To perform these two tests, type the following commands

    ./riscv -r ./code/input/I/I.input > ./code/out/I/I.trace

    python3 emulator_tester.py I/I

    ./riscv -r ./code/input/I/L.input > ./code/out/I/L.trace

    python3 emulator_tester.py I/L

    For the following types, the golden reference files are also in the code/ref folder, following the same naming convention and the commands are also similar. A complete set of commands are provided in ***test_emulator.sh***.

  - [10 points] S-type instructions in code/input/S/S.input. 10 instruction in total and each counts for 1 point. Note this only partially tests the correctness of S-type instructions: it doesn't test the correctness of the written memory values, but simply

makes sure the CPU registers are not accidentally modified. Correctness test for S-type instructions is integrated in entire program test.

- [10 points] SB-type instructions in code/input/SB/SB.input. 10 instruction in total and each counts for 1 point.
- [10 points] U-type instructions in code/input/U/U.input. 10 instruction in total and each counts for 1 point.
- [9 points] UJ-type instructions in code/input/UJ/UJ.input. 9 instruction in total and each counts for 1 point.

- **[30 points] Test 2: test for entire program,** including
  - The following two tests are for end-to-end program test, where each program includes a mix of different types of instructions. Once you passed the test for each type of instructions, this part of test should be relatively straightforward. Here we want to execute the program dynamically, i.e., determine the number of instructions based on the dynamic program execution, so we have to pass an extra -e option to the emulator. -e option will instruct the emulator to call the execute function in an infinite loop and exit until it encounters a special "ecall" instruction (we already provide the code to execute the "ecall" instruction). Don't try the -e option in the single type instruction test, as those input programs don't include the "ecall" instruction and thus the emulator will never stop (due to the infinite loop).
  - [15 points] multiply test: code/input/multiply.input. 75 instructions dynamically executed in total (excluding "ecall") and each counts for 0.2 point. To perform this test, type the following commands

    ./riscv -r -e ./code/input/multiply.input > ./code/out/multiply.trace

    python3 emulator_tester.py multiply
  - [15 points] random test: code/input/random.input. 15 instructions dynamically executed in total (excluding "ecall") and each counts for 1 point. To perform this test, type the following commands

    ./riscv -r -e ./code/input/random.input > ./code/out/random.trace

    python3 emulator_tester.py random
  - You may need to debug the code to pass these two tests.

- **[15 points] Test 3: test for blind set of instructions without references**
  - Unlike Test 1 and Test 2, there are no step marks for Test 3, since even one failing instruction in a real-world application makes the entire program fail. Successful execution of each of the 3 blind tests is awarded with 5 points; otherwise, you get 0 point for each failed blind test. These tests check for corner cases in the emulator.
  - **!Important! Remove the invalid instruction(s)** from code/input/blind_set1.input and code/input/blind_set2.input before starting the test. This is to make sure that your emulator does not see the invalid instructions, which would give you different outputs than expected. **Hint:** You can look at the disassembler output based on your lab 2 to check which instructions are invalid.

- [5 points] blind_set1: code/input/blind_set1.input. To perform this test:
  ./riscv -r ./code/input/blind_set1.input > ./code/out/blind_set1.trace
- [5 points] blind_set2: code/input/blind_set2.input. To perform this test:
  ./riscv -r ./code/input/blind_set2.input > ./code/out/blind_set2.trace
- [5 points] blind_set3: code/input/blind_set3.input. To perform this test:
  ./riscv -r -e ./code/input/blind_set3.input > ./code/out/blind_set3.trace
- If you had thoroughly emulated the execution of all instructions listed in the lab 2 table, this part of the test should be straightforward.
- The golden reference file is not provided (that's why it's called a blind set) and you need to manually execute the instructions to verify the correctness of your output. **Hint:** you may execute these instructions in the Venus simulator (Tutorial 2).

- **Guidance for workload distribution**. Below is an example distribution, you may use a different distribution. **Please document your workload distribution in the updated *WorkDistForGrading.csv* file inside the lab3 folder. In the TA tutorial, TA will show you how to fill in this .csv file. Please strictly follow the format to make it easy for the auto-grading; otherwise, you will lose 10% of the total points of this lab.**
  - **Student 1 responsible for 55 points**: R-type instructions (7 points), I-type instructions except loads (13 points), SB-type instructions (10 points); random test (15 points), blind_set1 (5 points) and blind_set3 tests (5 points).
  - **Student 2 responsible for 55 points:** Load instructions (6 points), S-type instructions (10 points), U-type instructions (10 points), UJ-type instructions (9 points); multiply test (15 points), and blind_set2 test (5 points).

Assignment Submission:

Your lab assignment 3 will be submitted electronically through Canvas. You will need to submit a single ***YOUR_LAB_GROUP.zip*** file (e.g., LA01_01.zip). **Please watch the TA tutorial.** This format makes it easy for the TAs to do auto-grading. **If you fail to comply with this format, you will lose 10% of the total points of this lab**. To zip your files in Linux,

1. *Go to your lab3 folder (assume you renamed your lab2 folder to lab3)*
2. *make deepclean //**make sure you clean up your files***
3. *cd .. //go one level up*
4. *mv lab3 **YOUR_LAB_GROUP** //rename your lab3 folder using your lab group (e.g., LA01_01)*
5. *zip -r YOUR_LAB_GROUP.zip YOUR_LAB_GROUP //zip all your lab3 files into a single .zip*

Submission Deadline:

You need to meet the deadline: every 10 minutes late for submission, you lose 10% of the points; that is, 100 minutes late, and you will get zero for this lab.

Lab Demonstration:

To finalize your lab 3 points, you will have to demo your lab code to your TA in the lab demo sessions that you enrolled**.**

Each student group has around 10 minutes to explain their code to the TA. Only code from your Canvas submission is allowed in the lab demo. If you fail to do the demo (without a medical note), you will be awarded a zero on this lab assignment. **If it is determined that you do not understand your code being evaluated (the part you are responsible for based on your workload distribution), you will be awarded zero and considered as CHEATING on this lab assignment.** Also please show up on the demo day on time (TAs will send out your scheduled time), otherwise you will lose 10% of the lab 3 points.