# Homework 2: Single Cycle RISC-V Simulator

**Due Date: 2024/09/24**

**Name:**
**Computing ID:**
**Collaborator(s):**

**Guidelines:**

In this lab, we will incrementally develop a single cycle RISC-V simulator, which will support most RISC-V **RV32I** instructions. In the end, this simulator can read from a RISC-V executable binary, parse the binary encoding to get the correct instructions, simulate the execution and produce the right value inside each register.

The whole simulation framework will be built using python, no RISC-V toolchain (e.g., riscv-32-gcc compiler) is needed (but if you're interested in that, check page 6). All testcase binaries along with their source codes, disassembled codes are distributed. You will only need to install two related python packages, which has been shown below (any version should work):

1. numpy, through `[conda/pip] install numpy`
2. pyelftools, through `pip install pyelftools`

All the code is tested using python-3.11.5, numpy-1.24.3, pyelftools-0.31
Any OS should work.

---

### Task 0: Overview

The entry point of this project is in entrypoint.py, what parameters can be accepted is specified in function _parse_args(). An example execution command would be:

```
$ cd src
$ python ./entrypoint.py ../tests/sum100
```

There is an auto_run.sh script that can launch all test cases and store the corresponding output into a directory named your_solution. Please use this script to generate all your outputs. Meanwhile, standard outputs are stored in a directory named samples. There is also an auto_compare.sh script that can be used to compare your outputs with the samples.

At the end of this homework (also see page 5 for submission guidelines), you are required to submit the whole project in a zip file. Make sure your output format is the same as samples.

To have an overview of the whole project, please understand how **main** function (entrypoint.py) works. It will first create a **RISCVSIM** object, which will bind with few structures like register file, memory, etc. It will work as a simulated RISCV CPU.

Then the program will be loaded from an executable file (in ELF format). While loading these binary instructions, they will be parsed and stored into memory structure. And entry point of this program will be returned.

Finally, this entry point (an address) will be set as the initial PC for the simulator to start execution. All the execution simulation steps are warped in another class named **SIMULATOR**, which will also be bind with the **RISCVSIM** (riscvsim.py). Each instruction will be executed in exact one cycle through the function **SIMULATOR.cycle()**.

After simulation is done, it will dump all the registers, and then verify whether the simulation results are correct.

---

### [20 points] Task 1: Constants Definitions

**Finish all the implementations needed inside the file definitions.py. And have an overview of this file.**

This file includes most RISC-V related constants, such as special instruction encoding (e.g., NOP), instruction filed mask and offset and so on. Most of these values will be used in other files for controlling the RISCV simulator. And the comments in this file described clearly the meaning of each constant and where it will be used.

Noticed that all requirements which involve your implementations are indicated using "**TASK--IMPLEMENTATION NEEDED**" and "**YOUR_CODE_HERE**" (same for later tasks), search this to locate all the tasks.

Use the guidelines mentioned in that file, also reference [1-5] will be useful during your implementation. **It's recommended to always have [1] on your side while doing this homework.**

Do not modify other parts of the codes (including function signature), if you wish to add debug information or something else, make sure they are removed (commented) in the end when you submit your code (you can also use the **LOG** class in structure.py to set your debug level).

---

## [20 points] Task 2: RISCVSIM Structure

**Finish the __init__ function of class RISCVSIM (defined in riscvsim.py), all the components like program counter (PC), register file (RF) and memory (MEM) are specified in structure.py. Therefore, you will first need to finish the implementation of these structures and then bind them to RISCVSIM.**

1. **Implement REG()**
2. **Implement RF()**
3. **Implement MEM()**
4. **Implement STATS()**
5. **Implement RISCVSIM.__init__()**

Guidelines are provided in these related files, and your tasks are specified just as in Task 1. You will also need to use some constants you defined in Task 1.

---

## [20 points] Task 3 & 4: Load the Program and Dump the Instructions

**Have a look at parser.py and the load_program() function, understand how to extract binary instruction from an executable file (which is in ELF little endian format). You can find it from [3]. There is no implementation needed.**

**Now the program is loaded into the memory structure you implemented, and you will need to implement the INSTRUCTION class to correctly parse these binary strings.**

1. **Implement ISA masks and encodings in isa_encoding.py.**
2. **Understand the structure rv32i_t.**
3. **Implement INSTRUCTION class.**

a. **Implement get_opcode()**
b. **Implement getter of rs1, rs2 and rd.**
c. **Implement get_sign_extend()**
d. **Implement get_imm_b()**

Then comment line 60 and line 63 with # in entrypoint.py. Now you should be able to see the output of **prog.dump()** (line 57). You can examine whether the current simulator can extract the correct RISCV instructions from executable, parse them and print all fields involved with the instruction.

Check all output to make sure they're correct (compare with the standard outputs, which will also be distributed along with this document), once **Task 4** is finished, you can proceed to the cycle level simulation tasks. Remember to remove the comment of line 60 and line 63.

---

## [20 points] Task 5-A:

**In riscvsim.py, finish the A part of SIMULATOR class. Understand how cycle() works.**

1. **Complete run()**
2. **Implement IF()**
3. **Implement ID()**

---

## [20 points] Task 5-B:

**In riscvsim.py, finish the B part of SIMULATOR class.**

1. **Implement EX()**
2. **Implement alu_fn()**
3. **Implement mem_fn()**
4. **Implement ctrl_fn()**

Both part A and part B are trying to implement functional simulation of a RISC-V program. As we don't consider pipeline in this homework, every instruction will be finished in exact one cycle, so the timing simulation is kind of not investigated.

After **Task 5** the simulator should be able to produce complete results. You can launch auto testing using auto_run.sh. Check samples/* to see the standard output. Make sure you get the same results (you can use auto_compare.sh to see the differences) ☺

**Submission Guideline:**

1. The homework will be distributed as a zip file named: **HW2-riscv-sim.zip**
2. After unzipping this file, you will get only one folder: **HW2-riscv-sim/**
3. Make sure your code will output the same as samples and place your log files into **your_solution/** (can be done through auto_run.sh).
4. When you submit, rename the folder **HW2-riscv-sim/** to **HW2-riscv-sim-ID/**.
   e.g., HW2-riscv-sim-xqg5sq/
5. Then compress the whole folder into **HW2-riscv-sim-ID.zip** and submit it to GradeScope.
   e.g., HW2-riscv-sim-xqg5sq.zip

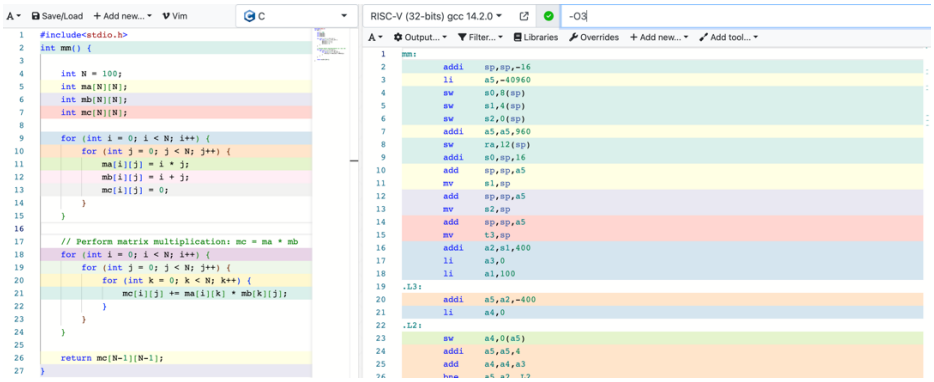
Make sure when you unzip this file, you should get and only get one folder named **HW2-riscv-sim-ID/**, and inside that folder should include samples/, src/, tests/, your_solution/.

## Cross Compile a RISCV Program

If you are interested in using this simulator to simulate your program or extend this simulator for fun. Here are the steps **[DO NOT SUBMIT YOUR EXTENSION]:**

1. Install requirements [mentioned in the link below]
2. Install riscv-gnu-toolchain [might take hours]
   ```
   $ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
   $ cd riscv-gnu-toolchain
   $ mkdir build
   $ cd build
   $ ../configure --prefix=[path to install] --with-arch=rv32im
   ```
3. Export the bin/ folder to PATH (write to your shell config file)
   ```
   $ export PATH=[path to install]/bin:$PATH
   ```
4. Check compilers are installed
   ```
   $ which riscv32-unknown-elf-gcc
   -> [path to install]/bin/riscv32-unknown-elf-gcc
   ```
5. Write a C function

   You can use https://gcc.godbolt.org/ to generate its assembly code by selecting the compiler to RISC-V (32-bits) gcc. You can also set optimization flags.



6. Turn this assembly into a simulator compatible format file (save as xxx.S)

   Use these lines to replace the first label (e.g. mm:)

```
        .text
        .align  2
        .globl  _start          # Declare entry point
_start:                         # Entry point label
        lui     sp, 0x80f10     # Match our memory layout
        xxxxxxxx                # Other codes
```

7. Replace the last `ret` or `jalr/jr` instruction to `ebreak` to force simulation completion.
8. Use the provided makefile and link script (link.ld) to compile the assembly file.
   ```
   $ make
   -> generate executable and xxx.objdump
   ```
   Now you can simulate this executable [make sure the instructions are supported, or you extend it]
9. To validate results, you can use spike and pk, which are also inside this toolchain.

   They follows standard GNU build steps [just like you make the compiler], notice the config for pk: `../configure –prefix=[path to install] --host=riscv32-unknown-elf --with-arch=rv32im_zicsr_zifencei`

**References**

[1] RISC-V Reference Card: [LINK]
[2] RISC-V ISA Documentation: [LINK]
[3] Executable and Linkable Format (ELF) [LINK]
[4] RISCV Reference Card 2: [LINK]
[5] RISCV ISA Manual: [LINK]