PA #2: Building a Simple RISC-V (RV32I) Simulator

Due 11:59PM, May 23th

1. Overview

This project is to build a simulator of a subset of the RISC-V instruction set. The simulator loads a RISC-V binary into a simulated memory and executes the instructions. Instruction execution will change the states of registers and memory.

2. Simulation Details

For a given input RISC-V binary (the output binary file from the assembler built-in Project 1), the simulator must be able to mimic the behaviors of the RISC-V ISA execution.

2.1 States

The simulator must maintain the system states, which consist of the necessary register set (`x0-x31`, `PC`) and the memory. The register and memory must be created when the simulation begins.

2.2 Loading an input binary

For a given input binary, the loader must identify the text and data section sizes. The text section must be loaded to the simulated memory from the address `0x00400000`. The data section must be loaded to the simulated memory from the address `0x10000000`. In this project, the simple loader does not create the stack region.

2.3 Initial states

- PC: The initial value is `0x00400000`.
- Registers: All values of `x0` to `x31` are set to zero.
- Memory: You may assume all initial values are zero, except for the loaded text and data sections.

2.4 Instruction execution

With the current PC, 4 bytes from the memory is read. The simulator must parse the binary instruction and identify what the instruction is and what the operands are. Based on the RISC-V ISA, the simulator must accurately mimic the execution, which will update either the PC, registers, or memory.

2.5 Completion

The simulator must stop after executing a given number of instructions.

2.6 Supported instruction set (same as Project 1)

ADD	SUB	ADDI	AND	ANDI	BEQ	BNE
JAL	JALR	AUIPC	LUI	LW	LA*	XORI
OR	ORI	SLTIU	SLTU	SLLI	SRLI	SW

2.7 Caution

Do not use the output files from the result of Project 1 (assembler). Even though your achieved score from the last Project was 100 points, there can be some minor bugs in the assembler. Therefore, using the output from Project 1 can lead to unsuspected results in Project 2.

3. Getting and Understanding the Skeleton Code

Like Project 1, you can download the skeleton code from the GitHub repository to the server or local machines. Then you are ready to start the project.

→ Go to the following page: https://github.com/Sys-KU/kece343-comm307-project2

3.1 Task #1

To simulate the given RISC-V binary, you need to load the binary into the simulated memory. In the `loader.c` file, there is an incomplete code for loading the binary file. It reads the first line of the binary file to extract the text section size, but the remaining steps are not implemented. You should complete the `load_program()` function as indicated by **`TODO`**.

3.2 Task #2

Once you successfully load the binary into the simulated memory, you can fetch an instruction from the simulated memory. We provide the implementation of `fetch()` function. It just reads 32B data from the text memory. Then, you are supposed to implement the following two functions in the `proc.c` file as indicated by **`TODO`**.

```
struct inst_t decode(int word)
void execute(struct inst_t inst)
```

In `proc.c`, the `cycle()` function invokes `fetch()`, `decode()`, and `execute()` functions and then adjusts the program counter and ticks of the `g_processor` object. In the `decode()` function, you need to interpret a given 32 bit instruction as an argument and then compose a `struct inst_t` data type. Once you decode an instruction, you can simulate the execution of the instruction. You may need to read data from the simulated memory or write data to the simulated memory. For the `R-type` instructions, you should update the register values in the simulated processor (the `g_processor` object). For the jump instructions such as `J` and `JAL`, the program counter register

should be updated.

4. Building and Running the Simulator

As Project 1, we provide the Makefile to build and test your code. If you are not familiar with how Makefile works, we highly recommend you visit this website.

4.1 Building your code

Step1: Let's move the directory we cloned

```
$ git clone git@github.com:Sys-KU/kece343-comm307-project2.git
$ ls
kece343-comm307-project2

# TIPS: Try to use the Tab key when navigating directories or files
$ cd kece343-comm307-project2
```

Step 2: Trying to build the skeleton codes

```
$ ls
loader.c    loader.h    main.c    Makefile    mem.c    mem.h    proc.c    proc.h
README.md  sample_input  sample_output  util.c  util.h

$ make # the warning messages come from the skeleton code
gcc -g -Wall -std=gnu99 -O3 -c main.c -o build/main.o
gcc -g -Wall -std=gnu99 -O3 -c util.c -o build/util.o
gcc -g -Wall -std=gnu99 -O3 -c proc.c -o build/proc.o
proc.c: In function 'decode':
proc.c:37:11: warning: unused variable 'buffer' [-Wunused-variable]
   37 |     char* buffer = dec_to_bin(word);
      |           ^~~~~~
proc.c:36:64: warning: unused variable 'imm' [-Wunused-variable]
    36 |       char func7[8], rs2[6], rs1[6], func3[4], rd[6], opcode[8],
imm[21] = {0};
      |                                                                ^~~
proc.c:36:53: warning: unused variable 'opcode' [-Wunused-variable]
    36 |       char func7[8], rs2[6], rs1[6], func3[4], rd[6], opcode[8],
imm[21] = {0};
      |                                                     ^~~~~~
proc.c:36:46: warning: unused variable 'rd' [-Wunused-variable]
    36 |       char func7[8], rs2[6], rs1[6], func3[4], rd[6], opcode[8],
imm[21] = {0};
```

```
      |                                                     ^~
proc.c:36:36: warning: unused variable ‘func3’ [-Wunused-variable]
    36 |       char func7[8], rs2[6], rs1[6], func3[4], rd[6], opcode[8],
imm[21] = {0};
      |                                        ^~~~~
proc.c:36:28: warning: unused variable ‘rs1’ [-Wunused-variable]
    36 |       char func7[8], rs2[6], rs1[6], func3[4], rd[6], opcode[8],
imm[21] = {0};
      |                               ^~~
proc.c:36:20: warning: unused variable ‘rs2’ [-Wunused-variable]
    36 |       char func7[8], rs2[6], rs1[6], func3[4], rd[6], opcode[8],
imm[21] = {0};
      |                     ^~~
proc.c:36:10: warning: unused variable ‘func7’ [-Wunused-variable]
    36 |       char func7[8], rs2[6], rs1[6], func3[4], rd[6], opcode[8],
imm[21] = {0};
      |          ^~~~~
proc.c:42:12: warning: ‘inst’ is used uninitialized [-Wuninitialized]
   42 |     return inst;
      |            ^~~~
proc.c:38:19: note: ‘inst’ declared here
   38 |     struct inst_t inst;
      |                   ^~~~
gcc -g -Wall -std=gnu99 -O3 -c mem.c -o build/mem.o
gcc -g -Wall -std=gnu99 -O3 -c loader.c -o build/loader.o
loader.c: In function ‘load_program’:
loader.c:44:9: warning: unused variable ‘data_size’ [-Wunused-variable]
   44 |     int data_size = 0;
      |         ^~~~~~~~~
loader.c:39:9: warning: unused variable ‘i’ [-Wunused-variable]
   39 |     int i = 0;
      |         ^
gcc  -o  kece343-comm307-project2  build/main.o  build/util.o  build/proc.o
build/mem.o build/loader.o
```

Step 3: You can find the executable binary (kece343-comm307-project2)

```
$ ls
build   kece343-comm307-project2   loader.c   loader.h   main.c   Makefile
mem.c   mem.h   proc.c   proc.h   README.md   sample_input   sample_output
util.c  util.h
```

4.2 Testing the assembler we built

Step 1: Our simulator requires a RISC-V binary file

```
$ ./kece343-comm307-project2
Usage: ./kece343-comm307-project2 [-d] [-n num_instr] RISC_V_binary
```

The skeleton code handles the two options.

- `-d`: Print the register file content for each instruction execution.
- `-n`: Number of instructions simulated

The default output is the PC and register file content after the completion of the given number of instructions. If `-d` option is set, the register must be printed for every instruction execution.

Step 2: Running the assembler with the provided sample inputs

```
$ ls sample_input/
example0.o  example0.s  example1.o  example1.s  example2.o  example2.s
example3.o  example3.s  example4.o  example4.s  example5.o  example5.s
example6.o  example6.s  example7.o  example7.s


# Run the simulator without any options
$./kece343-comm307-project2 sample_input/example1.o
[INFO] Simulating for 100 cycles...

[INFO] Current register values :
-------------------------------------
PC: 0x00400020
Registers:
x0: 0x00000000
x1: 0x00000000
x2: 0x00000000
x3: 0x00000000
x4: 0x00000000
x5: 0x00000000
x6: 0x00000000
x7: 0x00000000
x8: 0x10000000
x9: 0x10000004
x10: 0x00000000
```

```
x11: 0x00000017
x12: 0x00000000
x13: 0x00000000
x14: 0x00000000
x15: 0x00000000
x16: 0x00000000
x17: 0x00000017
x18: 0x00000000
x19: 0x00000000
x20: 0x00000000
x21: 0x00000000
x22: 0x00000000
x23: 0x00000000
x24: 0x00000000
x25: 0x00000000
x26: 0x00000000
x27: 0x00000000
x28: 0x00000000
x29: 0x00000000
x30: 0x00000000
x31: 0x00000000

[INFO] Memory content [0x10000000..0x1000001f] :
-------------------------------------
0x10000000: 0x00000064
0x10000004: 0x000000c8
0x10000008: 0x12345678
0x1000000c: 0x00000000
0x10000010: 0x00000000
0x10000014: 0x00000000
0x10000018: 0x00000000
0x1000001c: 0x00000000
```

If you type the command line as above, the output file shows PC, register values, and memory region for data. Note that the above result should be shown after you complete the project. In the first run without writing the code by yourself, every register values and memory region will be filled with zeros. The functions for printing the memory and register values are provided in the `proc.c`, and `mem.c` files.

4.3 Comparing the generated output with the reference output

```
$ ./kece343-comm307-project2 sample_input/example1.o | diff -Naur
sample_output/example1 -
--- sample_output/example1      2025-05-07 14:19:03.556639340 +0000
+++ -   2025-05-07 14:24:48.102035266 +0000
@@ -1,8 +1,9 @@
 [INFO] Simulating for 100 cycles...
+[INFO] Simulator halted

 [INFO] Current register values :
 -------------------------------------
```

```
-PC: 0x00400020
+PC: 0x0040006c
 Registers:
 x0: 0x00000000
 x1: 0x00000000
@@ -12,16 +13,16 @@
 x5: 0x00000000
 x6: 0x00000000
 x7: 0x00000000
-x8: 0x10000000
-x9: 0x10000004
+x8: 0x00000000
+x9: 0x00000000
 x10: 0x00000000
-x11: 0x00000017
+x11: 0x00000000
 x12: 0x00000000
 x13: 0x00000000
 x14: 0x00000000
 x15: 0x00000000
 x16: 0x00000000
-x17: 0x00000017
+x17: 0x00000000
 x18: 0x00000000
 x19: 0x00000000
 x20: 0x00000000
@@ -39,9 +40,9 @@

 [INFO] Memory content [0x10000000..0x1000001f] :
 -----------------------------------
-0x10000000: 0x00000064
-0x10000004: 0x000000c8
-0x10000008: 0x12345678
+0x10000000: 0x00000000
+0x10000004: 0x00000000
+0x10000008: 0x00000000
 0x1000000c: 0x00000000
 0x10000010: 0x00000000
 0x10000014: 0x00000000


# The below command will generate the same output as above
$ make test_1
```

To figure out which registers are different from the reference output, we use the `diff` utility or you can use `Makefile` to test individual samples. Since the skeleton code does not generate the correct output, it prints all the lines that are different. If you implement the code as intended, the `diff` utility will print nothing like the below. It means that your output file is the same as the reference output in the `sample_output` directory.

```
$ ./kece343-comm307-project2 sample_input/example1.o | diff -Naur
sample_output/example1 -
$
```

5. Grading Policy

Grades will be given based on the 8 test cases project provided in the `sample_input` directory. Your simulator should print the exact same output as the files in the `sample_output` directory.

We will be automating the grading procedure by seeing if there are any differences between the files in the `sample_output` directory and the result of your simulator executions. Please make sure that your outputs are identical to the files in the `sample_output` directory.

You are encouraged to use the `diff` command to compare your outputs to the provided outputs. If there are any differences (including whitespaces), the diff program will print the different lines. If there are no differences, nothing will be printed as mentioned before. Furthermore, we have provided a simple checking mechanism in the `Makefile`. Executing the following command will automate the checking procedure.

```
$ make test
```

There are 8 test codes to be graded, and you will be granted **10 points** for each correct binary code, and being **`Correct`** means that every digit and location is the same as the given output of the example. If a digit is not the same, you will receive **0 points** for the example.

6. Submission

Before submitting your code to `gradescope`, it is highly recommended to complete the work on your local Linux system environment. In this project, you just need to upload the `loader.c` and `proc.c` files to `gradescope` (**Do not modify file name**). After then, you should check that your code works like your local environment. Of course, you can test your code on `gradescope` as many as you want. Please make sure that you can see the same result on the submission site as well.

In this assignment, you are supposed to submit a 1-page document that should include the followings:

1) Explain the structure of the provided skeleton and how it works as per your understanding
2) Describe how you implemented the `decode` and `execute` codes
3) What you learned from this assignment
4) (Optional) Suggest anything to improve this assignment if you have

- You do not need to include the cover page. Please just put your name and student ID
- Do not include any code or screenshots in the document

- Your document should be the PDF format. If you turn in doc or hwp format, you will get 0 points for this document.

7. Updates/Announcements

If there are any updates to the project, including additional inputs or outputs, or changes, we will post a notice on the LMS, and will send you an e-mail using the LMS. Please check the notice or your e-mail for any updates.

8. Misc

We will accept your late submissions, but your score will lose up to 50%. Please do not give up the project.

Be aware of plagiarism! Although it is encouraged to discuss with others and refer to extra materials, **copying other students or opening code publicly is strictly banned**. The TAs will compare your source code with other team's code. If you are caught, you will receive a penalty for plagiarism.

Last semester, we found a couple of plagiarism cases through an automated tool. Please do not try to cheat TAs. If you have any requests or questions regarding administrative issues (such as late submission due to an unfortunate accident, `gradescope` is not working), please send an e-mail to the TA (sgjeon@csl.korea.ac.kr).