PA #1: Implementing an RISC-V (RV32I) Assembler

Due 11:59 PM, April 15th

1. Overview

This project aims to implement an RISC-V (subset) ISA assembler. The assembler is the tool that converts assembly codes to a binary file. The goal of this project is to help you understand the RISC-V ISA instruction set and become familiar with the principles of assemblers.

Our assembler is a simplified version that does not support the linking process, and thus you do not need to add the symbol and relocation tables for each file. In this project, only one assembled file will be the whole program.

You should implement the assembler, which can convert a subset of the instruction set shown in the following table. In addition, your assembler must handle labels for jump/branch targets and labels for the static data section.

2. Instruction Set

The detailed information regarding instructions is on the attached RISC-V green sheet page.

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

- Instructions for signed (`add`, `sub`) and unsigned operations (`sltiu`, `sltu`, `slli`, `srli`) need to be implemented.
- Only loads and stores with 4B word need to be implemented.
- The assembler must support decimal and hexadecimal numbers (`0x`) for the immediate field, and `.data` section.
- The name of registers is always "`x<n>`", n is from 0 to 31.
- `la` (load address) is a pseudo instruction; it should be converted to one or two assembly instructions. The example is shown below.

```
la x<n>, VAR1 # VAR1 is a label in the data section. It should be decomposed into two instructions, auipc and addi, to represent its 32 bit address.

auipc x<n>, D[31:12] # Load the upper 20 bits of D (where D = VAR1 - PC)
addi x<n>, x<n>, D[11:0] # Add the lower 12 bits of D to obtain the full address of VAR1
```

Case-1) When `D` is `0x0FCFFFFC`, we need `auipc` and `addi` instructions.
You need to add an additional constant '`0x800`' to get the proper result. Check **Section 7. Updates/Announcements** for more detailed information.

```
auipc x2, 0x0FD00  # hi20 = (0x0FCFFFFC+0x00000800) >> 12
addi x2, x2, 0xFFC # lo12 = D - (hi20 << 12)
```

Case-2) When `D` is `0x0FC00000`, the lower 12-bit address is `0x0`. In such a case, we can omit the `addi` instruction like the below.

```
auipc x2, 0x0FC00
```

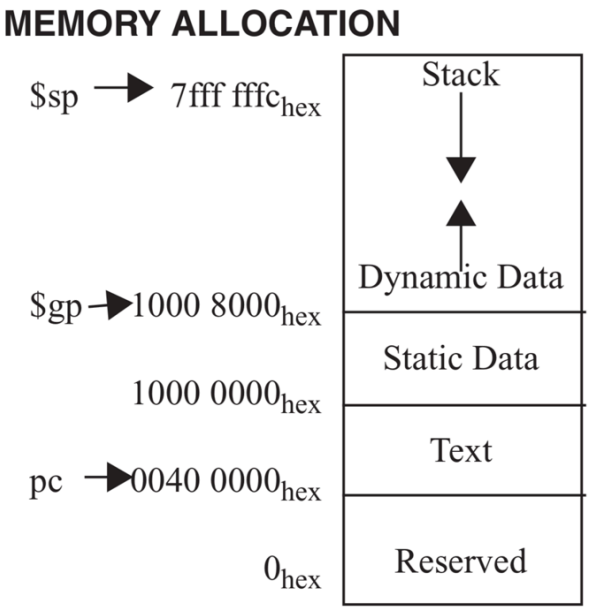
2.1 Directives

`.text`
- indicates that the following items are stored in the user text segment, typically instructions
- It always starts from `0x00400000`

`.data`
- indicates that the following data items are stored in the data segment
- It always starts from `0x10000000`

`.word`
- store n 32-bit quantities in successive memory words

You can assume that the `.data` and `.text` directives appear only once, and the `.data` must appear before the `.text` directive. Assume that each word in the data section is initialized (Each word has an initial value). In the following figure, we illustrate the memory map used in our projects.


MEMORY ALLOCATION
$sp
7fff fffc hex
Stack
Dynamic Data
$gp
1000 8000 hex
Static Data
1000 0000 hex
Text
pc
0040 0000 hex
Reserved
0 hex

2.2 Input format

```
    .data
array:  .word   3
    .word   123
    .word   4346
array2: .word   0x11111111
    .text
    .globl main
main:
    addi    x2, x0, 1024
    add     x3, x2, x2
    or      x4, x3, x2
    slli    x6, x5, 16
    lui     x28, 2
    addi    x28, x28, 0x70F
    add     x7, x6, t0
    sub     x8, x7, x2
    or      x9, x4, x3
    xori    x9, x9, -1
    ori     x10, x2, 255
    srli    x11, x6, 5
    la      x4, array2
    and     x13, x11, x5
    andi    x14, x4, 100
    lui     x17, 0x640
    addi    x2, x0, 0xa
```

Here is one of the input files we will use. As mentioned in Section 2.1, each input file consists of two sections, `data,` and `text`. In this example, `array` and `array2` are data.

2.3 Output format

The output of the assembler is an object file. We use a simplified custom format.

- The first two words (32bits) are the size of the `text` and `data` section.
- The next bytes are the instructions in binary. The length must be equal to the specified `text` section length.
- After the `text` section, the rest of the bytes are the initial value of each data in the `data` section.

The following must be the final binary format:

```
<text section size>
<data section size>
<instruction 1>
…
<instruction n>
<initial value of each data in the data section>
```

3. Getting the Skeleton Code

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-project1

Be sure to read the **README.md** file for some useful information. It includes an explanation of each file and which files you are allowed to modify for this project.

If you do not want to use the skeleton code, it is allowed to write code from scratch. However, you are supposed to follow the input and output file format because the grading script works on the provided `sample_input` and `sample_output` files described in the following section.

4. Building and Running the Assembler

As in the previous assignment, 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 http://github.com/sys-ku/kece343-comm307-project1.git
$ ls
kece343-comm307-project1

# TIPS: Try to use the Tab key when navigating directories or files
$ cd kece343-comm307-project1
```

Step 2: Trying to build the skeleton codes

```
$ ls
assembler.c  assembler.h  handout  main.c  Makefile 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 -o assembler.o assembler.c
assembler.c: In function 'record_text_section':
assembler.c:143:13:   warning:   variable   'rs'   set   but   not   used
[-Wunused-but-set-variable]
  143 |         int rs2, rs1, rd, imm;
      |             ^~
assembler.c:142:16: warning: unused variable 'idx' [-Wunused-variable]
  142 |         int i, idx = 0;
```

```
      |                ^~~
assembler.c:142:13: warning: unused variable ‘i’ [-Wunused-variable]
  142 |         int i, idx = 0;
      |             ^
assembler.c:140:14: warning: unused variable ‘label’ [-Wunused-variable]
  140 |         char label[32] = {0};
      |              ^~~~~
assembler.c:139:14: warning: unused variable ‘op’ [-Wunused-variable]
  139 |         char op[32] = {0};
      |              ^~
assembler.c:138:14: warning: unused variable ‘inst’ [-Wunused-variable]
  138 |         char inst[0x1000] = {0};
      |              ^~~~
gcc -o assembler assembler.o
```

Step 3: You can find a new directory (`./build`) containing all the object files, as well as the executable binary (`assembler`)

```
$ ls
assembler   assembler.c   assembler.h   build   handout   main.c   Makefile
README.md  sample_input  sample_output  util.c  util.h
```

4.2 Testing the assembler we built

Step 1: Our assembler requires an input file containing assembly codes

```
$ ./assembler
Usage: ./assembler <*.s>
Example: ./assembler sample_input/example?.s
```

Step 2: Running the assembler with the provided sample inputs

```
$ ls sample_input/
example0.s  example1.s  example2_mod.s  example3.s  example4.s  example5.s
example6.s  example7.s

$ ./assembler sample_input/example0.s
```

Step 3: Checking the output file

```
$ ls sample_input/
example0.s  example0.o  example1.s   example2_mod.s  example3.s  example4.s
      example5.s  example6.s  example7.s
```

The output file (in this case `example0.o`) is generated in the same directory where the input file is located.

4.3 Comparing the generated output with the reference output

```
$ diff -Naur sample_input/example0.o sample_output/example0.o
--- sample_input/example0.o     2022-09-27 18:06:07.246969199 +0900
+++ sample_output/example0.o    2022-09-27 18:05:27.874952448 +0900
@@ -0,0 +1,7 @@
+00000000000000000000000000010000
+00000000000000000000000000000100
+00111100000001010001000000000000
+00111100000001100001000000000000
+00000000101001100101000000100000
+00000000101001100101100000100010
+00000000000000000000000001100100

# The below command will generate the same output as above
$ make test_0
```

To figure out which lines of generated binary code 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 below. It means that your output file is the same as the reference output in the `sample_output` directory.

```
$ diff -Naur sample_input/example0.o sample_output/example0.o
$
```

4.4 Tips

While working on this assignment, you may need to have a debugger that is able to step through each line of your code. The most powerful tool for debugging is to use `printf` wherever you want to dig into it. That is a good starting point, but we highly recommend you be familiar with GDB which reduces the burden of debugging. Please refer to Section 6.2.2 GDB in the handout of Project-0.

Also, it would be good to design your own simple test cases while debugging your code. For instance, if you want to test your code for R format instructions, you can just write a few lines of input assembly file that excludes I and J formation instructions. This is kind of a test-driven development.

5. Grading Policy

Grades will be given based on the examples provided for this project in the `sample_input` directory. Your assembler 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. 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 code segments to be graded (last three cases will be hidden in `gradescope`) 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 a **0 score** 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 used in project0. In this project, you just need to upload the `assembler.c` file 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 symbol table (i.e., `make_symsbol_table()`)
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.

- **25.05.07: Update to Address‑Calculation Procedure (AUIPC + ADDI)**
    - When splitting a 32-bit offset into the 20-bit “hi20” immediate (for AUIPC) and the 12-bit signed “lo12” immediate (for ADDI), we add 0x800 (half of a 4 KiB page) before shifting right by 12 bits. This guarantees that
        1) Proper rounding occurs when the lower 12 bits of the offset are ≥ 0x800, so the page boundary in hi20 << 12 is correctly advanced by one.
        2) The resulting lo12 = offset – (hi20 << 12) always fits within the signed 12-bit range (–2048…+2047).
        3) We avoid overflow or out-of-range immediates in the ADDI instruction, ensuring the final reconstructed address equals the original offset.
    - You can check related commit history in our github project. (https://github.com/Sys-KU/kece343-comm307-project1/commit/de5bc7deaf9434bd627bb5f497fa88d46e478dd8)

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 TAs (yjcheon@csl.korea.ac.kr).