Project 2: RISC-V Assembler

EEL 4768 Computer Architecture: Spring 2026

Project Introduction

In this project phase, we will continue to build on your assembly experience by designing an assembler which processes RISC-V assembly code and converts it into its respective binary representation. Similar to Project 1, you are expected to have prior experience with writing and debugging assembly languages.

Write the assembler using the 32-bit RISC-V ISA, ***only using instructions found in the provided RV32I reference card posted in webcourses***. Using any instructions outside of the base RV32I instruction set will result in a zero for the specific problem. This assignment should be completed by yourself. You may discuss the problems with other students but must work on your own answers.

You are permitted to use generative artificial intelligence (e.g. ChatGPT); if you choose to do so, indicate where you use AI in your submission, and why you used it. Please ensure your usage of LLMs adheres to the course policies.

Environment setup

Your assembler can be writing in one of two languages of your choice.

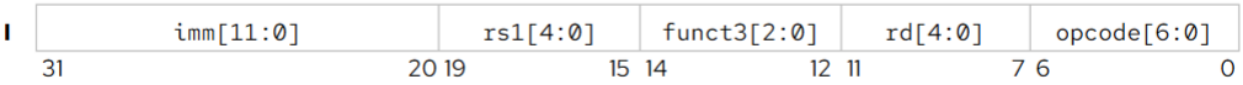
- C++
- Python

Assembler instructions

A sample assembly file (.s) will be provided. Your job is to convert those instructions into its binary representation. For example, take the instruction

```
addi t0, x0, 5
```

This is an immediate arithmetic instruction and thus follows the I type instruction.



This breaks the instruction down into

- The immediate value 5
- The source register x0
- The function add immediate
- The destination register t0
- The opcode which selects the immediate instruction type

If we break this down into each's bit representation, you would get

Bitfield	Hex	Binary
imm[11:0]	0x005	0000 0000 0101
rs1[4:0]	0x00	0000 0
funct3[2:0]	0x0	000
rd[4:0]	0x05	0010 1
opcode[6:0]	0x13	001 0011
Full instruction	0x00500293	0000 0000 0101 0000 0000 0010 1001 0011

Your program should automate this process for each instruction type (R, I, S, B, U, J) included in the RISC-V reference card. It should also be capable of handling memory initialized after .data, and labels, as well as .text and .global directives.

.global "label" is used to select the first label the file starts at. For example, .global main would start at the main label.

The output of your program should be two files. One .bin file, which is the binary file, and one .hex.txt file of the hex equivalent output.

Your program should take in a base path and the name of the assembly file, and should be named assembler.py or assembler.cpp. For example,

Python:

```
python assembler.py /autograder/source/ add_shift.s
```

C++:

```
g++ -std=c++17 -o /autograder/source/assembler /autograder/source/
assembler.cpp
```

```
/autograder/source/assembler /autograder/source/
/autograder/source/add_shift.s
```

The output of the program should match the name of the input .s file. For example, if the input file is add_shift.s, then the output should be ***add_shift.bin*** and ***add_shift.hex.txt***.

If your assembler does not match the format of these commands, the autograder will not be able to grade it. Please reach out if you have any confusion about this.

An add_shift.s file will be supplied to be used to test your program and is the first test of the autograder. Example output should match what is given below.

Example output

add_shift.bin

00000000010100000000001010010011

00010001000101000010011001100011

00010011001110010011001000100010

add_shift.hex.txt

0x00500293

0x001028AB

0x0312582A

Submission

Submit your program directly to the autograder. You can submit multiple files, but none of them can be nested in subfolders or subdirectories. Name the main file ***assembler.py or assembler.cpp***.