

Question 1.

Implement an RISC-V-LC Assembler.

Answer:

In this lab, we are going to complete some of the instructions in the `asm.c` file. There are few main points to mark and it is going to be stated in this lab report. In the following part, I am going to demonstrate how I complete this lab.

For integer R-I Instructions, they have the same pattern as `ADDI` instruction, which is I-Type instruction. So basically I can just modify the `ADDI` instruction to add corresponding `funct3`(bit12-14) to the binary output and change some of the argument parsing of `arg3` for the shift functions like `SLLI`, `SRLI`, `SRAI` instructions. Here is one of my implementations of assembling integer R-I Instructions instruction, this case using `SRAI` instruction.

```
1 //other cases
2 else if (is_opcode(opcode) == SRAI) {
3     binary = (0x04 << 2) + 0x03;
4     binary += (reg_to_num(arg1, line_no) << 7);
5     binary += (0x05 << 12);
6     binary += (reg_to_num(arg2, line_no) << 15);
7     binary += (lower5bit(arg3, line_no) << 20);
8     binary += (0x010 << 26);
9 }
10 //continue on other cases
```

Line 9 adds the `funct3` to the binary, line 11 extract the lower 5 bits of the immediate which stands for the number of bits to be shifted, and line 11 is adding value to binary to define the `funct7` of the corresponding instruction.

For Integer R-R operations, they have the same pattern as `ADD` instruction, which is R-Type instruction. So basically I can just modify the `ADD` instruction to add corresponding `funct3`(bit12-14) and `funct7`(bit25-31) to the binary output. Here is one of my implementations of assembling integer R-R Instructions instruction, this case using `SRA` instruction.

```
1 //other cases
2 else if (is_opcode(opcode) == SRA) {
3     /* Lab2-1 assignment */
4     binary = (0x0C << 2) + 0x03;
5     binary += (reg_to_num(arg1, line_no) << 7);
6     binary += (0x05 << 12);
7     binary += (reg_to_num(arg2, line_no) << 15);
8     binary += (reg_to_num(arg3, line_no) << 20);
9     binary += (0x020 << 25);
10 }
11 //continue other cases
```

Line 6 add corresponding `funct3` to the binary, line 7 and 8 handle corresponding register arguments, line 9 add corresponding `funct7` to the binary.

For JALR instruction, it uses I-Type format. So the assembling format is similar to ADDI instruction. Since JALR have only 2 arguments, and the arg2 contains both the offset immediate and the register, we have to use the function `parse_regs_indirect_addr()` to separate the arg2. Then put the corresponding part into the the binary.

```
1 //other cases
2 else if (is_opcode(opcode) == JALR) {
3     binary = (0x019 << 2) + 0x03;
4     binary += (reg_to_num(arg1, line_no) << 7);
5     binary += (reg_to_num(parse_regs_indirect_addr(arg2, line_no)->reg,
6         line_no) << 15);
7     binary += ((parse_regs_indirect_addr(arg2, line_no)->imm) << 20);
8 }
9 //continue other cases
```

For JAL instruction, it uses special J-Type format. So the assembling logic is different to JALR instruction.

```
1 //other cases
2 else if (is_opcode(opcode) == JAL) {
3     binary = (0x01b << 2) + 0x03;
4     binary += (reg_to_num(arg1, line_no) << 7);
5     int val = handle_label_or_imm(line_no, arg2, label_table,
6         number_of_labels);
7     int offset = val - addr;
8     binary += ((offset & 0x100000) << (31 - 20)); //bit20
9     binary += ((offset & 0x7FE) << 20); //bit10:1
10    binary += ((offset & 0b100000000000) << (20 - 11)); //bit11
11    binary += ((offset & 0xFF000)); //bit19:12
12 }
13 //continue other cases
```

Since JAL have special implementation in the 20-bit jump immediate, masking and careful shifting is needed to put the immediate in the correct position.

bit20 of the immediate only need to shift (31 - 20) in order to be in the bit31 of the instruction.

bit10:1 is shifted to left 20 in order to align the first bit to bit21.

bit11 is masked separately then shifted by (20-11)bit to place at bit20.

bit19:12 are of the same position of the insturction so no shifting is needed.

For conditional branches, they are in B-Type format, which requires careful masking and shifting as well. Since example is given, the parsing and shifting of immediate is done for us so all we need is just to modify the implementation and add the funct3 to the instructions to differ them from each other.

For Load Instructions, they implement I-Type formatting. However, they only take 2 arguments so we have to separate the offset immediate and the register from arg2 by calling the `parse_regs_indirect_addr()` and carefully put the register into position of rd and immediate respectively. Since example is given for the LB instruction, so all we need to do is to add the funct3 to the binary that differs from each other branch fuctions.

For Store Instructions, they implement specail S-Type formatting. They only take 2 arguments so we have to separate the offset immediate and the register from arg2 by calling the `parse_regs_indirect_addr()` and put them into correct position. And since they have similar structure as branch instructions execpt the immediate part, we only need to modify the position of immediate.

The following are the screenshots of the lab results.

Figure 1: make validate result

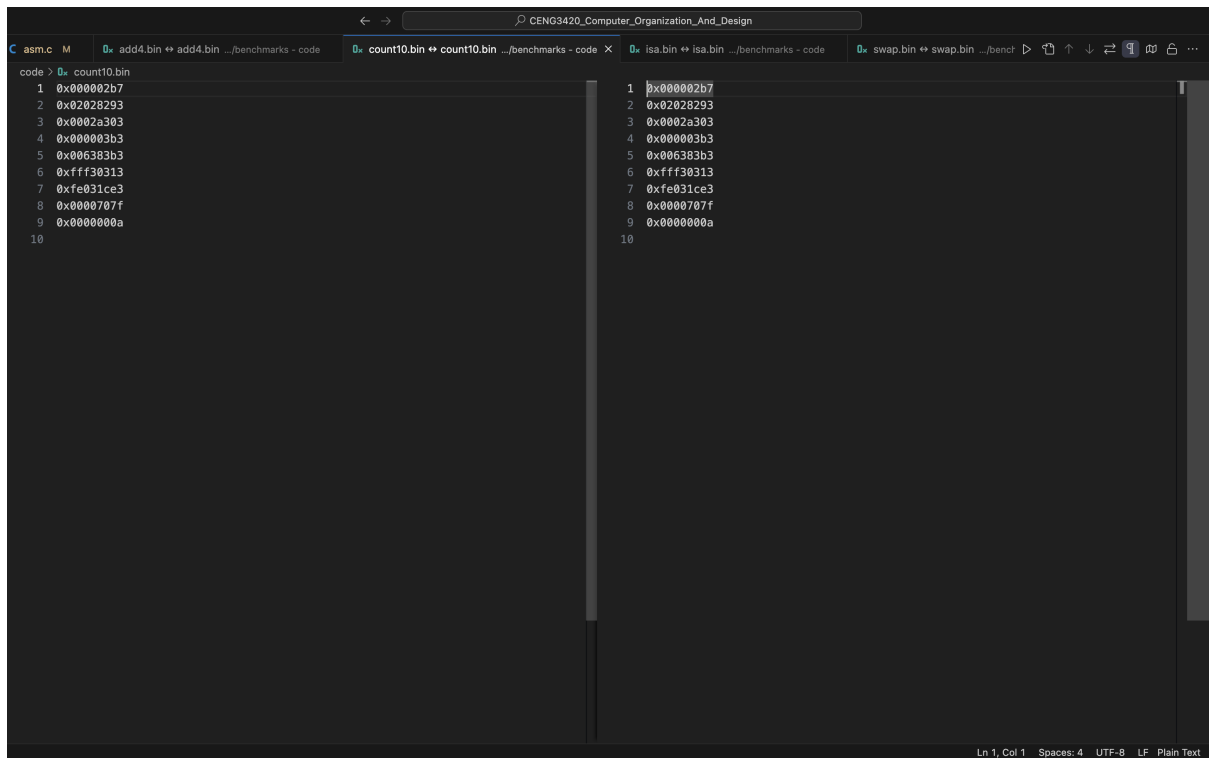
```
>  
make validate  
bash tools/validate.sh  
tools/./benchmarks/isa.asm  
[INFO]: Processing input file: tools/./benchmarks/isa.asm  
[INFO]: Writing result to output file: isa.bin  
tools/./benchmarks/swap.asm  
[INFO]: Processing input file: tools/./benchmarks/swap.asm  
[INFO]: Writing result to output file: swap.bin  
tools/./benchmarks/add4.asm  
[INFO]: Processing input file: tools/./benchmarks/add4.asm  
[INFO]: Writing result to output file: add4.bin  
tools/./benchmarks/count10.asm  
[INFO]: Processing input file: tools/./benchmarks/count10.asm  
[INFO]: Writing result to output file: count10.bin  
[INFO]: You have passed the Lab.  
~/.Documents/CUHK/Year_2/Spring/CENG3420_Computer_Organization_And_Design/code > on P lab2.1 +1 19 at 23:43:53
```

The following screenshots are the comparison of suggested answer(Left) and my output(Right). No highlighting means no different.

Figure 2: add4.bin

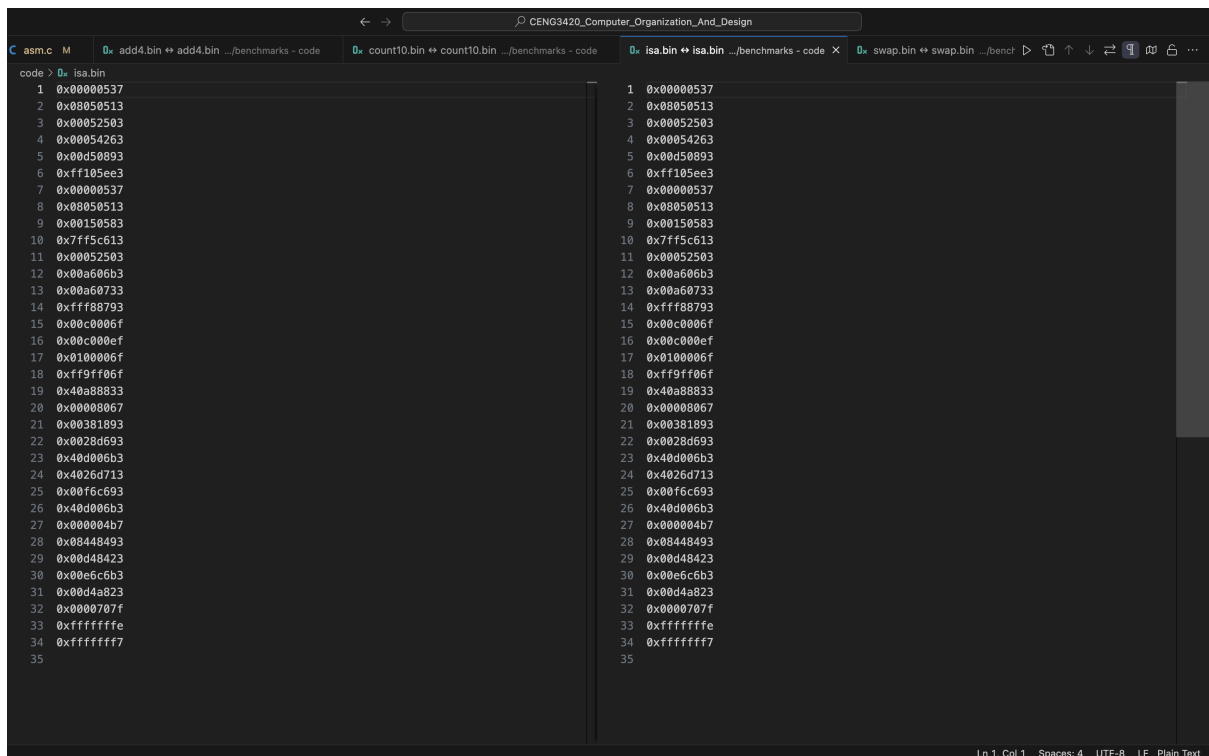
code > 0x add4.bin	0x add4.bin ↔ add4.bin .../benchmarks - code	0x count10.bin ↔ count10.bin .../benchmarks - code	0x isa.bin ↔ isa.bin .../benchmarks - code	0x swap.bin ↔ swap.bin .../benchmarks - code
1 0x000005b7	1 0x000005b7	1 0x000005b7	1 0x000005b7	1 0x000005b7
2 0x03458593	2 0x03458593	2 0x03458593	2 0x03458593	2 0x03458593
3 0x0005a583	3 0x0005a583	3 0x0005a583	3 0x0005a583	3 0x0005a583
4 0x00000637	4 0x00000637	4 0x00000637	4 0x00000637	4 0x00000637
5 0x03860613	5 0x03860613	5 0x03860613	5 0x03860613	5 0x03860613
6 0x00062603	6 0x00062603	6 0x00062603	6 0x00062603	6 0x00062603
7 0xffff5893	7 0xffff5893	7 0xffff5893	7 0xffff5893	7 0xffff5893
8 0x00160613	8 0x00160613	8 0x00160613	8 0x00160613	8 0x00160613
9 0xfe059ce3	9 0xfe059ce3	9 0xfe059ce3	9 0xfe059ce3	9 0xfe059ce3
10 0x000006b7	10 0x000006b7	10 0x000006b7	10 0x000006b7	10 0x000006b7
11 0x03860693	11 0x03860693	11 0x03860693	11 0x03860693	11 0x03860693
12 0x00c6a023	12 0x00c6a023	12 0x00c6a023	12 0x00c6a023	12 0x00c6a023
13 0x0000707f	13 0x0000707f	13 0x0000707f	13 0x0000707f	13 0x0000707f
14 0x00000004	14 0x00000004	14 0x00000004	14 0x00000004	14 0x00000004
15 0xffffffffb	15 0xffffffffb	15 0xffffffffb	15 0xffffffffb	15 0xffffffffb
16	16	16	16	16

Figure 3: count10.bin



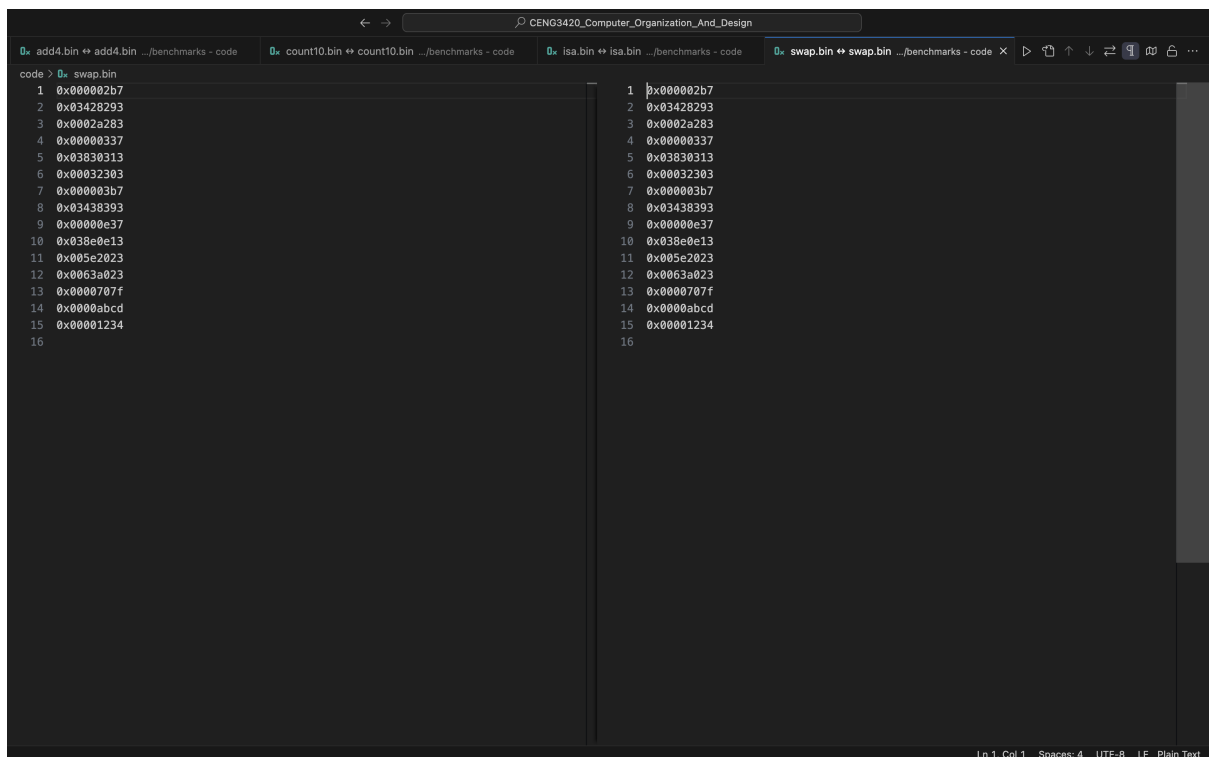
```
code > 0x count10.bin
1 0x000002b7
2 0x02020293
3 0x0002a303
4 0x000003b3
5 0x006383b3
6 0xffff30313
7 0xfe031ce3
8 0x0000707f
9 0x0000000a
10
```

Figure 4: isa.bin



```
code > 0x isa.bin
1 0x00000537
2 0x00050513
3 0x00052503
4 0x00054263
5 0x00d50093
6 0xff105ee3
7 0x00000537
8 0x00050513
9 0x00150583
10 0x7ff5c613
11 0x00052503
12 0x00a606b3
13 0x00a60733
14 0xffff88793
15 0x00c0006f
16 0x00c000ef
17 0x0100006f
18 0xff9ff06f
19 0x40a88833
20 0x00000067
21 0x00381893
22 0x0028d693
23 0x40d006b3
24 0x4026d713
25 0x00f6c693
26 0x40d006b3
27 0x000004b7
28 0x00448493
29 0x00d48423
30 0x00e6c6b3
31 0x00d4a823
32 0x0000707f
33 0xfffffffffe
34 0xfffffffff7
35
```

Figure 5: swap.bin



```
code > 0x swap.bin
1 0x000002b7
2 0x03428293
3 0x0002a283
4 0x00000337
5 0x03830313
6 0x00032303
7 0x000003b7
8 0x03438393
9 0x00000e37
10 0x038e0e13
11 0x005e2023
12 0x0063a023
13 0x0000707f
14 0x0000abcd
15 0x00001234
16

1 0x000002b7
2 0x03428293
3 0x0002a283
4 0x00000337
5 0x03830313
6 0x00032303
7 0x000003b7
8 0x03438393
9 0x00000e37
10 0x038e0e13
11 0x005e2023
12 0x0063a023
13 0x0000707f
14 0x0000abcd
15 0x00001234
16
```