Name: Yu Ching Hei
SID: 1155193237
email: chyu2@cse.cuhk.edu.hk

Question 1.

Implement an RISCV-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.

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

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

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

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

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

Figure 2: add4.bin

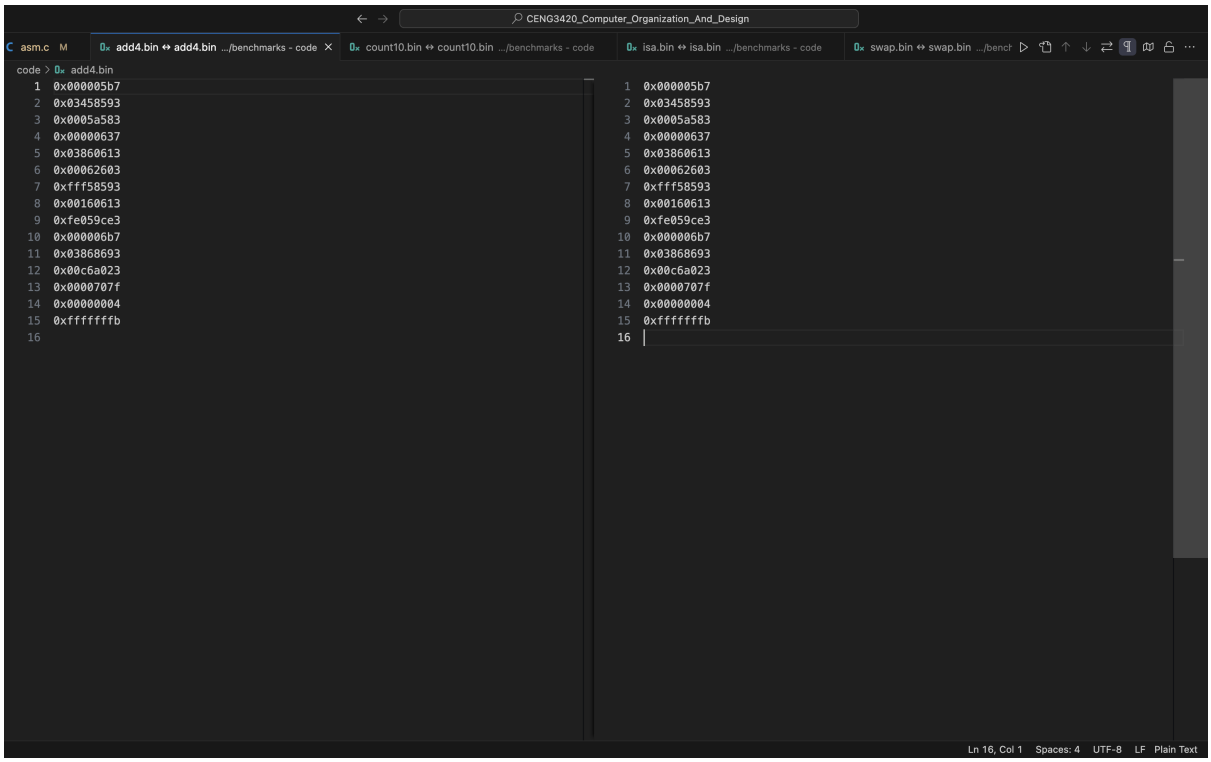

Figure 3: count10.bin

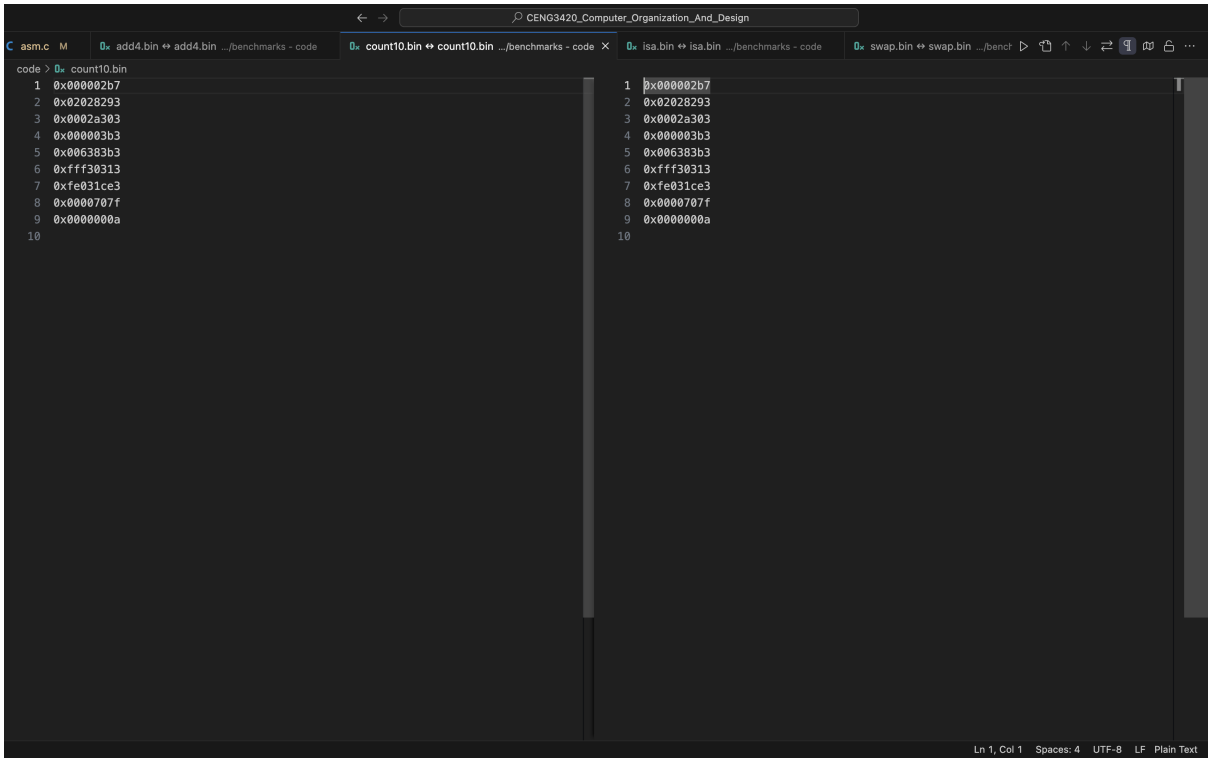

Figure 4: isa.bin

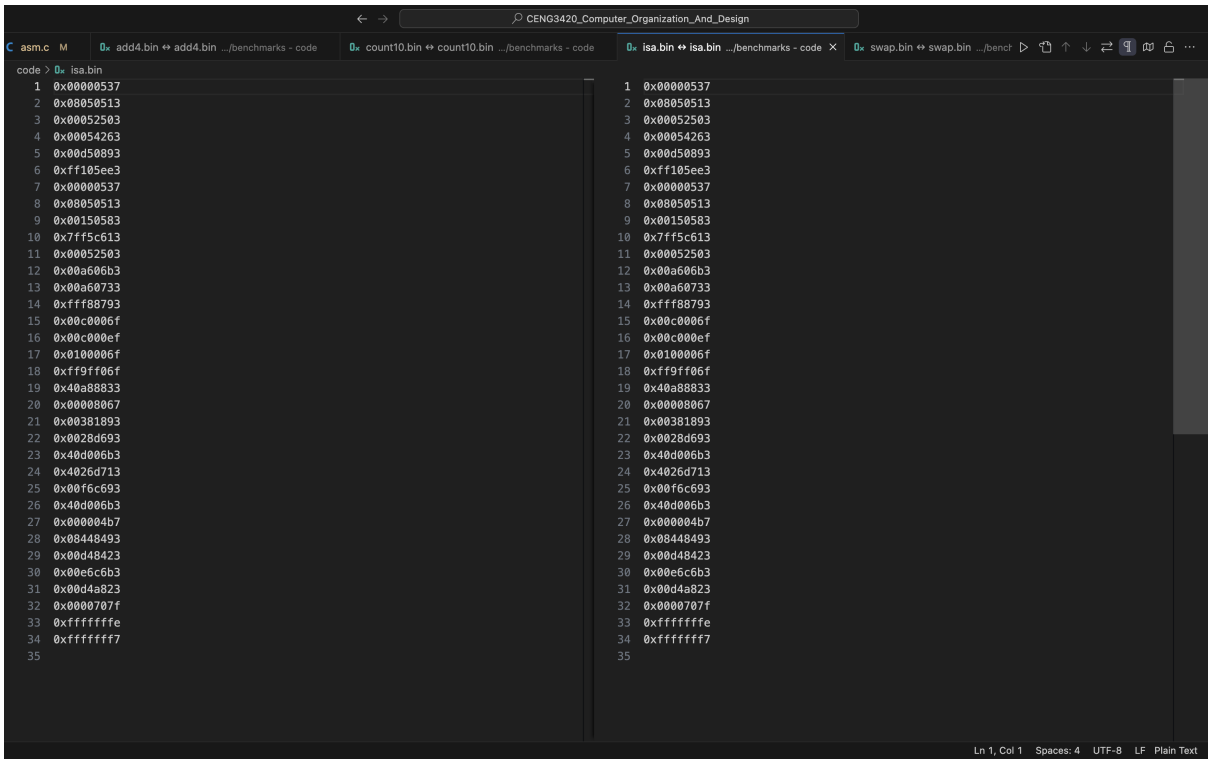

Figure 5: swap.bin