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

Question 1.

Implement an RISCV-LC ISA Simulator.

Answer:

In this lab, we are going to complete some of the instructions in the sim.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.

Let's say we have to complete the `handle_slli` function: Since we know that it is a I-type instructions, we can refer to the sample implementation of the `handle_addi` function

```
1 void handle_addi(unsigned int cur_inst) {
2     unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);
3     int imm12 = sext(MASK31_20(cur_inst), 12);
4     NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] + imm12;
5 }
```

since the parsing is the same for all integer R-I instructions, but only differs by the logic, I just have to modify line 4 by changing the "+" operator to other operators like "≪" for `slli`, "∧" for `ori` etc. Basically most integer R-I instructions like `slli, xori, srli, ori, andi, lui` can be implemented in the same way except `srai`.

Since the result of `srai` instruction is MSB-extended, we have to put the result into the `sext()` fucntion before storing it into the rd.

```
NEXT_LATCHES.REGS[rd] = sext(CURRENT_LATCHES.REGS[rs1] >> imm5, 32);
```

Then we have the integer R-R type instructions. These instructions implement R-Type format. So we are refering to the R-Type instruction example

```
1 void handle_add(unsigned int cur_inst) {
2     unsigned int rd = MASK11_7(cur_inst),
3                  rs1 = MASK19_15(cur_inst),
4                  rs2 = MASK24_20(cur_inst);
5     NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1]
6                             + CURRENT_LATCHES.REGS[rs2];
7 }
```

To complete the R-Type instructions, it shares the same idea of I type, by changing the "+" operator in line 6 to other operators like "≪" for `sll`, "∧" for `or` etc. Same for R-R instructions, `sra` requires special handling as it has MSB-extended return value. so the last line becomes

```
NEXT_LATCHES.REGS[rd] = sext(CURRENT_LATCHES.REGS[rs1]
                        >> CURRENT_LATCHES.REGS[rs2], 32);
```

For unconditional jump `JAL`, it uses J-Type format which has the 20-bit jimm20 specially formatted into the form of $[20|10:1|11|19:12]$. Therefore, there is no simple masking but we have to mask the current instruction binary stream part by part. The construction of jump immediate is done as below:

```
int imm20 = (MASK19_12(cur_inst) << 12);
    imm20 += (MASK20(cur_inst) << 11);
    imm20 += (MASK30_21(cur_inst) << 1);
    imm20 += (MASK31(cur_inst) << 20);
```

then the PC + 4 of current state will be stored into the rd if provided and the sign-extended jump immediate is added to the current PC as a offset then stored to the PC of next state to let the FSM jump to the desire location in the next state

For unconditional jump `JALR`, is uses I-Type format so it shares the same parsing format as the I-Type as implemented in the previous part. But the main different is we are jumping to an address reletive to an address stored in register. Therefore, instead of adding the jump offest directly to the next state PC, the next state PC is set to the address stored in rs1 that is offset by the immediate value.

For conditional branches, there is a given example of beq, so we can just mimic the parsing logic done in beq and change the branching logic

```
1 void handle_beq(unsigned int cur_inst) {
2     unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);
3     int imm12 = (MASK31(cur_inst) << 12) + \
4             (MASK7(cur_inst) << 11) + \
5             (MASK30_25(cur_inst) << 5) + \
6             (MASK11_8(cur_inst) << 1);
7     if (CURRENT_LATCHES.REGS[rs1] == CURRENT_LATCHES.REGS[rs2])
8         NEXT_LATCHES.PC = (sext(imm12, 12) + CURRENT_LATCHES.PC);
9 }
```

we just have to modify the "+" in line 8 to implement different branching logic like "`!=`" for `bne`, ">=" for `bge` and "<" for `blt`.

For load instructions, they are in I-Type format so the parsing is the same as implemented in the previous part. By following the example of the `lb`, we can formulate the instructions to read the memory from the address that is offset by the immediate. After retrieving the value from the memory, I applied corresponding masking fucntion to the value so that the output will be of the desired length.

For store instructions, they have similar instruction parsing as the conditional branch instructions, so the main focus will be the storing mechanism. This is the implementation of my `handle_sw` fucntion.

```
void handle_sw(unsigned int cur_inst) {
    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);
    int imm12 = (MASK11_7(cur_inst));
    imm12 += (MASK31_25(cur_inst) << 5);
    int startAddr = CURRENT_LATCHES.REGS[rs1] + sext(imm12, 12);
    MEMORY[startAddr] = (MASK7_0(CURRENT_LATCHES.REGS[rs2]));
    MEMORY[startAddr + 1] = (MASK15_8(CURRENT_LATCHES.REGS[rs2]));
    MEMORY[startAddr + 2] = (MASK23_16(CURRENT_LATCHES.REGS[rs2]));
    MEMORY[startAddr + 3] = (MASK31_24(CURRENT_LATCHES.REGS[rs2]));
}
```

In my code, I have separated the data to be stored to the memory into 4 different sections and store them into the memory byte by byte separately. For word(32-bit), split into 4 chunks; for half(16-bit), split into 2 chunks.

The following are the screenshots of the corresponding similations.

Figure 1: add4.bin

`../figs/add4.png`

Figure 2: count10.bin

../figs/count10.png

Figure 3: isa.bin

../figs/isa.png

Figure 4: swap.bin

`../figs/swap.png`