Lab 2.2 Report

1. Problem analysis

   Lab 2.2 is the next episode of lab 2.1. In lab 2.1, we have simulated the assembly process that turns the assembly code into binary machine code. And in lab 2.2, we are going to keep making use of these machine codes and perform real instructions, like changing the registers and data memory, simulating the real instruction execution process.

2. My understanding of the instruction execution process

   By examining the *sim.c* and *sim.h* files, I have a basic understanding about the whole instruction execution process.

   (a) Terminologies and data structures

   (i) System latches

   In my understanding, system latches is the system of the combination of the 32 registers and the PC counter. It is just the place to store information in the processor. System latches is simulated by the structure struct_system_latches, where it has 2 members: integer PC that stores the value of the PC and integer array REGS that stores all the register value of the 32 registers.

   (ii) Special base addresses

   CODE_BASE_ADDR: the base address to locate source codes.

   TRAPVEC_BASE_ADDR: exceptions & interruptions' base address.

   This is one of the important things I learned from this lab: in practice, actually exceptions and interruptions are stored in their unified position in memory, and if the pc just to that address, the system will know there is an error.

   (iii) Memory

   The simulator uses C arrays to simulate the memory. It is an array named MEMORY that has limited total amount BYTES_IN_MEM and is of type unsigned char, which is of 1 byte that is the same with the real situation.

   (b) Important functions and MACROS

   (i) The MASK macros. In this simulating code, we need to frequently retrieve some specific bits of the instructions. In order to avoid tedious and unclear bit wise and operation everywhere, we can use a macro to clear the things. A typical MASK is like this:

   #define MASK6_0(x) ((x) & 0x7F)

   Where the return value is the input bitwise and a masking value, so all the bits in the masking area remain and all other bits are set to 0 due to the property of and logic.

   (c) Simulation process

   The simulator is actually a user-interact system. After running this file, we

are actually in a user interface where we can let the simulator execute the binary machine code and examine the registers and memories.

Here are the possible instructions:

```
go              -  run a program till the end
run n           -  execute a program for n instructions
mdump low high  -  dump memory from low address to high address
rdump           -  dump the register & bus values
?/h             -  display the help menu
quit            -  exit the simulator
```

(d) Program logic:

The program starts with the main function. In the main function, the program first get the binary file from the console and open it, then it starts to initialize like allocating space for memory, etc. Then it goes into a while loop in which the get_command function is executed and will get command from the user interface. It can show the values in specific memories or registers, and it can also execute instructions.

The execution process is handled by the run functions. It runs a specific numbers of instructions by calling the cycle() function a specific amount of times, where each time the cycle() function executes exactly one instruction.

In the cycle instruction, the status of instruction count and latches is updated, and each instruction is handled by the handle_instruction() function, which is what we should implement.

In handle_instruction() function, it first does some previous preparations like updating pc, then it uses a switch logic to identify which instruction the current cycle is by examining the opcode, func3, func7, etc. In lab 2.2 we should implement the details of handle_instruction() function, including the switch logic and the function for handling specific instructions.

3. Switch logic implementation

The instruction identifying logic (decoding) in handle_instruction() function is done by a nested branching. On the outmost side is the switch function to decide on the general types of instruction (I, halt, R, U, jal, halr, B, load and store) by its opcode. On the second stage of the decoding, a nested switch is implemented to identify the specific instructions by func3. And if we still need to identify specific instructions by the 25..31 bits, a if_else logic could also be implemented. In the given code, the I and halt type logic is already implemented, so we need to finish the rest.

(a) R type instructions

The opcode of R type instructions is 0110011, which can be done by shifting 0x0C left 2 bits and add 0x03. And this number is put into the outmost switch logic to identify this type of instruction.

After knowing the type of instructions, we still have to examine func3 using another nested switch to know which instructions it is. Possible cases are 0,

1, 4, 5, 6, 7, corresponding to add/sub, sll, xor, srl/sra, or and respectively.

For add/sub and srl/sra, we also have to check the 25..31 bits to get the specific instructions. This can be done by using MASK31_25 to fetch the 25..31 bits and use an if_else to check which value it is. If it is 0, then the instruction should be add or srl according to func3, and otherwise it should be sub or sra according to func3.

Inside each instruction cases, handle_instruction() call the specific handle_xxx() function to execute each specific functions, which will be discussed in the next section.

If none of the switch cases matches, then there should be some mistakes in the input file, this error is handled by the error function.

(b) U type

The U type instruction here is only lui, so we don't need so many branching logics.

We just test if the opcode is 0110111, which can be done by shifting 0x0D left 2 bits and add 0x03. If it is, handle_instruction() call the specific handle_lui() function to execute specific function, which will be discussed in the next section.

(c) jalr

The opcode is 1100111, which can be done by shifting 0x19 left 2 bits and add 0x03. It is unique so we don't need so many branching logics. We just test if the opcode is 1100111, If it is, handle_instruction() call the specific handle_jalr() function to execute specific function, which will be discussed in the next section.

(d) jal

The opcode is 1101111, which can be done by shifting 0x1b left 2 bits and add 0x03. It is unique so we don't need so many branching logics. We just test if the opcode is 1101111, If it is, handle_instruction() call the specific handle_jal() function to execute specific function, which will be discussed in the next section.

(e) B type

The opcode of B type instructions is 1100011, which can be done by shifting 0x18 left 2 bits and add 0x03. And this number is put into the outmost switch logic to identify this type of instruction.

After knowing the type of instructions, we still have to examine func3 using another nested switch to know which instructions it is. Possible cases are 0, 1, 4, 5 corresponding to beq, bne, blt, bge respectively.

Inside each instruction cases, handle_instruction() call the specific handle_xxx() function to execute each specific functions, which will be discussed in the next section.

If none of the switch cases matches, then there should be some mistakes in the input file, this error is handled by the error function.

(f) Load instructions

The opcode of store type instructions is 0000011, which is 0x03. And this number is put into the outmost switch logic to identify this type of instruction.

After knowing the type of instructions, we still have to examine func3 using another nested switch to know which instructions it is. Possible cases are 0, 1, 2 corresponding to lb, lh, lw respectively.

Inside each instruction cases, handle_instruction() call the specific handle_xxx() function to execute each specific functions, which will be discussed in the next section.

If none of the switch cases matches, then there should be some mistakes in the input file, this error is handled by the error function.

(g) store instructions

The opcode of store type instructions is 0100011, which can be done by shifting 0x08 left 2 bits and add 0x03. And this number is put into the outmost switch logic to identify this type of instruction.

After knowing the type of instructions, we still have to examine func3 using another nested switch to know which instructions it is. Possible cases are 0, 1, 2 corresponding to sb, sh, sw respectively.

Inside each instruction cases, handle_instruction() call the specific handle_xxx() function to execute each specific functions, which will be discussed in the next section.

If none of the switch cases matches, then there should be some mistakes in the input file, this error is handled by the error function.

The code is shown in the main code section.

4. Handle specific instructions

In handle_xxx() functions we perform the decode, exe, wb of each instructions.

(a) Handle shift immediates

This includes handle_slli, handle_srli, handle_srai. In these functions we have to decode rd, rs1, calculate immediate value, and perform exe and wb by calculating the result and update the system latches.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7, rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15.

The immediate value is 5 bits so can be retrieve by masking the 20..24 bits with the MASK24_20. Since it is not signed value, so we DO NOT need to use the sext() function to perform sign extend, to extend the 5 bits into 32 bits.

The result is calculated by shifting CURRENT_LATCHES.REGS[rs1] by the immediate value, and assign it to NEXT_LATCHES.REGS[rd], so this register will be updated after the this cycle is finished.

(b) Handle immediate logic

This includes handle_xori, handle_ori, handle_andi. In these functions we have to decode rd, rs1, calculate immediate value, and perform exe and wb

by calculating the result and update the system latches.

The immediate value is 12 bits so can be retrieve by masking the 20..31 bits with the MASK31_20. Since it is signed value and we need to extend the 12 bits into 32 bits, we have to use the sext() function to perform sign extend.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7, rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15.

The result is calculated by performing logic operation on the CURRENT_LATCHES.REGS[rs1] with the immediate value, and assign it to NEXT_LATCHES.REGS[rd], so this register will be updated after the this cycle is finished.

(c) Handle lui

In handle_lui we have to decode rd, calculate immediate value, and load the immediate value to upper 20 bits of rd.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7.

The immediate value is 20 bits so can be retrieve by masking the 12..31 bits with the MASK31_20. Since it is signed value and we need to extend the 20 bits into 32 bits, we have to use the sext() function to perform sign extend.

To load the value into upper 20 bits of rd, we need to update NEXT_LATCHES.REGS[rd] to the immediate value left shifted 12 bits.

(d) Handle register-register arithmetic

This includes handle_add, handle_sub. In these functions we have to decode rd, rs1, rs2, and perform exe and wb by calculating the result and update the system latches.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7, rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15, rs2 can be decoded by first retrieving the 20..24 bits using MASK24_20.

The result is calculated by performing arithmetic operation on the CURRENT_LATCHES.REGS[rs1] with CURRENT_LATCHES.REGS[rs2], and assign it to NEXT_LATCHES.REGS[rd], so this register will be updated after the this cycle is finished.

(e) Handle register-register shift

This includes handle_sll, handle_srl, handle_sra. In these functions we have to decode rd, rs1, rs2, and perform exe and wb by calculating the result and update the system latches.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7, rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15, rs2 can be decoded by first retrieving the 20..24 bits using MASK24_20.

The result is calculated by performing shifting the CURRENT_LATCHES.REGS[rs1] with the amount CURRENT_LATCHES.REGS[rs2], and assign it to NEXT_LATCHES.REGS[rd], so this register will be updated after the this cycle is finished.

(f) Handle register-register logic

This includes handle_xor, handle_or, handle_and. In these functions we have to decode rd, rs1, rs2, and perform exe and wb by calculating the result and update the system latches.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7, rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15, rs2 can be decoded by first retrieving the 20..24 bits using MASK24_20.

The result is calculated by performing logic operation on the CURRENT_LATCHES.REGS[rs1] with CURRENT_LATCHES.REGS[rs2], and assign it to NEXT_LATCHES.REGS[rd], so this register will be updated after the this cycle is finished.

(g) jalr

In handle_jalr we have to decode rd, rs1, calculate immediate value, and update pc and rd.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7. rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15.

The immediate value is 12 bits so can be retrieve by masking the 20..31 bits with the MASK31_20. Since it is signed value and we need to extend the 12 bits into 32 bits, we have to use the sext() function to perform sign extend.

To make sure that we can jump back, we should update NEXT_LATCHES.REGS[rd] to CURRENT_LATCHES.PC +4, which is the address of the next instruction.

To jump, we should update NEXT_LATCHES.PC to the sum of CURRENT_LATCHES.REGS[rs1] and the immediate value. So that the next instruction will be the place to branch to.

(h) jal

In handle_jal we have to decode rd, calculate immediate value, and update pc and rd.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7.

The immediate value is 21 bits and the order is mixed, so we have to retrieve it part by part and perform adding and shifting. The bit 0 is always 0 so we can ignore it. The bit 1..10 is at bit 21..30 of the instruction, so can be retrieved by masking the 21..30 bits with the MASK30_21 and shift it 1 bit and add it to the binary value. The bit 11 is at bit 20 of the instruction, so can be retrieved by masking the 20^{th} bit with the MASK20 and shift it 11 bits and add it to the binary value. The bit 12..19 is at bit 12..19 of the instruction, so can be retrieved by masking the 12..19 bits with the MASK19_12 and shift it 12 bits and add it to the binary value. The bit 20 is at bit 31 of the instruction, so can be retrieved by masking the 31 bit with the MASK31 and shift it 31 bits and add it to the binary value.

To make sure that we can jump back, we should update NEXT_LATCHES.REGS[rd] to CURRENT_LATCHES.PC +4, which is the address of the next instruction.

To jump, we should update NEXT_LATCHES.PC to the sum of

CURRENT_LATCHES.REGS[rs1] and the immediate value sign extended to 32 bits (similar to I-type instructions, the immediate could also be negative). So that the next instruction will be the place to branch to.

(i) Handle branches

This includes handle_beq, handle_bne, handle_blt, handle_bge. In these functions we have to decode rs1, rs2, calculate immediate value, test the condition and update pc.

The rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15, rs2 can be decoded by first retrieving the 20..24 bits using MASK24_20.

The immediate value is 13 bits, and the order is mixed, so we have to retrieve it part by part and perform adding and shifting. The bit 0 is always 0 so we can ignore it. The bit 1..4 is at bit 8..11 of the instruction, so can be retrieved by masking the 8..11 bits with the MASK11_8 and shift it 1 bit and add it to the binary value. The bit 5..10 is at bit 25..30 of the instruction, so can be retrieved by masking the 20..30 bit with the MASK30_25 and shift it 5 bits and add it to the binary value. The bit 11 is at bit 7 of the instruction, so can be retrieved by masking the 17^{th} bits with the MASK7 and shift it 11 bits and add it to the binary value. The bit 12 is at bit 31 of the instruction, so can be retrieved by masking the 31^{st} bits with the MASK31 and shift it 12 bits and add it to the binary value.

Then we need to test the conditions based on CURRENT_LATCHES.REGS[rs1] and CURRENT_LATCHES.REGS[rs2]. If the condition is met, we need to jump, so we should update NEXT_LATCHES.PC to the sum of CURRENT_LATCHES.PC and the immediate value sign extended to 32 bits (similar to I-type instructions, the immediate could also be negative). So that the next instruction will be the place to branch to.

(j) Load

This includes handle_lb, handle_lh, handle_lw. In these functions we have to decode rd, rs1, calculate immediate value, and update rd by the value retrieved from memory.

The rd can be decoded by first retrieving the 7..11 bits using MASK11_7. rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15.

The immediate value is 12 bits so can be retrieve by masking the 20..31 bits with the MASK31_20. Since it is signed value and we need to extend the 12 bits into 32 bits, we have to use the sext() function to perform sign extend.

To retrieve the value from memory, we need to access the MEMORY array. For different instructions, we might need to access MEMORY with different offsets.

Here I take lw as an example since it needs to access biggest amount of memory and all other instructions memory access is included in lw. To access the n^{th} (n is 0, 1, 2, 3) byte starting from the given address calculated by sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1], we can use MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + n] to get the

value. Then we need to left shift it n*8 bits to its corresponding position and add it to the NEXT_LATCHES.REGS[rd].

(k) Store

This include handle_sb, handle_sh, handle_sw. In these functions we have to decode rs1, rs2, calculate immediate value, and store the value to memory.

The rs1 can be decoded by first retrieving the 15..19 bits using MASK19_15, rs2 can be decoded by first retrieving the 20..24 bits using MASK24_20.

The immediate value is 12 bits but it is separated to 2 parts, so we have to retrieve it part by part and perform adding and shifting. The bit 0..4 is at bit 7..11 of the instruction, so can be retrieved by masking the 7..11 bits with the MASK11_7. The bit 5..11 is at bit 25..31 of the instruction, so can be retrieved by masking the 25..31 bits with the MASK31_25 and shift it 5 bit and add it to the binary value. Finally a sign extension is applied.

To store the value to memory, we need to access the MEMORY array. For different instructions, we might need to access MEMORY with different offsets.

Here I take sw as an example since it needs to access the biggest amount of memory and all other instructions memory access is included in sw. To access the n^{th} (n is 0, 1, 2, 3) byte starting from the given address calculated by sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1], we can write each bytes of the value to MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + n]. The byte 1, 2, 3, 4 of the immediate is gotten by MASK7_0, MASK15_8, MASK23_16 and MASK31_24 of the value CURRENT_LATCHES.REGS[rs2].

The code is shown in the main code section.

5. My mistakes

During coding, I made some mistakes and I think it is worthwhile to be recorded in the lab report.

(a) The memory access is by bytes.

At first when I was performing at first wasn't by bytes, but is just assigning or retrieving the whole 32 bits of the memory altogether using a MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1]]. I have bebugged for a long time. I think the reason to make this mistake is that I haven't read the code entirely and didn't notice that memory array is of type unsigned char which is 8 bits.

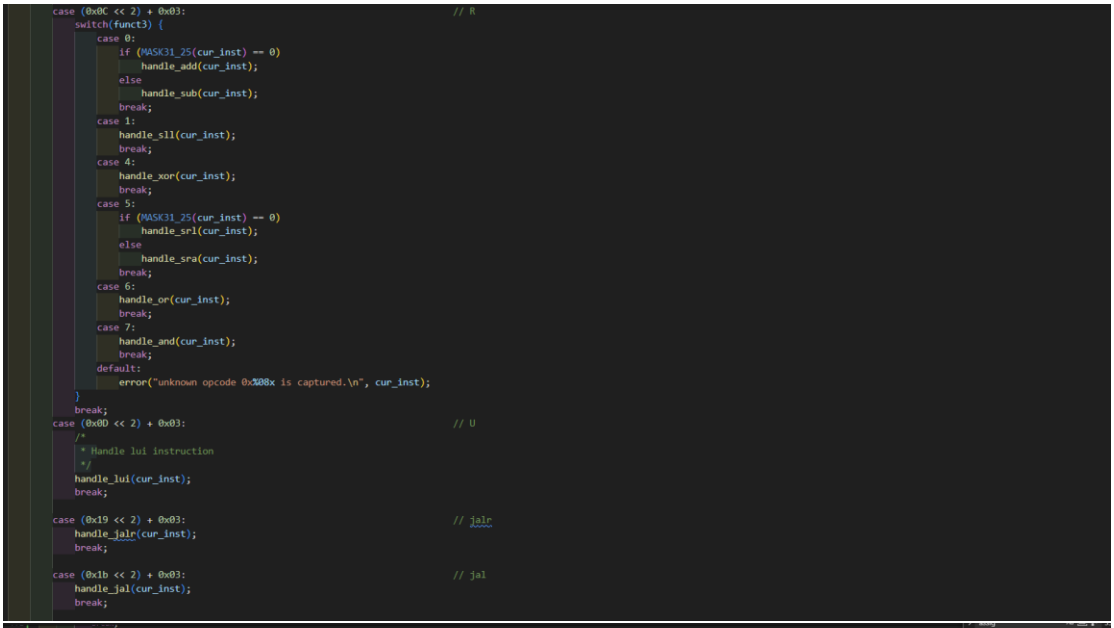
(b) PC update

To update the next PC, for example in jal, I initially just updated it on CURRENT_LATCHES.PC, ie. CURRENT_LATCHES.PC += sext(imm21, 21). Then executing the isa.bin, the program goes to a infinite loop. Then I found that this is because we should update it on the next PC, ie. NEXT_LATCHES.PC = CURRENT_LATCHES.PC + sext(imm21, 21). This is because in the implementation, before the instruction is handled, the NEXT_LATCHES.PC is already updated to CURRENT_LATCHES.PC+4 and will

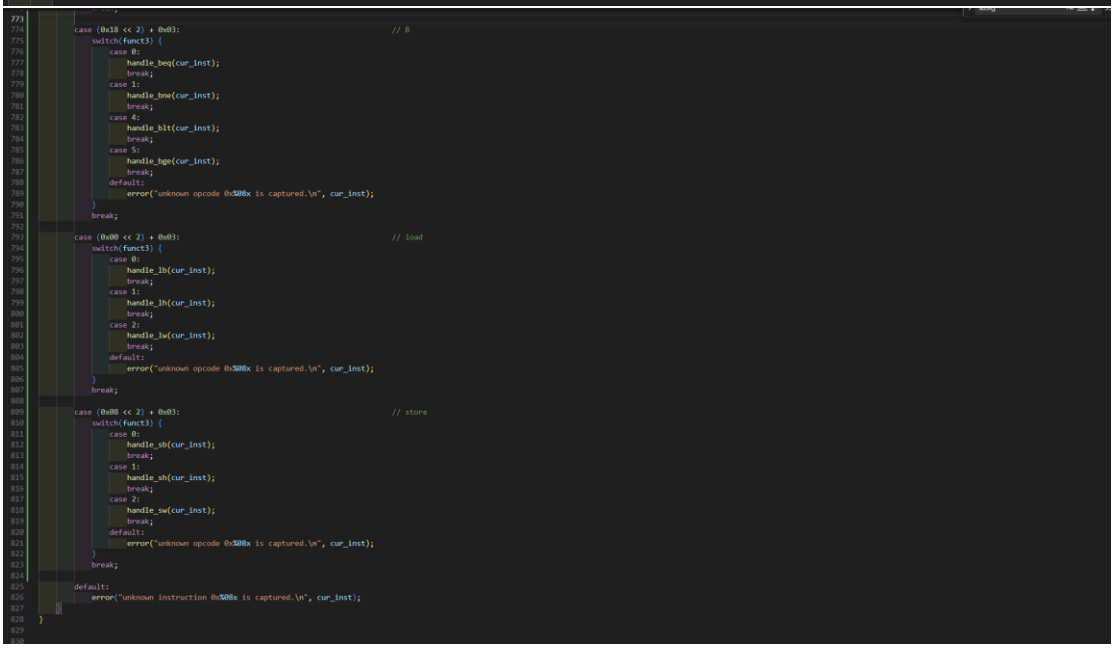
no longer update if no special branch. So updating CURRENT_LATCHES.PC is of no use. This can be understood with the 5 stages of instructions: pc+4 is done in the instruction fetch stage, which is before, EXE and WB.

6. Main code:

(a) Handle_instruction



```
case (0x0C << 2) + 0x03:                                    // R
    switch(funct3) {
        case 0:
            if (MASK31_25(cur_inst) == 0)
                handle_add(cur_inst);
            else
                handle_sub(cur_inst);
            break;
        case 1:
            handle_sll(cur_inst);
            break;
        case 4:
            handle_xor(cur_inst);
            break;
        case 5:
            if (MASK31_25(cur_inst) == 0)
                handle_srl(cur_inst);
            else
                handle_sra(cur_inst);
            break;
        case 6:
            handle_or(cur_inst);
            break;
        case 7:
            handle_and(cur_inst);
            break;
        default:
            error("unknown opcode 0x%08x is captured.\n", cur_inst);
    }
    break;
case (0x0D << 2) + 0x03:                                    // U
    /*
     * Handle lui instruction
     */
    handle_lui(cur_inst);
    break;

case (0x19 << 2) + 0x03:                                    // jalr
    handle_jalr(cur_inst);
    break;

case (0x1b << 2) + 0x03:                                    // jal
    handle_jal(cur_inst);
    break;
```

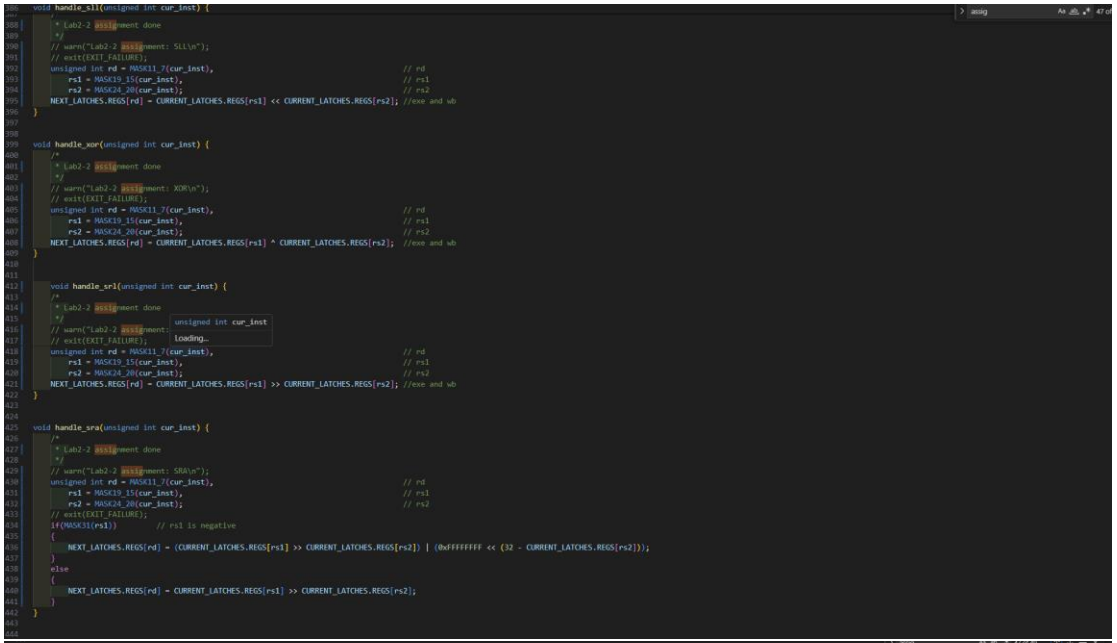


```
773
774     case (0x18 << 2) + 0x03:                                // B
775         switch(funct3) {
776             case 0:
777                 handle_beq(cur_inst);
778                 break;
779             case 1:
780                 handle_bne(cur_inst);
781                 break;
782             case 4:
783                 handle_blt(cur_inst);
784                 break;
785             case 5:
786                 handle_bge(cur_inst);
787                 break;
788             default:
789                 error("unknown opcode 0x%08x is captured.\n", cur_inst);
790         }
791         break;
792
793     case (0x00 << 2) + 0x03:                                // load
794         switch(funct3) {
795             case 0:
796                 handle_lb(cur_inst);
797                 break;
798             case 1:
799                 handle_lh(cur_inst);
800                 break;
801             case 2:
802                 handle_lw(cur_inst);
803                 break;
804             default:
805                 error("unknown opcode 0x%08x is captured.\n", cur_inst);
806         }
807         break;
808
809     case (0x08 << 2) + 0x03:                                // store
810         switch(funct3) {
811             case 0:
812                 handle_sb(cur_inst);
813                 break;
814             case 1:
815                 handle_sh(cur_inst);
816                 break;
817             case 2:
818                 handle_sw(cur_inst);
819                 break;
820             default:
821                 error("unknown opcode 0x%08x is captured.\n", cur_inst);
822         }
823         break;
824
825     default:
826         error("unknown instruction 0x%08x is captured.\n", cur_inst);
827 }
828 }
829
830
```

(b) Hande_xxx

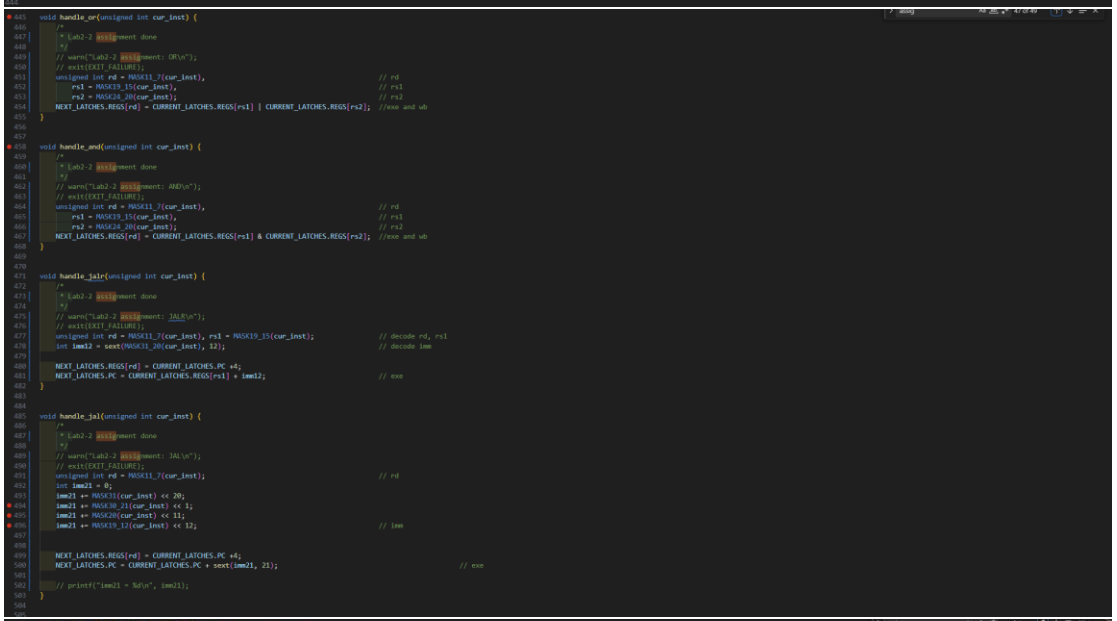
```
void handle_addi(unsigned int cur_inst) {
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);              // decode rd, rs1
    int imm12 = sext(MASK31_20(cur_inst), 12);                                    // decode imm
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] + imm12;                    // exe and wb
}



void handle_slli(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SLLI\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);              // decode rd, rs1
    int imm5 = MASK24_20(cur_inst);                                               // decode imm
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] << imm5;                    // exe and wb
}



void handle_xori(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: XORI\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);              // decode rd, rs1
    int imm12 = sext(MASK31_20(cur_inst), 12);                                    // decode imm
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] ^ imm12;                    // exe and wb
}


void handle_srli(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SRLI\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);              // decode rd, rs1
    int imm5 = MASK24_20(cur_inst);                                               // decode imm
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] >> imm5;                    // exe and wb
}


void handle_srai(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SRAI\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);              // decode rd, rs1
    int imm5 = MASK24_20(cur_inst);                                               // decode imm
    if(MASK31(rs1))         // rs1 is negative
    {
        NEXT_LATCHES.REGS[rd] = (CURRENT_LATCHES.REGS[rs1] >> imm5) | (0xFFFFFFFF << (32 - imm5));
    }
    else
    {
        NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] >> imm5;
    }                                                                             // exe and wb
}
```

```
void handle_ori(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: ORI\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);              // decode rd, rs1
    int imm12 = sext(MASK31_20(cur_inst), 12);                                    // decode imm
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] | imm12;                    // exe and wb
}


void handle_andi(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: ANDI\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);              // decode rd, rs1
    int imm12 = sext(MASK31_20(cur_inst), 12);                                    // decode imm
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] & imm12;                    // exe and wb
}

void handle_lui(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: LUI\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst);                                         // rd
    int imm20 = sext(MASK31_12(cur_inst), 20);                                    // imm
    NEXT_LATCHES.REGS[rd] = imm20 << 12;                                          // exe and wb

}


void handle_add(unsigned int cur_inst) {
    unsigned int rd = MASK11_7(cur_inst),                                         // rd
        rs1 = MASK19_15(cur_inst),                                                // rs1
        rs2 = MASK24_20(cur_inst);                                                // rs2
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] + CURRENT_LATCHES.REGS[rs2];  //exe and wb
}


void handle_sub(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SUB\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst),                                         // rd
        rs1 = MASK19_15(cur_inst),                                                // rs1
        rs2 = MASK24_20(cur_inst);                                                // rs2
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] - CURRENT_LATCHES.REGS[rs2];  //exe and wb
}
```

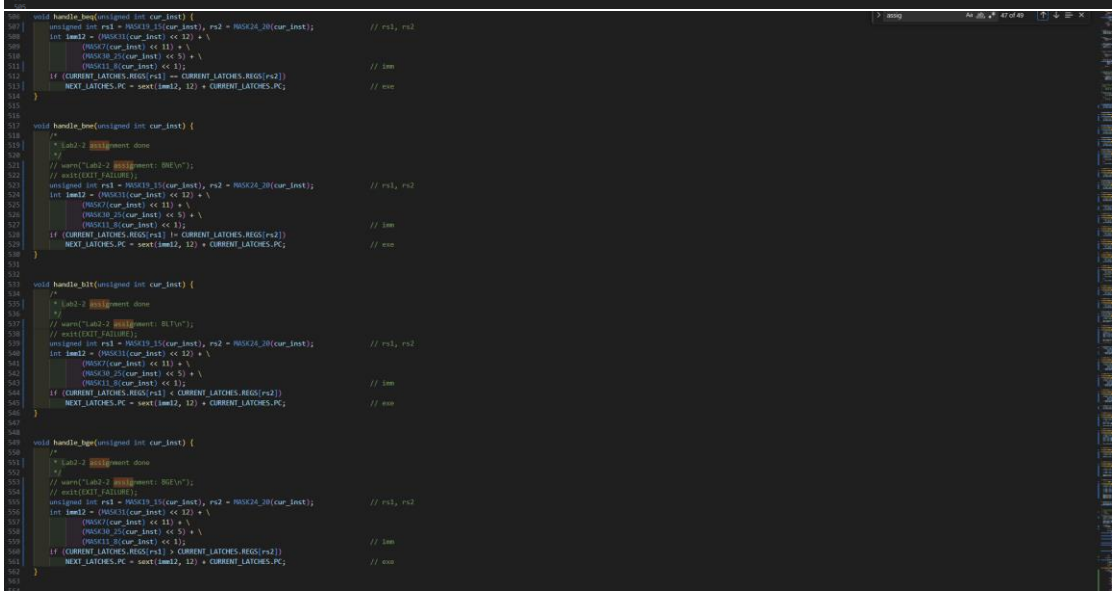


```
void handle_sll(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SLL\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst),                                   // rd
        rs1 = MASK19_15(cur_inst),                                          // rs1
        rs2 = MASK24_20(cur_inst);                                          // rs2
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] << CURRENT_LATCHES.REGS[rs2]; //exe and wb
}


void handle_xor(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: XOR\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst),                                   // rd
        rs1 = MASK19_15(cur_inst),                                          // rs1
        rs2 = MASK24_20(cur_inst);                                          // rs2
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] ^ CURRENT_LATCHES.REGS[rs2];  //exe and wb
}


    void handle_srl(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment:
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst),                                   // rd
        rs1 = MASK19_15(cur_inst),                                          // rs1
        rs2 = MASK24_20(cur_inst);                                          // rs2
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] >> CURRENT_LATCHES.REGS[rs2]; //exe and wb
}


void handle_sra(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SRA\n");
    unsigned int rd = MASK11_7(cur_inst),                                   // rd
        rs1 = MASK19_15(cur_inst),                                          // rs1
        rs2 = MASK24_20(cur_inst);                                          // rs2
    // exit(EXIT_FAILURE);
    if(MASK31(rs1))         // rs1 is negative
    {
        NEXT_LATCHES.REGS[rd] = (CURRENT_LATCHES.REGS[rs1] >> CURRENT_LATCHES.REGS[rs2]) | (0xFFFFFFFF << (32 - CURRENT_LATCHES.REGS[rs2]));
    }
    else
    {
        NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] >> CURRENT_LATCHES.REGS[rs2];
    }
}
```

Loading...



```
void handle_or(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: OR\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst),                                   // rd
        rs1 = MASK19_15(cur_inst),                                          // rs1
        rs2 = MASK24_20(cur_inst);                                          // rs2
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] | CURRENT_LATCHES.REGS[rs2];  //exe and wb
}


void handle_and(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: AND\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst),                                   // rd
        rs1 = MASK19_15(cur_inst),                                          // rs1
        rs2 = MASK24_20(cur_inst);                                          // rs2
    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] & CURRENT_LATCHES.REGS[rs2];  //exe and wb
}


void handle_jalr(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: JALR\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);        // decode rd, rs1
    int imm12 = sext(MASK31_20(cur_inst), 12);                              // decode imm

    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.PC +4;
    NEXT_LATCHES.PC = CURRENT_LATCHES.REGS[rs1] + imm12;                    // exe
}


void handle_jal(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: JAL\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst);                                   // rd
    int imm21 = 0;
    imm21 += MASK31(cur_inst) << 20;
    imm21 += MASK30_21(cur_inst) << 1;
    imm21 += MASK20(cur_inst) << 11;
    imm21 += MASK19_12(cur_inst) << 12;                                     // imm


    NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.PC +4;
    NEXT_LATCHES.PC = CURRENT_LATCHES.PC + sext(imm21, 21);                                 // exe

    // printf("imm21 = %d\n", imm21);
}
```



```
void handle_beq(unsigned int cur_inst) {
    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);      // rs1, rs2
    int imm12 = (MASK31(cur_inst) << 12) + \
            (MASK7(cur_inst) << 11) + \
            (MASK30_25(cur_inst) << 5) + \
            (MASK11_8(cur_inst) << 1);                                      // imm
    if (CURRENT_LATCHES.REGS[rs1] == CURRENT_LATCHES.REGS[rs2])
        NEXT_LATCHES.PC = sext(imm12, 12) + CURRENT_LATCHES.PC;             // exe
}


void handle_bne(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: BNE\n");
    // exit(EXIT_FAILURE);
    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);      // rs1, rs2
    int imm12 = (MASK31(cur_inst) << 12) + \
            (MASK7(cur_inst) << 11) + \
            (MASK30_25(cur_inst) << 5) + \
            (MASK11_8(cur_inst) << 1);                                      // imm
    if (CURRENT_LATCHES.REGS[rs1] != CURRENT_LATCHES.REGS[rs2])
        NEXT_LATCHES.PC = sext(imm12, 12) + CURRENT_LATCHES.PC;             // exe
}


void handle_blt(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: BLT\n");
    // exit(EXIT_FAILURE);
    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);      // rs1, rs2
    int imm12 = (MASK31(cur_inst) << 12) + \
            (MASK7(cur_inst) << 11) + \
            (MASK30_25(cur_inst) << 5) + \
            (MASK11_8(cur_inst) << 1);                                      // imm
    if (CURRENT_LATCHES.REGS[rs1] < CURRENT_LATCHES.REGS[rs2])
        NEXT_LATCHES.PC = sext(imm12, 12) + CURRENT_LATCHES.PC;             // exe
}


void handle_bge(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: BGE\n");
    // exit(EXIT_FAILURE);
    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);      // rs1, rs2
    int imm12 = (MASK31(cur_inst) << 12) + \
            (MASK7(cur_inst) << 11) + \
            (MASK30_25(cur_inst) << 5) + \
            (MASK11_8(cur_inst) << 1);                                      // imm
    if (CURRENT_LATCHES.REGS[rs1] > CURRENT_LATCHES.REGS[rs2])
        NEXT_LATCHES.PC = sext(imm12, 12) + CURRENT_LATCHES.PC;             // exe
}
```

```c
void handle_lb(unsigned int cur_inst) {
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);                    // rd, rs1
    int imm12 = MASK31_20(cur_inst);                                                    // imm
    NEXT_LATCHES.REGS[rd] = sext(MASK7_0(MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1]]), 8);    // exe and wb
}


void handle_lh(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: LH\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);                    // rd, rs1
    int imm12 = MASK31_20(cur_inst);                                                    // imm

    int half = 0;
    half += MASK7_0(MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1]]);               // 1st byte
    half += MASK7_0(MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 1]) << 8;  // 2nd byte

    NEXT_LATCHES.REGS[rd] = sext(half, 16);    // exe and wb
}

void handle_lw(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: LW\n");
    // exit(EXIT_FAILURE);
    unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);                    // rd, rs1
    int imm12 = MASK31_20(cur_inst);                                                    // imm
    int word = 0;
    word += MASK7_0(MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1]]);               // 1st byte
    word += MASK7_0(MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 1]) << 8;  // 2nd byte
    word += MASK7_0(MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 2]) << 16; // 3rd byte
    word += MASK7_0(MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 3]) << 24; // 4th byte


    NEXT_LATCHES.REGS[rd] = word;    // exe and wb
}


void handle_sb(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SB\n");
    // exit(EXIT_FAILURE);
    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);                  // rs1, rs2
    int imm12 = MASK11_7(cur_inst) + (MASK31_25(cur_inst) << 5);                        // imm

    MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1]] = MASK7_0(CURRENT_LATCHES.REGS[rs2]);   // exe and wb
}
```

```c
void handle_sh(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SH\n");
    // exit(EXIT_FAILURE);

    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);                  // rs1, rs2
    int imm12 = MASK11_7(cur_inst) + (MASK31_25(cur_inst) << 5);                        // imm

    int byte1 = MASK7_0(CURRENT_LATCHES.REGS[rs2]);
    int byte2 = MASK15_8(CURRENT_LATCHES.REGS[rs2]);

    MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1]] = byte1;
    MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 1] = byte2;                    // exe and wb
}


void handle_sw(unsigned int cur_inst) {
    /*
     * Lab2-2 assignment done
     */
    // warn("Lab2-2 assignment: SW\n");
    // exit(EXIT_FAILURE);
    unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);                  // rs1, rs2
    int imm12 = MASK11_7(cur_inst) + (MASK31_25(cur_inst) << 5);                        // imm

    int byte1 = MASK7_0(CURRENT_LATCHES.REGS[rs2]);
    int byte2 = MASK15_8(CURRENT_LATCHES.REGS[rs2]);
    int byte3 = MASK23_16(CURRENT_LATCHES.REGS[rs2]);
    int byte4 = MASK31_24(CURRENT_LATCHES.REGS[rs2]);


    MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1]] = byte1;
    MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 1] = byte2;
    MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 2] = byte3;
    MEMORY[sext(imm12, 12) + CURRENT_LATCHES.REGS[rs1] + 3] = byte4;                    // exe and wb
}
```

7. Console results

(a) Make:

```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-2/code$ make
cc -Wall -std=c99 -Wno-return-type -O3  sim.c util.c -o sim
sim.c: In function 'get_command':
sim.c:152:5: warning: ignoring return value of 'scanf' declared with attribute 'warn_unused_result' [-Wunused-result]
  152 |     scanf("%s", buffer);
      |     ^~~~~~~~~~~~~~~~~~~
sim.c:162:13: warning: ignoring return value of 'scanf' declared with attribute 'warn_unused_result' [-Wunused-result]
  162 |             scanf("%i %i", &start, &stop);
      |             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
sim.c:179:17: warning: ignoring return value of 'scanf' declared with attribute 'warn_unused_result' [-Wunused-result]
  179 |                 scanf("%d", &cycles);
      |                 ^~~~~~~~~~~~~~~~~~~~
In file included from /usr/include/string.h:535,
                 from util.h:18,
                 from util.c:12:
In function 'strncpy',
    inlined from 'copy_str' at util.c:88:8:
/usr/include/x86_64-linux-gnu/bits/string_fortified.h:95:10: warning: '__builtin_strncpy' output truncated before terminating nul copying as many bytes from a string as its length
 [-Wstringop-truncation]
   95 |   return __builtin___strncpy_chk (__dest, __src, __len,
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   96 |                                   __glibc_objsize (__dest));
      |                                   ~~~~~~~~~~~~~~~~~~~~~~~~~
util.c: In function 'copy_str':
util.c:88:15: note: length computed here
   88 |         ptr = strncpy(tgt, src, strlen(src));
      |               ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

There are some warnings but it turns out to be some version issues and is about the unsafe use of some string manipulation functions, so we can ignore it in this assignment.

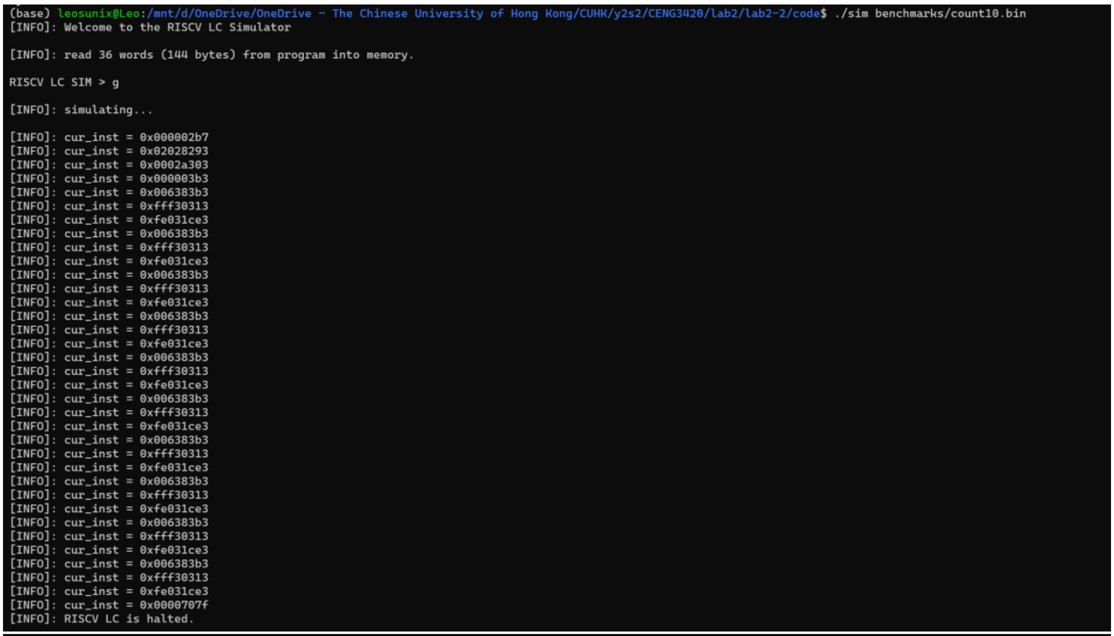
(b) Isa.bin

```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-2/code$
./sim benchmarks/isa.bin
[INFO]: Welcome to the RISCV LC Simulator

[INFO]: read 136 words (544 bytes) from program into memory.
```

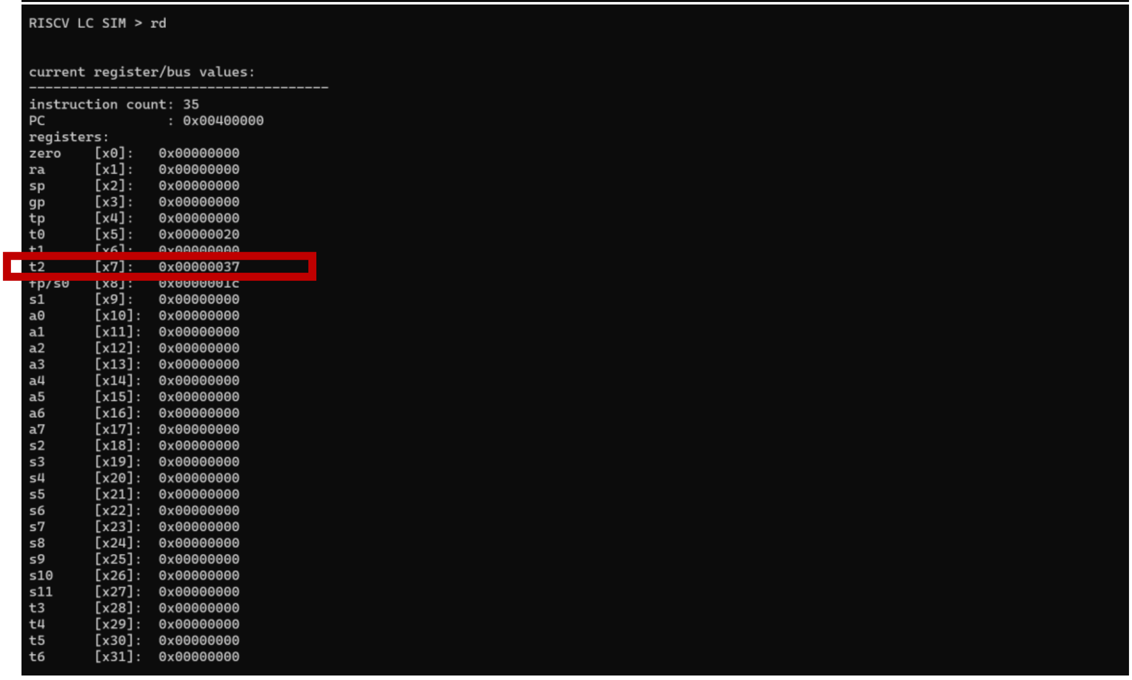
```
RISCV LC SIM > g

[INFO]: simulating...

[INFO]: cur_inst = 0x00000537
[INFO]: cur_inst = 0x08050513
[INFO]: cur_inst = 0x00052503
[INFO]: cur_inst = 0x00054263
[INFO]: cur_inst = 0x00d50893
[INFO]: cur_inst = 0xff105ee3
[INFO]: cur_inst = 0x00000537
[INFO]: cur_inst = 0x08050513
[INFO]: cur_inst = 0x00150583
[INFO]: cur_inst = 0x7ff5c613
[INFO]: cur_inst = 0x00052503
[INFO]: cur_inst = 0x00a606b3
[INFO]: cur_inst = 0x00a60733
[INFO]: cur_inst = 0xfff88793
[INFO]: cur_inst = 0x00c0006f
[INFO]: cur_inst = 0xff9ff06f
[INFO]: cur_inst = 0x00c000ef
[INFO]: cur_inst = 0x40a88833
[INFO]: cur_inst = 0x00008067
[INFO]: cur_inst = 0x0100006f
[INFO]: cur_inst = 0x00381893
[INFO]: cur_inst = 0x0028d693
[INFO]: cur_inst = 0x40d006b3
[INFO]: cur_inst = 0x4026d713
[INFO]: cur_inst = 0x00f6c693
[INFO]: cur_inst = 0x40d006b3
[INFO]: cur_inst = 0x000004b7
[INFO]: cur_inst = 0x08448493
[INFO]: cur_inst = 0x00d48423
[INFO]: cur_inst = 0x00e6c6b3
[INFO]: cur_inst = 0x00d4a823
[INFO]: cur_inst = 0x0000707f
[INFO]: RISCV LC is halted.
```

```
RISCV LC SIM > rd


current register/bus values:
--------------------------------------
instruction count: 32
PC              : 0x00400000
registers:
zero    [x0]:   0x00000000
ra      [x1]:   0x00000040
sp      [x2]:   0x00000000
gp      [x3]:   0x00000000
tp      [x4]:   0x00000000
t0      [x5]:   0x00000000
t1      [x6]:   0x00000000
t2      [x7]:   0x00000000
fp/s0   [x8]:   0x0000007c
s1      [x9]:   0x00000084
a0      [x10]:  0xfffffffe
a1      [x11]:  0xffffffff
a2      [x12]:  0xfffff800
a3      [x13]:  0xffffffee
a4      [x14]:  0xffffffff
a5      [x15]:  0x0000000a
a6      [x16]:  0x0000000d
a7      [x17]:  0x00000068
s2      [x18]:  0x00000000
s3      [x19]:  0x00000000
s4      [x20]:  0x00000000
s5      [x21]:  0x00000000
s6      [x22]:  0x00000000
s7      [x23]:  0x00000000
s8      [x24]:  0x00000000
s9      [x25]:  0x00000000
s10     [x26]:  0x00000000
s11     [x27]:  0x00000000
t3      [x28]:  0x00000000
t4      [x29]:  0x00000000
t5      [x30]:  0x00000000
t6      [x31]:  0x00000000

RISCV LC SIM > mdump 0x84 0x94


memory content [0x00000084..0x00000094]:
--------------------------------------
  0x00000084 (132) : 0xfffffff7
  0x00000088 (136) : 0x00000000
  0x0000008c (140) : 0x00000017
  0x00000090 (144) : 0x00000000
  0x00000094 (148) : 0xffffffee
```

(c) Count10

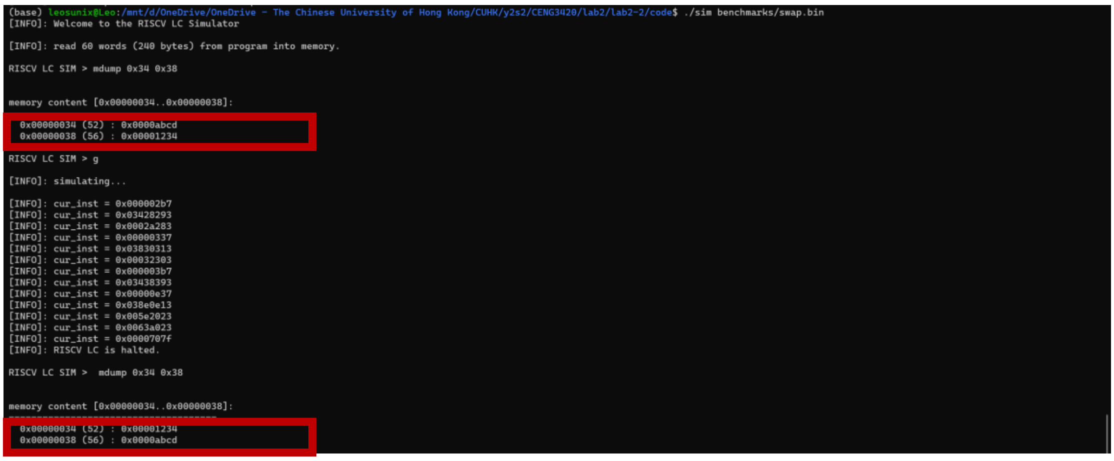


```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-2/code$ ./sim benchmarks/count10.bin
[INFO]: Welcome to the RISCV LC Simulator

[INFO]: read 36 words (144 bytes) from program into memory.

RISCV LC SIM > g

[INFO]: simulating...

[INFO]: cur_inst = 0x000002b7
[INFO]: cur_inst = 0x02028293
[INFO]: cur_inst = 0x0002a303
[INFO]: cur_inst = 0x000003b3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x006383b3
[INFO]: cur_inst = 0xfff30313
[INFO]: cur_inst = 0xfe031ce3
[INFO]: cur_inst = 0x0000707f
[INFO]: RISCV LC is halted.
```



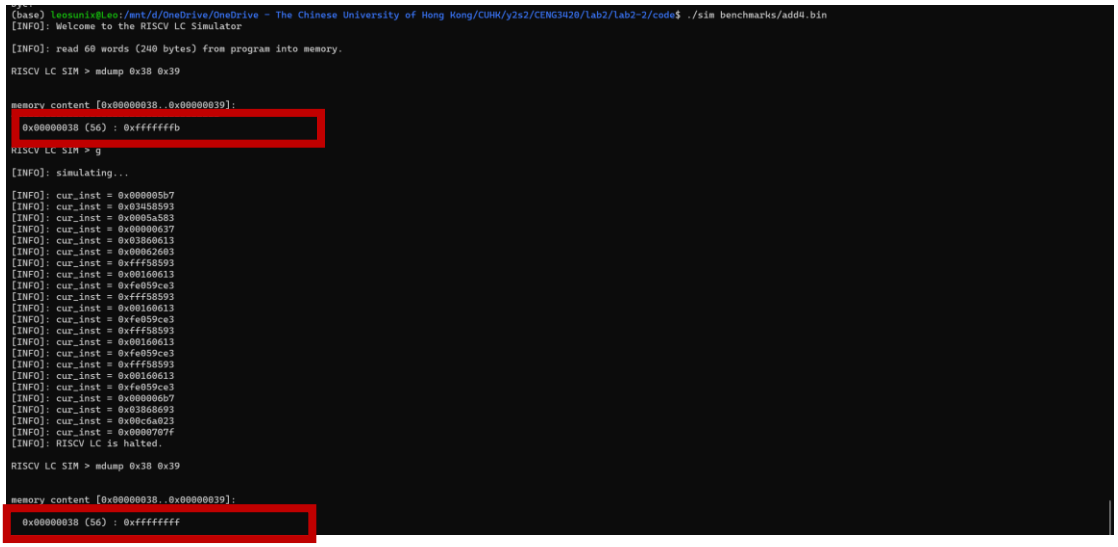
```
RISCV LC SIM > rd


current register/bus values:
-------------------------------------
instruction count: 35
PC                : 0x00400000
registers:
zero     [x0]:   0x00000000
ra       [x1]:   0x00000000
sp       [x2]:   0x00000000
gp       [x3]:   0x00000000
tp       [x4]:   0x00000000
t0       [x5]:   0x00000020
t1       [x6]:   0x00000000
t2       [x7]:   0x00000037
fp/s0    [x8]:   0x0000001c
s1       [x9]:   0x00000000
a0       [x10]:  0x00000000
a1       [x11]:  0x00000000
a2       [x12]:  0x00000000
a3       [x13]:  0x00000000
a4       [x14]:  0x00000000
a5       [x15]:  0x00000000
a6       [x16]:  0x00000000
a7       [x17]:  0x00000000
s2       [x18]:  0x00000000
s3       [x19]:  0x00000000
s4       [x20]:  0x00000000
s5       [x21]:  0x00000000
s6       [x22]:  0x00000000
s7       [x23]:  0x00000000
s8       [x24]:  0x00000000
s9       [x25]:  0x00000000
s10      [x26]:  0x00000000
s11      [x27]:  0x00000000
t3       [x28]:  0x00000000
t4       [x29]:  0x00000000
t5       [x30]:  0x00000000
t6       [x31]:  0x00000000
```

(d) Swap



```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-2/code$ ./sim benchmarks/swap.bin
[INFO]: Welcome to the RISCV LC Simulator

[INFO]: read 60 words (240 bytes) from program into memory.

RISCV LC SIM > mdump 0x34 0x38


memory content [0x00000034..0x00000038]:

  0x00000034 (52) : 0x0000abcd
  0x00000038 (56) : 0x00001234

RISCV LC SIM > g

[INFO]: simulating...

[INFO]: cur_inst = 0x000002b7
[INFO]: cur_inst = 0x03428293
[INFO]: cur_inst = 0x0002a283
[INFO]: cur_inst = 0x00000337
[INFO]: cur_inst = 0x03830313
[INFO]: cur_inst = 0x00032303
[INFO]: cur_inst = 0x000003b7
[INFO]: cur_inst = 0x03438393
[INFO]: cur_inst = 0x00000e37
[INFO]: cur_inst = 0x038e0e13
[INFO]: cur_inst = 0x005e2023
[INFO]: cur_inst = 0x0063a023
[INFO]: cur_inst = 0x0000707f
[INFO]: RISCV LC is halted.

RISCV LC SIM >  mdump 0x34 0x38


memory content [0x00000034..0x00000038]:
-------------------------------------
  0x00000034 (52) : 0x00001234
  0x00000038 (56) : 0x0000abcd
```

(e) Add4


(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-2/code$ ./sim benchmarks/add4.bin
[INFO]: Welcome to the RISCV LC Simulator
[INFO]: read 60 words (240 bytes) from program into memory.
RISCV LC SIM > mdump 0x38 0x39
memory content [0x00000038..0x00000039]:
0x00000038 (56) : 0xfffffffb
RISCV LC SIM > g
[INFO]: simulating...
[INFO]: cur_inst = 0x000005b7
[INFO]: cur_inst = 0x03458593
[INFO]: cur_inst = 0x0005a583
[INFO]: cur_inst = 0x00000637
[INFO]: cur_inst = 0x03860613
[INFO]: cur_inst = 0x00062603
[INFO]: cur_inst = 0xfff58593
[INFO]: cur_inst = 0x00160613
[INFO]: cur_inst = 0xfe059ce3
[INFO]: cur_inst = 0xfff58593
[INFO]: cur_inst = 0x00160613
[INFO]: cur_inst = 0xfe059ce3
[INFO]: cur_inst = 0xfff58593
[INFO]: cur_inst = 0x00160613
[INFO]: cur_inst = 0xfe059ce3
[INFO]: cur_inst = 0xfff58593
[INFO]: cur_inst = 0x00160613
[INFO]: cur_inst = 0xfe059ce3
[INFO]: cur_inst = 0x000006b7
[INFO]: cur_inst = 0x03868693
[INFO]: cur_inst = 0x00c6a023
[INFO]: cur_inst = 0x0000707f
[INFO]: RISCV LC is halted.
RISCV LC SIM > mdump 0x38 0x39
memory content [0x00000038..0x00000039]:
0x00000038 (56) : 0xffffffff

Reference:

TextBook -Computer Organization and Design_ The Hardware Software Interface [RISC-V Edition]

opcodes-rv32i reference document

risc-v-asm-manual.pdf

riscv-spec-20191213.pdf