Lab 2.1 Report

1. Problem analysis

   In lab 2.1, we are trying to use C language to simulate the assembly process, which means when we input the assembly language file to the *asm* executable, it will turn the assembly instructions to the binary machine code that the machines can understand.

   The given *asm.c* has finished most of the text manipulation process for us, and the only thing we need to do is to focus on the *inst_to_binary()* function, which translates the RV32I instructions to machine binaries based on the already parsed opcode and argument strings.

2. My understanding of the assembly process

   By examining the *asm.c* and *asm.h* files, I have a basic understanding about the whole assembly process.

   (a) Terminologies

      (i) opcode

         The "opcode" in some the places in the file is slightly different from the opcode that we say that represents the [0..6] bits in the binary instruction. It can also mean the instruction names, for example *addi*, *lw*, *etc*

      (ii) label

         The meaning of label is the same with label in assembly language. Notably, should only consist of numbers and alphabets or pure alphabets.

      (iii) arch_regs

         the real name and alias for the registers.

      (iv) regs_indirect_addr

         Forms like 8(t0). In the structure, the reg part is t0 and the imm part is 8.

      (v) binary

         The binary machine code after assembly.

   (b) Data structures

      (i) truct_regs_indirect_addr

         This structure is going to be one element of the array *label_table*, where each elements stores a label appeared in the input binary file. Especially, the *label[ ]* is a string storing the label itself.

   (c) Assembly process

      (i) The whole program starts with the main function. It first takes 2 arguments in the console as the input, the first is the .asm file that stores the assembly languages, and the second is the output file where the binary code will be stored.

      (ii) The *assemble()* function is the main function for assembly. It first handles the labels and puts all relevant information into the *label_table.* Then it uses the *parse_inst()* function to parse each lines of instructions. By examining the type of instructions, it will handle the special instructions including *halt*, *fill*, *la* specially, which is already implemented. And we will implement the normal instructions by finishing the *inst_to_binary()* function.

(d) Some special functions

(i) int reg_to_num(char *, int): input a string of reg, return the encoding (the number) of the register.

(ii) int handle_label_or_imm(int, char *, struct_label_table *, int): if is a label, return its address, if is an imm, return its int value (assure it must be in width 20).

(iii) int validate_imm(char *, int, int): first convert the int string (1st param) to integer, then 2nd param specifies validation. If is -1, don't validate and directly return, else 2nd param is the width, if the int is in the width, can return, else error.

(iv) int is_opcode(char *): input a token (string) and check if it is a opcode. If is, return the index of the operation in the opcode table, if is not, return MISMATCH.

(v) int is_arg(char *): possible return can be: if is an empty string, retrun MISMATCH; if is is_label type label, return VALID_LABEL; if is register, return ARCH_REGS; if is immediate, return IMM; if is is_regs_indirect_addr, return REGS_INDIRECT_ADDR; else, still return REGS_INDIRECT_ADDR.

3. *inst_to_binary()* implementation

The variable *binary* is the opcode. Actually, this whole function is to fill in *binary* according to different operation types.

(a) Integer Register-Immediate Instructions except for *lui*

(i) Description: I type instructions, all the opcode is 0010011, and have rd, rs1, imm.
Example: addi a0[arg1], a1[arg2], 3[arg3].

(ii) addi
The opcode is 0010011, which can be done by shifting 0x04 left 2 bits and adding 0x03. Since opcode is [0:6], so no left shift should be made.
rd and rs1 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7 and 15 bits respectively since the start from bit 7 and 15.
func3 is 0, so don’t need to do anything.
Immediate value can be got by doing MASK11_0 to the return value of validate_imm to only get the first 12 bits of the return (because in I type, immediate value has only 12 bits) and left shifting 20 bits.

(iii) Immediate shift including *slli*, *srli*, *srai*

The opcode is 0010011, which can be done by shifting 0x04 left 2 bits and adding 0x03. Since opcode is [0:6], so no left shift should be made.

rd and rs1 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7 and 15 bits respectively since the start from bit 7 and 15.

func3 is 3, 5, 5 repectively, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.

For shifting, the immediate value is separated to 2 parts. The first part is the shifting amount. It only takes 5 bits since the max amount to be shifted in a word is 2^5-1 = 31. And the upper bits are used to indicate arithmetic shift (32 from bit 25) and logical shift (0 from bit 25). So, to retrieve shift amount we need to use the lower5bit and shift it left 20 bits, and only srai has a non-zero bits after bit 25, so we need to shift 32 for 25 bits, or equivalently shift 16 for 26 bits.

(iv) Logical immediate: *xori*, *ori*, *andi*

The opcode is 0010011, which can be done by shifting 0x04 left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

rd and rs1 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7 and 15 bits respectively since the start from bit 7 and 15.

func3 is 4, 6, 7 repectively, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.

Immediate value can be gotten by doing MASK11_0 to the return value of validate_imm to only get the first 12 bits of the return (because in I type, immediate value has only 12 bits) and left shifting 20 bits.

(b) *lui* instruction

(i) Description: U type instruction, all the opcode is 0110111, and have 20-bit imm.

(ii) Implementation

The opcode is 0110111, which can be done by shifting 0x0D left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

The immediate value is the upper 20 bits, so we can do a masking by using a bitwise and to the return value of handle_label_or_imm and 0xFFFFF000.

(c) Integer Register-Regiser Instructions

(i) Description: R type instructions, all the opcode is 0110011, and have rd, rs1, rs2.

Example: add a0[arg1], a1[arg2], a2[arg3]

(ii) *add*, *sub*

The opcode is 0110011, which can be done by shifting 0x0C left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

rd and rs1, rs2 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7, 15 and 20 bits respectively since the start from bit 7, 15 and 20.

func3 is 0, so don't need to do anything.

The func7 is 0 and 32 respectively from the bit 25, so we should shift the values left 25 bits.

(iii) Shift including *sll*, *srl*, *sra*

The opcode is 0110011, which can be done by shifting 0x0C left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

rd and rs1, rs2 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7, 15 and 20 bits respectively since the start from bit 7, 15 and 20.

func3 is 3, 5, 5 repectively, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.

The func7 is 0, 0 and 32 respectively from the bit 25, so we should shift the values left 25 bits.

(v) Logical operation: *xor*, *or*, *and*

The opcode is 0110011, which can be done by shifting 0x0C left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

rd and rs1, rs2 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7, 15 and 20 bits respectively since the start from bit 7, 15 and 20.

func3 is 4, 6, 7 repectively, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.

The func7 is 0, 0 and 0 respectively from the bit 25, so we should shift the values left 25 bits.

(d) Unconditional Jumps

(i) *jalr*: I type, eg. jalr zero, (0)ra.

The opcode is 1100111, which can be done by shifting 0x19 left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

rd and rs1 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7 and 15 bits respectively since the start from bit 7 and 15.

func3 is 0, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.

Immediate value can be gotten by doing MASK11_0 to the return value of validate_imm to only get the first 12 bits of the return (because in this type, immediate value has only 12 bits) and left shifting 20 bits.

(ii) *jal*: J type, eg jal rd[arg1], label[arg2]

The opcode is 1101111, which can be done by shifting 0x1b left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

rd can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7 bits since the start from bit 7.

The immediate value of jal is the offset form the address of current instruction to the label, which can be calculated by subtracting the current address (addr) from the arg2 which can be retrieved by handle_label_or_imm.

For imm[19:12], it's at the original position, so we only need to mask it by 0xff000. For imm[11], imm[10:1] and imm[20], their positions in binary and in immediate number are not the same, so after masking with 0x800, 0x7fe and 0x100000, it still needs to be shifted left 9, 20, 11 respectively.

(e) Conditional Branches

(i) Description: B type, eg beq s1[arg1], s2[agr2], label[arg3]

(ii) Implementation

The opcode is 1100011, which can be done by shifting 0x18 left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.

rs1, rs2 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 15 and 20 bits respectively since the start from bit 15 and 20.

func3 is 0, 1, 4, 5 repectively, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.

The value of branch is the offset form the address of current instruction to the label, which can be calculated by subtracting the current address (addr) from the arg2 which can be retrieved by handle_label_or_imm. For imm[11], imm[4:1], imm[10:5] and imm[12], their positions in binary and in immediate number are not the same, so after masking with 0x800, 0x1E, 0x7E0 and 0x100000, it still needs to be shifted left 9, right 7, left 20, 19 respectively.

(f) Load

(i) Description: L type, eg lb a0[arg1], 0(t0)[arg2]

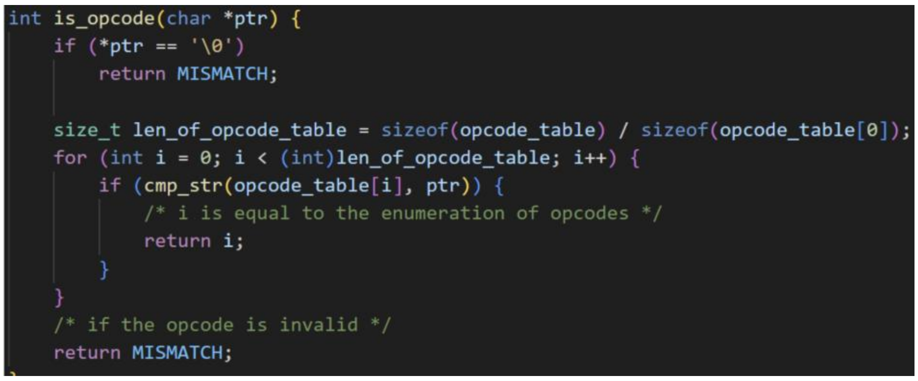
(ii) Implementation

The opcode is 0000011, which can be done by adding 0x03. Since opcode is [0:6], so no left shift should be made.

For lb, lh, lw, func3 is 0, 1, 2 repectively, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.
rd and rs1 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 7 and 15 bits respectively since the start from bit 7 and 15.
Immediate value can be got by doing MASK11_0 to the return value of validate_imm to only get the first 12 bits of the return (because in this type, immediate value has only 12 bits) and left shifting 20 bits.
One thing to note is that for load type, the arg2 to is of regs_indirect_addr which is discussed before, so the value of rs1 and imm should be retrieved from parse_regs_indirect_addr function.

(g) store
  (iii) Description: S type, eg sb rs2[arg1], 0(rs1)[arg2]
  (iv) Implementation
  The opcode is 0100011, which can be done by shifting 0x08 left 2 bits and add 0x03. Since opcode is [0:6], so no left shift should be made.
  rs1 and rs2 can be retrieved by using reg_to_num() that turns the register name to integer number, then shifting 15 and 20 bits respectively since the start from bit 15 and 20.
  For sb, sh, sw, func3 is 0, 1, 2 repectively, so we just need to shift the corresponding value 12 bits since func3 starts at bit 12.
  Immediate value can be got by doing MASK11_0 to the return value of validate_imm to only get the first 12 bits of the return (because in this type, immediate value has only 12 bits) and left shifting 7 and 20 bits to the corresponding lower and higher position.
  One thing to note is that for store type, the arg2 to is of regs_indirect_addr which is discussed before, so the value of rs1 and imm should be retrieved from parse_regs_indirect_addr function.

4. Sidetrack: bug report
(a)
When reading the code, I found that in the is_opcode function, =MISMATCH is the integer 13, and the no. 13 element in opcode table is sra. So if the opcode is wrong, it will return "sra".



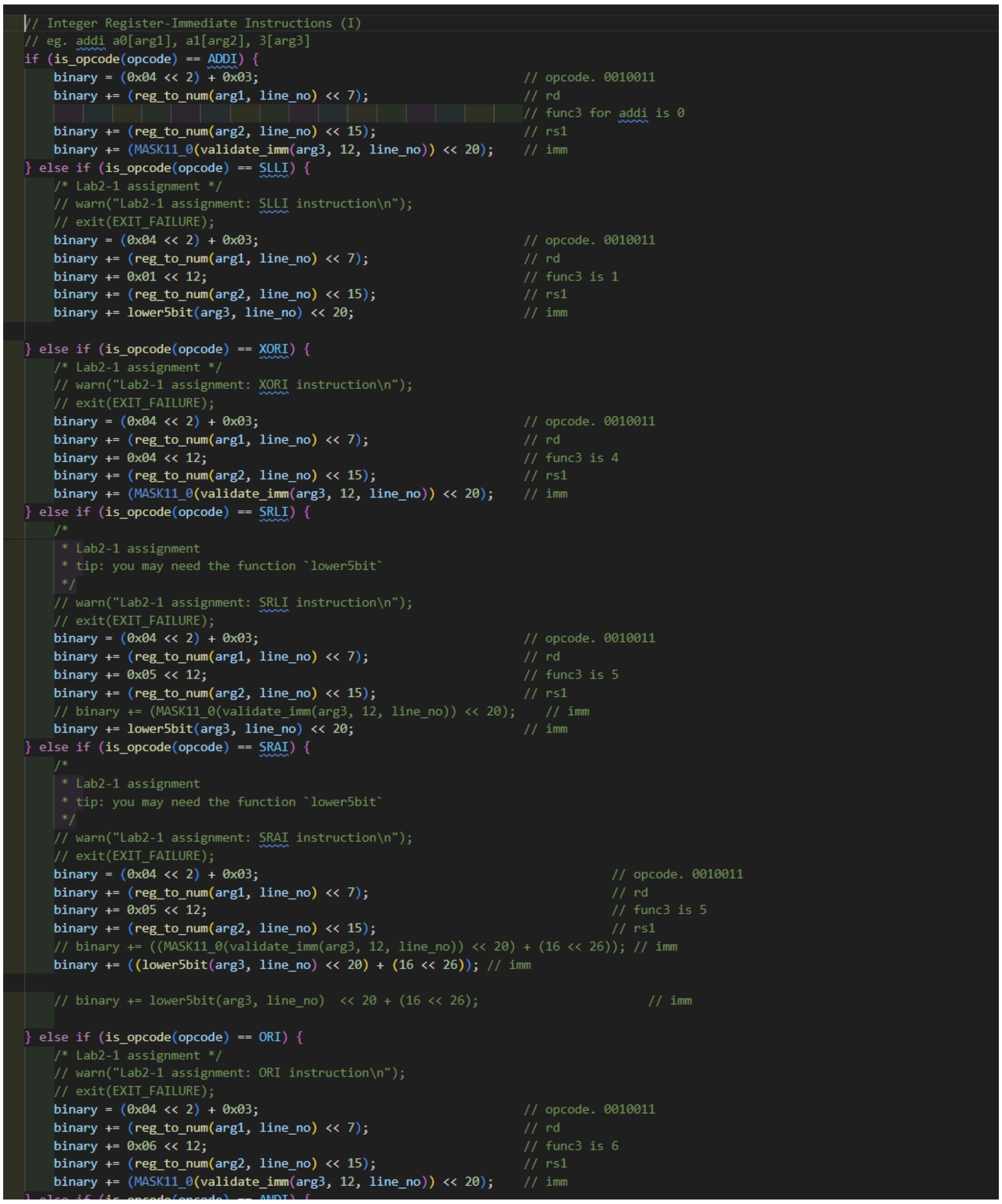
```
int is_opcode(char *ptr) {
    if (*ptr == '\0')
        return MISMATCH;

    size_t len_of_opcode_table = sizeof(opcode_table) / sizeof(opcode_table[0]);
    for (int i = 0; i < (int)len_of_opcode_table; i++) {
        if (cmp_str(opcode_table[i], ptr)) {
            /* i is equal to the enumeration of opcodes */
            return i;
        }
    }
    /* if the opcode is invalid */
    return MISMATCH;
```

I found the bug and reported it on the piazza. The solution is that I can use a bigger number to represent MISMATCH.

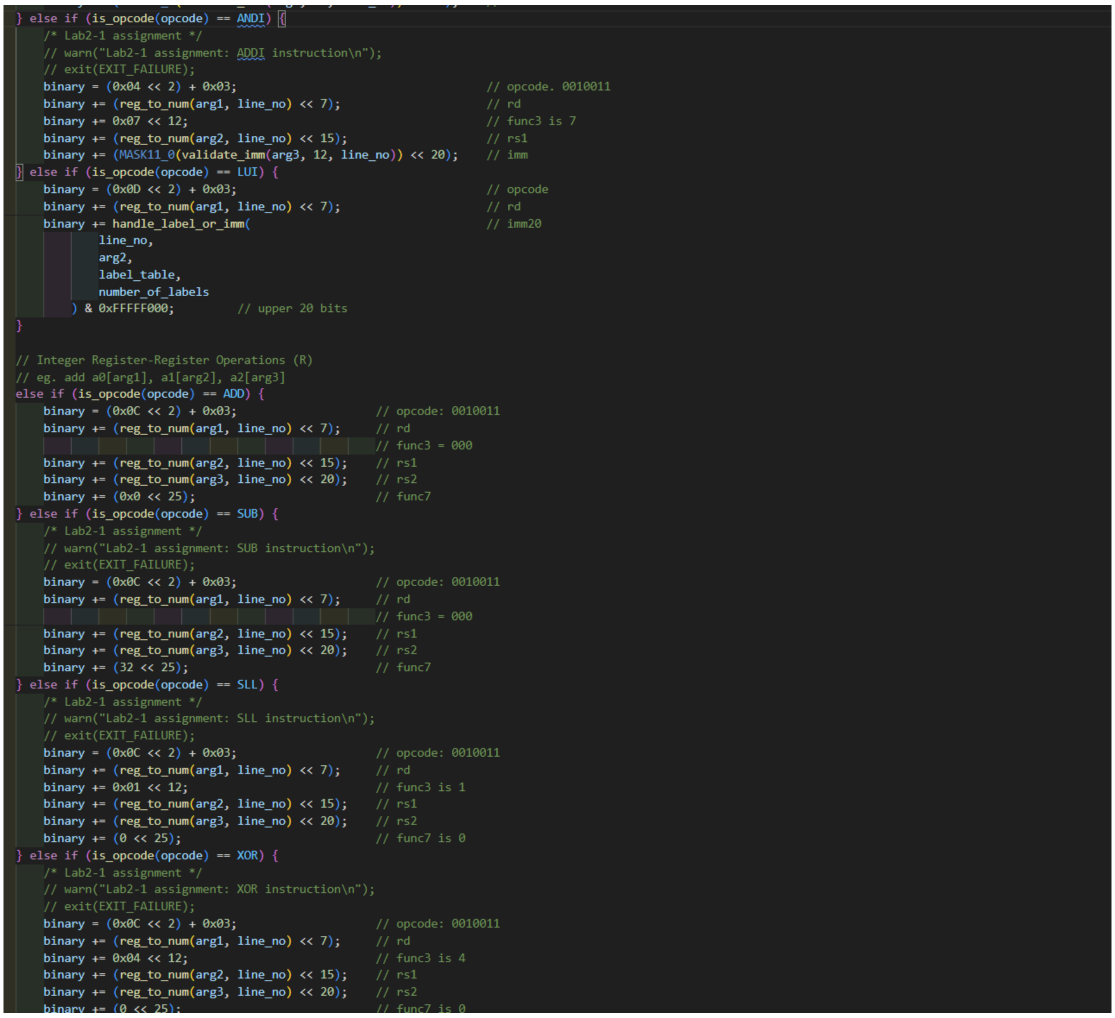
(b)

In the document for shift instructions, we use 31..26(6 bits) to distinguish arithmetic and logic shift, and 25:20(6 bits) to represent the shift amount. However, since the shift amount can only be 5 bits at most (0~31), the 25th bit is left uncared. I asked on the piazza and found that this don't really cause an error. But it is still better to write as 31..26 to be 32.

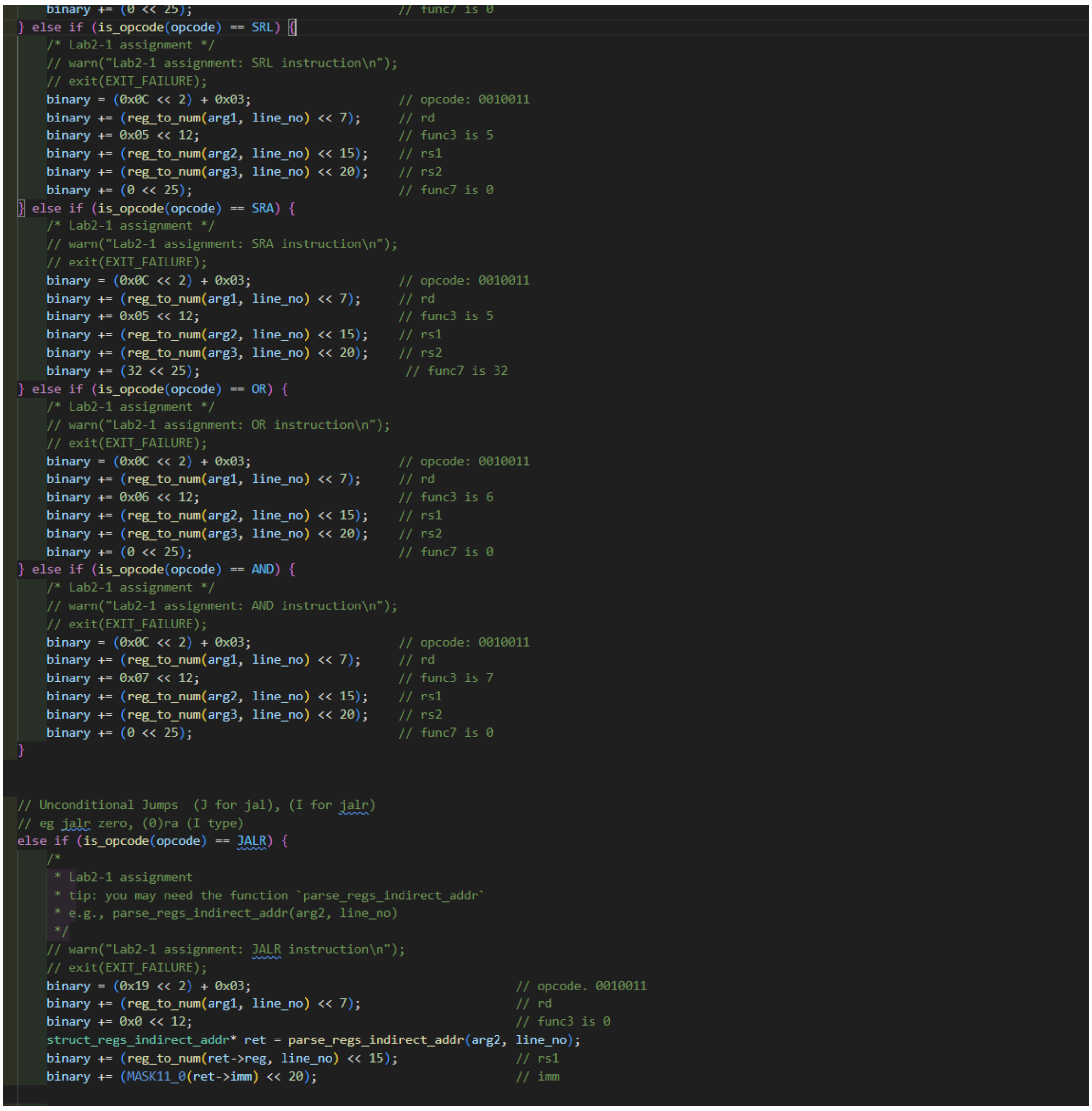
5. Main code:



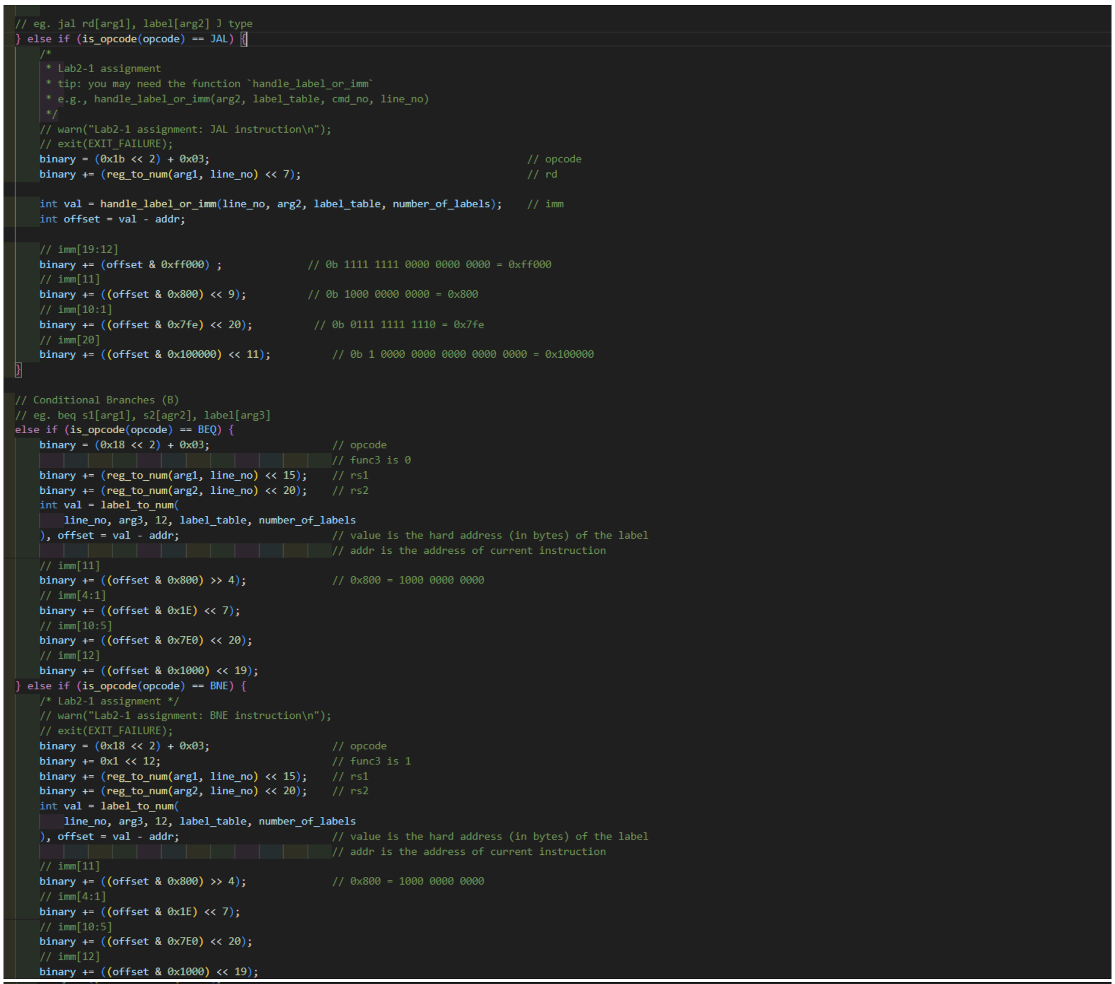
```c
// Integer Register-Immediate Instructions (I)
// eg. addi a0[arg1], a1[arg2], 3[arg3]
if (is_opcode(opcode) == ADDI) {
    binary = (0x04 << 2) + 0x03;                                    // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);                     // rd
                                                                    // func3 for addi is 0
    binary += (reg_to_num(arg2, line_no) << 15);                    // rs1
    binary += (MASK11_0(validate_imm(arg3, 12, line_no)) << 20);    // imm
} else if (is_opcode(opcode) == SLLI) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: SLLI instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x04 << 2) + 0x03;                                    // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);                     // rd
    binary += 0x01 << 12;                                           // func3 is 1
    binary += (reg_to_num(arg2, line_no) << 15);                    // rs1
    binary += lower5bit(arg3, line_no) << 20;                       // imm

} else if (is_opcode(opcode) == XORI) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: XORI instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x04 << 2) + 0x03;                                    // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);                     // rd
    binary += 0x04 << 12;                                           // func3 is 4
    binary += (reg_to_num(arg2, line_no) << 15);                    // rs1
    binary += (MASK11_0(validate_imm(arg3, 12, line_no)) << 20);    // imm
} else if (is_opcode(opcode) == SRLI) {
    /*
     * Lab2-1 assignment
     * tip: you may need the function `lower5bit`
     */
    // warn("Lab2-1 assignment: SRLI instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x04 << 2) + 0x03;                                    // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);                     // rd
    binary += 0x05 << 12;                                           // func3 is 5
    binary += (reg_to_num(arg2, line_no) << 15);                    // rs1
    // binary += (MASK11_0(validate_imm(arg3, 12, line_no)) << 20);    // imm
    binary += lower5bit(arg3, line_no) << 20;                       // imm
} else if (is_opcode(opcode) == SRAI) {
    /*
     * Lab2-1 assignment
     * tip: you may need the function `lower5bit`
     */
    // warn("Lab2-1 assignment: SRAI instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x04 << 2) + 0x03;                                                // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);                                 // rd
    binary += 0x05 << 12;                                                       // func3 is 5
    binary += (reg_to_num(arg2, line_no) << 15);                                // rs1
    // binary += ((MASK11_0(validate_imm(arg3, 12, line_no)) << 20) + (16 << 26)); // imm
    binary += ((lower5bit(arg3, line_no) << 20) + (16 << 26)); // imm

    // binary += lower5bit(arg3, line_no)  << 20 + (16 << 26);                       // imm

} else if (is_opcode(opcode) == ORI) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: ORI instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x04 << 2) + 0x03;                                    // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);                     // rd
    binary += 0x06 << 12;                                           // func3 is 6
    binary += (reg_to_num(arg2, line_no) << 15);                    // rs1
    binary += (MASK11_0(validate_imm(arg3, 12, line_no)) << 20);    // imm
} else if (is_opcode(opcode) == ANDI) {
```



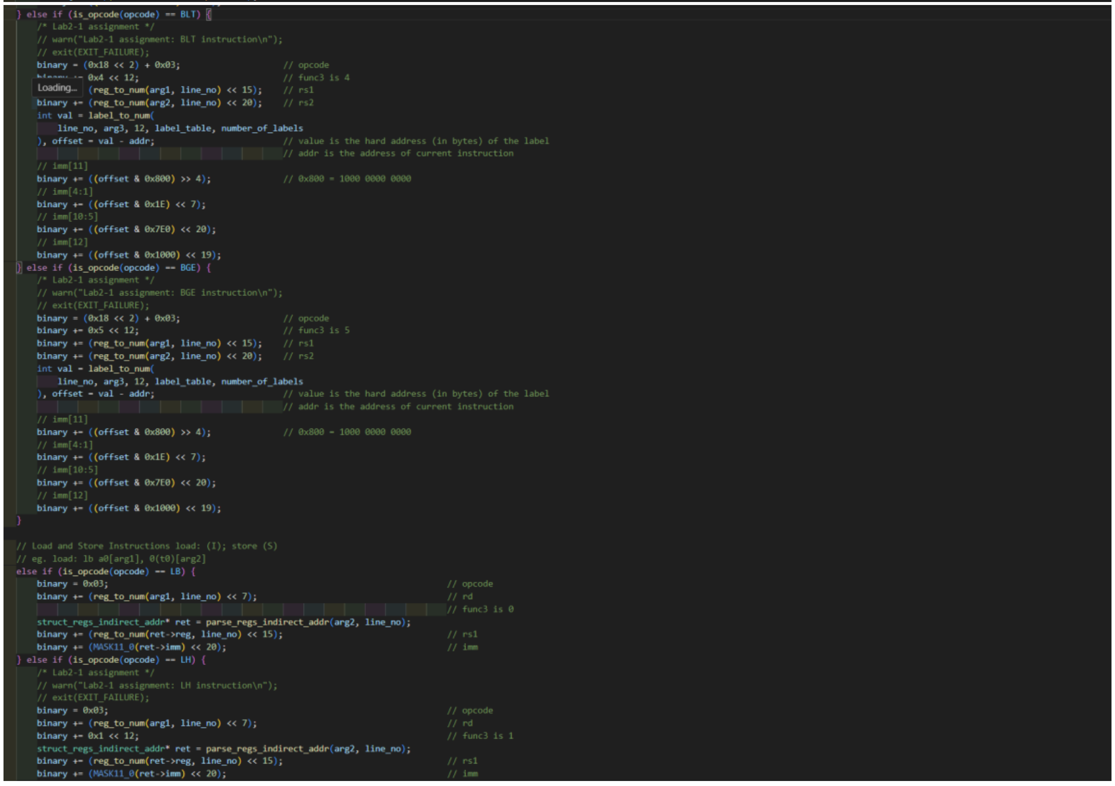
```c
} else if (is_opcode(opcode) == ANDI) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: ADDI instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x04 << 2) + 0x03;                                  // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);                   // rd
    binary += 0x07 << 12;                                         // func3 is 7
    binary += (reg_to_num(arg2, line_no) << 15);                  // rs1
    binary += (MASK11_0(validate_imm(arg3, 12, line_no)) << 20);  // imm
} else if (is_opcode(opcode) == LUI) {
    binary = (0x0D << 2) + 0x03;                                  // opcode
    binary += (reg_to_num(arg1, line_no) << 7);                   // rd
    binary += handle_label_or_imm(                                // imm20
            line_no,
            arg2,
            label_table,
            number_of_labels
        ) & 0xFFFFF000;         // upper 20 bits
}

// Integer Register-Register Operations (R)
// eg. add a0[arg1], a1[arg2], a2[arg3]
else if (is_opcode(opcode) == ADD) {
    binary = (0x0C << 2) + 0x03;                 // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);  // rd
                                                 // func3 = 000
    binary += (reg_to_num(arg2, line_no) << 15); // rs1
    binary += (reg_to_num(arg3, line_no) << 20); // rs2
    binary += (0x0 << 25);                       // func7
} else if (is_opcode(opcode) == SUB) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: SUB instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x0C << 2) + 0x03;                 // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);  // rd
                                                 // func3 = 000
    binary += (reg_to_num(arg2, line_no) << 15); // rs1
    binary += (reg_to_num(arg3, line_no) << 20); // rs2
    binary += (32 << 25);                        // func7
} else if (is_opcode(opcode) == SLL) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: SLL instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x0C << 2) + 0x03;                 // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);  // rd
    binary += 0x01 << 12;                        // func3 is 1
    binary += (reg_to_num(arg2, line_no) << 15); // rs1
    binary += (reg_to_num(arg3, line_no) << 20); // rs2
    binary += (0 << 25);                         // func7 is 0
} else if (is_opcode(opcode) == XOR) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: XOR instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x0C << 2) + 0x03;                 // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);  // rd
    binary += 0x04 << 12;                        // func3 is 4
    binary += (reg_to_num(arg2, line_no) << 15); // rs1
    binary += (reg_to_num(arg3, line_no) << 20); // rs2
    binary += (0 << 25);                         // func7 is 0
```



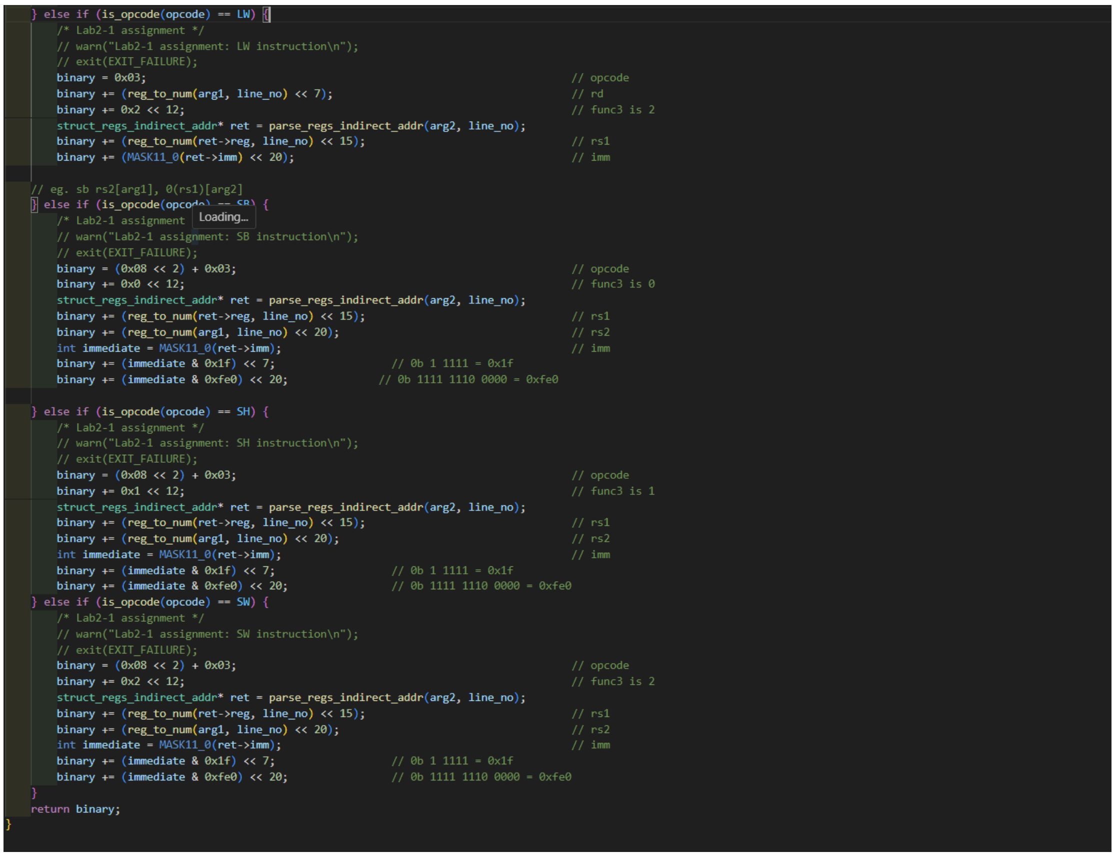
```c
    binary += (0 << 25);                          // func7 is 0
} else if (is_opcode(opcode) == SRL) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: SRL instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x0C << 2) + 0x03;                  // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);   // rd
    binary += 0x05 << 12;                         // func3 is 5
    binary += (reg_to_num(arg2, line_no) << 15);  // rs1
    binary += (reg_to_num(arg3, line_no) << 20);  // rs2
    binary += (0 << 25);                          // func7 is 0
} else if (is_opcode(opcode) == SRA) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: SRA instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x0C << 2) + 0x03;                  // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);   // rd
    binary += 0x05 << 12;                         // func3 is 5
    binary += (reg_to_num(arg2, line_no) << 15);  // rs1
    binary += (reg_to_num(arg3, line_no) << 20);  // rs2
    binary += (32 << 25);                          // func7 is 32
} else if (is_opcode(opcode) == OR) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: OR instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x0C << 2) + 0x03;                  // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);   // rd
    binary += 0x06 << 12;                         // func3 is 6
    binary += (reg_to_num(arg2, line_no) << 15);  // rs1
    binary += (reg_to_num(arg3, line_no) << 20);  // rs2
    binary += (0 << 25);                          // func7 is 0
} else if (is_opcode(opcode) == AND) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: AND instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x0C << 2) + 0x03;                  // opcode: 0010011
    binary += (reg_to_num(arg1, line_no) << 7);   // rd
    binary += 0x07 << 12;                         // func3 is 7
    binary += (reg_to_num(arg2, line_no) << 15);  // rs1
    binary += (reg_to_num(arg3, line_no) << 20);  // rs2
    binary += (0 << 25);                          // func7 is 0
}


// Unconditional Jumps  (J for jal), (I for jalr)
// eg jalr zero, (0)ra (I type)
else if (is_opcode(opcode) == JALR) {
    /*
     * Lab2-1 assignment
     * tip: you may need the function `parse_regs_indirect_addr`
     * e.g., parse_regs_indirect_addr(arg2, line_no)
     */
    // warn("Lab2-1 assignment: JALR instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x19 << 2) + 0x03;                              // opcode. 0010011
    binary += (reg_to_num(arg1, line_no) << 7);               // rd
    binary += 0x0 << 12;                                      // func3 is 0
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (reg_to_num(ret->reg, line_no) << 15);          // rs1
    binary += (MASK11_0(ret->imm) << 20);                     // imm
```



```c
// eg. jal rd[arg1], label[arg2] J type
} else if (is_opcode(opcode) == JAL) {
    /*
     * Lab2-1 assignment
     * tip: you may need the function `handle_label_or_imm`
     * e.g., handle_label_or_imm(arg2, label_table, cmd_no, line_no)
     */
    // warn("Lab2-1 assignment: JAL instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x1b << 2) + 0x03;                                            // opcode
    binary += (reg_to_num(arg1, line_no) << 7);                             // rd

    int val = handle_label_or_imm(line_no, arg2, label_table, number_of_labels);    // imm
    int offset = val - addr;

    // imm[19:12]
    binary += (offset & 0xff000) ;              // 0b 1111 1111 0000 0000 0000 = 0xff000
    // imm[11]
    binary += ((offset & 0x800) << 9);          // 0b 1000 0000 0000 = 0x800
    // imm[10:1]
    binary += ((offset & 0x7fe) << 20);          // 0b 0111 1111 1110 = 0x7fe
    // imm[20]
    binary += ((offset & 0x100000) << 11);          // 0b 1 0000 0000 0000 0000 0000 = 0x100000
}

// Conditional Branches (B)
// eg. beq s1[arg1], s2[agr2], label[arg3]
else if (is_opcode(opcode) == BEQ) {
    binary = (0x18 << 2) + 0x03;                    // opcode
                                                    // func3 is 0
    binary += (reg_to_num(arg1, line_no) << 15);    // rs1
    binary += (reg_to_num(arg2, line_no) << 20);    // rs2
    int val = label_to_num(
        line_no, arg3, 12, label_table, number_of_labels
    ), offset = val - addr;                         // value is the hard address (in bytes) of the label
                                                    // addr is the address of current instruction
    // imm[11]
    binary += ((offset & 0x800) >> 4);              // 0x800 = 1000 0000 0000
    // imm[4:1]
    binary += ((offset & 0x1E) << 7);
    // imm[10:5]
    binary += ((offset & 0x7E0) << 20);
    // imm[12]
    binary += ((offset & 0x1000) << 19);
} else if (is_opcode(opcode) == BNE) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: BNE instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x18 << 2) + 0x03;                    // opcode
    binary += 0x1 << 12;                            // func3 is 1
    binary += (reg_to_num(arg1, line_no) << 15);    // rs1
    binary += (reg_to_num(arg2, line_no) << 20);    // rs2
    int val = label_to_num(
        line_no, arg3, 12, label_table, number_of_labels
    ), offset = val - addr;                         // value is the hard address (in bytes) of the label
                                                    // addr is the address of current instruction
    // imm[11]
    binary += ((offset & 0x800) >> 4);              // 0x800 = 1000 0000 0000
    // imm[4:1]
    binary += ((offset & 0x1E) << 7);
    // imm[10:5]
    binary += ((offset & 0x7E0) << 20);
    // imm[12]
    binary += ((offset & 0x1000) << 19);
```



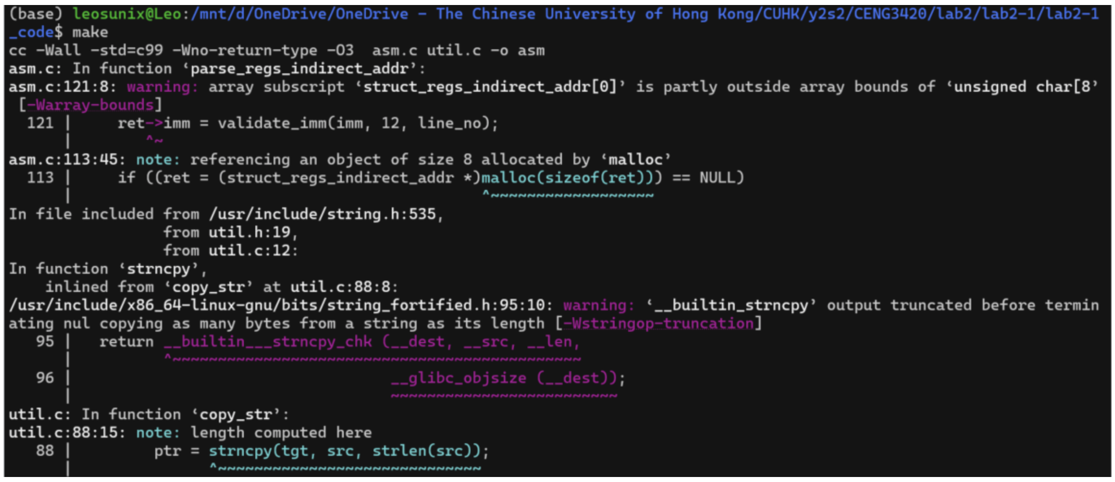
```c
} else if (is_opcode(opcode) == BLT) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: BLT instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x18 << 2) + 0x03;                    // opcode
    binary += 0x4 << 12;                            // func3 is 4
   Loading... (reg_to_num(arg1, line_no) << 15);    // rs1
    binary += (reg_to_num(arg2, line_no) << 20);    // rs2
    int val = label_to_num(
        line_no, arg3, 12, label_table, number_of_labels
    ), offset = val - addr;                         // value is the hard address (in bytes) of the label
                                                    // addr is the address of current instruction
    // imm[11]
    binary += ((offset & 0x800) >> 4);              // 0x800 = 1000 0000 0000
    // imm[4:1]
    binary += ((offset & 0x1E) << 7);
    // imm[10:5]
    binary += ((offset & 0x7E0) << 20);
    // imm[12]
    binary += ((offset & 0x1000) << 19);
} else if (is_opcode(opcode) == BGE) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: BGE instruction\n");
    // exit(EXIT_FAILURE);
    binary = (0x18 << 2) + 0x03;                    // opcode
    binary += 0x5 << 12;                            // func3 is 5
    binary += (reg_to_num(arg1, line_no) << 15);    // rs1
    binary += (reg_to_num(arg2, line_no) << 20);    // rs2
    int val = label_to_num(
        line_no, arg3, 12, label_table, number_of_labels
    ), offset = val - addr;                         // value is the hard address (in bytes) of the label
                                                    // addr is the address of current instruction
    // imm[11]
    binary += ((offset & 0x800) >> 4);              // 0x800 = 1000 0000 0000
    // imm[4:1]
    binary += ((offset & 0x1E) << 7);
    // imm[10:5]
    binary += ((offset & 0x7E0) << 20);
    // imm[12]
    binary += ((offset & 0x1000) << 19);
}

// Load and Store Instructions load: (I); store (S)
// eg. load: lb a0[arg1], 0(t0)[arg2]
else if (is_opcode(opcode) == LB) {
    binary = 0x03;                                                          // opcode
    binary += (reg_to_num(arg1, line_no) << 7);                             // rd
                                                                            // func3 is 0
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (reg_to_num(ret->reg, line_no) << 15);                        // rs1
    binary += (MASK11_0(ret->imm) << 20);                                   // imm
} else if (is_opcode(opcode) == LH) {
    /* Lab2-1 assignment */
    // warn("Lab2-1 assignment: LH instruction\n");
    // exit(EXIT_FAILURE);
    binary = 0x03;                                                          // opcode
    binary += (reg_to_num(arg1, line_no) << 7);                             // rd
    binary += 0x1 << 12;                                                    // func3 is 1
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (reg_to_num(ret->reg, line_no) << 15);                        // rs1
    binary += (MASK11_0(ret->imm) << 20);                                   // imm
```



```c
    } else if (is_opcode(opcode) == LW) {
        /* Lab2-1 assignment */
        // warn("Lab2-1 assignment: LW instruction\n");
        // exit(EXIT_FAILURE);
        binary = 0x03;                                                          // opcode
        binary += (reg_to_num(arg1, line_no) << 7);                             // rd
        binary += 0x2 << 12;                                                    // func3 is 2
        struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
        binary += (reg_to_num(ret->reg, line_no) << 15);                        // rs1
        binary += (MASK11_0(ret->imm) << 20);                                   // imm

    // eg. sb rs2[arg1], 0(rs1)[arg2]
    } else if (is_opcode(opcode) == SB) {
        /* Lab2-1 assignment  Loading...
        // warn("Lab2-1 assignment: SB instruction\n");
        // exit(EXIT_FAILURE);
        binary = (0x08 << 2) + 0x03;                                            // opcode
        binary += 0x0 << 12;                                                    // func3 is 0
        struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
        binary += (reg_to_num(ret->reg, line_no) << 15);                        // rs1
        binary += (reg_to_num(arg1, line_no) << 20);                            // rs2
        int immediate = MASK11_0(ret->imm);                                     // imm
        binary += (immediate & 0x1f) << 7;                  // 0b 1 1111 = 0x1f
        binary += (immediate & 0xfe0) << 20;              // 0b 1111 1110 0000 = 0xfe0

    } else if (is_opcode(opcode) == SH) {
        /* Lab2-1 assignment */
        // warn("Lab2-1 assignment: SH instruction\n");
        // exit(EXIT_FAILURE);
        binary = (0x08 << 2) + 0x03;                                            // opcode
        binary += 0x1 << 12;                                                    // func3 is 1
        struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
        binary += (reg_to_num(ret->reg, line_no) << 15);                        // rs1
        binary += (reg_to_num(arg1, line_no) << 20);                            // rs2
        int immediate = MASK11_0(ret->imm);                                     // imm
        binary += (immediate & 0x1f) << 7;                  // 0b 1 1111 = 0x1f
        binary += (immediate & 0xfe0) << 20;                // 0b 1111 1110 0000 = 0xfe0
    } else if (is_opcode(opcode) == SW) {
        /* Lab2-1 assignment */
        // warn("Lab2-1 assignment: SW instruction\n");
        // exit(EXIT_FAILURE);
        binary = (0x08 << 2) + 0x03;                                            // opcode
        binary += 0x2 << 12;                                                    // func3 is 2
        struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
        binary += (reg_to_num(ret->reg, line_no) << 15);                        // rs1
        binary += (reg_to_num(arg1, line_no) << 20);                            // rs2
        int immediate = MASK11_0(ret->imm);                                     // imm
        binary += (immediate & 0x1f) << 7;                  // 0b 1 1111 = 0x1f
        binary += (immediate & 0xfe0) << 20;                // 0b 1111 1110 0000 = 0xfe0
    }
    return binary;
}
```

6. Console results

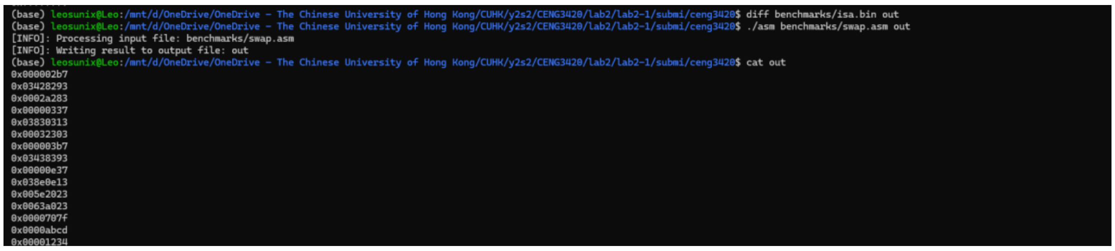
Make:



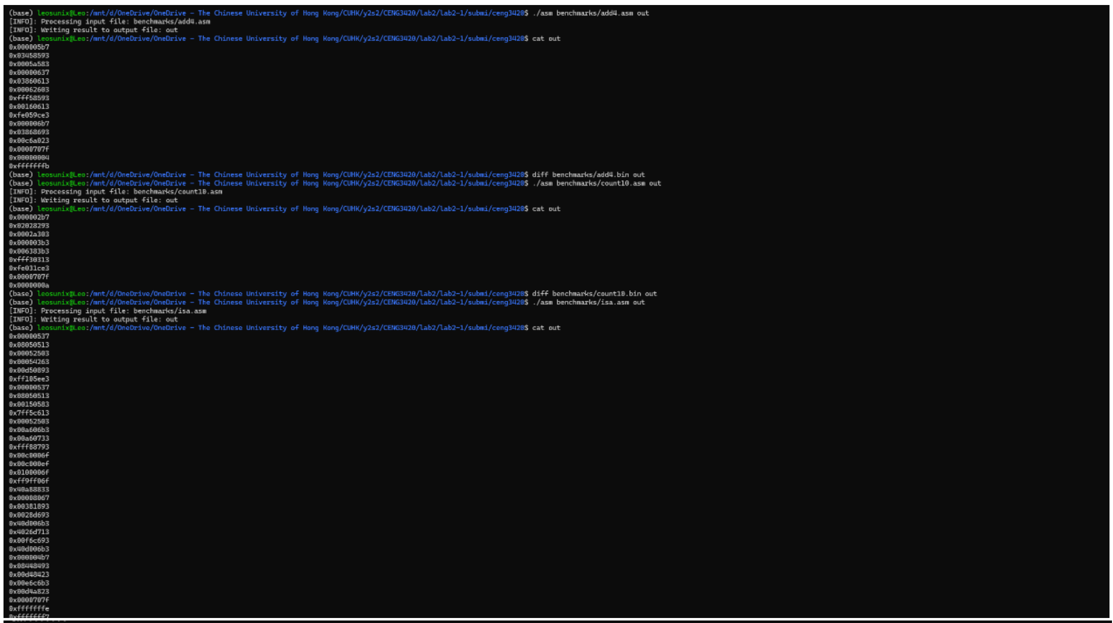
```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/lab2-1
_code$ make
cc -Wall -std=c99 -Wno-return-type -O3  asm.c util.c -o asm
asm.c: In function ‘parse_regs_indirect_addr’:
asm.c:121:8: warning: array subscript ‘struct_regs_indirect_addr[0]’ is partly outside array bounds of ‘unsigned char[8’
 [-Warray-bounds]
  121 |     ret->imm = validate_imm(imm, 12, line_no);
      |        ^~
asm.c:113:45: note: referencing an object of size 8 allocated by ‘malloc’
  113 |     if ((ret = (struct_regs_indirect_addr *)malloc(sizeof(ret))) == NULL)
      |                                             ^~~~~~~~~~~~~~~~~~~
In file included from /usr/include/string.h:535,
                 from util.h:19,
                 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 termin
ating 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.

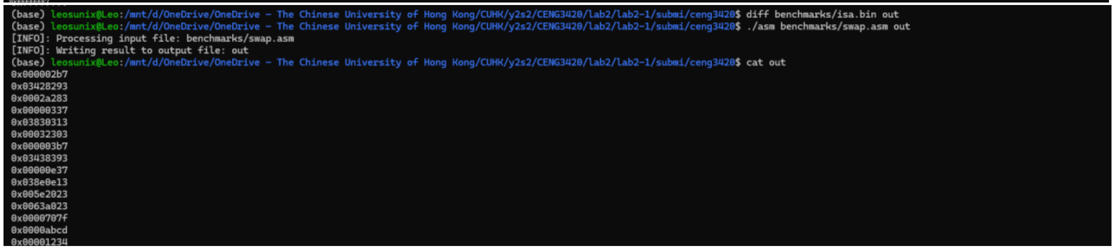
Running and comparing with diff:



```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ diff benchmarks/isa.bin out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ ./asm benchmarks/swap.asm out
[INFO]: Processing input file: benchmarks/swap.asm
[INFO]: Writing result to output file: out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ cat out
0x000002b7
0x03428293
0x0002a283
0x00000337
0x03830313
0x00032303
0x000003b7
0x03438393
0x00000e37
0x038e0e13
0x005e2023
0x0063a023
0x0000707f
0x0000abcd
0x00001234
```

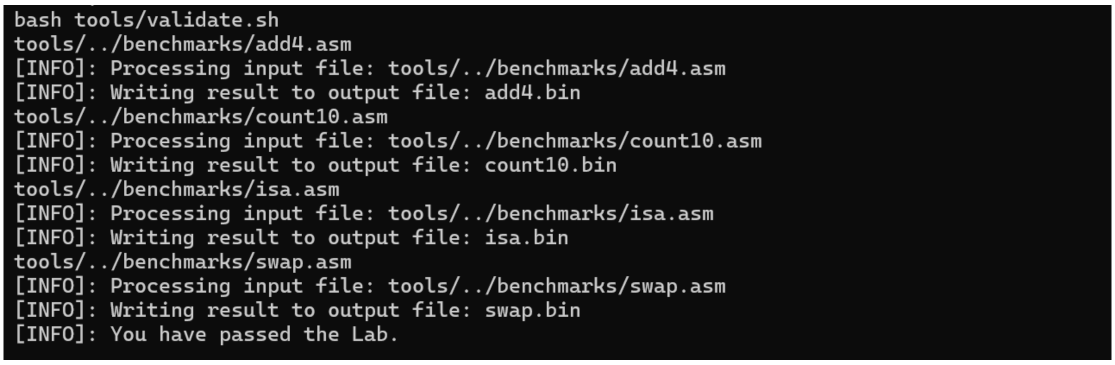


```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ ./asm benchmarks/add4.asm out
[INFO]: Processing input file: benchmarks/add4.asm
[INFO]: Writing result to output file: out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ cat out
0x000005b7
0x03458593
0x0005a583
0x00000637
0x03860613
0x00062603
0xfff58593
0x00160613
0xfe059ce3
0x000006b7
0x03868693
0x00c6a023
0x0000707f
0x00000004
0xfffffffb
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ diff benchmarks/add4.bin out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ ./asm benchmarks/count10.asm out
[INFO]: Processing input file: benchmarks/count10.asm
[INFO]: Writing result to output file: out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ cat out
0x000002b7
0x02028293
0x0002a303
0x000003b3
0x006383b3
0xfff30313
0xfe031ce3
0x0000707f
0x0000000a
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ diff benchmarks/count10.bin out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ ./asm benchmarks/isa.asm out
[INFO]: Processing input file: benchmarks/isa.asm
[INFO]: Writing result to output file: out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ cat out
0x00000537
0x08050513
0x00052503
0x00054263
0x00d50893
0xff105ee3
0x00000537
0x08050513
0x00150583
0x7ff5c613
0x00052503
0x00a606b3
0x00a60733
0xfff88793
0x00c0006f
0x00c000ef
0x0100006f
0xff9ff06f
0x40a88833
0x00008067
0x00381893
0x0028d693
0x40d806b3
0x4026d713
0x00f6c693
0x40d806b3
0x000004b7
0x00448493
0x00d48423
0x00e6c6b3
0x00d4a823
0x0000707f
0xfffffffe
0xfffffff7
```



```
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ diff benchmarks/isa.bin out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ ./asm benchmarks/swap.asm out
[INFO]: Processing input file: benchmarks/swap.asm
[INFO]: Writing result to output file: out
(base) leosunix@Leo:/mnt/d/OneDrive/OneDrive - The Chinese University of Hong Kong/CUHK/y2s2/CENG3420/lab2/lab2-1/submi/ceng3420$ cat out
0x000002b7
0x03428293
0x0002a283
0x00000337
0x03830313
0x00032303
0x000003b7
0x03438393
0x00000e37
0x038e0e13
0x005e2023
0x0063a023
0x0000707f
0x0000abcd
0x00001234
```

Using the make validate:



```
bash tools/validate.sh
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
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
[INFO]: You have passed the Lab.
```

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