# 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.

1. 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.

1. Terminologies
2. 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*

1. label

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

1. arch\_regs

the real name and alias for the registers.

1. regs\_indirect\_addr

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

1. binary

The binary machine code after assembly.

1. Data structures
2. 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.

1. Assembly process
2. 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.
3. 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.
4. Some special functions
5. int reg\_to\_num(char \*, int): input a string of reg, return the encoding (the number) of the register.
6. 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).
7. 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.
8. 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.
9. 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.
10. *inst\_to\_binary()* implementation

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

1. Integer Register-Immediate Instructions except for *lui*
2. Description: I type instructions, all the opcode is 0010011, and have rd, rs1, imm.

Example: addi a0[arg1], a1[arg2], 3[arg3].

1. 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.

1. 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 2­­5-1 = 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.

1. 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.

1. *lui* instruction
2. Description: U type instruction, all the opcode is 0110111, and have 20-bit imm.
3. 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.

1. Integer Register-Regiser Instructions
2. Description: R type instructions, all the opcode is 0110011, and have rd, rs1, rs2.

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

1. *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.

1. 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.

1. 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.

1. Unconditional Jumps
2. *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.

1. *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.

1. Conditional Branches
2. Description: B type, eg beq s1[arg1], s2[agr2], label[arg3]
3. 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.

1. Load
2. Description: L type, eg lb a0[arg1], 0(t0)[arg2]
3. 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.

1. store
2. Description: S type, eg sb rs2[arg1], 0(rs1)[arg2]
3. 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.

1. 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".

A computer screen with text

Description automatically generated

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 ask 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.

1. Main code:

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

A computer screen shot of text

Description automatically generated

1. Console results

Make:

A computer screen with white and blue text

Description automatically generated

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 to diff:

A blue lines on a black background

Description automatically generated

Using the make validate:

A screen shot of a computer

Description automatically generated

# 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