# 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.

1. 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.

1. Terminologies and data structures
2. 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.

1. 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.

1. 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 I byte that is the same with the real situation.

1. Important functions and MACROS
2. 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.

1. 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:

A screen shot of a computer program

Description automatically generated

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. Handle specific instructions

In handle\_xxx() functions we perform the decode, exe, wb of each instructions.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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 20th 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.

1. 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 17th 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 31st 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.

1. 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 nth(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].

1. 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 nth(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.

1. My mistakes

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

1. 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.

1. 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.

1. Main code:
2. Handle\_instruction

A black screen with a black background

Description automatically generated

A black screen with many small lines

Description automatically generated with medium confidence

1. Hande\_xxx

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A black screen with white text

Description automatically generated

A black screen with green and blue lines

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

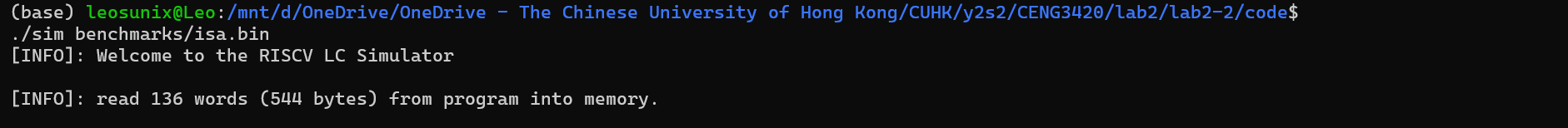
1. Console results
2. Make:

A computer screen with text on it

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.

1. Isa.bin



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Count10

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Swap

A screen shot of a computer

Description automatically generated

1. Add4

A computer screen with white text

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