# Lab 3.2 Report

1. Problem analysis

In lab 3.1 we are implementing the simulator of RISCV-LC, especially the memory operation part and the part that sends the value via bus.

1. My understandings of the RISCV-LC simulator structure
2. My understanding of the instruction execution process:

Just as in lab 3.1, in the cycle instruction, the core steps of the simulator are executed. In lab 3.2, we assume that other parts are already implemented, and we mainly focus on cycle\_memory and latch\_datapath\_values.

1. My understanding of some key implementations in the code
2. Some commonly used functions and macros
3. datasize\_mux ():

This function emulates the data size multiplexer. The goal of it is to tell what is the size of the data that the memory operation should operate on.

If the multiplexer chooses 0 (DATASIZE==0), then it will directly return 0. Else it will return the value based on the func3 of the instructions.

For memory operations. lb, lbu and sb are the instructions that are operating on bytes. Their func3 are 0x0, 0x4, 0x0 (0b000, 0b100, 0b000) respectively. So after the operation ~(funct3 & 0x3), the return value is always 0b 1111 1111 1111 1111 1111 1111 1111 1111, or -1.

For memory operations, lh, lhu, sh are the instructions that are operating on 2 bytes. Their func3 are 0x1, 0x5, 0x1 (0b001, 0x101, 0b001) respectively. So after the operation ~(funct3 & 0x3), the return value is always 0b 1111 1111 1111 1111 1111 1111 1111 1110, or -2.

For memory operations, lw, sw are the instructions that are operating on 4 bytes. Their func3 are 0x2 (0b10) respectively. So after the operation ~(funct3 & 0x3), the return value is always 0b 1111 1111 1111 1111 1111 1111 1111 1101, or -3.

1. ADDRnMUX

Functions addr1\_mux and addr2\_mux emulates the multiplexer ADDR1MUX and ADDR2MUX. The input are the potential output values of each multiplexers and the control value. The control value can be retrieved by the MACRO get\_ADDR1MUX and get\_ADDR2MUX.

1. Some key variables
2. MEM\_VAL: the temporary variable that stores the memory after each cycle finishes.
3. W: the indicator for writing enabled. If mio & W then it means we should write to the memory; if mio & !W then it means that we should do memory IO but we shouldn’t do memory write, which means we should read the memory.
4. BUS: the value of bus.
5. Handle cycle\_memory()

cycle\_memory() is the function to handle the operations for writing or reading to the main memory.

1. Writing

As discussed above, when W is true, we should write.

First, we should get the value of funct3, because the funct3 decides how many bytes we should manipulate. This can be done by using the mask\_val function to retrieve the 14..12 bits of the instructions.

Then by utilizing the datasize\_mux discussed above, with control signal get\_DATASIZE(CURRENT\_LATCHES.MICROINSTRUCTION), and input funct3, 0, we can map the result to how many bytes we should work on. Since it is little endian, the higher bits should be placed in higher addresses.

1. If the result is -1, we should write only one byte.

The first byte is the bits 7..0 of MDR, and the destination address is CURRENT\_LATCHES.MAR, so we can use MASK7\_0() to retrieve it, and assign it to MEMORY[CURRENT\_LATCHES.MAR].

1. If the result is -2, we should write the first 2 bytes.

The first byte is the bits 7..0 of MDR, and the destination address is CURRENT\_LATCHES.MAR, so we can use MASK7\_0() to retrieve it, and assign it to MEMORY[CURRENT\_LATCHES.MAR].

The second byte is the bits 15..8 of MDR, and the destination address is CURRENT\_LATCHES.MAR + 1, so we can use MASK15\_8() to retrieve it, and assign it to MEMORY[CURRENT\_LATCHES.MAR+1].

1. Otherwise, we should write all 4 bytes

The first byte is the bits 7..0 of MDR, and the destination address is CURRENT\_LATCHES.MAR, so we can use MASK7\_0() to retrieve it, and assign it to MEMORY[CURRENT\_LATCHES.MAR].

The second byte is the bits 15..8 of MDR, and the destination address is CURRENT\_LATCHES.MAR + 1, so we can use MASK15\_8() to retrieve it, and assign it to MEMORY[CURRENT\_LATCHES.MAR+1].

The third byte is the bits 23..16 of MDR, and the destination address is CURRENT\_LATCHES.MAR+2, so we can use MASK23\_116() to retrieve it, and assign it to MEMORY[CURRENT\_LATCHES.MAR+2].

The fourth byte is the bits 31..24 of MDR, and the destination address is CURRENT\_LATCHES.MAR + 3, so we can use MASK31\_24() to retrieve it, and assign it to MEMORY[CURRENT\_LATCHES.MAR+3].

The implementation is shown below:

A computer code on a black background

Description automatically generated

1. Reading

If W is false, then we should do reading. Same as writing, there are 3 situations of the output of datasize\_mux. Since it is little endian, the higher bits should be retrieved in higher addresses.

1. If the result is -1, we should read only one byte.

The byte is MEMORY[CURRENT\_LATCHES.MAR], and we can use sext\_unit() to sign extend it.

Then we can assign this value to MEM\_VAL.

1. If the result is -2, we should read 2 bytes.

The first byte is MEMORY[CURRENT\_LATCHES.MAR], the second byte is MEMORY[CURRENT\_LATCHES.MAR+1], so the value is (MEMORY[CURRENT\_LATCHES.MAR]) + (MEMORY[CURRENT\_LATCHES.MAR + 1] << 8. And we can use sext\_unit() to sign extend it.

Then we can assign this value to MEM\_VAL.

1. Otherwise, we should read 4 bytes.

The first byte is MEMORY[CURRENT\_LATCHES.MAR], the second byte is MEMORY[CURRENT\_LATCHES.MAR+1], The third byte is MEMORY[CURRENT\_LATCHES.MAR+2], the fourth byte is MEMORY[CURRENT\_LATCHES.MAR+3], so the value is MEMORY[CURRENT\_LATCHES.MAR]) + (MEMORY[CURRENT\_LATCHES.MAR + 1] << 8) + (MEMORY[CURRENT\_LATCHES.MAR + 2] << 16) + (MEMORY[CURRENT\_LATCHES.MAR + 3] << 24. And we can use sext\_unit() to sign extend it.

Then we can assign this value to MEM\_VAL.

A black screen with many small colored lines

Description automatically generated with medium confidence

1. Handle latch\_datapath\_values()

In this function, we update all the blocks with values from the bus according to LD signals.

1. Update register file

First we need to retrieve the register index. Since rd is in bits 11..7, we can use mask\_val(CURRENT\_LATCHES.IR, 11, 7) to get the index.

Then, we can set the corresponding register to the bus value, by NEXT\_LATCHES.REGS[reg\_idx] = BUS.

The implementation is shown below:

A computer screen with text

Description automatically generated

1. Update MAR

The update of MAR is simple, as long as the LD.MAR bit is true, we just assign the bus value to MAR.

The implementation is shown below:

A computer screen shot of a black screen

Description automatically generated

1. Update IR

The update of IR is simple, as long as the LD.IR bit is true, we just assign the bus value to IR.

The implementation is shown below:

A computer screen shot of a computer program

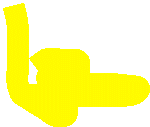
Description automatically generated

1. Update PC

The update of PC is more complex. From the graph we can see that, there are 2 sources from the PC MUX:

A diagram of a computer

Description automatically generated



The first source is the value of PC+4, since it is read when PCMUX is 0, I call it pc\_mux\_0\_value. The other value is the sum of the BUS value and PC-4, since it is read when PCMUX is 1, I call it pc\_mux\_1\_value. These 2 values are chosen depending on the control value of LD.PC.

So the PC value can be retrieved by the function pc\_mux, with parameters get\_PCMUX(CURRENT\_LATCHES.MICROINSTRUCTION), pc\_mux\_0\_valueand pc\_mux\_1\_value.

The implementation is shown below:

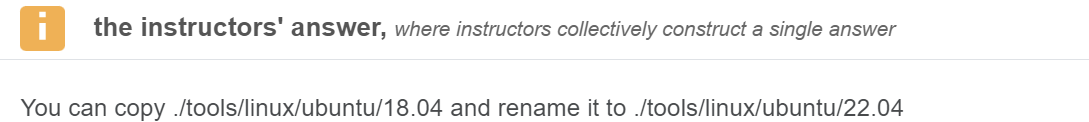
A screen shot of a computer

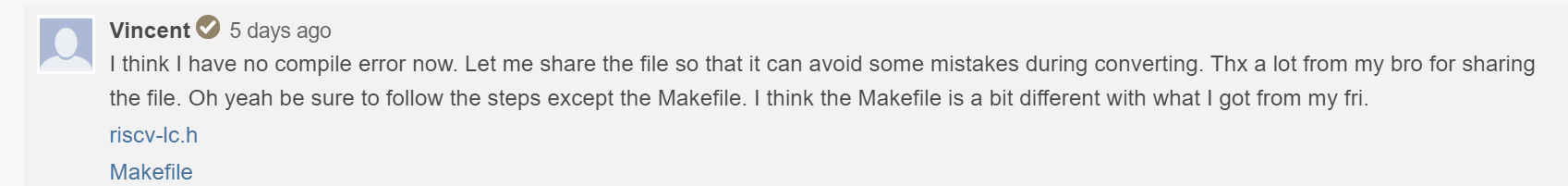
Description automatically generated

1. My mistakes and understandings
2. One mistake I had when doing the lab is that I missed considering the endianness of the data, by little endian rule, the higher bits should be placed in higher addresses, but I got it reversed. Resulting in mistakes.
3. When compiling, I got this mistake:



I checked on the piazza, and I found that it is the problem that the version of my ubuntu isn’t correct. So I followed this instruction and modified my path and Makefile, and the problem is solved.





1. Console results
2. Make:

A black screen with white 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.

1. Isa.bin

Running:

A screen shot of a computer

Description automatically generated

Result:

A computer screen with numbers and letters

Description automatically generated

1. Count10

Running:

A screenshot of a computer

Description automatically generated

Result:

A screenshot of a computer

Description automatically generated

1. Swap

Before:

A screen shot of a computer

Description automatically generated

Running:

A screenshot of a computer program

Description automatically generated

Result:

A black screen with a black background

Description automatically generated

1. Add4

Before

A screenshot of a computer

Description automatically generated

running:

A black background with white text

Description automatically generated

Result:

A black background with white letters and numbers

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

fsm.pdf

riscv-lc.pdf