# Note

Before I heard that we don’t need to include the textual explanations in report, I have already finished the report of Lab3.1 and Lab3.2. I asked on Piazza and I found out that I can still include the textual report in here.

# Lab 3.1 Report

1. Problem analysis

In lab 3.1 we are implementing the simulator of RISCV-LC, especially the finite state machine for stage update. Different from previous code, this time we are separating each instruction into many different stages, and for each stage the 33 control signal bits will be different, and we have to implement how are the 128 states interconnected (which is the finite state machine) and in each state, what are each signal.

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

The program starts with the main function. In the main function, the program first get the binary file from the console and opens it, then it starts to initialize like allocating space for memory, loading the control state into the code 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 core steps of the simulator are executed. The core steps are separated into 5 parts: eval\_micro\_sequencer, cycle\_memory, eval\_bus\_drivers, drive\_bus, datapath\_values. Which are basically to update each parts including PC, memory, BUS and so on in the microarchitecture for each instruction. These steps are determined by the control signals in the file uop.

1. My understanding of the finite state machine

There are 22 states in the finite state machine in total, but these ID (state number) are from 0 to 127, this is first good for future extensions, and also it is because the IR[6:0] and J6~J0 are 7 bits and is easy to map to 0 to 127. In the finite state machine graph there are 21 states, and the HALT state(127) is ignored. From the graph we can know that for each state, what processes are done and what are the next state for each state, so we can implement the uop file.

1. My understanding of the uop file

uop file is the file where we store the exact numbers of each state. It has 128 rows in total, which corresponds to the 128 states with state numbers 0 to 127. For each row, there are 33 columns, which correspond to 33 specific control signals for each states. Control signals include:

1. IRD: it determines if the state number of next state is from the instruction bits [6:0] (when IRD is 1) or from J6~J0 (together with B, READY, etc.) in ROM (when IRD is 0).
2. J6~J0: when IRD is 0, they partly determine the state transition.
3. LD: they basically enable whether each structure (PC, MAR, etc.) read and update their values in the current stage or not.
4. Gate: gate is enable signals for bus tristate driver. If they are enabled (1), it means that on the bus we can read its value, otherwise we disable it (0).
5. MUX: just the selection value of each multiplexer.
6. En: the enable value of output or input for register file and memory
7. RESET: a special signal to indicate whether to reset the whole RISCV-LC.
8. My understanding of some key implementations in the code
9. Some commonly used functions and macros
10. handle\_get\_element\_of\_error(x, id):

By masking and shifting, we can get specific digits of the input, so we can use MACROS to easily get the IRD, J, B etc. values.

1. MUX function:

MUX functions are the simulator of real multiplexers. For example, int blockBMUX(int, int, int): represent the B mux: return B or 0 according to LD.BEN.

1. Some key variables
2. struct\_system\_latches:

The latch is kind of the set that stores all the values for state, including PC, registers, MAR, IR, B, READY, MICROINDTRUCTION, and STATE\_NUMBER. We have 2 struct\_system\_latches which are CURRENT\_LATCHES and NEXT\_LATCHES that stores the value of the current and next stage.

1. Handle hard-wired 0

In the code, we need to handle the hard-wired 0: which means that no matter what operation is done to REG[0], in the next stage, the REG[0] is still zero. This can be done by setting NEXT\_LATCHES.REGS[0] to 0 at the eval\_micro\_sequencer() function:

A black background with green text

Description automatically generated

1. Handle the uop file
2. Preprocessing of the uop file

The uop matrix is really hard to read, so I decide to convert it to an excel file so that I can quickly locate specific rows and columns. So I wrote the to\_csv.c file to convert it to csv file by adding comma in-between:

A screenshot of a computer program

Description automatically generated

And after converting, we can open it in Excel with clearer layout (with other labels in yellow and undetermined values in green):

A white sheet with black dots and yellow lines

Description automatically generated

Then, we can modify each undetermined value.

1. State 1:

A diagram of a memory

Description automatically generated

In state 1, the next state depends on the value of READY, if READY is 0, then next state is 1 (0000001if READY is 1, the next state is 5 (0000101).

A diagram of a control signal

Description automatically generated

Looking at the control part we can see, if we set J0 to 1, and all others to 0, we can achieve this purpose since if J2 is 0, it is actually READY that determines bit2.

In this stage we update MDR to the value of memory at the address of MAR.

Therefore, no PC is updated, so LD.PC is 0;

No MAR is updated, so LD.MAR is 0;

MDR is updated, so LD.MDR is 1;

In this stage, we don’t need to read the values from register file, so RS1EN and RS2EN should both be disabled, set to 0. The RS2MUX should be set to 0 to choose the output of rs2 because in this stage, we don’t want to calculate with ALU and the 2 inputs of the ALU should be disabled. But there is no control for disabling the I immediate generator, so we can only let the RS2MUX choose the disabled RS2 to make the input disabled.

Since we want to read the data from memory to MDR, we should make the memory IO enabled, by setting MIO\_EN to 1.

1. State 2

A diagram of a memory

Description automatically generated

State 2 is almost the same with state 1, the only difference is that if READY is 0, then next state is 2 (0000010if READY is 1, the next state is 6 (0000110).

Looking at the control part we can see, if we set J1 to 1, and all others to 0, we can achieve this purpose since if J2 is 0, it is actually READY that determines bit2.

1. State 32

A diagram of a computer code

Description automatically generated with medium confidence

The next state is 33 (0b 0100001) so J0 and J5 should be set to 1, others 0.

This state, we load the value of rs2 into MDR, therefore,

No PC is updated, so LD.PC is 0;

No MAR is updated, so LD.MAR is 0;

MDR is updated, so LD.MDR is 1;

No instruction is updated, so LD.IR is 0;

No register is updated, so LD.REG is 0.

Since the next state number doesn’t depend on the value of B, so in the multiplexer we set LD.BEN to 0 meaning that we won’t choose the B value to determine the control signal.

For the tristate driver gates, since in this stage the value on the bus is the value of rs2, so only the gate that controls RS2 value should be enabled and all other values are disabled. Meaning that GatePC, GateMAR, GateMDR, GateALUSHF are set to 0, and GateRS2 is set to 1.

Since PC is not updated at this stage, so in theory we don’t need to determine the values of PCMUX, ADDR1MUX, ADDR2MUX because the LD.PC is disabled and these values are not used. But in practice, we set them to the default value, which is 0, meaning that PCMUX=0, ADDR1MUX, ADDR2MUX = 00.

Since the GateMAR is disabled, actually the value of MARMUX is unused and in theory we don’t need to use the value determined by MARMUX, but in practice, we set it to the default value, which is 0, meaning that MARMUX=0.

Since the value of MDR comes from the value of rs2 in the BUS, so that the input of MDR should be from the bus, so that MDRMUX should be 1 to choose from the BUS.

In this stage, we don’t need the value of Selection Logic part, so in theory the input of the ALU determined by RS2MUX doesn’t have to be determined, but in practice, we set it to default 0.

1. State 33

A diagram of a number

Description automatically generated

In state 33, the next stage determination is almost the same with state 1, the only difference is that if READY is 0, then next state is 33 (0100001) if READY is 1, the next state is 37 (0100101).

Looking at the control part we can see, if we set J1 and J5 to 1, and all others to 0, we can achieve this purpose since if J2 is 0, it is actually READY that determines bit2.

In this state, we load the value of MDR to the memory at the address MAR, therefore, no register is updated, so LD.REG is 0;

Since the next state number doesn’t depend on the value of B, so in the multiplexer we set LD.BEN to 0 meaning that we won’t choose the B value to determine the control signal.

For the tristate driver gates, since in this stage no value on the bus is needed to be transfered, so all gates should be disabled. Meaning that GatePC, GateMAR, GateMDR, GateALUSHF and GateRS2 are set to 0.

Since PC is not updated at this stage, so in theory we don’t need to determine the values of PCMUX, ADDR1MUX, ADDR2MUX because the LD.PC is disabled and these values are not used. But in practice, we set them to the default value, which is 0, meaning that PCMUX=0, ADDR1MUX, ADDR2MUX = 00.

Since the GateMAR is disabled, actually the value of MARMUX is unused and in theory we don’t need to use the value determined by MARMUX, but in practice, we set it to the default value, which is 0, meaning that MARMUX=0.

Since we don’t need to load anything to the MDR, so in theory we don’t need to determine MDRMUX, but in practice, we set it to the default value, which is 0, meaning that MDRMUX=0.

Since we don’t need to output anything from the register file, so we disable the output of rs1 and rs2, meaning that RS2EN and RS1EN are set to 0.

The RS2MUX should be set to 0 to choose the output of rs2 because in this stage, we don’t want to calculate with ALU and the 2 inputs of the ALU should be disabled. But there is no control for disabling the I immediate generator, so we can only let the RS2MUX choose the disabled RS2 to make the input disabled.

Since we need to modify the memory, so that MIO is enabled, meaning that MIO\_EN is set to 1, and since we need to write to memory, WE is enabled, set to 1.

Since the bit width should be selected by the control signal, the DATASIZE multiplexer should be set to 1.

1. State 51

A black text with a white background

Description automatically generated

Since the next state of this state does not depend on the instruction, we don’t need to read from the IR[6:0] and the IRD is set to 0.

Since the next state number is 0, then we should set all J6~J0 to 0 to represent 0.

In this state, we update the rd to be the value calculated by the operation on rs1 and rs2, therefore,

No PC is updated, so LD.PC is 0;

No MAR is updated, so LD.MAR is 0;

No MDR is updated, so LD.MDR is 0;

No instruction is updated, so LD.IR is 0;

Register file is updated, so LD.REG is 1.

Since the next state number doesn’t depend on the value of B, so in the multiplexer we set LD.BEN to 0 meaning that we won’t choose the B value to determine the control signal.

For the tristate driver gates, since in this stage the value on the bus is the value of the ALU, so only the gate that controls Selection logic value should be enabled and all other values are disabled. Meaning that GatePC, GateMAR, GateMDR, GateRS2 are set to 0, and GateALUSHF is set to 1.

Since PC is not updated at this stage, so in theory we don’t need to determine the values of PCMUX, ADDR1MUX because the LD.PC is disabled and these values are not used. But in practice, we set them to the default value, which is 0, meaning that PCMUX=0, ADDR1MUX=00.

Since in this stage we need to calculate the values by ALU based on the output of rs1 and rs2, so we should enable RS2EN and RS1EN, setting them to 1.

Since the input value of ALU is rs1 and rs2, so in the RS2MUX we should choose the value from RS2, setting the value of RS2MUX to 0.

Since in this stage, no memory operation is needed, so we should set MIO\_E and WE to 0. Since we don’t select the bitwidth because no memory operation is performed, we set it to 0 by making the multiplexer choose 0 (setting the DATASIZE to 0)

Since in this state we don’t reset the RISCV-LC, we do not enable reset, setting RESET to 0.

1. State 102

A black and white diagram

Description automatically generated

Since the next state of this state does not depend on the instruction, we don’t need to read from the IR[6:0] and the IRD is set to 0.

Since the next state number is 0, then we should set all J6~J0 to 0 to represent 0.

In this state, we update the value of PC to the value of the sum of rs1 + imm12, therefore,

PC is updated, so LD.PC is 1;

No MAR is updated, so LD.MAR is 0;

No MDR is updated, so LD.MDR is 0;

No instruction is updated, so LD.IR is 0;

No Register file is updated, so LD.REG is 0.

Since the next state number doesn’t depend on the value of B, so in the multiplexer we set LD.BEN to 0 meaning that we won’t choose the B value to determine the control signal.

For the tristate driver gates, since in this stage the PC value are calculated by the ALU with input immediate generator and rs1, so it should be the ALU below (highlighted in yellow):

A diagram of a flowchart

Description automatically generated

Since the output of this ALU should be transferred to the BUS by GateMAR then to the PC, so only the gate that controls GateMAR value should be enabled and all other values are disabled. Meaning that GatePC, GateALUSHF, GateMDR, GateRS2 are set to 0, and GateMAR is set to 1.

Since the PC is receiving the value from the addition result of imm and rs1 instead of from PC+4, so the PCMUX should be set to 1.

Since the immediate value is the 12 bits immediate from I instruction (for example, jalr), the immediate value should be generated from I Imm Gen, so the ADDR2MUX should select the value of 1, or 01. Since the other value fed into the ALU is rs1, so that the ADDR1MUX should choose the value of 2, or 10.

Since the calculation result is from the ALU highlighted in yellow, the MARMUX should be set to 0.

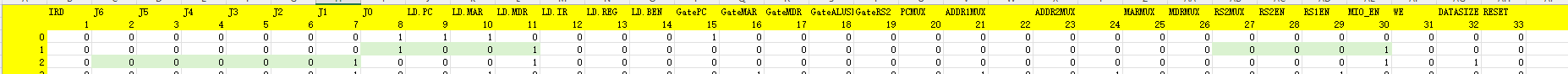
Since we don’t need to load anything to the MDR, so in theory we don’t need to determine MDRMUX, but in practice, we set it to the default value, which is 0, meaning that MDRMUX=0.

Since we only need the output of rs1 and don’t need the output of rs2, we only need to set RS1EN to 1 and set RS2EN to 0.

Since in this stage, no memory operation is needed, so we should set MIO\_E and WE to 0.

1. Final result of uop:

State 1 and 2



State 32 and 33



State 51



State 102



The final uop file:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

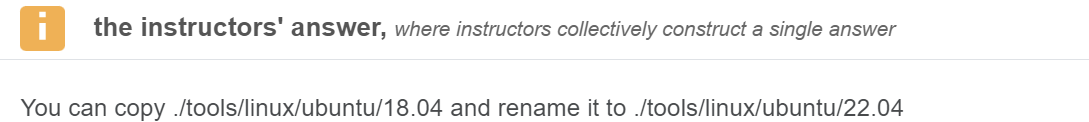
Description automatically generated

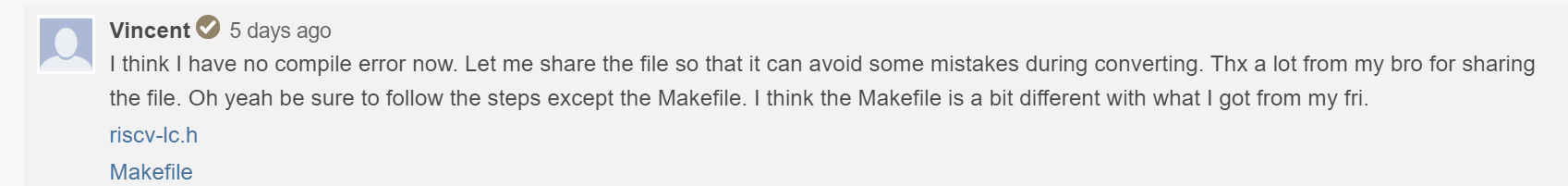
1. My mistakes and understandings

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

A screen shot of a computer

Description automatically generated

(rest is omitted)

Result:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Count10

Running

A screenshot of a computer

Description automatically generated

A black screen with white dots

Description automatically generated

(rest is omitted)

Result

A screenshot of a computer

Description automatically generated

1. Swap

Before

A screen shot of a computer

Description automatically generated

Running

A black screen with white text

Description automatically generated

(rest is omitted)

Result:

A black background with white text

Description automatically generated

1. Add4

Before:

A screenshot of a computer

Description automatically generated

Running:

A black and white screen

Description automatically generated

(rest is omitted)

Result:

A black screen with white text

Description automatically generated

# 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 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 is 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

MEM\_VAL: the temporary variable that stores the memory after each cycle finishes.

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.

BUS: the value of bus.

1. 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

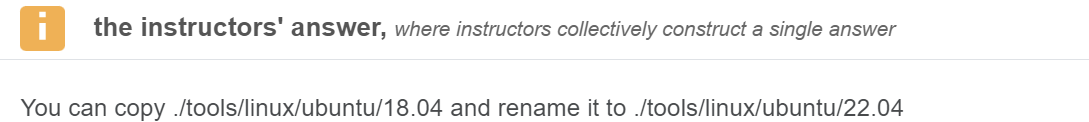
1. My mistakes and understandings

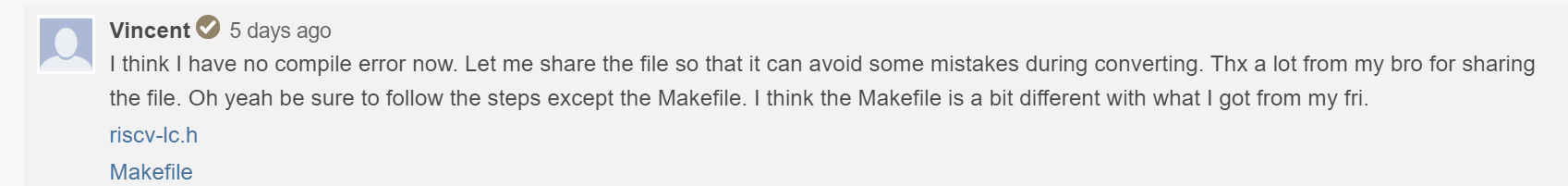
One mistake I had when doing the lab is that I missed considering the endianness of the data, by little endian rule, the higher bits should be placed in higher addresses, but I got it reversed. Resulting in mistakes.

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

A screenshot of a computer program

Description automatically generated

(rest is omitted)

Result:

A screenshot of a computer

Description automatically generated

A computer screen with numbers and letters

Description automatically generated

1. Count10

Running:

A screenshot of a computer

Description automatically generated

(rest is omitted)

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

(rest is omitted)

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

(rest is omitted)

Result:

A black background with white letters and numbers

Description automatically generated

# Lab 3.3

1. Console results
2. Make:

A screenshot of a computer program

Description automatically generated

1. Isa.bin

Running:

A screenshot of a computer

Description automatically generated

(rest is omitted)

Result:

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

1. Count10:

Running:

A screenshot of a computer

Description automatically generated

(rest is omitted)

Result:

A screenshot of a computer program

Description automatically generated

1. Swap

Before:

A black screen with white text

Description automatically generated

Running:

A screenshot of a computer program

Description automatically generated

(rest is omitted)

Result:

A black screen with white text

Description automatically generated

1. Add4

Before:

A black background with white text

Description automatically generated

Running:

A screenshot of a computer program

Description automatically generated

(rest is omitted)

Result:

A black 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

fsm.pdf

riscv-lc.pdf