CPU Based on Extended MINISYS ISA

Group Members

李子南 # 12011517

唐昕宇 # 12011439

何泽安 # 12011323

CS202 COMPUTER ORGANIZATION

2022 SPRING

FINAL PROJECT

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPT. OF COMPUTER SCIENCE AND ENGINEERING

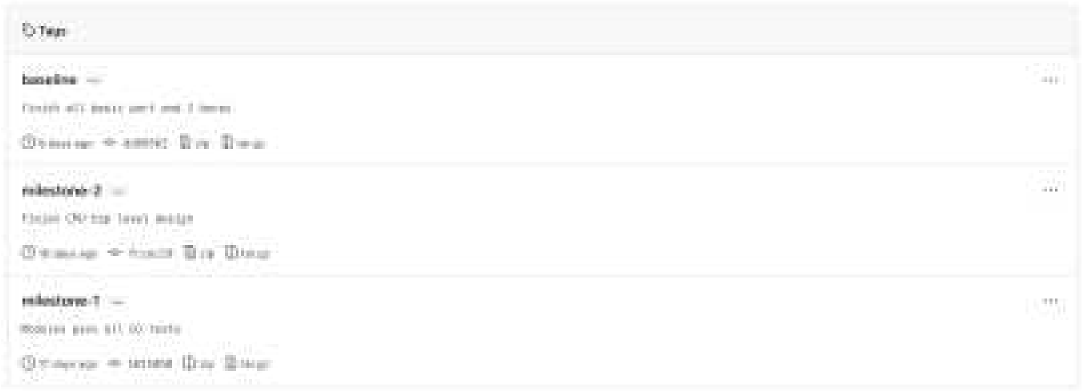
Contents

1 Developers Arrangement

Student ID	Name	Job	Contribution
12011517	Zinan Li	execution; test1; top level design; inst extension	33 %
12011439	Xinyu Tang	ifetch & dmem & memorio; UART; 2 additional IO	33 %
12011323	Zean He	controller & decoder; test2; report; inst extension	33 %

2 Milestone Changelog

The milestones in our implementation of CPU can be summarized as six parts: ① we divided the 5 foundational components of CPU (the three tasks in the OJ) and pass each of them; ③ writing a top model that instantiate the models and make them work together; ④ implemented the bonus part *UART*, which also helped us a lot during debugging the two test scenarios; ⑤ pass the two test asm; ⑥ adding mode bonus functions. We used git as VCS and record some milestones of development by adding tags.


Tags

baseline

Finish all basic part and 2 bonus

[illegible]

milestone-2

Finish CPU top level design

[illegible]

milestone-1

Modules pass all OJ tests

[illegible]

3 Architecture Design

3.1 Features

ISA Our ISA is base on the MINISYS ISA, then we checked all the MIPS32 instructions (try to compile them in the compiler) and found several supported ones. We summarized our supported instructions as below (instructions in MINISYS ISA are marked green in the left table, which is the MIPS32 ISA; the supported instructions that are of MIPS32 but not in MINISYS are listed in the right figure). They are encoded as MINISYS's encoding.

Registers In our decoder, there are 32 32-bit registers as the register file, and two 32-bit registers, *hi* and *lo*. We handle the error input by robust logic in our asm file.

Addressing The CPU is based on *Harvard architecture*, where the instruction memory is separated from the data memory. The data and instructions are stored in the RAM, aka. *0x00000000 - 0x00010000*.We also assigned address for IO ports on board, for example, the base address of board IO is *0xFFFFF000*, starting from that is the *RAM*, we also bind *0xFFFFC50* to *SegTubes*, *0xFFFFC70 - 0xFFFFC72* are for *switches*, *0xFFFFFC60-0xFFFFFC62* are *LEDs*. The enter button was the first bit load from *0xFFFFFC80*. The unit for board IO ports is based on bit, still the address is based on byte, thus we use quiet a lot *andi* / *ori* operations on the register's data.

Performance Our CPU is single cycled, thus CPI = 1 theoretically. We also did a experimental validation, by the data we calculated the actual CPI was 0.9978 $\approx$ 1, the bias may be mainly caused by the clock.

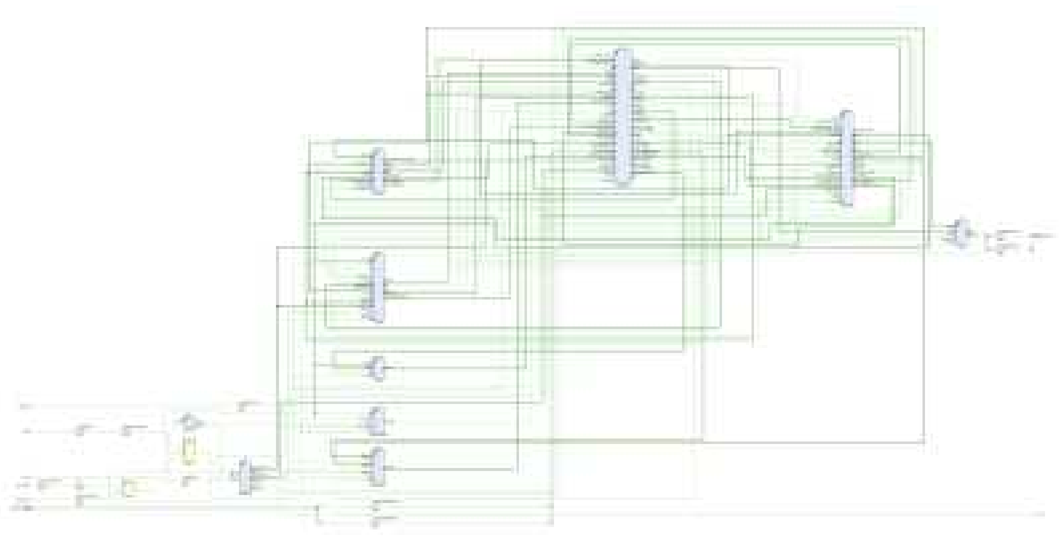
```
.text 0x0000
main:
lui $1,0xFFFF
ori $28,$1,0xF000
ori $1, $0, 0
ori $2, $0, 1
sll $2, $2, 27  # cpi = 2^28 / time
ori $3, $0, 0xFFFF
ori $4, $0, 1

sw $4, 0xC60($28)
  m_loop1:
    lw $8, 0xC80($28)
    beq $8, $0, m_loop1
  m_loop2:
    lw $8, 0xC80($28)
    bne $8, $0, m_loop2

sw $3, 0xC60($28)
  loop:
    addi $1, $1, 1
    bne $1, $2, loop

j main
```

Interfaces We used the *cpuclk* IP kernel that output two signals, 23 MHz for CPU and 10 MHz for UART. We also bind the reset button (P20), UART port (V18 and Y19), the enter button (P4, vibrated using the Y18 clock on board, its 100 MHz was divided inside the vibration model) was bind to enable the UART mode / getting ready to receive data; all other IO (LED, switch, SegTube, etc.) were not bind explicitly to the CPU, but can be accessed by address (starting from 0xFFFFF000).

3.2 Internal Structure

3.2.1 CPU Top

The top model instenlizes submodels explained below and connect wires between them properly. Note that since switches, LEDs contains too much repeated works and displaying them will make the figure not clean as below, they are thus omitted. A pdf file of it is placed separately and you may check it if necessary.



```verilog
module CPU_TOP (clk, fpga_rst, switches, leds, start_pg, rx, tx, button, DIG, Y);
    input clk, fpga_rst;  //reset
    input[23:0] switches; // switches' signal
    output[23:0] leds;    // leds' signal
    input button;
    output[7:0] DIG;      //tube' position
    output[7:0] Y;        //tube value
    input start_pg;
    input rx;
   output tx;
```

3.2.2 ALU

```
module executs32(Read_data_1,Read_data_2,Sign_extend,Function_opcode,Exe_opcode,ALUOp,
                 Shamt,ALUSrc,I_format,Zero,Jr,Sftmd,ALU_Result,Addr_Result,PC_plus_4);
input[31:0]  Read_data_1;        // Read data 1
input[31:0]  Read_data_2;        // Read data 2
input[31:0]  Sign_extend;        // Sign extend in 32 bit
input[5:0]   Function_opcode;      // instructions[5:0]
input[5:0]   Exe_opcode;          // instruction[31:26]
input[1:0]   ALUOp;             // { (R_format || I_format) , (Branch || nBranch) }
input[4:0]   Shamt;             // instruction[10:6], the amount of shift bits
input           Sftmd;            // 1 means this is a shift instruction
input        ALUSrc;      // 1 means the 2nd operand is an immedite (except beq，bne)
input        I_format;          // 1 means I-Type instruction except beq, bne, LW, SW
input        Jr;               // 1 means this is a jr instruction
input[31:0]  PC_plus_4;         // PC+4
output       Zero;              // 1 means the ALU_output_mux is zero, 0 otherwise
output reg[31:0] ALU_Result;    // the ALU calculation result
output[31:0] Addr_Result;        // the calculated instruction address
```

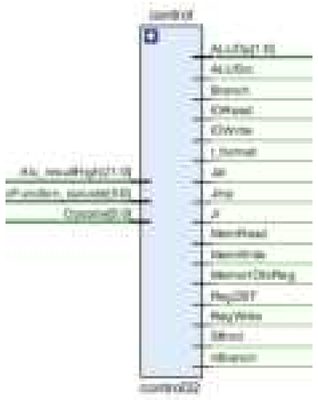
3.2.3 Decoder

The instructions are encoded under the MINISYS ISA, note that we extended several instructions from MIPS32, they are encoded as MIPS32 does. Also, to support *mult* whose ALU result is splitted into lo and hi, we add one more input port (32-bit).

```
module decode32(read_data_1, read_data_2,
                Instruction, mem_data,
                ALU_result, ALU_Hi, Jal,
                RegWrite, MemtoReg,
                RegDst, Sign_extend,
                clock, reset, opcplus4);

    input[31:0]  Instruction;              // instruction from memory
    input[31:0]  ALU_result;                        // alu result
    input[31:0]  ALU_Hi;                         // data from alu hi output
    input        RegWrite;
    input        RegDst;
    input        MemtoReg;
    input[31:0]  mem_data;                        // DATA RAM or I/O port
    input        Jal;                            // Is Jal instruction?
    input[31:0]  opcplus4;                       // from ifetch link_address
    input         clock, reset;
    output[31:0] read_data_1;
    output[31:0] read_data_2;
    output[31:0] Sign_extend;
```

3.2.4 Controller

It was mostly based on the MINISYS ISA, and we then add 8 extended instructions that are encoded using its corresponding encoding in MIPS32, we do the K-map on these.



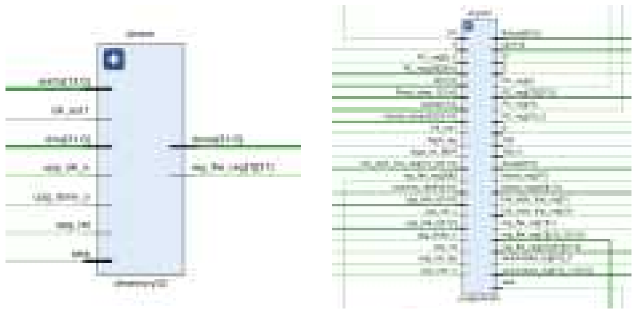
```
module control32(Opcode, Function_opcode, Alu_resultHigh, Branch, nBranch,
                 Jr, Jmp, Jal,
                 ALUSrc, ALUOp,
                 MemWrite, MemRead, IORead, IOWrite,
                 RegWrite, RegDST,
                 MemorIOtoReg, I_format, Sftmd);

    input[5:0]   Opcode;              // instruction[31..26] from Ifetch, 6 bits opcode
    input[5:0]   Function_opcode;     // instruction[5..0] from Ifetch, 6bits function coc
    input[21:0]  Alu_resultHigh;      // From the Alu unit Alu_Result[31..10]

    output       Branch;              // 1 indicate the instruction is "beq" , otherwise i
    output       nBranch;            // 1 indicate the instruction is "bne", otherwise it
    output       Jr;                  // 1 indicate the instruction is "jr", other wise i
    output       Jmp;                // 1 indicate the instruction is "j", otherwise it's
    output       Jal;                // 1 indicate the instruction is "jal", otherwise it
    output       ALUSrc;            // 1 indicate the 2nd data is immidiate (except "beq"
    output[1:0]  ALUOp;             // if the instruction is R-type or I_format, ALUOp is
    output       MemWrite;          // 1 indicate write data memory, otherwise it's not
    output       MemRead;           // 1 indicates that the instruction needs to read fro
    output       IORead;            // 1 indicates I/O read
    output       IOWrite;           // 1 indicates I/O write
    output       RegWrite;            // 1 indicates that the instruction needs to write
    output       RegDST;            // 1 indicate destination register is "rd"(R),otherwi
    output       MemorIOtoReg;      // 1 indicates that data needs to be read from memory
    output       I_format;          //  1 indicate the instruction is I-type but isn't "b
    output       Sftmd;             //  1 indicate the instruction is shift instruction;
```

3.2.5 Instruction Fetch

```
module Ifetc32(Instruction_i,branch_base_addr,Addr_result,
               Read_data_1,Branch,nBranch,Jmp,Jal,Jr,
               Comp,Zero,clock,reset,link_addr,
               rom_adr_o,Instruction_o);
input[31:0] Instruction_i;       // the instruction fetched from this prgrom
output[31:0] branch_base_addr; //(pc+4) to ALU which is used by branch type instruction
input[31:0]  Addr_result;      // the calculated address from ALU
input[31:0]  Read_data_1;      // the address of instruction used by jr instruction
input        Branch;           // while Branch is 1,it means current instruction is beq
input        nBranch;          // while nBranch is 1,it means current instruction is bnq
input        Jmp;              // while Jmp 1, it means current instruction is jump
input        Jal;              // while Jal is 1, it means current instruction is jal
input        Jr;               // while Jr is 1, it means current instruction is jr
input        Comp;             //from alu, 1 means bgez or blez is true
input       Zero;           //while Zero is 1, it means the ALUresult is zero
input       clock,reset;    //Clock and reset

output   [31:0] link_addr;     // (pc+4) to Decoder which is used by jal instruction
output   [13:0] rom_adr_o;
output   [31:0]Instruction_o;
```

3.2.6 Data Memory

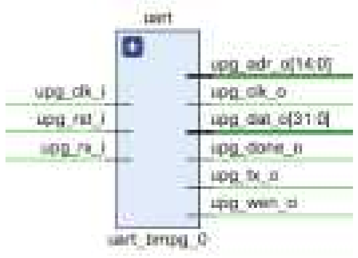
It receives the address and a read / write signal. It also supports UART mode. Internally, it uses *programrom.xci* IP kernel.



```
module dmemory32(ram_clk_i, ram_wen_i, ram_adr_i, ram_dat_i, ram_dat_o,
                 upg_rst_i, upg_clk_i, upg_wen_i,
                 upg_adr_i, upg_dat_i, upg_done_i);

    input ram_clk_i; // from CPU top
    input ram_wen_i; // from Controller
    input [13:0] ram_adr_i; // from alu_result of ALU
    input [31:0] ram_dat_i; // from read_data_2 of Decoder
    output [31:0] ram_dat_o; // the data read from data-ram

    // UART Programmer Pinouts
    input upg_rst_i; // UPG reset (Active High)
    input upg_clk_i; // UPG ram_clk_i (10MHz)
    input upg_wen_i; // UPG write enable
    input [13:0] upg_adr_i; // UPG write address
    input [31:0] upg_dat_i; // UPG write data
    input upg_done_i; // 1 if programming is finished
```

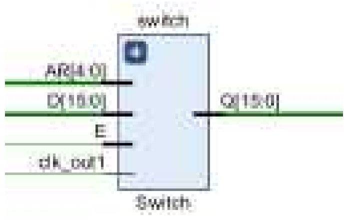
3.2.7 UART

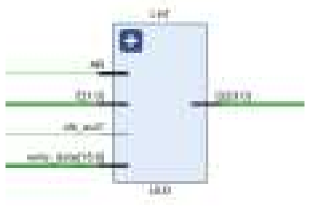
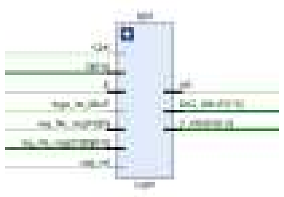
This model was provided by the *uart_bmpg_0.xci* IP kernel, we applied it in the top model.



3.2.8 Switch, LED, & SegTube

These three models take on the util IO function, and was mainly service for our addressing mechanism.



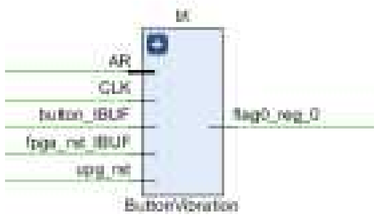



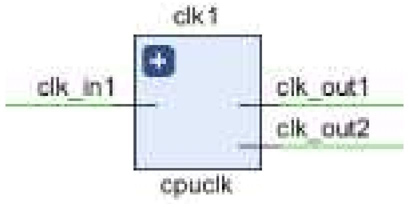
```
module LED(clk, rst, ledwrite, led, ledaddr,ledwdata, ledout);
    input clk;
    input rst;
    input ledwrite;                        // IOwrite
    input led;                          // signal from Memorio
    input[1:0] ledaddr;                 // last 2 bit address
    input[15:0] ledwdata;               // data to write
    output[23:0] ledout;            // output data

module Light(clk, rst, lightwrite, light, lightwdata, DIG, Y);
    input clk; //时钟信号
    input rst; //复位信号
    input lightwrite; //写信号
    input light; //从 memorio 来的 LED 片选信号
    input[15:0] lightwdata; //要写到数码管的 data, 共 3 位, 每位 5bit, 能显示 0-9a-w
    output [7:0] DIG; //向板子上输出的信号
    output [7:0] Y;

```

```
module Switch(clk, rst, switchread, switch,switchaddr, switchrdata, switch_i);
    input clk;
    input rst;
    input switch;                      //  signal from memorio
    input[1:0] switchaddr;              //  last 2 bits of address
    input switchread;                  //  IORead
    output [15:0] switchrdata;          //  signal to CPU
    input [23:0] switch_i;              //  signal from board
```

3.2.9 Button Vibration & Clock

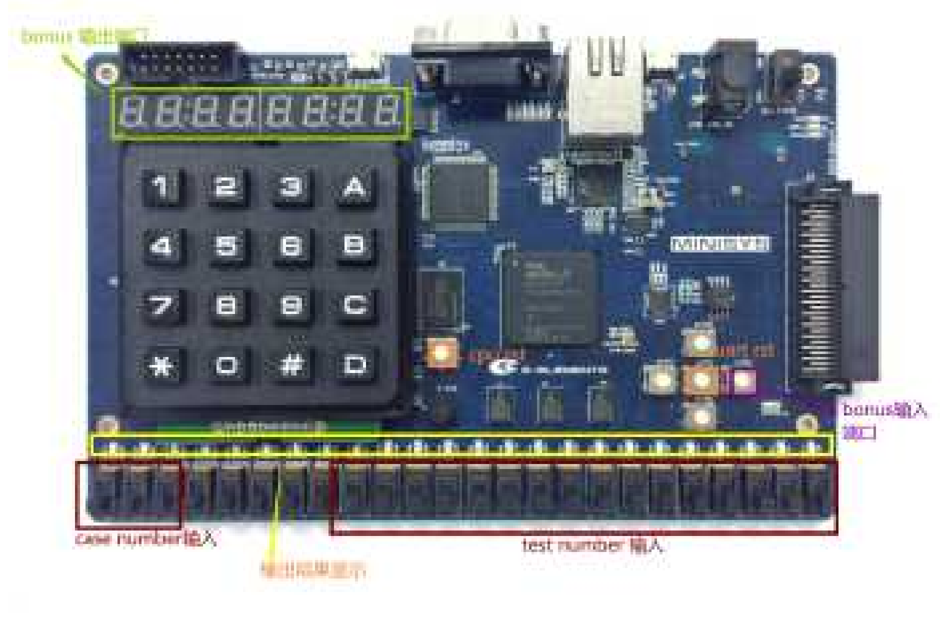
The button vibration is similar to the one in our digital logic's project. The ButtonVibration model was for the UART prepare button and the enter button, it includes a clock divider which receives CLK as input. The figure in right is the IP kernel that provides proper frequency for CPU and UART.





```
module ButtonVibration (
    input clk,
    input rst,
    input buttoncs,
    input button,
    output buttonout
);
```

4 Test Scenarios



Section	Method	Type	Description	Status
.v models	Simulate	Unit	Online judge & testbench: check waveform for all instructions	Pass
Test Assemblers	Simulate	Unit	Use separate .asm files to test part of code (eg. reading input)	Pass
Top module	On board	Integrated	With the specified simple coe, test if the LED lights as expected	Pass
			With the specified instruction, test if the LED lights as expected	Pass
UART	On board	Integrated	Try to program the board via UART, with the simple asm tested previously	Pass
SegTube	On board	Integrated	When sw specified value to its memory address, check whether it can display correctly	Pass
test1.asm	On board	Integrated	Input a number and check palindrome	Pass
			Input two numbers continuously at once	Pass
			Test a & b, a | b, a $\hat{b}$, sll, srl, sra	Pass
test2.asm	On board	Integrated	Input a number n, then continuously input n numbers	Pass
			Bubble sort, for both unsigned and signed numbers	Pass

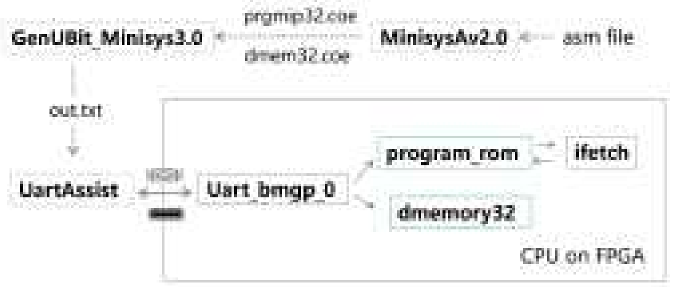
	Converting an 8-bit "signed" binary into its 2’s complement	Pass
	Finding the min / max value of a dataset, and do the subtraction operation	Pass
	Finding the number by the index of dataset and its index in the dataset	Pass
	Display information in turn in 5 sec	Pass

In the process of developing our CPU, we have sufficient unit tests for each submodel, only with the correct underlying models that work as expected for each instruction, can we develop more complex logics like the two test scenarios asm on it. Finally, we can summarize the results as: the five components are all correct for the extended ISA instructions (MINISYS + *mult* / *multu*, *mfhi* / *mflo*, *mthi* / *mtlo*, *blez* / *bgtz*, these are all the instruction that the provided compiler supports). And the top model connects all wires correctly, the two assemblers have correct logic.

5 Bonus Features

To make it clear, we summarize our bonus features here (again!) separately. We implemented UART, more IO interfaces, and ISA extending.

UART To program coe files in our FPGA conveniently, we add UART part in our CPU design. Ifetch and Dmemory module need to redesign because there is a new resource of data and instruction, so we need to judge where to get the sources. Prgrom module is picked out to make port connection easier. And uart_bmgp_0 ip kernel support the whole process. UART can reduce time to generate bitstream, which helps us debug more efficiently.



IO Interfaces Except for led and swithches, we add SegTube and button as additional IO devices. For each IO device we write a separate module to manage its read/write using its allocated address. The module of button also have the function of removing the vibration, which ensure the button work in a right way.

ISA Extending In order to extend the usage of our CPU, we modify some modules to add extended ISA instructions.(MINISYS + *mult* / *multu*, *mfhi* / *mflo*, *mthi* / *mtlo*, *blez* / *bgtz*, these are all the instruction that the provided compiler supports) We write test file for all the additional instructions, and they all work perfectly.

6 Problems and Summary

We were such an unfortunate that met lots of unexcepted things, such as, Prof. Wang didn't mention that the primitives output register selection of prgrom need to be untick when changing the IP kernel's section from 'A Single Port ROM' to 'A Single port RAM', which waste our whole night to find the problem.

Another problem we meet is the vibration on the switches. At first we thought that it's a bug from our asm files, but with our hard work, we find that the vibration leads to the unstable performance of the test file. We than change the swithch to button and we write a simple function in our asm file to block the button input.

The SegTube is also a big problem, we add this module before the additional instructions and it works perfectly on that version. But once we modify the modules of CPU in order to add some new instructions, we found that SegTube doesn't work at all. We do not cahnge the code of that module after it complete and all the other function on CPU work perfectly.

We meet some problem at the loop of the case8 of test2, we miscalculated the number of cycles needed and thus we waste a lot of time to fix a inexistent bug.

We learn a lot from this project. Writing the different module of CPU makes us learn how to work together. And thanks to the awkward Vivado, we learn how important the version management is. Thanks to Mr.Linus, we have the amazing git to manage our file and version. With git, we revert to the right version once and once again, it is no exaggeration to say that git saved our project.

Another important point is writing the asm file, writing the assemblers make us understand how difficult to write even a simple function in 1970s. It make us to think a problem as a computer, we learn how to write hardware friendly code and be grateful to high level programming language such as C++, java and python.