**Computer Architecture Final Project Report**

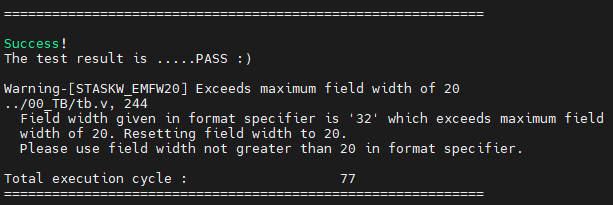
**Group 25**

**B10705007 劉冠甫**

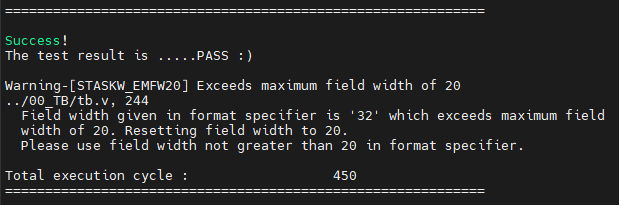
**B10705025 彭鈞道**

**◆ Record the execution cycle number of each instruction set**

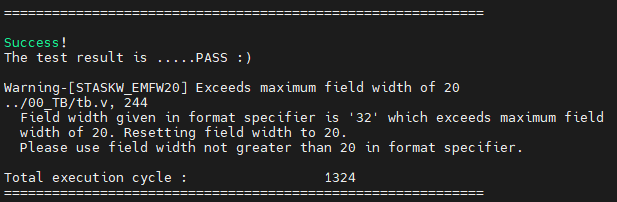
I0:



I1:



I3:

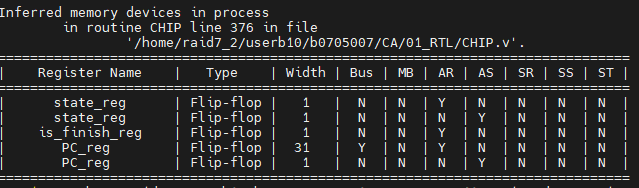


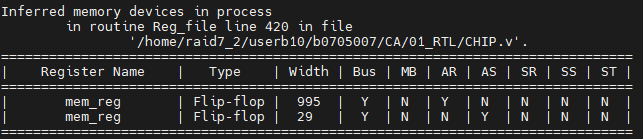
Execution Time Summary

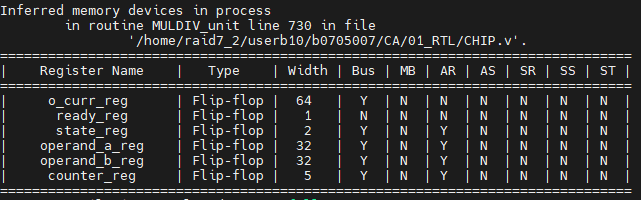
|  |  |
| --- | --- |
| Testbench Number | Cycles |
| I0 | 77 |
| I1 | 450 |
| I2 | The assembly of our homework 1 contains some instructions unsupported by this CPU |
| I3 | 1324 |

**◆ Snapshot the “Register table” in Design Compiler**

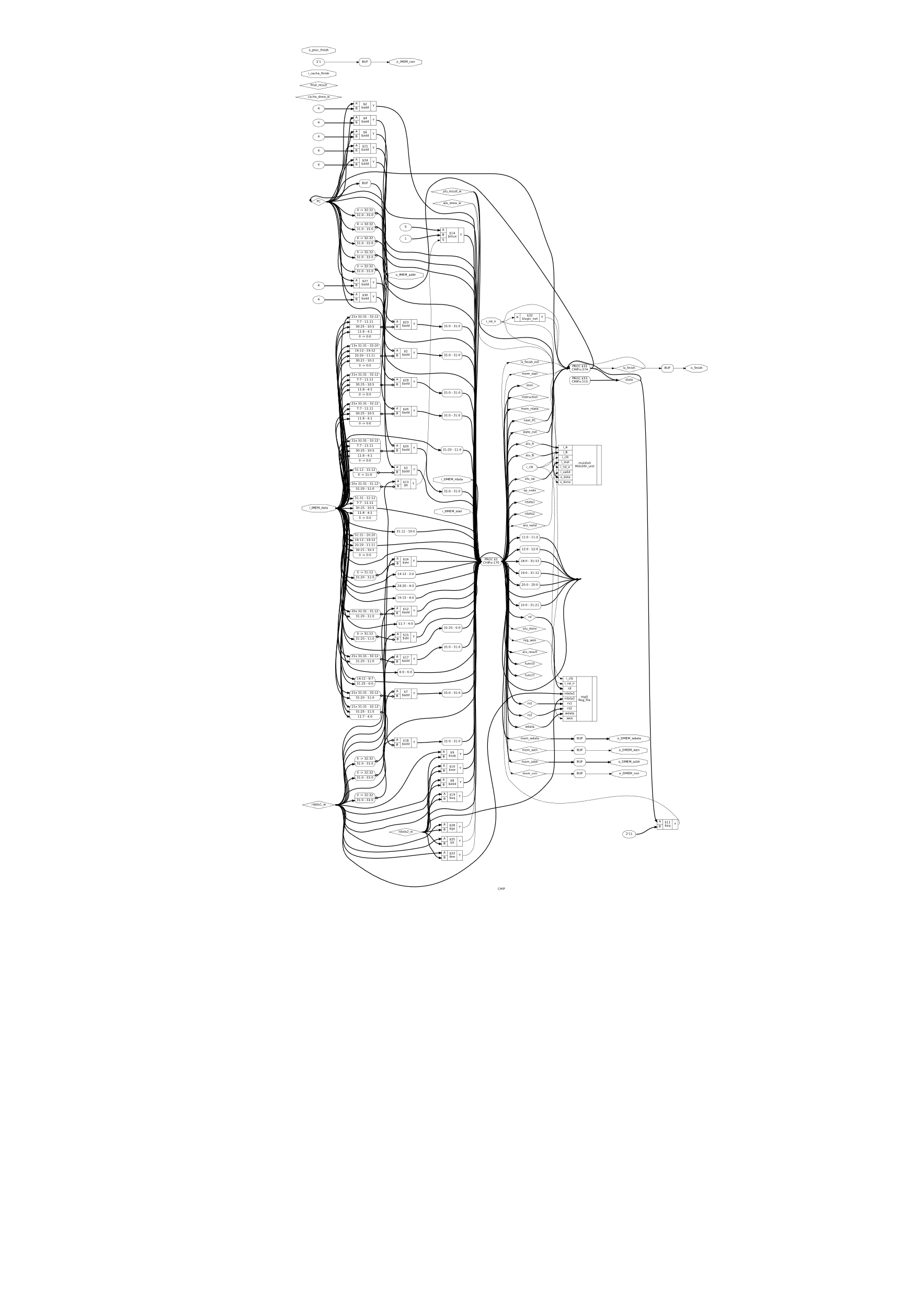
All inferred memory devices in the CHIP module, Reg\_file module, and MULDIV\_unit are flip-flop. There are no latches.

****

****

****

**◆ Draw the block diagram of your CPU architecture**

****

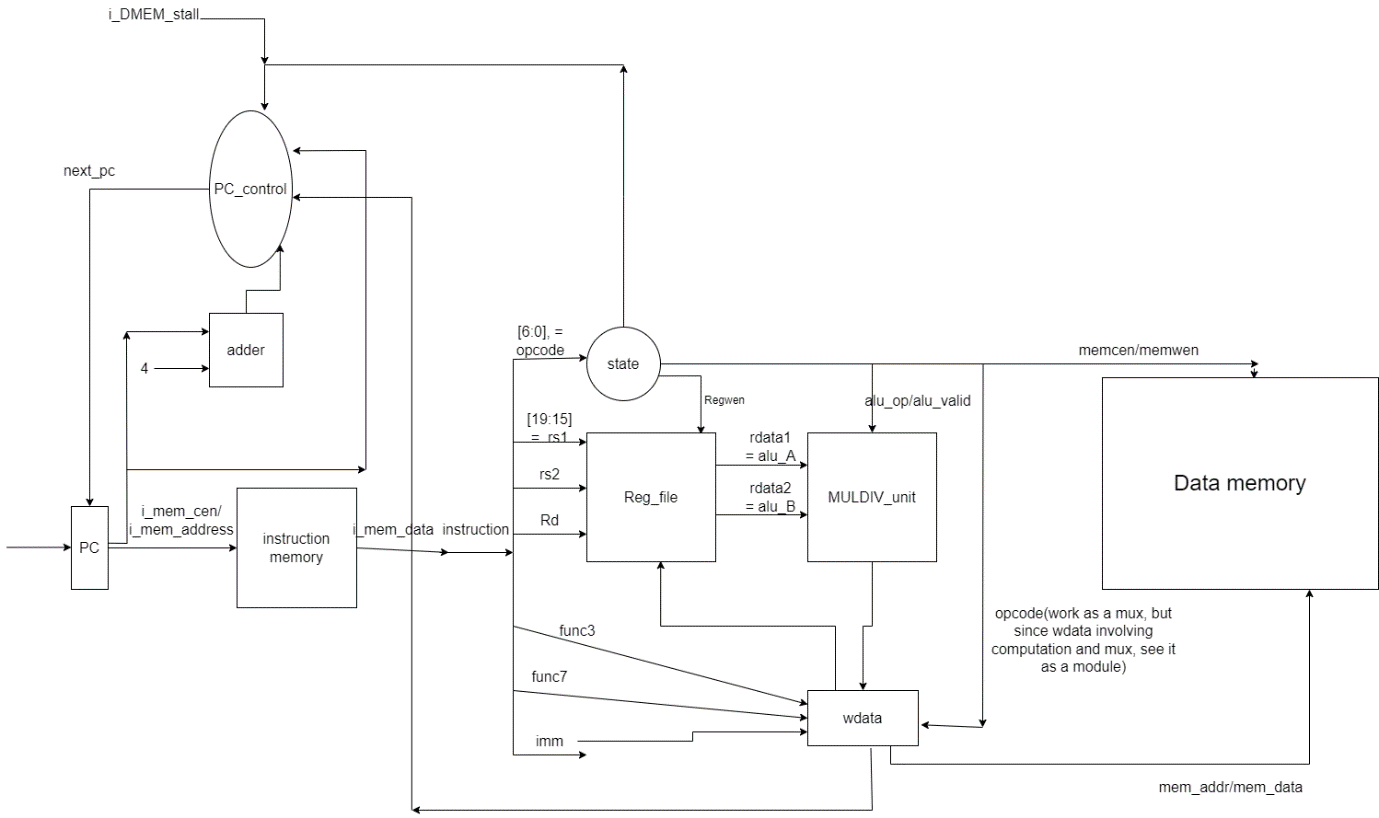
The graph is generated by the open source synthesis tool YoSys.

The steps of generating this block diagram is as below:

Navigate to the directory where the .v file exists. If Yosys is installed in the system, entering 'yosys' will open the Yosys command line interface. Input 'read\_verilog CHIP.v' to start compiling this file. Then, enter 'hierarchy -top CHIP' to specify the module that needs analysis. Only one module can be drawn at a time, so to check the 'CHIP,' comment out the places where 'Reg\_file,' 'MULDIV\_unit,' and 'Cache' are defined. Afterwards, input 'show' to automatically generate a block diagram.

For Yosys configurations, refer to https://github.com/YosysHQ/yosys.

Since the synthesized block diagram is difficult to read. The simplified block diagram is as below:



**◆ Describe how you design the data path of instructions not**

**referred in the lecture slides (jal, jalr, auipc, …)**

**JAL**

For JAL, first, we should set the register to be write enabled and write data of the register to PC+4 so that we can store the next instruction to be returned into rd. Also we need to set the imm value to {instruction[31], instruction[19:12], instruction[20], instruction[30:21], 1'b0}; and set the next\_PC to $signed({1'b0, PC}) + $signed(imm[20:0]) to make the PC in the next step to be the desired value.

**JALR**

Similar to JAL, but there are some differences as below:

1. imm will be instruction[31:20] instead.
2. Although we store the same data into rd, next\_PC will be $signed({1'b0, rdata1}) + $signed(imm[11:0]) instead.

**AUIPC**

First, we generate upper immediate as below:

imm[31:12] = instruction[31:12];

imm[11:0] = 0;

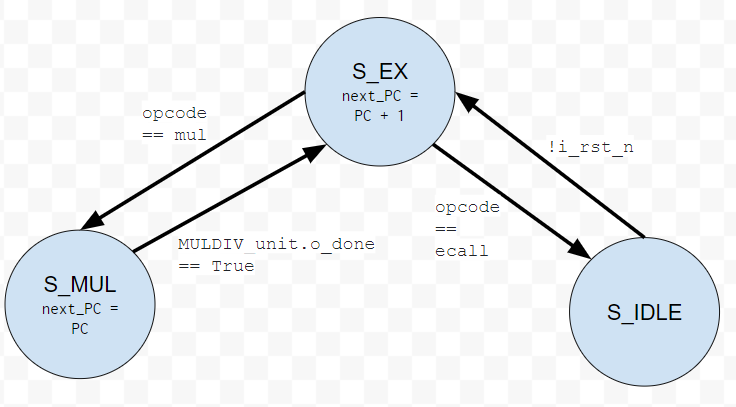
Then we set the write data of register to PC + imm to make rd the desired value.

**ECALL**

We will set is\_finish\_nxt = 1 to make the next cycle have the o\_finish set to high.

**◆ Describe how you handle multi-cycle instructions (mul, div …)**

For mul instruction, the following FSM is implemented.



If a multiplication procedure is called, the CPU will enter into S\_MUL, in the state the PC update will be frozen until the MULDIV\_unit.o\_done is set to high, indicating the multiplication is completed, then the CPU will be back into S\_EX.

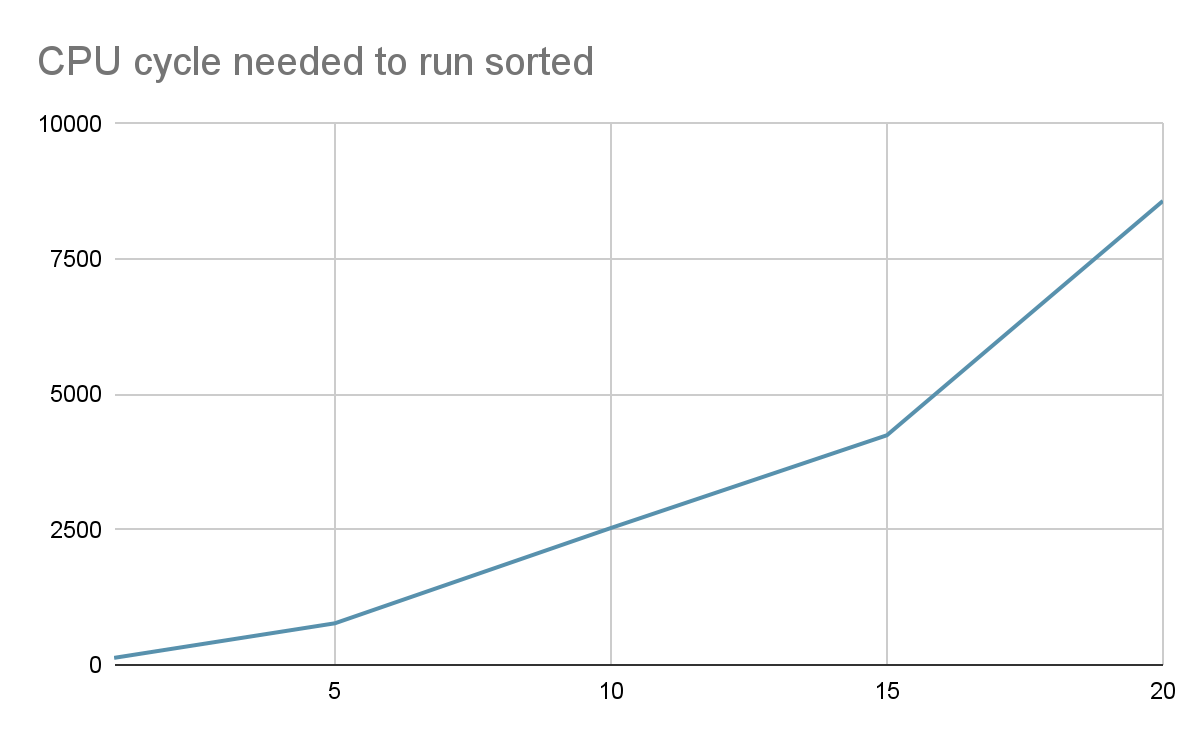
For lw, sw, we detect whether the i\_DMEM\_stall signal is set to high. If the signal is set to a high, it means that we should wait for memory data to be ready, then we will let PC <= PC instead of next\_PC in the sequential always block to stop the CPU from executing the next instruction.

**◆ Describe your observation**

I tried different settings of testbench parameters.

For I1 (factorial), the CPU can run the largest factorial supported by 32 bits (12!) within the time limit. (1010 cycles).

For I3 (sort), the CPU runtime is as the graph below. The x-axis is the length of the array and the y-axis is the CPU cycle.



It is roughly quadratic to the as expected if we are using a algorithm.

**◆ Work distribution**

We both wrote our own version of the CPU and submitted the one with the least bugs.

|  |  |
| --- | --- |
| 劉冠甫 | CPU 程式撰寫  報告撰寫（程式邏輯部分） |
| 彭鈞道 | CPU 程式撰寫  報告撰寫（block diagram） |