

Computer Organization

PA2

(Single Cycle Processor With R / I-format, Branch and Jump Instructions)

Student

B11107157 電機三乙 林明宏

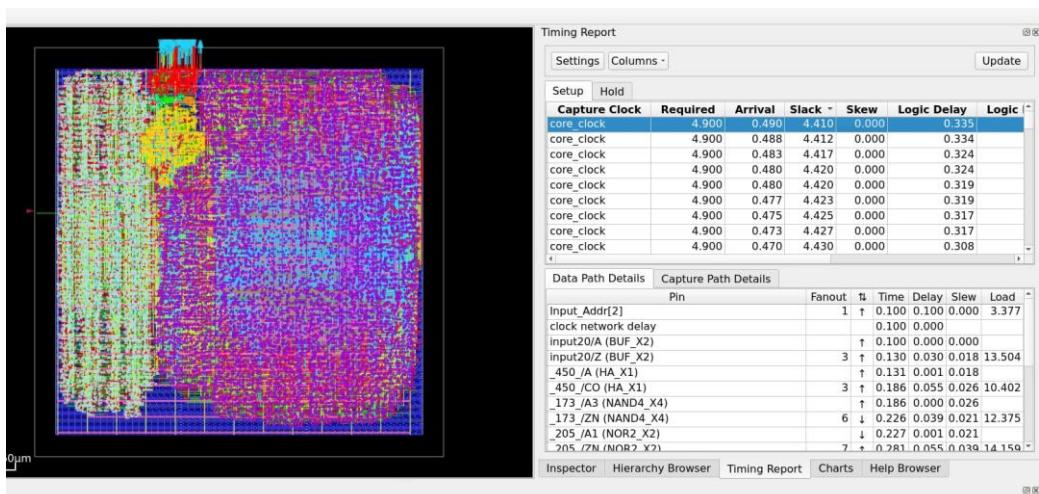
RTL Simulator & Synthesis Tool

*ModelSim-Intel FPGA Standard Edition, Version 20.1.1, windows
The OpenROAD-flow-scripts from github*



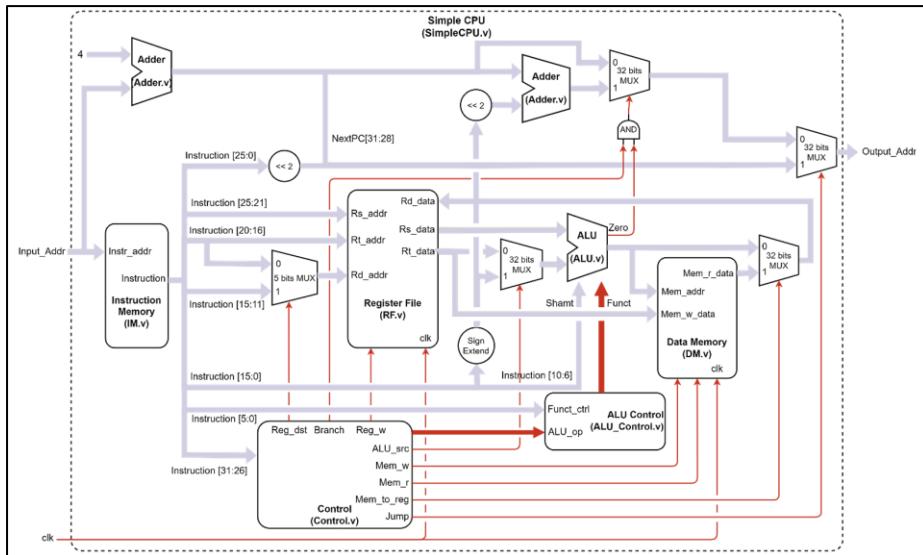
Slack & Area Report

MemSize = 128 | Critical Path Slack : 4.4097 | Area : 33743.912 um²
MemSize = 12 | Critical Path Slack : 4.4039 | Area : 14098.532 um²



a. Descriptions of how you implement each module

1. SimpleCPU.v



根據上面的架構圖去連接各個 Module 的 Data Path

並寫入 Mux、Shifter、AND Gate 去完成各個指令所需的功能。

值得注意的是這邊直接在 SimpleCPU 裡面直接做 PC+4 的動作，可以有效提高 Slack。

```

1  module SimpleCPU(
2      //Output
3      output reg [31:0] Output_Addr,
4      //Input
5      input wire [31:0] Input_Addr,
6      input wire clk
7  );
8
9  // Net -----
10 wire [31:0] Instruction,Rs_Data,Rt_Data,ALU_Result,Rd_Data_in,MemReadData;
11 wire [31:0] Addr,beq_Addr,Immediate,ALU_Src2;
12 wire RegWrite,Reg_dst,Branch;
13 wire ALU_src,Mem_w,Mem_r,Mem_to_reg,Jump,Zero_Flag;
14 wire [5:0] funct;
15 wire [1:0] ALU_OP;
16 wire [4:0] Rd_Addr_in;
17 wire [31:0] Immediate_shift;
18
19 // Assignment -----
20 assign Rd_Addr_in = (Reg_dst)? Instruction[15:11] : Instruction[20:16];
21 assign Rd_Data_in = (Mem_to_reg)? MemReadData : ALU_Result;
22 assign Immediate = {16{Instruction[15]}},Instruction[15:0];
23 assign ALU_Src2 = (ALU_src)? Immediate : Rt_Data;
24 assign Immediate_shift = {Immediate[29:0],2'd0};
25
26 always @(*) begin
27     if(Jump) Output_Addr = {Addr[31:28],Instruction[25:0],2'd0};
28     else if(Branch&&Zero_Flag) Output_Addr = beq_Addr;
29     else Output_Addr = Addr;
30 end
31
32 // Module Connection -----
33
34 // Adder Module
35 assign Addr = Input_Addr + 3'd4;
36 Adder Instruction_Adder(Addr,Immediate_shift,beq_Addr);
37
38 // Instruction Memory Module
39 IM Instr_Memory(Input_Addr,Instruction);
40
41 // Register File Module
42 RF Register_File(Instruction[25:21],Instruction[20:16],Rd_Addr_in,RegWrite,clk,Rd_Data_in,Rs_Data,Rt_Data);
43
44 // ArithmeticLogicUnit Module
45 ALU ArithmeticLogicUnit(Rs_Data,ALU_Src2,Instruction[10:6],funct,ALU_Result,Zero_Flag);
46
47 // ArithmeticLogicUnit Control Module
48 ALU_Control ALU_Control_base(ALU_OP,Instruction[5:0],funct);
49
50 // Control Module
51 Control Control_base(Instruction[29:26],Reg_dst,Branch,RegWrite,ALU_OP,ALU_src,Mem_w,Mem_r,Mem_to_reg,Jump);
52
53 // Data Memory Module
54 DM Data_Memory(MemReadData,ALU_Result,Rt_Data,Mem_w,Mem_r,clk);
55
endmodule

```

2. RF.v

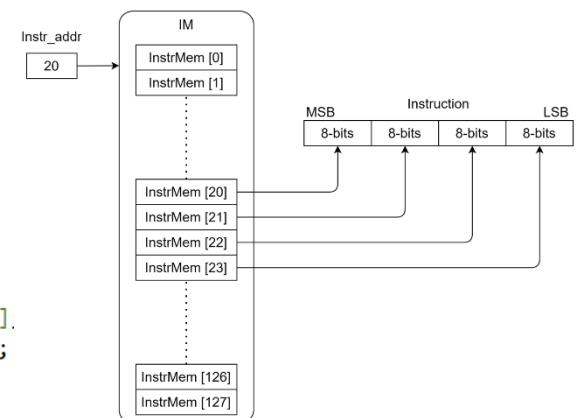
在這個 Module 裡面定義了 32 個 32bits 的 Register-File
並且當 RegWrite 為 High 時，可以從 Negative Edge 將資料寫入到指定的位址內。
而讀取資料的部分則不受 Clock 影響，可以直接讀取。

```
≡ RF.v
1 `define REG_MEM_SIZE 32
2 module RF(
3     //Input
4     input wire [4:0] Rs_Addr,
5     input wire [4:0] Rt_Addr,
6     input wire [4:0] Rd_Addr,
7     input wire RegWrite,
8     input wire clk,
9     input wire [31:0] Rd_Data,
10    //Output
11    output [31:0] Rs_Data,
12    output [31:0] Rt_Data
13 );
14 reg [31:0]R[0:`REG_MEM_SIZE - 1];
15 assign Rs_Data = R[Rs_Addr];
16 assign Rt_Data = R[Rt_Addr];
17
18 always @(negedge clk) begin
19     if(RegWrite) R[Rd_Addr] <= Rd_Data;
20     else;
21 end
22 endmodule
```

3. IM.v

根據右下圖的操作，將 Instruction Memory 中的值取出
並依照 Big Endian 的方式送入 Instruction 中。

```
1 `define INSTR_MEM_SIZE 128 // Bytes
2 module IM(
3     //Input
4     input wire [31:0] Instr_Addr,
5     //Output
6     output [31:0] Instr
7 );
8 reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
9 assign Instr[31:24] = InstrMem[Instr_Addr];
10 assign Instr[23:16] = InstrMem[Instr_Addr+32'd1];
11 assign Instr[15:8] = InstrMem[Instr_Addr+32'd2];
12 assign Instr[7:0] = InstrMem[Instr_Addr+32'd3];
13 endmodule
```

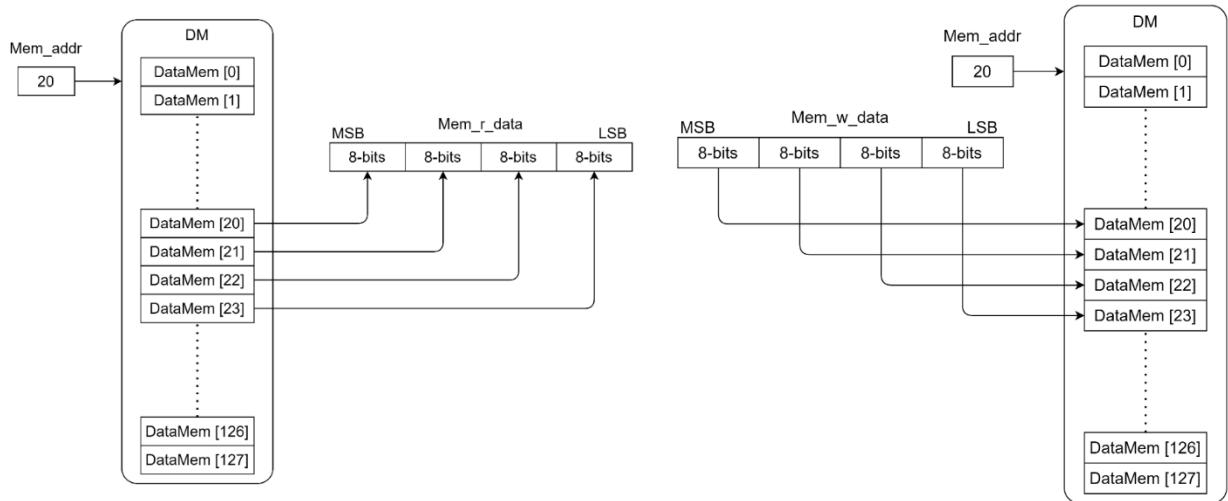


4. DM.v

根據下圖 Data Memory 讀寫操作完成該 Module 的功能。

當 MemWrite 為 High 時，在 Positive Edge 將資料寫入 Data Memory 裡面。

而 ReadData 則與 Clock 無關，當 MemRead 為 High 時即讀取 Memory 中的資料出來。



```

1  `define DATA_MEM_SIZE 128 // Bytes
2
3 module DM(
4   // Outputs
5   output [31:0] MemReadData,
6   // Inputs
7   input wire [31:0] MemAddr,
8   input wire [31:0] MemWriteData,
9   input wire MemWrite,
10  input wire MemRead,
11  input wire clk
12 );
13 reg [7:0]DataMem[0:DATA_MEM_SIZE - 1];
14
15 always @(posedge clk) begin
16   if(MemWrite) {DataMem[MemAddr],DataMem[MemAddr+1],DataMem[MemAddr+2],DataMem[MemAddr+3]} <= MemWriteData;
17   else;
18 end
19 assign MemReadData = (MemRead)? {DataMem[MemAddr],DataMem[MemAddr+1],DataMem[MemAddr+2],DataMem[MemAddr+3]} : 32'd0;
20
21 endmodule

```

5. Control.v

根據以下 10 個指令的操作，設定好每個控制線應有的操作。

Instruction	Example	Meaning	OpCode	Funct_ctrl	Funct
Add unsigned	Addu \$Rd,\$Rs,\$Rt	\$Rd = \$Rs + \$Rt	000000	100001	001001
Sub unsigned	Subu \$Rd,\$Rs,\$Rt	\$Rd = \$Rs - \$Rt	000000	100011	001010
Shift left logical	Sll \$Rd,\$Rs,Shamt	\$Rd = \$Rs << Shampt	000000	000000	100001
OR	Or \$Rd,\$Rs,\$Rt	\$Rd = \$Rs \$Rt	000000	100101	100101

Instruction	Example	Meaning	OpCode	Funct
Add imm unsigned	addiu \$Rt,\$Rs,Imm.	\$Rt = \$Rs + Imm.	001001	001001
Store word	Sw \$Rt,Imm.(\$Rs)	Mem.[\$Rs+Imm.] = \$Rt	101011	001001
Load word	Lw \$Rt,Imm.(\$Rs)	\$Rt = Mem.[\$Rs+Imm.]	100011	001001
Or Immediate	Ori \$Rt,\$Rs,Imm.	\$Rt = \$Rs Imm.	001101	100101

Instruction	Example	Meaning	OpCode	Funct
Branch on equal	Beq \$Rs,\$Rt,Imm.	if (\$Rs ≡ \$Rt) Output_Addr = Input_Addr + 4 + Imm.* 4	000100	001010
Jump	J Imm.	Output_Addr = NextPC[31:28] Imm.* 4	000010	001010

```

1  `define R_Format 6'b000000          37
2  `define Addi 6'b001001           38
3  `define Sw 6'b101011            39
4  `define Lw 6'b100011            40
5  `define Ori 6'b001101           41
6  `define Beq 6'b000100           42
7  `define Jump 6'b000010          43
8
9  module Control(
10    //Input
11    input wire [5:0] OpCode,
12    //Output
13    output reg Reg_dst,
14    output reg Branch,
15    output reg RegWrite,
16    output reg [1:0] ALU_OP,
17    output reg ALU_src,
18    output reg Mem_w,
19    output reg Mem_r,
20    output reg Mem_to_reg,
21    output reg Jump
22  );
23
24  always @(*) begin
25    case(OpCode)
26      `R_Format: begin // R-Format 6'b000000
27        ALU_OP = 2'b10;
28        RegWrite = 1'b1;
29        Reg_dst = 1'b1;
30        Branch = 1'b0;
31        ALU_src = 1'b0;
32        Mem_w = 1'b0;
33        Mem_r = 1'b0;
34        Mem_to_reg = 1'b0;
35        Jump = 1'b0;
36      end
37
38      `Addi: begin // I-Format Addi 6'b001001
39        ALU_OP = 2'b00;
40        RegWrite = 1'b1;
41        Reg_dst = 1'b0;
42        Branch = 1'b0;
43        ALU_src = 1'b1;
44        Mem_w = 1'b0;
45        Mem_r = 1'b0;
46        Mem_to_reg = 1'b0;
47        Jump = 1'b0;
48
49      end
50
51      `Sw: begin // I-Format Store Word 6'b101011
52        ALU_OP = 2'b00;
53        RegWrite = 1'b0;
54        Reg_dst = 1'b0;
55        Branch = 1'b0;
56        ALU_src = 1'b1;
57        Mem_w = 1'b1;
58        Mem_r = 1'b0;
59        Mem_to_reg = 1'b0;
60        Jump = 1'b0;
61
62      end
63
64      `Lw: begin // I-Format Load Word 6'b100011
65        ALU_OP = 2'b00;
66        RegWrite = 1'b1;
67        Reg_dst = 1'b0;
68        Branch = 1'b0;
69        ALU_src = 1'b1;
70        Mem_w = 1'b0;
71        Mem_r = 1'b1;
72        Mem_to_reg = 1'b0;
73        Jump = 1'b0;
74
75      end
76
77      `Ori: begin // I-Format Ori 6'b001101
78        ALU_OP = 2'b01;
79        RegWrite = 1'b1;
80        Reg_dst = 1'b0;
81        Branch = 1'b0;
82        ALU_src = 1'b1;
83        Mem_w = 1'b0;
84        Mem_r = 1'b0;
85        Mem_to_reg = 1'b0;
86        Jump = 1'b0;
87
88      end
89
90      `Beq: begin // I-Format Beq 6'b000100
91        ALU_OP = 2'b11;
92        RegWrite = 1'b0;
93        Reg_dst = 1'b0;
94        Branch = 1'b1;
95        ALU_src = 1'b0;
96        Mem_w = 1'b0;
97        Mem_r = 1'b0;
98        Mem_to_reg = 1'b0;
99        Jump = 1'b0;
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
endmodule

```

6. ALU_Control.v

一樣根據上面 10 個指令的操作，設定好每個控制線應有的操作。

```

1  `define ADD 6'b001001
2  `define SUB 6'b001010
3  `define SHIFT 6'b100001
4  `define OR 6'b100101
5
6  module ALU_Control(
7    //Input
8    input [1:0] ALU_OP,
9    input [5:0] funct_ctrl,
10   //Output
11   output reg [5:0] funct
12 );
13
14 always @(*) begin
15   case(ALU_OP)
16     2'b10:begin // R-Format
17       case(funct_ctrl)
18         6'b100001: funct = `ADD;
19         6'b100011: funct = `SUB;
20         6'b000000: funct = `SHIFT;
21         6'b100101: funct = `OR;
22         default:;
23       endcase
24     end
25     2'b00: funct = `ADD; // I-Format Add
26     2'b01: funct = `OR; // I-Format Or
27     2'b11: funct = `SUB; // I/J-Format Sub
28   endcase
29 endmodule

```

7. ALU.v

根據上面 10 個指令的操作，設定好每個指令應有的算術處理。

```
1 `define ADD 6'b001001
2 `define SUB 6'b001010
3 `define SHIFT 6'b100001
4 `define OR 6'b100101
5
6 module ALU(
7     //Input
8     input wire [31:0] Rs_Data,
9     input wire [31:0] Rt_Data,
10    input wire [4:0] shamt,
11    input wire [5:0] funct,
12    //Output
13    output reg [31:0] Rd_Data,
14    output Zero_Flag
15 );
16
17 always @(*) begin
18     case(funct)
19         `ADD: Rd_Data = Rs_Data + Rt_Data;
20         `SUB: Rd_Data = Rs_Data - Rt_Data;
21         `SHIFT: Rd_Data = Rs_Data << shamt;
22         `OR: Rd_Data = Rs_Data | Rt_Data;
23         default:;
24     endcase
25 end
26 assign Zero_Flag = (Rd_Data==32'd0);
27
28 endmodule
```

8. Adder.v

做指令位置的加法運算(考慮 Branch and Jump)。

```
1 `module Adder(
2     //Input
3     input wire [31:0] Src1,
4     input wire [31:0] Src2,
5     //Output
6     output [31:0] Output_Addr
7 );
8     assign Output_Addr = Src1 + Src2;
9
10 endmodule
```

b. Descriptions of how you test your modules and the execution results

Part.1

用以下四個 Pattern 驗證 R-Format 加法、減法、向左位移、OR 運算。

Pattern	Opcode	Rs_Addr	Rt_Addr	Rd_Addr	Shamt	Funct	Result(Rd_Data)
1	000000	10	11	20	0	001001	$160 + 10 = 170$
2	000000	12	13	21	0	001010	$2 - 1 = 1$
3	000000	14	15	24	3	100001	$3 \ll 3 = 27$
4	000000	17	18	27	4	100101	$100 55 = 119$

可以觀察以下 RF.out 去驗證 Result

Pattern	Result Addr (Rd_Addr)	Result (Rd_Data)	Correct
1	20	$aa_{hex} = 170_{dec}$	✓
2	21	$1_{hex} = 1_{dec}$	✓
3	24	$18_{hex} = 27_{dec}$	✓
4	27	$77_{hex} = 119_{dec}$	✓

Part.3

利用以下指令來測試 Branch 以及 Jump 的指令是否正確

```
beq $r19, $r0, 30
sub $r19, $r19, $r2
jump 0
```

而觀察 RF.dat 可知 \$r19 初始值為 64、\$r2 初始值為 3、\$r0 初始值為 1

故要減 21 次 Address 才會等於 30

當 beq 成立後會跳到 Address = (PC+4) + (Immediate*4) = (0+4) + (30*4) = 124

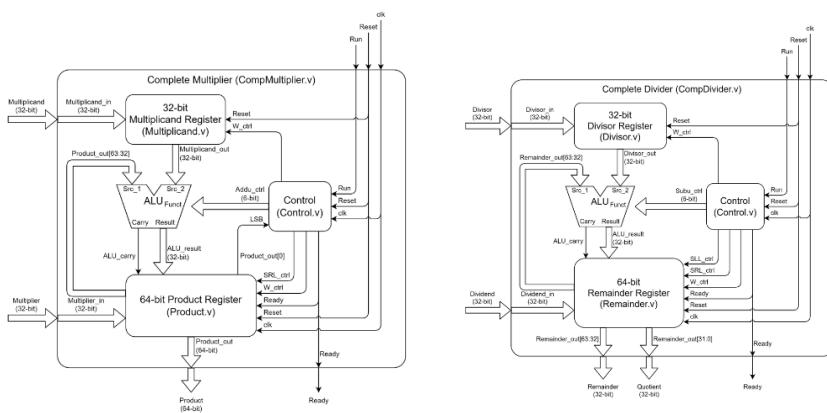
+ /tb_SimpleCPU/Input_Addr	-No Data-	4	8	0	4	8	0	124
+ /tb_SimpleCPU/Output_Addr	-No Data-	8	0	4	8	0	124	128
+ /tb_SimpleCPU/UUT/Immediate	-No Data-	429494...	0	30	429494...	0	30	4...
/tb_SimpleCPU/dlk	-No Data-							

故可以觀察到 RF.out 中的 \$r19 為 1

RF.dat		RF.out	
檔案	編輯	檔案	檢視
// Register File in Hex	00000001	00000001	00000001
0000_0001 // R[0]	0000_0001 // R[1]	00000003	
0000_0001 // R[2]	77777777		
7777_7777 // R[3]	7f7f7f7f		
7F7F_7F7F // R[4]	f7f7f7f7		
F7F7_F7F7 // R[5]	7fffffff		
7FFF_FFFF // R[6]	80000000		
8000_0000 // R[7]	fffff000		
FFFF_0000 // R[8]	0000ffff		
0000_FFFF // R[9]	0000000a		
0000_000A // R[10]	000000a0		
0000_00A0 // R[11]	00000002		
0000_0002 // R[12]	00000001		
0000_0001 // R[13]	00000003		
0000_0003 // R[14]	00000007		
0000_0007 // R[15]	00000002		
0000_0002 // R[16]	00000037		
0000_0037 // R[17]	00000064		
0000_0064 // R[18]	00000001		
0000_0040 // R[19]	00000000		

c. Datapath rethinking

If you were required to implement the multiplier and divider (from PA1) in a single-cycle CPU, how would you revise the datapath? Only a brief description or idea is needed.



從 PA1 的 Multiplier 跟 Divider 架構圖可知(如上圖)

只要添加對應的 Register 並用 Control 對 ALU 以及 Register 進行控制，即可完成操作。

但因為在 PA1 中的乘法與除法器共需要 32 以及 33 個 Clock 才可以算完一個乘法或除法操作。對於 Single Cycle 的 CPU 來說，一個指令必須在一個周期內做完，因此就算照上面的說法連接 Data Path，也沒有辦法達到正確的電路行為。

對於這個問題，以下是我的一些解決想法

1. 乘除法操作時固定 PC，當執行到乘法或是除法的時候，PC 不會改變直到乘法或除法操作結束，這樣下一個 Instruction 就不會送進來 IM 裡面打亂整個指令操作，整體而言是一個設計簡單但需要犧牲一些 Performance 的方式。
2. 讓乘除法單元的 Clock 頻率比整個 CPU 的 Clock 還要快，可以利用除頻器的概念，讓 CPU 的 RF、DM、接受到的是較低頻率(1/32)的 Clock，從而使得乘除法的操作可以在整個 CPU 的 Single Cycle 下完成，這個方法因為降低了整體 CPU 的頻率，所以會導致 Performance 下降，而如果要保持整個 CPU 還是跟原來相同的 Clock 頻率，讓乘除法單元的 Clock 變得很慢，可能會導致功耗增加或是時序約束問題。
3. 使用 Pipeline 來設計乘法與除法的操作，但是具體的設計需要考慮很多，比如 Hazard、時序等問題，雖然設計較前兩個來說更複雜，但 Performance 應該會很不錯。

d. Conclusion and insights

這次的 PA2 是讓我們實現一個簡單的 Single-Cycle 的 CPU，上學期在做數位系統設計實習的時候，本來想要用 FPGA 實現一個 MIPS 架構的 CPU，但因為期末報告是需要做遊戲相關的，因此 CPU 的開發就停滯了很大的時間，而在這學期的計組課上，總算是讓我把這個以前挖過的大坑填上了，雖然目前做的只是一顆十個指令的 MIPS CPU，但還是滿有成就感的，也可以讓我通過這個架構去延伸到更多指令的 MIPS CPU，充實自己的 Side Project，整體而言這次的 PA 對我來說也算簡單，其實就只是把計組上課學到的 CPU 架構的 Data Path 連接起來，而對於優化面積與時序方面，因為本次 PA 含有 Memory 以及 Register File 這兩個非常大面積的 Macro，所以面積能優化的並不多，關於時序的部分除了盡量優化自己的 Critical Path 以外，我還有上網找了相關的 MIPS CPU 優化的論文，比較大的時序優化是利用近似計算來達到，但因為跟本次 PA 方向不符，因此就沒有實作，總而言之，這次的 PA 讓我學習到很多，非常期待下一次實作 Pipeline 的 MIPS CPU !!!