Computer Organization

PA3

(5-stage pipelined processor with R/I Format, forwarding and hazard detection)

Student

B11107157 電機三乙 林明宏

RTL Simulator & Synthesis Tool

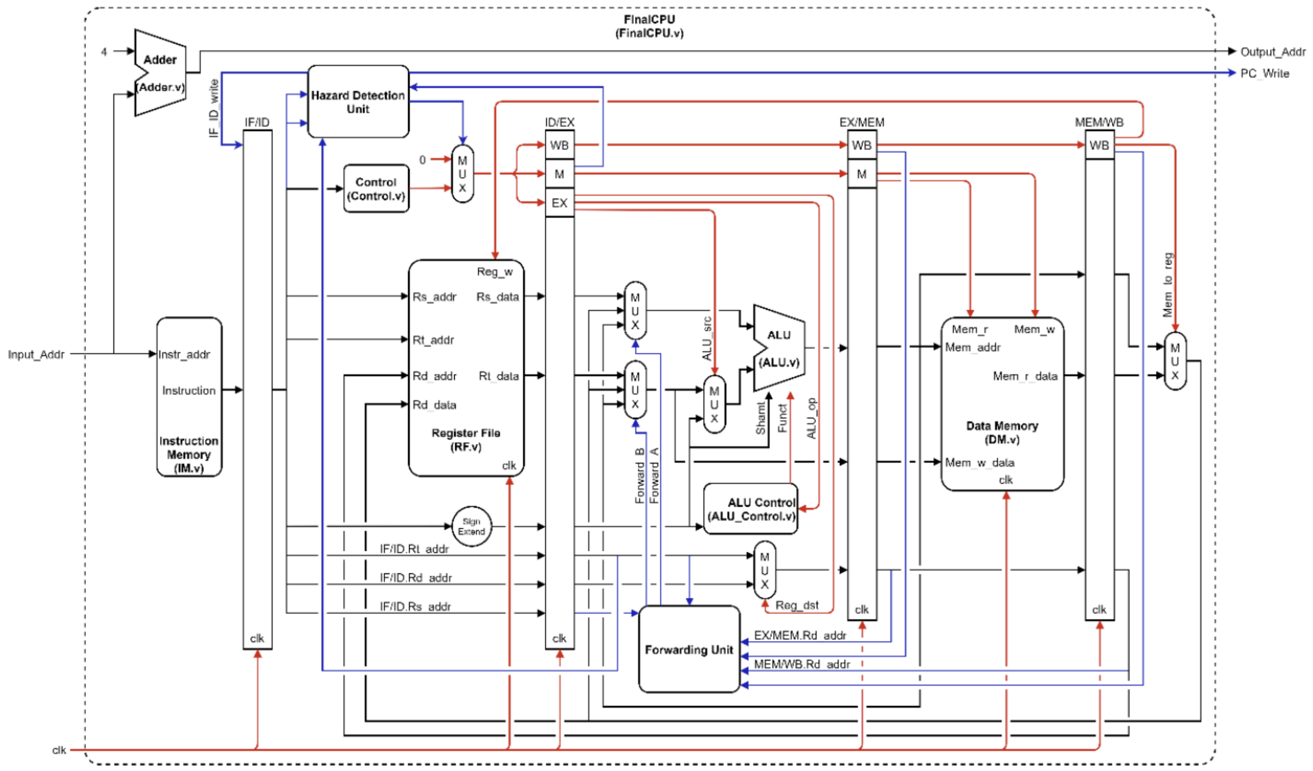
ModelSim-Intel FPGA Standard Edition, Version 20.1.1, windows
The OpenROAD-flow-scripts from github

Slack & Area Report

Critical Path Slack : 1.7721 | Area : 18500.034 um^2

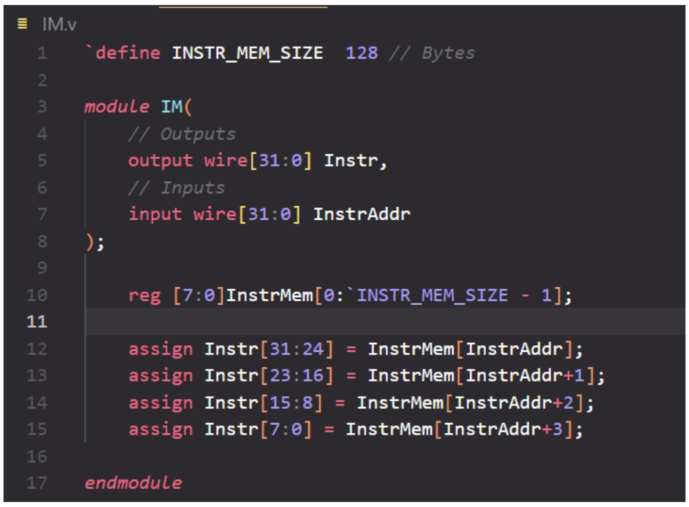
a. Descriptions of how you implement each module

1. FinalCPU.v



根據上述架構圖，建構每個 Module 的 Data Path，以及一些內部的 Mux 邏輯撰寫，至於 PC 的 Adder 則沒有再創一個新的 Module，而是直接用 assignment 對 Output Address + 4 ，可以有效提高運算速度
(我猜是因為加固定數，Openroad 會幫助我合成為較快速的 Constant Adder)。
建構好整體的 Data Path 後，我開始撰寫各個 Module 內部的功能。

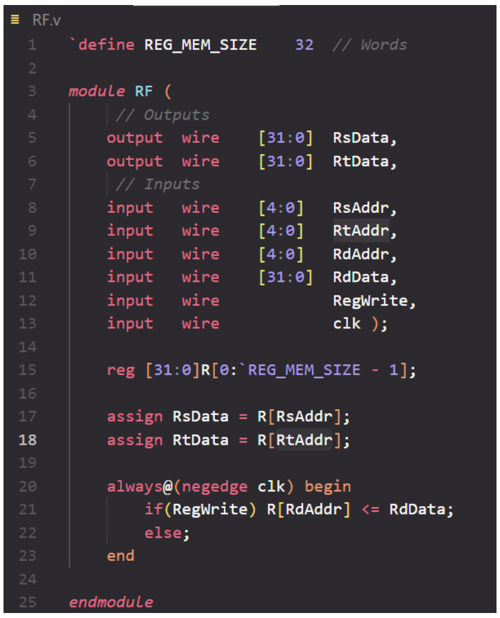
2. IM.v



```verilog
`define INSTR_MEM_SIZE  128 // Bytes

module IM(
    // Outputs
    output wire[31:0] Instr,
    // Inputs
    input wire[31:0] InstrAddr
);

    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];

    assign Instr[31:24] = InstrMem[InstrAddr];
    assign Instr[23:16] = InstrMem[InstrAddr+1];
    assign Instr[15:8] = InstrMem[InstrAddr+2];
    assign Instr[7:0] = InstrMem[InstrAddr+3];

endmodule
```

基本上和之前一樣，利用輸入的 Address，進去 Instruction Memory 找出指令並放到 Instruction 裡面。

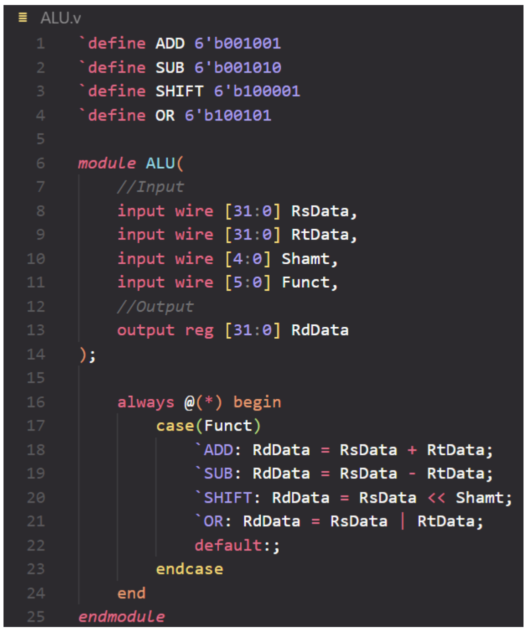
3. RF.v



```
`define REG_MEM_SIZE    32  // Words

module RF (
    // Outputs
    output  wire    [31:0]  RsData,
    output  wire    [31:0]  RtData,
    // Inputs
    input   wire    [4:0]   RsAddr,
    input   wire    [4:0]   RtAddr,
    input   wire    [4:0]   RdAddr,
    input   wire    [31:0]  RdData,
    input   wire            RegWrite,
    input   wire            clk );

    reg [31:0]R[0:`REG_MEM_SIZE - 1];

    assign RsData = R[RsAddr];
    assign RtData = R[RtAddr];

    always@(negedge clk) begin
        if(RegWrite) R[RdAddr] <= RdData;
        else;
    end

endmodule
```

與之前一樣，將 Instruction 分析後拆成 RsAddr、RtAddr，而 RdAddr、跟 RdData 則是從後面 WB 拉回來即可。

4. ALU.v



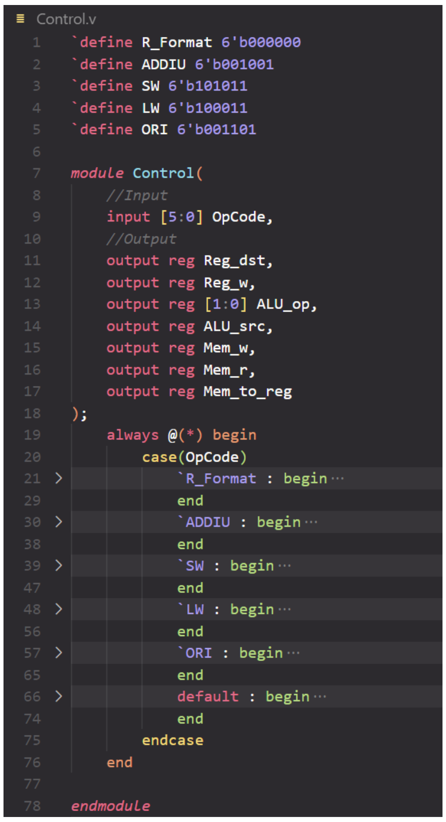
```
`define ADD 6'b001001
`define SUB 6'b001010
`define SHIFT 6'b100001
`define OR 6'b100101

module ALU(
    //Input
    input wire [31:0] RsData,
    input wire [31:0] RtData,
    input wire [4:0] Shamt,
    input wire [5:0] Funct,
    //Output
    output reg [31:0] RdData
);

    always @(*) begin
        case(Funct)
            `ADD: RdData = RsData + RtData;
            `SUB: RdData = RsData - RtData;
            `SHIFT: RdData = RsData << Shamt;
            `OR: RdData = RsData | RtData;
            default:;
        endcase
    end
endmodule
```

本次 PA 只實現加法、減法、位移跟 OR 操作，根據輸入的 Function 分配指定的運算式。

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 << Shamt	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

5. Control.v

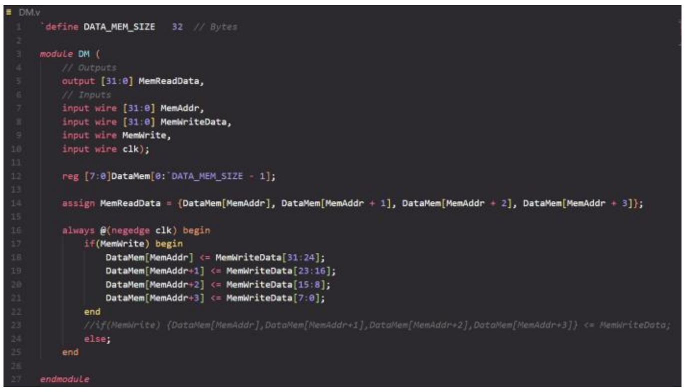


Control.v

```
`define R_Format 6'b000000
`define ADDIU 6'b001001
`define SW 6'b101011
`define LW 6'b100011
`define ORI 6'b001101

module Control(
    //Input
    input [5:0] OpCode,
    //Output
    output reg Reg_dst,
    output reg Reg_w,
    output reg [1:0] ALU_op,
    output reg ALU_src,
    output reg Mem_w,
    output reg Mem_r,
    output reg Mem_to_reg
);
    always @(*) begin
        case(OpCode)
            `R_Format : begin···
            end
            `ADDIU : begin···
            end
            `SW : begin···
            end
            `LW : begin···
            end
            `ORI : begin···
            end
            default : begin···
            end
        endcase
    end

endmodule
```

如上所示運算式，控制好每個 Control Signal，讓 ALU 和其他 Module 正確操作運算式。

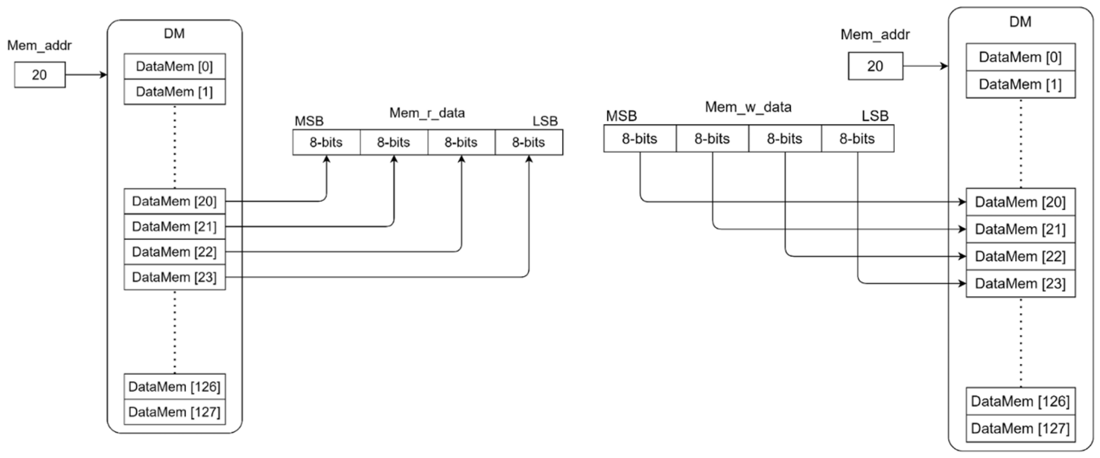
6. DM.v



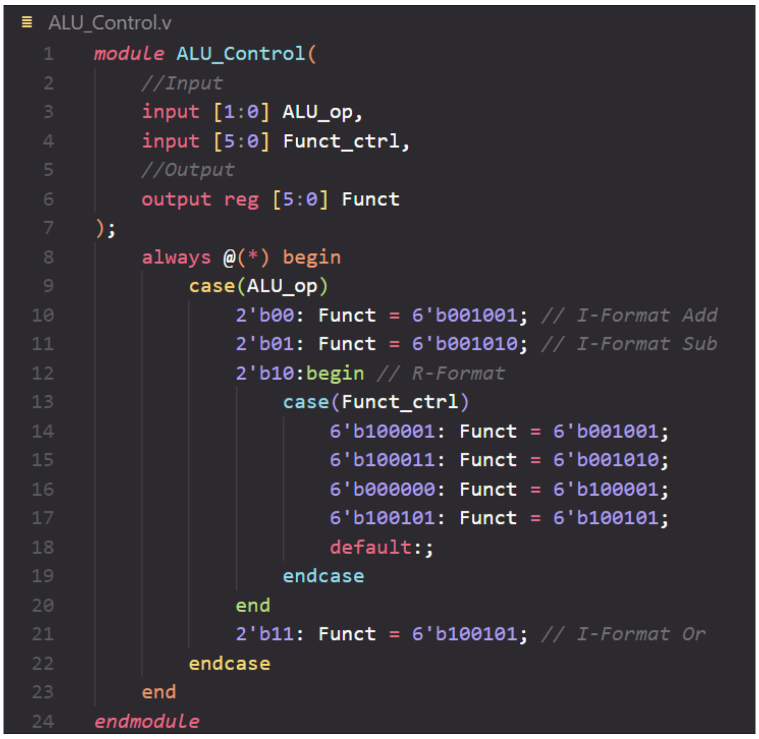
DM.v

```
`define DATA_MEM_SIZE   32  // Bytes

module DM (
    // Outputs
    output [31:0] MemReadData,
    // Inputs
    input wire [31:0] MemAddr,
    input wire [31:0] MemWriteData,
    input wire MemWrite,
    input wire clk);

    reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];

    assign MemReadData = {DataMem[MemAddr], DataMem[MemAddr + 1], DataMem[MemAddr + 2], DataMem[MemAddr + 3]};

    always @(negedge clk) begin
        if(MemWrite) begin
            DataMem[MemAddr] <= MemWriteData[31:24];
            DataMem[MemAddr+1] <= MemWriteData[23:16];
            DataMem[MemAddr+2] <= MemWriteData[15:8];
            DataMem[MemAddr+3] <= MemWriteData[7:0];
        end
        //if(MemWrite) {DataMem[MemAddr],DataMem[MemAddr+1],DataMem[MemAddr+2],DataMem[MemAddr+3]} <= MemWriteData;
        else;
    end

endmodule
```

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，讀取 Memory 中的資料出來。

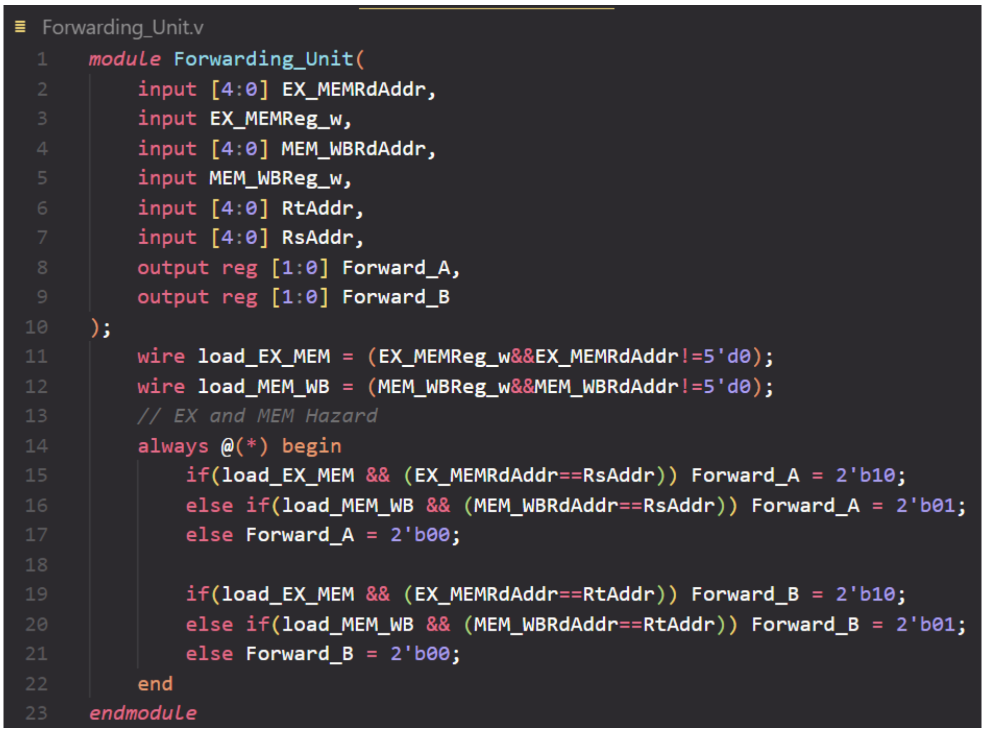


7. ALU_Control.v


ALU_Control.v
```verilog
module ALU_Control(
    //Input
    input [1:0] ALU_op,
    input [5:0] Funct_ctrl,
    //Output
    output reg [5:0] Funct
);
    always @(*) begin
        case(ALU_op)
            2'b00: Funct = 6'b001001; // I-Format Add
            2'b01: Funct = 6'b001010; // I-Format Sub
            2'b10:begin // R-Format
                case(Funct_ctrl)
                    6'b100001: Funct = 6'b001001;
                    6'b100011: Funct = 6'b001010;
                    6'b000000: Funct = 6'b100001;
                    6'b100101: Funct = 6'b100101;
                    default:;
                endcase
            end
            2'b11: Funct = 6'b100101; // I-Format Or
        endcase
    end
endmodule
```

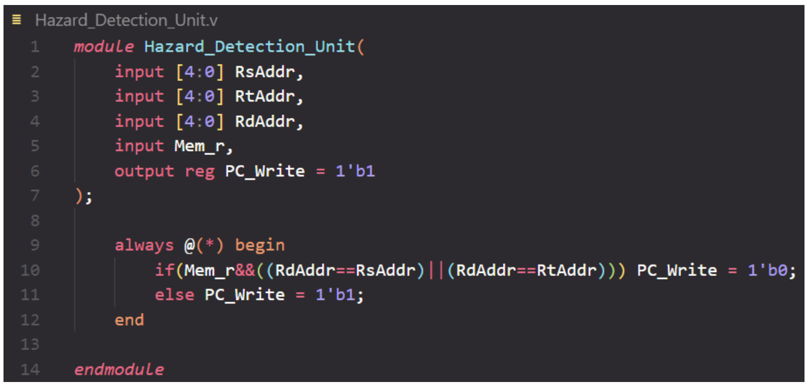
一樣根據上面指令的操作，根據 ALU_op 設定好控制信號 Funct 應有的操作。

8. Forwarding_Unit.v


Forwarding_Unit.v
```verilog
module Forwarding_Unit(
    input [4:0] EX_MEMRdAddr,
    input EX_MEMReg_w,
    input [4:0] MEM_WBRdAddr,
    input MEM_WBReg_w,
    input [4:0] RtAddr,
    input [4:0] RsAddr,
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
);
    wire load_EX_MEM = (EX_MEMReg_w&&EX_MEMRdAddr!=5'd0);
    wire load_MEM_WB = (MEM_WBReg_w&&MEM_WBRdAddr!=5'd0);
    // EX and MEM Hazard
    always @(*) begin
        if(load_EX_MEM && (EX_MEMRdAddr==RsAddr)) Forward_A = 2'b10;
        else if(load_MEM_WB && (MEM_WBRdAddr==RsAddr)) Forward_A = 2'b01;
        else Forward_A = 2'b00;

        if(load_EX_MEM && (EX_MEMRdAddr==RtAddr)) Forward_B = 2'b10;
        else if(load_MEM_WB && (MEM_WBRdAddr==RtAddr)) Forward_B = 2'b01;
        else Forward_B = 2'b00;
    end
endmodule
```

根據上課所學去撰寫 Forwarding Unit，在 EXE_MEMReg_w = 1 時代表目前這個指令會做 Register 寫入操作，並且寫入的 Address 不是 R0，當該 Address 等於 RsAddr 時代表需要複寫掉目前的 RsData，以避免 Data Hazard，同一個概念以此類推下去，即可完整整個 Forwarding Unit 的設計。

9. Hazard_Detection_Unit



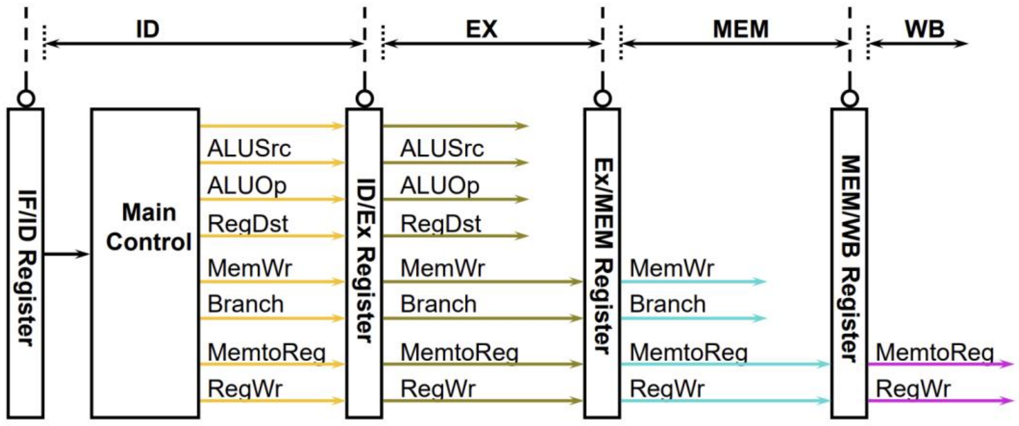
Hazard_Detection_Unit.v

```verilog
module Hazard_Detection_Unit(
    input [4:0] RsAddr,
    input [4:0] RtAddr,
    input [4:0] RdAddr,
    input Mem_r,
    output reg PC_Write = 1'b1
);

    always @(*) begin
        if(Mem_r&&((RdAddr==RsAddr)||(RdAddr==RtAddr))) PC_Write = 1'b0;
        else PC_Write = 1'b1;
    end

endmodule
```

根據上課所學建構 Hazard_Detection_Unit，當需要 Stall 時，把 hazard bit、IF_ID_write、PC_write 歸 0，使控制線歸 0、下一個指令停止寫入，以便達到 Stall，從上述可以發現，hazard bit、IF_ID_write、PC_write 其實在做同一件事情，所以可以把它縮減成一個 PC_write。

10. Stage Pipeline.v(IF_ID、ID_EX、EX_MEM、MEM_WB)

根據以下這張圖，完成每一個 Stage，Control Signal 的傳遞。



b. Describes the custom test and analyzes its results in each part then compare the timing, area and power against both the SimpleCPU (PA2) and the FinalCPU (PA3).

1. The custom test and analyzes its results in each part

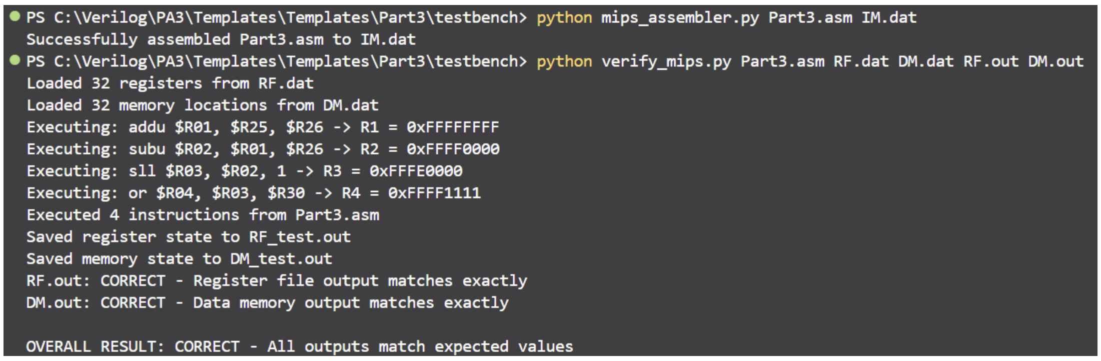
在測試結果的階段，我利用 Python 來幫助我測試結果，主要通過兩種程式，第一個是 MIPS Assembler，它可以幫助我將寫好的.asm 檔轉成 Instruction 並存入 IM.dat 中，供我的程式使用。第二個是 MIPS Simulator and Verify，它可以幫助我將.asm 檔的運算式讀取 DM.dat、RF.dat 內的資料並將正確答案算出並存入 DM_test.out 跟 RF_test.out，接著將 test_out 跟我用 RTL 透過 ModelSim 模擬出來的.out 檔做 Check，如果將正確代表 This Test Pattern Correct，如果有錯誤則會幫我標出錯誤，以此來增加驗證速度。

R_pipelineCPU

Test Pattern and Python Verify :

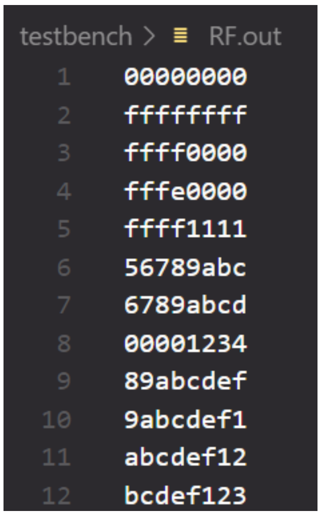
(Terminal will print the operation and correct answer in register or memory)

```
addu   $R01, $R25, $R26
subu   $R02, $R01, $R26
sll    $R03, $R02, 1
or     $R04, $R03, $R30
```



```
PS C:\Verilog\PA3\Templates\Templates\Part3\testbench> python mips_assembler.py Part3.asm IM.dat
Successfully assembled Part3.asm to IM.dat
PS C:\Verilog\PA3\Templates\Templates\Part3\testbench> python verify_mips.py Part3.asm RF.dat DM.dat RF.out DM.out
Loaded 32 registers from RF.dat
Loaded 32 memory locations from DM.dat
Executing: addu $R01, $R25, $R26 -> R1 = 0xFFFFFFFF
Executing: subu $R02, $R01, $R26 -> R2 = 0xFFFF0000
Executing: sll $R03, $R02, 1 -> R3 = 0xFFFE0000
Executing: or $R04, $R03, $R30 -> R4 = 0xFFFF1111
Executed 4 instructions from Part3.asm
Saved register state to RF_test.out
Saved memory state to DM_test.out
RF.out: CORRECT - Register file output matches exactly
DM.out: CORRECT - Data memory output matches exactly

OVERALL RESULT: CORRECT - All outputs match expected values
```

RF.out :



testbench > RF.out

```
00000000
ffffffff
ffff0000
fffe0000
ffff1111
56789abc
6789abcd
00001234
89abcdef
9abcdef1
abcdef12
bcdef123
```

I_pipelineCPU

Test Pattern and Python Verify :

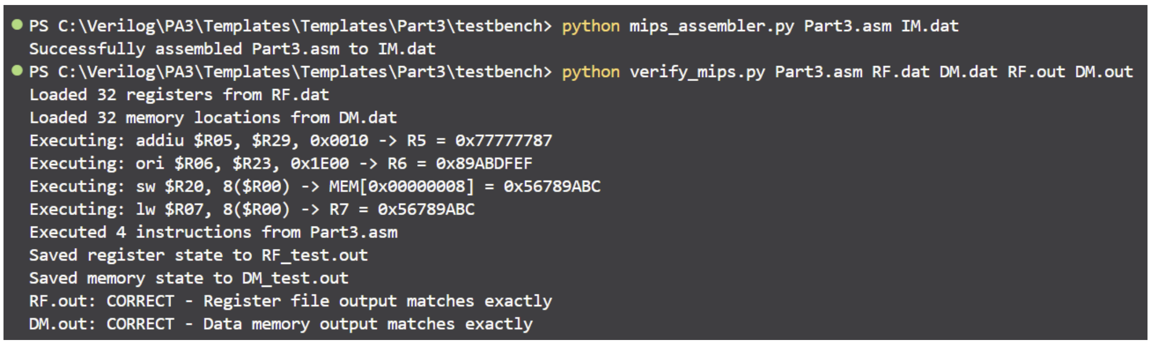
(Terminal will print the operation and correct answer in register or memory)

addiu $R05, $R29, 16

ori $R06, $R23, 0x1E00

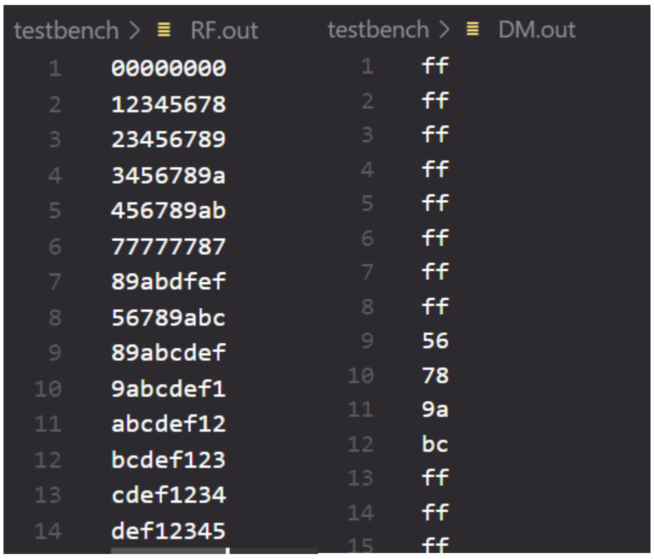
sw $R20, 8($R0)

lw $R07, 8($R0)



```
● PS C:\Verilog\PA3\Templates\Templates\Part3\testbench> python mips_assembler.py Part3.asm IM.dat
  Successfully assembled Part3.asm to IM.dat
● PS C:\Verilog\PA3\Templates\Templates\Part3\testbench> python verify_mips.py Part3.asm RF.dat DM.dat RF.out DM.out
  Loaded 32 registers from RF.dat
  Loaded 32 memory locations from DM.dat
  Executing: addiu $R05, $R29, 0x0010 -> R5 = 0x77777787
  Executing: ori $R06, $R23, 0x1E00 -> R6 = 0x89ABDFEF
  Executing: sw $R20, 8($R00) -> MEM[0x00000008] = 0x56789ABC
  Executing: lw $R07, 8($R00) -> R7 = 0x56789ABC
  Executed 4 instructions from Part3.asm
  Saved register state to RF_test.out
  Saved memory state to DM_test.out
  RF.out: CORRECT - Register file output matches exactly
  DM.out: CORRECT - Data memory output matches exactly
```

RF.out and DM.out :



```
testbench > RF.out
00000000
12345678
23456789
3456789a
456789ab
77777787
89abdfef
56789abc
89abcdef
9abcdef1
abcdef12
bcdef123
cdef1234
def12345
```

```
testbench > DM.out
ff
ff
ff
ff
ff
ff
ff
ff
56
78
9a
bc
ff
ff
ff
```

FinalCPU

Test Pattern and Python Verify :

(Terminal will print the operation and correct answer in register or memory)

addiu $R01, $R0, 10

addiu $R02, $R0, 5

addiu $R03, $R0, 15

addu $R04, $R01, $R02

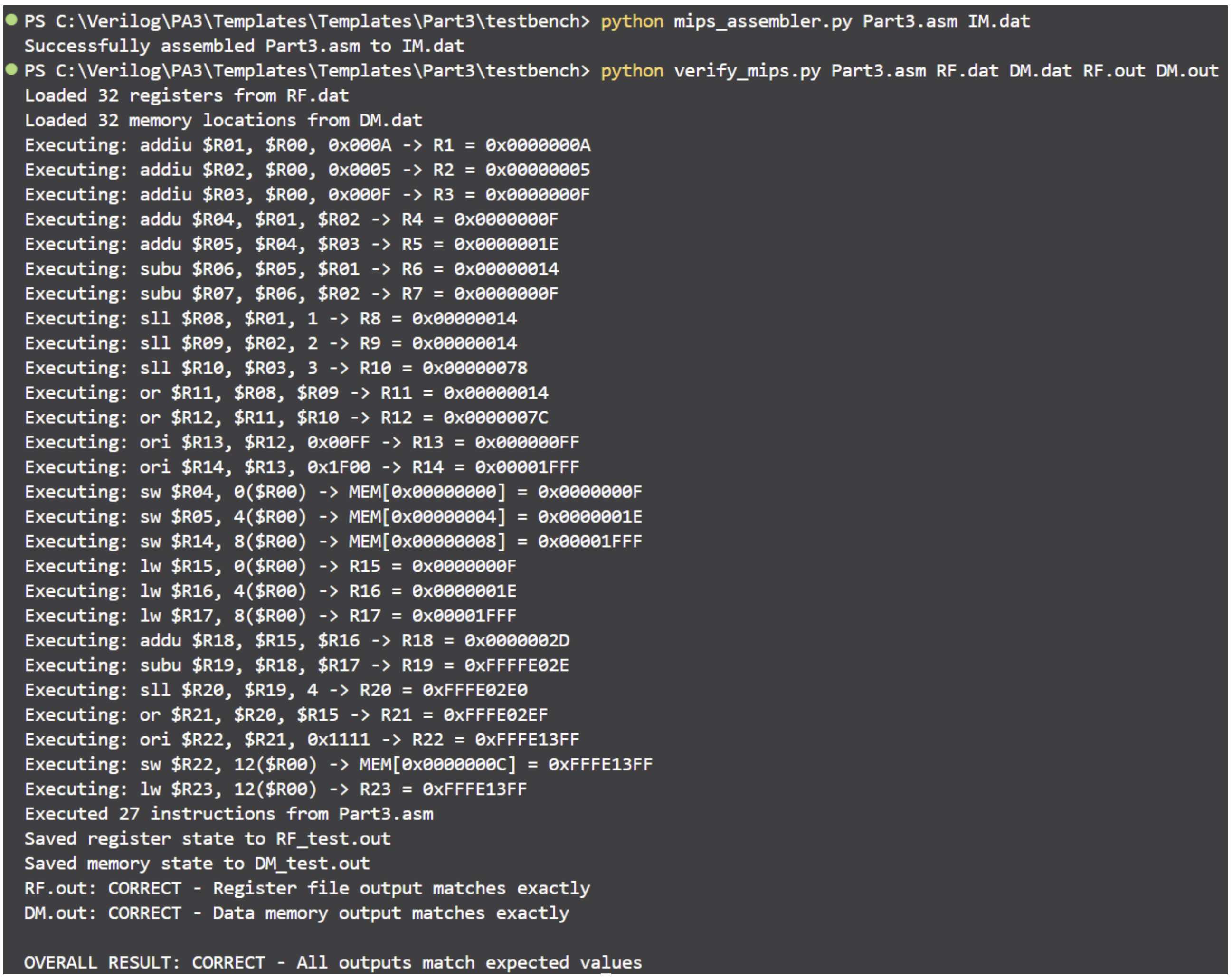
addu $R05, $R04, $R03

subu $R06, $R05, $R01

subu $R07, $R06, $R02

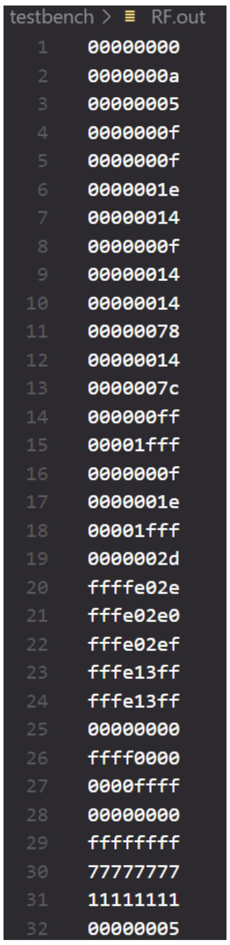
sll $R08, $R01, 1

```
sll     $R09, $R02, 2
sll     $R10, $R03, 3
or      $R11, $R08, $R09
or      $R12, $R11, $R10
ori     $R13, $R12, 0x00FF
ori     $R14, $R13, 0x1F00
sw      $R04, 0($R0)
sw      $R05, 4($R0)
sw      $R14, 8($R0)
lw      $R15, 0($R0)
lw      $R16, 4($R0)
lw      $R17, 8($R0)
addu    $R18, $R15, $R16
subu    $R19, $R18, $R17
sll     $R20, $R19, 4
or      $R21, $R20, $R15
ori     $R22, $R21, 0x1111
sw      $R22, 12($R0)
lw      $R23, 12($R0)
```



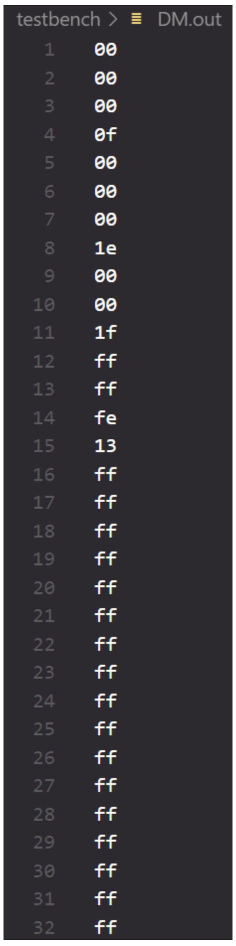
```
PS C:\Verilog\PA3\Templates\Templates\Part3\testbench> python mips_assembler.py Part3.asm IM.dat
Successfully assembled Part3.asm to IM.dat
PS C:\Verilog\PA3\Templates\Templates\Part3\testbench> python verify_mips.py Part3.asm RF.dat DM.dat RF.out DM.out
Loaded 32 registers from RF.dat
Loaded 32 memory locations from DM.dat
Executing: addiu $R01, $R00, 0x000A -> R1 = 0x0000000A
Executing: addiu $R02, $R00, 0x0005 -> R2 = 0x00000005
Executing: addiu $R03, $R00, 0x000F -> R3 = 0x0000000F
Executing: addu $R04, $R01, $R02 -> R4 = 0x0000000F
Executing: addu $R05, $R04, $R03 -> R5 = 0x0000001E
Executing: subu $R06, $R05, $R01 -> R6 = 0x00000014
Executing: subu $R07, $R06, $R02 -> R7 = 0x0000000F
Executing: sll $R08, $R01, 1 -> R8 = 0x00000014
Executing: sll $R09, $R02, 2 -> R9 = 0x00000014
Executing: sll $R10, $R03, 3 -> R10 = 0x00000078
Executing: or $R11, $R08, $R09 -> R11 = 0x00000014
Executing: or $R12, $R11, $R10 -> R12 = 0x0000007C
Executing: ori $R13, $R12, 0x00FF -> R13 = 0x000000FF
Executing: ori $R14, $R13, 0x1F00 -> R14 = 0x00001FFF
Executing: sw $R04, 0($R00) -> MEM[0x00000000] = 0x0000000F
Executing: sw $R05, 4($R00) -> MEM[0x00000004] = 0x0000001E
Executing: sw $R14, 8($R00) -> MEM[0x00000008] = 0x00001FFF
Executing: lw $R15, 0($R00) -> R15 = 0x0000000F
Executing: lw $R16, 4($R00) -> R16 = 0x0000001E
Executing: lw $R17, 8($R00) -> R17 = 0x00001FFF
Executing: addu $R18, $R15, $R16 -> R18 = 0x0000002D
Executing: subu $R19, $R18, $R17 -> R19 = 0xFFFFE02E
Executing: sll $R20, $R19, 4 -> R20 = 0xFFFE02E0
Executing: or $R21, $R20, $R15 -> R21 = 0xFFFE02EF
Executing: ori $R22, $R21, 0x1111 -> R22 = 0xFFFE13FF
Executing: sw $R22, 12($R00) -> MEM[0x0000000C] = 0xFFFE13FF
Executing: lw $R23, 12($R00) -> R23 = 0xFFFE13FF
Executed 27 instructions from Part3.asm
Saved register state to RF_test.out
Saved memory state to DM_test.out
RF.out: CORRECT - Register file output matches exactly
DM.out: CORRECT - Data memory output matches exactly

OVERALL RESULT: CORRECT - All outputs match expected values
```

RF.out and DM.out :



testbench > RF.out

```
00000000
0000000a
00000005
0000000f
0000000f
0000001e
00000014
0000000f
00000014
00000014
00000078
00000014
0000007c
000000ff
00001fff
0000000f
0000001e
00001fff
0000002d
ffffe02e
fffe02e0
fffe02ef
fffe13ff
fffe13ff
00000000
ffff0000
0000ffff
00000000
ffffffff
77777777
11111111
00000005
```



testbench > DM.out

```
00
00
00
0f
00
00
00
1e
00
00
1f
ff
ff
fe
13
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
```

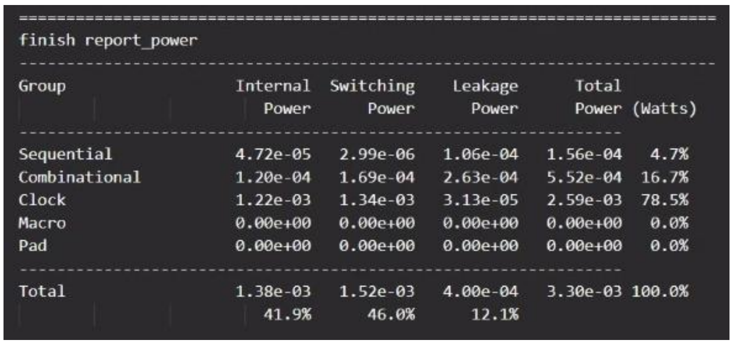
2. **compare the timing, area and power against both the SimpleCPU (PA2) and the FinalCPU (PA3)**

Instruction Memory Size = 128、Data Memory Size = 32

SimpleCPU :

Timing (Critical Path Slack) : 4.40 、Area : 17579

Power :

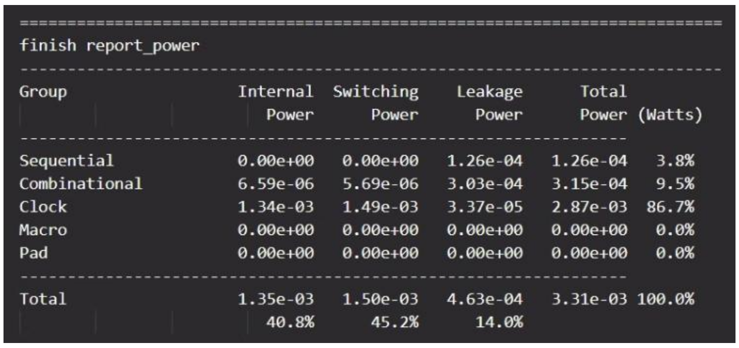


```
==========================================================================
finish report_power
--------------------------------------------------------------------------
Group                 Internal  Switching    Leakage      Total
                         Power      Power      Power      Power  (Watts)
-------------------------------------------------------------------
Sequential            4.72e-05   2.99e-06   1.06e-04   1.56e-04    4.7%
Combinational         1.20e-04   1.69e-04   2.63e-04   5.52e-04   16.7%
Clock                 1.22e-03   1.34e-03   3.13e-05   2.59e-03   78.5%
Macro                 0.00e+00   0.00e+00   0.00e+00   0.00e+00    0.0%
Pad                   0.00e+00   0.00e+00   0.00e+00   0.00e+00    0.0%
-------------------------------------------------------------------
Total                 1.38e-03   1.52e-03   4.00e-04   3.30e-03  100.0%
                         41.9%      46.0%      12.1%
```

FinalCPU :

Timing (Critical Path Slack) : 4.226(1.7260+2.5) 、Area : 18693

Power :



```
==========================================================================
finish report_power
--------------------------------------------------------------------------
Group                 Internal  Switching    Leakage      Total
                         Power      Power      Power      Power  (Watts)
-------------------------------------------------------------------
Sequential            0.00e+00   0.00e+00   1.26e-04   1.26e-04    3.8%
Combinational         6.59e-06   5.69e-06   3.03e-04   3.15e-04    9.5%
Clock                 1.34e-03   1.49e-03   3.37e-05   2.87e-03   86.7%
Macro                 0.00e+00   0.00e+00   0.00e+00   0.00e+00    0.0%
Pad                   0.00e+00   0.00e+00   0.00e+00   0.00e+00    0.0%
-------------------------------------------------------------------
Total                 1.35e-03   1.50e-03   4.63e-04   3.31e-03  100.0%
                         40.8%      45.2%      14.0%
```

因為 FinalCPU 有加入 Pipeline 與 Forward, Hazard Detection，因此面積與功耗較大。

c. **Memory Rethinking: If you were required to implement a multi-level cache in a pipelined CPU, how would you revise the datapath or design? Only a brief description or idea is needed**

在 5 stage pipeline CPU 中加入 multi-level cache，會增加 multi-level cache 的讀寫通路和控制邏輯，其 CPU 要先訪問 L1 cache，未命中才訪問 L2 或更低層。
並且需要設計快取控制器判斷命中與覆寫，去調整 pipeline 以處理因 multi-level cache 延遲造成的停頓（stall）。
同時要確保多層 cache 間資料一致性，並增加相關的快取狀態與控制信號來協調資料流和指令執行。

d. Conclusion and insights

這次的 Project 是要去實現 5-stage Pipeline with Forwarding and Hazard Detection，對我來說還滿有趣的，雖然在做 Pipeline 的 Data Path 需要細心一點，不能寫錯，但整體來說我覺得需要注意好整個 Data Path，這次的 PA 對我來說並不難，在課程結束後，我應該還會想要延伸寫寫看加入 Beq、Jump，讓整個 CPU 的指令集更加完整。
而在優化 Slack 與 Area 上面，我有做了幾個優化，第一個是將 PC+4 調整成+1，然後最後兩個 bit 固定為 0 的方法，再來就是調整 Sign-Extend 的位置，讓他往後一個 Stage，然後就是一些對於 default 的優化。

做完這次 Project，這堂課就差不多進入尾聲，我覺得收穫還不錯，通過這堂課，我累積一個製作 CPU 的作品集，希望在未來前往數位 IC 公司的路上，我可以多加延伸這個作品集，像是前面說的加入 Branch、Jump 的指令，甚至是加入 cache 的概念，進一步加快速度。
整體來說，我跟我的同學覺得這次的 Project 還是太簡單了，希望未來除了對 Area 跟 Critical Path Slack 可以加入 Power 的 Ranking，讓學弟妹們可以更投入在這堂課上面。

此外，我在這邊想給這堂課一些建議，我想在利用 Openroad 去合成，做優化這件事情想必讓大家都嚼盡腦汁，但是因為 Openroad 不確定性實在太高，常常我們在優化的時候都是一步一步調整，而在大家辛苦優化到一個自己滿意的階段後，到助教那邊卻要做替代 DM 或是調整 IM 之類的操作，這些做法都會讓大家的 Performance、Area 受到影響，而且受影響的幅度可大可小，充滿不確定性。
當然，大家都有受到影響，這樣很公平，但我認為這是一個在不公平的標準下所做的公平，我提出這個看法並不是要批評助教，因為我知道助教在這堂課上面其實已經花費了比我們修課的同學好幾百份的心力，我也非常感激助教們總是非常樂意的幫助我們去解答問題，只是我認為這個問題是我們許多修課同學都深有體會的，所以在這最後的心得中提出，我認為這個問題的解決方法有很多，最直觀的就是助教直接告訴我們同學們他的 DM 會怎麼改或...之類的，讓我們各位同學們可以套用這個標準，下去做優化。

以上，就是我對這堂課的小小建議，雖然不一定有用，但希望可以給你們參考參考，最後也特別感謝助教，在繁忙的學業之中還要抽空幫助我們解決許多問題，非常感謝有你們的幫助，讓我在修這門課的期間收穫很多~