

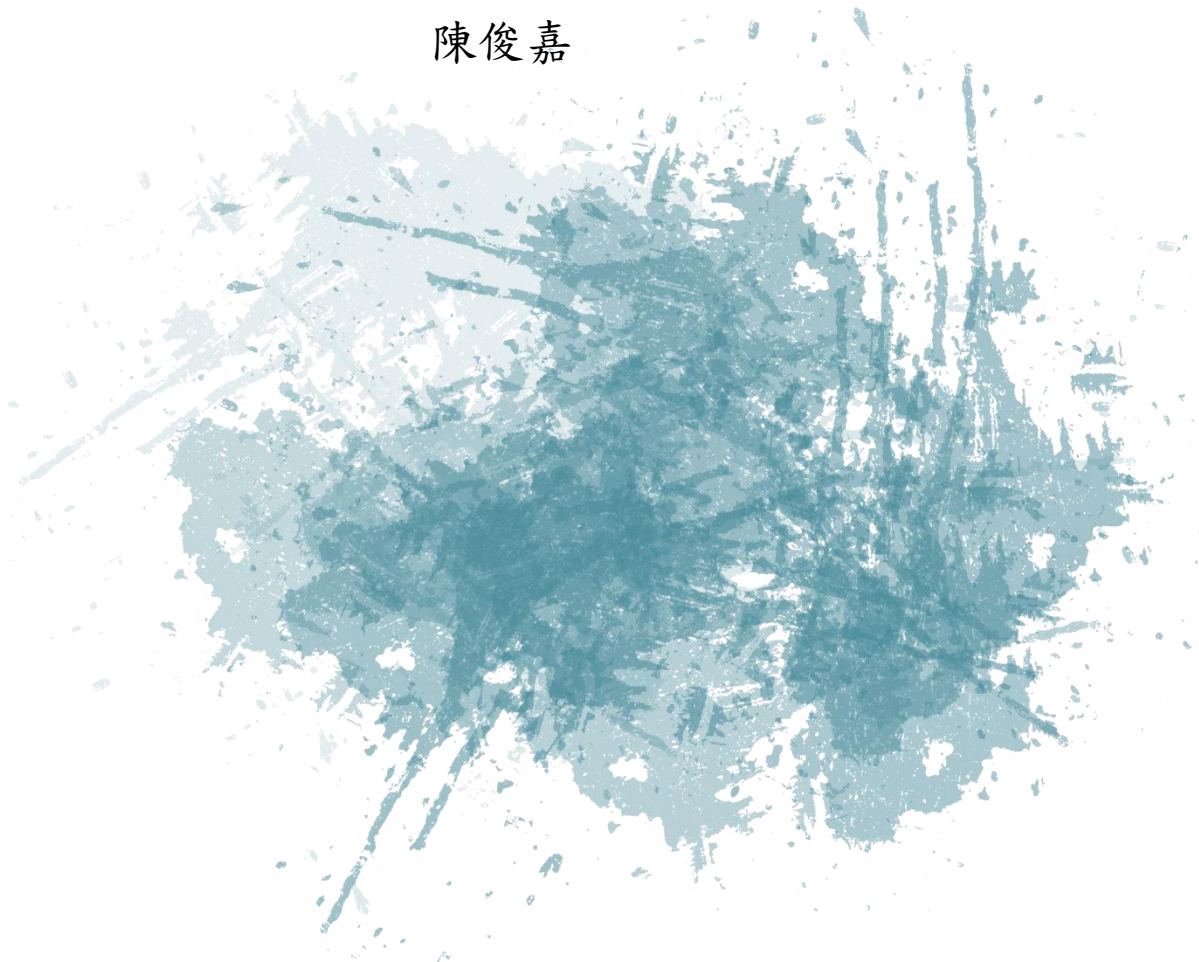
# Computer Organization Project

## 3

四電機三甲

B10830009

陳俊嘉



## Part III :Implement a 5-stage pipelined processor with forwarding and hazard detection.

- Adder Module

```
≡ Adder.v > {} Adder
1  module Adder (data1, data2, data_out) ;
2      output [31:0]data_out;
3      input [31:0]data1, data2;
4
5      assign data_out = data1 + data2;
6
7  endmodule
```

Adder Module 在這次PA中一樣是做計算下一個program counter的位置，data1會放目前的address，data2會放4，以達成每次位置加四的功能。

- Instruction Memory Module

```
29 `define INSTR_MEM_SIZE 128 // Bytes
30
31 /*
32  * Declaration of Instruction Memory for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module IM(
36     // Outputs
37     output wire [31:0]instr,
38     // Inputs
39     input [31:0]instr_addr
40 );
41
42 /*
43  * Declaration of instruction memory.
44  * CAUTION: DONT MODIFY THE NAME AND SIZE.
45  */
46 reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
47 assign instr = {InstrMem[instr_addr], InstrMem[instr_addr + 1], InstrMem[instr_addr + 2], InstrMem[instr_addr + 3]};
48 endmodule
```

Instruction Memory內儲存要執行的指令，以一個byte為單位，一次取四個byte (32bits)，為Big Endian，第一個指到的byte會放在MSB。

- IF/ID Module

```
IF_ID.v > {} IF_ID
1  module IF_ID(
2      //Output
3      output [31:0]instr_out,
4      //Inputs
5      input [31:0]instr_in,
6      input IF_ID_write,
7      input clk
8  );
9
10 reg [31:0]instr_reg;
11
12
13 always @(negedge clk) begin
14     if(IF_ID_write)begin
15         instr_reg <= instr_in;
16     end
17 end
18
19 assign instr_out = instr_reg;
20
21 endmodule
```

此Module主要作為儲存instruction fetch出來的指令的暫存器，為negative edge trigger，並透過IF\_ID\_write來處理要stall時的情況。

- RF Module

```
29 `define REG_MEM_SIZE 32 // Words
30
31 /*
32  * Declaration of Register File for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module RF(
36     // Outputs
37     output [31:0]src_data,
38     output [31:0]tar_data,
39     // Inputs
40     input [4:0]src_addr,
41     input [4:0]tar_addr,
42     input [4:0]dst_addr,
43     input [31:0]dst_data,
44     input clk,
45     input reg_write
46 );
47
48 /*
49  * Declaration of inner register.
50  * CAUTION: DONT MODIFY THE NAME AND SIZE.
51  */
52 reg [31:0]R[0:`REG_MEM_SIZE - 1];
53
54 //assign value by register point by address
55 assign src_data = R[src_addr];
56 assign tar_data = R[tar_addr];
57 always@(posedge clk)begin
58     //write result to register addressed by dst_data
59     if(reg_write)begin
60         R[dst_addr] <= dst_data;
61     end
62 end
63 endmodule
```

與之前的PA一樣，透過從instruction memory讀出來的source address 和target address 去找到相對應的資料並輸出，並透過reg write 訊號控制寫入資料在destination address指到的位置。

- Control Module

```

Control.v > {} Control
1  module Control(
2      //outputs
3      output reg [1:0]ALU_OP,
4      output reg reg_write,
5      output reg reg_dst,
6      output reg ALU_src,
7      output reg mem_write,
8      output reg mem_read,
9      output reg mem2reg,
10     output reg branch,
11     output reg jump,
12     //inputs
13     input [5:0]OP
14 );
15 always@(*)begin
16     case(OP)
17         //if is R type -> set ALU_OP = 10
18         6'd0:begin
19             ALU_OP <= 2'b10;
20             reg_write <= 1;
21             reg_dst <= 1;
22             ALU_src <= 0;
23             mem_write <= 0;
24             mem_read <= 0;
25             mem2reg <= 0;
26             branch <= 0;
27             jump <= 0;
28         end
29         //ADD immediate
30         6'b001100:begin
31             ALU_OP <= 2'b00;
32             reg_write <= 1;
33             reg_dst <= 0;
34             ALU_src <= 1;
35             mem_write <= 0;
36             mem_read <= 0;
37             mem2reg <= 0;
38             branch <= 0;
39             jump <= 0;
40         end
41         //SUB immediate
42         6'b001101:begin
43             ALU_OP <= 2'b01;
44             reg_write <= 1;
45             reg_dst <= 0;
46             ALU_src <= 1;
47             mem_write <= 0;
48             mem_read <= 0;
49             mem2reg <= 0;
50             branch <= 0;
51             jump <= 0;
52         end
53         //store
54         6'b010000:begin
55             ALU_OP <= 2'b00;
56             reg_write <= 0;
57             //reg_dst <= 0; Don't care
58             ALU_src <= 1;
59             mem_write <= 1;
60             mem_read <= 0;
61             //mem2reg <= 0; Don't care
62             branch <= 0;
63             jump <= 0;
64         end
65         //load
66         6'b010001:begin
67             ALU_OP <= 2'b00;
68             reg_write <= 1;
69             reg_dst <= 0;
70             ALU_src <= 1;
71             mem_write <= 0;
72             mem_read <= 1;
73             mem2reg <= 1;
74             branch <= 0;
75             jump <= 0;
76         end
77         //beq
78         6'b010011:begin
79             ALU_OP <= 2'b01;
80             reg_write <= 0;
81             //reg_dst <= 0; Don't care
82             ALU_src <= 0;
83             mem_write <= 0;
84             mem_read <= 0;
85             //mem2reg <= 1; Don't care
86             branch <= 1;
87             jump <= 0;
88         end
89         //jump
90         6'b011100:begin
91             ALU_OP <= 2'b01;
92             reg_write <= 0;
93             //reg_dst <= 0; Don't care
94             ALU_src <= 0;
95             mem_write <= 0;
96             mem_read <= 0;
97             //mem2reg <= 1; Don't care
98             branch <= 0;
99             jump <= 1;
100        end
101        default:begin
102            ALU_OP <= 2'b00;
103            reg_write <= 0;
104            reg_dst <= 0;
105            ALU_src <= 0;
106            mem_write <= 0;
107            mem_read <= 0;
108            mem2reg <= 0;
109            branch <= 0;
110            jump <= 0;
111        end
112    endcase
113 end
114 endmodule

```

Control Module 為處理大部分訊號的controller，會透過判斷instruction最前面的6個bits，也就是OP code，來決定要給其他相對應的component的訊號。

- Hazard Detection Module

```
1 module Hazard_Detection (  
2     //Outputs  
3     output reg PC_write,  
4     output reg stall,  
5     output reg IF_ID_write,  
6     //Inputs  
7     input [4:0]EX_tar_addr,  
8     input [4:0]ID_tar_addr,  
9     input [4:0]ID_src_addr,  
10    input EX_mem_read  
11 );  
12 initial begin  
13     stall <= 0;  
14     IF_ID_write <= 1;  
15     PC_write <= 1;  
16 end  
17 always @(*) begin  
18     if(EX_mem_read && ((ID_src_addr == EX_tar_addr) || (EX_tar_addr == ID_tar_addr)))begin  
19         stall <= 1;  
20         IF_ID_write <= 0;  
21         PC_write <= 0;  
22     end  
23     else begin  
24         stall <= 0;  
25         IF_ID_write <= 1;  
26         PC_write <= 1;  
27     end  
28 end  
29 endmodule
```

此module是用來判斷有沒有發生hazard的，並輸出PC write、stall和IF\_ID write訊號來決定要不要stall，而要stall的條件為當前的instruction會用到上個instruction會改變到的register。通常會在執行load指令時發生。

- MUX Module (for stall)

```
1 module Stall_MUX(  
2     //Outputs  
3     output reg [4:0]src_addr_out,  
4     output reg [4:0]I_dst_addr_out,  
5     output reg [1:0]ALU_OP_out,  
6     output reg reg_write_out,  
7     output reg reg_dst_out,  
8     output reg ALU_src_out,  
9     output reg mem_write_out,  
10    output reg mem_read_out,  
11    output reg mem2reg_out,  
12    output reg branch_out,  
13    output reg jump_out,  
14    //Inputs  
15    input [4:0]src_addr_in,  
16    input [4:0]I_dst_addr_in,  
17    input [1:0]ALU_OP_in,  
18    input reg_write_in,  
19    input reg_dst_in,  
20    input ALU_src_in,  
21    input mem_write_in,  
22    input mem_read_in,  
23    input mem2reg_in,  
24    input branch_in,  
25    input jump_in,  
26    input stall  
27 );  
28  
29 always @(*)begin  
30     if(stall)begin  
31         src_addr_out <= 0;  
32         I_dst_addr_out <= 0;  
33         ALU_OP_out <= 0;  
34         reg_write_out <= 0;  
35         reg_dst_out <= 0;  
36         ALU_src_out <= 0;  
37         mem_write_out <= 0;  
38         mem_read_out <= 0;  
39         mem2reg_out <= 0;  
40         branch_out <= 0;  
41         jump_out <= 0;  
42     end  
43     else begin  
44         src_addr_out <= src_addr_in;  
45         I_dst_addr_out <= I_dst_addr_in;  
46         ALU_OP_out <= ALU_OP_in;  
47         reg_write_out <= reg_write_in;  
48         reg_dst_out <= reg_dst_in;  
49         ALU_src_out <= ALU_src_in;  
50         mem_write_out <= mem_write_in;  
51         mem_read_out <= mem_read_in;  
52         mem2reg_out <= mem2reg_in;  
53         branch_out <= branch_in;  
54         jump_out <= jump_in;  
55     end  
56 end  
57  
58  
59 endmodule
```

這個mux是用來控制control訊號的，若發生hazard時，為了要stall，所以要將接下來的訊號都設為零。而因為在接下來的forwarding unit內會要判斷EX stage的src及tar address，所以在stall時，也會將src及tar address設為零。

- Sign Extension Module

若執行的指令為I type的，就需要透過此 module 將 instruction 的後 16bits，也就是 immediate，轉成 32 bits，才能給接下來的 ALU 進行運算。

```
1 module Sign_Extend(  
2     //output  
3     output reg [31:0]extend_out,  
4     //input  
5     input [15:0]immediate  
6 );  
7  
8 always @(*) begin  
9     if(immediate[15] == 0)begin  
10         extend_out <= {16'd0, immediate};  
11     end  
12     else if(immediate[15] == 1)begin  
13         extend_out <= {16'hFFFF, immediate};  
14     end  
15 end  
16  
17  
18 endmodule
```

- ID/EX Module

```
1 module ID_EX(  
2     //Outputs  
3     output wb_out,  
4     output ALU_src_out,  
5     output mem_read_out,  
6     output mem_write_out,  
7     output reg_dst_out,  
8     output mem2reg_out,  
9     output branch_out,  
10    output jump_out,  
11    output [1:0]ALU_OP_out,  
12    output [31:0]src_data_out,  
13    output [31:0]R_tar_data_out,  
14    output [31:0]I_tar_data_out,  
15    output [4:0]shamt_out,  
16    output [4:0]R_dst_addr_out,  
17    output [4:0]I_dst_addr_out,  
18    output [4:0]src_addr_out,  
19    output [5:0]funct_ctrl_out,  
20    //Inputs  
21    input wb_in,  
22    input ALU_src_in,  
23    input mem_read_in,  
24    input mem_write_in,  
25    input reg_dst_in,  
26    input mem2reg_in,  
27    input branch_in,  
28    input jump_in,  
29    input [1:0]ALU_OP_in,  
30    input [31:0]src_data_in,  
31    input [31:0]R_tar_data_in,  
32    input [31:0]I_tar_data_in,  
33    input [4:0]shamt_in,  
34    input [4:0]R_dst_addr_in,  
35    input [4:0]I_dst_addr_in,  
36    input [4:0]src_addr_in,  
37    input [5:0]funct_ctrl_in,  
38    input clk  
39 );
```

```
41 reg wb, ALU_src, mem_read, mem_write, reg_dst, mem2reg;  
42 reg branch, jump;  
43 reg [1:0]ALU_OP;  
44 reg [31:0]src_data, R_tar_data, I_tar_data;  
45 reg [4:0]shamt, R_dst_addr, I_dst_addr, src_addr;  
46 reg [5:0]funct_ctrl;  
47  
48 always @(negedge clk) begin  
49     wb <= wb_in;  
50     ALU_src <= ALU_src_in;  
51     mem_read <= mem_read_in;  
52     mem_write <= mem_write_in;  
53     mem2reg <= mem2reg_in;  
54     reg_dst <= reg_dst_in;  
55     branch <= branch_in;  
56     jump <= jump_in;  
57     ALU_OP <= ALU_OP_in;  
58     src_data <= src_data_in;  
59     R_tar_data <= R_tar_data_in;  
60     I_tar_data <= I_tar_data_in;  
61     src_addr <= src_addr_in;  
62     shamt <= shamt_in;  
63     R_dst_addr <= R_dst_addr_in;  
64     I_dst_addr <= I_dst_addr_in;  
65     funct_ctrl <= funct_ctrl_in;  
66 end  
67  
68 assign wb_out = wb;  
69 assign ALU_src_out = ALU_src;  
70 assign mem_read_out = mem_read;  
71 assign mem_write_out = mem_write;  
72 assign reg_dst_out = reg_dst;  
73 assign mem2reg_out = mem2reg;  
74 assign branch_out = branch;  
75 assign jump_out = jump;  
76 assign ALU_OP_out = ALU_OP;  
77 assign src_data_out = src_data;  
78 assign R_tar_data_out = R_tar_data;  
79 assign I_tar_data_out = I_tar_data;  
80 assign shamt_out = shamt;  
81 assign R_dst_addr_out = R_dst_addr;  
82 assign I_dst_addr_out = I_dst_addr;  
83 assign src_addr_out = src_addr;  
84 assign funct_ctrl_out = funct_ctrl;  
85  
86 endmodule
```

此module為一暫存器，在negative edge時存取在ID階段的輸出訊號，並輸出至EX stage 的component。

- Forwarding Unit Module

```
1 module Forward(  
2     //Outputs  
3     output reg [1:0]forward_src,  
4     output reg [1:0]forward_tar,  
5     //Inputs  
6     input [4:0]MEM_dst_addr,  
7     input [4:0]dst_addr,  
8     input [4:0]EX_src_addr,  
9     input [4:0]I_dst_addr,  
10    input MEM_reg_write,  
11    input reg_write  
12 );  
13 always@(*)begin  
14     /*  
15      * forward_src signal  
16      */  
17     //mem hazard -> forward data from the prior alu result  
18     if(MEM_reg_write && (MEM_dst_addr != 0) && (MEM_dst_addr == EX_src_addr))begin  
19         forward_src <= 2'b10;  
20     end  
21     //EX hazard -> forward data from data memory or earlier alu result  
22     else if(reg_write && (dst_addr != 0) && (MEM_dst_addr != EX_src_addr) && (dst_addr == EX_src_addr))begin  
23         forward_src = 2'b01;  
24     end  
25     else forward_src = 0;  
26     /*  
27      * forward_tar signal  
28      */  
29     //mem hazard  
30     if(MEM_reg_write && (MEM_dst_addr != 0) && (MEM_dst_addr == I_dst_addr))begin  
31         forward_tar <= 2'b10;  
32     end  
33     //EX hazard  
34     else if(reg_write && (dst_addr != 0) && (MEM_dst_addr != I_dst_addr) && (dst_addr == I_dst_addr))begin  
35         forward_tar = 2'b01;  
36     end  
37     else forward_tar = 0;  
38 end  
39 endmodule
```

判斷是否有需要用到forwarding，並輸出forward\_src及forward\_tar，給在ALU前的兩個MUX來選擇輸入ALU的資料。

- MUX Module

```
1 module MUX5(  
2     //Outputs  
3     output reg [4:0]data_out,  
4     //Inputs  
5     input [4:0]data0,  
6     input [4:0]data1,  
7     input select  
8 );  
9 always @(*) begin  
10     if(select)begin  
11         data_out <= data1;  
12     end else begin  
13         data_out <= data0;  
14     end  
15 end  
16  
17 endmodule
```

```
1 module MUX32(  
2     //Outputs  
3     output reg [31:0]data_out,  
4     //Inputs  
5     input [31:0]data0,  
6     input [31:0]data1,  
7     input select  
8 );  
9 always @(*) begin  
10     if(select)begin  
11         data_out <= data1;  
12     end else begin  
13         data_out <= data0;  
14     end  
15 end  
16  
17 endmodule
```

```
1 module MUX32_3(  
2     //Outputs  
3     output reg [31:0]data_out,  
4     //Inputs  
5     input [31:0]data0,  
6     input [31:0]data1,  
7     input [31:0]data2,  
8     input [1:0]select  
9 );  
10 always @(*) begin  
11     case(select)  
12         0: data_out <= data0;  
13         1: data_out <= data1;  
14         2: data_out <= data2;  
15         default:  
16             data_out <= 0;  
17     endcase  
18 end  
19  
20 endmodule
```

最左的為5-bit MUX，用來選擇destination address。中間的是32-bit MUX 用來選擇輸入ALU target值及選擇要寫回Register File的值。右邊的一樣是32-bit MUX，但是有三個in/output，用來判斷forward的data。



- ALU Control Module

```

1 module ALU_Control (
2     output reg [5:0]funcnt,
3     //inputs
4     input [5:0]funcnt_ctrl,
5     input [1:0]ALU_OP
6 );
7     always@(ALU_OP or funcnt_ctrl)begin
8         //R type
9         if(ALU_OP == 2'b10)begin
10             case(funcnt_ctrl)
11                 6'b001001: funcnt <= 6'b001001; //ADDU
12                 6'b001101: funcnt <= 6'b001010; //SUBU
13                 6'b100101: funcnt <= 6'b010010; //OR
14                 6'b000010: funcnt <= 6'b100010; //SRL
15             endcase
16         end
17         //I type that use ALU?for add only
18         else if(ALU_OP == 2'b00)begin
19             funcnt <= 6'b001001;
20         end
21         //I type that use ALU for subtract only
22         else if(ALU_OP == 2'b01)begin
23             funcnt <= 6'b001010;
24         end
25
26         else begin
27             funcnt <= 0;
28         end
29     end
30 endmodule

```

與之前的PA相同，主要透過 instruction 給的 function control 訊號及 controller 判斷過後輸出的 ALU\_OP 來判斷 ALU 需要做甚麼運算，並輸出 function 給 ALU 做對應的動作。

- ALU Module

```

1 module ALU (
2     //outputs
3     output reg [31:0]data_out,
4     output reg zero,
5     //inputs
6     input [31:0]src_data,
7     input [31:0]tar_data,
8     input [4:0]shamt,
9     input [5:0]funcnt
10 );
11     always@(*)begin
12         case(funcnt)
13             //ADDU
14             6'b001001: data_out <= src_data + tar_data;
15             //SUBU
16             6'b001010: data_out <= src_data - tar_data;
17             //OR
18             6'b010010: data_out <= src_data | tar_data;
19             //SRL
20             6'b100010: data_out <= src_data >> shamt;
21             default: data_out <= 32'd0;
22         endcase
23         zero <= (data_out == 0)? 1'b1 : 1'b0;
24     end
25
26 endmodule

```

此 module 為 CPU 中主要做運算的 component，透過 ALU controller 給的 function 來執行動作。



- EX/MEM Module

```
1  module EX_MEM(  
2      //outputs  
3      output wb_out,  
4      output mem_read_out,  
5      output mem_write_out,  
6      output mem2reg_out,  
7      output branch_out,  
8      output jump_out,  
9      output [31:0]mem_write_data_out,  
10     output [31:0]ALU_result_out,  
11     output [4:0]dst_addr_out,  
12     //Inputs  
13     input wb_in,  
14     input mem_read_in,  
15     input mem_write_in,  
16     input mem2reg_in,  
17     input branch_in,  
18     input jump_in,  
19     input [31:0]mem_write_data_in,  
20     input [31:0]ALU_result_in,  
21     input [4:0]dst_addr_in,  
22     input clk  
23 );  
24 reg wb, mem_read, mem_write, mem2reg, branch, jump;  
25 reg [31:0]ALU_result, mem_write_data;  
26 reg [4:0]dst_addr;  
27  
28 always @(negedge clk) begin  
29     wb <= wb_in;  
30     mem_read <= mem_read_in;  
31     mem_write <= mem_write_in;  
32     mem2reg <= mem2reg_in;  
33     branch <= branch_in;  
34     jump <= jump_in;  
35     mem_write_data <= mem_write_data_in;  
36     ALU_result <= ALU_result_in;  
37     dst_addr <= dst_addr_in;  
38  
39 end  
40  
41 assign wb_out = wb;  
42 assign mem_read_out = mem_read;  
43 assign mem_write_out = mem_write;  
44 assign mem2reg_out = mem2reg;  
45 assign branch_out = branch;  
46 assign jump_out = jump;  
47 assign mem_write_data_out = mem_write_data;  
48 assign ALU_result_out = ALU_result;  
49 assign dst_addr_out = dst_addr;  
50  
51  
52  
53 endmodule
```

與前面的IF/ID和ID/EX一樣，為儲存EX stage輸出至下一階段的暫存器，為negative edge trigger。

- DM Module

```
29 `define DATA_MEM_SIZE 128 // Bytes
30
31 /*
32  * Declaration of Data Memory for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module DM(
36     // Outputs
37     output reg [31:0] read_data,
38     // Inputs
39     input [31:0] write_data,
40     input [31:0] addr,
41     input mem_read,
42     input mem_write,
43     input clk
44 );
45
46 /*
47  * Declaration of data memory.
48  * CAUTION: DONT MODIFY THE NAME AND SIZE.
49  */
50 reg [7:0] DataMem[0:`DATA_MEM_SIZE - 1];
51
52 always@(posedge clk) begin
53     if(mem_read) begin
54         read_data <= {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]};
55     end
56
57     else if(mem_write) begin
58         {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]} <= write_data;
59     end
60 end
61 endmodule
```

Data memory 為除了Instruction memory及Register File以外的memory，可以透過load和store指令來存取其中的資料。

- MEM/WB Module

為五個stage中最後要寫回資料到Register File的暫存器。

```
1 module MEM_WB(
2     //outputs
3     output wb_out,
4     output mem2reg_out,
5     output [31:0] mem_read_data_out,
6     output [31:0] ALU_result_out,
7     output [4:0] dst_addr_out,
8     //Inputs
9     input wb_in,
10    input mem2reg_in,
11    input [31:0] mem_read_data_in,
12    input [31:0] ALU_result_in,
13    input [4:0] dst_addr_in,
14    input clk
15 );
16 reg wb, mem2reg;
17 reg [31:0] ALU_result, mem_read_data;
18 reg [4:0] dst_addr;
19 always @(negedge clk) begin
20     wb <= wb_in;
21     mem2reg <= mem2reg_in;
22     mem_read_data <= mem_read_data_in;
23     ALU_result <= ALU_result_in;
24     dst_addr <= dst_addr_in;
25 end
26
27 assign wb_out = wb;
28 assign mem2reg_out = mem2reg;
29 assign mem_read_data_out = mem_read_data;
30 assign ALU_result_out = ALU_result;
31 assign dst_addr_out = dst_addr;
32
33 endmodule
```

- Final CPU Module

```

29 module FinalCPU(
30     // Outputs
31     output wire      PCWrite,
32     output wire [31:0] Addr_Out,
33
34     // Inputs
35     input  wire [31:0] Addr_In,
36     input  wire      clk
37 );
38
39 wire [31:0] IM_OUT, INSTR;
40 wire [31:0] ID_SRC_DATA, SRC_DATA, FORWARD_SRC_DATA;
41 wire [31:0] ID_R_TAR_DATA, R_TAR_DATA, FORWARD_TAR_DATA;
42 wire [31:0] ID_I_TAR_DATA, I_TAR_DATA, TAR_DATA;
43 wire [31:0] ALU_OUT, MEM_ALU_OUT, WB_ALU_OUT;
44 wire [31:0] MEM_READ_DATA, WB_MEM_READ_DATA;
45 wire [31:0] MEM_WRITE_DATA;
46 wire [31:0] DST_DATA;
47 wire [5:0] FUNCT_CTRL, FUNCT;
48 wire [4:0] R_DST_ADDR, I_DST_ADDR, EX_DST_ADDR, MEM_DST_ADDR, DST_ADDR;
49 wire [4:0] EX_SRC_ADDR;
50 wire [4:0] ID_I_DST_ADDR, ID_SRC_ADDR;
51 wire [4:0] SHAMT;
52 wire [1:0] CTRL_ALU_OP, ID_ALU_OP, ALU_OP;
53 wire [1:0] FORWARD_TAR, FORWARD_SRC;
54 wire CTRL_REG_WRITE, ID_REG_WRITE, EX_REG_WRITE, MEM_REG_WRITE, REG_WRITE;
55 wire CTRL_REG_DST, ID_REG_DST, REG_DST;
56 wire CTRL_ALU_SRC, ID_ALU_SRC, ALU_SRC;
57 wire CTRL_MEM_WRITE, ID_MEM_WRITE, EX_MEM_WRITE, MEM_WRITE;
58 wire CTRL_MEM_READ, ID_MEM_READ, EX_MEM_READ, MEM_READ;
59 wire CTRL_MEM2REG, ID_MEM2REG, EX_MEM2REG, MEM_MEM2REG, MEM2REG;
60 wire CTRL_BRANCH, ID_BRANCH, EX_BRANCH, BRANCH;
61 wire CTRL_JUMP, ID_JUMP, EX_JUMP, JUMP;
62 wire ZERO_FLAG;
63 wire STALL, IF_ID_WRITE;
64
65 Adder Address_Adder(
66     //Outputs
67     .data_out(Addr_Out),
68     //Inputs
69     .data1(Addr_In),
70     .data2(32'd4)
71 );
72
73 /*
74  * Declaration of Instruction Memory.
75  * CAUTION: DONT MODIFY THE NAME.
76  */
77
78 IM Instr_Memory(
79     // Outputs
80     .instr(IM_OUT),
81     // Inputs
82     .instr_addr(Addr_In)
83 );
84
85 IF_ID IF_ID_Register(
86     //Outputs
87     .instr_out(INSTR),
88     //Inputs
89     .instr_in(IM_OUT),
90     .IF_ID_write(IF_ID_WRITE),
91     .clk(clk)
92 );
93
94 Hazard_Detection Hazard_Detection_Unit(
95     //Outputs
96     .PC_write(PCWrite),
97     .stall(STALL),
98     .IF_ID_write(IF_ID_WRITE),
99     //Inputs
100     .EX_tar_addr(I_DST_ADDR),
101     .ID_tar_addr(INSTR[20:16]),
102     .ID_src_addr(INSTR[25:21]),
103     .EX_mem_read(EX_MEM_READ)
104 );

```

```

103 Control_Controller(
104     //Outputs
105     .ALU_OP(CTRL_ALU_OP),
106     .reg_write(CTRL_REG_WRITE),
107     .reg_dst(CTRL_REG_DST),
108     .ALU_src(CTRL_ALU_SRC),
109     .mem_write(CTRL_MEM_WRITE),
110     .mem_read(CTRL_MEM_READ),
111     .mem2reg(CTRL_MEM2REG),
112     .branch(CTRL_BRANCH),
113     .jump(CTRL_JUMP),
114     //Inputs
115     .OP(INSTR[31:26])
116 );
117
118 Stall_MUX Stallation_MUX(
119     //Outputs
120     .src_addr_out(ID_SRC_ADDR),
121     .I_dst_addr_out(ID_I_DST_ADDR),
122     .ALU_OP_out(ID_ALU_OP),
123     .reg_write_out(ID_REG_WRITE),
124     .reg_dst_out(ID_REG_DST),
125     .ALU_src_out(ID_ALU_SRC),
126     .mem_write_out(ID_MEM_WRITE),
127     .mem_read_out(ID_MEM_READ),
128     .mem2reg_out(ID_MEM2REG),
129     .branch_out(ID_BRANCH),
130     .jump_out(ID_JUMP),
131     //Inputs
132     .src_addr_in(INSTR[25:21]),
133     .I_dst_addr_in(INSTR[20:16]),
134     .ALU_OP_in(CTRL_ALU_OP),
135     .reg_write_in(CTRL_REG_WRITE),
136     .reg_dst_in(CTRL_REG_DST),
137     .ALU_src_in(CTRL_ALU_SRC),
138     .mem_write_in(CTRL_MEM_WRITE),
139     .mem_read_in(CTRL_MEM_READ),
140     .mem2reg_in(CTRL_MEM2REG),
141     .branch_in(CTRL_BRANCH),
142     .jump_in(CTRL_JUMP),
143     //select signal
144     .stall(STALL)
145 );
146 /*
147  * Declaration of Register File.
148  * CAUTION: DONT MODIFY THE NAME.
149  */
150 RF_Register_File(
151     // Outputs
152     .src_data(ID_SRC_DATA),
153     .tar_data(ID_R_TAR_DATA),
154     // Inputs
155     .src_addr(INSTR[25:21]),
156     .tar_addr(INSTR[20:16]),
157     .dst_addr(DST_ADDR),
158     .dst_data(DST_DATA),
159     .reg_write(REG_WRITE),
160     .clk(clk)
161 );
162
163 Sign_Extend Sign_Extension(
164     //Output
165     .extend_out(ID_I_TAR_DATA),
166     //Input
167     .immediate(INSTR[15:0])
168 );

```

```

170 ID_EX_ID_EX_Register(
171     //Outputs
172     .wb_out(EX_REG_WRITE),
173     .ALU_src_out(ALU_SRC),
174     .mem_read_out(EX_MEM_READ),
175     .mem_write_out(EX_MEM_WRITE),
176     .reg_dst_out(REG_DST),
177     .mem2reg_out(EX_MEM2REG),
178     .branch_out(EX_BRANCH),
179     .jump_out(EX_JUMP),
180     .ALU_OP_out(ALU_OP),
181     .src_data_out(SRC_DATA),
182     .R_tar_data_out(R_TAR_DATA),
183     .I_tar_data_out(I_TAR_DATA),
184     .shamt_out(SHAMT),
185     .R_dst_addr_out(R_DST_ADDR),
186     .I_dst_addr_out(I_DST_ADDR),
187     .src_addr_out(EX_SRC_ADDR),
188     .funct_ctrl_out(FUNCT_CTRL),
189     //Inputs
190     .wb_in(ID_REG_WRITE),
191     .ALU_src_in(ID_ALU_SRC),
192     .mem_read_in(ID_MEM_READ),
193     .mem_write_in(ID_MEM_WRITE),
194     .reg_dst_in(ID_REG_DST),
195     .mem2reg_in(ID_MEM2REG),
196     .branch_in(ID_BRANCH),
197     .jump_in(ID_JUMP),
198     .ALU_OP_in(ID_ALU_OP),
199     .src_data_in(ID_SRC_DATA),
200     .R_tar_data_in(ID_R_TAR_DATA),
201     .I_tar_data_in(ID_I_TAR_DATA),
202     .shamt_in(INSTR[10:6]),
203     .R_dst_addr_in(INSTR[15:11]),
204     .I_dst_addr_in(ID_I_DST_ADDR),
205     .src_addr_in(ID_SRC_ADDR),
206     .funct_ctrl_in(INSTR[5:0]),
207     .clk(clk)
208 );
209
210 Forward_Forwarding_Unit(
211     //Outputs
212     .forward_src(FORWARD_SRC),
213     .forward_tar(FORWARD_TAR),
214     //Inputs
215     .MEM_dst_addr(MEM_DST_ADDR),
216     .dst_addr(DST_ADDR),
217     .EX_src_addr(EX_SRC_ADDR),
218     .I_dst_addr(I_DST_ADDR),
219     .MEM_reg_write(MEM_REG_WRITE),
220     .reg_write(REG_WRITE)
221 );
222
223 MUX32_3 Forward_SRC_MUX(
224     //output
225     .data_out(FORWARD_SRC_DATA),
226     //inputs
227     .data0(SRC_DATA),
228     .data1(DST_DATA),
229     .data2(MEM_ALU_OUT),
230     .select(FORWARD_SRC)
231 );
232
233 MUX32_3 Forward_TAR_MUX(
234     //output
235     .data_out(FORWARD_TAR_DATA),
236     //inputs
237     .data0(R_TAR_DATA),
238     .data1(DST_DATA),
239     .data2(MEM_ALU_OUT),
240     .select(FORWARD_TAR)
241 );

```

```

243     ALU_Control ALU_Controller(
244         //output
245         .func(FUNCT),
246         //inputs
247         .func_ctrl(FUNCT_CTRL),
248         .ALU_OP(ALU_OP)
249     );
250
251     MUX32 ALU_SRC_MUX(
252         //output
253         .data_out(TAR_DATA),
254         //inputs
255         .data0(FORWARD_TAR_DATA),
256         .data1(I_TAR_DATA),
257         .select(ALU_SRC)
258     );
259
260     ALU ALU_Unit(
261         //output
262         .data_out(ALU_OUT),
263         .zero(ZERO_FLAG),
264         //inputs
265         .src_data(FORWARD_SRC_DATA),
266         .tar_data(TAR_DATA),
267         .shamt(SHAMT),
268         .func(FUNCT)
269     );
270
271     MUX5 Reg_Dst_MUX(
272         .data_out(EX_DST_ADDR),
273         //inputs
274         .data0(I_DST_ADDR),
275         .data1(R_DST_ADDR),
276         .select(REG_DST)
277     );
278
279     EX_MEM EX_MEM_Register(
280         //outputs
281         .wb_out(MEM_REG_WRITE),
282         .mem_read_out(MEM_READ),
283         .mem_write_out(MEM_WRITE),
284         .mem2reg_out(MEM_MEM2REG),
285         .branch_out(BRANCH),
286         .jump_out(JUMP),
287         .mem_write_data_out(MEM_WRITE_DATA),
288         .ALU_result_out(MEM_ALU_OUT),
289         .dst_addr_out(MEM_DST_ADDR),
290         //Inputs
291         .wb_in(EX_REG_WRITE),
292         .mem_read_in(EX_MEM_READ),
293         .mem_write_in(EX_MEM_WRITE),
294         .mem2reg_in(EX_MEM2REG),
295         .branch_in(EX_BRANCH),
296         .jump_in(EX_JUMP),
297         .mem_write_data_in(FORWARD_TAR_DATA),
298         .ALU_result_in(ALU_OUT),
299         .dst_addr_in(EX_DST_ADDR),
300         .clk(clk)
301     );

```

```

303     DM Data_Memory(
304         //Output
305         .read_data(MEM_READ_DATA),
306         //Inputs
307         .write_data(MEM_WRITE_DATA),
308         .addr(MEM_ALU_OUT),
309         .mem_read(MEM_READ),
310         .mem_write(MEM_WRITE),
311         .clk(clk)
312     );
313
314     MEM_WB MEM_WB_Register(
315         //Outputs
316         .wb_out(REG_WRITE),
317         .mem2reg_out(MEM2REG),
318         .mem_read_data_out(WB_MEM_READ_DATA),
319         .ALU_result_out(WB_ALU_OUT),
320         .dst_addr_out(DST_ADDR),
321         //Inputs
322         .wb_in(MEM_REG_WRITE),
323         .mem2reg_in(MEM_MEM2REG),
324         .mem_read_data_in(MEM_READ_DATA),
325         .ALU_result_in(MEM_ALU_OUT),
326         .dst_addr_in(MEM_DST_ADDR),
327         .clk(clk)
328     );
329
330     MUX32 Dst_Data_MUX(
331         //output
332         .data_out(DST_DATA),
333         //inputs
334         .data0(WB_ALU_OUT),
335         .data1(WB_MEM_READ_DATA),
336         .select(MEM2REG)
337     );
338
339     endmodule

```

- Final CPU test bench

```

28 // Setting timescale
29 `timescale 10 ns / 1 ns
30
31 // Declarations
32 `define DELAY      1 // # * timescale
33 `define INSTR_SIZE 8 // bit width
34 `define INSTR_MAX  128 // bytes
35 `define INSTR_FILE  "testbench/IM.dat"
36 `define REG_SIZE   32 // bit width
37 `define REG_MAX    32 // words
38 `define REG_FILE    "testbench/RF.dat"
39 `define DATA_SIZE  8 // bit width
40 `define DATA_MAX   128 // bytes
41 `define DATA_FILE  "testbench/DM.dat"
42 `define OUTPUT_REG  "testbench/RF.out"
43 `define OUTPUT_DATA "testbench/DM.out"
44
45 // Declaration
46 `define LOW      1'b0
47 `define HIGH     1'b1
48
49 module tb_FinalCPU;
50
51     // Inputs
52     reg [31:0] AddrIn;
53
54     // Outputs
55     wire PCWrite;
56     wire [31:0] AddrOut;
57
58     // Clock
59     reg clk;
60
61     // Testbench variables
62     reg [`INSTR_SIZE-1 :0] instrMem [0:`INSTR_MAX-1];
63     reg [`REG_SIZE-1 :0] regMem [0:`REG_MAX-1];
64     reg [`DATA_SIZE-1 :0] dataMem [0:`DATA_MAX-1];
65     integer output_reg;
66     integer output_data;
67     integer i;
68
69     // Instantiate the Unit Under Test (UUT)
70     FinalCPU UUT(
71         // Outputs
72         .PCWrite(PCWrite),
73         .Addr_Out(AddrOut),
74         // Inputs
75         .Addr_In(AddrIn),
76         .clk(clk)
77     );

```

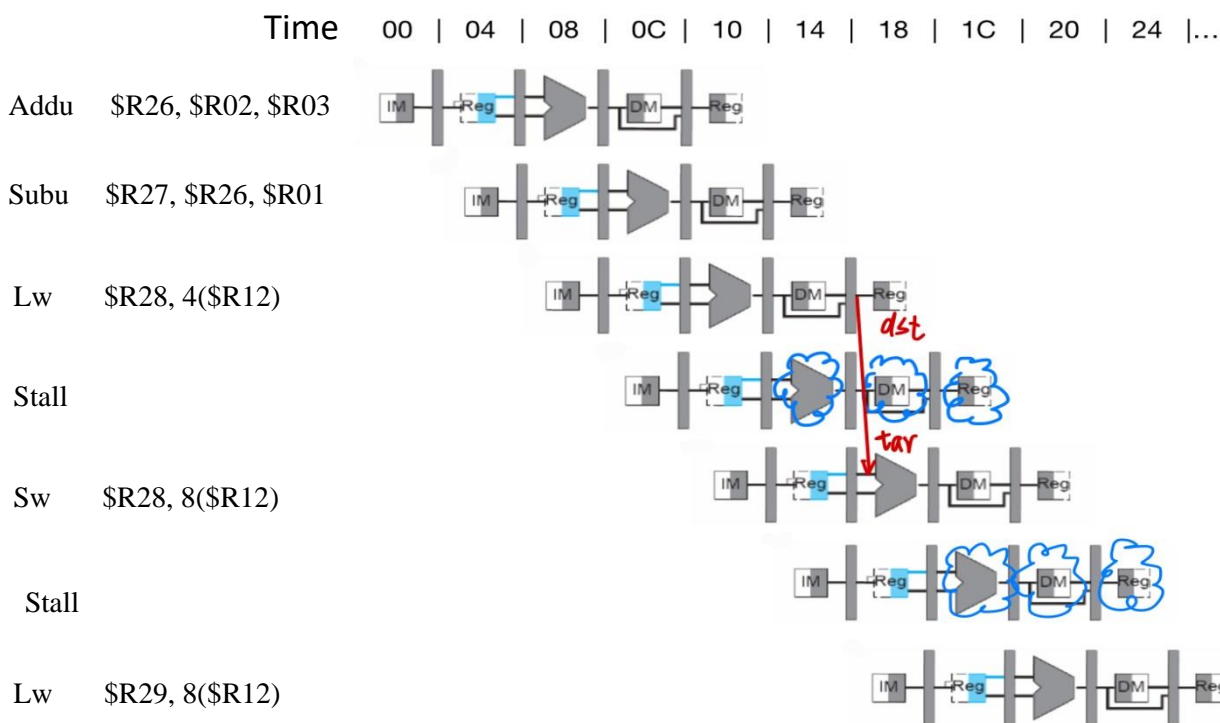
```

79     initial
80     begin : Preprocess
81         // Initialize inputs
82         AddrIn = 32'd0;
83         clk = `LOW;
84
85         // Initialize testbench files
86         $readmemh(`INSTR_FILE, instrMem);
87         $readmemh(`REG_FILE, regMem);
88         $readmemh(`DATA_FILE, dataMem);
89         output_reg = $fopen(`OUTPUT_REG);
90         output_data = $fopen(`OUTPUT_DATA);
91
92         // Initialize instruction memory
93         for (i = 0; i < `INSTR_MAX; i = i + 1)
94         begin
95             UUT.Instr_Memory.InstrMem[i] = instrMem[i];
96         end
97
98         // Initialize register file
99         for (i = 0; i < `REG_MAX; i = i + 1)
100         begin
101             UUT.Register_File.R[i] = regMem[i];
102         end
103
104         // Initialize data memory
105         for (i = 0; i < `DATA_MAX; i = i + 1)
106         begin
107             UUT.Data_Memory.DataMem[i] = dataMem[i];
108         end
109
110         #`DELAY; // Wait for global reset to finish
111     end
112
113     always
114     begin : ClockGenerator
115         #`DELAY;
116         clk <= ~clk;
117     end
118
119     always
120     begin : StimuliProcess
121         // Start testing
122         while (AddrIn < `INSTR_MAX - 4)
123         begin
124             @(negedge clk);
125             AddrIn <= (PCWrite) ? AddrOut : AddrIn;
126             @(posedge clk);
127         end
128
129         // Read out all register value
130         for (i = 0; i < `REG_MAX; i = i + 1)
131         begin
132             regMem[i] = UUT.Register_File.R[i];
133             $fwrite(output_reg, "%x\n", regMem[i]);
134         end
135
136         // Read out all memory value
137         for (i = 0; i < `DATA_MAX; i = i + 1)
138         begin
139             dataMem[i] = UUT.Data_Memory.DataMem[i];
140             $fwrite(output_data, "%x\n", dataMem[i]);
141         end
142
143         // Close output files for safety
144         $fclose(output_reg);
145         $fclose(output_data);
146
147         // Stop the simulation
148         $stop();
149     end
150
151 endmodule

```

- Final CPU Simulation

InstrAddr	Instruction	Binary Code	Hexadecimal
		OPCode   Rs   Rt   immediate	
0x0000_0000	addu \$R26, \$R02, \$R03	0b000000 00010 00011 11010 00000 001011	=> 0x00_43_D0_0B
0x0000_0004	subu \$R27, \$R26, \$R01	0b000000 11010 00001 11011 00000 001101	=> 0x03_41_D8_0D
0x0000_0008	lw \$R28, 4(\$R12)	0b010001 01100 11100 0000000000000100	=> 0x45_9C_00_04
0x0000_000C	sw \$R28, 8(\$R12)	0b010000 01100 11100 0000000000000100	=> 0x41_9C_00_08
0x0000_0010	lw \$R29, 8(\$R12)	0b010001 01100 11101 0000000000000100	=> 0x45_9D_00_08
0x0000_0014	addu \$R30, \$R29, \$R10	0b000000 11101 01010 11110 00000 001011	=> 0x03_AA_F0_0B



/tb_FinalCPU/AddrIn	0000007c	{00...}{00...}{00...}{00000010}{00...}{00000018}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}
/tb_FinalCPU/PCWrite	St1	
/tb_FinalCPU/AddrOut	00000080	{00...}{00...}{00...}{00000014}{00...}{0000001c}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}{00...}
/tb_FinalCPU/clk	1	
/tb_FinalCPU/output...	2	2
/tb_FinalCPU/output...	4	4
/tb_FinalCPU/j	128	128

在Adder\_in = 10的那個cycle，因為store 要用到load到register的值，所以hazard detection unit會讓PC write設為零，讓address不要繼續往下跑，並讓stall訊號為一，把後面的訊號都設為零以達到stall的效果。由上圖可知PC write 訊號正常升降。



## Conclusion and insights

這次的PA在前面的兩個Part其實我個人覺得蠻簡單的，因為大部分都是上次PA做過的，只要在每個stage中間再加上暫存器就好了，也沒有Hazard和forwarding的問題，所以前面其實沒有花太多的時間，經過上次PA的教訓後，在Top Module的接線也透過更易懂的命名還有更加的細心，讓我這次在前兩個Part中免於debug的凌虐。

因為在上課的時候，在講發生hazard及forwarding的條件那邊稍微沒有很專注在聽，所以這次做到part3的時候還花了一點時間去複習那部分才開始做hazard detection 和 forwarding unit這兩個module，但還是遇到了不少問題，更是花了我好幾天才解決問題。

上課的時候聽都覺得沒甚麼問題，但真的做了這次的PA才知道自己不太熟悉的地方，像是在address是0C的那個store要做stall，但我一開始就以為要在0C的那個cycle做stall，但其實因為是在ID的那個階段才能知道是否會有hazard，所以是在下一個cycle，也就是address是10的時候才會將PC write關掉。

在確定自己hazard detection的PC write那些訊號的部分沒有問題後，卻發現還是沒有正常的store值回去，所以我就改了Register file 位置是28(\$R28)的值，發現是forwarding的那兩個MUX還是用原本的訊號，所以才確定是forwarding unit的問題，但在詳細的複習那部分的條件後發現forwarding unit module的程式條件並沒有打錯，所以就把全部的圖畫出來(上頁中間的圖)才發現問題，因為雖然controller後面的MUX在stall的時候把全部的訊號都關掉了，但instruction給的source 和 target address還是會正常傳到EX的stage，並造成forwarding unit的誤判，所以雖然在PA3給的圖中應該沒有要將那兩個address消除，但我還是把那兩個address加進MUX裡面，這樣在stall的時候address就不會被傳進去了，最後問題就解決了。

這次的PA也是讓我學到了不少，雖然也是花了不少的時間，但是也順便複習了之前上課學的東西，也加深了不足的觀念。