



計算機組織 PA3

電機三甲 B11107056 黃敏聖

Area : 18850 Worst slack : 1.74

```
582   Chip area for top module '\FinalCPU': 18850.356000
583       of which used for sequential elements: 325.584000 (1.73%)
12 =====
13 finish report_worst_slack
14 -----
15 worst slack 1.74
```

# 壹、Part1

## 一、程式碼

R\_PipelineCPU :

```
 29 module R_PipelineCPU(
30   // Outputs
31   output [31:0] Output_Addr,
32   // Inputs
33   input  [31:0] Input_Addr,
34   input          clk
35 );
36   //stage 1 to 2
37   wire [31:0] Instruction_a;
38   //stage 2 to 3
39   reg [31:0] Instruction_b;
40   wire [4:0] contral_out_a1;//[4]=Reg_w, [3]=Mem_w, [2]=Mem_r, [1:0]=ALU_op
41   reg [4:0] contral_out_b1;
42   wire [31:0] RsData_a, RtData_a;
43   reg [31:0] RsData_b, RtData_b;
44   reg [10:0] ALU_something;
45   reg [4:0] RdAddr_b1;
46   //stage 3 to 4
47   reg [2:0] contral_out_b2;//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
48   wire [1:0] Funct;
49   reg [4:0] RdAddr_b2;
50   wire [31:0] ALU_out_a1;
51   reg [31:0] ALU_out_b1;
52   //stage 4 to 5
53   reg contral_out_b3;
54   reg [4:0] RdAddr_b3;
55   reg [31:0] ALU_out_b2;
56
57   always @(posedge clk) begin
58     //stage 1 to 2
59     Instruction_b <= Instruction_a;
60     //stage 2 to 3
61     contral_out_b1 <= contral_out_a1;
62     RsData_b <= RsData_a;
63     RtData_b <= RtData_a;
64     ALU_something <= Instruction_b[10:0];
65     RdAddr_b1 <= Instruction_b[15:11];
66     //stage 3 to 4
67     contral_out_b2 <= contral_out_b1[4:2];
68     RdAddr_b2 <= RdAddr_b1;
69     ALU_out_b1 <= ALU_out_a1;
70     //stage 4 to 5
71     contral_out_b3 <= contral_out_b2[2];
72     RdAddr_b3 <= RdAddr_b2;
73     ALU_out_b2 <= ALU_out_b1;
74   end
75
76   /*
77    * Declaration of Instruction Memory.
78    * CAUTION: DONT MODIFY THE NAME.
79    */
80   IM Instr_Memory(
81     // Outputs
82     .Instr(Instruction_a),
83     // Inputs
84     .InstrAddr(Input_Addr)
85   );
86
87   /*
88    * Declaration of Register File.
89    * CAUTION: DONT MODIFY THE NAME.
90    */
91   RF Register_File(
92     // Outputs
93     .RsData(RsData_a),
94     .RtData(RtData_a),
95     // Inputs
96     .RsAddr(Instruction_b[25:21]),
97     .RtAddr(Instruction_b[20:16]),
98     .RdAddr(RdAddr_b3),
99     .RdData(ALU_out_b2),
100    .RegWrite(contral_out_b3),
101    .clk(clk)
102  );
103
104  Adder ad0(
105    // Inputs
106    .Input_1_addr(Input_Addr),
107    .Input_2_addr(32'd4),
108    // Outputs
109    .Output_addr(Output_Addr)
110  );
111
112  ALU_control ac0(
113    // Inputs
114    .Funct_ctrl(ALU_something[5:0]),
115    .ALU_op(contral_out_b1[1:0]),
116    // Outputs
117    .Funct(Funct) //0:+ , 1:- , 2:<< , 3:|
118  );
119
120  ALU a0(
121    // Inputs
122    .Rs_data(RsData_b),
123    .Rt_data(RtData_b),
124    .shamt(ALU_something[10:6]),
125    .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
126    // Outputs
127    .Rd_data(ALU_out_a1),
128    .Zero()
129  );
130
131  control c0(
132    // Inputs
133    .Opcode(Instruction_b[31:26]),
134    // Outputs
135    .Reg_w(contral_out_a1[4]),
136    .Reg_dst(),
137    .ALU_src(),
138    .Mem_w(contral_out_a1[3]),
139    .Mem_r(contral_out_a1[2]),
140    .ALU_op(contral_out_a1[1:0])
141  );
142
143 endmodule
```

基本上就是將各個 module 合在一起，但是在每級之間加上暫存器，用來將複雜的計算切成 5 份同時進行。

Adder :

```
	Adder.v
1  module Adder (
2  	input [31:0] Input_1_adder, Input_2_adder,
3  	output [31:0] Output_adder
4  );
5  	assign Output_adder = Input_1_adder + Input_2_adder;
6 endmodule
```

這個module主要負責處理加法的運算，是用於在執行一串指令後，將指令執行完畢後，PC會自動加 4 來指向下一條指令的位置。

ALU\_control :

```
	ALU_control.v
1  module ALU_control (
2  	input [5:0] Funct_ctrl,
3  	input [1:0] ALU_op,
4  	output reg [1:0] Funct //0:+ , 1:- , 2:<< , 3:|
5  );
6
7  always @(*) begin
8  	case (ALU_op)
9   	2'b10: begin//R type
10    	case (Funct_ctrl)
11      6'b100001: Funct <= 2'd0;//+
12      6'b100011: Funct <= 2'd1;//-
13      6'b000000: Funct <= 2'd2;//<<
14      6'b100101: Funct <= 2'd3;//|
15      default: Funct <= 2'bz;
16     endcase
17  	end
18  	2'b01:begin// +
19     Funct <= 2'd0;
20  	end
21  	2'b11: begin// |
22     Funct <= 2'd3;
23  	end
24  	2'b00: begin// -
25     Funct <= 2'd1;
26  	end
27  	default: Funct <= 2'bz;
28  endcase
29 end
30 endmodule
```

這個module的功能是用來執行暫存器之間的運算。它根據來自ALU\_control的Funct信號來決定該執行哪種運算動作。Funct提供了指令的具體類型（加法、減法、left shift、或），從而指導該單元進行相應的計算。除了R-type指令以外，其他類型的指令（例如I-type）的後6位（即Funct\_ctrl字段）並不是用來直接分辨運算功能的。這是因為在這些指令中，ALU的操作功能並不完全依賴於Funct\_ctrl字段來確定，而是依賴於控制信號中的ALU\_op來進行判斷。當ALU\_op被設置時，控制信號會決定ALU應該執行哪種運算。

ALU :

```
ALU.v
1 module ALU (
2     input [31:0] Rs_data, Rt_data,
3     input [4:0] shamt,
4     input [1:0] Funct, //0:+ , 1:- , 2:<< , 3:|
5     output reg [31:0] Rd_data,
6     output Zero
7 );
8     assign Zero = (Rd_data == 32'd0) ? 1 : 0;
9
10    always @(*) begin
11        case (Funct)
12            2'd0: Rd_data <= Rs_data + Rt_data;
13            2'd1: Rd_data <= Rs_data - Rt_data;
14            2'd2: Rd_data <= Rs_data << shamt;
15            2'd3: Rd_data <= Rs_data | Rt_data;
16            default: Rd_data <= 32'bz;
17        endcase
18    end
19 endmodule
```

這個 module 的功能是用來執行暫存器之間的運算。它根據來自 ALU\_control 的 Funct 信號來決定該執行哪種運算動作。Funct 提供了指令的具體類型（加法、減法、left shift、或）, 從而指導該單元進行相應的計算。

Control :

```
control.v
1 module control (
2     input [5:0] Opcode,
3     output reg Reg_w, Reg_dst,
4     output reg ALU_src, Mem_w, Mem_r,
5     output reg [1:0] ALU_op
6 );
7     always @(*) begin
8         case (Opcode)
9             6'b000000: begin//I type lw
10                 Reg_w <= 1;
11                 Reg_dst <= 1;
12                 ALU_src <= 0;
13                 Mem_w <= 0;
14                 Mem_r <= 0;
15                 ALU_op <= 2'b10;
16             end
17             6'b001001:begin//I type addiu
18                 Reg_w <= 1;
19                 Reg_dst <= 0;
20                 ALU_src <= 1;
21                 Mem_w <= 0;
22                 Mem_r <= 0;
23                 ALU_op <= 2'b01;
24             end
25             6'b101011:begin//I type sw
26                 Reg_w <= 0;
27                 //Reg_dst <= 0;//可以不管
28                 ALU_src <= 1;
29                 Mem_w <= 0;
30                 Mem_r <= 0;
31                 ALU_op <= 2'b01;
32             end
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
```

```
6'b100011:begin//I type lw
    Reg_w <= 1;
    Reg_dst <= 0;
    ALU_src <= 1;
    Mem_w <= 0;
    Mem_r <= 1;
    ALU_op <= 2'b01;
end
6'b001101:begin//I type ori
    Reg_w <= 1;
    Reg_dst <= 0;
    ALU_src <= 1;
    Mem_w <= 0;
    Mem_r <= 0;
    ALU_op <= 2'b11;
end
default: begin//反正不要write就好
    Mem_w <= 0;
    Mem_r <= 0;
    Reg_w <= 0;
end
endcase
end
endmodule
```

這個 module 基本上就是負責控制各個多工選擇器 (mux), 其功能與 PA2 中實現的相同。所有 7 個 bit 的輸出訊號都會透過暫存器傳遞到下一個處理階段，確保數據的穩定與同步傳輸。

IM :

```
IM.v
30
31 /*
32  * Declaration of Instruction Memory for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module IM(
36     // Outputs
37     output reg[31:0] Instr,
38     // Inputs
39     input wire[31:0] InstrAddr
40 );
41 /*
42  * Declaration of inner register.
43  * CAUTION: DONT MODIFY THE NAME AND SIZE.
44  */
45 reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
46
47 always @(*) begin
48     Instr[31:24] <= InstrMem[InstrAddr];
49     Instr[23:16] <= InstrMem[InstrAddr + 32'd1];
50     Instr[15:8]  <= InstrMem[InstrAddr + 32'd2];
51     Instr[7:0]   <= InstrMem[InstrAddr + 32'd3];
52 end
53 endmodule
```

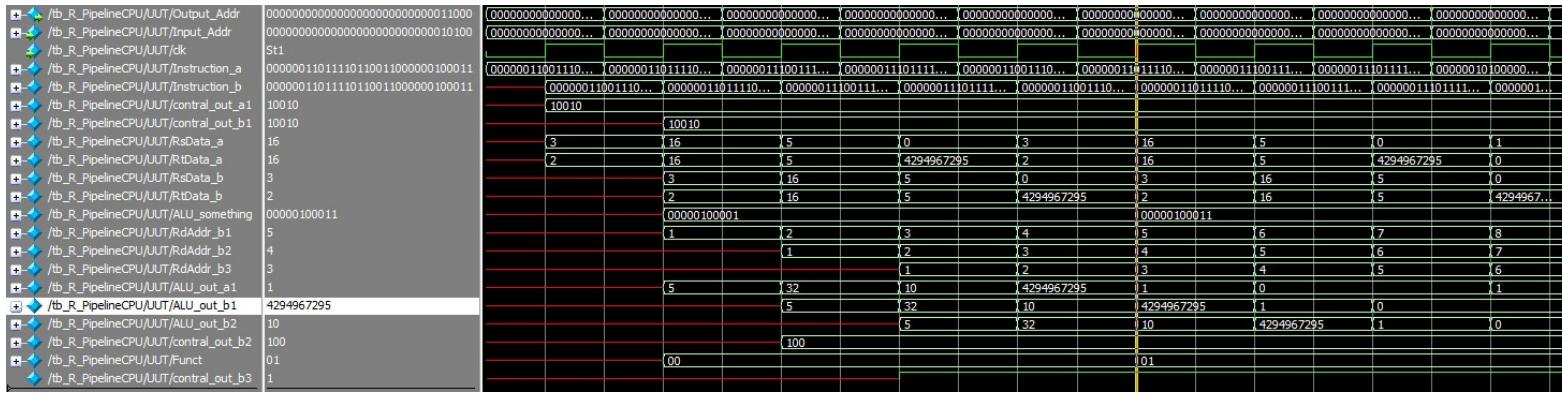
這個module負責將指令從記憶體中讀取出來。在此過程中，使用的是MIPS的Big Endian排法，也就是將記憶體的最低位元存儲資料的最高位元。

RF :

```
RF.v
25 /*
26  * Macro of size declaration of register memory
27  * CAUTION: DONT MODIFY THE NAME AND VALUE.
28  */
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 reg      [31:0] RsData,
38     output reg      [31:0] RtData,
39     // Inputs
40     input  wire     [4:0]  RsAddr,
41     input  wire     [4:0]  RtAddr,
42     input  wire     [4:0]  RdAddr,
43     input  wire     [31:0] RdData,
44     input  wire     RegWrite,
45     input  wire     clk );
46 /*
47  * Declaration of inner register.
48  * CAUTION: DONT MODIFY THE NAME AND SIZE.
49  */
50 reg [31:0]R[0:`REG_MEM_SIZE - 1];
51
52 always @(*) begin
53     RsData <= R[RsAddr];
54     RtData <= R[RtAddr];
55 end
56
57 always @(negedge clk) begin
58     if (RegWrite) R[RdAddr] <= RdData;
59 end
60 endmodule
```

這個module的功能是負責將Rs和Rt的值從暫存器中讀取出來，並將計算結果寫入Rd暫存器。這個module會根據暫存器地址進行查找。另一方面，當計算完成後，這個module還會將結果寫入Rd暫存器中，但這個寫入操作需要等待時鐘(clk)信號的觸發，並且必須滿足RegWrite信號為1的條件，這樣才會觸發對Rd暫存器的寫入。

## 二、波形圖



波形與 PA2 最大不同之處在於部分信號接腳被分成了暫存器前與暫存器後兩段，這樣的設計能將原本較複雜的功能拆分成五個階段，實現流水線 (pipeline) 效果。舉例來說，波形中的 RsData\_a 和 RsData\_b 信號就被延後了一個時鐘週期 (cycle)，這種分段處理讓整體運算更有效率，也提升了 CPU 的執行速度與吞吐量。

### 三、結果

```

000000 11001 11010 00001 00000 100001
add $1, $25, $26

opcode rs rt rd shamt funct
000000 11011 11011 00010 00000 100001
add $2, $27, $27

opcode rs rt rd shamt funct
000000 11100 11100 00011 00000 100001
add $3, $28, $28

opcode rs rt rd shamt funct
000000 11101 11110 00100 00000 100001
add $4, $29, $30

opcode rs rt rd shamt funct
000000 11001 11010 00101 00000 100011
sub $5, $25, $26

opcode rs rt rd shamt funct
000000 11011 11011 00110 00000 100011
sub $6, $27, $27

opcode rs rt rd shamt funct
000000 11100 11100 00111 00000 100011
sub $7, $28, $28

opcode rs rt rd shamt funct
000000 11101 11110 01000 00000 100011
sub $8, $29, $30

```

```

opcode rs rt rd shamt funct
000000 10100 00000 01010 10101 000000
sll $10, $20, $0

opcode rs rt rd shamt funct
000000 10110 00000 01011 10110 000000
sll $11, $22, $0

opcode rs rt rd shamt funct
000000 10111 00000 01100 11000 000000
sll $12, $23, $0

opcode rs rt rd shamt funct
000000 11000 00000 01101 11000 000000
sll $13, $24, $0

opcode rs rt rd shamt funct
000000 10100 10101 01110 00000 100101
or $14, $20, $21

opcode rs rt rd shamt funct
000000 10110 10110 01111 00000 100101
or $15, $22, $22

opcode rs rt rd shamt funct
000000 10111 11000 10000 00000 100101
or $16, $23, $24

opcode rs rt rd shamt funct
000000 11000 11000 10001 00000 100101
or $17, $24, $24

```

```

testbench > RF.dat
19 0000_003 / R[1]
20 0000_0064 // R[18]
21 0000_0030 // R[19]
22 0000_0001 // R[20]
23 0000_0010 // R[21]
24 0000_0002 // R[22]
25 0000_0003 // R[23]
26 0000_0004 // R[24]
27 0000_0003 // R[25]
28 0000_0002 // R[26]
29 0000_0010 // R[27]
30 0000_0005 // R[28]
31 0000_0000 // R[29]
32 FFFF_FFFF // R[30]
33 FFFF_FFFF // R[31]

```

```

testbench > RF.out
1 00000000
2 00000005
3 00000020
4 0000000a
5 ffffffff
6 00000001
7 00000000
8 00000000
9 00000001
10 0000ffff
11 00200000
12 00800000
13 03000000
14 04000000
15 00000011
16 00000002
17 00000007
18 00000004
19 00000064

```

	funct	RsAddr	RsData	RtAddr	RtDara	shamt	RdAddr	RdData
1	Add	25	0000_0003	26	0000_0002	0	1	0000_0005
2	Add	27	0000_0010	27	0000_0010	0	2	0000_0020
3	Add	28	0000_0005	28	0000_0005	0	3	0000_000a
4	Add	29	0000_0000	30	ffff_ffff	0	4	ffff_ffff
5	Sub	25	0000_0003	26	0000_0002	0	5	0000_0001
6	Sub	27	0000_0010	27	0000_0010	0	6	0000_0000
7	Sub	28	0000_0005	28	0000_0005	0	7	0000_0000
8	Sub	29	0000_0000	30	ffff_ffff	0	8	0000_0001
9	Sll	20	0000_0001	0	0000_0000	21	10	0020_0000
10	Sll	22	0000_0002	0	0000_0000	22	11	0080_0000
11	Sll	23	0000_0003	0	0000_0000	24	12	0300_0000
12	Sll	24	0000_0004	0	0000_0000	24	13	0400_0000
13	Or	20	0000_0001	21	0000_0010	0	14	0000_0011
14	Or	22	0000_0002	22	0000_0002	0	15	0000_0002
15	Or	23	0000_0003	24	0000_0004	0	16	0000_0007
16	Or	24	0000_0004	24	0000_0004	0	17	0000_0004

這個結果基本符合我的預期。第 8 個指令是將 0000\_0000 減去 ffff\_ffff，由於借位的關係，會補入一個進位 1，最終結果為 0000\_0001，並將其存入暫存器 8。至於 sll 指令（邏輯左移），例如第 12 條指令會將 0000\_0004 左移 24 位元，最後的結果 0400\_0000 被存入暫存器 13。整體來說，這些運算功能與 PA2 中的設計是一致的，確保了正確的資料處理與指令執行。

# 貳、Part2

## 一、程式碼

I\_PipelineCPU :

```
I_PipelineCPU.v
25  /*
26
29  module I_PipelineCPU(
30      // Outputs
31      output [31:0] Output_Addr,
32      // Inputs
33      input  [31:0] Input_Addr,
34      input          clk
35  );
36      //stage 1 to 2
37      wire [31:0] Instruction_a;
38      //stage 2 to 3
39      reg [31:0] Instruction_b;
40      wire [6:0] contral_out_a1;//[6]=ALU_src, [5]=Reg_dst, [4]=Reg_w, [3]=Mem_w, [2]=Mem_r, [1:0]=ALU_op
41      reg [6:0] contral_out_b1;
42      wire [31:0] RsData_a, RtData_a1;
43      reg [31:0] RsData_b, RtData_b1;
44      reg [15:0] ALU_something;
45      reg [4:0] RdAddr_b1;
46      reg [4:0] RtAddr_b;
47      //stage 3 to 4
48      reg [2:0] contral_out_b2;//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
49      wire [1:0] Funct;
50      reg [4:0] RdAddr_b2;
51      wire [31:0] ALU_out_a1;
52      reg [31:0] ALU_out_b;
53      reg [31:0] RtData_b2;
54      //stage 4 to 5
55      reg [1:0] contral_out_b3;//[1]=Reg_w, [0]=Mem_r
56      reg [4:0] RdAddr_b3;
57      reg [31:0] ALU_out_b2;
58      wire [31:0] mem_read_data_a;
59      reg [31:0] mem_read_data_b;
60
61      //mux
62      wire [31:0] ALU_src_out;
63      wire [4:0] Reg_dst_out;
64      wire [31:0] RdData_final;
```

```
67  always @ (posedge clk) begin
68      //stage 1 to 2
69      Instruction_b <= Instruction_a;
70      //stage 2 to 3
71      contral_out_b1 <= contral_out_a1;
72      RsData_b <= RsData_a;
73      RtData_b1 <= RtData_a1;
74      ALU_something <= Instruction_b[15:0];
75      RdAddr_b1 <= Instruction_b[15:11];
76      RtAddr_b <= Instruction_b[20:16];
77      //stage 3 to 4
78      contral_out_b2 <= contral_out_b1[4:2];
79      RdAddr_b2 <= Reg_dst_out;
80      ALU_out_b <= ALU_out_a1;
81      RtData_b2 <= RtData_b1;
82      //stage 4 to 5
83      contral_out_b3 <= {contral_out_b2[2], contral_out_b2[0]};
84      RdAddr_b3 <= RdAddr_b2;
85      ALU_out_b2 <= ALU_out_b;
86      mem_read_data_b <= mem_read_data_a;
87
88  end
89
90  assign ALU_src_out = contral_out_b1[6] ? {16'd0, ALU_something} : RtData_b1;
91  assign Reg_dst_out = contral_out_b1[5] ? RdAddr_b1 : RtAddr_b;
92  assign RdData_final = contral_out_b3[0] ? mem_read_data_b : ALU_out_b2;
```

```
93  /*
94   * Declaration of Instruction Memory.
95   * CAUTION: DONT MODIFY THE NAME.
96   */
97  DM Data_Memory(
98      //Outputs
99      .MemReadData(mem_read_data_a),
100     //Inputs
101     .MemAddr(ALU_out_b),
102     .MemWriteData(RtData_b2),
103     .MemWrite(contral_out_b2[1]),
104     .clk(clk)
105 );
106
107 /*
108  * Declaration of Instruction Memory.
109  * CAUTION: DONT MODIFY THE NAME.
110  */
111 IM Instr_Memory(
112     // Outputs
113     .Instr(Instruction_a),
114     // Inputs
115     .InstrAddr(Input_Addr)
116 );
```

```

118  /*
119   * Declaration of Register File.
120   * CAUTION: DONT MODIFY THE NAME.
121   */
122   RF Register_File(
123     // Outputs
124     .RsData(RsData_a),
125     .RtData(RtData_a1),
126     // Inputs
127     .RsAddr(Instruction_b[25:21]),
128     .RtAddr(Instruction_b[20:16]),
129     .RdAddr(RdAddr_b3),
130     .RdData(RdData_final),
131     .RegWrite(contral_out_b3[1]),
132     .clk(clk)
133 );
134
135   Adder ad0(
136     // Inputs
137     .Input_1_adder(Input_Addr),
138     .Input_2_adder(32'd4),
139     // Outputs
140     .Output_adder(Output_Addr)
141 );
142
143   ALU_control ac0(
144     // Inputs
145     .Funct_ctrl(ALU_something[5:0]),
146     .ALU_op(contral_out_b1[1:0]),
147     // Outputs
148     .Funct(Funct) //0:+ , 1:- , 2:<< , 3:|
149 );

```

```

151   ALU a0(
152     // Inputs
153     .Rs_data(RsData_b),
154     .Rt_data(ALU_src_out),
155     .shamt(ALU_something[10:6]),
156     .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
157     // Outputs
158     .Rd_data(ALU_out_a1)
159 );
160
161   control c0(
162     // Inputs
163     .Opcode(Instruction_b[31:26]),
164     // Outputs
165     .Reg_w(contral_out_a1[4]),
166     .Reg_dst(contral_out_a1[5]),
167     .ALU_src(contral_out_a1[6]),
168     .Mem_w(contral_out_a1[3]),
169     .Mem_r(contral_out_a1[2]),
170     .ALU_op(contral_out_a1[1:0])
171 );
172 endmodule

```

基本上就是將各個 module 合在一起，但是在每級之間加上暫存器，用來將複雜的計算切成 5 份同時進行。

## Adder :

```

Adder.v
1 module Adder (
2   input [31:0] Input_1_adder, Input_2_adder,
3   output [31:0] Output_adder
4 );
5   assign Output_adder = Input_1_adder + Input_2_adder;
6 endmodule

```

功能同 Part1。

## ALU\_contral :

```

ALU_contral.v
1 module ALU_control (
2   input [5:0] Funct_ctrl,
3   input [1:0] ALU_op,
4   output reg [1:0] Funct //0:+ , 1:- , 2:<< , 3:|
5 );
6
7   always @(*) begin
8     case (ALU_op)
9       2'b10: begin//R type
10         case (Funct_ctrl)
11           6'b100001: Funct <= 2'd0;//+
12           6'b100011: Funct <= 2'd1;//-
13           6'b000000: Funct <= 2'd2;//<<
14           6'b100101: Funct <= 2'd3;//|
15           default: Funct <= 2'bz;
16         endcase
17     end

```

```

18   2'b01:begin// +
19     |   Funct <= 2'd0;
20   end
21   2'b11: begin// |
22     |   Funct <= 2'd3;
23   end
24   2'b00: begin// -
25     |   Funct <= 2'd1;
26   end
27   default: Funct <= 2'bz;
28 endcase
29
30 end
31 endmodule

```

功能同 Part1。

## ALU :

```
ALU.v
1 module ALU (
2     input [31:0] Rs_data, Rt_data,
3     input [4:0] shamt,
4     input [1:0] Funct, //0:+ , 1:- , 2:<< , 3:|
5     output reg [31:0] Rd_data
6 );
7     always @(*) begin
8         case (Funct)
9             2'd0: Rd_data <= Rs_data + Rt_data;
10            2'd1: Rd_data <= Rs_data - Rt_data;
11            2'd2: Rd_data <= Rs_data << shamt;
12            2'd3: Rd_data <= Rs_data | Rt_data;
13            default: Rd_data <= 32'b0;
14        endcase
15    end
16 endmodule
```

功能同 Part1。

## Control :

```
control.v
1 module control (
2     input [5:0] Opcode,
3     output reg Reg_w, Reg_dst,
4     output reg ALU_src, Mem_w, Mem_r,
5     output reg [1:0] ALU_op
6 );
7     always @(*) begin
8         case (Opcode)
9             6'b000000: begin//R type
10                 Reg_w <= 1;
11                 Reg_dst <= 1;
12                 ALU_src <= 0;
13                 Mem_w <= 0;
14                 Mem_r <= 0;
15                 ALU_op <= 2'b10;
16             end
17             6'b001001:begin//I type addiu
18                 Reg_w <= 1;
19                 Reg_dst <= 0;
20                 ALU_src <= 1;
21                 Mem_w <= 0;
22                 Mem_r <= 0;
23                 ALU_op <= 2'b01;
24             end
25             6'b101011:begin//I type sw
26                 Reg_w <= 0;
27                 //Reg_dst <= 0;//可以不管
28                 ALU_src <= 1;
29                 Mem_w <= 1;
30                 Mem_r <= 0;
31                 ALU_op <= 2'b01;
32             end
33         endcase
34     end
35 endmodule
```

```
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
```

6'b100011:begin//I type lw  
Reg\_w <= 1;  
Reg\_dst <= 0;  
ALU\_src <= 1;  
Mem\_w <= 0;  
Mem\_r <= 1;  
ALU\_op <= 2'b01;  
end  
6'b001101:begin//I type ori  
Reg\_w <= 1;  
Reg\_dst <= 0;  
ALU\_src <= 1;  
Mem\_w <= 0;  
Mem\_r <= 0;  
ALU\_op <= 2'b11;  
end  
default: begin//反正不要write就好  
Mem\_w <= 0;  
Mem\_r <= 0;  
Reg\_w <= 0;  
end  
endcase  
end  
endmodule

功能同 Part1。

DM :

```
25  /*
26  * CAUTION: DONT MODIFY THE NAME AND VALUE.
27  */
28 `define DATA_MEM_SIZE 32 // Bytes
29
30 /*
31 * Declaration of Data Memory for this project.
32 * CAUTION: DONT MODIFY THE NAME.
33 */
34
35 module DM (
36   // Outputs
37   output wire [31:0] MemReadData,
38   // Inputs
39   input wire [31:0] MemAddr,
40   input wire [31:0] MemWriteData,
41   input wire MemWrite,
42   input wire clk );
43 /*
44 * Declaration of inner register.
45 * CAUTION: DONT MODIFY THE NAME AND SIZE.
46 */
47 reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
48
49 assign MemReadData = {DataMem[MemAddr], DataMem[MemAddr + 1], DataMem[MemAddr + 2], DataMem[MemAddr + 3]};
50
51 always @ (negedge clk) begin
52   if (MemWrite) begin
53     DataMem[MemAddr] <= MemWriteData[31:24];
54     DataMem[MemAddr+32'd1] <= MemWriteData[23:16];
55     DataMem[MemAddr+32'd2] <= MemWriteData[15:8];
56     DataMem[MemAddr+32'd3] <= MemWriteData[7:0];
57   end
58   else;
59 end
60 endmodule
```

這個module的功能是用來將資料寫入或讀出記憶體。資料的寫入和讀出都遵循Big Endian排法，這意味著資料的最高位元組會存放在記憶體的最低位元地址。此外，這個module與RF（寄存器檔案）相似，也需要等到時鐘信號（clk）和MemWrite控制信號的觸發，才能進行寫入操作。

IM :

```
IM.v
25 /*
26 */
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 reg[31:0] Instr,
38   // Inputs
39   input wire[31:0] InstrAddr
40 );
41 /*
42 * Declaration of inner register.
43 * CAUTION: DONT MODIFY THE NAME AND SIZE.
44 */
45 reg [7:0] InstrMem[0:`INSTR_MEM_SIZE - 1];
46
47 always @(*) begin
48   Instr[31:24] <= InstrMem[InstrAddr];
49   Instr[23:16] <= InstrMem[InstrAddr + 32'd1];
50   Instr[15:8] <= InstrMem[InstrAddr + 32'd2];
51   Instr[7:0] <= InstrMem[InstrAddr + 32'd3];
52 end
53 endmodule
```

功能同 Part1。

RF :

```
RF.v
25 /*
26 */
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 reg [31:0] RsData,
38   output reg [31:0] RtData,
39   // Inputs
40   input wire [4:0] RsAddr,
41   input wire [4:0] RtAddr,
42   input wire [4:0] RdAddr,
43   input wire [31:0] RdData,
44   input wire RegWrite,
45   input wire clk );
46 /*
47 * Declaration of inner register.
48 * CAUTION: DONT MODIFY THE NAME AND SIZE.
49 */
50 reg [31:0] R[0:`REG_MEM_SIZE - 1];
51
52 always @(*) begin
53   RsData <= R[RsAddr];
54   RtData <= R[RtAddr];
55 end
56
57 always @ (negedge clk) begin
58   if (RegWrite) R[RdAddr] <= RdData;
59 end
60 endmodule
```

功能同 Part1。

## 二、波形



波形設計幾乎與 Part1 相同，但新增了記憶體操作和立即值處理的功能。這些新增的部分同樣採用將運算流程拆分為五個階段的方式來執行，以實現流水線 (pipeline) 效果，提高整體執行效率與運算速度。

### 三、結果

```

opcode rs rd immidiate      opcode rs rd immidiate
001001 11010 00001 0001000100010001 001101 11011 01001 0111111111111111
addi $1, $26, 4369          ori $9, $27, 32767

opcode rs rd immidiate      opcode rs rd immidiate
001001 11101 00010 0111111111111111 101011 11001 10100 0000000000000000
addi $2, $29, 32767         sw $20, $25, 0

opcode rs rd immidiate      opcode rs rd immidiate
001001 11110 00011 0000000000000000 101011 11001 10101 0000000000000110
addi $3, $30, 0              sw $21, $25, 6

opcode rs rd immidiate      opcode rs rd immidiate
001001 11011 00100 0111111111111111 101011 11001 10100 0000000000001000
addi $4, $27, 32767         sw $20, $25, 8

opcode rs rd immidiate      opcode rs rd immidiate
001101 11000 00101 0111011101110111 100011 11001 01011 0000000000000000
ori $5, $24, 30583          lw $11, $25, 0

opcode rs rd immidiate      opcode rs rd immidiate
001101 11010 00110 0001000100010001 100011 11001 01100 0000000000000110
ori $6, $26, 4369          lw $12, $25, 6

opcode rs rd immidiate      opcode rs rd immidiate
001101 11101 00111 0111111111111111 100011 11001 01101 0000000000001000
ori $7, $29, 32767         lw $13, $25, 8

opcode rs rd immidiate
001101 11110 01000 0000000000000000
ori $8, $30, 0

```

testbench > RF.dat

```

17 0000_0007 // R[15]
18 0000_0002 // R[16]
19 0000_0037 // R[17]
20 0000_0064 // R[18]
21 0000_0030 // R[19]
22 0000_0011 // R[20]
23 1200_0000 // R[21]
24 0000_0000 // R[22]
25 0000_0000 // R[23]
26 0000_0000 // R[24]
27 0000_0000 // R[25]
28 0000_0001 // R[26]
29 0000_0030 // R[27]
30 0000_0000 // R[28]
31 0000_0010 // R[29]
32 FFFF_FFFF // R[30]
33 FFFF_FFFF // R[31]

```

testbench > RF.out

```

1 00000000
2 00001112
3 0000800f
4 ffffffff
5 0000802f
6 00007777
7 00001111
8 00007fff
9 ffffffff
10 00007fff
11 000000a
12 00000011
13 12000000
14 00000011
15 00000003

```

No memory

	Opcode	RsAddr	RsData	imidiate	RtAddr	RtData
1	Addi	26	0000_0001	1111	1	0000_1112
2	Addi	29	0000_0010	7fff	2	0000_800f
3	Addi	30	ffff_ffff	0000	3	ffff_ffff
4	Addi	27	0000_0030	7fff	4	0000_802f
5	Ori	24	0000_0000	7777	5	0000_7777
6	Ori	26	0000_0001	1111	6	0000_1111
7	Ori	29	0000_0010	7fff	7	0000_7fff
8	Ori	30	ffff_ffff	0000	8	ffff_ffff
9	Ori	27	0000_0030	7ffff	9	0000_7fff

testbench > DM.dat	testbench > DM.out
<pre> 1 // Data Memory in Hex 2 FF    // Addr = 0x00 3 FF    // Addr = 0x01 4 FF    // Addr = 0x02 5 FF    // Addr = 0x03 6 FF    // Addr = 0x04 7 FF    // Addr = 0x05 8 FF    // Addr = 0x06 9 FF    // Addr = 0x07 10 FF   // Addr = 0x08 11 FF   // Addr = 0x09 12 FF   // Addr = 0x0A 13 FF   // Addr = 0x0B 14 FF   // Addr = 0x0C 15 FF   // Addr = 0x0D 16 FF   // Addr = 0x0E 17 FF   // Addr = 0x0F </pre>	<pre> 1 00 2 00 3 00 4 11 5 ff 6 ff 7 12 8 00 9 00 10 00 11 00 12 11 13 ff 14 ff 15 ff 16 ff </pre>

With memory

	Opcode	RsAddr	RsData	RtAddr	RtData	imidiate	MemAddr	MemData
10	Sw	25	0000_0000	20	0000_0011	0	0~3	ffff_ffff => 0000_0011
11	Sw	25	0000_0000	21	1200_0000	6	6~9	ffff_ffff => 1200_0000
12	Sw	25	0000_0000	20	0000_0011	8	8~11	ffff_ffff => 0000_0011
13	Lw	25	0000_0000	11	0000_00a0 => 0000_0011	0	0~3	0000_0011
14	Lw	25	0000_0000	12	0000_0002 => 1200_0000	6	6~9	1200_0000
15	Lw	25	0000_0000	13	0000_0001 => 0000_0011	8	8~11	0000_0011

這些結果基本上與我的預期一致。前 9 個指令都是將 Rs 與立即值進行運算，然後將計算結果存入 Rt 暫存器。第 11 和第 12 個指令都涉及將數據寫入記憶體的位置 8 和 9，但由於第 12 個指令是後執行的，因此這兩個記憶體位置最終存放的是第 12 個指令的結果，覆蓋了第 11 個指令所寫入的資料。這樣的執行順序確保了資料的正確性和一致性，符合預期的運作流程。

# 參、Part3

## 一、程式碼

FinalCPU :

```
29 module FinalCPU(
30     // Outputs
31     output      PC_Write,
32     output [31:0] Output_Addr,
33     // Inputs
34     input   [31:0] Input_Addr,
35     input      clk
36 );
37
38     //stage 1 to 2
39     wire [31:0] Instruction_a;
40     wire [31:0] Instruction_b;
41     //stage 2 to 3
42     wire [6:0] contral_out_a1;//[6]=ALU_src, [5]=Reg_dst, [4]=Reg_w, [3]=Mem_w, [2]=Mem_r, [1:0]=ALU_op
43     wire [6:0] contral_out_b1;
44     wire [31:0] RsData_a, RtData_a1;
45     wire [31:0] RsData_b, RtData_b1;
46     wire [15:0] ALU_something;
47     wire [4:0] RdAddr_b1;
48     wire [4:0] RtAddr_b;
49     wire [4:0] RsAddr_b;
50     //stage 3 to 4
51     wire [2:0] contral_out_b2;//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
52     wire [4:0] RdAddr_b2;
53     wire [31:0] ALU_out_a1;
54     wire [31:0] ALU_out_b;
55     wire [31:0] RtData_b2;
56     //stage 4 to 5
57     wire [2:0] contral_out_b3;//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
58     wire [4:0] RdAddr_b3;
59     wire [31:0] ALU_out_b2;
60     wire [31:0] mem_read_data_a;
61     wire [31:0] mem_read_data_b;
62     //mux and temp
63     wire [1:0] Funct;
64     wire [31:0] ALU_src_out;
65     wire [4:0] Reg_dst_out;
66     wire [31:0] RdData_final;
67     wire contral_all_0;
68     wire [6:0] contral_out;
69     wire [1:0] forward_s, forward_t;
70     reg [31:0] RsData_forward, RtData_forward;
```

```
122 assign ALU_src_out = contral_out_b1[6] ? {16'd0 , ALU_something} : RtData_forward;
123 assign Reg_dst_out = contral_out_b1[5] ? RdAddr_b1 : RtAddr_b;
124 assign RdData_final = contral_out_b3[0] ? mem_read_data_b : ALU_out_b2;
125 assign contral_out_a1 = contral_all_0 ? 7'd0 : contral_out;
126 always @(*) begin
127     case (forward_s)
128         2'd0:RsData_forward = RsData_b;
129         2'd1:RsData_forward = ALU_out_b;
130         2'd2:RsData_forward = RdData_final;
131         default:;
132     endcase
133 end
134 always @(*) begin
135     case (forward_t)
136         2'd0:RtData_forward = RtData_b1;
137         2'd1:RtData_forward = ALU_out_b;
138         2'd2:RtData_forward = RdData_final;
139         default:;
140     endcase
141 end
```

```

72      reg_IF_ID r0(
73          .clk(clk),
74          .Instruction_a(Instruction_a),
75          //Outputs
76          .Instruction_b(Instruction_b)
77      );
78      reg_ID_EX r1(
79          //Inputs
80          .clk(clk),
81          .contral_out_a1(contral_out_a1),//[6]=ALU_src, [5]=Reg_dst, [4]=Reg_w, [3]=Mem_w, [2]=Mem_r, [1:0]=ALU_
82          .RsData_a(RsData_a),
83          .RtData_a1(RtData_a1),
84          .Instruction_b(Instruction_b[25:0]),
85          //Outputs
86          .contral_out_b1(contral_out_b1),
87          .RsData_b(RsData_b),
88          .RtData_b1(RtData_b1),
89          .ALU_something(ALU_something),
90          .RdAddr_b1(RdAddr_b1),
91          .RtAddr_b(RtAddr_b),
92          .RsAddr_b(RsAddr_b)
93      );
94      reg_EX_MEM r2(
95          //Inputs
96          .clk(clk),
97          .contral_out_b1(contral_out_b1[4:2]),
98          .Reg_dst_out(Reg_dst_out),
99          .ALU_out_a1(ALU_out_a1),
100         .RtData_forword(RtData_forword),
101         //Outputs
102         .contral_out_b2(contral_out_b2),//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
103         .RdAddr_b2(RdAddr_b2),
104         .ALU_out_b(ALU_out_b),
105         .RtData_b2(RtData_b2)
106     );
107     reg_MEM_WB r3(
108         //Inputs
109         .clk(clk),
110         .contral_out_b2(contral_out_b2),
111         .RdAddr_b2(RdAddr_b2),
112         .ALU_out_b(ALU_out_b),
113         .mem_read_data_a(mem_read_data_a),
114         //Outputs
115         .contral_out_b3(contral_out_b3),//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
116         .RdAddr_b3(RdAddr_b3),
117         .ALU_out_b2(ALU_out_b2),
118         .mem_read_data_b(mem_read_data_b)
119     );
120 
```

```

185     Adder ad0(
186         // Inputs
187         .Input_1_adder(Input_Addr),
188         .Input_2_adder(32'd4),
189         // Outputs
190         .Output_adder(Output_Addr)
191     );
192
193     ALU_control ac0(
194         // Inputs
195         .Funct_ctrl(ALU_something[5:0]),
196         .ALU_op(contral_out_b1[1:0]),
197         // Outputs
198         .Funct(Funct) //0:+ , 1:- , 2:<< , 3:|
199     );
200
201     ALU a0(
202         // Inputs
203         .Rs_data(RsData_forword),
204         .Rt_data(ALU_src_out),
205         .shamt(ALU_something[10:6]),
206         .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
207         // Outputs
208         .Rd_data(ALU_out_a1)
209     );

```

```

211     control c0(
212         // Inputs
213         .Opcode(Instruction_b[31:26]),
214         // Outputs
215         .Reg_w(contral_out[4]),
216         .Reg_dst(contral_out[5]),
217         .ALU_src(contral_out[6]),
218         .Mem_w(contral_out[3]),
219         .Mem_r(contral_out[2]),
220         .ALU_op(contral_out[1:0])
221     );

```

```

223     forwarding_unit f0(
224         // Inputs
225         .RdAddr_3_to_4(RdAddr_b2),
226         .RdAddr_4_to_5(RdAddr_b3),
227         .Reg_w_3_to_4(contral_out_b2[2]),
228         .Reg_w_4_to_5(contral_out_b3[2]),
229         .RsAddr_2_to_3(RsAddr_b),
230         .RtAddr_2_to_3(RtAddr_b),
231         // Outputs
232         .forward_s(forward_s),
233         .forward_t(forward_t)
234         //0 = normal, 1 = RdAddr_3_to_4, 2 = RdAddr_4_to_5
235     );
236
237     hazard_detection_unit hd0(
238         // Inputs
239         .Mem_r_2_to_3(contral_out_b1[2]),
240         .RtAddr_2_to_3(RtAddr_b),
241         .RtAddr_1_to_2(Instruction_b[20:16]),
242         .RsAddr_1_to_2(Instruction_b[25:21]),
243         // Outputs
244         .contral_all_0(contral_all_0),
245         .PC_Write(PC_Write)
246     );
247 endmodule

```

這個 module 與之前的設計相似，主要用於連接所有子模組，構成完整的 CPU 架構。為了確保能夠成功進行合成，將流水線 (pipeline) 相關的部分拆分並封裝到了子模組 (submodule) 中，這樣不僅提高了設計的模組化程度，也方便了維護與調試。此外，為了提升處理效率並避免流水線中的資料衝突，特別加入了 forwarding 和 hazard detection 的模組，用以解決資料前推和危險檢測問題，確保指令執行的正確性與穩定性。

Adder :

```

1 module Adder (
2     input [31:0] Input_1_adder, Input_2_adder,
3     output [31:0] Output_adder
4 );
5     assign Output_adder = Input_1_adder + Input_2_adder;
6 endmodule

```

功能同 Part1 。

ALU\_contral :

```

1 module ALU_control (
2     input [5:0] Funct_ctrl,
3     input [1:8] ALU_op,
4     output reg [1:0] Funct //0:+ , 1:- , 2:<< , 3:|
5 );
6
7     always @(*) begin
8         case (ALU_op)
9             2'b10: begin//R type
10                 case (Funct_ctrl)
11                     6'b100001: Funct <= 2'd0; //+
12                     6'b100011: Funct <= 2'd1; //-
13                     6'b000000: Funct <= 2'd2; //<<
14                     6'b100101: Funct <= 2'd3; //|
15                     default: Funct <= 2'bz;
16                 endcase
17             end

```

```

18
19
20
21
22
23
24
25
26
27
28
29
30
31

```

```

2'b01: begin // +
|   Funct <= 2'd0;
end
2'b11: begin // |
|   Funct <= 2'd3;
end
2'b00: begin // -
|   Funct <= 2'd1;
end
default: Funct <= 2'bz;
endcase
endmodule

```

功能同 Part1 。

ALU :

```
ALU.v
1 module ALU (
2     input [31:0] Rs_data, Rt_data,
3     input [4:0] shamt,
4     input [1:0] Funct, //0:+ , 1:- , 2:<< , 3:|
5     output reg [31:0] Rd_data
6 );
7     always @(*) begin
8         case (Funct)
9             2'd0: Rd_data <= Rs_data + Rt_data;
10            2'd1: Rd_data <= Rs_data - Rt_data;
11            2'd2: Rd_data <= Rs_data << shamt;
12            2'd3: Rd_data <= Rs_data | Rt_data;
13            default: Rd_data <= 32'b0;
14        endcase
15    end
16 endmodule
```

功能同 Part1。

Control :

```
control.v
1 module control (
2     input [5:0] Opcode,
3     output reg Reg_w, Reg_dst,
4     output reg ALU_src, Mem_w, Mem_r,
5     output reg [1:0] ALU_op
6 );
7     always @(*) begin
8         case (Opcode)
9             6'b000000: begin//R type
10                Reg_w <= 1;
11                Reg_dst <= 1;
12                ALU_src <= 0;
13                Mem_w <= 0;
14                Mem_r <= 0;
15                ALU_op <= 2'b10;
16            end
17            6'b001001:begin//I type addiu
18                Reg_w <= 1;
19                Reg_dst <= 0;
20                ALU_src <= 1;
21                Mem_w <= 0;
22                Mem_r <= 0;
23                ALU_op <= 2'b01;
24            end
25            6'b101011:begin//I type sw
26                Reg_w <= 0;
27                //Reg_dst <= 0;//可以不管
28                ALU_src <= 1;
29                Mem_w <= 1;
30                Mem_r <= 0;
31                ALU_op <= 2'b01;
32            end

```

```
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56

6'b100011:begin//I type lw
    Reg_w <= 1;
    Reg_dst <= 0;
    ALU_src <= 1;
    Mem_w <= 0;
    Mem_r <= 1;
    ALU_op <= 2'b01;
end
6'b001101:begin//I type ori
    Reg_w <= 1;
    Reg_dst <= 0;
    ALU_src <= 1;
    Mem_w <= 0;
    Mem_r <= 0;
    ALU_op <= 2'b11;
end
default: begin//反正不要write就好
    Mem_w <= 0;
    Mem_r <= 0;
    Reg_w <= 0;
end
endcase
endmodule
```

功能同 Part1。

DM :

```
25  /*
26   * Macro of size declaration of data memory
27   * CAUTION: DONT MODIFY THE NAME AND VALUE.
28   */
29 `define DATA_MEM_SIZE 32 // 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 wire [31:0] MemReadData,
38   // Inputs
39   input wire [31:0] MemAddr,
40   input wire [31:0] MemWriteData,
41   input wire MemWrite,
42   input wire clk );
43 /*
44  * Declaration of inner register.
45  * CAUTION: DONT MODIFY THE NAME AND SIZE.
46  */
47 reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
48
49 assign MemReadData = {DataMem[MemAddr], DataMem[MemAddr + 1], DataMem[MemAddr + 2], DataMem[MemAddr + 3]};
50
51 always @(negedge clk) begin
52   if (MemWrite) begin
53     DataMem[MemAddr] <= MemWriteData[31:24];
54     DataMem[MemAddr+32'd1] <= MemWriteData[23:16];
55     DataMem[MemAddr+32'd2] <= MemWriteData[15:8];
56     DataMem[MemAddr+32'd3] <= MemWriteData[7:0];
57   end
58   else;
59 end
60 endmodule
```

功能同 Part2。

IM :

```
25 /*
26  * Macro of size declaration of instruction memory
27  * CAUTION: DONT MODIFY THE NAME AND VALUE.
28  */
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 reg[31:0] Instr,
38   // Inputs
39   input wire[31:0] InstrAddr
40 );
41 /*
42  * Declaration of inner register.
43  * CAUTION: DONT MODIFY THE NAME AND SIZE.
44  */
45 reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
46
47 always @(*) begin
48   Instr[31:24] <= InstrMem[InstrAddr];
49   Instr[23:16] <= InstrMem[InstrAddr + 32'd1];
50   Instr[15:8] <= InstrMem[InstrAddr + 32'd2];
51   Instr[7:0] <= InstrMem[InstrAddr + 32'd3];
52 end
53 endmodule
```

功能同 Part1。

RF :

```
25  /*
26   * Macro of size declaration of register memory
27   * CAUTION: DONT MODIFY THE NAME AND VALUE.
28   */
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 reg    [31:0] RsData,
38   output reg    [31:0] RtData,
39   // Inputs
40   input  wire   [4:0]  RsAddr,
41   input  wire   [4:0]  RtAddr,
42   input  wire   [4:0]  RdAddr,
43   input  wire   [31:0] RdData,
44   input  wire    RegWrite,
45   input  wire        clk );
46 /*
47  * Declaration of inner register.
48  * CAUTION: DONT MODIFY THE NAME AND SIZE.
49  */
50 reg [31:0]R[0:`REG_MEM_SIZE - 1];
51
52 always @(*) begin
53   RsData <= R[RsAddr];
54   RtData <= R[RtAddr];
55 end
56
57 always @ (negedge clk) begin
58   if (RegWrite) R[RdAddr] <= RdData;
59 end
60 endmodule
```

功能同 Part1。

## Forwarding unit :

```
forwarding_unit.v
1  module forwarding_unit (
2    input [4:0] RdAddr_3_to_4, RdAddr_4_to_5,
3    input Reg_w_3_to_4, Reg_w_4_to_5,
4    input [4:0] RsAddr_2_to_3, RtAddr_2_to_3,
5    output reg [1:0] forward_s, forward_t
6    //0 = normal, 1 = RdAddr_3_to_4, 2 = RdAddr_4_to_5
7  );
8
9  //forward_s
10 always @(*) begin
11   if ((RdAddr_3_to_4 == RsAddr_2_to_3) && Reg_w_3_to_4)
12     forward_s = 2'd1;
13   else if ((RdAddr_4_to_5 == RsAddr_2_to_3) && Reg_w_4_to_5)
14     forward_s = 2'd2;
15   else
16     forward_s = 2'd0;
17 end
18
19 //forward_t
20 always @(*) begin
21   if ((RdAddr_3_to_4 == RtAddr_2_to_3) && Reg_w_3_to_4)
22     forward_t = 2'd1;
23   else if ((RdAddr_4_to_5 == RtAddr_2_to_3) && Reg_w_4_to_5)
24     forward_t = 2'd2;
25   else
26     forward_t = 2'd0;
27 end
28
29 endmodule
```

這個 module 的功能是用來處理當使用的 Rs 和 Rt 暫存器值尚未寫入時的情況，避免讀取到錯誤的資料。它透過偵測流水線階段四和階段五中 Rd 的位址是否與階段三中的 Rs 或 Rt 位址相同，來判斷是否存在資料相依性問題。如果發現兩者位址相同，表示目前需要使用的資料尚未從暫存器寫回，這時候該 module 會利用多工選擇器（Mux），直接將階段三從暫存器讀出的值替換掉原本預期的值，避免因資料尚未更新而造成的運算錯誤。這樣的設計有效防止了資料衝突，確保流水線運作時指令間的資料傳遞準確無誤，提升整體 CPU 的運算穩定性與效率。

## Hazard detection unit :

```
hazard_detection_unit.v
1 module hazard_detection_unit (
2     input Mem_r_2_to_3,
3     input [4:0] RtAddr_2_to_3,
4     input [4:0] RtAddr_1_to_2, RsAddr_1_to_2,
5     output reg contral_all_0, PC_Write
6 );
7     always @(*) begin
8         if (((RtAddr_2_to_3 == RsAddr_1_to_2) || (RtAddr_2_to_3 == RtAddr_1_to_2)) && Mem_r_2_to_3) begin
9             contral_all_0 <= 1;
10            PC_Write <= 0;
11        end
12        else begin
13            contral_all_0 <= 0;
14            PC_Write <= 1;
15        end
16    end
17 endmodule
```

這個 module 用來偵測當前剛進入流水線的指令是否與上一筆正在執行的 lw (load word) 指令存在資料相依性。如果偵測到下一條指令的 Rs 或 Rt 暫存器位置與上一筆 lw 指令所寫入的暫存器位置相同，表示存在相依性衝突，為了避免資料錯誤，該 module 會將下一筆指令的寫入控制信號全部設為 0，確保內容不被錯誤修改。這種做法有效防止在資料尚未從記憶體載入並寫回之前，後續指令提前讀取或寫入，從而維持流水線的資料一致性和運作正確性。偵測的具體方式是比對流水線階段二的 Rs 和 Rt 位址與 lw 指令目標暫存器的位址是否相同，若相同則觸發阻塞 (stall) 機制。

## Reg IF ID :

```
reg_IF_ID.v
1 module reg_IF_ID (
2     input clk,
3     input [31:0] Instruction_a,
4     output reg [31:0] Instruction_b
5 );
6     always @(posedge clk) begin
7         //stage 1 to 2
8         Instruction_b <= Instruction_a;
9     end
10 endmodule
```

此 module 是用來將資料從第一階段送到第二階段的暫存器。

## Reg ID EX :

```
reg_ID_EX.v
1 module reg_ID_EX (
2     input clk,
3     input [6:0] contral_out_a1,//[6]=ALU_src, [5]=Reg_dst, [4]=Reg_w, [3]=Mem_w, [2]=Mem_r, [1:0]=ALU_op
4     input [31:0] RsData_a, RtData_a1,
5     input [25:0] Instruction_b,
6     output reg [6:0] contral_out_b1,
7     output reg [31:0] RsData_b, RtData_b1,
8     output reg [15:0] ALU_something,
9     output reg [4:0] RdAddr_b1,
10    output reg [4:0] RtAddr_b,
11    output reg [4:0] RsAddr_b
12 );
13     always @(posedge clk) begin
14         //stage 2 to 3
15         contral_out_b1 <= contral_out_a1;
16         RsData_b <= RsData_a;
17         RtData_b1 <= RtData_a1;
18         ALU_something <= Instruction_b[15:0];
19         RdAddr_b1 <= Instruction_b[15:11];
20         RtAddr_b <= Instruction_b[20:16];
21         RsAddr_b <= Instruction_b[25:21];
22     end
23 endmodule
```

此 module 是用來將資料從第二階段送到第三階段的暫存器。

## Reg EX MEM :

```
reg_EX_MEM.v
1 module reg_EX_MEM (
2     input clk,
3     input [2:0] contral_out_b1,//contral_out_b1[4:2]
4     input [4:0] Reg_dst_out,
5     input [31:0] ALU_out_a1,
6     input [31:0] RtData_forword,
7     output reg [2:0] contral_out_b2,//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
8     output reg [4:0] RdAddr_b2,
9     output reg [31:0] ALU_out_b,
10    output reg [31:0] RtData_b2
11 );
12     always @(posedge clk) begin
13         //stage 3 to 4
14         contral_out_b2 <= contral_out_b1;
15         RdAddr_b2 <= Reg_dst_out;
16         ALU_out_b <= ALU_out_a1;
17         RtData_b2 <= RtData_forword;
18     end
19 endmodule
```

此 module 是用來將資料從第三階段送到第四階段的暫存器。

Reg MEM WB :

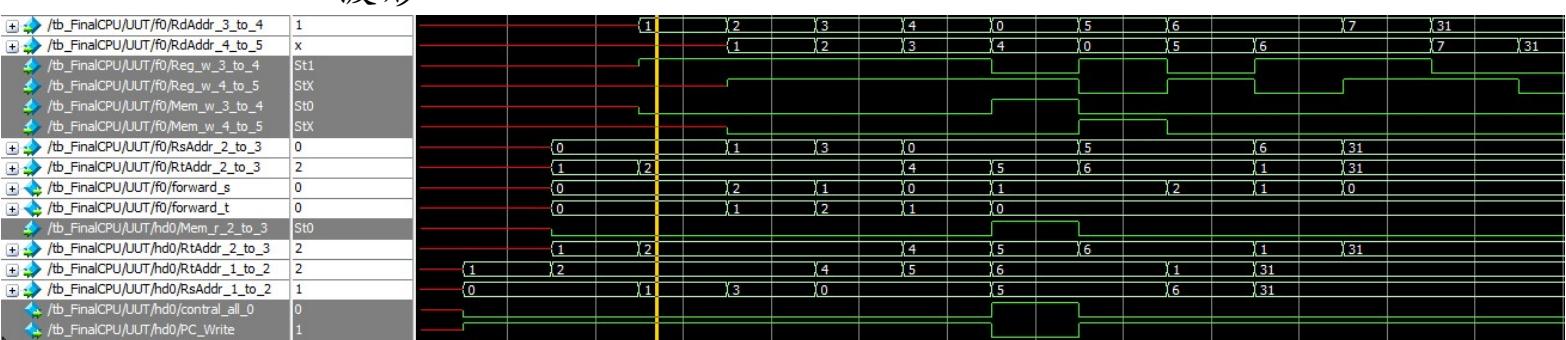
```

reg_MEM_WB.v
1 module reg_MEM_WB (
2     input clk,
3     input [2:0] contral_out_b2,
4     input [4:0] RdAddr_b2,
5     input [31:0] ALU_out_b,
6     input [31:0] mem_read_data_a,
7     output reg [2:0] contral_out_b3,//[2]=Reg_w, [1]=Mem_w, [0]=Mem_r
8     output reg [4:0] RdAddr_b3,
9     output reg [31:0] ALU_out_b2,
10    output reg [31:0] mem_read_data_b
11 );
12    always @(posedge clk) begin
13        //stage 4 to 5
14        contral_out_b3 <= contral_out_b2;
15        RdAddr_b3 <= RdAddr_b2;
16        ALU_out_b2 <= ALU_out_b;
17        mem_read_data_b <= mem_read_data_a;
18    end
19 endmodule

```

此 module 是用來將資料從第四階段送到第五階段的暫存器。

## 二、波形



從波形中可以觀察到，forwarding unit 會根據流水線中不同階段暫存器位址的匹配情況來決定轉發訊號的輸出。具體而言，若在第五階段（Write Back 階段）的 Rd 位址與第三階段（Execute 階段）的 Rs 或 Rt 位址相同，則 forwarding unit 會輸出值為 2，表示資料需從第五階段直接轉發；而如果是第四階段（Memory 階段）的 Rd 與第三階段的 Rs 或 Rt 相同，則輸出值為 1，表示資料由第四階段轉發。因為我的設計中 forward\_s 和 forward\_t 是分開處理的，所以兩者不會產生衝突或干擾，確保了轉發的準確與穩定。

接著，在 hazard detection unit 部分，如果發現第三階段的 MemRead 信號為 1（表示該指令為 lw 類型且正在從記憶體讀取資料），且其寫入暫存器位址與第二階段指令的 Rs 或 Rt 位址相同，系統就會將第二階段指令的所有寫入控制信號設為 0，阻止該指令寫入寄存器，並在下一個時鐘週期重新執行該指令。這種設計有效避免了資料尚未準備好時就被使用，從而解決了資料相依性問題，保證流水線的正確運作與數據一致性。

### 三、結果

```
opcode rs    rd    immediate
001001 00000 00001 0000000000000001
addi $1, $0, 1

opcode rs    rd    immediate
001001 00000 00010 0000000000000010
addi $2, $0, 2

opcode rs    rt    rd    shamt funct
000000 00001 00010 00011 00000 100001
add $3, $1, $2

opcode rs    rt    rd    shamt funct
000000 00011 00010 00100 00000 100011
sub $4, $3, $2

opcode rs    rd    immediate
101011 00000 00100 0000000000000000
sw $4, $0, 0

opcode rs    rd    immediate
100011 00000 00101 0000000000000000
lw $5, $0, 0

opcode rs    rd    immediate
001101 00101 00110 0000001000000000
ori $6, $5, 256

opcode rs    rt    rd    shamt funct
000000 00110 00001 00111 00000 100001
add $7, $6, $1
```

```
testbench > E RF.dat
1 // Register File in Hex
2 0000_0000 // R[0]
3 1234_5678 // R[1]
4 2345_6789 // R[2]
5 3456_789A // R[3]
6 4567_89AB // R[4]
7 5678_9ABC // R[5]
8 6789_ABCD // R[6]
9 0000_1234 // R[7]
10 89AB_CDEF // R[8]
11 9ABC_DEF1 // R[9]

testbench > E RF.out
1 00000000
2 00000001
3 00000002
4 00000003
5 00000001
6 00000001
7 00000101
8 00000102
9 89abcdef
10 9abcdef1
```

```
testbench > E DM.dat
1 // Data Memory in Hex
2 FF // Addr = 0x00
3 FF // Addr = 0x01
4 FF // Addr = 0x02
5 FF // Addr = 0x03
6 FF // Addr = 0x04
7 FF // Addr = 0x05
8 FF // Addr = 0x06
9 FF // Addr = 0x07
```

```
testbench > E DM.out
1 00
2 00
3 00
4 01
5 ff
6 ff
7 ff
8 ff
```

這個結果完全符合我的預期。系統不僅成功實現了前兩個部分（Part1 和 Part2）的功能，還能夠有效地偵測並處理資料冒險（data hazard）的情況。當出現資料相依性時，系統能及時透過轉發（forwarding）和危險偵測（hazard detection）機制進行調度，確保指令執行的正確性與流水線的流暢運作。這種完善的設計避免了因資料衝突造成的錯誤，讓整體架構更加穩定可靠，達到了我對這個系統的預期目標。

## 肆、比較 PA2 和 PA3 的面積和 Slack 和功耗

總體來說，PA3 的面積和功耗相比 PA2 有了顯著增加，這主要是因為增加了每個流水線階段之間的暫存器，以及額外加入了如 forwarding unit 這類模組來確保資料的正確傳遞與處理。這些額外的硬體元件雖然提升了電路的複雜度和面積需求，但同時也帶來了顯著的速度提升。由於將複雜的運算流程拆分成五個階段，每個時鐘週期只需處理其中一部分的工作，這大幅縮短了每個時鐘週期的長度。因此，整體系統可以以更高的頻率運行，提高了 CPU 的吞吐量和運算效率。換句話說，PA3 透過流水線技術在速度上獲得了明顯的優勢，雖然以面積為代價，但對於需要高速運算的應用場景來說，這樣的取捨是非常值得的。

## 伍、如果加入多層的 cache

如果要在流水線 CPU 中加入多層快取 (cache)，我可能會增加更多的流水線階段，讓每個階段專門負責尋找一層快取（例如 L1、L2 等）。同時，會設計並加入一個新的控制器模組，負責檢測每一層快取的命中 (hit) 或未命中 (miss) 狀態，以便正確地控制資料的流動和存取。這樣的設計能有效提升快取查找的效率，並確保資料在不同層級快取間的協調與一致性。

## 陸、心得

總體來說，這次的作業難度與 PA2 相當接近。只要能理解原理並依照課本上的判斷邏輯結合電路圖設計，基本上就能順利完成。就我個人看法來說，如果加入了 jump 和 beq 指令的支援，作業難度可能會顯著增加。這次助教或許是考慮到這部分較為複雜，因此暫時將它們移除，以避免學生遇到過大困難。不過，即使如此，這份作業仍然是一個非常寶貴的學習機會，讓我們能夠深入理解計算機組織的核心概念與實作細節，對未來的學習和實務應用都大有裨益。