

國立臺灣科技大學

電機工程系



計算機組織

作業報告

PA2

四電機三乙 | B10707128 | 劉杰閔

目錄

1.	R-format instruction supported CPU	2
a.	R_formatCPU	2
b.	Instruction Memory	5
c.	Register File	7
d.	Adder	11
e.	ALU.....	12
f.	ALU_Control	13
g.	Control	14
2.	I-format instruction supported CPU	15
a.	I_format CPU	15
b.	Instruction Memory	17
c.	Register File	19
d.	Data Memory	23
e.	Adder	26
f.	ALU.....	27
g.	ALU_Control	28
h.	Control	30
i.	MUX.....	34
3.	J-format (Simple CPU)	36
a.	Simple CPU	36
b.	Instruction Memory	43
c.	Register File	45
d.	Data Memory	49
e.	Adder	52
f.	ALU.....	53
g.	ALU_Control	54
h.	Control	56
i.	MUX.....	59
4.	作業總結與心得.....	61

1. R-format instruction supported CPU

a. R_formatCPU

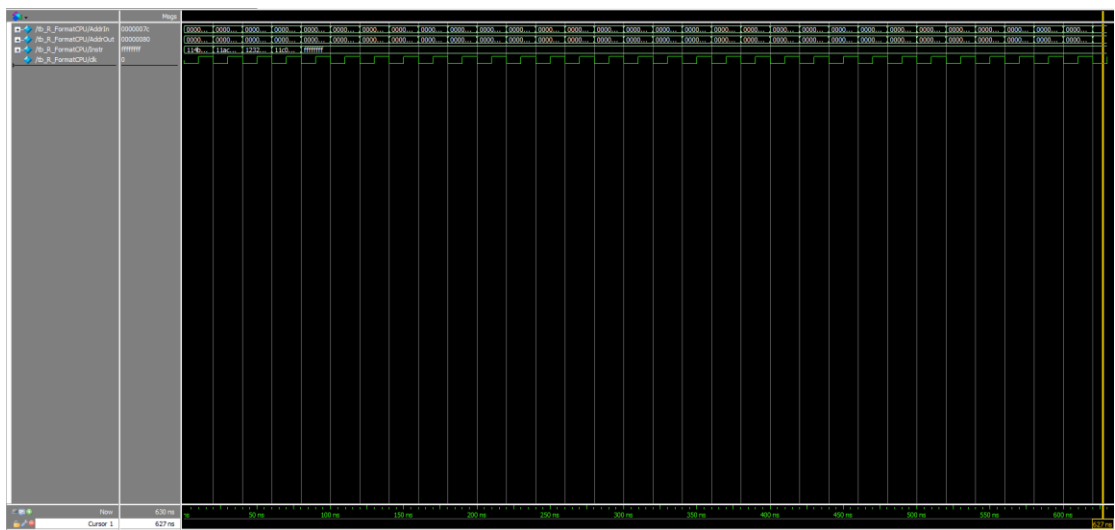
```
29 module R_FormatCPU(  
30     // Outputs  
31     output wire [31:0] AddrOut,  
32     output wire [31:0] Instr,  
33     // Inputs  
34     input wire [31:0] AddrIn,  
35     input wire clk  
36 );  
37 // wire [31:0] Instr;  
38 wire [31:0] ALU_result;  
39 wire [31:0] RsData;  
40 wire [31:0] RtData;  
41 wire [1:0] ALUOp;  
42 wire [5:0] Funct;  
43 wire RegWrite;  
44 /*  
45  * Declaration of Instruction Memory.  
46  * CAUTION: DONT MODIFY THE NAME.  
47  */  
48 IM Instr_Memory(  
49     // Outputs  
50     .Instr(Instr),  
51     // Inputs  
52     .InstrAddr(AddrIn)  
53 );  
54  
55 /*  
56  * Declaration of Register File.  
57  * CAUTION: DONT MODIFY THE NAME.  
58  */  
59 RF Register File(  
60     // Outputs  
61     .RsData(RsData),  
62     .RtData(RtData),  
63     // Inputs  
64     .clk(clk),  
65     .RegWrite(RegWrite),  
66     .RsAddr(Instr[25:21]),  
67     .RtAddr(Instr[20:16]),  
68     .RdAddr(Instr[15:11]),  
69     .RdData(ALU_result[31:0])  
70 );  
71
```

```

72  ✓ Adder adder(
73      // Outputs
74      .AddrOut(AddrOut),
75      // Inputs
76      .AddrIn(AddrIn)
77  );
78
79  ✓ ALU alu(
80      // Inputs
81      .Src1(RsData[31:0]),
82      .Src2(RtData[31:0]),
83      .Shamt(Instr[10:6]),
84      .Funct(Funct[5:0]),
85      // Outputs
86      .result(ALU_result[31:0])
87  );
88
89  ✓ Control controller(
90      // Inputs
91      .OpCode(Instr[31:26]),
92      // Outputs
93      .RegWrite(RegWrite),
94      .ALUOp(ALUOp[1:0])
95  );
96
97  ✓ ALU_Control ALU_controller(
98      // Inputs
99      .funct(Instr[5:0]),
100     .ALUOp(ALUOp[1:0]),
101     // Outputs
102     .Funct(Funct[5:0])
103  );
104
105  endmodule
106

```

tb_R_formatCPU 與題目所提供之檔案相同，因篇幅限制而不另行截圖。這個 R_formatCPU 是將所有模組集大成之結果。我在裡面宣告了一些 wire 讓他去相互連接。而下圖為 TestBench 模擬之結果及 RF.out 之輸出結果。模擬出來的結果我們看最後一個 clk，可以發現 7C+4=80H，因此推測是正確的。而下方右圖是經過執行後輸出出來的結果，而對比於題目提供之檔案(左圖)可以發現完全相同，因此 R_formatCPU 到此順利做完。



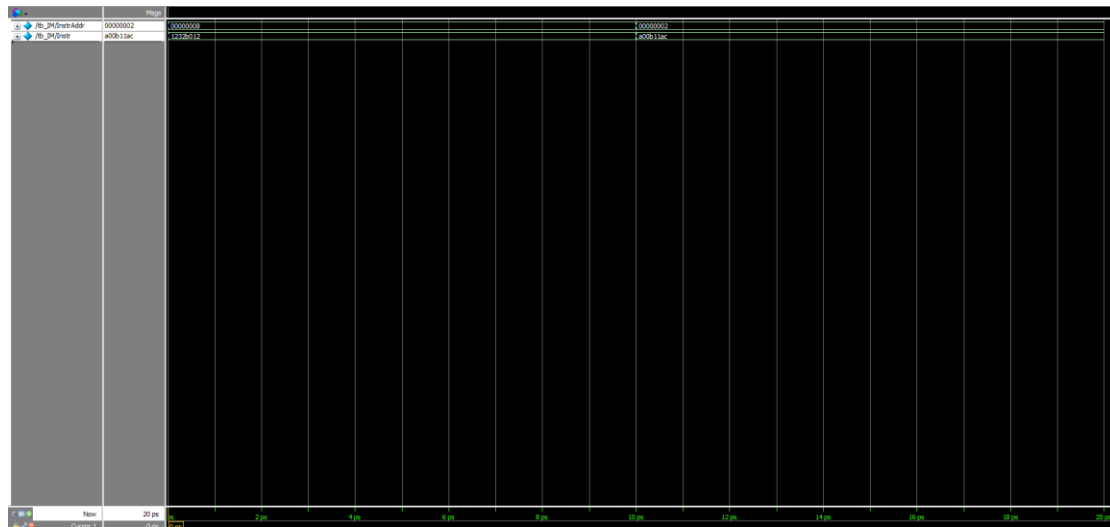
testbench > Answer > RF.out	testbench > RF.out
You, a day ago 1 author (You)	You, 7 hours ago 1 author (You)
1 00000000	1 00000000
2 00000001	2 00000001
3 00000002	3 00000002
4 77777777	4 77777777
5 7f7f7f7f	5 7f7f7f7f
6 f7f7f7f7	6 f7f7f7f7
7 7fffffff	7 7fffffff
8 80000000	8 80000000
9 ffff0000	9 ffff0000
10 0000ffff	10 0000ffff
11 00000011	11 00000011
12 00000023	12 00000023
13 00000017	13 00000017
14 00000090	14 00000090
15 00000100	15 00000100
16 00000250	16 00000250
17 00000300	17 00000300
18 00000037	18 00000037
19 00000064	19 00000064
20 00000030	20 00000030
21 00000034	21 00000034
22 00000079	22 00000079
23 00000024	23 00000024
24 00040000	24 00040000
25 00000000	25 00000000
26 00000000	26 00000000
27 00000000	27 00000000
28 00000000	28 00000000
29 00000000	29 00000000
30 00000000	30 00000000
31 ffffffff	31 ffffffff
32 ffffffff	32 ffffffff
33	33

b. Instruction Memory

IM這個module其實很簡單，只要輸入Instruction Address，接著我就到該位置去抓Instruction，因為這個系統是BIG-ENDIAN，因此我抓的順序就會是如我程式碼的方式去抓{0, 1, 2, 3}。那這個部分比較特別的是testbench，我參考了題目提供的tb_R_formatCPU，使用\$readmemh來抓資料。接著下圖為模擬之結果。

```
IM.v
29 `define INSTR_MEM_SIZE 128 // Bytes
30 `define Rtype_op 6'b000100
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 [31:0] InstrAddr
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]; // You, a day ago * put some component and is fine to design
47
48 always@(InstrAddr)
49 begin
50     Instr[31:0] = {InstrMem[InstrAddr], InstrMem[InstrAddr+1], InstrMem[InstrAddr+2], InstrMem[InstrAddr+3]};
51 end
52
53 endmodule
```

```
tb_IM.v
1 `define INSTR_FILE "testbench/IM.dat"
2 `define INSTR_MAX 128 // bytes
3 `define INSTR_SIZE 8 // bit width
4
5
6 module tb_IM;
7
8     integer i;
9     reg [31:0] InstrAddr;
10    wire [31:0] Instr;
11    reg [`INSTR_SIZE-1 : 0] instrMem [0:`INSTR_MAX-1];
12
13    IM Instruction_Memory(
14        .InstrAddr(InstrAddr),
15        .Instr(Instr)
16    );
17
18    initial begin : Preprocess
19
20        $readmemh(`INSTR_FILE, instrMem); // put the value into instrMem
21        // Initialize intruction memory
22        for (i = 0; i < `INSTR_MAX; i = i + 1)
23        begin
24            Instruction_Memory.InstrMem[i] = instrMem[i];
25        end
26    end
27
28    initial #20 $finish;
29
30    initial fork
31        #0 InstrAddr = 8; // Instr should be 12_32_B0_12.
32        #10 InstrAddr = 2; // Instr should be A0_0B_11_AC
33    join
34
35
36 endmodule
```



1. 第一次Address = 8, Instr should be 12_32_B0_12.
2. 第二次Address = 2, Instr should be A0_0B_11_AC.

下圖為IM.dat，供此部分參考。

```
testbench > IM.dat
You, a day ago | 1 author (You)
1 // Instruction Memory in Hex
2 11 // Addr = 0x00
3 4B // Addr = 0x01
4 A0> // Addr = 0x02
5 0B> // Addr = 0x03
6 11> // Addr = 0x04
7 AC> // Addr = 0x05
8 A8 // Addr = 0x06
9 0D // Addr = 0x07
10 12 // Addr = 0x08
11 32 // Addr = 0x09
12 B0 // Addr = 0x0A
13 12 // Addr = 0x0B
14 11 // Addr = 0x0C
15 C0 // Addr = 0x0D
16 BA // Addr = 0x0E
17 A6 // Addr = 0x0F
18 FF // Addr = 0x10
19 FF // Addr = 0x11
20 FF // Addr = 0x12
21 FF // Addr = 0x13
```

c. Register File

RM這個module其實不難，只要判斷RegWrite是否為1，來決定是否可以將值寫入Reg，那其他部分就是到Register的地址去抓值去輸出。我將RsData與RtData使用assign而非放在always裡面是因為我發現放在裡面會等到正緣觸發才把值送入ALU，那這樣的話就無法達到我們想要的效果了。

```
35 module RF(  
36     // Outputs  
37     output [31:0] RsData,  
38     output [31:0] RtData,  
39     // Inputs  
40     input RegWrite,  
41     input clk,  
42     input [4:0] RsAddr,  
43     input [4:0] RtAddr,  
44     input [4:0] RdAddr,  
45     input [31:0] RdData  
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 RsData = R[RsAddr];  
55 assign RtData = R[RtAddr];  
56  
57 always@(posedge clk)  
58 begin  
59     if(RegWrite == 1)  
60     begin  
61         R[RdAddr] = RdData;  
62     end  
63     else  
64     begin  
65         R[RdAddr] = R[RdAddr];  
66     end  
67 end  
68  
69  
70 endmodule
```



```

1  `define DELAY          5    // # * timescale
2  `define REG_SIZE      32    // bit width
3  `define REG_MAX       32    // words
4  `define REG_FILE      "testbench/RF.dat"
5
6  module tb_RF;
7
8      // Inputs
9      reg RegWrite;
10     reg clk;
11     reg [4:0] RsAddr;
12     reg [4:0] RtAddr;
13     reg [4:0] RdAddr;
14     reg [31:0] RdData;
15
16     // Outputs
17     wire [31:0] RsData;
18     wire [31:0] RtData;
19
20     integer i;
21     integer output_reg;
22     reg [`REG_SIZE-1 :0] regMem    [0:`REG_MAX-1];
23
24     RF Register File(
25         // Outputs
26         .RsData(RsData),
27         .RtData(RtData),
28         // Inputs
29         .RegWrite(RegWrite),
30         .clk(clk),
31         .RsAddr(RsAddr),
32         .RtAddr(RtAddr),
33         .RdAddr(RdAddr),
34         .RdData(RdData)
35     );

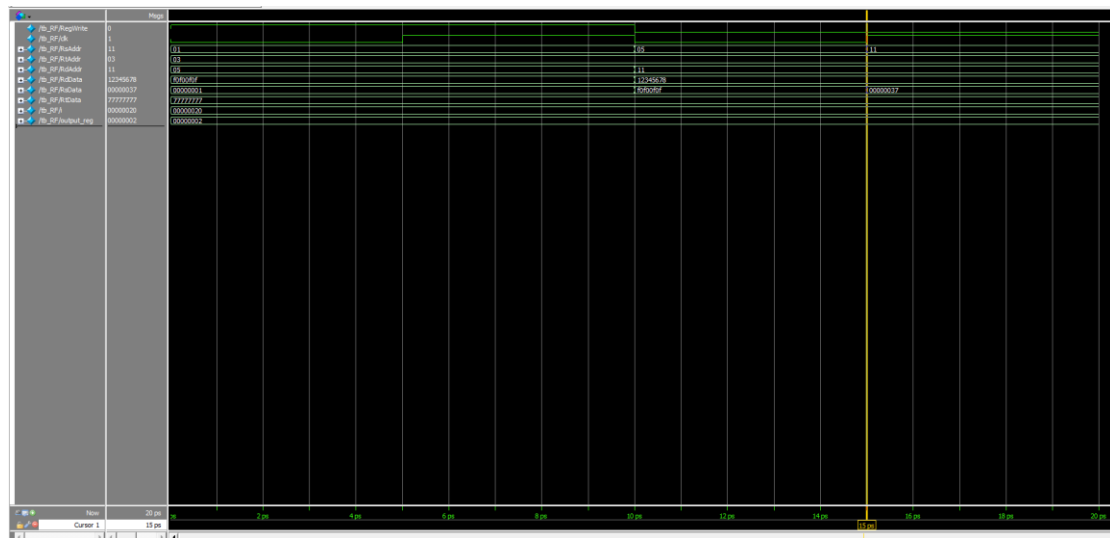
```

```

39     initial begin: Preprocess
40         // Initialize inputs
41         clk = 0;
42         RegWrite = 1;
43         RsAddr = 5'd 1; // Register R[1]
44         RtAddr = 5'd 3; // R[3]
45         RdAddr = 5'd 5; // R[5]
46         RdData = 32'h f0f0_0f0f;
47
48         $readmemh(`REG_FILE,regMem);
49
50         // Initialize register file
51         for (i = 0; i < `REG_MAX; i = i + 1)
52             begin
53                 Register_File.R[i] = regMem[i];
54             end
55     end
56
57     initial #20 $finish;
58
59
60     always begin : ClockGenerator
61         #`DELAY;
62         clk ≤ ~clk;
63     end
64
65
66     initial fork
67         #0 RegWrite = 1;
68         #0 RsAddr = 5'd 1;
69         #0 RtAddr = 5'd 3;
70         #0 RdAddr = 5'd 5;
71         #0 RdData = 32'h F0F0_0F0F;
72
73         #10 RegWrite = 0;
74         #10 RsAddr = 5'd 5; // to show where R[5] is been written or not.
75         #10 RtAddr = 5'd 3;
76         #10 RdAddr = 5'd 17;
77         #10 RdData = 32'h 1234_5678;
78         #15 RsAddr = 5'd 17; // to show where R[17] is been written or not.
79     join
80

```

而下圖這個就是模擬出來的結果，與IM相同的是，我用類似的指令去將值存入一個Reg，接著再去做模擬進行分析。



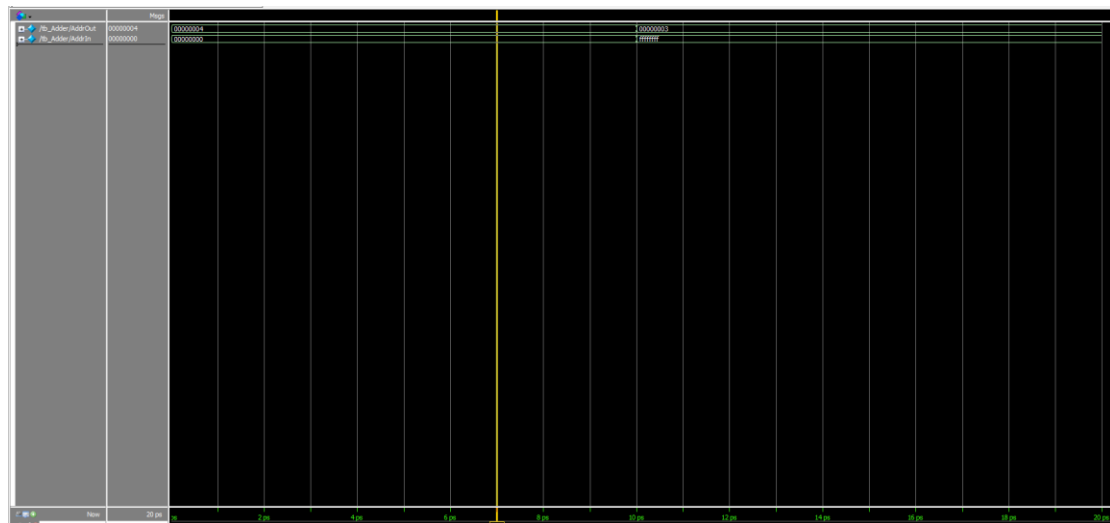
1. 一開始的時候我是設定讓值可以寫入Reg，所以我把R[5]=F0F0_0F0F
2. 在t=10後我去把RsAddr改成5(也就是第一次的Rd)，因此可以看到我第一次是否有成功寫入，而我們也可以看到RsData變成F0F0_0F0F了。那接著我將RegWrite改為0，試著把它放到R[17]。
3. 在t=15後我去把RsAddr改成17(也就是第二次的Rd)，因此可以看到我第二次因為我關掉RegWrite了，所以沒有成功寫入，因此可以確認RF可以正常工作。

d. Adder

```
Adderv x
Adderv You, 2 hours ago | 1 author (You)
1 module Adder
2 (
3     input [31:0] AddrIn,
4     output [31:0] AddrOut
5 );
6
7     assign AddrOut = AddrIn + 32'd 4;
8
9 endmodule
10

tb_Adderv.t.u x
1 module tb_Adder;
2
3     wire [31:0] AddrOut;
4     reg [31:0] AddrIn;
5
6     Adder A(
7         // Outputs
8         .AddrOut(AddrOut),
9         // Inputs
10        .AddrIn(AddrIn)
11    );
12
13    initial #20 $finish;
14    initial fork
15        #0 AddrIn = 32'd 0;
16
17        #10 AddrIn = 32'h FFFF_FFFF;
18    join
19
20 endmodule
```

Adder所做的事情非常簡單，因為在R-type的AdderOut僅僅需要能夠將AddrIn+4，因此我就只做了加4的動作，然後輸出。（我的加法是unsigned的加法，因為Address沒有負的。）



1. 第一次Input為0，輸出為4。
2. 第二次Input為FFFF_FFFF，輸出為3。

e. ALU

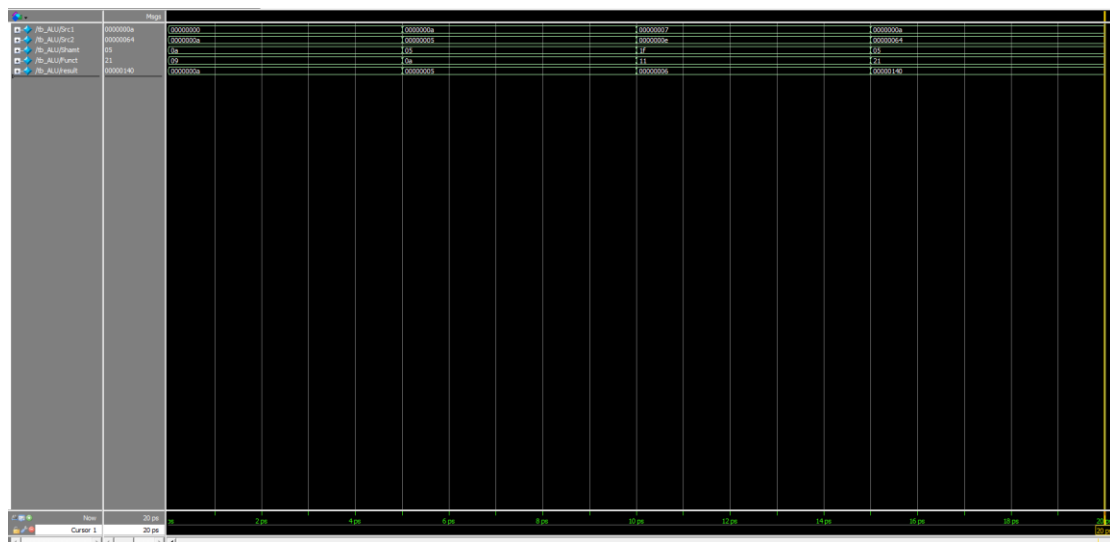
```

// tb_ALU.v
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 module tb_ALU;
7
8     reg [31:0] Src1;
9     reg [31:0] Src2;
10    reg [4:0] Shamt;
11    reg [5:0] Funct;
12    wire [31:0] result;
13
14    ALU AL1(
15        .Src1(Src1),
16        .Src2(Src2),
17        .Shamt(Shamt),
18        .Funct(Funct),
19        .result(result)
20    );
21
22    initial #20 $finish;
23    initial fork
24        #0 Src1 = 32'd 0;
25        #0 Src2 = 32'd 10;
26        #0 Funct = `addu;
27        #0 Shamt = 5'd 10; // result should be 10(hex).
28
29        #5 Src1 = 32'd 10;
30        #5 Src2 = 32'd 5;
31        #5 Funct = `subu;
32        #5 Shamt = 5'd 5; // result should be 5(hex).
33
34        #10 Src1 = 32'd 7; // 0111
35        #10 Src2 = 32'd 14; // 1110
36        #10 Funct = `AND;
37        #10 Shamt = 5'd 31; // result should be 6h (0110).
38
39        #15 Src1 = 32'd 10; // 1010
40        #15 Src2 = 32'd 100; // This value would not affect the result.
41        #15 Funct = `sll; // shift left logic
42        #15 Shamt = 5'd 5; // 10*2^5 = 320.(d) (140H)
43
44    join
45 endmodule
    
```

```

// ALU.v
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 module ALU(
7     input [31:0] Src1,
8     input [31:0] Src2,
9     input [4:0] Shamt,
10    input [5:0] Funct,
11    output reg [31:0] result
12);
13
14 always@(Funct or Shamt or Src1 or Src2)
15 begin
16     case (Funct)
17         `addu : result = Src1 + Src2;
18         `subu : result = Src1 - Src2;
19         `AND : result = Src1 & Src2;
20         `sll : result = Src1 << Shamt;
21     endcase
22 end
23
24 endmodule
    
```

ALU 主要是用來做Src1 and Src2的運算，由輸入的Funct來決定要做甚麼工作。下圖為由testbench產生之模擬結果。



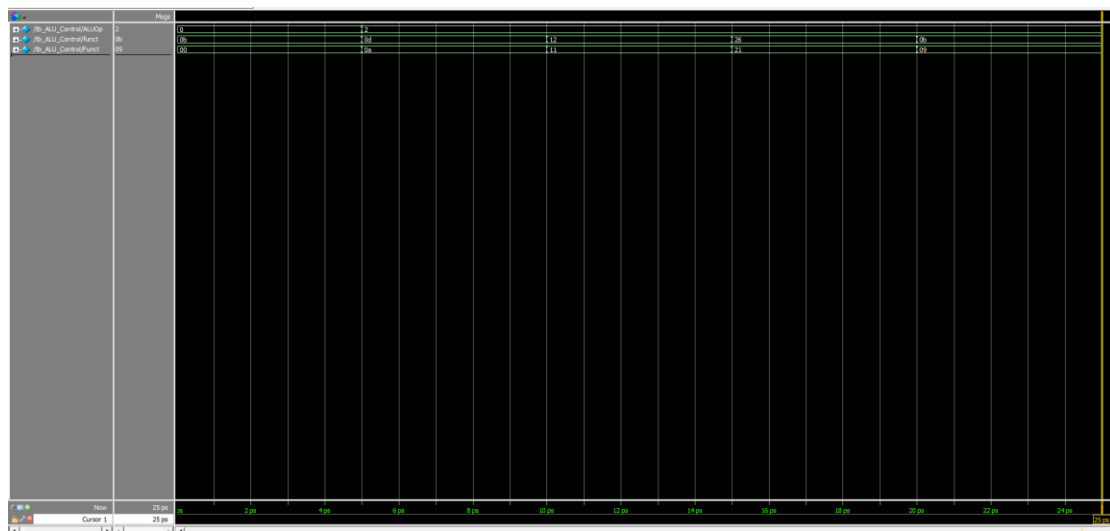
1. 第一次Input為0+A，輸出為A。
2. 第二次Input為A-5，輸出為5。
3. 第三次Input為(1f)&(11)，輸出為6。
4. 第四次Input為A<<5=A*2^5=320，輸出為140。

f. ALU_Control

```
ALU_Control
You, 5 minutes ago | 1 author (You)
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 `define input_addu 6'b 001011
7 `define input_subu 6'b 001101
8 `define input_and 6'b 010010
9 `define input_sll 6'b 100110
10 `define R_type 2'b10
11
12 module ALU_Control(
13     input [5:0] funct,
14     input [1:0] ALUOp,
15     output reg [5:0] Funct
16 );
17
18 always@([funct or ALUOp])
19 begin
20     case(ALUOp)
21     `R_type:
22     begin
23         case(funct)
24         `input_addu : Funct = `addu;
25         `input_subu : Funct = `subu;
26         `input_and : Funct = `AND;
27         `input_sll : Funct = `sll;
28         default: Funct = 0;
29         endcase
30     end
31     default: Funct = 0;
32     endcase
33 end
34 endmodule

tb_ALU_Control
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 `define input_addu 6'b 001011
7 `define input_subu 6'b 001101
8 `define input_and 6'b 010010
9 `define input_sll 6'b 100110
10 `define R_type 2'b10
11
12 module tb_ALU_Control;
13
14 reg [5:0] funct;
15 reg [1:0] ALUOp;
16 wire [5:0] Funct;
17
18 ALU_Control alu_controller(
19     .funct(funct),
20     .ALUOp(ALUOp),
21     .Funct(Funct)
22 );
23
24 initial #25 $finish;
25
26 initial fork
27     #0 ALUOp = 2'b0;
28     #0 funct = `input_addu; // Funct = 0;
29
30     #5 ALUOp = 2'b10;
31     #5 funct = `input_subu; // Funct = a;
32
33     #10 funct = `input_and; // Funct = 11h
34
35     #15 funct = `input_sll; // Funct = 21h
36
37     #20 funct = `input_addu; // Funct = 9h
38
39 join
40
41 endmodule
```

ALU_Control主要是用來控制ALU工作與否，及將instr的指令轉為ALU懂得function code。下圖為TestBench之模擬結果。



1. 第一次ALUOP為0，ALU不可以工作，Funct輸出為0。
2. 第二次開始ALUOP為2' b10，ALU可以工作，Funct輸出為Instr對應的subu。
3. 第三次Input為input_and，輸出為Instr對應的AND。
4. 第四次Input為input_sll，輸出為Instr對應的sll。
5. 第五次Input為input_addu，輸出為Instr對應的addu。

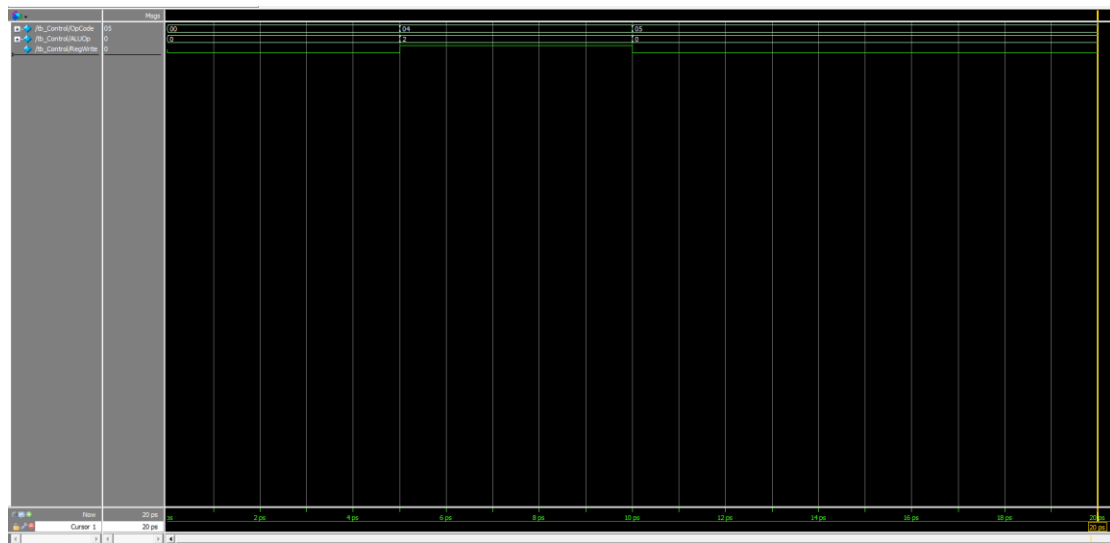
g. Control

Control這個module在R-type沒什麼太複雜的工作，只要依照Opcode的要求，來確認會不會把值寫入Reg裡面，來決定是否要送RegWrite的訊號。而因為R-type的所有指令都會使用到ALU，因此我們ALUOp只要是對的Opcode，我們一律送2'b10。

```
Control.v
You, 6 hours ago | 1 author (You)
1 module Control(
2     input  [5:0]  Opcode,
3     output reg   RegWrite,
4     output reg   [1:0]  ALUOp
5 );
6
7 always@(Opcode)
8 begin
9     case (Opcode)
10        6'd 4:
11        begin
12            RegWrite = 1'b 1; // R format.
13            ALUOp = 2'b10;
14        end
15        default:
16        begin
17            RegWrite = 0; // no this opcode.
18            ALUOp = 2'b00;
19        end
20    endcase
21 end
22
23 endmodule

tb_Control.v
1 module tb_Control;
2
3     reg    [5:0]  Opcode;
4     wire   [5:0]  RegWrite;
5     wire   [1:0]  ALUOp;
6
7     Control_controller(
8         .Opcode(Opcode),
9         .RegWrite(RegWrite),
10        .ALUOp(ALUOp)
11    );
12
13    initial #20 $finish;
14
15    initial fork
16        #0 Opcode = 6'd 0;
17        #5 Opcode = 6'd 4;
18        #10 Opcode = 6'd 5;
19    join
20
21 endmodule
22
23 endmodule
```

下圖為由testbench產生之模擬結果。



1. 第一次Opcode為0，RF不工作->RegWrite=0，ALUOP輸出為0。
2. 第二次Opcode為4，RF工作->RegWrite=1，ALUOP輸出為2'b10。
3. 第三次Opcode為0，RF不工作->RegWrite=0，ALUOP輸出為0。

2. I-format instruction supported CPU

a. I_format CPU

I_formatCPU，因篇幅限制而不特別截圖。因此在這個地方，只放上 DM.out 與 RF.out 來確認模擬結果是正確的。而下方右圖是 DM.out 經過執行後輸出出來的結果，而對比於題目提供之檔案(左圖)可以發現完全相同，除了 27~30 以外，其他結果皆為 FF。下下方右圖 RF.out 是經過執行後輸出出來的結果，而對比於題目提供之檔案(左圖)可以發現完全相同。

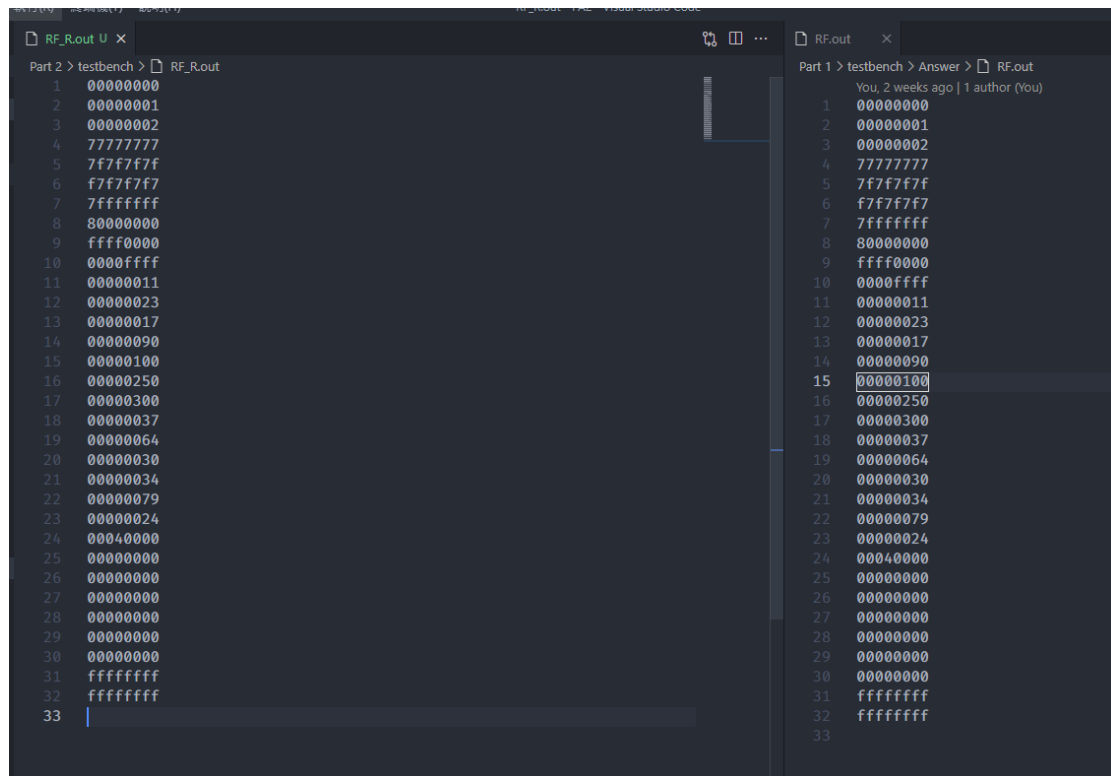
```
Part 2 > testbench > Answer > DM.out
13 FF
14 FF
15 FF
16 FF
17 FF
18 FF
19 FF
20 FF
21 FF
22 FF
23 FF
24 FF
25 FF
26 FF
27 00
28 00
29 00
30 1b
31 FF
32 FF
33 FF
34 FF
35 FF
36 FF
37 FF
38 FF
39 FF
40 FF
41 FF
42 FF
43 FF

Part 2 > testbench > DM.out
13 FF
14 FF
15 FF
16 FF
17 FF
18 FF
19 FF
20 FF
21 FF
22 FF
23 FF
24 FF
25 FF
26 FF
27 00
28 00
29 00
30 1b
31 FF
32 FF
33 FF
34 FF
35 FF
36 FF
37 FF
38 FF
39 FF
40 FF
41 FF
42 FF
43 FF
```

```
Part 2 > testbench > Answer > RF.out
1 00000000 You, 5 days ago | 1 author (You)
2 00000001
3 00000002
4 77777777
5 7f7f7f7f
6 f7f7f7f7
7 7fffffff
8 80000000
9 ffff0000
10 0000ffff
11 00000011
12 00000023
13 00000017
14 00000090
15 00000100
16 00000250
17 00000300
18 00000037
19 00000064
20 00000030
21 00000000
22 00000000
23 00000000
24 00000000
25 0000001b
26 00000008
27 0000001b
28 00000000
29 00000000
30 00000000
31 ffffffff
32 ffffffff

Part 2 > testbench > RF.out
1 00000000
2 00000001
3 00000002
4 77777777
5 7f7f7f7f
6 f7f7f7f7
7 7fffffff
8 80000000
9 ffff0000
10 0000ffff
11 00000011
12 00000023
13 00000017
14 00000090
15 00000100
16 00000250
17 00000300
18 00000037
19 00000064
20 00000030
21 00000000
22 00000000
23 00000000
24 00000000
25 0000001b
26 00000008
27 0000001b
28 00000000
29 00000000
30 00000000
31 ffffffff
32 ffffffff
```


實測R_format在I-type CPU上模擬之結果。可以看到完全相等，因此到這裡可以確認成功。



The screenshot shows a code editor with two side-by-side windows. The left window is titled 'RF_R.out U' and the right window is titled 'RF.out'. Both windows show a list of 33 hexadecimal values, which are identical in both. The values are as follows:

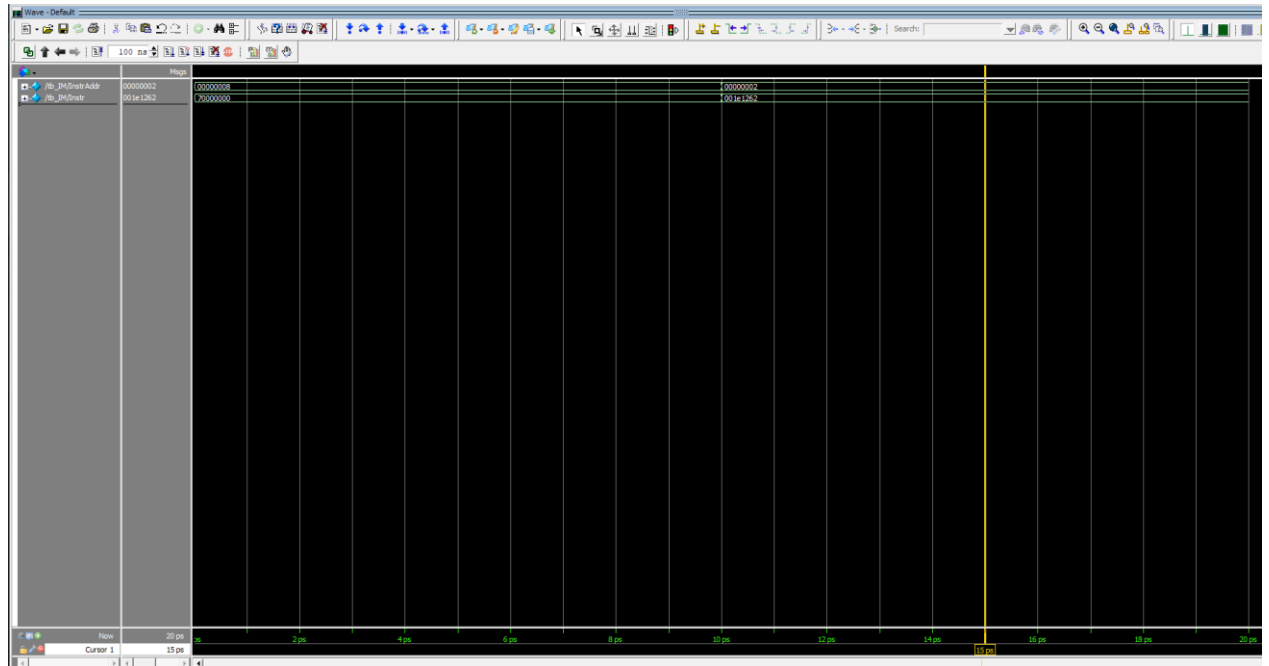
Line	RF_R.out	RF.out
1	00000000	00000000
2	00000001	00000001
3	00000002	00000002
4	77777777	77777777
5	7f7f7f7f	7f7f7f7f
6	f7f7f7f7	f7f7f7f7
7	7fffffff	7fffffff
8	80000000	80000000
9	ffff0000	ffff0000
10	0000ffff	0000ffff
11	00000011	00000011
12	00000023	00000023
13	00000017	00000017
14	00000090	00000090
15	00000100	00000100
16	00000250	00000250
17	00000300	00000300
18	00000037	00000037
19	00000064	00000064
20	00000030	00000030
21	00000034	00000034
22	00000079	00000079
23	00000024	00000024
24	00040000	00040000
25	00000000	00000000
26	00000000	00000000
27	00000000	00000000
28	00000000	00000000
29	00000000	00000000
30	00000000	00000000
31	ffffffff	ffffffff
32	ffffffff	ffffffff
33		

b. Instruction Memory

IM這個module其實很簡單，只要輸入Instruction Address，接著我就到該位置去抓Instruction，因為這個系統是BIG-ENDIAN，因此我抓的順序就會是如我程式碼的方式去抓{0, 1, 2, 3}。那這個部分比較特別的是testbench，我參考了題目提供的tb_R_formatCPU，使用\$readmemh來抓資料。接著下圖為模擬之結果。

```
IM.v
29 `define INSTR_MEM_SIZE 128 // Bytes
30 `define Rtype_op 6'b000100
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 [31:0] InstrAddr
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]; // You, a day ago * put some component and is fine to design
47
48 always@ (InstrAddr)
49 begin
50     Instr[31:0] = {InstrMem[InstrAddr], InstrMem[InstrAddr+1], InstrMem[InstrAddr+2], InstrMem[InstrAddr+3]};
51 end
52
53 endmodule
```

```
tb_IM.v
1 `define INSTR_FILE "testbench/IM.dat"
2 `define INSTR_MAX 128 // bytes
3 `define INSTR_SIZE 8 // bit width
4
5
6 module tb_IM;
7
8     integer i;
9     reg [31:0] InstrAddr;
10    wire [31:0] Instr;
11    reg [`INSTR_SIZE-1 : 0] instrMem [0:`INSTR_MAX-1];
12
13    IM Instruction_Memory(
14        .InstrAddr(InstrAddr),
15        .Instr(Instr)
16    );
17
18    initial begin : Preprocess
19
20        $readmemh(`INSTR_FILE, instrMem); // put the value into instrMem
21        // Initialize intruction memory
22        for (i = 0; i < `INSTR_MAX; i = i + 1)
23        begin
24            Instruction_Memory.InstrMem[i] = instrMem[i];
25        end
26    end
27
28    initial #20 $finish;
29
30    initial fork
31        #0 InstrAddr = 8; // Instr should be 12_32_B0_12.
32        #10 InstrAddr = 2; // Instr should be A0_0B_11_AC
33    join
34
35
36 endmodule
```



1. 第一次Address = 8, Instr should be 12_32_B0_12.
2. 第二次Address = 2, Instr should be A0_0B_11_AC.

下圖為IM.dat，供此部分參考。

```

Part 3 > testbench > IM.dat
You, 4 days ago | 1 author (You)
1 // Instruction Memory in Hex
2 4C // Addr = 0x00
3 13 // Addr = 0x01
4 00 // Addr = 0x02
5 1E // Addr = 0x03
6 12 // Addr = 0x04
7 62 // Addr = 0x05
8 98 // Addr = 0x06
9 23 // Addr = 0x07
10 70 // Addr = 0x08
11 00 // Addr = 0x09
12 00 // Addr = 0x0A
13 00 // Addr = 0x0B
14 FF // Addr = 0x0C
15 FF // Addr = 0x0D
16 FF // Addr = 0x0E
17 FF // Addr = 0x0F
18 FF // Addr = 0x10
19 FF // Addr = 0x11
20 FF // Addr = 0x12
21 FF // Addr = 0x13
22 FF // Addr = 0x14
23 FF // Addr = 0x15
24 FF // Addr = 0x16
25 FF // Addr = 0x17
26 FF // Addr = 0x18
27 FF // Addr = 0x19
28 FF // Addr = 0x1A
29 FF // Addr = 0x1B
  
```

c. Register File

RM這個module其實不難，只要判斷RegWrite是否為1，來決定是否可以將值寫入Reg，那其他部分就是到Register的地址去抓值去輸出。我將RsData與RtData使用assign而非放在always裡面是因為我發現放在裡面會等到正緣觸發才把值送入ALU，那這樣的話就無法達到我們想要的效果了。

```
35 module RF(  
36     // Outputs  
37     output [31:0] RsData,  
38     output [31:0] RtData,  
39     // Inputs  
40     input RegWrite,  
41     input clk,  
42     input [4:0] RsAddr,  
43     input [4:0] RtAddr,  
44     input [4:0] RdAddr,  
45     input [31:0] RdData  
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  
55 assign RsData = R[RsAddr];  
56 assign RtData = R[RtAddr];  
57  
58 always@(posedge clk)  
59 begin  
60     if(RegWrite == 1)  
61     begin  
62         R[RdAddr] = RdData;  
63     end  
64     else  
65     begin  
66         R[RdAddr] = R[RdAddr];  
67     end  
68 end  
69  
70  
71 endmodule  
72
```

```

1  `define DELAY          5    // # * timescale
2  `define REG_SIZE      32    // bit width
3  `define REG_MAX       32    // words
4  `define REG_FILE      "testbench/RF.dat"
5
6  module tb_RF;
7
8      // Inputs
9      reg RegWrite;
10     reg clk;
11     reg [4:0] RsAddr;
12     reg [4:0] RtAddr;
13     reg [4:0] RdAddr;
14     reg [31:0] RdData;
15
16     // Outputs
17     wire [31:0] RsData;
18     wire [31:0] RtData;
19
20     integer i;
21     integer output_reg;
22     reg [`REG_SIZE-1 :0] regMem    [0:`REG_MAX-1];
23
24     RF Register File(
25         // Outputs
26         .RsData(RsData),
27         .RtData(RtData),
28         // Inputs
29         .RegWrite(RegWrite),
30         .clk(clk),
31         .RsAddr(RsAddr),
32         .RtAddr(RtAddr),
33         .RdAddr(RdAddr),
34         .RdData(RdData)
35     );

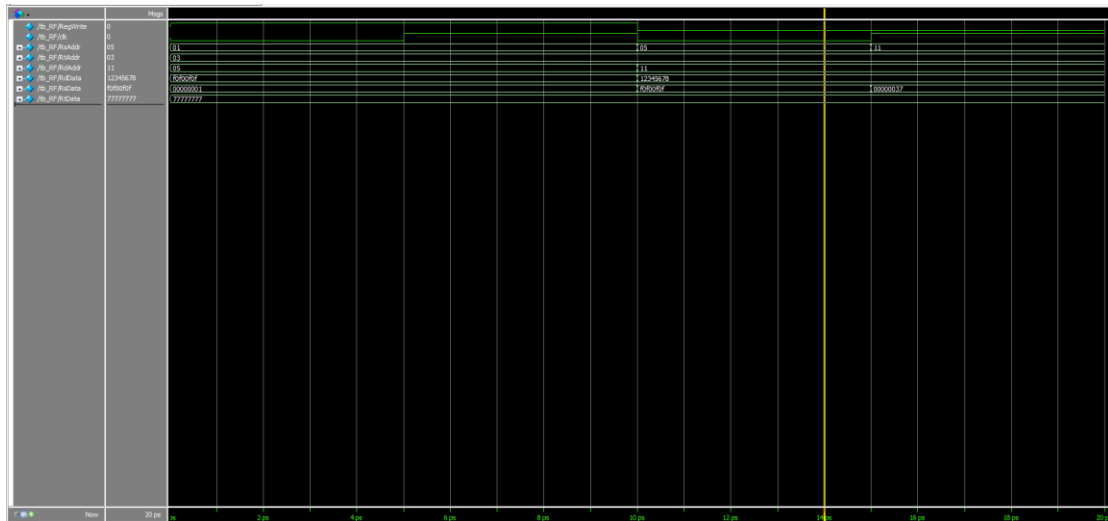
```

```

39     initial begin: Preprocess
40         // Initialize inputs
41         clk = 0;
42         RegWrite = 1;
43         RsAddr = 5'd 1; // Register R[1]
44         RtAddr = 5'd 3; // R[3]
45         RdAddr = 5'd 5; // R[5]
46         RdData = 32'h f0f0_0f0f;
47
48         $readmemh(`REG_FILE,regMem);
49
50         // Initialize register file
51         for (i = 0; i < `REG_MAX; i = i + 1)
52             begin
53                 Register_File.R[i] = regMem[i];
54             end
55     end
56
57     initial #20 $finish;
58
59
60     always begin : ClockGenerator
61         #`DELAY;
62         clk ≤ ~clk;
63     end
64
65
66     initial fork
67         #0 RegWrite = 1;
68         #0 RsAddr = 5'd 1;
69         #0 RtAddr = 5'd 3;
70         #0 RdAddr = 5'd 5;
71         #0 RdData = 32'h F0F0_0F0F;
72
73         #10 RegWrite = 0;
74         #10 RsAddr = 5'd 5; // to show where R[5] is been written or not.
75         #10 RtAddr = 5'd 3;
76         #10 RdAddr = 5'd 17;
77         #10 RdData = 32'h 1234_5678;
78         #15 RsAddr = 5'd 17; // to show where R[17] is been written or not.
79     join
80

```

而下圖這個就是模擬出來的結果，與IM相同的是，我用類似的指令去將值存入一個Reg，接著再去做模擬進行分析。



1. 一開始的時候我是設定讓值可以寫入Reg，所以我把R[5]=F0F0_0F0F
2. 在t=10後我去把RsAddr改成5(也就是第一次的Rd)，因此可以看到我第一次是否有成功寫入，而我們也可以看到RsData變成F0F0_0F0F了。那接著我將RegWrite改為0，試著把它放到R[17]。
3. 在t=15後我去把RsAddr改成17(也就是第二次的Rd)，因此可以看到我第二次因為我關掉RegWrite了，所以沒有成功寫入，可以確認RF正常工作。

```

You, 4 days ago | 1 author (You)
1 // Register File in Hex
2 0000_0000 // R[0]
3 0000_0001 // R[1]
4 0000_0002 // R[2]
5 7777_7777 // R[3]
6 7F7F_7F7F // R[4]
7 F7F7_F7F7 // R[5]
8 7FFF_FFFF // R[6]
9 8000_0000 // R[7]
10 FFFF_0000 // R[8]
11 0000_FFFF // R[9]
12 0000_0011 // R[10]
13 0000_0023 // R[11]
14 0000_0017 // R[12]
15 0000_0090 // R[13]
16 0000_0100 // R[14]
17 0000_0250 // R[15]
18 0000_0300 // R[16]
19 0000_0037 // R[17]
20 0000_0064 // R[18]
21 0000_0030 // R[19]
22 0000_0000 // R[20]
23 0000_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_0000 // R[26]
29 0000_0000 // R[27]
30 0000_0000 // R[28]
31 0000_0000 // R[29]
32 FFFF_FFFF // R[30]
33 FFFF_FFFF // R[31]

```

d. Data Memory

DM這個module其實也不難，只要判斷MemWrite跟MemRead是否為1，來決定是否可以將值寫入Memory或是把Memory的值給讀出來。我將MemReadData使用assign而非放在always裡面，與RF的原因相同。我發現放在裡面會等到下個正緣觸發才把值送出，那這樣的話就無法達到我們想要的效果了。

```
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 [31:0] MemReadData,
38     // Inputs
39     input [31:0] MemAddr,
40     input [31:0] MemWriteData,
41     input MemWrite,
42     input MemRead,
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 assign MemReadData = MemRead? {DataMem[MemAddr],DataMem[MemAddr+1],DataMem[MemAddr+2],DataMem[MemAddr+3]}:32'b0;
53 // You, a day ago + PART3 controller still working
54 always@(posedge clk)
55 begin
56     if(MemWrite == 1)
57     begin
58         {DataMem[MemAddr],DataMem[MemAddr+1],DataMem[MemAddr+2],DataMem[MemAddr+3]} = MemWriteData;
59     end
60     else;
61 end
62
63 endmodule
64
```



```

tb_DM.v 1, U X
Part 3 > tb_DM.v
1  `define DATA_SIZE      8    // bit width
2  `define DATA_MAX      128   // bytes
3  `define DATA_FILE     "testbench/DM.dat"
4  `define DELAY           5    // # * timescale
5
6  module tb_DM();
7      reg clk;
8      reg [31:0] MemAddr;
9      reg [31:0] MemWriteData;
10     reg MemWrite;
11     reg MemRead;
12     wire [31:0] MemReadData;
13     reg [`DATA_SIZE-1 :0] dataMem    [0:`DATA_MAX-1];
14     integer i;
15
16     DM_Data_Memory(
17         .MemAddr(MemAddr),
18         .MemWriteData(MemWriteData),
19         .MemWrite(MemWrite),
20         .MemRead(MemRead),
21         .clk(clk),
22         .MemReadData(MemReadData)
23     );
24

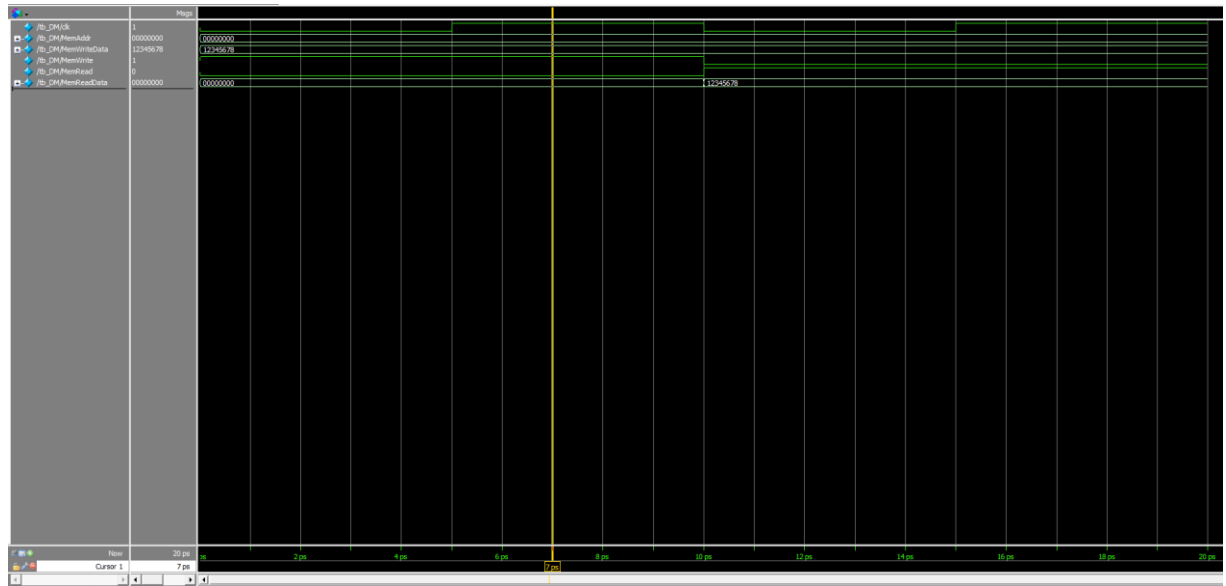
```

```

25     initial begin: Preprocess
26
27         clk = 0;
28
29         $readmemh(`DATA_FILE, dataMem);
30         // Initialize data memory
31         for (i = 0; i < `DATA_MAX; i = i + 1)
32             begin
33                 Data_Memory.DataMem[i] = dataMem[i];
34             end
35
36     end
37
38     always begin : ClockGenerator
39         #`DELAY;
40         clk ≤ ~clk;
41     end
42
43     initial #20 $finish;
44
45     initial fork
46         #0 MemWriteData = 32'h 1234_5678;
47         #0 MemAddr = 32'h 0000_0000;
48         #0 MemWrite = 1;
49         #0 MemRead = 0;
50
51         #10 MemRead = 1;
52         #10 MemWrite = 0;
53
54     join
55
56 endmodule

```

下圖這個就是由testbench模擬出來的結果。



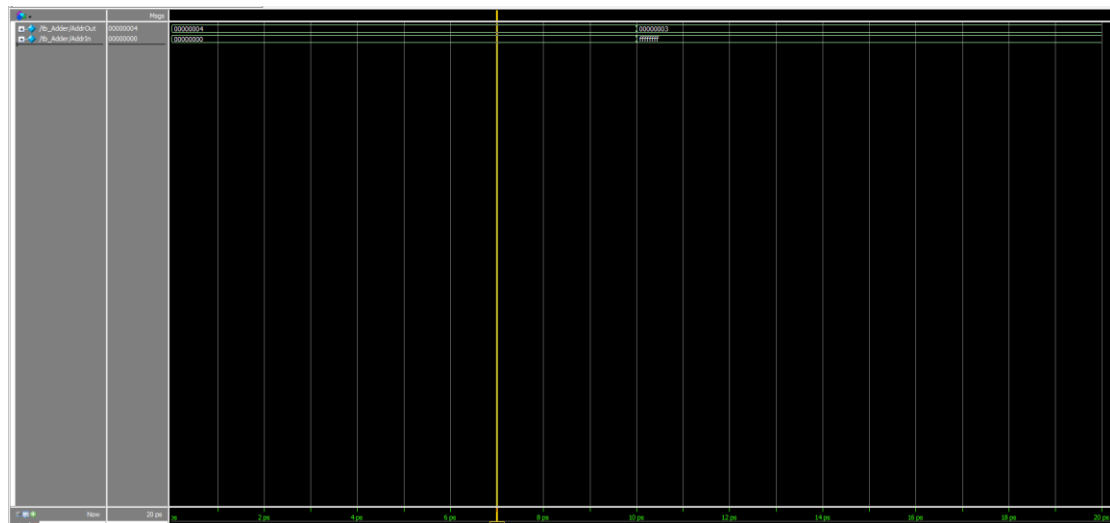
1. 一開始的時候我是設定讓值可以寫入Memory(MemWrite=1)，所以我把Memory{0, 1, 2, 3}=32'h1234_5678.
2. 在t=10後我把MemWrite關掉，把MemRead打開，Address不變，就可以發現我的MemReadData真的在上一個clk被寫入Memory了，輸出為32'h1234_5678.

e. Adder

```
Adderv x
Adderv
You, 2 hours ago | 1 author (You)
1 module Adder
2 (
3     input  [31:0] AddrIn,
4     output [31:0] AddrOut
5 );
6
7     assign AddrOut = AddrIn + 32'd 4;
8
9 endmodule
10

tb_Adderv.t.u x
tb_Adderv
1 module tb_Adder;
2
3     wire [31:0] AddrOut;
4     reg  [31:0] AddrIn;
5
6     Adder A(
7         // Outputs
8         .AddrOut(AddrOut),
9         // Inputs
10        .AddrIn(AddrIn)
11    );
12
13    initial #20 $finish;
14    initial fork
15        #0 AddrIn = 32'd 0;
16
17        #10 AddrIn = 32'h FFFF_FFFF;
18    join
19
20 endmodule
```

Adder所做的事情非常簡單，因為在R-type的AdderOut僅僅需要能夠將AddrIn+4，因此我就只做了加4的動作，然後輸出。（我的加法是unsigned的加法，因為Address沒有負的。）



1. 第一次Input為0，輸出為4。
2. 第二次Input為FFFF_FFFF，輸出為3。

f. ALU

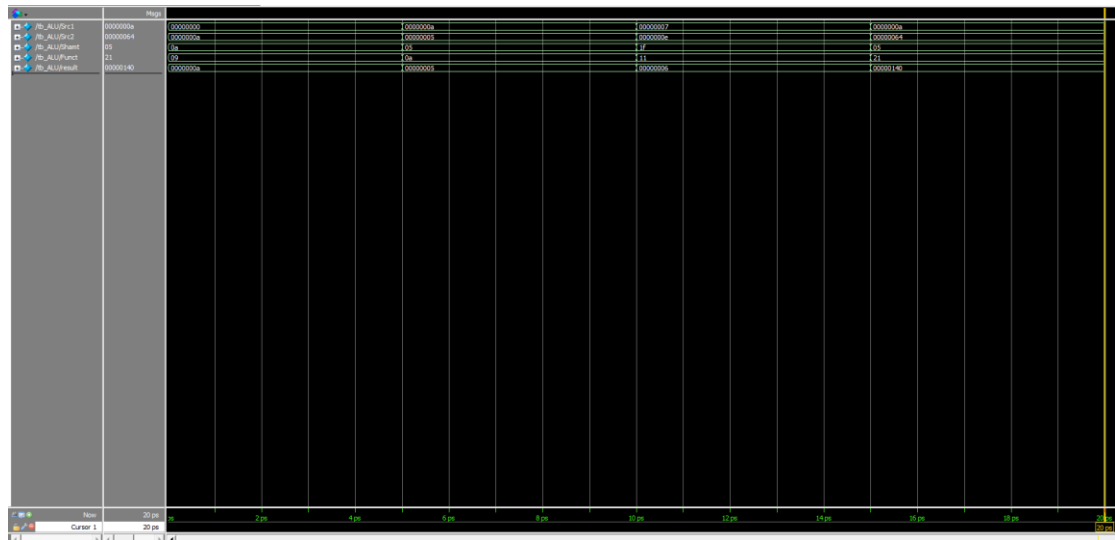
```

1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 module tb_ALU;
7
8     reg [31:0] Src1;
9     reg [31:0] Src2;
10    reg [4:0] Shamt;
11    reg [5:0] Funct;
12    wire [31:0] result;
13
14    ALU A1(
15        .Src1(Src1),
16        .Src2(Src2),
17        .Shamt(Shamt),
18        .Funct(Funct),
19        .result(result)
20    );
21
22    initial #20 $finish;
23    initial fork
24        #0 Src1 = 32'd 0;
25        #0 Src2 = 32'd 10;
26        #0 Funct = `addu;
27        #0 Shamt = 5'd 10; // result should be 10(hex).
28
29        #5 Src1 = 32'd 10;
30        #5 Src2 = 32'd 5;
31        #5 Funct = `subu;
32        #5 Shamt = 5'd 5; // result should be 5(hex).
33
34        #10 Src1 = 32'd 7; // 0111
35        #10 Src2 = 32'd 14; // 1110
36        #10 Funct = `AND;
37        #10 Shamt = 5'd 31; // result should be 6h (0110).
38
39        #15 Src1 = 32'd 10; // 1010
40        #15 Src2 = 32'd 100; // This value would not affect the result.
41        #15 Funct = `sll; // shift left logic
42        #15 Shamt = 5'd 5; // 10*2^5 = 320.(d) (140H)
43
44    join
45 endmodule
    
```

```

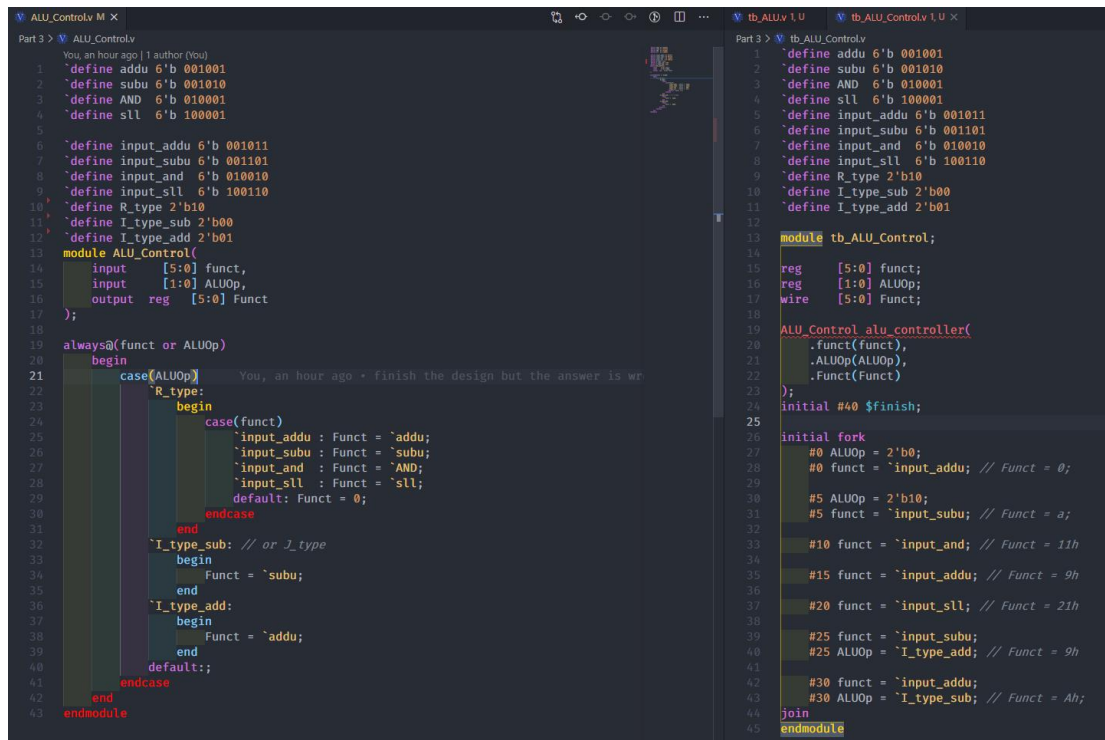
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 module ALU(
7     input [31:0] Src1,
8     input [31:0] Src2,
9     input [4:0] Shamt,
10    input [5:0] Funct,
11    output reg [31:0] result
12);
13
14 always@(Funct or Shamt or Src1 or Src2)
15 begin
16     case (Funct)
17         `addu : result = Src1 + Src2;
18         `subu : result = Src1 - Src2;
19         `AND : result = Src1 & Src2;
20         `sll : result = Src1 << Shamt;
21     endcase
22 end
23
24 endmodule
    
```

ALU 主要是用來做Src1 and Src2的運算，由輸入的Funct來決定要做甚麼工作。下圖為由testbench產生之模擬結果。



1. 第一次Input為0+A，輸出為A。
2. 第二次Input為A-5，輸出為5。
3. 第三次Input為(1f)&(11)，輸出為6。
4. 第四次Input為A<<5=A*2^5=320，輸出為140。

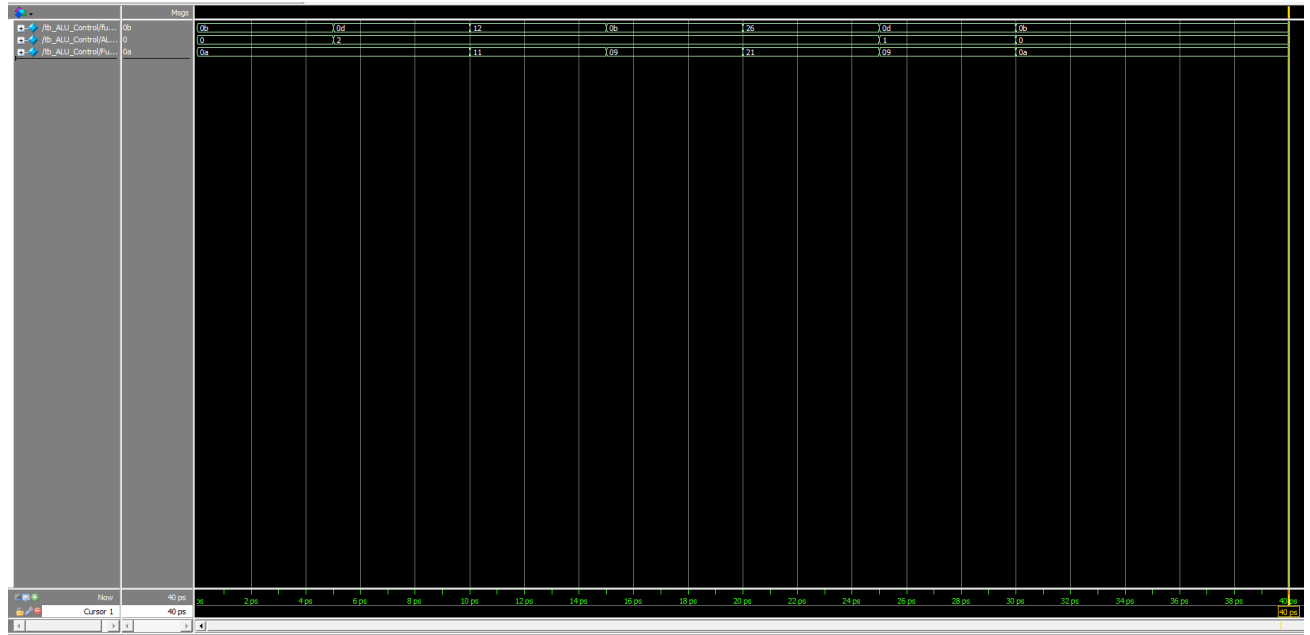
g. ALU_Control



```
Part 3 > ALU_Control.v
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 `define input_addu 6'b 001011
7 `define input_subu 6'b 001101
8 `define input_and 6'b 010010
9 `define input_sll 6'b 100110
10 `define R_type 2'b10
11 `define I_type_sub 2'b00
12 `define I_type_add 2'b01
13 module ALU_Control(
14     input [5:0] funct,
15     input [1:0] ALUOp,
16     output reg [5:0] Funct
17 );
18
19 always@(funct or ALUOp)
20 begin
21     case(ALUOp)
22         R_type:
23             begin
24                 case(funct)
25                     'input_addu : Funct = `addu;
26                     'input_subu : Funct = `subu;
27                     'input_and : Funct = `AND;
28                     'input_sll : Funct = `sll;
29                     default: Funct = 0;
30                 endcase
31             end
32         'I_type_sub: // or J_type
33             begin
34                 Funct = `subu;
35             end
36         'I_type_add:
37             begin
38                 Funct = `addu;
39             end
40         default:;
41     endcase
42 end
43 endmodule

Part 3 > tb_ALU_Control.v
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5 `define input_addu 6'b 001011
6 `define input_subu 6'b 001101
7 `define input_and 6'b 010010
8 `define input_sll 6'b 100110
9 `define R_type 2'b10
10 `define I_type_sub 2'b00
11 `define I_type_add 2'b01
12
13 module tb_ALU_Control;
14
15 reg [5:0] funct;
16 reg [1:0] ALUOp;
17 wire [5:0] Funct;
18
19 ALU_Control alu_controller(
20     .funct(funct),
21     .ALUOp(ALUOp),
22     .Funct(Funct)
23 );
24 initial #40 $finish;
25
26 initial fork
27     #0 ALUOp = 2'b0;
28     #0 funct = `input_addu; // Funct = 0;
29
30     #5 ALUOp = 2'b10;
31     #5 funct = `input_subu; // Funct = a;
32
33     #10 funct = `input_and; // Funct = 11h
34
35     #15 funct = `input_addu; // Funct = 9h
36
37     #20 funct = `input_sll; // Funct = 21h
38
39     #25 funct = `input_subu;
40     #25 ALUOp = `I_type_add; // Funct = 9h
41
42     #30 funct = `input_addu;
43     #30 ALUOp = `I_type_sub; // Funct = Ah;
44 join
45 endmodule
```

ALU_Control主要是用來控制ALU工作與否，及將instr的指令轉為ALU可以理解的function code。而在I-type跟J-type指令下，吃的主要就是ALUOp，由ALUOp來決定他們的function out. 下圖為TestBench之模擬結果。



1. 第一次ALUOP為0，ALU不可以工作，Funct輸出為0。
2. 第二次開始ALUOP為2' b10，ALU可以工作，Funct輸出為Instr對應的subu。
3. 第三次Input為input_and，輸出為Instr對應的AND。
4. 第四次Input為input_sll，輸出為Instr對應的sll。
5. 第五次Input為input_addu，輸出為Instr對應的addu。
6. 第六次ALUOp為 I-type-add， funct Input為input_subu，但是在I-type指令下，ALU不會理會funct，因此輸出為Instr對應的addu。
7. 第七次ALUOp為 I-type-sub， funct Input為input_addu，但是在I-type指令下，ALU不會理會funct，因此輸出為Instr對應的subu。

h. Control

Control這個module在整個CPU裡面是一個非常重要的角色。他掌管整顆CPU現在要做甚麼，不要做甚麼。雖然他極其重要，但是其實沒有甚麼太複雜的工作，只要依照Opcode的要求，來決定是否要送各種訊號。而因為R-type的所有指令都會使用到ALU，因此我們ALUOp只要是對的Opcode，我們一律送2'b10. 而對於I-type指令來說，只會有加法跟減法，因此除了subiu以外(2'b00)，其他的ALUOp都為2'b01，剩下的是J-type指令，因為branch用到的是減法，因此ALUOp為0。

```
You, seconds ago | 1 author (You)
1 // I-type sub
2 `define I_type_sub 2'b00
3 // I-type add
4 `define I_type_add 2'b01
5 module Control(
6     input  [5:0]  OpCode,
7     output reg    RegWrite,
8     output reg [1:0] ALUOp,
9     output reg    RegDst,
10    output reg    ALUSrc,
11    output reg    MemWrite, // write memory or not.
12    output reg    MemRead,
13    output reg    MemtoReg
14 );
15
16 always@(OpCode)
17 begin
18     case (OpCode)
19         6'd 4:
20             begin
21                 RegWrite = 1'b 1; // R format.
22                 ALUOp = 2'b 10;
23                 RegDst = 1; // R format.
24                 ALUSrc = 0;
25                 MemWrite = 0;
26                 MemRead = 0;
27                 MemtoReg = 0;
28             end
29         // I format.
30         6'd 12: // addiu
31             begin
32                 RegWrite = 1'b 1;
33                 ALUOp = `I_type_add;
34                 RegDst = 0; // I format → write into Rt
35                 ALUSrc = 1;
36                 MemWrite = 0;
37                 MemRead = 0;
38                 MemtoReg = 0;
39             end
40         // J format.
41         6'd 17:
42             begin
43                 RegWrite = 1'b 1;
44                 ALUOp = 0;
45                 RegDst = 0;
46                 ALUSrc = 0;
47                 MemWrite = 0;
48                 MemRead = 0;
49                 MemtoReg = 0;
50             end
51     endcase
52 end
```

```

40 6'd 13: // subiu
41 begin
42     RegWrite = 1'b 1;
43     ALUOp = `I_type_sub;
44     RegDst = 0; // I format → write into Rt
45     ALUSrc = 1;
46     MemWrite=0;
47     MemRead = 0;
48     MemtoReg = 0;
49 end
50 6'd 16: // sw
51 begin
52     RegWrite = 1'b 0;
53     ALUOp = `I_type_add;
54     RegDst = 1'b x; // I format → write into Rt
55     ALUSrc = 1;
56     MemWrite = 1;
57     MemRead = 0;
58     MemtoReg = 1'b x; // Since the SW would not read the value from memory.
59 end
60 6'd 17: // lw
61 begin
62     RegWrite = 1'b 1;
63     ALUOp = `I_type_add;
64     RegDst = 0; // I format → write into Rt
65     ALUSrc = 1;
66     MemWrite = 0;
67     MemRead = 1;
68     MemtoReg= 1;
69 end
70 default:
71 begin
72     MemWrite = 0;
73     RegWrite = 0;
74 end
75 endcase
76
77 end
78
79 endmodule

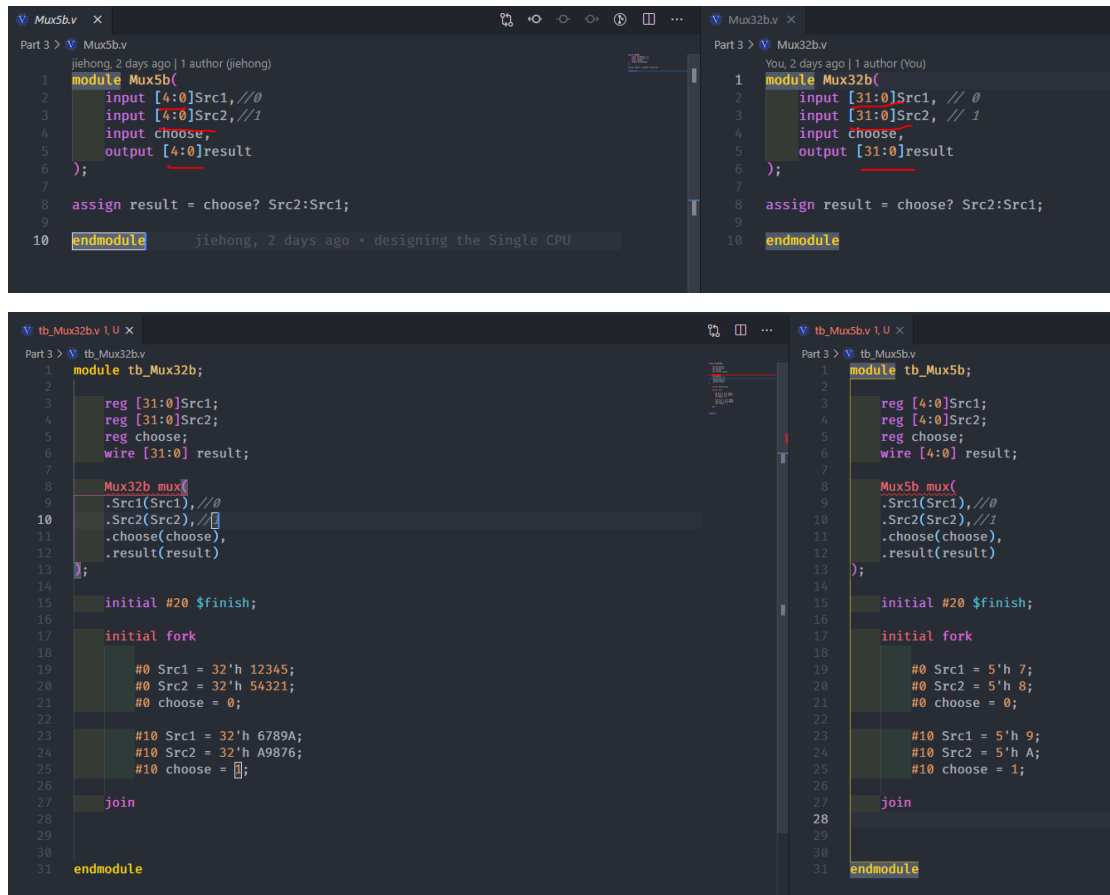
```


Part 2 > tb_Control.v

```
1  module tb_Control;
2
3      reg      [5:0] OpCode;
4      wire     RegWrite;
5      wire     [1:0] ALUOp;
6      wire     RegDst;
7      wire     ALUSrc;
8      wire     MemWrite;
9      wire     MemRead;
10     wire     MemtoReg;
11
12     Control controller(
13         .OpCode(OpCode),
14         .RegWrite(RegWrite),
15         .ALUOp(ALUOp),
16         .RegDst(RegDst),
17         .ALUSrc(ALUSrc),
18         .MemWrite(MemWrite),
19         .MemRead(MemRead),
20         .MemtoReg(MemtoReg)
21     );
22
23     initial #40 $finish;
24
25     initial fork
26         #0 OpCode = 6'd 0; // Not working
27
28         #5 OpCode = 6'd 4;
29
30         #10 OpCode = 6'd 12;
31
32         #15 OpCode = 6'd 13;
33
34         #20 OpCode = 6'd 16;
35
36         #25 OpCode = 6'd 17;
37
38     join
39 endmodule
```


i. MUX

MUX這二個module其實非常簡單，一個是5bit, 一個是32bit. 只要判斷choose選的是多少，就決定輸出要送哪一個輸入出來。



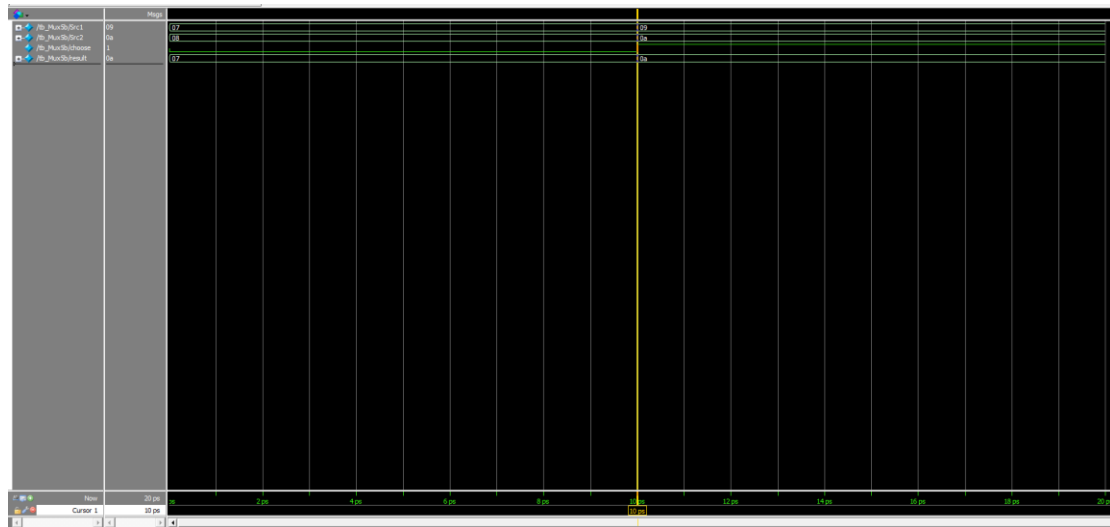
```
Part 3 > Mux5b.v
1 jiehong, 2 days ago | 1 author (jiehong)
2 module Mux5b(
3     input [4:0]Src1,//0
4     input [4:0]Src2,//1
5     input choose,
6     output [4:0]result
7 );
8 assign result = choose? Src2:Src1;
9
10 endmodule jiehong, 2 days ago • designing the Single CPU

Part 3 > Mux32b.v
1 You, 2 days ago | 1 author (You)
2 module Mux32b(
3     input [31:0]Src1, // 0
4     input [31:0]Src2, // 1
5     input choose,
6     output [31:0]result
7 );
8 assign result = choose? Src2:Src1;
9
10 endmodule

Part 3 > tb_Mux32b.v
1 module tb_Mux32b;
2
3     reg [31:0]Src1;
4     reg [31:0]Src2;
5     reg choose;
6     wire [31:0] result;
7
8     Mux32b mux(
9         .Src1(Src1),//0
10        .Src2(Src2),//1
11        .choose(choose),
12        .result(result)
13    );
14
15    initial #20 $finish;
16
17    initial fork
18
19        #0 Src1 = 32'h 12345;
20        #0 Src2 = 32'h 54321;
21        #0 choose = 0;
22
23        #10 Src1 = 32'h 6789A;
24        #10 Src2 = 32'h A9876;
25        #10 choose = 1;
26
27    join
28
29
30
31 endmodule

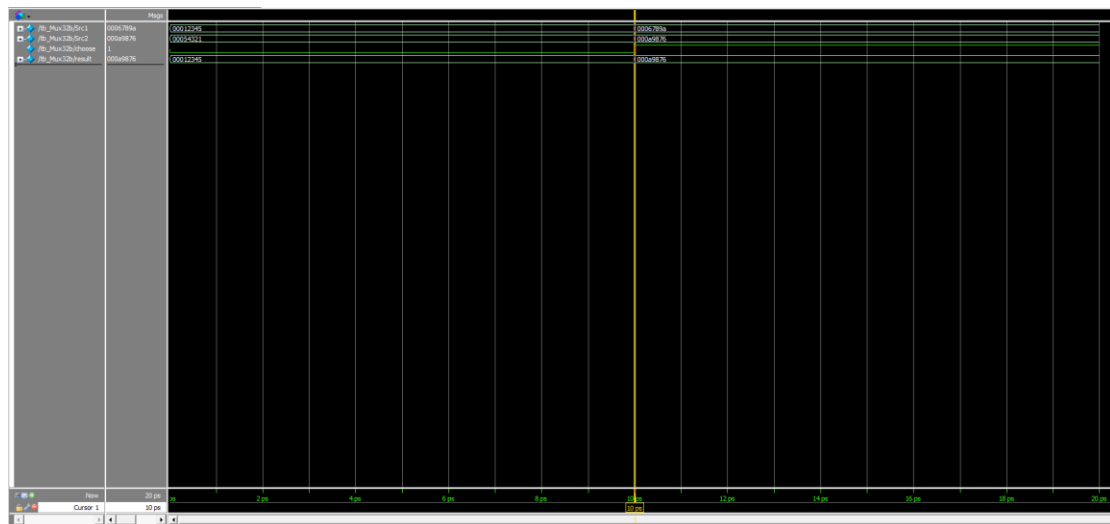
Part 3 > tb_Mux5b.v
1 module tb_Mux5b;
2
3     reg [4:0]Src1;
4     reg [4:0]Src2;
5     reg choose;
6     wire [4:0] result;
7
8     Mux5b mux(
9         .Src1(Src1),//0
10        .Src2(Src2),//1
11        .choose(choose),
12        .result(result)
13    );
14
15    initial #20 $finish;
16
17    initial fork
18
19        #0 Src1 = 5'h 7;
20        #0 Src2 = 5'h 8;
21        #0 choose = 0;
22
23        #10 Src1 = 5'h 9;
24        #10 Src2 = 5'h A;
25        #10 choose = 1;
26
27    join
28
29
30
31 endmodule
```

下圖MUX5b這個testbench模擬出來的結果。只要判斷choose選的是多少，就決定輸出要送哪一個輸入出來。



1. 一開始的時候Src1 = 7, Src2 = 8, choose = 0, 因此result會選擇Src1, 輸出為07.
2. T=10的時候Src1 = 9, Src2 = A, choose = 1, 因此result會選擇Src2, 輸出為0A.

下圖MUX32b這個testbench模擬出來的結果。只要判斷choose選的是多少，就決定輸出要送哪一個輸入出來。



1. 一開始的時候Src1 = 12345, Src2 = 54321, choose = 0, 因此result會選擇Src1, 輸出為12345.
2. T=10的時候Src1 = 6789A, Src2 = A9876, choose = 1, 因此result會選擇Src2, 輸出為A9876.

3. J-format (Simple CPU)

a. Simple CPU

```
29 module SimpleCPU(  
30     // Outputs  
31     output wire [31:0] AddrOut,  
32     // Inputs  
33     input wire [31:0] AddrIn,  
34     input wire clk  
35 );  
36  
37 // Adder  
38 wire [31:0] AdderDataOut1;  
39 wire [31:0] AdderDataOut2;  
40  
41 // ALU  
42 wire [31:0] ALU_result;  
43 wire zeroFlag;  
44  
45 // ALU controller  
46 wire [5:0] Funct;  
47  
48 // MUX  
49 wire [31:0] MUX32A_result; // middle left  
50 wire [31:0] MUX32B_result; // middle right  
51 wire [31:0] MUX32C_result; // top left  
52 wire [31:0] MUX32D_result; // top right  
53 wire [4:0] MUX5_result;  
54 wire [31:0] Sign_Extend; // input for mux and Adder  
55  
56 // Register File  
57 wire [31:0] RsData;  
58 wire [31:0] RtData;  
59  
60 // Controller  
61 wire RegWrite;  
62 wire RegDst;  
63 wire ALUSrc;  
64 wire MemWrite;  
65 wire MemRead;  
66 wire MemtoReg;  
67 wire Jump;  
68 wire Branch;  
69 wire [1:0] ALUOp;  
70  
71 // Data Memory  
72 wire [31:0] MemReadData;  
73 // IM  
74 wire [31:0] Instr;  
75  
76 assign Sign_Extend = Instr[15]?{16'hFFFF,Instr[15:0]}:{16'h0000,Instr[15:0]};
```

```

83  ~  IM Instr_Memory(
84      // Outputs
85      .Instr(Instr),
86      // Inputs
87      .InstrAddr(AddrIn)
88  );
89
90  ~  Adder adder1(
91      // Outputs
92      .DataOut(AdderDataOut1),
93      // Inputs
94      .Src1(32'd4),
95      .Src2(AddrIn)
96  );
97
98  ~  Adder adder2(
99      // Outputs
100     .DataOut(AdderDataOut2),
101     // Inputs
102     .Src1(AdderDataOut1),
103     .Src2(Sign_Extend << 2)
104 );
105
106 ~  Mux5b mux5b(
107     .Src1(Instr[20:16]),
108     .Src2(Instr[15:11]),
109     .choose(RegDst),
110     .result(MUX5_result)
111 );
112
113 ~  /*
114     * Declaration of Register File.
115     * CAUTION: DONT MODIFY THE NAME.
116     */
117 ~  RF Register_File(
118     // Outputs
119     .RsData(RsData),
120     .RtData(RtData),
121     // Inputs
122     .clk(clk),
123     .RegWrite(RegWrite),
124     .RsAddr(Instr[25:21]),
125     .RtAddr(Instr[20:16]),
126     .RdAddr(MUX5_result),
127     .RdData(MUX32B_result)
128 );

```

```

130     Mux32b_A32(
131         .Src1(RtData),
132         .Src2(Sign_Extend),
133         .result(MUX32A_result),
134         .choose(ALUSrc)
135     );
136
137     ALU_alu(
138         .Src1(RsData),
139         .Src2(MUX32A_result),
140         .Shamt(Instr[10:6]),
141         .Funct(Funct),
142         .result(ALU_result),
143         .Zero(zeroFlag)
144     );
145
146     Mux32b_B32(
147         .Src1(ALU_result),
148         .Src2(MemReadData),
149         .choose(MemtoReg),
150         .result(MUX32B_result)
151     );
152
153     Mux32b_C32(
154         .Src1(AdderDataOut1),
155         .Src2(AdderDataOut2),
156         .result(MUX32C_result),
157         .choose(Branch & zeroFlag)
158     );
159
160     Mux32b_D32(
161         .Src1(MUX32C_result),
162         .Src2({AdderDataOut1[31:28], Instr[25:0], 2'b00}), // 4 + 26 + 2 = 32;
163         .result(AddrOut),
164         .choose(Jump)
165     );

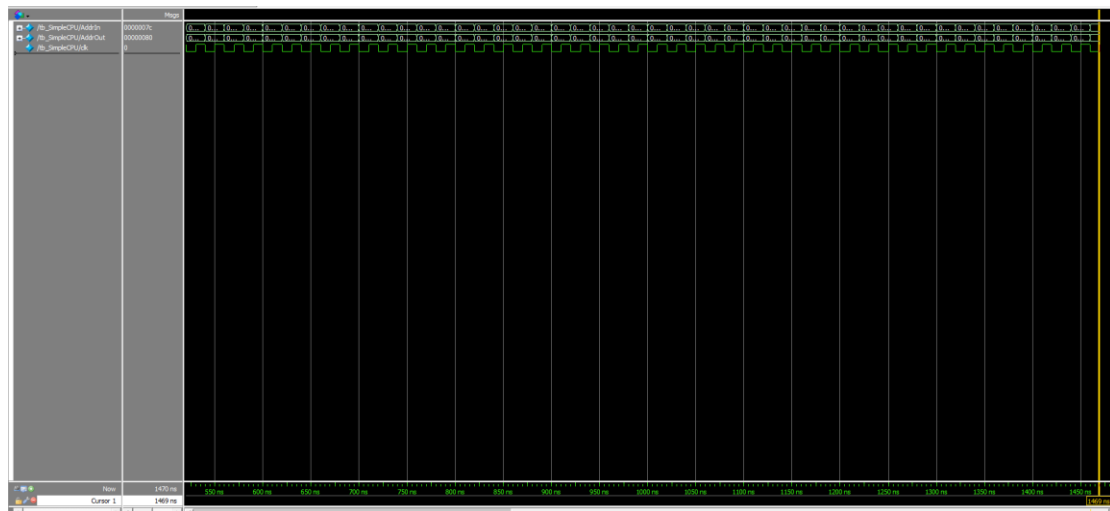
```

```

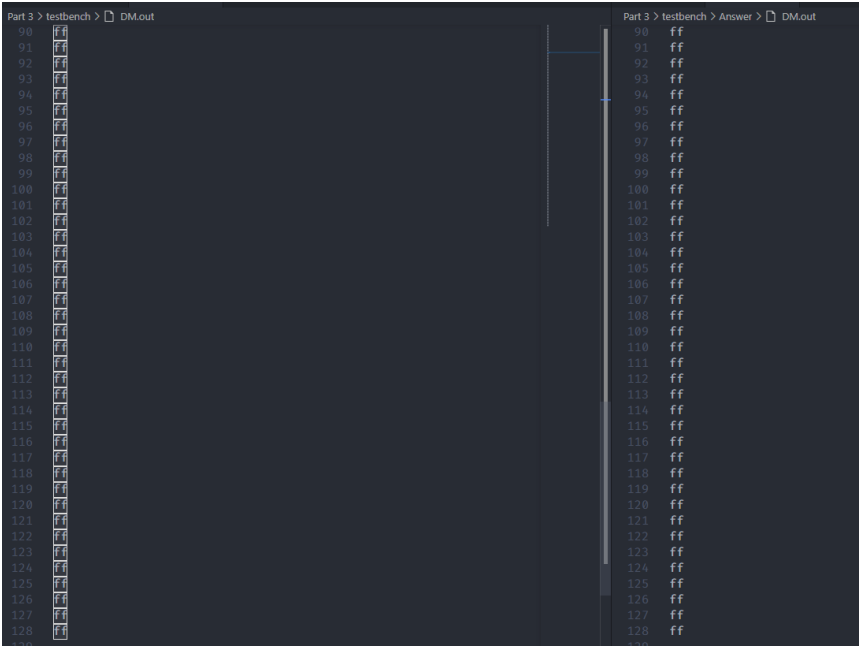
169     DM_Data_Memory(
170         // Outputs
171         .MemReadData(MemReadData),
172         // Inputs
173         .MemAddr(ALU_result),
174         .MemWriteData(RtData),
175         .MemWrite(MemWrite),
176         .MemRead(MemRead),
177         .clk(clk)
178     );
179
180     Control_controller(
181         .OpCode(Instr[31:26]),
182         .RegDst(RegDst),
183         .Branch(Branch),
184         .RegWrite(RegWrite),
185         .ALUSrc(ALUSrc),
186         .MemWrite(MemWrite),
187         .MemRead(MemRead),
188         .MemtoReg(MemtoReg),
189         .Jump(Jump),
190         .ALUOp(ALUOp)
191     );
192
193     ALU_Control_alu_controller(
194         .funct(Instr[5:0]),
195         .ALUOp(ALUOp),
196         .Funct(Funct)
197     );
198 endmodule

```

tb_SimpleCPU 與題目所提供之檔案相同，因篇幅限制而不另行截圖。這個 SimpleCPU 是將所有模組集大成之結果。我在裡面宣告了一些 wire 讓他去相互連接。這裡面比較特別的是有一個被我宣告的 wire 叫作 SignExtend, 是去把輸進來的 16bit 的 Immediate 值，延長到 32bit. 接著因為在 103 行，得對 SignExtend 左移兩位的動作，我沒有使用一個新的模組來幫我處理，而是直接寫在裡面讓他左移兩格等等比較簡單且少次的操作，我僅僅在 SimpleCPU 去處理。下圖為 TestBench 模擬之結果及 RF.out, DM.out 之輸出結果。模擬出來的結果我們看最後一個 clk，可以發現時間是 t=1470，根據 Part3. ASM 我們可以發現我們的指令是當 $R19 == R0$ 的時候，就跳到 30 那邊。而一開始 $R[19]=30H$ 不會等於 $R[0]$ ，因此下一行會要求我們去進行一個 subu 的動作， $R[2] = 2H$ ，下一行又會無跳件跳到 0 的位置，這時候我們可以計算 $30H/2H=24(decimal)$ ，又從模擬之波行觀察，可以發現每個 clk 佔 $t = 20$ ，而我們會做 3 個指令，所以做完一次就是 $20*3=60$ 。又我們要做 24 次我們才可以讓指令跳到 30 那邊，因此我們計算 $60*24=1440$ ，而跳到 1440 後，又經過一個 clk， $1440+20=1460(AddrIn=7C)$ ，讓 $PC=PC+4$ ，這時候 $PC=80$ ，又等到正緣觸發($1460+10=1470$)，就超過了 testbench 之限制，模擬結束。因此我們可以判定我們的結果是正確的。

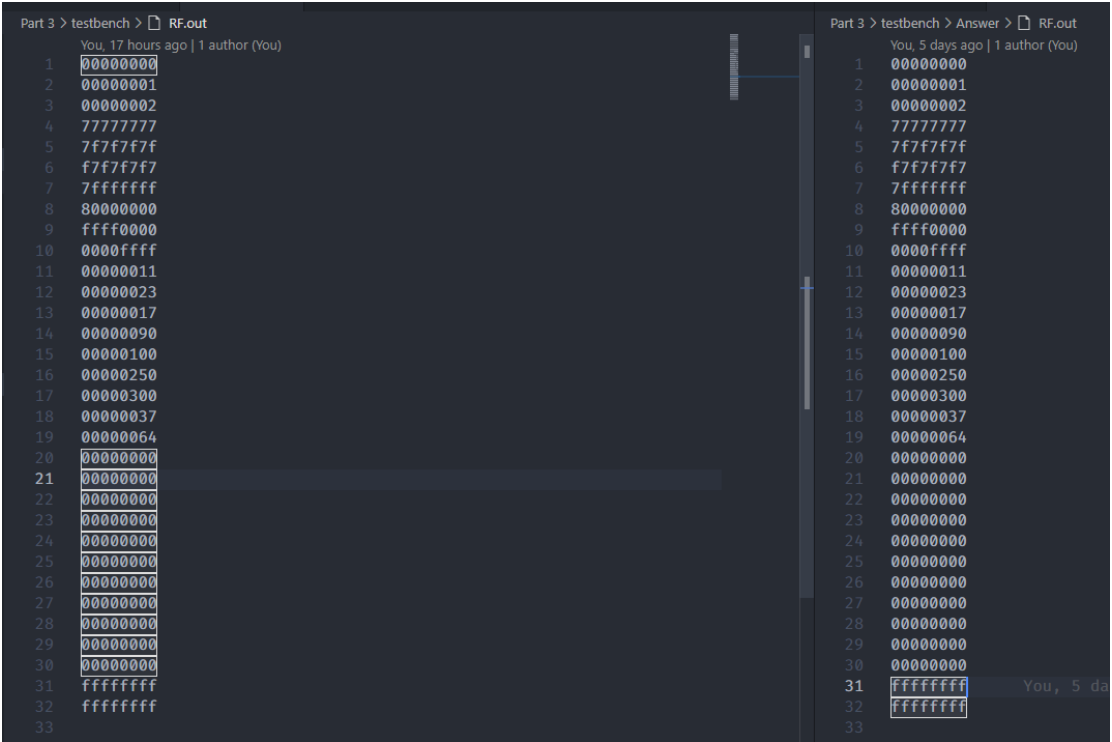


而下方右圖是DM.out經過執行後輸出出來的結果(全部都為FF) 而對比於題目提供之檔案(右圖)可以發現完全相同。



The image shows two side-by-side terminal windows. The left window is titled 'Part 3 > testbench > DM.out' and displays a list of 28 lines, each containing the text 'ff'. The right window is titled 'Part 3 > testbench > Answer > DM.out' and also displays a list of 28 lines, each containing the text 'ff'. A vertical scrollbar is visible between the two windows, indicating they are being compared line by line.

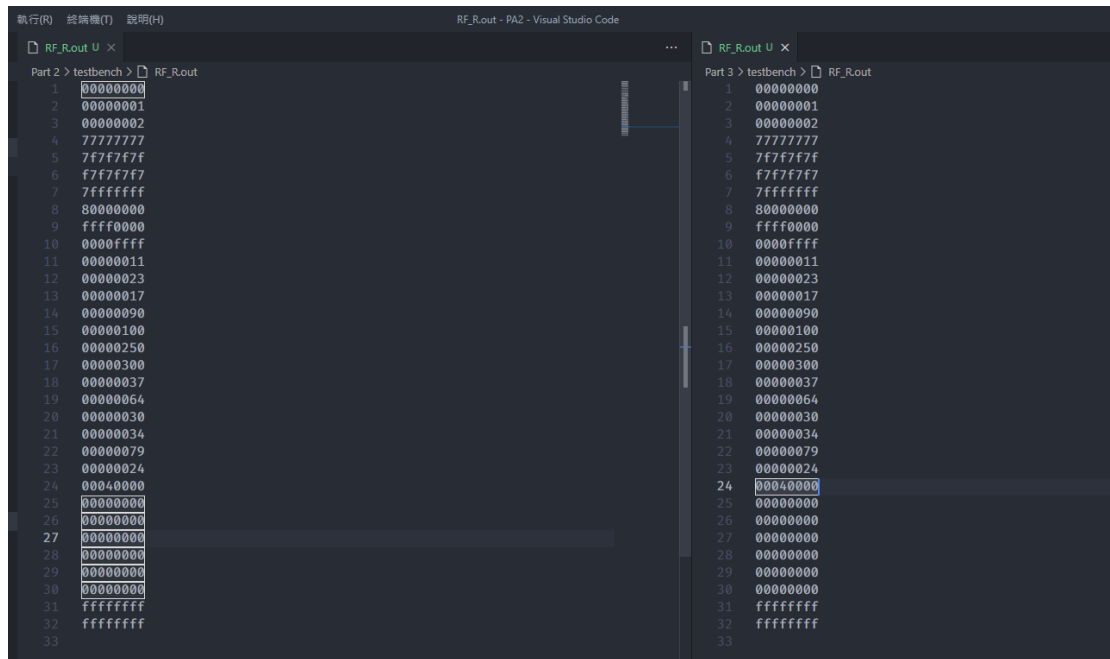
而下方右圖是RF.out(J-type)經過執行後輸出出來的結果，而對比於題目提供之檔案(右圖)可以發現完全相同，因此SimpleCPU到此順利做完。



The image shows two side-by-side terminal windows. The left window is titled 'Part 3 > testbench > RF.out' and displays a list of 33 lines of hexadecimal values. The right window is titled 'Part 3 > testbench > Answer > RF.out' and displays a list of 33 lines of hexadecimal values. A vertical scrollbar is visible between the two windows, indicating they are being compared line by line.

Line	Left RF.out	Right RF.out
1	00000000	00000000
2	00000001	00000001
3	00000002	00000002
4	77777777	77777777
5	7f7f7f7f	7f7f7f7f
6	f7f7f7f7	f7f7f7f7
7	7fffffff	7fffffff
8	80000000	80000000
9	ffff0000	ffff0000
10	0000ffff	0000ffff
11	00000011	00000011
12	00000023	00000023
13	00000017	00000017
14	00000090	00000090
15	00000100	00000100
16	00000250	00000250
17	00000300	00000300
18	00000037	00000037
19	00000064	00000064
20	00000000	00000000
21	00000000	00000000
22	00000000	00000000
23	00000000	00000000
24	00000000	00000000
25	00000000	00000000
26	00000000	00000000
27	00000000	00000000
28	00000000	00000000
29	00000000	00000000
30	00000000	00000000
31	ffffffff	ffffffff
32	ffffffff	ffffffff
33		

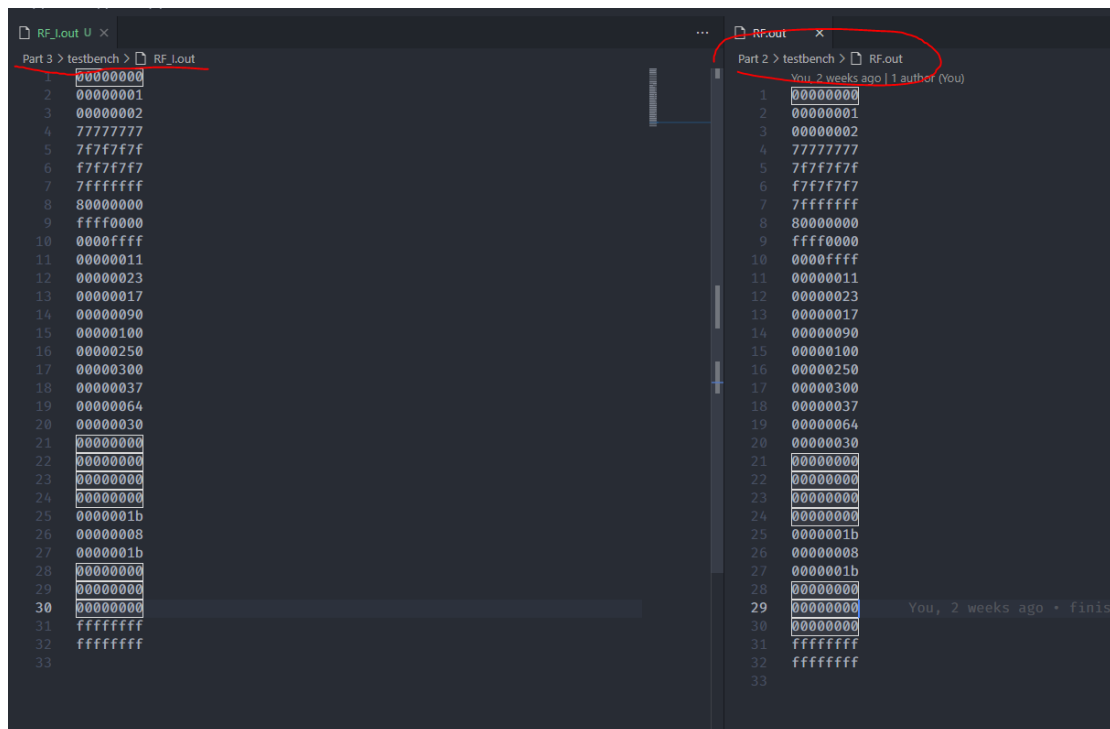
下圖是Rtype指令在SimpleCPU上模擬後輸出的結果，我們可以對比part2的RF_R.out輸出，可以發現一模一樣，因此可以確認成功，



```
Part 2 > testbench > RF_R.out
1 00000000
2 00000001
3 00000002
4 77777777
5 7f7f7f7f
6 f7f7f7f7
7 7fffffff
8 80000000
9 ffff0000
10 0000ffff
11 00000011
12 00000023
13 00000017
14 00000090
15 00000100
16 00000250
17 00000300
18 00000037
19 00000064
20 00000030
21 00000034
22 00000079
23 00000024
24 00040000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 ffffffff
32 ffffffff
33

Part 3 > testbench > RF_R.out
1 00000000
2 00000001
3 00000002
4 77777777
5 7f7f7f7f
6 f7f7f7f7
7 7fffffff
8 80000000
9 ffff0000
10 0000ffff
11 00000011
12 00000023
13 00000017
14 00000090
15 00000100
16 00000250
17 00000300
18 00000037
19 00000064
20 00000030
21 00000034
22 00000079
23 00000024
24 00040000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 ffffffff
32 ffffffff
33
```

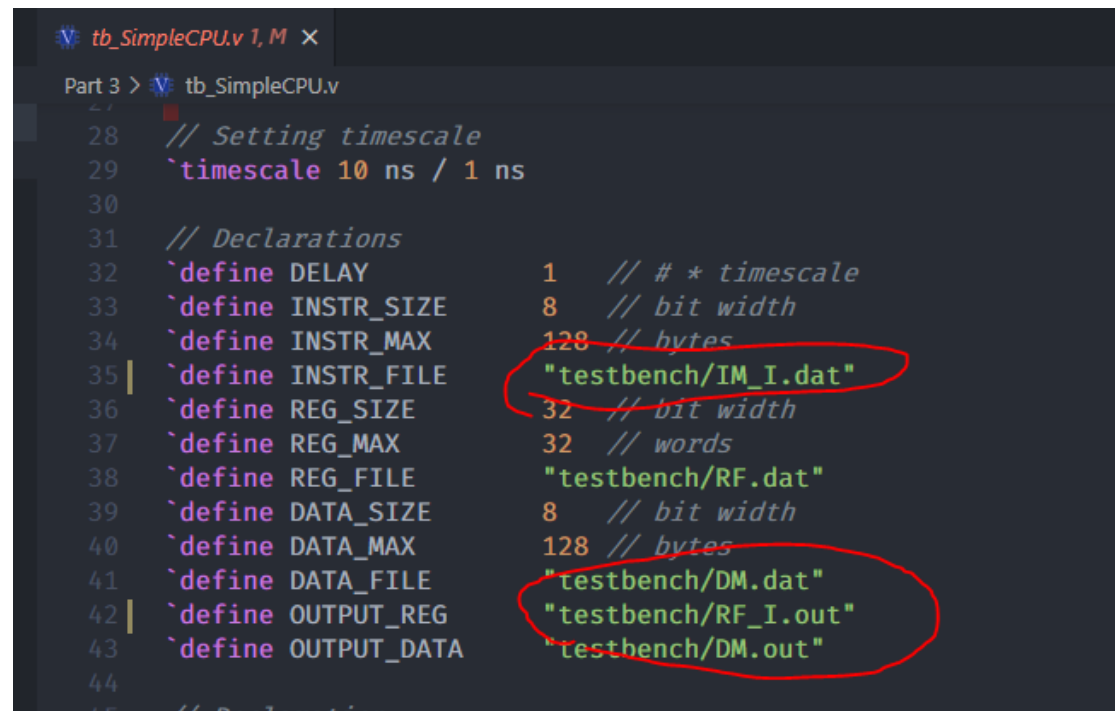
下圖是Itype指令在SimpleCPU上模擬後輸出的結果，我們可以對比part2的RF.out輸出，可以發現一模一樣，因此可以確認成功，



```
Part 3 > testbench > RF_I.out
1 00000000
2 00000001
3 00000002
4 77777777
5 7f7f7f7f
6 f7f7f7f7
7 7fffffff
8 80000000
9 ffff0000
10 0000ffff
11 00000011
12 00000023
13 00000017
14 00000090
15 00000100
16 00000250
17 00000300
18 00000037
19 00000064
20 00000030
21 00000000
22 00000000
23 00000000
24 00000000
25 0000001b
26 00000008
27 0000001b
28 00000000
29 00000000
30 00000000
31 ffffffff
32 ffffffff
33

Part 2 > testbench > RF.out
You, 2 weeks ago | 1 author (You)
1 00000000
2 00000001
3 00000002
4 77777777
5 7f7f7f7f
6 f7f7f7f7
7 7fffffff
8 80000000
9 ffff0000
10 0000ffff
11 00000011
12 00000023
13 00000017
14 00000090
15 00000100
16 00000250
17 00000300
18 00000037
19 00000064
20 00000030
21 00000000
22 00000000
23 00000000
24 00000000
25 0000001b
26 00000008
27 0000001b
28 00000000
29 00000000
30 00000000
31 ffffffff
32 ffffffff
33
You, 2 weeks ago • finish
```

比較值得注意的是我測試這些指令是否成功的方法是在testbench改變他們讀及輸出的檔名。讓他們得以讀入不同IM，且可輸出不同的RF。



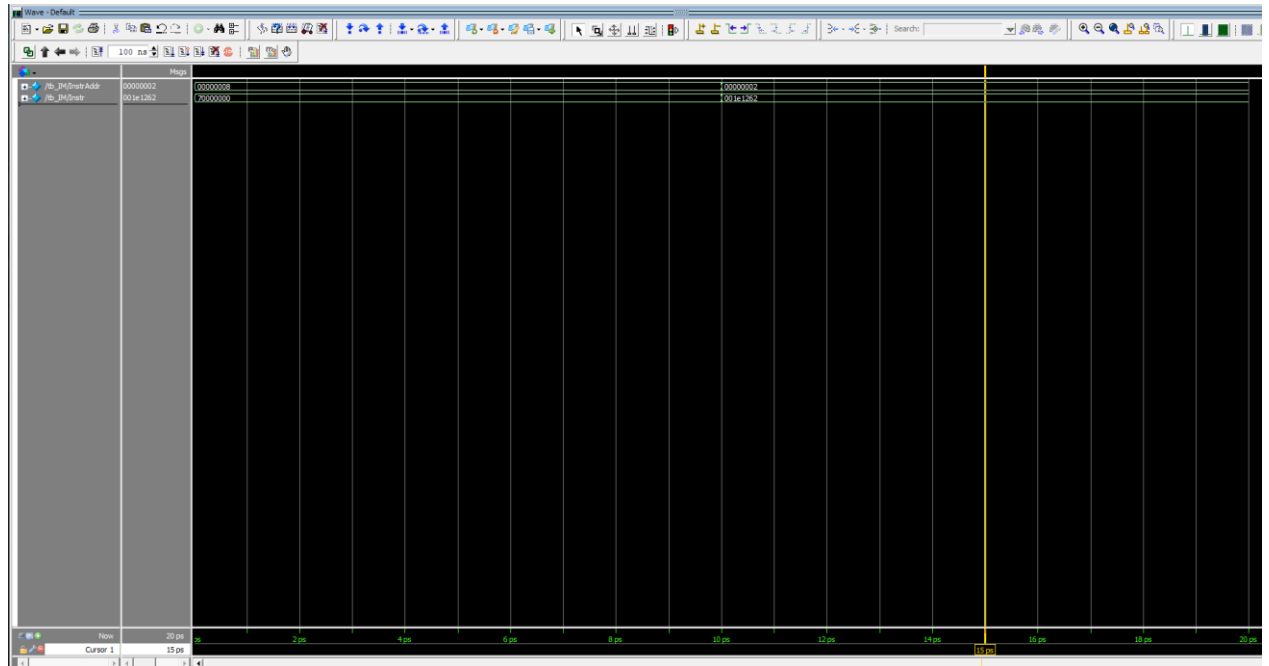
```
tb_SimpleCPU.v 1, M X
Part 3 > tb_SimpleCPU.v
28 // Setting timescale
29 `timescale 10 ns / 1 ns
30
31 // Declarations
32 `define DELAY 1 // # * timescale
33 `define INSTR_SIZE 8 // bit width
34 `define INSTR_MAX 128 // bytes
35 `define INSTR_FILE "testbench/IM_I.dat"
36 `define REG_SIZE 32 // bit 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_I.out"
43 `define OUTPUT_DATA "testbench/DM.out"
44
45 // Declaration
```

b. Instruction Memory

IM這個module其實很簡單，只要輸入Instruction Address，接著我就到該位置去抓Instruction，因為這個系統是BIG-ENDIAN，因此我抓的順序就會是如我程式碼的方式去抓{0, 1, 2, 3}。那這個部分比較特別的是testbench，我參考了題目提供的tb_R_formatCPU，使用\$readmemh來抓資料。接著下圖為模擬之結果。

```
IM.v
29 `define INSTR_MEM_SIZE 128 // Bytes
30 `define Rtype_op 6'b000100
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 [31:0] InstrAddr
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]; // You, a day ago * put some component and is fine to design
47
48 always@ (InstrAddr)
49 begin
50     Instr[31:0] = {InstrMem[InstrAddr], InstrMem[InstrAddr+1], InstrMem[InstrAddr+2], InstrMem[InstrAddr+3]};
51 end
52
53 endmodule
```

```
tb_IM.v
1 `define INSTR_FILE "testbench/IM.dat"
2 `define INSTR_MAX 128 // bytes
3 `define INSTR_SIZE 8 // bit width
4
5
6 module tb_IM;
7
8     integer i;
9     reg [31:0] InstrAddr;
10    wire [31:0] Instr;
11    reg [`INSTR_SIZE-1 : 0] instrMem [0:`INSTR_MAX-1];
12
13    IM Instruction_Memory(
14        .InstrAddr(InstrAddr),
15        .Instr(Instr)
16    );
17
18    initial begin : Preprocess
19
20        $readmemh(`INSTR_FILE, instrMem); // put the value into instrMem
21        // Initialize intruction memory
22        for (i = 0; i < `INSTR_MAX; i = i + 1)
23        begin
24            Instruction_Memory.InstrMem[i] = instrMem[i];
25        end
26    end
27
28    initial #20 $finish;
29
30    initial fork
31        #0 InstrAddr = 8; // Instr should be 12_32_B0_12.
32        #10 InstrAddr = 2; // Instr should be A0_0B_11_AC
33    join
34
35
36 endmodule
```



1. 第一次Address = 8, Instr should be 12_32_B0_12.

2. 第二次Address = 2, Instr should be A0_0B_11_AC.

下圖為IM.dat，供此部分參考。

```

Part 3 > testbench > IM.dat
You, 4 days ago | 1 author (You)
1 // Instruction Memory in Hex
2 4C // Addr = 0x00
3 13 // Addr = 0x01
4 00 // Addr = 0x02
5 1E // Addr = 0x03
6 12 // Addr = 0x04
7 62 // Addr = 0x05
8 98 // Addr = 0x06
9 23 // Addr = 0x07
10 70 // Addr = 0x08
11 00 // Addr = 0x09
12 00 // Addr = 0x0A
13 00 // Addr = 0x0B
14 FF // Addr = 0x0C
15 FF // Addr = 0x0D
16 FF // Addr = 0x0E
17 FF // Addr = 0x0F
18 FF // Addr = 0x10
19 FF // Addr = 0x11
20 FF // Addr = 0x12
21 FF // Addr = 0x13
22 FF // Addr = 0x14
23 FF // Addr = 0x15
24 FF // Addr = 0x16
25 FF // Addr = 0x17
26 FF // Addr = 0x18
27 FF // Addr = 0x19
28 FF // Addr = 0x1A
29 FF // Addr = 0x1B
  
```

c. Register File

RM這個module其實不難，只要判斷RegWrite是否為1，來決定是否可以將值寫入Reg，那其他部分就是到Register的地址去抓值去輸出。我將RsData與RtData使用assign而非放在always裡面是因為我發現放在裡面會等到正緣觸發才把值送入ALU，那這樣的話就無法達到我們想要的效果了。

```
35 module RF(  
36     // Outputs  
37     output [31:0] RsData,  
38     output [31:0] RtData,  
39     // Inputs  
40     input RegWrite,  
41     input clk,  
42     input [4:0] RsAddr,  
43     input [4:0] RtAddr,  
44     input [4:0] RdAddr,  
45     input [31:0] RdData  
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  
55 assign RsData = R[RsAddr];  
56 assign RtData = R[RtAddr];  
57  
58 always@(posedge clk)  
59 begin  
60     if(RegWrite == 1)  
61     begin  
62         R[RdAddr] = RdData;  
63     end  
64     else  
65     begin  
66         R[RdAddr] = R[RdAddr];  
67     end  
68 end  
69  
70  
71 endmodule  
72
```

```

1  `define DELAY          5    // # * timescale
2  `define REG_SIZE      32    // bit width
3  `define REG_MAX       32    // words
4  `define REG_FILE      "testbench/RF.dat"
5
6  module tb_RF;
7
8      // Inputs
9      reg RegWrite;
10     reg clk;
11     reg [4:0] RsAddr;
12     reg [4:0] RtAddr;
13     reg [4:0] RdAddr;
14     reg [31:0] RdData;
15
16     // Outputs
17     wire [31:0] RsData;
18     wire [31:0] RtData;
19
20     integer i;
21     integer output_reg;
22     reg [`REG_SIZE-1 : 0] regMem    [0: `REG_MAX-1];
23
24     RF Register File(
25         // Outputs
26         .RsData(RsData),
27         .RtData(RtData),
28         // Inputs
29         .RegWrite(RegWrite),
30         .clk(clk),
31         .RsAddr(RsAddr),
32         .RtAddr(RtAddr),
33         .RdAddr(RdAddr),
34         .RdData(RdData)
35     );

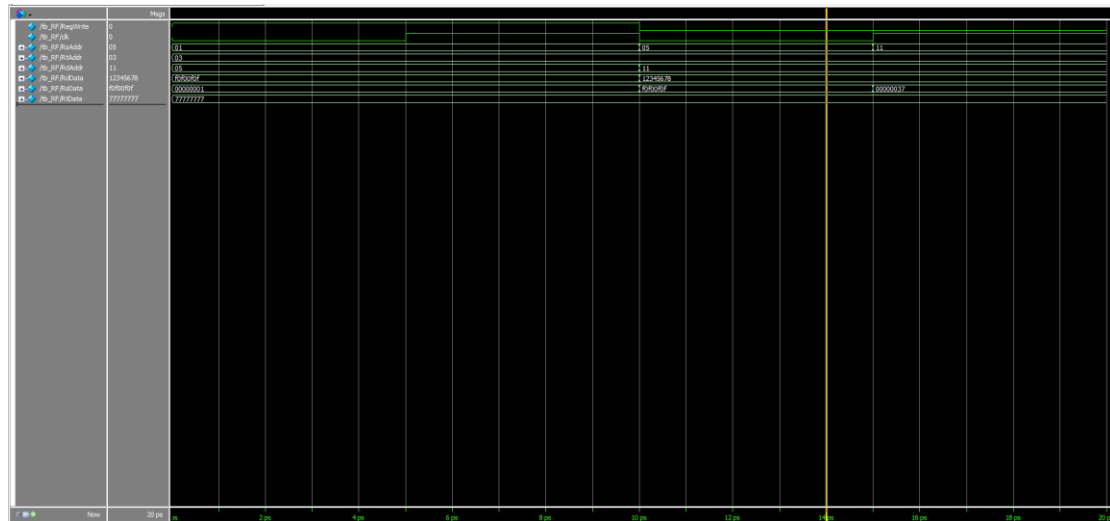
```

```

39     initial begin: Preprocess
40         // Initialize inputs
41         clk = 0;
42         RegWrite = 1;
43         RsAddr = 5'd 1; // Register R[1]
44         RtAddr = 5'd 3; // R[3]
45         RdAddr = 5'd 5; // R[5]
46         RdData = 32'h f0f0_0f0f;
47
48         $readmemh(`REG_FILE,regMem);
49
50         // Initialize register file
51         for (i = 0; i < `REG_MAX; i = i + 1)
52             begin
53                 Register_File.R[i] = regMem[i];
54             end
55     end
56
57     initial #20 $finish;
58
59
60     always begin : ClockGenerator
61         #`DELAY;
62         clk ≤ ~clk;
63     end
64
65
66     initial fork
67         #0 RegWrite = 1;
68         #0 RsAddr = 5'd 1;
69         #0 RtAddr = 5'd 3;
70         #0 RdAddr = 5'd 5;
71         #0 RdData = 32'h F0F0_0F0F;
72
73         #10 RegWrite = 0;
74         #10 RsAddr = 5'd 5; // to show where R[5] is been written or not.
75         #10 RtAddr = 5'd 3;
76         #10 RdAddr = 5'd 17;
77         #10 RdData = 32'h 1234_5678;
78         #15 RsAddr = 5'd 17; // to show where R[17] is been written or not.
79     join
80

```

而下圖這個就是模擬出來的結果，與IM相同的是，我用類似的指令去將值存入一個Reg，接著再去做模擬進行分析。



4. 一開始的時候我是設定讓值可以寫入Reg，所以我把R[5]=F0F0_0F0F
5. 在t=10後我去把RsAddr改成5(也就是第一次的Rd)，因此可以看到我第一次是否有成功寫入，而我們也可以看到RsData變成F0F0_0F0F了。那接著我將RegWrite改為0，試著把它放到R[17]。
6. 在t=15後我去把RsAddr改成17(也就是第二次的Rd)，因此可以看到我第二次因為我關掉RegWrite了，所以沒有成功寫入，可以確認RF正常工作。

```

You, 4 days ago | 1 author (You)
1 // Register File in Hex
2 0000_0000 // R[0]
3 ✓ 0000_0001 // R[1]
4 0000_0002 // R[2]
5 ✓ 7777_7777 // R[3]
6 7F7F_7F7F // R[4]
7 F7F7_F7F7 // R[5]
8 7FFF_FFFF // R[6]
9 8000_0000 // R[7]
10 FFFF_0000 // R[8]
11 0000_FFFF // R[9]
12 0000_0011 // R[10]
13 0000_0023 // R[11]
14 0000_0017 // R[12]
15 0000_0090 // R[13]
16 0000_0100 // R[14]
17 0000_0250 // R[15]
18 ✓ 0000_0300 // R[16]
19 ✓ 0000_0037 // R[17]
20 0000_0064 // R[18]
21 0000_0030 // R[19]
22 0000_0000 // R[20]
23 0000_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_0000 // R[26]
29 0000_0000 // R[27]
30 0000_0000 // R[28]
31 0000_0000 // R[29]
32 FFFF_FFFF // R[30]
33 FFFF_FFFF // R[31]

```

d. Data Memory

DM這個module其實也不難，只要判斷MemWrite跟MemRead是否為1，來決定是否可以將值寫入Memory或是把Memory的值給讀出來。我將MemReadData使用assign而非放在always裡面，與RF的原因相同。我發現放在裡面會等到下個正緣觸發才把值送出，那這樣的話就無法達到我們想要的效果了。

```
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 [31:0] MemReadData,
38     // Inputs
39     input [31:0] MemAddr,
40     input [31:0] MemWriteData,
41     input MemWrite,
42     input MemRead,
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 assign MemReadData = MemRead? {DataMem[MemAddr],DataMem[MemAddr+1],DataMem[MemAddr+2],DataMem[MemAddr+3]}:32'b0;
53 // You, a day ago + PART3 controller still working
54 always@(posedge clk)
55 begin
56     if(MemWrite == 1)
57     begin
58         {DataMem[MemAddr],DataMem[MemAddr+1],DataMem[MemAddr+2],DataMem[MemAddr+3]} = MemWriteData;
59     end
60     else;
61 end
62
63 endmodule
64
```

```

tb_DM.v 1, U X
Part 3 > tb_DM.v
1  `define DATA_SIZE      8    // bit width
2  `define DATA_MAX      128   // bytes
3  `define DATA_FILE     "testbench/DM.dat"
4  `define DELAY           5    // # * timescale
5
6  module tb_DM();
7      reg clk;
8      reg [31:0] MemAddr;
9      reg [31:0] MemWriteData;
10     reg MemWrite;
11     reg MemRead;
12     wire [31:0] MemReadData;
13     reg [`DATA_SIZE-1 :0] dataMem    [0:`DATA_MAX-1];
14     integer i;
15
16     DM_Data_Memory(
17         .MemAddr(MemAddr),
18         .MemWriteData(MemWriteData),
19         .MemWrite(MemWrite),
20         .MemRead(MemRead),
21         .clk(clk),
22         .MemReadData(MemReadData)
23     );
24

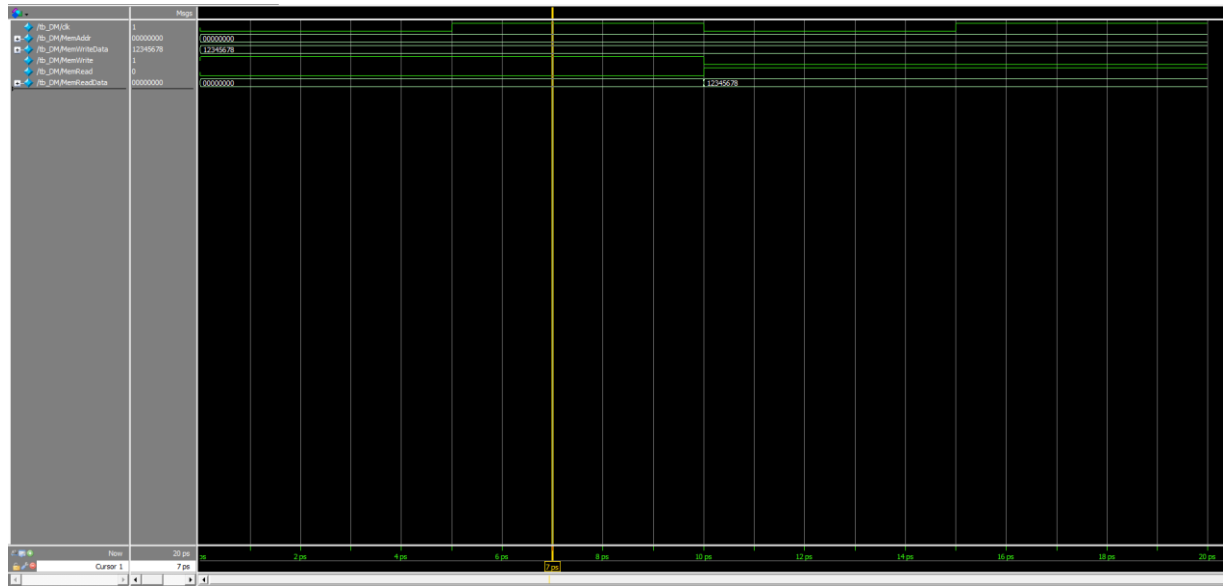
```

```

25     initial begin: Preprocess
26
27         clk = 0;
28
29         $readmemh(`DATA_FILE, dataMem);
30         // Initialize data memory
31         for (i = 0; i < `DATA_MAX; i = i + 1)
32             begin
33                 Data_Memory.DataMem[i] = dataMem[i];
34             end
35
36     end
37
38     always begin : ClockGenerator
39         #`DELAY;
40         clk ≤ ~clk;
41     end
42
43     initial #20 $finish;
44
45     initial fork
46         #0 MemWriteData = 32'h 1234_5678;
47         #0 MemAddr = 32'h 0000_0000;
48         #0 MemWrite = 1;
49         #0 MemRead = 0;
50
51         #10 MemRead = 1;
52         #10 MemWrite = 0;
53
54     join
55
56 endmodule

```

下圖這個就是由testbench模擬出來的結果。



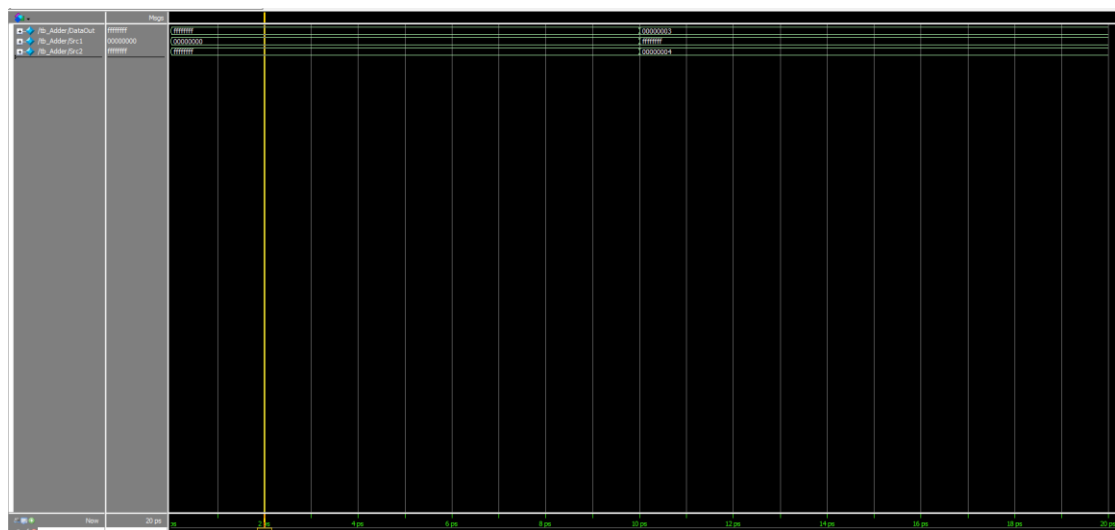
3. 一開始的時候我是設定讓值可以寫入Memory(MemWrite=1)，所以我把Memory{0, 1, 2, 3}=32'h1234_5678.
4. 在t=10後我把MemWrite關掉，把MemRead打開，Address不變，就可以發現我的MemReadData真的在上一個clk被寫入Memory了，輸出為32'h1234_5678.

e. Adder

```
Part 3 > tb_Adder.v
1 module tb_Adder;
2
3     wire [31:0] DataOut;
4     reg [31:0] Src1;
5     reg [31:0] Src2;
6
7     Adder A(
8         // Outputs
9         .DataOut(DataOut),
10        // Inputs
11        .Src1(Src1),
12        .Src2(Src2)
13    );
14
15    initial #20 $finish;
16    initial fork
17        #0 Src1 = 32'd 0;
18        #0 Src2 = 32'h FFFF_FFFF;
19
20        #10 Src1 = 32'h FFFF_FFFF;
21        #10 Src2 = 32'h 0000_0004;
22    join
23
24 endmodule

Part 3 > Adder.v
jiehong, a day ago | 1 author (jiehong)
1 module Adder
2 (
3     input [31:0] Src1,
4     input [31:0] Src2,
5     output [31:0] DataOut
6 );
7
8     assign DataOut = Src1 + Src2;
9
10 endmodule
11
```

Adder所做的事情非常簡單，因為在的DataOut僅僅需要能夠將Src1+Src2，因此我就只做了相加的動作，然後輸出。（我的加法是unsigned的加法，因為Address沒有負的。）



1. 第一次Input為FFFF_FFFF+0，輸出為FFFF_FFFF。
2. 第二次Input為4+FFFF_FFFF，輸出為3。

f. ALU

```

Part1 > ALU
1  *define addu 6'b 001001
2  *define subu 6'b 001010
3  *define AND 6'b 010001
4  *define sll 6'b 100001
5
6  module ALU(
7      input  [31:0] Src1,
8      input  [31:0] Src2,
9      input  [4:0]  Shamt,
10     input  [5:0]  Funct,
11     output reg [31:0] result,
12     output reg Zero
13 );
14
15 always@(Funct or Shamt or Src1 or Src2)
16 begin
17     case (Funct)
18         *addu : result = Src1 + Src2;
19         *subu : result = Src1 - Src2;
20         *AND  : result = Src1 & Src2;
21         *sll  : result = Src1 << Shamt;
22         default: result = result; // if Funct is not same as above, then the r
23     endcase
24     if(result == 0)
25         Zero = 1;
26     else
27         Zero = 0;
28 end
29
30 endmodule

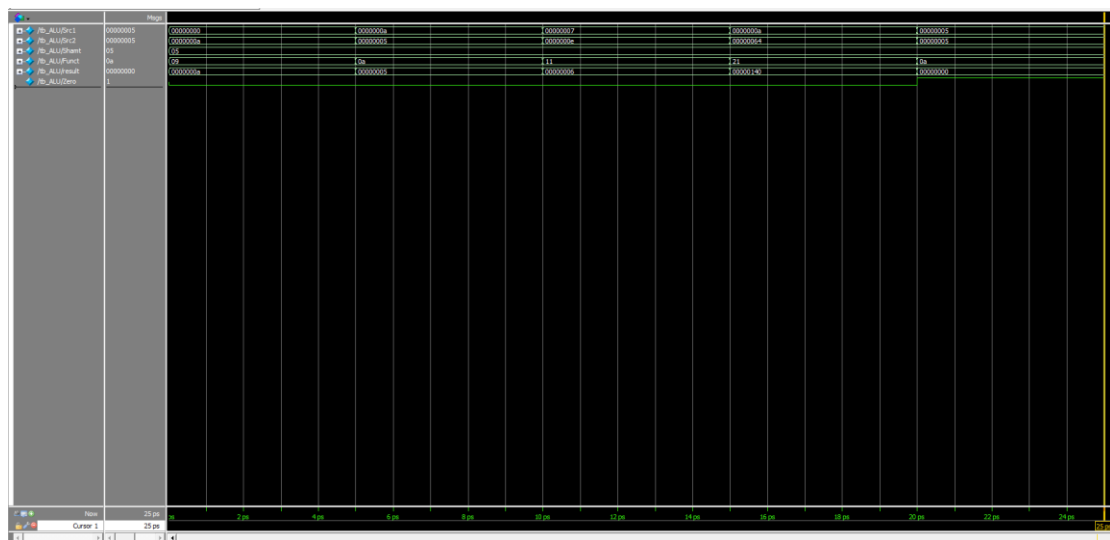
```

```

Part3 > tb_ALU
1  *define addu 6'b 001001
2  *define subu 6'b 001010
3  *define AND 6'b 010001
4  *define sll 6'b 100001
5
6  module tb_ALU;
7
8      reg  [31:0] Src1;
9      reg  [31:0] Src2;
10     reg  [4:0]  Shamt;
11     reg  [5:0]  Funct;
12     wire [31:0] result;
13     wire      Zero;
14
15     ALU AL(
16         .Src1(Src1),
17         .Src2(Src2),
18         .Shamt(Shamt),
19         .Funct(Funct),
20         .result(result),
21         .Zero(Zero)
22     );
23
24     initial #25 $finish;
25     initial fork
26         #0 Src1 = 32'd 0;
27         #0 Src2 = 32'd 10;
28         #0 Funct = *addu;
29         #0 Shamt = 5'd 5; // result should be a(hex).
30
31         #5 Src1 = 32'd 10;
32         #5 Src2 = 32'd 5;
33         #5 Funct = *subu;
34
35         #10 Src1 = 32'd 7; // 0111
36         #10 Src2 = 32'd 14; // 1110
37         #10 Funct = *AND;
38
39         #15 Src1 = 32'd 10; // 1010
40         #15 Src2 = 32'd 100; // This value would not affect the result.
41         #15 Funct = *sll; // shift left logic
42
43         #20 Src1 = 32'd 5;
44         #20 Src2 = 32'd 5;
45         #20 Funct = *subu;
46     join
47 endmodule

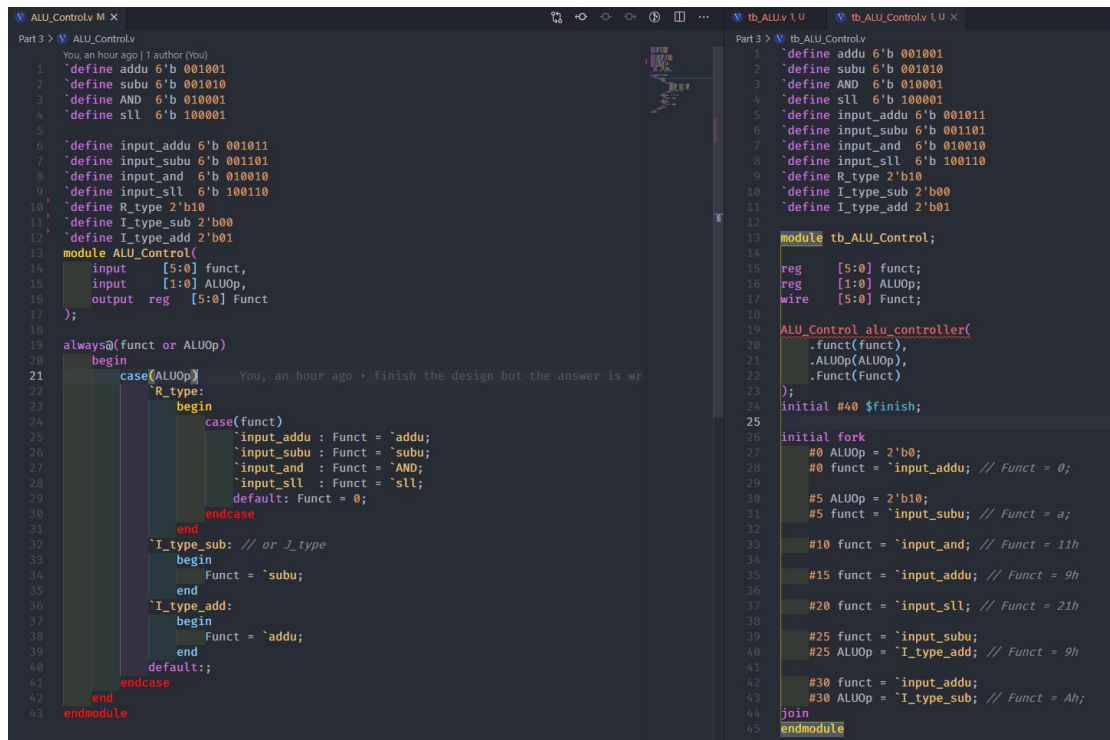
```

ALU 主要是用來做Src1 and Src2的運算，由輸入的Funct來決定要做甚麼工作。會依照結果是否為0決定zero Flag。下圖為由testbench產生之模擬結果。



1. 第一次Input為0+A，輸出為A。
2. 第二次Input為A-5，輸出為5。
3. 第三次Input為(1f)&(11)，輸出為6。
4. 第四次Input為A<<5=A*2^5=320，輸出為140。
5. 第五次Input為5-5=0，輸出為140，Zero Flag =1。

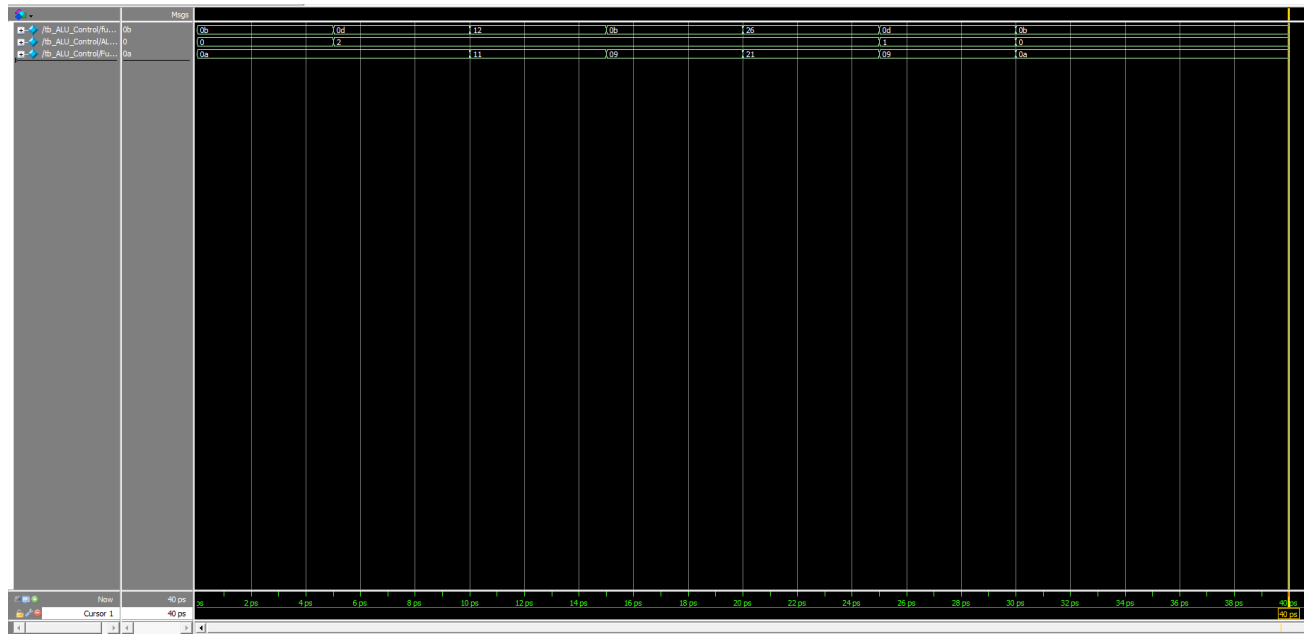
g. ALU_Control



```
Part 3 > ALU_Control.v M X
You, an hour ago | 1 author (You)
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5
6 `define input_addu 6'b 001011
7 `define input_subu 6'b 001101
8 `define input_and 6'b 010010
9 `define input_sll 6'b 100110
10 `define R_type 2'b10
11 `define I_type_sub 2'b00
12 `define I_type_add 2'b01
13 module ALU_Control(
14     input [5:0] funct,
15     input [1:0] ALUOp,
16     output reg [5:0] Funct
17 );
18
19 always@(funct or ALUOp)
20 begin
21     case(ALUOp)
22         R_type:
23             begin
24                 case(funct)
25                     'input_addu : Funct = `addu;
26                     'input_subu : Funct = `subu;
27                     'input_and : Funct = `AND;
28                     'input_sll : Funct = `sll;
29                     default: Funct = 0;
30                 endcase
31             end
32         'I_type_sub: // or J_type
33             begin
34                 Funct = `subu;
35             end
36         'I_type_add:
37             begin
38                 Funct = `addu;
39             end
40         default:;
41     endcase
42 end
43 endmodule

Part 3 > tb_ALU_Control.v
1 `define addu 6'b 001001
2 `define subu 6'b 001010
3 `define AND 6'b 010001
4 `define sll 6'b 100001
5 `define input_addu 6'b 001011
6 `define input_subu 6'b 001101
7 `define input_and 6'b 010010
8 `define input_sll 6'b 100110
9 `define R_type 2'b10
10 `define I_type_sub 2'b00
11 `define I_type_add 2'b01
12
13 module tb_ALU_Control;
14
15 reg [5:0] funct;
16 reg [1:0] ALUOp;
17 wire [5:0] Funct;
18
19 ALU_Control alu_controller(
20     .funct(funct),
21     .ALUOp(ALUOp),
22     .Funct(Funct)
23 );
24 initial #40 $finish;
25
26 initial fork
27     #0 ALUOp = 2'b0;
28     #0 funct = `input_addu; // Funct = 0;
29
30     #5 ALUOp = 2'b10;
31     #5 funct = `input_subu; // Funct = a;
32
33     #10 funct = `input_and; // Funct = 11h
34
35     #15 funct = `input_addu; // Funct = 9h
36
37     #20 funct = `input_sll; // Funct = 21h
38
39     #25 funct = `input_subu;
40     #25 ALUOp = `I_type_add; // Funct = 9h
41
42     #30 funct = `input_addu;
43     #30 ALUOp = `I_type_sub; // Funct = Ah;
44 join
45 endmodule
```

ALU_Control主要是用來控制ALU工作與否，及將instr的指令轉為ALU可以理解的function code。而在I-type跟J-type指令下，吃的主要就是ALUOp，由ALUOp來決定他們的function out. 下圖為TestBench之模擬結果。



1. 第一次ALUOP為0，ALU不可以工作，Funct輸出為0。
2. 第二次開始ALUOP為2'b10，ALU可以工作，Funct輸出為Instr對應的subu。
3. 第三次Input為input_and，輸出為Instr對應的AND。
4. 第四次Input為input_sll，輸出為Instr對應的sll。
5. 第五次Input為input_addu，輸出為Instr對應的addu。
6. 第六次ALUOp為 I-type-add, funct Input為input_subu，但是在I-type指令下，ALU不會理會funct, 因此輸出為Instr對應的addu.
7. 第七次ALUOp為 I-type-sub, funct Input為input_addu，但是在I-type指令下，ALU不會理會funct, 因此輸出為Instr對應的subu.

h. Control

Control這個module在整個CPU裡面是一個非常重要的角色。他掌管整顆CPU現在要做甚麼，不要做甚麼。雖然他極其重要，但是其實沒有甚麼太複雜的工作，只要依照Opcode的要求，來決定是否要送各種訊號。而因為R-type的所有指令都會使用到ALU，因此我們ALUOp只要是對的Opcode，我們一律送2'b10. 而對於I-type指令來說，只會有加法跟減法，因此除了subiu以外(2'b00)，其他的ALUOP都為2'b01，剩下的是J-type指令，因為branch用到的是減法，因此ALUOP為0。

```
You, 28 minutes ago | 1 author (You)
1 // I-type sub
2 `define I_type_sub 2'b00
3 // I-type add
4 `define I_type_add 2'b01
5 module Control(
6     input    [5:0]    OpCode,
7     output reg         RegWrite,
8     output reg  [1:0] ALUOp,
9     output reg         RegDst,
10    output reg         ALUSrc,
11    output reg         MemWrite, // write memory or not.
12    output reg         MemRead,
13    output reg         MemtoReg,
14    output reg         Jump,
15    output reg         Branch
16 );
17
18 always@(OpCode)
19 begin
20     case (OpCode)
21     6'd 4:
22     begin
23         RegWrite = 1'b 1; // R format.
24         ALUOp = 2'b 10;
25         RegDst = 1; // R format.
26         ALUSrc = 0;
27         MemWrite = 0;
28         MemRead = 0;
29         MemtoReg = 0;
30         Jump = 0;
31         Branch = 0;
32     end
33     // I format.
```

```

33 // I format.
34 6'd 12: // addiu
35     begin
36         RegWrite = 1'b 1;
37         ALUOp = `I_type_add;
38         RegDst = 0; // I format → write into Rt
39         ALUSrc = 1;
40         MemWrite = 0;
41         MemRead = 0;
42         MemtoReg = 0;
43         Jump = 0;
44         Branch = 0;
45     end
46 6'd 13: // subiu
47     begin
48         RegWrite = 1'b 1;
49         ALUOp = `I_type_sub;
50         RegDst = 0; // I format → write into Rt
51         ALUSrc = 1;
52         MemWrite = 0;
53         MemRead = 0;
54         MemtoReg = 0;
55         Jump = 0;
56         Branch = 0;
57     end
58 6'd 16: // sw
59     begin // You, a day ago • PART3 controller still working
60         RegWrite = 1'b 0;
61         ALUOp = `I_type_add;
62         // RegDst = 1'b x; // I format → write into Rt
63         ALUSrc = 1;
64         MemWrite = 1;
65         MemRead = 0;
66         Jump = 0;
67         Branch = 0;
68         // MemtoReg = 1'b x; // Since the SW would not read the value
69     end

```

```

70     end
71 6'd 17: // lw
72     begin
73         RegWrite = 1'b 1;
74         ALUOp = `I_type_add;
75         RegDst = 0; // I format → write into Rt
76         ALUSrc = 1;
77         MemWrite = 0;
78         MemRead = 1;
79         MemtoReg = 1;
80         Jump = 0;
81         Branch = 0;
82     end
83 6'd 19: // beq
84     begin
85         ALUOp = 2'b00;
86         RegDst = 1'bx;
87         Jump = 0;
88         Branch = 1;
89         RegWrite = 0;
90         ALUSrc = 0;
91         MemWrite = 0;
92         MemRead = 0;
93         MemtoReg = 1'bx;
94     end
95 6'd 28: // j
96     begin // You, a day ago • PART3 controller still working
97         ALUOp = 2'b00;
98         RegDst = 1'b x;
99         Jump = 1;
100        Branch = 0;
101        RegWrite = 0;
102        ALUSrc = 1;
103        MemWrite = 0;
104        MemRead = 0;
105        MemtoReg = 0;
106    end
107 default:
108     begin
109         MemWrite = 0;
110         RegWrite = 0;
111         Jump = 0;
112         Branch = 0;
113     end
114 endcase
115 end
endmodule

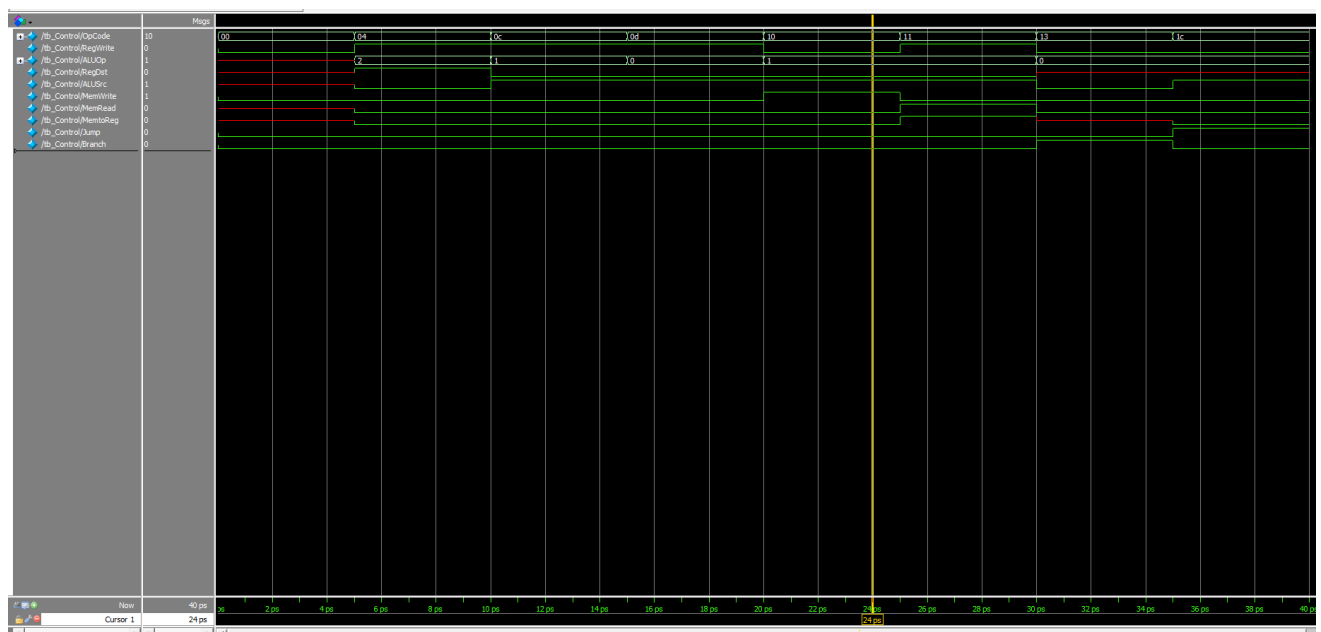
```

```

Part 3 > tb_Control.v
1  module tb_Control;
2
3      reg    [5:0] OpCode;
4      wire   RegWrite;
5      wire   [1:0] ALUOp;
6      wire   RegDst;
7      wire   ALUSrc;
8      wire   MemWrite;
9      wire   MemRead;
10     wire   MemtoReg;
11     wire   Jump;
12     wire   Branch;
13
14     Control_controller(
15         .OpCode(OpCode),
16         .RegWrite(RegWrite),
17         .ALUOp(ALUOp),
18         .RegDst(RegDst),
19         .ALUSrc(ALUSrc),
20         .MemWrite(MemWrite),
21         .MemRead(MemRead),
22         .MemtoReg(MemtoReg),
23         .Jump(Jump),
24         .Branch(Branch)
25     );
26
27     initial #40 $finish;
28
29     initial fork
30         #0 OpCode = 6'd 0; // Not working
31
32         #5 OpCode = 6'd 4;
33
34         #10 OpCode = 6'd 12;
35
36         #15 OpCode = 6'd 13;
37
38         #20 OpCode = 6'd 16;
39
40         #25 OpCode = 6'd 17;
41
42         #30 OpCode = 6'd 19;
43
44         #35 OpCode = 6'd 28;
45     join
46 endmodule

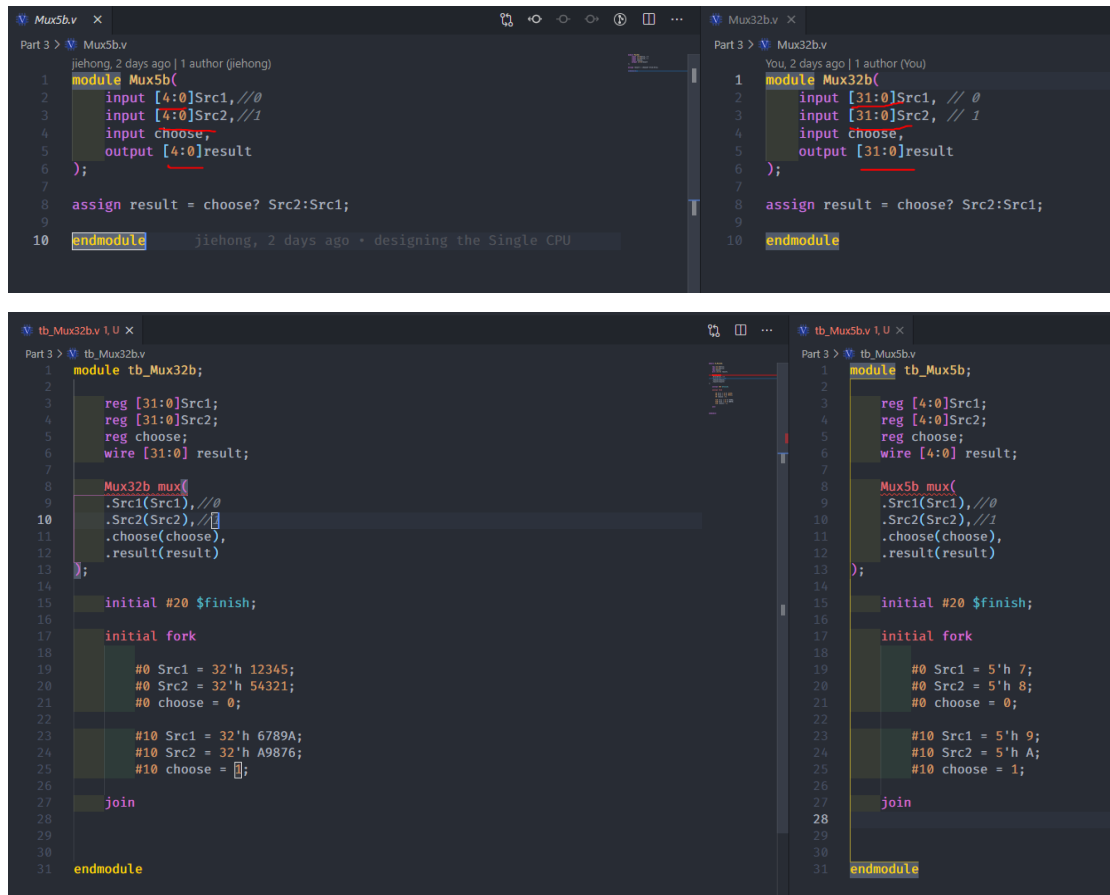
```

由tb_Control.v這個testbench產生之模擬結果，由於input與Control之case順序相同，輸出也皆相同，故不特別列出。



i. MUX

MUX這二個module其實非常簡單，一個是5bit, 一個是32bit. 只要判斷choose選的是多少，就決定輸出要送哪一個輸入出來。



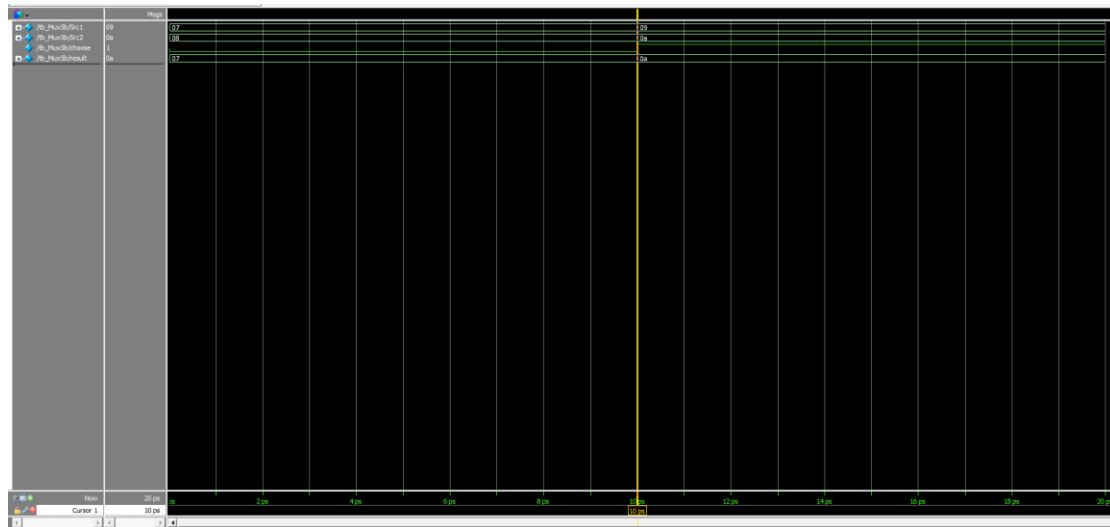
```
Part 3 > Mux5b.v
1 jiehong, 2 days ago | 1 author (jiehong)
2 module Mux5b(
3     input [4:0]Src1,//0
4     input [4:0]Src2,//1
5     input choose,
6     output [4:0]result
7 );
8 assign result = choose? Src2:Src1;
9
10 endmodule jiehong, 2 days ago • designing the Single CPU

Part 3 > Mux32b.v
1 You, 2 days ago | 1 author (You)
2 module Mux32b(
3     input [31:0]Src1, // 0
4     input [31:0]Src2, // 1
5     input choose,
6     output [31:0]result
7 );
8 assign result = choose? Src2:Src1;
9
10 endmodule

Part 3 > tb_Mux32b.v
1 module tb_Mux32b;
2
3     reg [31:0]Src1;
4     reg [31:0]Src2;
5     reg choose;
6     wire [31:0] result;
7
8     Mux32b mux(
9         .Src1(Src1),//0
10        .Src2(Src2),//1
11        .choose(choose),
12        .result(result)
13    );
14
15    initial #20 $finish;
16
17    initial fork
18
19        #0 Src1 = 32'h 12345;
20        #0 Src2 = 32'h 54321;
21        #0 choose = 0;
22
23        #10 Src1 = 32'h 6789A;
24        #10 Src2 = 32'h A9876;
25        #10 choose = 1;
26
27    join
28
29
30
31 endmodule

Part 3 > tb_Mux5b.v
1 module tb_Mux5b;
2
3     reg [4:0]Src1;
4     reg [4:0]Src2;
5     reg choose;
6     wire [4:0] result;
7
8     Mux5b mux(
9         .Src1(Src1),//0
10        .Src2(Src2),//1
11        .choose(choose),
12        .result(result)
13    );
14
15    initial #20 $finish;
16
17    initial fork
18
19        #0 Src1 = 5'h 7;
20        #0 Src2 = 5'h 8;
21        #0 choose = 0;
22
23        #10 Src1 = 5'h 9;
24        #10 Src2 = 5'h A;
25        #10 choose = 1;
26
27    join
28
29
30
31 endmodule
```

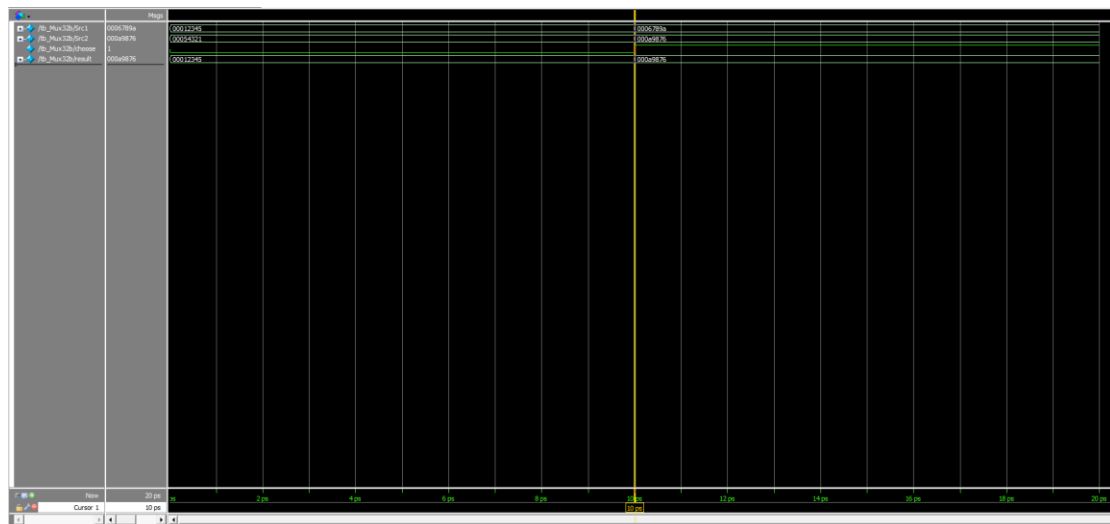
下圖MUX5b這個testbench模擬出來的結果。只要判斷choose選的是多少，就決定輸出要送哪一個輸入出來。



3. 一開始的時候Src1 = 7, Src2 = 8, choose = 0, 因此result會選擇Src1, 輸出為07.

4. T=10的時候Src1 = 9, Src2 = A, choose = 1, 因此result會選擇Src2, 輸出為0A.

下圖MUX32b這個testbench模擬出來的結果。只要判斷choose選的是多少，就決定輸出要送哪一個輸入出來。



3. 一開始的時候Src1 = 12345, Src2 = 54321, choose = 0, 因此result會選擇Src1, 輸出為12345.

4. T=10的時候Src1 = 6789A, Src2 = A9876, choose = 1, 因此result會選擇Src2, 輸出為A9876.

4. 作業總結與心得

這次的作業，從4/28星期三提早到4/27星期二出了。幸運的是，因為疫情的關係，整間學校停課一周，讓我原本非常排滿滿的生活，有了可以喘息的空間，因此這次的作業，因為不用去學校上課，我從4/27星期二晚上開始思考如何做起，大概做到5/2星期天，大約是4~5天的時間，每天大概花4~6個小時左右處理這個作業，途中有遇到非常令人不知所云的BUG，也花了很多時間在理解BUG的由來，在理解到BUG是怎麼出現的以後再去思考要怎麼DEBUG. 結果在反覆測試後及與同學討論後想到了各自不同的解決方法，且都可以順利執行，真的非常開心。

不過也有一次，做到半夜凌晨三點，不知為何，臭蟲打不死的經驗，只好熄燈去睡覺。隔天一早問了同學，結果他聽我的敘述就知道我的bug是因為在controller沒有做好default的處理，一分鐘不到就解決了。這也讓我體會到經驗與討論的重要，有時苦幹一整天，不如與朋友討論。這次的經驗也讓我深深知道default處理的重要，以後再設計case的時候就會更加注意了。

最後我覺得最重要的是，比起急著開始打程式，還不如先想好要怎麼設計，真的確定好邏輯是對的，都比一開始就急著打但是毫無頭緒還快。比起上次，這次我有先規劃再進行實作，比起上次的速度真的快了非常多，希望以後這堂課的不論是考試或是project，也可以像這次一樣順利度過。