



國立台灣科技大學
電機工程系

計算機組織

(EE3005301)

HW2

班級：四電機三甲

學號：B10507004

指導老師：陳雅淑

姓名：游照臨

中華民國 108 年 5 月 20 日

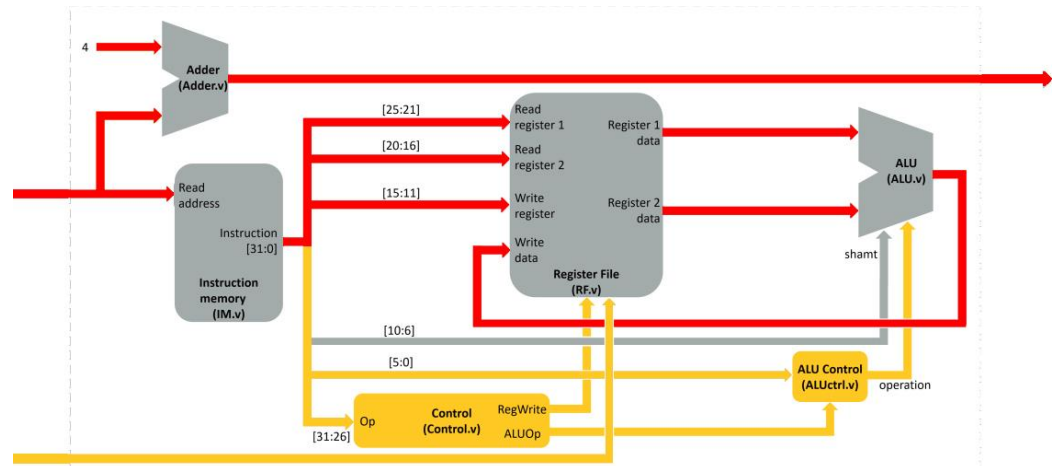
壹、階段 1

一、程式功能

(一) ADD 指令

1. ADD 指令位於 Instruction Memory 的第 0 位置，其指令為
add \$t0, \$t1,\$t2
2. 在 im 中將其指令轉換為 instruction 的 binary 為

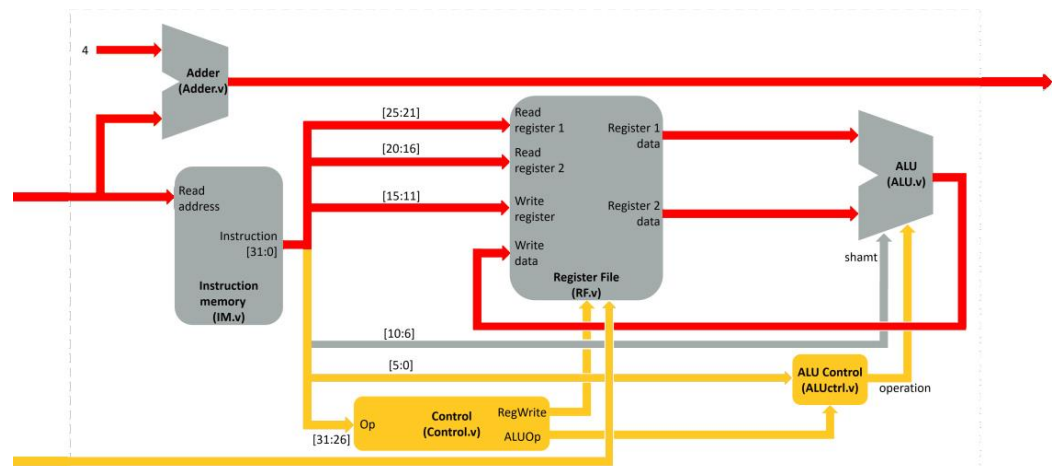
```
Instr[0] = {6'd54,5'd9,5'd10,5'd8,5'd0,6'd21};
```
3. Data(紅)/Control(黃) Path



(二) SUB 指令

1. SUB 指令位於 Instruction Memory 的第 4 位置，其指令為
sub \$t1, \$t1, \$t2
2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[1] = {6'd54,5'd9,5'd10,5'd9,5'd0,6'd22};
```
3. Data(紅)/Control(黃) Path



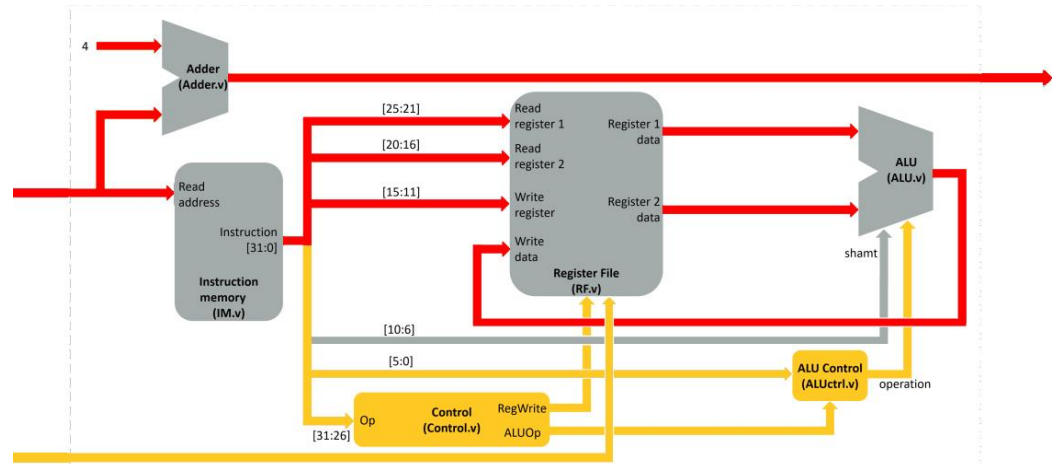
(三) AND 指令

1. AND 指令位於 Instruction Memory 的第 8 位置，其指令為
and \$t3, \$t5, \$t3

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[2] = {6'd54, 5'd13, 5'd11, 5'd11, 5'd0, 6'd23};
```

3. Data(紅)/Control(黃) Path



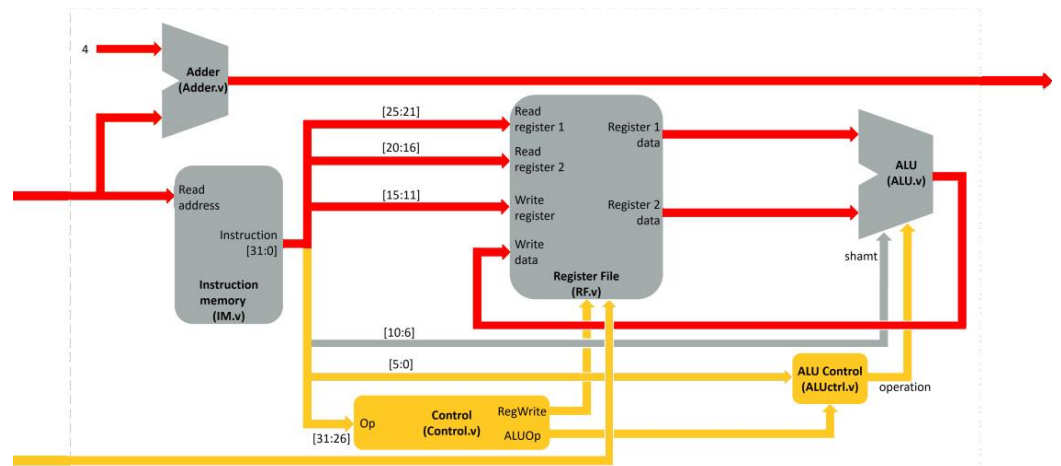
(四) OR 指令

1. OR 指令位於 Instruction Memory 的第 12 位置，其指令為
or \$t5, \$t3, \$t5

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[3] = {6'd54, 5'd11, 5'd13, 5'd13, 5'd0, 6'd24};
```

3. Data(紅)/Control(黃) Path



(五) SHR 指令

1. SHR 指令位於 Instruction Memory 的第 16 位置，其指令為
shr \$t3, \$t4, 2

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[4] = {6'd54, 5'd0, 5'd12, 5'd11, 5'd2, 6'd25};
```

(六) SHL 指令

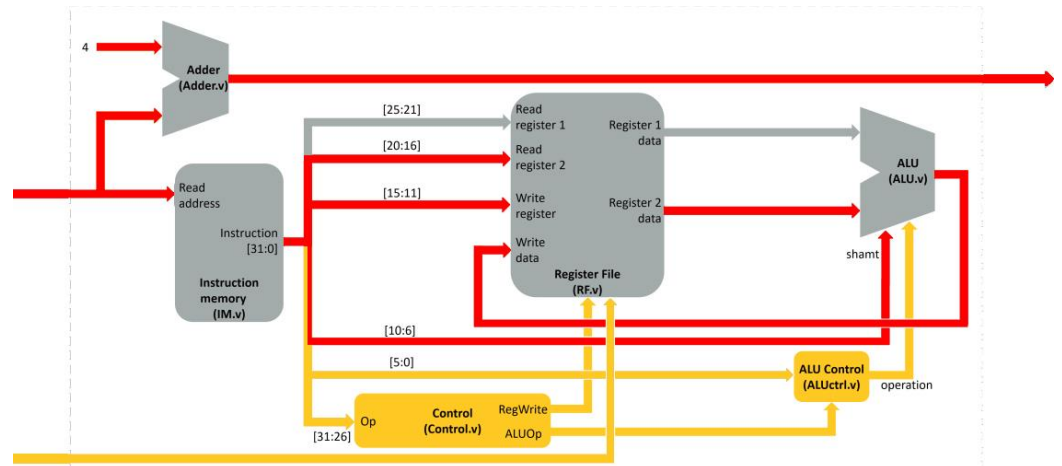
1. SHL 指令位於 Instruction Memory 的第 20 位置，其指令為

shl \$t5, \$t7, 5

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[5] = {6'd54, 5'd0, 5'd15, 5'd13, 5'd5, 6'd26};
```

3. Data(紅)/Control(黃) Path



二、程式碼

(一) Adder.v

1. 程式碼

(1)程式碼

```
1. module Adder(  
2.     input [31:0] data1,  
3.     input [31:0] data2,  
4.     output [31:0] data_o  
5. );  
6.  
7. assign data_o = data1 + data2;  
8.  
9. endmodule
```

(2)解釋

Adder 模組為一個 32bit 輸入，32bit 輸出的雙輸入、單輸出加法器，其實現方式為使用 Verilog 中的 Data Flow Model。

2. TestBench

(1)程式碼

```
1. module tb_Adder;  
2.     reg [31:0] data1;
```

```

3. reg [31:0] data2;
4. wire [31:0] data_o;
5. integer i;
6.
7. Adder uut(
8.     data1,
9.     data2,
10.    data_o
11.);
12.
13.initial begin
14.for(i = 0; i<31;i=i+1)begin
15.    #10
16.    data1 = i;
17.    data2 = i*10;
18.end
19.end
20.endmodule

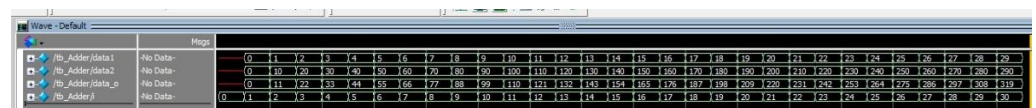
```

(2)解釋

透過一個 for 迴圈，使第一個輸入資料為 i，第二個資料為 i*10 依照迴圈依序送入 Adder 中進行加法的動作。

3. 模擬結果

(1)模擬結果



(2)模擬結果分析

根據模擬結果可以觀察到，輸出值確實等於兩個輸入值相加，因此程式功能正確。

(二) ALU.v

1. 程式碼

(1)程式碼

```

1. module ALU(input [4:0] shamt,
2.             input [31:0] src1,
3.             input [31:0] src2,
4.             input [5:0] operation,
5.             output reg [31:0] result,

```

```

6.         output zero);
7.
8.  /*
9.   add 27
10.  sub 28
11.  and 29
12.  or 30
13.  srl 31
14.  sll 32
15.  */
16.
17. assign zero = result==0?1:0;
18.
19. always@(*)begin
20.     case(operation)
21.         6'd27:
22.             result <= src1 + src2;
23.         6'd28:
24.             result <= src1 - src2;
25.         6'd29:
26.             result <= src1 & src2;
27.         6'd30:
28.             result <= src1 | src2;
29.         6'd31:
30.             result <= src2 >> shamt;
31.         6'd32:
32.             result <= src2 << shamt;
33.         default:
34.             result <= 0;
35.     endcase
36. end
37. endmodule

```

(2)解釋

透過 ALU 的 OP Code 定義，定義出 27 為加法；28 為減法；29 為邏輯與運算；30 為邏輯或運算；31 為位元右移；32 為位元左移，並且當計算結果為 0 時，zero 腳位需要輸出高電位。

2. TestBench

(1)程式碼

```
1. module tb_ALU;
2.
3. reg [4:0] shamt;
4. reg [31:0] src1;
5. reg [31:0] src2;
6. reg [5:0] operation;
7. wire [31:0] result;
8. wire zero;
9.
10. ALU uut(
11.     shamt,
12.     src1,
13.     src2,
14.     operation,
15.     result,
16.     zero
17. );
18.
19. initial begin
20. src1 = 1;
21. src2 = 3;
22. operation = 27;
23. //1+3
24.
25. #10
26. src1 = 87;
27. src2 = 87;
28. operation = 28;
29. //87-87 = 0, zero = 1
30.
31. #10
32. src1 = 123;
33. src2 = 456;
34. operation = 29;
35. // 123 & 456 = 72
36.
37. #10
```

```
38.operation = 30;
39.// 123 | 546 = 507
40.
41.#10
42.src1 = 10;
43.shamt = 3;
44.operation = 31;
45.// 10 >> 3 = 1;
46.
47.// #10
48.operation = 32;
49.//0 << 3 = 80
50.
51.#10
52.operation = 0;
53.end
54.
55.endmodule
```



```

4.     output reg[5:0] operation
5. );
6.
7. always@(funct or ALUOp)begin
8.     if(ALUOp == 3'b010)begin
9.         case(funct)
10.            6'd21:
11.                operation = 27;
12.            6'd22:
13.                operation = 28;
14.            6'd23:
15.                operation = 29;
16.            6'd24:
17.                operation = 30;
18.            6'd25:
19.                operation = 31;
20.            6'd26:
21.                operation = 32;
22.        endcase
23.    end
24. end
25.
26. endmodule

```

(2)解釋

此程式碼的主要功能為，將 funct 與 ALUOp 兩者的輸入，轉換為 ALU 的 Operation，在階段 1 時 ALUOp 一定會等於 3'b010，因此只須考慮 funct 為 21 至 26 的狀況，依照題本第 3 至 4 頁的對照，將指令對應到指定的 ALU opcode。

2. TestBench

(1)程式碼

```

1. module tb_ALUctrl;
2. reg [5:0] funct;
3. reg [2:0] ALUOp;
4. wire [5:0] operation;
5. integer i;
6.

```

```

7. ALUctrl uut(
8.     funct,
9.     ALUOp,
10.    operation
11.);
12.
13.initial begin
14.    ALUOp = 3'b010;
15.    for(i=21;i<27;i=i+1)begin
16.        #10
17.        funct = i;
18.    end
19.    #10
20.    ALUOp = 3'b000;
21.end
22.
23.endmodule

```

(2)解釋

此 TestBench 透過迴圈的方式，依序測試輸入之 funct 為 21 至 26 時的 ALUOp 是否正確。

3. 模擬結果

(1)模擬結果

| Msgs | | | 21 | 22 | 23 | 24 | 25 | 26 | |
|-----------------------|-----------|----|----|----|----|----|----|----|--|
| /tb_ALUctrl/funct | -No Data- | 2 | | | | | | | |
| /tb_ALUctrl/ALUOp | -No Data- | | | | | | | | |
| /tb_ALUctrl/operation | -No Data- | 21 | | | | | | | |
| /tb_ALUctrl/i | -No Data- | 21 | 22 | 23 | 24 | 25 | 26 | 27 | |

(2)模擬結果分析

經測試，當此模組接收到指定的 funct 時，會回傳正確的 ALU OP Code。

(四) Control.v

1. 程式碼

(1)程式碼

```

1. module Control(
2.     input [5:0]Op,
3.     output reg RegWrite,
4.     output reg [2:0]ALUOp
5. );
6.

```

```

7.  always@(Op)begin
8.      RegWrite <= 1;
9.      case(Op)
10.         6'd54:
11.             ALUOp <= 3'b010;
12.         default:
13.             ALUOp <= 3'b000;
14.
15.     endcase
16.
17. end
18.
19. endmodule

```

(2)解釋

在 Part1 時，接收到的 instruction 只會為 54，而且 RegWrite 可以永遠設定為 1，因此在 Control 輸出的 ALUOp 只需要固定為 3'b010 即可。

2. TestBench

(1)程式碼

```

1. module tb_Control;
2. reg [5:0]Op;
3. wire RegWrite;
4. wire [2:0]ALUOp;
5. integer i;
6.
7. Control uut(
8.     Op,
9.     RegWrite,
10.    ALUOp
11.);
12.
13. initial begin
14.     for(i=50;i<60;i=i+1)begin
15.         #10
16.         Op = i;
17.     end
18. end
19.

```

```
20.endmodule
```

(2)解釋

本 TestBench 透過迴圈測試當 Op Code 為 50 至 59，是否 Control 模組皆會回傳正確的 ALU Op。

3. 模擬結果

(1)模擬結果

| | | Msgs | | | | | | | | | | | |
|----------------------|----------|------|----|----|----|----|----|----|----|----|----|--|--|
| /tb_Control/Op | No Data- | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | | | |
| /tb_Control/RegWrite | No Data- | | | | | | | | | | | | |
| /tb_Control/ALUOp | No Data- | 0 | | | | 2 | 0 | | | | | | |
| /tb_Control/ | No Data- | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | | |

(2)模擬結果分析

透過觀察測試結果，可以發現 ALU Op 在正確的 54 時會輸出 3'b010，其餘狀況下皆會輸出 0，因此測試正確。

(五) IM.v

1. 程式碼

(1)程式碼

```
1. module IM(input [31:0] Addr_in,
2.           output reg [31:0] instr);
3.
4.   reg [31:0] Instr[31:0];
5.
6.   initial begin
7.       Instr[0] = {6'd54,5'd9,5'd10,5'd8,5'd0,6'd21};
8.       Instr[1] = {6'd54,5'd9,5'd10,5'd9,5'd0,6'd22};
9.
10.      Instr[2] = {6'd54,5'd13,5'd11,5'd11,5'd0,6'd23};
11.      Instr[3] = {6'd54,5'd11,5'd13,5'd13,5'd0,6'd24};
12.      Instr[4] = {6'd54,5'd0,5'd12,5'd11,5'd2,6'd25};
13.      Instr[5] = {6'd54,5'd0,5'd15,5'd13,5'd5,6'd26};
14.      instr = Instr[0];
15.   end
16.
17.   always@(Addr_in)begin
18.       instr <= Instr[Addr_in>>2];
19.   end
20.
21.endmodule
```

(2)解釋

如本文(一)所示，將組合語言翻譯為機器語言後，把指定的 Instruction memory address 除以四(右移兩格)後，把指令給輸出。

2. TestBench

(1)程式碼

```
1. module tb_IM;
2. reg [31:0] Addr_in;
3. wire [31:0] instr;
4. integer i;
5.
6. IM uut(
7.     Addr_in,
8.     instr
9. );
10.
11. initial begin
12.     for(i=0;i<=20;i=i+4)begin
13.         #10
14.         Addr_in = i;
15.     end
16.     #10
17.     Addr_in = 0;
18.
19. end
20.
21. endmodule
```

(2)解釋

本 Testbench 透過 for 迴圈依序放入不同的 Address，測試 IM 的輸出是否為正確的值。

3. 模擬結果

(1)模擬結果

| | | Mips | | | | | | | | | |
|----------------|----------|------------|------------|------------|------------|------------|------------|----|--|--|--|
| | | 0 | 4 | 8 | 12 | 16 | 20 | | | | |
| /tb_IM/Addr_in | No Data- | | | | | | | | | | |
| /tb_IM/instr | No Data- | 3643424789 | 3643426838 | 3651885079 | 3647825944 | 3624687769 | 3624888666 | | | | |
| /tb_IM/i | No Data- | 0 | 4 | 8 | 12 | 16 | 20 | 24 | | | |

(2)模擬結果分析

經由模擬結果可以觀察到，輸入指定的 instruction address，皆可以讀取到正確的 instruction。

(六) RF.v

1. 程式碼

(1)程式碼

```
1. module RF(input clk,
2.             input RegWrite,
3.             input [4:0] RSaddr,
4.             input [4:0] RTaddr,
5.             input [4:0] RDaddr,
6.             input [31:0] RDdata,
7.             output reg [31:0] RTdata,
8.             output reg [31:0] src1);
9.
10.
11.     reg [31:0] REGISTER[31:0];
12.
13.     initial begin
14.         REGISTER[0] = 32'd0;
15.         REGISTER[1] = 32'd11;
16.         REGISTER[2] = 32'd370;
17.         REGISTER[3] = 32'd183;
18.         REGISTER[4] = 32'd91;
19.         REGISTER[5] = 32'd234;
20.         REGISTER[6] = 32'd53;
21.         REGISTER[7] = 32'd127;
22.         REGISTER[8] = 32'd317;
23.         REGISTER[9] = 32'd179;
24.         REGISTER[10] = 32'd101;
25.         REGISTER[11] = 32'd161;
26.         REGISTER[12] = 32'd152;
27.         REGISTER[13] = 32'd39;
28.         REGISTER[14] = 32'd39;
29.         REGISTER[15] = 32'd44;
30.         REGISTER[16] = 32'd29;
31.         REGISTER[17] = 32'd334;
32.         REGISTER[18] = 32'd245;
```

```

33.     REGISTER[19] = 32'd19;
34.     REGISTER[20] = 32'd2;
35.     REGISTER[21] = 32'd13;
36.     REGISTER[22] = 32'd262;
37.     REGISTER[23] = 32'd185;
38.     REGISTER[24] = 32'd180;
39.     REGISTER[25] = 32'd180;
40.     REGISTER[26] = 32'd198;
41.     REGISTER[27] = 32'd178;
42.     REGISTER[28] = 32'd235;
43.     REGISTER[29] = 32'd22;
44.     REGISTER[30] = 32'd1000;
45.     REGISTER[31] = 32'd75;
46. end
47.
48. always@(posedge clk or RSaddr or RTaddr or RDaddr)begin
49.     RTdata <= REGISTER[RTaddr];
50.     src1   <= REGISTER[RSaddr];
51. end
52.
53. always@(negedge clk)begin
54.     if (RegWrite)begin
55.         REGISTER[RDaddr] <= RDdata;
56.     end
57. end
58. endmodule

```

(2)解釋

透過本模組，可以對佔存器進行讀取或是寫入的動作，當信號為正緣 clk，或是 address 變動時，會進行讀取的動作；當信號為負緣 clk，則進行寫入的動作

2. TestBench

(1)程式碼

```

1. module tb_RF;
2.
3. // RF Inputs
4. reg  clk = 0 ;
5. reg  RegWrite = 0 ;
6. reg  [4:0] RSaddr = 0 ;

```

```

7. reg [4:0] RTaddr = 0 ;
8. reg [4:0] RDaddr = 0 ;
9. reg [31:0] RDdata = 0 ;
10.
11. // RF Outputs
12. wire [31:0] RTdata;
13. wire [31:0] srcl;
14. integer i;
15.
16. RF u_RF(
17.     clk,
18.     RegWrite,
19.     RSaddr,
20.     RTaddr,
21.     RDaddr,
22.     RDdata,
23.     RTdata,
24.     srcl
25. );
26.
27. initial
28. begin
29.     clk = 1;
30.     for(i=0;i<32;i=i+1)begin
31.         #10
32.         clk = !clk;
33.         #10
34.         clk = !clk;
35.         RSaddr = i;
36.         RTaddr = 31-i;
37.     end
38.
39.     RegWrite = 1;
40.     for(i=0;i<32;i=i+1)begin
41.         #10
42.         clk = !clk;
43.         #10
44.         clk = !clk;

```



```

45.    RDaddr = i;
46.    RDdata = i*10;
47.    end
48.
49.    for(i=0;i<32;i=i+1)begin
50.        #10
51.        clk = !clk;
52.        #10
53.        clk = !clk;
54.        RSaddr = i;
55.        end
56.
57. end
58.
59. endmodule

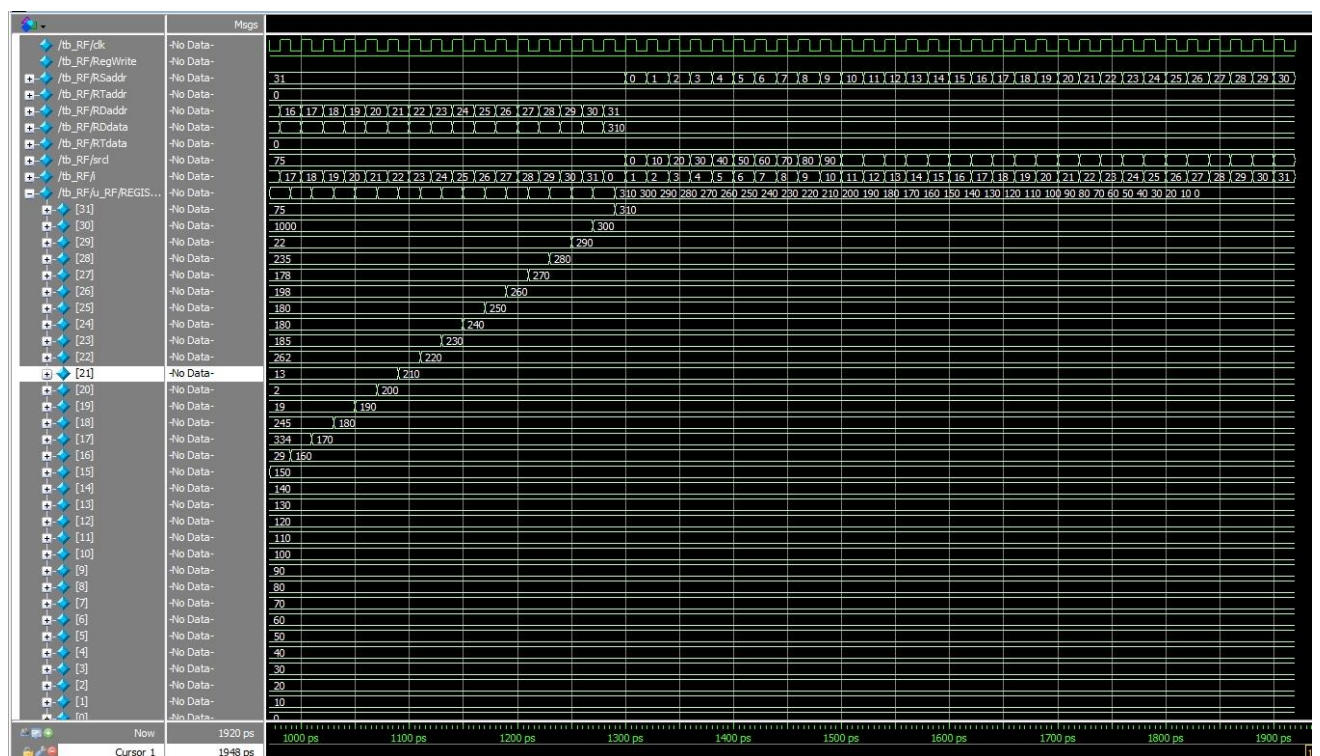
```

(2)解釋

透過 for 迴圈測試對於站存器的讀取以及寫入動作。

3. 模擬結果

(1)模擬結果



(2)模擬結果分析

經過 Testbench，可以觀察到無論是讀取或是寫入的動作，RF 皆能正常運作。

(七) SingleCPU.v

1. 程式碼

(1)程式碼

```
1. module SingleCPU(input [31:0] Addr_in,
2.                  input clk,
3.                  output [31:0] Addr_o
4.                  // output [31:0] ALU_out,
5.                  // output [5:0]operation,
6.                  // output [31:0] RTdata,
7.                  // output [31:0] srcl,
8.                  // output [31:0] instruction
9.                  );
10.
11.   wire [31:0]instruction;
12.   wire [31:0]ALU_out;
13.
14.   wire [31:0] RTdata;
15.   wire [31:0] srcl;
16.
17.   wire RegWrite;
18.   wire [2:0]ALUOp;
19.
20.   wire [5:0]operation;
21.
22.   wire zero;
23.
24.   Adder adder(
25.     32'd4,
26.     Addr_in,
27.     Addr_o
28.   );
29.
30.   IM im(
31.     Addr_in,
32.     instruction
```

```

33. );
34.
35. RF rf(
36.     clk,
37.     RegWrite,
38.     instruction[25:21],
39.     instruction[20:16],
40.     instruction[15:11],
41.     ALU_out,
42.     RTdata,
43.     src1);
44.
45. Control ctrl(
46.     instruction[31:26],
47.     RegWrite,
48.     ALUOp
49. );
50.
51. ALUctrl aluctrl(
52.     instruction[5:0],
53.     ALUOp,
54.     operation);
55.
56. ALU alu(
57.     instruction[10:6],
58.     src1,
59.     RTdata,
60.     operation,
61.     ALU_out,
62.     zero
63. );
64.
65.
66.
67.
68. endmodule

```

(2)解釋

將上述的模組全部依照架構圖接起來，即為 SingleCPU 模組，此模組在階段 1 僅可以執行 R-Format 的指令。

2. TestBench

(1)程式碼

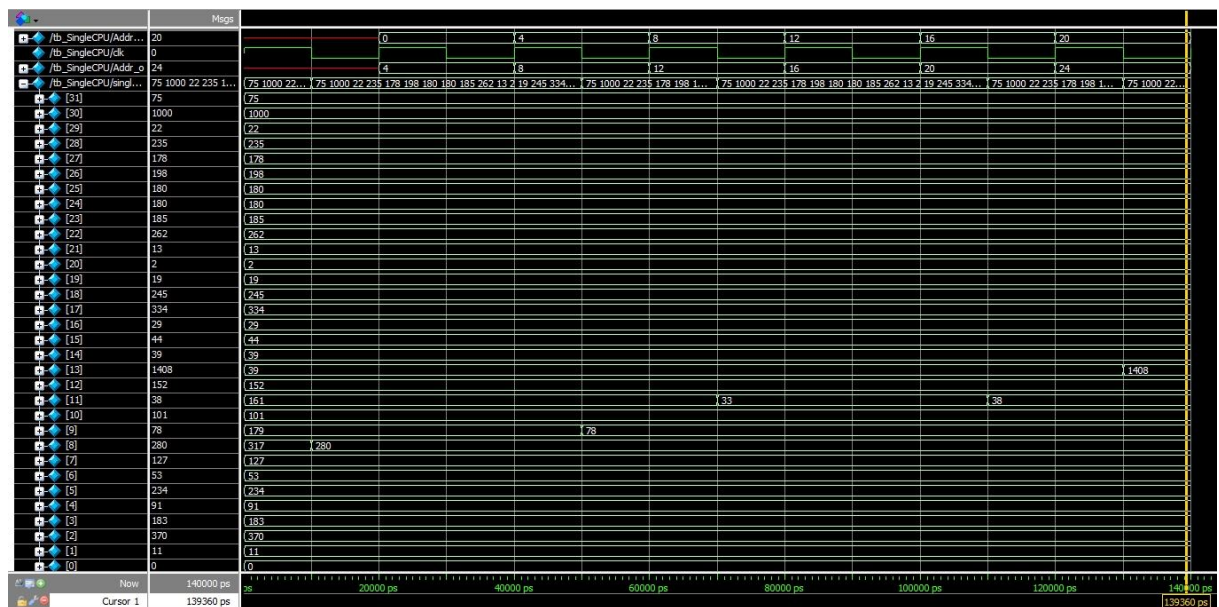
```
1. `timescale 1ns / 1ps
2. module tb_SingleCPU;
3.   reg [31:0] Addr_in;
4.   reg clk;
5.   wire [31:0] Addr_o;
6.   integer i;
7.
8.   SingleCPU singlecpu(
9.       Addr_in,
10.      clk,
11.      Addr_o
12. );
13.
14. initial begin
15.   clk <= 1;
16.   for(i=0;i<28;i=i+4)begin
17.       #10
18.       clk <= !clk;
19.       #10
20.       clk <= !clk;
21.       Addr_in <= i;
22.   end
23. end
24.
25. endmodule
```

(2)解釋

透過 for 迴圈依序放入指定的 instruction address，可以依照 instruction memory 中的指令執行。

3. 模擬結果

(1)模擬結果



(2)模擬結果分析

經過測試，CPU 可以順利地執行 Instruction memory 中位置第 0 至 20 的 MIPS R-type 指令。

貳、階段 2

一、程式功能

(一) SW 指令

1. SW 指令位於 Instruction Memory 的第 24、36、40 位置，其指令為

sw \$t0, 2(\$t3)

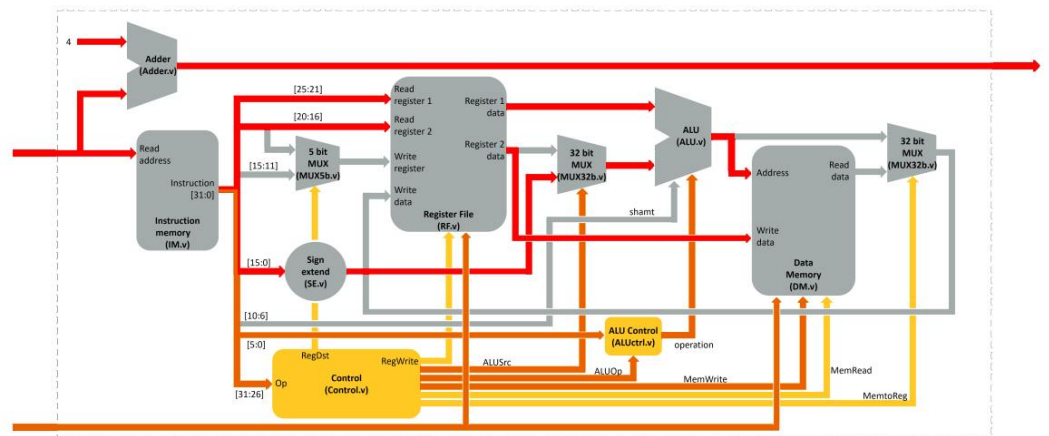
sw \$t0, 2(\$t2)

sw \$s4, 4(\$t1)

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[6] = {6'd39, 5'd11, 5'd8, 16'd2};  
Instr[9] = {6'd39, 5'd10, 5'd8, 16'd2};  
Instr[10] = {6'd39, 5'd9, 5'd20, 16'd4};
```

3. Data(紅)/Control(橙) Path



(二) LW 指令

1. LW 指令位於 Instruction Memory 的第 28、32 位置，其指令為

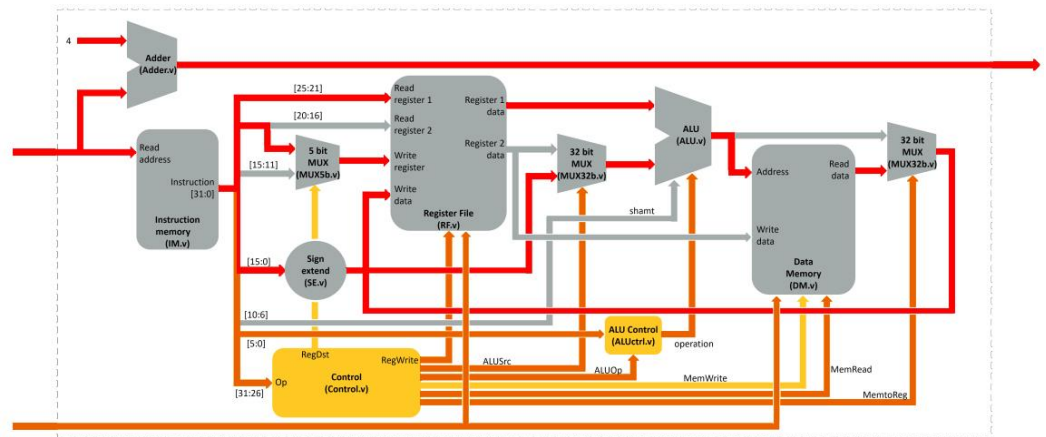
lw \$s3, 2(\$t3)

lw \$s4, 3(\$t3)

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[7] = {6'd40, 5'd11, 5'd19, 16'd2};  
Instr[8] = {6'd40, 5'd11, 5'd20, 16'd3};
```

3. Data(紅)/Control(橙) Path



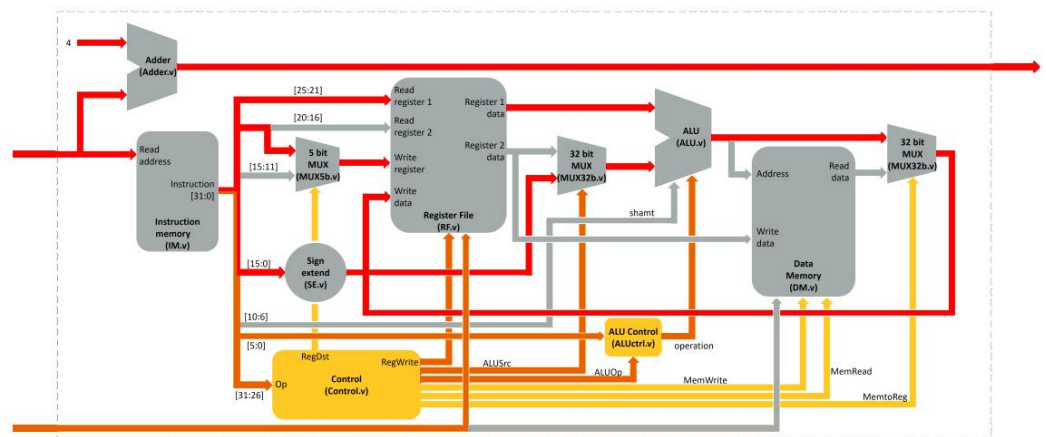
(三) ADDI 指令

1. ADDI 指令位於 Instruction Memory 的第 44、48 位置，其指令為
 $\text{addi } \$s5, \$s4, 40$
 $\text{addi } \$s6, \$s5, 22$

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[11] = {6'd41, 5'd20, 5'd21, 16'd40};
Instr[12] = {6'd41, 5'd21, 5'd22, 16'd22};
```

3. Data(紅)/Control(橙) Path



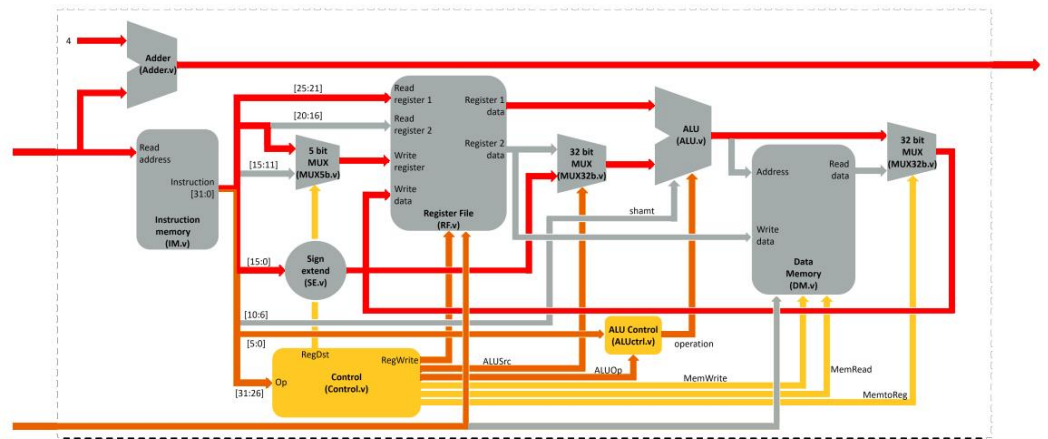
(四) SUBI 指令

1. SUBI 指令位於 Instruction Memory 的第 52、56，其指令為
 $\text{subi } \$s3, \$s6, 8$
 $\text{subi } \$s7, \$s0, 2$

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[13] = {6'd42, 5'd22, 5'd19, 16'd8};
Instr[14] = {6'd42, 5'd16, 5'd23, 16'd2};
```

3. Data(紅)/Control(橙) Path



二、程式碼

(一) Adder.v

1. 程式碼

(1)程式碼

```

1. module Adder(
2.     input [31:0] data1,
3.     input [31:0] data2,
4.     output [31:0] data_o
5. );
6.
7. assign data_o = data1 + data2;
8.
9. endmodule

```

(2)解釋

Adder 模組為一個 32bit 輸入，32bit 輸出的雙輸入、單輸出加法器，其實現方式為使用 Verilog 中的 Data Flow Model。

2. TestBench

(1)程式碼

```

1. module tb_Adder;
2.     reg [31:0] data1;
3.     reg [31:0] data2;
4.     wire [31:0] data_o;
5.     integer i;
6.
7.     Adder uut(
8.         data1,
9.         data2,

```



```

10.    data_o
11.);
12.

13.initial begin
14.for(i = 0; i<31;i=i+1)begin
15.    #10
16.    data1 = i;
17.    data2 = i*10;
18.end
19.end
20.endmodule

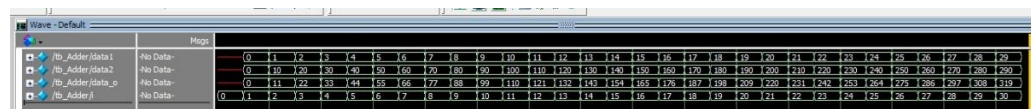
```

(2)解釋

透過一個 for 迴圈，使第一個輸入資料為 i，第二個資料為 i*10 依照迴圈依序送入 Adder 中進行加法的動作。

3. 模擬結果

(1)模擬結果



(2)模擬結果分析

根據模擬結果可以觀察到，輸出值確實等於兩個輸入值相加，因此程式功能正確。

(二) ALU.v

1. 程式碼

(1)程式碼

```

1. module ALU(input [4:0] shamt,
2.             input [31:0] src1,
3.             input [31:0] src2,
4.             input [5:0] operation,
5.             output reg [31:0] result,
6.             output zero);
7.
8. /*
9.  add 27
10. sub 28
11. and 29
12. or 30

```

```

13. srl 31
14. sll 32
15. */
16.
17. assign zero = result==0?1:0;
18.
19. always@(*)begin
20.     case(operation)
21.         6'd27:
22.             result <= src1 + src2;
23.         6'd28:
24.             result <= src1 - src2;
25.         6'd29:
26.             result <= src1 & src2;
27.         6'd30:
28.             result <= src1 | src2;
29.         6'd31:
30.             result <= src2 >> shamt;
31.         6'd32:
32.             result <= src2 << shamt;
33.         default:
34.             result <= 0;
35.     endcase
36. end
37. endmodule

```

(2)解釋

透過 ALU 的 OP Code 定義，定義出 27 為加法；28 為減法；29 為邏輯與運算；30 為邏輯或運算；31 為位元右移；32 為位元左移，並且當計算結果為 0 時，zero 腳位需要輸出高電位。

2. TestBench

(1)程式碼

```

1. module tb_ALU;
2.
3.     reg [4:0] shamt;
4.     reg [31:0] src1;
5.     reg [31:0] src2;
6.     reg [5:0] operation;

```

```

7. wire [31:0] result;
8. wire zero;
9.

10. ALU uut(
11.     shamt,
12.     src1,
13.     src2,
14.     operation,
15.     result,
16.     zero
17. );
18.
19. initial begin
20. src1 = 1;
21. src2 = 3;
22. operation = 27;
23. //1+3
24.
25. #10
26. src1 = 87;
27. src2 = 87;
28. operation = 28;
29. //87-87 = 0, zero = 1
30.
31. #10
32. src1 = 123;
33. src2 = 456;
34. operation = 29;
35. // 123 & 456 = 72
36.
37. #10
38. operation = 30;
39. // 123 | 546 = 507
40.
41. #10
42. src1 = 10;
43. shamt = 3;
44. operation = 31;

```



```

11.         operation <= 27;
12.         6'd22:
13.         operation <= 28;
14.         6'd23:
15.         operation <= 29;
16.         6'd24:
17.         operation <= 30;
18.         6'd25:
19.         operation <= 31;
20.         6'd26:
21.         operation <= 32;
22.     endcase
23. end
24. else if (ALUOp == 3'b000)begin //lw sw addi
25.     operation <= 27; //add
26. end
27. else if (ALUOp == 3'b001)begin //sbui
28.     operation <= 28; //sub
29.
30. end
31. end
32.
33. endmodule

```

(2)解釋

與 Part1 相比，ALU op 多了 IType 的指令，其中如果 ALU Op 為 0 時，需要做加法，operation 為 27；當 ALU Op 為 1 時，需要做減法，operation 為 28。

2. Testbench

(1)程式碼

```

1. module tb_ALUctrl;
2.     reg [5:0] funct;
3.     reg [2:0] ALUOp;
4.     wire [5:0] operation;
5.     integer i;
6.
7.     ALUctrl uut(
8.         funct,

```

```

9.     ALUOp,
10.    operation
11.);
12.

13.initial begin
14.    ALUOp = 3'b010;
15.    for(i=21;i<27;i=i+1)begin
16.        #10
17.        funct = i;
18.    end
19.
20.    #10
21.    ALUOp = 3'b000; //要做加法
22.
23.    #10
24.    ALUOp = 3'b001; //要做減法
25.
26.    #10
27.    ALUOp = 3'b010; //do nothing
28.end
29.
30.endmodule

```

(2)解釋

分別測試 R type、I type 的 ALU OP 為 0 或 1 時，要分別做加法或減法。

3. 模擬結果

(1)模擬結果

| | | Mems | | | | | | | | | | | | | | | |
|-----------------------|----------|------|--|--|--|--|--|--|--|--|--|--|--|--|----|--|----|
| /tb_ALUctrl/funct | No Data- | | | | | | | | | | | | | | | | |
| /tb_ALUctrl/ALUOp | No Data- | 2 | | | | | | | | | | | | | 0 | | 1 |
| /tb_ALUctrl/operation | No Data- | | | | | | | | | | | | | | 27 | | 28 |
| /tb_ALUctrl/I | No Data- | 21 | | | | | | | | | | | | | | | |

(2)分析

透過觀察模擬結果可以觀察到，ALUctrl 可以正確的在不同狀況下給予 ALU OP code。

(四) Control.v

1. 程式碼

(1)程式碼

```

1. module Control(
2. input [5:0]Op,

```

```

3. output reg RegDst,
4. output reg MemRead,
5. output reg MemtoReg,
6. output reg [2:0]ALUOp,
7. output reg MemWrite,
8. output reg ALUSrc,
9. output reg RegWrite
10.);
11.

12. always@(Op)begin
13.     case(Op)
14.         6'd54: begin// Rtyp
15.             ALUOp <= 3'b010;
16.             RegDst <= 1'b1;
17.             ALUSrc <= 1'b0;
18.             MemtoReg <= 1'b0;
19.             RegWrite <= 1'b1;
20.             MemRead <= 1'b0;
21.             MemWrite <= 1'b0;
22.         end
23.
24.         6'd39: begin// sw
25.             ALUOp <= 3'b000;
26.             RegDst <= 1'bz;
27.             ALUSrc <= 1'b1;
28.             MemtoReg <= 1'bz;
29.             RegWrite <= 1'b0;
30.             MemRead <= 1'b0;
31.             MemWrite <= 1'b1;
32.         end
33.
34.         6'd40: begin// lw
35.             ALUOp <= 3'b000;
36.             RegDst <= 1'b0;
37.             ALUSrc <= 1'b1;
38.             MemtoReg <= 1'b1;
39.             RegWrite <= 1'b1;

```

```

40.         MemRead <= 1'b1;
41.         MemWrite <= 1'b0;
42.     end
43.
44.     6'd41: begin//addi
45.         ALUOp <= 3'b000;
46.         RegDst <= 1'b0;
47.         ALUSrc <= 1'b1;
48.         MemtoReg <= 1'b0;
49.         RegWrite <= 1'b1;
50.         MemRead <= 1'b1;
51.         MemWrite <= 1'b0;
52.     end
53.     6'd42: begin //subi
54.         ALUOp <= 3'b001;
55.         RegDst <= 1'b0;
56.         ALUSrc <= 1'b1;
57.         MemtoReg <= 1'b0;
58.         RegWrite <= 1'b1;
59.         MemRead <= 1'b1;
60.         MemWrite <= 1'b0;
61.     end
62.     default: begin
63.         ALUOp <= 3'bzzz;
64.         RegDst <= 1'bz;
65.         ALUSrc <= 1'bz;
66.         MemtoReg <= 1'bz;
67.         RegWrite <= 1'bz;
68.         MemRead <= 1'bz;
69.         MemWrite <= 1'bz;
70.     end
71. endcase
72.
73. end
74.
75. endmodule

```

(2)解釋

與 Part1 相比，Control 的功能多了很多，需要處理

Rtype 指令、sw、lw、addi、subi 等指令，對於各模組與多工器給予不同的控制信號，詳細過程如上所示。

2. Testbench

(1)程式碼

```
1. module tb_Control;
2.
3. reg [5:0]Op;
4. wire RegDst;
5. wire MemRead;
6. wire MemtoReg;
7. wire [2:0]ALUOp;
8. wire MemWrite;
9. wire ALUSrc;
10. wire RegWrite;
11.
12. Control uut(
13.     Op,
14.     RegDst,
15.     MemRead,
16.     MemtoReg,
17.     ALUOp,
18.     MemWrite,
19.     ALUSrc,
20.     RegWrite
21. );
22.
23. initial begin
24.     Op = 6'd54;
25.     #10
26.     Op = 6'd39;
27.     #10
28.     Op = 6'd40;
29.     #10
30.     Op = 6'd41;
31.     #10
32.     Op = 6'd42;
33.     #10
34.     Op = 6'd00;
```

```

35.
36. end
37.
38. endmodule

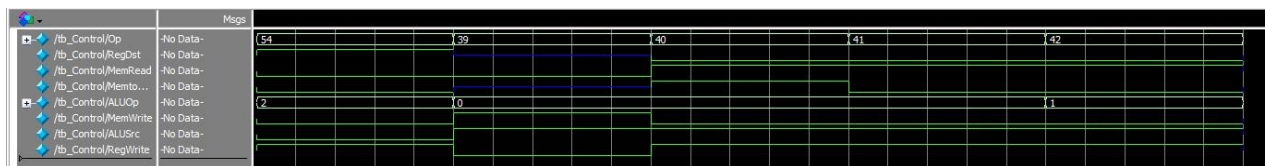
```

(2)解釋

本 Testbench 給予了不同的 Op Code，預期透過 Control 模組可以產生出不同的結果，用以控制 CPU 內及 Data memory 等各模組的控制信號。

3. 模擬結果

(1)模擬結果



(2)分析

經過 TestBench 後，Control 模組順利的產生了正確的控制信號，一切皆如預期。

(五) DM.v

1. 程式碼

(1)程式碼

```

1. module DM(input clk,
2.             input [31:0] addr,
3.             input [31:0] data,
4.             input MemRead,
5.             input MemWrite,
6.             output reg [31:0] DM_data);
7.
8.     integer i;
9.     reg [7:0] Mem[127:0];
10.
11.
12.     initial begin
13.         for(i = 0; i < 128; i = i + 1) begin
14.             Mem[i] = 8'd0;
15.         end
16.     end
17.
18.     always@(clk or MemWrite) begin

```

```

19.         if(clk & MemWrite)begin
20.             Mem[addr] <= data[31:24];
21.             Mem[addr + 1] <= data[23:16];
22.             Mem[addr + 2] <= data[15:8];
23.             Mem[addr + 3] <= data[7:0];
24.         end
25.     end
26.
27.     always@(*)begin
28.         if(MemRead)begin
29.             DM_data <=
{Mem[addr],Mem[addr+1],Mem[addr+2],Mem[addr+3]};
30.         end
31.     end
32.
33. endmodule

```

(2)解釋

Data Memory 預設所有數字皆為 0，可以透過 clk 及 Memory Write 來進行寫入的動作，其餘狀態下皆可讀取。

2. Testbench

(1)程式碼

```

1. module tb_DM;
2.     reg clk;
3.     reg [31:0] addr;
4.     reg [31:0] data;
5.     reg MemRead;
6.     reg MemWrite;
7.     wire [31:0] DM_data;
8.
9.     DM uut(
10.        clk,
11.        addr,
12.        data,
13.        MemRead,
14.        MemWrite,

```

```

15.    DM_data
16.);
17.
18.initial begin
19.    clk = 0;
20.    #10
21.    clk <= !clk;
22.    MemRead <= 0;
23.    MemWrite <= 1;
24.    data <= 32'd10;
25.    addr <= 32'd10;
26.    #10
27.    clk<= !clk;
28.    #10
29.    clk<= !clk;
30.    MemRead <= 1;
31.    MemWrite <= 0;
32.    addr <= 32'd10;
33.    #10
34.    clk <= !clk;
35.
36.end
37.
38.endmodule

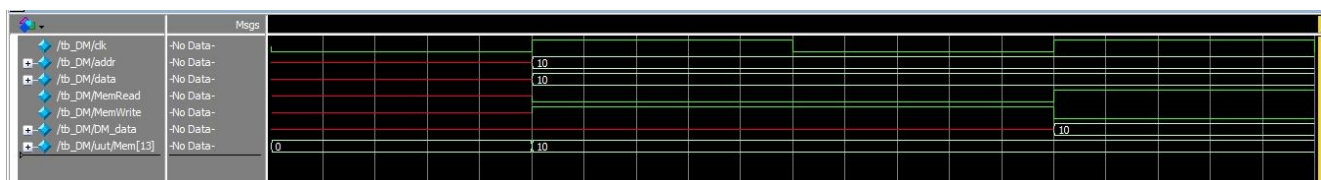
```

(2)解釋

測試寫入一記憶體值，並將其讀取出來。

3. 模擬結果

(1)模擬結果



(2)分析

Data Memory 可以順利的寫入及讀取資料。

(六) IM.v

1. 程式碼

(1)程式碼

```

1. module IM(input [31:0] Addr_in,
2.           output reg [31:0] instr);
3.
4.     reg [31:0] Instr[31:0];
5.
6.     initial begin
7.         Instr[0] = {6'd54,5'd9,5'd10,5'd8,5'd0,6'd21};
8.         Instr[1] = {6'd54,5'd9,5'd10,5'd9,5'd0,6'd22};
9.         Instr[2] = {6'd54,5'd13,5'd11,5'd11,5'd0,6'd23};
10.        Instr[3] = {6'd54,5'd11,5'd13,5'd13,5'd0,6'd24};
11.        Instr[4] = {6'd54,5'd0,5'd12,5'd11,5'd2,6'd25};
12.        Instr[5] = {6'd54,5'd0,5'd15,5'd13,5'd5,6'd26};
13.
14.        Instr[6] = {6'd39,5'd11,5'd8,16'd2};
15.        Instr[7] = {6'd40,5'd11,5'd19,16'd2};
16.        Instr[8] = {6'd40,5'd11,5'd20,16'd3};
17.        Instr[9] = {6'd39,5'd10,5'd8,16'd2};
18.        Instr[10] = {6'd39,5'd9,5'd20,16'd4};
19.        Instr[11] = {6'd41,5'd20,5'd21,16'd40};
20.        Instr[12] = {6'd41,5'd21,5'd22,16'd22};
21.        Instr[13] = {6'd42,5'd22,5'd19,16'd8};
22.        Instr[14] = {6'd42,5'd16,5'd23,16'd2};
23.
24.        instr = Instr[0];
25.    end
26.
27.    always@(Addr_in)begin
28.        instr <= Instr[Addr_in>>2];
29.    end
30.
31. endmodule

```

(2)解釋

如上功能解釋的部分，除了 Part1 的部分外，另外新增了記憶體位置 24 至 56 的 iType 指令 sw、lw、addi、subi。

2. Testbench

(1)程式碼

```
1. module tb_IM;
2.   reg [31:0] Addr_in;
3.   wire [31:0] instr;
4.   integer i;
5.
6.   IM uut(
7.     Addr_in,
8.     instr
9.   );
10.
11. initial begin
12.   for(i=0;i<=20;i=i+4)begin
13.     #10
14.     Addr_in = i;
15.   end
16.   #10
17.   Addr_in = 0;
18.
19. end
20. endmodule
```

(2)解釋

依序給予不同的 instruction memory 位置，測試回傳的 instruction 是否正確。

3. 模擬結果

(1)模擬結果

| | | Mags | | | | | | | | | |
|----------------|--------------|------------|---|------------|------------|------------|------------|------------|--|--|--|
| /tb_IM/Addr_in | No Data | | | | | | | | | | |
| | /tb_IM/instr | | | | | | | | | | |
| | /tb_IM/i | | | | | | | | | | |
| | | | 0 | 4 | 8 | 12 | 16 | 20 | | | |
| | | 3643424789 | | 3643426838 | 3651885079 | 3647825944 | 3624687769 | 3624888666 | | | |
| | | 0 | 4 | 8 | 12 | 16 | 20 | 24 | | | |

(2)分析

給予不同的 instruction memory 位置，發現 IM 回傳的值與預期相同，因此此模組設計正確。

(七) MUX5b.v

1. 程式碼

(1)程式碼

```
1. module MUX5b(
```

```

2.     input [4:0] data1,
3.     input [4:0] data2,
4.     input select,
5.     output [4:0] data_o
6. );
7.
8. assign data_o = select?data2:data1;
9.
10. endmodule

```

(2)解釋

本模組為一個 5bit 的多工器，透過 select 信號可以選擇 data2 或 data1 輸出。

2. Testbench

(1)程式碼

```

1. module tb_MUX5b;
2. reg [4:0] data1;
3. reg [4:0] data2;
4. reg select;
5.
6. wire [4:0] data_o;
7.
8. MUX5b uut(
9.     data1,
10.    data2,
11.    select,
12.    data_o
13.);
14.
15. initial begin
16.     data1 = 5'd12;
17.     data2 = 5'd37;
18.     select = 1'b1;
19.     #10
20.     select = 1'b0;
21.
22.     #10
23.     select = 1'b1;
24. end

```

```
25.
26. endmodule
```

(2)解釋

本 Testbench 測試個別放入兩個不同的 data1 及 data2，並透過 select 信號測試是否能正確的輸出 data1 與 data2。

3. 模擬結果

(1)模擬結果

| | Msgs | |
|------------------|-----------|----|
| /tb_MUX3b/data1 | -No Data- | 12 |
| /tb_MUX3b/data2 | -No Data- | 5 |
| /tb_MUX3b/select | -No Data- | |
| /tb_MUX3b/data_o | -No Data- | 5 |

(2)分析

經過 Testbench 測試，本模組的輸出可透過 select 控制，是一個正確的多工器。

(八) MUX32b.v

1. 程式碼

(1)程式碼

```
1. module MUX32b(
2.     input [31:0] data1,
3.     input [31:0] data2,
4.     input select,
5.     output [31:0] data_o
6. );
7.
8. assign data_o = select?data2:data1;
9. // assign data_o = data1;
10.
11. endmodule
```

(2)解釋

本模組為一個 32bit 的多工器，透過 select 信號可以選擇 data2 或 data1 輸出。

2. Testbench

(1)程式碼

```
1. module tb_MUX32b;
2.     reg [31:0] data1;
3.     reg [31:0] data2;
4.     reg select;
5.
```






```
6. wire [31:0] data_o;
7.
8. MUX32b uut(
9.     data1,
10.    data2,
11.    select,
12.    data_o
13.);
14.
15.initial begin
16.    data1 = 32'd314159;
17.    data2 = 32'd123456;
18.    select = 1'b1;
19.    #10
20.    select = 1'b0;
21.
22.    #10
23.    select = 1'b1;
24.end
25.
26.endmodule
```

(2)解釋

本 Testbench 測試個別放入兩個不同的 data1 及 data2，並透過 select 信號測試是否能正確的輸出 data1 與 data2。

3. 模擬結果

(1)模擬結果

| | | Mops | |
|---|-------------------|---------|---------------|
|  | /fb_MUX32b/data1 | No Data | 314159 |
|  | /fb_MUX32b/data2 | No Data | 123456 |
| | /fb_MUX32b/select | No Data | |
|  | /fb_MUX32b/data_o | No Data | 123456 314159 |

(2)分析

經過 Testbench 測試，本模組的輸出可透過 select 控制，是一個正確的多工器。

(九) RF.v

1. 程式碼

(1)程式碼

```
1. module RF(input clk,
2.           input RegWrite,
```

```

3.         input [4:0] RSaddr,
4.         input [4:0] RTaddr,
5.         input [4:0] RDaddr,
6.         input [31:0] RDdata,
7.         output reg [31:0] RTdata,
8.         output reg [31:0] src1);
9.
10.
11.    reg [31:0] REGISTER[31:0];
12.
13.    initial begin
14.        REGISTER[0] = 32'd0;
15.        REGISTER[1] = 32'd11;
16.        REGISTER[2] = 32'd370;
17.        REGISTER[3] = 32'd183;
18.        REGISTER[4] = 32'd91;
19.        REGISTER[5] = 32'd234;
20.        REGISTER[6] = 32'd53;
21.        REGISTER[7] = 32'd127;
22.        REGISTER[8] = 32'd317;
23.        REGISTER[9] = 32'd179;
24.        REGISTER[10] = 32'd101;
25.        REGISTER[11] = 32'd161;
26.        REGISTER[12] = 32'd152;
27.        REGISTER[13] = 32'd39;
28.        REGISTER[14] = 32'd39;
29.        REGISTER[15] = 32'd44;
30.        REGISTER[16] = 32'd29;
31.        REGISTER[17] = 32'd334;
32.        REGISTER[18] = 32'd245;
33.        REGISTER[19] = 32'd19;
34.        REGISTER[20] = 32'd2;
35.        REGISTER[21] = 32'd13;
36.        REGISTER[22] = 32'd262;
37.        REGISTER[23] = 32'd185;
38.        REGISTER[24] = 32'd180;
39.        REGISTER[25] = 32'd180;
40.        REGISTER[26] = 32'd198;

```

```

41.     REGISTER[27] = 32'd178;
42.     REGISTER[28] = 32'd235;
43.     REGISTER[29] = 32'd22;
44.     REGISTER[30] = 32'd1000;
45.     REGISTER[31] = 32'd75;
46. end
47.
48. always@(posedge clk or RSaddr or RTaddr or RDaddr)begin
49.     RTdata <= REGISTER[RTaddr];
50.     srcl   <= REGISTER[RSaddr];
51. end
52.
53. always@(negedge clk)begin
54.     if (RegWrite)begin
55.         REGISTER[RDaddr] <= RDdata;
56.     end
57. end
58. endmodule

```

(2)解釋

透過本模組，可以對佔存器進行讀取或是寫入的動作，當信號為正緣 clk，或是 address 變動時，會進行讀取的動作；當信號為負緣 clk，則進行寫入的動作

2. TestBench

(1)程式碼

```

1. module tb_RF;
2.
3. // RF Inputs
4. reg  clk = 0 ;
5. reg  RegWrite = 0 ;
6. reg  [4:0] RSaddr = 0 ;
7. reg  [4:0] RTaddr = 0 ;
8. reg  [4:0] RDaddr = 0 ;
9. reg  [31:0] RDdata = 0 ;
10.
11. // RF Outputs
12. wire [31:0] RTdata;
13. wire [31:0] srcl;
14. integer i;

```

```

15.
16. RF u_RF(
17.     clk,
18.     RegWrite,
19.     RSaddr,
20.     RTaddr,
21.     RDaddr,
22.     RDdata,
23.     RTdata,
24.     src1
25. );
26.
27. initial
28. begin
29.     clk = 1;
30.     for(i=0;i<32;i=i+1)begin
31.         #10
32.         clk = !clk;
33.         #10
34.         clk = !clk;
35.         RSaddr = i;
36.         RTaddr = 31-i;
37.     end
38.
39.     RegWrite = 1;
40.     for(i=0;i<32;i=i+1)begin
41.         #10
42.         clk = !clk;
43.         #10
44.         clk = !clk;
45.         RDaddr = i;
46.         RDdata = i*10;
47.     end
48.
49.     for(i=0;i<32;i=i+1)begin
50.         #10
51.         clk = !clk;
52.         #10

```

```

53.     clk = !clk;
54.     RSaddr = i;
55.     end
56.

57. end
58.
59. endmodule

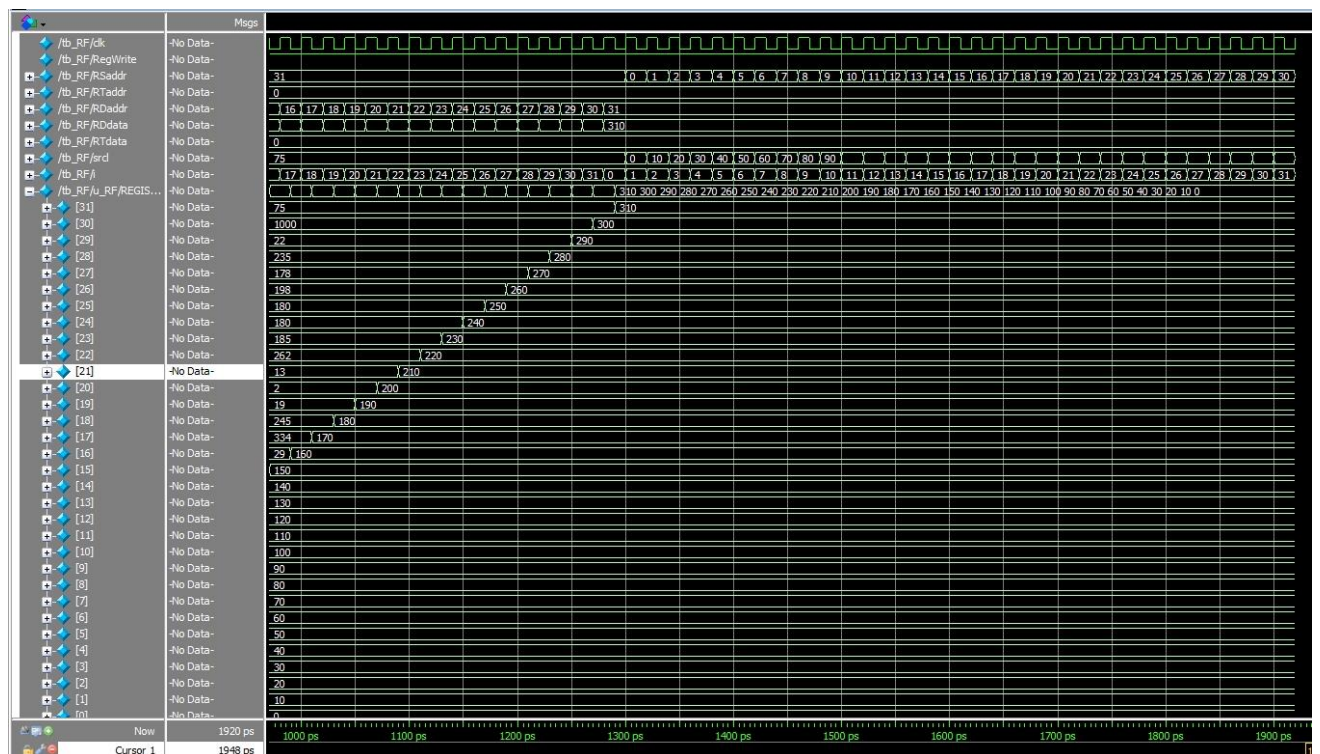
```

(2)解釋

透過 for 迴圈測試對於站存器的讀取以及寫入動作。

3. 模擬結果

(1)模擬結果



(2)模擬結果分析

經過 Testbench，可以觀察到無論是讀取或是寫入的動作，RF 皆能正常運作。

(十) SE.v

1. 程式碼

(1)程式碼

```

1. module SE(
2.     input [15:0] data_i,
3.     output [31:0] data_o
4. );

```

```

5. assign data_o = data_i[15]?{16'hffff,data_i}:{16'h0000,data_i};
6.
7. endmodule

```

(2)解釋

本模組可以使輸入的 16bit 有號數擴展成 32bit 有號數。

2. Testbench

(1)程式碼

```

1. module tb_SE;
2. reg [15:0] data_i;
3. wire [31:0] data_o;
4. integer i;
5.
6. SE uut(
7.     data_i,
8.     data_o
9. );
10.
11. initial begin
12.     for(i=0;i<16'hffff;i=i+5)begin
13.         #10
14.         data_i = i;
15.     end
16. end
17.
18. endmodule

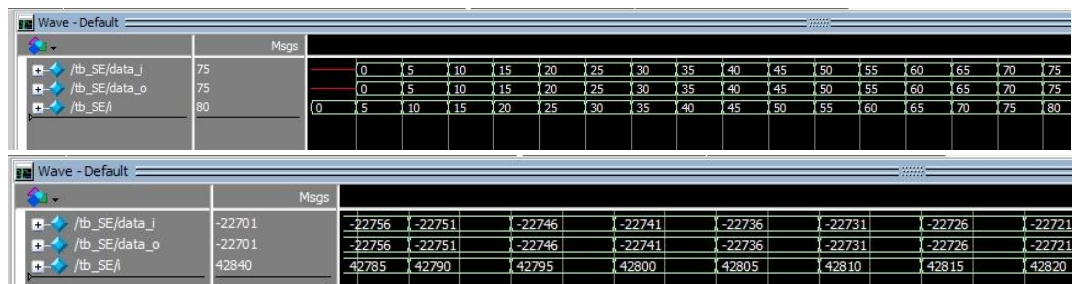
```

(2)解釋

透過 for 迴圈測試是否可以順利的處理正號與負號的狀態。

3. 模擬結果

(1)模擬結果



(2)分析

經測試，正負兩種情形的運作皆正常。

(十一) SingleCPU.v

1. 程式碼

(1)程式碼

```
1. module SingleCPU(  
2.     input [31:0] Addr_in,  
3.     input clk,  
4.     output [31:0] Addr_o  
5.     // output [31:0] alu_result,  
6.     // output ALUSrc  
7.     // output [31:0]mux32b1_out,  
8.     // output [31:0]dm_out  
9. );  
10.  
11. wire [31:0]instruction;  
12.  
13. // ctrl 勿 out  
14. wire RegDst;  
15. wire MemRead;  
16. wire MemtoReg;  
17. wire [2:0]ALUOp;  
18. wire MemWrite;  
19. wire ALUSrc;  
20. wire RegWrite;  
21.  
22. wire [4:0]write_register;  
23.  
24. wire [31:0]se_out;  
25.  
26. wire [31:0]write_data;  
27.  
28. wire [31:0]register1_out;  
29. wire [31:0]register2_out;  
30.  
31. wire [31:0]mux32b1_out;
```

```

32.
33. wire [5:0]operation;
34.
35. wire [31:0]alu_result;
36.
37. wire zero;
38.
39. wire [31:0]dm_out;
40.
41. Adder adder(
42.     .data1(4),
43.     .data2(Addr_in),
44.     .data_o(Addr_o)
45. );
46.
47. IM im(
48.     .Addr_in(Addr_in),
49.     .instr(instruction)
50. );
51.
52. MUX5b mux5b(
53.     .data1(instruction[20:16]),
54.     .data2(instruction[15:11]),
55.     .select(RegDst),
56.     .data_o(write_register)
57. );
58.
59. SE se(
60.     .data_i(instruction[15:0]),
61.     .data_o(se_out)
62. );
63.
64. Control control(
65.     .Op(instruction[31:26]),
66.     .RegDst(RegDst),
67.     .MemRead(MemRead),
68.     .MemtoReg(MemtoReg),
69.     .ALUOp(ALUOp),

```



```

70.     .MemWrite(MemWrite),
71.     .ALUSrc(ALUSrc),
72.     .RegWrite(RegWrite)
73. );
74.
75. RF rf(
76.     .clk(clk),
77.     .RegWrite(RegWrite),
78.     .RSaddr(instruction[25:21]),
79.     .RTaddr(instruction[20:16]),
80.     .RDaddr(write_register),
81.     .RDdata(write_data),
82.     .RTdata(register2_out),
83.     .src1(register1_out)
84. );
85.
86. MUX32b mux32b1(
87.     .data1(register2_out),
88.     .data2(se_out),
89.     .select(ALUSrc),
90.     .data_o(mux32b1_out)
91. );
92.
93. ALU alu(
94.     .shamt(instruction[10:6]),
95.     .src1(register1_out),
96.     .src2(mux32b1_out),
97.     .operation(operation),
98.     .result(alu_result),
99.     .zero(zero)
100. );
101.
102. ALUctrl aluctrl(
103.     .funct(instruction[5:0]),
104.     .ALUOp(ALUOp),
105.     .operation(operation)
106. );
107.

```

```

108.    DM dm(
109.        .clk(clk),
110.        .addr(alu_result),
111.        .data(register2_out),
112.        .MemRead(MemRead),
113.        .MemWrite(MemWrite),
114.        .DM_data(dm_out)
115.    );
116.
117.    MUX32b mux32b2(
118.        .data1(alu_result),
119.        .data2(dm_out),
120.        .select(MemtoReg),
121.        .data_o(write_data)
122.    );
123.
124.    endmodule

```

(2)解釋

將上述的模組全部依照架構圖接起來，即為 SingleCPU 模組，此模組在階段 2 可以執行 R-Format、I-Format 的指令。

2. Testbench

(1)程式碼

```

1. module tb_SingleCPU;
2.   reg [31:0] Addr_in;
3.   reg clk;
4.   wire [31:0] Addr_o;
5.   // wire [31:0]alu_result;
6.   // wire ALUSrc;
7.   // wire [31:0]mux32b1_out;
8.   // wire [31:0]dm_out;
9.   integer i;
10.
11. SingleCPU uut(
12.     Addr_in,
13.     clk,
14.     Addr_o

```

```

15.    // alu_result,
16.    // ALUSrc
17.    // mux32b1_out,
18.    // dm_out
19.);
20.
21.initial begin
22.    i <= 0;
23.    clk <= 0;
24.    #10
25.    clk <= !clk;
26.    for(i=0;i<64;i=i+4)begin
27.        #10
28.        clk <= !clk;
29.        #10
30.        clk <= !clk;
31.        Addr_in <= i;
32.    end
33.
34.end
35.
36.endmodule

```

(2)解釋

透過迴圈依序從位址 0 存取到 56，將第 2 階段以上的所有內容都執行一次。

3. 模擬結果

(1)模擬結果

參、階段 3(a)

一、程式功能

(一) BEQ 指令

1. BEQ 指令位於 Instruction Memory 的第 60 至 72 位置，其指令為

beq \$t4, \$t6, 4

beq \$t4, \$t8, 4

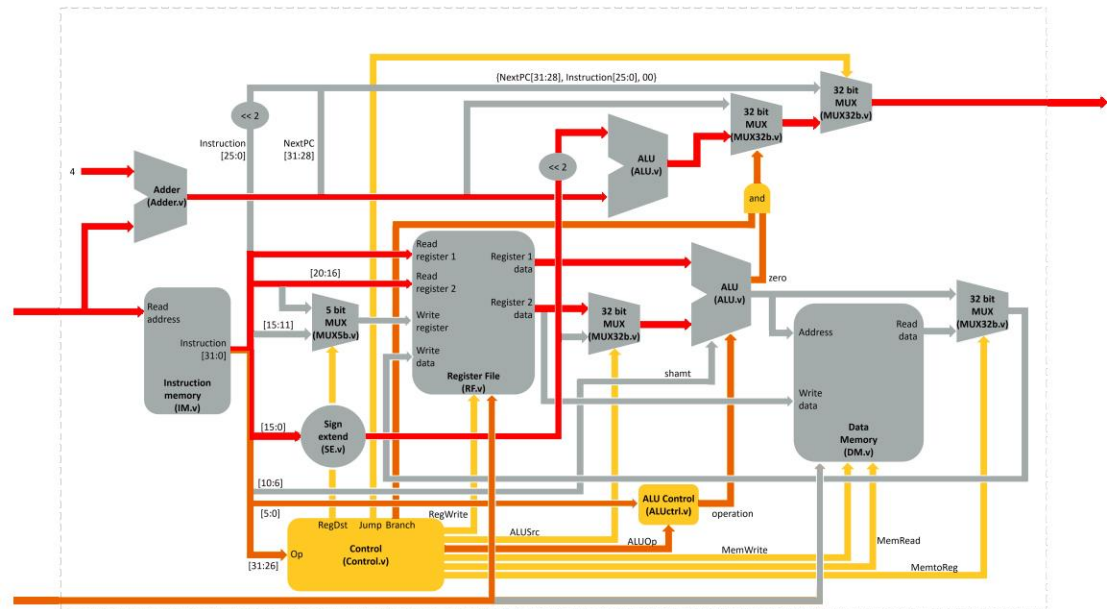
beq \$s3, \$s5, 4

beq \$t8, \$t9, 1

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[15] = {6'd31, 5'd12, 5'd14, 16'd4};  
Instr[16] = {6'd31, 5'd12, 5'd24, 16'd4};  
Instr[17] = {6'd31, 5'd19, 5'd21, 16'd4};  
Instr[18] = {6'd31, 5'd24, 5'd25, 16'd1};
```

3. Data(紅)/Control(橙) Path



(二) JUMP 指令

1. J 指令位於 Instruction Memory 的第 12 位置，其指令為

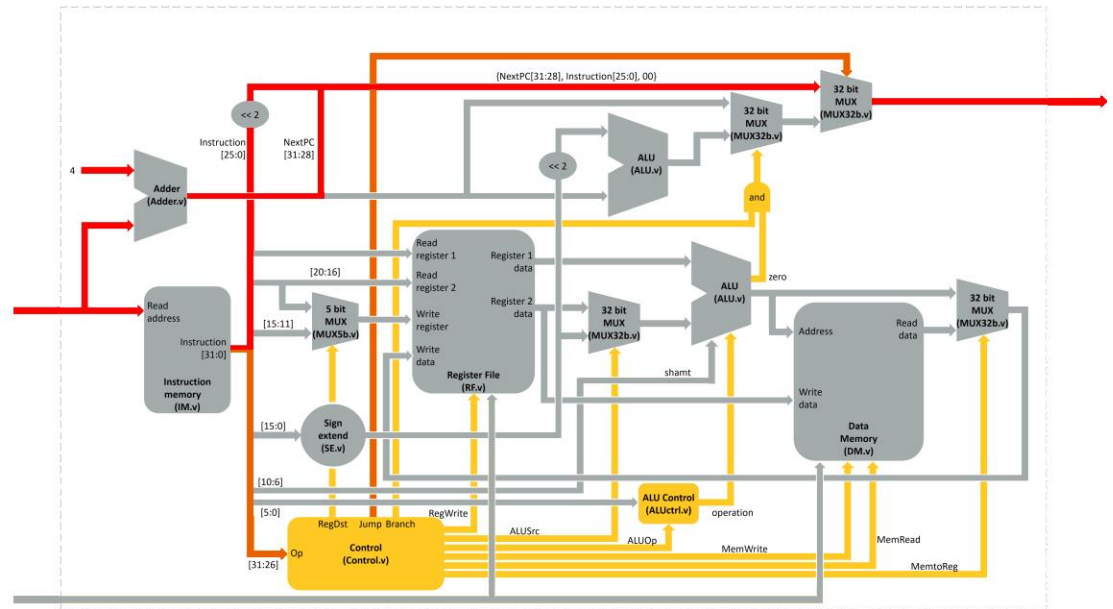
j 125

j 18

2. 在 im 中將其指令轉換為 instruction 的 binary 為

```
Instr[19] = {6'd32, 26'd125};  
Instr[20] = {6'd32, 26'd18};
```

3. Data(紅)/Control(橙) Path



二、程式碼

(一) Adder.v

1. 程式碼

(1) 程式碼

```

10. module Adder(
11.     input [31:0] data1,
12.     input [31:0] data2,
13.     output [31:0] data_o
14.);
15.
16. assign data_o = data1 + data2;
17.
18. endmodule

```

(2) 解釋

Adder 模組為一個 32bit 輸入，32bit 輸出的雙輸入、單輸出加法器，其實現方式為使用 Verilog 中的 Data Flow Model。

2. TestBench

(1) 程式碼

```

21. module tb_Adder;
22. reg [31:0] data1;
23. reg [31:0] data2;
24. wire [31:0] data_o;
25. integer i;
26.

```

```

27. Adder uut(
28.     data1,
29.     data2,
30.     data_o
31.);
32.

33. initial begin
34.     for(i = 0; i<31;i=i+1)begin
35.         #10
36.         data1 = i;
37.         data2 = i*10;
38.     end
39. end
40. endmodule

```

(2)解釋

透過一個 for 迴圈，使第一個輸入資料為 i，第二個資料為 i*10 依照迴圈依序送入 Adder 中進行加法的動作。

3. 模擬結果

(1)模擬結果

| Time (ns) | data1 | data2 | data_o |
|-----------|-------|-------|--------|
| 0 | 0 | 0 | 0 |
| 10 | 1 | 10 | 11 |
| 20 | 2 | 20 | 22 |
| 30 | 3 | 30 | 33 |
| 40 | 4 | 40 | 44 |
| 50 | 5 | 50 | 55 |
| 60 | 6 | 60 | 66 |
| 70 | 7 | 70 | 77 |
| 80 | 8 | 80 | 88 |
| 90 | 9 | 90 | 99 |
| 100 | 10 | 100 | 110 |
| 110 | 11 | 110 | 121 |
| 120 | 12 | 120 | 132 |
| 130 | 13 | 130 | 143 |
| 140 | 14 | 140 | 154 |
| 150 | 15 | 150 | 165 |
| 160 | 16 | 160 | 176 |
| 170 | 17 | 170 | 187 |
| 180 | 18 | 180 | 198 |
| 190 | 19 | 190 | 209 |
| 200 | 20 | 200 | 220 |
| 210 | 21 | 210 | 231 |
| 220 | 22 | 220 | 242 |
| 230 | 23 | 230 | 253 |
| 240 | 24 | 240 | 264 |
| 250 | 25 | 250 | 275 |
| 260 | 26 | 260 | 286 |
| 270 | 27 | 270 | 297 |
| 280 | 28 | 280 | 308 |
| 290 | 29 | 290 | 319 |
| 300 | 30 | 300 | 330 |

(2)模擬結果分析

根據模擬結果可以觀察到，輸出值確實等於兩個輸入值相加，因此程式功能正確。

(二) ALU.v

1. 程式碼

(1)程式碼

```

1. module ALU(input [4:0] shamt,
2.             input [31:0] src1,
3.             input [31:0] src2,
4.             input [5:0] operation,
5.             output reg [31:0] result,
6.             output zero);
7.
8. /*
9.  add 27

```

```

10. sub 28
11. and 29
12. or 30
13. srl 31
14. sll 32
15. */
16.
17. assign zero = result==0?1:0;
18.
19. always@(*)begin
20.     case(operation)
21.         6'd27:
22.             result <= src1 + src2;
23.         6'd28:
24.             result <= src1 - src2;
25.         6'd29:
26.             result <= src1 & src2;
27.         6'd30:
28.             result <= src1 | src2;
29.         6'd31:
30.             result <= src2 >> shamt;
31.         6'd32:
32.             result <= src2 << shamt;
33.         default:
34.             result <= 0;
35.     endcase
36. end
37. endmodule

```

(2)解釋

透過 ALU 的 OP Code 定義，定義出 27 為加法；28 為減法；29 為邏輯與運算；30 為邏輯或運算；31 為位元右移；32 為位元左移，並且當計算結果為 0 時，zero 腳位需要輸出高電位。

2. TestBench

(1)程式碼

```

1. module tb_ALU;
2.
3. reg [4:0] shamt;

```



```

4. reg [31:0] src1;
5. reg [31:0] src2;
6. reg [5:0] operation;
7. wire [31:0] result;
8. wire zero;
9.

10. ALU uut(
11.     shamt,
12.     src1,
13.     src2,
14.     operation,
15.     result,
16.     zero
17. );
18.
19. initial begin
20. src1 = 1;
21. src2 = 3;
22. operation = 27;
23. //1+3
24.
25. #10
26. src1 = 87;
27. src2 = 87;
28. operation = 28;
29. //87-87 = 0, zero = 1
30.
31. #10
32. src1 = 123;
33. src2 = 456;
34. operation = 29;
35. // 123 & 456 = 72
36.
37. #10
38. operation = 30;
39. // 123 | 546 = 507
40.
41. #10

```

```
42.src1 = 10;
43.shamt = 3;
44.operation = 31;
45.// 10 >> 3 = 1;
46.
47.// #10
48.operation = 32;
49.//0 << 3 = 80
50.
51.#10
52.operation = 0;
53.end
54.
55.endmodule
```

```

8.     if(ALUOp == 3'b010)begin
9.         case(funcnt)
10.            6'd21:
11.                operation <= 27;
12.            6'd22:
13.                operation <= 28;
14.            6'd23:
15.                operation <= 29;
16.            6'd24:
17.                operation <= 30;
18.            6'd25:
19.                operation <= 31;
20.            6'd26:
21.                operation <= 32;
22.        endcase
23.    end
24.    else if (ALUOp == 3'b000)begin //lw sw addi
25.        operation <= 27; //add
26.    end
27.    else if (ALUOp == 3'b001)begin //sbui
28.        operation <= 28; //sub
29.    end
30.    else if (ALUOp == 3'b101)begin
31.        operation <= 28; //sub
32.    end
33.end
34.
35.endmodule

```

(2)解釋

暨先前第 2 階段所寫的 ALUCtrl，在第 3 階段需要新增一個 ALUOp == 3'b101 的狀況，也就是判斷 beq 時，ALU 需要執行減法。

2. Testbench

(1)程式碼

```

1. module tb_ALUctrl;
2.     reg [5:0] funct;
3.     reg [2:0] ALUOp;
4.     wire [5:0] operation;

```

```

5. integer i;
6.
7. ALUctrl uut(
8.     funct,
9.     ALUOp,
10.    operation
11.);
12.
13.initial begin
14.    ALUOp = 3'b010;
15.    for(i=21;i<27;i=i+1)begin
16.        #10
17.        funct = i;
18.    end
19.
20.    #10
21.    ALUOp = 3'b000; //要做加法
22.
23.    #10
24.    ALUOp = 3'b001; //要做減法
25.
26.    #10
27.    ALUOp = 3'b101; //減法
28.
29.    #10
30.    ALUOp = 3'b010; //do nothing
31.end
32.
33.endmodule

```

(2)解釋

除了保持先前的狀態皆可以正常運行外，當 ALUOp = 3'b101 時需要做減法。

3. 模擬結果

(1)模擬結果

| | | Mags | | | | | | | | | | | |
|-----------------------|----------|------|----|----|----|----|----|----|----|---|--|--|--|
| /tb_ALUctrl/funct | No Data- | 21 | 22 | 23 | 24 | 25 | 26 | 0 | 1 | 5 | | | |
| /tb_ALUctrl/ALUOp | No Data- | 27 | 28 | 29 | 30 | 31 | 32 | 27 | 28 | | | | |
| /tb_ALUctrl/operation | No Data- | 21 | 22 | 23 | 24 | 25 | 26 | 27 | | | | | |
| /tb_ALUctrl/i | No Data- | | | | | | | | | | | | |

(2)分析

ALUctrl 的動作如預期，在 beq 時會做減法，其他狀態與先前 Part2 相同。

(四) Control.v

1. 程式碼

(1)程式碼

```
1. module Control(  
2.   input [5:0]Op,  
3.   output reg RegDst,  
4.   output reg MemRead,  
5.   output reg MemtoReg,  
6.   output reg [2:0]ALUOp,  
7.   output reg MemWrite,  
8.   output reg ALUSrc,  
9.   output reg RegWrite,  
10.  
11. output reg Jump,  
12. output reg Branch  
13.);  
14.  
15. always@(Op)begin  
16.   case(Op)  
17.     6'd54: begin// Rtyp  
18.       ALUOp <= 3'b010;  
19.       RegDst <= 1'b1;  
20.       ALUSrc <= 1'b0;  
21.       MemtoReg <= 1'b0;  
22.       RegWrite <= 1'b1;  
23.       MemRead <= 1'b0;  
24.       MemWrite <= 1'b0;  
25.       Branch <= 1'b0;  
26.       Jump <= 1'b0;  
27.     end  
28.  
29.     6'd39: begin// sw  
30.       ALUOp <= 3'b000;  
31.       RegDst <= 1'bz;  
32.       ALUSrc <= 1'b1;
```

```

33.         MementoReg <= 1'bz;
34.         RegWrite <= 1'b0;
35.         MemRead <= 1'b0;
36.         MemWrite <= 1'b1;
37.         Branch <= 1'b0;
38.         Jump <= 1'b0;
39.     end
40.
41.     6'd40: begin// lw
42.         ALUOp <= 3'b000;
43.         RegDst <= 1'b0;
44.         ALUSrc <= 1'b1;
45.         MementoReg <= 1'b1;
46.         RegWrite <= 1'b1;
47.         MemRead <= 1'b1;
48.         MemWrite <= 1'b0;
49.         Branch <= 1'b0;
50.         Jump <= 1'b0;
51.     end
52.
53.     6'd41: begin//addi
54.         ALUOp <= 3'b000;
55.         RegDst <= 1'b0;
56.         ALUSrc <= 1'b1;
57.         MementoReg <= 1'b0;
58.         RegWrite <= 1'b1;
59.         MemRead <= 1'b1;
60.         MemWrite <= 1'b0;
61.         Branch <= 1'b0;
62.         Jump <= 1'b0;
63.     end
64.     6'd42: begin //subi
65.         ALUOp <= 3'b001;
66.         RegDst <= 1'b0;
67.         ALUSrc <= 1'b1;
68.         MementoReg <= 1'b0;
69.         RegWrite <= 1'b1;
70.         MemRead <= 1'b1;

```

```

71.         MemWrite <= 1'b0;
72.         Branch <= 1'b0;
73.         Jump <= 1'b0;
74.     end
75.
76.     6'd31:begin //beq
77.         ALUOp <= 3'b101;
78.         RegDst <= 1'bz;
79.         ALUSrc <= 1'b0;
80.         MemtoReg <= 1'bz;
81.         RegWrite <= 1'b0;
82.         MemRead <= 1'b0;
83.         MemWrite <= 1'b0;
84.         Branch <= 1'b1;
85.         Jump <= 1'b0;
86.     end
87.
88.     6'd32:begin //j
89.         ALUOp <= 3'b101;
90.         RegDst <= 1'bz;
91.         ALUSrc <= 1'b0;
92.         MemtoReg <= 1'bz;
93.         RegWrite <= 1'b0;
94.         MemRead <= 1'b0;
95.         MemWrite <= 1'b0;
96.         Branch <= 1'b0;
97.         Jump <= 1'b1;
98.     end
99.
100.    default: begin
101.        ALUOp <= 3'bzzz;
102.        RegDst <= 1'bz;
103.        ALUSrc <= 1'bz;
104.        MemtoReg <= 1'bz;
105.        RegWrite <= 1'bz;
106.        MemRead <= 1'bz;
107.        MemWrite <= 1'bz;
108.        Jump <= 1'bz;

```

```

109.         end
110.     endcase
111.
112. end
113.
114. endmodule

```

(2)解釋

新增了 Op Code 等於 31 及 32，也就是 beq 以及 j 兩個指令時，對於其他模組的控制信號。

2. Testbench

(1)程式碼

```

1. module tb_Control;
2.
3. reg [5:0]Op;
4. wire RegDst;
5. wire MemRead;
6. wire MemtoReg;
7. wire [2:0]ALUOp;
8. wire MemWrite;
9. wire ALUSrc;
10. wire RegWrite;
11. wire Jump;
12. wire Branch;
13. Control uut(
14.     Op,
15.     RegDst,
16.     MemRead,
17.     MemtoReg,
18.     ALUOp,
19.     MemWrite,
20.     ALUSrc,
21.     RegWrite,
22.     Jump,
23.     Branch
24. );
25.

```



```

26. initial begin
27.     Op = 6'd54;
28.     #10
29.     Op = 6'd39;
30.     #10
31.     Op = 6'd40;
32.     #10
33.     Op = 6'd41;
34.     #10
35.     Op = 6'd42;
36.
37.     #10
38.     Op = 6'd31;
39.     #10
40.     Op = 6'd32;
41.
42.     #10
43.     Op = 6'd00;
44. end
45.
46. endmodule

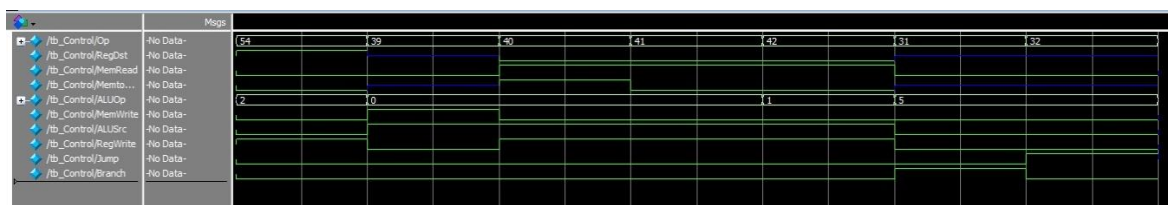
```

(2)解釋

除了先前訂一個 Op Code 之外，多增加測試 beq 以及 j 的指令。

3. 模擬結果

(1)模擬結果



(2)分析

測試結果與預期相同，因此模組設計正確。

(五) DM.v

1. 程式碼

(1)程式碼

```

1. module DM(input clk,

```

```

2.      input [31:0] addr,
3.      input [31:0] data,
4.      input MemRead,
5.      input MemWrite,
6.      output reg [31:0] DM_data);
7.
8.      integer i;
9.      reg [7:0]Mem[127:0];
10.
11.
12.     initial begin
13.         for(i = 0;i<128;i = i+1)begin
14.             Mem[i] = 8'd0;
15.         end
16.     end
17.
18.     always@(clk or MemWrite)begin
19.         if(clk & MemWrite)begin
20.             Mem[addr] <= data[31:24];
21.             Mem[addr + 1] <= data[23:16];
22.             Mem[addr + 2] <= data[15:8];
23.             Mem[addr + 3] <= data[7:0];
24.         end
25.     end
26.
27.     always@(*)begin
28.         if(MemRead)begin
29.             DM_data <=
30.                 {Mem[addr],Mem[addr+1],Mem[addr+2],Mem[addr+3]};
31.         end
32.     end
33. endmodule

```

(2)解釋

Data Memory 預設所有數字皆為 0，可以透過 clk 及 Memory Write 來進行寫入的動作，其餘狀態下皆可讀取。

2. Testbench

(1)程式碼

```
1. module tb_DM;
2. reg clk;
3. reg [31:0] addr;
4. reg [31:0] data;
5. reg MemRead;
6. reg MemWrite;
7. wire [31:0] DM_data;
8.
9. DM uut(
10.     clk,
11.     addr,
12.     data,
13.     MemRead,
14.     MemWrite,
15.     DM_data
16. );
17.
18. initial begin
19.     clk = 0;
20.     #10
21.     clk <= !clk;
22.     MemRead <= 0;
23.     MemWrite <= 1;
24.     data <= 32'd10;
25.     addr <= 32'd10;
26.     #10
27.     clk <= !clk;
28.     #10
29.     clk <= !clk;
30.     MemRead <= 1;
31.     MemWrite <= 0;
32.     addr <= 32'd10;
33.     #10
34.     clk <= !clk;
35.
```

```

36. end
37.
38. endmodule

```

(2)解釋

測試寫入一記憶體值，並將其讀取出來。

3. 模擬結果

(1)模擬結果



(2)分析

Data Memory 可以順利的寫入及讀取資料。

(六) IM.v

1. 程式碼

(1)程式碼

```

1. module IM(input [31:0] Addr_in,
2.           output reg [31:0] instr);
3.
4.   reg [31:0] Instr[31:0];
5.
6.   initial begin
7.       Instr[0] = {6'd54,5'd9,5'd10,5'd8,5'd0,6'd21};
8.       Instr[1] = {6'd54,5'd9,5'd10,5'd9,5'd0,6'd22};
9.       Instr[2] = {6'd54,5'd13,5'd11,5'd11,5'd0,6'd23};
10.      Instr[3] = {6'd54,5'd11,5'd13,5'd13,5'd0,6'd24};
11.      Instr[4] = {6'd54,5'd0,5'd12,5'd11,5'd2,6'd25};
12.      Instr[5] = {6'd54,5'd0,5'd15,5'd13,5'd5,6'd26};
13.
14.      Instr[6] = {6'd39,5'd11,5'd8,16'd2};
15.      Instr[7] = {6'd40,5'd11,5'd19,16'd2};
16.      Instr[8] = {6'd40,5'd11,5'd20,16'd3};
17.      Instr[9] = {6'd39,5'd10,5'd8,16'd2};
18.      Instr[10] = {6'd39,5'd9,5'd20,16'd4};
19.      Instr[11] = {6'd41,5'd20,5'd21,16'd40};
20.      Instr[12] = {6'd41,5'd21,5'd22,16'd22};

```

```

21. Instr[13] = {6'd42,5'd22,5'd19,16'd8};
22. Instr[14] = {6'd42,5'd16,5'd23,16'd2};
23.
24. //beq
25. Instr[15] = {6'd31,5'd12,5'd14,16'd4};
26. Instr[16] = {6'd31,5'd12,5'd24,16'd4};
27. Instr[17] = {6'd31,5'd19,5'd21,16'd4};
28. Instr[18] = {6'd31,5'd24,5'd25,16'd1};
29.
30. //j
31. Instr[19] = {6'd32,26'd125};
32. Instr[20] = {6'd32,26'd18};
33.
34. instr = Instr[0];
35. end
36.
37. always@(Addr_in)begin
38.     instr <= Instr[Addr_in>>2];
39. end
40.
41. endmodule

```

(2)解釋

新增了指令位址 60 至 80，也就是 beq 以及 j 指令的部分，並確保先前的資料依然正確。

2. Testbench

(1)程式碼

```

1. module tb_IM;
2. reg [31:0] Addr_in;
3. wire [31:0] instr;
4. integer i;
5.
6. IM uut(
7.     Addr_in,
8.     instr
9. );
10.

```

```

11. initial begin
12.     for(i=0;i<=88;i=i+4)begin
13.         #10
14.         Addr_in = i;
15.     end
16.     #10
17.     Addr_in = 0;
18.
19. end
20. endmodule

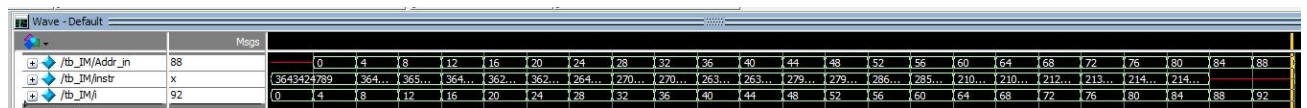
```

(2)解釋

依序測試記憶體中的所有指令。

3. 模擬結果

(1)模擬結果



(2)分析

所有指令皆可以正常的輸出。

(七) MUX5b.v

1. 程式碼

(1)程式碼

```

1. module MUX5b(
2.     input [4:0] data1,
3.     input [4:0] data2,
4.     input select,
5.     output [4:0] data_o
6. );
7.
8. assign data_o = select?data2:data1;
9.
10. endmodule

```

(2)解釋

本模組為一個 5bit 的多工器，透過 select 信號可以選擇 data2 或 data1 輸出。

2. Testbench

(1)程式碼

```

1. module tb_MUX5b;

```

```

2. reg [4:0] data1;
3. reg [4:0] data2;
4. reg select;
5.
6. wire [4:0] data_o;
7.
8. MUX5b uut(
9.     data1,
10.    data2,
11.    select,
12.    data_o
13.);
14.
15.initial begin
16.    data1 = 5'd12;
17.    data2 = 5'd37;
18.    select = 1'b1;
19.    #10
20.    select = 1'b0;
21.
22.    #10
23.    select = 1'b1;
24.end
25.
26.endmodule

```

(2)解釋

本 Testbench 測試個別放入兩個不同的 data1 及 data2，並透過 select 信號測試是否能正確的輸出 data1 與 data2。

3. 模擬結果

(1)模擬結果

| | Msgs | |
|------------------|-----------|----|
| /tb_MUX5b/data1 | -No Data- | 12 |
| /tb_MUX5b/data2 | -No Data- | 5 |
| /tb_MUX5b/select | -No Data- | |
| /tb_MUX5b/data_o | -No Data- | 5 |

(2)分析

經過 Testbench 測試，本模組的輸出可透過 select 控制，是一個正確的多工器。

(八) MUX32b.v

1. 程式碼

(1)程式碼

```
1. module MUX32b(  
2.     input [31:0] data1,  
3.     input [31:0] data2,  
4.     input select,  
5.     output [31:0] data_o  
6. );  
7.  
8. assign data_o = select?data2:data1;  
9. // assign data_o = data1;  
10.  
11.endmodule
```

(2)解釋

本模組為一個 32bit 的多工器，透過 select 信號可以選擇 data2 或 data1 輸出。

2. Testbench

(1)程式碼

```
1. module tb_MUX32b;  
2. reg [31:0] data1;  
3. reg [31:0] data2;  
4. reg select;  
5.  
6. wire [31:0] data_o;  
7.  
8. MUX32b uut(  
9.     data1,  
10.    data2,  
11.    select,  
12.    data_o  
13.);  
14.  
15.initial begin  
16.    data1 = 32'd314159;  
17.    data2 = 32'd123456;  
18.    select = 1'b1;  
19.    #10  
20.    select = 1'b0;
```



```

21.
22.     #10
23.     select = 1'b1;
24. end
25.
26. endmodule

```

(2)解釋

本 Testbench 測試個別放入兩個不同的 data1 及 data2，並透過 select 信號測試是否能正確的輸出 data1 與 data2。

3. 模擬結果

(1)模擬結果

| | Msgs | |
|-------------------|-----------|--------|
| /tb_MUX32b/data1 | -No Data- | 314159 |
| /tb_MUX32b/data2 | -No Data- | 123456 |
| /tb_MUX32b/select | -No Data- | |
| /tb_MUX32b/data_o | -No Data- | 123456 |
| | | 314159 |

(2)分析

經過 Testbench 測試，本模組的輸出可透過 select 控制，是一個正確的多工器。

4. 程式碼

(1)程式碼

```

1. module RF(input clk,
2.             input RegWrite,
3.             input [4:0] RSaddr,
4.             input [4:0] RTaddr,
5.             input [4:0] RDaddr,
6.             input [31:0] RDdata,
7.             output reg [31:0] RTdata,
8.             output reg [31:0] src1);
9.
10.
11.     reg [31:0] REGISTER[31:0];
12.
13.     initial begin
14.         REGISTER[0] = 32'd0;
15.         REGISTER[1] = 32'd11;
16.         REGISTER[2] = 32'd370;
17.         REGISTER[3] = 32'd183;
18.         REGISTER[4] = 32'd91;

```

```

19.     REGISTER[5]  = 32'd234;
20.     REGISTER[6]  = 32'd53;
21.     REGISTER[7]  = 32'd127;
22.     REGISTER[8]  = 32'd317;
23.     REGISTER[9]  = 32'd179;
24.     REGISTER[10] = 32'd101;
25.     REGISTER[11] = 32'd161;
26.     REGISTER[12] = 32'd152;
27.     REGISTER[13] = 32'd39;
28.     REGISTER[14] = 32'd39;
29.     REGISTER[15] = 32'd44;
30.     REGISTER[16] = 32'd29;
31.     REGISTER[17] = 32'd334;
32.     REGISTER[18] = 32'd245;
33.     REGISTER[19] = 32'd19;
34.     REGISTER[20] = 32'd2;
35.     REGISTER[21] = 32'd13;
36.     REGISTER[22] = 32'd262;
37.     REGISTER[23] = 32'd185;
38.     REGISTER[24] = 32'd180;
39.     REGISTER[25] = 32'd180;
40.     REGISTER[26] = 32'd198;
41.     REGISTER[27] = 32'd178;
42.     REGISTER[28] = 32'd235;
43.     REGISTER[29] = 32'd22;
44.     REGISTER[30] = 32'd1000;
45.     REGISTER[31] = 32'd75;
46. end
47.
48. always@(posedge clk or RSaddr or RTaddr or RDaddr)begin
49.     RTdata <= REGISTER[RTaddr];
50.     src1   <= REGISTER[RSaddr];
51. end
52.
53. always@(negedge clk)begin
54.     if (RegWrite)begin
55.         REGISTER[RDaddr] <= RDdata;
56.     end

```

```
57.     end
58. endmodule
```

(2)解釋

透過本模組，可以對佔存器進行讀取或是寫入的動作，當信號為正緣 clk，或是 address 變動時，會進行讀取的動作；當信號為負緣 clk，則進行寫入的動作

5. TestBench

(1)程式碼

```
1. module tb_RF;
2.
3. // RF Inputs
4. reg  clk = 0 ;
5. reg  RegWrite = 0 ;
6. reg  [4:0]  RSaddr = 0 ;
7. reg  [4:0]  RTaddr = 0 ;
8. reg  [4:0]  RDaddr = 0 ;
9. reg  [31:0] RDdata = 0 ;
10.
11. // RF Outputs
12. wire [31:0] RTdata;
13. wire [31:0] srcl;
14. integer i;
15.
16. RF u_RF(
17.     clk,
18.     RegWrite,
19.     RSaddr,
20.     RTaddr,
21.     RDaddr,
22.     RDdata,
23.     RTdata,
24.     srcl
25. );
26.
27. initial
28. begin
29.     clk = 1;
30.     for(i=0;i<32;i=i+1)begin
```

```

31.    #10
32.    clk = !clk;
33.    #10
34.    clk = !clk;
35.    RSaddr = i;
36.    RTaddr = 31-i;
37.    end
38.
39.    RegWrite = 1;
40.    for(i=0;i<32;i=i+1)begin
41.        #10
42.        clk = !clk;
43.        #10
44.        clk = !clk;
45.        RDaddr = i;
46.        RDdata = i*10;
47.        end
48.
49.        for(i=0;i<32;i=i+1)begin
50.            #10
51.            clk = !clk;
52.            #10
53.            clk = !clk;
54.            RSaddr = i;
55.            end
56.
57.    end
58.
59. endmodule

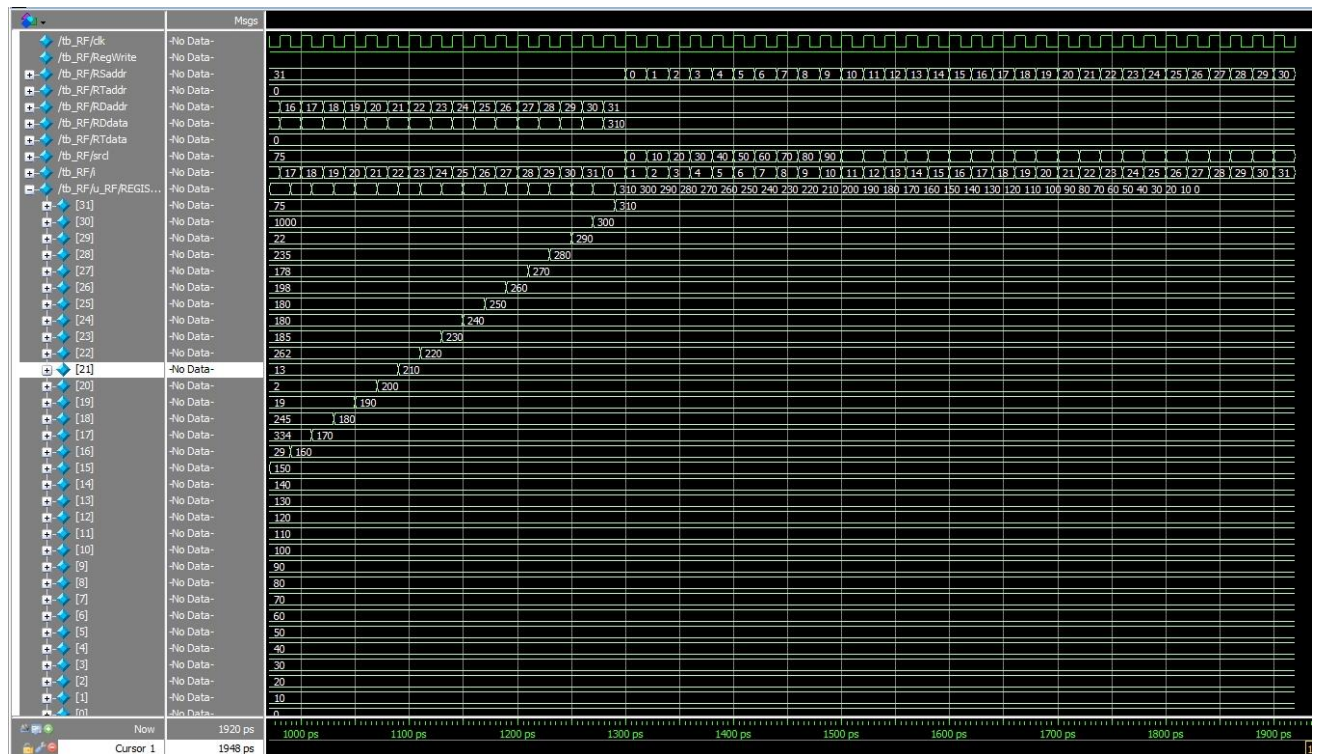
```

(2)解釋

透過 for 迴圈測試對於寄存器的讀取以及寫入動作。

6. 模擬結果

(1)模擬結果



(2)模擬結果分析

經過 Testbench，可以觀察到無論是讀取或是寫入的動作，RF 皆能正常運作。

(九) SE.v

1. 程式碼

(1)程式碼

```
8. module SE(
9.     input [15:0] data_i,
10.    output [31:0] data_o
11.);
12.assign data_o = data_i[15]?{16'hffff,data_i}:{16'h0000,data_i};
13.
14.endmodule
```

(2)解釋

本模組可以使輸入的 16bit 有號數擴展成 32bit 有號數。

2. Testbench

(1)程式碼

```
19.module tb_SE;
20.reg [15:0] data_i;
21.wire [31:0] data_o;
22.integer i;
```

```

23.
24. SE uut(
25.     data_i,
26.     data_o
27.);
28.
29. initial begin
30.     for(i=0;i<16'hffff;i=i+5)begin
31.         #10
32.         data_i = i;
33.     end
34. end
35.
36. endmodule

```

(2)解釋

透過 for 迴圈測試是否可以順利的處理正號與負號的狀態。

3. 模擬結果

(1)模擬結果

| Wave - Default | | | Msgs | | | | | | | | | | | | | | | |
|----------------|----|--|------|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| /tb_SE/data_i | 75 | | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 |
| /tb_SE/data_o | 75 | | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 |
| /tb_SE/i | 80 | | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 |

| Wave - Default | | | Msgs | | | | | | | | | | | | | | | |
|----------------|--------|--|--------|--------|--|--------|--|--------|--|--------|--|--------|--|--------|--|--------|--|--|
| /tb_SE/data_i | -22701 | | -22756 | -22751 | | -22746 | | -22741 | | -22736 | | -22731 | | -22726 | | -22721 | | |
| /tb_SE/data_o | -22701 | | -22756 | -22751 | | -22746 | | -22741 | | -22736 | | -22731 | | -22726 | | -22721 | | |
| /tb_SE/i | 42840 | | 42785 | 42790 | | 42795 | | 42800 | | 42805 | | 42810 | | 42815 | | 42820 | | |

(2)分析

經測試，正負兩種情形的運作皆正常。

(十) Shiftleft2.v

1. 程式碼

(1)程式碼

```

1. module Shiftleft2(
2.     input [31:0]data_i,
3.     output [31:0]data_o
4. );
5. assign data_o = data_i << 2;
6. endmodule

```

(2)解釋

將輸入的 32bit 值左移兩格，並輸出。

2. Testbench

(1)程式碼

```
1. module tb_Shiftleft2;
2. reg [31:0]data_i;
3. wire [31:0]data_o;
4.
5. Shiftleft2 uut(
6.     data_i,
7.     data_o
8. );
9.
10. initial begin
11.     data_i = 32'd132;
12.     #10
13.     data_i = 32'd111;
14.     #10
15.     data_i = 32'd25;
16.     #10
17.     data_i = 0;
18. end
19.
20. endmodule
```

(2)解釋

輸入依序放入 3 個數字，預期模組會將輸入的數字左移兩個位元。

3. 模擬結果

(1)模擬結果

| | Msgs | | | | | | | | | | | | | |
|----------------------|-----------|-----|--|--|--|-----|--|--|--|-----|--|--|--|--|
| tb_Shiftleft2/data_i | No Data-- | 132 | | | | 111 | | | | 25 | | | | |
| tb_Shiftleft2/data_o | No Data-- | 528 | | | | 444 | | | | 100 | | | | |

(2)分析

輸出都等於輸入乘以 4，也就是左移兩個位元。

(十一) SingleCPU.v

1. 程式碼

(1)程式碼

```
1. module SingleCPU(  
2.     input [31:0] Addr_in,  
3.     input clk,  
4.     output [31:0] Addr_o  
5. );  
6.  
7. //adder out  
8. wire [31:0]adder_o;  
9.  
10. //im out  
11. wire [31:0]instruction;  
12.  
13. //mux5b out  
14. wire [4:0]RDAddr;  
15.  
16. //control out  
17. wire RegDst;  
18. wire MemRead;  
19. wire MemtoReg;  
20. wire [2:0]ALUOp;  
21. wire MemWrite;  
22. wire ALUSrc;  
23. wire RegWrite;  
24. wire Jump;  
25. wire Branch;  
26.  
27. //se out  
28. wire [31:0]se_out;  
29.  
30. // RF_in  
31. wire [31:0]write_data;  
32. // RF out  
33. wire [31:0]register1_out;  
34. wire [31:0]register2_out;  
35.  
36. //mux32b1 out  
37. wire [31:0]mux32b1_out;
```



```

38.
39. //Aluctrl out
40. wire [5:0]operation;
41.
42. //alu out
43. wire [31:0]alu_result;
44. wire zero;
45.
46. //dm out
47. wire [31:0]dm_out;
48.
49. // alu1 in
50. wire [31:0]alu1_in;
51.
52. //alu1 out
53. wire [31:0]alu1_result;
54. wire zero1;
55.
56. // and_out
57. wire and_out;
58.
59. wire [31:0]mux32b3_o;
60.
61. Adder adder(
62.     .data1(4),
63.     .data2(Addr_in),
64.     .data_o(add_o)
65. );
66.
67. IM im(
68.     .Addr_in(Addr_in),
69.     .instr(instruction)
70. );
71.
72. MUX5b mux5b(
73.     .data1(instruction[20:16]),
74.     .data2(instruction[15:11]),
75.     .select(RegDst),

```

```

76.     .data_o(RDaddr)
77. );
78.
79. SE se(
80.     .data_i(instruction[15:0]),
81.     .data_o(se_out)
82. );
83.
84. Control control(
85.     .Op(instruction[31:26]),
86.     .RegDst(RegDst),
87.     .MemRead(MemRead),
88.     .MemtoReg(MemtoReg),
89.     .ALUOp(ALUOp),
90.     .MemWrite(MemWrite),
91.     .ALUSrc(ALUSrc),
92.     .RegWrite(RegWrite),
93.     .Jump(Jump),
94.     .Branch(Branch)
95. );
96.
97. RF rf(
98.     .clk(clk),
99.     .RegWrite(RegWrite),
100.     .RSaddr(instruction[25:21]),
101.     .RTaddr(instruction[20:16]),
102.     .RDaddr(RDaddr),
103.     .RDdata(write_data),
104.     .RTdata(register2_out),
105.     .srcl(register1_out)
106. );
107.
108. MUX32b mux32b1(
109.     .data1(register2_out),
110.     .data2(se_out),
111.     .select(ALUSrc),
112.     .data_o(mux32b1_out)
113. );

```

```

114.
115.     ALU alu(
116.         .shamt(instruction[10:6]),
117.         .src1(register1_out),
118.         .src2(mux32b1_out),
119.         .operation(operation),
120.         .result(alu_result),
121.         .zero(zero)
122.     );
123.
124.     ALUctrl aluctrl(
125.         .funct(instruction[5:0]),
126.         .ALUOp(ALUOp),
127.         .operation(operation)
128.     );
129.
130.     DM dm(
131.         .clk(clk),
132.         .addr(alu_result),
133.         .data(register2_out),
134.         .MemRead(MemRead),
135.         .MemWrite(MemWrite),
136.         .DM_data(dm_out)
137.     );
138.
139.     MUX32b mux32b2(
140.         .data1(alu_result),
141.         .data2(dm_out),
142.         .select(MemtoReg),
143.         .data_o(write_data)
144.     );
145.
146.     Shiftleft2 sl1(
147.         .data_i(se_out),
148.         .data_o(alu1_in)
149.     );
150.
151.     ALU alu1(

```

```

152.         .shamt(5'b0),
153.         .src1(alu1_in),
154.         .src2(adder_o),
155.         .operation(6'd27), //add
156.         .result(alu1_result),
157.         .zero()
158.     );
159.
160.     and (and_out,Branch,zero);
161.
162.     MUX32b mux32b3(
163.         .data1(adder_o),
164.         .data2(alu1_result),
165.         .select(and_out),
166.         .data_o(mux32b3_o)
167.     );
168.
169.     MUX32b mux32b4(
170.         .data1(mux32b3_o),
171.         .data2({adder_o[31:28],instruction[25:0],2'b00}),
172.         .select(Jump),
173.         .data_o(Addr_o)
174.     );
175.
176.     endmodule

```

(2)解釋

將上述的 model 全部接線接起來，形成完整的 CPU 模組。

2. Testbench

(1)程式碼

```

1. module tb_SingleCPU;
2. reg [31:0] Addr_in;
3. reg clk;
4. wire [31:0] Addr_o;
5.

```

```

6. integer i;
7.
8. SingleCPU uut(
9.     Addr_in,
10.    clk,
11.    Addr_o
12.);
13.
14.initial begin
15.    i <= 0;
16.    clk <= 0;
17.    #10
18.    clk <= !clk;
19.    for(i=0;i<88;i=i+4)begin
20.        #10
21.        clk <= !clk;
22.        #10
23.        clk <= !clk;
24.        Addr_in <= i;
25.    end
26.
27.end
28.
29.endmodule

```

(2)解釋

為了方便測試 jump 及 beq 指令，所以我先將 Address_in 與 out 獨立開來，手動放資料給 Address_in，可以方便觀察 Address_out 的位子

3. 助教提供之 Testbench

(1)程式碼

```

1. `timescale 1ns / 1ps
2.
3. module SingleCPU_tb;
4.
5.     // Inputs
6.     reg [31:0] Addr_in;
7.     reg clk;

```

```

8.
9.     // Outputs
10.    wire [31:0] Addr_o;
11.
12.    // Instantiate the Unit Under Test (UUT)
13.    SingleCPU uut (
14.        .Addr_in(Addr_in),
15.        .clk(clk),
16.        .Addr_o(Addr_o)
17.    );
18.
19.    initial begin
20.        // Initialize Inputs
21.        Addr_in = 0;
22.        clk = 0;
23.
24.        // Wait 100 ns for global reset to finish
25.        #50000 $finish;
26.
27.        // Add stimulus here
28.    end
29.
30.    always begin
31.        #10 clk <= ~clk;
32.        #10 clk <= ~clk;
33.        Addr_in <= Addr_o;
34.    end
35.
36. endmodule

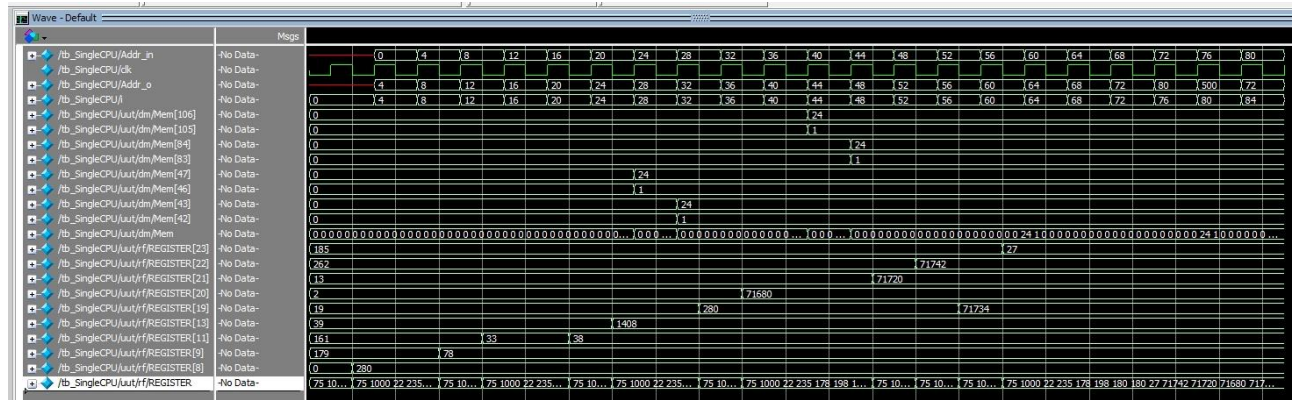
```

(2)解釋

指定 Addr_in 的值為 0，並不斷的將輸出的 Address 拉回來轉發成輸入的 Address

4. 模擬結果(自己的 Testbench)

(1)模擬結果



(2)分析

結果如預期，會執行跳轉，指令，先前指令也都執行正常。

肆、階段 3(b)

一、程式功能

(一) C 語言

```
1. int Num1 = 666;
2. int Num2 = 777;
3. int Num3;
4. Num1 = Num1 + 111;
5. Num2 = NUM2 - 111;
6. if(Num1 > Num2)
7.     Num3 = 888;
8. else
9.     Num3 = 555;
```

(二) 組合語言

```
1. addi $s0, $zero, 666
2. # num1 = 666
3.
4. addi $s1, $zero, 777
5. # num2 = 777
6.
7. addi $s0, $s0, 111
8. # num1 = num1 + 111
9.
10. subi $s1, $s1, 111
11. # num2 = num2 - 111
12.
13. sub $t0, $s2, $s1
14. # t0 = s2 - s1
15.
16. srl $t0, $t0, 31
17. # $t0 = $t0 >> 31
18.
19. addi $t1, $zero, 1
20. # $t1 = 1
21.
22. beq $t0, $t1, 2
23. # if ($t0 == $t1 == 1) , jump
```



```

24.
25. addi $s2 $zero, 555
26. # s2 = 555 (no jump)
27.
28. beq $t0,$t0,1
29.
30. addi $s2 $zero, 888
31. # s2 = 888 (jump)
32.
33. addi $t0, $zero, 0
34. sw $s0, 0, $t0
35. sw $s1, 4, $t0
36. sw $s2, 8, $t0
37. # 寫入資料到 memory

```

二、程式碼

(一) IM.v

1. 程式碼

```

1. module IM(input [31:0] Addr_in,
2.             output reg [31:0] instr);
3.
4.     reg [31:0] Instr[31:0];
5.
6.     initial begin
7.         Instr[0] = {6'd41,5'd0,5'd16,16'd666};
8.         Instr[1] = {6'd41,5'd0,5'd17,16'd777};
9.         Instr[2] = {6'd41,5'd16,5'd16,16'd111};
10.        Instr[3] = {6'd42,5'd17,5'd17,16'd111};
11.        Instr[4] =
            {6'd54,5'd18,5'd17,5'd8,5'd0,6'd22};
12.        Instr[5] = {6'd54,5'd0,5'd8,5'd8,5'd31,6'd25};
13.        Instr[6] = {6'd41,5'd0,5'd9,16'd1};
14.
15.        Instr[7] = {6'd31,5'd8,5'd9,16'd2};
16.        Instr[8] = {6'd41,5'd0,5'd18,16'd555};
17.        Instr[9] = {6'd31,5'd0,5'd0,16'd1};
18.
19.        Instr[10] = {6'd41,5'd0,5'd18,16'd888};
20.

```

```

21.          Instr[11] = {6'd41,5'd0,5'd8,16'd0};
22.          Instr[12] = {6'd39,5'd8,5'd16,16'd0};
23.          Instr[13] = {6'd39,5'd8,5'd17,16'd4};
24.          Instr[14] = {6'd39,5'd8,5'd18,16'd8};
25.
26.          instr = Instr[0];
27.      end
28.
29.      always@(Addr_in)begin
30.          instr <= Instr[Addr_in>>2];
31.      end
32.
33.endmodule
34.
35./*
36.addi $s0, $zero, 666
37.# num1 = 666
38.
39.addi $s1, $zero, 777
40.# num2 = 777
41.
42.addi $s0, $s0, 111
43.# num1 = num1 + 111
44.
45.subi $s1, $s1, 111
46.# num2 = num2 - 111
47.
48.sub $t0, $s2, $s1
49.# t0 = s2 - s1
50.
51.srl $t0, $t0, 31
52.# $t0 = $t0 >> 31
53.
54.addi $t1, $zero, 1
55.# $t1 = 1
56.
57.beq $t0, $t1, 2
58.# if ($t0 == $t1 == 1) , jump

```

```

59.
60.addi $s2 $zero, 555
61.# s2 = 555 (no jump)
62.
63.beq $t0,$t0,1
64.
65.addi $s2 $zero, 888
66.# s2 = 888 (jump)
67.

68.addi $t0, $zero, 0
69.sw $s0, 0, $t0
70.sw $s1, 4, $t0
71.sw $s2, 8, $t0
72.# 寫入資料到 memory
73.*/

```

2. 解釋

將上述的組合語言給編碼成機器語言後，放入 instruction memory 中。

(二) Testbench

1. 程式碼

```

1. module tb_SingleCPU;
2. reg [31:0] Addr_in;
3. reg clk;
4. wire [31:0] Addr_o;
5. integer i;
6.
7. SingleCPU uut(
8.     Addr_in,
9.     clk,
10.    Addr_o
11.);
12.
13.initial begin
14.    i <= 0;
15.    clk <= 0;
16.    #10
17.    clk <= !clk;

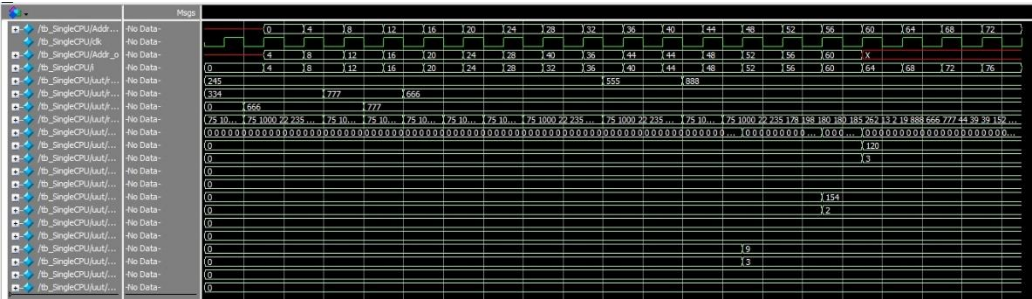
```

```
18.    for(i=0;i<80;i=i+4)begin
19.        #10
20.        clk <= !clk;
21.        #10
22.        clk <= !clk;
23.        Addr_in <= i;
24.    end
25.
26.end
27.
28.endmodule
```

2. 解釋

依序送入位址，以便執行程式。

(三) 程式執行結果



程式執行完後，Num3 的值為 888。

伍、心得

在這一次的專案中，在處理 instruction memory 時，我上網查了許多 MIPS 的資料，發現到其實正規的 MIPS 並沒有 subi 指令，而且有一部分 OP Code 與專案也有些許差異，由於手動把指令翻譯成機器碼十分的耗時，因此我也在寫專案的過程中自己透過 python 編寫了一個對於專案專用的 MIPS 語法組譯器，程式碼如下。

```
#!/usr/bin/env python
# coding: utf-8

# In[3]:

reg = {}
for i in range(32):
    if i == 0:
        reg[i] = 'zero'
    elif i == 1:
        reg[i] = 'at'
    elif i == 2 or i==3:
        reg[i] = 'v' + str(i-2)
    elif i>=4 and i<= 7:
        reg[i] = 'a' + str(i-4)
    elif i >=8 and i<=15:
        reg[i] = 't' + str(i-8)
    elif i == 24 or i==25:
        reg[i] = 't' + str(i-24+8)
    elif i >=16 and i <=23:
        reg[i] = 's' + str(i-16)
    elif i==26 or i == 27:
        reg[i] = 'k' + str(i-26)
    elif i==28:
        reg[i] = 'gp'
    elif i==29:
        reg[i] = 'sp'
    elif i==30:
        reg[i] = 'fp'
    elif i==31:
        reg[i] = 'ra'
```

```

# In[119]:

reg_a = {}
for i in reg.keys():
    reg_a[reg[i]] = i

# In[118]:

def R_convert(instr):
    op = instr[0]
    a = instr[1]
    b = instr[2]
    c = instr[3]

    funct = {
        'add' : 21,
        'sub' : 22,
        'and' : 23,
        'or' : 24,
        'srl' : 25,
        'sll' : 26
    }

    out = '{'
    # op
    if op in ['add', 'sub', 'and', 'or', 'srl', 'sll']:
        out += "6'd54,"

    # rs
    if op in ['add', 'sub', 'and', 'or']:
        out += "5'd{}",".format(reg_a[b])
    elif op in ['srl', 'sll']:
        out += "5'd0,"

    #rt

```

```

    if op in ['add', 'sub', 'and', 'or']:
        out += "5'd{}",".format(reg_a[c])
    elif op in ['srl', 'sll']:
        out += "5'd{}",".format(reg_a[b])

#rd
    if op in ['add', 'sub', 'and', 'or', 'srl', 'sll']:
        out += "5'd{}",".format(reg_a[a])

#shamt
    if op in ['add', 'sub', 'and', 'or']:
        out += "5'd0,"
    elif op in ['srl', 'sll']:
        out += "5'd{}",".format(c)

#funct
    if op in ['add', 'sub', 'and', 'or', 'srl', 'sll']:
        out += "6'd{}".format(func[op])

    out += '}'
    return out

# In[120]:

r_instr = [
    'add t0 t1 t2',
    'sub t1 t1 t2',
    'and t3 t5 t3',
    'or t5 t3 t5',
    'srl t3 t4 2',
    'sll t5 t7 5'
]

# In[122]:

def R_instrs(r_instr):

```

```

    for i in range(len(r_instr)):
        s = 'Instr[{}] = '.format(i) + R_convert(r_instr[i].split(' '))
+ ';'
        print(s)

# In[123]:

R_instrs(r_instr)

# In[75]:

def I_convert(instr):
    op = instr[0]
    if op in ['sw', 'lw']:
        rs = instr[3]
        im = instr[2]
        rt = instr[1]
    elif op in ['addi', 'subi']:
        rt = instr[1]
        rs = instr[2]
        im = instr[3]
    elif op == 'beq':
        rs = instr[1]
        rt = instr[2]
        im = instr[3]

    out = '{'

    #op
    if op == 'sw':
        out += "6'd39,"
    elif op == 'lw':
        out += "6'd40,"
    elif op == 'addi':
        out += "6'd41,"
    elif op == 'subi':

```



```

        out += "6'd42,"
    elif op == 'beq':
        out += "6'd31,"

    #rs
    if op in ['sw', 'lw', 'addi', 'subi', 'beq']:
        out += "5'd{},".format(reg_a[rs])

    #rt
    if op in ['sw', 'lw', 'addi', 'subi', 'beq']:
        out += "5'd{},".format(reg_a[rt])

    #im
    out += "16'd{}".format(im)

    out += "}"

    return out

```

```
# In[76]:
```

```

i_instr = [
    'sw t0 2 t3',
    'lw s3 2 t3',
    'lw s4 3 t3',
    'sw t0 2 t2',
    'sw s4 4 t1',
    'addi s5 s4 40',
    'addi s6 s5 22',
    'subi s3 s6 8',
    'subi s7 s0 2',
    'beq t4 t6 4',
    'beq t4 t8 4',
    'beq s3 s5 4',
    'beq t8 t9 1'
]

```

```

# In[128]:

def I_instrs(i_instr):
    for i in range(len(i_instr)):
        print('Instr[{}] = '.format(i) + I_convert(i_instr[i].split('
'))) + ';'

# In[129]:

I_instrs(i_instr)

# In[132]:

def J_convert(instr):
    op = instr[0]
    im = instr[1]
    out = "{"
    if op == 'j':
        out += "6'd32,"
        out += "26'd{}".format(im)
    out += "}"
    return out

# In[133]:

j_instr = [
    'j 125',
    'j 18'
]

# In[146]:

```

```
def J_instrs(j_instr):
    for i in range(len(j_instr)):
        print('Instr[{}] = {}'.format(i) + J_convert(j_instr[i].split('
'))) + ';'

# In[147]:

J_instrs(j_instr)
```

```
# In[152]:
```

```
a = ['addi s0 zero 666',
      'addi s1 zero 777',
      'addi s0 s0 111',
      'subi s1 s1 111'
    ]
I_instrs(a)
```

```
# In[151]:
```

```
a = [
      'sub t0 s2 s1',
      'srl t0 t0 31'
    ]
R_instrs(a)
```

```
# In[153]:
```

```
a = [
      'addi t1 zero 1',
      'beq t0 t1 1',
      'addi s2 zero 555',
    ]
```

```
'addi s2 zero 888',  
'addi t0 zero 0',  
'sw s0 0 t0',  
'sw s1 4 t0',  
'sw s2 8 t0'  
]  
I_instrs(a)
```

```
# In[ ]:
```

非常感謝老師的教導，讓我知道處理不同 Format 的 CPU 指令，完全透過自己手刻一個 CPU 真的是一件非常有成就感的事情，很謝謝老師可以安排這種作業讓我們了解了 CPU 背後的運作方式。

今天剛好看到了一篇新聞，提到中美貿易戰，川普下令讓美國的企業高通、Google、intel 等公司對華為停止供貨，如果這篇新聞屬實的話，代表華為公司的手機、伺服器等产品最重要的系統核心部分就無法透過美國的技術，需要自己研發了。但是經過我自己設計了一次 CPU 後，感覺連個普通的大學生都有辦法寫出最基本的 CPU，我想華為應該也有能力可以自己研發出他們專用的 CPU 吧？或許研發的效能不及 intel 及高通，但基本的使用我覺得應該是沒有問題的吧？

謝謝助教在這次 Project 上面的協助，透過 email 問助教問題，基本上助教都是秒回的，回應時間非常非常的快。而這次的作業批改還需要看我這邊寫的 100 頁報告，真是辛苦助教了，謝謝！

這一次的作業我差不多花了 10 個小時寫程式與 debug，花了 6 個小時寫報告，加起來總共寫了 16 個小時，感覺難度好像比前一次作業，乘除法器簡單了許多！