PA2: Single Cycle MIPS CPU

Part3:

Area: 14188.09μm^2 slack: 4.364

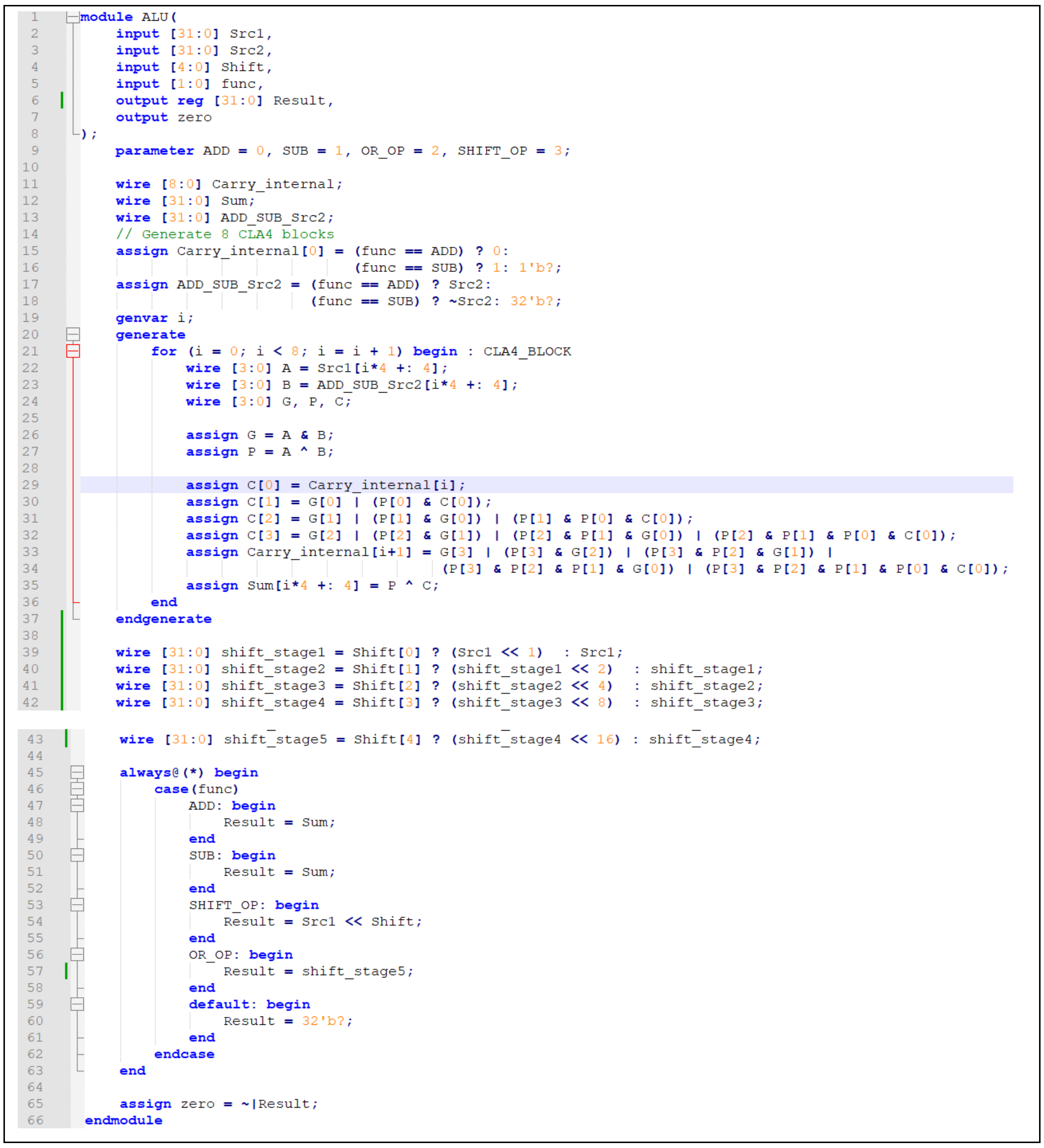
學生：李勁磊

學號：B11107048

一、Screenshots and descriptions of each module:

ALU:

利用自己寫的 adder 跟 shift 功能以減小面積

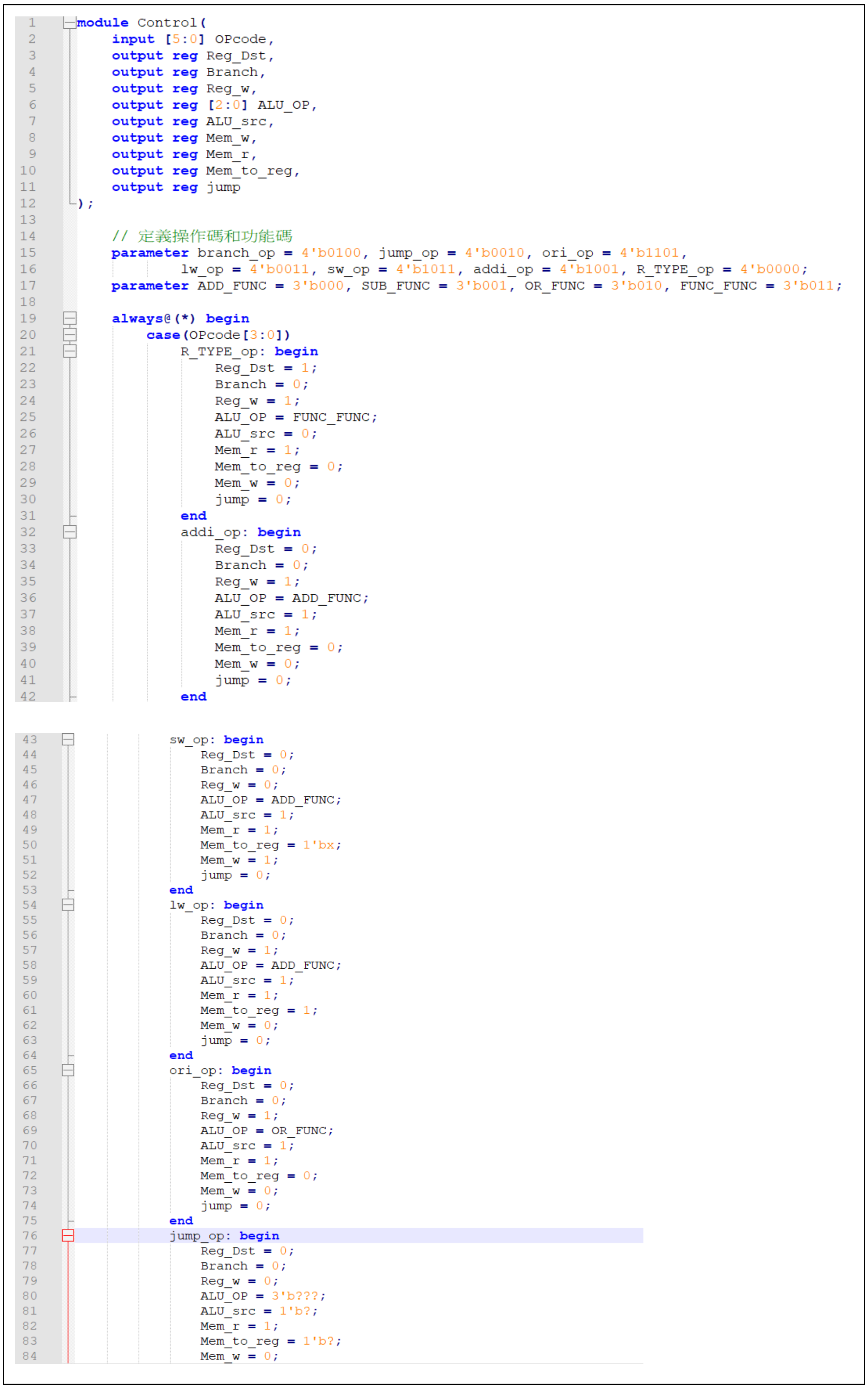


```
1  module ALU(
2      input [31:0] Src1,
3      input [31:0] Src2,
4      input [4:0] Shift,
5      input [1:0] func,
6      output reg [31:0] Result,
7      output zero
8  );
9      parameter ADD = 0, SUB = 1, OR_OP = 2, SHIFT_OP = 3;
10
11     wire [8:0] Carry_internal;
12     wire [31:0] Sum;
13     wire [31:0] ADD_SUB_Src2;
14     // Generate 8 CLA4 blocks
15     assign Carry_internal[0] = (func == ADD) ? 0:
16                                (func == SUB) ? 1: 1'b?;
17     assign ADD_SUB_Src2 = (func == ADD) ? Src2:
18                           (func == SUB) ? ~Src2: 32'b?;
19     genvar i;
20     generate
21         for (i = 0; i < 8; i = i + 1) begin : CLA4_BLOCK
22             wire [3:0] A = Src1[i*4 +: 4];
23             wire [3:0] B = ADD_SUB_Src2[i*4 +: 4];
24             wire [3:0] G, P, C;
25
26             assign G = A & B;
27             assign P = A ^ B;
28
29             assign C[0] = Carry_internal[i];
30             assign C[1] = G[0] | (P[0] & C[0]);
31             assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & C[0]);
32             assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & C[0]);
33             assign Carry_internal[i+1] = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) |
34                                          (P[3] & P[2] & P[1] & G[0]) | (P[3] & P[2] & P[1] & P[0] & C[0]);
35             assign Sum[i*4 +: 4] = P ^ C;
36         end
37     endgenerate
38
39     wire [31:0] shift_stage1 = Shift[0] ? (Src1 << 1)  : Src1;
40     wire [31:0] shift_stage2 = Shift[1] ? (shift_stage1 << 2)  : shift_stage1;
41     wire [31:0] shift_stage3 = Shift[2] ? (shift_stage2 << 4)  : shift_stage2;
42     wire [31:0] shift_stage4 = Shift[3] ? (shift_stage3 << 8)  : shift_stage3;
43     wire [31:0] shift_stage5 = Shift[4] ? (shift_stage4 << 16) : shift_stage4;
44
45     always@(*) begin
46         case(func)
47             ADD: begin
48                 Result = Sum;
49             end
50             SUB: begin
51                 Result = Sum;
52             end
53             SHIFT_OP: begin
54                 Result = Src1 << Shift;
55             end
56             OR_OP: begin
57                 Result = shift_stage5;
58             end
59             default: begin
60                 Result = 32'b?;
61             end
62         endcase
63     end
64
65     assign zero = ~|Result;
66 endmodule
```

1~7	宣告接腳
14~37	執行加法運算
38~43	執行 Shift 運算
45~63	控制輸出訊號

Control:

將每個狀態的 control signal 分別寫出，讓每個 block 可以正常運作

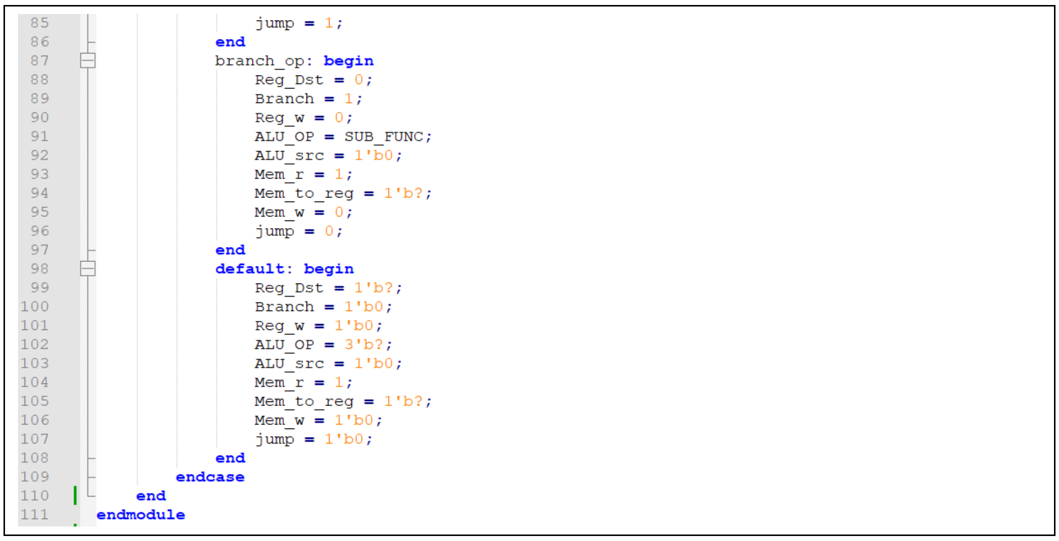


```verilog
module Control(
    input [5:0] OPcode,
    output reg Reg_Dst,
    output reg Branch,
    output reg Reg_w,
    output reg [2:0] ALU_OP,
    output reg ALU_src,
    output reg Mem_w,
    output reg Mem_r,
    output reg Mem_to_reg,
    output reg jump
);

    // 定義操作碼和功能碼
    parameter branch_op = 4'b0100, jump_op = 4'b0010, ori_op = 4'b1101,
            lw_op = 4'b0011, sw_op = 4'b1011, addi_op = 4'b1001, R_TYPE_op = 4'b0000;
    parameter ADD_FUNC = 3'b000, SUB_FUNC = 3'b001, OR_FUNC = 3'b010, FUNC_FUNC = 3'b011;

    always@(*) begin
        case(OPcode[3:0])
            R_TYPE_op: begin
                Reg_Dst = 1;
                Branch = 0;
                Reg_w = 1;
                ALU_OP = FUNC_FUNC;
                ALU_src = 0;
                Mem_r = 1;
                Mem_to_reg = 0;
                Mem_w = 0;
                jump = 0;
            end
            addi_op: begin
                Reg_Dst = 0;
                Branch = 0;
                Reg_w = 1;
                ALU_OP = ADD_FUNC;
                ALU_src = 1;
                Mem_r = 1;
                Mem_to_reg = 0;
                Mem_w = 0;
                jump = 0;
            end

            sw_op: begin
                Reg_Dst = 0;
                Branch = 0;
                Reg_w = 0;
                ALU_OP = ADD_FUNC;
                ALU_src = 1;
                Mem_r = 1;
                Mem_to_reg = 1'bx;
                Mem_w = 1;
                jump = 0;
            end
            lw_op: begin
                Reg_Dst = 0;
                Branch = 0;
                Reg_w = 1;
                ALU_OP = ADD_FUNC;
                ALU_src = 1;
                Mem_r = 1;
                Mem_to_reg = 1;
                Mem_w = 0;
                jump = 0;
            end
            ori_op: begin
                Reg_Dst = 0;
                Branch = 0;
                Reg_w = 1;
                ALU_OP = OR_FUNC;
                ALU_src = 1;
                Mem_r = 1;
                Mem_to_reg = 0;
                Mem_w = 0;
                jump = 0;
            end
            jump_op: begin
                Reg_Dst = 0;
                Branch = 0;
                Reg_w = 0;
                ALU_OP = 3'b???;
                ALU_src = 1'b?;
                Mem_r = 1;
                Mem_to_reg = 1'b?;
                Mem_w = 0;
```

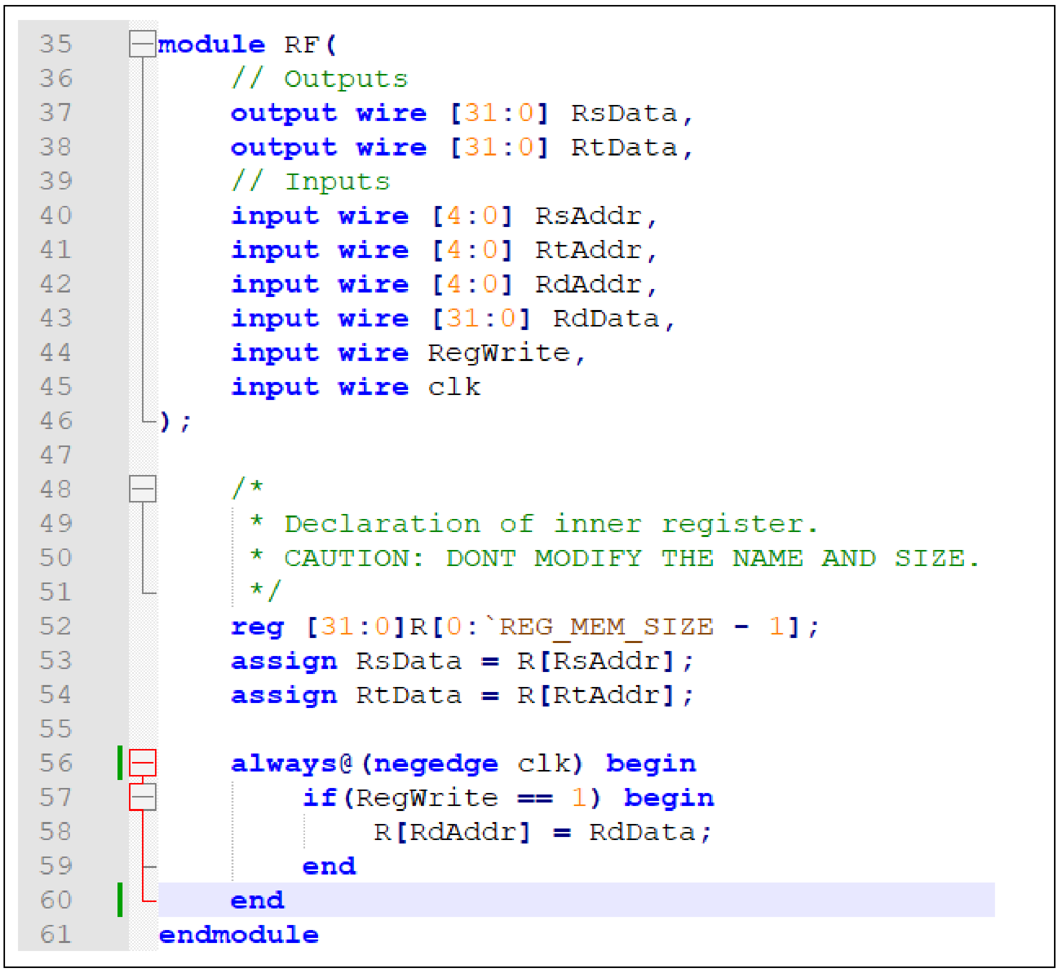


```
 85                  jump = 1;
 86              end
 87              branch_op: begin
 88                  Reg_Dst = 0;
 89                  Branch = 1;
 90                  Reg_w = 0;
 91                  ALU_OP = SUB_FUNC;
 92                  ALU_src = 1'b0;
 93                  Mem_r = 1;
 94                  Mem_to_reg = 1'b?;
 95                  Mem_w = 0;
 96                  jump = 0;
 97              end
 98              default: begin
 99                  Reg_Dst = 1'b?;
100                  Branch = 1'b0;
101                  Reg_w = 1'b0;
102                  ALU_OP = 3'b?;
103                  ALU_src = 1'b0;
104                  Mem_r = 1;
105                  Mem_to_reg = 1'b?;
106                  Mem_w = 1'b0;
107                  jump = 1'b0;
108              end
109          endcase
110      end
111  endmodule
```

1~11	宣告接腳
12~110	將每種可能寫出來

RF:

在 negedge 時把資料寫入

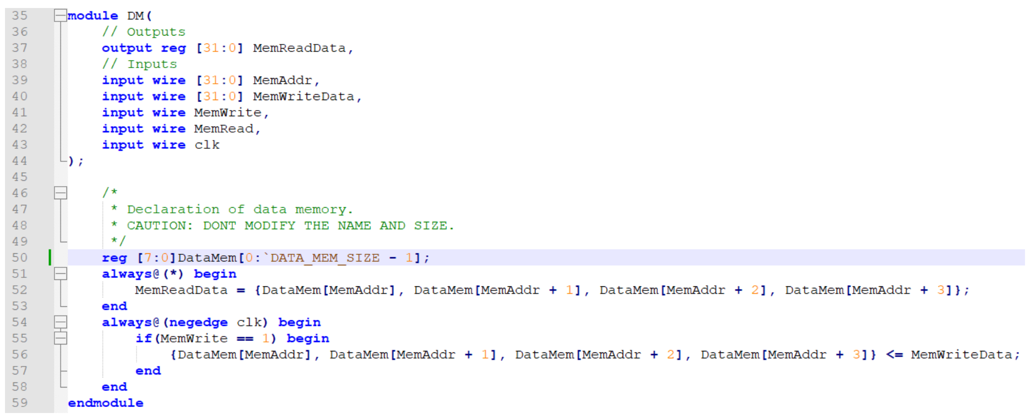


```verilog
35 module RF(
36     // Outputs
37     output wire [31:0] RsData,
38     output wire [31:0] RtData,
39     // Inputs
40     input wire [4:0] RsAddr,
41     input wire [4:0] RtAddr,
42     input wire [4:0] RdAddr,
43     input wire [31:0] RdData,
44     input wire RegWrite,
45     input wire clk
46 );
47 
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     assign RsData = R[RsAddr];
54     assign RtData = R[RtAddr];
55 
56     always@(negedge clk) begin
57         if(RegWrite == 1) begin
58             R[RdAddr] = RdData;
59         end
60     end
61 endmodule
```

35~46	宣告接腳
48~61	控制讀出跟寫入訊號

DM

讓資料讀取用 combinational 寫入用 negedge

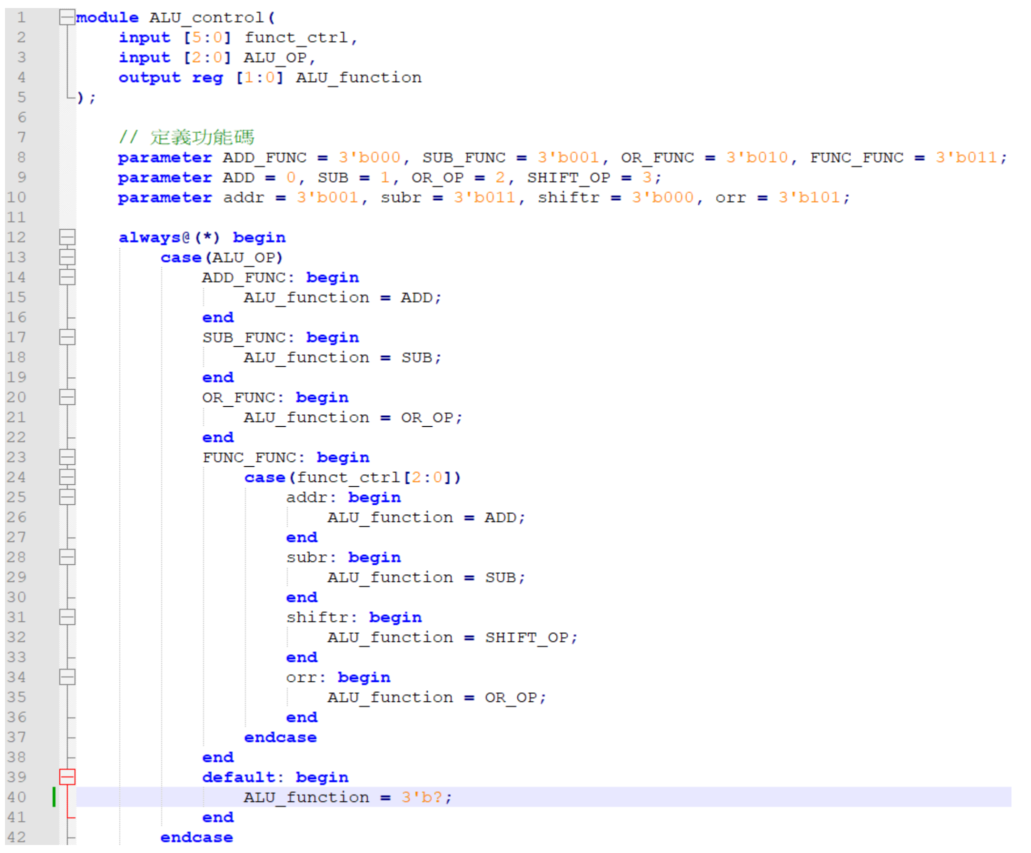


```
35 module DM(
36     // Outputs
37     output reg [31:0] MemReadData,
38     // Inputs
39     input wire [31:0] MemAddr,
40     input wire [31:0] MemWriteData,
41     input wire MemWrite,
42     input wire MemRead,
43     input wire clk
44 );
45 
46     /*
47      * Declaration of data memory.
48      * CAUTION: DONT MODIFY THE NAME AND SIZE.
49      */
50     reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
51     always@(*) begin
52         MemReadData = {DataMem[MemAddr], DataMem[MemAddr + 1], DataMem[MemAddr + 2], DataMem[MemAddr + 3]};
53     end
54     always@(negedge clk) begin
55         if(MemWrite == 1) begin
56             {DataMem[MemAddr], DataMem[MemAddr + 1], DataMem[MemAddr + 2], DataMem[MemAddr + 3]} <= MemWriteData;
57         end
58     end
59 endmodule
```

35~44	宣告接腳
49~59	控制讀出跟寫入訊號

ALU_control

根據 func_ctrl 跟 ALU_OP 看要控制 ALU 做甚麼運算

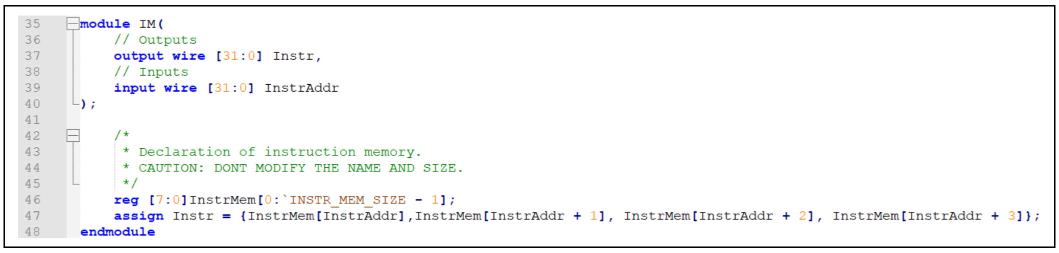


```
1  module ALU_control(
2      input [5:0] funct_ctrl,
3      input [2:0] ALU_OP,
4      output reg [1:0] ALU_function
5  );
6  
7      // 定義功能碼
8      parameter ADD_FUNC = 3'b000, SUB_FUNC = 3'b001, OR_FUNC = 3'b010, FUNC_FUNC = 3'b011;
9      parameter ADD = 0, SUB = 1, OR_OP = 2, SHIFT_OP = 3;
10     parameter addr = 3'b001, subr = 3'b011, shiftr = 3'b000, orr = 3'b101;
11 
12     always@(*) begin
13         case(ALU_OP)
14             ADD_FUNC: begin
15                 ALU_function = ADD;
16             end
17             SUB_FUNC: begin
18                 ALU_function = SUB;
19             end
20             OR_FUNC: begin
21                 ALU_function = OR_OP;
22             end
23             FUNC_FUNC: begin
24                 case(funct_ctrl[2:0])
25                     addr: begin
26                         ALU_function = ADD;
27                     end
28                     subr: begin
29                         ALU_function = SUB;
30                     end
31                     shiftr: begin
32                         ALU_function = SHIFT_OP;
33                     end
34                     orr: begin
35                         ALU_function = OR_OP;
36                     end
37                 endcase
38             end
39             default: begin
40                 ALU_function = 3'b?;
41             end
42         endcase
```

1~4	宣告接腳
12~41	將每種可能寫出來

IM:

使用 assign 把 IM 的資料傳入 CPU

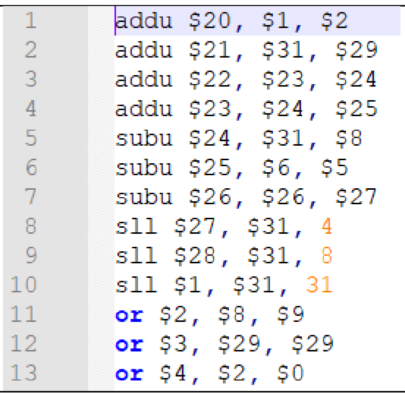


```verilog
module IM(
    // Outputs
    output wire [31:0] Instr,
    // Inputs
    input wire [31:0] InstrAddr
);

    /*
     * Declaration of instruction memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
    assign Instr = {InstrMem[InstrAddr],InstrMem[InstrAddr + 1], InstrMem[InstrAddr + 2], InstrMem[InstrAddr + 3]};
endmodule
```

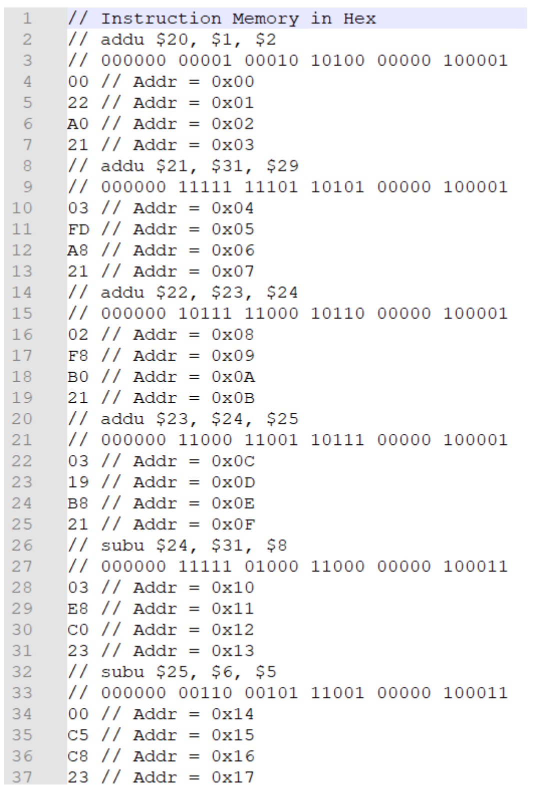
二、Descriptions test commands for each module:

R type:

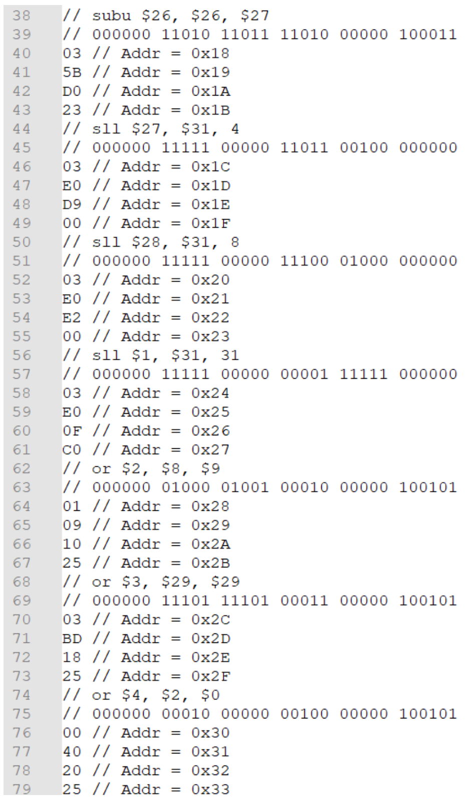


```
addu $20, $1, $2
addu $21, $31, $29
addu $22, $23, $24
addu $23, $24, $25
subu $24, $31, $8
subu $25, $6, $5
subu $26, $26, $27
sll $27, $31, 4
sll $28, $31, 8
sll $1, $31, 31
or $2, $8, $9
or $3, $29, $29
or $4, $2, $0
```

分別測試:
正常的數值相加
最大最小值的相加
正常的數值相減
小減大
0 減 0
左移
OR 功能測試

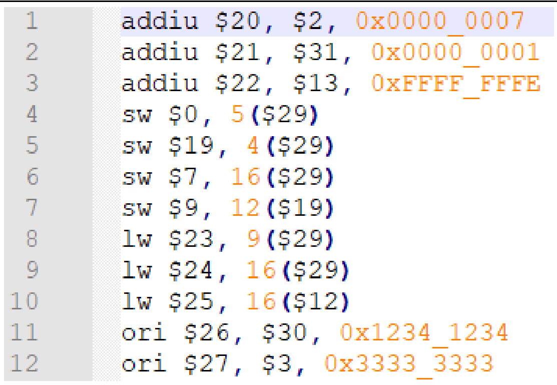


```
// Instruction Memory in Hex
// addu $20, $1, $2
// 000000 00001 00010 10100 00000 100001
00 // Addr = 0x00
22 // Addr = 0x01
A0 // Addr = 0x02
21 // Addr = 0x03
// addu $21, $31, $29
// 000000 11111 11101 10101 00000 100001
03 // Addr = 0x04
FD // Addr = 0x05
A8 // Addr = 0x06
21 // Addr = 0x07
// addu $22, $23, $24
// 000000 10111 11000 10110 00000 100001
02 // Addr = 0x08
F8 // Addr = 0x09
B0 // Addr = 0x0A
21 // Addr = 0x0B
// addu $23, $24, $25
// 000000 11000 11001 10111 00000 100001
03 // Addr = 0x0C
19 // Addr = 0x0D
B8 // Addr = 0x0E
21 // Addr = 0x0F
// subu $24, $31, $8
// 000000 11111 01000 11000 00000 100011
03 // Addr = 0x10
E8 // Addr = 0x11
C0 // Addr = 0x12
23 // Addr = 0x13
// subu $25, $6, $5
// 000000 00110 00101 11001 00000 100011
00 // Addr = 0x14
C5 // Addr = 0x15
C8 // Addr = 0x16
23 // Addr = 0x17
```



```
// subu $26, $26, $27
// 000000 11010 11011 11010 00000 100011
03 // Addr = 0x18
5B // Addr = 0x19
D0 // Addr = 0x1A
23 // Addr = 0x1B
// sll $27, $31, 4
// 000000 11111 00000 11011 00100 000000
03 // Addr = 0x1C
E0 // Addr = 0x1D
D9 // Addr = 0x1E
00 // Addr = 0x1F
// sll $28, $31, 8
// 000000 11111 00000 11100 01000 000000
03 // Addr = 0x20
E0 // Addr = 0x21
E2 // Addr = 0x22
00 // Addr = 0x23
// sll $1, $31, 31
// 000000 11111 00000 00001 11111 000000
03 // Addr = 0x24
E0 // Addr = 0x25
0F // Addr = 0x26
C0 // Addr = 0x27
// or $2, $8, $9
// 000000 01000 01001 00010 00000 100101
01 // Addr = 0x28
09 // Addr = 0x29
10 // Addr = 0x2A
25 // Addr = 0x2B
// or $3, $29, $29
// 000000 11101 11101 00011 00000 100101
03 // Addr = 0x2C
BD // Addr = 0x2D
18 // Addr = 0x2E
25 // Addr = 0x2F
// or $4, $2, $0
// 000000 00010 00000 00100 00000 100101
00 // Addr = 0x30
40 // Addr = 0x31
20 // Addr = 0x32
25 // Addr = 0x33
```

I type:



```
addiu $20, $2, 0x0000_0007
addiu $21, $31, 0x0000_0001
addiu $22, $13, 0xFFFF_FFFE
sw $0, 5($29)
sw $19, 4($29)
sw $7, 16($29)
sw $9, 12($19)
lw $23, 9($29)
lw $24, 16($29)
lw $25, 16($12)
ori $26, $30, 0x1234_1234
ori $27, $3, 0x3333_3333
```

分別測試:

正常加法

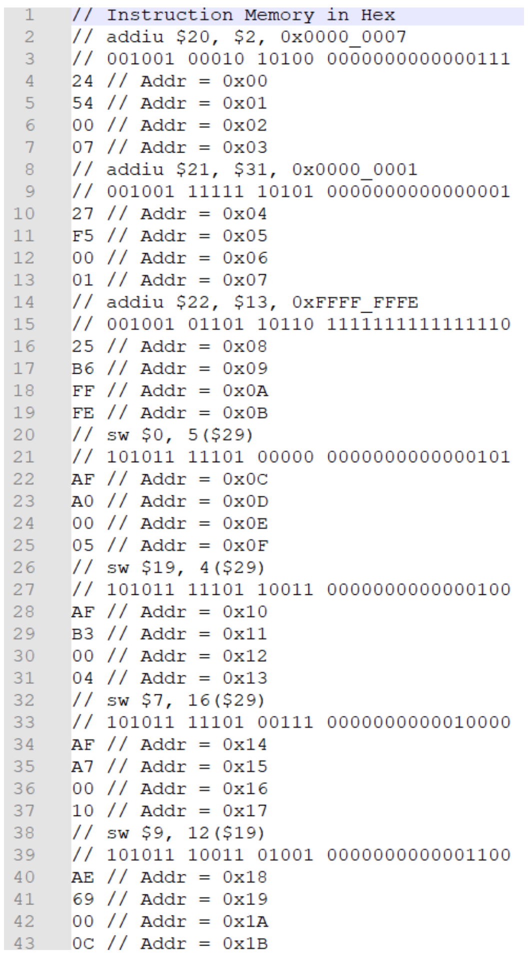
最大值加 1

零加其他數值

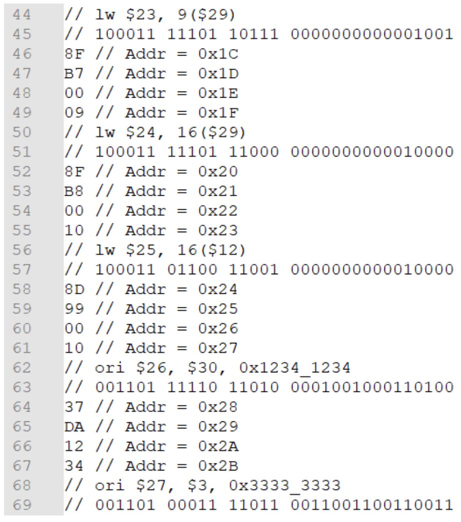
存數值

讀數值

OR 功能測試

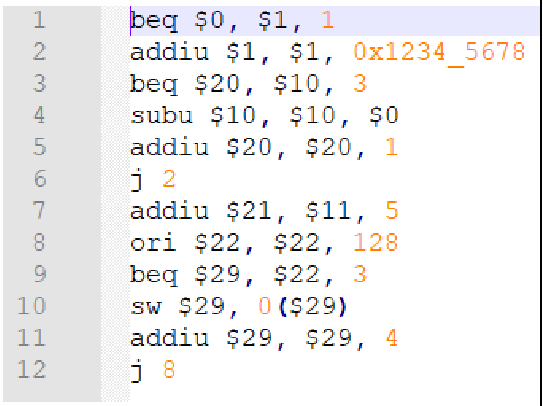


```
// Instruction Memory in Hex
// addiu $20, $2, 0x0000_0007
// 001001 00010 10100 0000000000000111
24 // Addr = 0x00
54 // Addr = 0x01
00 // Addr = 0x02
07 // Addr = 0x03
// addiu $21, $31, 0x0000_0001
// 001001 11111 10101 0000000000000001
27 // Addr = 0x04
F5 // Addr = 0x05
00 // Addr = 0x06
01 // Addr = 0x07
// addiu $22, $13, 0xFFFF_FFFE
// 001001 01101 10110 1111111111111110
25 // Addr = 0x08
B6 // Addr = 0x09
FF // Addr = 0x0A
FE // Addr = 0x0B
// sw $0, 5($29)
// 101011 11101 00000 0000000000000101
AF // Addr = 0x0C
A0 // Addr = 0x0D
00 // Addr = 0x0E
05 // Addr = 0x0F
// sw $19, 4($29)
// 101011 11101 10011 0000000000000100
AF // Addr = 0x10
B3 // Addr = 0x11
00 // Addr = 0x12
04 // Addr = 0x13
// sw $7, 16($29)
// 101011 11101 00111 0000000000010000
AF // Addr = 0x14
A7 // Addr = 0x15
00 // Addr = 0x16
10 // Addr = 0x17
// sw $9, 12($19)
// 101011 10011 01001 0000000000001100
AE // Addr = 0x18
69 // Addr = 0x19
00 // Addr = 0x1A
0C // Addr = 0x1B
```



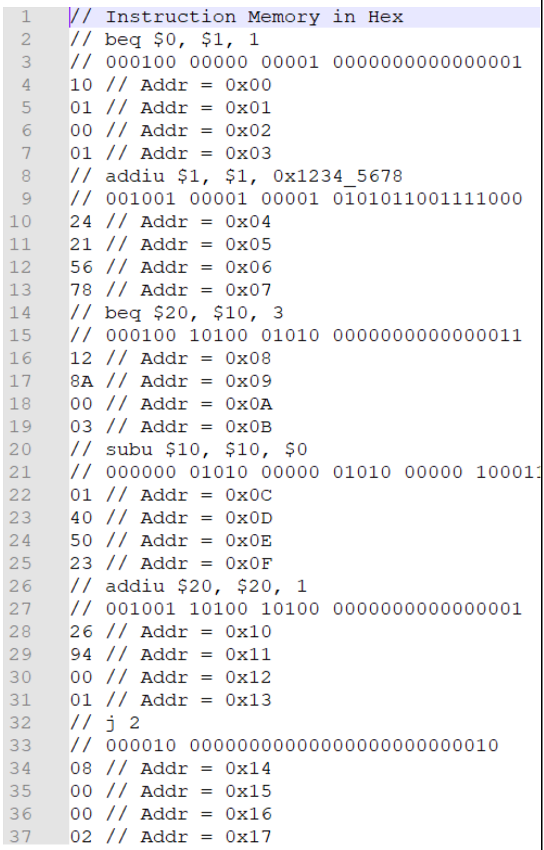
```
// lw $23, 9($29)
// 100011 11101 10111 0000000000001001
8F // Addr = 0x1C
B7 // Addr = 0x1D
00 // Addr = 0x1E
09 // Addr = 0x1F
// lw $24, 16($29)
// 100011 11101 11000 0000000000010000
8F // Addr = 0x20
B8 // Addr = 0x21
00 // Addr = 0x22
10 // Addr = 0x23
// lw $25, 16($12)
// 100011 01100 11001 0000000000010000
8D // Addr = 0x24
99 // Addr = 0x25
00 // Addr = 0x26
10 // Addr = 0x27
// ori $26, $30, 0x1234_1234
// 001101 11110 11010 0001001000110100
37 // Addr = 0x28
DA // Addr = 0x29
12 // Addr = 0x2A
34 // Addr = 0x2B
// ori $27, $3, 0x3333_3333
// 001101 00011 11011 0011001100110011
```

J type:

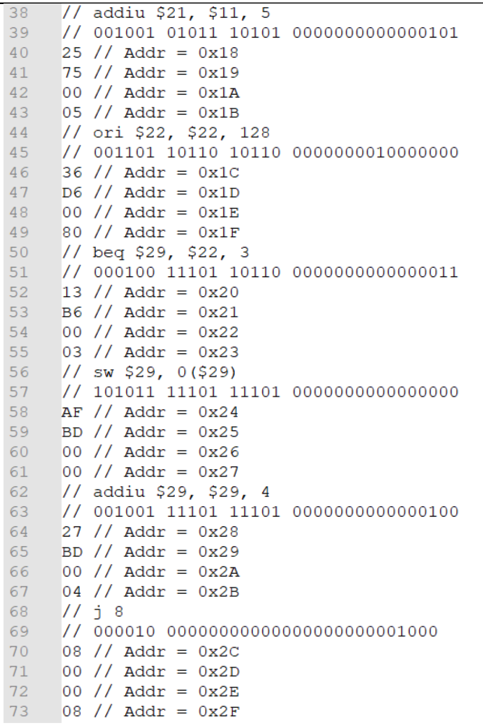


```
beq $0, $1, 1
addiu $1, $1, 0x1234_5678
beq $20, $10, 3
subu $10, $10, $0
addiu $20, $20, 1
j 2
addiu $21, $11, 5
ori $22, $22, 128
beq $29, $22, 3
sw $29, 0($29)
addiu $29, $29, 4
j 8
```

測試 beq j 跟其他功能組合的功能



```
// Instruction Memory in Hex
// beq $0, $1, 1
// 000100 00000 00001 0000000000000001
10 // Addr = 0x00
01 // Addr = 0x01
00 // Addr = 0x02
01 // Addr = 0x03
// addiu $1, $1, 0x1234_5678
// 001001 00001 00001 0101011001111000
24 // Addr = 0x04
21 // Addr = 0x05
56 // Addr = 0x06
78 // Addr = 0x07
// beq $20, $10, 3
// 000100 10100 01010 0000000000000011
12 // Addr = 0x08
8A // Addr = 0x09
00 // Addr = 0x0A
03 // Addr = 0x0B
// subu $10, $10, $0
// 000000 01010 00000 01010 00000 10001
01 // Addr = 0x0C
40 // Addr = 0x0D
50 // Addr = 0x0E
23 // Addr = 0x0F
// addiu $20, $20, 1
// 001001 10100 10100 0000000000000001
26 // Addr = 0x10
94 // Addr = 0x11
00 // Addr = 0x12
01 // Addr = 0x13
// j 2
// 000010 00000000000000000000000010
08 // Addr = 0x14
00 // Addr = 0x15
00 // Addr = 0x16
02 // Addr = 0x17
```



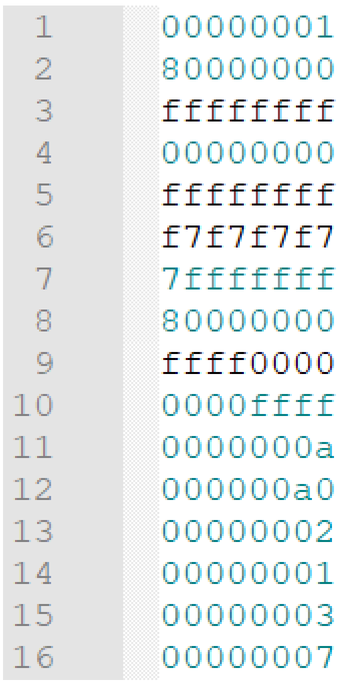
```
// addiu $21, $11, 5
// 001001 01011 10101 0000000000000101
25 // Addr = 0x18
75 // Addr = 0x19
00 // Addr = 0x1A
05 // Addr = 0x1B
// ori $22, $22, 128
// 001101 10110 10110 0000000010000000
36 // Addr = 0x1C
D6 // Addr = 0x1D
00 // Addr = 0x1E
80 // Addr = 0x1F
// beq $29, $22, 3
// 000100 11101 10110 0000000000000011
13 // Addr = 0x20
B6 // Addr = 0x21
00 // Addr = 0x22
03 // Addr = 0x23
// sw $29, 0($29)
// 101011 11101 11101 0000000000000000
AF // Addr = 0x24
BD // Addr = 0x25
00 // Addr = 0x26
00 // Addr = 0x27
// addiu $29, $29, 4
// 001001 11101 11101 0000000000000100
27 // Addr = 0x28
BD // Addr = 0x29
00 // Addr = 0x2A
04 // Addr = 0x2B
// j 8
// 000010 00000000000000000000001000
08 // Addr = 0x2C
00 // Addr = 0x2D
00 // Addr = 0x2E
08 // Addr = 0x2F
```

三、Stimulation Result:

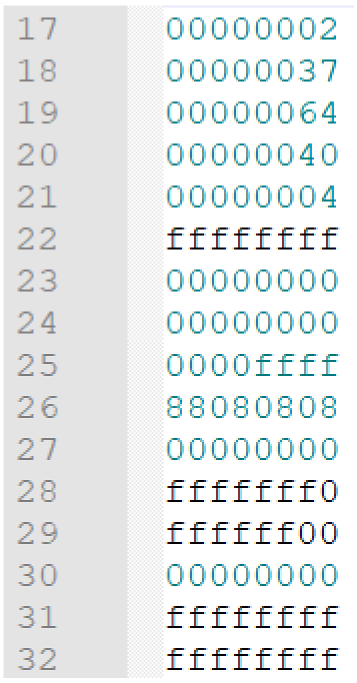
R type:

輸出符合預期

RF output



```
1   00000001
2   80000000
3   ffffffff
4   00000000
5   ffffffff
6   f7f7f7f7
7   7fffffff
8   80000000
9   ffff0000
10  0000ffff
11  0000000a
12  000000a0
13  00000002
14  00000001
15  00000003
16  00000007
```

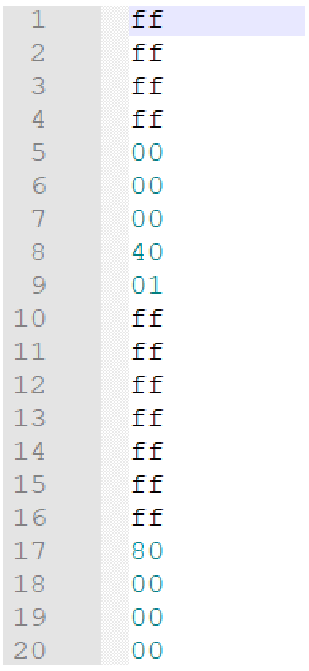


```
17  00000002
18  00000037
19  00000064
20  00000040
21  00000004
22  ffffffff
23  00000000
24  00000000
25  0000ffff
26  88080808
27  00000000
28  fffffff0
29  ffffff00
30  00000000
31  ffffffff
32  ffffffff
```

I type:

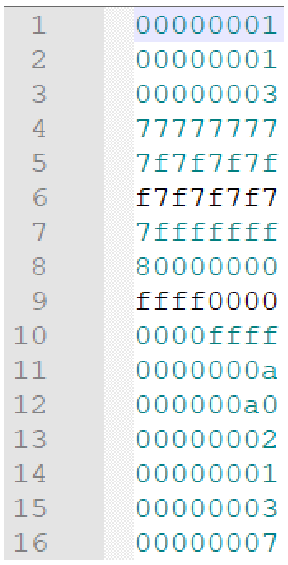
輸出結果符合預期

DM out

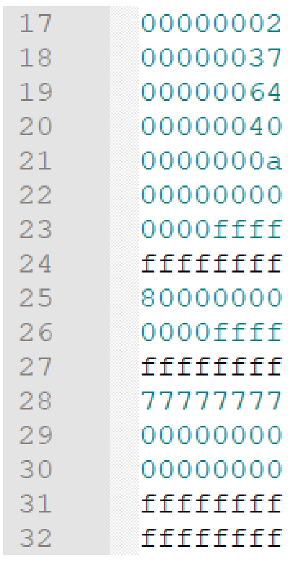


```
1   ff
2   ff
3   ff
4   ff
5   00
6   00
7   00
8   40
9   01
10  ff
11  ff
12  ff
13  ff
14  ff
15  ff
16  ff
17  80
18  00
19  00
20  00
```

RF out



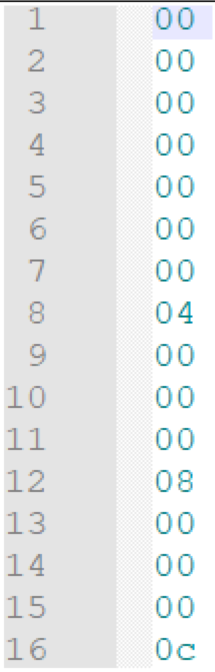
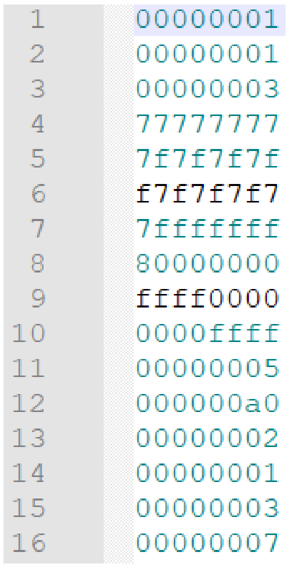
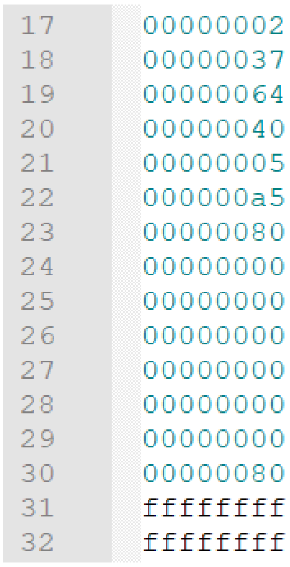
```
1   00000001
2   00000001
3   00000003
4   77777777
5   7f7f7f7f
6   f7f7f7f7
7   7fffffff
8   80000000
9   ffff0000
10  0000ffff
11  0000000a
12  000000a0
13  00000002
14  00000001
15  00000003
16  00000007
```



```
17  00000002
18  00000037
19  00000064
20  00000040
21  0000000a
22  00000000
23  0000ffff
24  ffffffff
25  80000000
26  0000ffff
27  ffffffff
28  77777777
29  00000000
30  00000000
31  ffffffff
32  ffffffff
```

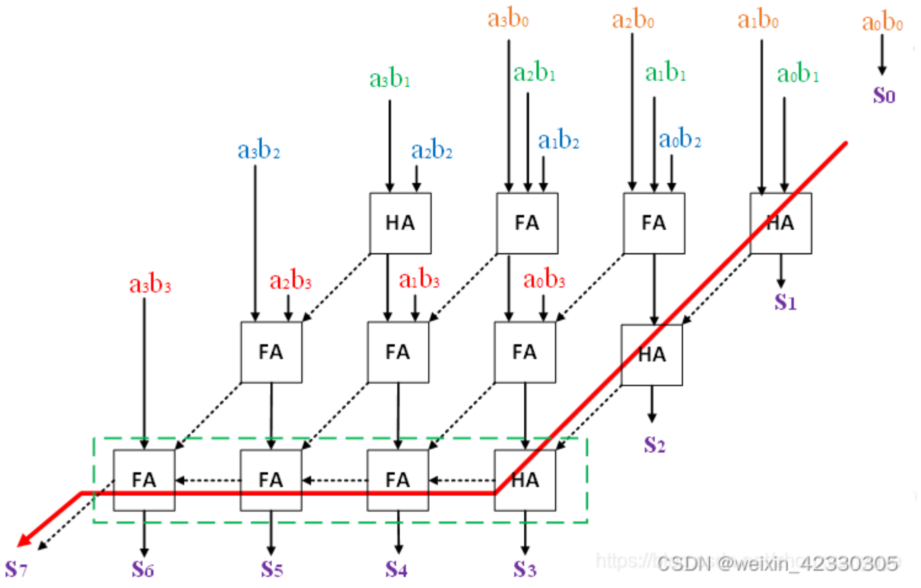
J type:

輸出結果符合預期

DM out (只截圖一部分，因為檔案太長)	RF out	
 1 00 2 00 3 00 4 00 5 00 6 00 7 00 8 04 9 00 10 00 11 00 12 08 13 00 14 00 15 00 16 0c	 1 00000001 2 00000001 3 00000003 4 77777777 5 7f7f7f7f 6 f7f7f7f7 7 7fffffff 8 80000000 9 ffff0000 10 0000ffff 11 00000005 12 000000a0 13 00000002 14 00000001 15 00000003 16 00000007	 17 00000002 18 00000037 19 00000064 20 00000040 21 00000005 22 000000a5 23 00000080 24 00000000 25 00000000 26 00000000 27 00000000 28 00000000 29 00000000 30 00000080 31 ffffffff 32 ffffffff

四、Implement multiplier and divider in a single cycle CPU

如果要執行 single cycle 乘法或加法，可以利用很多加法器，組成專門計算乘法除法的 block，然後根據 Opcode 控制它，另外 Register 需要增加 HIGH, LOW 兩個 register 用以存計算結果，類似下圖的計算方法。計算的方法是先將每個 bit 之間之間的結果算出來，然後再把它們組合起來，這樣可以很快速的知道乘法的結果，但是這樣會需要很大的面積。



五、Conclusion and insight on this homework.:

這次作業我做了以下幾點優化 1.自己寫 ALU 中的加法器讓面積縮小 2.自己寫 ALU 中的位移讓面積縮小 3.簡化 control 的邏輯，以上這些優化都讓我的面積可以變得更小。尤其是對 instruction 輸入的 Opcode, Funct 做優化，分別只要 4bit, 3bit 就可以判斷出功能。

這次的作業合成非常的神奇，因為我在合成的時候 slack 上上下下，一下 0.5，一下 4.多，require time 還可以變成 7.多，完全沒有辦法優化。最後我發現如果寫一個 32bit MUX 反而會比直接使用 synthesis 工具合成出來的好很多，slack 也提升很多。