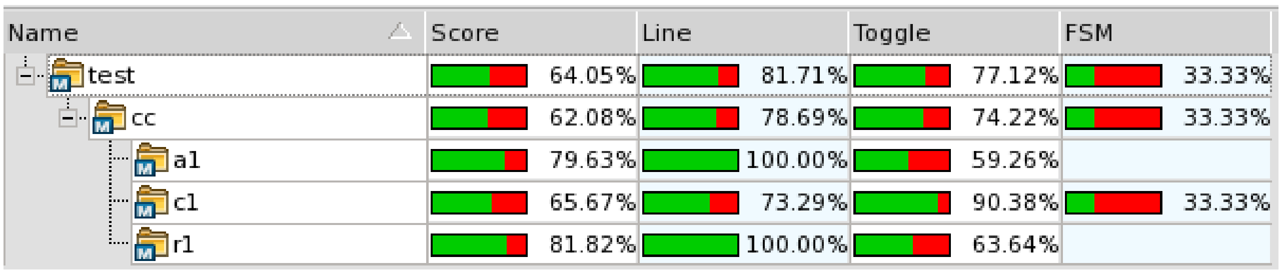
2022 NYCU Computer Aided Design - Homework 3 Report

IEE 陳冠瑋 310510221 / Dec. 4, 2022

1 <u>Why coverage rate can't reach 100% in problem 1 (with coverage rate pictures)</u>

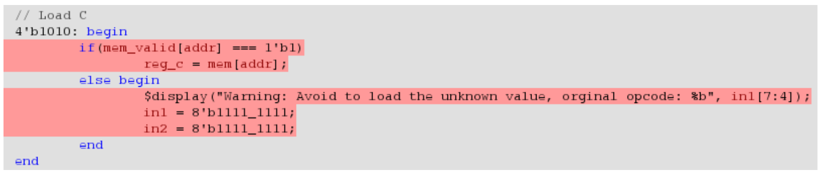
Problem 1. Generate 20 patterns randomly (test_random.v) to analyze coverage



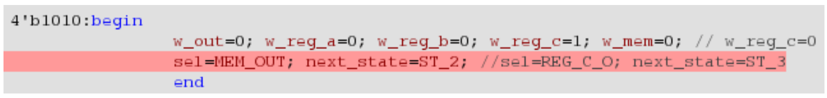
Name	Score	Line	Toggle	FSM
test	64.05%	81.71%	77.12%	33.33%
cc	62.08%	78.69%	74.22%	33.33%
a1	79.63%	100.00%	59.26%	
c1	65.67%	73.29%	90.38%	33.33%
r1	81.82%	100.00%	63.64%	

Fig1. 隨機產生 20 筆測資所測試出來之 coverage rate

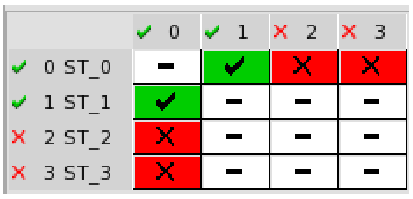
使用 random 的方式所產生的測試資料，如果數量不足夠多的話，是沒辦法 cover 所有的測試項目。因為，隨機產生的測試資料不一定能完全測試我們每一種功能的設計，原因可能是因為隨機產生的測試資料裡面沒有包含全部的情況或是 corner case，可能是重複輸入同一個訊號、測試了同一種功能，導致沒辦法 cover 到每一行的 code、每個 register 的 transition，或是整個電路的每個 state。因此，20 筆測試資料無法達到 100%的 coverage rate。



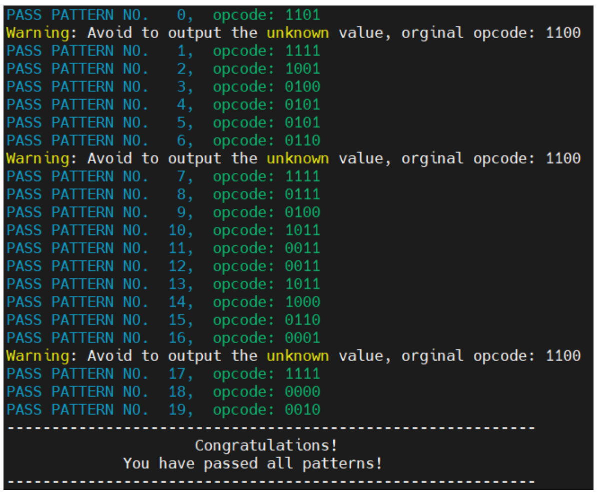
```
// Load C
4'b1010: begin
        if(mem_valid[addr] === 1'b1)
                reg_c = mem[addr];
        else begin
                $display("Warning: Avoid to load the unknown value, orginal opcode: %b", in1[7:4]);
                in1 = 8'b1111_1111;
                in2 = 8'b1111_1111;
        end
end
```



```
4'b1010:begin
            w_out=0; w_reg_a=0; w_reg_b=0; w_reg_c=1; w_mem=0; // w_reg_c=0
            sel=MEM_OUT; next_state=ST_2; //sel=REG_C_O; next_state=ST_3
            end
```



	✓ 0	✓ 1	✗ 2	✗ 3
✓ 0 ST_0	—	✓	✗	✗
✓ 1 ST_1	✓	—	—	—
✗ 2 ST_2	✗	—	—	—
✗ 3 ST_3	✗	—	—	—



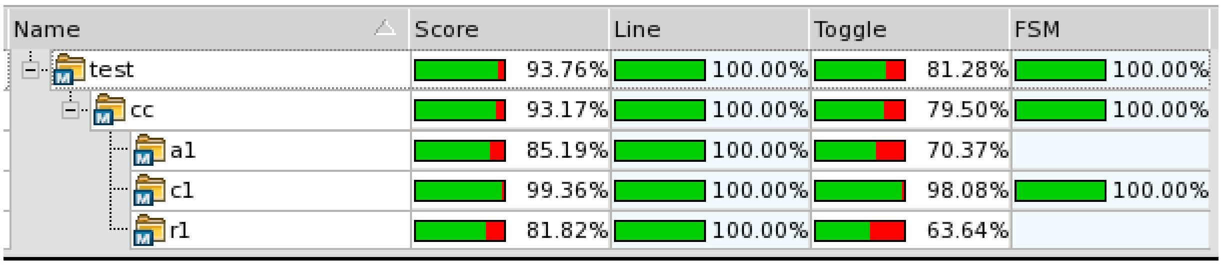
```
PASS PATTERN NO.   0,  opcode: 1101
Warning: Avoid to output the unknown value, orginal opcode: 1100
PASS PATTERN NO.   1,  opcode: 1111
PASS PATTERN NO.   2,  opcode: 1001
PASS PATTERN NO.   3,  opcode: 0100
PASS PATTERN NO.   4,  opcode: 0101
PASS PATTERN NO.   5,  opcode: 0101
PASS PATTERN NO.   6,  opcode: 0110
Warning: Avoid to output the unknown value, orginal opcode: 1100
PASS PATTERN NO.   7,  opcode: 1111
PASS PATTERN NO.   8,  opcode: 0111
PASS PATTERN NO.   9,  opcode: 0100
PASS PATTERN NO.  10,  opcode: 1011
PASS PATTERN NO.  11,  opcode: 0011
PASS PATTERN NO.  12,  opcode: 0011
PASS PATTERN NO.  13,  opcode: 1011
PASS PATTERN NO.  14,  opcode: 1000
PASS PATTERN NO.  15,  opcode: 0110
PASS PATTERN NO.  16,  opcode: 0001
Warning: Avoid to output the unknown value, orginal opcode: 1100
PASS PATTERN NO.  17,  opcode: 1111
PASS PATTERN NO.  18,  opcode: 0000
PASS PATTERN NO.  19,  opcode: 0010
-----------------------------------------------------------
                 Congratulations!
             You have passed all patterns!
-----------------------------------------------------------
```

Fig2. 測試資料沒有 cover 到特定電路功能之情況

由 Fig2.可知，隨機產生的測試資料中並沒有 4'b1010 (Load C) 這樣的 opcode，導致電路的這項功能沒有被覆蓋到。所以 code coverage 就沒有覆蓋到 4'b1010 的情況、FSM coverage 就沒有覆蓋到 ST_2 的情況。

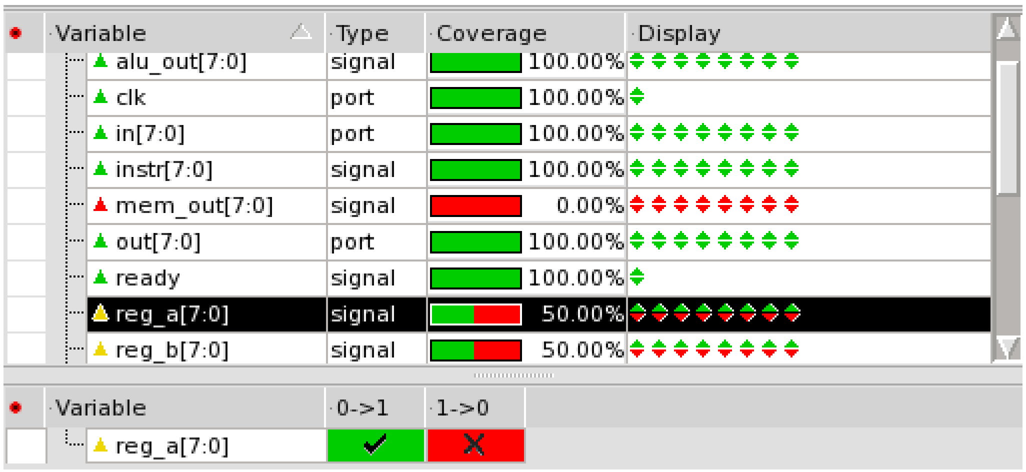
2 Analyze problem 2 results (with coverage rate pictures)

Problem 2. Base on CPU operation, create some patterns to test the original cpu_bug.v with coverage analysis. Can you use less number of test patterns than random approach?



Name	Score	Line	Toggle	FSM
test	93.76%	100.00%	81.28%	100.00%
cc	93.17%	100.00%	79.50%	100.00%
a1	85.19%	100.00%	70.37%	
c1	99.36%	100.00%	98.08%	100.00%
r1	81.82%	100.00%	63.64%	

Fig3. 使用 16 個不同 8-bit CPU 的指令作為測試資料之 coverage rate



Variable	Type	Coverage
alu_out[7:0]	signal	100.00%
clk	port	100.00%
in[7:0]	port	100.00%
instr[7:0]	signal	100.00%
mem_out[7:0]	signal	0.00%
out[7:0]	port	100.00%
ready	signal	100.00%
reg_a[7:0]	signal	50.00%
reg_b[7:0]	signal	50.00%

Variable	0->1	1->0
reg_a[7:0]	✓	✗

Fig4. 使用 16 個不同的 8-bit CPU 指令作為測試資料的 toggle coverage 情況

可以，我使用 16 個不同的 CPU 指令作為測試資料，並且同時考慮有效的指令順序，可以使 coverage 提升(Fig3.)，line coverage 與 FSM coverage 都可以到達 100%。但 toggle coverage (Fig4.) 由於測試資料還不夠全面，所以無法到達 100%。

3 **Find the bugs in the cpu_bug.v (if any) and correct them**

(1) ram_data 是 16 bits 不是 17 bits

```
reg[7:0] ram_data[15:0]; // ram_data[16:0]
```

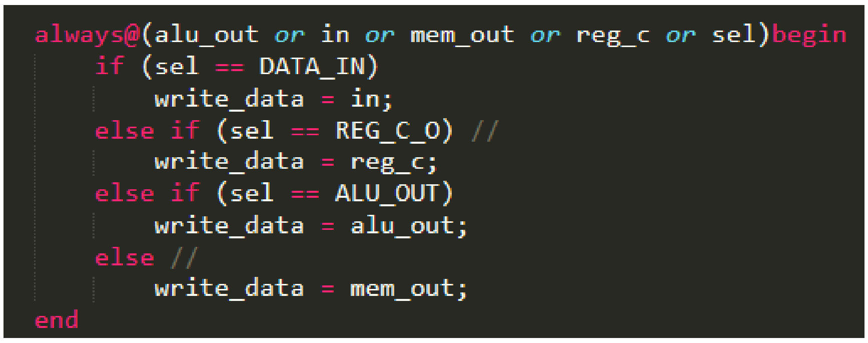
(2) Sequential Logic 使用 Non-Blocking Assignment (<=)

```
always@(posedge clk)begin
    if (w_mem == 1'b0)
        data_out <= ram_data[address];
    else if(w_mem == 1'b1)
        ram_data[address] <= data_in;
    else
        data_out <= 8'hz;
end
endmodule
```

```
always@(posedge clk)begin
    if(reset)begin
        instr <= 0;
        reg_a <= 0;
        reg_b <= 0;
        reg_c <= 0;
        out <= 0;
    end
    else begin
        instr <= in;
        if(ready)begin
            if (w_reg_c)
                reg_c <= write_data;
            else if (w_reg_a)
                reg_a <= write_data;
            else if (w_reg_b)
                reg_b <= write_data;
            else if (w_out)
                out <= write_data;
            else
                out <= 0;
        end
    end

end
```

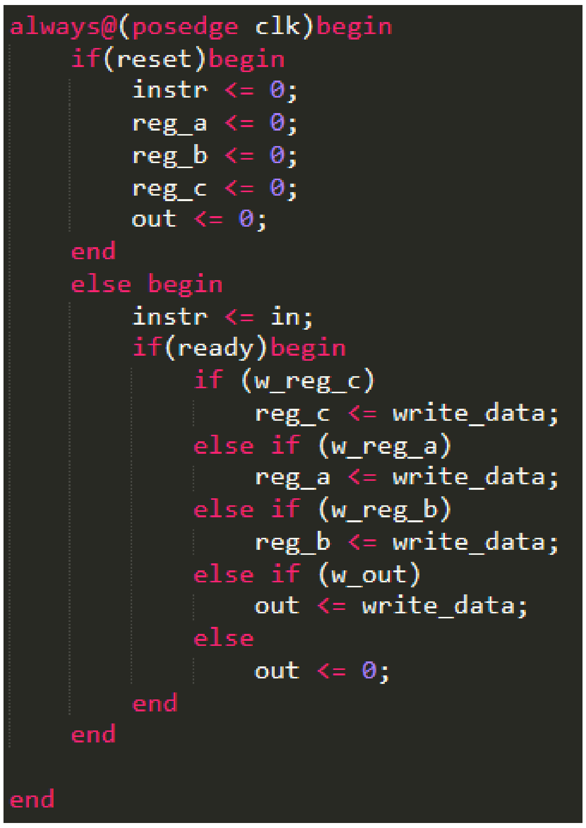
```
always@(posedge clk)begin
    if (reset)
        current_state <= ST_0;
    else
        current_state <= next_state;
end
```

(3) 缺少 sel == REG_C_O 與 sel == MEM_OUT 的情況



```
always@(alu_out or in or mem_out or reg_c or sel)begin
    if (sel == DATA_IN)
        write_data = in;
    else if (sel == REG_C_O) //
        write_data = reg_c;
    else if (sel == ALU_OUT)
        write_data = alu_out;
    else //
        write_data = mem_out;
end
```

(4) 缺少 w_out == 1 的情況

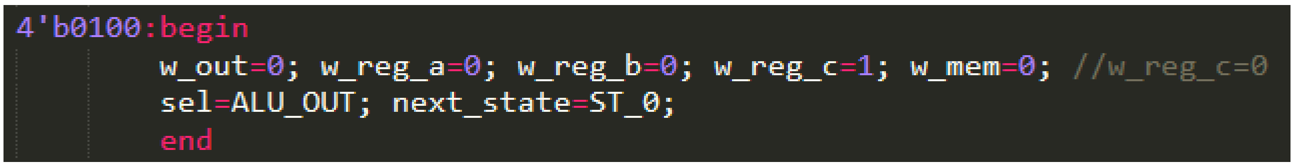


```
always@(posedge clk)begin
    if(reset)begin
        instr <= 0;
        reg_a <= 0;
        reg_b <= 0;
        reg_c <= 0;
        out <= 0;
    end
    else begin
        instr <= in;
        if(ready)begin
            if (w_reg_c)
                reg_c <= write_data;
            else if (w_reg_a)
                reg_a <= write_data;
            else if (w_reg_b)
                reg_b <= write_data;
            else if (w_out)
                out <= write_data;
            else
                out <= 0;
        end
    end

end
```

(5) 一些錯誤的訊號

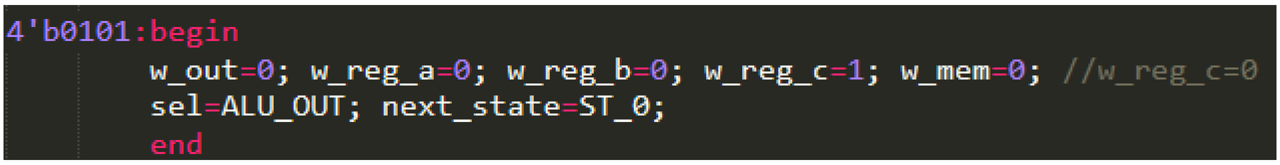
current_state

4’b0100 (C = A + B), Correct: w_reg_c = 1



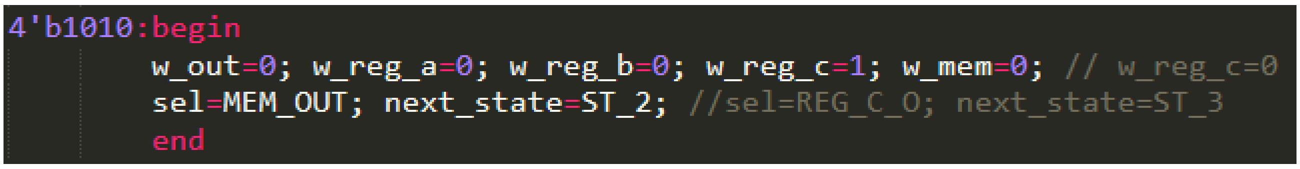
```
4'b0100:begin
        w_out=0; w_reg_a=0; w_reg_b=0; w_reg_c=1; w_mem=0; //w_reg_c=0
        sel=ALU_OUT; next_state=ST_0;
        end
```

4’b0101 (C = -A), Correct: w_reg_c = 1



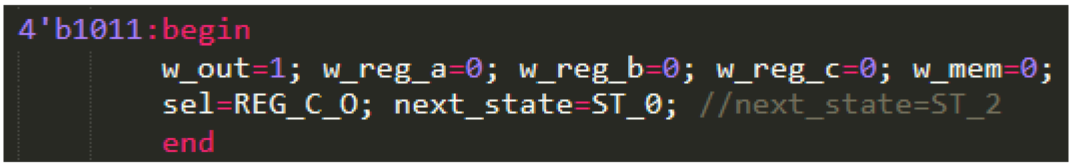
```
4'b0101:begin
        w_out=0; w_reg_a=0; w_reg_b=0; w_reg_c=1; w_mem=0; //w_reg_c=0
        sel=ALU_OUT; next_state=ST_0;
        end
```

4’b1010 (Load C), Correct: w_reg_c = 1, sel = MEM_OUT, next_state = ST_2



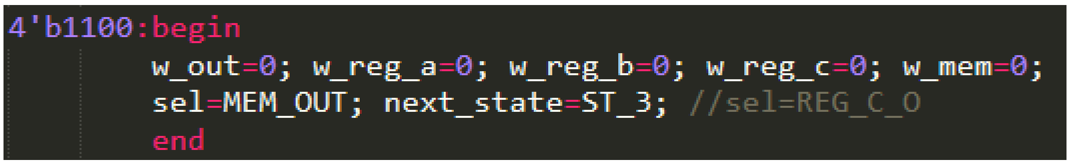
```
4'b1010:begin
        w_out=0; w_reg_a=0; w_reg_b=0; w_reg_c=1; w_mem=0; // w_reg_c=0
        sel=MEM_OUT; next_state=ST_2; //sel=REG_C_O; next_state=ST_3
        end
```

4’b1011 (Output C), Correct: next_state = ST_0



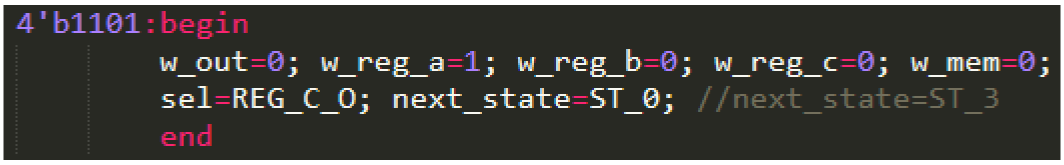
```
4'b1011:begin
        w_out=1; w_reg_a=0; w_reg_b=0; w_reg_c=0; w_mem=0;
        sel=REG_C_O; next_state=ST_0; //next_state=ST_2
        end
```

4’b1100 (Output Mem), Correct: sel = MEM_OUT



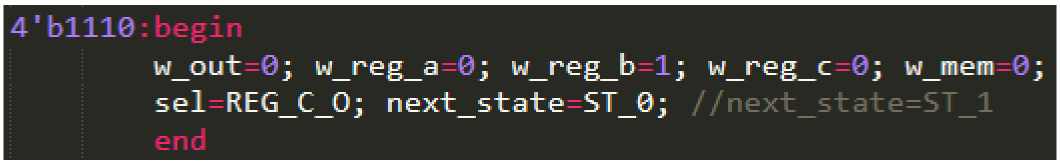
```
4'b1100:begin
        w_out=0; w_reg_a=0; w_reg_b=0; w_reg_c=0; w_mem=0;
        sel=MEM_OUT; next_state=ST_3; //sel=REG_C_O
        end
```

4’b1101 (A = C), Correct: next_state = ST_0



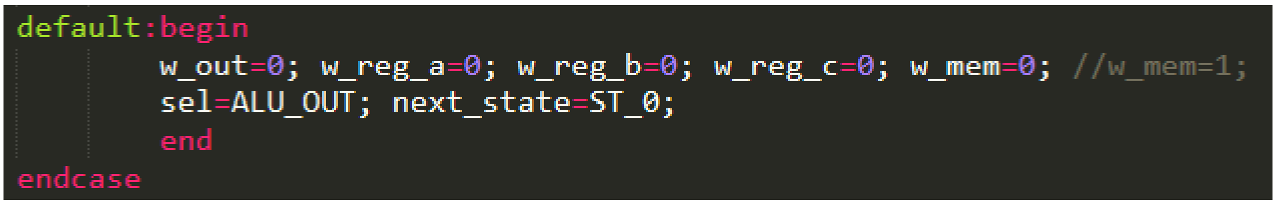
```
4'b1101:begin
        w_out=0; w_reg_a=1; w_reg_b=0; w_reg_c=0; w_mem=0;
        sel=REG_C_O; next_state=ST_0; //next_state=ST_3
        end
```

4’b1110 (B = C), Correct: next_state = ST_1



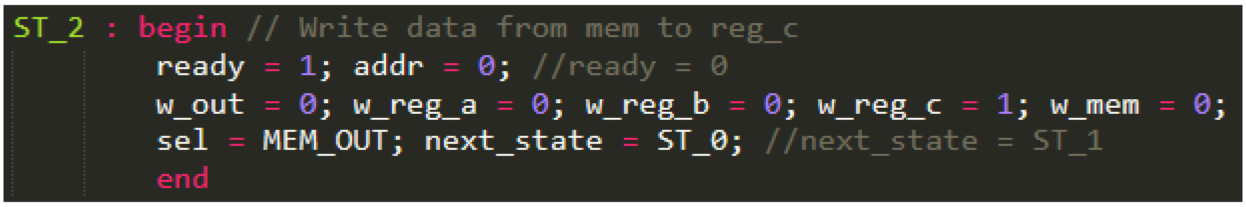
```
4'b1110:begin
        w_out=0; w_reg_a=0; w_reg_b=1; w_reg_c=0; w_mem=0;
        sel=REG_C_O; next_state=ST_0; //next_state=ST_1
        end
```

default, Correct: w_mem = 0



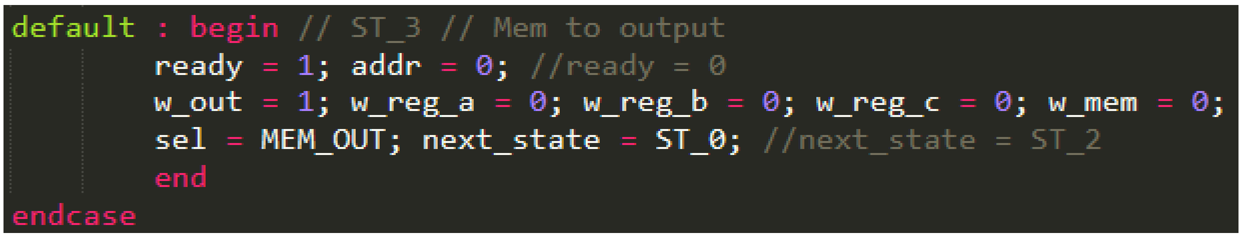
```
default:begin
        w_out=0; w_reg_a=0; w_reg_b=0; w_reg_c=0; w_mem=0; //w_mem=1;
        sel=ALU_OUT; next_state=ST_0;
        end
endcase
```

ST_2, Correct: next_state = ST_0



```
ST_2 : begin // Write data from mem to reg_c
        ready = 1; addr = 0; //ready = 0
        w_out = 0; w_reg_a = 0; w_reg_b = 0; w_reg_c = 1; w_mem = 0;
        sel = MEM_OUT; next_state = ST_0; //next_state = ST_1
        end
```

把 ST_3 的情況改為 default，否則原本的 combinational 電路 case 條件沒寫滿，有可能會出現 latch, Correct: ready = 0, next_state = ST_0



```
default : begin // ST_3 // Mem to output
        ready = 1; addr = 0; //ready = 0
        w_out = 1; w_reg_a = 0; w_reg_b = 0; w_reg_c = 0; w_mem = 0;
        sel = MEM_OUT; next_state = ST_0; //next_state = ST_2
        end
endcase
```

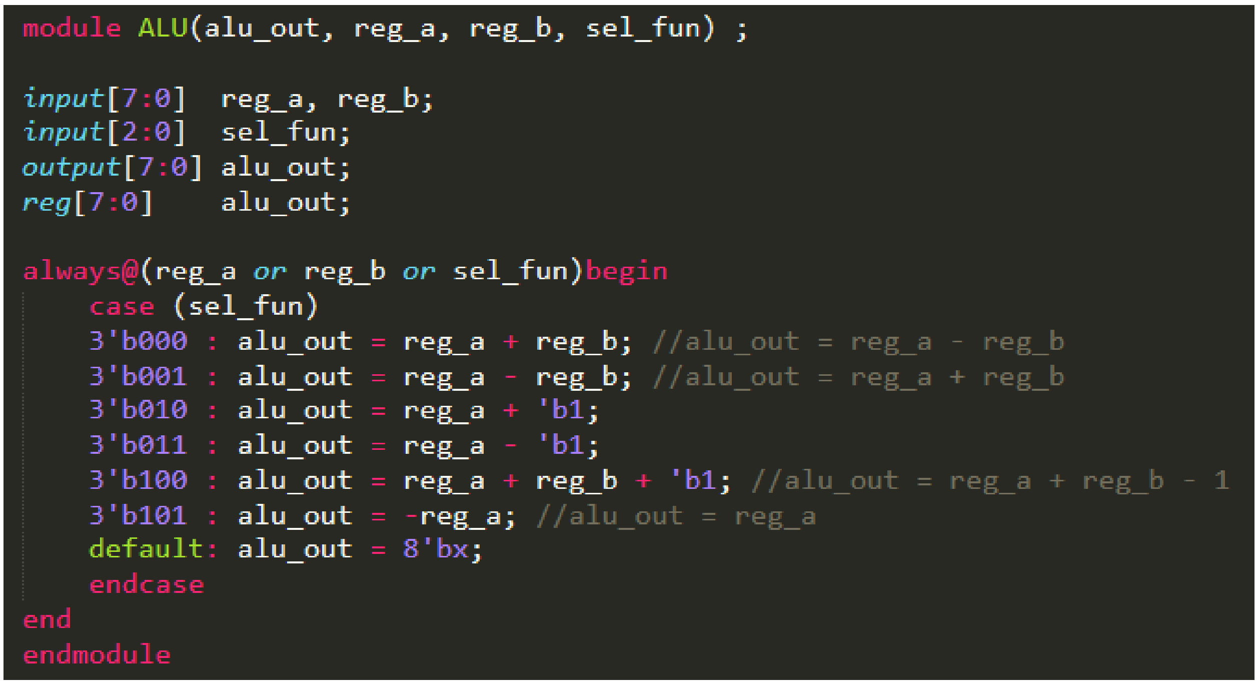
ALU

3’b000, Correct: alu_out = reg_a + reg_b

3’b001, Correct: alu_out = reg_a - reg_b

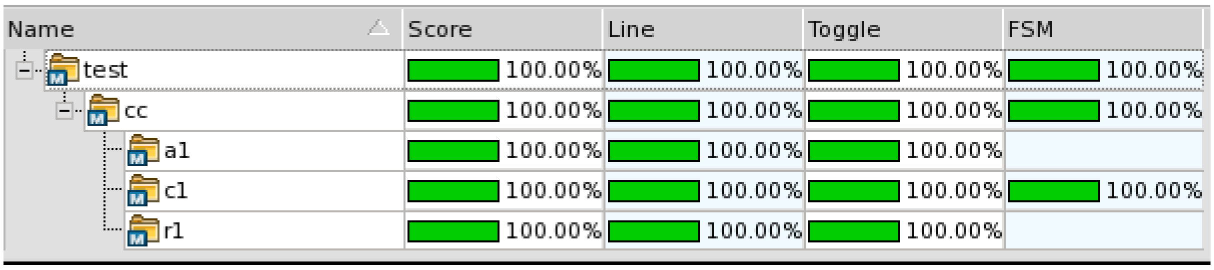
3’b100, Correct: alu_out = reg_a + reg_b + ’b1

3’b100, Correct: alu_out = -reg_a



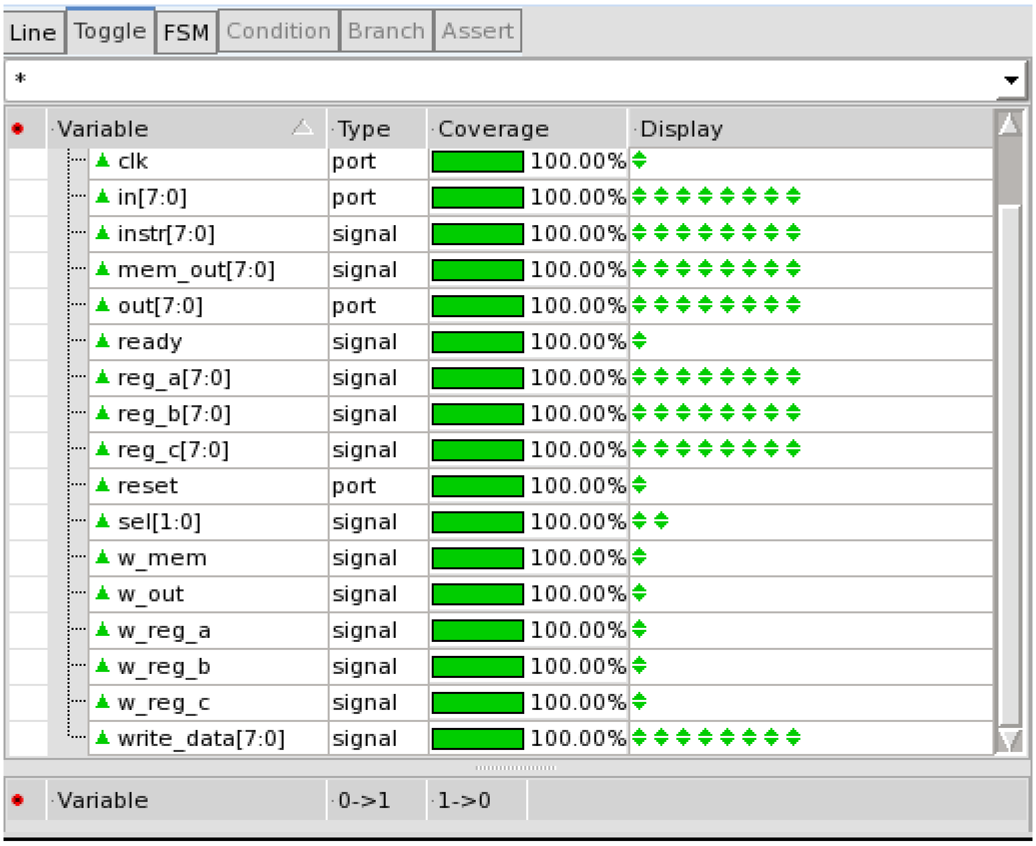
```
module ALU(alu_out, reg_a, reg_b, sel_fun) ;

input[7:0]  reg_a, reg_b;
input[2:0]  sel_fun;
output[7:0] alu_out;
reg[7:0]    alu_out;

always@(reg_a or reg_b or sel_fun)begin
    case (sel_fun)
    3'b000 : alu_out = reg_a + reg_b; //alu_out = reg_a - reg_b
    3'b001 : alu_out = reg_a - reg_b; //alu_out = reg_a + reg_b
    3'b010 : alu_out = reg_a + 'b1;
    3'b011 : alu_out = reg_a - 'b1;
    3'b100 : alu_out = reg_a + reg_b + 'b1; //alu_out = reg_a + reg_b - 1
    3'b101 : alu_out = -reg_a; //alu_out = reg_a
    default: alu_out = 8'bx;
    endcase
end
endmodule
```

4 What did you do to increase the coverage rate (with 100% coverage rate pictures)



Name	Score	Line	Toggle	FSM
test	100.00%	100.00%	100.00%	100.00%
cc	100.00%	100.00%	100.00%	100.00%
a1	100.00%	100.00%	100.00%	
c1	100.00%	100.00%	100.00%	100.00%
r1	100.00%	100.00%	100.00%	

Fig5. 使用 20 個設計過的 pattern 作為測試資料的 coverage rate

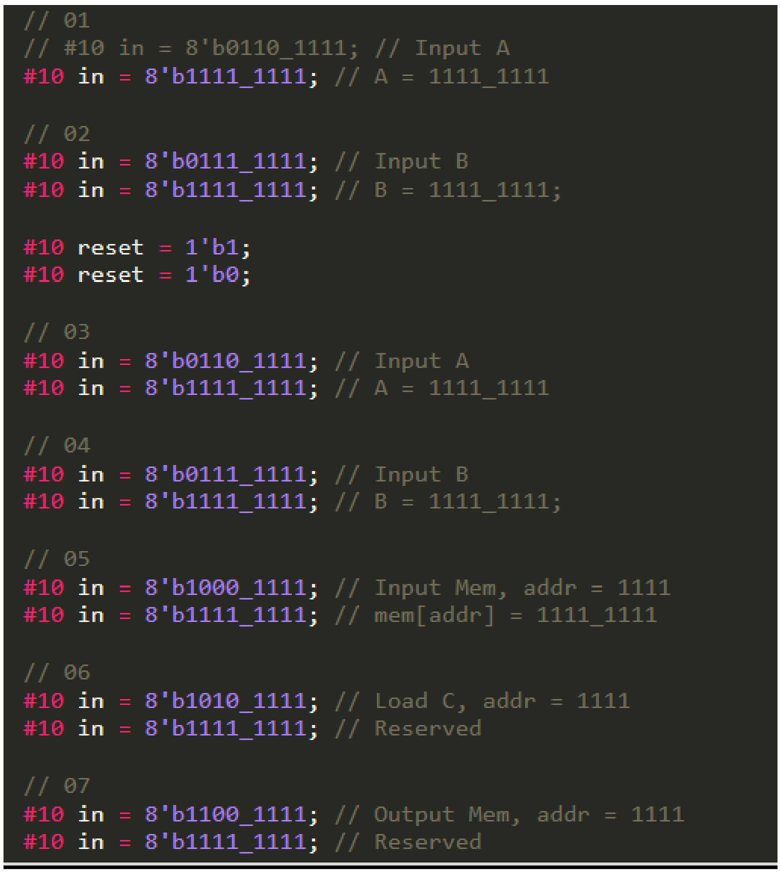


Variable	Type	Coverage	Display
clk	port	100.00%	
in[7:0]	port	100.00%	
instr[7:0]	signal	100.00%	
mem_out[7:0]	signal	100.00%	
out[7:0]	port	100.00%	
ready	signal	100.00%	
reg_a[7:0]	signal	100.00%	
reg_b[7:0]	signal	100.00%	
reg_c[7:0]	signal	100.00%	
reset	port	100.00%	
sel[1:0]	signal	100.00%	
w_mem	signal	100.00%	
w_out	signal	100.00%	
w_reg_a	signal	100.00%	
w_reg_b	signal	100.00%	
w_reg_c	signal	100.00%	
write_data[7:0]	signal	100.00%	

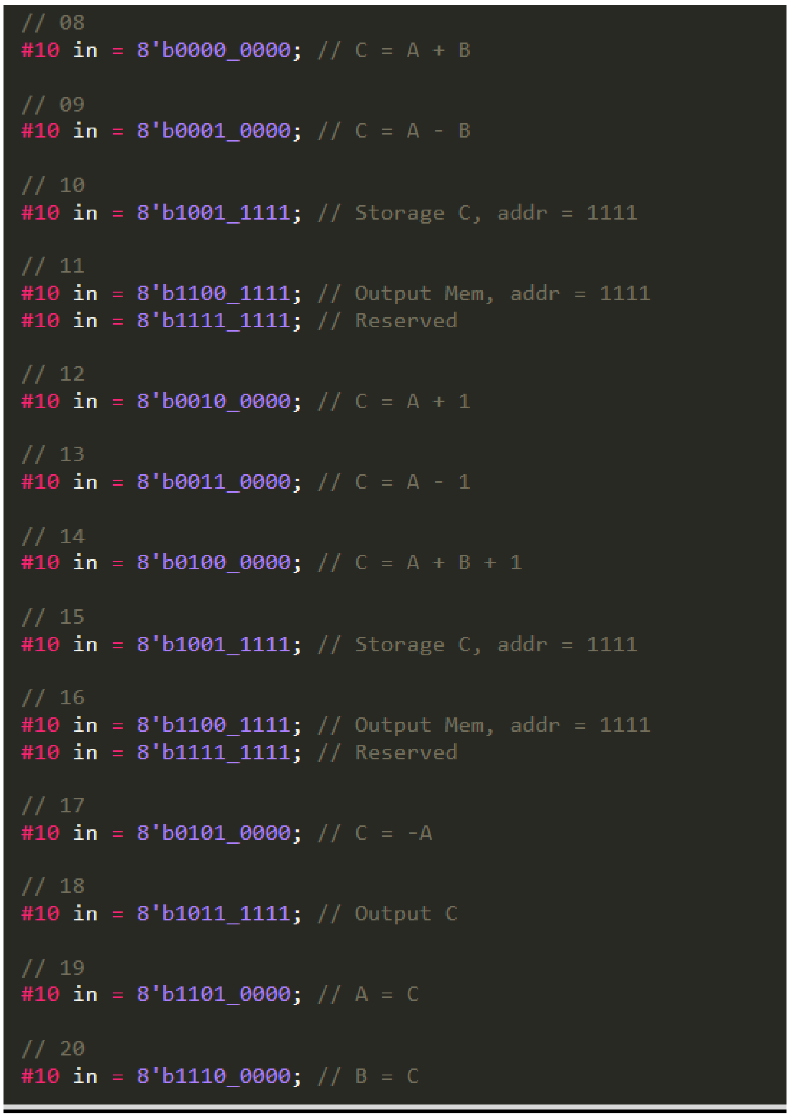
Variable	0->1	1->0

Fig6. 使用 20 個設計過的 pattern 作為測試資料的 toggle coverage

將所有 reference 的 opcode 依照有意義的方式執行過一次，同時考慮 corner case，並將 toggle 不足的地方補齊，直接設計可以覆蓋全部 coverage 的 pattern，使用 20 個 pattern 就可以達到 100% coverage。



```
// 01
// #10 in = 8'b0110_1111; // Input A
#10 in = 8'b1111_1111; // A = 1111_1111

// 02
#10 in = 8'b0111_1111; // Input B
#10 in = 8'b1111_1111; // B = 1111_1111;

#10 reset = 1'b1;
#10 reset = 1'b0;

// 03
#10 in = 8'b0110_1111; // Input A
#10 in = 8'b1111_1111; // A = 1111_1111

// 04
#10 in = 8'b0111_1111; // Input B
#10 in = 8'b1111_1111; // B = 1111_1111;

// 05
#10 in = 8'b1000_1111; // Input Mem, addr = 1111
#10 in = 8'b1111_1111; // mem[addr] = 1111_1111

// 06
#10 in = 8'b1010_1111; // Load C, addr = 1111
#10 in = 8'b1111_1111; // Reserved

// 07
#10 in = 8'b1100_1111; // Output Mem, addr = 1111
#10 in = 8'b1111_1111; // Reserved
```



```
// 08
#10 in = 8'b0000_0000; // C = A + B

// 09
#10 in = 8'b0001_0000; // C = A - B

// 10
#10 in = 8'b1001_1111; // Storage C, addr = 1111

// 11
#10 in = 8'b1100_1111; // Output Mem, addr = 1111
#10 in = 8'b1111_1111; // Reserved

// 12
#10 in = 8'b0010_0000; // C = A + 1

// 13
#10 in = 8'b0011_0000; // C = A - 1

// 14
#10 in = 8'b0100_0000; // C = A + B + 1

// 15
#10 in = 8'b1001_1111; // Storage C, addr = 1111

// 16
#10 in = 8'b1100_1111; // Output Mem, addr = 1111
#10 in = 8'b1111_1111; // Reserved

// 17
#10 in = 8'b0101_0000; // C = -A

// 18
#10 in = 8'b1011_1111; // Output C

// 19
#10 in = 8'b1101_0000; // A = C

// 20
#10 in = 8'b1110_0000; // B = C
```

5 **What did you learn in this project**

透過 coverage analysis 可以了解到自己設計的 pattern 對於這個 design 考慮的夠不夠全面。發現 code coverage 小的時候，可能代表自己寫的 pattern 有些 corner case 沒有考慮到，所以沒有執行到那些 code，或者是自己本身的 design 寫了一些沒有意義的 code，導致永遠不會進到那些條件中。發現 FSM coverage 小的時候，就可以回去看自己寫的 pattern 是不是打中全部的 state，如果有些 state 沒有打中，可以嘗試設計一些 pattern 可以進到那些 state，如此一來才能完整測試整個電路的功能是否正常。發現 toggle coverage 小的，可以試試看多加一些 pattern 的組合，讓所有 register 的 transition 的情況都有辦法發生。從這次的作業就可以體會到，只要好好利用 CAD tool，像是 Verdi，就可以更有效率的設計 pattern 與修正 design 的 bug。