2025/10/09

實驗五

序向邏輯練習

姓名：林承羿
學號：01257027
班級：資工 3A
E-mail：IanLin6225@gmail.com

※注意

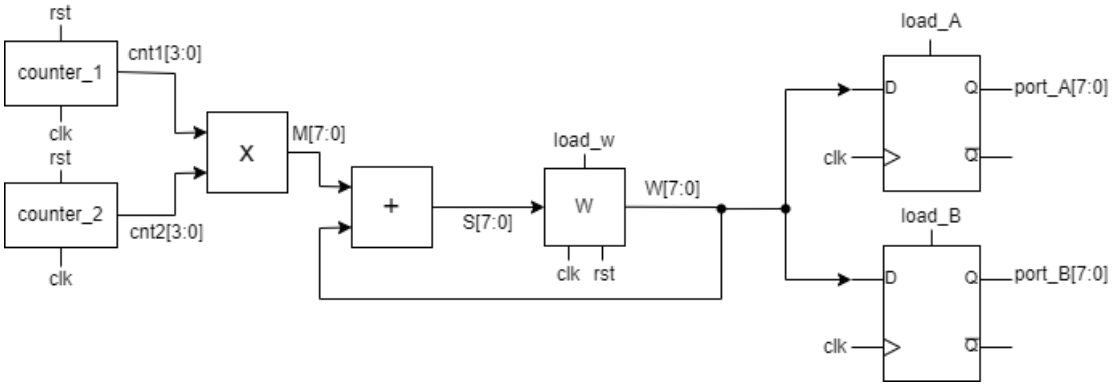
1. 繳交時一律轉 PDF 檔
2. 繳交期限為下周上課前
3. 一人繳交一份
4. 檔名請按照作業檔名格式進行填寫，未依照格式不予批改
5. 檔名範例：學號_姓名_HW5

1、 乘法累加器(Multiply Accumulate, MAC)

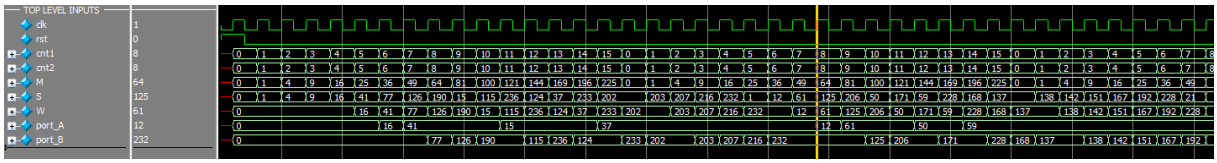
●實驗說明：

1. 實現一個乘法累加器，W = W + cntl * cnt2。
2. cntl、cnt2 同時從 0 數到 15，再歸零重數。
3. 當 S 大於等於 8'd10 時，再寫入 W。
4. 當 W 值小於 8'd63 時，將值傳到 port_A，大於等於 8'd63 時，則傳到 port_B。
5. 用 FSM 控制 load_w、load_A、load_B。
6. port_A、port_B 記得 reset。

●系統硬體架構方塊圖（接線圖）：



● 參考波形



●系統架構程式碼、測試資料程式碼與程式碼說明

截圖請善用 **win+shift+S**

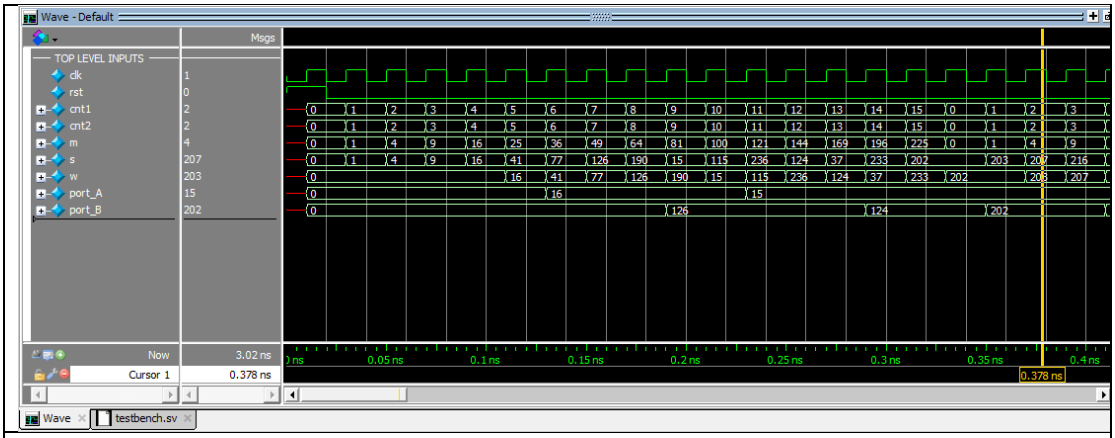
1. 錯誤經驗

mymac.sv > mymac

```
module mymac(
    input logic rst, clk,
    output logic [7:0] port_A, port_B
);
    logic [3:0] cnt1, cnt2;
    logic [7:0] m, s, w;
    logic load_w, load_a, load_b;
    typedef enum logic [1:0] { start = 2'd0, lw = 2'd1, la = 2'd2, lb = 2'd3 } state_t;
    state_t ps, ns;
    // counter 1, 2
    always_ff @(posedge clk) begin
        if (rst) begin
            cnt1 <= 4'd0;
            cnt2 <= 4'd0;
        end
        else begin
            cnt1 <= cnt1 + 1;
            cnt2 <= cnt2 + 1;
        end
    end

    // multiple cnt1, cnt2 every time
    assign m = cnt1 * cnt2;

    // s = w + m
    assign s = w + m;

    // W flip-flop
    always_ff @(posedge clk) begin
        if (rst) begin
            w <= 8'd0;
        end
        else begin
            if (load_w) begin
                w <= s;
            end
        end
    end
```

```
39
40      // port_A flip-flop
41      always_ff @(posedge clk) begin
42          if (rst) begin
43              port_A <= 8'd0;
44          end
45          else begin
46              if (load_a) begin
47                  port_A <= w;
48              end
49          end
50      end
51
52      // port_B flip-flop
53      always_ff @(posedge clk) begin
54          if (rst) begin
55              port_B <= 8'd0;
56          end
57          else begin
58              if (load_b) begin
59                  port_B <= w;
60              end
61          end
62      end
63
64      // initial FSM state to "start"
65      always_ff @(posedge clk) begin
66          if (rst) begin
67              ps <= start;
68          end
69          else begin
70              ps <= ns;
71          end
72      end
```

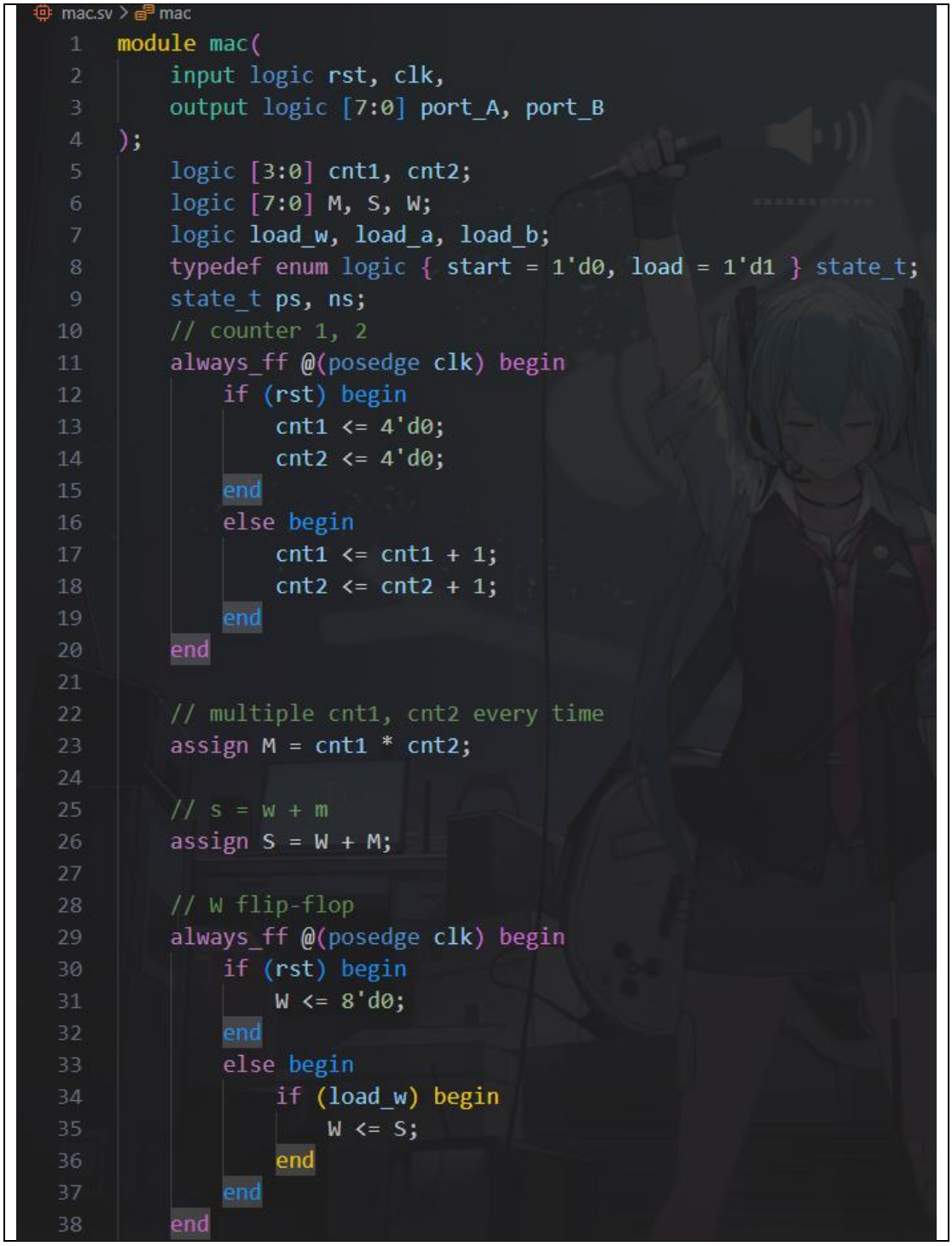
```
73
74      // FSM controller
75      always_comb begin
76          load_w = 0;
77          load_a = 0;
78          load_b = 0;
79          ns = ps;
80          case (ps)
81              start: begin
82                  ns = lw;
83              end
84              lw: begin
85                  if (s >= 8'd10) begin
86                      load_w = 1;
87                      ns = la;
88                  end
89              end
90              la: begin
91                  if (s >= 8'd10) begin
92                      load_w = 1;
93                  end
94                  if (w < 8'd63) begin
95                      load_a = 1;
96                      ns = lw;
97                  end
98                  else begin
99                      ns = lb;
100                 end
101             end
102             lb: begin
103                 if (s >= 8'd10) begin
104                     load_w = 1;
105                 end
106                 if (w >= 8'd63) begin
107                     load_b = 1;
108                     ns = lw;
109                 end
110             end
111         endcase
112     end
113 endmodule
```



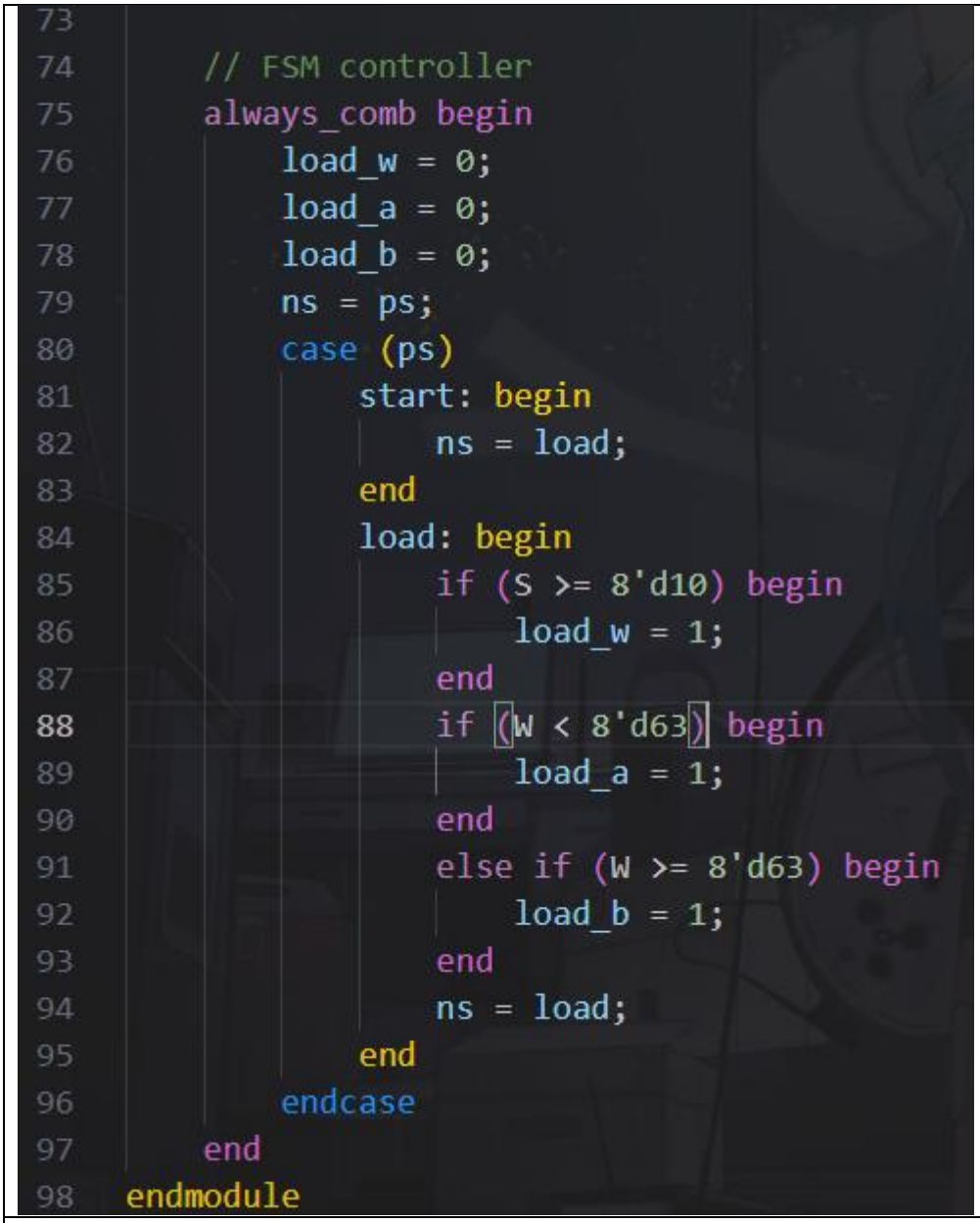
這是錯誤的，這想法是以 load W 作為開機 reset 後的第一個狀態，先檢查 A 是否符合需求再檢查 B，這兩個狀態都可能跳回 W 狀態。

問題是波型圖跑出來可以看到檢查 A 時，可能 B 符合要求，但狀態還沒跑到 B 因此會有許多缺值，檢查 A 時也是如此。

2. 正確結果


mac.sv > mac
module mac(
input logic rst, clk,
output logic [7:0] port_A, port_B
);
logic [3:0] cnt1, cnt2;
logic [7:0] M, S, W;
logic load_w, load_a, load_b;
typedef enum logic { start = 1'd0, load = 1'd1 } state_t;
state_t ps, ns;
// counter 1, 2
always_ff @(posedge clk) begin
if (rst) begin
cnt1 <= 4'd0;
cnt2 <= 4'd0;
end
else begin
cnt1 <= cnt1 + 1;
cnt2 <= cnt2 + 1;
end
end
// multiple cnt1, cnt2 every time
assign M = cnt1 * cnt2;
// s = w + m
assign S = W + M;
// W flip-flop
always_ff @(posedge clk) begin
if (rst) begin
W <= 8'd0;
end
else begin
if (load_w) begin
W <= S;
end
end
end

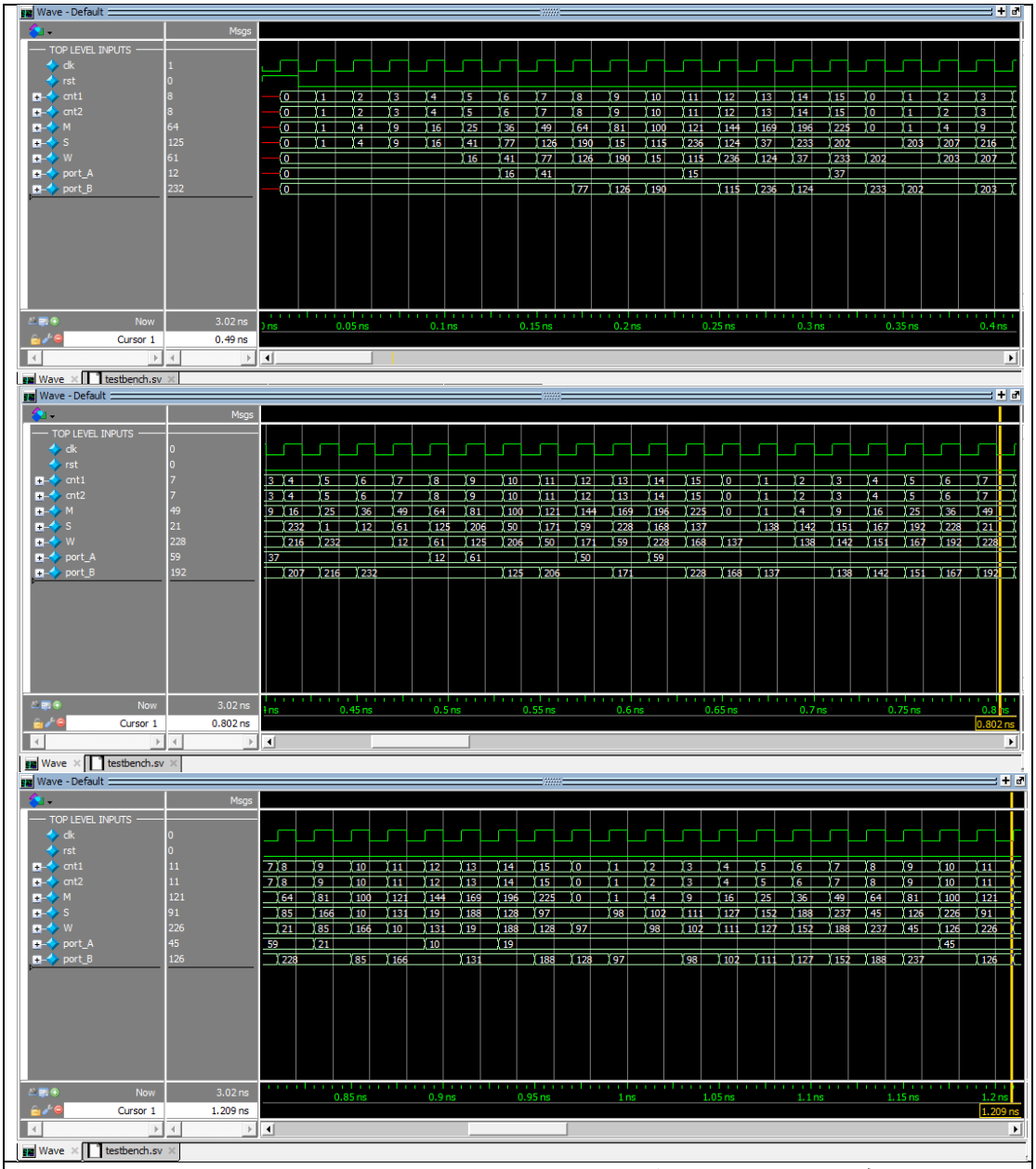
```
39
40      // port_A flip-flop
41      always_ff @(posedge clk) begin
42          if (rst) begin
43              port_A <= 8'd0;
44          end
45          else begin
46              if (load_a) begin
47                  port_A <= W;
48              end
49          end
50      end
51
52      // port_B flip-flop
53      always_ff @(posedge clk) begin
54          if (rst) begin
55              port_B <= 8'd0;
56          end
57          else begin
58              if (load_b) begin
59                  port_B <= W;
60              end
61          end
62      end
63
64      // initial FSM state to "start"
65      always_ff @(posedge clk) begin
66          if (rst) begin
67              ps <= start;
68          end
69          else begin
70              ps <= ns;
71          end
72      end
```



```
73
74      // FSM controller
75      always_comb begin
76          load_w = 0;
77          load_a = 0;
78          load_b = 0;
79          ns = ps;
80          case (ps)
81              start: begin
82                  ns = load;
83              end
84              load: begin
85                  if (S >= 8'd10) begin
86                      load_w = 1;
87                  end
88                  if (W < 8'd63) begin
89                      load_a = 1;
90                  end
91                  else if (W >= 8'd63) begin
92                      load_b = 1;
93                  end
94                  ns = load;
95              end
96          endcase
97      end
98  endmodule
```

可以看到跟錯誤的比起來我省去了分別 **load A, B** 的狀態，只保留了 **reset** 後進入的狀態 **start**。如果要更省可以讓 **reset** 後直接進入 **load** 狀態(可以把 **FSM** 拔掉，只剩下 **if** 判斷)。

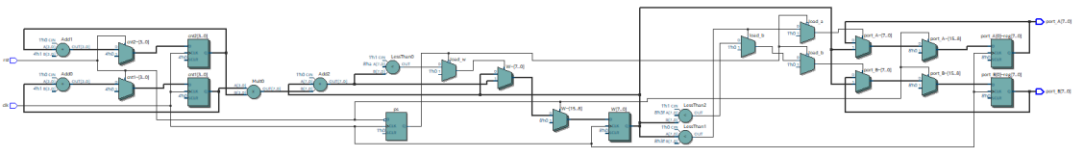
●模擬結果與結果說明：



可以看到跟之前錯誤的比起來結果修正了許多，在同一個狀態中判斷是否 load_a, load_b, load_w 避免漏掉因狀態錯誤而為判斷到的結果。

●結論與心得：

從此次的實驗中，我更深刻的體會到了如何撰寫狀態機，並非 compile 過後沒問題的狀態機都是可以依照理想結果產出的，尤其是當許多 flag 要一起判斷決定誰輸出時需要將所有判斷寫在同一個狀態內才可以確保不漏掉答案。



本人對照了給定的電路圖，看著自己 code 轉出的電路圖，大致上都符合圖示描述，架構上應與給定的想法差不多。