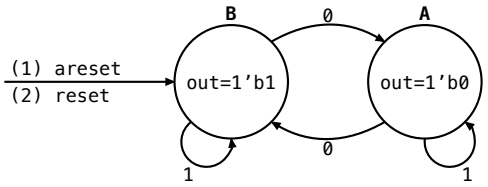
HDLbits 有限狀態機練習

Practice of FSM on HDLbits

Hylu

Problem 1　Simple FSM 1

題目：分別以（1）異步復位；（2）同步復位；實作下圖所示一位輸入、一位輸出、二狀態、復位狀態爲 B 的摩爾狀態機（Moore state machine）。



1.1　Asynchronous reset

Listing 1.1: 以異步復位實作

```
module top_module(
    input clk,
    input areset,    // Asynchronous reset to state B
    input in,
    output out
);
    parameter A=1'b0, B=1'b1; // to avoid truncating warning
    reg state, next_state;
    // State transition logic, a combinational always block
    always @(*) begin
        case (state)
            A: next_state = in? A: B;
            B: next_state = in? B: A;
        endcase
    end
    // State flip-flops with a-reset, a sequential always block
    always @(posedge clk, posedge areset) begin
        state <= areset? B: next_state;
    end
    // Output logic
    assign out = (state == A)? 1'b0: 1'b1;
endmodule
```

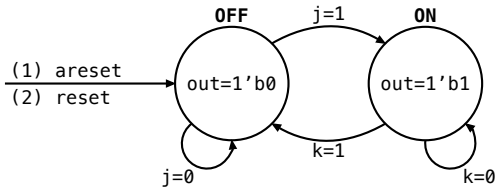
1.2 Synchronous reset

Listing 1.2: 以同步復位實作

```
module top_module(
    input clk,
    input reset,    // Synchronous reset to state B
    input in,
    output out
);
    parameter A=1'b0, B=1'b1; // to avoid truncating warning
    reg state, next_state;
    // State transition logic, a combinational always block
    always @(*) begin
        case (state)
            A: next_state = in? A: B;
            B: next_state = in? B: A;
        endcase
    end
    // State flip-flops, a sequential always block
    always @(posedge clk) begin
        state <= reset? B: next_state;
    end
    // Output logic
    assign out = (state == A)? 1'b0: 1'b1;
endmodule
```

Problem 2 Simple FSM 2 (asynchronous reset)

題目：分別以（1）異步復位；（2）同步復位；實作下圖所示兩位輸入、一位輸出、二狀態的摩爾狀態機（Moore state machine）。



2.1 Asynchronous reset

Listing 2.1: 以異步復位實作

```
module top_module(
    input clk,
    input areset,    // Asynchronous reset to OFF
    input j,
    input k,
    output out
); //
    parameter OFF=1'b0, ON=1'b1;
    reg state, next_state;
    // State transition logic, combinational always
    always @(*) begin
        case (state)
            OFF: next_state = j? ON: OFF;
            ON : next_state = k? OFF: ON;
        endcase
    end
    // State flip-flops with asynchronous reset, sequential always
    always @(posedge clk, posedge areset) begin
        state <= areset? OFF: next_state;
    end
    // Output logic
    assign out = (state == ON)? 1'b1: 1'b0;
endmodule
```

2.2 Synchronous reset

Listing 2.2: 以同步復位實作

```
module top_module(
    input clk,
    input reset,    // synchronous reset to OFF
    input j,
    input k,
    output out
); //
    parameter OFF=1'b0, ON=1'b1;
    reg state, next_state;
    // State transition logic, combinational always
    always @(*) begin
        case (state)
            ON: next_state = k? OFF: ON;
            OFF: next_state = j? ON: OFF;
        endcase
    end
    // State flip-flops with synchronous reset, sequential always
    always @(posedge clk) begin
        state <= reset? OFF: next_state;
    end
    // Output logic
    assign out = (state == ON)? 1'b1: 1'b0;
endmodule
```

Problem 3 Simple one-hot state transitions 3

題目：以下是一個一位輸入、一位輸出、四狀態的摩爾狀態機的狀態轉換表。

State	Next State		Output
	in=0	in=1	
A	A	B	0
B	C	B	0
C	A	D	0
B	C	B	1

(a) 給定狀態編碼 `A=0, B=1, C=2, D=3`，實作組合邏輯；

(b) 給定獨熱（onehot）狀態編碼 `A=4'b0001, B=4'b0010, C=4'b0100, D=4'b1000`，實作組合邏輯；

(c) 實作異步復位；

(d) 實作同步復位。

3.1 Simple state transition

Listing 3.1: 順序編碼狀態轉換組合邏輯

```
module top_module(
    input in,
    input [1:0] state,
    output [1:0] next_state,
    output out
); //
    parameter A=2'd0, B=2'd1, C=2'd2, D=2'd3;
    // State transition logic: next_state = f(state, in)
    always @(*) begin
        case (state)
            A: next_state = in? B: A;
            B: next_state = in? B: C;
            C: next_state = in? D: A;
            D: next_state = in? B: C;
        endcase
    end
    // Output logic:  out = f(state) for a Moore state machine
    assign out = (state == D)? 1'b1: 1'b0;

endmodule
```

3.2 Simple one-hot state transition

Listing 3.2: 獨熱編碼狀態組合邏輯

```
module top_module(
    input in,
    input [3:0] state,
    output [3:0] next_state,
    output out
); //
    parameter A=2'd0, B=2'd1, C=2'd2, D=2'd3;
    // State transition logic: Derive an equation for each state flip-flop.
    assign next_state[A] = (~in) & (state[A] | state[C]);
    assign next_state[B] = in & (state[A] | state[B] | state[D]);
    assign next_state[C] = (~in) & (state[B] | state[D]);
    assign next_state[D] = in & state[C];
    // Output logic:
    assign out = state[D];
endmodule
```

3.3 Asynchronous reset

Listing 3.3: 異步復位

```
module top_module(
    input clk,
    input in,
    input areset,
    output out
); //
    parameter A = 2'd0, B = 2'd1, C = 2'd2, D = 2'd3;
    reg[1:0] state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            A: next_state = in? B: A;
            B: next_state = in? B: C;
            C: next_state = in? D: A;
            D: next_state = in? B: C;
        endcase
    end
    // State flip-flops with asynchronous reset
    always @(posedge clk or posedge areset) begin
        state <= areset? A: next_state;
    end
    // Output logic
    assign out = (state == D)? 1'b1: 1'b0;
endmodule
```

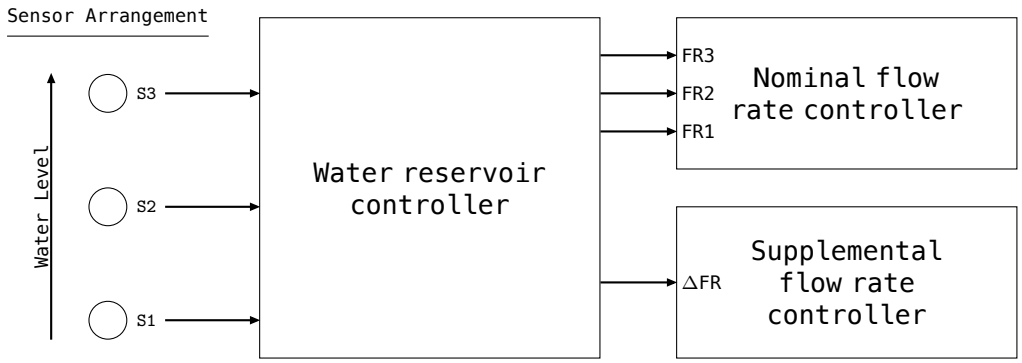
3.4 Synchronous reset

Listing 3.4: 同步復位

```
module top_module(
    input clk,
    input in,
    input reset,
    output out
); //
    parameter A = 2'd0, B = 2'd1, C = 2'd2, D = 2'd3;
    reg[1:0] state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            A: next_state = in? B: A;
            B: next_state = in? B: C;
            C: next_state = in? D: A;
            D: next_state = in? B: C;
        endcase
    end
    // State flip-flops with synchronous reset
    always @(posedge clk) state <= reset? A: next_state;
    // Output logic
    assign out = (state == D)? 1'b1: 1'b0;
endmodule
```

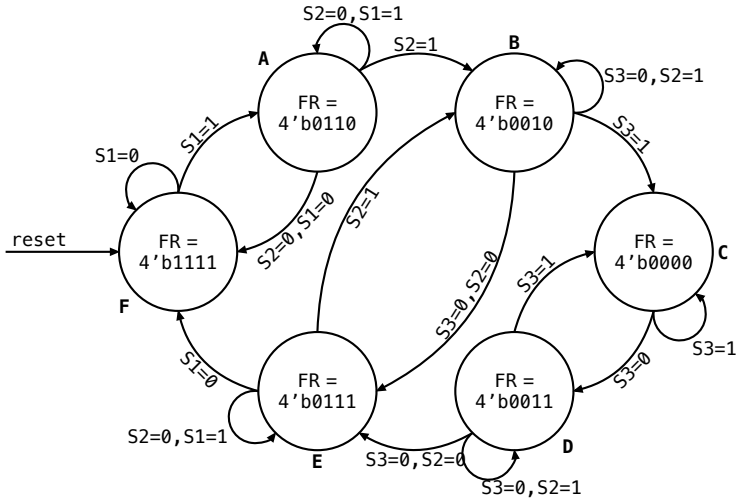
Problem 4 Design a Moore FSM

題目：一個大蓄水池，爲了維持足夠高的水位以便供水，在豎直方向上放置了三個傳感器，各間隔 5 英吋。若水位超過最高的傳感器（S_3），輸入流量應該爲零。若水位低於最低的傳感器（S_1），流量應該拉到最大值（即名目閥與輔助閥都打開）。若水位在最高與最低傳感器之間，則流量由兩個因素決定：現在的水位與上一次傳感器訊號變化的水位。每一個水位都對應一個名目流量，如下表所示。若訊號變化表明先前的水位低於目前的水位，則流量應爲名目流量；反之，應該進一步打開輔助閥（由 ΔFR 控制）以提高流量。畫出該蓄水池控制器的摩爾狀態機，輸入爲 `S1, S2, S3`，輸出爲 `FR1, FR2, FR3, ΔFR`。此外，同步復位狀態爲空。



水位	傳感器激活	名目輸入流量
高於 S3	S1, S2, S3	無
介於 S3 與 S2	S1, S2	FR1
介於 S2 與 S1	S1	FR1, FR2
低於 S1	None	FR1, FR2, FR3

解答：狀態轉換圖如下。



圖中，把四個輸出封裝成向量 `FR[3:0]={fr3, fr2, fr1, dfr}`。設定水位四個狀態的名稱分別爲「滿（full）、高（high）、低（low）、空（empty）」，由於 `dfr` 受到水位升降影響，輸出狀態一共有六種：

狀態名稱	含義	FR[3:0]	狀態名稱	含義	FR[3:0]
A	水位從空升到低	4'b0110	D	水位從滿降到高	4'b0011
B	水位從低升到高	4'b0010	E	水位從高降到低	4'b0111
C	水位滿	4'b0000	F	水位空	4'b1111

Listing 4.1: 設計代碼

```
module top_module (
    input clk,
    input reset,
    input [3:1] s,
    output fr3,
    output fr2,
    output fr1,
    output dfr
);
    parameter F=3'd0, A=3'd1, B=3'd2, C=3'd3, D=3'd4, E=3'd5;
    reg[2:0] state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            A: next_state = s[2]? B: (s[1]? A: F);
            B: next_state = s[3]? C: (s[2]? B: E);
            C: next_state = s[3]? C: D;
            D: next_state = s[3]? C: (s[2]? D: E);
            E: next_state = s[2]? B: (s[1]? E: F);
            default: next_state = s[1]? A: F;
        endcase
    end
    // State flip-flops with synchronous reset
    always @(posedge clk) state <= reset? F: next_state;
    // Output logic
    always @(*) begin
        case (state)
            A: {fr3,fr2,fr1,dfr} = 4'b0110;
            B: {fr3,fr2,fr1,dfr} = 4'b0010;
            C: {fr3,fr2,fr1,dfr} = 4'b0000;
            D: {fr3,fr2,fr1,dfr} = 4'b0011;
            E: {fr3,fr2,fr1,dfr} = 4'b0111;
            default: {fr3,fr2,fr1,dfr} = 4'b1111;
        endcase
    end
endmodule
```

Problem 5 Lemmings

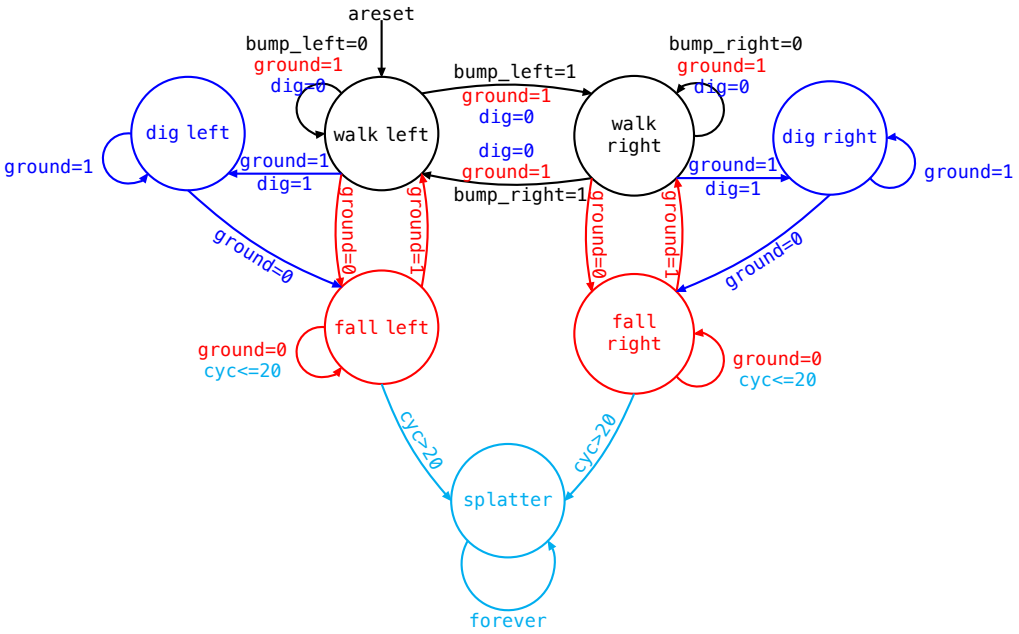
遊戲「旅鼠總動員」有一種頭腦簡單的小動物，簡單到我們可以用一個有限狀態機建模。

(a) 在旅鼠總動員的二維世界，旅鼠可以處於兩種狀態：往左行走或往右行走。若撞到障礙物，牠會轉向。具體而言，如果旅鼠左側撞到，牠就會往右；反之右側撞到，牠就會往左。如果兩邊同時撞到，牠會走原本的反方向。設計一個二狀態、二位輸入、一位輸出的摩爾狀態機描述這一行爲，其狀態圖即下圖的黑色部分。

(b) 除了往左、往右、撞到後改變方向外，若腳下的地面消失（`ground=0`），旅鼠還會墜落並慘叫「啊啊啊啊！」。當地面重現（`ground=1`），旅鼠會沿著墜落之前的方向前進。墜落時、地面剛消失尚未墜落時、地面剛出現尚在墜落時，撞到障礙物不會改變方向。設計一個有限狀態機描述這一行爲，其狀態圖即下圖擴展到紅色部分。

(c) 除了行走與墜落之外，旅鼠還能做些有用的工作，比如挖掘（當 `dig=1` 時開始挖掘）。挖掘行爲發生於旅鼠走在地面上時（即沒有在墜落，且 `ground=1`），且可以一直挖掘到牠挖穿地面（`ground=0`）。這時，由於沒有在地面上，旅鼠會墜落（啊啊啊啊！）然後在落地後沿原本的方向行走。如上題，挖掘時撞到不會影響方向；在墜落期間或地面消失時挖掘指令會被忽略。換句話說，行走的旅鼠可以執行墜落、挖掘或轉向，倘若這些行爲嘗試同時出現，則墜落的優先級最高、轉向的優先級最低。擴展先前的狀態機描述旅鼠的行爲，其狀態圖即下圖擴展到藍色部分。

(d) 旅鼠固然能行走、墜落、挖掘，但也絕非不死之身。若旅鼠墜落太久才碰到地面，牠會粉身碎骨。具體而言，若旅鼠跌落超過 20 個週期才碰到地面，牠會粉身碎骨，永遠無法繼續行走、墜落或挖掘（四個輸出都是 0）直到復位。墜地時間沒有上限，旅鼠只會在接觸地面時粉身碎骨，在半空中不會。擴展先前的狀態機描述旅鼠的行爲，其狀態圖即下圖擴展到青色部分。



5.1 Lemmings1

Listing 5.1: 旅鼠總動員 1

```
module top_module(
    input clk,
    input areset,    // Freshly brainwashed Lemmings walk left.
    input bump_left,
    input bump_right,
    output walk_left,
    output walk_right
); //
    parameter WALK_LEFT=1'b0, WALK_RIGHT=1'b1;
    reg state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            WALK_LEFT: next_state = bump_left? WALK_RIGHT: WALK_LEFT;
            WALK_RIGHT: next_state = bump_right? WALK_LEFT: WALK_RIGHT;
        endcase
    end
    // State flip-flops with asynchronous reset
    always @(posedge clk, posedge areset) begin
        state <= areset? WALK_LEFT: next_state;
    end
    // Output logic
    assign walk_left = (state == WALK_LEFT);
    assign walk_right = (state == WALK_RIGHT);
endmodule
```

5.2 Lemmings2

```
module top_module(
    input clk,
    input areset,    // Freshly brainwashed Lemmings walk left.
    input bump_left,
    input bump_right,
    input ground,
    output walk_left,
    output walk_right,
    output aaah
);
    parameter WL=2'd0, WR=2'd1; // walk left / right
    parameter FL=2'd2, FR=2'd3; // fall after walk left / right
    reg[1:0] state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            WL: next_state = ground? (bump_left ? WR: WL): FL;
            WR: next_state = ground? (bump_right? WL: WR): FR;
            FL: next_state = ground? WL: FL;
            FR: next_state = ground? WR: FR;
        endcase
    end
    // State Flip-flops with asynchronous reset
    always @(posedge clk or posedge areset)
        state <= areset? WL: next_state;
    // Output logic
    always @(*) begin
        case (state)
            WL: {walk_left, walk_right, aaah} = 3'b100;
            WR: {walk_left, walk_right, aaah} = 3'b010;
            default   : {walk_left, walk_right, aaah} = 3'b001;
        endcase
    end
endmodule
```

5.3 Lemmings3

Listing 5.2: 旅鼠總動員 3

```
module top_module(
    input clk,
    input areset,    // Freshly brainwashed Lemmings walk left.
    input bump_left,
    input bump_right,
    input ground,
    input dig,
    output walk_left,
    output walk_right,
    output aaah,
    output digging
);
    parameter WL=3'd0, DL=3'd1, FL=3'd2;// (walk / dig / fall) left
    parameter WR=3'd3, DR=3'd4, FR=3'd5;// (walk / dig / fall) right
    reg[2:0] state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            WL: next_state = ground?(dig?DL:(bump_left?WR:WL)):FL;
            DL: next_state = ground?DL:FL;
            FL: next_state = ground?WL:FL;
            WR: next_state = ground?(dig?DR:(bump_right?WL:WR)):FR;
            DR: next_state = ground?DR:FR;
            FR: next_state = ground?WR:FR;
            default: next_state = state;
        endcase
    end
    // State flip-flops
    always @(posedge clk or posedge areset)
        state <= areset? WL: next_state;
    // Output logic
    assign aaah = (state == FL) | (state == FR);
    assign digging = (state == DL) | (state == DR);
    assign walk_left = (state == WL);
    assign walk_right = (state == WR);
endmodule
```

5.4 Lemmings4

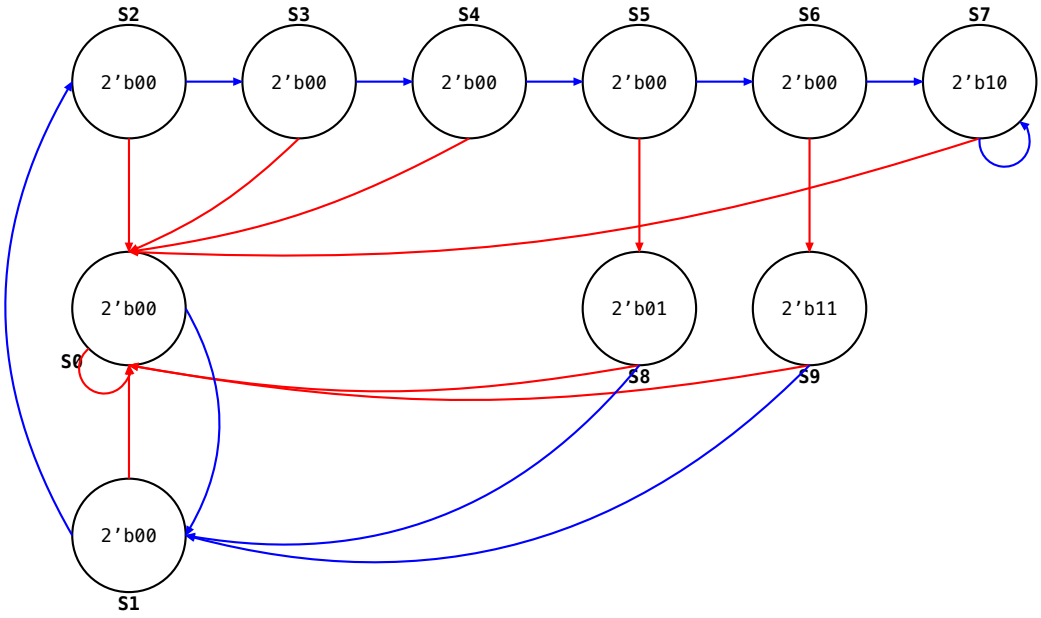
Listing 5.3: 旅鼠總動員 4

```
module top_module(
    input clk,
    input areset,    // Freshly brainwashed Lemmings walk left.
    input bump_left,
    input bump_right,
    input ground,
    input dig,
    output walk_left,
    output walk_right,
    output aaah,
    output digging
);
    parameter WL=3'd0, FL=3'd1, DL=3'd2;// walk/fall/dig left
    parameter WR=3'd3, FR=3'd4, DR=3'd5;// walk/fall/dig right
    parameter SP=3'd6;                  // splatter
    parameter SP_CNTR=5'd20;            //splatter time
    reg[2:0] state, next_state;
    reg[4:0] cntr;
    // State transition logic
    always@ (*) begin
        case (state)
            WL: next_state = ground?(dig?DL:(bump_left?WR:WL)):FL;
            DL: next_state = ground?DL:FL;
            FL: next_state = ground?((cntr>SP_CNTR)?SP:WL):FL;
            WR: next_state = ground?(dig?DR:(bump_right?WL:WR)):FR;
            DR: next_state = ground?DR:FR;
            FR: next_state = ground?((cntr>SP_CNTR)?SP:WR):FR;
            default: next_state = state;
        endcase
    end
    // counter
    always @(posedge clk or posedge areset) begin
        cntr <= areset? 5'd0: (cntr > SP_CNTR)? cntr: (ground? 5'd0: cntr+1'b1);
    end
    // State flip-flops
    always @(posedge clk or posedge areset) begin
        state <= areset? WL: next_state;
    end
    // Output logic
```

```
    assign aaah = (state == FL | state == FR);
    assign digging = (state == DL | state == DR);
    assign walk_left = (state == WL);
    assign walk_right = (state == WR);
endmodule
```

Problem 6 FSM onehot

下圖狀態機採用獨熱編碼，`state[0]` 到 `state[9]` 與狀態 S0 到 S9 一一對應。僅實作該狀態機的狀態轉換及輸出邏輯部分，當前狀態由 `state[9:0]` 提供，據此產生 `next_state[9:0]` 與兩位輸出。「憑觀察」推導其獨熱編碼的邏輯表示式。



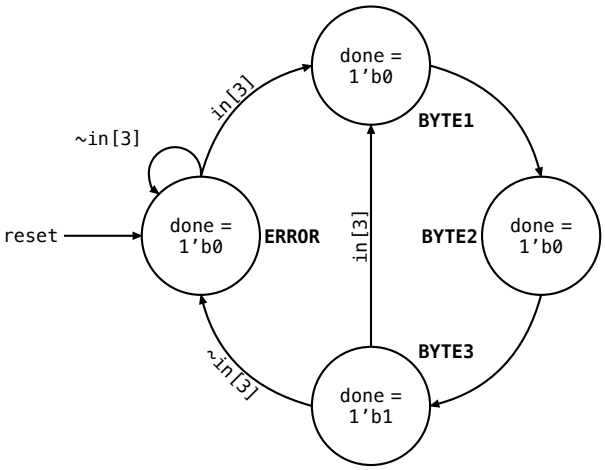

```
module top_module(
    input in,
    input [9:0] state,
    output [9:0] next_state,
    output out1,
    output out2
);
    // State transition logic
    assign next_state[0] = (~in) & ({state[9:7], state[4:0]} != 8'b0);
    assign next_state[1] = in & (state[0] | state[8] | state[9]);
    assign next_state[2] = in & state[1];
    assign next_state[3] = in & state[2];
    assign next_state[4] = in & state[3];
    assign next_state[5] = in & state[4];
    assign next_state[6] = in & state[5];
    assign next_state[7] = in & (state[6] | state[7]);
    assign next_state[8] = (~in) & state[5];
    assign next_state[9] = (~in) & state[6];
    // Output logic
    assign out1 = state[8] | state[9];
    assign out2 = state[7] | state[9];
endmodule
```

Problem 7　PS/2 封包解析器

PS/2 滑鼠協議發送三字節長的訊息。然而，在連續字節流中，訊息的始末並不明顯。唯一的提示是，每三字節訊息的第一個字節總是滿足 `bit[3]=1`（不過，後兩個字節的 `bit[3]` 根據資料可以是 1 或 0）。

(a) 現需要一個狀態機在輸入字節流中尋找訊息邊界。這裡使用的演算法是在看到 `bit[3]=1` 前丟棄所有的字節。這個滿足要求的字節會被視爲訊息的第一個字節，在 3 個字節都收到後，由訊號（`done`）示意接收訊息。該狀態機在成功收到第三個字節後應立即發出 `done` 訊號。

(b) 在以上能從 PS/2 字節流中解析出三字節訊息的狀態機的基礎上，增加一個無論何時收到封包都能輸出該 24 位元（3 字節）訊息的資料路徑（`out_bytes[23:16]` 是第一字節，`out_bytes[15:8]` 是第二字節，以此類推）。當 `done` 訊號拉高時，`out_bytes` 必須輸出有效值，其他時候輸出任意資料皆可（即都會被視爲無效值）。



```
module top_module(
    input clk,
    input [7:0] in,
    input reset,    // Synchronous reset
    output done
);  // (a) part, FSM
    parameter ERROR=2'd0, BYTE1=2'd1, BYTE2=2'd2, BYTE3=2'd3;
    reg[1:0] state, next_state;
    // State transition logic (combinational)
    always @(*) begin
        case (state)
            BYTE1: next_state = BYTE2;
            BYTE2: next_state = BYTE3;
            default: next_state = in[3]? BYTE1: ERROR;
        endcase
    end
    // State flip-flops (sequential)
    always @(posedge clk) begin
        state <= reset? ERROR: next_state;
    end
    // Output logic
    assign done = (state == BYTE3);
    // (b) part: Datapath to store incoming bytes
    reg[23:0] data;
    always @(posedge clk) data <= {data[15:0], in};
    assign out_bytes = data;
endmodule
```

Problem 8 Serial Receiver

在許多（老式）通訊協議中，每一個資料字節都與一個起始位元與一個終止位元一同傳送，幫助接收器從位元流中界定字節。其中一個常見的做法，是用 1 位起始位元（0）、8 位資料位元、一位終止位元（1）。即便沒在傳輸，線路仍保持在邏輯 1（即閒置）。

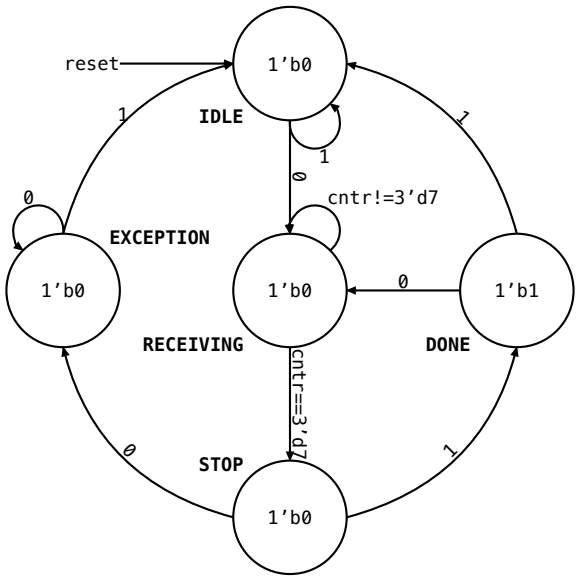
(a) 設計一個有限狀態機，當傳入位元流時，辨識位元是否正確傳送。它要能辨識出起始位元、等待 8 位資料後確認終止位元正確。若終止位元與預期不相符，則該狀態機必須等到收到終止位元，才能開始接收下一資料字節。

(b) 上一小題完成了可以正確從位元流裡辨識何時成功接收字節的狀態機，在此基礎上增加一個資料路徑，在 `done=1` 時輸出正確接收到的資料字節。`out_byte` 只需要在 `done=1` 時有效，其他情況可以爲任意值。請注意該協議先送低位元。

(c) 我們想要給接收器增加奇偶校驗。奇偶校驗會在每一個字節上額外增加一位元。這裡使用奇校驗，亦即這 9 位元裡要有奇數個 1。例如，`101001011` 滿足奇校驗（有 5 個 1），`001001011` 則不滿足。

修改上兩小題的 FSM 以進行奇校驗。唯有正確接收、通過奇校驗時才觸發 `done` 訊號。與上兩小題類似，該 FSM 應該識別起始位元、等待 9 位元（資料與校驗）、確認終止位元是否正確。倘若終止位元與預期不相符，FSM 必須等到收到終止位元才能開始接收下一字節。

輸入流的奇校驗模組已經提供，是一個帶復位的 T 觸發器。預期的使用方式是，給它一個輸入位元流，在適合的時機復位以便正確計算每一字節的 1 的數量。



Listing 8.1: 校驗器

```
module parity (
    input clk,
    input reset,
    input in,
    output reg odd
);
```

```
    always @(posedge clk)
        if (reset) odd <= 0;
        else if (in) odd <= ~odd;
endmodule
```

8.1 Simple Serial Receiver

```
module top_module(
    input clk,
    input in,
    input reset,    // Synchronous reset
    output done
);  // (a) FSM state
    parameter IDLE=3'd0, RECV=3'd1, STOP=3'd2;
    parameter DONE=3'd3, EXCE=3'd4, CNTR=3'd7;
    reg[2:0] state, next_state, cntr;
    // State transition logic
    always @(*) begin
        case (state)
            RECV: next_state = (cntr == CNTR)? STOP: RECV;
            STOP: next_state = in? DONE: EXCE;
            EXCE: next_state = in? IDLE: EXCE;
            default: next_state = in? IDLE: RECV;
        endcase
    end
    // counter
    always @(posedge clk)
        cntr <= (reset | state!=RECV)? 3'd0: cntr+1'b1;
    // State flip-flops
    always @(posedge clk) state <= reset? IDLE: next_state;
    // Output logic
    assign done = (state==DONE);
    // (b) Datapath to latch input bits.
    always @(posedge clk)
        out_byte[cntr] <= (state==RECV)? in: out_byte[cntr];
endmodule
```

8.2 Serial Receiver with Parity Check

```
module top_module(
    input clk,
    input in,
    input reset,    // Synchronous reset
    output [7:0] out_byte,
    output done
);  // (a) FSM state
    parameter IDLE=3'd0, RECV=3'd1, STOP=3'd2;
    parameter DONE=3'd3, EXCE=3'd4, CNTR=4'd8;
    reg[2:0] state, next_state;
    reg[3:0] cntr;
    reg correct, new_correct;
    // State transition logic
    always @(*) begin
        case (state)
            RECV: next_state = (cntr == CNTR)? STOP: RECV;
            STOP: next_state = in? DONE: EXCE;
            EXCE: next_state = in? IDLE: EXCE;
            default: next_state = in? IDLE: RECV;
        endcase
    end
    // counter
    always @(posedge clk)
        cntr <= (reset | state!=RECV)? 4'd0: cntr+1'b1;
    // State flip-flops
    always @(posedge clk) state <= reset? IDLE: next_state;
    // Output logic
    assign done = (state==DONE) & correct;
    // (b) Datapath to latch input bits.
    always @(posedge clk)
        out_byte[cntr] <= (state==RECV)? in: out_byte[cntr];
    // (c) Add parity checking.
    parity U0(
        .clk(clk),
        .reset(state!=RECV),
        .in(in),
        .odd(new_correct)
    );
    always @(posedge clk) correct <= reset? 1'b0: new_correct;
endmodule
```

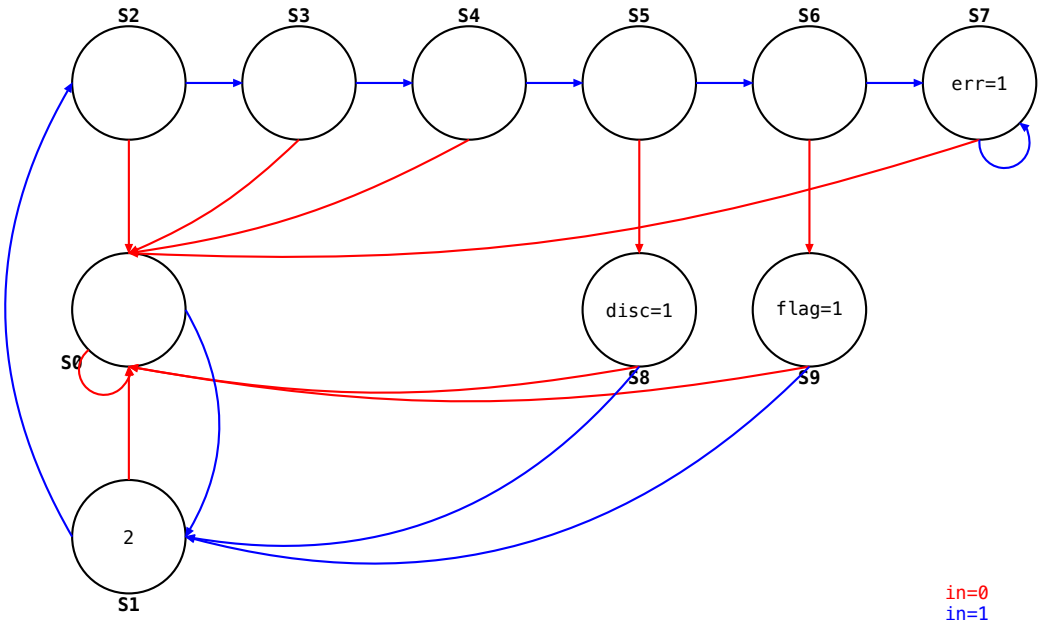
Problem 9 Sequence Recognition

同步高級數據鏈路控制（High-Level Data Link Control, HDLC）訊框（framing）含有一個數據連續位元流解碼器，尋找指示訊框始末的位元模式（即封包）。看到 6 個連續的 1（例如 `01111110`）是指示訊框邊界的「旗標」（flag）。爲避免資料流意外含有「旗標」，發訊端每連續 5 個 `1` 就會插入一個 `0`，收訊端則應該檢測到該 0 並忽略之。在發現連續 7 個及以上的 1 時，也應該拋出錯誤訊號。設計一個有限狀態機識別以下三種序列：

- `0111110`：發現應該拋棄一個位元（disc）
- `01111110`：訊框邊界旗標（flag）
- `01111111...`：錯誤（大於等於 7 個 1）（err）

若該 FSM 復位，它應該回到前一個輸出爲 0 的狀態。

解答：採用展開所有狀態的寫法，則狀態轉換圖如下（類似 FSM onehot）。



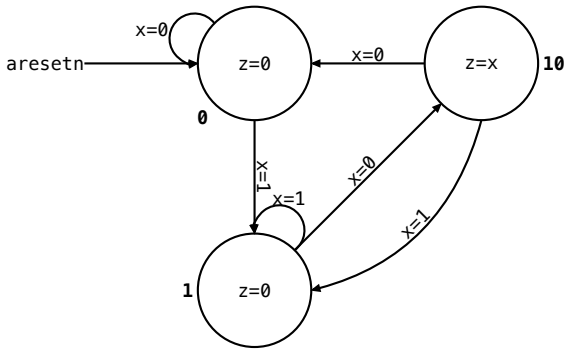

```
module top_module(
    input clk,
    input reset,    // Synchronous reset
    input in,
    output disc,
    output flag,
    output err
);  // using onehot encoding
    reg[9:0] state, next_state;
    // State transition logic
    assign next_state[0] = (~in) & ({state[9:7], state[4:0]} != 8'b0);
    assign next_state[1] = in & (state[0] | state[8] | state[9]);
    assign next_state[6:2] = {5{in}} & state[5:1];
    assign next_state[7] = in & (state[6] | state[7]);
```

```
    assign next_state[9:8] = {2{~in}} & state[6:5];
    // State flip-flop
    always @(posedge clk) state <= reset? 10'd1: next_state;
    // Output logic
    assign {flag, disc, err} = state[9:7];
endmodule
```

Problem 10 Design a Mealy FSM

實作一個米利型有限狀態機，從輸入訊號 **x** 識別 **101** 序列。該 FSM 的輸出 **z** 會在檢測到序列 **101** 時觸發。此外，該 FSM 要有一個低電位有效異步復位訊號。該狀態機只容許 3 個狀態，必須能識別重疊序列。

解答：

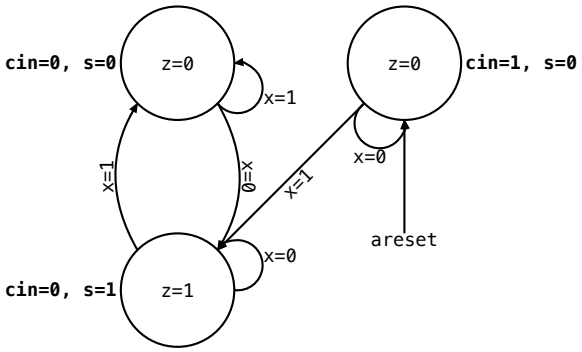



```
module top_module (
    input clk,
    input aresetn,    // Asynchronous active-low reset
    input x,
    output z
);  // Mealy FSM
    parameter SEQ0=2'd0, SEQ1=2'd1, SEQ10=2'd2;
    reg[1:0] state, next_state;
    // State transition logic
    assign next_state = x? SEQ1: ((state == SEQ1)? SEQ10: SEQ0);
    // State flip-flop
    always @(posedge clk or negedge aresetn)
        state <= (~aresetn)? SEQ0: next_state;
    // Output logic
    assign z = x & (state==SEQ10);
endmodule
```

Problem 11 Serial 2's Complementer

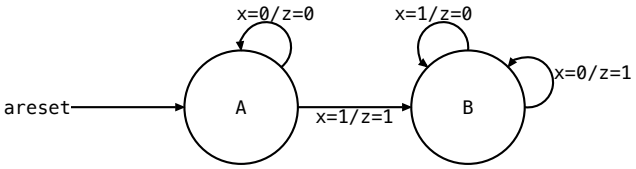
設計一個 1 位輸入、1 位輸出的循序二補數器（serial 2's complementer）摩爾狀態機。輸入（x）是該數字從低位到高位的一系列位元（每個時鐘週期一位），輸出（z）則是輸入的二補數。該狀態機接收任意長度的輸入，需要一個異步復位訊號，它界定了輸入的長度。

11.1 Moore FSM




```
module top_module (
    input clk,
    input areset,
    input x,
    output z
);  // FSM
    parameter S00=2'd0, S01=2'd1, S10=2'd2;
    reg[1:0] state, next_state;
    // State transition logic
    assign next_state = (state==S10)? (x? S01: S10): (x? S00: S01);
    // State flip-flop
    always @(posedge clk or posedge areset)
        state <= areset? S10: next_state;
    // Output logic
    assign z = (state==S01);
endmodule
```

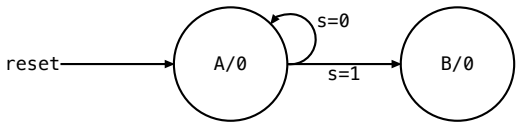
11.2 Mealy FSM




```
module top_module (
    input clk,
    input areset,
    input x,
    output z
);  // Mealy FSM
    parameter A=1'b0, B=1'b1;
    reg state, next_state;
    // State transition logic
    assign next_state = (state==A & (~x))? A: B;
    // State flip-flop
    always @(posedge clk or posedge areset)
        state <= areset? A: next_state;
    // Output logic
    assign z = (state==A)? x: (~x);
endmodule
```

Problem 12 Q3 FSM

(a) 考慮一個輸入爲 `s, w` 的有限狀態機。假設該狀態機從復位態 A 開始。只要 `s=0`，該 FSM 會始終處於狀態 A；一旦 `s=1`，狀態立即移動到 B，如下所示。在狀態 B，該狀態機在後續 3 個時鐘週期檢查輸入 `w` 的值。如果其中正好兩個週期有 `w=1`，則下一個週期輸出 `z=1`，否則 `z=0`。狀態機會繼續檢查後續三個時鐘週期，以此類推。



(b) 給定以下狀態轉換表，實作有限狀態機，復位態爲 000。

Present state y[2:0]	Next state Y[2:0]		Output z
	x=0	x=1	
000	000	001	0
001	001	100	0
010	010	001	0
011	001	010	1
100	011	100	1

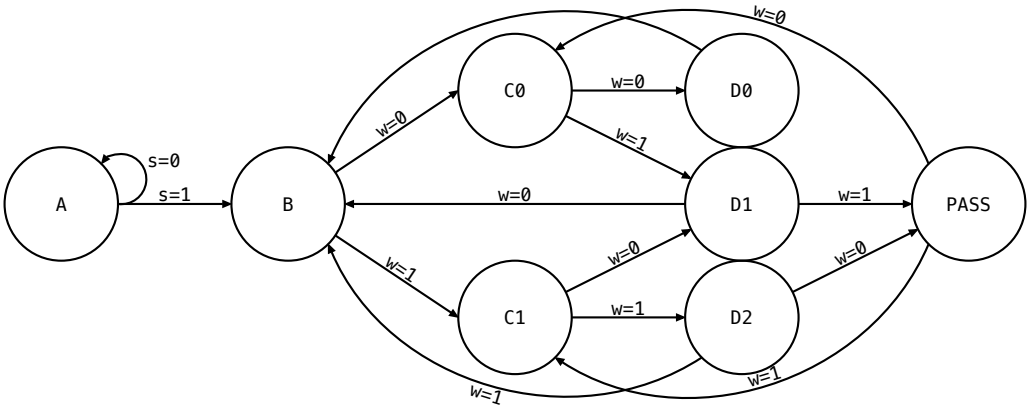
(c) 仍使用上表，實作計算 `Y[0], z` 的邏輯電路。

思路：

(a) 僅計算三個時鐘週期內的 1 的數量，無需考慮順序或位元重疊。因此，僅需考慮第零個週期看到 0/1（記爲 C0/C1）、第一個週期看到 0/1/2 個 1（記爲 D0/D1/D2）、第二個週期是否看到 2 個 1（B/PASS）等狀態。加上復位態 A，一共 8 個狀態。

(b) 後兩題則是簡單的查表狀態機。

12.1 FSM a




```
module top_module (
    input clk,
    input reset,   // Synchronous reset
    input s,
    input w,
    output z
);  // FSM
    parameter A=3'd0, B=3'd1, C0=3'd2, C1=3'd3;
    parameter D0=3'd4, D1=3'd5, D2=3'd6, PASS=3'd7;
    reg[2:0] state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            A: next_state = s? B: A;
            C0: next_state = w? D1: D0;
            C1: next_state = w? D2: D1;
            D0: next_state = B;
            D1: next_state = w? PASS: B;
            D2: next_state = w? B: PASS;
            default: next_state = w? C1: C0;
        endcase
    end
    // State flip=flop
    always @(posedge clk) state <= reset? A: next_state;
    // Output logic
    assign z = (state==PASS);
endmodule
```

12.2 FSM b

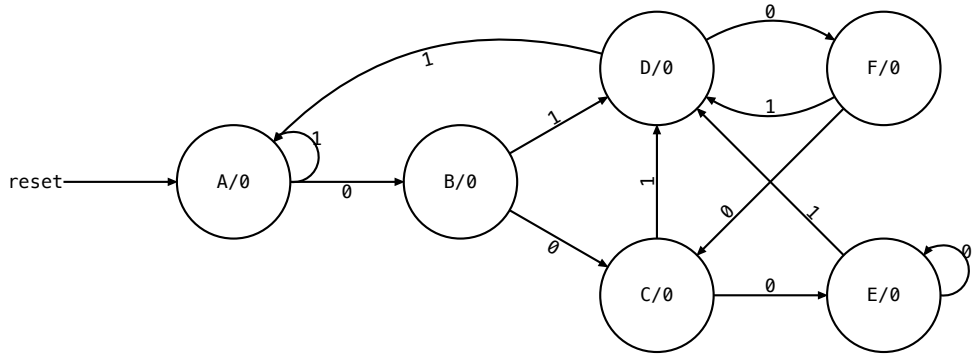
```
module top_module (
    input clk,
    input reset,   // Synchronous reset
    input x,
    output z
);  // FSM
    reg[2:0] state, next_state;
    // State transition logic
    always @(*) begin
        case (state)
            3'b001: next_state = x? 3'b100: 3'b001;
            3'b010: next_state = x? 3'b001: 3'b010;
            3'b011: next_state = x? 3'b010: 3'b001;
            3'b100: next_state = x? 3'b100: 3'b011;
            default: next_state = x? 3'b001: 3'b000;
        endcase
    end
    // State flip-flop
    always @(posedge clk) state <= reset? 3'b000: next_state;
    // Output logic
    assign z = (state==3'b011) | (state==3'b100);
endmodule
```

12.3 FSM c

```
module top_module (
    input clk,
    input [2:0] y,
    input x,
    output Y0,
    output z
);  // logic functions
    assign Y0 = (y==3'b000 | y==3'b010)? x: ~x;
    assign z = (y==3'b011) | (y==3'b100);
endmodule
```

Problem 13 Q6 FSM

下圖所示狀態機，一位輸入 `w`、一位輸出 `z`。



(a) 假設現在要以三個觸發器、狀態編碼 `y[3:1]=000,001,...,101` 表示狀態 A～F 實作狀態機，畫出狀態轉換表，並實作 `y[2]` 的狀態轉換邏輯。

(b) 假設狀態以獨熱編碼賦值，`y[6:1]=000001,000010,...,100000` 分別對應 A～F。憑觀察寫下 Y2 與 Y4 的邏輯。

(c) 完整實作這個 FSM。

13.1 Q6 FSM next state logic

狀態轉換表爲

present state y[3:1]	next state Y[3:1]		Output z
	w=0	w=1	
000	001	000	0
001	010	011	0
010	100	011	0
011	101	000	0
100	100	011	1
101	010	011	1

```
module top_module (
    input [3:1] y,
    input w,
    output Y2
);  // Only Y2 logic
    always @(*) begin
        case (y)
            3'b000: Y2 = 1'b0;
            3'b001: Y2 = 1'b1;
            3'b011: Y2 = 1'b0;
            3'b101: Y2 = 1'b1;
            default: Y2 = w;
        endcase
    end
endmodule
```

13.2 Q6 FSM onehot next state logic

狀態轉換表爲

present state y[6:1]	next state Y[6:1]		Output z
	w=0	w=1	
000001	000010	000001	0
000010	000100	001000	0
000100	010000	001000	0
001000	100000	000001	0
010000	010000	001000	1
100000	000100	001000	1

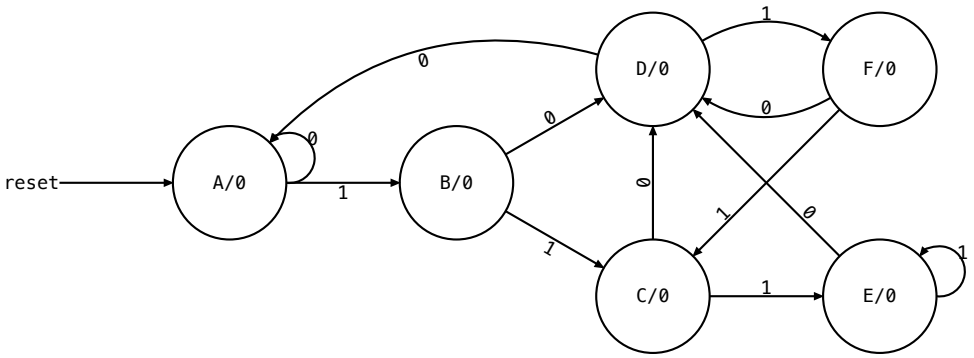
```
module top_module (
    input [6:1] y,
    input w,
    output Y2,
    output Y4
);
    assign Y2 = y[1] & (~w);
    assign Y4 = w & (y[2] | y[3] | y[5] | y[6]);
endmodule
```

13.3 Q6 FSM completed

```
module top_module (
    input clk,
    input reset,     // synchronous reset
    input w,
    output z
);  // FSM (a)
    parameter A=3'd0, B=3'd1, C=3'd2, D=3'd3, E=3'd4, F=3'd5;
    reg[2:0] y, Y;
    // State transition logic
    always @(*) begin
        case (y)
            A: Y = w? A: B;
            C: Y = w? D: E;
            D: Y = w? A: F;
            E: Y = w? D: E;
            default: Y = w? D: C;
        endcase
    end
    // State flip-flop
    always @(posedge clk) y <= reset? A: Y;
    // Output logic
    assign z = y==E | y==F;
endmodule
```

Problem 14 Q2 FSM 2012

考慮以下狀態轉換圖。(顯然它與 Q6FSM 只有輸入值剛好相反)



(a) 假設使用獨熱編碼 `y[5:0]` 表示狀態，憑觀察寫下 Y1、Y3 的邏輯表示式。

(b) 寫下該 FSM 完整的 Verilog 代碼。請使用分離的 **always** 塊實作狀態表與狀態觸發器。

14.1 Q2b: One-hot FSM equations

```
module top_module (
    input [5:0] y,
    input w,
    output Y1,
    output Y3
);
    assign Y1 = w & y[0];
    assign Y3 = (~w) & (y[1] | y[2] | y[4] | y[5]);
endmodule
```

14.2 Q2a FSM

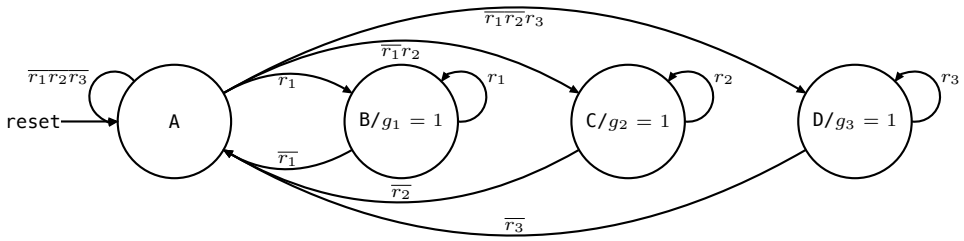
由於狀態機幾乎與 Q6FSM 一樣，這裡採用獨熱編碼，避免做法重複。

```
module top_module (
    input clk,
    input reset,   // Synchronous active-high reset
    input w,
    output z
);  // one-hot
    reg[5:0] s, next_s;
    // State transition logic
    assign next_s[0] = (~w) & (s[0] | s[3]);
    assign next_s[1] = w & s[0];
    assign next_s[2] = w & (s[1] | s[5]);
    assign next_s[3] = (~w) & (s[1] | s[2] | s[4] | s[5]);
    assign next_s[4] = w & (s[2] | s[4]);
    assign next_s[5] = w & s[3];
    // State flip-flop
    always @(posedge clk) s <= reset? 6'd1: next_s;
    // Output logic
    assign z = s[4] | s[5];
endmodule
```

Problem 15 Q2 FSM 2013

15.1 Part (a)

考慮下圖所示的狀態圖。



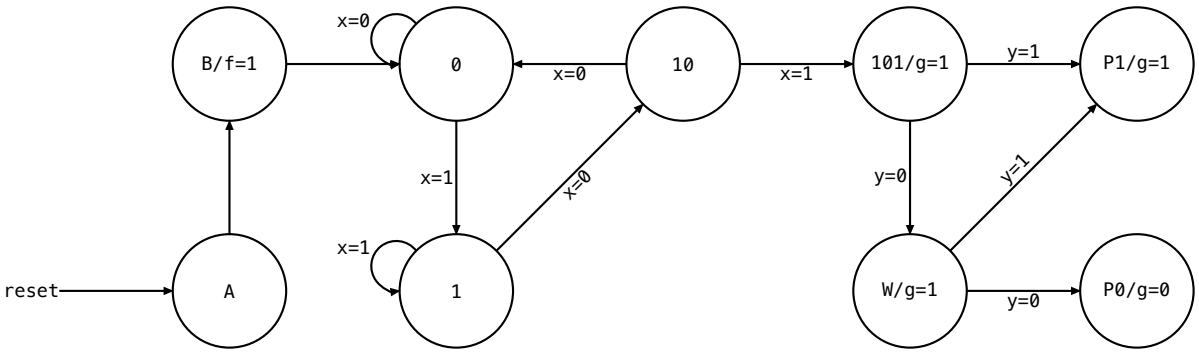
這個狀態機是一個仲裁器電路（arbiter circuit），用以控制三個提出請求的設備對資源的訪問權。每個設備以設定訊號 `r[i]=1` 請求資源，其中 $i = 1, 2, 3$。每個 `r[i]` 都是 FSM 的輸入，分別表示三台設備。只要沒有收到請求，該 FSM 處於狀態 A。若接收到一個或多個請求，該 FSM 決定哪個設備允許使用資源，並改變狀態使得該設備的 `[g[i]]` 訊號爲 1。每個 `g[i]` 是該 FSM 的輸出。該系統有優先級，設備 1 比設備 2 更優先，設備 3 的優先級最低。於是舉例來說，當狀態機在狀態 A 時，設備 3 只有在當它是唯一提出請求的設備時，才會獲准。若某個設備獲得 FSM 准許，該設備只要提出請求（`r[i]=1`）就會持續獲准。請寫出完整的 Verilog 代碼表示這個狀態機。

```
module top_module (
    input clk,
    input resetn,    // active-low synchronous reset
    input [3:1] r,   // request
    output [3:1] g   // grant
);  // FSM
    parameter A=2'd0, B=2'd1, C=2'd2, D=2'd3;
    reg[1:0] s, next_s;
    // State transition logic
    always @(*) begin
        case (s)
            A: next_s = r[1]? B: (r[2]? C: (r[3]? D: A));
            B: next_s = r[1]? B: A;
            C: next_s = r[2]? C: A;
            D: next_s = r[3]? D: A;
        endcase
    end
    // State flip-flop
    always @(posedge clk) s <= (~resetn)? A: next_s;
    // Output logic
    assign g = {(s==D), (s==C), (s==B)};
endmodule
```

15.2 Part (b)

考慮一個控制某種馬達的有限狀態機，兩位輸入 `x, y` 來自馬達，產生兩位輸出 `f, g` 控制馬達。另有一個時鐘輸入 `clk` 與一個復位輸入訊號 `resetn`。

該 FSM 應如下運作。只要復位訊號有效，則一直處於初始狀態 A。當復位訊號失效，在下一個時鐘邊緣，該 FSM 應設定 `f=1`，維持一個時鐘週期。「此後」，該 FSM 監測輸入 `x`，若該訊號連續三個週期產生 `1,0,1` 值，在下一個時鐘週期應設 `g=1`。維持 `g=1` 時，該 FSM 應監測輸入訊號 `y`，若 `y` 在最多兩個時鐘週期之內取值爲 1，該 FSM 應該永久維持 `g=1`（直到復位）。但如果 `y` 沒有在兩個時鐘週期內取 1，則該 FSM 應該永久取 0（直到復位）。




```
module top_module (
    input clk,
    input resetn,    // active-low synchronous reset
    input x,
    input y,
    output f,
    output g
);  // FSM
    parameter A=4'd0, B=4'd1, C0=4'd2, C1=4'd3, C2=4'd4;
    parameter C5=4'd5, P1=4'd6, W=4'd7, P0=4'd8;
    reg[3:0] s, next_s;
    // State transition logic
    always @(*) begin
        case (s)
            A: next_s = B;
            B: next_s = C0;
            C0: next_s = x? C1: C0;
            C1: next_s = x? C1: C2;
            C2: next_s = x? C5: C0;
            C5: next_s = y? P1: W;
            W: next_s = y? P1: P0;
            default: next_s = s;
        endcase
    end
    // State flip-flop
```

```
    always @(posedge clk) s <= (~resetn)? A: next_s;
    // Output logic
    assign f = (s==B);
    assign g = (s==C5 | s==P1 | s==W);
endmodule
```