

**11010EECS207001Logic Design Lab**

**LAB4 : Finite State Machines**

**TEAM9**

**組長：108062213 顏浩昀**

**組員：106062304 黃鈺舒**

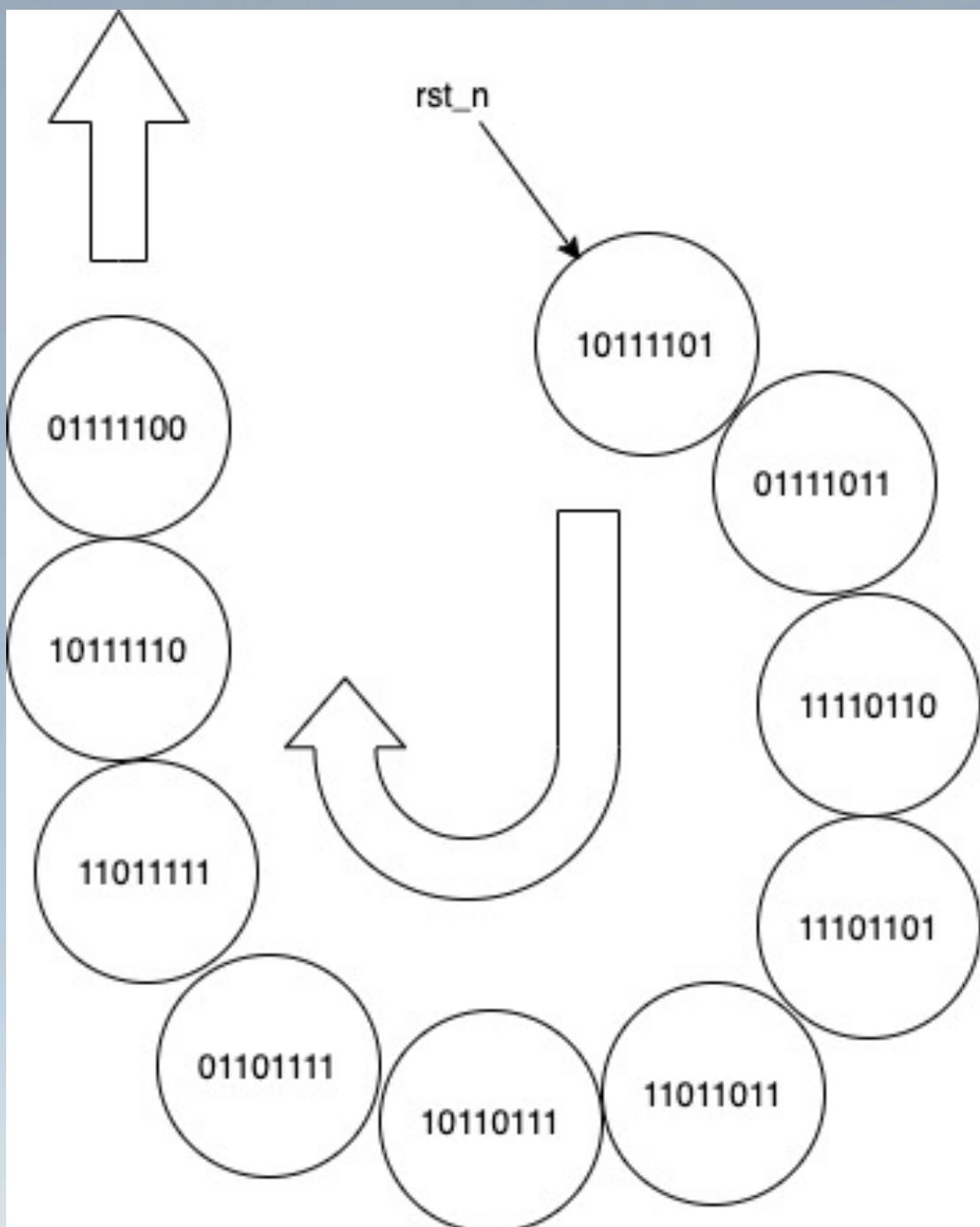
**Prof. Chun-Yi Lee**

**2021.11.11**

# **Basic Part**

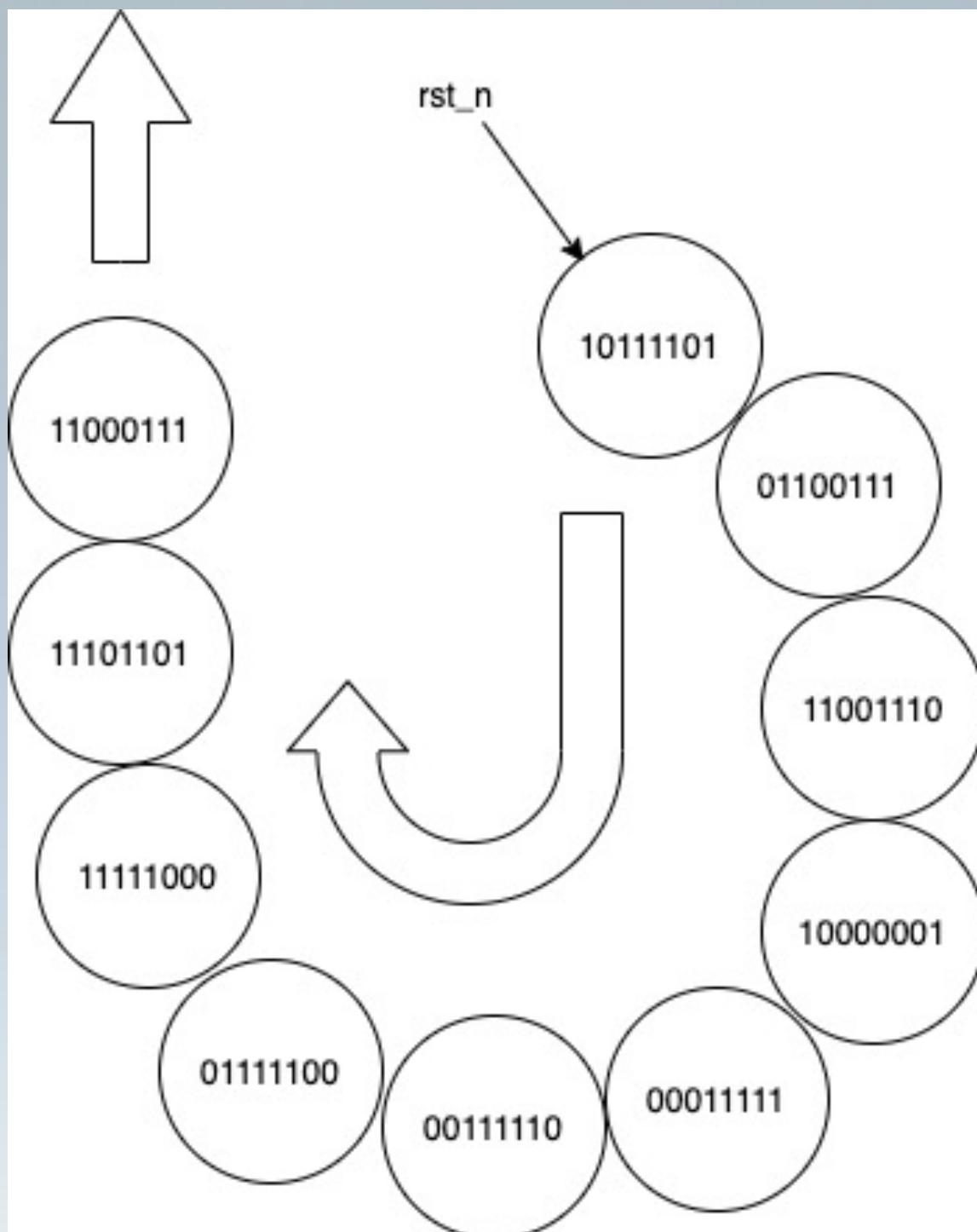
### Basic Q3. Many-to-One LFSR

#### State Transition Diagram



## Basic Q4. One-to-Many LFSR

State Transition Diagram



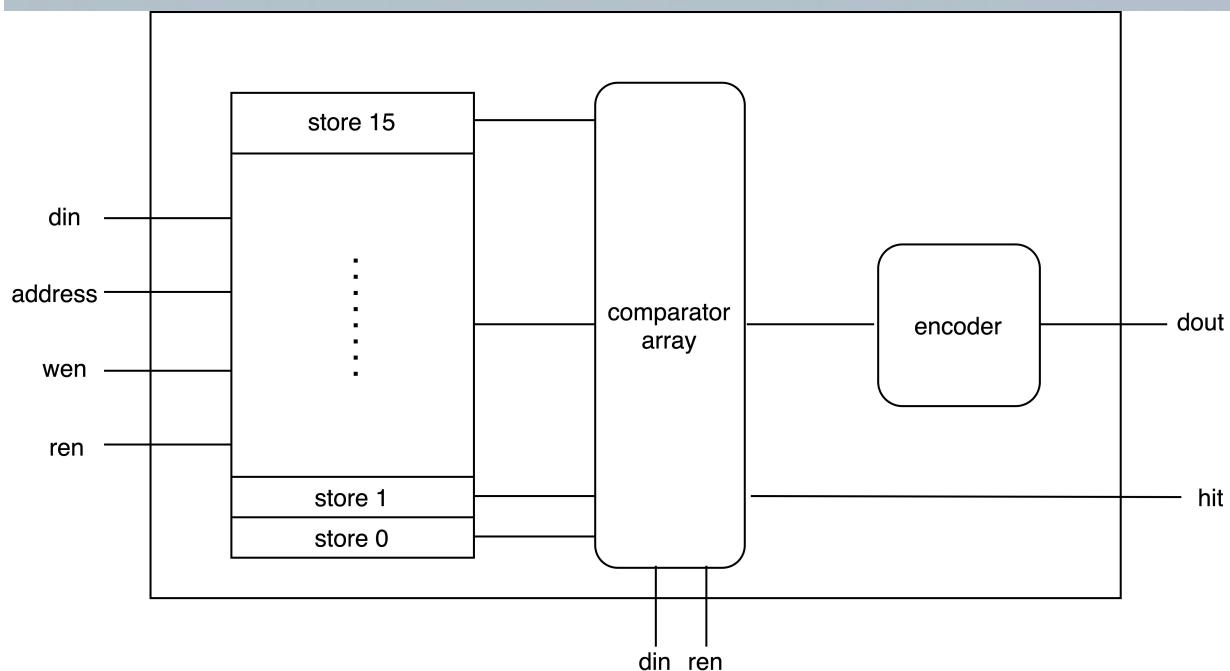
Explanation :

如果一開始reset時將DFFs設為8'd0，無論是一對多或許多對一，都會不斷的只有0這個數字在傳遞，因為兩者都是透過XOR讓state產生變化，如果將全部的值都設為0，XOR後的值仍然必為0，也就代表整個DFFs不會出現1，也就沒有任何state transition了。

# **Content-addressable memory**

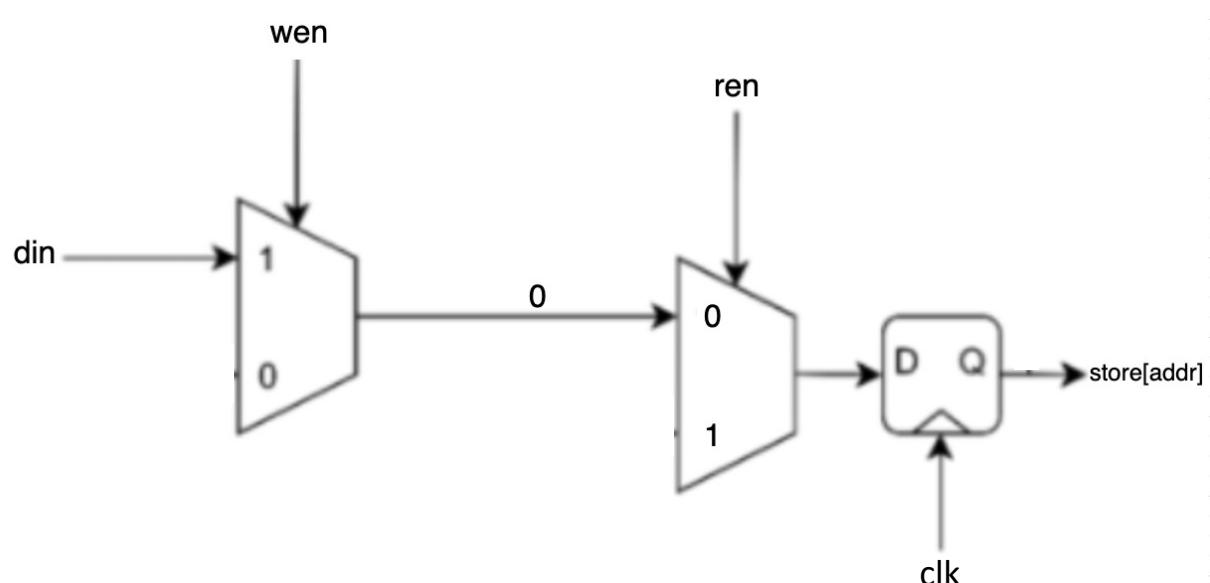
## I. Diagram

### (1) 基本架構



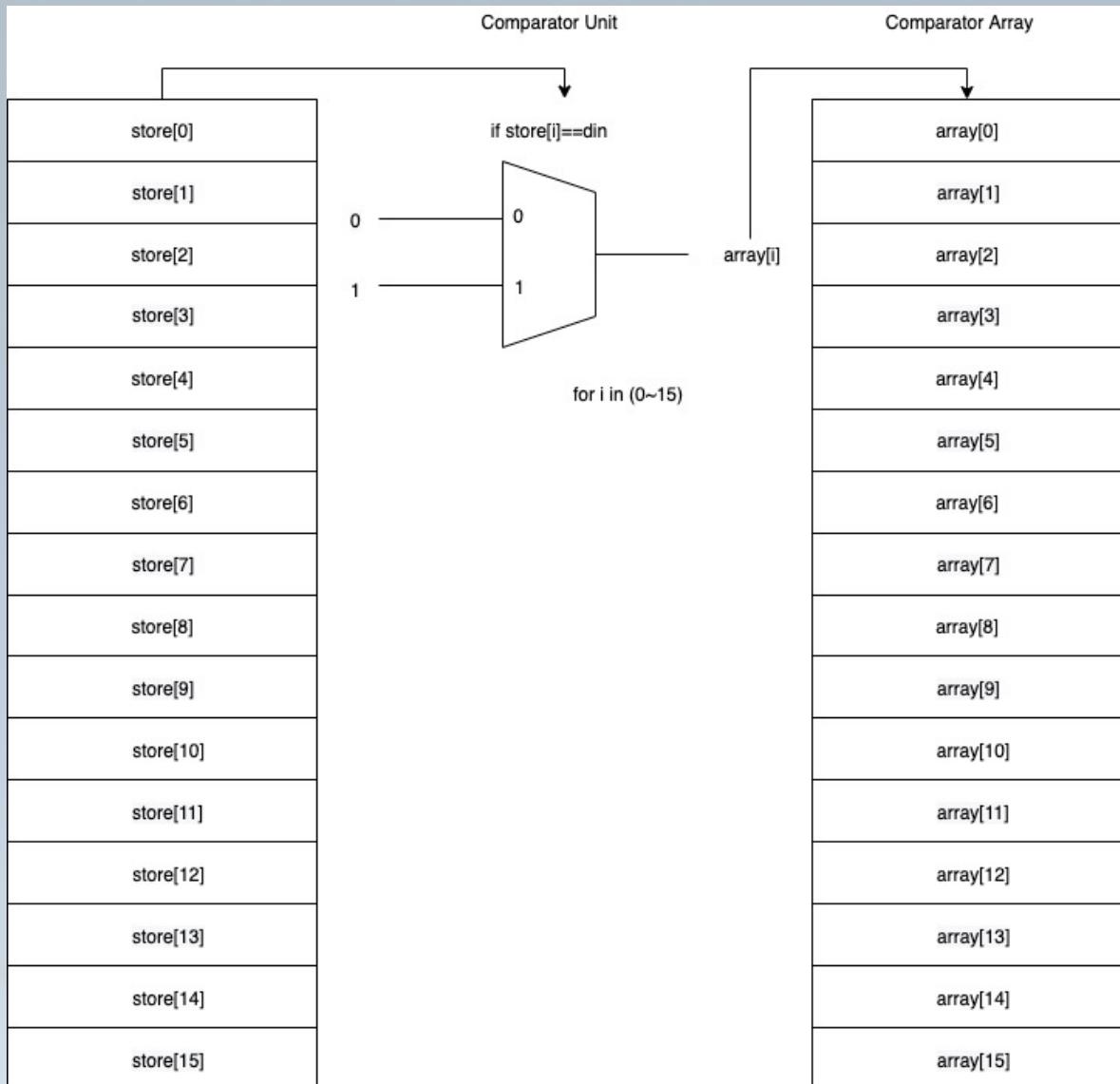
### (2) 物件電路設計

#### (i) Store



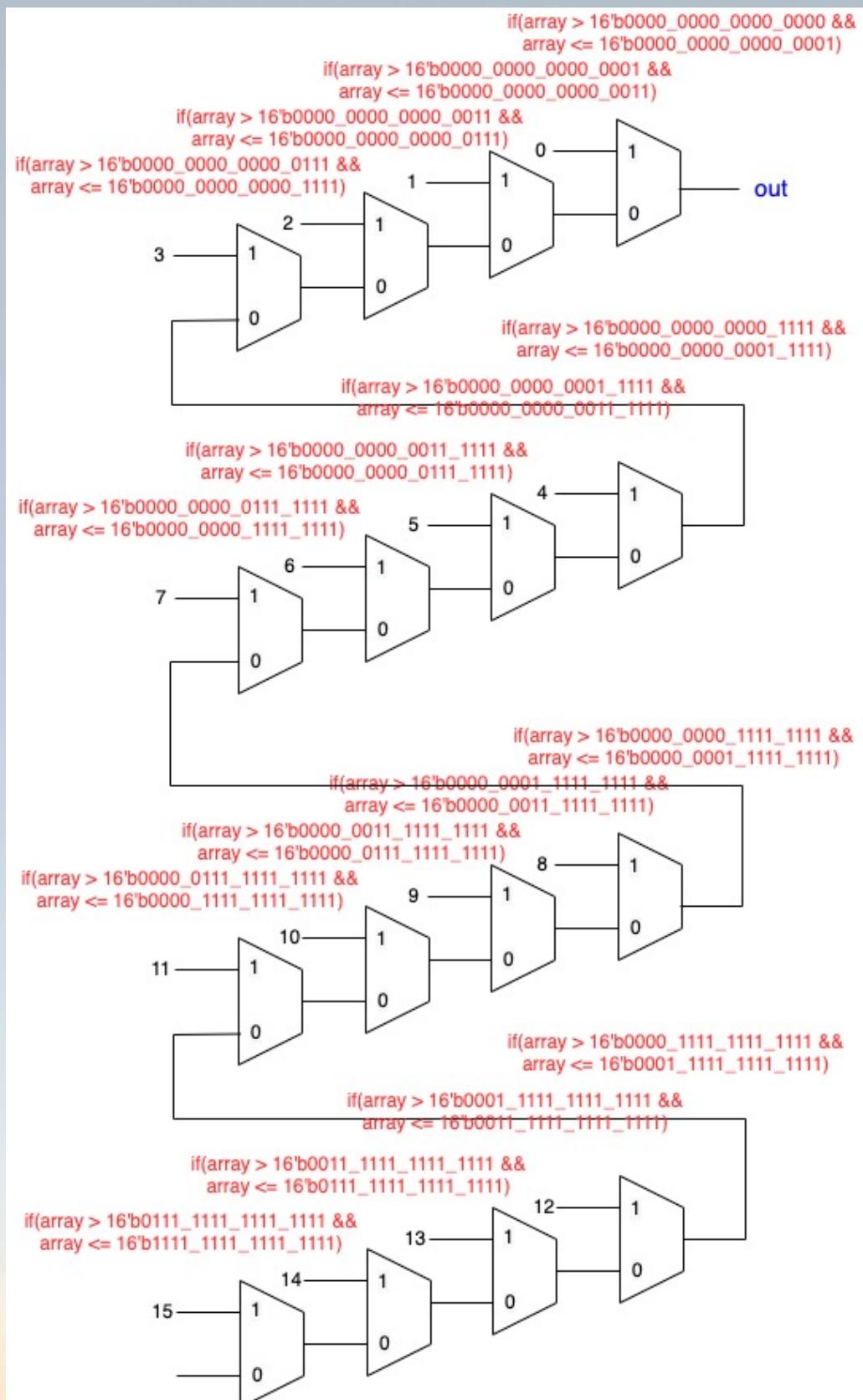
## I. Diagram

### (ii) comparator array



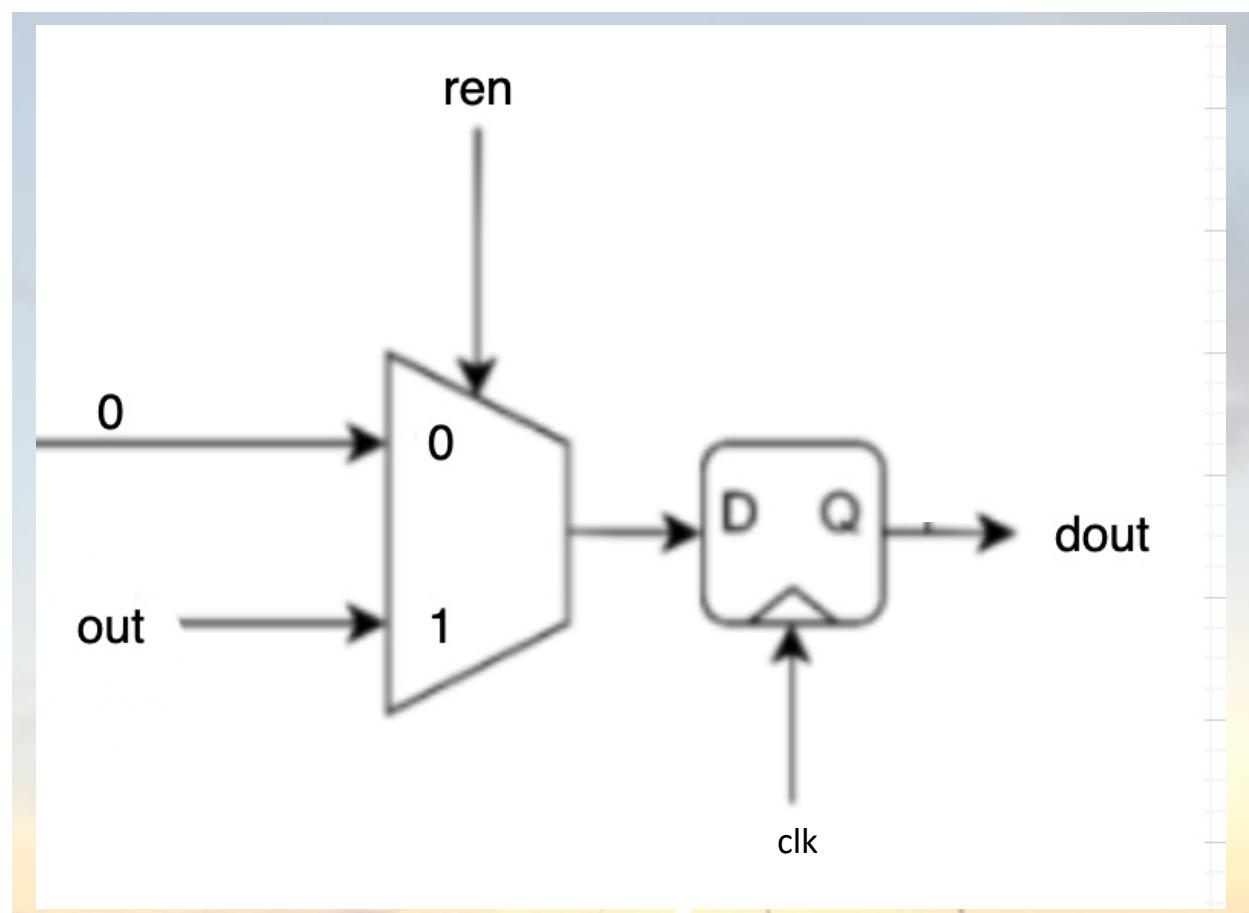
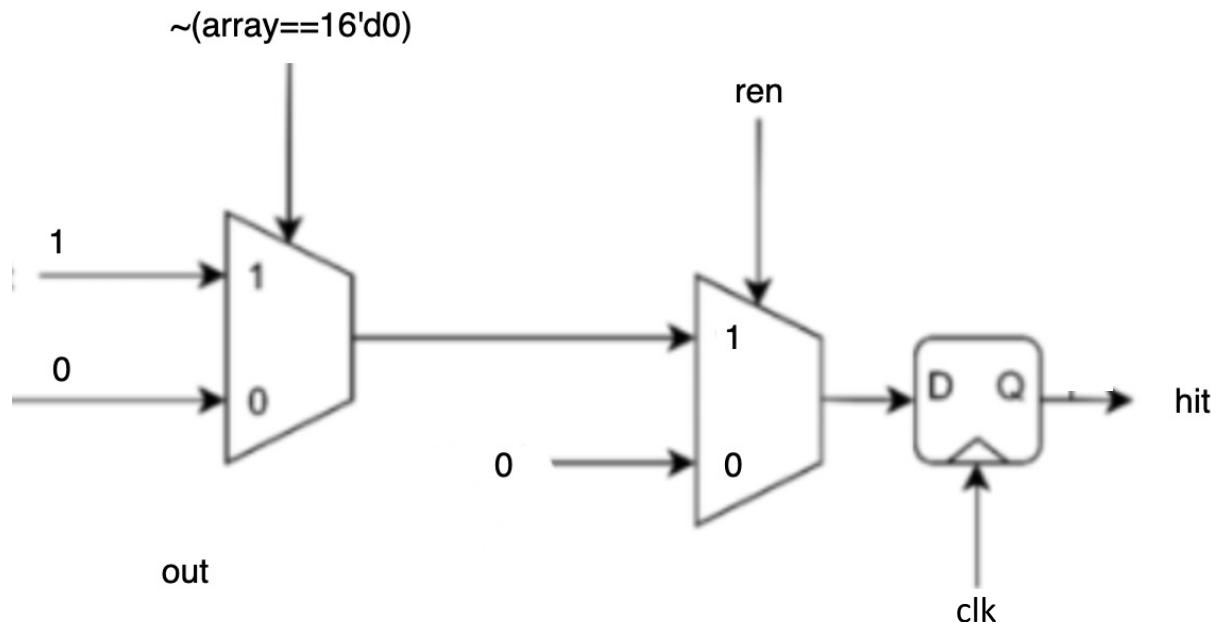
## I. Diagram

### (iii) Encoder



## I. Diagram

(iv) hit and dout



## II. Explanation

我設計的CAM分為兩大個module，分別是Memory+Comparator Array及Encoder。如果執行write，將store[addr]值設為din，這時可以直接判斷dout及hit為0。如果執行read，則會將din與store中各個address中的值比較，如果address i內的值與din相同則將comparator[i]設為1，反之為0。

```
always@(*) begin
    if(store[0] == din) begin
        array[0] = 1'b1;
    end
    else begin
        array[0] = 1'b0;
    end

    if(store[1] == din) begin
        array[1] = 1'b1;
    end
    else begin
        array[1] = 1'b0;
    end
```

comparator Array的結果則會放入Encoder去判斷哪些位置的值與din相同，並且因為只需判斷address最大者即可，因此我們只需要判斷array是否在特定區間就可以判斷dout（右界代表第i個bit為1的最大值，代表要使out = i時array的最大值）。

```
always@(*) begin
    if(array > 16'b0000_0000_0000_0000 && array <= 16'b0000_0000_0000_0001) begin
        out = 4'd0;
    end
    else if(array > 16'b0000_0000_0000_0001 && array <= 16'b0000_0000_0000_0011) begin
        out = 4'd1;
    end
    else if(array > 16'b0000_0000_0000_0011 && array <= 16'b0000_0000_0000_0111) begin
        out = 4'd2;
    end
```

hit的判斷比較簡單，只要comparator array==0則hit=0，否則hit都是1。

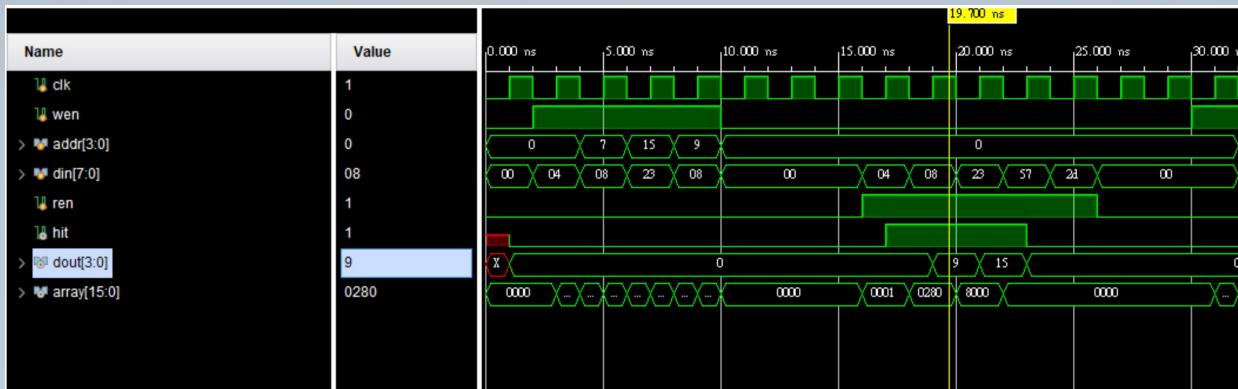
```
if(ren == 1'b1) begin
    dout <= out;
    hit <= ~(array==16'd0);
end
```

### III. Testbench and Waveform

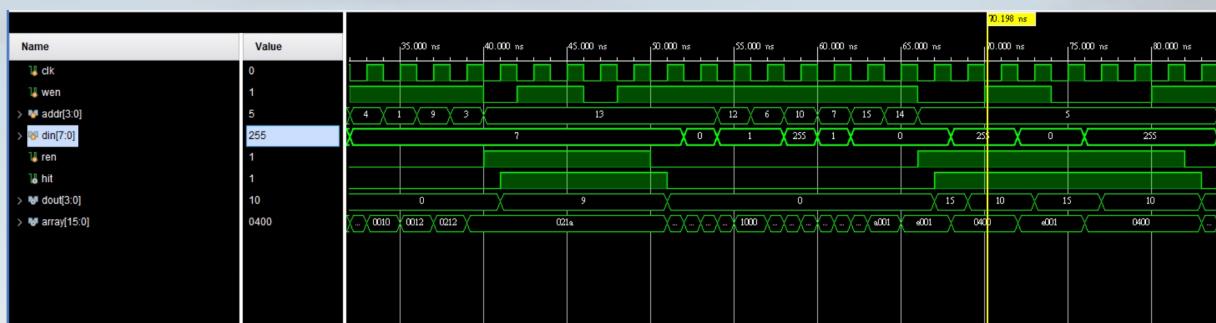
本題我們加入array值偵測是否有輸出正確addr bitmap，有放值的addr位置為1，其餘為0，共16 bits。

testbench分成四個階段：

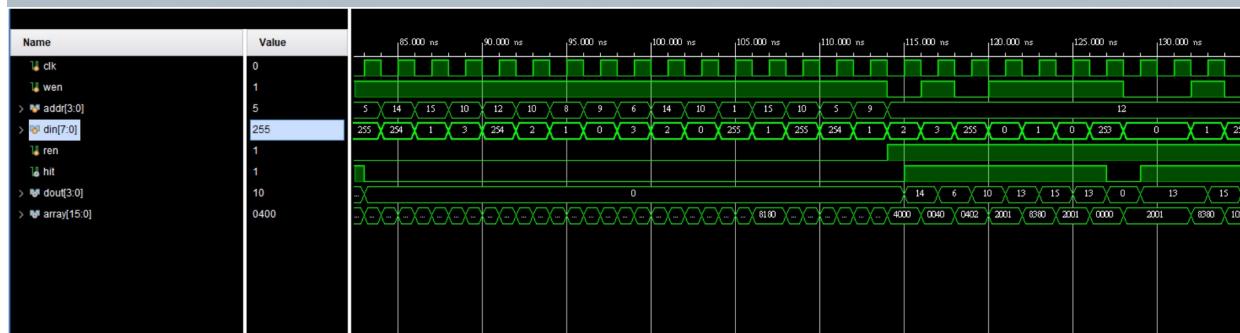
第一階段測試hit與是否會輸出reg最大值，可以看到當ren=1，dout會選擇最大值有放din之最大值addr輸出到dout:



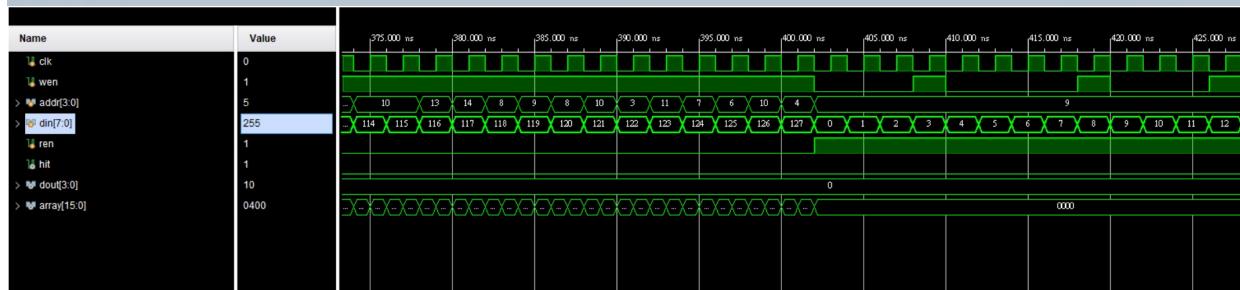
第二階段選取單一數字與小範圍隨機數字隨機放入不同addr，檢查是否會輸出最大值addr，可以看到din=7時，最大值是9沒錯;當din=0,1,225隨機選擇addr放入時，寫出的dout也是有放入那個數字的addr中，最大的addr。



第三階段選取大範圍隨機數字隨機放入不同addr，檢查是否會輸出最大值addr，與第二階段一樣的檢查方式：



第四階段iterate所有din到隨機addr，先寫入再寫出：



測資code：

```
repeat(5)begin
    @(negedge clk);
    din = 7;
    addr = ($random)%16;
end

    din = 1'b0;
    ren = 1'b1;
    wen = 1'b0;

repeat(5)begin
    @(negedge clk);
    wen = ($random)%2;
    din = 7;
end
```

```
repeat(16)begin
    @(negedge clk);
    din = ($random)%4;
    addr = ($random)%16;
end

    din = 1'b0;
    ren = 1'b1;
    wen = 1'b0;

repeat(16)begin
    @(negedge clk);
    wen = ($random)%2;
    din = ($random)%4;
```

```
repeat(8)begin
    @(negedge clk);
    din = ($random)%2;
    addr = ($random)%16;
end

    din = 1'b0;
    ren = 1'b1;
    wen = 1'b0;

repeat(8)begin
    @(negedge clk);
    wen = ($random)%2;
    din = ($random)%2;
```

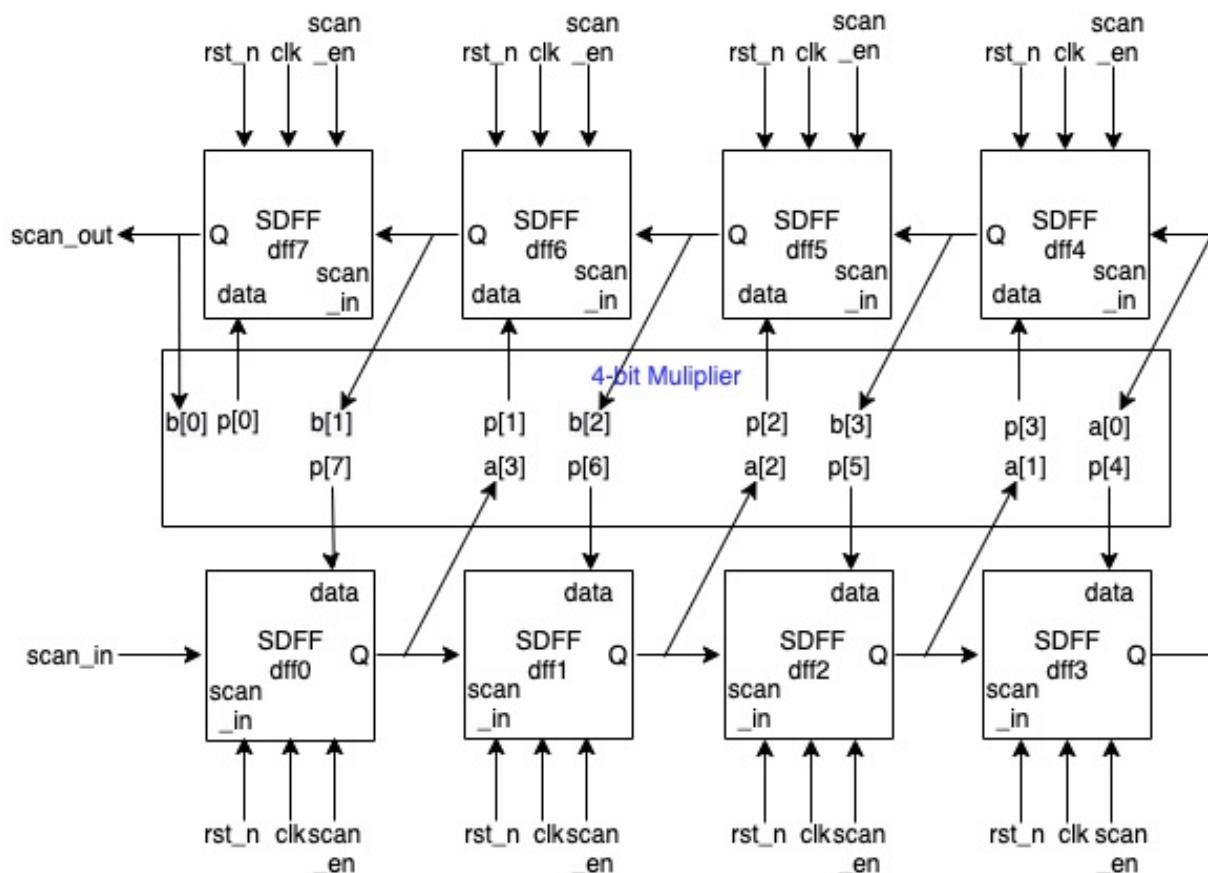
```
repeat(2**7)begin
    @(negedge clk);
    din = din+1'b1;
    addr = ($random)%16;
end

    din = 1'b0;
    ren = 1'b1;
    wen = 1'b0;
repeat(2**7)begin
    @(negedge clk);
    wen = ($random)%2;
    din = din+1'b1;
```

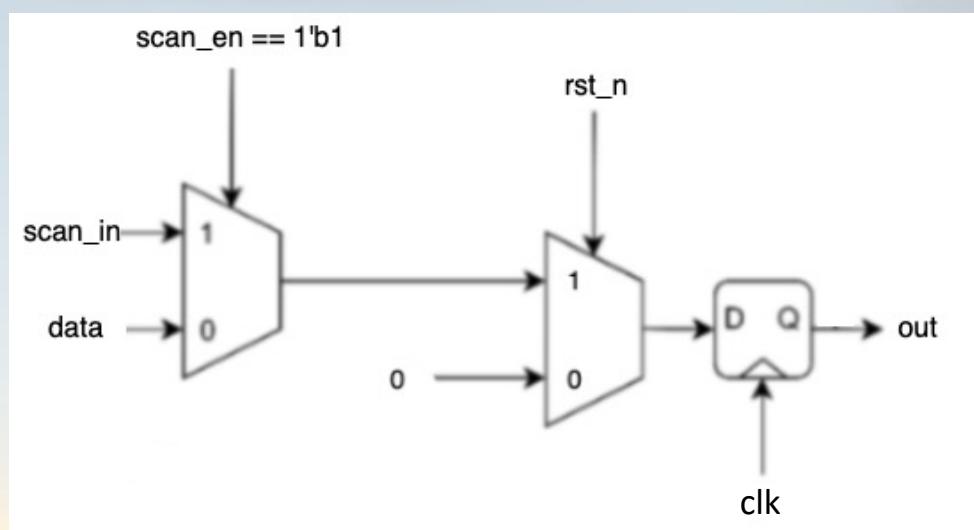
# **Scan chain design**

## I. Diagram

### (1) Circuit Design



### (2) SDFF Circuit Design



## II. Explanation

首先先製作Scan DFF（以下簡稱SDFF），與一般的DFF不同之處是要透過enable signal選擇輸出哪一個input signal。（詳見下方實作程式碼）

```
always@(posedge clk) begin
    if(!rst_n) begin
        out <= 1'b0;
    end
    else begin
        if(scan_en == 1'b1) begin
            out <= scan_in;
        end
        else begin
            out <= data;
        end
    end
end
```

接著依照Spec將SDFF串接起來並把SDFF的output也接入Multiplier中（最後一個SDFF的output同時也是scan\_out）。

而Multiplier的output則會依序接到SDFF的数据input。

```
Multiplier mul(.a(a), .b(b), .out(mul_out));
SDFF dff0( .clk(clk), .rst_n(rst_n), .scan_in(scan_in), .scan_en(scan_en), .data(mul_out[7]), .out(a[3]) );
SDFF dff1( .clk(clk), .rst_n(rst_n), .scan_in(a[3]), .scan_en(scan_en), .data(mul_out[6]), .out(a[2]) );
SDFF dff2( .clk(clk), .rst_n(rst_n), .scan_in(a[2]), .scan_en(scan_en), .data(mul_out[5]), .out(a[1]) );
SDFF dff3( .clk(clk), .rst_n(rst_n), .scan_in(a[1]), .scan_en(scan_en), .data(mul_out[4]), .out(a[0]) );
SDFF dff4( .clk(clk), .rst_n(rst_n), .scan_in(a[0]), .scan_en(scan_en), .data(mul_out[3]), .out(b[3]) );
SDFF dff5( .clk(clk), .rst_n(rst_n), .scan_in(b[3]), .scan_en(scan_en), .data(mul_out[2]), .out(b[2]) );
SDFF dff6( .clk(clk), .rst_n(rst_n), .scan_in(b[2]), .scan_en(scan_en), .data(mul_out[1]), .out(b[1]) );
SDFF dff7( .clk(clk), .rst_n(rst_n), .scan_in(b[1]), .scan_en(scan_en), .data(mul_out[0]), .out(b[0]) );

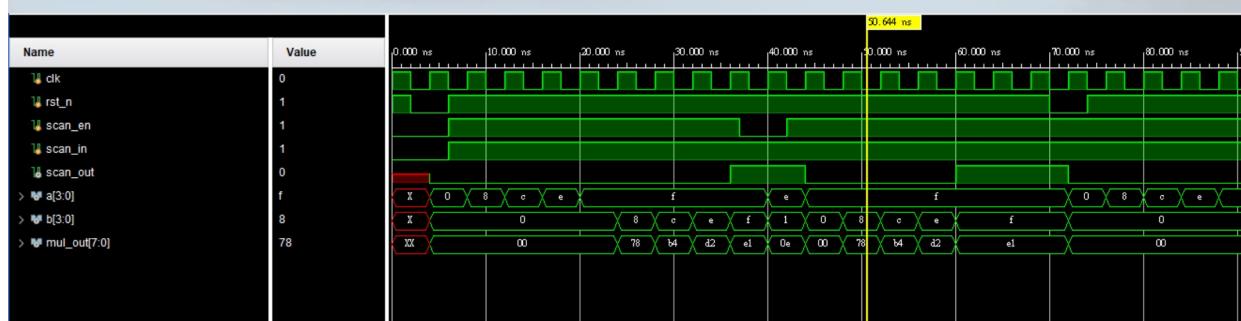
assign scan_out = b[0];
```

### III. Testbench and Waveform

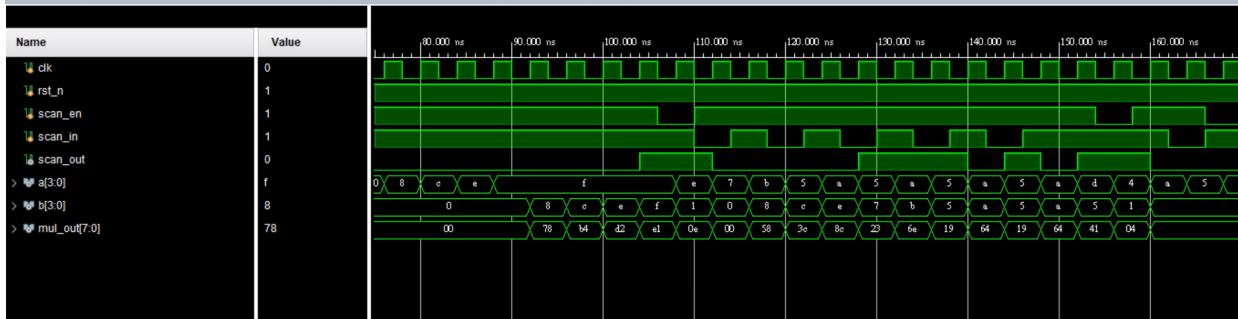
本題特別加入a,b,mul\_out方便測試程式是否有依照規定的behavior來運作，也方便我們偵測錯誤。

testbench共分成兩個階段：

第一階段測試a,b初始值是否正確輸入，經過7個clk、8個posedge之後，scan\_en拉下1個clk後回到1，檢查在這個phase輸出的p0~p7是否等於我們前面在第八個clk的mul\_out，也就是  $a \times b$ ，可以觀察到自從  $\text{scan\_en}=0$  的posedge capture值之後，從第40ns開始的scan\_out值在後八個clk依序為10000111，因為會先輸出p0最後才是p8，所以數字是反過來的a,b乘積  $p = 11100001$ ，而a,b在capture前分別為15與15, 因此  $15 \times 15 = \text{fxf} = 225 = 11100001$ ，代表scan\_out有正確輸出，並在70ns的時候檢查用rst\_n的歸零效果。而在  $\text{scan\_en}=1$  時，sdff之間傳遞的就會是原本的a與b，capture後的data輸出完畢後，輸出的就會是b[0]。



第二階段隨機拉scan\_en值與scan\_in值，測試sdff是否可以正確傳遞capture的data或scan\_in的a,b:



測資code：

```
#2
rst_n = 0;
repeat(8)begin
@ (negedge clk)
rst_n = 1;
scan_in = 1;
scan_en = 1;
end
rst_n = 1;
scan_in = 1;
scan_en = 1;
#3
scan_in = scan_in;
scan_en = 0;
#2

@(negedge clk)
scan_en=1;

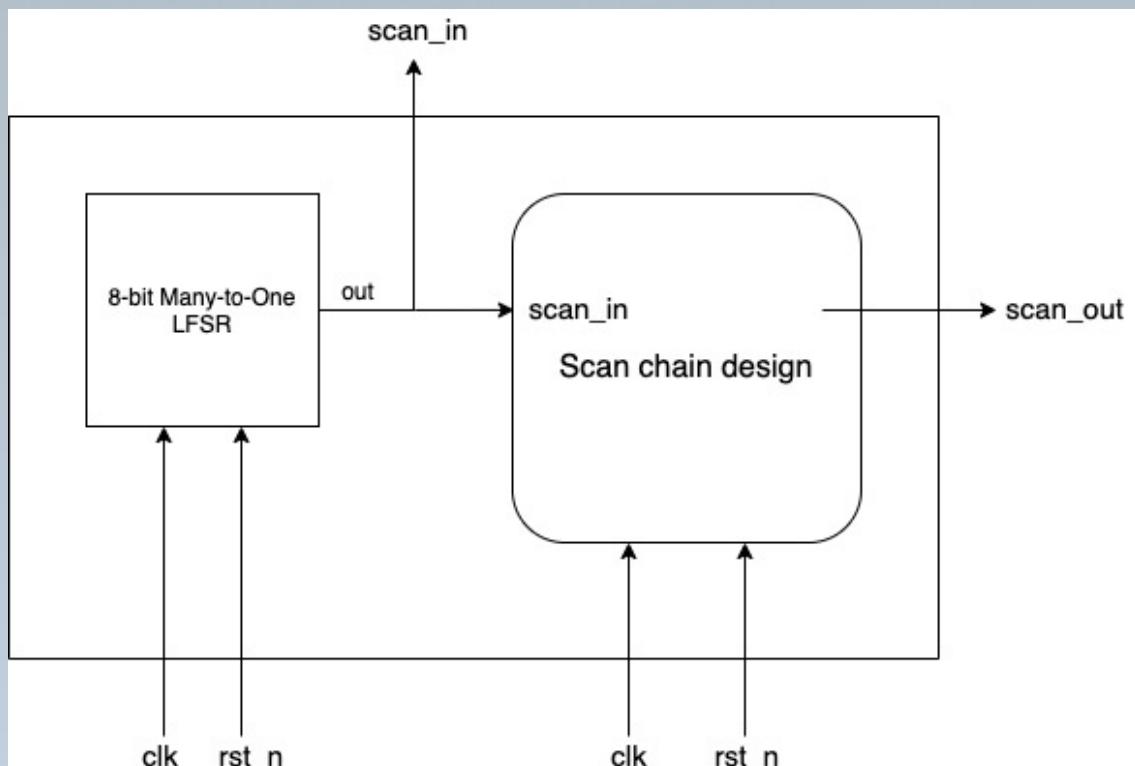
repeat(7)begin
@ (negedge clk)
scan_en = 1;

end
```

# Built-in self test

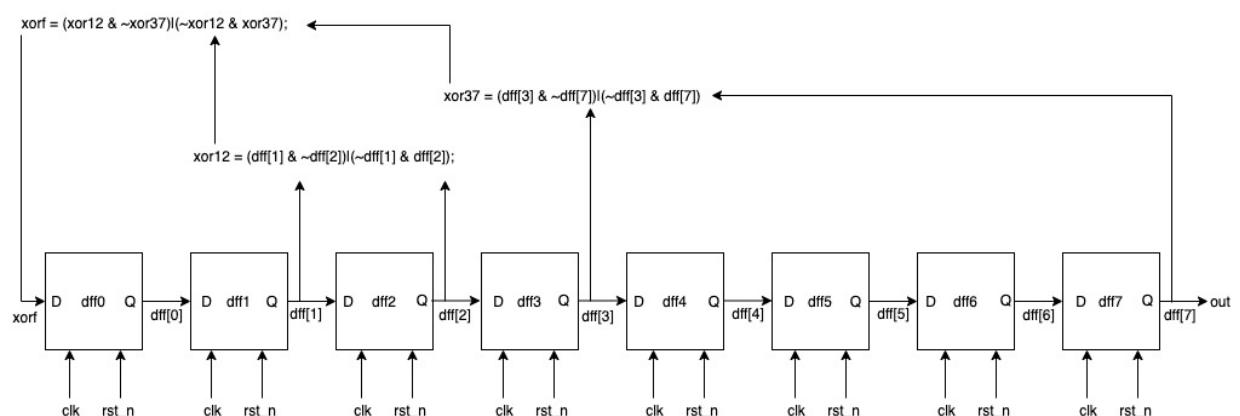
## I. Diagram

### (1) Circuit Design



### (2) LFSR Circuit Design

8-bit Many-to-One LFSR



## II. Explanation

這題用到LFSR與Scan-chain。由於在第二題已經講述Scan-chain design的部分，這邊主要講的是LFSR的做法。

根據spec要求，我們使用Basic Question 3的Many-to-one LFSR，作法是將dff接起來一個一個傳遞，並且以

$(\text{dff}[1] \wedge \text{dff}[2]) \wedge (\text{dff}[3] \wedge \text{dff}[7])$  的方式產生新的 $\text{dff}[0]$ 。

(下圖為LFSR部分實作)

```
else begin
    dff[7:1] <= dff[6:0];
    dff[0] <= xorf;
end
end

always @(*) begin
    xor12 = (dff[1] & ~dff[2])|(~dff[1] & dff[2]);
    xor37 = (dff[3] & ~dff[7])|(~dff[3] & dff[7]);
    xorf = (xor12 & ~xor37)|(~xor12 & xor37);
end
```

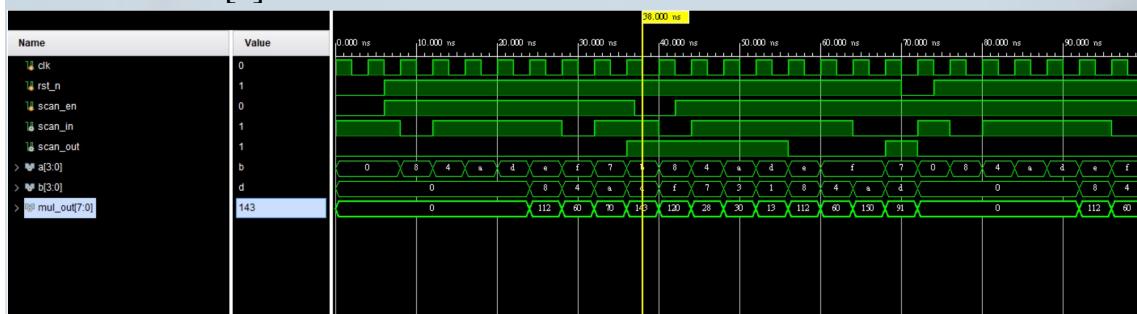
接著將 $\text{dff}[7]$ 的值作為 $\text{scan\_in}$ 輸入給Scan-chain，再透過操控 $\text{scan\_en}$  signal決定Scan-chain要進行的動作。

### III. Testbench and Waveform

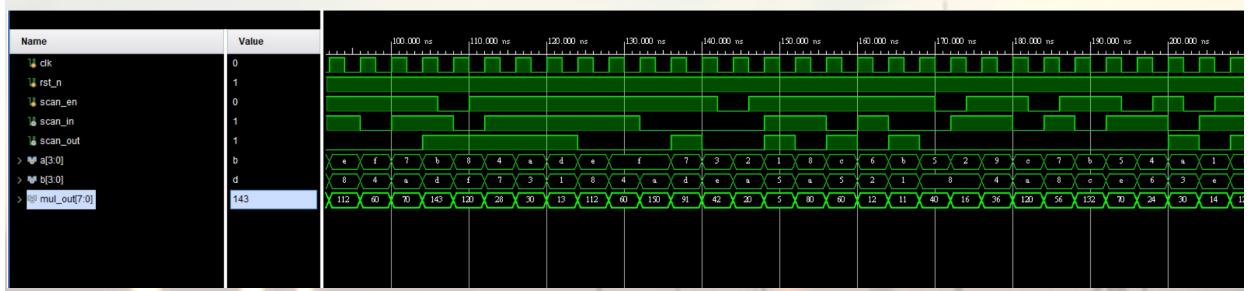
本題也加入a,b,mul\_out方便測試程式是否有依照規定的behavior來運作，也方便我們偵測錯誤。

testbench共分成兩個階段：

第一階段觀察a,b初始值是否為LFSR值之正確輸入，經過7個clk、8個posedge之後，scan\_en拉下1個clk後回到1，檢查在這個phase輸出的p0~p7是否等於我們前面在第八個clk的mul\_out，也就是  $a \times b$ ，可以觀察到自從scan\_en=0的posedge capture值之後，從第40ns開始的scan\_out值在後八個clk依序為 11110001，因為會先輸出p0最後才是p8，所以數字是反過來的a,b乘積  $p = 10001111$ ，而a,b在capture前分別為b與d，也就是LFSR的第八個out, 因此  $143 = b \times d = 11 \times 13 = 8f = 10001111$ ，代表scan\_out有正確輸出，並在70ns的時候檢查用rst\_n的歸零效果。而在scan\_en=1時，sdff之間傳遞的就會是原本的a與b，capture後的data輸出完畢後，輸出的就會是b[0]。



第二階段隨機拉scan\_en值，測試sdff是否可以正確傳遞capture的数据或從LFSR scan\_in正確的值:

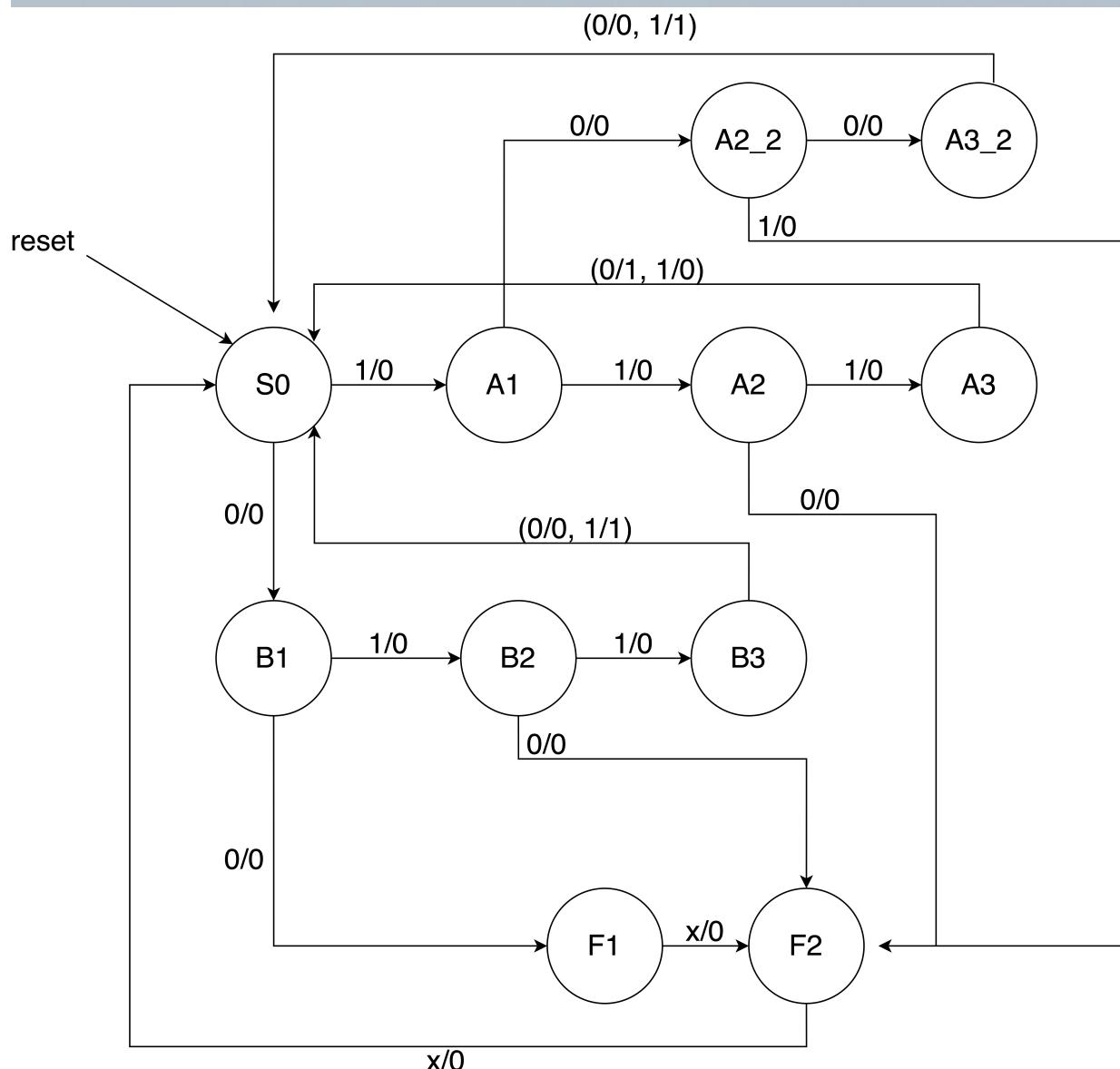


# **Mealy machine sequence detector**

## I. Diagram

### (1) State Transition

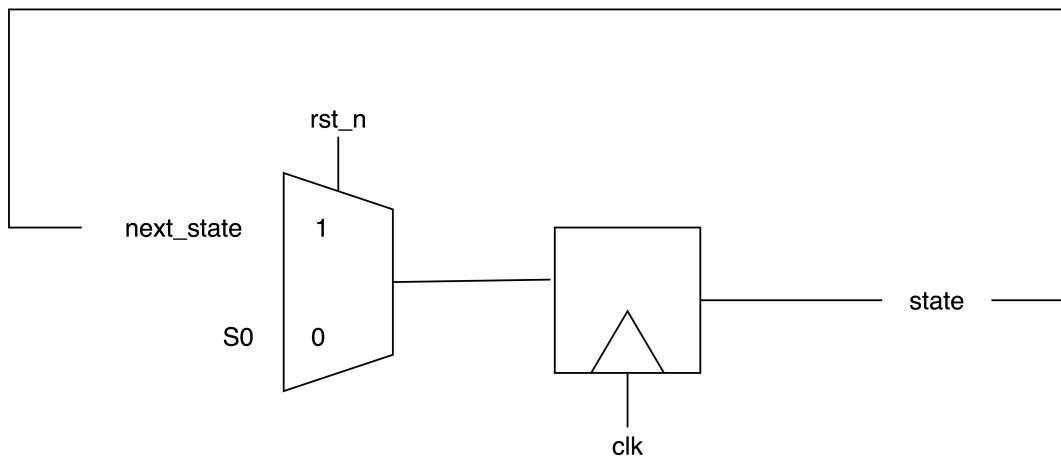
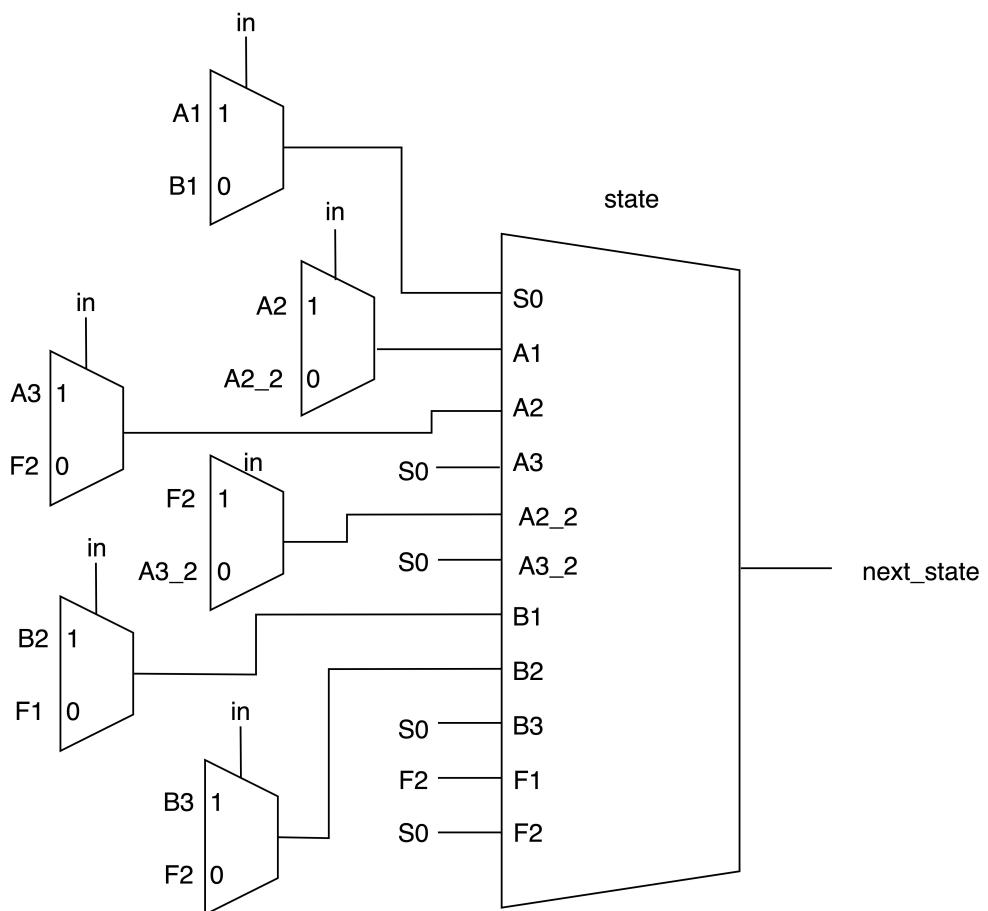
(in/out)



## I. Diagram

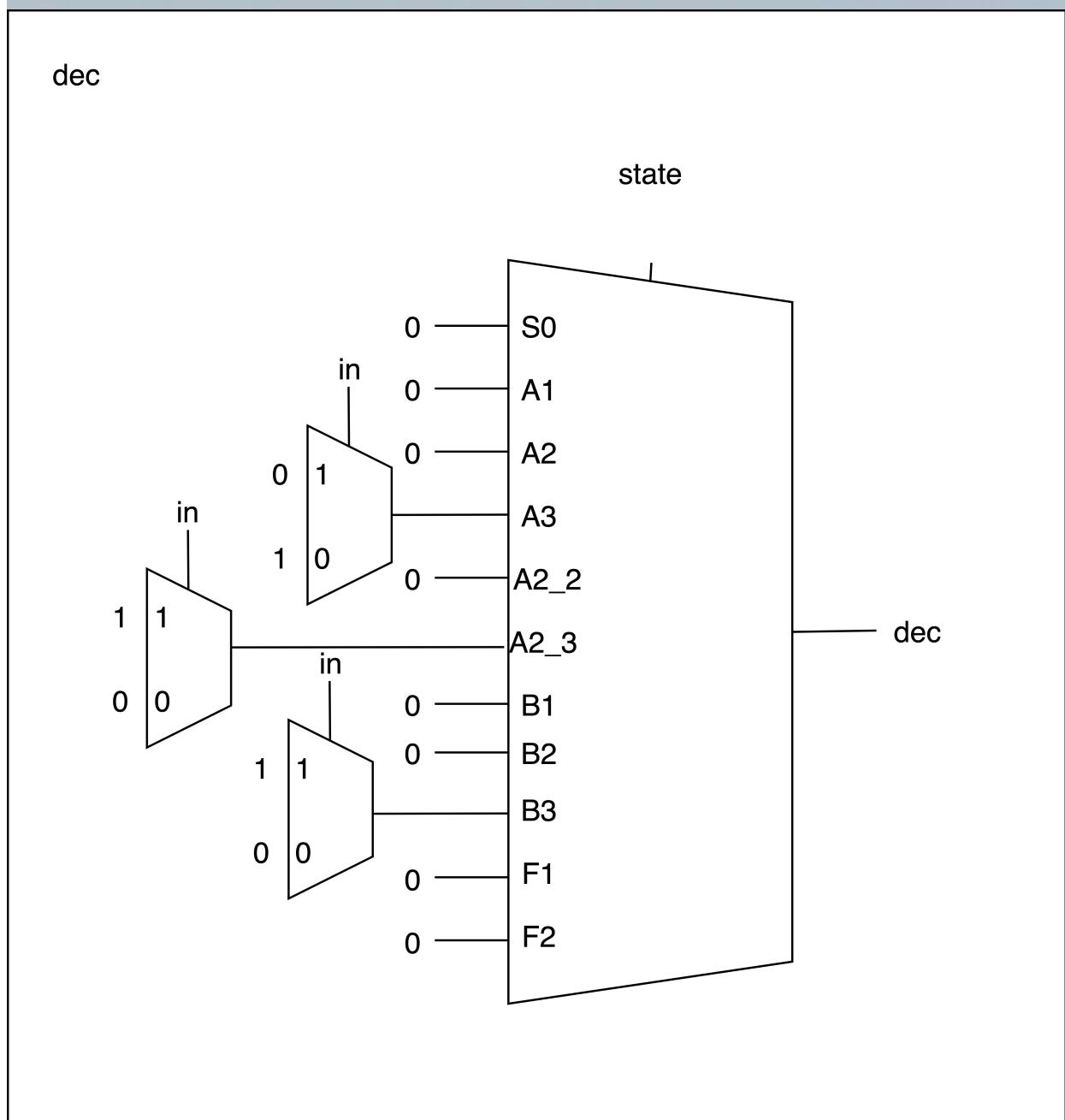
### (1) Circuit Design

State  
Transition



## I. Diagram

### (1) Circuit Design



## II. Explanation

這題是Mealy machine的實作，根據input的變化，我們得出前述的state與in/out組合。在實作的部分，sequential的部分僅僅做state的變化，剩餘包含判斷next\_state, out都是在combinational部分進行。在combinational部分則是根據不同的state與input值，去決定next\_state與應有的output。

(下圖為sequential circuit實作)

```
always@(posedge clk) begin
    if(!rst_n) begin
        state <= S0;
    end
    else begin
        state <= next_state;
    end
end
```

```
S0:begin
    if(in==1'b0)begin
        next_state = B1;
        dec = 1'b0;
    end
    else begin
        next_state = A1;
        dec = 1'b0;
    end
end
A1:begin
    if(in==1'b0)begin
        next_state = A2_2;
        dec = 1'b0;
    end
    else begin
        next_state = A2;
        dec = 1'b0;
    end
end
```

關於state transition的部分於下方進中說明：

在一開始S0時會根據input1/0分為兩大路線，我們稱為A/B。在A路線中，我們期待出現1110/1001讓out=1，因此在state A1也會根據input1/0分出兩條路線，我們稱為A/A\_2。而B路線則是要等到0111出現時才會使out=1。

接著就是state F1及F2，因為每四個bit才會redetect，因此如果中間的路線走到「不可能出現out=1」的情形時，我們會將其放入state FX，這樣無論input是什麼，out皆為0。

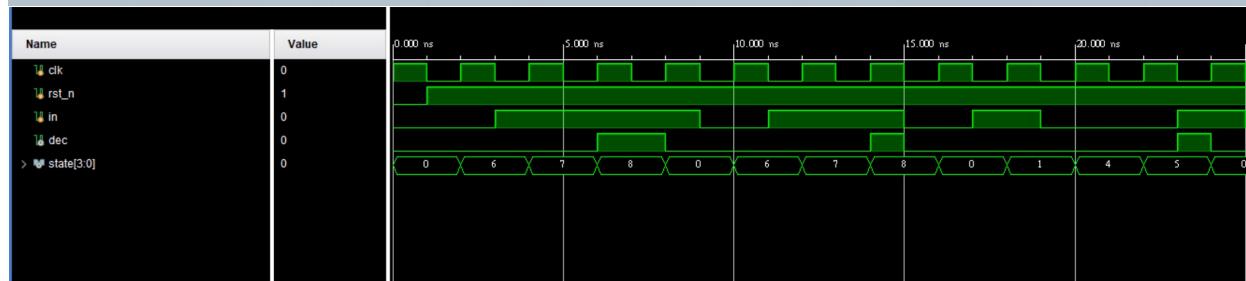
```
F1:begin
    next_state = F2;
    dec = 1'b0;
end
F2:begin
    next_state = S0;
    dec = 1'b0;
end
```

(左圖為state FX的實作)

### III. Testbench and Waveform

本題特別加入state測試程式的正確性與方便偵錯。給每一個state一個數字方便detect現在的state與dec是否是正確的。

觀察根據input，state的改變為S0->B1->B2->B3..以此類推。



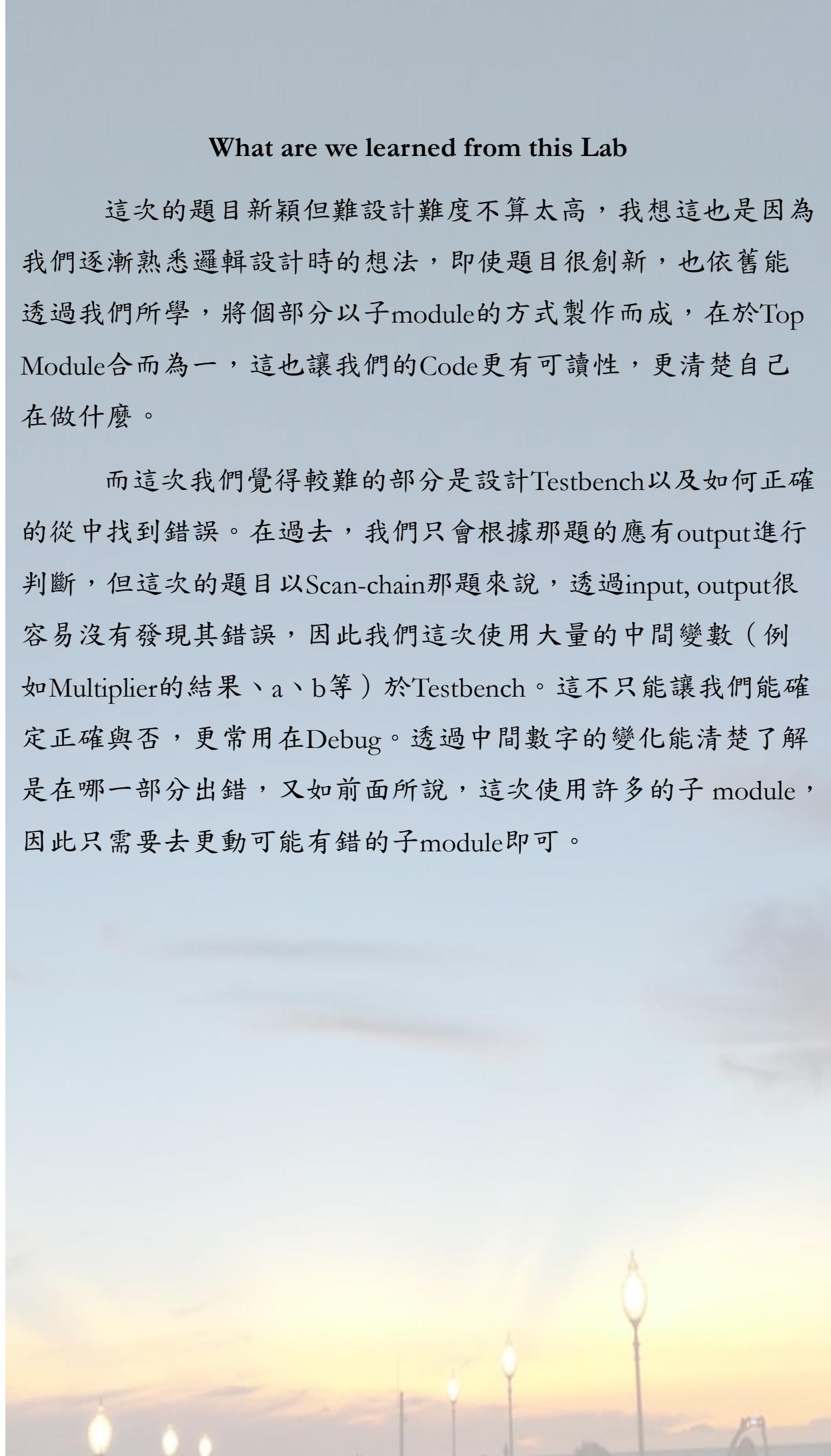
測資code:

```
initial begin
    @ (negedge clk) rst_n = 1'b1;
    @ (posedge clk)
    @ (negedge clk) in = 1'b1;
    @ (posedge clk)
    @ (negedge clk) in = 1'b1;
    @ (posedge clk)
    @ (negedge clk) in = 1'b1;
    @ (posedge clk)
    @ (negedge clk) in = 1'b0;
    @ (posedge clk)
```

## What are we learned from this Lab

這次的題目新穎但難設計難度不算太高，我想這也是因為我們逐漸熟悉邏輯設計時的想法，即使題目很創新，也依舊能透過我們所學，將個部分以子module的方式製作而成，在於Top Module合而為一，這也讓我們的Code更有可讀性，更清楚自己在做什麼。

而這次我們覺得較難的部分是設計Testbench以及如何正確的從中找到錯誤。在過去，我們只會根據那題的應有output進行判斷，但這次的題目以Scan-chain那題來說，透過input, output很容易沒有發現其錯誤，因此我們這次使用大量的中間變數（例如Multiplier的結果、a、b等）於Testbench。這不只能讓我們能確定正確與否，更常用在Debug。透過中間數字的變化能清楚了解是在哪一部分出錯，又如前面所說，這次使用許多的子 module，因此只需要去更動可能有錯的子module即可。



## Cooperation

108062213 顏浩昀：

Advance Question 構想與實作、Mealy machine seq state transition 繪製、Mealy Machine seq 電路圖繪製、Report 製作

106062304 黃鈺舒：

Advance Question Debug、Testbench 設計、4題電路繪製、Basic State Transition 繪製、Report 製作