Lab 3 report

2020/10/22

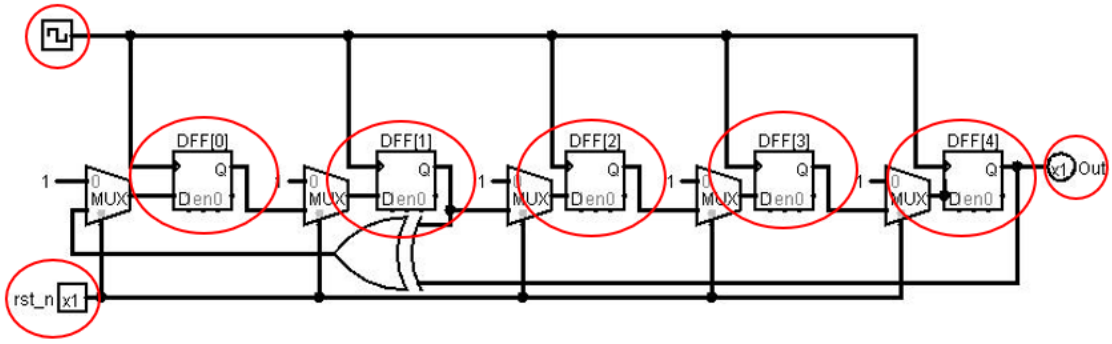
108062125 高敦晉

108062229 陳皇佑

Advanced Question 1

Explanation Verilog Module & Block Diagram

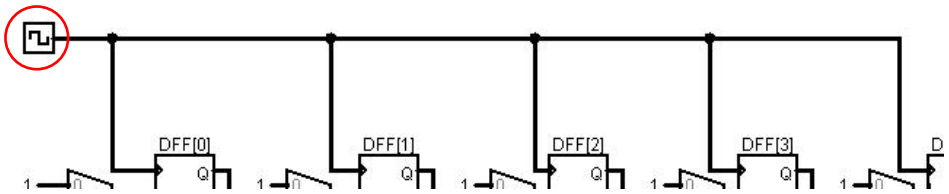
```
3   module LFSR (clk, rst_n, out);
4   input clk, rst_n;
5   output out;
6
7   reg [4:0] DFF;
```



這裡 input/output 基本跟給的模板一樣。

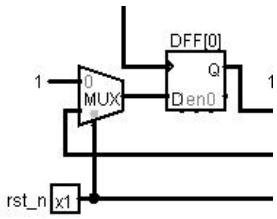
用五個 reg 來處理 D-flipflop 的動作。

```
9  always @(posedge clk)begin
```



Posedge trigger。這東西指的是 Clock 信號。

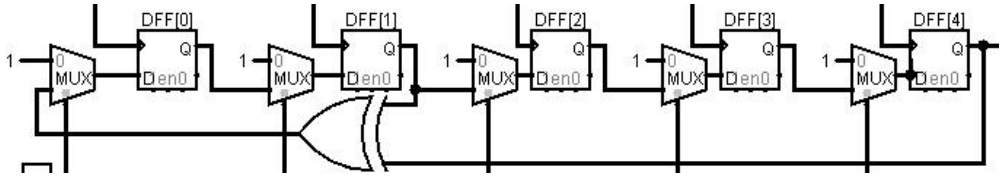
```
10     if(rst_n === 1'b0)
11         DFF[4:0] <= 5'b11111;
```



依照題目的定義，在 rst_n 為 0 時，將所有 D-flipflop 重置為 0

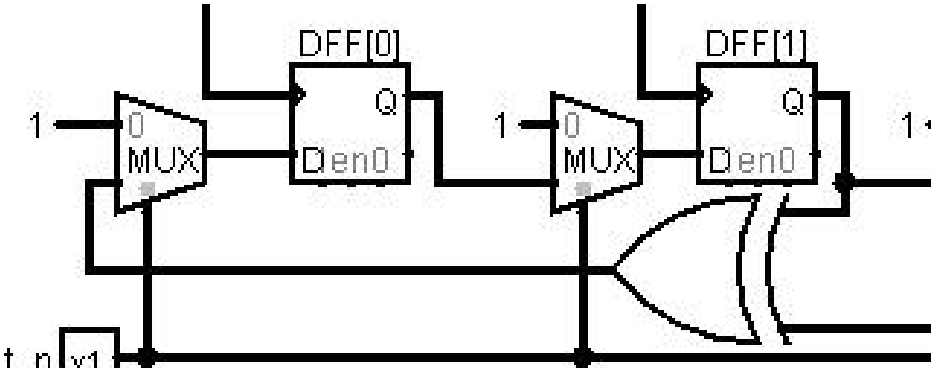
圖以 DFF[0]為例。

```
12    else begin
13        DFF[0] <= DFF[1] ^ DFF[4];
14        DFF[1] <= DFF[0];
15        DFF[2] <= DFF[1];
16        DFF[3] <= DFF[2];
17        DFF[4] <= DFF[3];
18    end
```



每個 D-flipflop 前面都接一個 MUX，以 rst_n 做為 selector，rst_n 為 0 時傳入 1，rst_n 為 1 時傳入上一個 D-flipflop 的值。

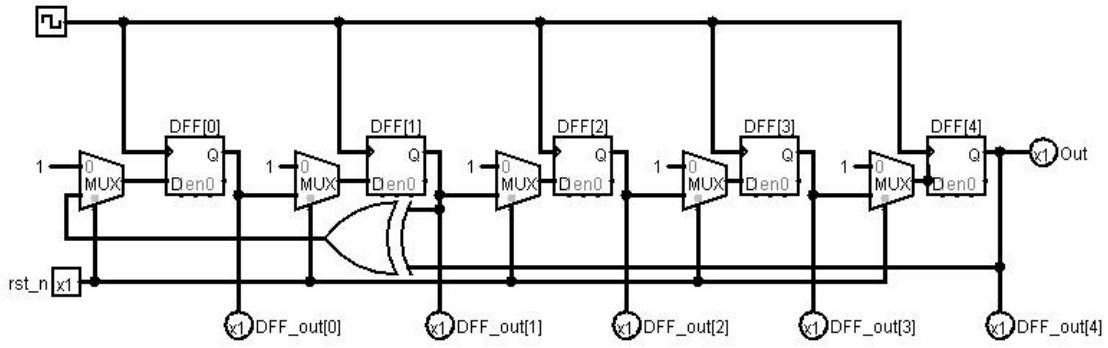
除了 DFF[0] 以外的 D-flipflop 都是將上一個 D-flipflop 的值接下來 MUX 的 1 位置，DFF[0]前的 MUX1 的位置則是用 DFF[1] 以及 DFF[4]的值做 XOR 得到。



在 DFF[0] 前面的 MUX，0 的位置接的是 DFF[4]和 DFF[1]的 XOR 值

為了能夠在 testbench 看裡面的 D-flipflop 信號

我後來又做了一個 LFSR_lookinside 的 module




```
28   output [4:0] DFF_out;
29   reg [4:0] DFF;
```

```
44   assign DFF_out = DFF;
```

整體上跟 LFSR 一模一樣，只差在多拉每個 D-flipflop 的信號出來做 Output。

How to test design

```
3   module Lab3_AQ1_LSFR_tb;
4   // global clock
5   reg CLK = 1'b0;
6   // input
7   reg rst_n = 1'b0;
8   // output
9   wire Out;
10  wire Out_2;
11  wire [4:0] DFFout;
```

宣告基本要用的 reg, wire

CLK -> clock

rst_n -> Reset 信號

Out -> 第一個 module 的 Output

Out_2 -> 第二個 module 的 Output

DFFout -> 第二個 module 的 D-flipflop 信號拉出來

```
LFSR testing_instance(
    .clk(CLK),
    .rst_n(rst_n),
    .out(Out)
);
LFSR_lookinside testing_inside(
    .clk(CLK),
    .rst_n(rst_n),
    .out(Out_2),
    .DFF_out(DFFout)
);
always #2 CLK = ~CLK;
```

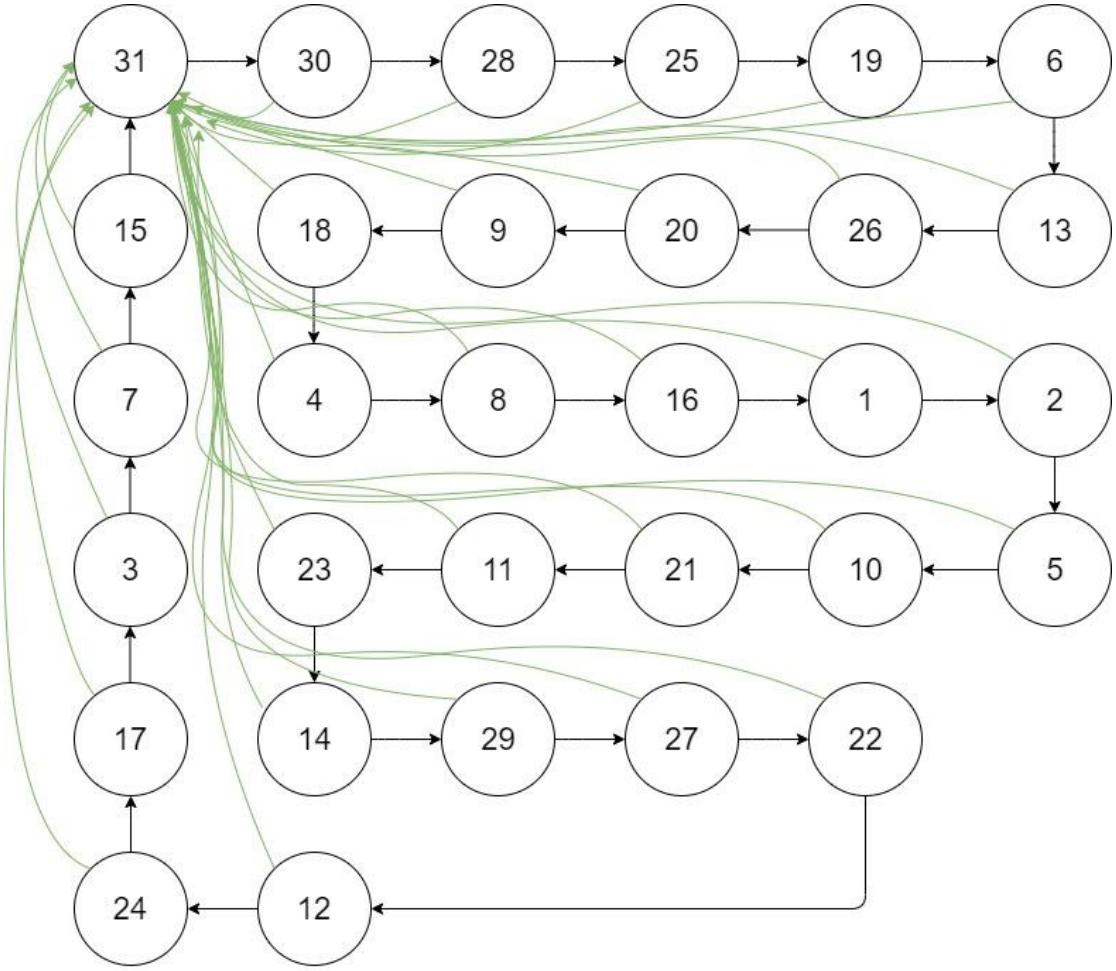
接好 module 以及設置好 CLK 讓他每 #2 跳一次

```
initial begin
    rst_n = 1'b0;

    repeat(2**7)begin
        @(negedge CLK)begin
            rst_n = 1'b1;
        end
    end

    repeat(2**4)begin
        @(negedge CLK)
            rst_n = 1'b0;
    end

    repeat(2**5)begin
        @(negedge CLK)
            rst_n = 1'b1;
    end
    $finish;
end
```

這部分其實就是讓他在 1，然後看一下各個 state Transition 的狀況，然後讓他 reset，在跳一次，確認沒問題就好

State Transition Diagram

下面附上 State Transition Diagram，轉成十進制



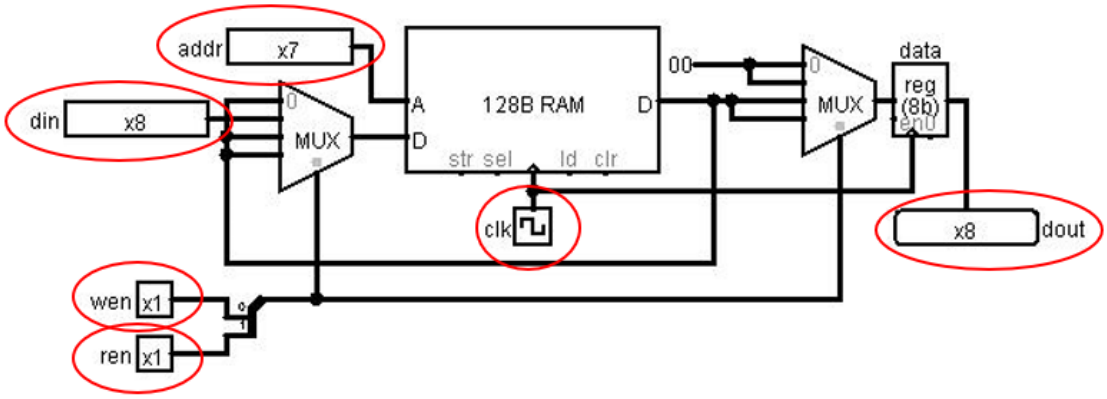
Black Line for RESET == 1'b=1　　Green Line for RESET == 1'b0

此外有測過 RESET = 0 時，將 DFF[4:0]都設成 0 時，它的 state transition 就都會一直在 0。

這個狀況蠻好理解的，因為這個 design 裏頭，能夠帶給他們 D-flipflop 新氣象的只有接在 DFF[0] 上面的 XOR，或是 reset 成 11111，但是現在 reset 變成了 00000，0 XOR 0 又是 0，因此 state 就 hold 在 0 上面了。

Advanced Question 2

Explanation Verilog Module & Block Diagram



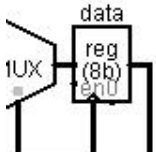

```
3   module Memory (clk, ren, wen, addr, din, dout);
4   input clk;
5   input ren, wen;
6   input [7-1:0] addr;
7   input [8-1:0] din;
8   output [8-1:0] dout;
```

基本的 input/output

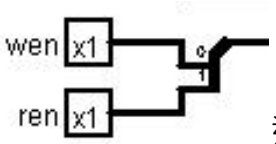
```
10   reg [7:0]Memory_array[128-1:0];
11   reg [7:0] data;
```

Memory_array -> 記憶體們，用了 128 個 8bit 的記憶體

A-> address, 左邊的 D->data input(write), 右邊的 D-> data output(read), 下方中間->clock, str -> write if 1, ld -> read if one(沒接的話就是又讀又寫)，不過我後面用 MUX 解決這件事，後面解釋)

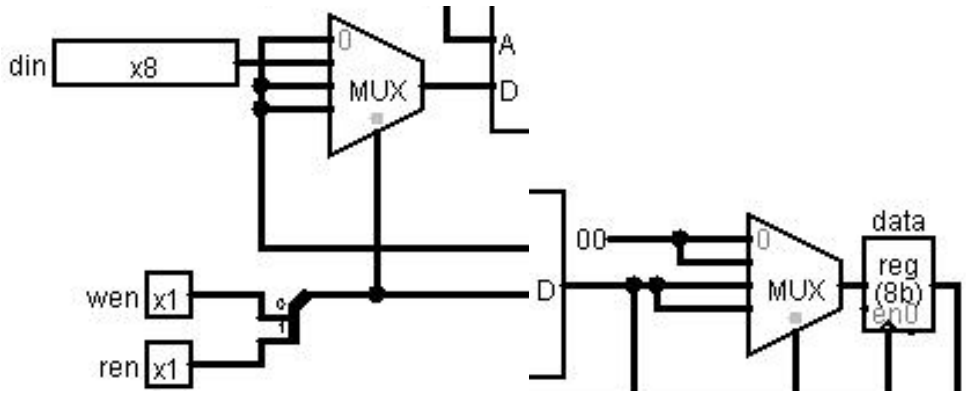


Data -> 用來輸出 dout 用的 8bit reg。



這張是指將兩個 1 bit 的 wire 合成一個 2bit 的 wire 線，沒有實質上的 verilog code，僅是畫圖時用的示意圖而已。

```
case({ren, wen})
2'b00:
begin
    data <= 8'b00000000;
    Memory_array[addr[6:0]] <= Memory_array[addr[6:0]];
end
2'b01:
begin
    data <= 8'b00000000;
    Memory_array[addr[6:0]] <= din;
end
2'b10:
begin
    data <= Memory_array[addr[6:0]];
    Memory_array[addr[6:0]] <= Memory_array[addr[6:0]];
end
2'b11:
begin
    data <= Memory_array[addr[6:0]];
    Memory_array[addr[6:0]] <= Memory_array[addr[6:0]];
end
endcase
```



做個 case，然後判別 ren, wen 的四種組合，分別接再 MUX 上

這裡 Memory_array 是從右邊的 D 接回來的，將自己寫 write 給自己，等同於沒有變。

00 : Do nothing data 設成 8’ b00000000，Memory_array 不變接給自己

01 : Write, data 設成 8’ b00000000，Memory_array 接給 din

10 : Read, data 設成要讀取的 Memory array, Memory_array 接給自己

11 : Read, data 設成要讀取的 Memory array, Memory_array 接給自己

其實我最一開始寫的版本，是

00 : Do nothing data 設成 8’ b00000000

01 : Write, data 設成 8’ b00000000，Memory_array 接給 din

10 : Read, data 設成要讀取的 Memory array

11 : Read, data 設成要讀取的 Memory array

想說該階段沒有用的的 register，就不特地聲明它要不變，後來上 iLMS 問過，且回去翻過講義後，了解到為了避免合成出 unintended latch，要把 if-else 寫好，所以特別寫成了 case，把每個狀況聲明清楚。

How to test design

```
3   module Memory_tb;
4   // clock;
5   reg CLK = 0;
6   // input
7   reg REN;
8   reg WEN;
9   reg [7:0]DIN;
10  reg [6:0]ADR;
11  reg [7:0]curANS;
12  // output
13  wire [7:0]DOUT;
14  // debug using
15  integer phase = 0;
16  reg [7:0]out_mem[127:0];
```

一些需要用到的 reg，input/output 基本上跟 module 差不多。

Out_mem -> 在 testbench，也就是外面模擬另一個 128bytes 的記憶體(128 個 8bit)，用來 check 裡面外面的記憶體是否一樣，有沒有哪裡沒改到。

Integer phase -> debug 用的資訊。

```
17   Memory testing_Memory(
18       .clk(CLK),
19       .ren(REN),
20       .wen(WEN),
21       .addr(ADR),
22       .din(DIN),
23       .dout(DOUT)
24   );
25
26   always #3 CLK = ~CLK;
```

將 module 接好，CLK 設定成#3 一跳。

```
initial begin
    {REN, WEN} = 2'b01;
    ADR = 7'b000000;
    DIN = $random;
    // write something into all memory
    repeat(2**7)begin
        @(posedge CLK)begin
            out_mem[ADR[6:0]] = DIN;
        end
        @(negedge CLK)begin
            DIN = $random;
            ADR = ADR + 1'b1;
        end
    end
```

先將 128 個 address 全部寫入 random 的數字，一開始將{REN, WEN}設成 01 代表寫入狀態。接著 2**7 = 128 寫入，一邊維護外部的 memory-array out_mem。

```
42      {REN, WEN} = 2'b00;
43      #10
44      {REN, WEN} = 2'b10;
```

從 write 過渡到 read 中間，先設為 00 表示 do nothing，等個 #10，設成 10 表示開始 read。

```
// random read something out;
repeat(2**5)begin
    // I used to think can't task
    @(posedge CLK)begin
        curANS = out_mem[ADR];
    end
    @(negedge CLK)begin
        ADR = $random;
        DIN = 8'b00000000;
        testreadout;
    end
end
```

Random 讀出一些 data，並用 task testreadout check 讀出來的 data 是否和外面維護的 memory 讀到的 data 一樣，negedge 時做更改 address 以及 task testreadout。

```
 96  task testreadout;
 97  begin
 98      if(DOUT != curANS)begin
 99          $display("ERROR!");
100          $write("Phase: %d. Address at: %h ", phase, ADR);
101          $write("DOUT: %h but it should be %h\n", DOUT, curANS);
102      end
103  end
104  endtask
```

Task testreadout 內容如上

就簡單的 check 現在讀出來的跟 curANS 有沒有一樣，if not 輸出現在在哪個 phase debug，讀的位置是哪裡，輸出是多少，正確輸出是多少。提供 debug 資訊。

```
58          phase = phase + 1;
```

換到下一個 phase。

```
// write and read
{REN, WEN} = 2'b00;
ADR = $random;
```

```
88          @(negedge CLK)begin
89              {REN, WEN} = {REN, WEN} + 2'b01;
90              ADR = $random;
91              DIN = $random;
92          end
```

第二個 phase 要測的是讓 REN, WEN 一直+1，每次 address 跟 data-in 都 random，看看 memory module 在各種情況下的 behavior 有沒有符合預期。

```
63          @(posedge CLK)begin
64              case({REN, WEN})
65              2'b00:
66              begin // nothing
67                  curANS <= 8'b00000000;
68                  out_mem[ADR] <= out_mem[ADR];
69              end
70              2'b01:
71              begin // write
72                  curANS <= 8'b00000000;
73                  out_mem[ADR] <= DIN;
74              end
75              2'b10:
76              begin // read
77                  curANS <= out_mem[ADR];
78                  out_mem[ADR] <= out_mem[ADR];
79              end
80              2'b11:
81              begin // read
82                  curANS <= out_mem[ADR];
83                  out_mem[ADR] <= out_mem[ADR];
84              end
85              endcase
86              testreadout;
87          end
```

這邊基本上就根據現在的{REN, WEN}進行操作，有個問題點是這裡的實作幾乎快跟 Memory module 裏頭一模一樣，問題是我也沒想到比較好的方式來測試，單純就用 Wave 圖以及 task 判斷正確性。

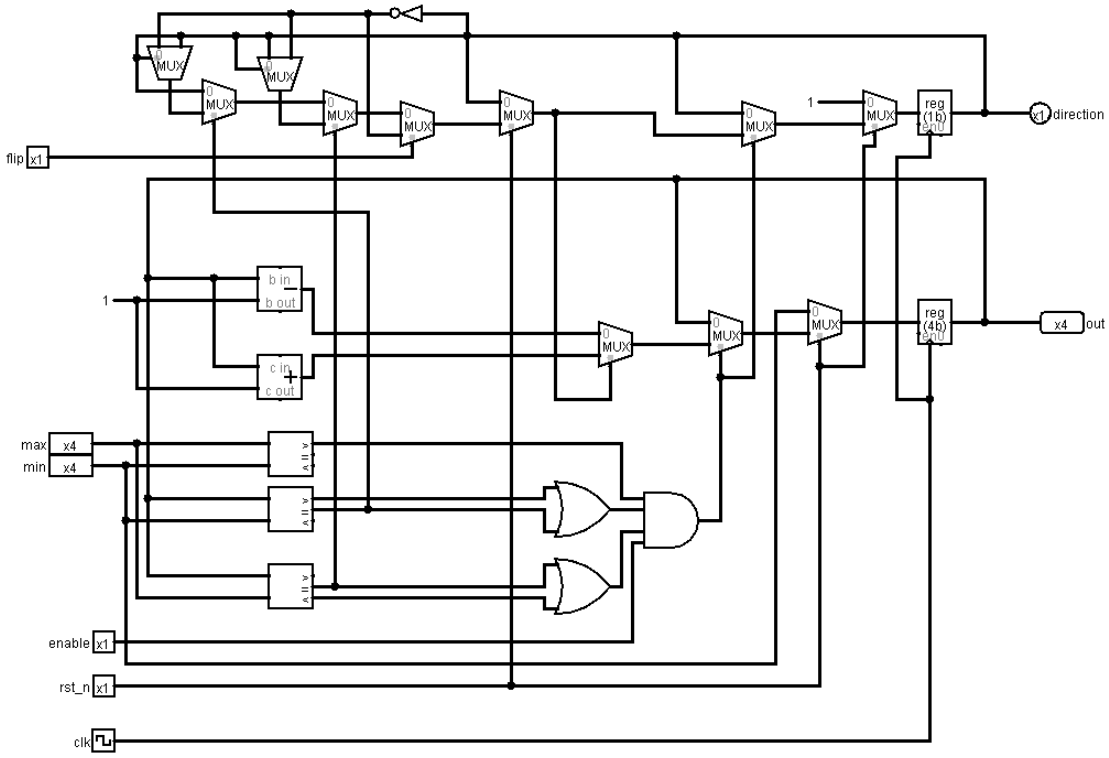
What I have learned:

這次主要學到的是 if-else 以及 case 的正確寫法，寫 if-else 時每多接一個 else if，在合成時就會多新增一個 2 to 1 MUX，寫成 case 時會使用一個多工器將所有 case 放入 input 然後做 selector。如果能夠寫成 case 就盡量寫成 case 以減少 gate 數量。

另外在 if 沒有寫出 else 或是 case 沒有把所有狀況定義出來時，會生成 unintended latch，要避免這種情況。

另外就是熟悉了寫在 Behavior Level 寫出的 sequential circuit 實際上 register 的運作模式。在畫 block diagram 時，才一邊認真了解自己寫出的 verilog 會合成怎樣的 circuit，還因此重寫了一兩次 verilog 的設計。

AQ3



設計構想：

設計 sequential circuit 內含 counter 和 direction。

```
if(rst_n == 1'b0) begin
    cnt <= min;
    direction <= 1'b1;
end
```

Reset = 0 的時候，把 reset 和 direction 設成初始值，

```
if(enable == 1'b1 && (max > min) && (cnt >= min) && (cnt <= max)) begin
    if(nxt_direction == 1'b1)begin
        cnt <= cnt + 1'b1;
    end
    else begin
        cnt <= cnt - 1'b1;
    end
    direction <= nxt_direction;
end
else begin
    cnt <= cnt;
    direction <= direction;
end
```

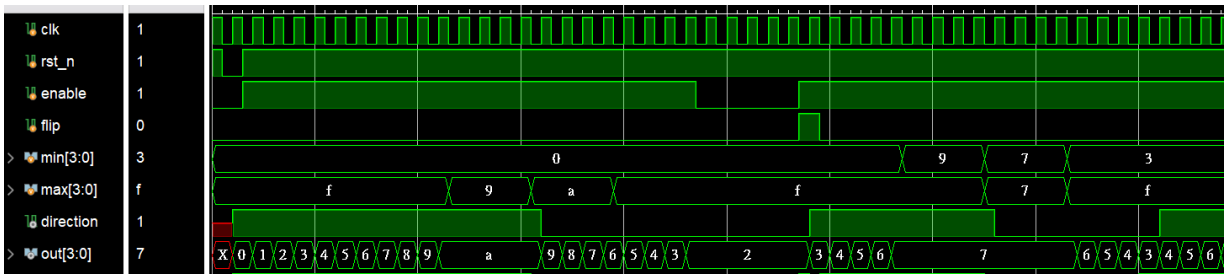
在 reset = 1 的時候，先判斷 max > min、counter ≤ max、counter ≥ min 是否成立。如果成立，以下一個方向決定 counter 要加一或減一，然後使 direction = 下一個 direction。如果不成立，counter 和 direction 的值維持不變。

```
always @ (cnt, max, min, flip, rst_n, direction) begin
    if(rst_n == 1'b1) begin
        if(flip == 1'b1) nxt_direction = ~direction;
        else begin
            if(cnt == max) begin
                if(direction == 1'b1) nxt_direction = ~direction;
                else nxt_direction = direction;
            end
            else if(cnt == min) begin
                if(direction == 1'b0) nxt_direction = ~direction;
                else nxt_direction = direction;
            end
            else nxt_direction = direction;
        end
    end
    else nxt_direction = direction;
end
```

把 nxt_direction 設計成 combinational circuit，當 reset 為 0，nxt_direction = direction。Reset = 1 時，如果 flip = 1，nxt_direction = direction 的相反。Flip = 0：若 counter = max 且 direction = 1，nxt_direction = direction 的相反，否則等於

direction。若 counter = min 且 direction = 0，nxt_direction = direction 的相反，否則等於 direction。若都不是 nxt_direction = direction。

Testbench 設計構想：



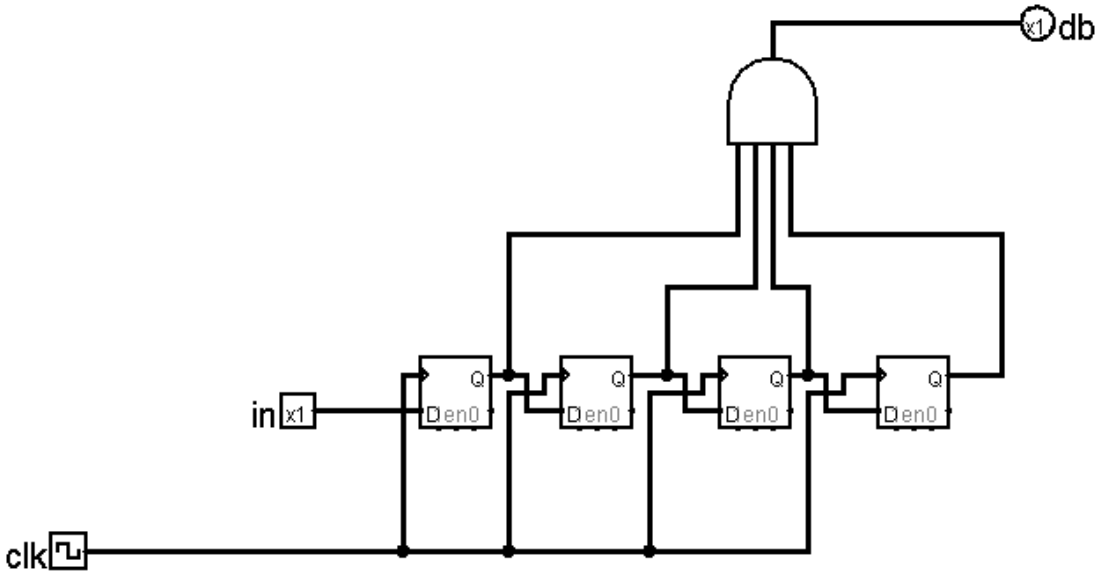
Max 改為 9 測若 cnt > max，output 和 direction 會不會維持定值。Max 改 10 測 cnt 和 direction 遇到上極限會不會正常運作。Enable 改 0 測 output 和 direction 會不會維持定值。Flip 改 1 測 direction 會不會變反向。Min 改 9 測若 cnt< min，output 和 direction 會不會維持定值。Min 和 max 改 7 測，若 output = min = max，output 和 direction 會不會維持定值。Min 改 3 測 cnt 和 direction 遇到下極限會不會正常運作。

開發過程中的問題/學習：

一開始沒有解決 direction 在 reset 的時候為 1 的問題，下一個 posedge clk 時，direction 會反轉。最後拆成 combinational 和 sequential 兩種 block 寫，變得容易預測結果，也比較容易修改邏輯，只要在 next direction 的變動設條件，就可以解決了。

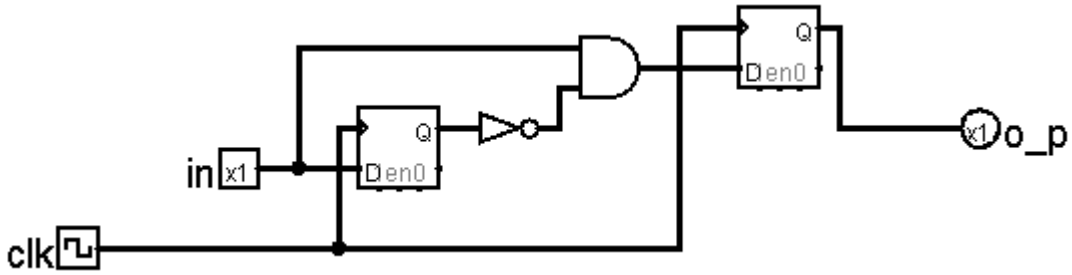
FPGA

設計構想：




```
module debounce(db, in, clk);
input in, clk;
output db;

reg [3:0] dff;

always @ (posedge clk) begin
    dff[3:1] <= dff[2:0];
    dff[0] <= in;
end

assign db = ( (dff == 4'b1111) ? 1'b1 : 1'b0);

endmodule
```

Debounce 是讓 input 經過 4 個 dff，確保要四個 clk cycle 連續為 1。




```
module one_pulse(o_p, in, clk);
input in, clk;
output o_p;

reg o_p, in_delay;

always @ (posedge clk) begin
    o_p <= in &( !in_delay);
    in_delay <= in;
end

endmodule
```

One_pulse 是 in 跟 in_delay 的相反做 and，結果用 dff 輸出，這樣

確保訊號 1 只會維持一個 clk cycle。

```
always @ (posedge clk)
    begin
        cnt <= nxt_cnt;
        if (cnt === 25'b1_1111_1111_1111_1111_1111_1111) clk1_2_25 <= 1'b1;
        else clk1_2_25 <= 1'b0;
    end

always @ (*) begin
    if(cnt === 25'b1_1111_1111_1111_1111_1111_1111) nxt_cnt = 25'd0;
    else nxt_cnt = cnt + 1'b1;
end
```

```
always @ (posedge clk)
    begin
        cnt <= nxt_cnt;
        if (cnt === 16'b1111_1111_1111_1111) clk1_2_16 <= 1'b1;
        else clk1_2_16 <= 1'b0;
    end

always @ (*) begin
    if(cnt === 16'b1111_1111_1111_1111) nxt_cnt = 16'd0;
    else nxt_cnt = cnt + 1'b1;
end
```

Clock divider，用一個 dff 做計數器，225 是數到 2 的 25 次方，216 是數到 2 的 16 次方，之後 counter 歸 0，output 轉 1。功用是延長 clk cycle 長度。

```
Clock_Divider_225 cd225(clk, clk_225);
Clock_Divider_216 cd216(clk, clk_216);

debounce db_f(flip_db, flip, clk);
debounce db_r(rst_n_db, rst_n, clk);
one_pulse op_f(flip_db_op, flip_db, clk_225);
one_pulse op_r(rst_n_db_op, rst_n_db, clk_225);
```

對 flip、rst_n 兩個按鈕做 debounce 和 one_pulse，debounce 傳入原始 clk，這樣只要一按 db_output 會馬上跳一，one_pulse 傳入調慢 2 的 25 次方頻率的 clk，為了讓 one_pulse 長度達到 counter 的 clk 長度。

```
not(not_rst_n_dp_op, rst_n_db_op);
```

因為 reset 是等於 0 初始化，所以把處理完的 reset 取反。

```
Parameterized_Ping_Pong_Counter pppc(clk_225, not_rst_n_dp_op, enable, flip_db_op, max, min, dir, num);
```

把參數放進 AQ3 的 counter 內。

```
always @ (posedge clk_216) begin
    if(not_rst_n_dp_op == 1'b0) begin
        an <= 4'b1110;
        case (an)
        4'b1110:
        begin
            an <= 4'b0111;
            case (num)
            4'b0000: out <= 7'b000_0001;
            4'b0001: out <= 7'b000_0001;
            4'b0010: out <= 7'b000_0001;
            4'b0011: out <= 7'b000_0001;
            4'b0100: out <= 7'b000_0001;
            4'b0101: out <= 7'b000_0001;
            4'b0110: out <= 7'b000_0001;
            4'b0111: out <= 7'b000_0001;
            4'b1000: out <= 7'b000_0001;
            4'b1001: out <= 7'b000_0001;
            4'b1010: out <= 7'b100_1111;
            4'b1011: out <= 7'b100_1111;
            4'b1100: out <= 7'b100_1111;
            4'b1101: out <= 7'b100_1111;
            4'b1110: out <= 7'b100_1111;
            4'b1100: out <= 7'b100_1111;
            4'b1101: out <= 7'b100_1111;
            4'b1110: out <= 7'b100_1111;
            4'b1111: out <= 7'b100_1111;
            default : out <= 7'b101_1111;
            endcase
        end
        4'b0111:
        begin
            an <= 4'b1011;
            case (num)
            4'b0000: out <= 7'b000_0001;
            4'b1010: out <= 7'b000_0001;
            4'b0001: out <= 7'b100_1111;
            4'b1011: out <= 7'b100_1111;
            4'b0010: out <= 7'b001_0010;
            4'b1100: out <= 7'b001_0010;
            4'b0011: out <= 7'b000_0110;
            4'b1101: out <= 7'b000_0110;
            4'b0100: out <= 7'b100_1100;
            4'b1110: out <= 7'b100_1100;
            4'b0101: out <= 7'b010_0100;
            4'b1111: out <= 7'b010_0100;
            4'b0110: out <= 7'b010_0000;
            4'b0111: out <= 7'b000_1111;
            4'b1000: out <= 7'b000_0000;
            4'b1001: out <= 7'b000_0100;
            default : out <= 7'b011_1111;
            endcase
        end
        4'b1011:
        begin
            an <= 4'b1101;
            out <= 7'b001_1101;
        end
        4'b1101:
        begin
            an <= 4'b1110;
            out <= 7'b001_1101;
        end
        default : out <= 7'b111_0111;
        endcase
```

用 2 的 16 次方 clk，讓肉眼觀察不到各個 digit 跳動。

若 reset，an 從 3 跑到 0，輸出各個 digit 的 num and direction output。

```
4'b0111:
begin
    an <= 4'b1011;
    case (num)
    4'b0000: out <= 7'b000_0001;
    4'b1010: out <= 7'b000_0001;
    4'b0001: out <= 7'b100_1111;
    4'b1011: out <= 7'b100_1111;
    4'b0010: out <= 7'b001_0010;
    4'b1100: out <= 7'b001_0010;
    4'b0011: out <= 7'b000_0110;
    4'b1101: out <= 7'b000_0110;
    4'b0100: out <= 7'b100_1100;
    4'b1110: out <= 7'b100_1100;
    4'b0101: out <= 7'b010_0100;
    4'b1111: out <= 7'b010_0100;
    4'b0110: out <= 7'b010_0000;
    4'b0111: out <= 7'b000_1111;
    4'b1000: out <= 7'b000_0000;
    4'b1001: out <= 7'b000_0100;
    default : out <= 7'b011_1111;
    endcase
```

```
else begin
    case (an)
    4'b1110:
    begin
        an <= 4'b0111;
        case (num)
        4'b0000: out <= 7'b000_0001;
        4'b0001: out <= 7'b000_0001;
        4'b0010: out <= 7'b000_0001;
        4'b0011: out <= 7'b000_0001;
        4'b0100: out <= 7'b000_0001;
        4'b0101: out <= 7'b000_0001;
        4'b0110: out <= 7'b000_0001;
        4'b0111: out <= 7'b000_0001;
        4'b1000: out <= 7'b000_0001;
        4'b1001: out <= 7'b000_0001;
        4'b1010: out <= 7'b100_1111;
        4'b1011: out <= 7'b100_1111;
        4'b1100: out <= 7'b100_1111;
        4'b1101: out <= 7'b100_1111;
        4'b1110: out <= 7'b100_1111;
        4'b1111: out <= 7'b100_1111;
        default : out <= 7'b101_1111;
```

```
4'b1011:
begin
    an <= 4'b1101;
    if(dir == 1'b1) out <= 7'b001_1101;
    else out <= 7'b110_0011;
end
4'b1101:
begin
    an <= 4'b1110;
    if(dir == 1'b1) out <= 7'b001_1101;
    else out <= 7'b110_0011;
end
default : an <= 4'b1110;
endcase
```

Reset 不等於 0，an 從 3 到 0，依 counter 輸出的 out 和 direction 決定各 digit 的 output。

開發過程中的問題/學習：

因為想直觀的做 seven segment 輸出，就用很冗的 case 寫，在燒進板子時，因為 reset 是 0 的時候觸發，抓了很久的 bug 才發現。Debounce 和 one_pulse 一開始都用最慢的 clk 當參數，導致要按很久，才有反應。最後應該兩個都傳入原始 clk，使得一按下去就觸發，再把 output 訊號做延長，但一直想不到怎麼正確的延長訊號。最後才知道用一個新的 counter 算固定時間，以那個 counter 做延長。我們有用 led 燈確認 counter 有無正確運行及按鈕的訊號值，讓 debug 輕鬆不少。

Contribution:

高敦晉：AdvancedQuestion1, AdvancedQuestion2

陳皇佑：AQ3, FPGA