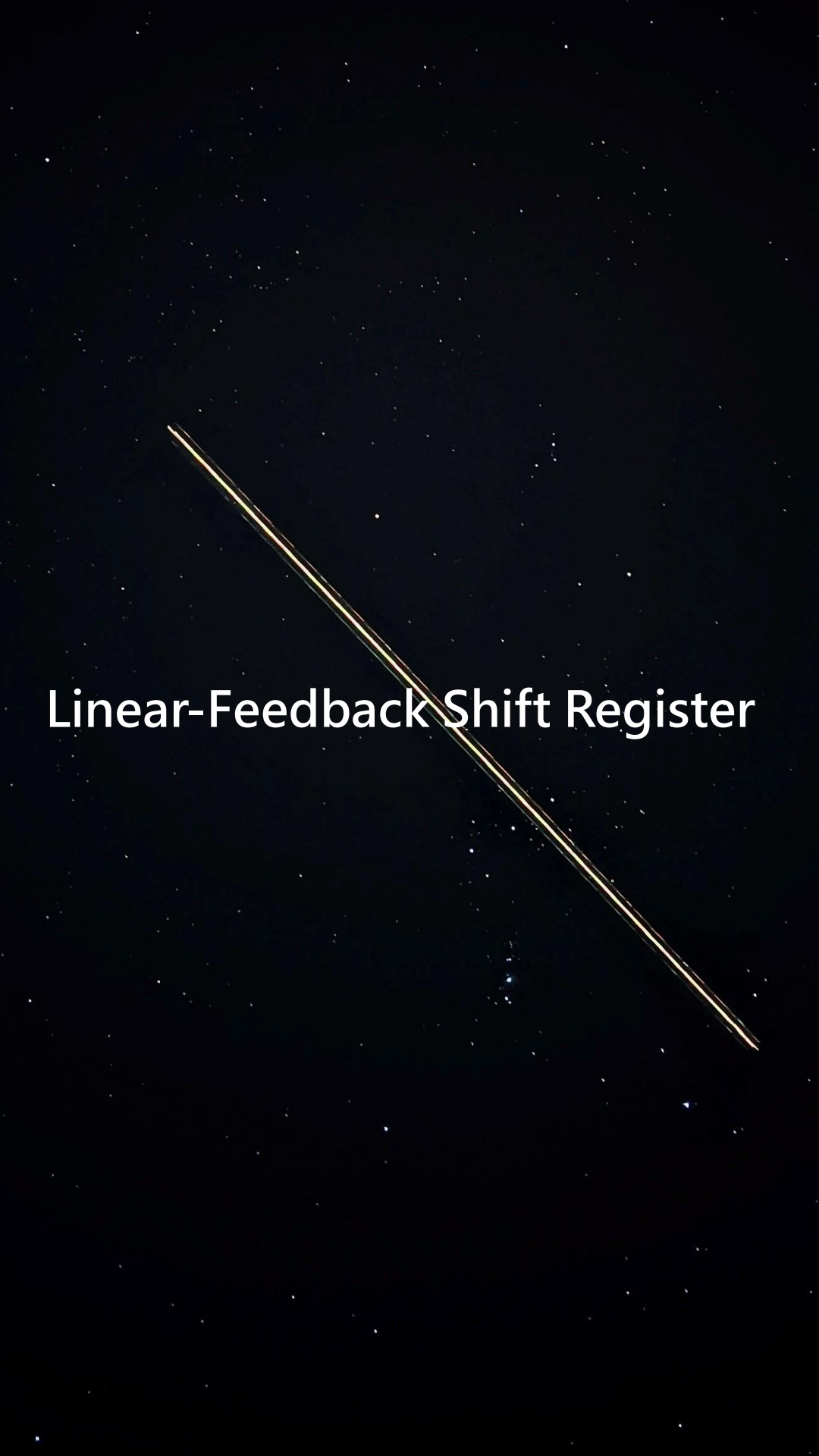
Logic design lab

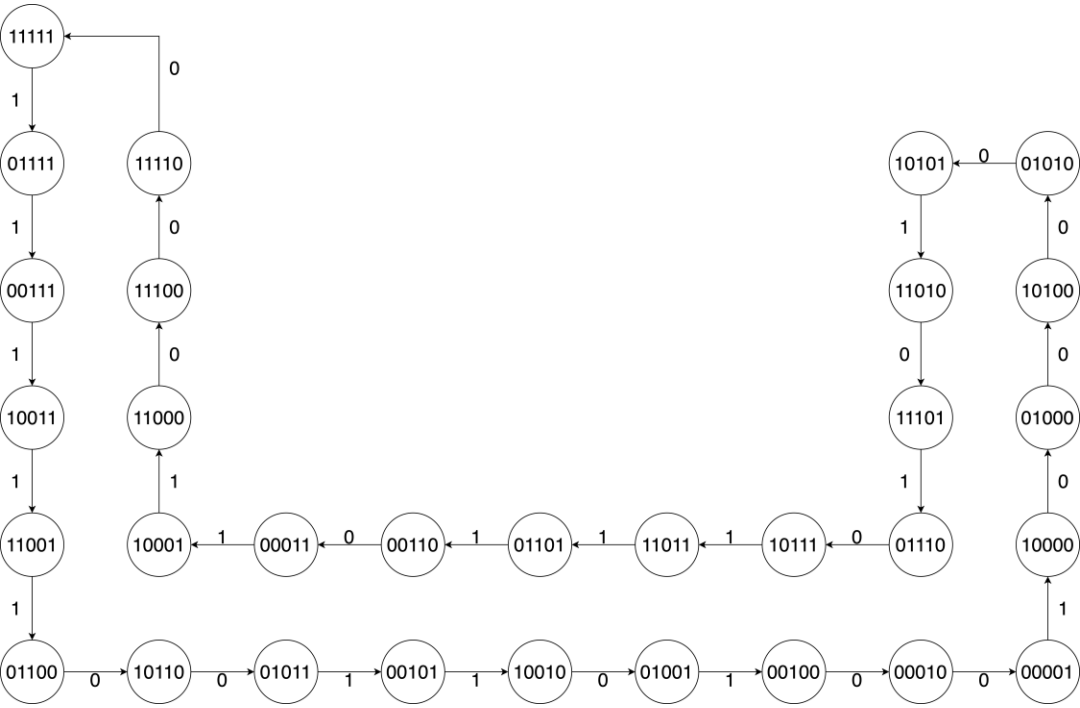
Lab3_Team5_report

2020/10/22

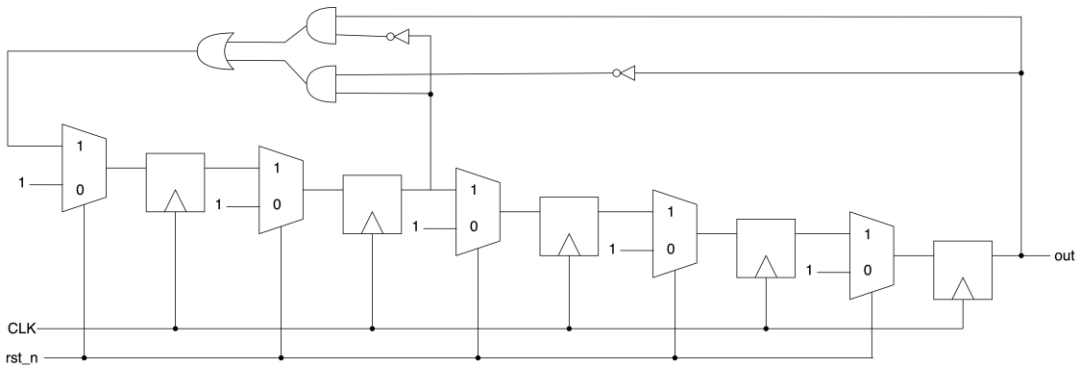

Linear-Feedback Shift Register

I. Diagram

(i) State diagram



(ii)電路設計圖



II. Explanation

依題意，將clk觸發設計為posedge。在always block中，先考慮是否會reset，如果會將DFF設為5bit 11111。反之，則進行shift運算，將DFF[3:0]依序shift至DFF[4:1]，DFF[4]的值則會成為out。以及透過and、or將DFF[1]及DFF[4]的值做xor運算成為DFF[0]的值。

```
always @(posedge clk)begin
    if(rst_n == 1'b0)begin
        dff <= 5'b11111;
    end
    else begin
        dff[4] <= dff[3];
        dff[3] <= dff[2];
        dff[2] <= dff[1];
        dff[1] <= dff[0];
        dff[0] <= (~dff[1] & dff[4]) | (dff[1] & ~dff[4]);
    end
end
assign out = dff[4];
```

接著解釋為何reset不是5bit 00000。如果reset為00000，shift一格後，新的DFF[0]在DFF[1]^DFF[4]運算後會是0。也就是在shift之後，整個DFF仍然是00000，將會進入無止境的輪迴。因此我們將reset設為11111，才能正常的去shift並跑過各種state。

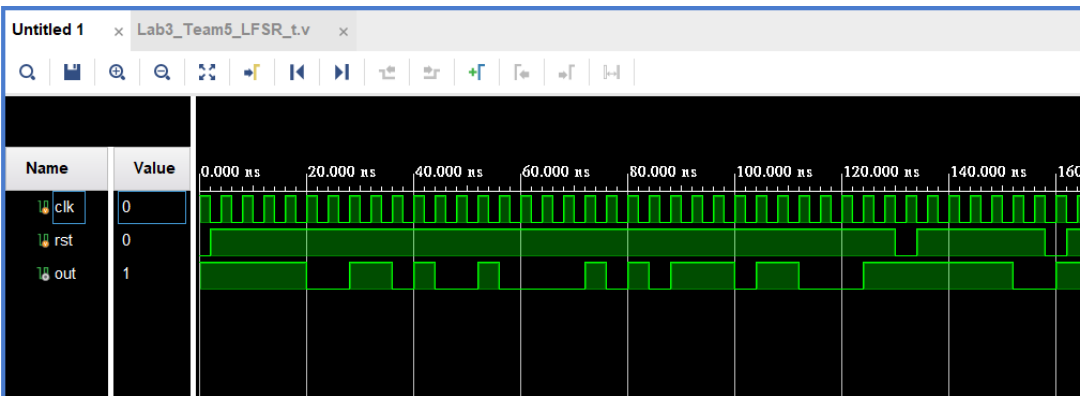
III. Testbench

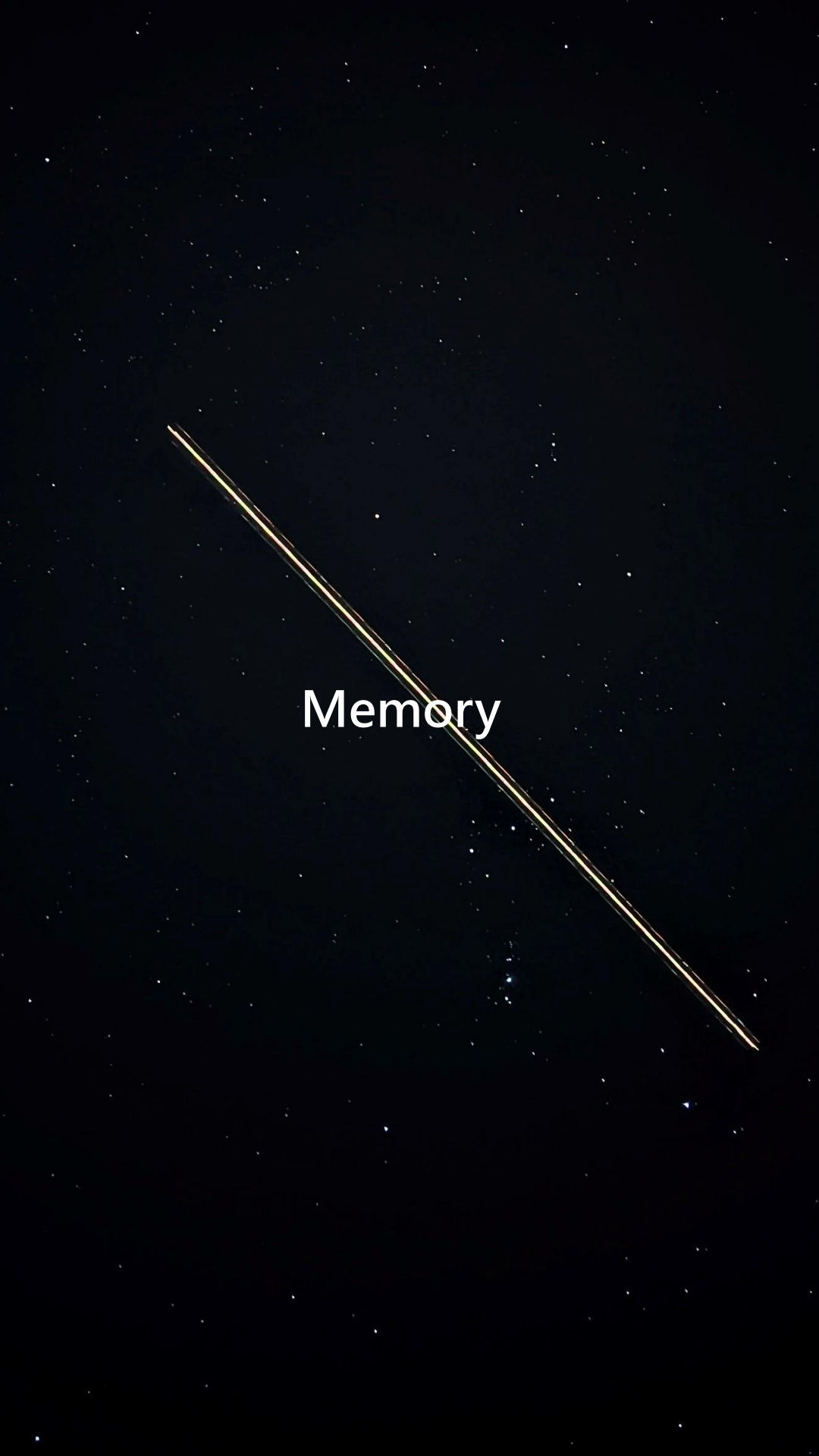
首先觀察State Diagram，我們發現總共有31個state，因此在設計Testbench時，我們先讓所有狀態都出現過一次，確定在所有狀況下都能照我們希望的樣子完成shift（見左圖）。

再確認過所有的state都正常之後，我們要測試reset的功能，因此繼續shift，並在negedge的時候隨機的插入reset訊號，測試reset功能正常（見右圖）。

```
rst =1 ;
repeat(2**5)begin
    @(posedge clk);
end
```

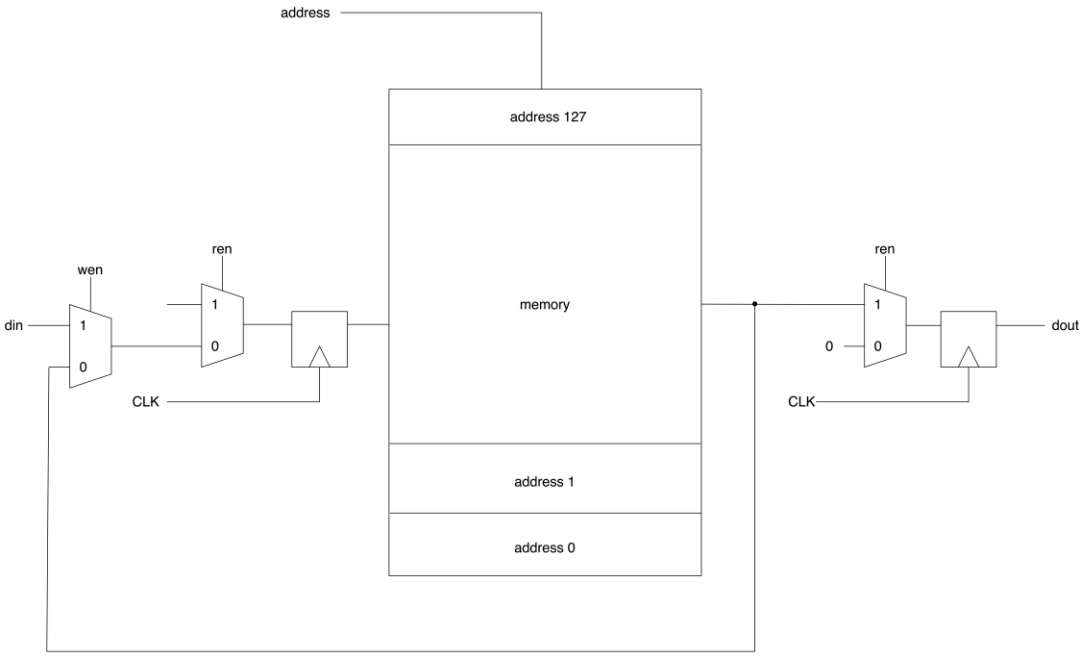
```
repeat(2**5)begin
    @(negedge clk);
    rst = ($random)%2;
end
```




Memory

I. Diagram

(i) 電路設計圖



II. Explanation

首先設計memory，將memory設計為可以存放128個adress，且每一格可以存8bits的數值的array。

```
reg [8-1:0] mem [128-1:0];
```

接著依題意設計為posedge trigger，然後因為在ren和wen同時為1的情況下，會執行read指令，因此先判斷ren。如果ren==1，dout=memory[address]。反之，繼續判斷wen，若wen=1，將din寫入memory[address]。若ren, wen皆為0，則不執行任何動作。

```
always @(posedge clk)begin

    if(ren == 1'b1)begin
        dout <= mem[addr];
    end
    else begin
        if(wen==1'b1)begin
            mem[addr] <= din;
        end
        else begin
        end
    end
end
```

III. Testbench

首先，先將ren訊號設定為0，讓wen訊號為1，開始將memory中所有的address逐一寫入din（見左下圖）。在寫入完成後將ren訊號設定為1，wen訊號為random%2，此舉可以測定ren訊號為1時無論wen訊號為何，都只進行讀取值的動作。同時，我們將address的值由127至0，讓dout可以獲取memory每一個address的值（見右下圖）。最後我們用肉眼比對每一個從memory取出的dout與當初放入memory的din是否相等。

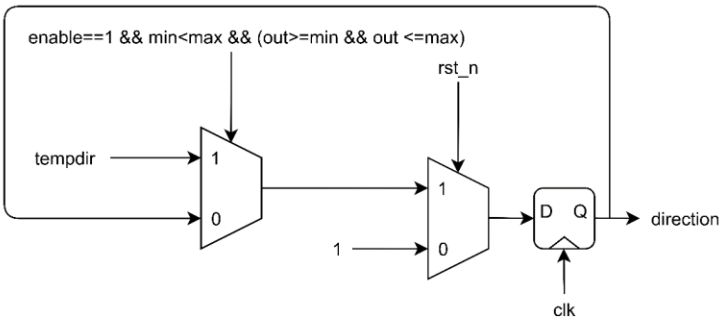
```
read = 1'b0;
write = 1'b1;
repeat(2**7)begin
    @(negedge clk);
    in = ($random)%128;
    address = address+1'b1;
end
```

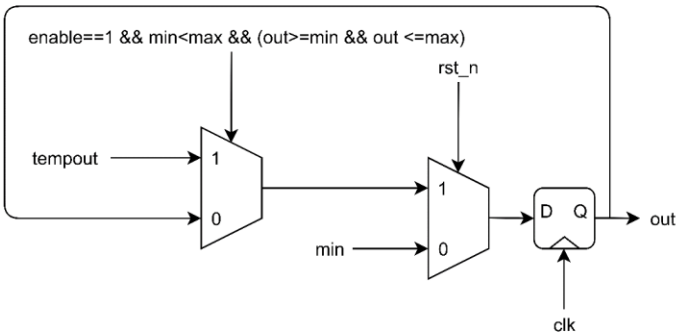
```
in = 1'b0;
read = 1'b1;

repeat(2**7)begin
    @(negedge clk);
    write = ($random)%2;
    address = address-1'b1;
end
```

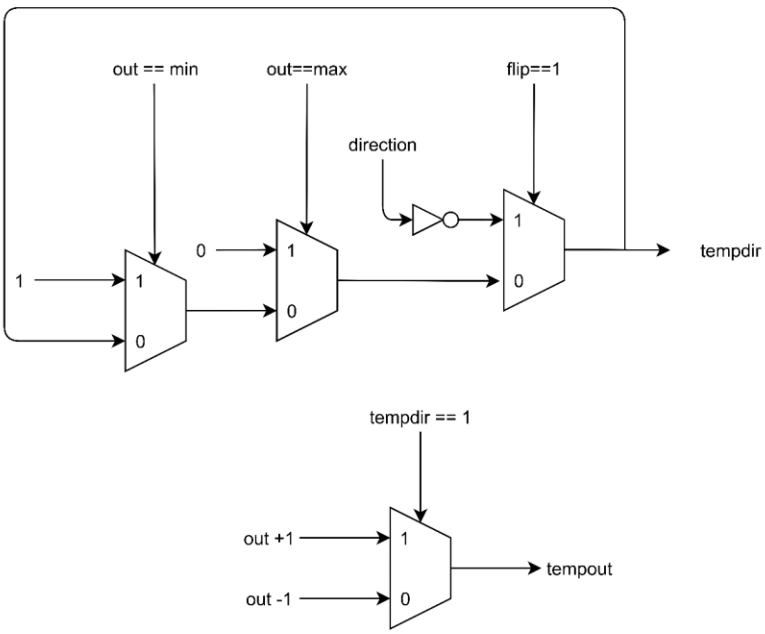
4-bit Ping-Pong Counter

• I. Logic Design Diagram

依序為direction實作、out實作，tempdir實作，tempout實作。







- II. Explanation

這次Ping Pong counter 實作中，使用tempdir、tempout兩個reg去預先計算output out跟direction下一個clk的值會是多少， tempdir、tempout 使用combinational 接法去計算值，而out、direction則是受到clk的控制在每個posedge clk才會從tempout、tempdir取值。

(左圖)而代表目前方向的tempdir會受到out是否等於max、min，或是flip是否等於1影響而改變值為1(向上) 或者是0(向下)。

(左下圖)代表輸出數字的tempout 則會受到tempdir是否等於1影響，若tempdir 為1，tempout即等於out+1，反之則等於out-1。

(右圖)而out、direction則是在posedge clk 的時候去檢測，若是rst_n為0則將out、direction初始化為min、1，反之則檢測enable是否等於1、min是否小於max、out是否大於等於min、out是否小於等於max，若皆成立則將tempout的值給out、將tempdir的值給direction。

```
always @(*) begin
    if(flip == 1) begin
        tempdir = ~direction;
    end
    else begin
        tempdir = tempdir;
    end

    if(out==max) begin
        tempdir = 1'b0;
    end
    else begin
        tempdir = tempdir;
    end
    if(out ==min) begin
        tempdir = 1'b1;
    end
    else begin
        tempdir = tempdir;
    end
end
```

```
always @(posedge clk) begin
    if(rst_n==0) begin
        out <=min;
        direction <=1;
    end
    else begin
        if(enable==1 && min<max && (out>=min && out <=max) ) begin
            out <=tempout;
            direction <= tempdir;
        end
        else begin
        end
    end
end
```

```
always @(*) begin
    if( tempdir==1'b1 ) begin
        tempout = out+1;
    end
    else begin
        tempout = out-1;
    end
end
```

- III. Testbench

(圖左上)本次測試有三個階段，第一階段測試在無flip的狀況下讓counter在max以及min之間來回。

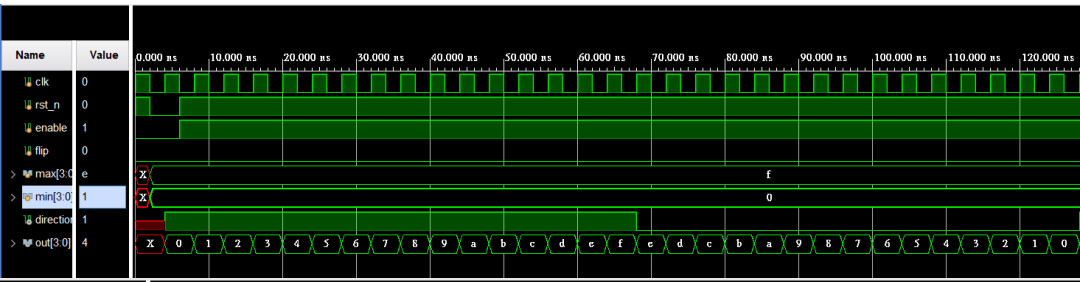
(圖中上)第二階段測試在max>min的不同max以及min的狀況中加入flip測試訊號。

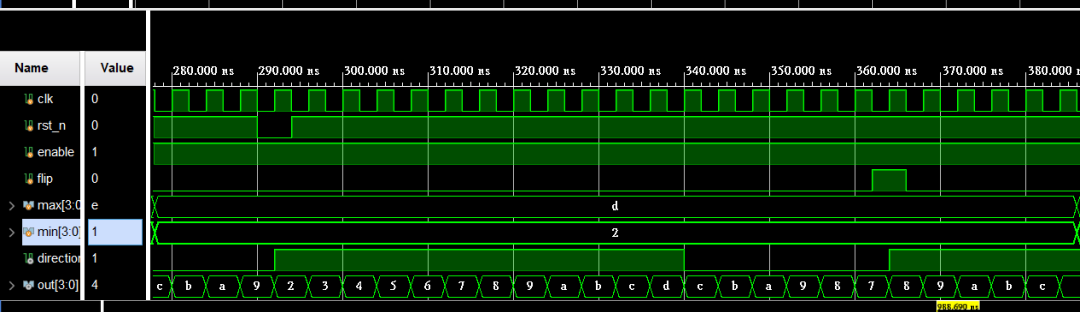
(圖右上)第三階段則是測試max<min的狀況，觀察out以及direction是否會保持原先的值。

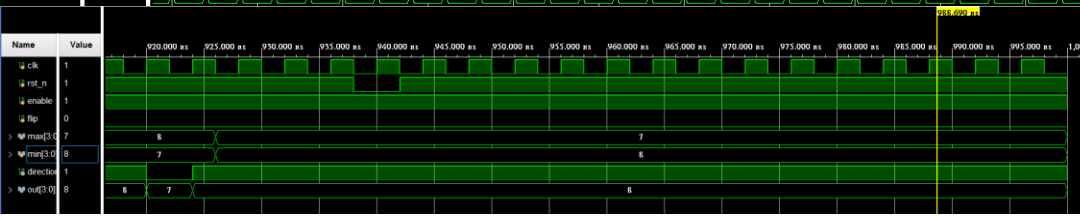
```
@(negedge clk);
    max = 15;
    min = 0;
    rst_n = 1'b0;
@(negedge clk);
    rst_n = 1'b1;
    enable = 1'b1;
#(`CYC * 30);

//test flip
#(`CYC * 5);
@ (negedge clk);
    flip = 1'b1;
@ (negedge clk);
    flip = 1'b0;
#(`CYC * 5);
```

```
repeat(8)begin
    @(negedge clk);
        max = max-1;
        min = min+1;
    #(`CYC * 3);  //test
    @(negedge clk);
        rst_n = 1'b0;
    @(negedge clk);
        rst_n = 1'b1;
        enable = 1'b1;
    #(`CYC * 12);
```

```
    //test flip
    #(`CYC * 5);
    @ (negedge clk);
        flip = 1'b1;
    @ (negedge clk);
        flip = 1'b0;
    #(`CYC * 5);
end
```



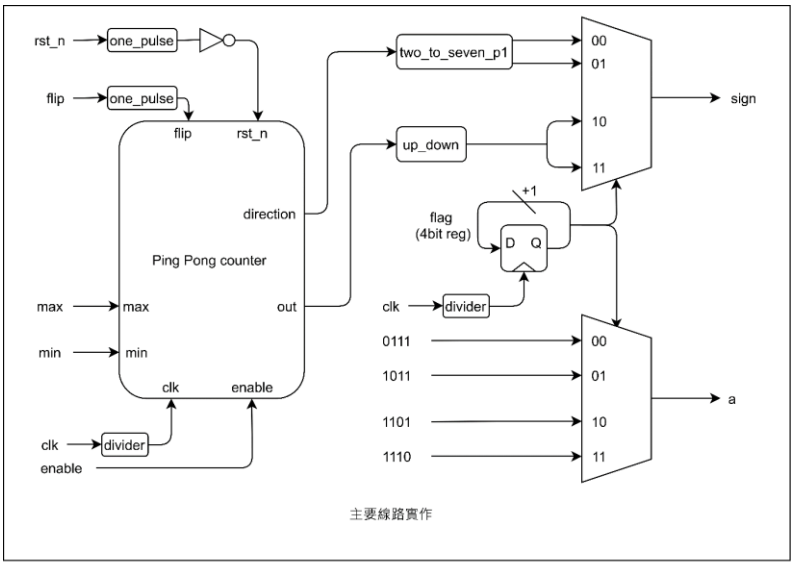




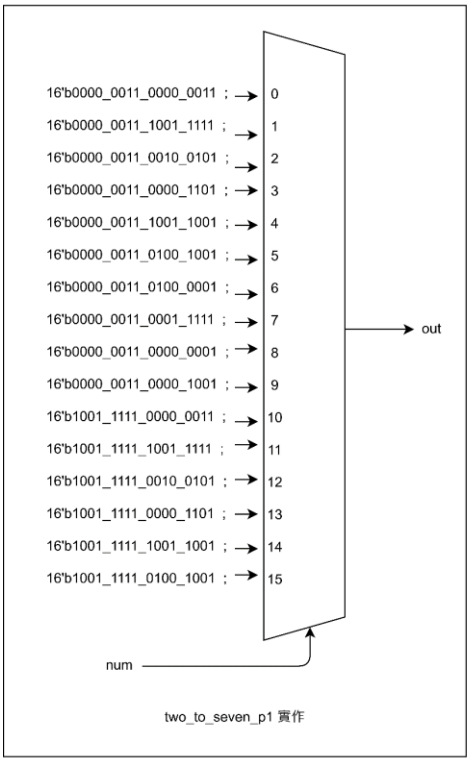
4-bit Ping-Pong Counter FPGA

- I. Logic Design Diagram

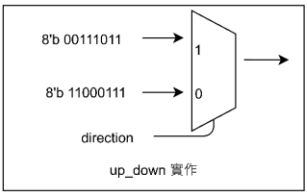
主要線路中 sign 代表七段顯示器要顯示的值，a則對應哪一個燈要亮起，而debounce也已經實作在one_pulse內部了，因此主要線路物理按鍵input只有接上one_pulse module處理，而在two_to_seven_p1 中會將目前要顯示的數字轉為8+8bit的訊號分別給顯示器的左邊兩位數、up_down 則負責將direction轉為上或下的箭頭顯示訊號。



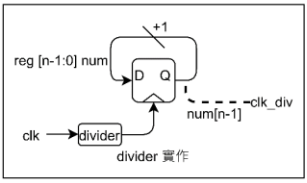
主要線路實作



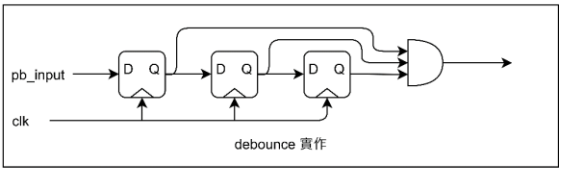
two_to_seven_p1 實作



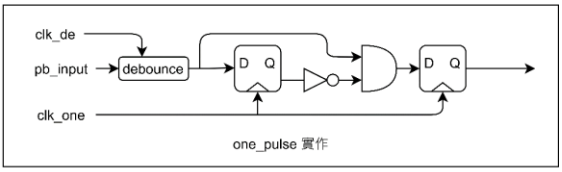
up_down 實作



divider 實作



debounce 實作



one_pulse 實作

- II. Explanation

本次FGPA實作，主要是在Q3的module外層做一個module處理輸入輸出，在輸入訊號處，對rst_n、flip做one_pulse、debounce的處理再接入Q3 module，而在Q3輸出後將它的結果decode成為FPGA版上七段顯示器的訊號。

在輸入接口上使用debounce處理是為了減少FPGA版上物理按鍵的雜訊干擾，使訊號穩定後才送入其他module處理，one_pulse處理，則是為了在長壓物理按鍵時不會送入多於一次的訊號，避免發生重複觸發的狀況。

為了在視覺上達成七段顯示器四位數顯示不同的效果，我們使用原先100Mhz的頻率除頻2的18次方，所產生新的頻率去刷新七段顯示器的四位數字，在這個頻率每跳動四次就會將顯示器的四個位數都顯示過一遍。而數字跳動的頻率則是使用100Mhz的頻率除頻2的25次方實作。

而本次實作重要的部件之一 divider，在製作時刻意寫成可以依據指定數字去輸出不同頻率的clk，如附圖，如此一來，就不用為了需要輸出不同頻率的clk而去多寫額外的module了。

```
module divider #(parameter n = 25) (clk, clk_div);
    // parameter n = 25;
    input clk;
    output clk_div;
    reg [n-1:0]num = 0;
    wire [n-1:0]next_num;

    always @(posedge clk) begin
        num <= next_num;
    end
    assign next_num = num + 1;
    assign clk_div = num[n-1];
endmodule
```

What are we learned from this Lab

在這次的lab實作中，我們充分體會到code架構的重要性。在Advanced Q3及FPGA實作時，我們遇到不少問題，最後我們發覺。使用combinational電路先計算出next的值，然後等到clock trigger時在將值做更新的這一個架構，可以避免掉許多的錯誤，也可以讓debug更容易。

而這也是我們第二次寫有關七段顯示器的題目，這次比起之前多開放了if else、case等寫法，除了可以實作更多功能同時也讓我們coding更加順暢，想必我們未來得要繼續精進熟練相關的寫法。

另外，我們在與不同組學生在相互討論時發覺，許多組別都發生類似的錯誤。因為reset訊號在FPGA按鈕按下時為1，但在寫simulation時，reset訊號等於0的時候會reset。許多人一開始沒有注意到這個細節，使得FPGA燒錄後，一直維持在reset狀態不會進入count的環節，也因此不知道自己錯在哪裡，大家搞半天才發現問題其實不大，也讓一起討論的我們不知該哭還是該笑。

而這次lab3的題目，不論是basic還是advenced，都有些許題目和隔壁邏輯設計班曾經、或是目前的題目有些相似之處，而也因為我們曾經幫助隔壁班同學debug，在這次lab處理可以算是較為有經驗一些，也算是一種利人利己吧。

Cooperation

108062211 黃子瑋：

Ping Pong Counter 程式與report、FPGA實作與report、各Testbench debug

108062213 顏浩昀：

Linear-Feedback Shift Register實作與report、Memory實作與report、各Testbench設計