Lab3 Report

Team25

組長：蔡登瑞、組員：蔡政諺

Contents

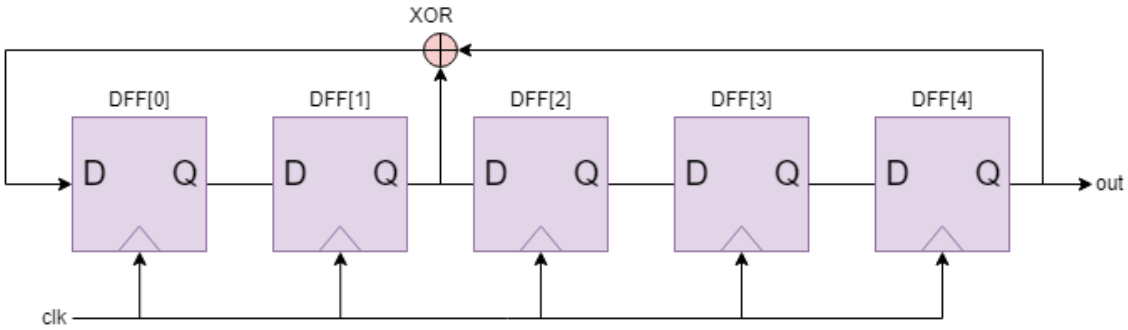
- **Designs**

1. LFSR

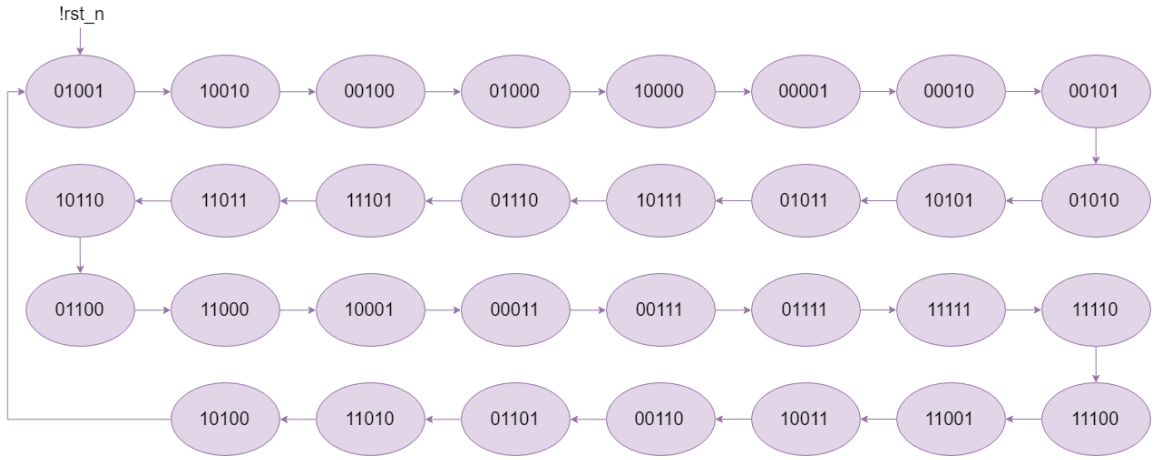
這題我開了一個 DFF[4:0]的陣列，在每一個 posedge clk 時，DFF 內的值會 shift 給下一位(DFF[4:1] <= DFF[3:0])，而 DFF[0]的值會是 DFF[1]與 DFF[4]XOR 出來的值(DFF[0] <= DFF[1]^DFF[4])。最後再 assign out = DFF[4]，就可以把 reg 型別的 DFF[4]接到 wire 型別的 out 了。

如果 reset 的值是 5'b00000 的話，在 posedge clk 時，DFF[4:1]得到 DFF[3:0]的值(3'b000)，DFF[0]得到 DFF[1]與 DFF[4]XOR 出來的值(0^0 = 0)，新的 DFF[4:0]的值仍為 5'b00000，不論經過幾個 cycle 都不會變。

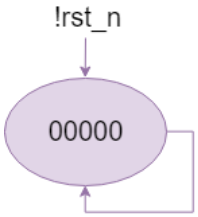
Gate-level diagram



State Transition Diagram



△ Case we reset DFF[4:0] to 5'b01001



△ Case we reset DFF[4:0] to 5'b00000

2. Memory

這題我開了一個 64*8 的陣列 mem，用於儲存 din 傳入的值。接著寫一個 always block，當 ren = 0 時，代表 read enable(無論 wen 為 1 或 0)，就將 mem 裡指定地址(addr)內的資料丟給 dout，但如果資料不存在，就輸出 8'b0；當 wen = 0 且 ren = 1 時，代表 write enable，就將 din 丟到 mem 裡的指定地址(addr)內，意即將想要的值存進 memory 裡，此時 dout 固定為 8'b0。此外，當 wen = 1 且 ren = 1 時，此 module 不執行任何動作，且 dout 固定為 8'b0。

Gate-level diagram

3. Parameterized Ping Pong Counter

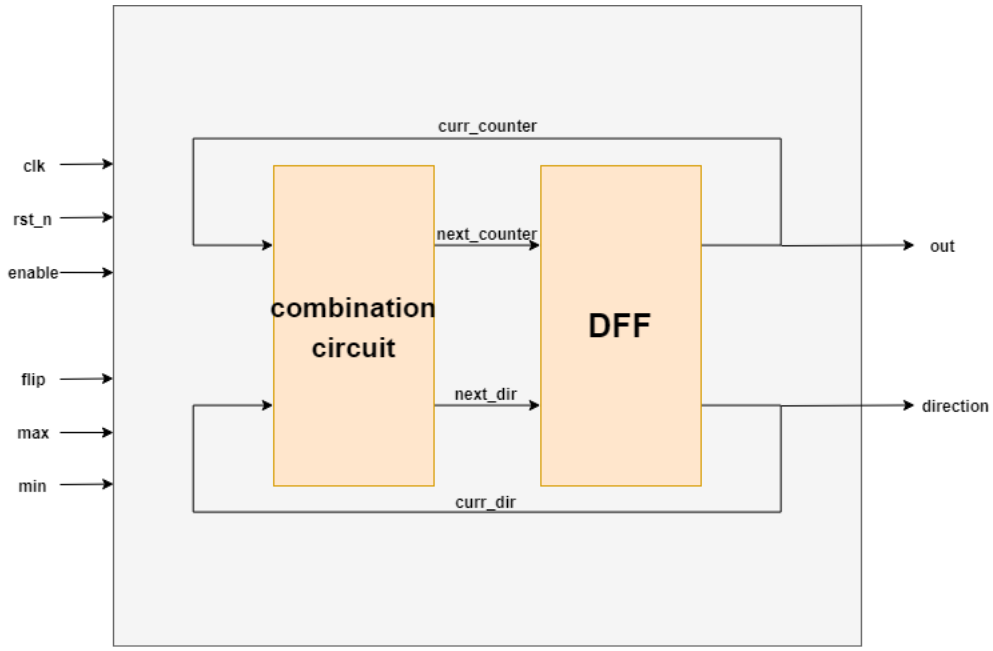
這題我們用了兩個 always block，一個是 DFF，一個是 combinational circuit。並分別各用兩個 reg 存取 current 跟 next 的 counter 和 clk。

在 DFF 裡，只做 synchronously reset，且將 next 的值賦予給 current。

而在另一個 always block 裡，用一些 if-else 的判斷式去判斷 current 的值是要怎麼處理，處理後存在 next 裡。等到下一個 posedge clk 來的時候，在 DFF 裡賦予給 current 值。

最後把 curr_counter assign 給 out、curr_dir assign 給 direction。這樣就完成了 Parameterized Ping Pong Counter。

Gate-level diagram



4. Parameterized Ping Pong Counter_fpga

設計想法：

首先，這題我們用了三個 clk，分別是 10ns、2^17x10ns、2^26x10ns，10ns 是 base clock，其他兩個都是以 base clock 為基準做 Clock Divider 得來的。2^17x10ns 是拿來作為顯示器的 AN 交替顯示的 clock，2^26x10ns 是 Ping Pong Counter 的 counter 的 clock。2^26x10ns 大約是半秒，也就是說 count 的間隔大約是半秒的時間。除了 PingPongCounter 以及 AN_replace 之外，所有的 sequential circiuts 都是以 base clock—10ns 來作為 clk，而 PingPongCounter 和 AN_replace 都是在自己的 module 裡面將 base clock 做 Clock Divider 的處理，而不是在 Top_module 裡面先做好 clock divider 後再輸入到小 module 裡面。

處理好需要用到什麼 clock 的問題後，接者要處理的是 flip 跟 rst_n 如何才不會造成 delay 或是 error。由於說每按一次按鍵，就會做一次 flip 或 rst_n，不管是按了多久，都只會算一次，因此就需要用到 debounce 跟 one_pulse 以 base clock 的 clk 來做訊號的處理。

一開始，我們是把已經除頻過好的 clock 拿去當作 one_pulse 的 clk。由於 one_pulse 會有一個 cycle 的 delay，而且必須按著按鍵維持一段時間後，在下一個 cycle 才會得到 one_pulse 的值，再下個 cycle 才會讓成功 flip 或 reset，也就是說，最多會 delay 兩個 cycle。可是這不是我們想要的。

然而以 base clock 處理成 one_pulse 的 flip 跟 rst_n，對於除頻過後的 clk 來說，很難去偵測到變化，後來我們想到說可以把 flip 跟 rst_n 變化的值存起來，存取的時間長短則是除頻過後的 clk (2^17x10ns, 2^26x10ns) 的一個 cycle。寫法就是下圖中紅色方框的地方。在 DFF 判斷的時候就用存起來的值去判斷(粉色方框)，而不是一開始輸入進 input 的 flip 跟 rst_n。這樣一來就不會造成 delay 兩個 cycle 的狀況

```
module Parameterized_Ping_Pong_Counter (clk, rst_n, enable, flip, max, min, direction, out);
  input clk, rst_n, enable, flip;
  input [4-1:0] max, min;
  output direction;
  output [4-1:0] out;

  reg [4-1:0] curr_counter, next_counter;
  reg curr_dir, next_dir;
  reg flip_for_PPC, rst_n_for_PPC;
  wire clk2_26;

  Divider_by_2_26 forDIGIT ( .clk(clk), .rst_n(rst_n), .clk2_26(clk2_26) );

  assign out = curr_counter;
  assign direction = curr_dir;

  always @(posedge clk2_26, posedge flip) begin
    if (flip) flip_for_PPC = 1'b1;
    else flip_for_PPC = 1'b0;
  end

  always @(posedge clk2_26, negedge rst_n) begin
    if (!rst_n) rst_n_for_PPC = 1'b0;
    else rst_n_for_PPC = 1'b1;
  end

  always @(posedge clk2_26) begin
    if (!rst_n_for_PPC) begin
      curr_counter <= min;
      curr_dir <= 1'b0;
    end
    else begin
      curr_counter <= next_counter;
      curr_dir <= next_dir;
```

Module 說明：

Parameterized_Ping_Pong_Counter：

以 10ns 為基準進行除頻，再以除頻過後的 clock (2^26x10ns)為 clk，每隔一個 cycle，count 一次。是以第三題為基準，再加上一些處理 flip 跟 rst_n 的東西。

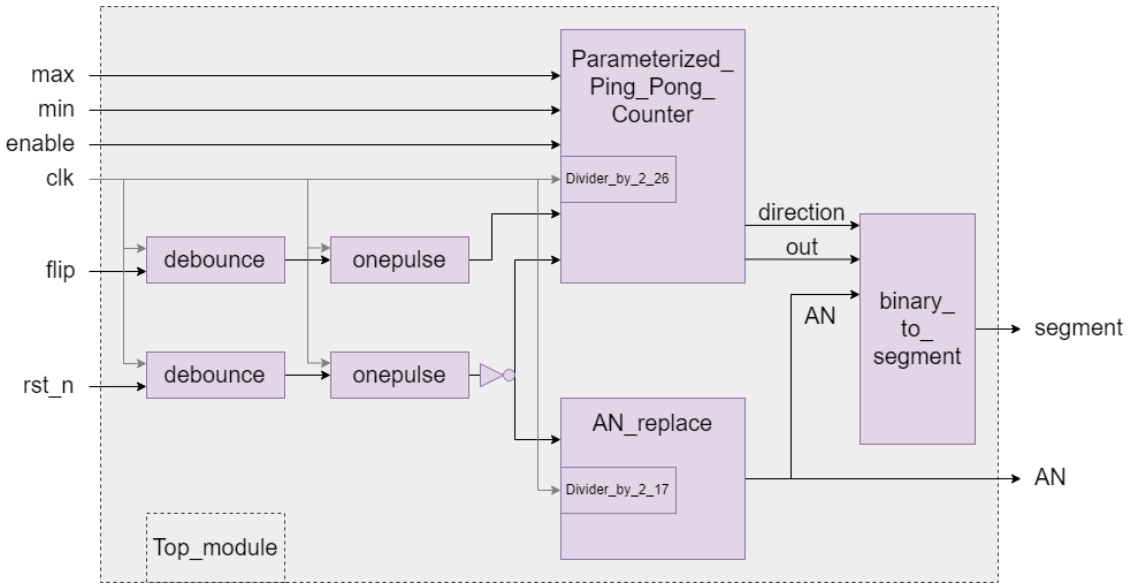
AN_replace：

以 10ns 為基準進行除頻，再以除頻過後的 clock (2^17x10ns)為 clk，每隔一個 cycle 變換一次要顯示的 digit 位置。

Binary_to_segment：

將 Ping Pong Counter 處理好的 out 依據 AN 和 direction 的不同，轉換成最後要在七段顯示器上顯示的樣子。

Gate-level diagram



- **How we test our designs?**

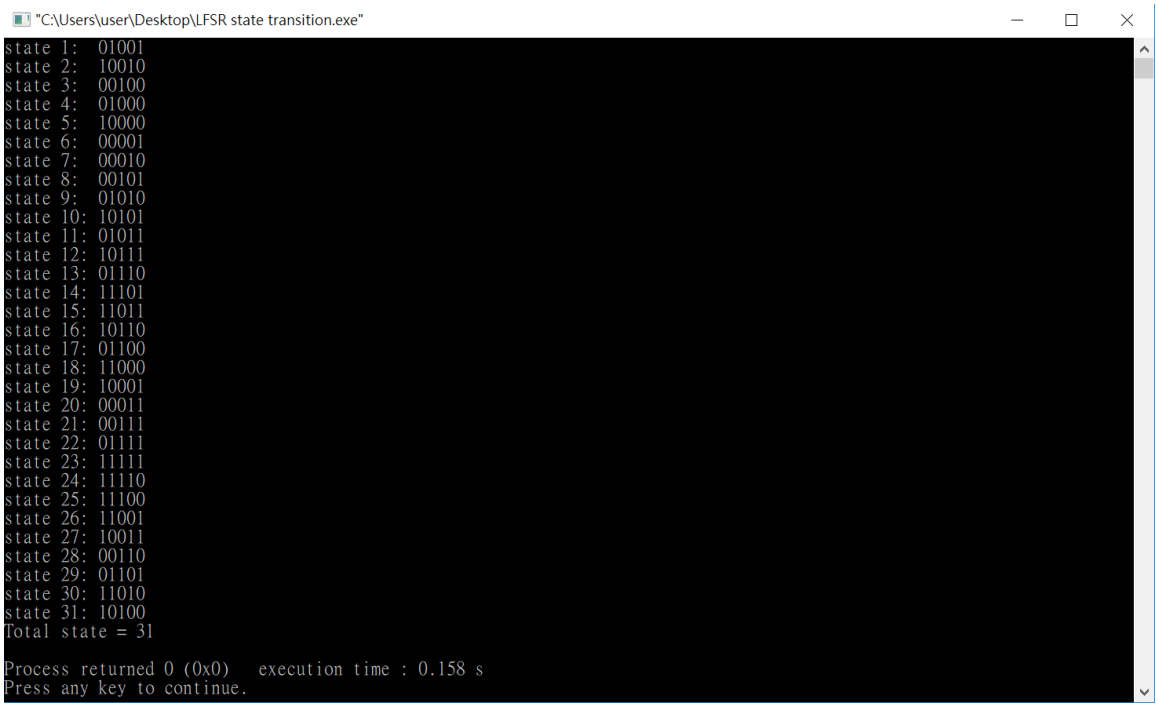
1. LFSR

先於 negedge clk 時將 rst_n 拉到 0，再於下一個 negedge clk 時將 rst_n 拉回 1，接下來跑若干個 cycle 即可。我跑了 32 個 cycle，因為最多可能產生 $2^5 = 32$ 種 state。

這題我寫了一個 C++ code 去跑 DFF[4:0]的 state transition，即可與波形圖交叉檢驗 out(DFF[4])是否正確。

Start here | LFSR state transition.cpp

```cpp
#include<iostream>
using namespace std;

int main(){

    int DFF[5];
    DFF[4] = 0;
    DFF[3] = 1;
    DFF[2] = 0;
    DFF[1] = 0;
    DFF[0] = 1;
    int state_count = 0;

    while(1){
        state_count++;
        cout << "state " << state_count << ": ";
        if(state_count<10) cout << " ";
        cout << DFF[4] << DFF[3] << DFF[2] << DFF[1] << DFF[0] << endl;

        int tmp = DFF[4];
        DFF[4] = DFF[3];
        DFF[3] = DFF[2];
        DFF[2] = DFF[1];
        DFF[1] = DFF[0];
        if(tmp&&!DFF[2]||!tmp&&DFF[2]) DFF[0] = 1;
        else DFF[0] = 0;


        if( DFF[4] == 0 &&
            DFF[3] == 1 &&
            DFF[2] == 0 &&
            DFF[1] == 0 &&
            DFF[0] == 1) break;
    }
    cout << "Total state = " << state_count << endl;

    return 0;
}
```



2. Memory

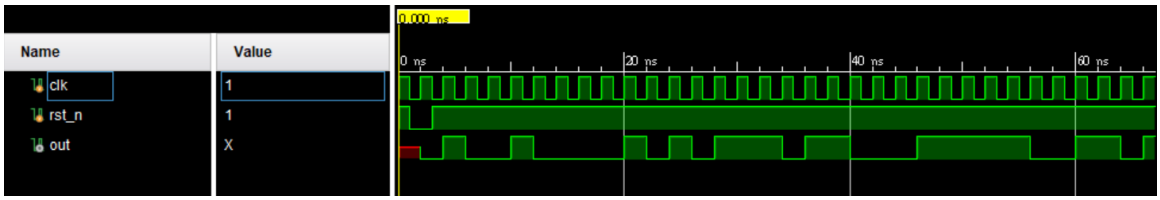
這題我是參照作業 pdf 所提供的波形圖去寫 testbench。其中測試了(wen, ren) = (0, 1)、(wen, ren) = (1, 1)、(wen, ren) = (1, 0)三種情況，若 simulation 結果與 pdf 中的圖形相同，就表示 design 無誤。除此之外，我還測試了(wen, ren) = (0, 0)時，是否正確執行 read operation。也測試在 read enable 的情況下，若讀的 address 裡還沒有存東西，dout 輸出結果是否為 8'b0。

3. Parameterized Ping Pong Counter

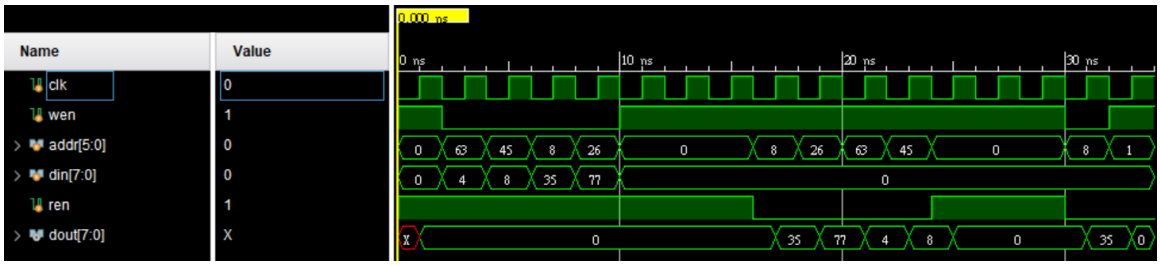
在這題的 testbench 中，一開始我設 max = 15，min = 0 跑若干個 cycle，以測試一般情況下 Parameterized Ping Pong Counter 是否能正常運作。接著我設 flip = 1 一個 cycle，測試 flip 的功能。過幾個 cycle 之後，再設 en(enable) = 0 跑若干個 cycle，測試 enable 的功能。過幾個 cycle 之後，我重新設定 max = 11，min = 8，並將 rst_n 設為 0 一個 cycle，測試 reset、min、max 的功能。跑若干個 cycle 後，再重新設定 max = 0，min = 15，測試 min>max 時，有沒有將值 hold 住。

- **Waveforms**

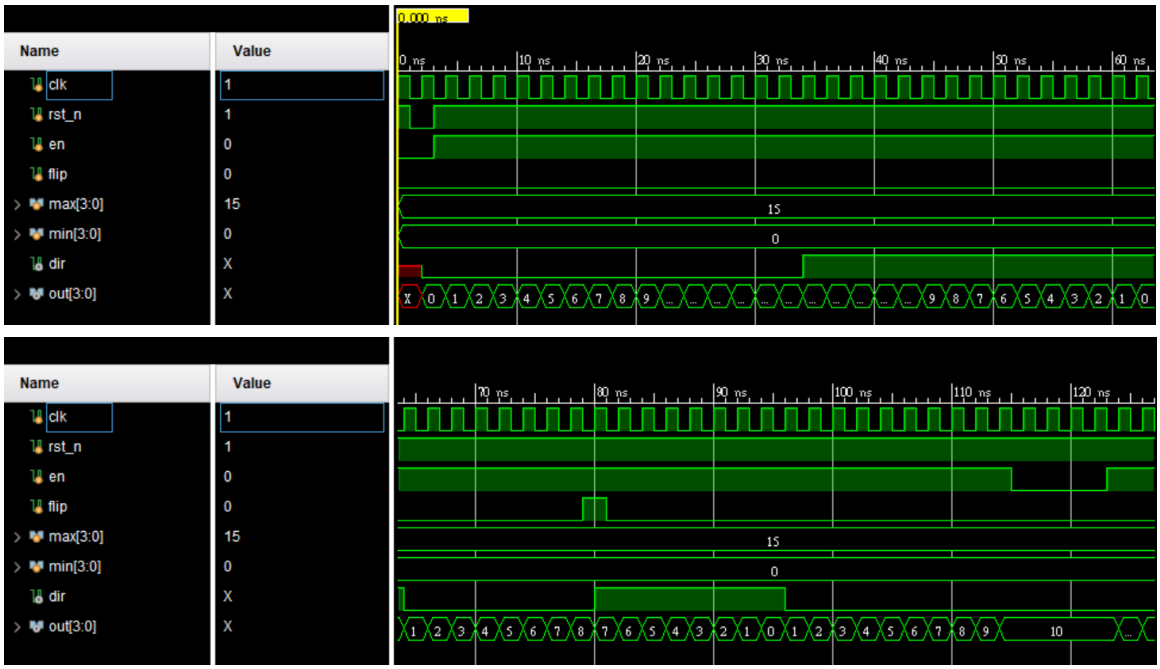
1. LFSR

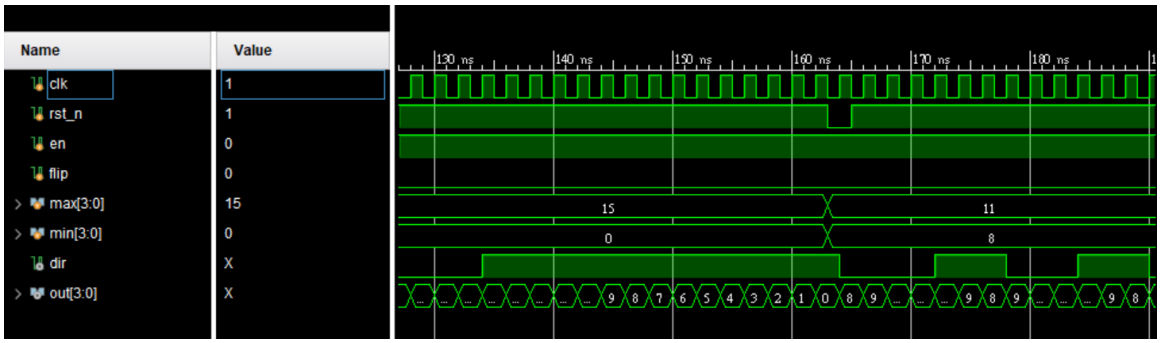


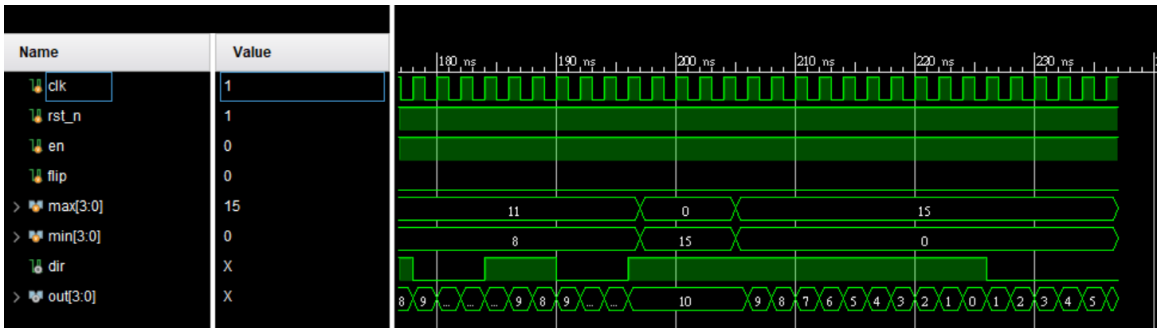
2. Memory



3. Parameterized Ping Pong Counter




Name
Value
clk
1
rst_n
1
en
0
flip
0
max[3:0]
15
min[3:0]
0
dir
X
out[3:0]
X
130 ns
140 ns
150 ns
160 ns
170 ns
180 ns
15
11
0
8
9 8 7 6 5 4 3 2 1 0 8 9 9 8 9 9 8


Name
Value
clk
1
rst_n
1
en
0
flip
0
max[3:0]
15
min[3:0]
0
dir
X
out[3:0]
X
180 ns
190 ns
200 ns
210 ns
220 ns
230 ns
11
0
15
8
15
0
8 9 9 8 9 10 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5

- **What we have learned from Lab3?**

這次的 Lab 我們從 Gate Level Description 進展到 Behavioral Modeling，開始練習使用 Always Block 和 Assign 等等的寫法。有了這種寫法，我們就可以用更簡單的語法詮釋出更複雜的邏輯。

Contribution List

- 蔡登瑞
  - 實作 FPGA
- 蔡政諺
  - 寫 testbench
  - 畫 state transition diagram
- 共同完成
  - 寫 code
  - 畫 gate-level diagram
  - 寫 report