Lab 3 Gate-Level Modeling

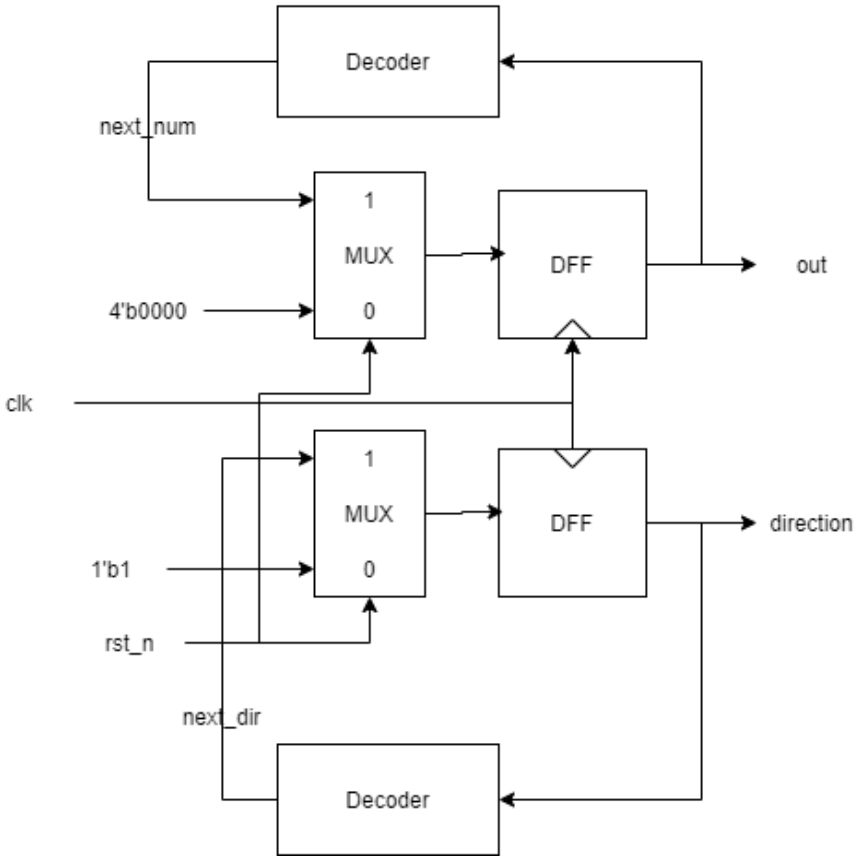
實驗報告

組長:劉奇泓 109033135

組員:洪聖祥 109062315

1. 4-bit Ping-Pong Counter

Block diagram:



4-bit ping pong counter input 有 clk , rst_n, 和 enable, output 有 out[3:0]和 direction。

我的 ping pong counter 大致上可以分為 4 個部分。2 個 sequential 電路和 2 個 combinational 電路。

一個 combinational 電路是用來計算下一個 direction 是什麼值 (code 中的變數為 next_dir)。我們發現，當 out[3:0]到達最上最下界時，下一個 direction 的值方向會改變(1 變 0，0 變 1)，

next_dir 的值跟當下 out[3:0]的值有關。也就是說當 out[3:0]等於 4’b0000 時，next_dir 的值需要變成 1’b1 而 out[3:0]等於 4’b1111 時，next_dir 的值需要變成 1’b0。

```
always@(*)begin
    if(enable == 1'b1)begin
        if(out == 4'b0000 ) next_dir = 1'b1;
        else if(out == 4'b1111) next_dir = 1'b0;
        else next_dir = direction;
    end
    else next_dir = direction;
end
```

而另一個 combinational 電路是用來計算下一個 out[3:0]是什麼值(code 中的變數為 next_num[3:0])。我們發現，next_num 的值跟 next_dir 的值有關。也就是說當 next_dir==1’b1 時，next_num 要加 1，反之，next_dir==1’b0 時，next_num 要減 1。

```
always@(*)begin
    if(enable == 1'b1)begin
        if(next_dir == 1'b1 ) next_num = out + 4'b0001;
        else next_num = out - 4'b0001;
    end
    else next_num = out;
end
```

兩個 combinational 電路都需要處理 enable==0。Enable==0 時，direction 和 out[3:0]都要維持不變。因此我們可以在兩個 combinational 電路中各自再加一個條件是 enable==0 時，next_num = out(下一個 out 跟現在 out 是一樣的)，next_dir = direction(下一個 direction 方向跟現在方向是一樣的)。

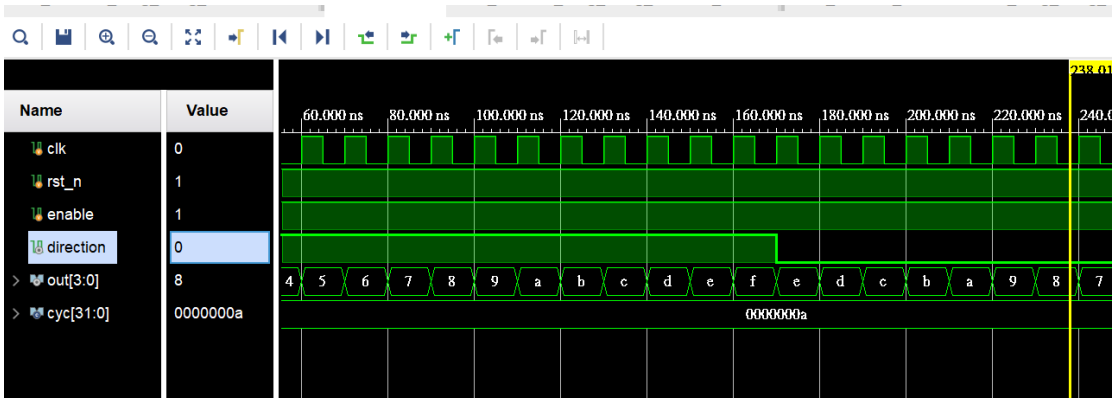
2 個 Sequential 電路都是用來儲存並更新 out[3:0]和 direction 的

值。電路中 rst_n 需要是 synchronize，在 clk 的 positive edge 上 rst_n==1’b1 時，更新 direction 和 out 的值，在 rst_n==1’b0 時，初始化 direction=1’b1 out[3:0]==4’b0000。

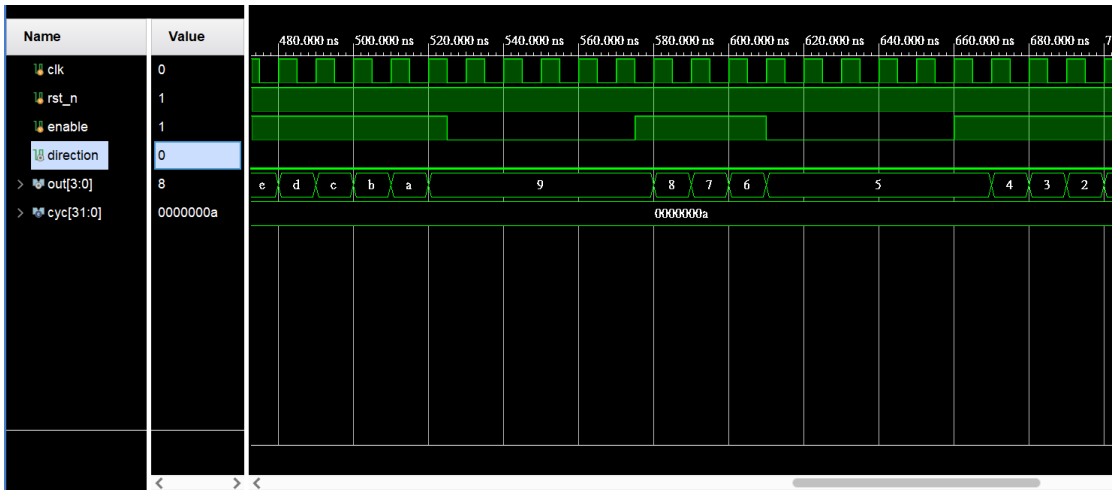
```
always @(posedge clk)
begin
    if(rst_n == 1'b1)begin
        out <= next_num;
        direction <= next_dir;
    end
    else begin
        out <= 4'b0000;
        direction <= 1'b1;
    end
end
always@(*)begin
```

波形圖：

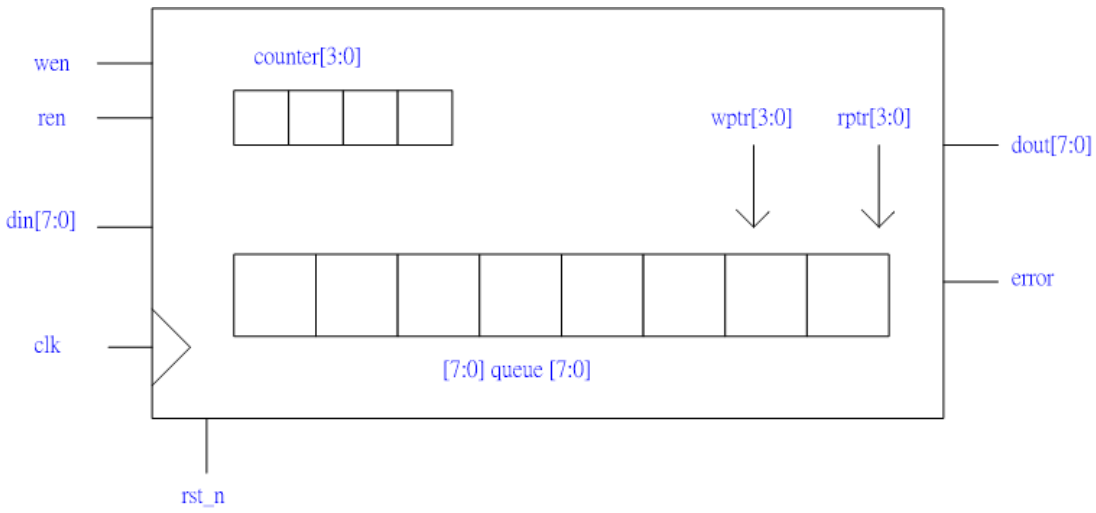
下圖是 enable == 1’b1 時的狀況，當 direction==1’b1 可以看見到每一個 clock 的 positive edge，out[3:0]會加 1。當 out == 4’b1111 時，下一個 out 會減 1。



下圖是 enable == 1’b0 時的狀況，當 enable == 1’b0 時，out 和 direction 都會維持原來的值直到 enable == 1’b1 才會開始運算。

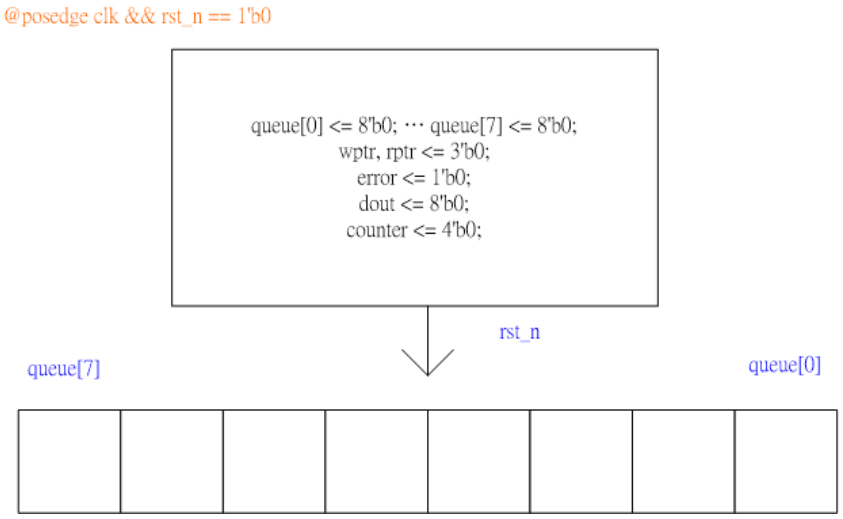


2. First-In First Out (FIFO) Queue



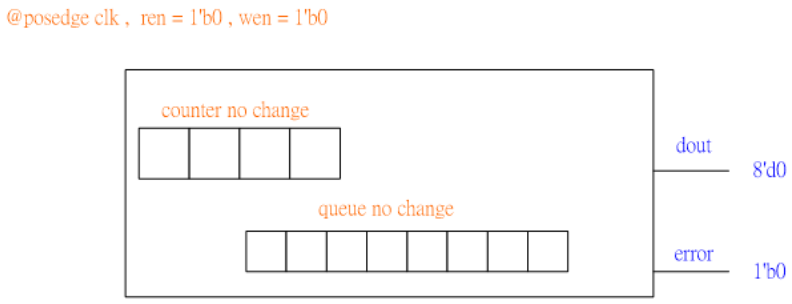

```
module FIFO_8(clk, rst_n, wen, ren, din, dout, error);
input clk;
input rst_n;
input wen, ren;
input [7:0] din;
output reg [7:0] dout;
output reg error;
reg [7:0] queue[7:0];
reg [2:0] wptr, rptr; // write pointer and read pointer
reg [2:0] counter;
```

FIFO queue 我們的 input 有 clk, rst_n, 有 ren 和 wen 來執行寫入和讀出的訊號，還有 din[7:0] 輸入寫入 queue 的內容；output 有 dout[7:0] 輸出 queue 的內容，error 輸出來表示出現錯誤；module 內部有 8 個 8-bits 的 queue，有 counter[2:0] 來記錄 queue 有多少儲存量，有 wptr，rptr 指在儲存資訊的開頭與結尾。如果 wptr 及 rptr 移動到左端 queue[7]，再加一就會 overflow 而回到 queue[0]的位置，符合 circular queue。



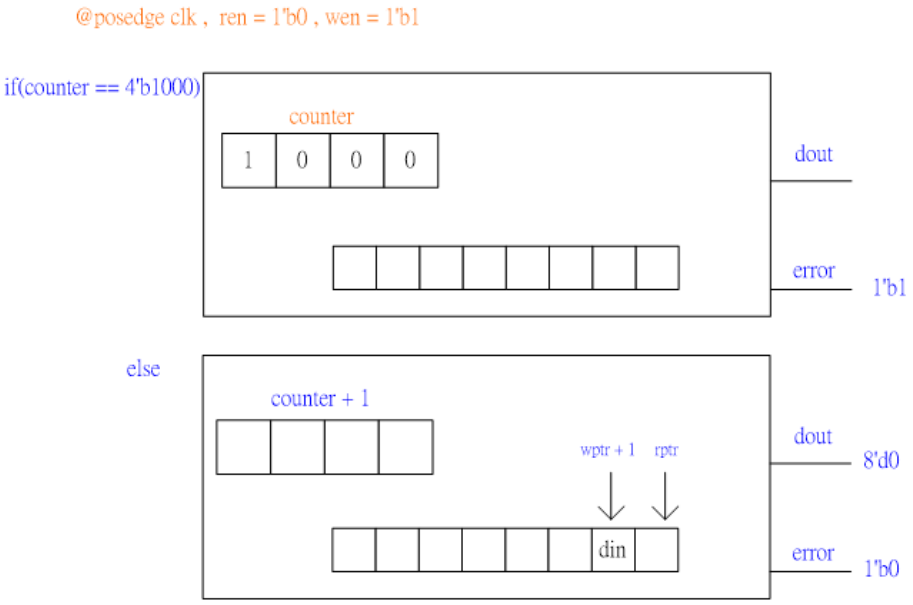
```
always@ (posedge clk)
begin
    if(rst_n == 1'b0) begin
        wptr <= 3'd0;
        rptr <= 3'd0;
        error <= 1'b0;
        dout <= 8'd0;
        queue[0] <= 8'd0;
        queue[1] <= 8'd0;
        queue[2] <= 8'd0;
        queue[3] <= 8'd0;
        queue[4] <= 8'd0;
        queue[5] <= 8'd0;
        queue[6] <= 8'd0;
        queue[7] <= 8'd0;
        counter <= 3'd0;
        end
```

在 clk posedge 上，如果 rst_n 就會進行 reset(synchronous reset)，把 wptr、rptr 都指到 queue[0]的位置，並把 queue 內容全部歸零，output 的 dout、error 都輸出 0。



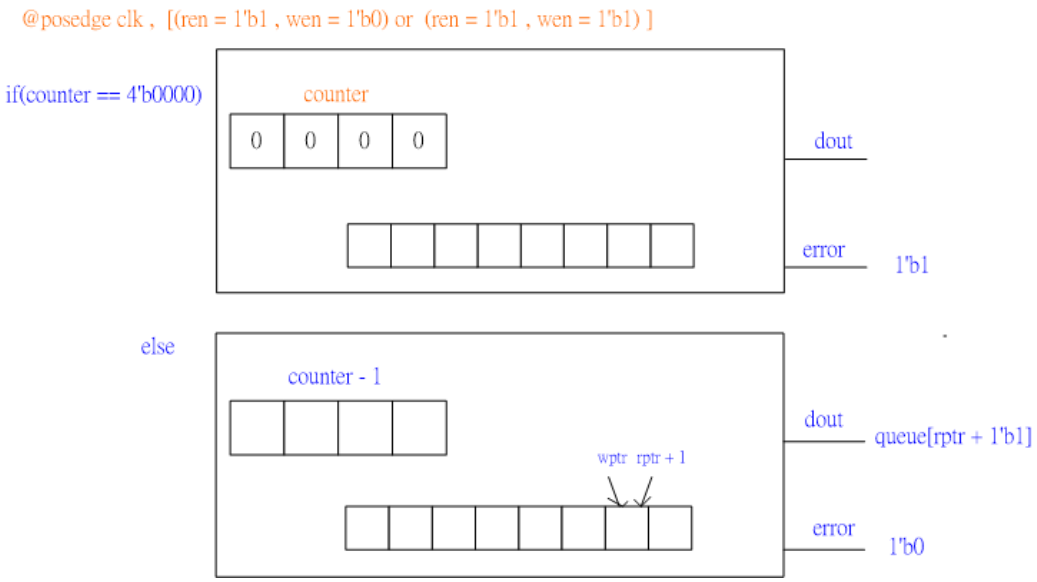

```
else
 case ({ren, wen})
 2'b00: begin
        error <= 1'b0;
        dout [7:0] <= 8'd0;
        counter <= counter;
        end
```

如果 rst_n = 1，且 ren、wen 都為 0 時不進行任何 write、read 的動作，error 和 dout 都輸出 0，counter 也不變。




```
2'b01: begin
   if(counter == 3'b111) error <= 1'b1;
   else begin
       error <= 1'b0;
       queue[wptr + 1'b1][7:0] <= din;
       dout[7:0] <= 8'd0;
       wptr <= wptr + 1'b1;
       counter <= counter + 3'd1;
       end
   end
```

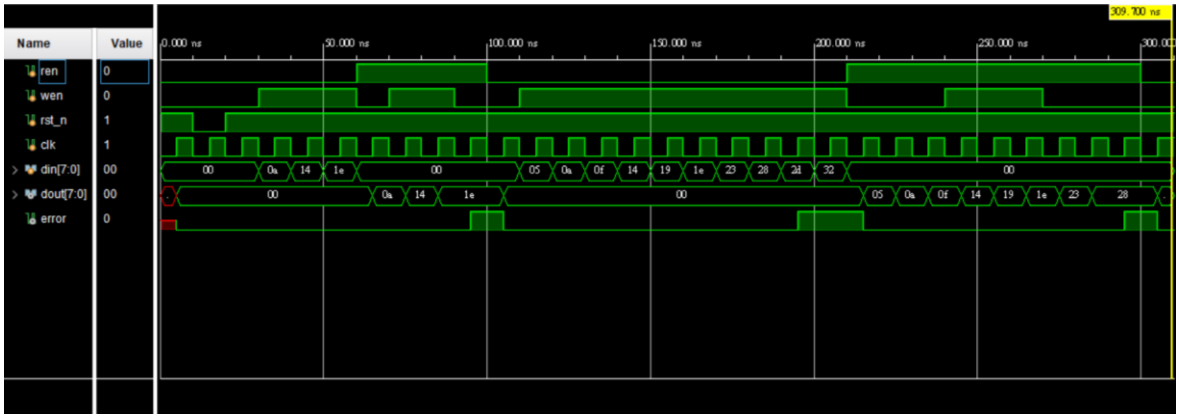
當 ren = 0、wen = 0 時進行 write 的動作，在 wptr 的左邊一格寫入 din、wptr 往左移一格，counter + 1，dout、error 輸出 0。但這時如果 counter = 3’b111 的話表示 queue 已經全滿了，就不會進行以上的動作並輸出 error = 1。




```
2'b10, 2'b11: begin
   if(counter == 3'b000) error <= 1'b1;
   else begin
       error <= 1'b0;
       dout[7:0] <= queue[rptr + 1'b1];
       rptr <= rptr + 1'b1;
       counter <= counter - 3'd1;
       end
   end
endcase
end

endmodule
```

當 ren = 1、wen = 0 或是 ren = 1、wen = 1 時會進行 read 的動作，讀出 rptr 左邊一格的內容，由 dout 輸出，同時 rptr 往左移一格，error 輸出 0，counter - 1，這時如果 counter = 0 表示 queue 內是空的，就不會進行以上的動作並輸出 error = 1。

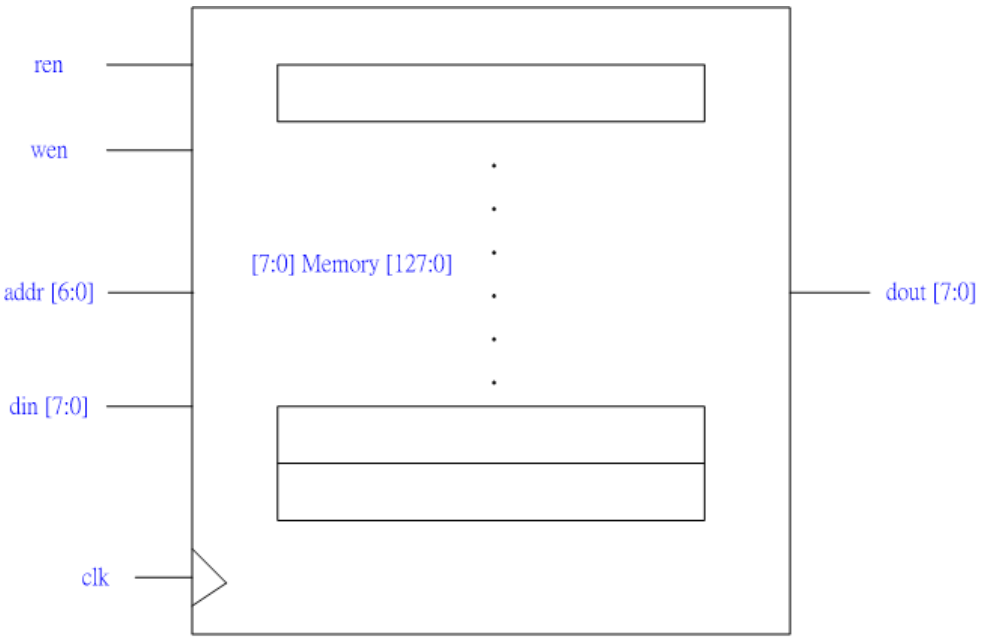
波形圖：



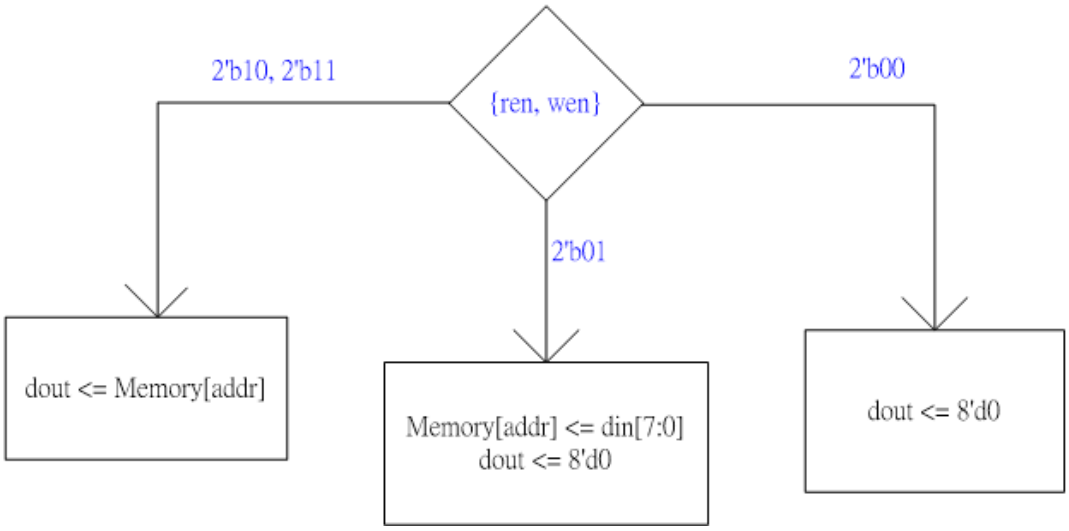
一開始先 reset，然後寫入三組內容再讀出，確認是否有按照 first in first out 讀出，並確認在 wen = 0、ren = 1 或 wen = 1、ren = 1 時只執行 read，嘗試讀出第四組內容時 error = 1；然後把 queue 寫滿，確認在寫入八組後就不再寫入且 error 輸出 1 最後全部讀出，確認之前輸入的 3 組數字不會再出現，並確認 queue 清空後要再讀出時 error 輸出 1。

3. Multi-Bank Memory

- Memory



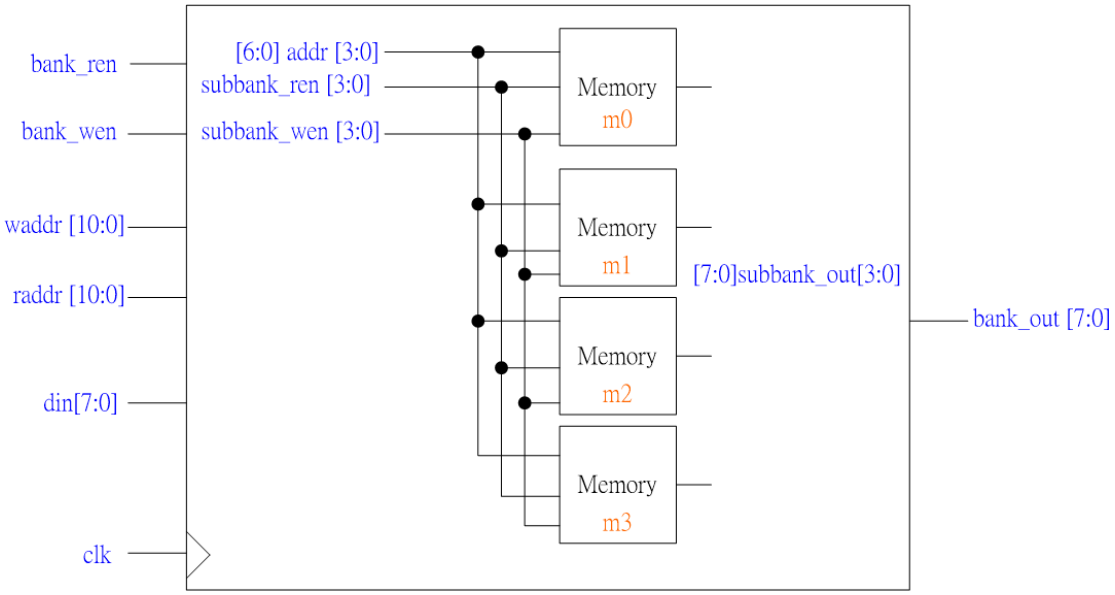
使用到 basic question 2 製作的 Memory，input 有 clk，ren、wen 下達 read 和 write 的訊號，addr 提供寫入的地址，din 提供輸入 Memory 的內容，output 有 dout 輸出 Memory 的內容。



```
always @(posedge clk)begin
    case({ren, wen})
    2'b00: dout <= 8'd0;
    2'b01: begin
           Memory[addr] <= din[7:0];
           dout <= 8'd0;
           end
    2'b10, 2'b11: dout <= Memory[addr];
    endcase

end
```

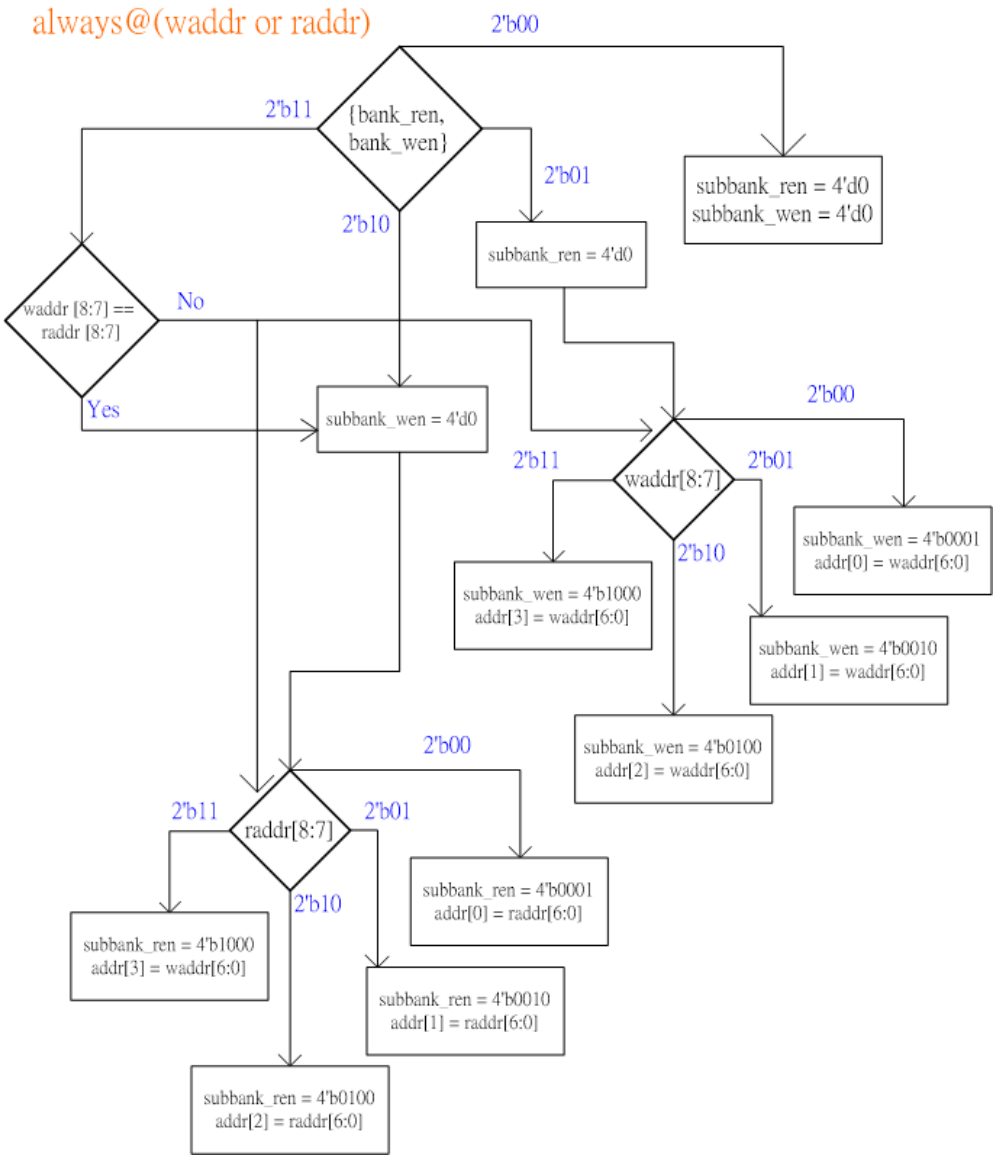
在 posedge clk 上測試 ren、wen 的值，如果{ren, wen} = 00 表示沒有 write 和 read，輸出 dout = 0，{ren, wen} = 01 表示執行 write，將 din 寫入 Memory 中指定的 address，並輸出 dout = 0，{ren, wen} = 10 或 11 執行 read，把 Memory 指定的 address 的內容由 dout 輸出。

- Bank



Bank 的 input 有 clk 送進四個 subbank(Memory)，bank_ren、bank_wen 選擇要 read 或是 write，waddr[8:0]、raddr[8:0]是

要 write、read 的地址，前兩位[8:7]選擇 subbank、後七位[6:0]選擇 subbank 內的 address，din 是要儲存的內容；output 為 dout 輸出 Memory 的內容；bank 內部有[6:0]addr[3:0]連接四個 subbank 的 addr[6:0]，subbank_ren[3:0]連接四個 subbank 的 ren，subbank_wen[3:0]連接四個 subbank 的 wen，四個 subbank 輸出 subbank_out 最後經由 bank_out 輸出。



此 always block 只要 wddr 和 raddr 改變了就會執行一次，先判斷{bank_ren, bank_wen}，如果是 00 則不 write 也不 read，四個 subbank 的 ren、wen 都設為 0。

如果{bank_ren, bank_wen} = 01 就要執行 write，先把 subbank_ren 都設為 0(不 read)，再由 waddr[8:7]找到要 write 的 subbank，把指定的 subbank_wen 設為 1，其餘為 0，再 waddr[6:0]傳給指定的 subbank 的 addr 寫入 Memory。

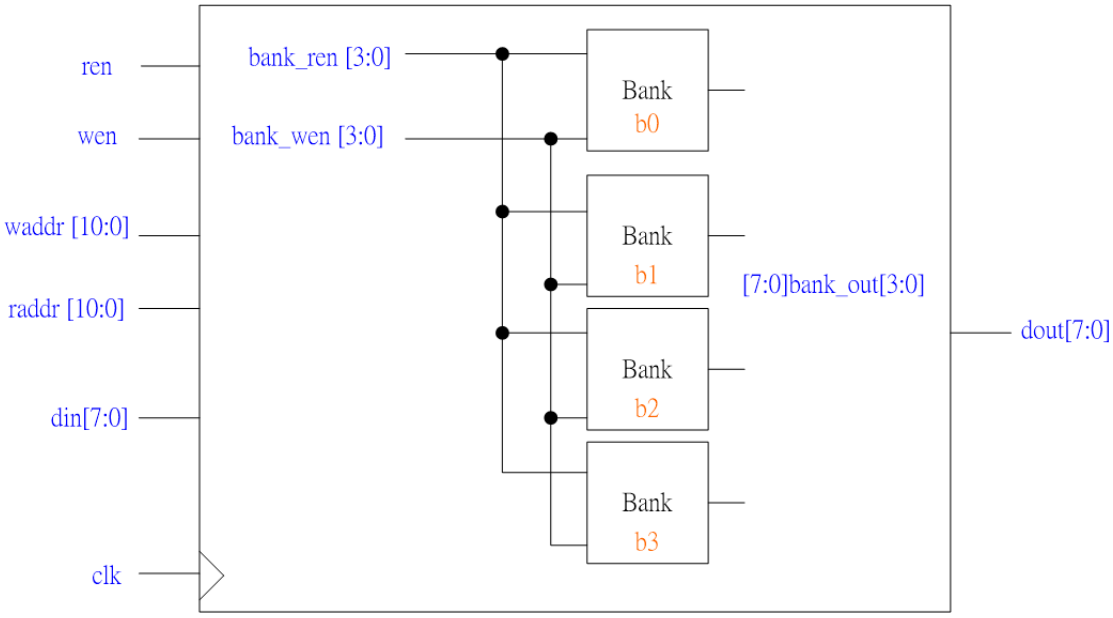
如果{bank_ren, bank_wen} = 10 就要執行 read，先把 subbank_wen 都設為 0(不 write)，再由 raddr[8:7]找到要 read 的 subbank，把指定的 subbank_ren 設為 1，其餘為 0，再 raddr[6:0]傳給指定的 subbank 的 addr 將內容從 Memory 讀出。

如果{bank_ren, bank_wen} = 11 要先判斷 waddr[10:9] == raddr[10:9](是否要同時 write、read 同個 subbank)，如果是的話就只執行 read 的動作，如果否就同時判斷 waddr[8:7]、raddr[8:7]，在兩個 subbank 同時執行 read、write 的動作。

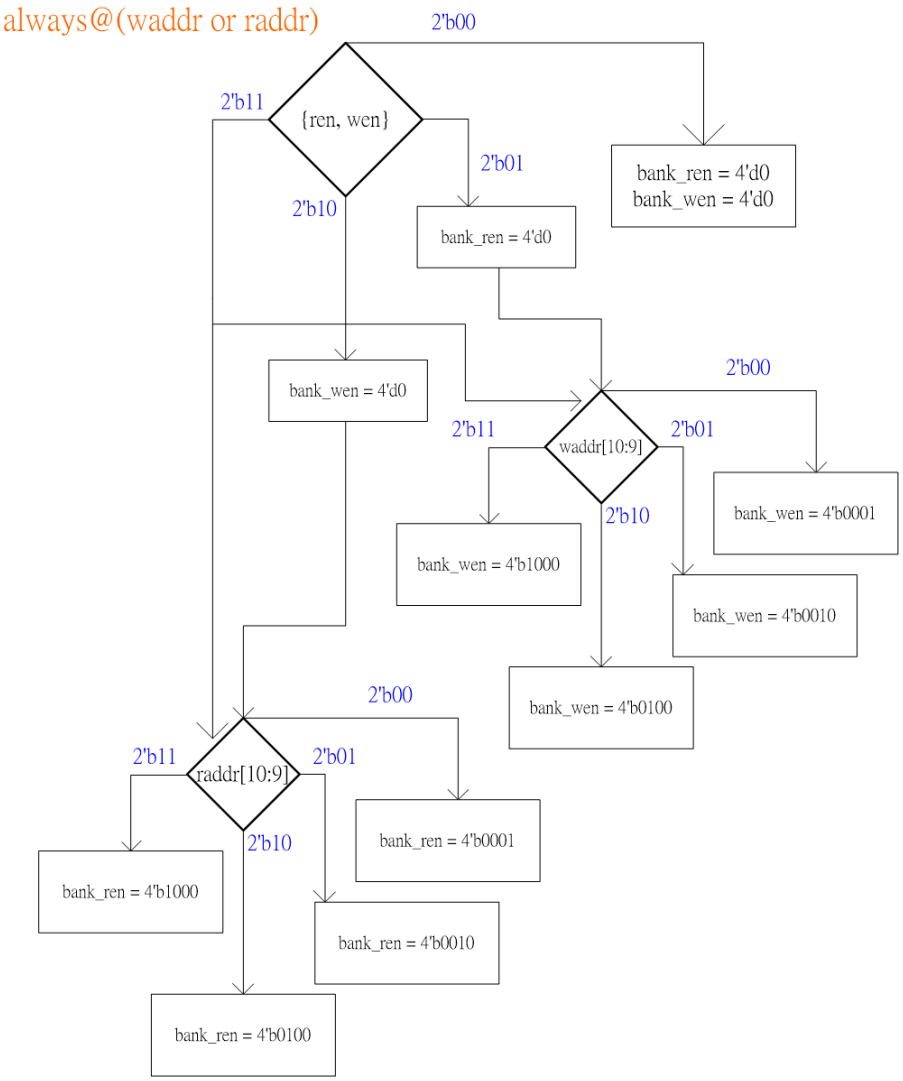
```
Memory m0 (clk, subbank_ren[0], subbank_wen[0], addr[0], din, subbank_out[0]);
Memory m1 (clk, subbank_ren[1], subbank_wen[1], addr[1], din, subbank_out[1]);
Memory m2 (clk, subbank_ren[2], subbank_wen[2], addr[2], din, subbank_out[2]);
Memory m3 (clk, subbank_ren[3], subbank_wen[3], addr[3], din, subbank_out[3]);

assign bank_out = subbank_out[0] | subbank_out[1] | subbank_out[2] | subbank_out[3];
```

最後得到四個 subbank 的 output，因為除了要讀出的 subbank 之外其他的 subbank_out 都是 0，所以 bank 的 output bank_out 就是四個 subbank_out 的 bitwise or。

- Multi bank memory



Multi bank memory 內部有 4 個 bank，用 bank_ren[3:0]、bank_wen[3:0]輸入四個 bank 的 subbank_ren、subbank_wen，輸出 bank_out 再連接到 dout。



此 always block 只要 wddr 和 raddr 改變了就會執行一次，先判斷{ren, wen}，如果是 00 則不 write 也不 read，四個 bank 的 ren、wen 都設為 0。

如果{ren, wen} = 01 就要執行 write，先把 bank_ren 都設為 0(不 read)，再由 waddr[10:9]找到要 write 的 bank，把指定的 bank_wen 設為 1，其餘為 0。

如果{ren, wen} = 10 就要執行 read，先把 bank_wen 都設為 0(不 write)，再由 raddr[10:9]找到要 read 的 bank，把指定的

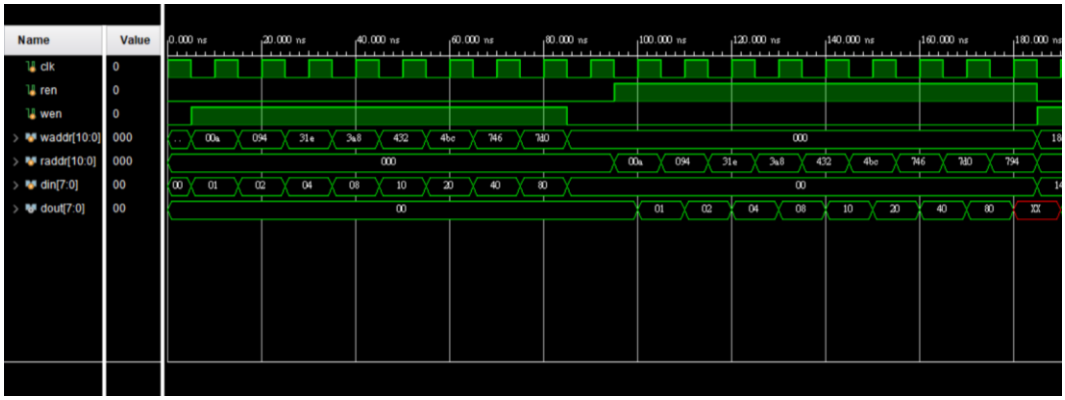
bank_ren 設為 1，其餘為 0。

如果{ren, wen} = 11 要同時判斷 waddr[10:9]、raddr[10:9]，在兩個 bank 同時執行 read、write 的動作。

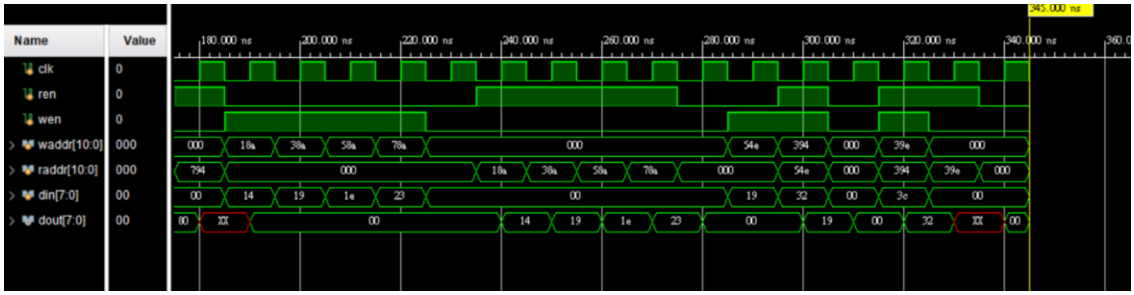
```
Bank b0(clk, bank_ren[0], bank_wen[0], waddr, raddr, din, bank_out[0]);
Bank b1(clk, bank_ren[1], bank_wen[1], waddr, raddr, din, bank_out[1]);
Bank b2(clk, bank_ren[2], bank_wen[2], waddr, raddr, din, bank_out[2]);
Bank b3(clk, bank_ren[3], bank_wen[3], waddr, raddr, din, bank_out[3]);

assign dout = bank_out[0] | bank_out[1] | bank_out[2] | bank_out[3];
endmodule
```

最後得到四個 bank 的 output，因為除了要讀出的 bank 之外其他的 bank_out 都是 0，所以 dout 就是四個 bank_out 的 bitwise or。

波形圖:

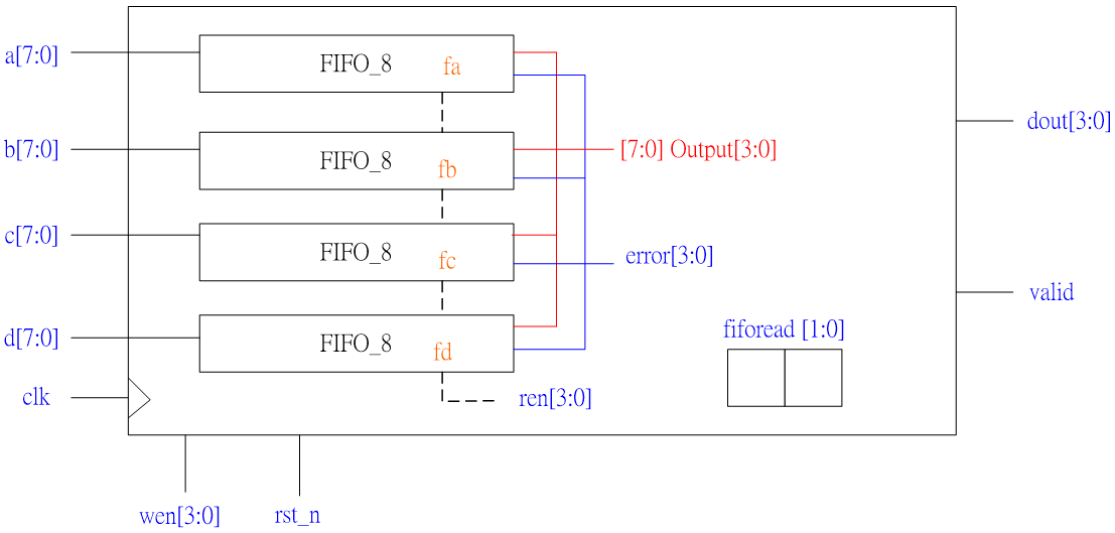


首先測試在每個 subbank 都寫入一組數字再照順序一一讀出看是否 write、read 的功能正常，再測試讀出一個額外的 address 是否會得到空的內容。



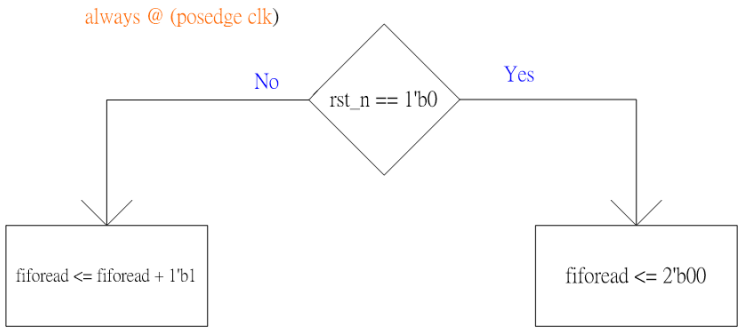
再來測試寫到 4 個 bank 的相同 subbank、address 是否能成功 read、write，最後測試是否**能**同時在不同的 subbank write、read，以及是否**不能**在同個 subbank write、read。

4. Round-Robin FIFO Arbiter



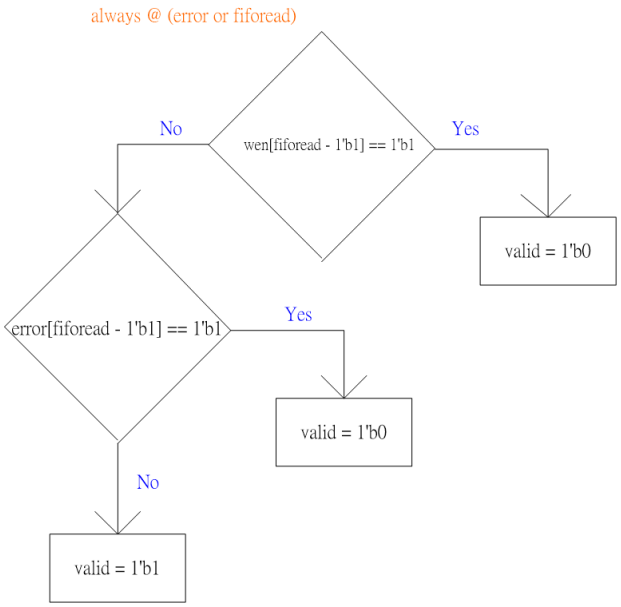
Round-Robin FIFO Arbiter 使用到 4 個在 advanced question 製作的 fifo queue，由 a、b、c、d input 儲存內容在 queue 中，其他 input 還有 clk、rst_n，wen[3:0]輸入 4 個 queue 的 wen 訊號。output 有 dout[3:0]，以及 valid 在無法正常輸出時給出訊號。Round-Robin

FIFO Arbiter 內部有 ren[3:0]提供 4 個 queue ren 訊號，[7:0] Output[3:0]輸出 queue 的 output，error[3:0]輸出 queue 的 error bit，fiforead[1:0]可以判斷輪到哪個 queue 輸出。



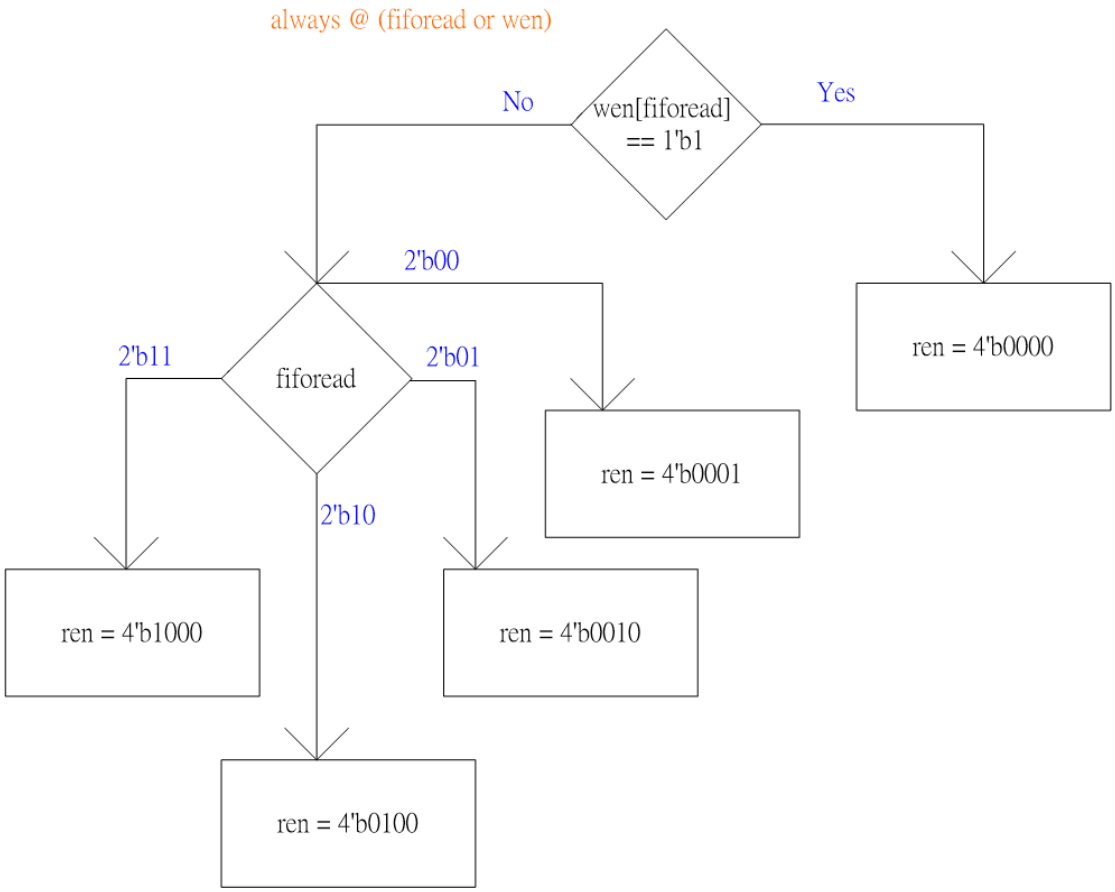

```
always @ (posedge clk) begin
    if(rst_n == 1'b0) fiforead <= 2'b00;
    else fiforead <= fiforead + 1'b1;
end
```

此 always block 在 posedge clk 時執行，判斷 rst_n == 0，如果是就將 fiforead 歸零；如果否則將 fiforead + 1。



```
always @ (error or fiforead)
begin
    if(wen[fiforead - 1'b1] == 1'b1) valid = 1'b0;
    else if(error[fiforead - 1'b1] == 1'b1) valid = 1'b0;
    else valid = 1'b1;
end
```

此 always block 在 error 或 fiforead 變化時執行，先判斷要 output 的 queue 是否正要被寫入，如果是就將 valid 設為 1，否則再判斷前一個 cycle 的 queue 是否回傳 error bit = 1，是則 valid 就設為 0，否則 valid 設為 1。



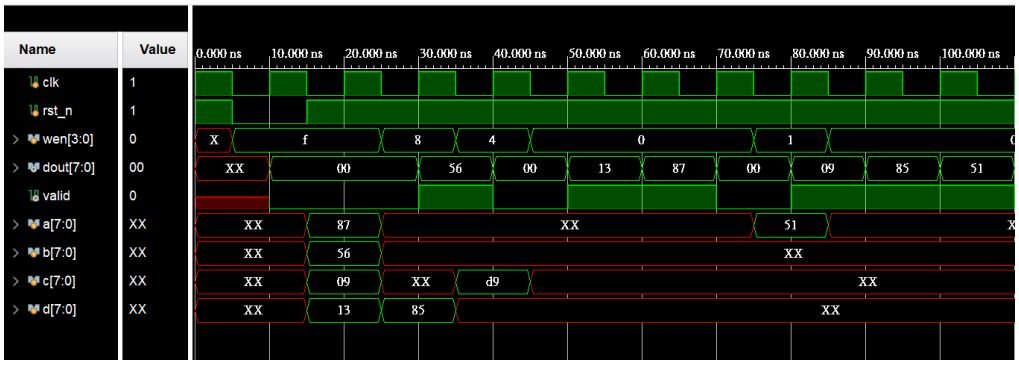
```
always @ (fiforead or wen)
begin
    if(wen[fiforead] == 1'b1)
        ren = 4'b0000;
    else
        case(fiforead)
        2'b00: ren = 4'b0001;
        2'b01: ren = 4'b0010;
        2'b10: ren = 4'b0100;
        2'b11: ren = 4'b1000;
        endcase

end
```

此 always block 在 fiforead 或 wen 變化時執行，如果 fiforead 輪到的 queue 正要被寫入時就讓 ren = 0000，讓該 queue 的 write 不會受到干擾。如果 queue 沒有要被寫入的話就再判斷 fiforead 在哪一個 state，fiforead = 00 為 queue a 輸出，ren 設為 0001 使其輸出，fiforead = 01 為 queue b 輸出，ren 設為 0010 使其輸出，fiforead = 10 為 queue c 輸出，ren 設為 0100 使其輸出，fiforead = 11 為 queue d 輸出，ren 設為 1000 使其輸出。

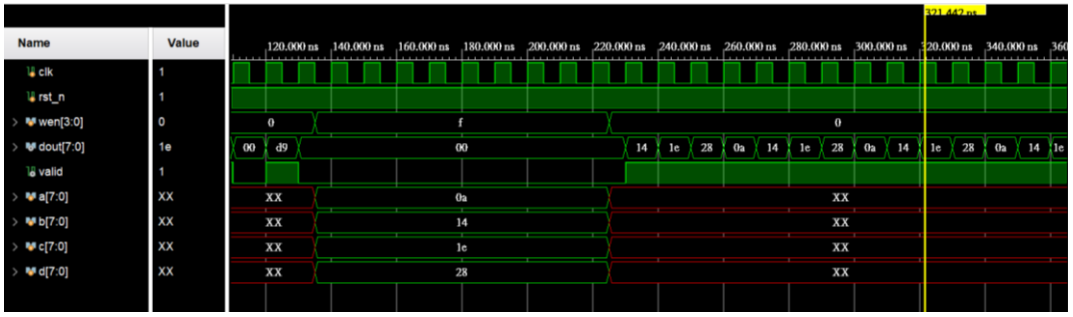
```
assign dout = valid ? Output[fiforead - 1'b1] : 8'd0;

FIFO_8 fa(clk, rst_n, wen[0], ren[0], a, Output[0], error[0]);
FIFO_8 fb(clk, rst_n, wen[1], ren[1], b, Output[1], error[1]);
FIFO_8 fc(clk, rst_n, wen[2], ren[2], c, Output[2], error[2]);
FIFO_8 fd(clk, rst_n, wen[3], ren[3], d, Output[3], error[3]);

endmodule
```

最後輸出 dout 判斷是否為 valid 可以執行 output，如果是的話就輸出上個 cycle 輪替到的 queue 輸出的 Output，否則 dout = 0。

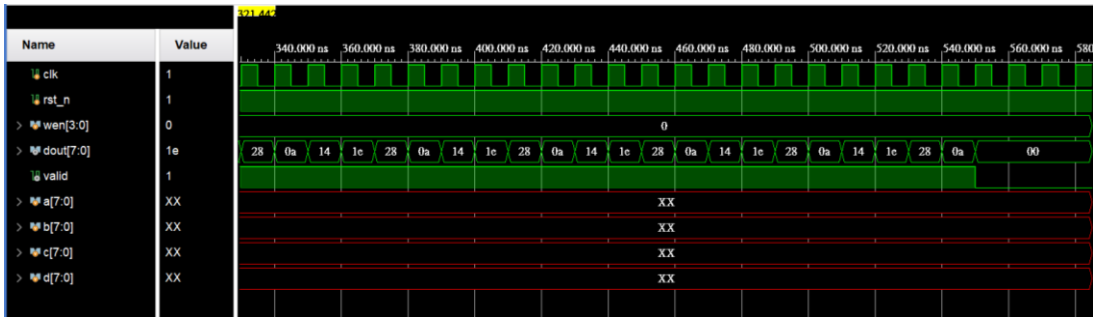
波形圖:



先按照講解 ppt 上的波形範例測試，確認波形無誤



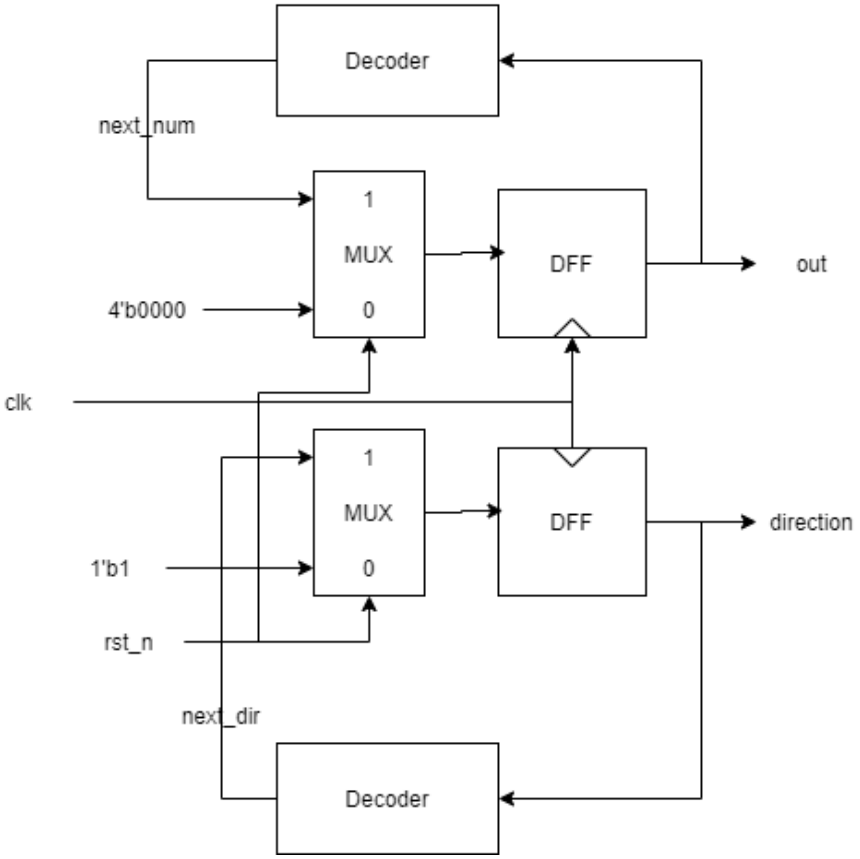
再來把 4 個 queue 清空，然後再把 4 個 queue 寫滿，並嘗試在每個 queue 寫入第 9 組數字，測試讀取的情況



確認每個 queue 只能存取並讀出 8 筆資料，並確認在 queue 清空後 valid = 0、dout = 0。

5. 4-bit Parameterized Ping-Pong Counter with max and min

Block diagram:



4-bit parameterized ping pong counter with max and min 有 6 個 input 和兩個 output。

這題的寫法跟第一題的 ping pong counter 很類似，是由三個部分組成的。2 個 sequential 電路和 2 個 combinational 電路組成。

其中一個 combinational 電路是用來計算下一個 direction 是什麼值(code 中的變數為 next_dir)。因為 input 的值蠻多的，所以我們

必須先弄清楚什麼 input 會改變 next_dir。首先是 enable，enable==1’b1 才會進行運算，再來是 max > min 的條件，接著是 min <= out <= max，最後是再判斷是否有 flip。有 flip 的話，next_dir 要跟現在的 direction 反向，沒有 flip 的話，要判斷 out 的值是否是 max 或是否是 min。是的 max 話，next_dir 要變成 1’b0，而如果是 min 的話，next_dir 要變成 1’b1。上述所說的是 next_dir 會改變值，所以我們知道只要有其中一個條件不符合，next_dir 就會維持現在 direction 的值(也就是 next_dir = direction)

```
41
42  always@(*)begin
43      if(enable == 1'b1)begin
44          if(max > min)begin
45              if(out > max || out < min) next_dir = direction;
46              else begin
47                  if(flip == 1'b1) next_dir <= ~direction;
48                  else begin
49                      if(out == min) next_dir = 1'b1;
50                      else if(out == max) next_dir = 1'b0;
51                      else next_dir = direction;
52                  end
53              end
54          end
55          else next_dir = direction;
56      end
57      else next_dir = direction;
58  end
59
60  endmodule
```

另一個 combinational 電路是用來計算下一個 out 是什麼值(code 中的變數為 next_num)。跟計算 next_dir 很類似的地方是條件很多地方是一樣的。首先是 enable，enable==1’b1 才會進行運算，再

來是 max > min 的條件，接著是 min <= out <= max，最後是再判斷 next_dir 的值，如果 next_dir 是 1’b1，next_num 加 4’b0001，如果 next_dir 是 1’b0，next_num 減 4’b0001。上述所說的是 next_num 會改變值，所以我們知道只要有其中一個條件不符合，next_num 就會維持現在 direction 的值(也就是 next_num = out)

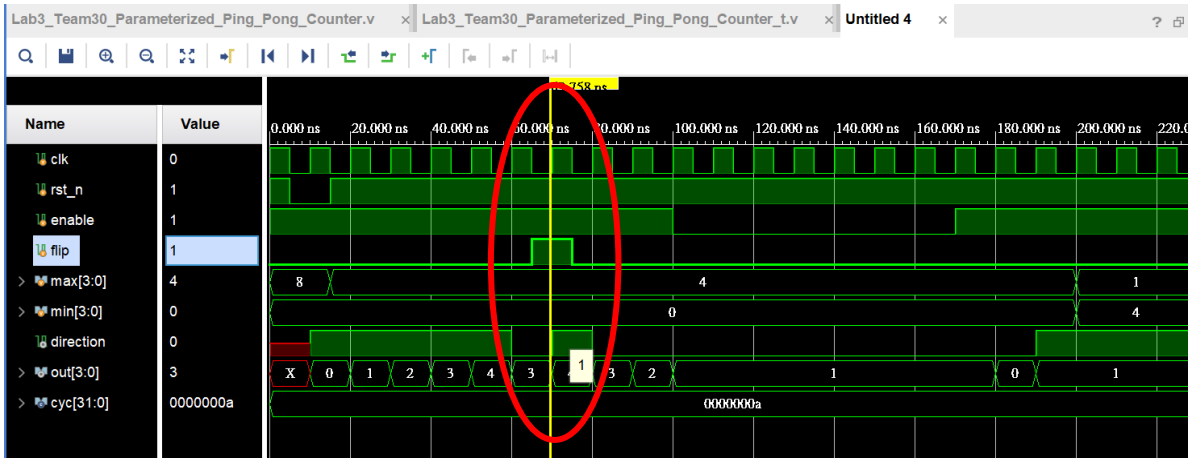
```
27
28      always@(*)begin
29          if(enable == 1'b1)begin
30              if(max > min)begin
31                  if(out > max || out < min)next_num = out;
32                  else begin
33                      if(next_dir == 1'b1) next_num = out + 4'b0001;
34                      else next_num = out - 4'b0001;
35                  end
36              end
37              else next_num = out;
38              end
39          else next_num = out;
40      end
```

2 個 Sequential 電路是個自用來儲存並更新 out[3:0]和 direction 的值。電路中 rst_n 需要是 synchronize，在 clk 的 positive edge 上 rst_n==1’b1 時，更新 direction 和 out 的值，在 rst_n==1’b0 時，初始化 direction=1’b1 out[3:0]==4’b0000。

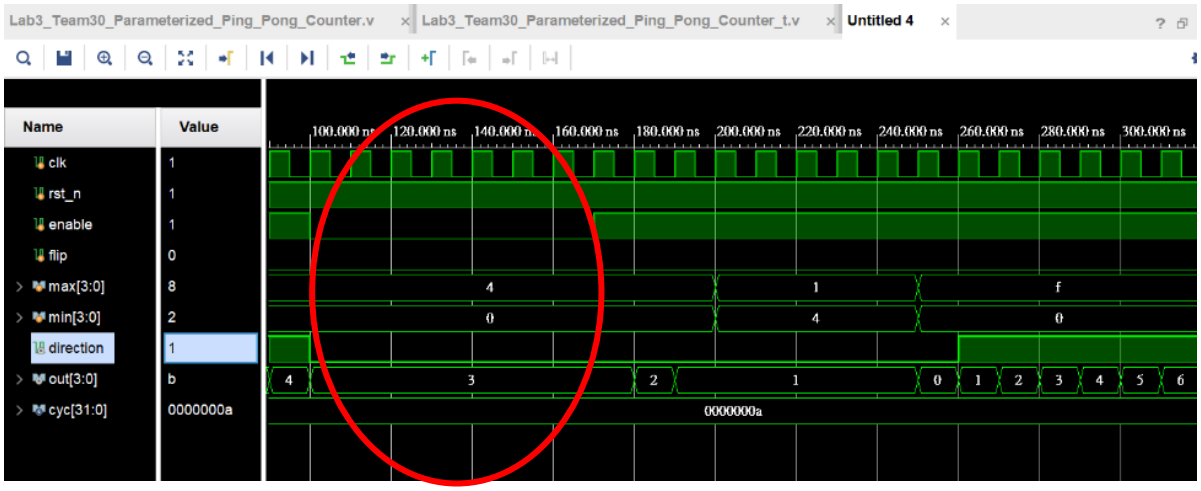
```
16
17      always@(posedge clk)begin
18          if(rst_n == 1'b1)begin
19              out <= next_num;
20              direction <= next_dir;
21          end
22          else begin
23              out <= min;
24              direction <= 1'b1;
25          end
26      end
```

波形圖：

下圖是當 enable == 1’b1，max > min、min <= out <= max 時 flip==1’b1 的波形圖。因為 spec 上說 flip 是 one cycle in length，所以當 flip==1’b1 時一定會正緣觸發，direction 在這個 cycle 跟前一個是反向的。

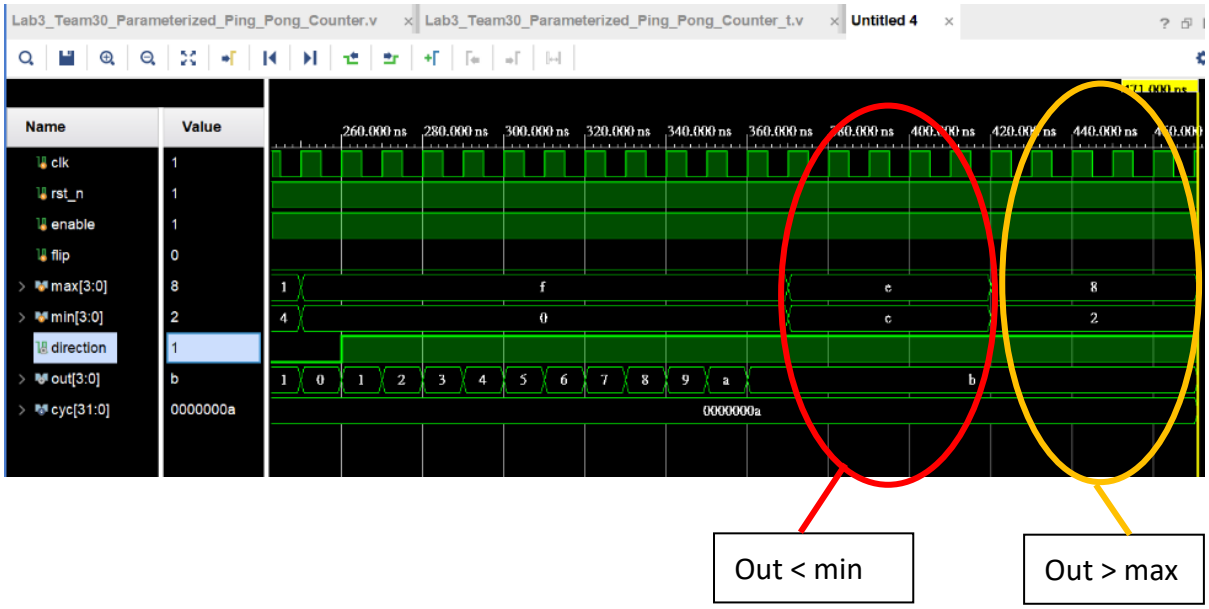


下圖是 enable==1’b0 時的波形圖，可以發現在 enable == 1’b0 時，direction 和 out 都維持不變。



下圖是 out> max 和 out < min 時的情形，可以發現不管是 out <

max 或 out < min，out 和 direction 都會維持不變。



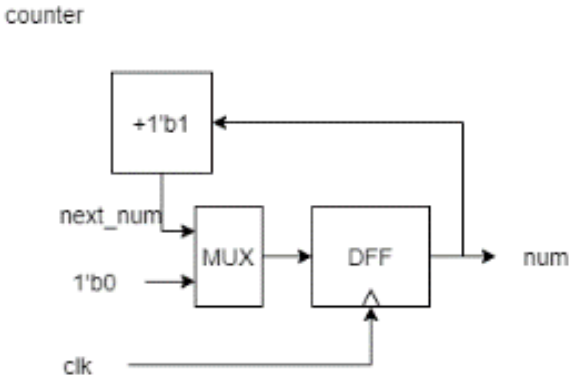
6. fpga 題

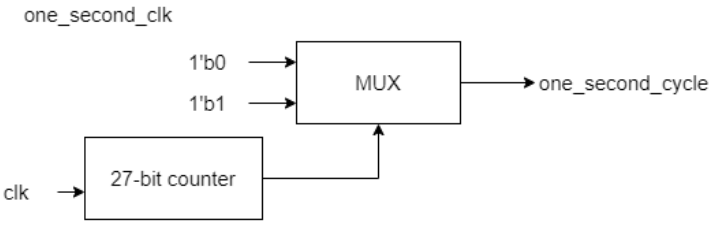
Fpga 題是第 5 題的延伸，但跟第 5 題比較不一樣的是，fpga 的 clock 跟 7 段顯示的 clock 是不一樣的，這樣才有數字 1 秒 1 秒累加，1 秒 1 秒遞減的效果。因此，我造了另一個除頻的 clock(每 1 秒 1 個 cycle)來實作這題。

Fpga 題主要有 7 個 submodule，1 個 clock divider、2 個 debounce curcuit、2 個 one pulse circuit、1 個 7 段顯示器的 module 和改良過後的 parameterized ping pong counter。

Clock divider:

Block diagram:





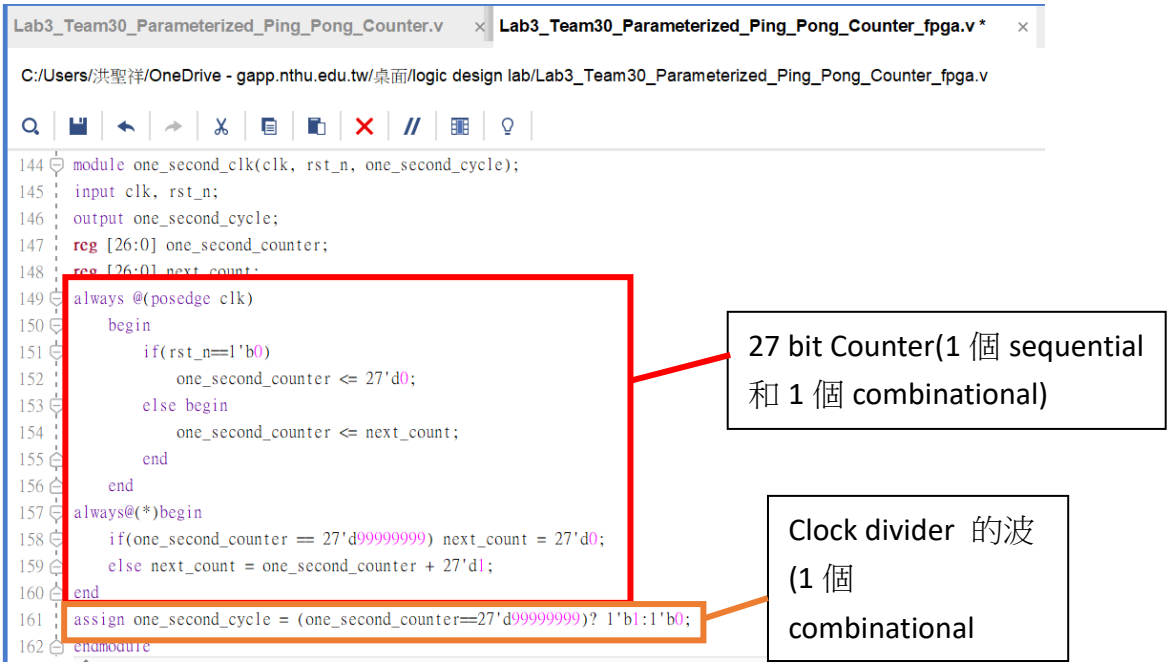
Clock divder 的作用是實作一個用 fpga 的 clock 除頻過後變成 1 秒的 clock。因為 fpga 的 clock 是 100MHz，也就是週期 10 的-8 次方秒。所以我就用 1 個 sequential circuit 和 2 個 combinational Circuit 來做(也就是 1 個 counter 和 1 個 combinational)

其中一個 combinational 電路是用來計算 counter 的下一個值是什麼(code 中的變數為 next_count)。如果 one_second_counter==10 的 8 次方-1，next_count 加歸 0，反之，one_second_counter 加 1

Sequential curcuit 用來儲存更新 one_second_counter 的值。如果 rst_n==1’ b0，one_second_counter 等於 0，反之，更新 one_second_counter 的值(one_second_counter = next_count)

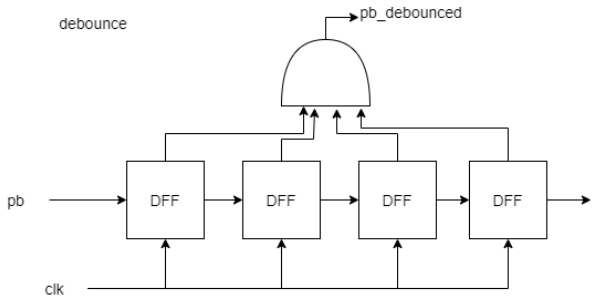
另 1 個 combinational curcuit 作用是來形成除頻過後的波型(變數

為 reg one_second_cycle)。當 one_second_counter 等於 10 的 8 次方減 1，one_second_cycle 等於 1 反之為 0。



Debounce circuit:

Block diagram:



Debounce circuit 由 1 個 combinational circuit 和 4 個 sequential circuit (4 個 D flip flop)所組成。

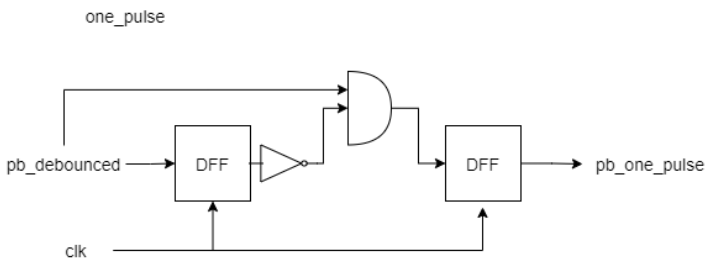
Combinational circuit 來判斷 4 個 d flip flop 是否都是 1，如果是 output pb_debounced 等於 1 反之為 0。

Sequential circuit 來儲存更新 d flip flop 的值，每一個 clock cycle 的 positive edge，input pb 餵給第一個 d flip flop，第一個 d flip flop 的值餵給第二個 d flip flop，依此類推。

```
129
130 module debounce (pb_debounced, pb, clk);
131 output pb_debounced;
132 input pb;
133 input clk;
134
135 reg [3:0] DFF;
136 always @(posedge clk)
137 begin
138     DFF[3:1] <= DFF[2:0];
139     DFF[0] <= pb;
140 end
141 assign pb_debounced = ((DFF == 4'b1111) ? 1'b1 : 1'b0);
142 endmodule
143
```

One pulse circuit:

Block diagram:



One pulse circuit 的作用是要 output 一個 clock cycle 的波。

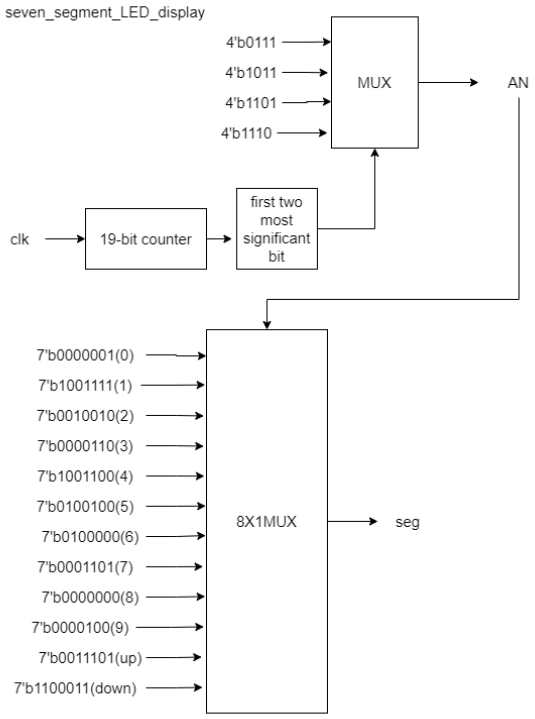
Input 是 debounced 過後的波和 clock。

```
239 module onepulse (PB_debounced, CLK, PB_one_pulse);
240 input PB_debounced;
241 input CLK;
242 output PB_one_pulse;
243 reg PB_one_pulse;
244 reg PB_debounced_delay;
245 always @(posedge CLK) begin
246     PB_one_pulse <= PB_debounced & (! PB_debounced_delay);
247     PB_debounced_delay <= PB_debounced;
248 end
249 endmodule
```

7 segment display module:

（modulename:seven_segment_LED_display）

Block diagram:



7 segment display module input 有 clk, rst_n, out, direction, output 有 AN, seg。fpga 上有 4 個 7 段顯示器，但因為 4 個 7 段顯示器都是由同一個 7 bit 訊號來傳，導致如果我們把 4 個 7 段顯示器都打開，4 個顯示器都會呈現一樣的數字。因此我們利用視覺暫留，每 1 毫秒交錯顯示第一個顯示器、第二個顯示…。Module 裡面有 counter, mux。

首先，我先製造一個 counter(fpga 的 clock 週期是 10^-8sec，所

以 10^8 個 fpga 的 clock 是 1ms)名叫 refresh_counter，counter 的實作跟前面一樣。因為 2^17 大約是 10^5，當我們數 2^17 個 clock，時間是 1ms，這正是我們要的秒數。我先宣告一個 19bit 的 register 最前面的 2 個 most significant bit 可以用來選擇不同的顯示器。

```
174
175 always@(posedge clk)begin
176     if(rst_n == 1'b0)begin
177         refresh_counter = 19'b0000000000000000000;
178     end
179     else begin
180         refresh_counter <= next_refresh_count;
181     end
182 end
183 assign next_refresh_count = refresh_counter + 19'b0000000000000000001;
184 assign activating_LED = refresh_counter[18:17];
185
```

再來，我們可以實作一個 mux 來以前面 2 個 bit 來選擇不同的顯示器。2’b00 代表第 1 個顯示器(AN = 4’b0111)，2’b01 代表第 2 個顯示器(AN = 4’b1011)，2’b10 代表第 3 個顯示器(AN = 4’b1101)，2’b11 代表第 4 個顯示器(AN = 4’b1110)

```
186 always@(*)begin
187     case(activating_LED)
188     2'b00:begin
189         AN = 4'b0111;
190     end
191     2'b01:begin
192         AN = 4'b1011;
193     end
194     2'b10:begin
195         AN = 4'b1101;
196     end
197     2'b11:begin
198         AN = 4'b1110;
199     end
200     default:begin
201         AN = 4'b1110;
202     end
203     endcase
204 end
```

最後，我們可以再實作一個 mux 來判斷 seg[7:0]的值是什麼。

當 out>=10(in decimal)，第一個顯示器要顯示 1，反之，顯示 0。

接下來決定第二個顯示器要顯是哪個數字(個位數)，第三和第四個顯示器要決定要顯是哪個符號網上的符號或往下的符號(direction 決定)

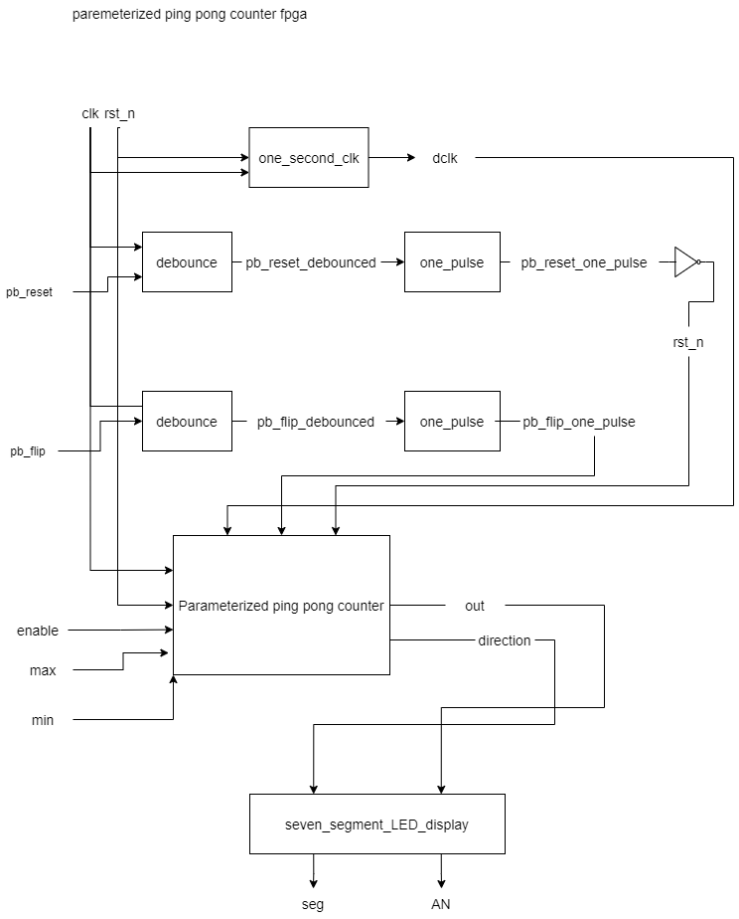
C:/Users/洪聖祥/OneDrive - gapp.nthu.edu.tw/桌面/logic design lab/Lab3_Team30_Parameterizec

```
always@(*)begin
    if(AN == 4'b0111)begin
        if(out >= 4'b1010)seg = 7'b1001111; //1
        else seg = 7'b0000001;//0
    end
    else if(AN == 4'b1011)begin
        if(out == 4'b0000 || out == 4'b1010)seg = 7'b0000001;//0
        else if(out == 4'b0001 || out == 4'b1011)seg = 7'b1001111; //1
        else if(out == 4'b0010 || out == 4'b1100)seg = 7'b0010010;//2
        else if(out == 4'b0011 || out == 4'b1101)seg = 7'b0000110;//3
        else if(out == 4'b0100 || out == 4'b1110)seg = 7'b1001100;//4
        else if(out == 4'b0101 || out == 4'b1111)seg = 7'b0100100;//5
        else if(out == 4'b0110)seg = 7'b0100000;//6
        else if(out == 4'b0111)seg = 7'b0001101;//7
        else if(out == 4'b1000)seg = 7'b0000000;//8
        else seg = 7'b0000100;//9
    end
    else if (AN == 4'b1101 || AN == 4'b1110)begin
        if(direction == 1'b1)begin
            seg = 7'b0011101;//up
        end
        else if(direction == 1'b0)begin
            seg = 7'b1100011;//down
        end
        else seg = 7'b1111111;
    end
    else seg = 7'b1111111;
end

endmodule
```

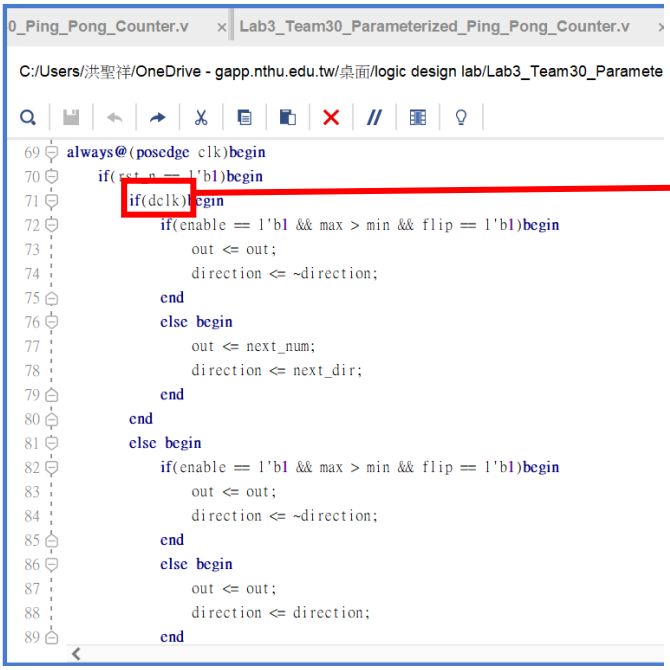
Top module:

(Lab3_Team30_Parameterized_Ping_Pong_Counter_fpga)

Block diagram:



這裡的 parameterized ping pong counter 跟第 5 題有點不一樣，這裡的 parameterized ping pong counter 可以吃進 clock divider 的 clock。我們希望數字在顯示時，變動的數字是每 1sec 改變一次。而且我們希望按下按鈕時可以馬上改變數字的顯示或上下符號，所以也要吃進原來 fpga 的 clock(10^-8sec)。只要在原本的 code 的 always block 中加入如果 dclk==1’b1，out 和 direction 會更新，如果 dclk==1’b0，out 和 direction 不變。



dclk 是除頻過後的 clock，如果 dclk==1'b1，代表要更新 out 和 direction 的值

7. 心得

洪聖祥: 這次 lab 花了很久的時間分清楚哪些地方是 sequential circuit 那些地方是 combinational circuit。尤其是 fpga 題，要處理 button 的問題還有 fpga 題在處理 clock 上需要有點技巧。因為一個 clock 是 fpga 的 clock，還有一個 clock 是顯示數字更新時的 clock，讓我花了蠻多時間處理這個問題。在經過一次次的嘗試後終於可以符合 spec 的要求完成。雖然我每次都說希望下次 lab 可以更得心應手，但經過一次又一次的 lab，每次題目越來越難，反而越花越多時間，寫 verilog 的思維也跟 C 不一樣。但我相信花這

些時間絕對是值得的。

劉奇泓:這次的 lab 第一次開始寫 seqential circuit，我花了蠻長的時間弄懂 seqential circuit 的時序問題，尤其是在第三題連接多層的 submodule 時花了我蠻長的時間 debug，之後才發現問題是在我在上層的 module 寫 always@(posedge clk)時出現了底層與上層有時間差的問題，也讓我知道之後有遇到類似的情況要更加謹慎。希望這些經驗都能累積起來，在下次的 lab 提升效率，使我有能力處理更多難關。

8. 分工

洪聖祥:advanced question 1、5、fpga 題，report 1、5、fpga 題

劉奇泓: basic question 2 block diagram ，advanced question 2、3、4，report 2、3、4 題