

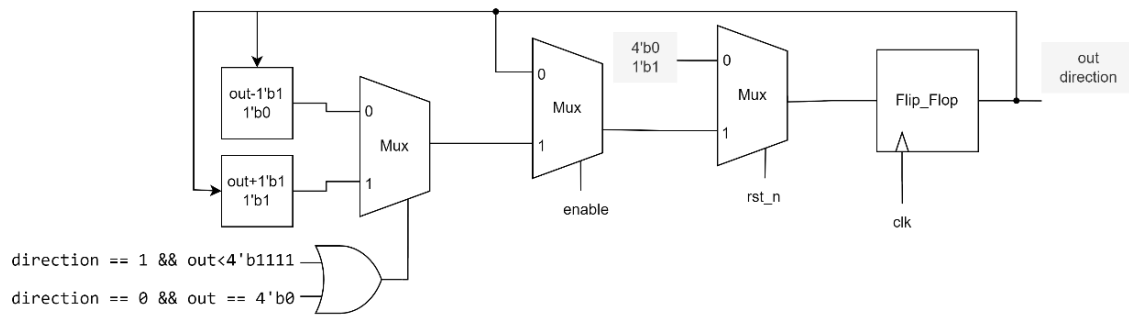
Lab 3 report

組員：110062221 李品萱

110062213 唐翊雯

I. 4-bit Ping-Pong Counter

如下圖，這題我們需要實作一個 Ping-Pong Counter。



首先在 combinational block 的部份我們先用 next_out 及 next_direction 接下一次

clock 起來的時候要傳過 Flip Flop 的值。為了讓這個 Counter counting up 時能做

到 Spec 中數到 15 後將 direction 從 1 變為 0 及 counting down 數到 0 後將

direction 從 0 變為 1，我們觀察到只有 Counting up 的 out 在 0 到 14 的區間內

或 Counting down 的 out 為 0 時，會使 next_direction = 1'b1，next_out =

out+1'b1，因此我們用 if-else 去做判斷，而它 synthesize 的結果即為 Mux。Mux

由左而右是 priority 由低至高，最左邊的 Mux 如上所述，而接下來這個 Mux 則

會藉由 enable 判斷是否要 hold 原值或繼續傳 next_out 及 next_direction；rst_n

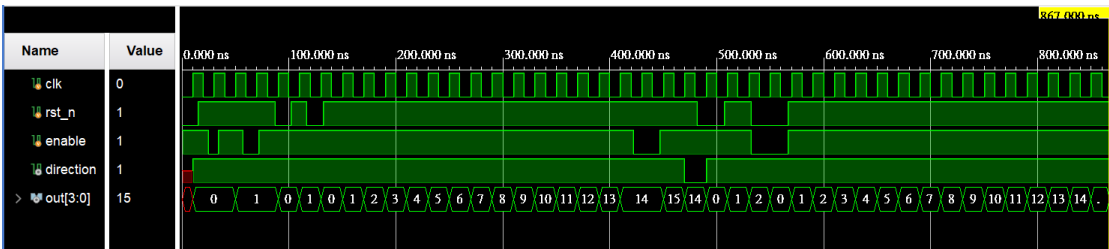
則會判斷是否要將 next_direction 設為 1 及 next_out 設為 0。在以上的

combinational block 做完之後，在每次 clock 起來時這個值就會通過 Flip Flop，

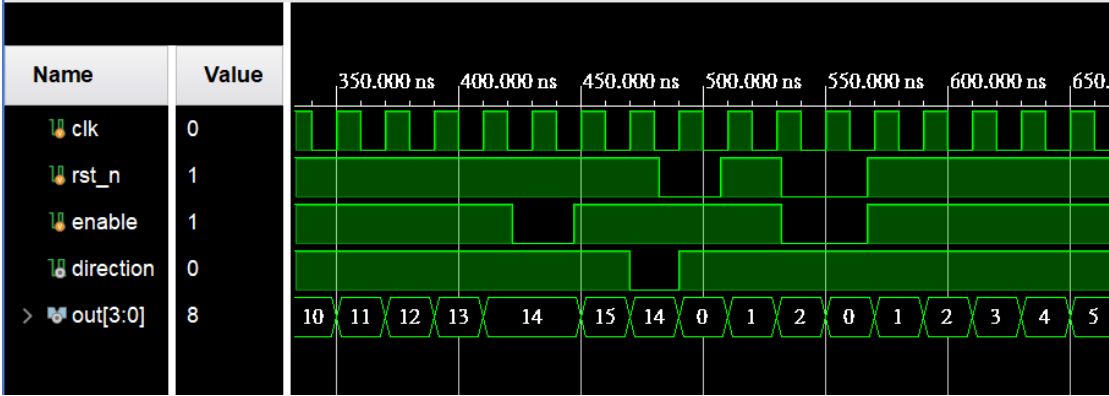
出來的值即為 out 與 direction。簡單來說，combinational block 會負責處理各個

條件下 next_direction 及 next_out 的值，而 sequential block 則負責配合 clock 及 reset 處理最後的 out 及 direction。

在下面的波型圖中我們可以看到，在 clk 起來時才會 trigger 到 rst_n 為 0，在此之前都是 Unknow 的狀態，並將我們的 counter 設定初始值，在 out 為 1 時會造成 hold value 的情形是因為我們的(rst_n, enable) = (1, 0)的結果。

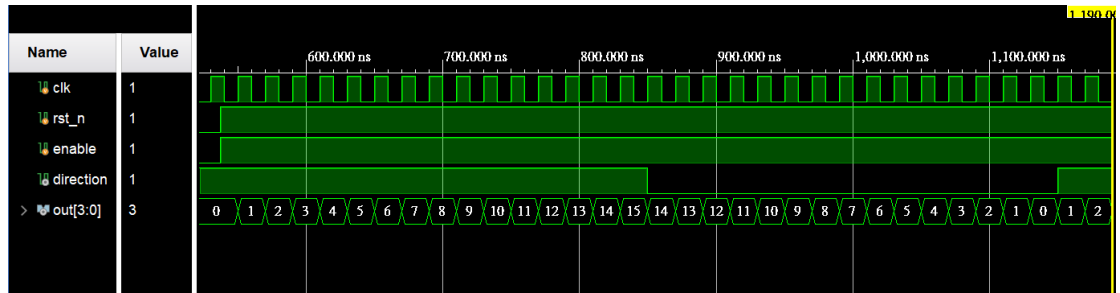


接下來我們在 Counter 運算過程中進行測試(rst_n, enable)為(1, 0), (0, 1), (0, 0)的情況，如下圖。在 out 為 14 後經過一個 clock，enable 為 0 會被 trigger 到，此時如預期的會使 counter hold value，而後由於已經數到 15，direction 會變為 0 並開始往下數；接下來我們讓 rst_n 為 0，clock 起來時 Counter 也確實使 out 回到 0，direction 回到 1；至於(0, 0)的情況則是由於 rst_n 已經是 0，且它的優先權是最高的，因此便會如 waveform 顯示的直接 reset。



下圖我們測試從 0 數到 15 再數回 0 的狀況，確認我們設計的 ping-pong counter

可以正確地執行 count up, count down 及轉換 direction 的動作。



testbench 的設計我們每 10 個單位時間 (配合 timescale 即為 10ns) 會將 clock

反向，藉此產生 clock 的週期變化，其他 input 主要則是使用延遲一段時間去

raise 不同信號確認以上所述的各種情況，部分如下圖。

```
always #10 clk = ~clk;

initial begin
    #15
    rst_n = 1'b1;
    #10
    enable = 1'b0;
    #9
    enable = 1'b1;
    rst_n = 1'b0;
    #8
```

II. First-In First Out (FIFO) Queue

這題要設計一個 FIFO 的環狀 Queue，支援三種操作：寫入新的元素、讀取並 pop 掉最早寫進的值、初始化整個 queue。我們另外開了三個 reg：head、rear、ct，來記錄這個 queue 的資訊。head 會記錄當前最早寫進來的元素在 queue 中的位置；rear 會記錄下一個元素被寫進來時應該要儲存的位置，意即當前 queue 中最後一個元素的下一個位置；ct 則是當前 queue 中的元素數量。

對於寫入新的元素這個操作，我們的實作方法如下：首先檢查 queue 是否已滿，如果 $ct == 8$ ，代表元素的數量恰等於 queue 的 size，這時將無

法再寫入元素，須將 `error` 設為 `1'b1`。否則代表 `queue` 還未滿，這時我們先將要放進來的元素存在 `queue[rear]` 的位置，再更新 `rear`，讓 `rear = rear + 1'b1`，維持 `rear` 的定義（下一個元素被寫進來時應該要儲存的位置）。最後將 `error` 設為 `0`、並更新 `ct`，讓 `ct = ct + 1'b1`（`queue` 中元素的數量增加 1）。

而對於讀取並 `pop` 掉最早寫進的值這個操作，我們的做法如下：首先檢查 `queue` 是否為空，如果 `ct == 0`，代表元素的數量為 `0`，這時將無法讀取元素，須將 `error` 設為 `1'b1`。否則代表 `queue` 非空，這時我們先將 `queue[head]` 的值傳給 `dout`，再更新 `head`，讓 `head = head + 1'b1`，維持 `head` 的定義（當前最早寫進來的元素在 `queue` 中的位置）。最後將 `error` 設為 `0`、並更新 `ct`，讓 `ct = ct - 1'b1`（`queue` 中元素的數量減少 1）。

對於初始化整個 `queue` 這個操作，我們將 `head`、`rear` 與 `ct` 三者的值都設為 `0`，即完成清空 `queue`、對其初始化的工作。

現在，根據題目給的 `input`，總共會有三種情況，分別對應到這三種不同的操作：

(1) `rst_n == 0`

在這個條件下，根據題目要求，我們執行上述提到初始化 `queue` 的操作，並且將 `dout` 與 `error` 都設為 `0`。

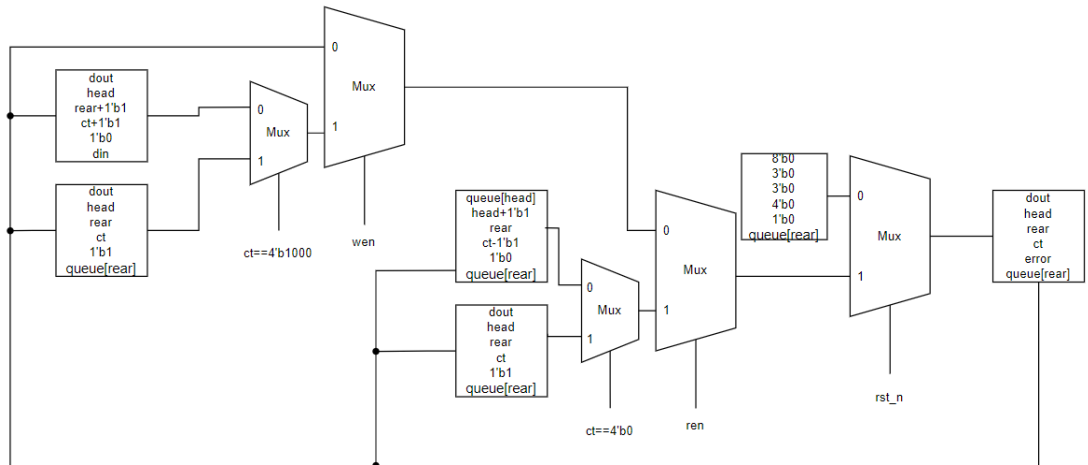
(2) `rst_n == 1 && ren == 1'b1`

在這個條件下，根據題目，我們執行上述提到「讀取並 `pop` 掉最早寫進的值」的操作。

(3) `rst_n == 1 && ren == 1'b0 && wen == 1'b1`

在這個條件下，根據題目，我們執行上述提到「寫入新的元素」的操作。

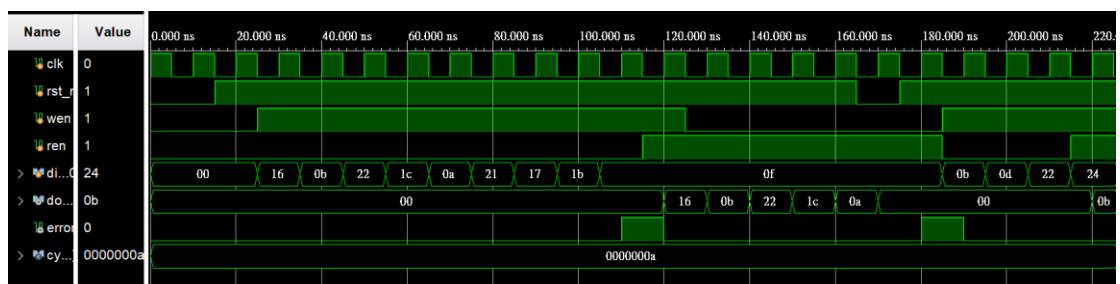
上述的所有實作我們都放在 `always block` 裡面並用 `posedge clk triggered`，並且使用 `non-blocking assignment` 的方法來完成，`block diagram` 如下：



Testbench 的部分，除了最基本的讀取與寫入外，主要要測試的重點有下列幾項：

- reset 後是否有確實將 queue 清空
- queue 為空但仍收到讀取指令時，是否做好 error handle
- queue 已滿但仍收到寫入指令時，是否做好 error handle
- ren 與 wen 同時為 1'b0 時，是否執行 read operation

因此我們在設計 testbench 時，調整參數使以上四種情況都出現，以檢查 code 的正確性：在將 rst_n 設為 1 後，我們先 write 進 9 個數字，確認 queue 已滿但仍收到寫入指令時，error 會被設為 1 且 queue 中的元素不會被更動。接著我們做讀取的動作，確認讀取的順序遵循 FIFO。接下來我們做 reset，將 rst_n 設為 0 再設為 1 並立即做讀取，檢查 reset 後是否有確實將 queue 清空以及 queue 為空但仍收到讀取指令時，是否將 error 確實設為 1'b1 並維持 queue 不被更動。最後再對 queue 做一些讀寫操作確認 reset 後的 queue 確實沒有問題，最後的 waveform 如下：



III. Multi-Bank Memory

這題要求我們使用 basic question 2 實作的 module 來做出 Multi-Bank Memory。我們最一開始直接使用 16 個 basic question 2 的 Memory module 作為此題 module 的 16 個 sub-bank，並將他們由 4'b0000~4'b1111 做編號，而並沒有另外分 bank。然而，這樣做會造成讀取的 index 含有變數，無法好好切分 sequential circuit 與 combinational circuit。因此後來我們更改成先開 4 個 bank 的 module，每一個 bank 的 module 裡面再使用 4 個 basic question 2 的 Memory module 作為 sub-bank。

根據 *ren* 以及 *raddr* 的值，要決定是否做讀取操作以及要對哪一個 bank 做讀取。這裡分成兩個情況：

(1) *ren* == 1'b0

此情況下不需要做讀取操作，我們將 *renb* 設為 4'b0，使所有的 bank 都不做讀取。

(2) *ren* == 1'b1

此情況下須對編號為 *raddr*[10:9] 的 bank 做讀取操作，此時我們應將 *renb*[*raddr*[10:9]] 設為 1'b1，而 *renb* 其他位置的值設為 1'b0。此項操作類似於 decoder，我們可以利用位移運算來達成：

讓 $renb = 1 \ll raddr[10:9]$ ，即可達成前述要求。

與前一段類似，根據 *wen* 以及 *waddr* 的值，要決定是否做寫入操作以及要對哪一個 bank 做寫入。一樣分成兩個情況：

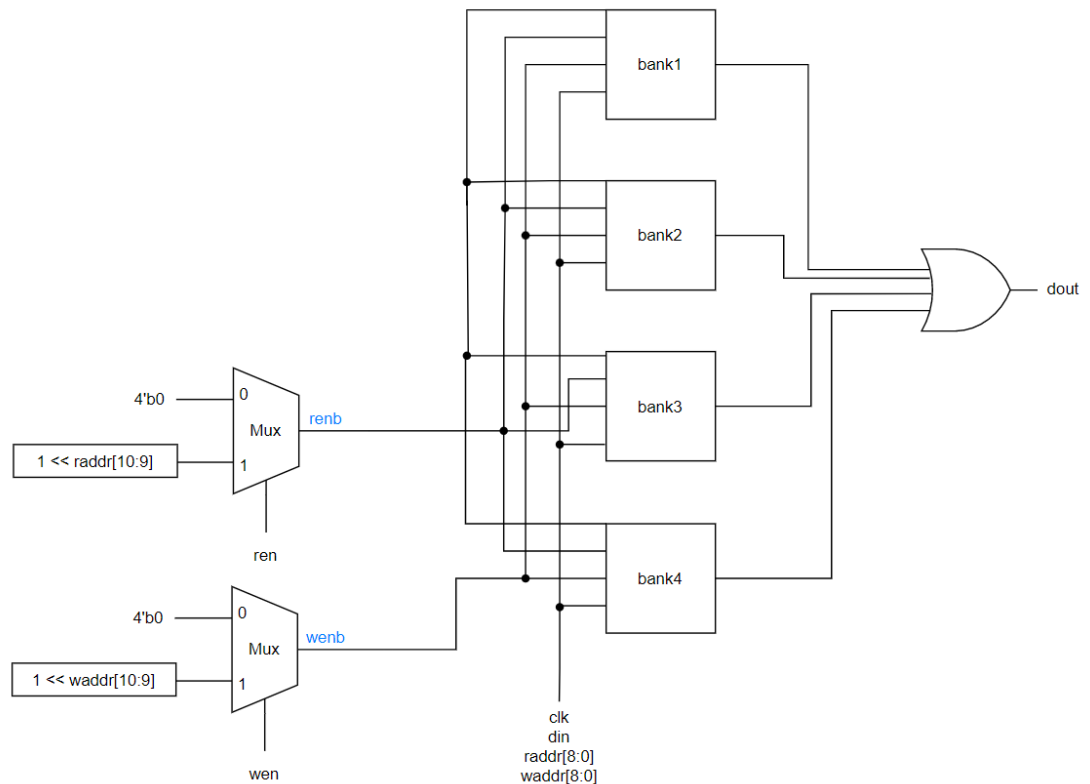
(3) *wen* == 1'b0

此情況下不需要做寫入操作，我們將 *wenb* 設為 4'b0，使所有 bank 都不做寫入。

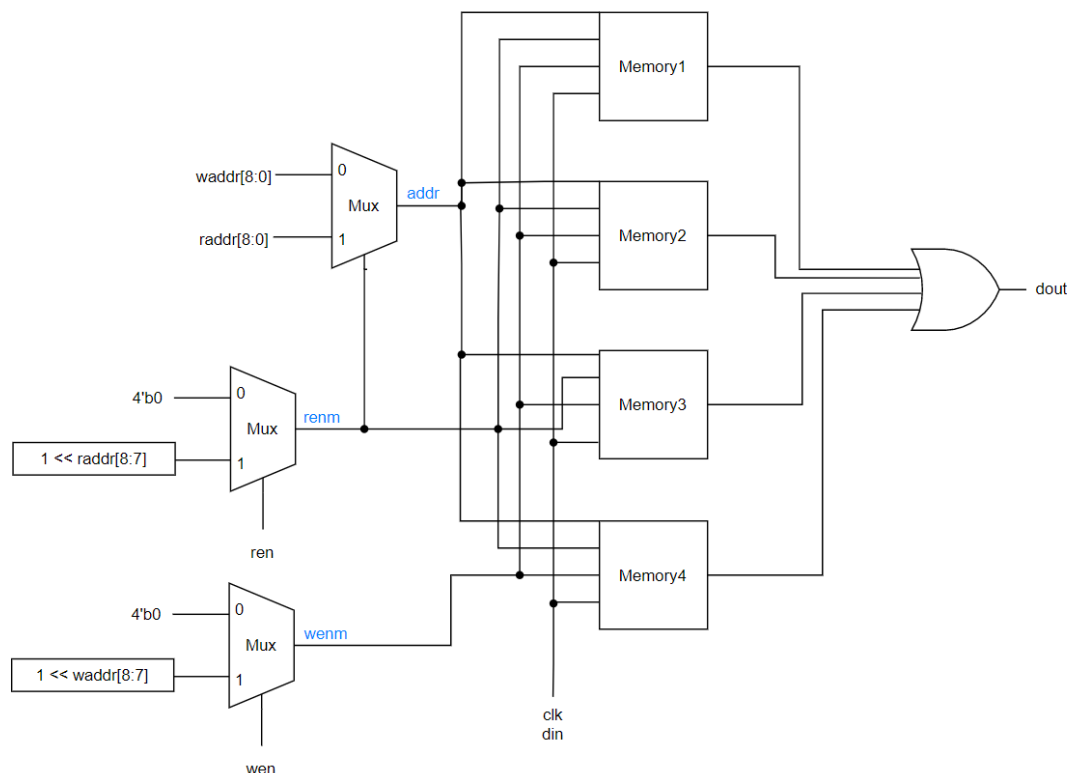
(4) *wen* == 1'b1

根據我們前面的定義，此情況下須對編號為 *waddr*[10:9] 的 bank 做讀取操作，此時我們應將 *wenb*[*waddr*[10:9]] 設為 1'b1，而 *wenb* 其他位置的值設為 1'b0。一樣可以利用位移運算來達成：讓 $wenb = 1 \ll waddr[10:9]$ ，即可達成前述要求。

最後，dout 的值即為所有 bank 的 output 取 bitwise or。以上操作都是用 combinatial circuit 實作，block diagram 如下：



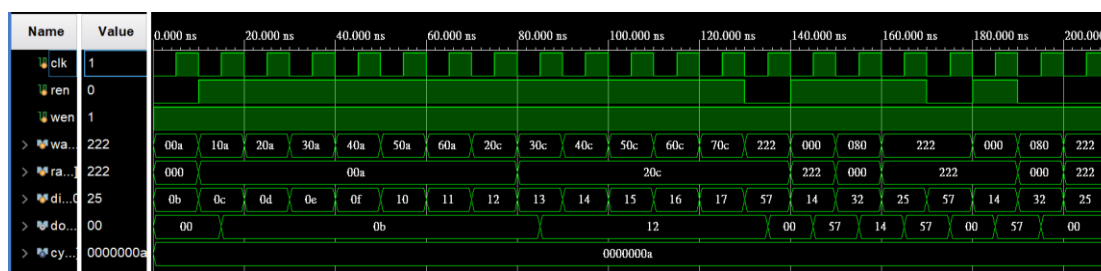
而每個 bank 裡面的實作與前述類似，差別在於對每個 memory module 而言，不再需要給 raddr、waddr 兩個位址，而是根據 ren 以及 wen 的值決定要傳哪一個 address 進去。由於讀取較寫入有較大的優先序，因此我們以 ren 作為判斷，若該 memory 的 $renm == 1'b1$ ，我們便讓 $addr = raddr[6:0]$ ，否則讓 $addr = waddr[6:0]$ 。Block diagram 如下：



testbench 的部分，除了最基本的操作，我們還測試了各種情況，包含：

- 對同一個 sub-bank 同時進行讀取與寫入
- 在 $raddr == waddr$ 的狀況下只寫入不讀取，確認是否寫入成功
- 對同一個位置重複寫入，確認其紀錄的值是否正確

而最終的波形圖如下：

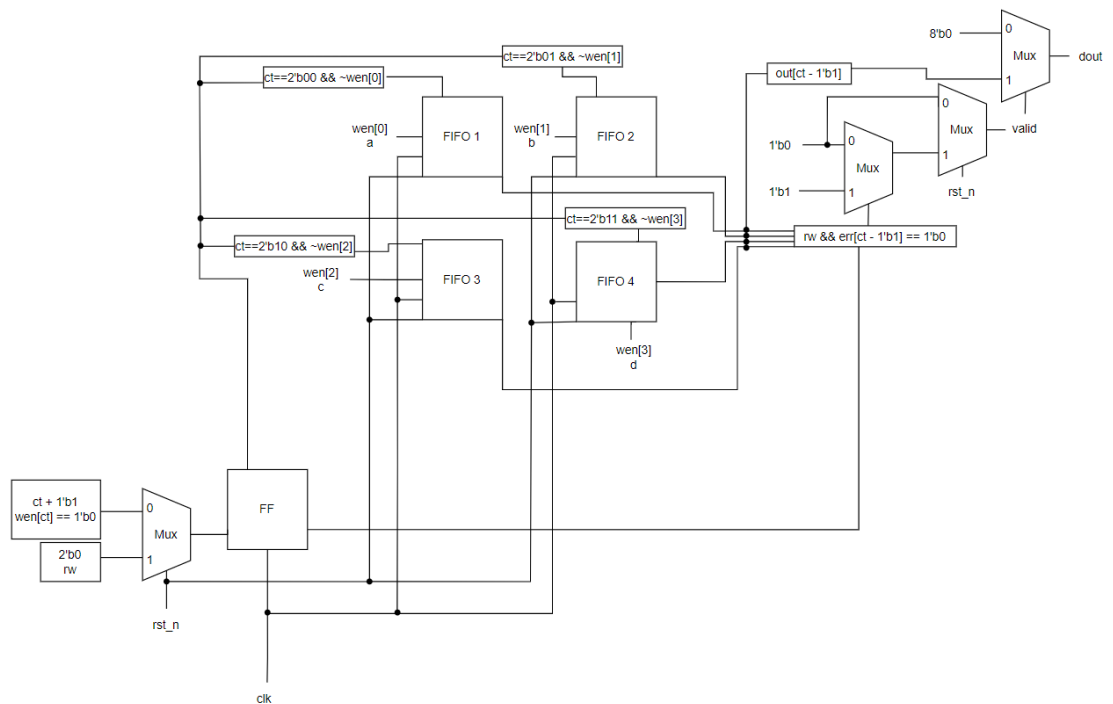


iv. Round-Robin FIFO Arbiter

這題要沿用 advance 2 的 module，實作 Round-Robin FIFO Arbiter。根據題目要求，我們開了 4 個 queue 的 module，並分別幫它們多開了 wire 接它們的 output (dout 與 error)，和多開了 reg 來作為這四個 queue 的 ren。

我們另外開了一個變數 `ct`，來記錄當前讀取到第幾個 `queue`。我們將 `ct` 宣告為 2 個 bits 的 `reg`，這樣在它數完第 3 個 `queue` (0-indexed) 之後再加一，會恰好溢位並回到第 0 個 `queue`，恰符合題目要求。而維護 `ct` 的方法如下：先判斷 `rst_n` 是否為 `1'b0`，如果是的話代表要 `reset`，`ct` 必須重新設為 `2'b0`。否則的話，讓 `ct = ct + 1'b1`，往下一個 `queue` 移動。此外，我們另外維護一個等一下會用到的 `reg`，記錄前一刻是否有在正在讀取的那個 `queue` 中，做寫入的動作。我們將此變數命名為 `rw`，它會與 `combinatinal circuit` 中 `valid` 的值有關。這段操作我們使用 `sequential` 的寫法，寫在 `always block` 內，並用 `posedge clk triggered`，讓 `ct` 能在 `clk` 變動時改值，達到題目的要求。

接著 `combinatinal circuit` 的部分，我們寫了另一個 `always block`，對 `ren`、`vaild` 以及 `dout` 作更新。我們將 `ren` 宣告成 4 個 bits 的 `reg`，分別代表四個 `queue` 的 `ren`。因為題目要求若同時對同一個 `queue` 進行讀取和寫入操作時，應只做寫入而不做讀取，因此這裡 `ren[i]` 的值應該要設為 `ct == i && ~wen[i]`。而 `valid` 的部分，主要與三件事有關：`rst_n` 是否為 0、前一刻是否有寫入（有寫入便不能讀取），以及前一刻若有讀取是否讀取成功。這時判斷就需要用到在 `sequential circuit` 中更新的 `rw`，以得知前一刻是否有寫入。因此考慮 `valid` 的值時，我們先考慮 `rst_n` 是否為 0，若為 0 則 `valid` 為 `1'b0`。若 `rst_n` 不為 0，則 `valid` 的值為 `rw && err[ct - 1'b1] == 1'b0`，這邊注意 `ct` 要減一，因為是判斷前一刻的讀取是否成功。最後 `dout` 的部分，如果 `valid` 為 `1'b0`，則 `dout` 為 `8'b0`，否則 `dout` 的值為 `out[ct - 1'b1]`，一樣因為讀的是前一刻的值所以 `ct` 要減一，整體的 `block diagram` 如下：

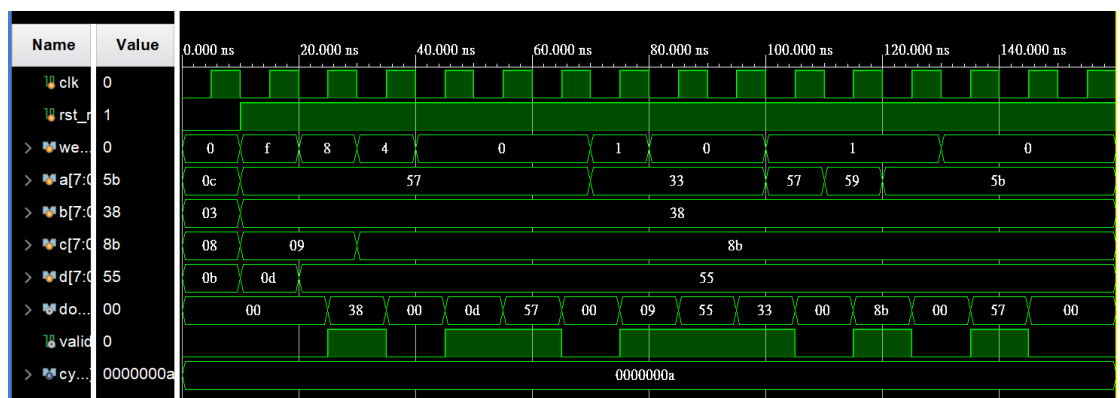


Testbench 的部分，主要需測試的重點有下列幾項：

- 要讀取的 queue 為空時，是否做好 error handle
- 同時寫入與讀取 queue 時，是否只做寫入操作
- 讀取順序是否正確

我們讓以上這幾種狀況都在 testbench 中出現，以驗證 code 的正確性，

waveform 如下圖：

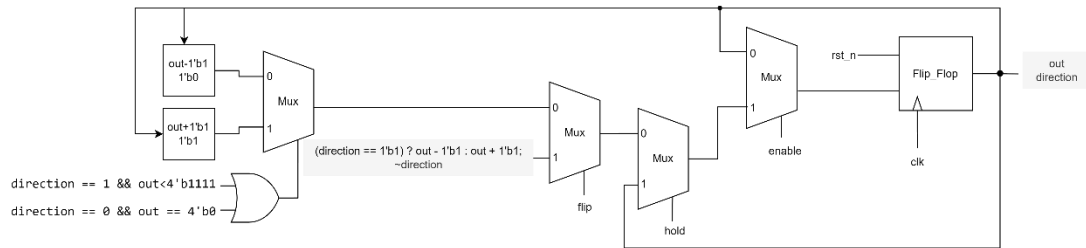


v. 4-bit Parameterized Ping-Pong Counter

這一題與第一題非常相似，差別在於多了 flip、max、min 等 control signal，如

下圖，block diagram 的 hold 如下所示。

```
assign hold = ((max<min)||((out>max)||((out<min)||((max == min) && (min == out)))) ? 1'b1 : 1'b0;
```



與第一題相同的 Mux 這邊就不贅述，我們可以先看以 flip 為 reset signal 的

Mux，flip 為 1 時會將 direction 反向，out count 的方向也會與原本相反；而

hold 我們將這題新增的條件直接以 assign 的方式接起來去判斷是否要維持當前的

的值。經過途中這些 Mux 之後我們就會得到最後的值再通過暫存器輸出。

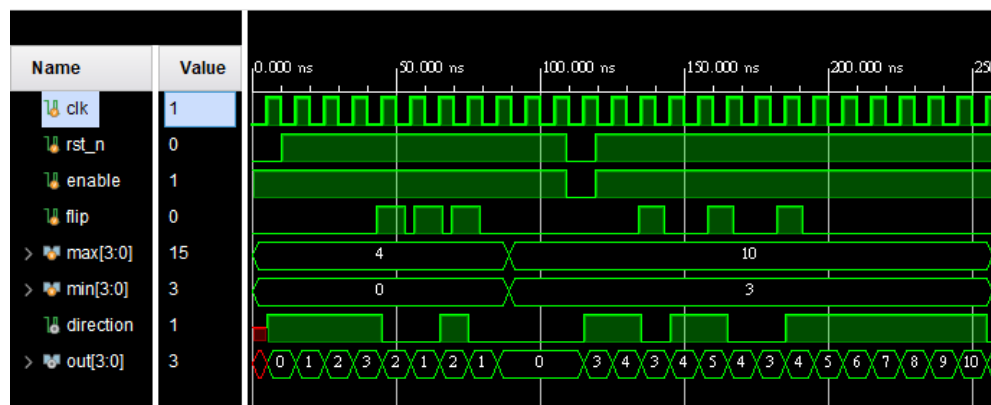
下面的波型圖中，我們先將(max, min)設為(4, 0)，做出與 spec 相同的 waveform

初步確認我們的設計應該是正確的，這邊我們也設計了連續 flip 的狀況，能看到

direction 及 out 都有做出相對的改變。接下來我們將(max, min)設為(3, 10)，

剛 trigger 到新的值的時候，由於 out < min，因此 out 會維持它原先的值一直到

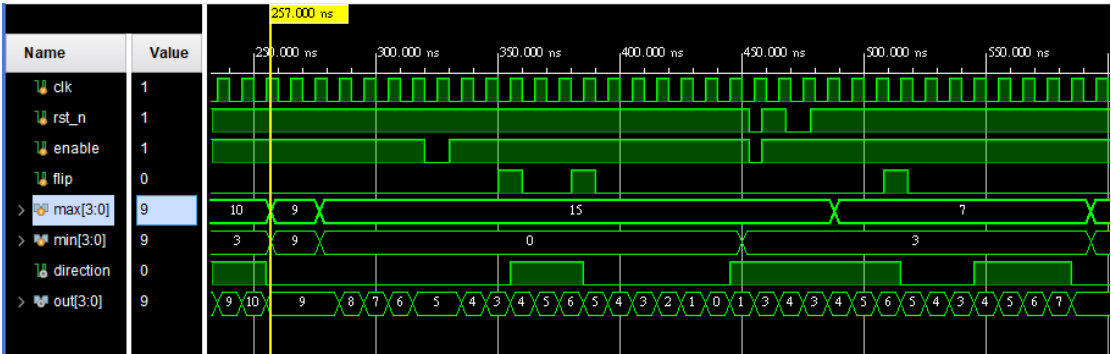
rst_n 為 0 時，direction 為 1，out 被設為 min，它才符合繼續 count 的條件。



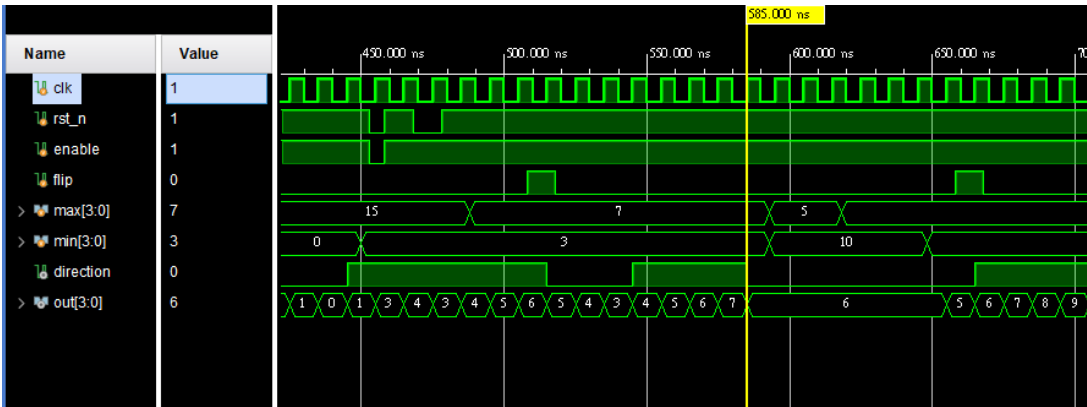
接下來我們測 min==max==output 的情況，counter 此時會維持其原值(out and

direction)，之後我們也繼續測 flip 及 enable，皆符合預期。而由於前面我們都

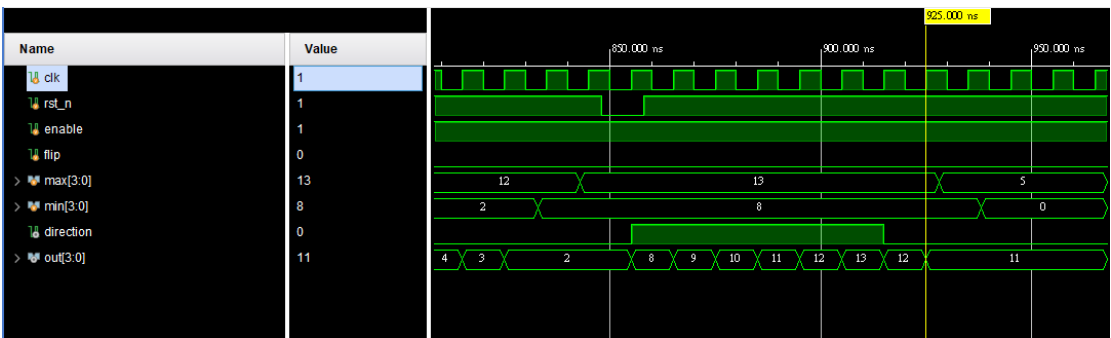
是同時改動 max 及 min，這邊我們試了先改動 min 的 case，也都有出現正常的結果。



接下來我們測 $\text{max} < \text{min}$ ，此時 counter 會維持其當前值。



下圖中我們測 $\text{out} > \text{max}$ ，此時 counter 亦會維持當前值。

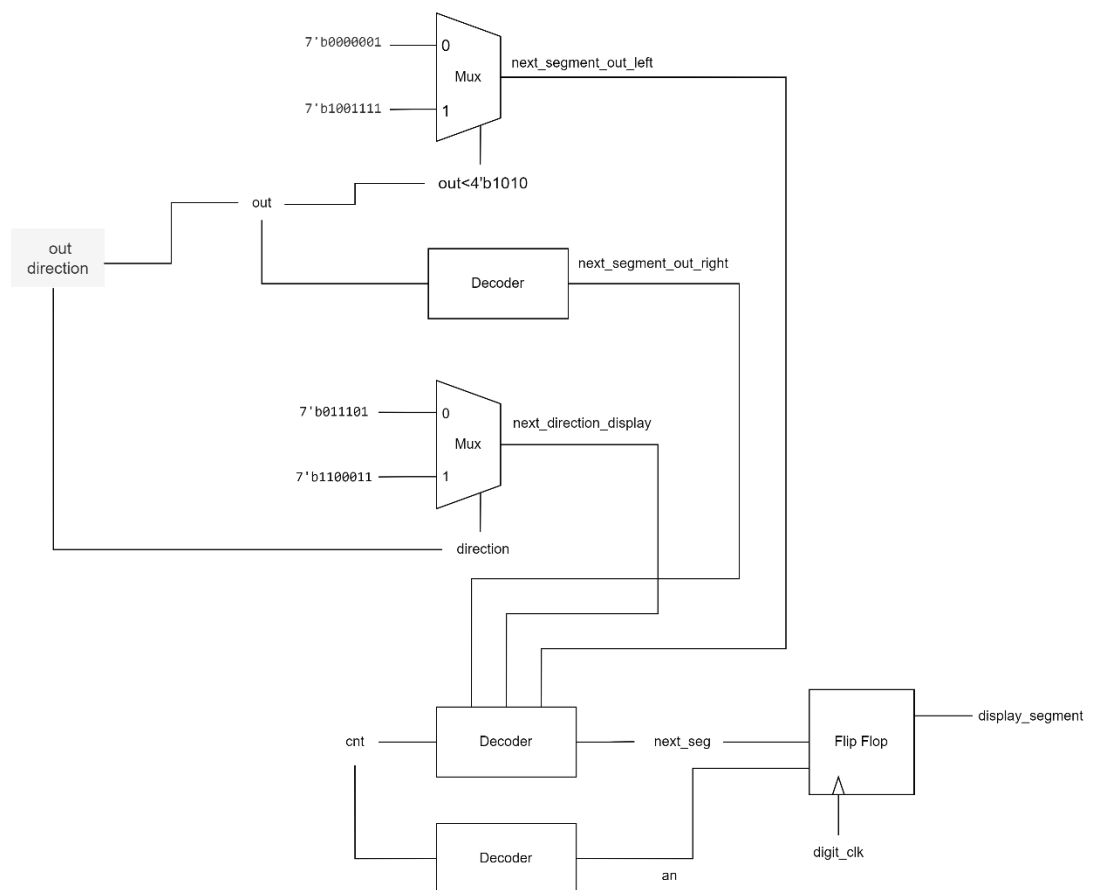


這個 testbench 我們的目的是測試其是否符合 spec 的要求，因此除了檢查 flip

及 enable 是否能正常運作之外，我們主要對特殊狀況都進行測試。

vi. 4-bit Parameterized Ping-Pong Counter on FPGA

如下圖，這題我們將上一題實作的呈現於 FPGA 上，由於上一題已說明過 counter 的 out 及 direction，這邊我們針對 display 的部分進行說明。首先，我們用 Mux 及 Decoder 為 out 對應到它要在 seven segment 顯示的值，Mux 處理的是十位顯示的值及 direction，Decoder 則處理個位數的值。而後，如同上課提過的，seven segment 一次只能顯示一個 digit，因此我們用 cnt 決定它顯示的先後順序，而此處我們用的 clock 是除頻過的 digit_clk，在每個 positive edge digit_clk，我們的 display_segment 就會收到 next_seg 的值，即為它要顯示的值。



◆ Clock divider

由於板子給的 clock 變化很快，為了方便肉眼觀察，我們需要為 clock 做除頻，

我們的作法是將除頻過的 clock 送進負責計算 out 及 direction 的 module。除頻

的部分上課時教授給的建議值是 $1/2^{17}\text{clk}$ ，但我們在操作時發現 $1/2^{24}\text{clk}$ 會

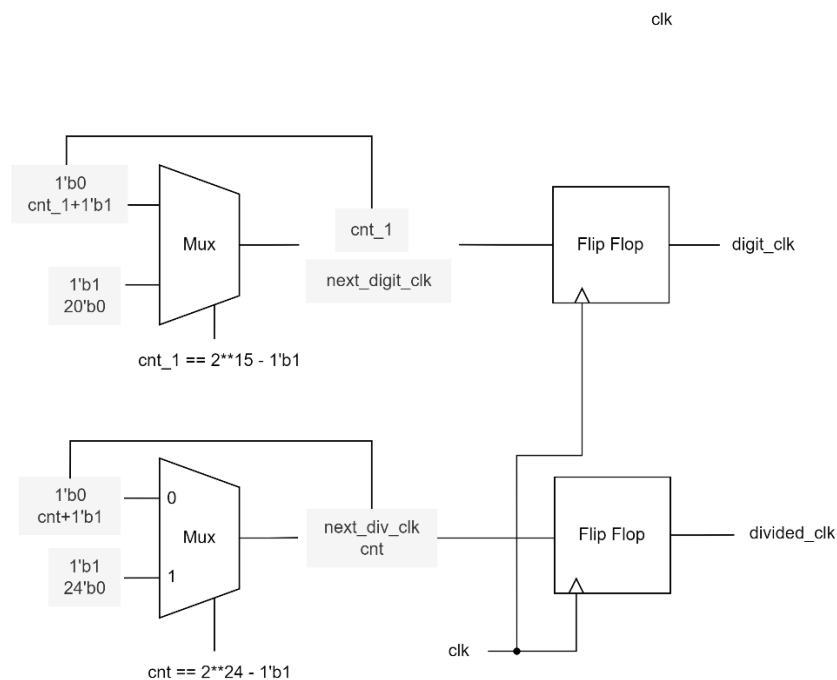
較為剛好，如下圖。我們運用的是在 basic lab 做過的 Clock Divider 加以衍伸，

用 cnt 數我們要在第幾個 clock 時將對應的 divided_clk 做更動。

```
always@(posedge clk)begin
    if(cnt == 2**24 - 1'b1)begin
        cnt <= 24'b0;
        next_div_clk <= 1'b1;
    end
    else begin
        next_div_clk <= 1'b0;
        cnt <= cnt+1'b1;
    end

    if(cnt_1 == 2**15 - 1'b1)begin
        cnt_1 <= 20'b0;
        next_digit_clk <= 1'b1;
    end
    else begin
        next_digit_clk <= 1'b0;
        cnt_1 <= cnt_1+1'b1;
    end
end
endmodule
```

其 block diagram 如下圖。



後來在與另一班同學討論到除頻時，發現我們使用了不同的除頻方式，因此以

下我們也用這個方法做出了相應的設計，code 如下圖。與前一種做法不同的

是，這邊我們直接指定 `divided_clk = cnt[23]`，這個作法的原理是，假設以一個

3bits 的 count 來說，每次加一結果如下。

- 000 -> 001 -> 010 -> 011 -> 100 -> 101 -> 110 -> 111

我們可以觀察到 `count [0]` 每次都會反向，`count [1]` 每兩次反向，`count [2]` 每四

次反向，若配合 clock 則會使 `count [0]` 產生週期為 $1/2\text{clk}$ 的方波，`count [1]` 產

生週期為 $1/2^2\text{clk}$ 的方波，`count [2]` 產生週期為 $1/2^3\text{clk}$ 的方波，由此可知

`count[n]` 會產生週期為 $1/2^{(n+1)}\text{clk}$ 的方波，因此 `cnt[23]` 即為週期 $1/2^{24}\text{clk}$ 的

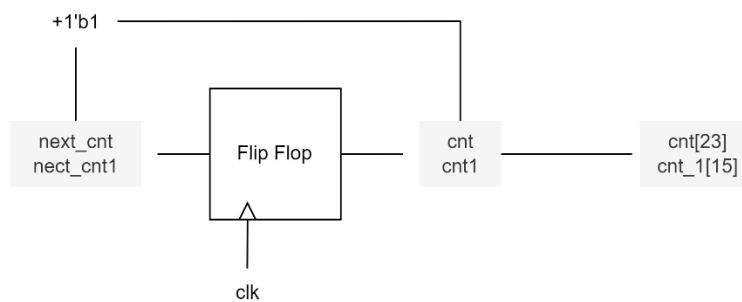
方波，`digit_clk = cnt_1[15]` 也是同樣原理。

```

always@(posedge clk)begin
    cnt<=next_cnt;
    cnt_1<=next_cnt_1;
end
assign next_cnt = cnt + 1'b1;
assign next_cnt_1 = cnt_1 + 1'b1;
assign divided_clk = cnt[23];
assign digit_clk = cnt_1[15];
endmodule

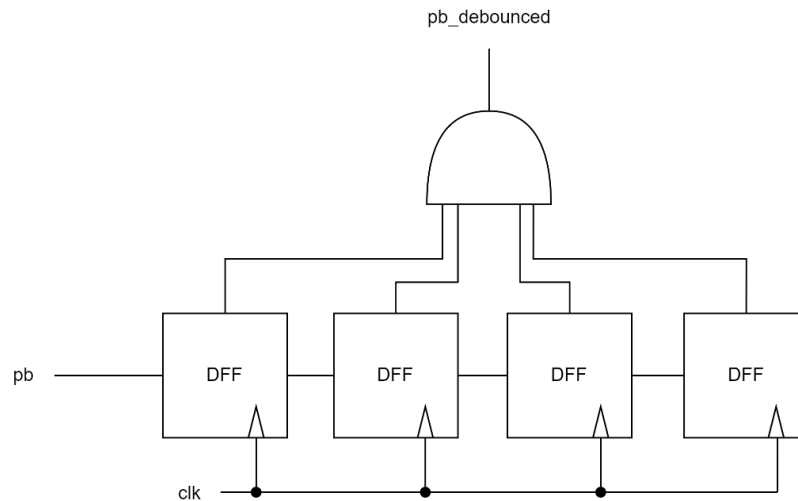
```

其 block diagram 如下圖。

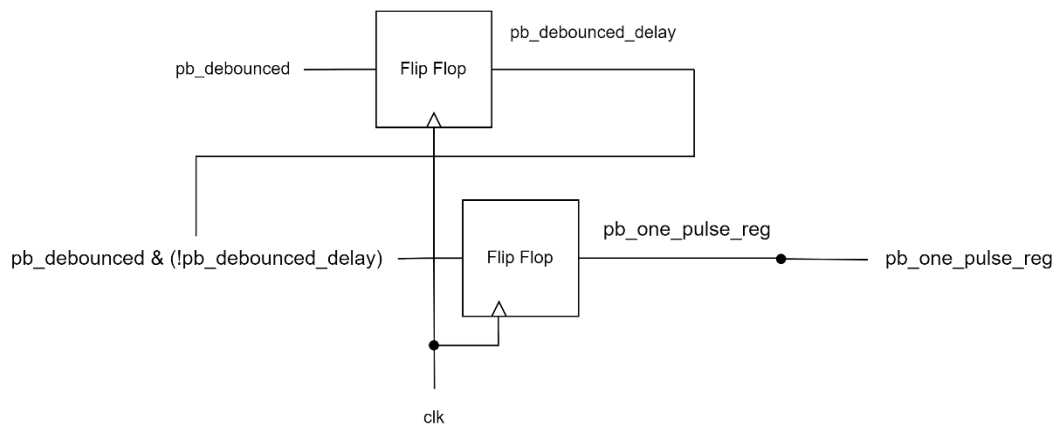


這兩個做法會產生不同的 clock waveform，第一種作法是做出 clock high 的寬度與原本的 clock signal 相同的 clock，第二種作法則會產生類似放大版的 clock 信號，區間為 1 與區間為 0 的寬度相同；但這兩種做法都能產生正確的結果，因為我們只要 divide 過的 clock 為 high 的瞬間，所以只要這個瞬間是正確的，就能使後續觸發的信號正常運作。

◆ Debounce and One pulse



(Debounce)



(One pulse)

另外，我們還需要對 `bottom` 做 `debounce` 及 `one pulse`，這邊上課時都有講解過，所以可以直接實做出來。One pulse Circuit 首先會將 `debounce` 過的結果做延遲一個 `clock` 及反向，而後將其與 `debounce` 的結果 `and` 起來送入 DFF，即會得到結果。這麼做的原因在於 `bottom` 可能被按著不只一個 `clk`，但我們只想要按下去時產生一個 `clk` 的變化。`Debounce` 的部分則是為了去除 `push bottom` 產生的 `glitches`，送出乾淨的信號，因此我們用四個 DFF 過濾掉它，即可完成 `debounce`。

這題我們沿用上一題的 module 並稍作修改，如：在 push reset bottom 時它會給出 1，但我們要的是 reset 為 0 時 reset，因此我們先做 not 再送進上一題的 module。對於這題我們分不同 module 處理 debounce、one pulse、clock divider、計算 out and direction 及 FPGA display，在 display 的部分則寫好 7'b 對應亮哪個 Pin 再將其接好，即可 display 4-bit Parameterized Ping-Pong Counter。

vii. Summary

這次的 Lab 我們接觸了 Clock divider、One pulse、debounce 等新的東西，也學到如何養成好的 coding style。Basic Lab 的時候還不太清楚自己在幹嘛，只是按照 spec 的指示給條件，在進一步寫到 advance 時才比較有概念，但由於接觸的不多，這次我們在實作過程也出現比前幾次都要多的問題，除了需要更仔細的思考 combinational 與 sequential 如何符合我們期待的設計之外，也遇到 vivado 不同的報錯及警告並藉由網路資料解決它，這個過程也使我們去思考自己寫出來的 code 實際上會接出怎樣的電路。

在 Ping-Pong Counter 的部分，我們一開始的寫法是將 rst_n 及 enable 放在 sequential block 內，但會出現 enable 要 hold 值時，我們不太確定是否可以寫 out <= out 這種 code，或者直接將 else 後面空白，但也不確定這樣會產生什麼樣的電路；詢問助教後，我們得到直接將它們放在 combinational block 裡面用 next 值來接的建議，真的大夢初醒，非常感謝助教。

在 FIFO Queue 的部分，我們學到如何實作一個 circular queue。實作過程中需小心的思考如何取值刪值、何時移動 head，何時移動 rear、如何知道 queue 為空以及為滿等等，思考並學習到許多實作上的細節。

而 Multi-Bank Memory 以及 Round-Robin FIFO Arbiter 的部分，最難的是要思考如何好好切割 combinational circuit 以及 sequential circuit。如果沒有

處理好的話，會導致 clk 延遲、在非 posedge 的時後改值等問題。雖然此時 output 看起來大致正確，修正起來卻是浩大的工程，這也提醒了我們先畫圖的重要性。

FPGA 的部分則讓我們進一步思考 clock 在整個過程的運作，也加入了 push bottom 帶來的新觀念，對這些東西也有了更好的理解。

從 gate-level 到這次的 modeling techniques，我們對整個電路有更完整的了解，這次主要是要對 combinational 及 sequential 有較為清楚的概念及養成良好的 coding 習慣，我們也確實在這方面有所收穫，不過或許是慢慢接近期中了，因此近期綜合結果如下圖。



VIII. Contributions

- Code:

I. 4-bit Ping-Pong Counter [By 唐翊雯](#)

II. First-In First Out (FIFO) Queue [By 李品萱](#)

III. Multi-Bank Memory By 李品萱

IV. Round-Robin FIFO Arbiter By 李品萱

V. 4-bit Parameterized Ping-Pong Counter By 唐翊雯

VI. 4-bit Parameterized Ping-Pong Counter on FPGA

By 唐翊雯

● **Report:**

兩人描述各自在 code 部分負責的題目及畫電路圖。