

11010EECS207001Logic Design Lab

LAB3 : Sequential Circuits

TEAM9

組長 : 108062213 顏浩昀

組員 : 106062304 黃鈺舒

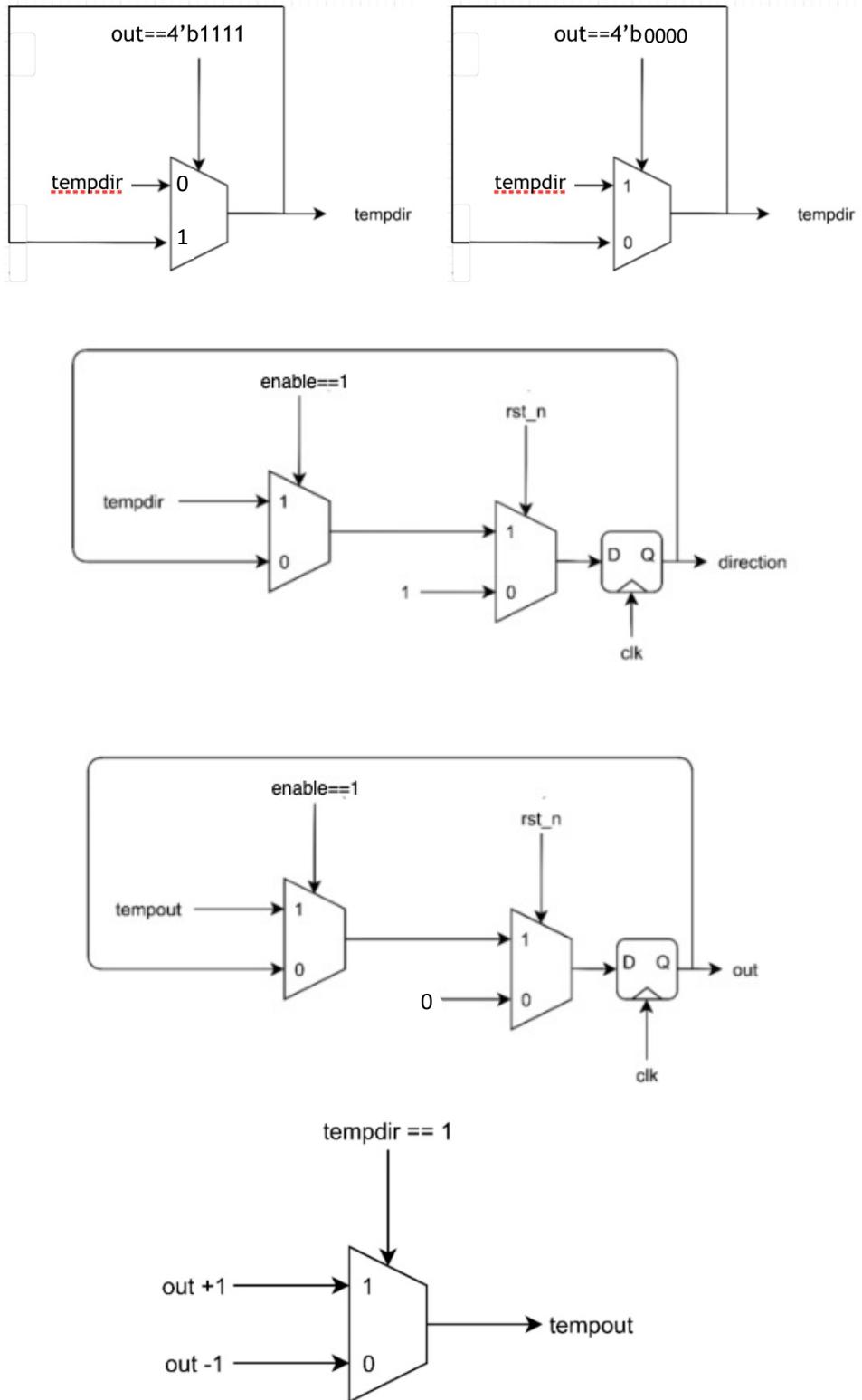
Prof. Chun-Yi Lee

2021.10.28

4-bit Ping-Pong Counter

I. Diagram

(i) 電路設計圖



II. Explanation

一開始rst_n會將out與direction設為初始值，並在每個clk的posedge同時sequentially的以電路圖畫出的方法做運算

- 1.out接收tempdir得到tempout
- 2.得到tempdir與tempout後檢查enable是否通過，在將這兩個值餵給direction,out，也就是說，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。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。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0、1，反之 則檢測enable是否等於1，成立則將tempout的值給out、將tempdir的值給direction。

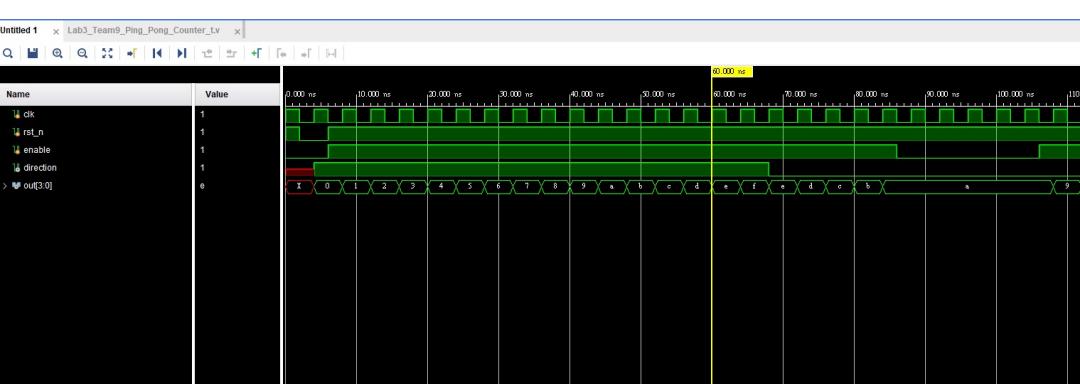
III. Testbench and Waveform

本題testbench分成兩個階段，enable設1之後檢查rst_n為0與1的情況，最後分別檢查enable為0與1的情況。

(下圖為部分實作)

```
initial begin
    //test all Dout 0 to 15
    @(negedge clk);
        max = 15;
        min = 0;
        rst_n = 1'b0;
    @(negedge clk);
        rst_n = 1'b1;
        enable = 1'b1;
        #(`CYC * 30);
```

(下圖為部分波形圖)

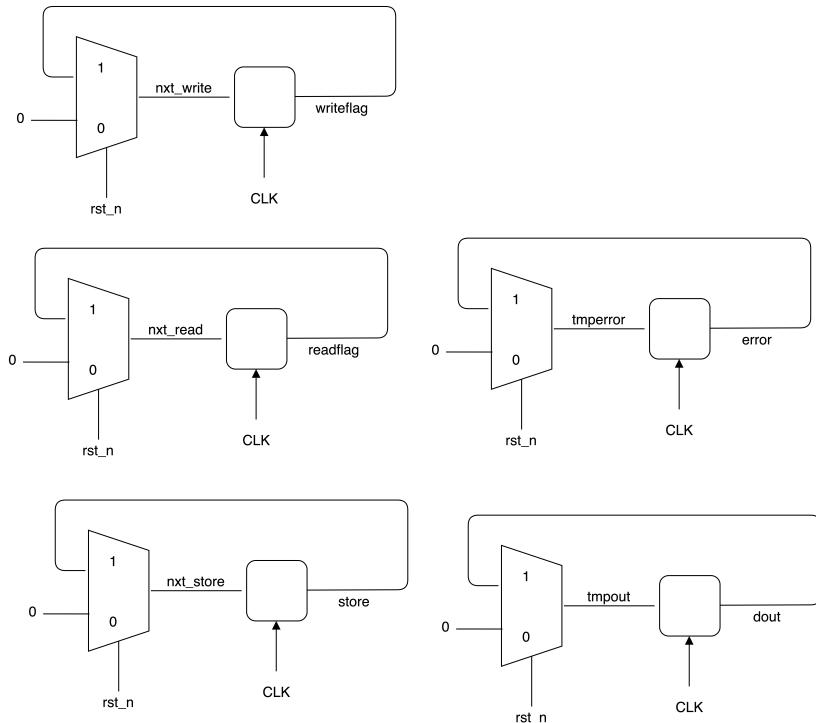


FIFO

I. Diagram

(i) 電路設計圖

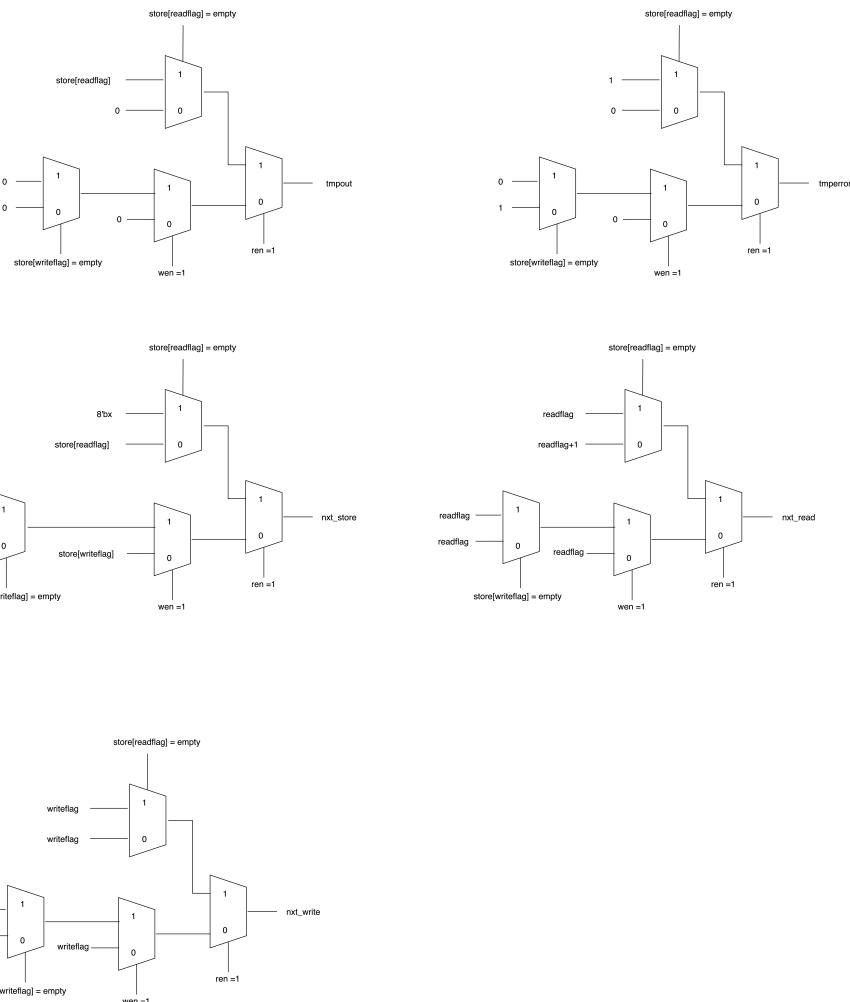
Sequential



I. Diagram

(i) 電路設計圖

combinational



II. Explanation

在sequential的部分，只負責更新error、dout、readflag、writeflag以及store的值。

(下圖為更新方式，省略reset部分)

```
else begin
    if(ren === 1'b1) begin
        store[readflag] <= nxt_store;
    end
    else begin
        if(wen === 1'b1) begin
            store[writeflag] <= nxt_store;
        end
        else begin
        end
    end
    readflag <= nxt_read;
    writeflag <= nxt_write;
    dout <= tmpout;
    error <= tmperror;
end
```

在combinational部分則是負責計算各種next值。因為read priority會高於write，所以如果ren=1，就可以忽略wen，只需要判斷read的位置是否為empty。

```
if(ren === 1'b1) begin
    if(store[readflag] !== 8'b0 && store[readflag] !== 8'bx && store[readflag] !== 8'bz) begin
        tmpout = store[readflag];
        nxt_store = 8'bx;
        tmperror = 1'b0;
        nxt_read = readflag+3'b001;
        nxt_write = writeflag;
    end
    else begin
        tmperror = 1'b1;
        tmpout = 1'b0;
        nxt_store = store[readflag];
        nxt_write = writeflag;
        nxt_read = readflag;
    end
end
```

相反的ren=0時，就去判斷wen與write的位置是否已經有東西。

```
else begin
    if(wen === 1'b1) begin
        if(store[writeflag] === 8'b0 || store[writeflag] === 8'bz || store[writeflag] === 8'bx) begin
            nxt_store = din;
            tmperror = 1'b0;
            tmpout = 8'b0;
            nxt_write = writeflag+3'b001;
            nxt_read = readflag;
        end
    end
```

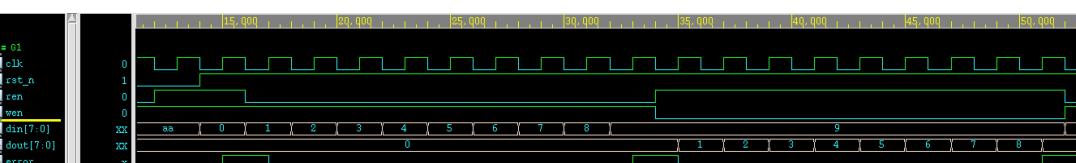
最後解釋nxt_write、nxt_read、nxt_store的判斷方式。如果有確定可以write，nxt_write = writeflag+1，代表下一次write的位置改變。同理，可以成功read時，nxt_read = readflag+1，下一次就可以讀下一個位置的值。nxt_store則是分為四個條件，如果是進行read且成功read值，nxt_store = 8'bx，代表即將將store的該位置清空；如果進行write且成功，nxt_store=din，代表即將將store該位置填入新的值；如果read失敗或write失敗nxt_store則為store的值，代表不去做更動（作法都以分佈於上方各圖中）。

III. Testbench and Waveform

這題的Testbench總共分為兩大部分。第一步分測試是不是能夠正常讀寫。首先先測試write狀況，將din=1~8分別依序寫入，接著嘗試寫入din=9，這時會因為full無法再寫入新的值，並且使error=1。接著測試read狀況，因為這時候[0]~[7]的位置應該都有資料，因此repeat (8+1)次去讀出所有資料再透過波形圖判斷讀出的資料是否正確；後面多出的那一次repeat是用來測試繼續 read empty會使error signal為1。（見下圖）

```
@(negedge clk)
    rst_n = 1'b1; //ren == 1, error = 1 at begin
    din = 8'b0;
for (idx = 1; idx < 10; idx = idx+1)begin //write 1~8 to store[0]~store[7], and error=1 when full(idx=9)
    @(negedge clk)
        ren = 1'b0;
        wen = 1'b1;
        din = idx;
end
repeat(9) begin //read from store[0]~store[7], and error=1 when empty
    @(negedge clk)
        wen = 1'b0;
        ren = 1'b1;
end
```

（下圖為本段測試waveform）



第二部分要測試在read、write互換以及在中途reset的狀況。

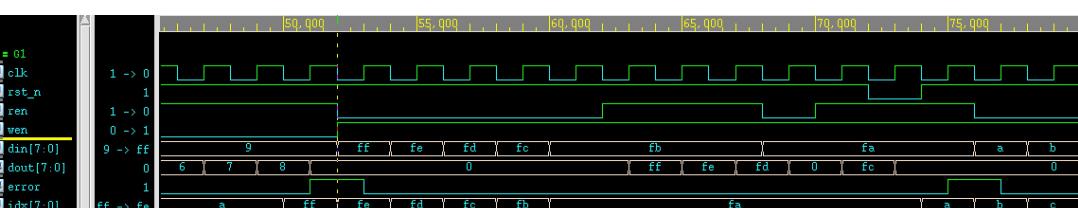
首先是寫入5個值，然後試著讀出3個值，再轉換到write輸入一個值再轉到read輸出一個值。

接著進行reset。reset後先read一次，確定error訊號，也代表queue為empty狀態。最後就簡單測試write與read。

(下圖為實作，僅節錄至reset結束時)

```
//Test interrupt reset and large din
for (idx = 255; idx > 250; idx = idx-1)begin      //write 255~251 to store[0]~store[4]
    @(negedge clk)
        ren = 1'b0;
        wen = 1'b1;
        din = idx;
end
repeat(3) begin      //read from store[0]~store[2]
    @(negedge clk)
        ren = 1'b1;
end
@(negedge clk)
    ren = 1'b0;
    wen = 1'b1;
    din = 8'd250;
@(negedge clk)
    ren = 1'b1;
@(negedge clk)
    rst_n = 1'b0;
@(negedge clk)
    rst_n = 1'b1;
    ren = 1'b1;      //empty -> error = 1
```

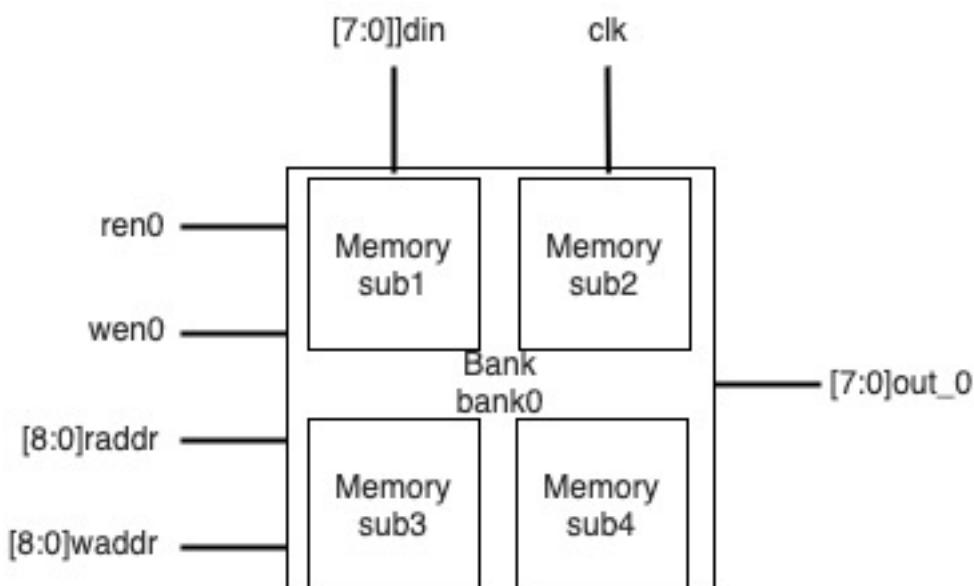
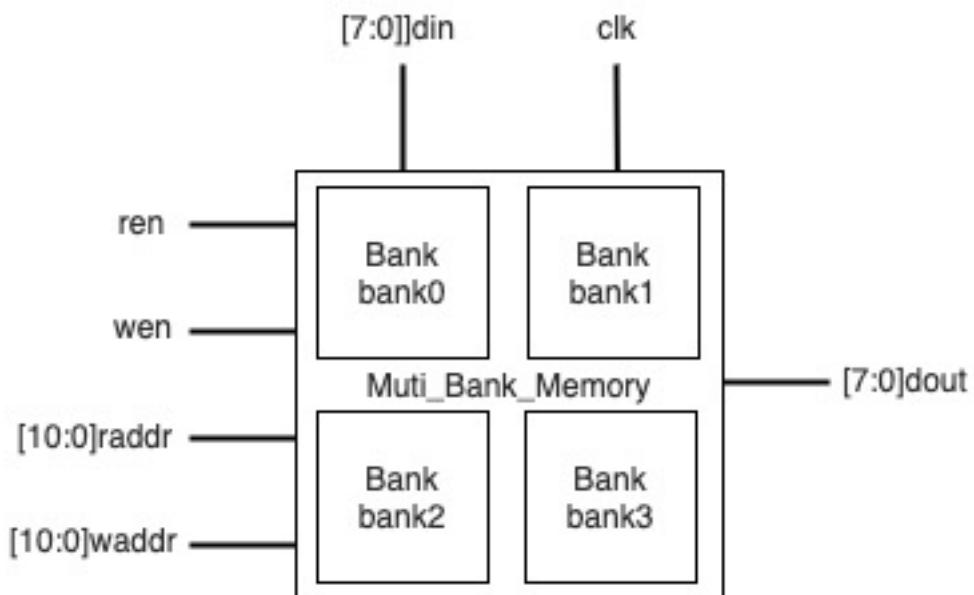
(下圖為本段測試waveform)



Muti-Bank Memory

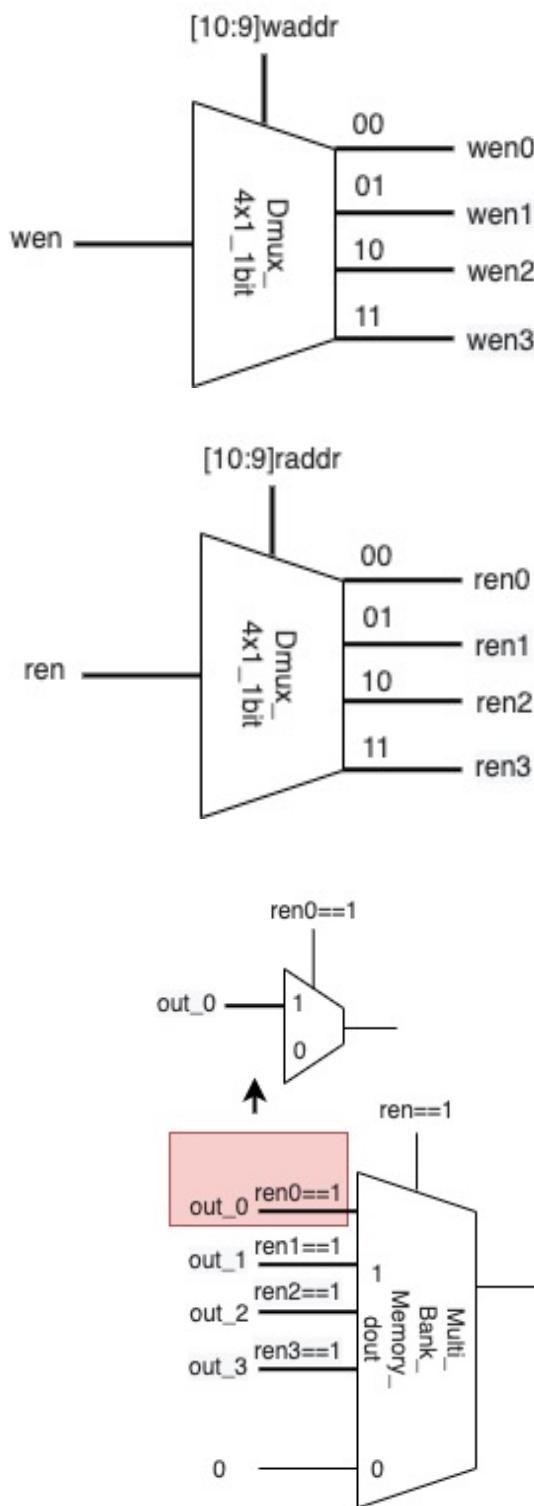
I. Diagram

(i) 電路設計圖 -Hierarchy



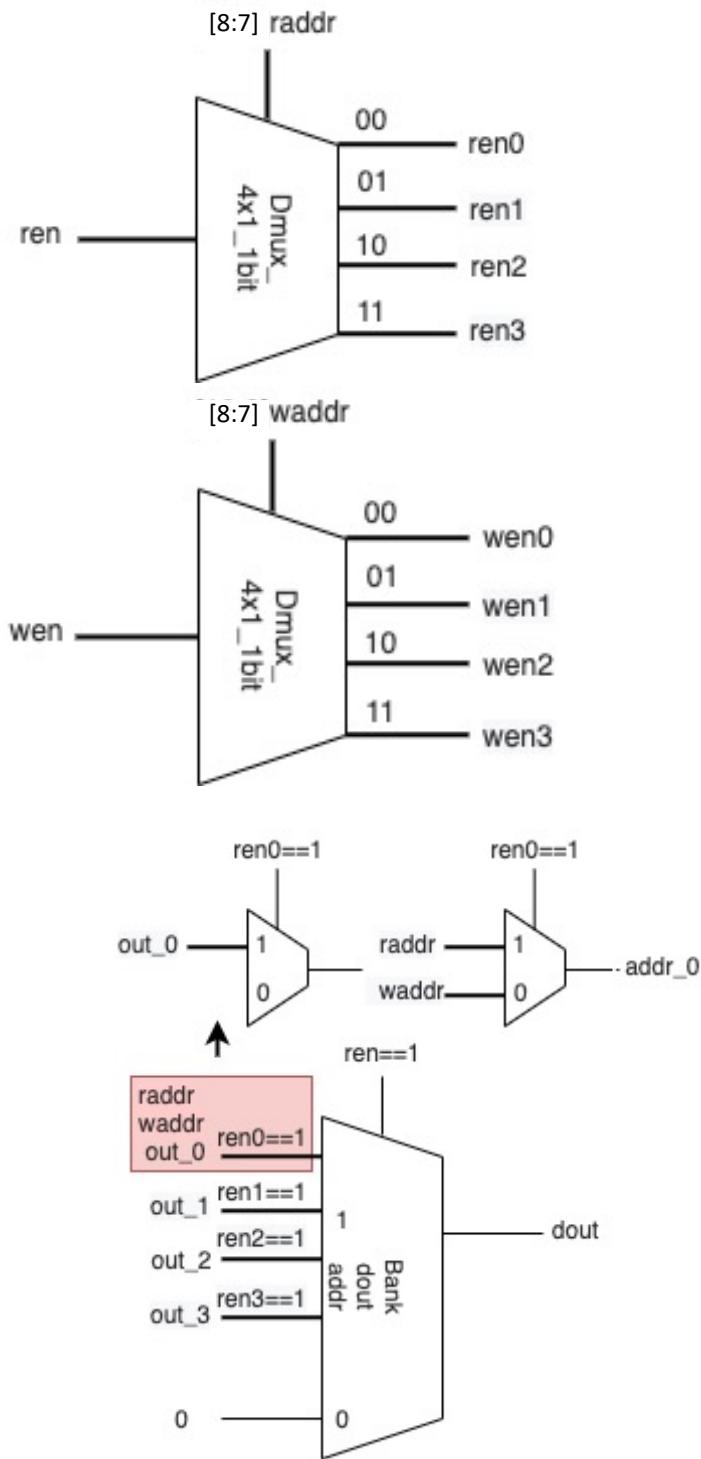
I. Diagram

Muti_Bank_Memory 第一層架構



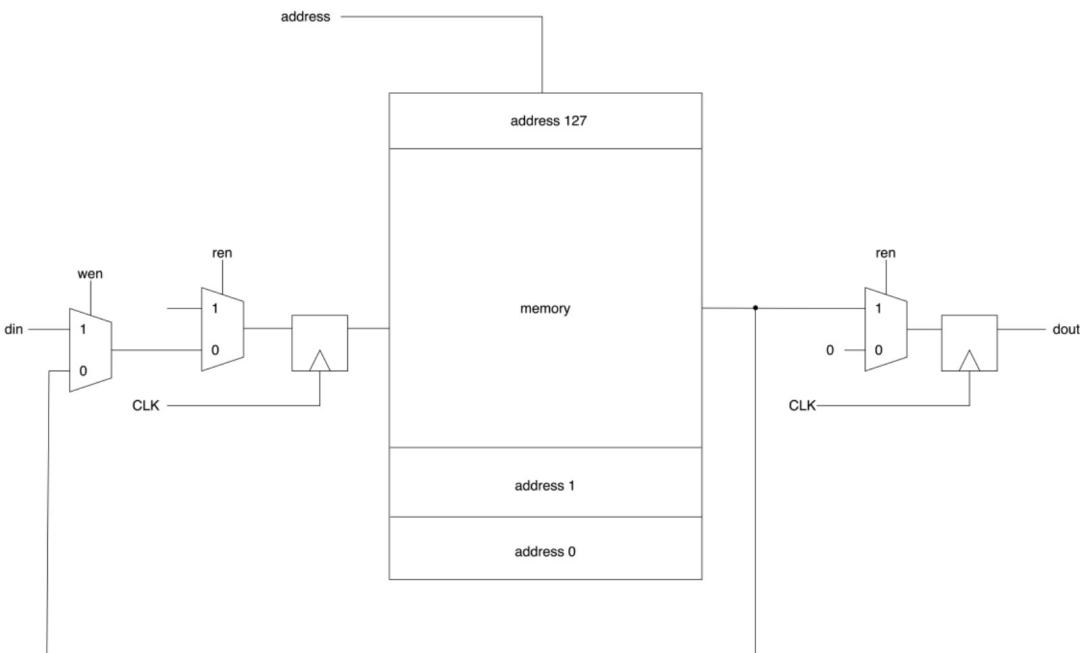
I. Diagram

subbank 第二層架構



I. Diagram

Memory Unit Module



II. Explanation

此題題目要求指定用 memory module 為底層 unit 設置一個三層架構，在第一層設置四個 subbank，每一個 subbank 設置四個 memory，最上層傳入 (waddr, raddr, wen, ren, 目標數字 din) 等值，會一層一層傳入底下的 module，並在中間做處理。

第一層作法：

將此層 input 傳入 bank，並且做 ren, wen 個別化的動作，以此來分辨要讀或寫入哪個 bank，使用 1-to-4 demux 去將此層原始的 wen, ren 寫入選擇到的 wen0~4, ren0~4。假如選擇到編號為 0 之 bank 就將 wen 寫入 wen0，以此類推。

(以下為 code 實作方式)

```
Bank bank0(.clk(clk), .ren(ren_0), .wen(wen_0), .waddr(waddr[8:0]), .raddr(raddr[8:0]), .din(din), .dout(out_0));
Bank bank1(.clk(clk), .ren(ren_1), .wen(wen_1), .waddr(waddr[8:0]), .raddr(raddr[8:0]), .din(din), .dout(out_1));
Bank bank2(.clk(clk), .ren(ren_2), .wen(wen_2), .waddr(waddr[8:0]), .raddr(raddr[8:0]), .din(din), .dout(out_2));
Bank bank3(.clk(clk), .ren(ren_3), .wen(wen_3), .waddr(waddr[8:0]), .raddr(raddr[8:0]), .din(din), .dout(out_3));

Dmux_4x1_1bit selwen(.in(wen), .sel(waddr[10:9]), .out1(wen_0), .out2(wen_1), .out3(wen_2), .out4(wen_3));
Dmux_4x1_1bit selren(.in(ren), .sel(raddr[10:9]), .out1(ren_0), .out2(ren_1), .out3(ren_2), .out4(ren_3));
```

II. Explanation

至於第二層 subbank 稍微比較複雜，因為第一層沒有選擇究竟現在是讀還是寫，傳入 memory 執行的動作是要用 raddr 還是 waddr，因此在這層我們需要判斷讀寫 ren, wen 來選擇傳入的 addr，如果這層的 ren==1 代表要寫，因為 ren 優先權高於高於 wen，傳到下一層的 addr 設為 raddr，若不讀則 dout 輸出為 0，傳到上一層，以這樣的架構，我們最終能將 ren, wen, addr 傳到最下層 memory 使用 clk sequential 的做讀取與寫入 reg，並依照時脈將 dout 一層一層網上傳，最終完成三層 bank memory。

(右圖為選擇往下傳遞 address 的實作 code)

為了處理 dout 不能在 poedge 以外的時間做更動，我們在主 module 及 bank module 皆使用一個 tmp_ren 去紀錄要接哪一個傳出來的 out。因為如果直接用 ren 與 raddr 作為判斷依據馬上會因為 negedge 的改動而影響值。

(如下圖實作)

```
always @(posedge clk) begin
    tmp_ren <= {ren_3, ren_2, ren_1, ren_0};
end

always @(*) begin
    if(tmp_ren == 4'b0001) begin
        dout = out_0;
    end

    else if(tmp_ren == 4'b0010) begin
        dout = out_1;
    end

    else if(tmp_ren == 4'b0100) begin
        dout = out_2;
    end

    else if(tmp_ren == 4'b1000) begin
        dout = out_3;
    end

    else begin
        dout = 8'b0;
    end
end
```

```
always @(*) begin
    if(ren == 1'b1) begin
        if(ren_0 == 1'b1) begin
            addr_0 = raddr[6:0];
            addr_1 = waddr[6:0];
            addr_2 = waddr[6:0];
            addr_3 = waddr[6:0];
        end

        if(ren_1 == 4'b0010) begin
            addr_1 = raddr[6:0];
            addr_2 = waddr[6:0];
            addr_3 = waddr[6:0];
        end

        if(ren_2 == 4'b0100) begin
            addr_2 = raddr[6:0];
            addr_3 = waddr[6:0];
            addr_1 = waddr[6:0];
        end

        if(ren_3 == 4'b1000) begin
            addr_3 = raddr[6:0];
            addr_2 = waddr[6:0];
            addr_1 = waddr[6:0];
        end
    end
    else begin
        addr_3 = waddr[6:0];
        addr_2 = waddr[6:0];
        addr_1 = waddr[6:0];
        addr_0 = waddr[6:0];
    end
end
```

III. Testbench and Waveform

本題testbench會檢查在不同組合的bank中讀寫同一addr能不能正確地完成讀寫，ex:若前一步將值寫入bank1101+addr中，之後從bank0111+addr中應該讀不到東西。

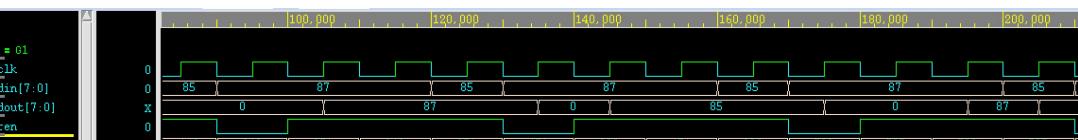
以iteration的方法將{a,addr}中的a+1去產生不同bank與位址的組合。ren與wen輪流去設為(0,0)(0,1)(1,0)(1,1)去檢查，直到檢查完所有情況，以確保程式的正確性。

1. 讀寫同時發生在同一個bank同一個addr的話只會讀

2. 讀寫同時發生在不同bank同一個addr的話可以同時進行
(下圖為範例)

```
@(negedge clk)
waddr ={ a,7'd87};
raddr ={ b,7'd87};
din = 8'd87;
ren = 1'b0;
wen = 1'b1;
@(negedge clk)
waddr = { a,7'd87};
raddr ={ b,7'd15};
din = 8'd87;
ren = 1'b1;
wen = 1'b1;
a = a+1'b1;
```

(下圖為waveform部分截圖)

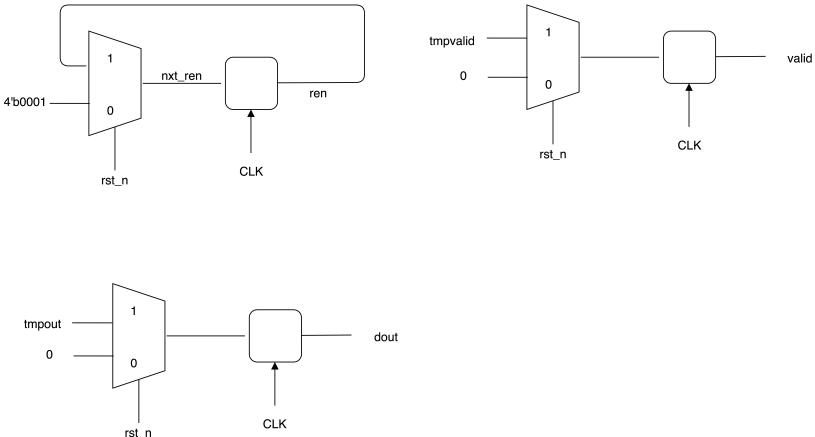


Round-Robin FIFO Arbiter

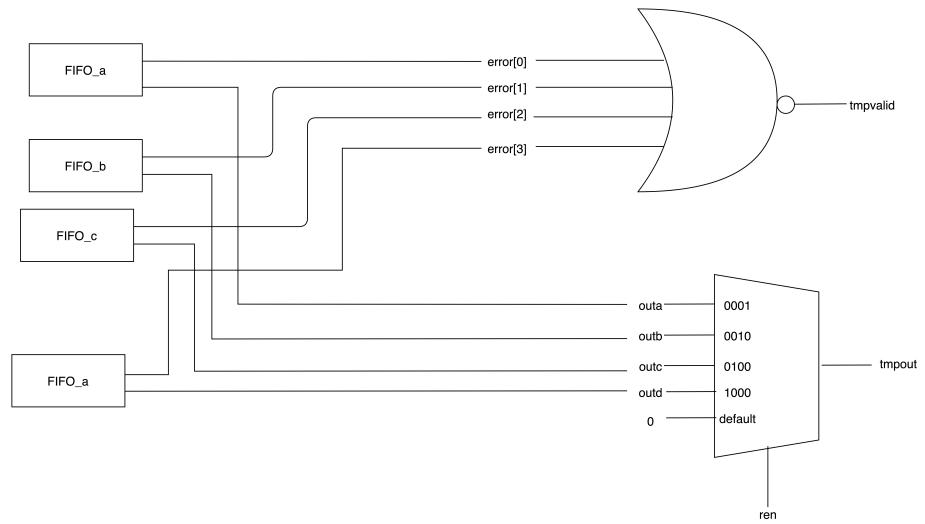
I. Diagram

(1) 電路設計圖 (Round-Robin Arbiter part)

RRA
Sequential

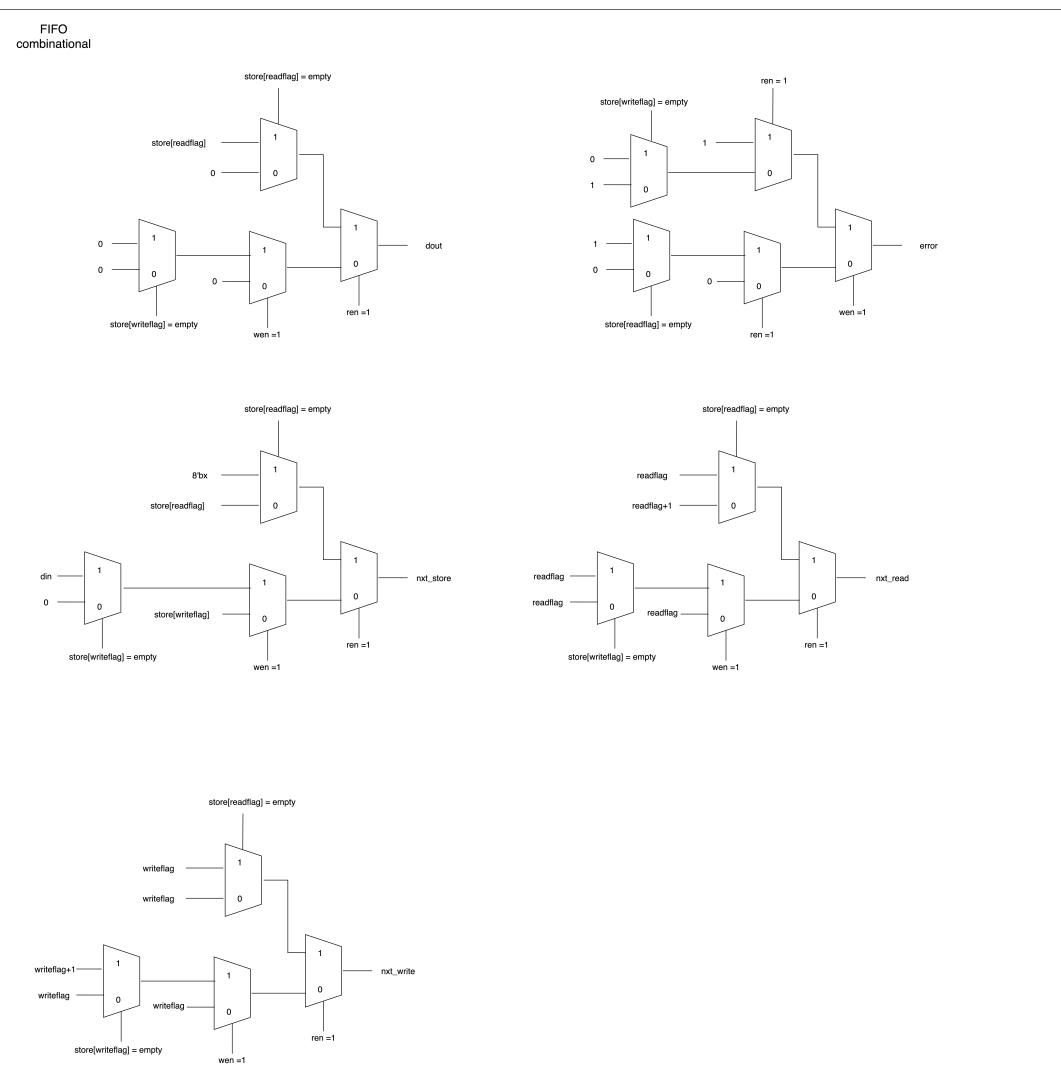
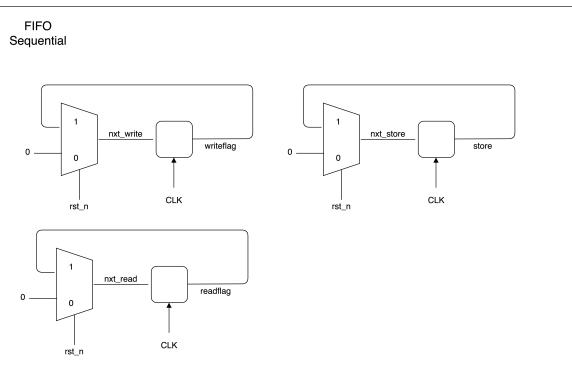


RRA
Combinational



I. Diagram

(1) 電路設計圖 (FIFO part)



II. Explanation

首先先說明FIFO part。與第二題的做法大同小異，差別在於原本隨著clock trigger才會更新的error與dout，現在做成combinational，會隨時改變。會這樣更動的原因是因為FIFO part其實也是Round-Robin-Arbitrer的combinational circuit的其中一段，如果用clock trigger會使dout與error晚一個clock傳入tmpout與tmpvalid，然後晚一個clock cycle輸出。

(更動範例如下圖)

```
if(wen === 1'b1) begin
    if(store[writeflag] === 8'b0 || store[writeflag] === 8'bx || store[writeflag] === 8'bz) begin
        if(ren === 1'b1) begin
            error = 1'b1;
            dout = 8'b0;
            nxt_store = din;
            nxt_write = writeflag+3'b001;
            nxt_read = readflag;
        end
    end
end
```

另外就是因為在這題當中read、write同時進行時，write priority較高，且error訊號會被拉起，因此改為先判斷wen再判斷ren，這部分也可以從上方的圖看出來。

接著說明Round-Robin-Arbitrer的部分。分為sequential與combinational兩部分，sequential只做valid、dout及ren的更新。valid與dout為題目指定輸出，ren則是用來選擇要去read哪一個FIFO的選擇器，會不斷循環。

在combinational部分，分為tmpout、tmpvalid及nxt_ren三個重要的值。tmpout會透過case的方式，從4個FIFO的dout選一個。

(右圖為tmpout選擇方式)

```
case (ren)
  4'b0001: begin
    tmpout = outa;
  end
  4'b0010: begin
    tmpout = outb;
  end
  4'b0100: begin
    tmpout = outc;
  end
  4'b1000: begin
    tmpout = outd;
  end
  default: begin
    tmpout = 8'b0;
  end
endcase
```

tmpvalid則是將4個FIFO的error透過nor的方式得到。

(可見下圖)

```
always@(*) begin
  tmpvalid = !(error[0]|error[1]|error[2]|error[3]);
end
```

ren代表要選的 FIFO，依照0001、0010、0100、1000、0001……的循環去改變，nxt_ren會根據現在的ren狀況去改變值。

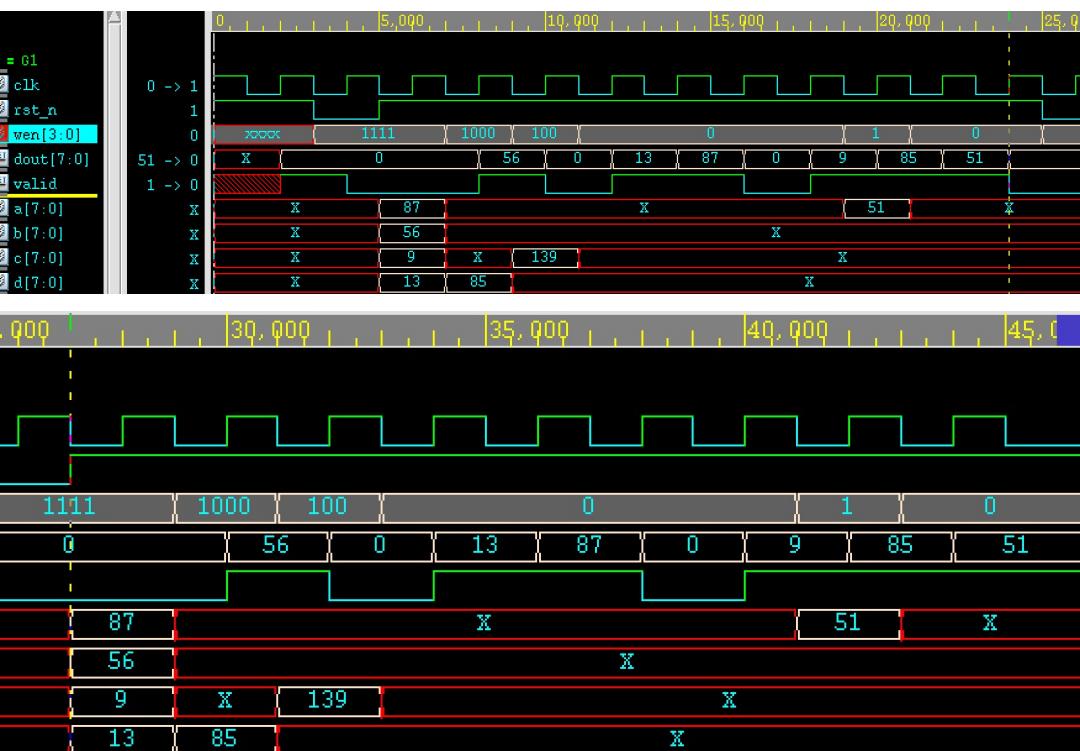
(下圖為nxt_ren改變規則)

```
nxt_ren = (ren==4'b1000)?(4'b0001):(ren<<1);
```

III. Testbench and Waveform

這個testbench設計概念是：因為spec的圖形完整，而且有測試到同時讀寫、讀到空值的錯誤，我們需要添加的就是判斷有沒有順利reset。因此我們沿用spec上的測資，並且將其repeat 2次，中間添加reset的狀況。

(下圖分別為在中途reset前後的waveform)

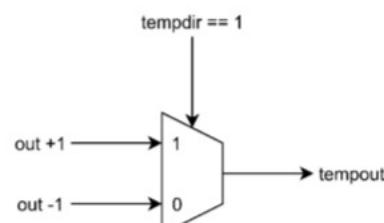
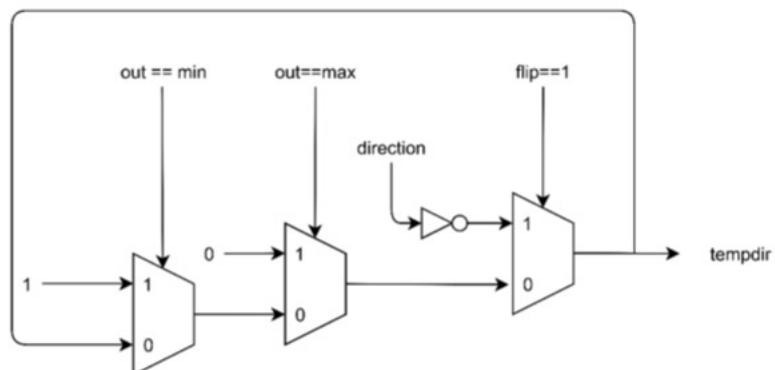
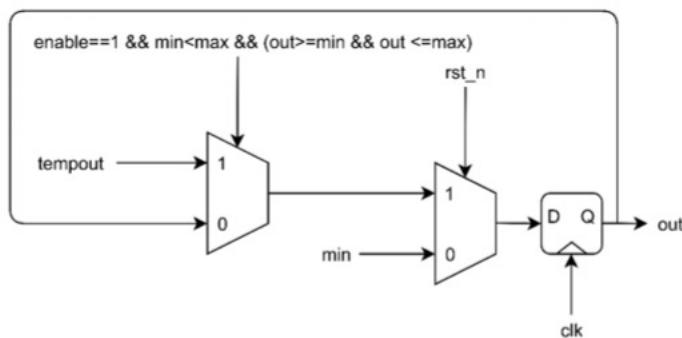
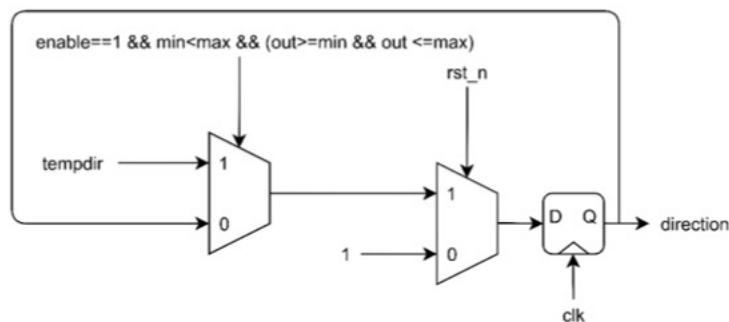


從上方圖片可以看出我們的dout與valid訊號更改的時間與spec是相同的，而且reset後原本FIFO都有清空（否則讀到C時應該dout = 139），也有重新計算讀取順序。

Parameterized Ping-Pong Counter

• I. Logic Design Diagram

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



II. explantion

一開始rst_n會將out與direction設為初始值，並在每個clk的posedge同時sequentially的以電路圖畫出的方法做運算

1.out接收tempdir得到tempout

2.得到tempdir與tempout後檢查enable是否通過，是否在min與max之間，在將這兩個值餵給direction,out

3.tmpout要記得如果等於min或max要跳往反方向，以及吃flip訊號決定要不要變成反向的direction，也就是現在方向的反向也就是說，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。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。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，成立則將tempout的值給out、將tempdir的值給direction。

這題比第一題加上了min跟max的條件，初始化為min，與第一題明顯的差異是min max flip控制了tmpdir，進而影響到下一步direction應該輸出的值。

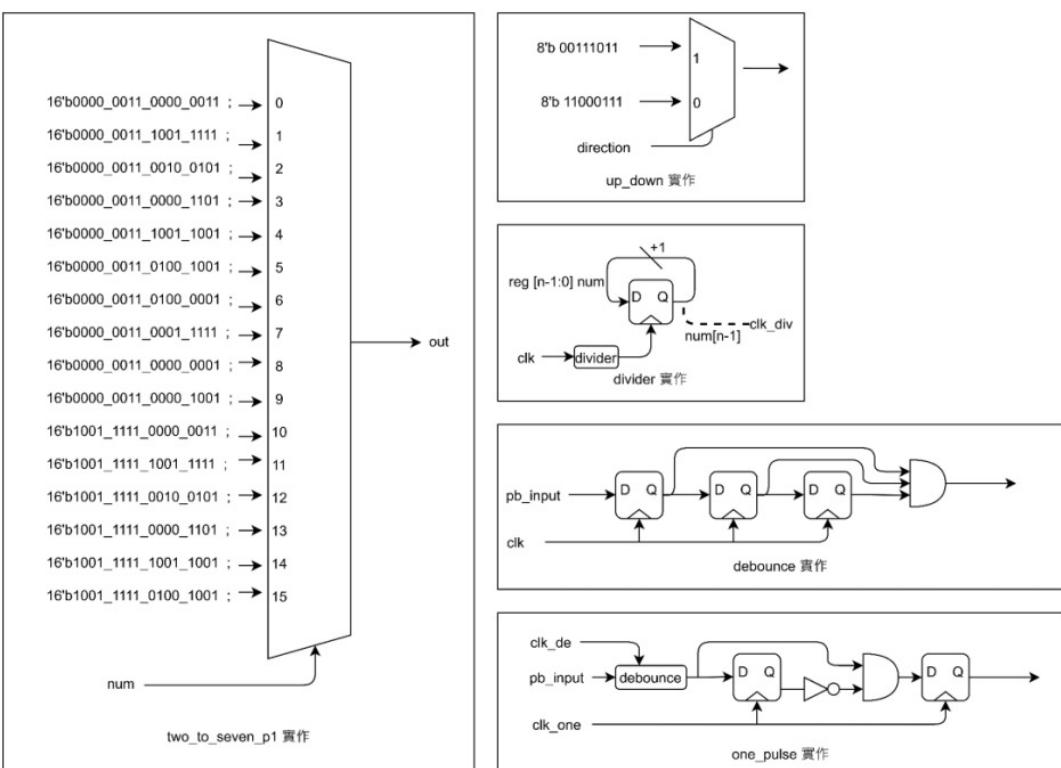
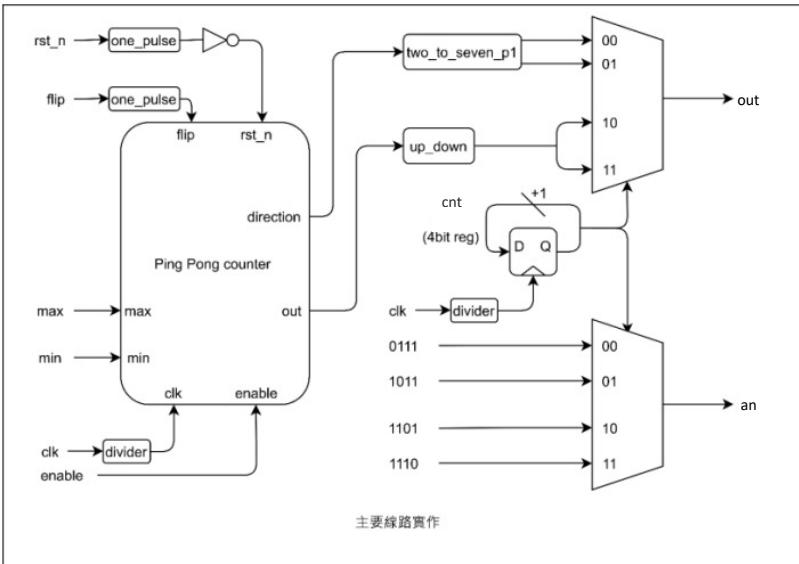
II. explantion

這題比第一題加上了min跟max的條件，初始化為min，與第一題明顯的差異是min max flip控制了tmpdir，進而影響到下一步direction應該輸出的值。

4-bit Ping-Pong Counter FPGA

I. Diagram

(i) 電路設計圖



II.Explanation

為了接pingpong counter 傳出來的direction與out信號，我們用case去用傳出的out數字選fpga上的7segment顯示器訊號，得到tmpout，再用direction選出7segment方向顯示器訊號，得到tmpdir，再用一個always block去把tmpdir與tmpout餵給顯示器要顯示的direction與out，要注意的是因為要將信號週期放大 2^{17} 倍，我們用了clk_18去跑cnt來輪流顯示四位數字的an訊號。而數字跳動為了要能夠辨識，用 2^{25} 次方，接近一秒的頻率來顯示。在rst與flip button上野時做了debounce 訊號，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。

• 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。

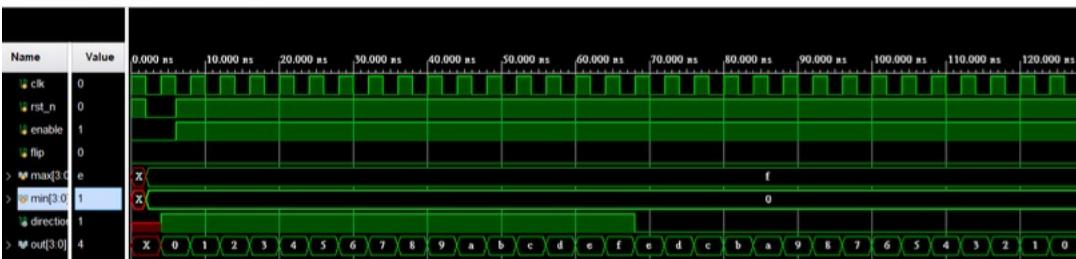
```

@(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);
end

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



What are we learned from this Lab

這次的作業我們在三、四兩題花上許多時間。在Multi-Bank-Memory的部分，我們一開始使用了錯誤的思考方式，用軟體呼叫function的方式去思考一層層的module，結果一直著眼於無法有效維持memory這件事，花費了大量時間。這也再次提醒我們，硬體與軟體的構思方式截然不同，養成良好的書寫與思考習慣可以有效的增加工作效率。

而在Round-Robin-Arbitrer我們則是遇到clock計算的障礙。先是多延遲了一個clock cycle去找output值，又因為round-robin-arbiter的特性，要多過好幾個clock cycle才會回到我們想要的位置讀取剛剛未找到的值，因此整體晚了3~4個cycle才找到值。後來我們發現sequential與combinational circuit不只是在同一個module下可以運用，更可以活用於相關聯的多個module中。將其中一個module用來進行combinational circuit的部分，而非每一個module都必須要包含sequential與combinational，反而可以好好得到我們想要的value。

Cooperation

108062213 顏浩昀：

FIFO module 實作、FIFO testbench、Round-Robin-Arbiter 實作、Round-Robin-Arbiter testbench、Multi-Bank-Memory module 實作、Parameterized-Ping-pong counter testbench、FPGA debug、report 製作

106062304 黃鈺舒：

Ping-pong counter 實作、Ping-pong counter testbench、Multi-Bank-Memory 構想、Multi-Bank-Memory testbench、Parameterized-Ping-pong counter module 實作、FPGA 實作、report 製作