

Lab3_Team1_Report

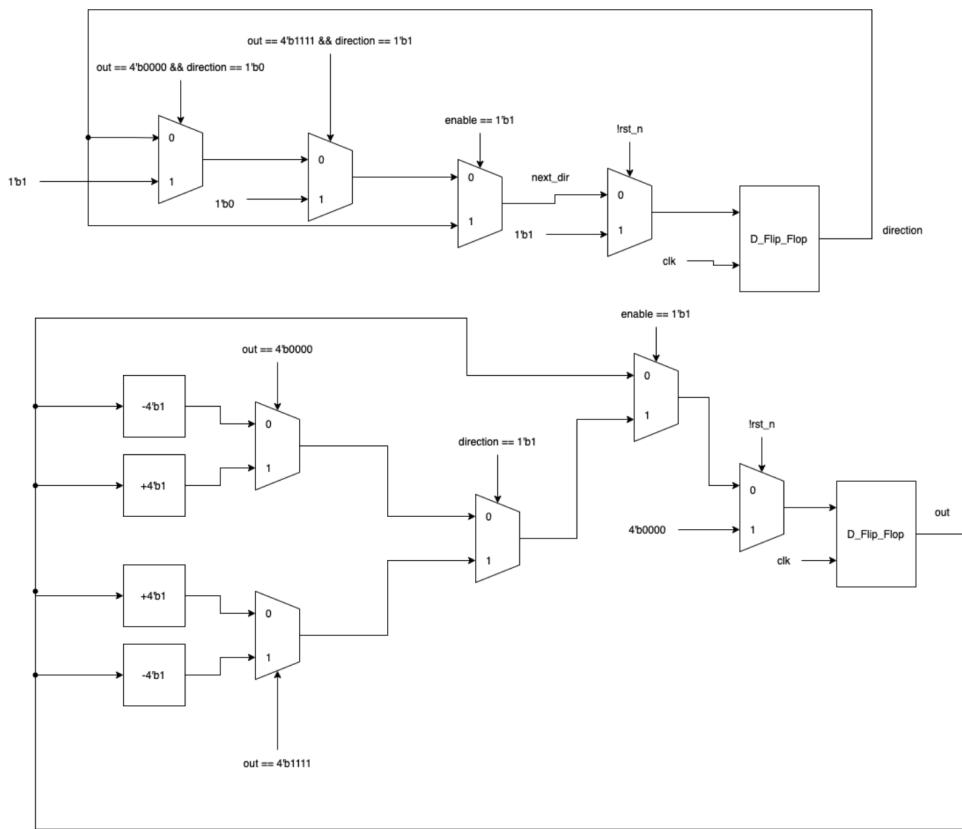
組員

107071016 施泳瑜

109062320 朱季葳

Advanced Question1

Drawing of the design of Verilog Advanced Question 1



Requirements

1. $\text{rst_n} = 0$
 - $\text{out} = 4'\text{b}0000$
 - $\text{direction} = 1'\text{b}1$
2. $\text{rst_n} = 1$
 - $\text{enable} = 1$
 - counter begin its operation
 - $\text{enable} = 0$
 - counter hold its current value

Design Explanation

我們的設計分為三個部分

1. 當正緣觸發時，更新output的值

- `rst_n = 0`
 - 依據requirement所要求的，更新out和direction的值
- `rst_n = 1`
 - 將`next_dir`的值用來更新`direction`
 - 將`next_out`的值用來更新`out`

2. 依據不同的狀況來更新`next_dir`的值

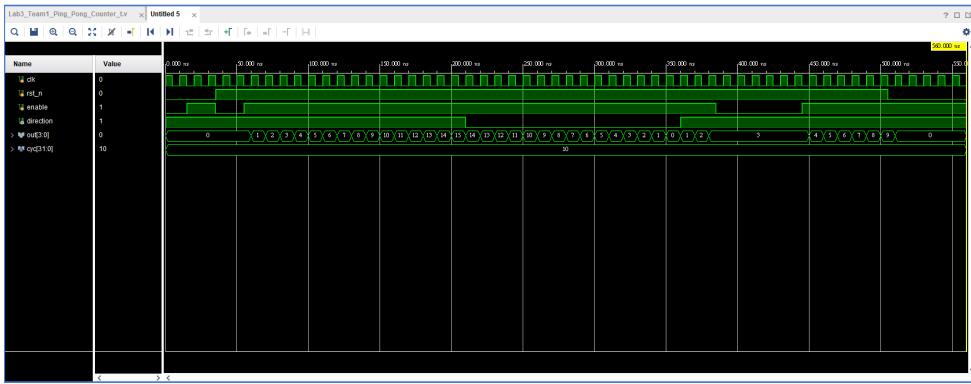
- `enable = 0`
 - `next_dir`維持在前一個cycle的值
- `enable = 1`
 - `out == 4'b1111 && direction == 1'b1`
 - 當out到最大值且方向為正數時，在下一個cycle更改方向為倒數
 - `out == 4'b0000 && direction == 1'b0`
 - 當out到最小值且方向為倒數時，在下一個cycle更改方向為正數
 - else
 - 維持原本的方向

3. 依據不同的狀況來更新`next_out`的值

- `enable = 0`
 - `next_out`維持在前一個cycle的值
- `enable = 1`
 - `direction = 1`
 - 當out為最大值時，在下一個cycle將其減一
 - 反之，則將其加一，因為此刻的方向為正數
 - `direction = 0`
 - 當out為最小值時，在下一個cycle將其加一
 - 反之，則將其減一，因為此刻的方向為倒數

Testbench Design & Result Explanation

波形圖截圖



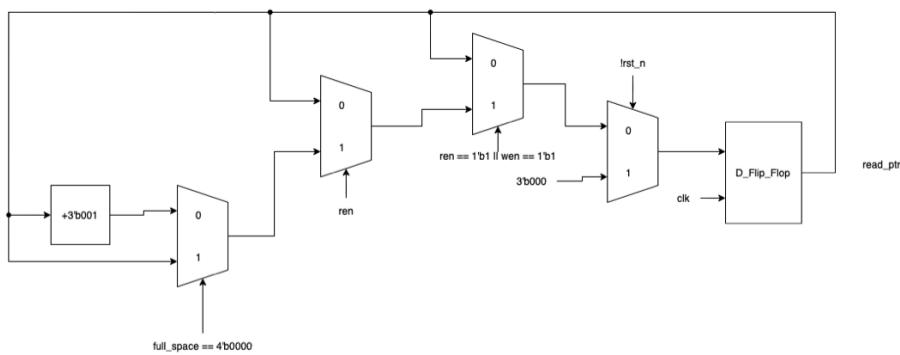
TESTBENCH設計解釋

1. 將rst_n設為0來進行reset以及將enable設為0來測試當進行reset且enable為0的時候，output是否為reset時應有的值
2. 將rst_n設為0來進行reset以及將enable設為1來測試當進行reset且enable為1的時候，output是否為reset時應有的值
3. 將rst_n設為1以及將enable設為0來測試當enable為0的時候，counter是否有維持原本應有的值
4. 將rst_n設為1以及將enable設為1並維持32個cycle來測試當enable為1的時候，counter是否有正常運作
5. 將enable設為0來再次測試counter是否能維持在原本的狀態
6. 將rst_n設為0來再次檢查reset正常運作

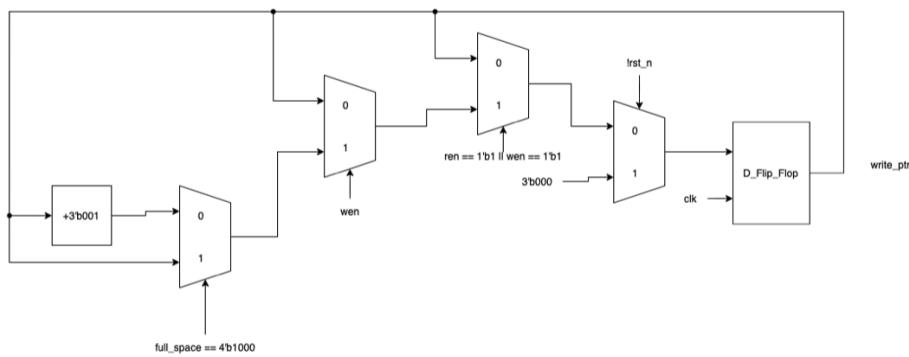
Advanced Question2

Drawing of the design of Verilog Advanced Question 2

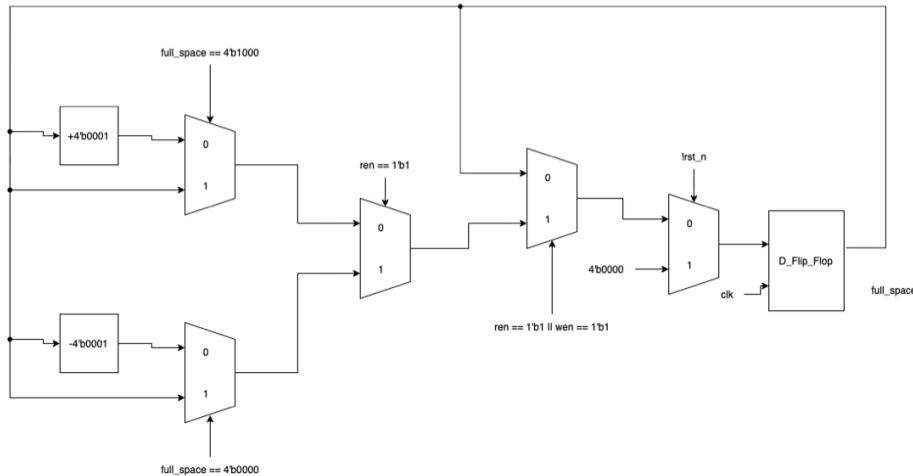
- read_ptr



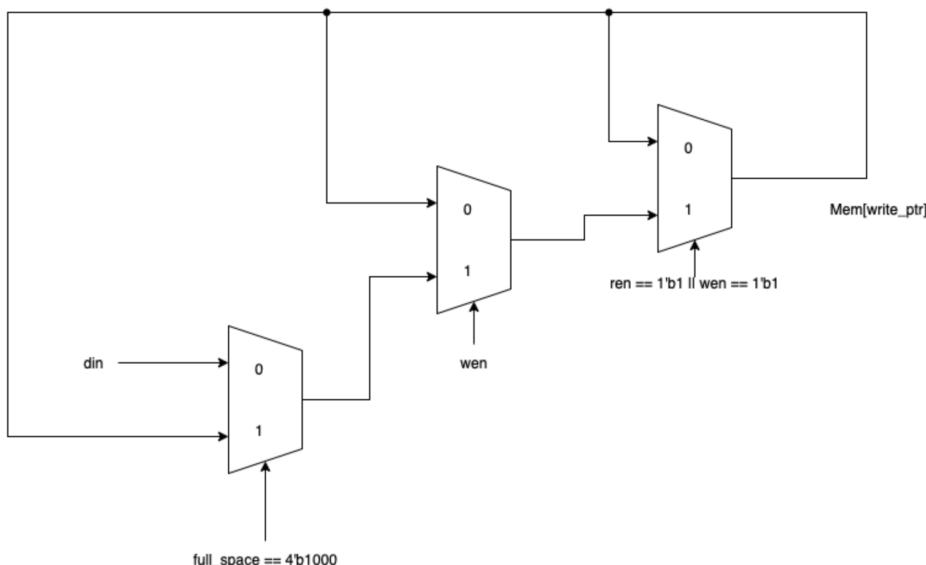
- write_ptr



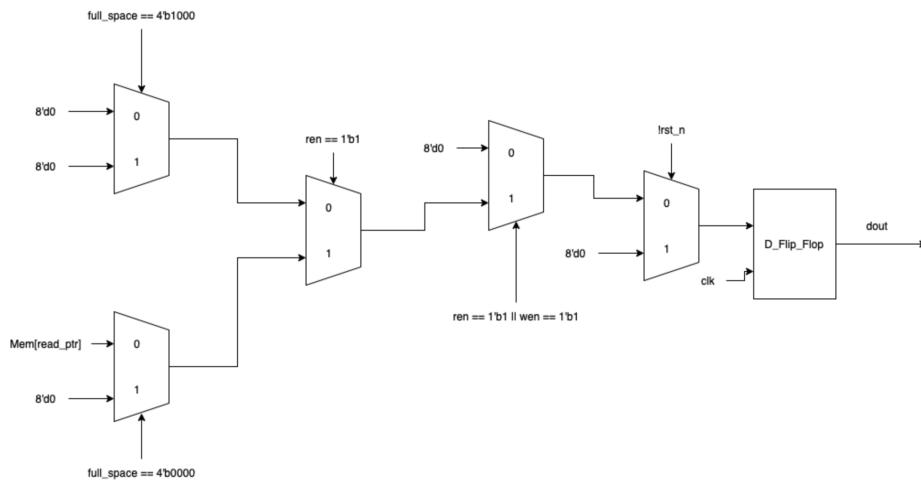
- full_space



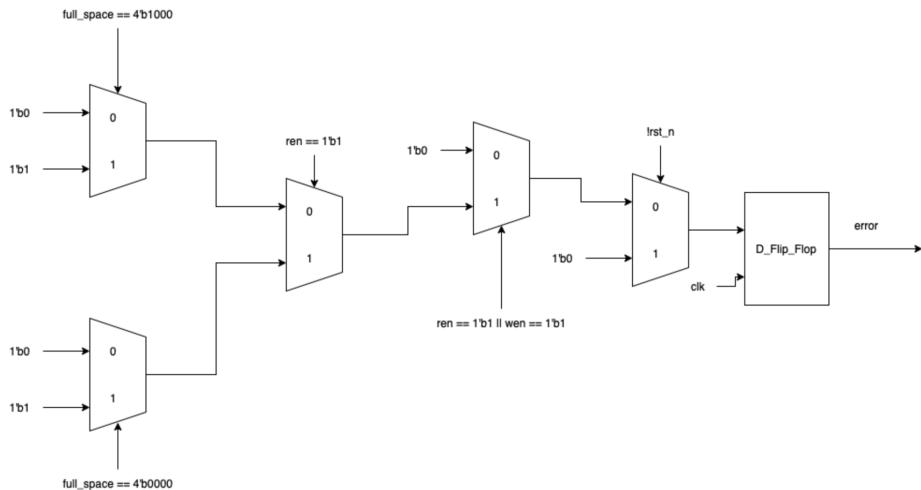
- Mem



- dout



- error



Requirements

1. 實作一個FIFO(pop的時候pop掉最先進入的值，像在排隊一樣)
2. ren = 1時，讀出在FIFO裡最久的值，並pop掉他
3. wen = 1時，寫入din的值到FIFO
4. ren = wen = 1時，做讀取的動作
5. ren = 1時，當FIFO裡是空的時候error = 1'b1
6. wen = 1時，當FIFO裡是滿的時候error = 1'b1

Design Explanation

我們的設計分為四個部分

1. 當正緣觸發時，更新read_ptr, write_ptr, full_space, dout, error的值
 - rst_n == 0:
 - reset read_ptr, write_ptr, full_space, dout, error 的值為0 (預設0為 empty) · 使得FIFO回到最初

始沒有寫入任何值的狀態

- $\text{rst_n} == 1$:
 - 將 next_read_ptr 的值用來更新 read_ptr
 - 將 next_write_ptr 的值用來更新 write_ptr
 - 將 next_full_space 的值用來更新 full_space
 - 將 next_dout 的值用來更新 dout
 - 將 next_error 的值用來更新 error

2. $\text{ren} = 1$ (因為只要 ren 等於1，無論 wen 的值等於多少，都進行讀取的動作)

- **read empty:**當 full_space 的值為0時，代表FIFO裡面沒有任何資料，也就不能讀取任何值，也因此達到 error 的條件
 - $\text{next_error} = 1'b1$
 - $\text{next_dout} = 8'b0$ (don't care)
- **read data:**此時FIFO內有資料可以讀取，因此我們讀取 read_ptr 所指向的那個位置的資料(read_ptr 會指向最舊的資料，並且在讀取完資料的下一個cycle指到下一個data的位置(因為FIFO會依照循環的方式去讀寫資料，所以假如現在 $\text{read_ptr} = 3'b000$ ，則當資料pop掉之後， read_ptr 會指向 $3'b001$ ，在寫入資料的時候也是一樣的))
 - $\text{next_error} = 1'b0$
 - $\text{next_dout} = \text{Mem}[\text{read_ptr}]$
 - 將 next_full_space 減掉 $4'b0001$ ，因為在讀取完資料之後要pop掉該資料

3. $\text{wen} = 1(\text{ren} = 0)$

- **write full:**當 full_space 的值為 $4'd8$ 時，代表FIFO被寫滿了，所以不能夠再寫入了，也因此達到 error 的條件
 - $\text{next_error} = 1'b1$
 - $\text{next_dout} = 8'b0$ (don't care)
- **write data:**此時的FIFO內仍有空間可以寫入資料，因此我們將資料寫入 write_ptr 所指向的位置，並在下一個cycle時將其指到下一個待寫入的位置
 - $\text{next_error} = 1'b1$
 - $\text{next_dout} = 8'b0$ (don't care)

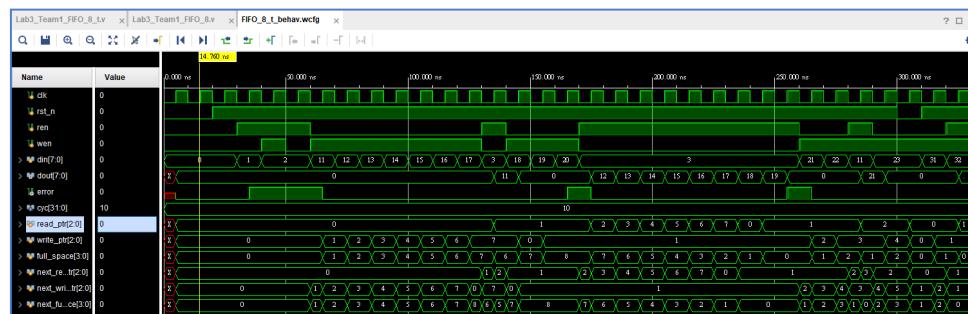
- Mem[write_ptr] = din
- 將next_full_space加上4'b0001，因為在寫入一個資料之後，被填滿的空間會多一個

4. ren = wen = 0

- 沒有任何功能要實踐，所以不會有error，dout的值也不重要
 - next_error = 1'b0
 - next_dout = 8'b0(don't care的值一律預設為0)

Testbench Design & Result Explanation

波形圖截圖



TESTBENCH設計解釋

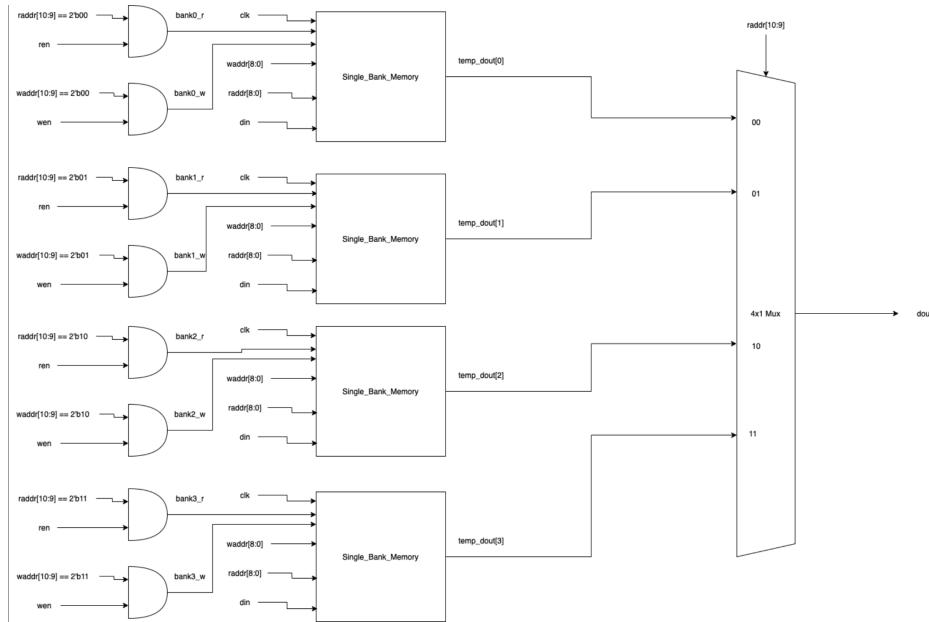
1. rst_n降下來一個cycle進行reset
2. 將rst_n拉起來
3. 測試read empty (ren = 1, wen = 0)
4. 測試read empty (ren = 1, wen = 1)
5. 確認剛剛把wen拉起來的時候沒有把任何值寫入
6. 開始將資料寫入(din = 8'd11 ~ din = 8'd17)，寫入完成後，full space為7
7. 讀出一個值，並確認值為8'd11(最先寫入的值)，此時full space為6
8. 再將三筆資料寫入(din = 8'd18~din = 8'd20)，並確認在想要寫入8'd20的時候，error升起，因為寫完8'd19之後，就沒有空間可以再寫入了
9. 進行9個cycle的read，確認有依序讀出剛剛寫入的值，還有在第9個cycle的時候error有升起
10. 寫入din = 8'd21~din = 8'd23
11. reset
12. 寫入din = 8'd31

13. 確認讀出來的值是剛剛寫入的8'd31而非8'd21~8'd23來驗證reset有成功

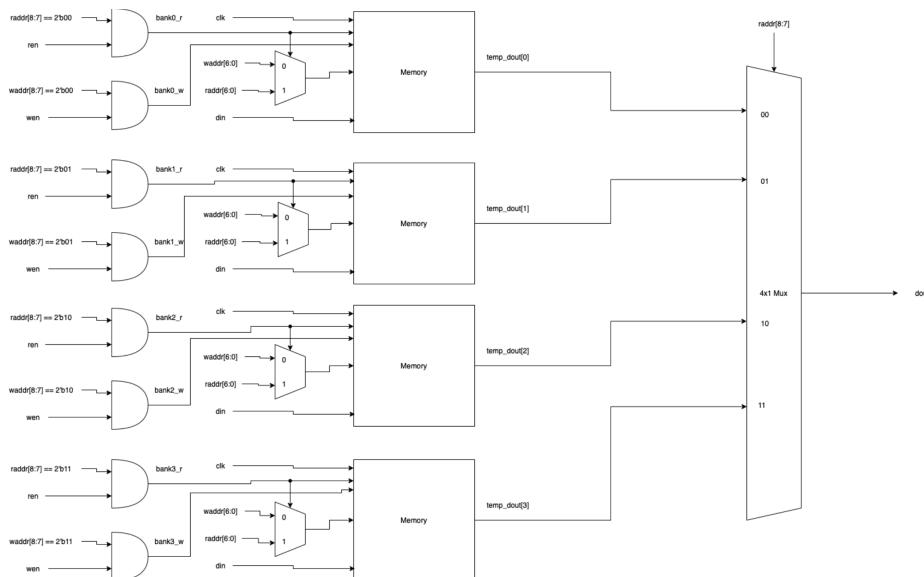
Advanced Question3

Drawing of the design of Verilog Advanced Question 3

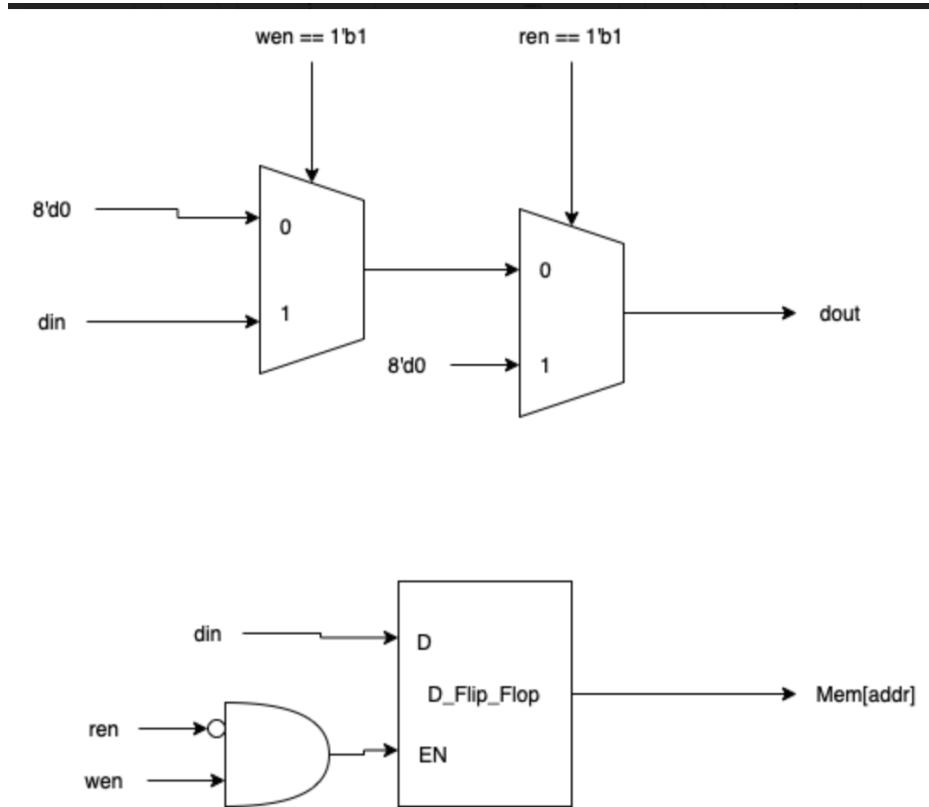
- Top module: Multi_Bank_Memory



- Single_Bank_Memory



- Memory



Requirements

1. 設計一個Memory hierarchy，裏面有四個bank，每個bank裡面有四個sub-bank，共16個sub-bank
2. sub-bank使用basic question2所實作的memory
3. waddr和raddr中the most significant four bits代表哪個bank的哪個sub-bank，而剩下的bit代表要寫入/讀出的資料位於memory中的位置
4. 在不同的sub-bank中，是可以同時進行讀寫的
5. 在同一個sub-bank中，就只能讀或者寫

Design Explanation

我們在multi_bank_memory中instantiate四個single_bank_memory，而在single_bank_memory中，再instantiate 四個memory。而multi_bank_memory以及single_bank_memory的方式極為相似，所以我們將在以下統一說明。

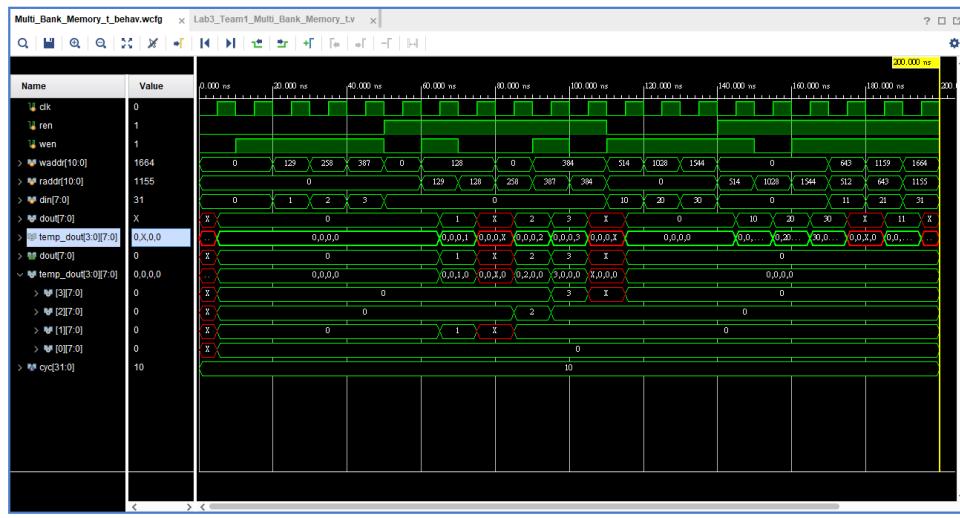
- 在bank0_w ~ bank3_w的部分，我們將wen與代表bank address的waddr[10:9]做運算，並分別接到所屬的 instantiated single_bank_memory中接到wen的port
 - e.g. bank0_w = wen && (waddr[10:9] == 2'b00)
代表假如要寫入資料的bank是bank0，則bank0_w = 1, bank1_w = bank2_w = bank3_w = 0

同理，在single_bank_memory裡，wen則是和 (waddr[8:7])做運算，並分別將其接到所屬memory中接到 wen的port

- 在bank0_r ~ bank3_r的部分則和第一個部分相似，只不過把第一個部分的wen改成ren，還有waddr改成raddr。
- 在multi_bank_memory module中，我們將clk, waddr[8:0], raddr[8:0], din分別接到四個instantiated single_bank_memory的input port中。
至於single_bank_memory module的部分，我們則將 bank0_r ~ bank3_r作為selector輸入Mux來決定要輸入到 memory的addr的port是waddr[6:0]還是raddr[6:0]，如果該sub-bank要讀取資料，則將raddr[6:0]接到addr的port；反之，則將waddr[6:0]接入。
 - e.g. if bank0_r == 1, then bank0_addr = raddr[6:0]
else bank0_addr = waddr[6:0]
- dout的部分，我們則將temp_dout輸入到always block的 sensitivity list中，並在temp_dout有變化的時候，依據 raddr所指出的bank/sub-bank為何，來將相對應的 temp_dout接到dout
 - e.g. in multi_bank_memory:if waddr[10:9] == 2'b00,
then dout = temp_dout[0]
 - e.g. in single_bank_memory:if waddr[8:7] == 2'b00,
then dout = temp_dout[0]

Testbench Design & Result Explanation

波形圖截圖



TESTBENCH設計解釋

1. check single bank

- write data to bank0, sub-bank0
- write data to bank0, sub-bank1
- write data to bank0, sub-bank2
- write data to bank0, sub-bank3

read the data written just now :

- read bank0, sub-bank0($\text{ren} = 1, \text{wen} = 0$)
- read bank0, sub-bank1($\text{ren} = 1, \text{wen} = 1, \text{raddr}[10:7] = \text{waddr}[10:7]$)
- read bank0, sub-bank1 to check in the previous cycle, nothing is written to it
- read bank0, sub-bank2($\text{ren} = 1, \text{wen} = 0$)
- read bank0, sub-bank3($\text{ren} = 1, \text{wen} = 1, \text{raddr}[10:7] = \text{waddr}[10:7]$)
- read bank0, sub-bank3 to check in the previous cycle, nothing is written to it

2. check other banks

- write data to bank1, sub-bank0
- write data to bank2, sub-bank0
- write data to bank3, sub-bank0

read the data written just now :

- read bank1, sub-bank0
- read bank2, sub-bank0
- read bank3, sub-bank0

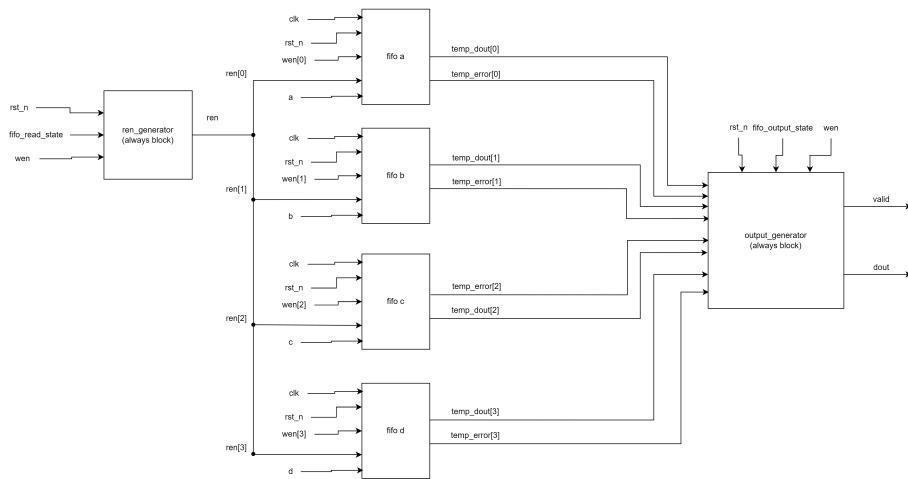
check read and write can be serviced simultaneously if they are directed for different sub-banks:

- read bank1, sub-bank0 (with nothing is written in the desired address) and write data to bank1, sub-bank1
- read bank1, sub-bank1 (data written just now), and write data to bank2, sub-bank1

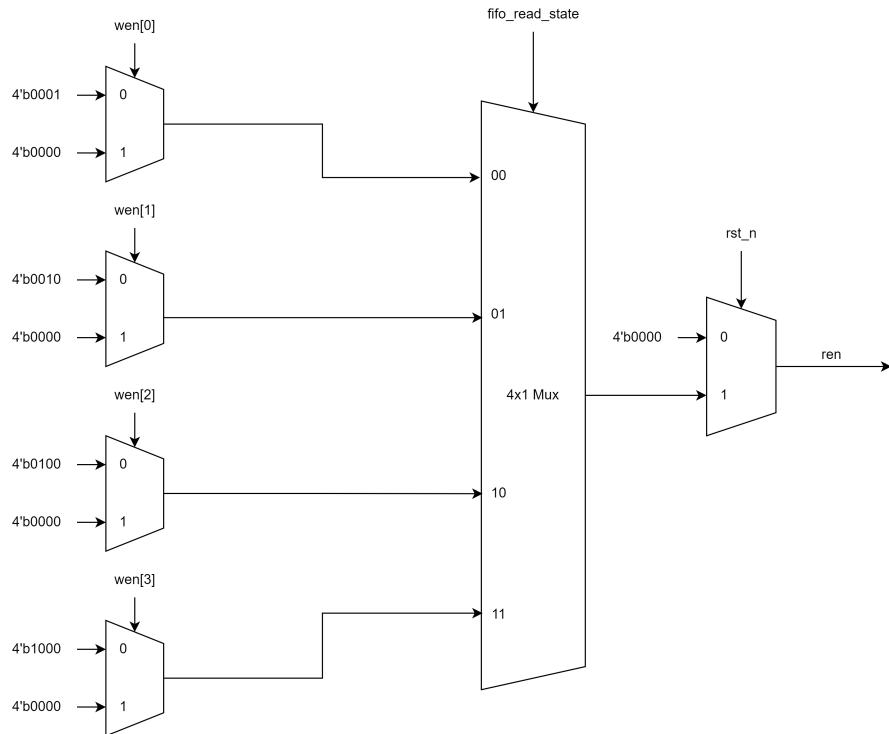
Advanced Question4

Drawing of the design of Verilog Advanced Question 4

- Top module

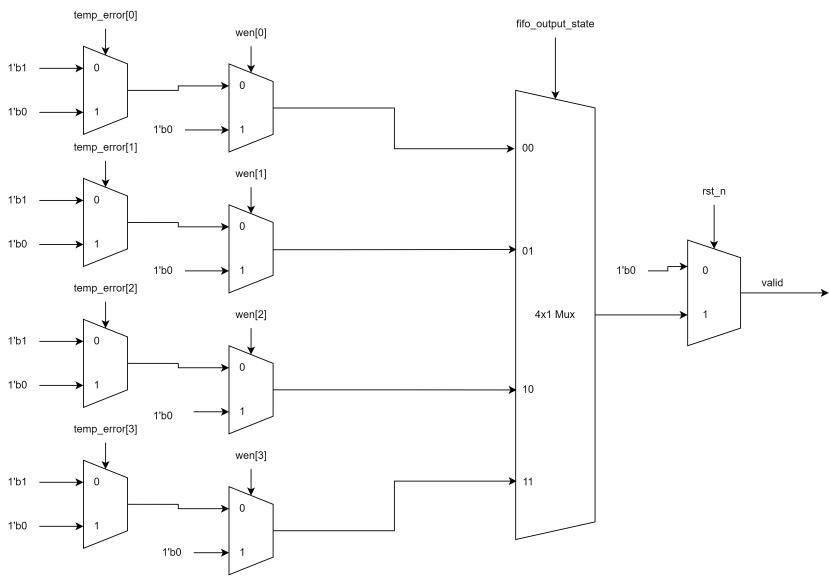


- ren_generator(always block)

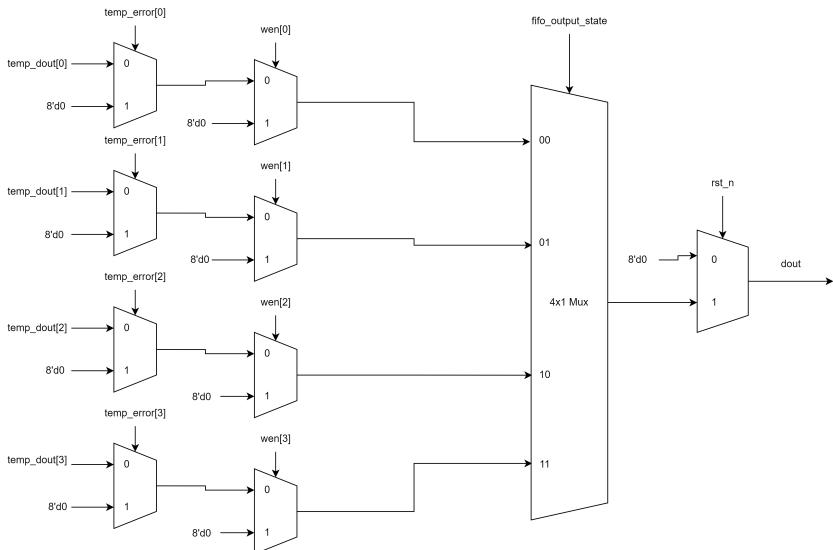


- output_generator(always block)

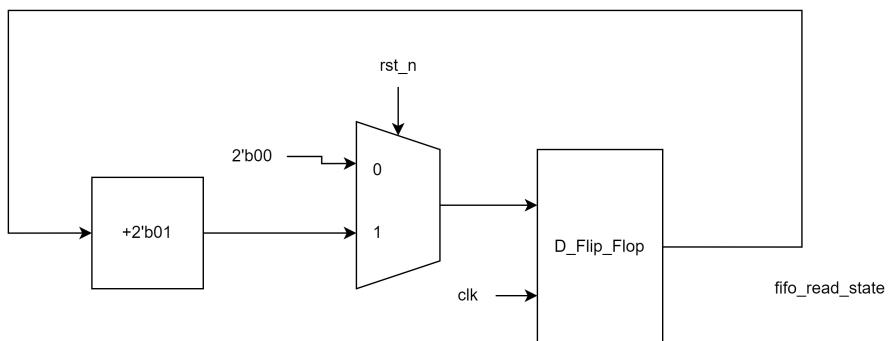
- valid



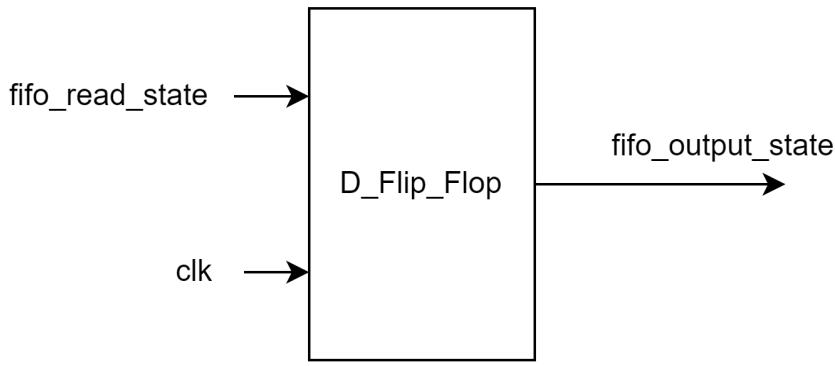
- dout



- fifo_read_state | use in ren_generator

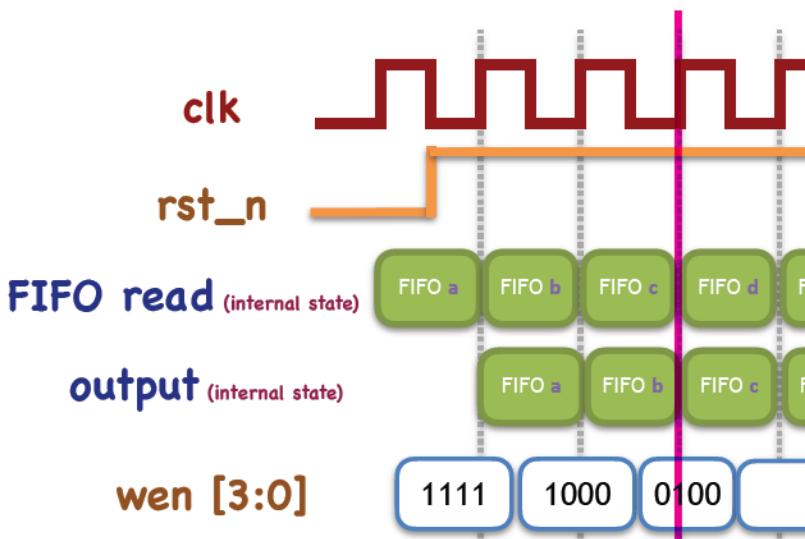


- fifo_output_state | use in output_generator



Requirements

1. 利用在 advanced question 2 實做出的 FIFO，實做出 FIFO*4 輸出 (8-bit output)*4，各別間 independent
2. 將輸出的 (8-bit output)*4 丟入一個 Round Robin Arbiter，以此控制各別的 ren 信號
3. wen 信號由外部提供；FIFO*4 的 input data : a、b、c、d 也是由外部提供
4. 而有2項 invalid 的特殊狀況要特別處理，
◦ 某 FIFO 被 arbiter 訪問到，但卻在讀取 (wen[number]=1)時，視為無效。如下圖：



解釋：FIFO c 在此時被 arbiter 訪問到，但因同時在做
讀取的動作，此訪問將視為無效。

- arbiter 訪問到的 FIFO 輸出之 error = 1 時，視為無效。

※ 處理方式

- a. valid = 0
- b. dout = 原本 dout (不從FIFO中讀取任何值)

5. 其他讀取成功的case : valid = 1 / dout = 從FIFO中讀取值之新 dout

Design Explanation

PS. Round Robin Arbiter簡稱為RRA

我們的設計將拆分為 sequential & combinational 者:
[簡述兩者使用到的參數]

1. sequential 實踐更新 fifo_read_state & fifo_output_state
 - fifo_read_state : Round Robin Arbiter 輪到要 read 的項
 - fifo_output_state : Round Robin Arbiter 輪到要 output 的項
 - ✖ 以上兩者 fifo_read_state / fifo_output_state 區分 read / output 兩者選取的 FIFO 項

2. combinational

- ▷ module:ren_generator
 - a. 在 fifo_read_state 或 wen 變更時，更新 read signal
- ▷ module:output_generator
 - b. 在 temp_dout 或 temp_error 變更時，更新 output (包括valid, dout) 值

[詳述模組設計想法]

1. 以 sequential 的方式，依據時脈訊號 clock 上升瞬間更新以下值，
 - rst_n = 0 : 初始化。
此時 RRA 選取 FIFO a 為 read 項(設定 fifo_read_state = 2'b00)
 - rst_n = 1 : 更新，不論 read / output，皆將所選取的 FIFO 輪為下一個，做兩件事：
 - a. 將 fifo_read_state 加 2'b01
 - b. fifo_output_state 為原本的 fifo_read_state

2. 以 combinational 的方式更新以下值，

- ▷ module : ren_generator

- step1 : fifo_read_state 的訊號值可得 RRA 輪到要 read 的 FIFO 項為何
- step2 : 判斷此 FIFO 是否正在 write
 - 「是」則設此項 read = 0 · 代表不可讀
 - 「否」則設此項 read = 1 · 代表可讀

除了此 FIFO 項的 read 可能設成 1 之外 · 其餘 3-bit read 項為 0 · 可得 4-bit ren 值。

PS. 整理所有 ren 不為 4b'0000 的狀況於以下[表1] · 其餘代表全部 FIFO 都不可讀。

fifo_read_state	wen	ren
0 (a)	wen[0]==0	4'b0001
1 (b)	wen[1]==0	4'b0010
2 (c)	wen[2]==0	4'b0100
3 (d)	wen[3]==0	4'b1000
其餘所有組合		4'b0000

⌚ module : output_generator

- step1 : fifo_output_state 的訊號值可得 RRA 輪到要 output 的 FIFO 項為何
- step2 : 判斷此 FIFO 是否正在 write
 - 「是」則設此項 invalid => dout = 0
 - 「否」則進入 step3
- step3 : 判斷此 FIFO 的 error 項是否為 1 (read/write is issued to empty/full FIFO)
 - 「是」則設此項 invalid => dout = 0
 - 「否」則 valid => 更新 dout

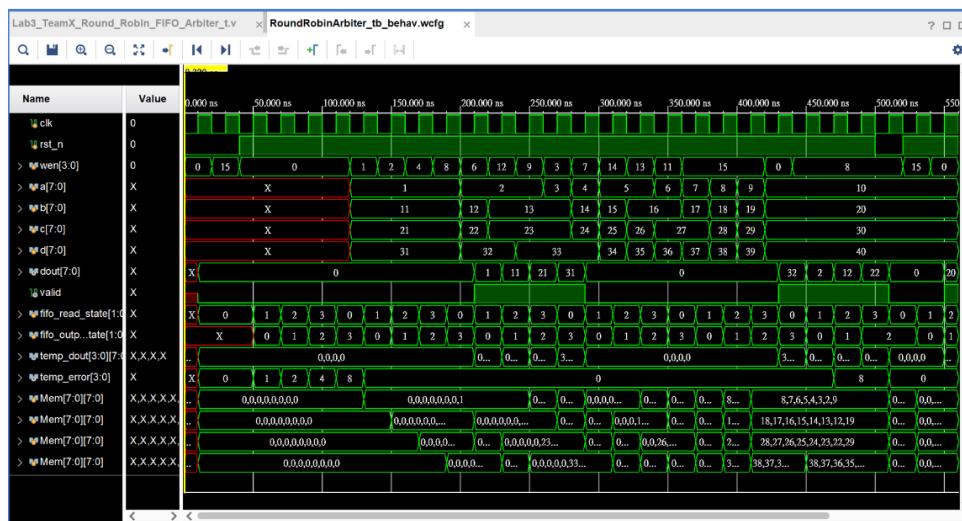
PS. 整理所有非 invalid 的狀況於以下[表2] · 其餘 invalid 的項 dout = 0 。

(以下將 **fifo_output_state** 簡寫為 **state** / **temp_error** 簡寫為 **t_e** / **temp_out** 簡寫為 **t_o**)

state	wen	t_e	dout	valid
0 (a)	wen[0]==0	t_e[0]==0	t_o[0]	1
1 (b)	wen[1]==0	t_e[1]==0	t_o[1]	1
2 (c)	wen[2]==0	t_e[2]==0	t_o[2]	1
3 (d)	wen[3]==0	t_e[3]==0	t_o[3]	1
其餘所有組合			8'b0	0

Testbench Design & Result Explanation

波形圖截圖



TESTBENCH設計說明

起初初始化如下並會在一開始維持一個 cycle :

- I. clk = 0 → 初始 clock 於 0
- II. rst_n = 0 → 啟動 reset 機制
- III. wen = 4'b0000

接著：

1. rst_n = 0 / wen = 4'b1111 的狀況維持一個 cycle

- rst_n = 0 : 進行 reset
- wen = 4'b1111 : 4-bit 皆拉起，先用圖中 Mem signal 確認 FIFO a,b,c,d 尚未把任何值寫入

2. rst_n = 1 / wen = 4'b0000 的狀況維持四個 cycle

- wen = 4'b0000 : 保持尚未寫入值到任何 FIFO 中的狀態

3. wen 以四個 cycle 分別設定成 4'b0001 / 4'b0010 / 4'b0100 / 4'b1000，檢查 write to FIFO 的情況，input 保持 $a = 8'd1$ / $b = 8'd11$ / $c = 8'd21$ / $d = 8'31 \Rightarrow$ 檢查 FIFO 空，read empty FIFO a/b/c/d 之 error 特殊情況會使得 invalid & dout = 0

- ⇒ 檢查 FIFO a write data + read b + output a 的情境
 - wen = 4'b0001 : 將 $a = 1$ 寫入 FIFO a
 - read b -> 此時 FIFO b 為空
 - output a -> 此時 FIFO a 正在被寫入，所以 invalid。

(各 cycle 詳細說明)

- ⇒ 檢查 FIFO b write data + read c + output b 的情境
 - wen = 4'b0010 : 將 $b = 11$ 寫入 FIFO b
 - read c -> 此時 FIFO c 為空
 - output b -> 此時 FIFO b 正在被寫入，所以 invalid。
- ⇒ 檢查 FIFO c write data + read d + output c 的情境
 - wen = 4'b0100 : 將 $c = 21$ 寫入 FIFO c
 - read d -> 此時 FIFO d 為空
 - output c -> 此時 FIFO c 正在被寫入，所以 invalid。
- ⇒ 檢查 FIFO d write data + read a + output d 的情境
 - wen = 4'b1000 : 將 $d = 31$ 寫入 FIFO d
 - read a -> 讀取 $a = 1$
 - output d -> 此時 FIFO d 正在被寫入，所以 invalid。

PS. 為方便辨識 testbench 是否正確，讀取值有特別設計過，如以下：

FIFO Memory	值
FIFO a 第n個值	n
FIFO b 第n個值	1n
FIFO c 第n個值	2n
FIFO d 第n個值	3n

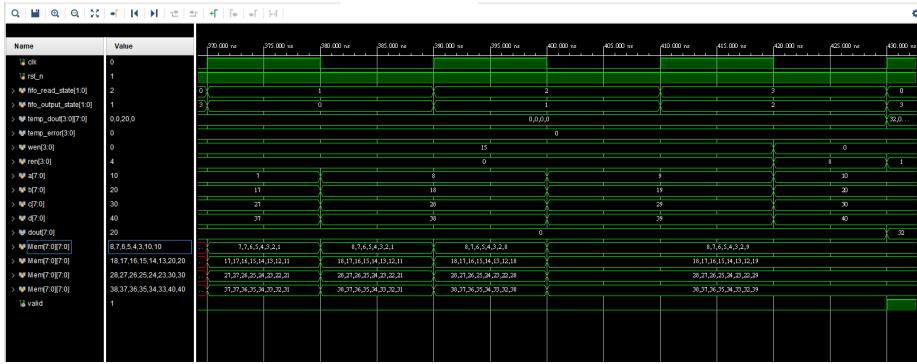
4. 依序進行以下：

- ⇒ 檢查 FIFO b/c write data + read b + output a 的情境

- wen = 4'b0110 : 將 b=12/c=22 寫入 FIFO b/c
 - read b -> 讀取 b = 11
 - output a -> 先前 read 到的 a = 1 輸出
- 檢查 FIFO c/d write data + read c + output b 的情境
 - wen = 4'b1100 : 將 c=23/d=32 寫入 FIFO c/d
 - read c -> 讀取 b = 21
 - output b -> 先前 read 到的 b = 11 輸出
- 檢查 FIFO a/d write data + read d + output c 的情境
 - wen = 4'b1001 : 將 a=2/d=33 寫入 FIFO a/d
 - read d -> 讀取 b = 31
 - output c -> 先前 read 到的 c = 21 輸出
- 檢查 FIFO a/b write data + read a + output d 的情境
 - wen = 4'b0011 : 將 a=3/b=13 寫入 FIFO a/b
 - read a -> 讀取 a = 2
 - output d -> 先前 read 到的 d = 31 輸出
- FIFO a/b/c write data + read b + output a 的情境
 - wen = 4'b0111 : 將 a=4/b=14/c=24 寫入個別 FIFO
 - read b -> 讀取 b = 22
 - output a -> 此時 FIFO a 正在被寫入，所以 invalid。
- FIFO b/c/d write data + read c + output b 的情境
 - wen = 4'b1110 : 將 b=15/c=25/d=34 寫入個別 FIFO
 - read c -> 讀取 c = 32
 - output b -> 此時 FIFO b 正在被寫入，所以 invalid。
- FIFO a/c/d write data + read d + output c 的情境
 - wen = 4'b1101 : 將 a=5/c=26/d=35 寫入個別 FIFO
 - read d -> 讀取 d = 42
 - output c -> 此時 FIFO c 正在被寫入，所以 invalid。
- FIFO a/b/d write data + read a + output d 的情境
 - wen = 4'b1011 : 將 a=6/b=16/d=36 寫入個別 FIFO
 - read a -> 讀取 a = 3
 - output d -> 此時 FIFO d 正在被寫入，所以 invalid。

依循 wen/ read/ output 的查驗方法得證設計出的所有
testbench 皆無誤，以下不再列舉，講述情境為主

- ⇒ FIFO a/b/c/d write data + read b + output a 的情境
- ⇒ FIFO a/b/c/d write data + read c + output b 的情境
- ⇒ FIFO a/b/c/d write data + read d + output c 的情境 ⇒ 檢查 FIFO 已滿，無法再 write 之 error 特殊情況會使得 invalid & dout = 0



- ⇒ 不寫入值到任何 FIFO 中 + read a + output d 的情境 ⇒ valid!=0
- ⇒ FIFO d write data + read b + output a 的情境 ⇒ output 項不為雖滿卻要寫入的 d 項，因而 valid!=0
- ⇒ FIFO d write data + read c + output b 的情境 ⇒ 同上
- ⇒ FIFO d write data + read d + output c 的情境 ⇒ 同上

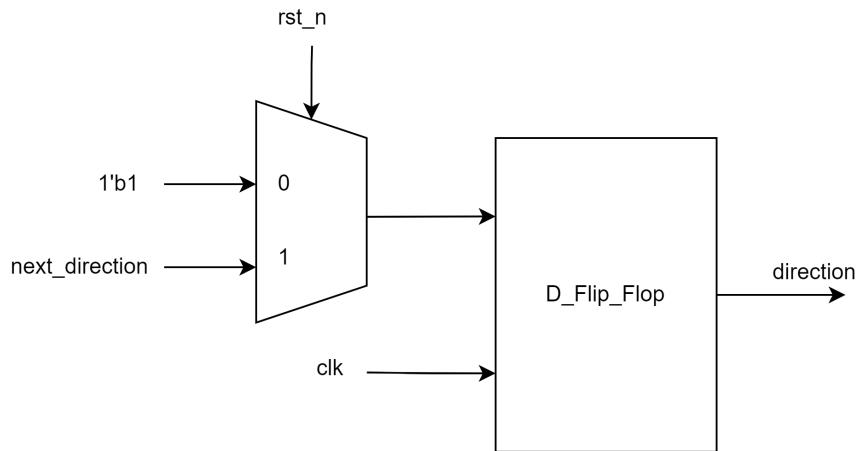
以上所有的狀況都檢查成功，符合預期✓

5. 再次 reset ⇒ 檢查 reset 狀況，呈現 dout = 0 & valid = 0，符合預期✓
6. 設定 wen = 4'b1111 & input 為 a = 10 / b = 20 / c = 30 / d = 40 ⇒ 檢查是否有寫入，從 Mem 可得結果符合預期 ✓
7. 設定 wen = 4'b0000 & input 為 a = 10 / b = 20 / c = 30 / d = 40 ⇒ 檢查上一個 cycle 寫入的值是否能正確被讀出來。因為經過 reset 之後，read_ptr, write_ptr 又都被歸零，會指向 FIFO 的第一個位置，所以在前一個 cycle 寫入的值應位於各 FIFO 的第一個位置，且能在此步驟進行第一次讀出時讀出來，符合預期✓

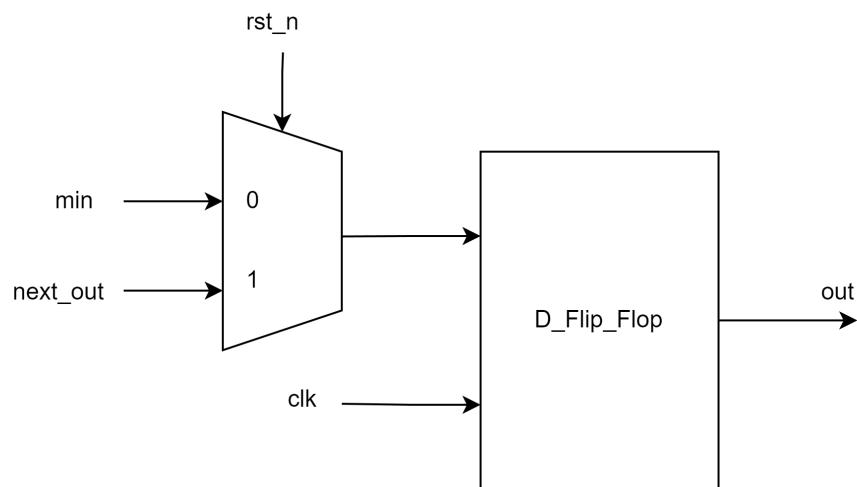
Advanced Question5

Drawing of the design of Verilog Advanced Question 5

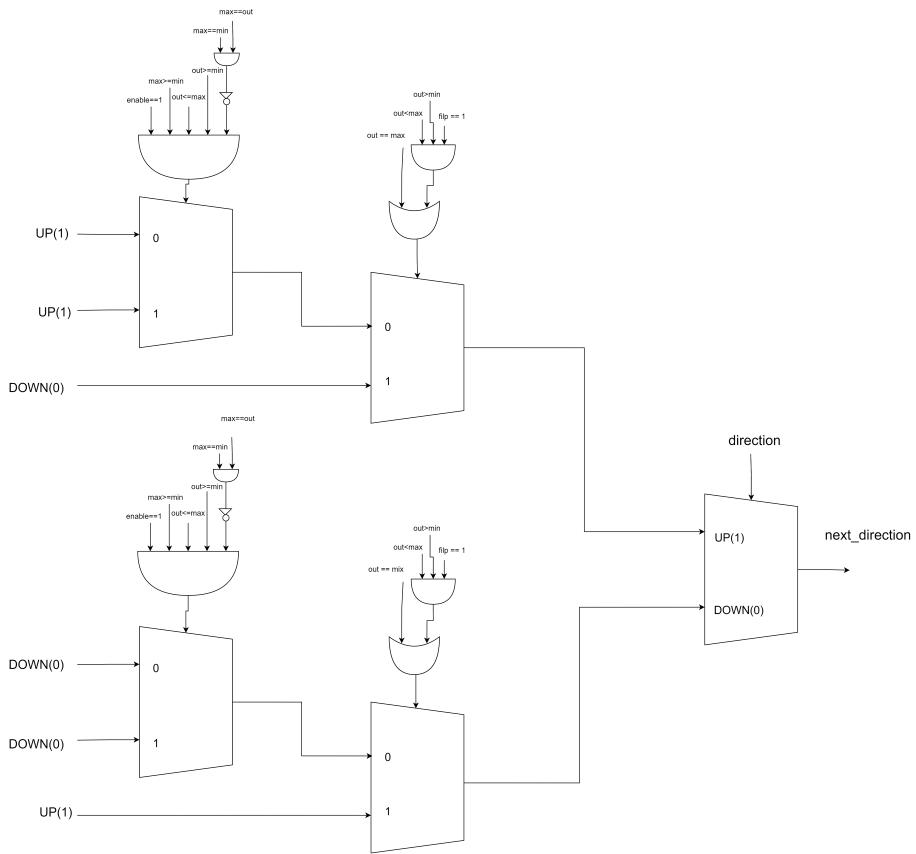
- direction



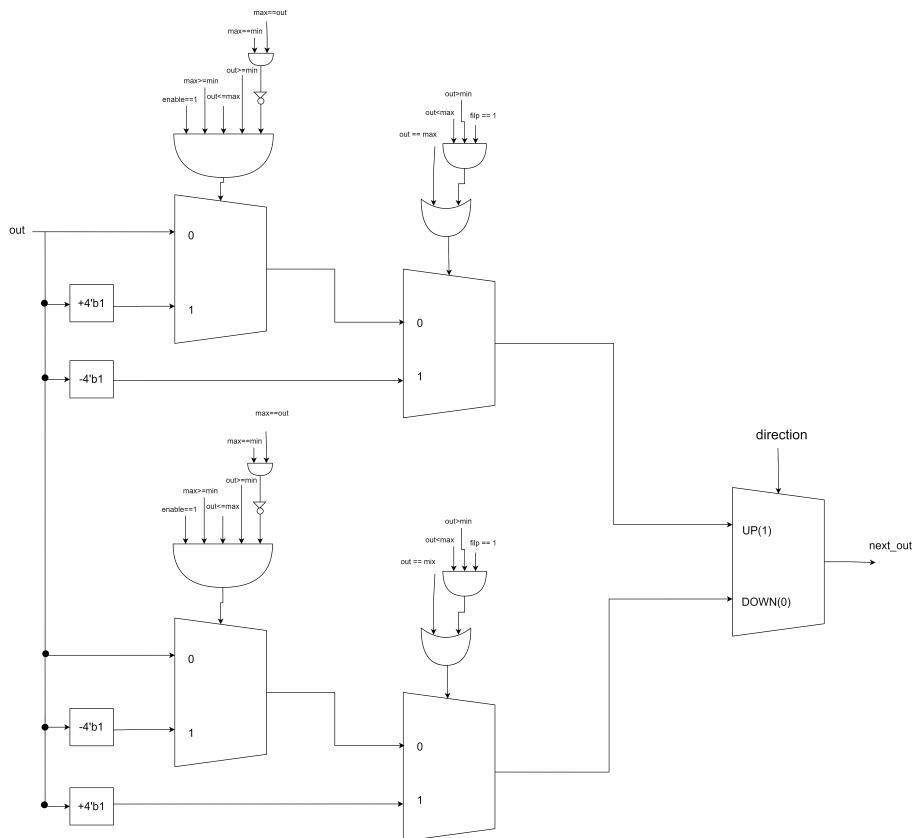
- out



- next_direction generation(always block)



- next_out generation(always block)



Requirements

(相似於advanced question1)

1. rst_n

- $\text{rst_n} = 0 \rightarrow$ 初始化
 - a. $\text{direction} = 1'b1$
 - b. $\text{out} = \text{min}$

2. enable

- $\text{enable} = 1$
 - counter begins its operation · that is
 - a. $\text{direction} =$ 更新後的 direction
 - b. $\text{out} =$ 更新後的 out
- $\text{enable} = 0$
 - counter holds its current value
 - a. $\text{direction} =$ 不更新 direction
 - b. $\text{out} =$ 不更新 out

3. max and min

- $\text{max} < \text{min}$
 - counter begins its operation
- $\text{max} > \text{min}$ (不合理狀況!)
 - counter holds its current value
- $\text{counter} > \text{max}$ 或 $\text{min} > \text{counter}$ (不合理狀況!)
 - counter holds its current value

4. filp

- $\text{filp} = 1$
 - counter flips its direction · that is
 - a. $\text{direction} =$ 反向的 direction
 - b. $\text{out} =$ 更新後的 out

Design Explanation

我們的設計將拆分為 sequential&combinational 兩者:

1. 以 sequential 的方式 · 依據時脈訊號 clock 上升的瞬間更新 output(out & direction) 的值 ·
 - $\text{rst_n} = 0$: 初始话 out & direction
 - $\text{rst_n} = 1$: 按照以上 requirements · 更新 out & direction
2. 以 combinational 的方式更新 output(out & direction) 的值 ·

- 原方向為 UP/DOWN 且符合以下：

- ① enable = 1
 - ② max >= min
 - ③ out <= max
 - ④ out >= min
 - ⑤ out = max = min

則開始做變化，否則維持在前一個 cycle 的 output 值。

- A、做變化情況如下：

■ UP的狀況下

- a. $\text{out} = \text{max} / \text{filp} = 1$ (且符合 out 在目標區間中)
: counter begins its operation · that is $\text{out} -= 1$
 $\& \text{direction} = \sim \text{direction}$
 - b. 包含 $\text{out} != \text{max} \& \text{filp} != 1$ 之其他狀況：
counter begins its operation · that is $\text{out} += 1$
 $\& \text{direction} = \text{direction}$

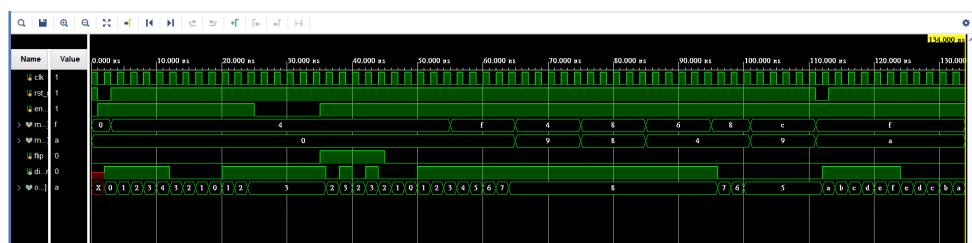
■ DOWN 的狀況下

- a. $\text{out} = \text{min} / \text{filp} = 1$ (且符合 out 在目標區間中) :
counter begins its operation · that is $\text{out} -= 1$
 $\& \text{direction} = \sim \text{direction}$
 - b. 包含 $\text{out} != \text{min} \& \text{filp} != 1$ 之其他狀況 :
counter begins its operation · that is $\text{out} += 1$
 $\& \text{direction} = \text{direction}$

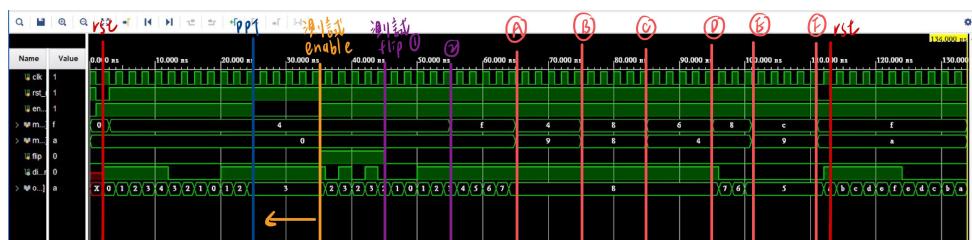
- B、不做變化，維持在前一個 cycle 的 output (out / direction) 值

Testbench Design & Result Explanation

原圖



+ 註解



1. rst_n 降下來的狀況維持一個 cycle，進行 reset

- rst_n = 0
- enable = 1

純粹測試 reset 狀況，out 被設定為 min 值 0 & direction 被設定為 1'b1，符合預期√

2. rst_n 拉起，設定 max/min 與投影片相同做初步觀察

- rst_n = 1
- max 設成 4
- min 設成 0

結果與給的 waveform 相同，out 呈現 0、1、2、3、4、3、2、1、0、1、2、3，符合預期√

3. 測試 enable 的切換

- enable = 0

4. 測試 flip 的切換

- ① flip = 1 維持五個 cycle，out 呈現 2、3、2、3、2 & direction 呈現 0、1、0、1、0，為彈掉現象
- ② flip = 0 維持五個 cycle，out 呈現 2、1、0、1、2 & direction 呈現 0、0、0、1、1，為不彈跳繼續 count 的現象

-> 以上兩種情況皆符合預期√

5. 跑不同 min, max 值(各維持 5 個 cycle time 觀察)

- Ⓐ min = 0 & max = 15 的情況，out 持續往上加，呈現 4、5、6、7、8 & direction 呈現 1、1、1、1、1
- Ⓑ min = 9 & max = 4，是特殊情況 “max > min” 的測試，持續呈現 out = 8 & direction = 1，維持原值
- Ⓒ min = 8 & max = 8，是特殊情況 “max == min == out” 的測試，持續呈現 out = 8 & direction = 1，維持原值
- Ⓓ min = 4 & max = 6，是特殊情況 “counter > max” 的測試，持續呈現 out = 8 & direction = 1，維持原值
- Ⓔ min = 4 & max = 8，回歸一下正常狀況，呈現 7、6、5 & direction 呈現 0、0、0
- Ⓕ min = 12 & max = 9，是特殊情況 “counter < min” 的測試，持續呈現 out = 5 & direction = 0，維持原值

-> 以上六種正常或特殊情況皆符合預期✓

6. 再次 `rst_n` 降下來的狀況維持一個 cycle，進行 reset

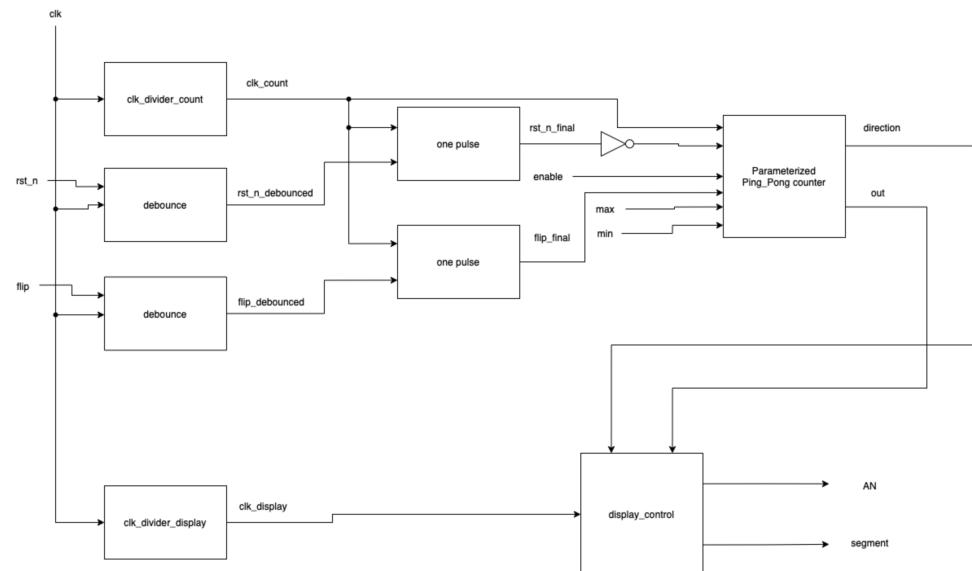
-> `out` 再度被設定回 min 值 10 & `direction` 被設定為 1'b1，符合預期✓

7. `rst_n` 拉起，繼續觀察

-> `out` 呈現 10、11、12、13、14、15、14、13、12、11、10 & `direction` 呈現 1、1、1、1、1、1、0、0、0、0、0，符合預期✓

FPGA Demonstration 1

Drawing of the design of FPGA Demonstration 1



Design Explanation

分成 Counter&Display 兩塊做設計，並將所需使用到的物件模組化，如以下：

1. Parameterized_Ping_Pong_Counter

使用到 advanced question 5 設計的 4-bit Parameterized Ping-Pong Counter，為 Counter。

2. display_control

在這個 function 中，控制 AN 與 7-bit segment 的顯示，為 Display。

- 在 AN 的部分，由於我們要讓四個 digit 都亮起，而在硬體上的實作是利用視覺暫留的方式讓他輪流亮，看起來就像是一直一起亮，所以設定 clock period，

一次亮起一個 digit，實踐讓他看起來像同時 display 的方法，否則同時只會都顯示同一個數字。

現階段的 AN	下個 clock 的 AN
0111	1011
1011	1101
1101	1110
1110	0111

- 在 7-bit segment 的部分，因四個 digit 在每個 clk 要顯示的各自不同，所以我們使用 case(AN) 來設計，表明在此 clk 此 AN 所應顯示的：

- 最左邊的 digit (AN = 4'b0111)

此位置顯示計數器「十位數」的部分，而 4-bit 的 max&min 會讓計數器顯示 0 到 15，因此在十位數的部分至多只會顯示到 1，也就是只有 0、1 兩個可能。我們的設計是當計數器值大於 10，代表有進位而要顯示 1，此位置輸入訊號 7'b1001111；若沒有進位，要顯示 0，此位置輸入訊號 7'b0000001。

- 第二位 digit (AN = 4'b1011)

此位置顯示計數器「個位數」的部分，而 4-bit 的 max&min 會讓計數器顯示 0 到 15，因此在個位數的部分 0~9 都有可能會顯示。因此按照 16 個計數器可能值，分別對應到相對要顯示的個位數字，如以下：

計數器 (counter)	第二位 digit 顯示 (seven_segment)
0(4'b0000)	7'b0000001
1(4'b0001)	7'b1001111
2(4'b0010)	7'b0010010
3(4'b0011)	7'b0000110
4(4'b0100)	7'b1001100
5(4'b0101)	7'b0100100
6(4'b0110)	7'b0100000
7(4'b0111)	7'b0001111
8(4'b1000)	7'b0000000
9(4'b1001)	7'b0001100
10(4'b1010)	7'b0000001
11(4'b1011)	7'b1001111
12(4'b1100)	7'b0010010
13(4'b1101)	7'b0000110
14(4'b1110)	7'b1001100
15(4'b1111)	7'b0100100

- 第三/四位 digit (AN = 4'b1101/4'b1110)

這兩個位置顯示計數器方向 (direction) 的部分，因此如果計數器往上數 (direction = 1)，則按照投影片所顯示的，兩者皆應設為 7'b0011101；而如果計數器往下數 (direction = 0)，兩者則按照投影片所顯示的，皆應設為 7'b1100011。

- 此外，加上 defalut 設定為 7'b1111111 以確保 case 的穩固，會在所有項目都不符合 (AN為四者以外的情況) 下執行。

3. clk_divider_count → 用於

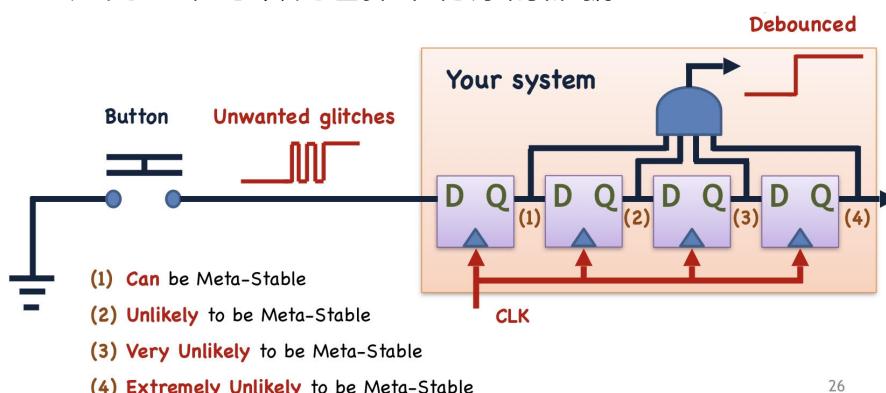
Parameterized_Ping_Pong_Counter module 中是給 Counter 的 clock，輸入 clk，輸出頻率為 $1/2^{25}$ clk 的 clk_count，用於

Parameterized_Ping_Pong_Counter 的計數及 one_pulse 的處理。(原本 Basys 3 的 clock 為 100MHz (= 10ns per clock cycle))

4. clk_divider_display → 用於 display_control module 中是給 Display 的 clock，輸入 clk，輸出頻率為 $1/2^{17}$ clk 的 clk_display，利於 display_control 的近似同步顯示。

5. debounce → 一個 clock 最多處理一次 button(包括 rst_n 及 flip)按下的情況

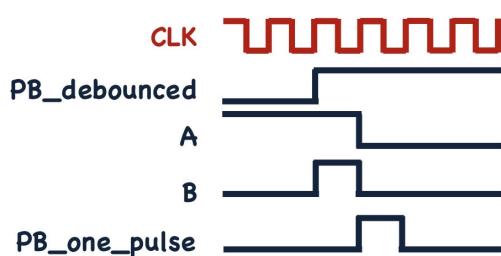
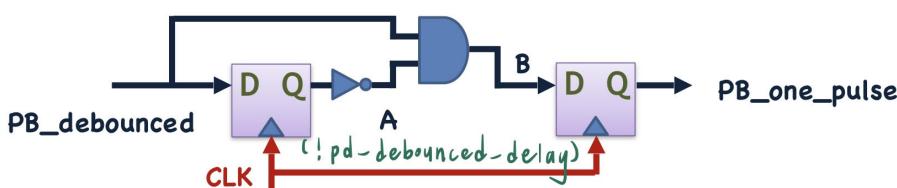
因 FPGA 使用到 push buttons 的方式，因此將老師上課所教的 Debounce Circuit 機制(Lecture 4 p.27)應用於此，透過宣告一連串的 DFF，將 register 層層傳遞來處理訊號的 meta-stability 問題，只有在全是 1 才是 1、全是 0 才是 0，如此一來可以處理掉不乾淨的訊號。



26

6. one_pulse → button(包括 rst_n 及 flip)按下的情況要維持一個 clock

延續 FPGA 使用到 push buttons 的方式，而當 button 被按一次必須要延續一個 clock-cycle，因此將老師上課所教的 One Pulse Circuit 機制(Lecture 4 p.30)應用於此。



29

I/O pin assignment

A ` Clock (clk)

Input	package pin
clk	W5

B ` Switches (enable ` max ` min)

enable : SW[15]

Input	package pin
enable	R2

max : SW[14:11]

Input	package pin
max[0]	R3
max[1]	W2
max[2]	U1
max[3]	T1

mix : SW[10:7]

Input	package pin
min[0]	W13
min[1]	V2
min[2]	T3
min[3]	T2

B ` Buttons (DOWN for flip / UP for rst_n)

Input	package pin
flip	U17
rst_n	T18

C ` 7-segment display

Output	package pin
AN[0]	U2
AN[1]	U4
AN[2]	V4
AN[3]	W4
segment[0]	U7
segment[1]	V5
segment[2]	U5
segment[3]	V8
segment[4]	U8
segment[5]	W6
segment[6]	W7

Contribution

朱季葳：advanced question 1、2、3; code of advanced question 4 & FPGA

施泳瑜：advanced question 5; report of advanced question 4 & FPGA

What we have learned from Lab3

- 在第二題的部分，原本我們打算比較read_ptr與write_ptr的值來判斷是否read empty or write full，但是後來發現這樣的寫法不但容易遺漏掉很多種不同的情況，又會讓判斷式變得非常雜亂，所以就捨棄掉這樣的做法。而且其實題目已經給了很明顯該怎麼做的答案：總共有8個位置，read empty就代表沒有位置是被佔據的，所以紀錄已填滿的位置自然為0，反之，當以填滿的位置到達上限8的時候，我們要再寫入就是不被允許的。而用這樣的紀錄方式也使我們的code變得更加簡潔易懂，也更好寫。
- 在第四題的部分，時序問題的解決相對困難，因為 FIFO read / output 會 delay 一個 clock cycle，所以在處理上我們加入兩個 state - fifo_read_state & fifo_output_state 分別去做判斷，還有善用sensitivity list使得只有在某些特定wire的訊號改變時，會wake up always block，讓

output register裡面所存的值得已更新。透過以上較能清楚處理所遇上的時序問題而完成這題作業。

3. 在fpga題的部分，一開始因為沒有注意到reset訊號的問題所以卡了一段時間，後來才想起按下鍵所代表的含義是reset但是我們在parameterized ping pong counter所實作的是rst_n，所以在把reset wire接進去module之前要加一個inverter，但是由於一開始的時候忘記這件事情，所以把代表reset訊號的wire也取名叫做rst_n了qq，後來因為板子成功燒起來了，所以就想說不要再做更動了，下次如果遇到的話會記得改成reset的。
4. 在這次的作業中包括第三題還有第四題實作的部分，sensitivity list真的幫助了我們解決了許多時序相關的問題，也使得我們的code簡潔了許多。