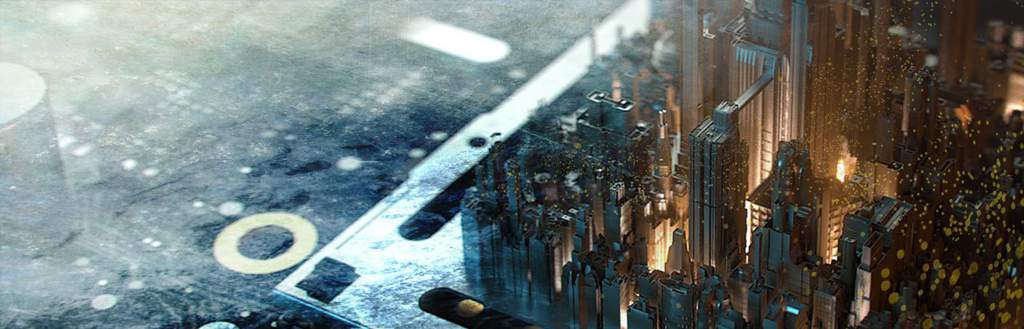
****

**Lab 3 report**

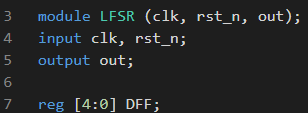
**2020/10/22**

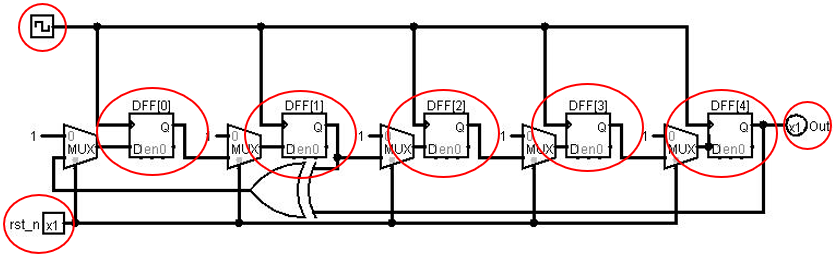
**108062125 高敦晉**

**108062229 陳皇佑**

**Advanced Question 1**

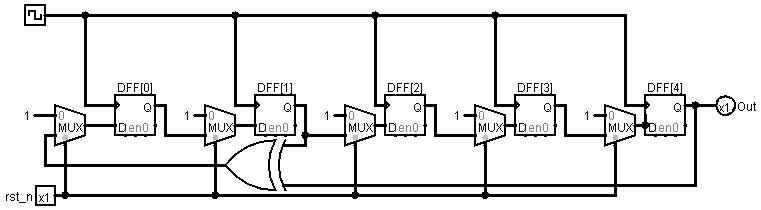
**Explanation Verilog Module & Block Diagram**



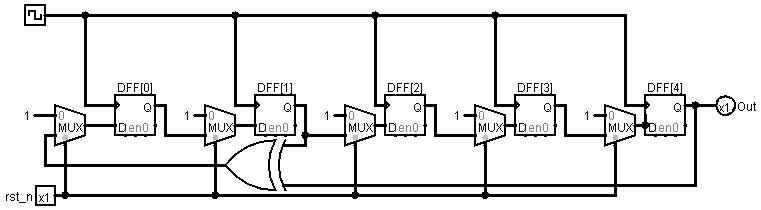


這裡 input/output 基本跟給的模板一樣。

用五個 reg 來處理 D-flipflop的動作。

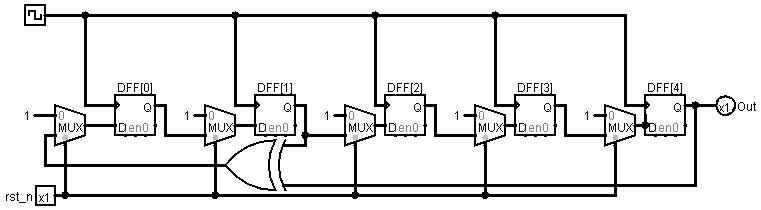
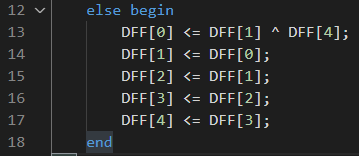


Posedge trigger。這東西指的是Clock信號。



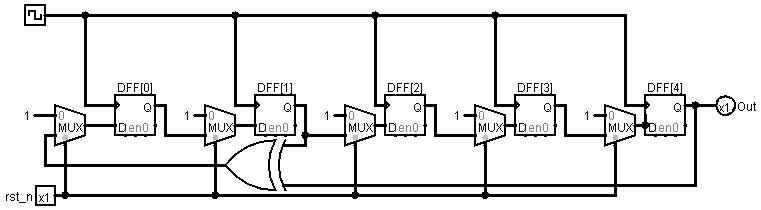
依照題目的定義，在rst\_n 為 0時，將所有D-flipflop重置為0

圖以DFF[0]為例。



每個D-flipflop 前面都接一個MUX，以rst\_n做為selector，rst\_n 為 0時傳入 1，rst\_n為1時傳入上一個D-flipflop的值。

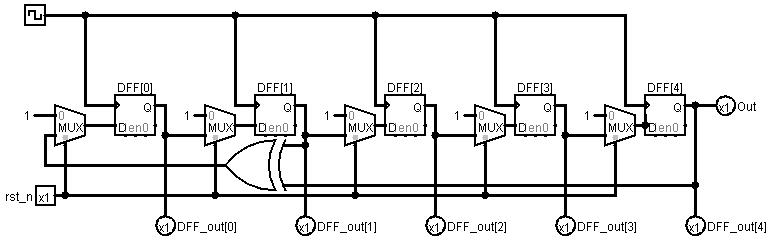
除了 DFF[0] 以外的 D-flipflop都是將上一個 D-flipflop的值接下來MUX的1位置，DFF[0]前的MUX1的位置則是用DFF[1] 以及 DFF[4]的值做XOR得到。



在DFF[0] 前面的MUX，0 的位置接的是DFF[4]和DFF[1]的XOR值

為了能夠在testbench看裡面的D-flipflop信號

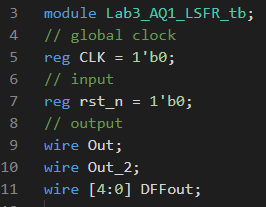
我後來又做了一個LFSR\_lookinside的module





整體上跟LFSR一模一樣，只差在多拉每個D-flipflop的信號出來做Output。

**How to test design**



宣告基本要用的reg, wire

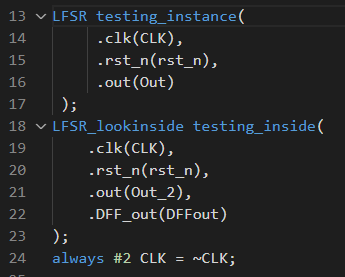
CLK -> clock

rst\_n -> Reset信號

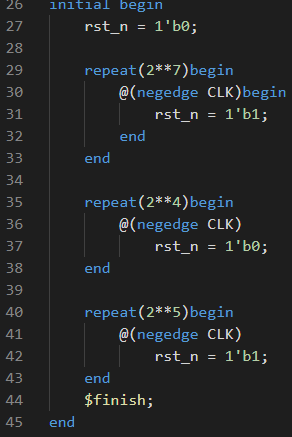
Out -> 第一個module的Output

Out\_2 -> 第二個module的Output

DFFout -> 第二個module的D-flipflop信號拉出來



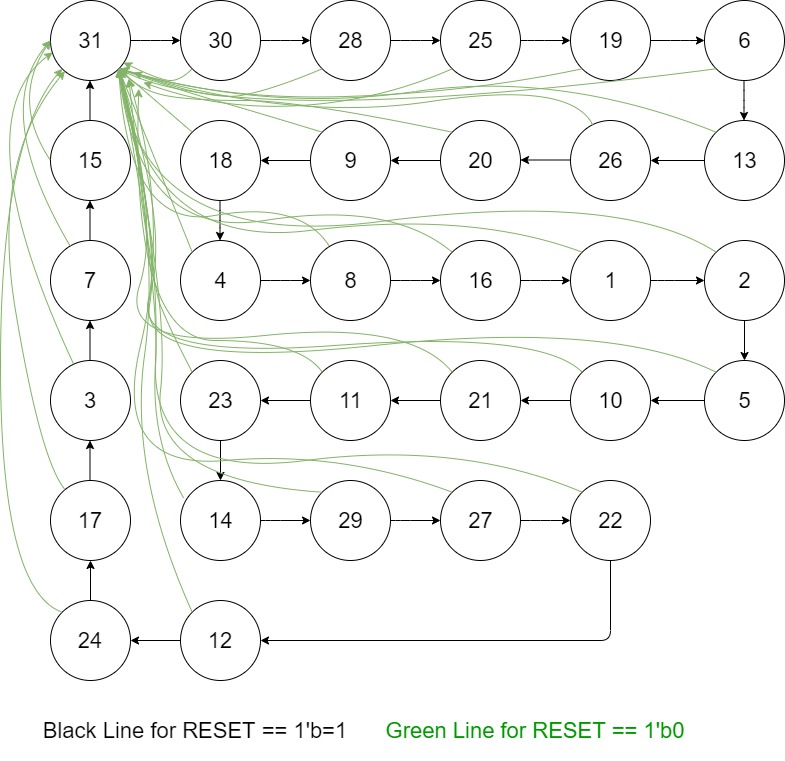
接好module 以及設置好 CLK 讓他每 #2跳一次



這部分其實就是讓他在1，然後看一下各個state Transition的狀況，然後讓他reset，在跳一次，確認沒問題就好

**State Transition Diagram**

下面附上State Transition Diagram，轉成十進制

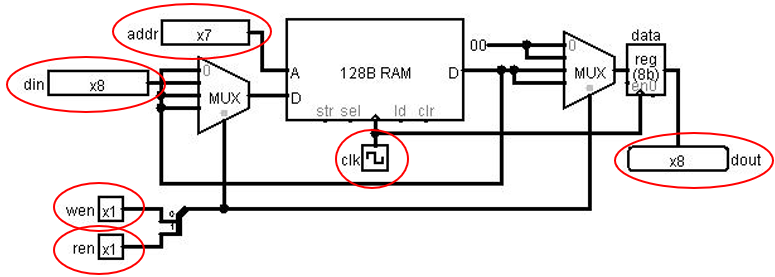
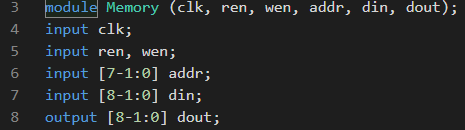


此外有測過RESET = 0時，將DFF[4:0]都設成0時，它的state transition就都會一直在0。

這個狀況蠻好理解的，因為這個design裏頭，能夠帶給他們D-flipflop新氣象的只有接在DFF[0] 上面的XOR，或是reset成11111，但是現在reset變成了00000，0 XOR 0 又是 0，因此state就hold在0上面了。

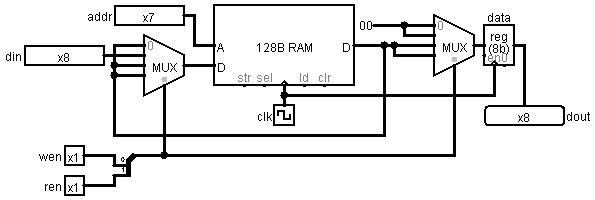
**Advanced Question 2**

**Explanation Verilog Module & Block Diagram**

**** ****

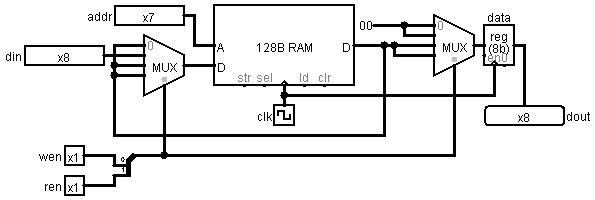
基本的input/output

****

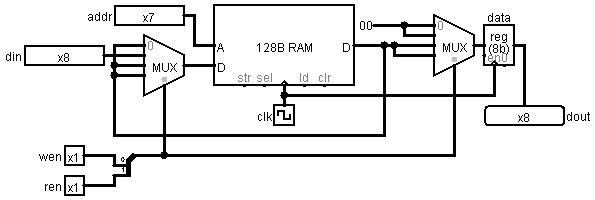
****

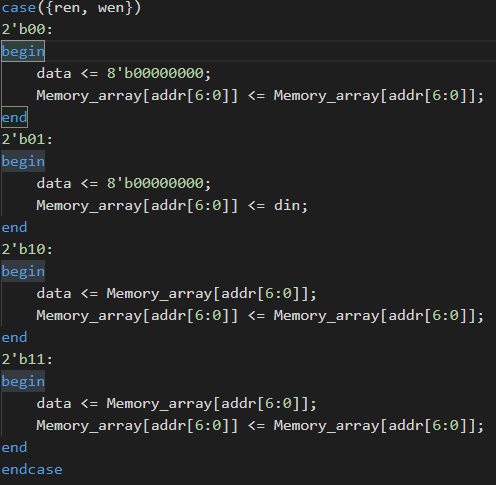
Memory\_array -> 記憶體們，用了128個8bit的記憶體

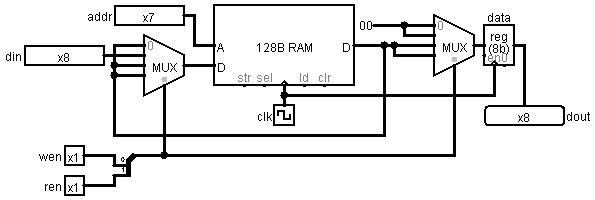
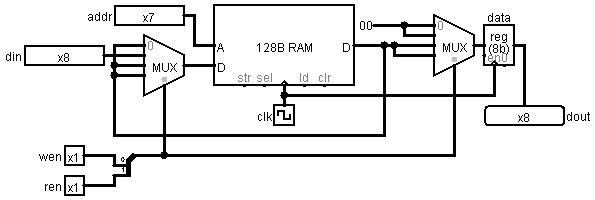
A-> address, 左邊的D->data input(write), 右邊的D-> data output(read), 下方中間->clock, str -> write if 1, ld -> read if one(沒接的話就是又讀又寫)，不過我後面用MUX解決這件事，後面解釋)

****

Data -> 用來輸出dout用的8bit reg。

****這張是指將兩個1 bit的wire合成一個2bit的wire線，沒有實質上的verilog code，僅是畫圖時用的示意圖而已。

,

****

做個case，然後判別ren, wen的四種組合，分別接再MUX上

這裡Memory\_array是從右邊的D接回來的，將自己寫write給自己，等同於沒有變。

00 : Do nothing data 設成 8’b00000000，Memory\_array不變接給自己

01 : Write, data 設成 8’b00000000，Memory\_array接給din

10 : Read, data 設成要讀取的Memory array, Memory\_array接給自己

11 : Read, data 設成要讀取的Memory array, Memory\_array接給自己

其實我最一開始寫的版本，是

00 : Do nothing data 設成 8’b00000000

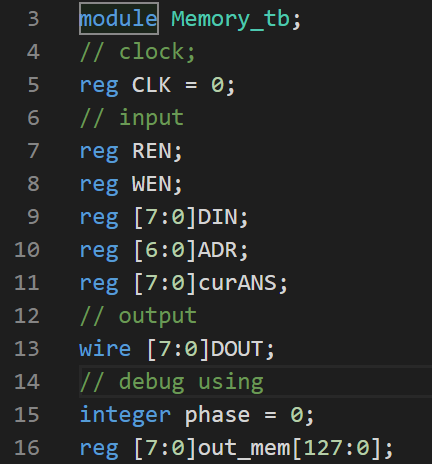
01 : Write, data 設成 8’b00000000，Memory\_array接給din

10 : Read, data 設成要讀取的Memory array

11 : Read, data 設成要讀取的Memory array

想說該階段沒有用的的register，就不特地聲明它要不變，後來上iLMS問過，且回去翻過講義後，了解到為了避免合成出unintended latch，要把if-else寫好，所以特別寫成了case，把每個狀況聲明清楚。

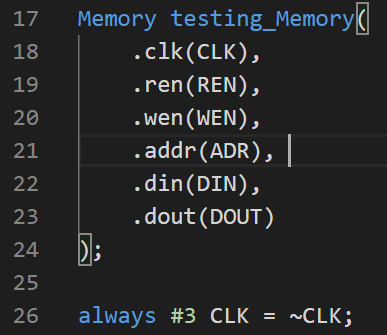
**How to test design**



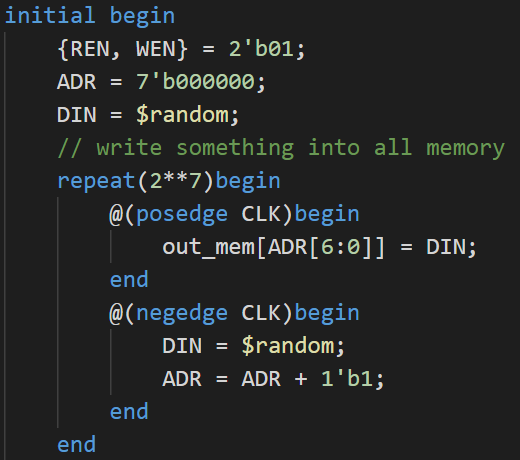
一些需要用到的reg，input/output基本上跟module差不多。

Out\_mem -> 在testbench，也就是外面模擬另一個128bytes的記憶體(128個8bit)，用來check 裡面外面的記憶體是否一樣，有沒有哪裡沒改到。

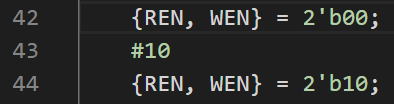
Integer phase -> debug用的資訊。



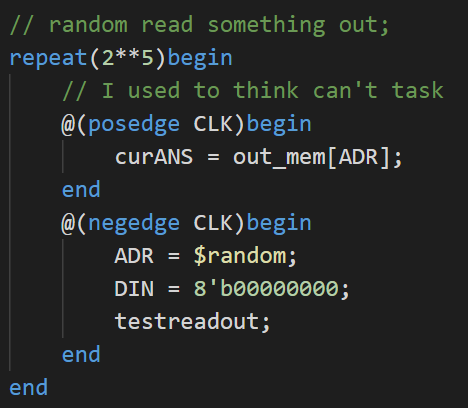
將module接好，CLK設定成#3一跳。



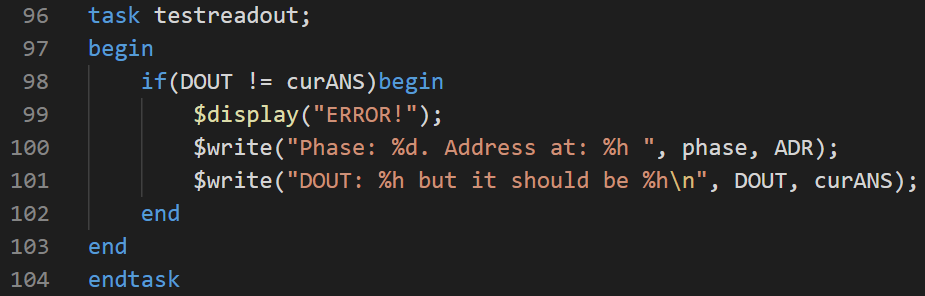
先將128個 address全部寫入random的數字，一開始將{REN, WEN}設成01代表寫入狀態。接著2\*\*7 = 128寫入，一邊維護外部的memory-array out\_mem。



從write過渡到read中間，先設為00表示do nothing，等個#10，設成10表示開始read。



Random讀出一些data，並用task testreadout check 讀出來的data是否和外面維護的memory讀到的data一樣，negedge時做更改address以及task testreadout。

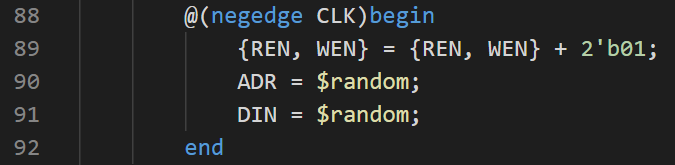
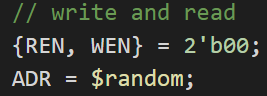


Task testreadout內容如上

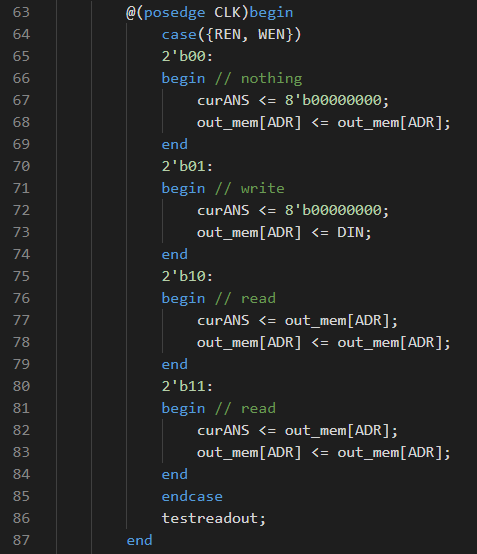
就簡單的check現在讀出來的跟curANS有沒有一樣，if not輸出現在在哪個phase debug，讀的位置是哪裡，輸出是多少，正確輸出是多少。提供debug資訊。



換到下一個phase。



第二個phase要測的是讓REN, WEN一直+1，每次address跟data-in都random，看看memory module在各種情況下的behavior有沒有符合預期。



這邊基本上就根據現在的{REN, WEN}進行操作，有個問題點是這裡的實作幾乎快跟Memory module裏頭一模一樣，問題是我也沒想到比較好的方式來測試，單純就用Wave圖以及task判斷正確性。