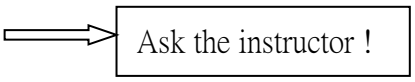
Verilog Laboratory Exercise

Design Preparation

unix% cd ~

unix% tar xvf /******/ vloglab_21f.tar



unix% cp /******/cshrc .cshrc

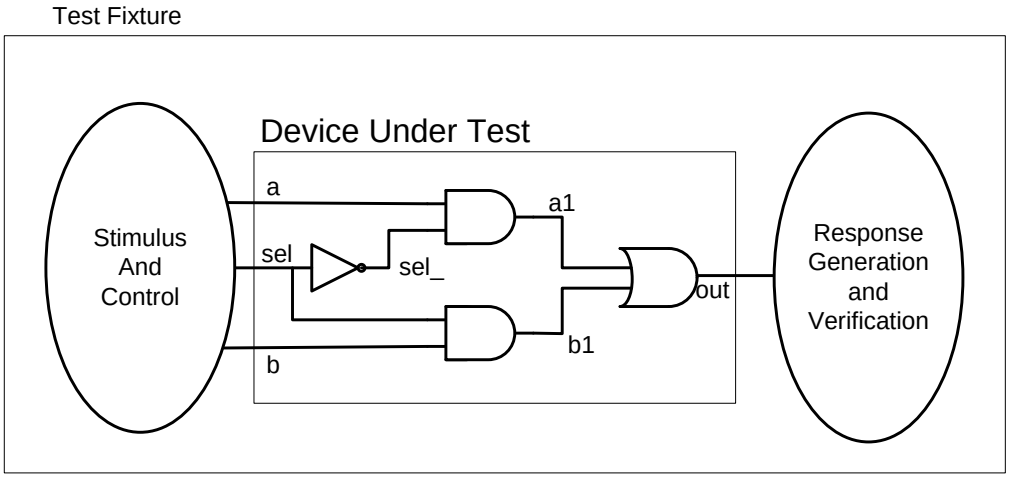
unix% source .cshrc

Start

unix% cd vloglab_20s

Lab1 : 2-1 MUX

- 請設計一個『**2 對 1 的多工器**』
- Specifications
  - ✓ Module name : mux
  - ✓ Input pins : a, b, sel
  - ✓ Output pins : out
  - ✓ Symbol View :



1. 切換工作目錄到 **Lab1** 資料夾. 此資料夾內包含 **mux.v** 及 **mux_text.v** 兩個檔案。在 **terminal** 內執行以下指令：

   unix% cd Lab1

2. 使用以上的 **DUT** 電路圖撰寫 **mux.v** 的 RTL Code，其 RTL Code 可參考以下範例（本範例含 Gate-Delay）:

```
module mux(out,a,b,sel);
// Port declarations
output      out;
input       a,b,sel;
// The netlist
not                 not1(sel_, sel);
and     #1      and1(a1, a, sel_);
and     #1      and2(b1, b, sel);
or      #2      or1(out, a1, b1);
endmodule
```

3. 執行 Simulation。

使用 NC-Verilog 執行 simulation 的話，則使用以下指令（建議使用）:

```
unix%    ncverilog   mux_test.v   mux.v   +access+r
```

「Note：在 NC-Verilog 執行的指令內時加入 +access+r 這個 option 是為了可以在 Waveform Tool 內看到訊號的波形變化。如果不打算看 Waveform 的話，則可以不用加此 option。」

4. 執行 silumation 完成後的結果將如下所示。

```
ncsim> run
                0   a = x,   b = x,   sel = x,    out = x
               10   a = 0,   b = 0,   sel = 0,    out = x
               13   a = 0,   b = 0,   sel = 0,    out = 0
               20   a = 1,   b = 0,   sel = 1,    out = 0
               30   a = 1,   b = 1,   sel = 0,    out = 0
               33   a = 1,   b = 1,   sel = 0,    out = 1
               40   a = 0,   b = 1,   sel = 1,    out = 1
               50   a = 0,   b = 1,   sel = 0,    out = 1
               53   a = 0,   b = 1,   sel = 0,    out = 0
               60   a = 1,   b = 0,   sel = 0,    out = 0
               63   a = 1,   b = 0,   sel = 0,    out = 1
               70   a = 0,   b = 0,   sel = 1,    out = 1
               73   a = 0,   b = 0,   sel = 1,    out = 0
               80   a = 1,   b = 1,   sel = 1,    out = 0
               83   a = 1,   b = 1,   sel = 1,    out = 1
Simulation complete via $finish(1) at time 180 NS + 0
./mux_test.v:22             #100 $finish;
```

Wave Tool and Waveform Database

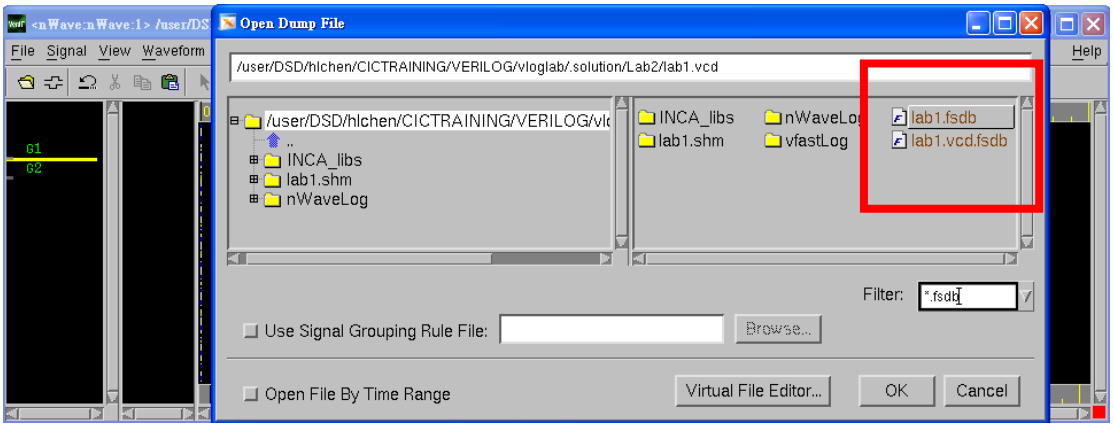
➢ Verdi – nWave

1. In the command shell, open the waveform tool from the same directory where you started the simulation.

   To invoke **nWave**, enter :

   unix% nWave &

2. In the waveform window, select **File – Open** . The file browser appears.



   I. double click the **lab1.fsdb** database

   II. click **OK**

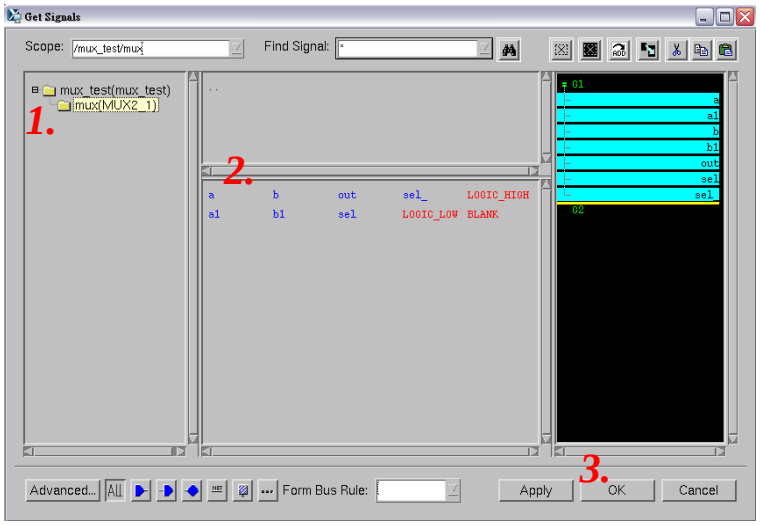
3. From the **nWave** window, select **signal – get signal**.

   The Get signal window appears.

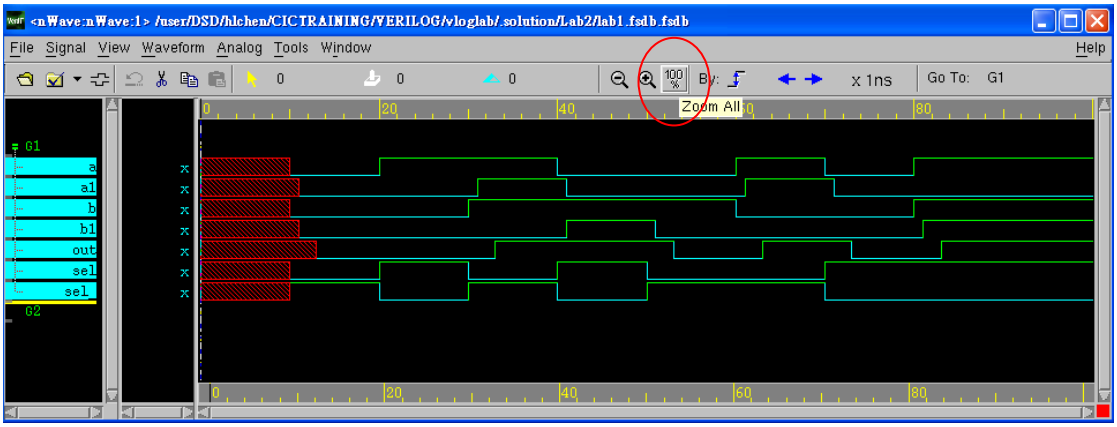
   I. You can select **mux_test** or **mux** to find the I/O signal of the module.

   II. Select the signal to scope.

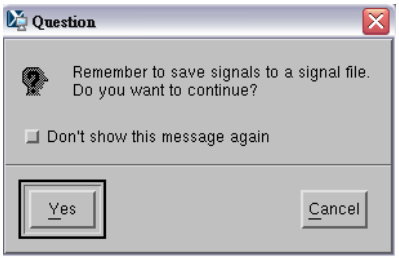
   III. Click **OK**



4. Then the waveform will be shown in the **nWave** browser.



5. From the **nWave** menu, select **File – Exit.**



6. Click **Yes**.

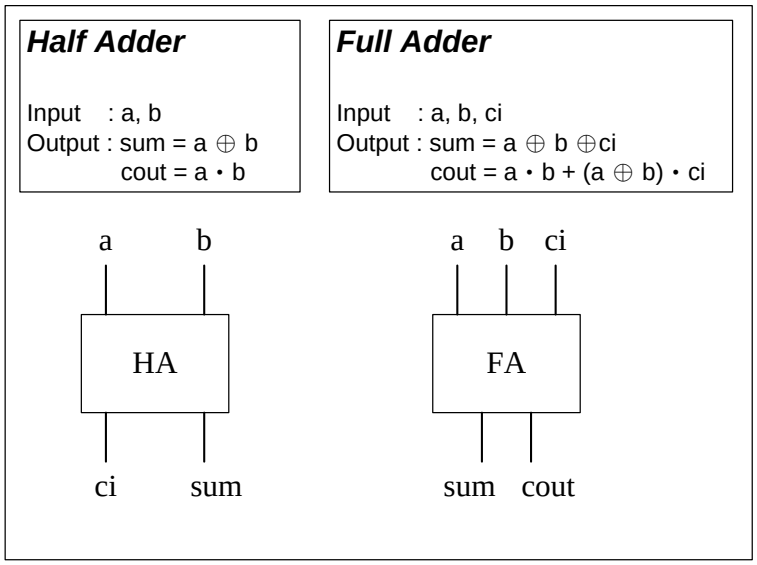
   The **nWave** window closes.

Lab2a : Full Adder Module Design

- 請設計一個 Full-Adder 模組
- Specifications
  - ✓ Module name : fa
  - ✓ Input pins : a, b, ci
  - ✓ Output pins : sum, cout
  - ✓ Function : { cout, sum } = a + b + ci;
  - ✓ Truth Table :

a	b	ci	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- **Solution : Half-Adder and Full-Adder**



1. 改變工作目錄到 Lab2a 下. 此目錄下包含 fa.v, ha.v 及 fa_test.v 等三個檔案。

   unix% cd Lab2a

2. 在 fa.v 檔案中撰寫您的 full-adder module 程式. 在 full-adder module 中使用下列的 module header 及 port interface :

```
module   fa(a, b, ci, sum, cout);
      output   sum, cout;
      input    a, b, ci;
      ......
endmodule
```

這個 Full-adder module 可以由兩個 Half-adder 與一些邏輯閘組成。請撰寫 Half-adder module(**ha.v**)程式, 並使用下列的 module header 及 port interface：

```
module   ha(a, b, sum, cout);
      output   sum, cout;
      input        a, b;
      ......
Endmodule
```

3. 請創造一個 verilog control file 並命名為 **vlog.f** 。

   unix% touch vlog.f

4. 在此 vlog.f 中指名 design file 及 testbench file 檔案名稱，以及您所需要的 command-line options。

```
fa_test.v
fa.v
ha.v
+access+r
```

接下來請使用 **vlog.f** 檔案來進行模擬，請執行以下的命令 ：

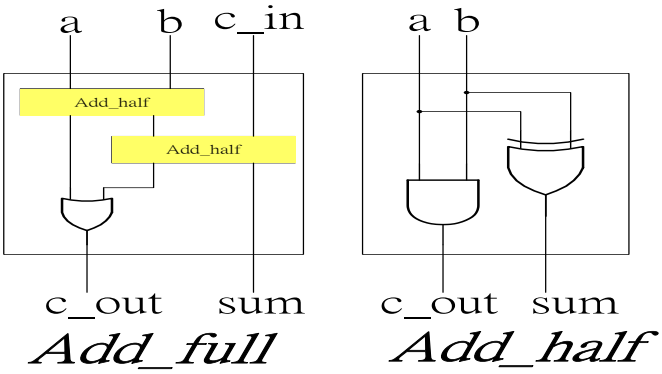
unix% ncverilog -f vlog.f

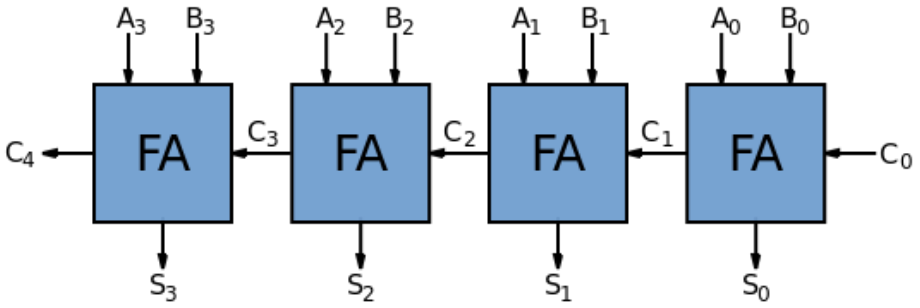
5. 如果您的 design 功能正確的話，其模擬結果將如下所示 ：

```
**********************
function test pass !!
**********************
```

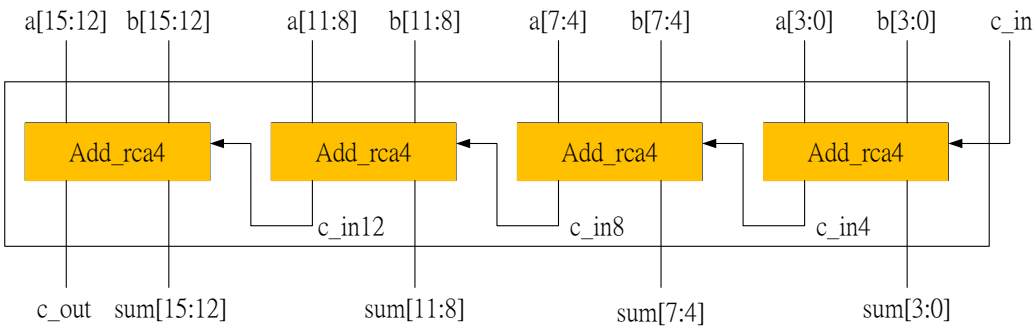
Lab2b : A 16-bit ripple-carry adder

- Please modeling an Arithmetic Logic Unit (ALU)
- Specifications
  - ✓ Module name : rca16
  - ✓ Input pins : a[15:0], b[15:0], c_in
  - ✓ Output pins : c_out, sum[15:0]
  - ✓ Symbol view :





Add_rca4



Add_rca16

1. 切換工作目錄到 **Lab2b** 資料夾。這資料夾下包含測試檔 **rca16_test.v** 及設計檔 **rca16.v** 。

   unix% cd Lab2b

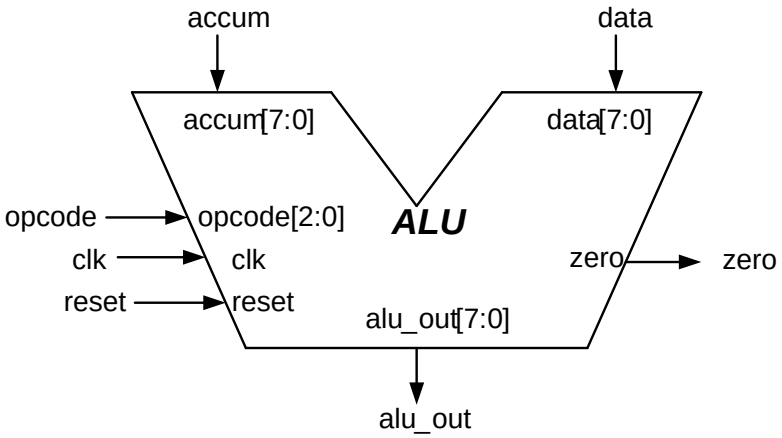
2. 依模組需求撰寫 **rca16.v** 檔案。本 Ripple Carry Adder（Adder_rca16）架構為 16 個 bits，其由 4 個 4 bits 的 ripple carry adder（Add_rca4）組成，而每個 Add_rca4 則是由 4 個 Add_full 組成，每個 Add_full 又可由 2 個 Add_half 及 1 個 OR gate 組成。如上圖所示。

3. 當您撰寫完成這 alu 模組之後，請使用 **rca16_test.v** 檔案作為測試檔來測試此電路。
   您可使用 **NC-Verilog** 或 **Verilog-XL** 來跑 simulation。
   例如：**unix%** ncverilog rca16_test.v rca16.v +access+r

4. 本測試檔已完成自動比對的功能，請檢查 simulation 的輸出結果。

Lab3a : ALU

- Please modeling an Arithmetic Logic Unit (ALU)
- Specifications
  - ✓ Module name : alu
  - ✓ Input pins : accum[7:0], data[7:0], opcode[2:0], clk, reset
  - ✓ Output pins : zero, alu_out[7:0]
  - ✓ Symbol view :



1. 切換工作目錄到 **Lab3a** 資料夾。這資料夾下包含測試檔 **alu_test.v** 及設計檔 **alu.v** 。

   unix% cd Lab3a

2. 依模組需求撰寫 **alu.v** 檔案 ．您必須撰寫依照上述的 symbol view 撰寫 port interface.並依照下列的規範撰寫此 alu 的功能描述。

   I. 所有輸入及輸出（除了「zero」訊號以外）需使用 clock 的正緣（rising edge）來觸發動作。

   II. 同步 reset 架構。當 reset 為 1 時表示 reset 啟動，此時 alu_out 訊號輸出為 0。

   III. 當 accum 輸入為 0 時，zero 訊號輸出為 1；而當 accum 輸入不為 0 時，zero 訊號輸出為 0。並且 zero 訊號不需理會 reset 訊號的動作，也不須跟 clock 同步。

3. 使用以下 3-bits 的 opcode 訊號值來定義其 alu 的操作模式。當 opcode 輸入為其他任意值(包含 unknown)時，其 alu_out 訊號輸出為 0。

opcode	ALU operation
000	Pass accum
001	accum + data (add)
010	accum - data (subtraction)
011	accum AND data (bit-wise AND)
100	accum XOR data (bit-wise XOR)
101	accum 取 2 補數 (2's complement)
110	accum*5 + accum/8
111	假如(accum >= 32)，則 alu_out=data 否則 alu_out=data 取 1 補數

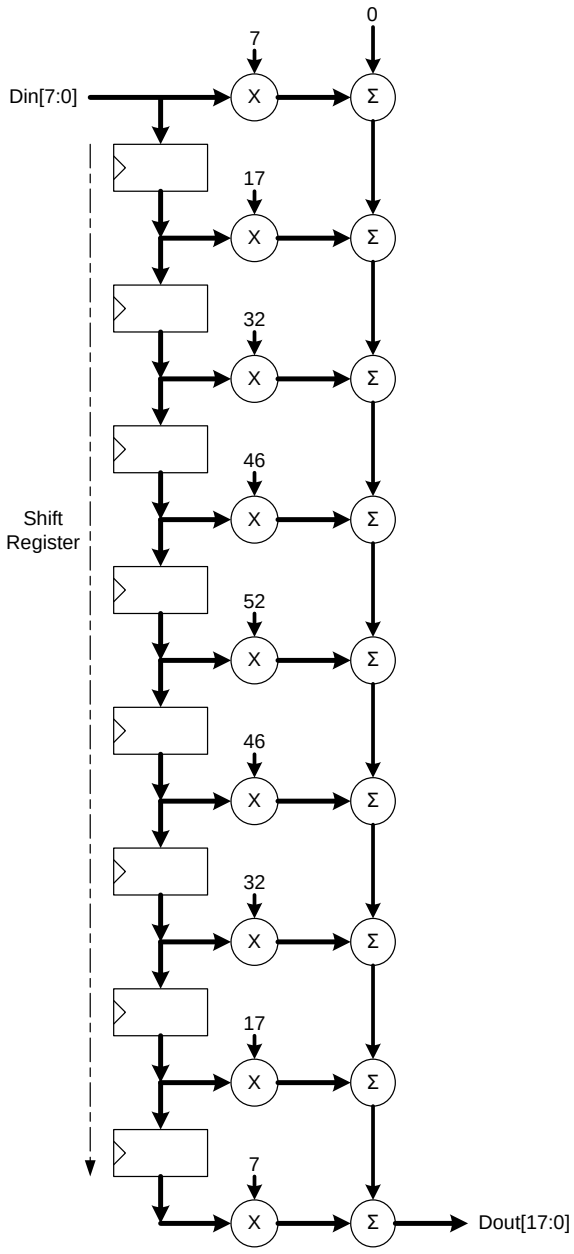
備註 ： 本題所有訊號及運算均視為無號數運算即可，且不需考慮溢位問題

4. 當您撰寫完成這 alu 模組之後，請使用 **alu_test.v** 檔案作為測試檔來測試此 alu。
   您可使用 **NC-Verilog** 或 **Verilog-XL** 來跑 simulation。
   例如：**unix%** ncverilog alu_test.v alu.v +access+r

5. 本測試檔已完成自動比對的功能，請檢查 simulation 的輸出結果。

Lab3b : FIR

- Please modeling an Gaussian lowpass FIR filter
- Specifications
  - ✓ Module name : FIR
  - ✓ Input pins : Din[7:0], clk, reset
  - ✓ Output pins : Dout[17:0]
  - ✓ Symbol view :



1. 切換工作目錄到 **Lab3b** 資料夾。這資料夾下包含測試檔 **FIR_test.v** 及設計檔 **FIR.v** 。

   unix% cd Lab3b

2. 依模組需求撰寫 **FIR.v** 檔案 ．您必須撰寫依照上述的 symbol view 撰寫 port interface.並依照下列的規範撰寫此 FIR 的功能描述。

   I. 同步 reset 架構。當 reset 為 1 時表示 reset 啟動，此時 Shift Register 的輸入皆為 0。因此此時的輸出值為 Din[7:0]乘上 7 的結果。

   II. 依本題 symbol view 所示可知其輸出訊號為 Shift Register、各階系數及輸入訊號分別進行相乘及累加之後的結果，因此 Dout 輸出不須與 colck 同步。

   III. 提示 1：您可使用 for 迴圈來撰寫 Shift Register 的動作。

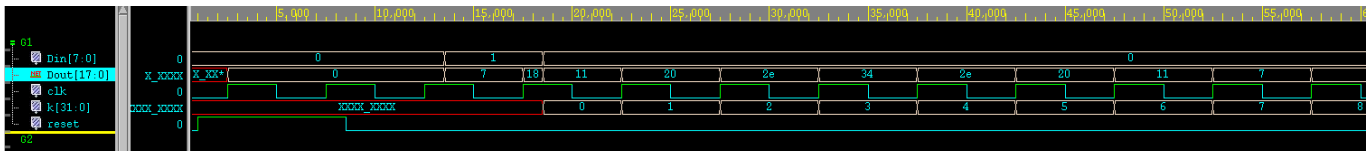
   IV. 提示 2：您可宣告一 Register Array(Memory Array)來代表 Shift Register 架構。

3. 當您撰寫完成這 FIR 模組之後，請使用 **FIR_test.v** 檔案作為測試檔來測試此 FIR。

   您可使用 **NC-Verilog** 或 **Verilog-XL** 來跑 simulation。

   例如：**unix%** ncverilog FIR_test.v FIR.v +access+r

4. 您可參考以下輸出確認輸出結果是否正確。



   依序在每個 clock 負緣可以抓取到 7 → 11 → 20 → 2e → 34 → 2e → 20 → 11 → 7 → 0 ..... (16 進制)。

Lab4a : Bit-stream Pattern Detector

- Please design a serial input bit-stream pattern detector module.
- Specifications
  - ✓ Module name : fsm_bspd
  - ✓ Input pins : clk, reset, bit_in
  - ✓ Output pins : det_out
  - ✓ Function : serial input bit-stream pattern detector

    Using finite state Mealy-machine. "*det_out*" is to be low(logic 0), unless the input bit-stream is "0010" sequentially.

    Example :

    bit_in : .... 0 0 1 0 0 1 0 0 0 1 0 0...

    det_out : .... 0 0 0 1 0 0 1 0 0 0 1 0...

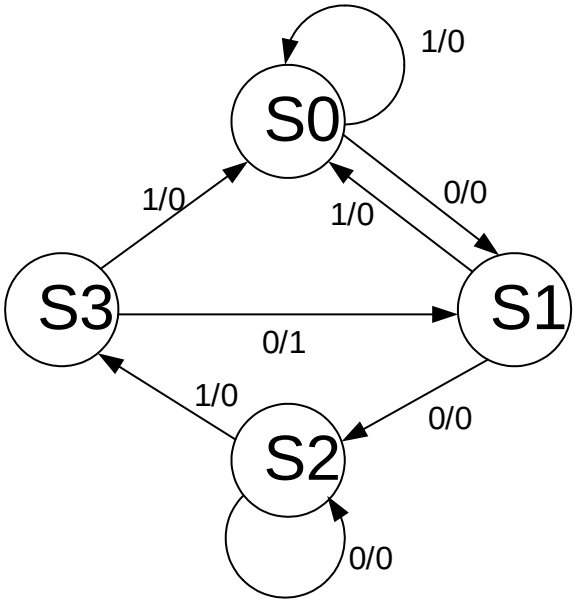
1. 切換工作目錄到到 **Lab7** 資料夾，此資料夾內包含測試檔 **fsm_test.v** 及設計檔 **fsm_bspd.v**。其中 **fsm_bspd.v** 只完成了 module header 的部分。

2. 撰寫 **fsm_bspd.v** 檔案內容並完成上述所提之 serial input bit-stream pattern detector 功能。請使用 Mealy-Machine 狀態機實現。
   I. 所有輸入訊號請使用 clock (clk) 的 rising edge 來觸發動作。
   II. 您可使用下列任何模式來設計此模組。
      A. Separates CS、NS and OL
      B. Combine CS and NS，Separate OL
      C. Combine NS and OL，Separate CS
   III. 您可自訂數個狀態來完成此設計。
      A. Less than 4 states：S0 (00)，S1 (01)，S2 (10)，S3 (11)
      B. Between 5 and 8 states：S0 (000)，S1 (001)，S2 (010)，S3 (011)，S4 (100)，S5 (101)，S6 (110)，S7 (111)。

3. 當完成您的 fsm_bspd 模組後，請用 **NC-Verilog** 或 **Verilog-XL** 跑 simulation 看看。如果有錯誤產生的話，請修正，直到正確無誤為止。

4. 您可以參考以下狀態圖與狀態表來完成您的 fsm_bspd 設計。

State Diagram

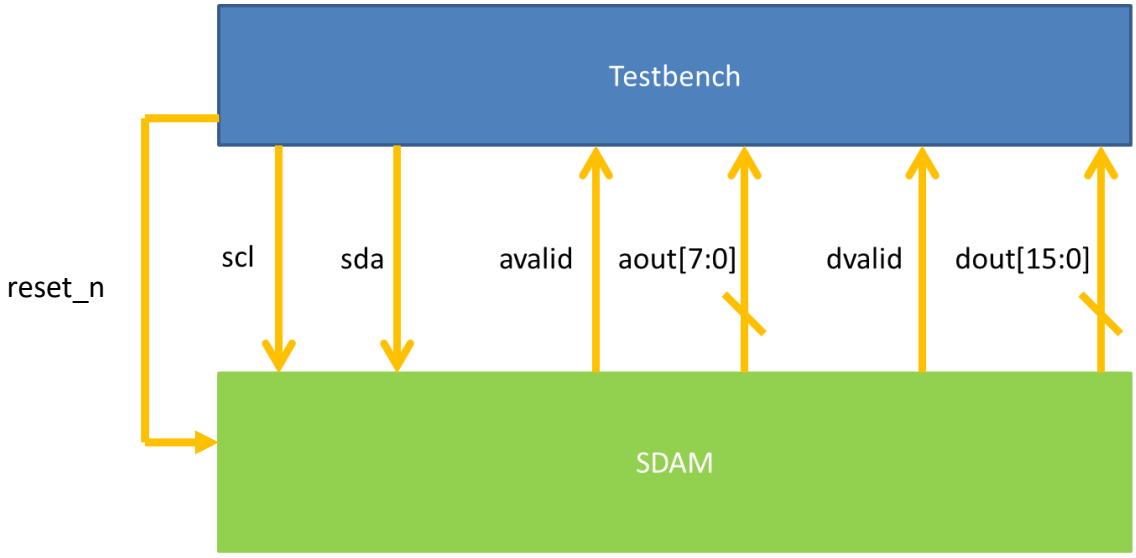


State Table

Present state	Next state		Present output	
	X = 0	X = 1	X = 0	X = 1
S0	S1	S0	0	0
S1	S2	S0	0	0
S2	S2	S3	0	0
S3	S1	S0	1	0

Lab4b : Serial Data Access Module

- Please modeling an Serial Data Access Module
- Specifications
  - ✓ Module name : SDAM
  - ✓ Input pins : scl, sda, **reset_n**
  - ✓ Output pins : dvalid, dout[15:0] , avalid, aout[7:0]
  - ✓ Symbol view :



1. 切換工作目錄到 **Lab4b** 資料夾，這資料夾內包含已完成的測試檔 **testbench.v** 及設計檔 **SDAM.v**。在這個 lab 中，您必須撰寫設計檔 **SDAM.v** 的內容，並使用 **textbench** 模組去測試 **SDAM** 模組的功能是否正常。

   unix% cd Lab4b

2. 本電路為一序列資料輸入存取電路，如 Symbol view 所式。其中 scl 為序列時脈輸入訊號，sda 為序列資料輸入訊號。dvalid 及 dout 為並列資料輸出訊號，avalid 及 aout 為並列位址輸出訊號。

3. reset_n 為 Low active，表示當 reser_n 為 0 時啟動。

4. 動作上，測試電路 Testbench 會利用 scl 和 sda 將序列資料入到 SDAM。SDAM 需依照下圖說明依序接收序列資料及位址；並在收到序列資料及位址後將之轉成並列資料輸出(利用 dvalid 及 dout)及並列位址輸出(avalid 及 aout)。時序說明如下:

   T1. 一開始 Testbench 將 sda 設為 1，此時系統為閒置的狀態。

   T2. 當時脈正緣時，Testbench 將 sda 設為 0 表示指令觸發，SDAM 開始執行指令動作。

   T3. 由 sda 位準為 1 或 0 來指示要進行何種動作。1 表示進行寫入資料；0 表示進行讀取資料。本題只會出現 1。

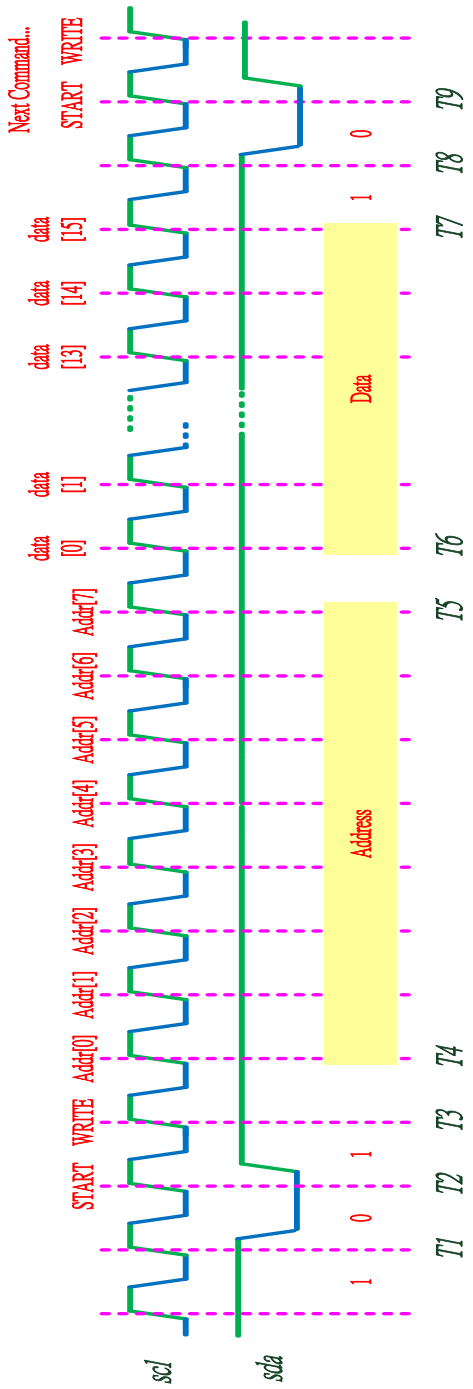
   T4-T5. 這段時間共有 8 個時脈週期，Testbench 依序將位址利用 sda 由 LSB 開始送出。

T6-T7. 這段時間共有 16 個時脈週期，Testbench 依序將資料利用 sda 由 LSB 開始送出。

T8. 接下來，Testbench 再次將 sda 設為 1，表示 Testbench 系統再次回到閒置的狀態。等同 T1 時間點。

T9. 當時脈正緣時，Testbench 將 sda 設為 0 表示指令觸發，SDAM 開始執行下一筆指令動作。等同 T2 時間點。

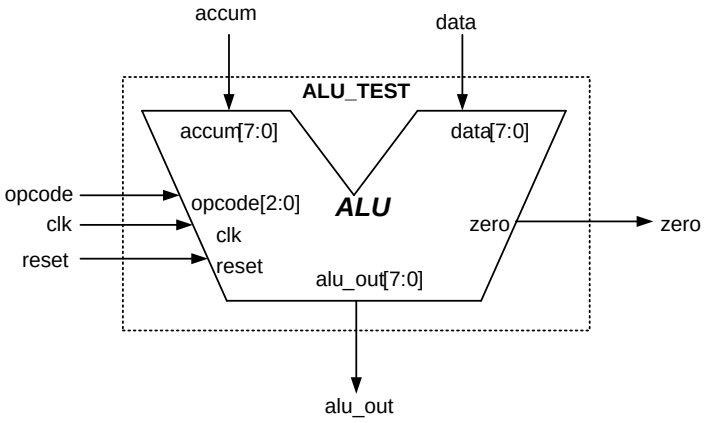
於 T8 時間點之後，SDAM 須將 avalid 及 dvalid 設為 1 表示此時輸出的 aout 及 dout 為有效輸出，且 **avalid 和 dvalid 僅可以持續一個 scl 週期**。Testbench 偵測到這筆輸出後，會立刻進行結果比對，之後才會有下一筆資料輸入。若 avalid 或 dvalid 沒有設定為 1，則 testbench 將會進行等待動作，電路將不會下一筆輸入。



5. 本題將自動比對，請檢查您的結果並視情況修正。

Lab5 : Testbench of ALU(Lab3a)

- Please modeling an Testbench of Arithmetic Logic Unit (ALU)
- Specifications
  - ✓ Module name : alu_test
  - ✓ Interface with ALU : accum[7:0], data[7:0], opcode, clk, reset, zero, alu_out
  - ✓ Symbol view :



1. 切換工作目錄到 **Lab5b** 資料夾，這資料夾內包含測試檔 **alu_test.v** 及已完成的設計檔 **alu.v**。在這個 lab 中，您必須撰寫測試檔 **alu_test.v** 的內容，並使用 alu_test 模組去測試 alu 模組的功能是否正常。

   unix% cd Lab5b

2. 為了測試 alu 模組功能，在 **alu_test.v** 測試檔中，我們必須撰寫對於測試 alu 模組功能所需之各種測試向量，例如:測試 reset 功能、測試 opcode 功能及測試 zero 訊號功能等測試向量。請依照下列指示撰寫您的 alu_test 測試模組功能：

   I. 在 alu_test 模組內，首先您必須先 instance alu 模組，您可使用 call by orders 或 call by name 的方法來宣告。

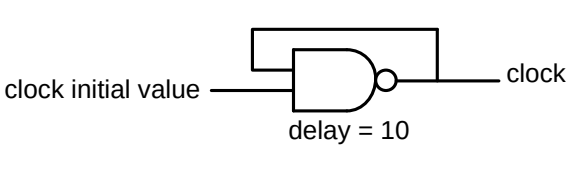
      alu alu(.clk(clk), .opcode(opcode), …….);

   II. 接下來您必須對 alu_test 模組中有使用到的訊號宣告其 data type，例如：reg、wire 或其他 data type。

   III. 在功能的部分，由於 alu 模組必須有個 clock 訊號，因此我們必須在 alu_test 模組內建立 clock 產生器，並將其產生的 clock (clk) 訊號連接到 alu 模組的 clock (clk) 訊號。您可參考以下 structural clock 做法來設計：

      A. 定義 clock 週期為 20ns。其中 10ns 為 High，另外 10ns 為 Low。

      B. Clock 的初始狀態為 unknown。



time	clk
0	x
10	1
20	0

IV. 在功能部分，我們需要測試各種 opcode 的輸入情況及 reset 的功能是否正常。

A. 將 reset 啟動或停止動作來測試 alu 模組的 reset 是否有正常動作，當 reset 訊號為 1 時代表 reset 動作。

B. 您可以使用「for loop」來自動產生 8 個 opcodes 訊號的值，並將之送入 alu 模組以便觀察功能。

C. 為測試 alu 模組收到 opcode 訊號為 unknown 時的反應，您必須將 opcode 訊號設定為 unknown。

D. 測試 zero 訊號的功能（accum 訊號為 0 時，zero 訊號才會為 1）。

E. 為方便完成以上規範，您可在 testbench 內使用 initial block。

V. 在驗證的部分，或許可以加上 waveform display 的宣告，以方便觀察波形變化。

```
initial     begin
        $shm_open("alu.shm");    // SHM Database
        $shm_probe("AC");
        $fsdbDumpfile("alu.fsdb");    // FSDB Database
        $fsdbDumpvars();
end
```

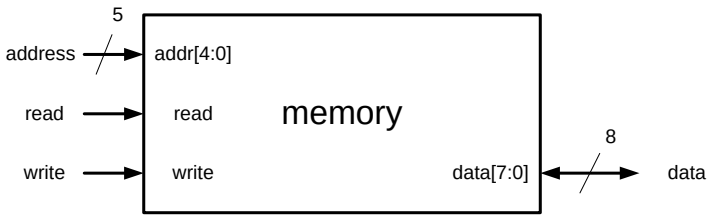
VI. 我們可以加入 monitor 或 display 等輸出宣告，以方便 simulation 時就可觀察訊號輸入與相對應的訊號輸出關係。

3. 撰寫完成 **alu_test.v** 檔案後，您可使用 **NC-Verilog** 進行 simulation。

4. 檢查您給予 alu_test 模組對 alu 模組的輸入的訊號及其相對應的輸出是否正常？是否有其他狀況尚未考慮到的，請補充！

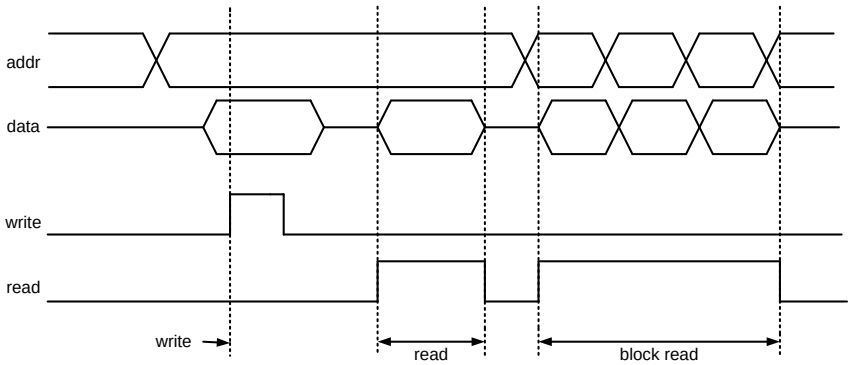
Lab6 : Memory

- Modeling a Memory with a Bidirectional Data Bus.
- Specifications
  - ✓ Module name : mem
  - ✓ Input pins : addr[4:0], read, write
  - ✓ inout pins : data[7:0]
  - ✓ Symbol view :



1. 切換工作目錄到 **Lab6** 資料夾，此資料夾下包含測試檔 **mem_test.v** 及設計檔 **mem.v**。

2. 此 lab 的設計檔 **mem.v** 內容只完成 module header 的部分，特別注意其中 data 訊號是 bidirectional 的。請依照以下規範撰寫 memory 模組的內容：
   I. 使用 memory register array，並命名為 memory。此 memory register array 須符合以下規範：
      A. The MSB of each word is bit **7**
      B. The LSB of each word is bit **0**
      C. The first address is address **0**
      D. The last address is address **31** (hex 1F)
   II. 使用非同步存取方式此 memory model，並使用兩個控制訊號做為 memory model 讀取（read）或寫入（write）的控制。



   III. 在 write 控制訊號的 positive edge 發生時，此時在 data 訊號線出現的資料將會被寫入 memory 中，其寫入位址由當下 addr 訊號線之資料所定義。您可以使用 procedural assignment 來撰寫這部分的 code。
   IV. 當 read 控制訊號值為 High 時，memory 將以目前 addr 訊號線之資料做為位址，將 memory 內該位址之內容藉由 data 訊號線讀出。

V. 此 memory 須支援 block read 功能。當 read 控制訊號線維持在 High 的狀態下，若 addr 訊號線之資料改變，則被讀出資料之位址亦會隨之改變，因此可連續讀到不同位址之資料。

VI. 當 read 控制訊號值為 Low 時，其 read 控制將呈現不致能的狀態，在此情況下 read 控制訊號對 data 訊號線將呈現 high-impedance 的狀態。亦即在此情形下若 write 控制訊號值為 high 的話，則寫入的動作將可正常執行。

3. 使用 **mem_test.v** 檔來測試 **mem.v** 設計檔。執行完 simulation 後，您會發現有 error information 出現。請查看 **mem_test.v** 檔案內容，並思考以下問題：

   I. data 訊號線為一 bidirectional port，其 data type 宣告為 wire

   II. 當在 procedural block 中，我們將資料值設給 data 訊號線時，此 data 訊號線是否應該宣告為 reg 的 data type。

4. 為修正上述 error information 的問題，請修改 **mem_test.v** 檔案內容後再跑一次 simulation 看看，直到 error information 消失為止。

   提示：您可在 procedural assignment 中使用 shadow register 來修正。

   i. procedural assignment 內容主要用在 write 的情況下，您可對 procedural block 內的 data 訊號線，另外宣告使用一個 register。

   ii. 使用 continuous assignment 及 conditional operator 來決定當 read 不啟動時，可將 procedural block 內的 register 內容寫入到 memory。若 read 啟動時，表示 memory 要做讀取的動作，因此 continuous assignment 的結果須為 High-impednace 以避免互相衝突。

5. 請使用 **mem_test.v** 檔案及 **mem.v** 檔案執行 simulation，並確定其功能正常。

```
Setting all memory cells to zero…
Reading from one memory address…
Setting all memory cells to alternating patterns…
Doing block read from five memory addresses…

Completed Memory Tests With 0 Errors!
```

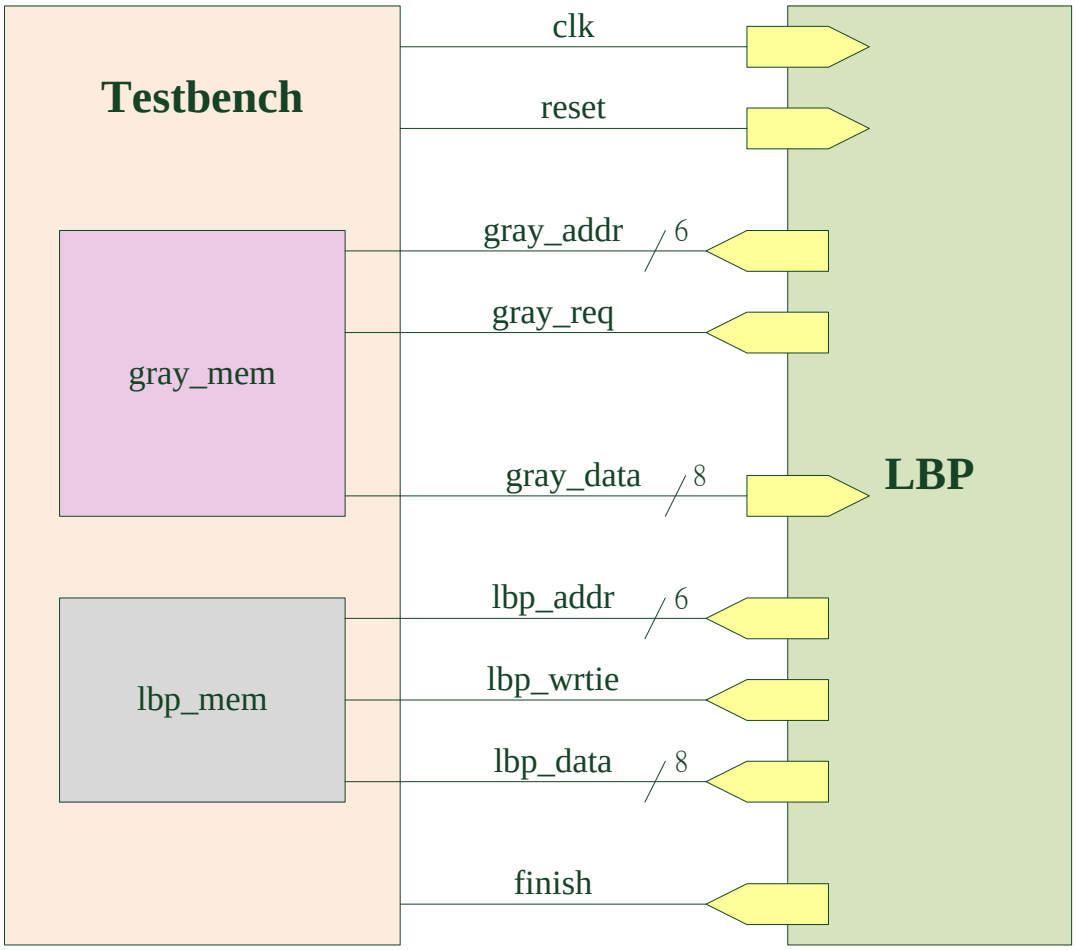
Lab7 : Local Binary Pattern Encoder

1. 問題描述

請完成一 Local Binary Patterns (後文以 LBP 表示)，輸入為一灰階影像，此灰階影像存放於 Testbench 的灰階圖像記憶體模組(*gray_mem*)中，LBP 須發送訊號至 Testbench 以索取灰階影像資料，再對灰階影像中每個 pixel 各自進行獨立運算，運算後的結果請寫入 Testbench 的局部二值模式記憶體模組(*lbp_mem*)內，並在整張影像訊號處理完成後，將 *finish* 訊號拉為 High，接著系統會自動進行比對整張影像資料的正確性。

2. 設計規格

2.1 系統方塊圖



圖一、系統架構圖

2.2 輸出入訊號和記憶體描述

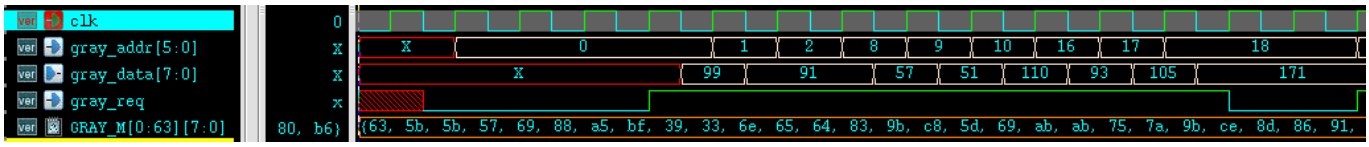
表一、輸入/輸出信號

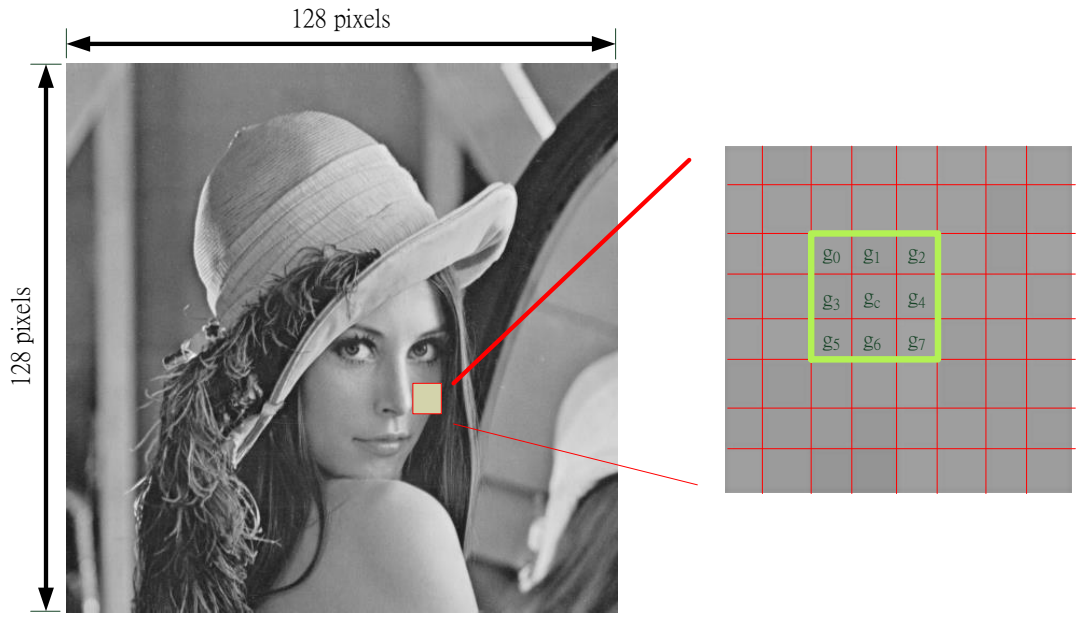
Signal Name	I/O	Width	Simple Description
clk	I	1	本系統為同步於時脈正緣之同步設計。
reset	I	1	高位準”非”同步(active high asynchronous)之系統重置信號。
gray_addr	O	6	灰階圖像位址匯流排。LBP 端需透過此匯流排向 Testbench 的 gray_mem 索取該位址的灰階影像資料。 每一個週期僅能索取一個位址的資料。 題目不限制位址及資料的索取次數。
gray_req	O	1	灰階圖像**讀取**致能訊號。當為 High 時，表示 LBP 端要向 gray_mem 讀取灰階圖像資料。
gray_data	I	8	灰階圖像資料匯流排。Testbench 端利用此匯流排將 gray_mem 的灰階圖像資料送到 LBP 端。
lbp_addr	O	6	局部二值模式位址匯流排。LBP 端利用此位址將經 LBP 運算完成後之資料儲存至 lbp_mem 中。
lbp_write	O	1	局部二值模式資料**寫入**致能訊號。當為 High 時，表示 LBP 端要將 LBP 運算結果寫入 lbp_mem 中。
lbp_data	O	8	局部二值模式資料匯流排。LBP 端需透過此匯流排將 LBP 運算結果傳送到 lbp_mem 中。
finish	O	1	LBP 運算完畢之通知訊號。當所有的灰階圖像資料經過個別運算完畢且儲存後，需將 *finish* 訊號拉為 High，以通知 Testbench，開始進行所有資料之比對。

2.3 系統功能描述及動作時序

本電路功能為當 *reset* 結束後， LBP 端可立即開始對 Testbench 進行動作。當 Testbench 在每個時脈訊號**負緣觸發**時若偵測到 *gray_req* 訊號為 High 時表示 LBP 端對 gray_mem 要求索取灰階圖像資料，此時 Testbench 會依 *gray_addr* 匯流排所指示的位址將灰階圖像記憶體內的位址資料由 *gray_data* 匯流排輸入 LBP 端。

時序規格請參考下圖所示。

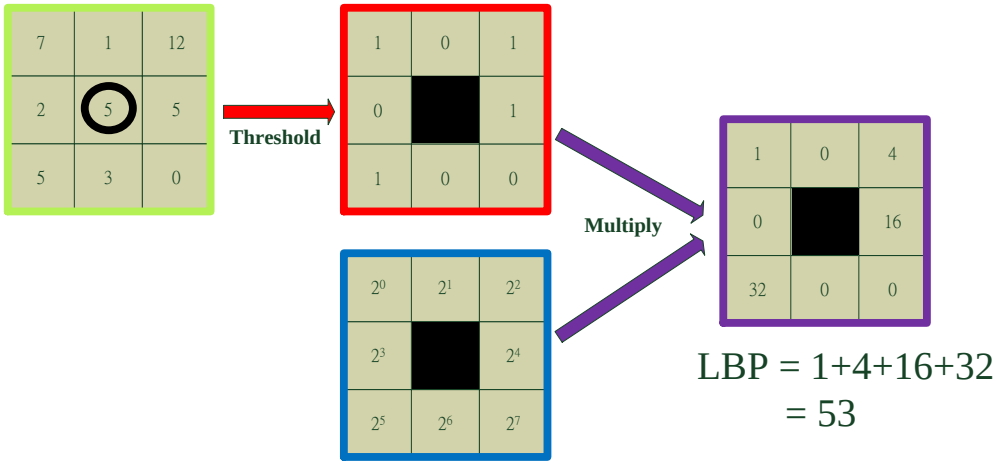




由 *gray_data* 輸入的有效灰階圖像資料須經過 3x3 的 LBP 編碼才可得到區域二值模式資料，編碼方式為利用每個 pixel 及其相鄰的數個 pixel 的相對應關係來計算，以如上圖所示的灰階圖像架構來說明，若待處理 pixel 位置為 P_C(位置為(x,y),灰階值為 g_c)，由 P_C 向外延伸 1 個 pixel 為其編碼區域 $P_{n(n=1\sim8)}$(灰階值為 g_p，p=0~7)，則 LBP 編碼為…

$$LBP(x,y)=\sum_{p=0}^{P-1}s(g_p-g_c)2^p \qquad \text{，而 } s(z)=\begin{cases}1, & z\geq 0\\ 0, & otherwise\end{cases}$$

舉例說明，若編碼區域內容如下圖所示，g_c 則為左方 3x3 區域中間的圓圈位置，則利用上式所計算出各 g_p 的 Threshold 值 $s(z)$就如中間上方圖示所示，將各 g_p 的 Threshold 值乘上各位置的權重值 2^p (中間下方圖示)就可以得到如右方圖示的運算結果，因此 pixel P_C 的 LBP 運算結果就是將右方圖示框框內內所有值相加即可得到 pixel P_C 的 LBP 編碼為 53(十進制)。



這張圖的所有 pixel 都計算完成各自的 LBP 編碼值後，接著要將編碼結果存到 Testbench 的 lbp_mem 內，題目規定由 gray_mem 的第 k 個位址 pixel 所讀取的灰階圖像資料經 LBP 運算後的結果須存到 lbp_mem 的第 k 位址；

lbp_mem 的寫入方式如下，當 *lbp_write* 的負緣觸發時，就會將目前 *lbp_data* 上的內容寫入到 lbp_mem 的 *lbp_addr* 所指示的位址內。

時序規格圖請參考下圖所示。

當所有 pixel 都處理完畢後，請將 finish 訊號拉為 High，接著 Testbench 就會開始進行結果驗證。

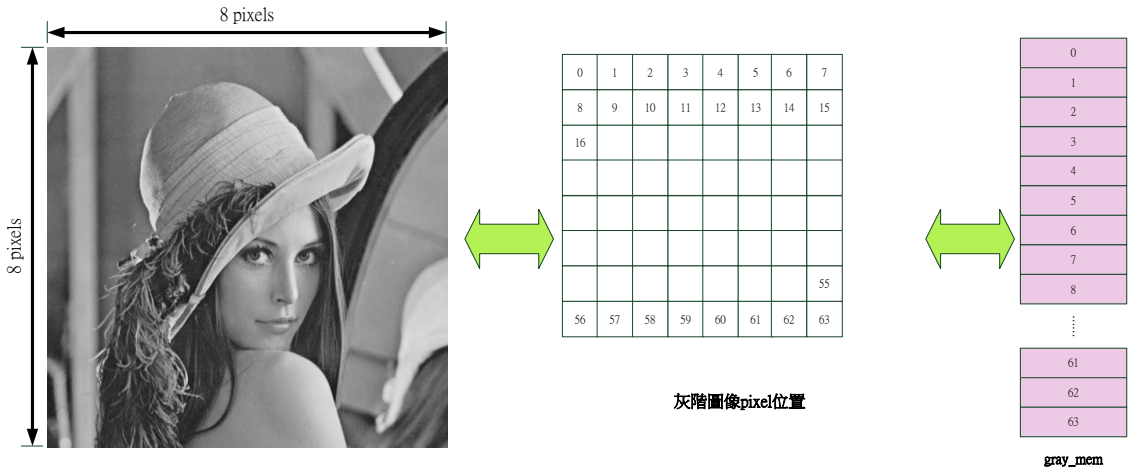
為簡化題目難度，灰階圖像最外圍一圈的 pixel 不須做 LBP 運算，並且這一圈的 pixel 在 lbp_mem 的數值為 0，Testbench 會自動初始化整個 lbp_mem 的所有位址的數值為 0。

為再次簡化題目難度，所以本題灰階圖像的長與寬只有 8x8 共 64 個 pixel。

同學可參考”5.參考資料”就可以知道這張灰階圖像(gray_mem)中各 pixel 的值及 LBP 編碼後的結果(lbp_mem)。

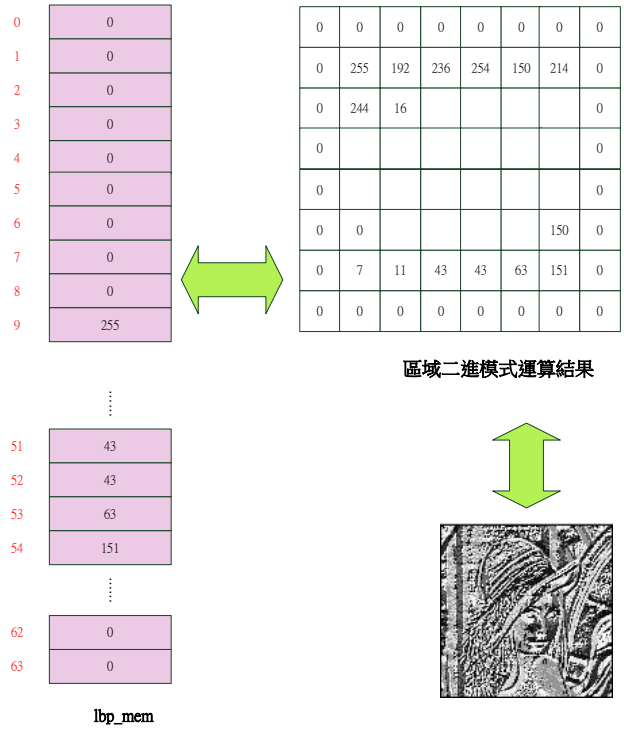
2.3.1 灰階圖像及 gray_mem 對應方式

灰階圖像大小固定為 8x8 pixels，每個 pixel 為 8bit 灰階(每個 8bit 灰階圖像 pixel 的值介於 0 到 255 之間)，因此 Testbench 的灰階圖像記憶體模組(*gray_mem*)共有 64 個位址用以存放各 pixel 的灰階圖像資料，圖像與記憶體模組的對應方式如下圖所示。



2.3.2 LBP 處理結果及 lbp_mem 對應方式

LBP 圖像為 8x8 pixels，每個 pixel 為 8bit，因此 Testbench 的 lbp_mem 共有 64 個位址用以存放各 pixel 的處理結果，本題目規定最外圍一圈 pixel 的值須為 0，因此 LBP 處理結果及局部二值模式記憶體的對應方式及處理結果應如下圖所示。



3. 模擬

Lab 提供一組測試樣本，可依下面範例來進行模擬:

ncverilog 指令範例如下：

ncverilog testfixture.v LBP.v +access+r

4. 結果及驗證

若模擬結果正確的話，則會出現" Congratulations "及文字提示。

若有錯誤的話，則會出現錯誤原因提示，請依提示修正您的 code。

5. 參考資料

以下為 MATLAB 所模擬的結果，因此若要將以下內容的 X 軸及 Y 軸位置與硬體的位址相對應，請自行將以下內容的 X 軸及 Y 軸位置減 1 即可。

Gray_mem 內容:

	1	2	3	4	5	6	7	8
1	99	91	91	87	105	136	165	191
2	57	51	110	101	100	131	155	200
3	93	105	171	171	117	122	155	206
4	141	134	145	160	140	127	150	208
5	167	167	170	166	145	132	142	207
6	174	179	176	165	148	135	137	204
7	164	174	172	159	140	130	133	195
8	154	162	164	152	131	127	128	182

LBP_mem 內容:

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	255	192	236	254	150	214	0
3	0	244	16	8	252	246	150	0
4	0	252	246	99	105	252	150	0
5	0	248	96	40	105	253	150	0
6	0	0	8	43	41	61	150	0
7	0	7	11	43	43	63	151	0
8	0	0	0	0	0	0	0	0