數位系統實驗筆記

實驗一　組合邏輯設計—比較器

一、　組合邏輯之設計步驟

1. 對問題之文字描述決定需要輸入輸出變數之數目
2. 依輸入、輸出變數關係，求真值表
3. 列出各輸出變數的布林函數
4. 化簡、畫出邏輯圖

二、　比較器

大小比較器即是用來比較 A 與 B 兩數而決定其相對大小的組合電路。

Ex：1 位元比較器

A	B	G (A>B)	E (A=B)	L (A<B)
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

由左表知：

G=AB'

E=A'B'+AB=(A⊕B)'

L=A'B

Ex：2 位元比較器(參考 1 位元比較器推出)

(1)A=B→ $A_1=B_1$ 且 $A_0=B_0$　　則 $AEB=E_1E_0$

(2)A<B→ $A_1<B_1$ 或 ($A_1=B_1$ 且 $A_0<B_0$)　則 $ALB=L_1+E_1L_0$

(3)　A>B→ $A_1>B_1$ 或 ($A_1=B_1$ 且 $A_0>B_0$)　則 $AGB=G_1+E1G_0$

方法一:

```
/* comp */
module comp (A, B, AEB, AGB, ALB);
input [1:0] A, B;
output AEB, AGB, ALB;
reg AEB, AGB, ALB;
always @(A, B)
   begin
      if(A>B)
         begin
            AGB = 1'b1;
            ALB = 1'b0;
            AEB = 1'b0;
         end
      else if(A<B)
         begin
            AGB = 1'b0;
            ALB = 1'b1;
            AEB = 1'b0;
         end
      else
         begin
            AGB = 1'b0;
            ALB = 1'b0;
            AEB = 1'b1;
         end
            end
endmodule
```

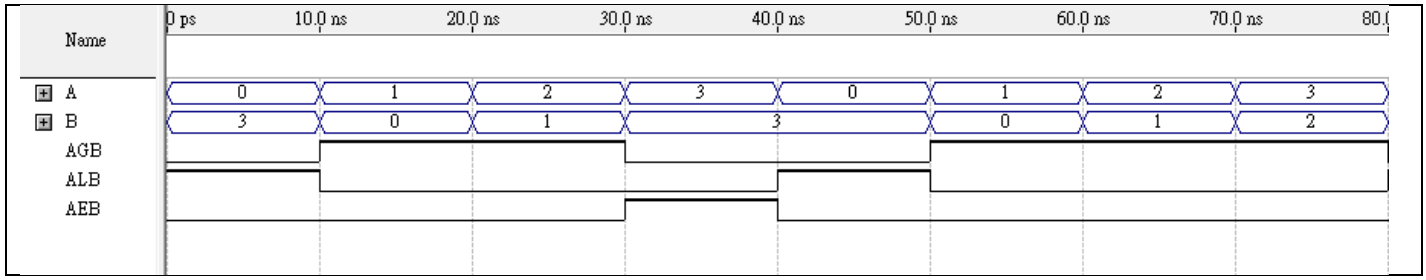
方法二:

```
/* comp_2   */
module comp_2 (A, B, AEB, AGB, ALB);
input [1:0] A, B;
output AEB, AGB, ALB;
reg AEB, AGB, ALB;
always @(A, B)
   begin
      AEB = Equal(A[1],B[1]) & Equal(A[0],B[0]);
      AGB = Grate(A[1],B[1]) | (Equal(A[1],B[1]) & Grate(A[0],B[0]));
      ALB = Less(A[1],B[1]) | (Equal(A[1],B[1]) & Less(A[0],B[0]));
   end
function Equal;
input a, b;
Equal = ~(a ^ b);
endfunction

function Grate;
input a, b;
Grate = a & (~b);
endfunction

function Less;
input a, b;
Less = (~a) & b;
endfunction
endmodule
```

波行模擬


Name
0 ps
10.0 ns
20.0 ns
30.0 ns
40.0 ns
50.0 ns
60.0 ns
70.0 ns
A
B
AGB
ALB
AEB
0 1 2 3 0 1 2 3
3 0 1 3 0 1 2

實驗二　解 / 編碼器

※解碼器

在數位系統中，資料或數目的傳遞，只有兩種狀態，也就是〝0〞、〝1〞，且數位系統中操作結果的二進制碼，必須經過一種改變為十進制系統之設備，這種編譯的工作稱為「解碼」。

一般而言，一個 n 位元的二進碼可表示出 2^n 種訊息，而所謂的解碼器能將二進位資料的 n 個輸入碼轉換成最多 2^n 種輸出中唯一輸出的組合電路。有 n 個輸入，m 個輸出的解碼器稱為 nXm 解碼器。

(一)2 對 4 解碼器 ：　二個輸入被解碼成四個輸出

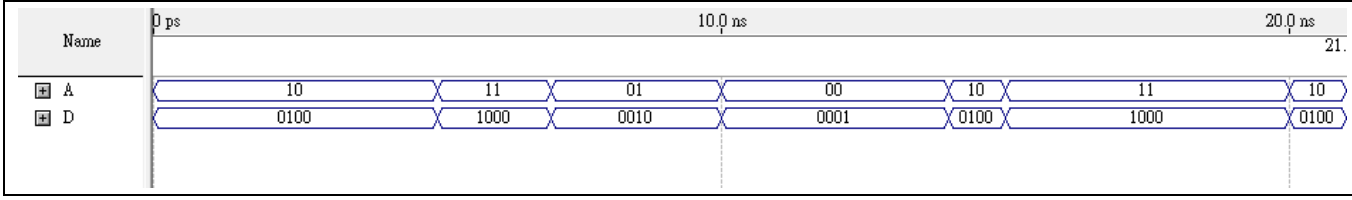
如 Truth table 所示

輸入		輸出			
A1	A0	D3	D2	D1	D0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

以 Case 方式撰寫 2 對 4 解碼器

```
1  /* decoder2_4  */
2  module decoder2_4 (A, D);
3  input [1:0] A;
4  output [3:0] D;
5  reg [3:0]D;
6  always @(A)
7     case( A )
8     2'b00 : D = 4'b0001;
9     2'b01 : D = 4'b0010;
10    2'b10 : D = 4'b0100;
11    2'b11 : D = 4'b1000;
12    endcase
13 endmodule
```

波型模擬



(二)2*4 Decoder with Enable:　大多數的解碼器都有一條或更多條的致能輸入以控制線路工作。

輸入			輸出			
E	A1	A0	Y3	Y2	Y1	Y0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0

左表，當 E="0" 時，此線路被抑制，此時，無論輸入值為何，沒有

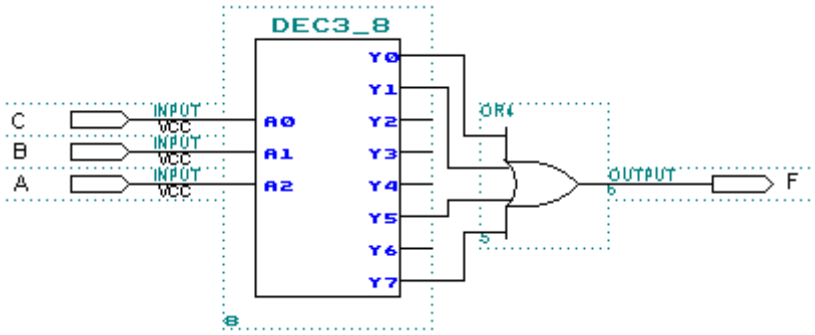
(三)組合電路的製作

解碼器可提供 n 個輸入變數的 2^n 個最小項。由於任何布林函數均可表成最小項之和，故我們可以用解碼器來產生最小項，且加上一個外接的 OR 閘即可產生其邏輯和。

製作方法：任何含有 n 個輸入和 m 個輸出的組合電路均可用一個 n 對 2^n 線解碼器及 m 個 OR 閘作

範例：試用一個解碼器與 OR 閘製作此布林函數　$F(A,B,C)=\Sigma(0,1,5,7)$

因為有三個變數故需 3x8Decoder



	A	B	C	標準積（最小項）	標準和（大項）
0	0	0	0	A’B’C’	A+B+C
1	0	0	1	A’B’C	A+B+C’
2	0	1	0	A’BC’	A+B’+C
3	0	1	1	A’BC	A+B’+C’
4	1	0	0	AB’C’	A’+B+C
5	1	0	1	AB’C	A’+B+C’
6	1	1	0	ABC’	A’+B’+C
7	1	1	1	ABC	A’+B’+C’

Σ(0,1,5,7)=A′B′C′+A′B′C+AB′C+ABC

※編碼器

解碼器接受 N 位元輸入碼，僅僅激發一條輸出線為高電位，而編碼器的作用與解碼器剛好相反，它具有 2^n（或少於）個輸入線與 N 個輸出線，其輸出訊號對應於 2^n 個輸入變數所代表的二位數碼。

(一)4 對 2 編碼器

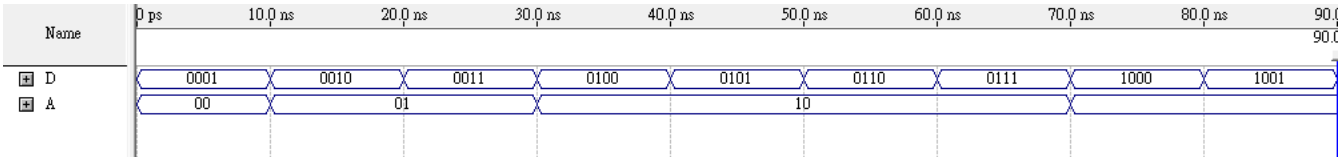
如 Truth table 所示

輸入				輸出	
D3	D2	D1	D0	A1	A0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

以 Case 方式撰寫 4 對 2 編碼器

```
1  /* encoder4_2 */
2  module encoder4_2 (D0, D1, D2, D3, A);
3  input D0, D1, D2, D3;
4  output [1:0] A;
5  reg [1:0] A;
6  always @(D0 or D1 or D2 or D3)
7     case({D3, D2, D1, D0})
8     4'b0001 : A = 2'b00;
9     4'b0010 : A = 2'b01;
10    4'b0100 : A = 2'b10;
11    4'b1000 : A = 2'b11;
12    endcase
13 endmodule
```

波型模擬



(二) 4 對 2 線優先權編碼器

優先權編碼器是一個可以執行優先權功能的組合電路，其操作為：如果同一時間有兩個或更多的輸入等於 1 時，具有最高優先權的輸入就可先做處理。

如 Truth table 所示

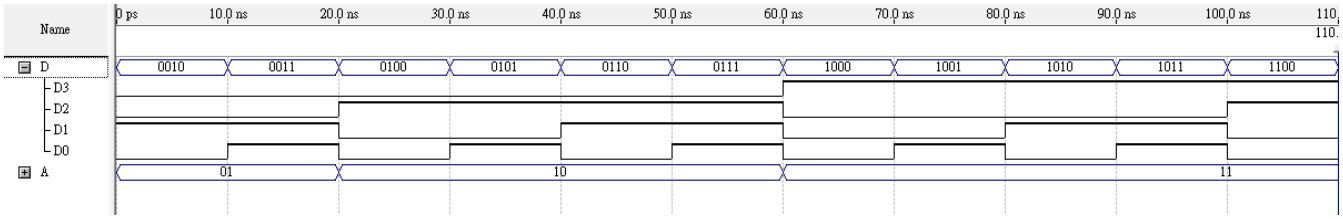
輸入				輸出	
D3	D2	D1	D0	A1	A0
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

1.如左表所示：輸入 D3 具有最高的優先權，所以無論其他輸入值為何，當此輸入為 1 時，A1A0 的輸出便為 11(二進位 3)；同理 D2 為次高優先權。
2. X.：表示隨意條件(Don't care)

以 if … else if … else 方式撰寫 4 對 2 優先權編碼器

```verilog
/* priority_encoder */
module priority_encoder (D3, D2, D1, D0, A);
input D3, D2, D1, D0;
output [1:0] A;
reg [1:0] A;
always @(D3 or D2 or D1 or D0)
   if(D3==1'b1)
      A = 2'b11;
   else if({D3, D2}==2'b01)
      A = 2'b10;
   else if({D3, D2, D1}==3'b001)
      A = 2'b01;
   else if({D3, D2, D1, D0}==4'b0001)
      A = 2'b00;
   else
      A = 2'bxx;
endmodule
```

波型模擬



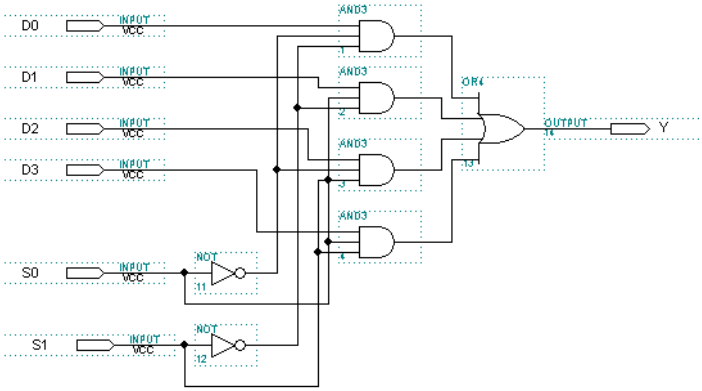
實驗三 多工與解多工器

※多工器(Mux)

多工器 Mux 本質上即是個電子開關，它能由 n 個輸入線中選取一個而連接到輸出線上，至於選取那條輸入線，則由一組控制線，稱為位址選擇線來決定。一般而言，2^n 條輸入線必須有 n 條選擇線，由選擇線的位元組合可決定選擇那一條輸入線。

Ex：4 對 1 多工器

4 對 1 多工器：4 輸入多工器，4 個輸入端 D0~D3，兩位元的位址選擇線 S0 S1，共有四種選擇信號，每種選擇信號使其相對應 AND gate 進入有效狀態最後從 output 端出來。



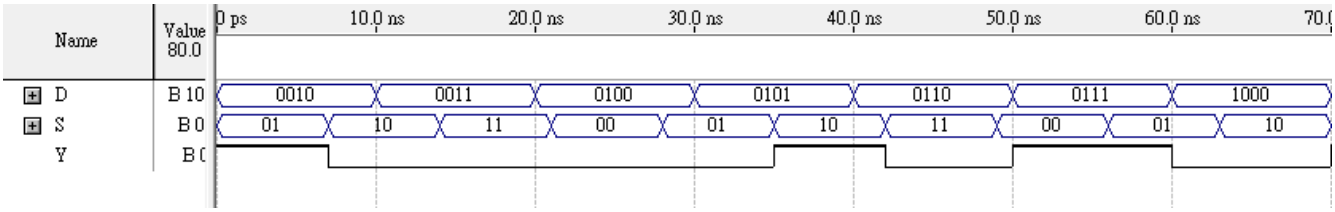
S1	S0	Y(data output)
0	0	D0
0	1	D1
1	0	D2
1	1	D3

工作方式：看 S1S0=10 的情況，gate0. 1. 3 之輸出皆為 0，而 gate2 的輸出信號則視 D2 而定，因此 OR 閘的輸出便等於 D2。

以 Case 方式撰寫 4 對 1 多工器

```
1  /*     MUL4_1    */
2  module MUL4_1 (D, S, Y);
3  input [3:0] D;
4  input [1:0] S;
5  output Y;
6  reg Y;
7  always @(D or S)
8     case(S)
9        2'b00:Y = D[0];
10       2'b01:Y = D[1];
11       2'b10:Y = D[2];
12       2'b11:Y = D[3];
13    endcase
14 endmodule
```

波型模擬

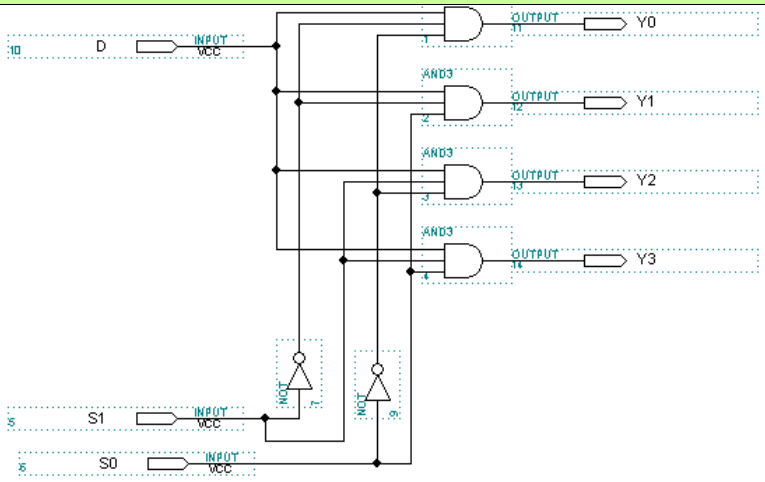


※解多工器(Demux)

解多工器的動作恰與多工器相反，它是將單一的輸入資料送至多個選取的輸出線上。

Ex：1 對 4 解多工器

1 對 4 解多工器：1 輸入解多工器，4 個輸出端 Y0~Y3，兩位元的位址選擇線 S0 S1，共有四種選擇信號，每種選擇信號使其相對應 AND gate 進入有效狀態最後從 output 端出來。



select		output			
S1	S0	Y3	Y2	Y1	Y0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

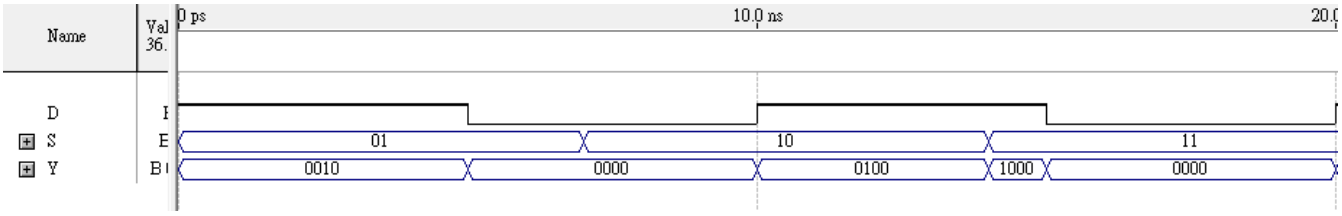
工作方式：當 S1S0=10，輸入 D 被導入 Y2，而其它的輸出都維持不動作的 logic "0"

以巢狀 Case 方式撰寫 1 對 4 解多工器

```
/* DEMUL1_4 */
module DEMUL1_4 (D, S, Y);
input D;
input [1:0] S;
output [3:0] Y;
reg [3:0] Y;

always @(D or S)
   case(S[1])
      1'b0:
         case(S[0])
            1'b0   :Y = {3'b000, D};
            default:Y = {2'b00, D, 1'b0};
         endcase
      1'b1:
         case(S[0])
            1'b0   :Y = {1'b0, D, 2'b00};
            default:Y = {D, 3'b000};
         endcase
   endcase
endmodule
```

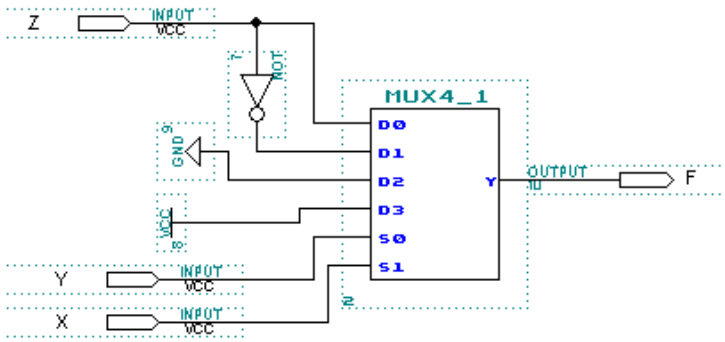
波型模擬



※布林函數的製作：Boolean Function Implemention

可用一個有 n-1 條選擇輸入的多工器，來製作一個 n 個變數的布林函數。取其中 n-1 個變數連接到多工器的選擇線，剩下的一個變數當作多工器的輸入。

Ex: $F(X,Y,Z)=\Sigma m(1,2,6,7)$

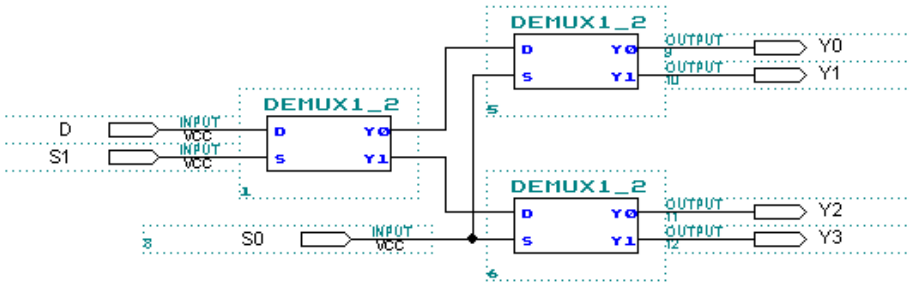


X	Y	Z	F	
0	0	0	0	F=Z
0	0	1	1	
0	1	0	1	F=Z’
0	1	1	0	
1	0	0	0	F=0
1	0	1	0	
1	1	0	1	F=1
1	1	1	1	

Step1:列出布林函數的直值表
Step2:表上的前 n-1 個變數要接到多工器的選擇 input
Step3:最後一個變數來表示其 output，其可為 0、1、變數、變數的補數，接到 Data input。

※(解)多工器的擴充

Ex: 由 3 個 1*2 解多工器組成 1*4 解多工器



以 Component 方式，由 3 個 1*2 解多工器組成 1*4 解多工器

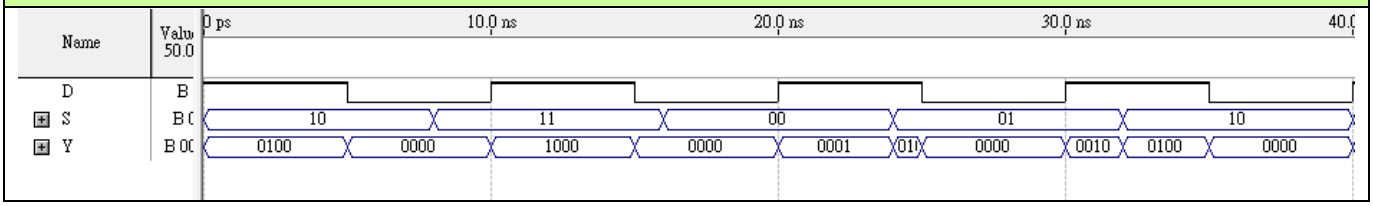
【1*2 解多工器之 component 元件】:

```
1  /* DEMUL1_2 */
2  module DEMUL1_2 (D, S, Y0, Y1);
3  input D, S;
4  output Y0, Y1;
5
6  assign Y0 = D & (~S);
7  assign Y1 = D & S;
8  endmodule
```

【1*4 解多工器】:

```
1  /* DEMUL1_4 */
2  module DEMUL1_4 (D, S, Y);
3  input D;
4  input [1:0] S;
5  output [3:0] Y;
6  wire E, F;
7
8  DEMUL1_2 DM1(D, S[1], E, F);
9  DEMUL1_2 DM2(E, S[0], Y[0], Y[1]);
10 DEMUL1_2 DM3(F, S[0], Y[2], Y[3]);
11
12 endmodule
```

波型模擬



實驗四 加/減法器

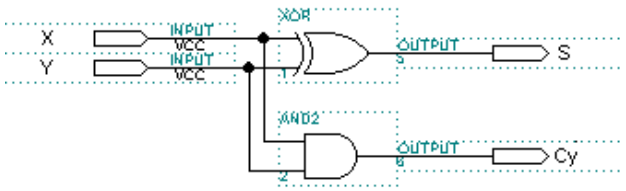
算術電路是個用來執行二進位數字或 BCD 表示的十進位數字的算術運算，如：加、減、乘和除法。由於表示的數碼不同，運算方法也就因而不同，用二進碼表示的數目，它們的運算則就稱為二進運算，用 BCD 碼表示的數目，它們的運算規則就稱為 BCD 運算。

一、半加/全加器(Adder)

基本加法器各級的輸入來自加數、被加數以及前一級的進位，結果產生一個「和」和「進位」輸出，且此一進位輸出會送到下一級(較高位級)。

★半加器：較簡單，只接受加數和被加數做為輸入，由於並沒接受進位，故可用為加法器的最低位級。

X	Y	Cy(進位)	S(和)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

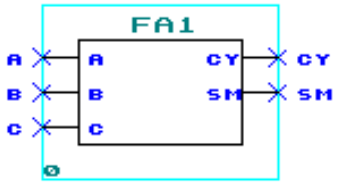


$S=X'Y+XY'=X\oplus Y$

$Cy=XY$

★全加器：包括三個輸入與兩個輸出(和、進位)。

輸入			輸出	
A	B	C	Cy	Sm
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



Sm = f(A,B,C) = Σ(1,2,4,7)

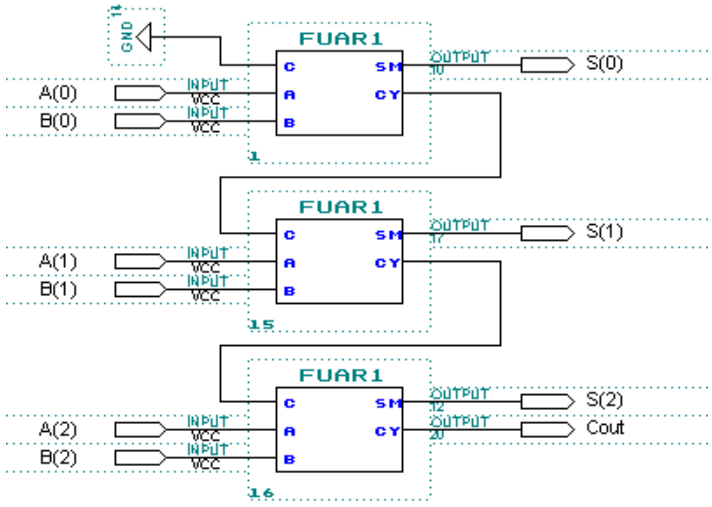
=A’B’C+ABC’+AB’C’+ABC = A⊕B⊕C

Cy = f(A,B,C) = Σ(3,5,6,7)

=A’BC+AB’C+ABC’+ABC = AB+AC+BC

★多位元加法器：若想要完成兩個位元以上的二進制加法工作，可使用全加器的串接。因最低位元不用考慮進位的問題，所以可以使用**半加器**或是使用**全加器但要將進位輸入以“0“輸入**，其它的位數則用全加器，且將前一級的進位輸出連接到下一級的進位輸入。

Ex:三位元加法器



方法一、以 component 方式撰寫三位元加法器

一位元加法器之 component 元件:

```
1  module FULLADDER_1 (Ci, A, B, Sm, Cy);
2  input Ci, A, B;
3  output Sm, Cy;
4  assign Sm = A ^ B ^ Ci;
5  assign Cy = (A & B) | (A & Ci) | (B & Ci);
6  endmodule
```

三位元加法器

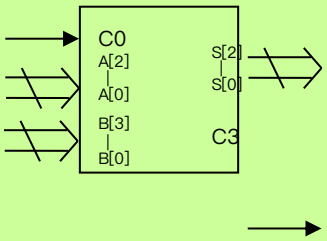
```
1  module FULLADDER_3(A ,B ,S ,Cout);
2    input [2:0] A ,B;
3    output [2:0]S;
4    output Cout;
5    wire s1 ,s2;
6    FULLADDER_1 F1(0  ,A[0] ,B[0] ,S[0] ,s1);
7    FULLADDER_1 F2(s1 ,A[1] ,B[1] ,S[1] ,s2);
8    FULLADDER_1 F3(s2 ,A[2] ,B[2] ,S[2] ,Cout);
9    endmodule
```

方法二、以 for 迴圈方式撰寫

1、 描述

```
                  C0
      A[2]  A[1]  A[0]
+     B[2]  B[1]  B[0]
-------------------------
C3    S[2]  S[1]  S[0]
```

2. 方塊圖



1 位元的全加器布林函式為:

$S[0] = A[0] \oplus B[0] \oplus C0$

$C[1] = A[0]B[0] + A[0]C0 + B[0]C0$

依此類推，下一個位元為:

$S[1] = A[1] \oplus A[1] \oplus C1$

$C[2] = A[1]B[1] + A[1]C1 + B[1]C1$

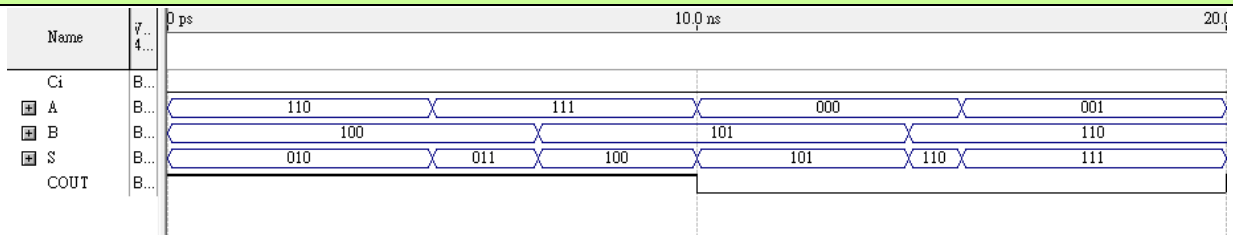
因此其通式為:

$S[i] = A[i] \oplus B[i] \oplus Ci$

$C[i+1] = A[i]B[i] + A[i]Ci + B[i]Ci$

```
1  module fulladder_3bits(A,B,Cin,S,Cout);
2
3  input [2:0]A,B;
4  input Cin;
5  output [2:0]S;
6  output Cout;
7
8  reg [2:0]S;
9  reg Cout,C;
10
11 integer i;
12
13 always @(A or B or Cin or C)
14     begin
15         C = Cin;
16         for (i=0 ; i<3 ; i=i+1)
17             begin
18                 S[i] = A[i] ^ B[i] ^ C;
19                 C = A[i]&B[i] | A[i]&C | B[i]&C;
20             end
21         Cout = C;
22     end
23 endmodule
```

波型圖模擬

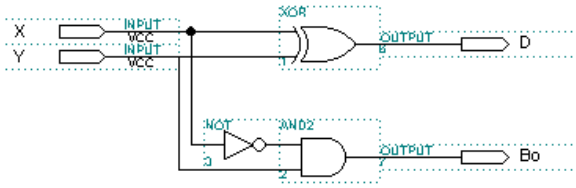


二、半減/全減器(Subtractor)

同於加法器，減法器的輸入有被減數、減數以及前一級的借位輸入，而產生的輸出有「差」和「借位輸出」，此種減法器稱為全減器，如不考慮前一級的位輸入，則稱為半減器。

★半減器：

輸入		輸出	
X(被減數)	Y(減數)	D(差)	Bo(借位)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



$D = X'Y+XY' = X \oplus Y$

$Bo = X'Y$

★全減器：

輸入			輸出	
X(被減數)	Y(減數)	Bi(借位輸入)	D(差)	Bo(借位輸出)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$D=X'Y'Bi + X'YBi'+ XY'Bi'+ XYBi$

$=X \eth Y \eth Bi$

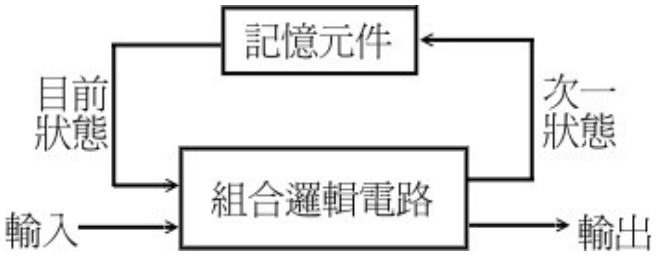
$Bo=X'Y'Bi+X'YBi'+X'YBi+XYBi$

$=X'(Y \eth Bi)+YBi$

實驗五　Latch

先前我們所學的數位電路皆屬於組合邏輯電路，此種電路的目標輸出完全由目前的輸入決定；雖然有不少電路都是由組合邏輯電路所組成，不過大都包含有記憶元件或延遲元件，這種與**時間**因素有關的電路，則稱為循序邏輯(Sequential Logic)電路。

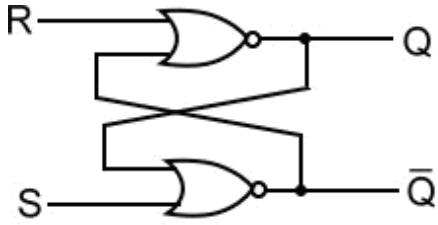
此種循序電路和我們之前所學的組合電路最大的不同在於，循序電路的輸出或系統的次一狀態(next state，簡稱次態)是由目前輸入及目前狀態來決定的。循序電路一般又可分為同步(Synchronous)電路和非同步(Asynchronous)電路兩種，前者指的是循序電路的輸出和狀態的改變是跟隨時脈(Clock)的變化而改變，而後者的輸出和狀態則是隨著輸入的改變而改變。



由於循序電路的目前狀態會影響到次態(參考左圖)，所以我們需要用到記憶元件來記錄狀態，而最基本的記憶元件在同步電路中為正反器(Flip Flop)，非同步電路中則為閂鎖(Latch)；它有一組互為補數關係的輸出 Q 和 Q’(Q bar)，其一次的輸出結果可以一直維持直到再次受到輸入訊號的激勵才會改變。就是這種記憶性質使得正反器在計數器、暫存器等循序電路的設計上扮演了很重要的角色，接下來我們將介紹四種最基本 Latch，而瞭解其運作正是學習循序電路最重要的第一步。

★ SR Latch

SR Latch 的電路圖如下圖 A 所示，它是由兩個 NOR gate 所組成，包含了 S(SET)和 R(RESET)兩個輸入端以及 Q 和 Q’兩個互補的輸出端。



A) SR Latch 電路圖

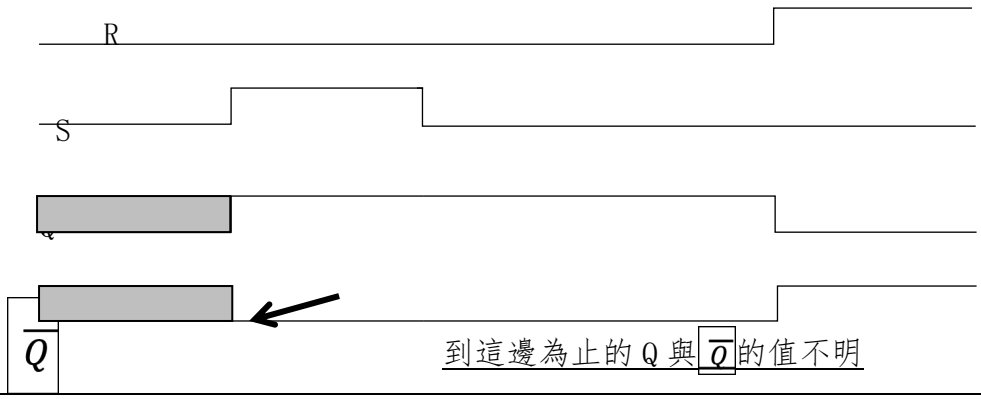
S	R	Q_{t+1}	$\overline{Q}_{t+1}$
0	0	Q_t	$\overline{Q}_t$
0	1	0	1
1	0	1	0
1	1	undefined	

B) 狀態表-1

S	R	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

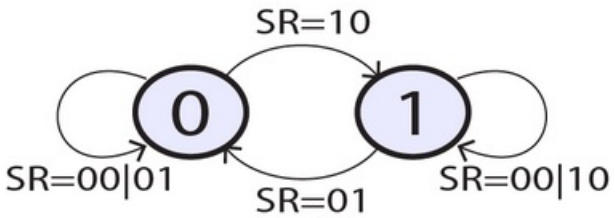
C) 狀態表-2

剛開始給正反器電路電源時，Q 與 $\overline{Q}$ 的值不一定。因為對 R = ‘0’，S = ‘0’ 的狀態來說，Q 與 $\overline{Q}$ 的值都有可能是’0’ 或 ‘1‘ 。唯獨不會兩個都是同樣的值。如下圖：

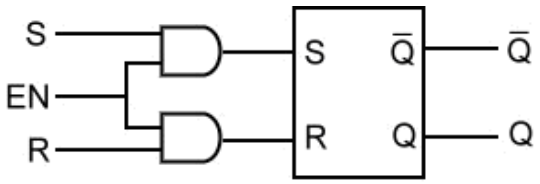


狀態表是用來描述電路的運作及狀態變化，它可用來列舉輸入、輸出及狀態的時序關係值，由上圖 B 來看，可以發現狀態有分為目前狀態(present state，簡稱前態)和次一狀態(next state，簡稱次態)兩種，在時間 t 時當給定輸入和前態，在時間 t+1 時可決定次態值。狀態表的推導是將輸入和前態的所有可能的二進位組合列出，再依電路的函式關係列出次態和輸出值即可完成。我們可由狀態表推得卡諾圖，再由卡諾圖推導出電路的輸出方程式，如將圖 B 的 Qt 當成已知條件，再利用卡諾圖化簡便可推得 SR 正反器的輸出方程式為：$Q_{t+1}=R'(S+Q_t)$

★ SR Latch



D) SR Latch 狀態圖



E) SR Latch with EN

上圖 D 為 SR Latch 的狀態圖，狀態圖和狀態表一樣，都是用來描述電路的行為及變化，由狀態圖更可直接看出狀態的遷移變化情形。一般以一個圓或方塊來表示一種狀態，狀態間的轉變則以箭頭來連結表達，狀態隨著輸入值(寫在箭號的旁邊)的變化可能保持在原來的狀態不變(即箭頭指回原本的這個圈圈)或遷移至另一個狀態(即箭號指向另一個圈圈)。

基本上，電路圖、狀態圖、狀態表和電路方程式都可看出一個電路的行為(功能)，只要知道其中一項，便能推導出其他三項：給你一個電路圖，你就要能從圖推出方程式，進而完成狀態圖及狀態表；給你狀態圖，你就要能從狀態圖推出狀態表，再由狀態表寫出卡諾圖進而推導出電路方程式，有了電路方程式便能畫出電路圖。

```
module test(S, R,Q, Q_B);
```

```
input S, R;
output Q, Q_B;
reg Q, Q_B;
always@(S, R)
begin
Q = ~(R|Q_B);
Q_B = ~(S|Q);
end
endmodule
```

★ 撰寫 Verilog 的注意事項

※ 在撰寫序向邏輯電路的程式時，有一個地方需要特別注意，由於在序向邏輯電路中，會有輸出反接回來成回輸入訊號的情形，如上圖 A 中的輸出 Q 被反接回來和 S 一起做 NOR 運算，但是在 Verilog 中，**輸出是不能拿來做任何運算的**，因此，一定要記得設 **reg** 來替代 output 做運算，等最後運算結束後，再將值傳給真正的 OUTPUT。

※ 區塊式程序指定：所謂區塊式程序指定即為敘述中，執行方式是由上往下依序執行，其特性與一班高階語言相同，它是將右邊表示式處理完後的結果，立刻指定給左邊的物件。

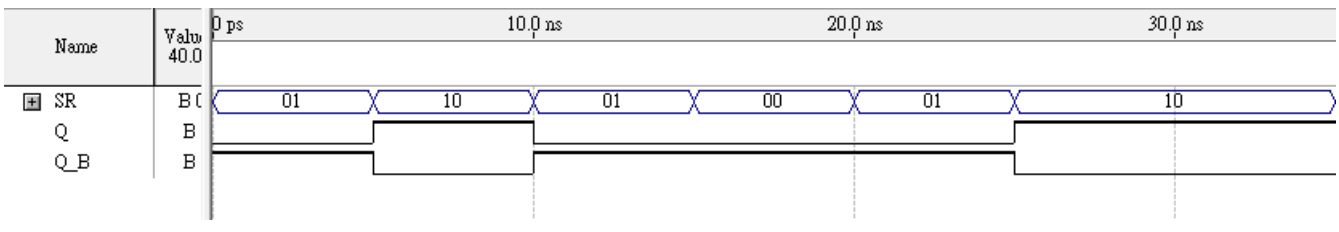
物件名稱 = 表示式

※ **非區塊式程序指定：所謂非區塊式程序指定就是在整個區塊敘述中，當時脈訊號發生預期的變化時，所指定的物件內容也會被改變。**

物件名稱 <= 表示式

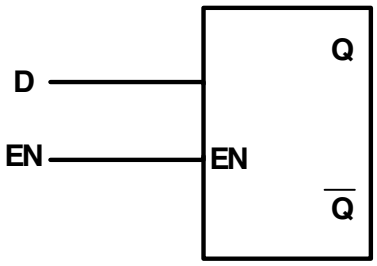
```
1  /* RSLatch  */
2  module RSLatch (S, R, Q, Q_B);
3  input S, R;
4  output Q, Q_B;
5  reg Q, Q_B;
6  always @(S or R)
7     case({S, R})
8        2'b00:begin Q <=Q; Q_B <= Q_B; end
9        2'b01:begin Q <=1'b0; Q_B <= 1'b1; end
10       2'b10:begin Q <=1'b1; Q_B <= 1'b0; end
11    endcase
12 endmodule
```

波行模擬



★D Latch

D Latch 的作用非常單純，當 EN 為 0 時，這個 Latch 不作用，即次態恆等於前態；當 EN 為 1 時，則不論 D Latch 的前態為何，它的次態恆等於輸入 D 的值，即 Q(t+1)=D，這個 D 指的就是 DATA。它的電路圖如下圖 F 所示，圖 G 則是它的狀態表。



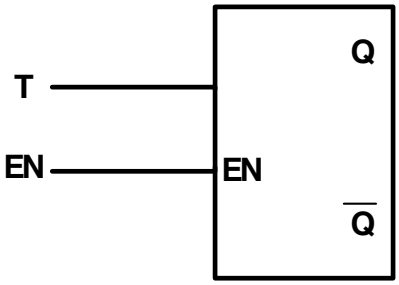
F) D Latch 電路圖

EN	D	Q_t	Q_{t+1}
0	X	0	0
0	X	1	1
1	0	X	0
1	1	X	1

G) 狀態表

★ T Latch

T Latch 和 D Latch 很相似，只是當 T 為 0 時，輸出端 Q 的值將保持不變，即 Q(t+1)=Q(t)；當 T 為 1 時，輸出將是前態的補數，即 Q(t+1)= NOT Q(t)。下圖 H 為它的電路圖，圖 I 則是狀態表，由狀態表及卡諾圖我們可得到 Q(t+1)的函式為：Q(t+1)=T xor Q(t)



H) T Latch 電路圖

EN	T	Q_t	Q_{t+1}
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

I) 狀態表

實驗六 邊緣觸發正反器 與 除頻電路

邊緣觸發正反器(Edge-Triggered Flip-Flop)

latch 運作時，只有輸入數值一有改變輸出就會隨之立即改變，但是這個現象在實際電路執行時會造成很大的不穩定問題，而為了避免這個問題，才有正反器的產生。

◎邊緣觸發式正反器(Edge-Triggered Flip-Flop)

利用 Verilog 來撰寫邊緣觸發的行為，有一個很簡單的判斷語法：

正緣觸發

```
  module JK-FF-B(…);
  …
always @(posedge Clock)
Q <= (J & ~Q)|(~K & Q);
assign Qbar = ~Q;
endmodule
```

負緣觸發

```
 module JK-FF-B(…);
  …
always @(negedge Clock)
Q <= (J & ~Q)|(~K & Q);
assign Qbar = ~Q;
endmodule
```

```
module
JK_FF_example01_SD(J,K,CLK,Q,Qno
t);
     output Q,Qnot;
     input J,K,CLK;

     reg Q;
     assign Qnot = ~ Q ;

     always @ (posedge CLK)
     case ({J,K})
          2'b00: Q = Q;
          2'b01: Q = 1'b0;
          2'b10: Q = 1'b1;
          2'b11: Q = ~ Q;
     endcase
endmodule
```

Q(t)	Q(t + 1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK* Flip-Flop

Q(t)	Q(t + 1)	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T* Flip-Flop

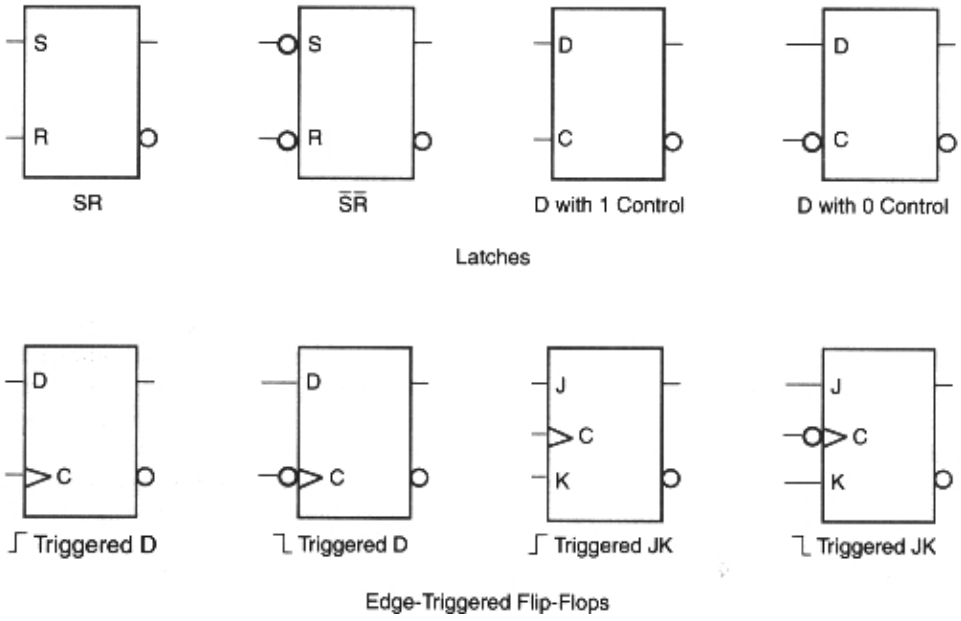
```
module Toggle_flip_flop_1 (Q, T, Clk,
rst);
 output   Q;
 input T, Clk,;
 reg   Q;

 always @ (posedge Clk, negedge rst)
  if (!rst) Q <= 1'b0; //若 rst 不等
於 1 時,
  else if (T) Q <= !Q; //若 rst=1,
且 T=1 時, 將!Q 輸入 Q
endmodule

module Toggle_flip_flop_2 (Q, T, Clk,
rst);
 output   Q;
 input T, Clk, rst;
 wire  DT;
 assign DT = T ^ Q;
 D_flip_flop_AR M0 (Q, DT, Clk, rst);
endmodule

 module Toggle_flip_flop_3 (Q, T, Clk,
rst);
 output   Q;
 input T, Clk, rst;
 reg   Q;
```

```
 always @ (posedge Clk, negedge rst)
  if (!rst) Q <= 1'b0;
  else  Q <= Q ^ T;
endmodule
```

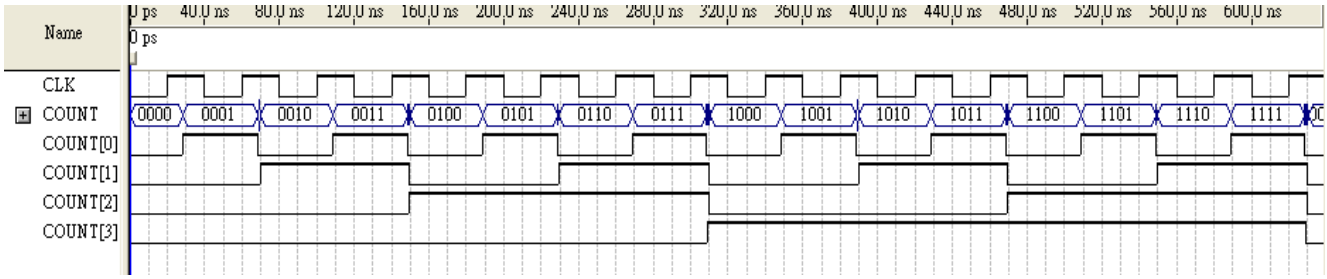


(圖)Latch 和 Flip-Flop 符號

除 頻 電 路

我們所使用的數位模擬版上有一顆振盪頻率為 16MHz 的振盪器，它決定了產生數位電路的時脈週期，由於這個頻率太高（一秒內振盪了 16,000,000 次），肉眼無法辨識，所以我們必須利用除頻電路來降低它的頻率。

除頻電路的原理其實很簡單，從計數器來看就可明白。一個正緣觸發的 4 位元的上數計數器的波形結果下：



仔細觀察波形結果，不難發現，COUNT[0]的波形變化週期剛好是 CLK 的 2 倍、COUNT[1]是 4 倍、COUNT[2]是 8 倍、COUNT[3]是 16 倍，以頻率來說，就是除以 2、除以 4、除以 8、除以 16。

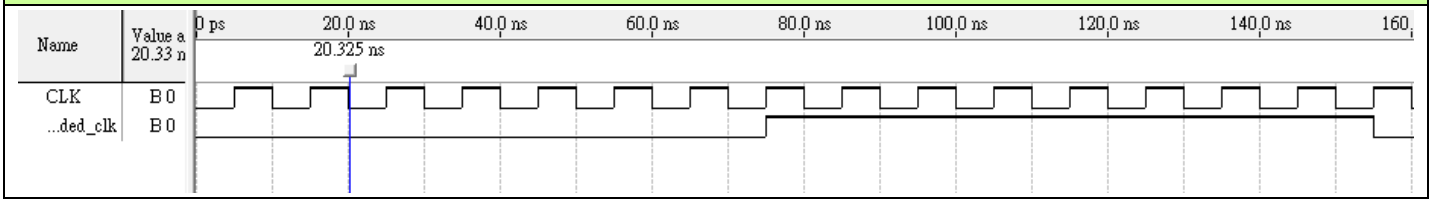
而由於我們所使用的數位板的振盪頻率有 16M，因此我們需設計一個除以 16M 的除頻電路，讓它變成一秒運算一次，這樣才可以肉眼觀察到執行的結果。

$$\frac{16MHz}{16} = 1\ M\ HZ = 1 \times 10^6\ HZ$$

除以 16 的除頻電路之程式如下:

```
1  /* Divider16 */
2  module Divider_16(CLK, divided_clk);
3  input CLK;
4  output divided_clk;
5  reg [3:0] cnt;
6  always@(posedge CLK)
7  if(cnt == 15)
8     cnt <= 0;
9  else
10    cnt <= cnt + 1;
11 assign divided_clk = cnt[3];
12 endmodule
```

波行模擬



現在，請參照上面的程式寫法，自行撰寫一除以 16M 的除頻電路。

假設輸出名稱：Q

電路功能：每秒 Q 會變化一次，由亮變暗，或由暗變亮。(即一直做反向)

寫好程式後請將電路 download 到數位模擬板上，輸出 Q 指定給 LED 燈 D1。

注意，clk 訊號指定之腳位為 PIN_18，且 IP1_12 需短路!

(往後的電路 clk 都是這樣設定，請記下。)

※請將寫好的 16M 除頻電路 upload 到你的網路信箱，之後的電路都需要用到它喔!

```
/* Divider16 */
module Divider16 (clk, div_CLK);
input clk;
output div_CLK;
reg [23:0] cnt;
always@(posedge clk)
    if (cnt==15999999)
        cnt <= 0;
    else
        cnt <= cnt+1;
assign div_CLK = cnt[23];
endmodule
```