Lab 2 Gate-Level Modeling

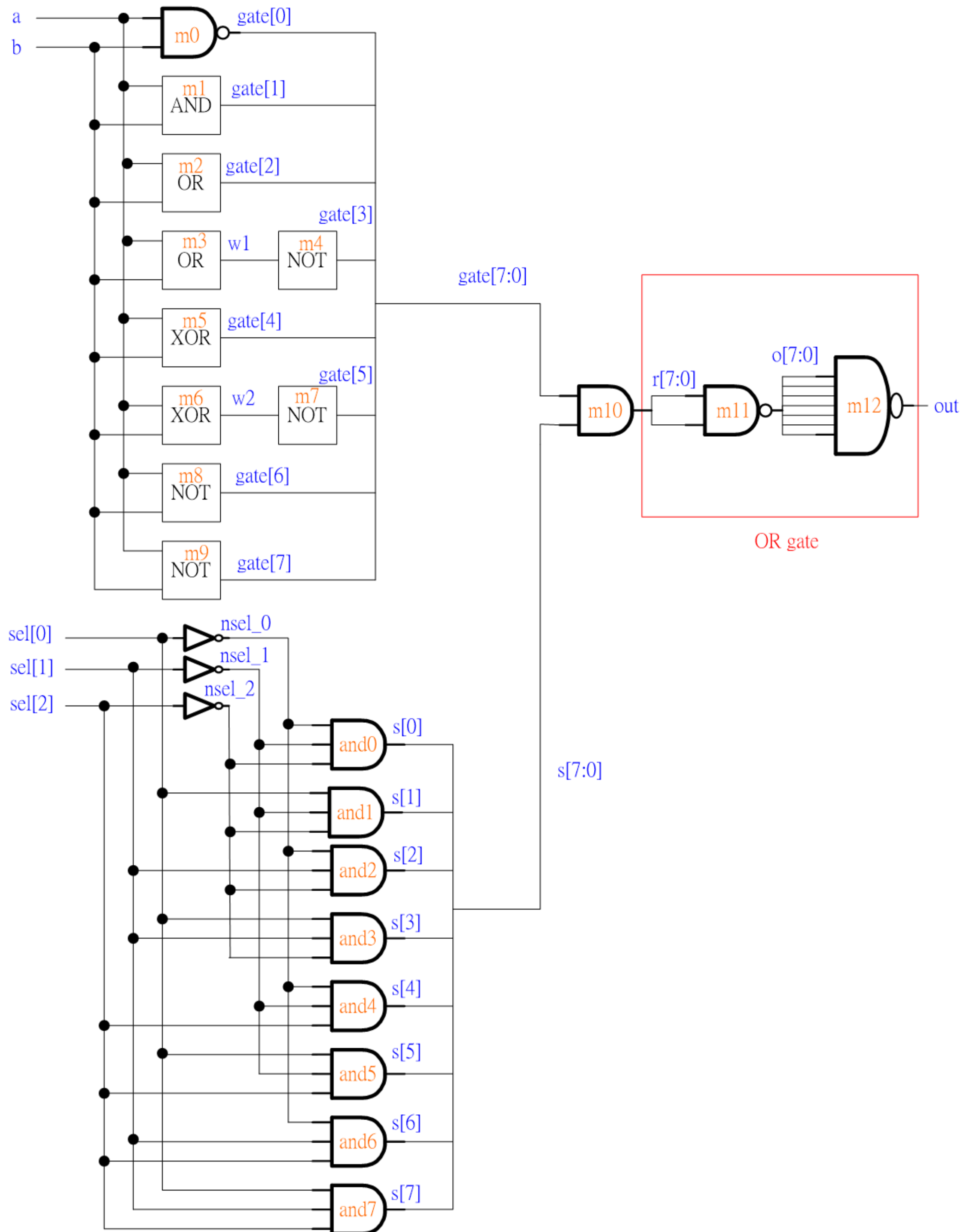
實驗報告

組長:劉奇泓 109033135

組員:洪聖祥 109062315

1. 8-bit ripple carry adder

- NAND gates only



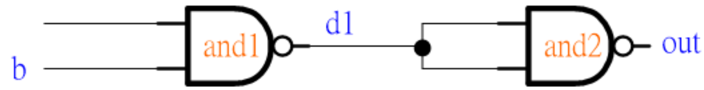
```
module NAND_Implement (a, b, sel, out);
input a, b;
input [2:0] sel;
output out;
wire nsel_0, nsel_1, nsel_2;
wire [7:0] gate, s, r, o;
wire w1, w2;

NOT not0(sel[0], nsel_0);
NOT not1(sel[1], nsel_1);
NOT not2(sel[2], nsel_2);

threebitAND and0(nsel_2, nsel_1, nsel_0, s[0]);
threebitAND and1(nsel_2, nsel_1, sel[0], s[1]);
threebitAND and2(nsel_2, sel[1], nsel_0, s[2]);
threebitAND and3(nsel_2, sel[1], sel[0], s[3]);
threebitAND and4(sel[2], nsel_1, nsel_0, s[4]);
threebitAND and5(sel[2], nsel_1, sel[0], s[5]);
threebitAND and6(sel[2], sel[1], nsel_0, s[6]);
threebitAND and7(sel[2], sel[1], sel[0], s[7]);
```

```
nand m0 (gate[0], a, b); // nand gate
AND  m1 (a, b, gate[1]); // and gate
OR   m2 (a, b, gate[2]); // or gate
OR   m3 (a, b, w1);
NOT  m4 (w1, gate[3]); // nor gate
XOR m5 (a, b, gate[4]);// xor gate
XOR m6 (a, b, w2);
NOT m7 (w2, gate[5]);// xnor gate
NOT  m8 (a, gate[6]);// not gate
NOT  m9 (a, gate[7]);// not gate
AND m10 [7:0] (gate, s, r);

nand m11 [7:0] (o, r, r);
nand m12(out, o[0], o[1], o[2], o[3], o[4], o[5], o[6], o[7]);

endmodule
```

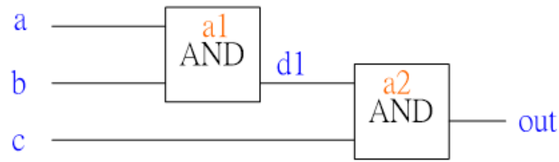
Ripple carry adder 會使用到 basic question 製作的 nand gates only、3 input majority gate 以及 full adder。Nand gates only 中所有的 gates 都是以 nand gate 為基底製作，我們可以依照順序用 sel 選擇 nand、and、or、xor、xnor 及 not gate。圖中 m10 的 and gate 是將 gate[7:0] 及 s[7:0]依照相對應的位數 and，以實現用 sel 選擇要輸出哪個 function。m11 及 m12 等效於一個 8-bit OR gate，最後輸出 out。

用 nand gate 製作的 gates:

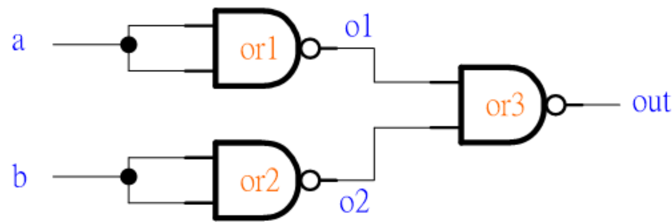
AND



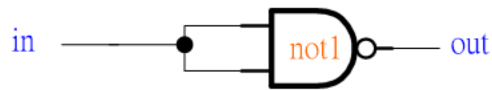
threebitAND



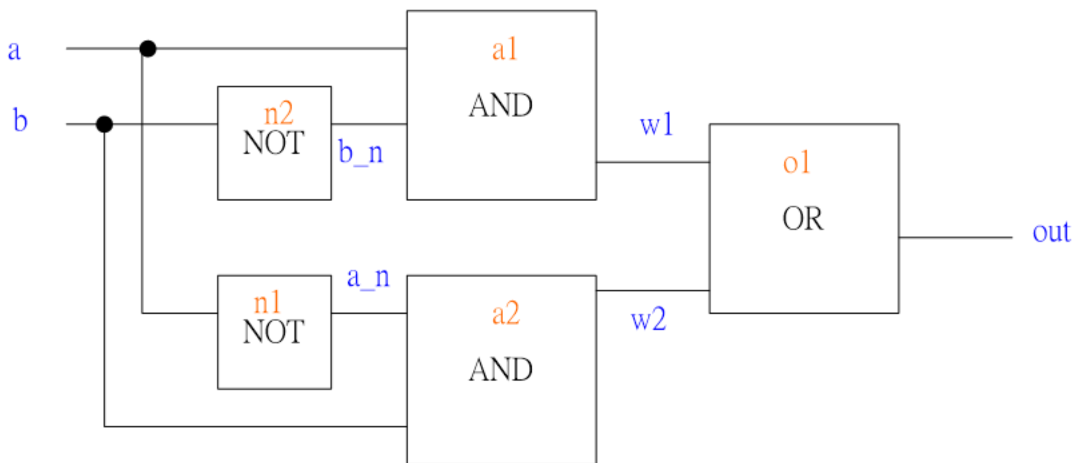
OR



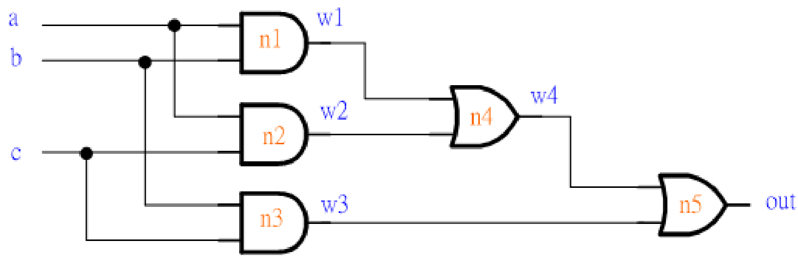
NOT



XOR



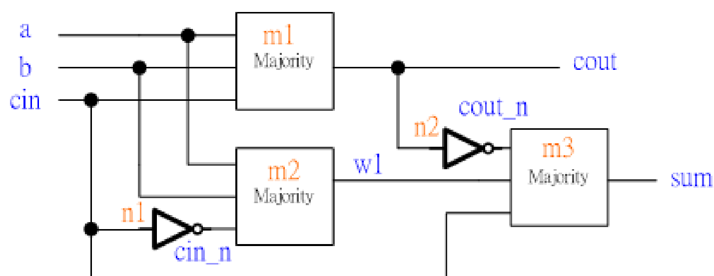
- 3 input majority gate




```
module Majority(a, b, c, out);
input a, b, c;
output out;
wire w1, w2, w3, w4;

NAND_Implement n1(a, b, 3'b001, w1);
NAND_Implement n2(a, c, 3'b001, w2);
NAND_Implement n3(b, c, 3'b001, w3);
NAND_Implement n4(w1, w2, 3'b010, w4);
NAND_Implement n5(w3, w4, 3'b010, out);

endmodule
```

majority gate 的 gates 都是使用 nand gate only 中製作的 gates。majority gates 只要在 a、b、c 三個 input 有兩個以上是 1 時就會輸出 1。

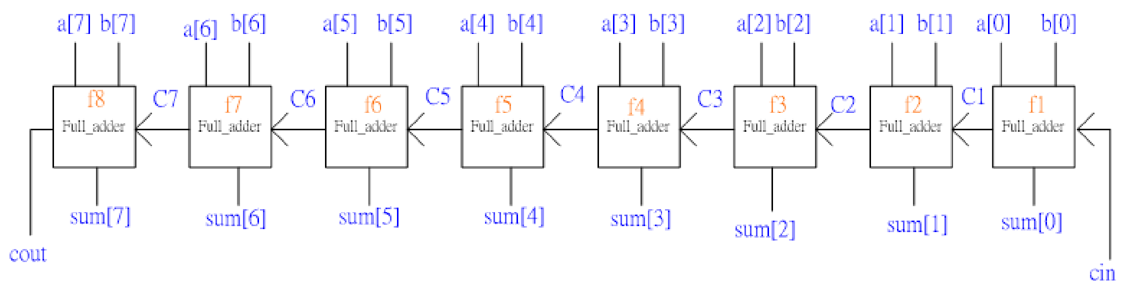
- **Full adder**



```
module Full_Adder (a, b, cin, cout, sum);
input a, b, cin;
output cout, sum;
wire wl;
wire cin_n, cout_n;

NAND_Implement nl(cin, 1'b0, 3'b110, cin_n);
NAND_Implement n2(cout, 1'b0, 3'b110, cout_n);
Majority ml(a, b, cin, cout);
Majority m2(a, b, cin_n, wl);
Majority m3(cout_n, wl, cin, sum);

endmodule
```

Full adder 使用在 basic question 中，以 3 input majority gate 製作，not gate 則使用 nand gates only 製作的 not gate。m1 是測試 a、b、cin 是否有兩個以上是 1，如果是的話就必定進位，並輸出 cout = 1。m2 與 m3 的部分會測試 a、b、cin 的值，只有在其中一個或全部 input 都是 1 的時候才會輸出 sum = 1。

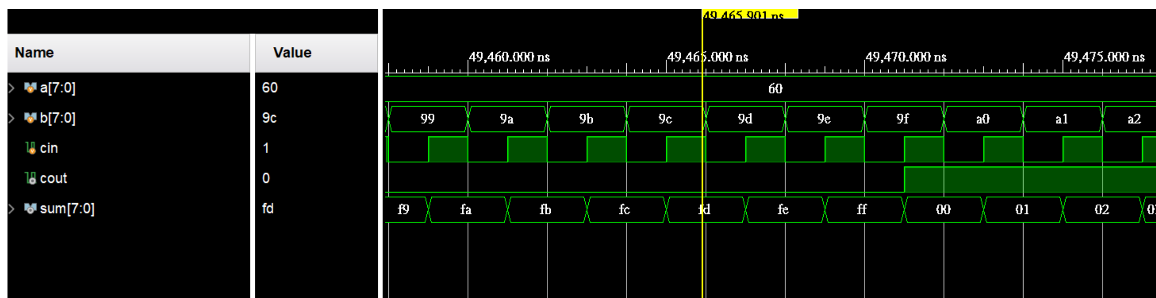
- **8-bit ripple carry adder**



```
module Ripple_Carry_Adder(a, b, cin, cout, sum);
input [7:0] a, b;// input
input cin;// carry in
output cout;// carry out
output [7:0] sum;// output
wire C1, C2, C3, C4, C5, C6, C7;// carry bits

// instantiate 8 full adder to construct a 8-bit ripple carry adder
Full_Adder f1(a[0], b[0], cin, C1, sum[0]);
Full_Adder f2(a[1], b[1], C1, C2, sum[1]);
Full_Adder f3(a[2], b[2], C2, C3, sum[2]);
Full_Adder f4(a[3], b[3], C3, C4, sum[3]);
Full_Adder f5(a[4], b[4], C4, C5, sum[4]);
Full_Adder f6(a[5], b[5], C5, C6, sum[5]);
Full_Adder f7(a[6], b[6], C6, C7, sum[6]);
Full_Adder f8(a[7], b[7], C7, cout, sum[7]);

endmodule
```

從左到右總共 8 個 1 bit full adder，a、b 為兩個相加的數，從最前面開始為 LSB，a[0]、b[0]與 cin(carry in)相加，輸出該位數的值 sum[0]，並把 carry out C1 傳遞到下一個 full adder 進行下個位數的加法，持續這樣的步驟到 MSB，最後相加的結果為 sum，如果有進位則 cout = 1。

波形圖：

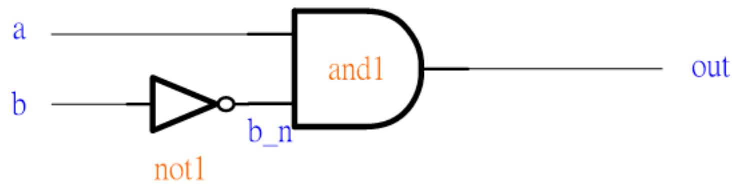


```
module Lab2_Team30_Ripple_Carry_Adder_t;
reg [7:0] a, b;// input signal
reg cin;// carry in
wire cout;// carry out
wire [7:0] sum;// output wire

// instantiate the main module
Ripple_Carry_Adder m1(a, b, cin, cout, sum);

initial begin
a = 8'b00000000;
b = 8'b00000000;
cin  = 1'b0;// set all input signals to 0

// test all the input combinations, chane the signal every 1 ns
repeat(2 ** 17)begin
    #1 {a, b, cin} = {a, b, cin} + 1'b1;
end

#1 $finish;// finish the test
```

用 repeat 迴圈嘗試 a、b、cin 的所有組合，以測試 sum 及 cout 是否產生錯誤。

2. Decode and execute

- Universal gate

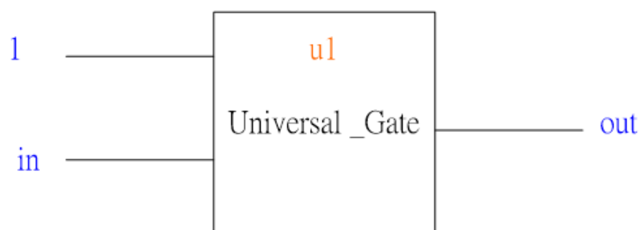


使用一個 AND 及 NOT 來實現 universal gate

以 universal gate 製作的 basic logic gates:

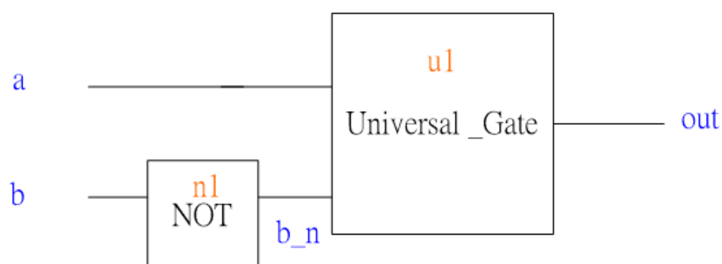
NOT:

in 輸入會經過 not，因此和 1 and 後輸出 out = in’



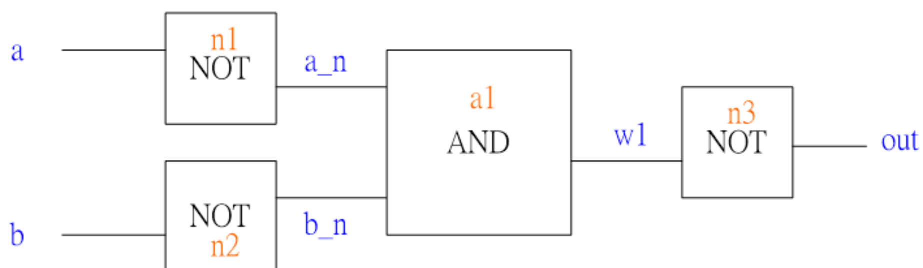
AND:

b 經 NOT 輸入至 universal gate 後會再經過 not，互相抵銷，out = a • b



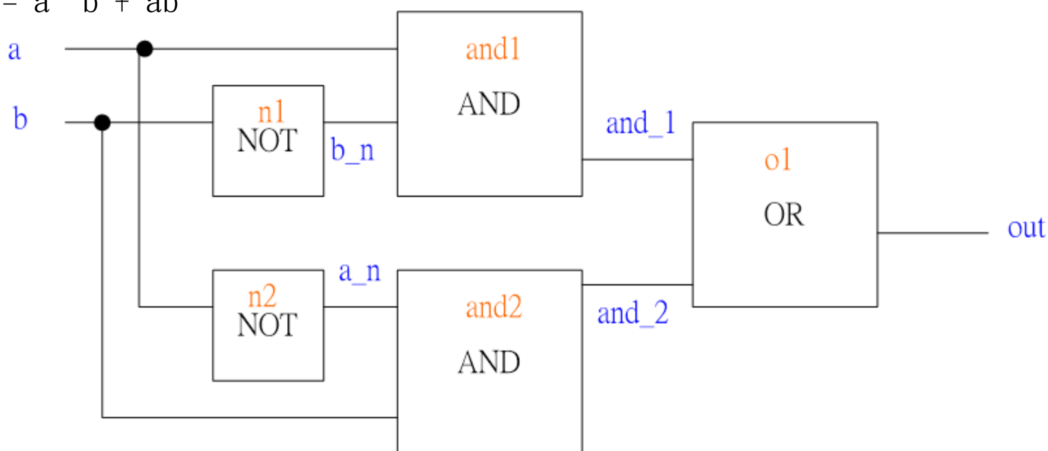
OR:

(a’ • b’)’
= a + b

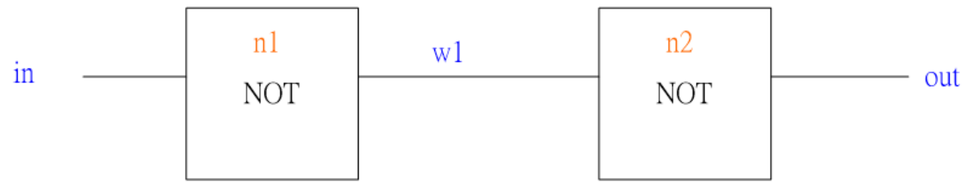


XOR:

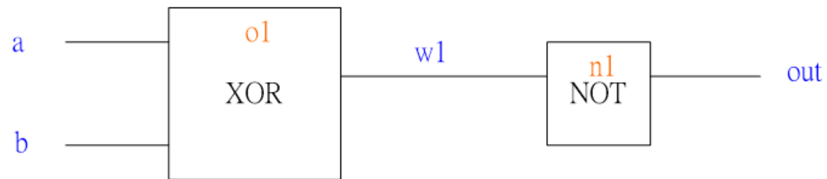
a⊕b = a’ b + ab’



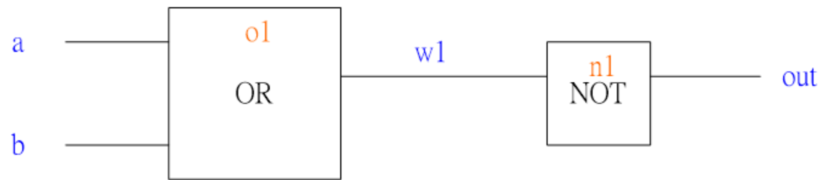
buffer:



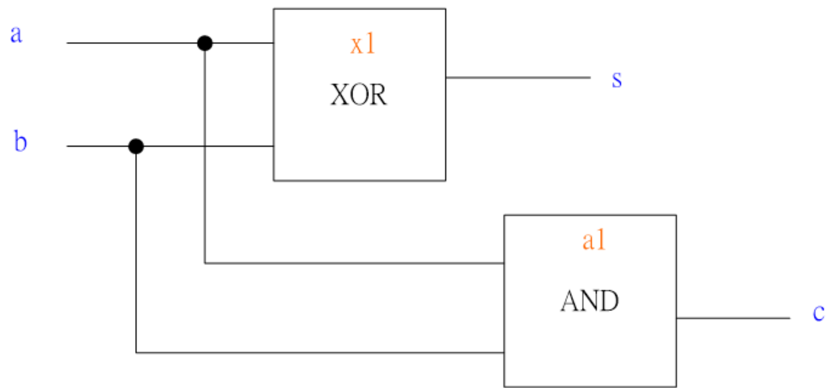
XNOR:



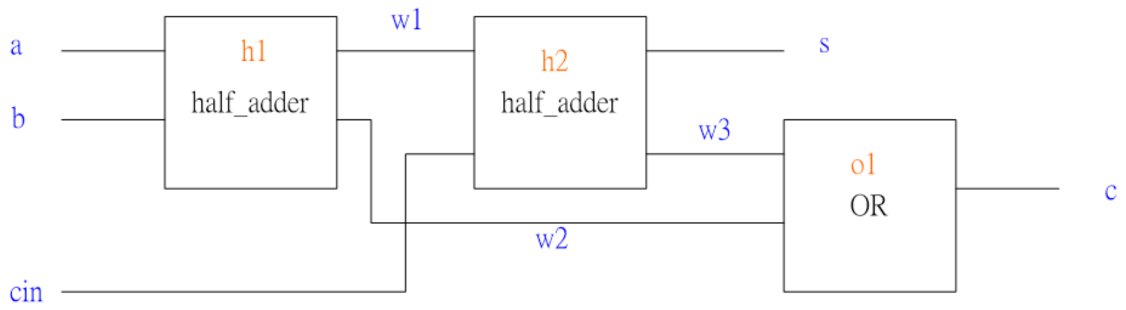
NOR:



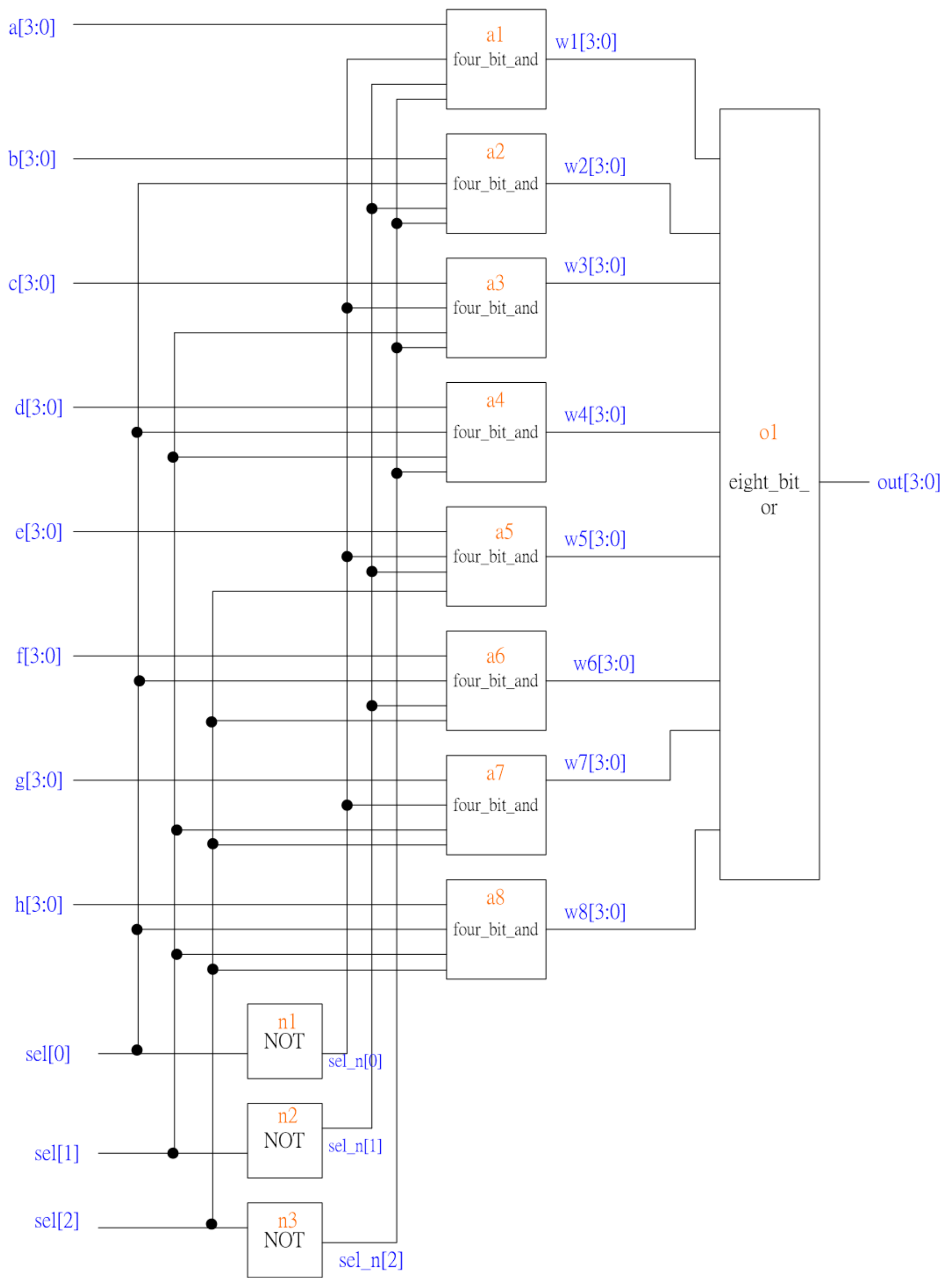
Half adder:



Full adder:

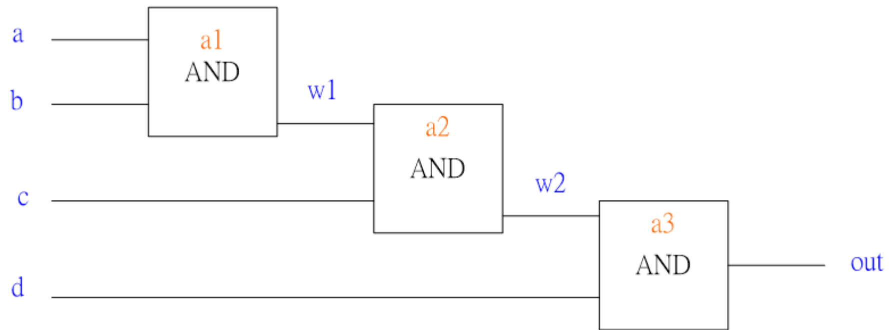


- 4-bit 8X1 MUX

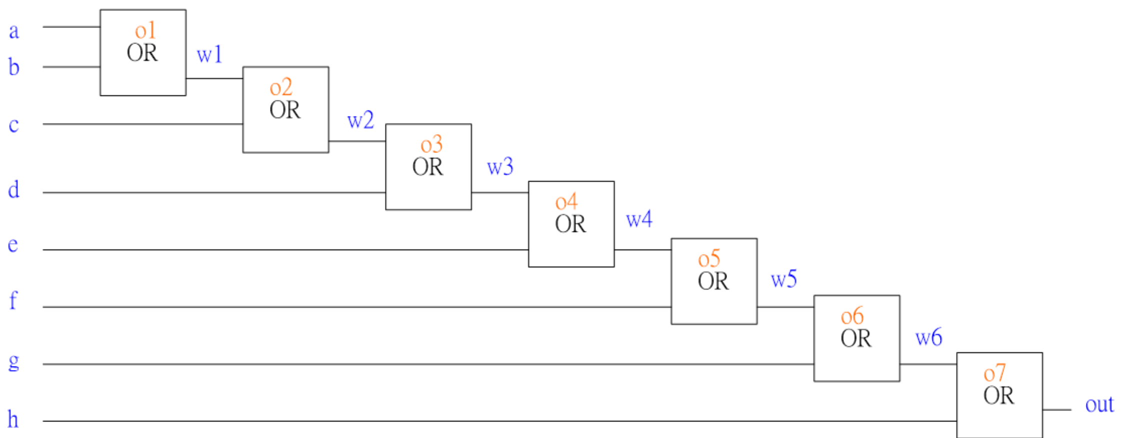


此為在選擇執行的 function 時會使用到的 MUX

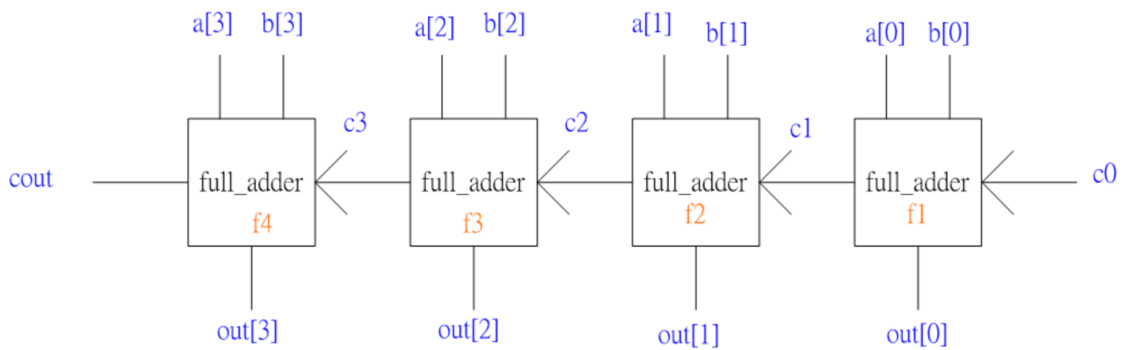
four bit and:



eight bit or:



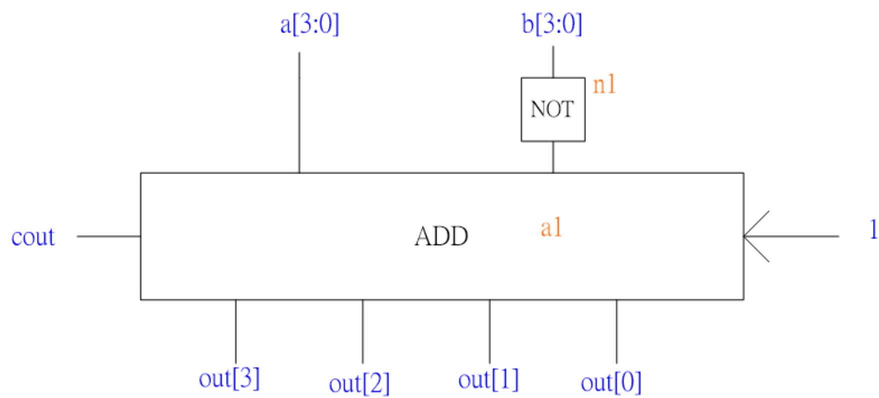
- ADD



```
module ADD (a, b, c0, out, cout);
  input [3:0] a, b;
  input c0;
  output [3:0] out;
  output cout;
  wire c1, c2, c3, c4;
  // instantiate 4 full adders to construct a 4-bit ripple carry adder
  full_adder f1(a[0], b[0], c0, out[0], c1);
  full_adder f2(a[1], b[1], c1, out[1], c2);
  full_adder f3(a[2], b[2], c2, out[2], c3);
  full_adder f4(a[3], b[3], c3, out[3], cout);

endmodule
```

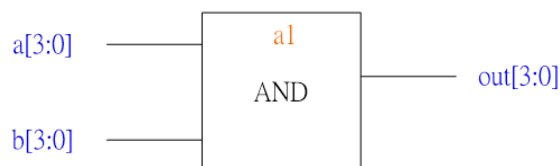
ADD 使用四個 1-bit full adder，輸出 out[3:0]和 cout。

- SUB



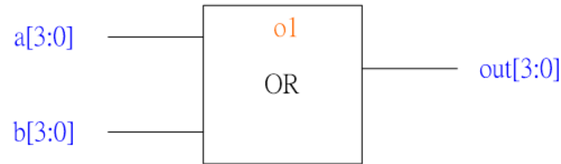
把減數 b 用 NOT gate 產生 1’s complement，並將 cin 輸入 1，使 b 變成 2’s complement，再加回 a 後即能得到 subtraction 的結果，輸出 out[3:0]和 cout。

- BITWISE AND



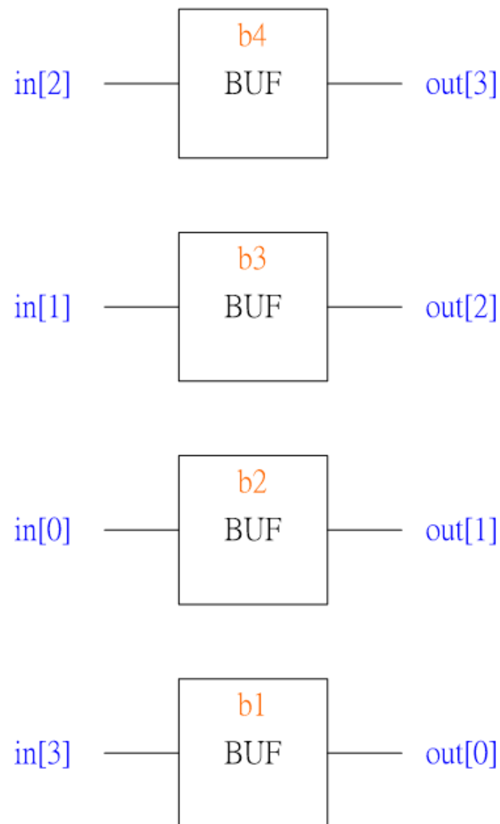
a, b 的 4 個 bit 對應的位數經過 AND 得到 out

- BITWISE OR



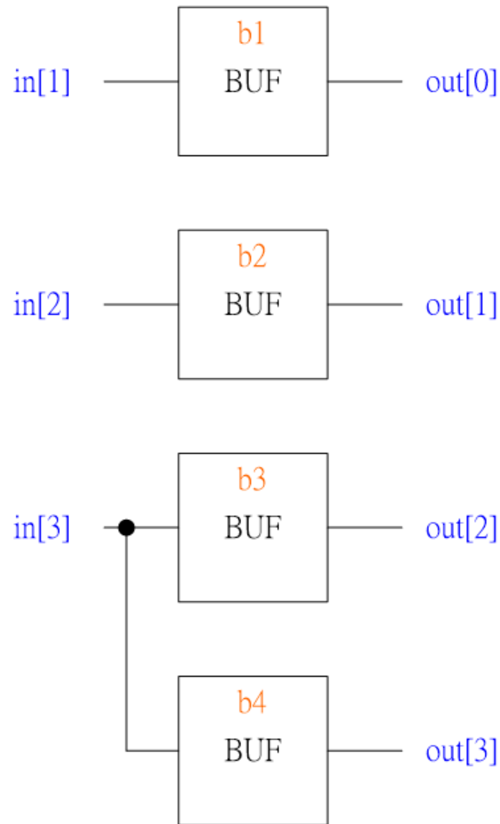
a, b 的 4 個 bit 對應的位數經過 OR 得到 out

- RS CIR. LEFT SHIFT



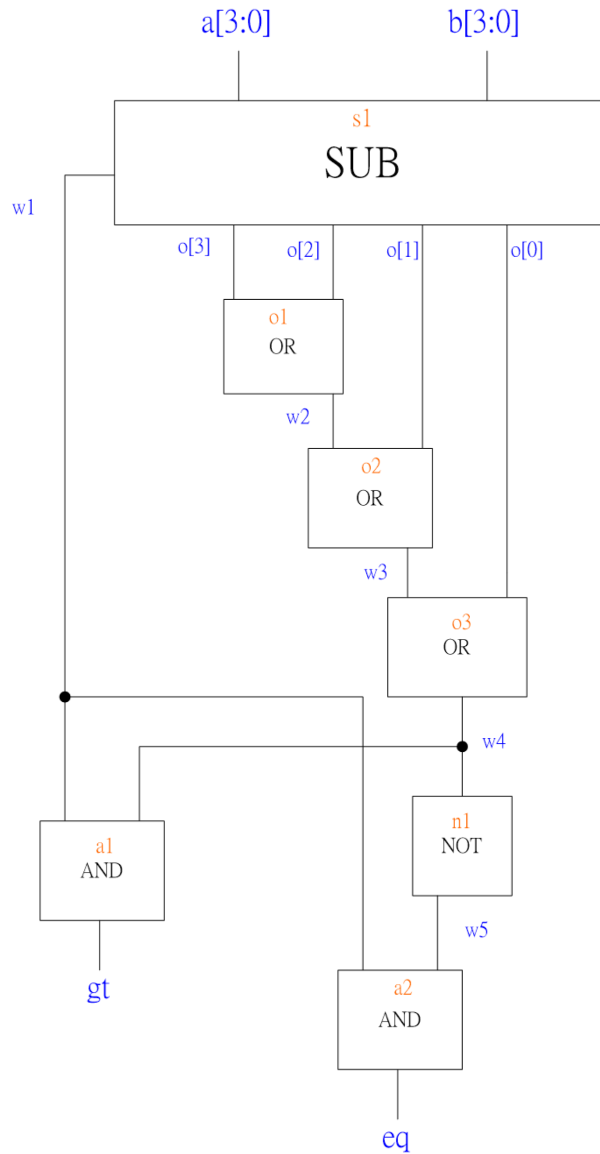
input 經過一個 buffer，把 MSB 輸出到 output 的 LSB，其餘的 bits 都往後推移一位

- RT ARI. RIGHT SHIFT



input 經過一個 buffer，把 LSB 捨去，其餘的 bits 往前推移一位輸出到 output，in[3]則同時輸出到 out[2], out[3]以保持原本的 sign bit。

- COMPARE

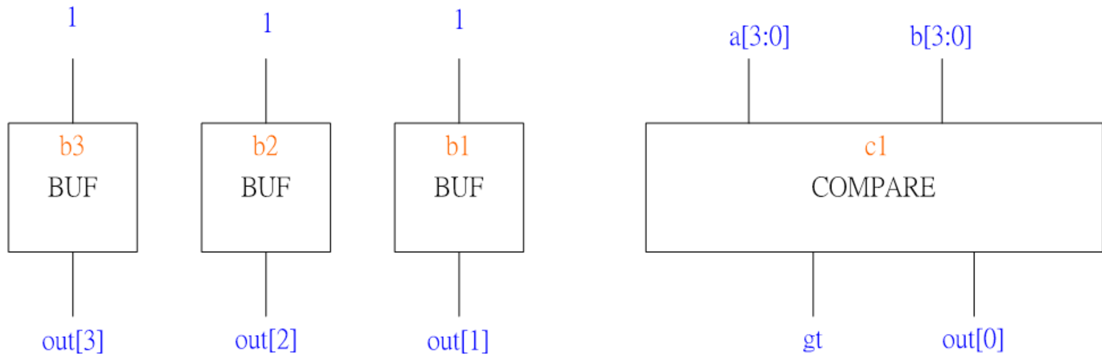



```
module COMPARE(a, b, gt, eq);
input [3:0] a, b;
output gt, eq;
wire [3:0] o;
wire w1, w2, w3, w4, w5;
SUB s1(a, b, o, w1); // instantiate a SUB module

// w4 = 1 when o = 4'b0000
OR o1(o[3], o[2], w2);
OR o2(o[1], w2, w3);
OR o3(o[0], w3, w4);
NOT n1(w4, w5);

AND a1(w1, w4, gt);
AND a2(w1, w5, eq);

endmodule
```

comparator 使用兩數相減的方式來比較大小，當 a = b 時相減的結果一定是 10000，所以只會在 w1 = 1 及 o[3:0] = 0000 時 eq 才會

輸出 1；當 a > b 時 w1 = 1，且 o 不全為 0，因此只會在 w1 = 1 及 o 有一個以上不為 0 時 gt 才會輸出 1。

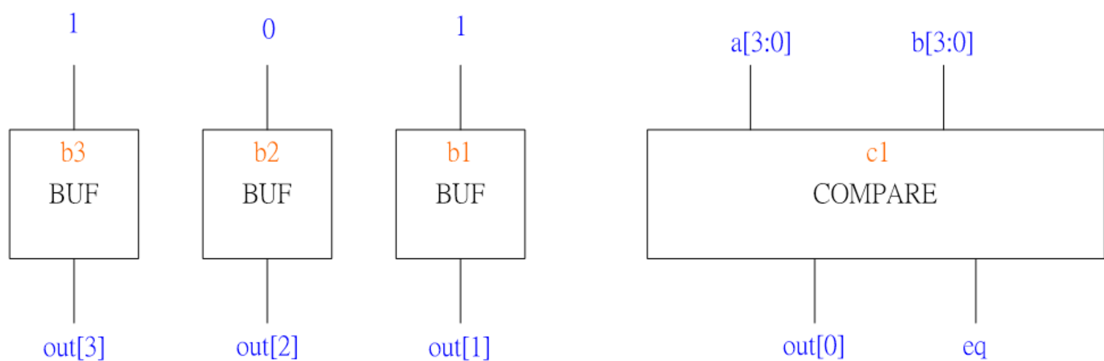
- COMPARE EQ




```
module COMPARE_EQ(a, b, out);
input [3:0] a, b;
output [3:0] out;
wire gt;
// use 3 buffer to assign 1'b1 to out[1], out[2], out[3]
BUF b1(1'b1, out[1]);
BUF b2(1'b1, out[2]);
BUF b3(1'b1, out[3]);
// instantiate a COMPARE module
// connect eq with out[0]
COMPARE c1(a, b, gt, out[0]);

endmodule
```

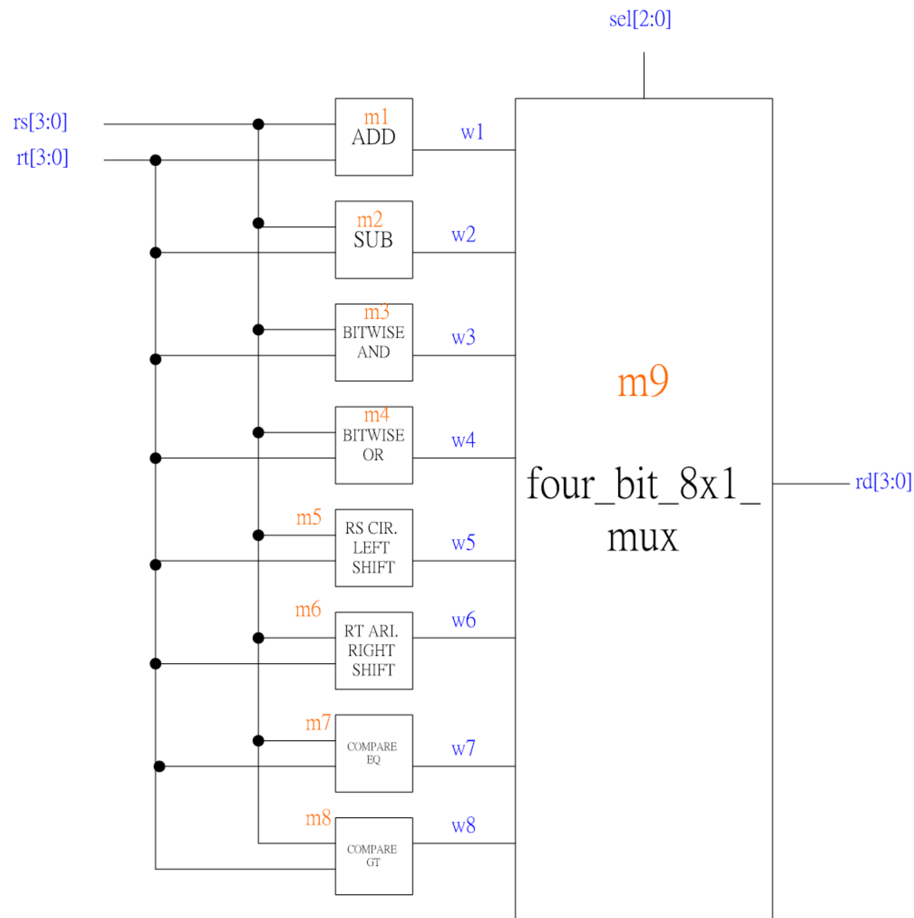
把 COMPARE 的 output eq 接到 out[0]

- COMPARE GT



把 COMPARE 的 output gt 接到 out[0]

- Decode and execute



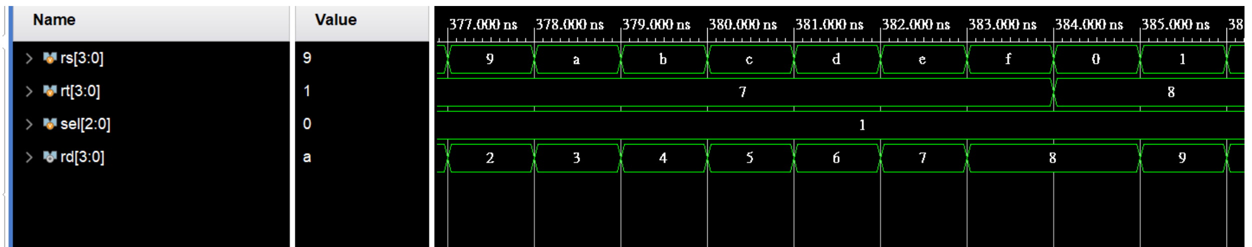

```
module Decode_And_Execute(rs, rt, sel, rd);
 input [3:0] rs, rt; // input signal
 input [2:0] sel; // selection signal
 output [3:0] rd; // output signal
 wire [3:0] w1, w2, w3, w4, w5, w6, w7, w8;
 wire cout1, cout2;// the carry out of ADD and SUB, which I don't need

 // all the instructions
 ADD m1 (rs, rt, 1'b0, w1, cout1);// cin = 1'b0
 SUB m2 (rs, rt, w2, cout2);
 BITWISE_AND m3 (rs, rt, w3);
 BITWISE_OR m4 (rs, rt, w4);
 RS_CIR_LEFT_SHIFT m5 (rs, w5);
 RT_ARI_RIGHT_SHIFT m6 (rt, w6);
 COMPARE_EQ m7 (rs, rt, w7);
 COMPARE_GT m8 (rs, rt, w8);

 // input results into 8x1 MUX to select output
 four_bit_8x1_mux m9(w1, w2, w3, w4, w5, w6, w7, w8, sel, rd);

endmodule
```

把 rs[3:0]、rt[3:0]訊號輸入將全部的 function 都執行完畢後，

再用 sel[2:0]和 MUX 選擇要輸出的訊號。cout1 和 cout2 分別是

ADD 和 SUB 輸出的 carry out，因為最後結果 rd 只有 4 bits，因此 cout 被接到 wire 而不輸出到 output。

波形圖:



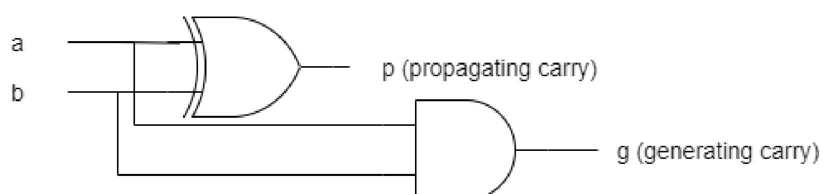

```
module Lab2_Team30_Decode_And_Execute_t;
reg [3:0] rs, rt;// input signal
reg [2:0] sel;// selection bits
wire [3:0] rd; // output wire

// instantiate the main module
Decode_And_Execute m1 (rs, rt, sel, rd);

initial begin
rs = 4'b0000;
rt = 4'b0000;
sel = 3'b000;// all input are 0s

// test every input combinations, change signals every 1 ns
repeat(2 ** 11)begin
    #1 {sel, rt, rs} = {sel, rt, rs} + 1'b1;
end

#1 $finish; // finish the test
end
```

用 repeat 迴圈嘗試 sel、rt、rs 的所有組合，以測試 function 以及 rd 是否產生錯誤。

從下圖開始 and xor or gate 的圖都是用 basic question 用 nand gate 實作

3. 8-bit carry-lookahead (CLA) adder

Half adder

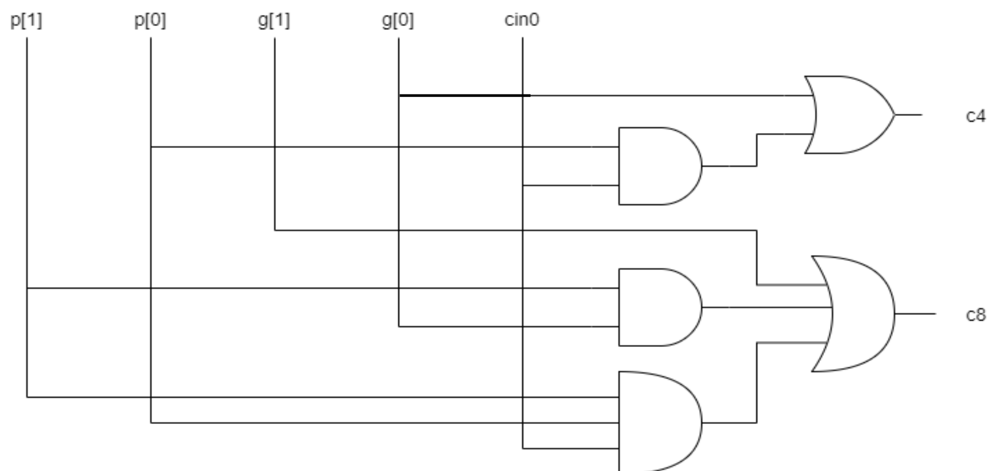


```
module half_adder_1bit(a, b, p, g);
input a, b;
output p, g;
//propagating function
XOR x0(a, b, p);
//generating function
AND a0(a, b, g);
endmodule
```

Half adder 2 個 output p 和 g。p 代表 propagating carry 表示當 input carry(cin)是 1 時才會進位也就是只有當 a 或 b 是其中一個是 1 時 p 才會是 1。 g 代表 generating carry 表示不管有沒有 input carry(cin)都會進位也就是只有當 a 和 b 是 1 時 g 才會是 1。

$$p = a \oplus b$$

$$g = a * b$$



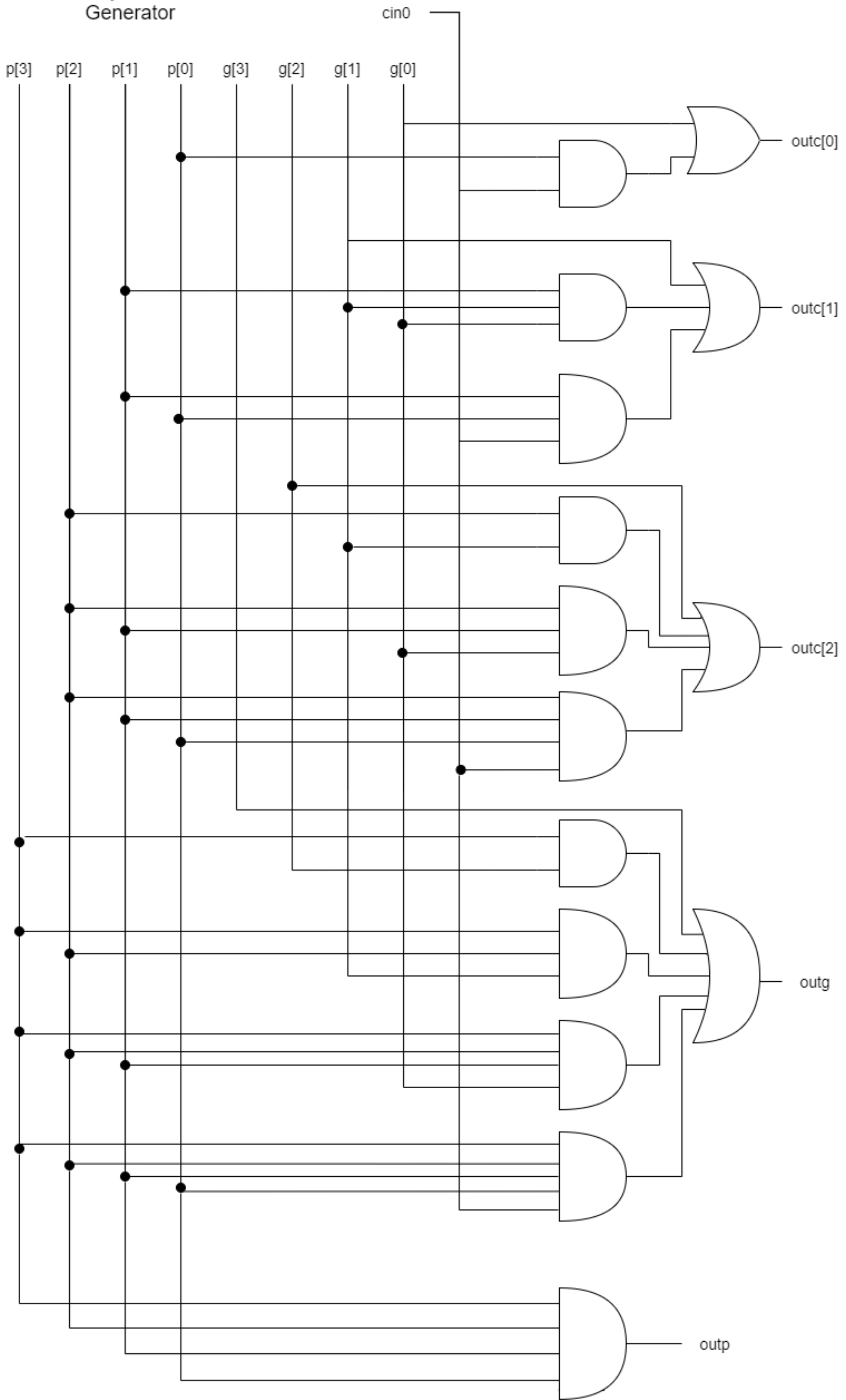
```
145
146 module CLA_Gen2_bits(p, g, cin0, c4, c8);
147 input [1:0] p, g;
148 input cin0;
149 output c4, c8;
150 wire [2:0] tmp;
151
152 AND a0(p[0], cin0, tmp[0]);
153 OR or0(tmp[0], g[0], c4);
154
155 AND a1(p[1], g[0], tmp[1]);
156 AND_3bits a2(cin0, p[0], p[1], tmp[2]);
157 OR_3bits or1(tmp[1], tmp[2], g[1], c8);
158 endmodule
```

2bit Carry-Look-Ahead Generator 有 2 個 outputs。上圖中 p[0]表示在我們要計算 8bit a+b 中 a[0]~a[3]和 b[0]~b[3]的 propagating carry，p[1]表示 a[4]~a[7]和 b[4]~b[7]的 propagating carry。g[0]表示 a[0]~a[3]和 b[0]~b[3]的 generating carry，g[1]表示 a[4]~a[7]和 b[4]~b[7]的 generating carry。

我們要知道是否要進位可以藉由 p、g 和 cin0(前一個位數是否進位)決定。因此有一個等式:

C(i+1) = G(i)+P(i)C(i)

上面的等式可以理解成要進位的話 generating carry = 1(可自行進位，像 2 進位的 1+1=10)或是前一個位數有進位而且 propagating carry = 1(像是 11+01 = 100，第二個 bit propagating carry = 1 而且前一個位數有進位)。將上面等式展開就可以得到上面的邏輯設計圖。


4-bit Carry-Look-Ahead
Generator
cin0
p[3]
p[2]
p[1]
p[0]
g[3]
g[2]
g[1]
g[0]
outc[0]
outc[1]
outc[2]
outg
outp

4-bit Carry-Look-Ahead Generator output 跟 2-bit Carry-Look-Ahead Generator 類似但我們想要把它的 output 接上 2-bit Carry-Look-Ahead Generator 以計算 C4 和 C8。因此本來要算 C4 變成計算 p[0,3]和 g[0,3]，本來要算 C8 變成計算 p[4,7]和 g[4,7]。p[0,3]表示第 0bit 到第 3bit 是否可以產生 propagating carry 而 g[0,3]表示第 0bit 到第 3bit 是否可以產生 generating carry，同理 p[4,7]和 g[4,7]。

我們可以利用前面的等式 C(i+1) = G(i)+P(i)C(i)來計算各個 bit 的 carry。因此本來要算的

C4 =g[3]+g[2]p[3]+g[1]p[2]p[3]+g[0]p[1]p[2]p[3]+c0p[0]p[1]p[2]p[3]

因為 g[0,3] = 1 時不管有沒有 c0 第四個 bit 都會自行進位，所以我們可以將 C4 最後一項刪掉就可以得到 g[0,3]。而因為 p[0,3] = 1 時 c0 要 = 1 才會進位。因此當 p[0]~p[3] = 1 時 p[0,3]才會 = 1。也就是 p[0,3] = p[0]p[1]p[2]p[3]。同理可知 p[4,7]和 g[4,7]

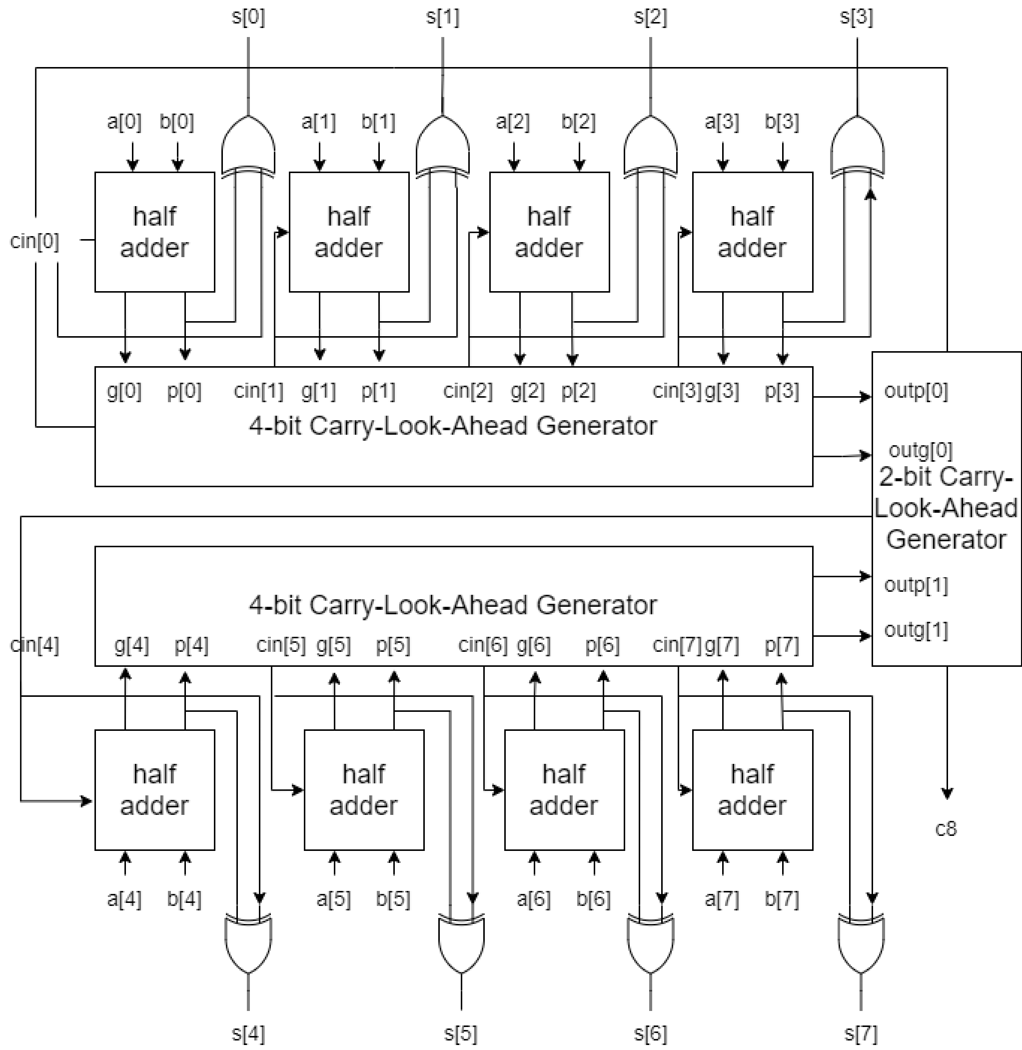
與 2-bit Carry-Look-Ahead Generator 不一樣的地方:

```
131
132 AND a3(p[2], g[1], tmp[3]);
133 AND_3bits a4(p[2], p[1], g[0], tmp[4]);
134 AND_4bits a5(cin0, p[0], p[1], p[2], tmp[5]);
135 OR_4bits or2(tmp[3], tmp[4], tmp[5], g[2], outc[2]);
136
137 AND a6(p[3], g[2], tmp[6]);
138 AND_3bits a7(p[3], p[2], g[1], tmp[7]);
139 AND_4bits a8(p[3], p[2], p[1], g[0], tmp[8]);
140 OR_4bits or3(tmp[6], tmp[7], tmp[8], g[3], outg);
141
142 AND_4bits a10(p[0], p[1], p[2], p[3], outp);
143 endmodule
144
```

計算 g[0,3]

計算 p[0,3]

8-bit Carry Look Ahead Adder




```
module Carry_Look_Ahead_Adder_8bit(a, b, c0, s, c8);
input [7:0] a, b;
input c0;
output [7:0] s;
output c8;
wire [7:0] p, g ;
wire tmp;
wire [7:0] cin;
wire [1:0] outp, outg;
NOT n0(c0, tmp);
NOT n1(tmp, cin[0]);
CLA_Gen4_bits gen0(p[3:0], g[3:0], cin[0], outp[0], outg[0], cin[3:1]);
CLA_Gen4_bits gen1(p[7:4], g[7:4], cin[4], outp[1], outg[1], cin[7:5]);
CLA_Gen2_bits gen2(outp, outg, cin[0], cin[4], c8);
half_adder_1bit ha[7:0](a, b, p, g);
calculate_sum cs[7:0](cin, p, s);
endmodule
```

8-bit Carry-Look-Ahead adder 由 8 個 half adder、2 個 4-bit Carry-Look-Ahead Generator、1 個 2-bit Carry-Look-Ahead Generator 組成和 XOR gate 組成。我把計算 sum 的部分分開是因為我希望可以清楚表示 half adder 這個 module 的功能，不然也可以把計算 sum 的部分和 half adder 合起來就跟助教的圖一樣。

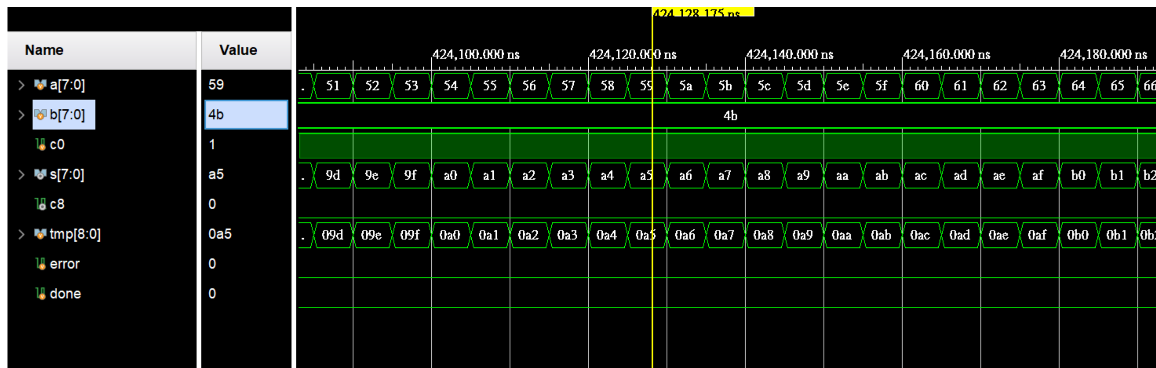
這裡計算 sum 的部分跟 full adder 的邏輯一樣，full adder 的 sum 是 sum = a⊕b⊕cin。因為 p[i]表示第 i 個 bit 的 propagating carry 也就是 p[i] = a[i] ⊕ b[i]。我們只要再跟 c[i]做 XOR 就好了。因此，s[i] = p[i] ⊕ cin[i]。

其他部分就跟前面一樣，4 個 half adder 接 4-bit Carry-Look-Ahead Generator，2 個 4-bit Carry-Look-Ahead Generator 接 2-bit Carry-Look-Ahead Generator 就完成 8-bit Carry-Look-Ahead adder 了。

優點:

Ripple carry adder 跟 Carry-Look-Ahead adder 不一樣的是 ripple carry adder 跟我們用直式計算加法一樣，都是先算完前一位數後再計算後一位數，缺點是有 gate delay。Carry-Look-Ahead adder 則是提前先計算所有位數的 generating carry propagating carry cin 的關係，所以所有位數是平行運算不會有 gate delay。

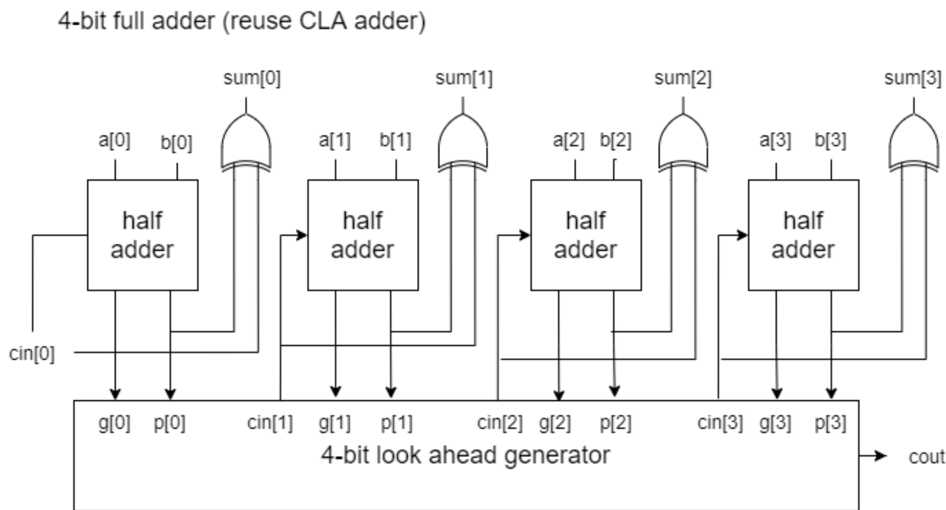
波形圖：




```verilog
29  reg error = 1'b0, done = 1'b0;
30  Carry_Look_Ahead_Adder_8bit CLA(a, b, c(
31  initial begin
32  a = 8'b00000000;
33  b = 8'b00000000;
34  c0 = 1'b0;
35  tmp = 9'b000000000;
36  repeat(2)begin
37      repeat (2 ** 8)begin
38          repeat (2 ** 8)begin
39              tmp = a+b+c0;
40              #1 error = 1'b0;
41              if (tmp!={c8, s})
42                  error = 1'b1;
43              #4
44              a = a + 8'b00000001;
45
46          end
47          b = b + 8'b00000001;
48      end
49      c0 = c0 + 1'b1;
50  end
51  #1 error = 1'b0;
52  #4 done = 1'b1;
53  #5done = 1'b0;
54  $finish; // finish the test
55  end
56  endmodule
57
```

用 repeat 窮舉 a、b、c0 的所有可能並以 tmp 為標準確認 adder 是否有錯(跟 exhuasted testing 類似)

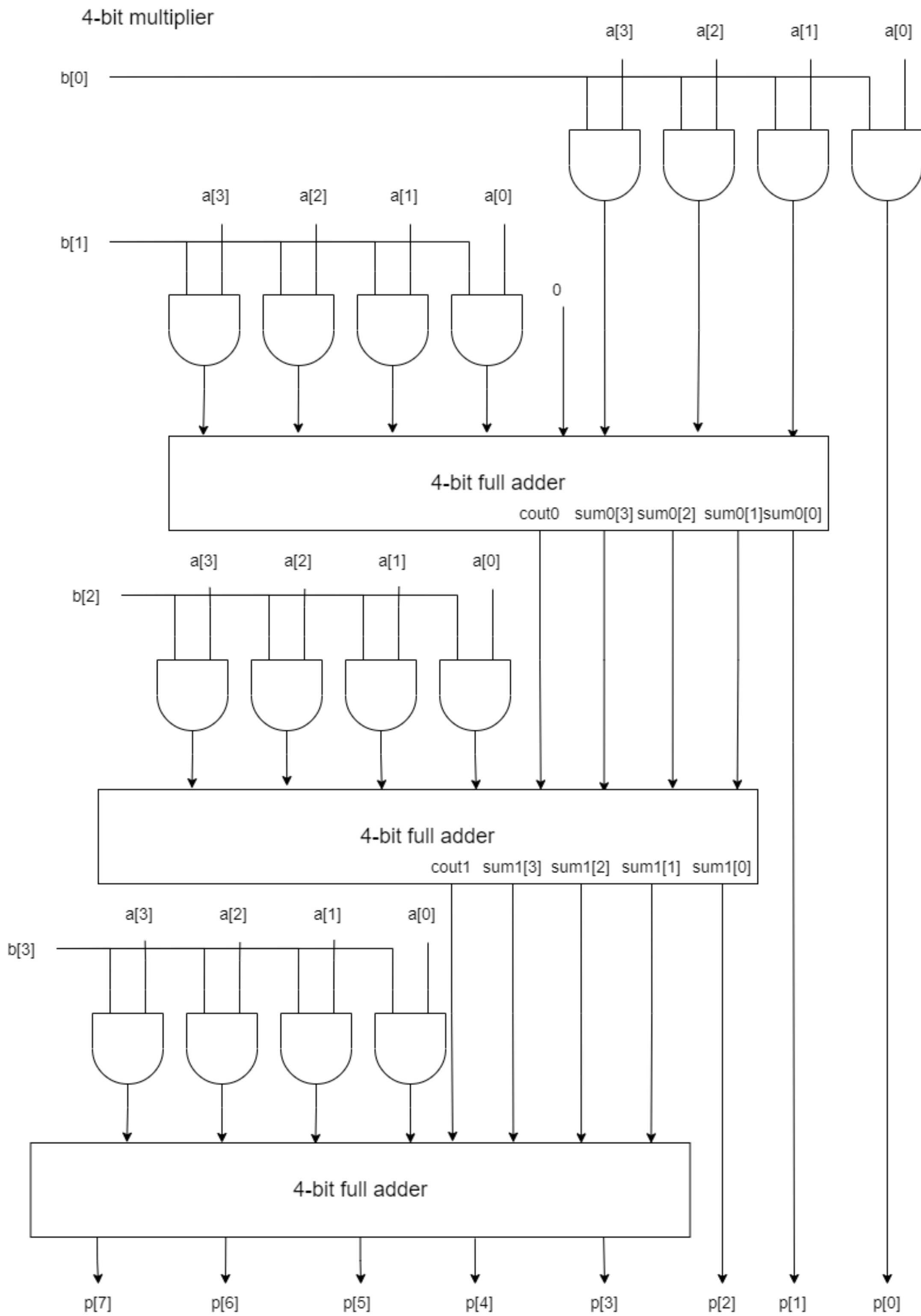
4. 4-bit multiplier



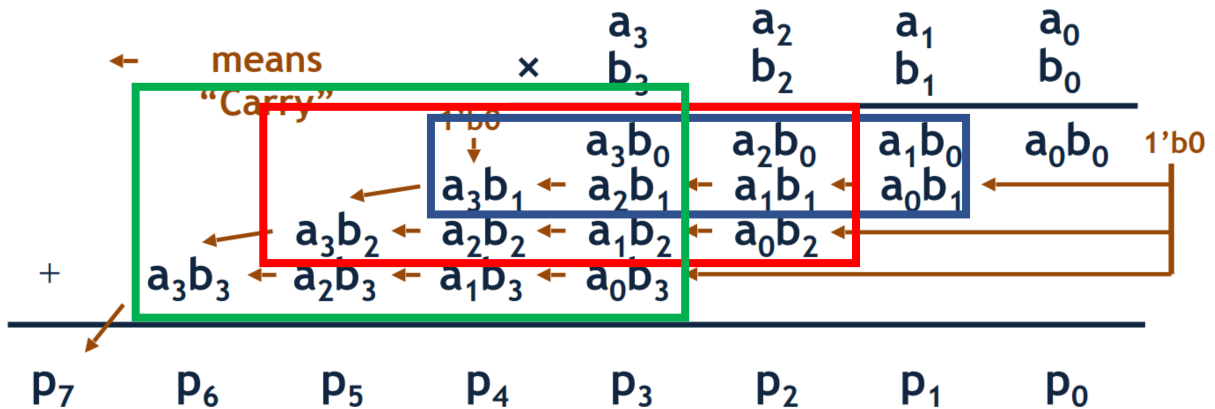
4-bit full adder 我們沿用之前的 8-bit Carry-Look-Ahead Adder 中取我們要的 4 bits。因為只有 4 bits，所以我們的 4-bit look ahead generator 跟之前的設計有些許不同。我們直接計算 cin[4]來當 cout。一樣是用 C(i+1) = G(i)+P(i)C(i)這個等式。不一樣的地方:

```
70
71  AND a6(p[3], g[2], tmp[6]);
72  AND_3bits a7(p[3], p[2], g[1], tmp[7]);
73  AND_4bits a8(p[3], p[2], p[1], g[0], tmp[8]);
74  AND_5bits a9(p[3], p[2], p[1], p[0], cin0, tmp[9]);
75  OR_5bits or3(tmp[6], tmp[7], tmp[8], tmp[9], g[3], cout);
76
```

直接計算 cout

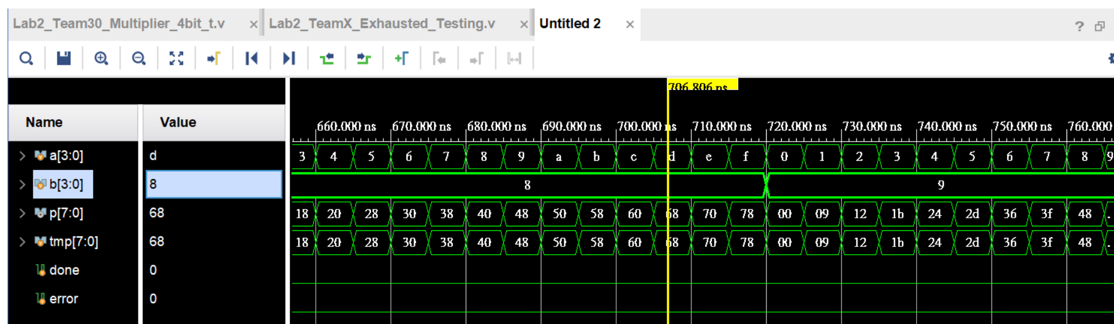

4-bit multiplier
a[3]
a[2]
a[1]
a[0]
b[0]
a[3]
a[2]
a[1]
a[0]
b[1]
0
4-bit full adder
cout0
sum0[3]
sum0[2]
sum0[1]
sum0[0]
a[3]
a[2]
a[1]
a[0]
b[2]
4-bit full adder
cout1
sum1[3]
sum1[2]
sum1[1]
sum1[0]
a[3]
a[2]
a[1]
a[0]
b[3]
4-bit full adder
p[7]
p[6]
p[5]
p[4]
p[3]
p[2]
p[1]
p[0]

4-bit multiplier 由 3 個 4-bit adder 和 AND gate 組成。我們可以由下圖知道:



我們可以先用 AND gate 來算 a[0]b[0]、a[1]b[0]…a[3]b[3]。圖中藍色方框、紅色方框、綠色方框代表 3 個 4-bit adder，接下來就可以算出答案了。

波形圖:




```
30  a = 4'b0000;
31  b = 4'b0000;
32  done = 1'b0;
33  error = 1'b0;
34  tmp = 7'b0000000;
35  repeat(2**4)begin
36      repeat(2 ** 4)begin
37      tmp = a*b;
38      #1
39      if (tmp!=p)
40          error = 1'b1;
41      else
42          error = 1'b0;
43      #4
44      a = a+ 4'b0001;
45      end
46      b = b + 4'b0001;
47  end
48  #1 error = 1'b0;
49  #4 done = 1'b1;
50  #5done = 1'b0;
```

用 repeat 窮舉 a、b 的所有可能並以 tmp 為標準確認 multiplier 是否有錯(跟 exhuasted testing 類似)

5. Exhausted testbench

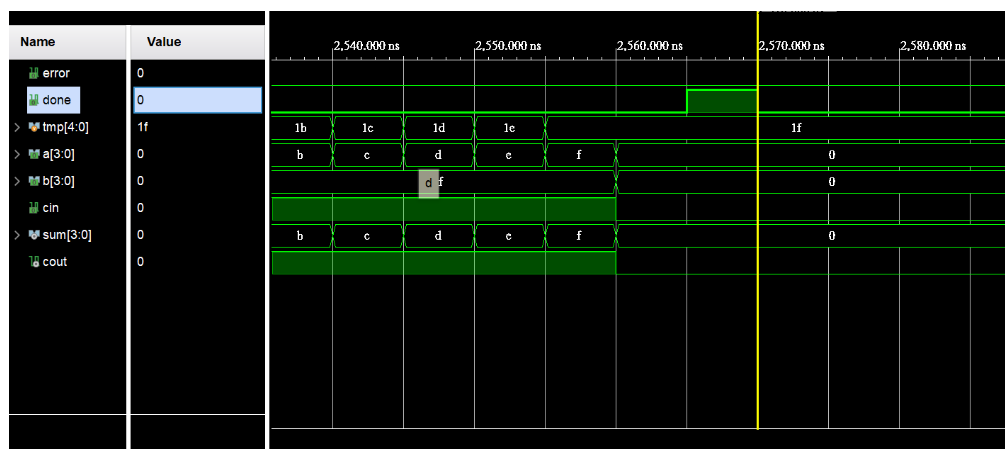
```
35        repeat(2)begin
36            repeat(2 ** 4)begin
37                repeat(2 ** 4)begin
38                tmp = a + b + cin;
39                #1
40                if(tmp!={cout, sum})
41                    error = 1'b1;
42                else
43                    error = 1'b0;
44                #4
45                 a = a + 4'b0001;
46                end
47                b = b + 4'b0001;
48            end
49            cin = cin + 1'b1;
50        end
51    #1 error = 1'b0;
52    #4 done = 1'b1;
53    #5 done = 1'b0;
54    end
55
```

確定 adder module 是否正確

Exhuasted testbench 用 3 個 repeat 來窮舉所有可能的 input 第一層累加 cin，第二層累加 b，第三層層累加 a。我自己定義一個 reg tmp 來當作測試的標準。在最裡層的迴圈中先計算 a+b+cin 並存進 tmp。過 1ns 後判斷 tmp 是否等於{cout, sum}，如果是 error 設為 1。在測試最後一個 a+b+cin 如果是錯的，因為 error 只能維持 5ns 而且在 error 測試後只有延長 4ns 所以在最後再延長 1ns 並將 error 設為 0。Done 的部分則是延長 4ns 再設為 1 經過 5ns 再設為 0。

波形圖：

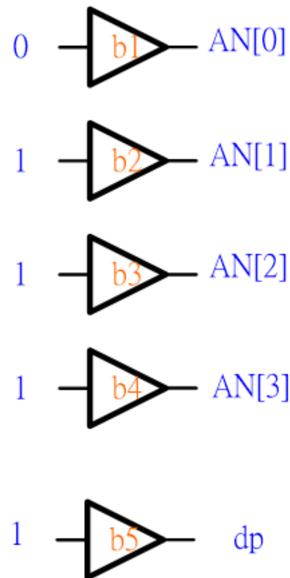
下圖為 done 確實在結束所有測資後 5ns done 拉起 5ns 再放下



下圖為為了確認 error 有確實運作把 ripple carry adder 的 4bit 改成 half adder，確認 error 確實在輸入新的值的 1ns 後如果是錯誤的將 error 拉起。

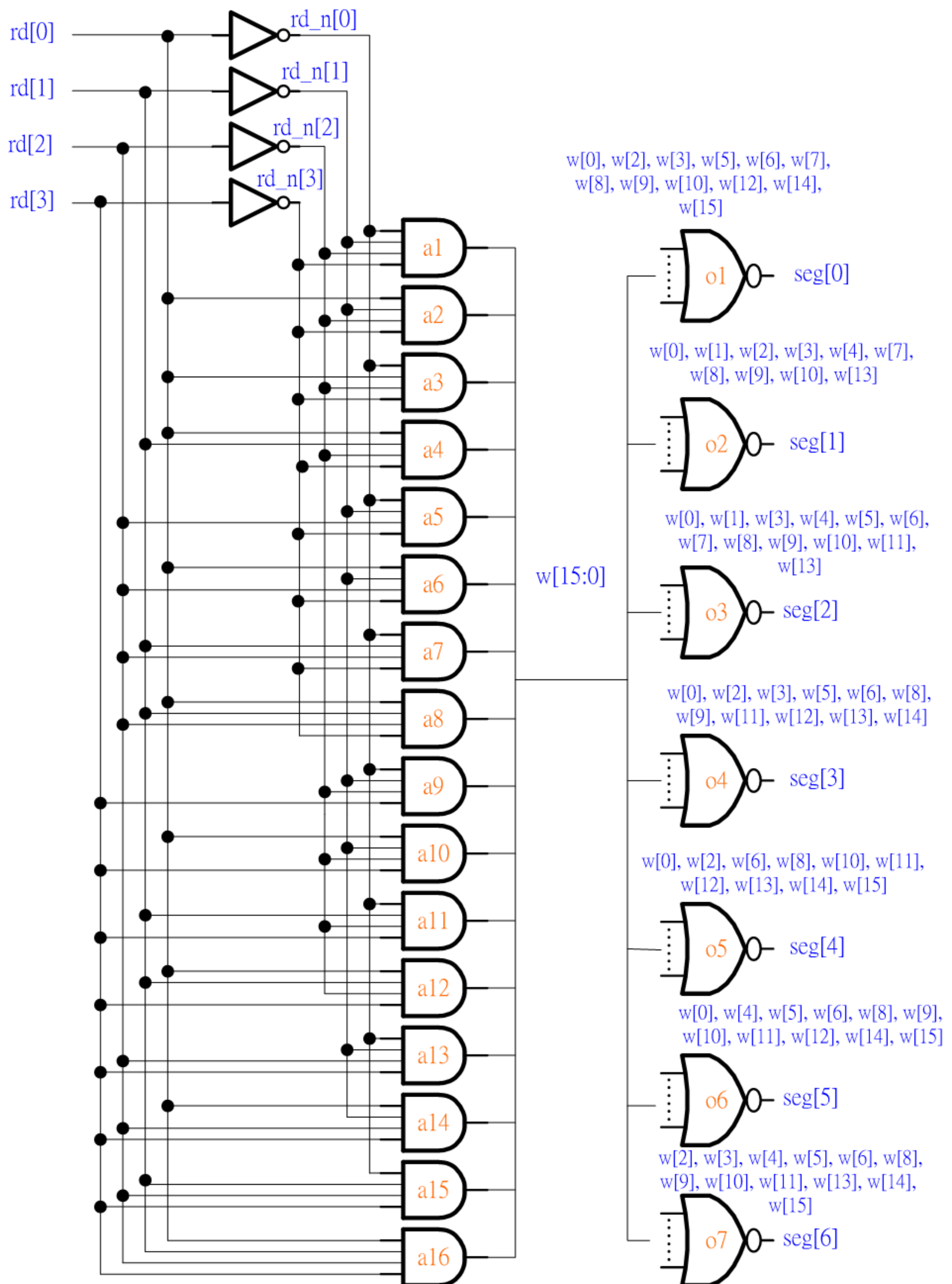
6. FPGA demonstration

- AN、dp



AN[3:0]為七段顯示器的 enable 訊號，dp 為小數點 enable 訊號，output 和 input 之間以 buffer 連接，因為只要顯示最右邊的七段顯示器，因此 AN[0] = 0 其他 = 1，不需要小數點因此 dp = 1。

- seg



用 4X16 decoder 來把 rd[3:0]的結果用 16 bits 來表示，w[0] 到 w[15]就依序代表 rd[3:0] 的數值為 0000 到 1111，接下來再

將 w[0]~w[15]根據七段顯示器的每一段分別在 rd 是什麼數字時亮起，把 w[0]~w[15]分配給 seg[0]~seg[6]，seg[0]~seg[6]分別代表七段顯示器中的 segment A~G，因為顯示器在訊號為 0 時亮起，因此 w 和 seg 之間以 NOR gate 連接。

7. Basic question: half adder 與 full adder 間的差別

Half adder: 有二個輸入端與兩個輸出端。

輸入加數 a 與被加數 b，輸出 sum 與 cout，不考慮前一位數的進位，因此只能使用在 1 bit 的加法

Full adder: 有三個輸入端與兩個輸出端。

輸入加數 a、被加數 b 與前一位的進位 cin，輸出 sum 與 cout，可以把多個 full adder 連在一起，cout 連接到下個位數的 cin，進行多位數的加法。

主要的差別就在 full adder 多了一個 cin 因此可以直接處理前一個 bit 的進位問題。

8. 心得

洪聖祥:這次 lab 要寫 CLA adder 雖然有上次的經驗，但因為此前完全沒學過這種 adder，因此花了蠻久才理解它的構造，幸好有組員幫忙。而且助教給的 CLA generator output 跟網路上的有些許不同，花了蠻多時間才理解要怎麼寫。這次 lab 我學到 hierarchical modules 的重要性，將 modules 寫進 module 裡能將 code 表達得更有條理。而且這次 lab 限制只能用 nand gate，如果將所有 code 寫進同一個 module 裡將會顯得雜亂沒條理。希望藉由這次經驗下次 lab 能更得心應手。

劉奇泓:這次的 lab 除了要寫出基本的 adder 之外，第二題的 decode and execute 要處理多個 function，並選擇要輸出的結果，最後再寫到 fpga，因此花了較多時間設計。這次的 lab 學到了蠻多的，我想出 comparator 其實可以用現成的減法器再加上一些 gate 就能完成，也熟悉了使用 fpga 中的七段顯示器，學會在 testbench 中使用簡單的迴圈就能列舉所有 input 組合，也學到要將邏輯圖完整規劃完再打 code 才能打出有條理的程式碼。希望這些經驗都能累積起來，在下次的 lab 提升效率，使我有能力處理更多難關。

9. 分工

洪聖祥:advanced question 3、4、5題，report 3、4、5題

劉奇泓: basic question logic diagram ，advanced question 1、2、fpga題，report 1、2、fpga題