Lab 2 report

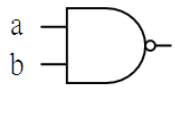
組員：110062221 李品萱
110062213 唐翊雯

I. (Gate Level) 8-bit ripple carry adder (RCA)

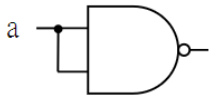
首先，這題我們使用了 Basic question 1 中以 NAND Gate 實作的 basic logic gates，以及在 Basic question 3 中以 NAND Gate 實作的 Full Adder。

- Basic question 1：

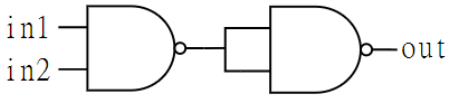
  以下為用 NAND Gate 實作的 Basic Logic Gates:



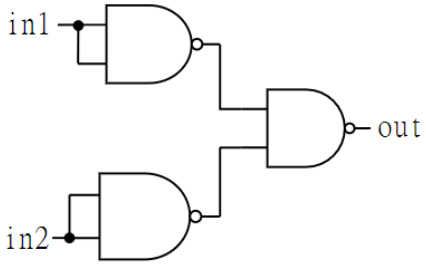
Nand



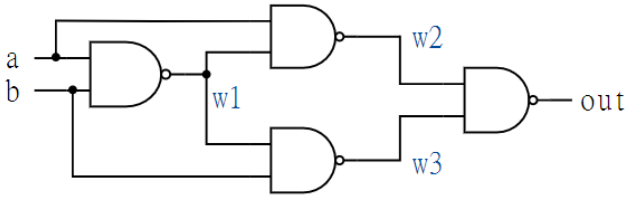
Not



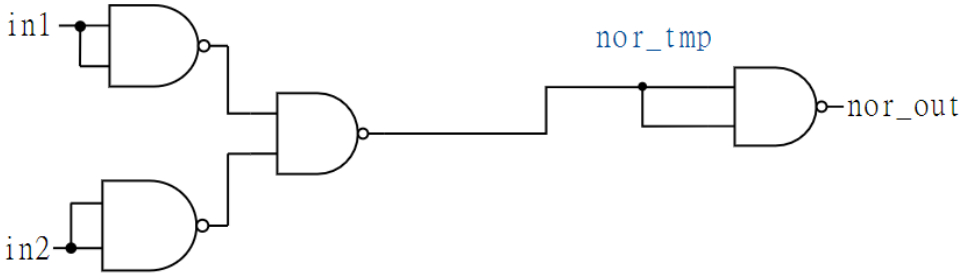
And



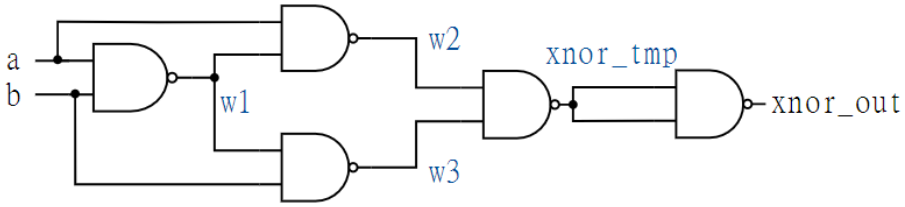
Or



Xor

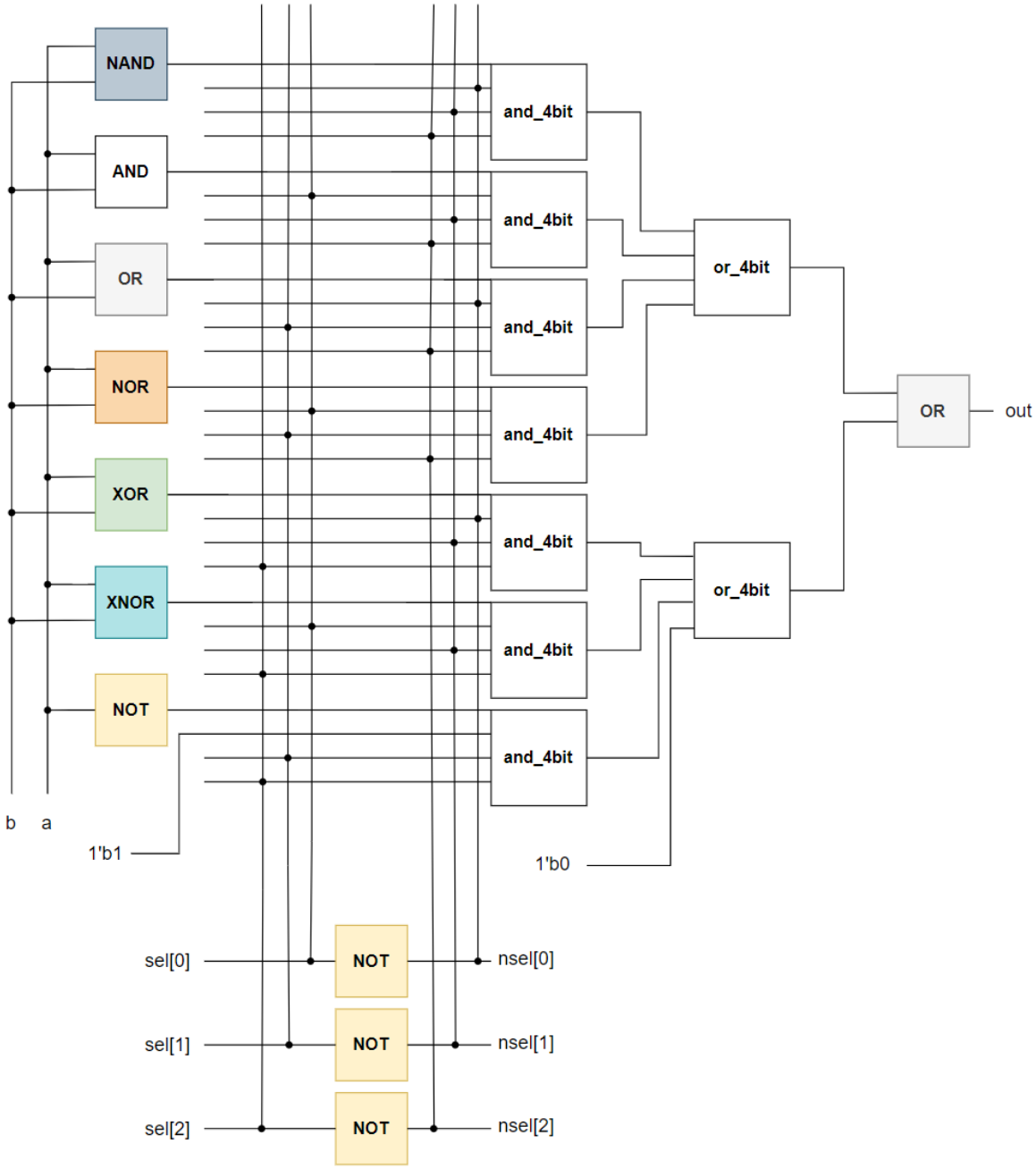


Nor

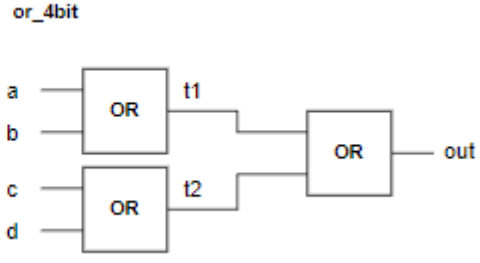


Xnor

Select 的部分，我們沿用上面實作出來的 basic logic gate，並用 mux 的想法實作，circuit 如下:

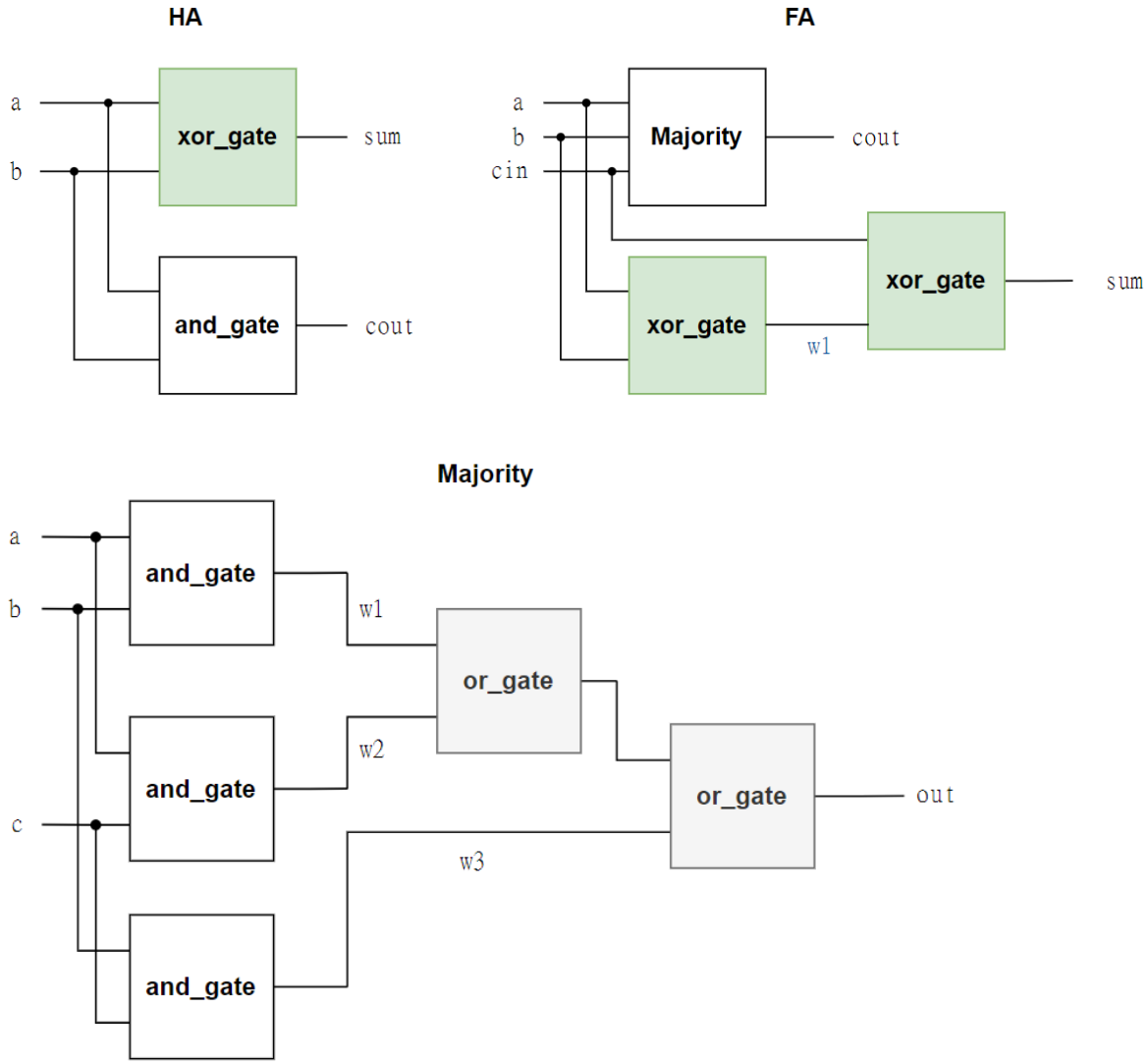


其中上面使用到 and_4bit 與 or_4bit 兩個 module，功能分別為對四個 bit 取 and 和對 4 個 bit 取 or，circuit 如下：



- Basic question 3：

  Half Adder、Full Adder 的 Circuit 分別如下：

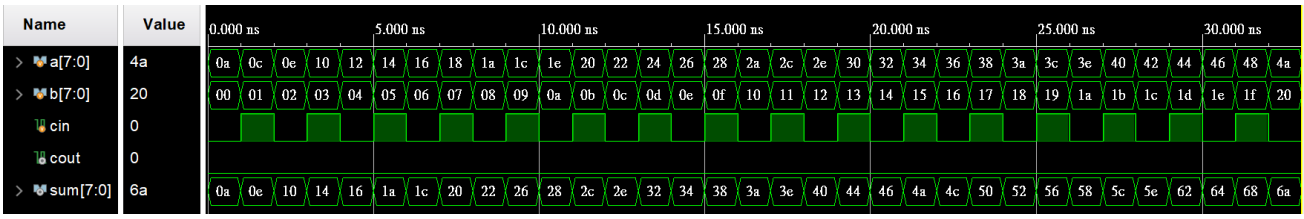


兩個 Adder 的差別在於，Half Adder 只需要處理兩個 bit 相加，而 Full Adder 要處理三個，因此兩者在處理 sum 與 cout 的時候做法不太相同。

處理 sum 的時候，如果要相加的 bit 中，1 的數量為奇數個，則 sum = 1，否則 sum = 0（因為被進位了，或是原本的 input 皆為 0）。因此在實作 Half Adder 時，我們使用 Xor 來得到 sum（若 a != b 表示有奇數個 1，否則有偶數個）。而在實作 Full Adder 時，我們使用了兩個 Xor Gate 來得到 sum。第一個 Xor Gate 若 output 1，則表示 a、b 中有奇數個 1，這時再跟 cin 做 Xor，若 cin 為 1 則總共有偶數個 1，output 為 0，反之總共有奇數個 1，output 為 1；而若第一個 Xor Gate output 0，則表示 a、b 中有偶數個 1，這時再跟 cin 做 Xor，若 cin 為 1 則總共有奇數個 1，output 為 1，反之總共有偶數個 1，output 為 0。因此這樣的寫法可以得到 sum 的值。

cout 的部分，如果 input 中 1 的數量 ≥ 2，則 cout 為 1，否則為 0。因此 Half Adder 的 cout 只需要將 a、b 做 and，即為所求。至於 Full Adder 的 cout，只要 a & b、a & cin、b & cin 任一個成立，cout 的值便為 1，我們只要將這三項的值做 or 即為 cout 的值，而這恰好是 Basic Question 2 實作的 circuit，可以直接拿來使用。

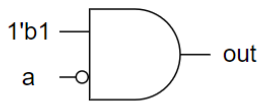
有了 Basic Question 1 與 Basic question 3 的 module，在做 8-bit ripple-carry adder 時，我們只要像題目給定的 circuit 一樣，使用 8 個 Basic question 3 的 Full Adder 並全部接起來就完成了。Testbench 的部分，我們讓 a 的值每次加 2、讓 b 與 cin 的值每次加 1，並賦予 a 和 b 不同的初始值，確認 output 是否正確。



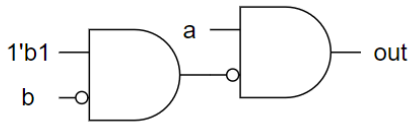
II. (Gate Level) Decode and execute

這題首先要使用給定的 universal gate，實作出其他所有的 basic logic gate，它們的 circuit 分別如下：

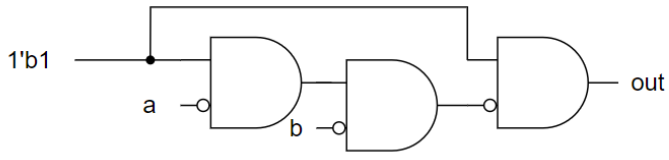
NOT



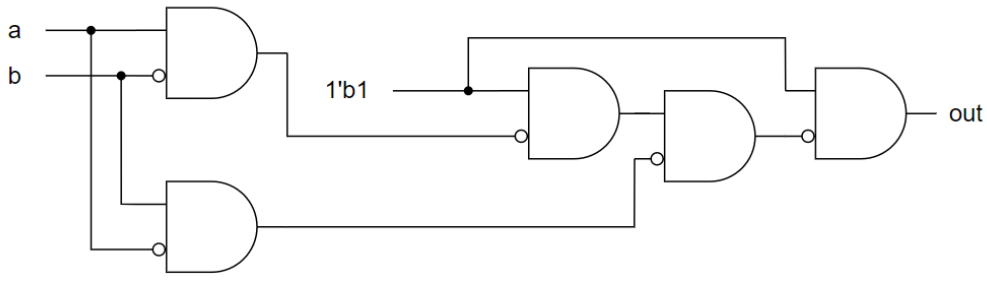
AND



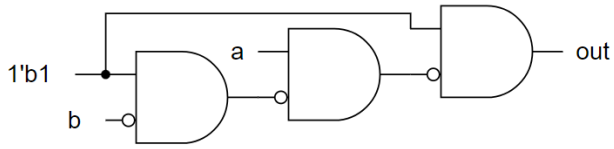
OR



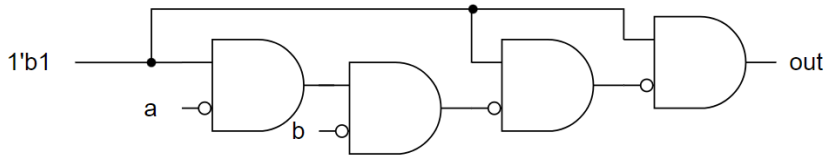
XOR



NAND

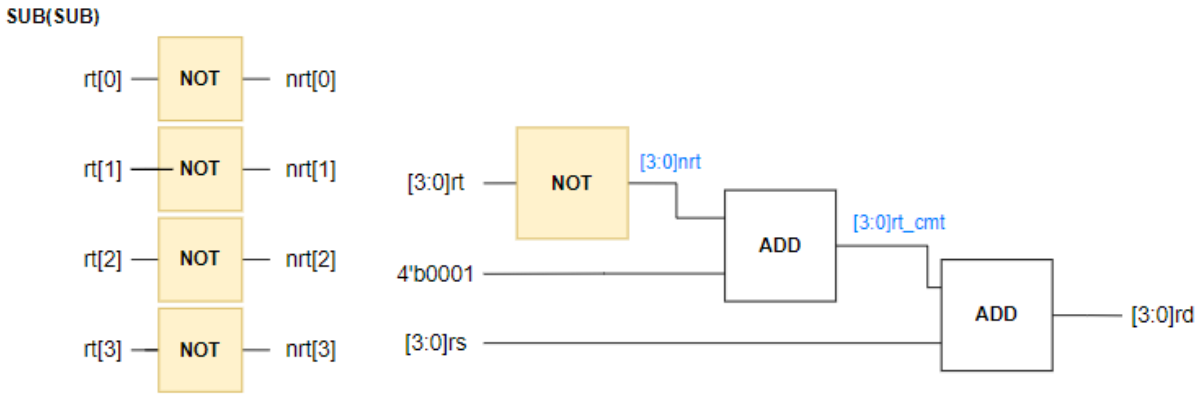


NOR

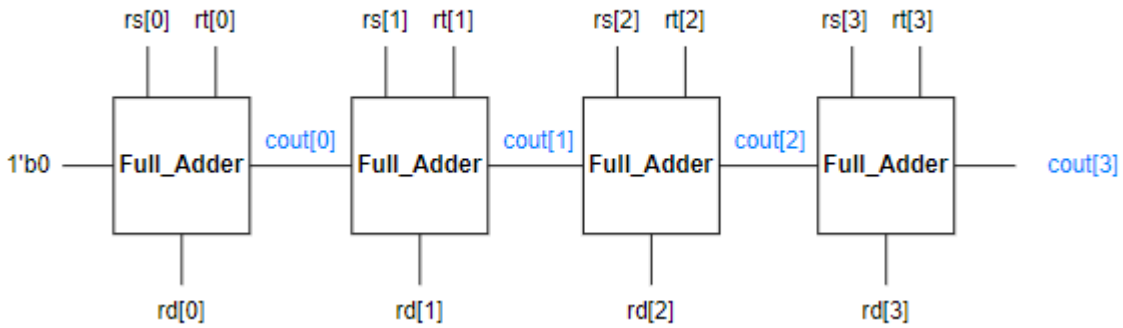


我們基於這些 basic logic gate，實作了題目所要求的各個 function，它們的 circuit 分別如下：

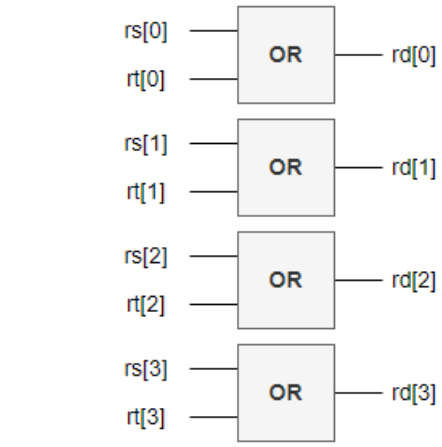
SUB(SUB)



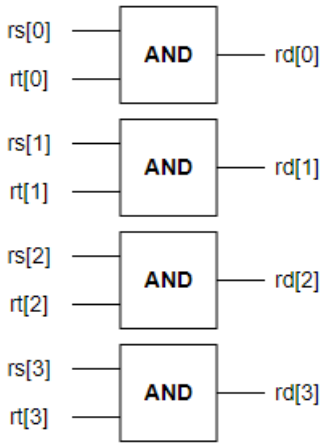
ADD(ADD)



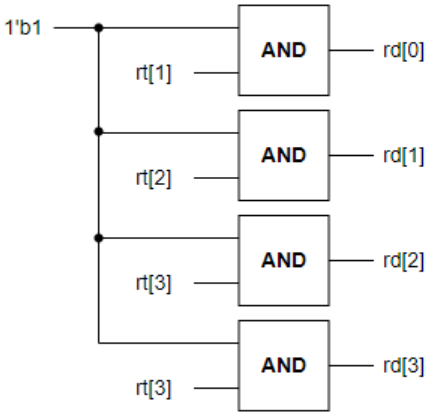
BITWISE_OR
(BITWISE OR)



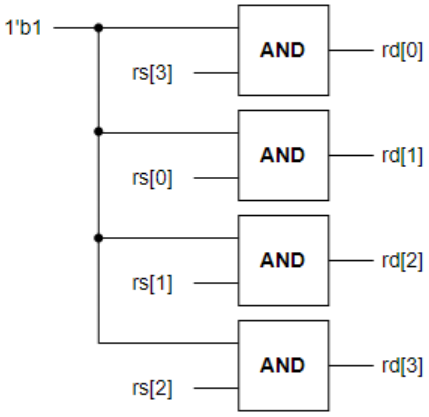
BITWISE_AND
(BITWISE AND)



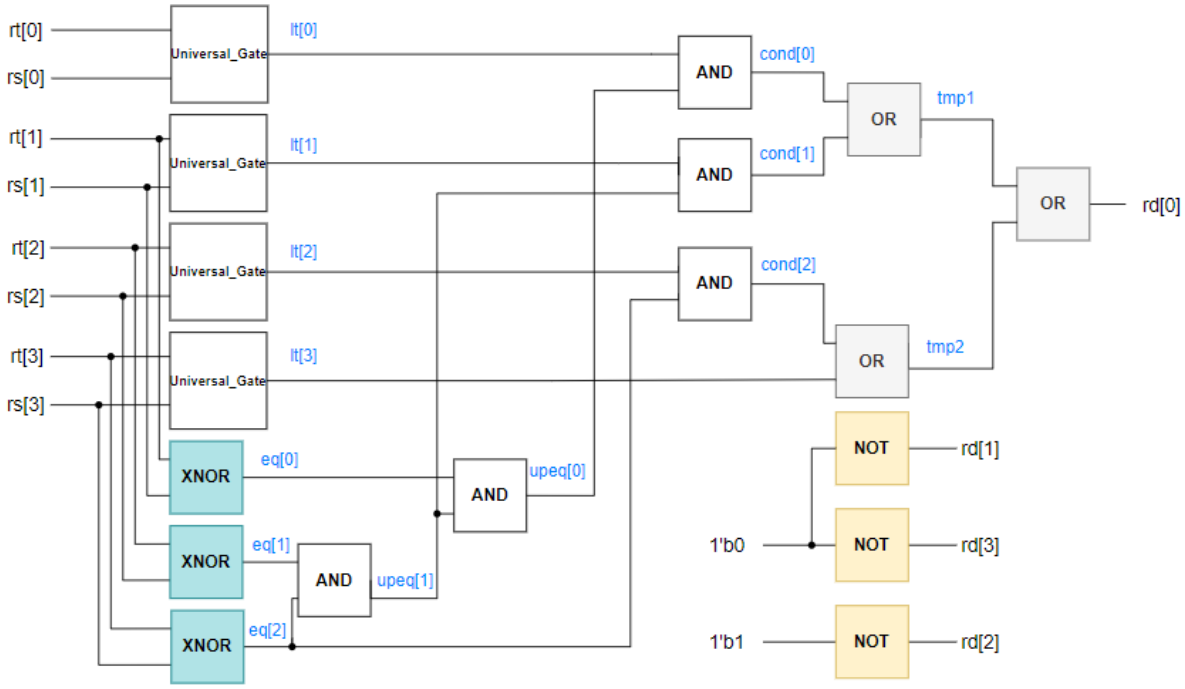
RIGHT_SHIFT
(RT ARI. RIGHT SHIFT)

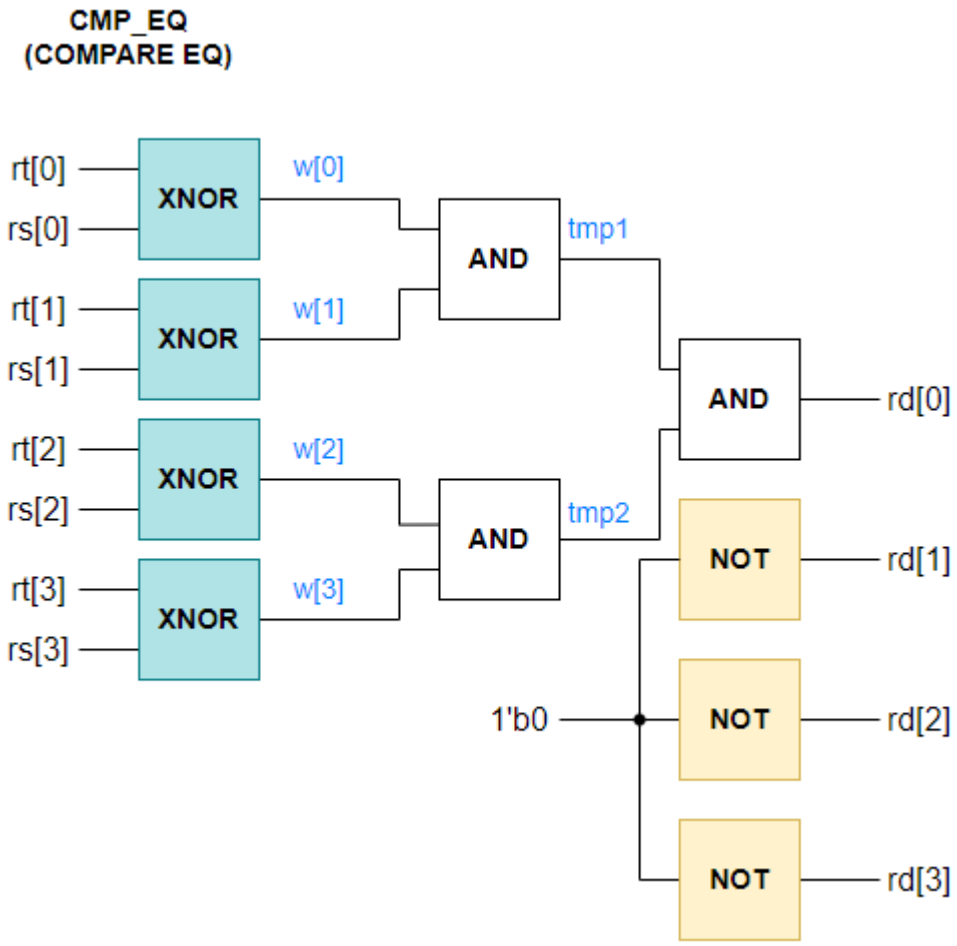


LEFT_SHIFT
(RS CIR. LEFT SHIFT)



CMP_LT
(COMPARE LT)





SUB 的實作使用到了二補數：首先，$rd = rs - rt$ 這個 function 等同於 $rd = rs + (-rt)$。因此我們先對 rt 取負號，即計算它的二補數，接著再利用 ADD 這個 module 將 rs 與 rt 相加，得到我們要的輸出。計算二補數的方法是將所有的 bit 都倒過來再加一，因此我們先使用之前實作的 NOT gate，對 rt 所有的 bit 都取 not，再將它與 4'b0001，一樣利用 ADD 這個 module 將它們相加起來。

ADD 這個 function 的實作則是使用到了 4 個 Full Adder。基本上，每一個 Full Adder 的實作的方式與 basic question 3 的實作相同，只是把所有 basic logic gates 的實作從使用 NAND 改為使用題目規定的 universal gate。如上圖把 4 個 Full Adder 接在一起，便可以實作出 $rd = rs + rt$ 這個 function。

BITWISE OR 的實作相對單純：將 rs 與 rt 的每一個 bit 分別作 OR，便是這個 function 要的輸出。BITWISE AND 的實作也很相似：將 rs 與 rt 的每一個 bit 分別作 AND 即可。

RT ARI. RIGHT SHIFT 的部分，題目已經說明 rd 的每一個 bit 分別要是 rt 哪一個 bit 的值，這邊我們的做法是對這些值與 1'b1 取 AND，輸出即為所求。RS CIR. LEFT SHIFT 也

很相似，一樣，題目已經說明 rd 的每一個 bit 分別要是 rs 哪一個 bit 的值，對這些值與 1'b1 取 AND，輸出即為所求。

COMPARE LT 的部分，我們實作的想法如下：比較兩個二進位數字的大小，首先要比較最高位的 bit，若不同則該 bit 比較出來的大小關係即為兩數的大小關係。若相同，則繼續往下一個較低位的 bit 比較，依此類推。因此若要符合 $rs < rt$，必須符合下列條件的其中一項：

- $rs[3] < rt[3]$
- $rs[3] == rt[3]$ && $rs[2] < rt[2]$
- $rs[3:2] == rt[3:2]$ && $rs[1] < rt[1]$
- $rs[3:1] == rt[3:1]$ && $rs[0] < rt[0]$

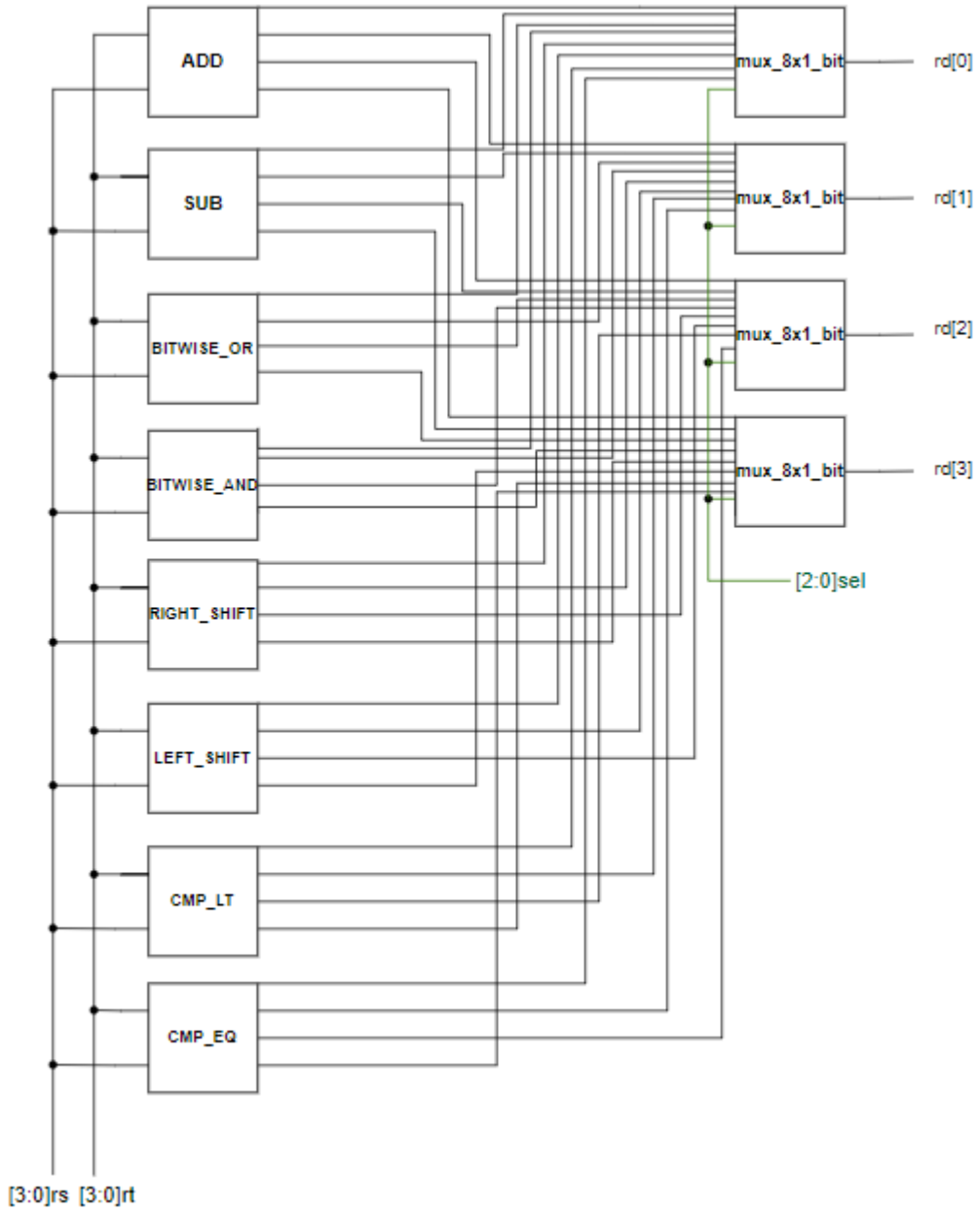
因此，我們分別用 logic gate 得到上述關係式的結果，再將所有結果取 OR，output 即為 rd[0]。 rd[3:1] 的部分，題目已經指定每個 bit 的值，這邊我們直接使用三個 NOT gate：輸入分別放題目要求的值的相反；輸出分別放 rd[0]、rd[1]、rd[2]，便能達到題目要求。

COMPARE EQ 的實作，我們的想法如下：比較兩個二進位數字的是否相等，要比較兩數的每一個 bit 是否都相同。也就是說，若 $rt == rs$，下列的值皆須為 true：

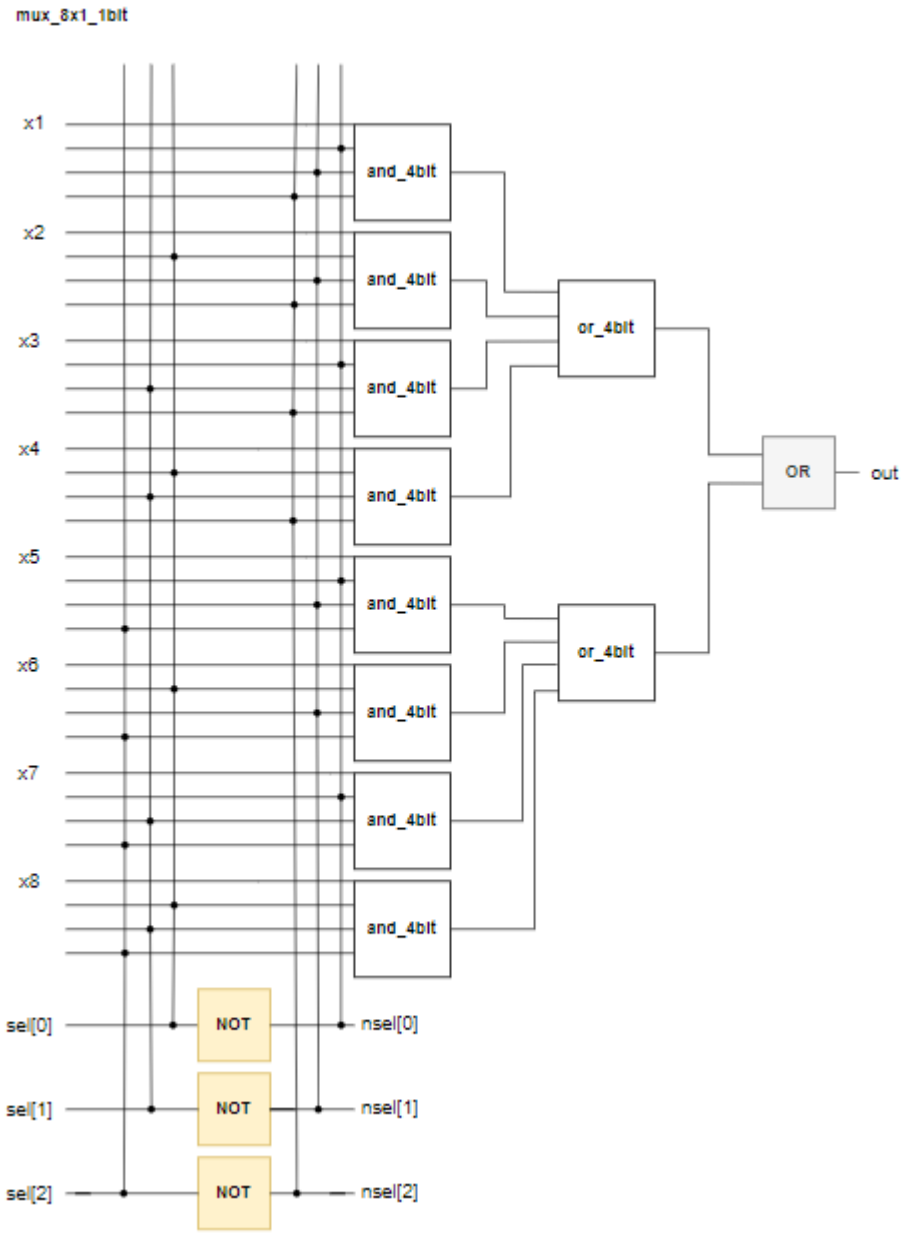
- $rs[3] == rt[3]$
- $rs[2] == rt[2]$
- $rs[1] == rt[1]$
- $rs[0] == rt[0]$

判斷相等的部分我們使用 XNOR 實作，若兩 bit 作 XNOR 的值為 true，即代表兩 bit 相等。最後把上述 4 個值取 AND，即為 rd[0] 的值。rd[3:1] 的部分與 COMPARE LT 的部分相同，使用三個 NOT gate 給 rd[3:1] 題目所要求的值。

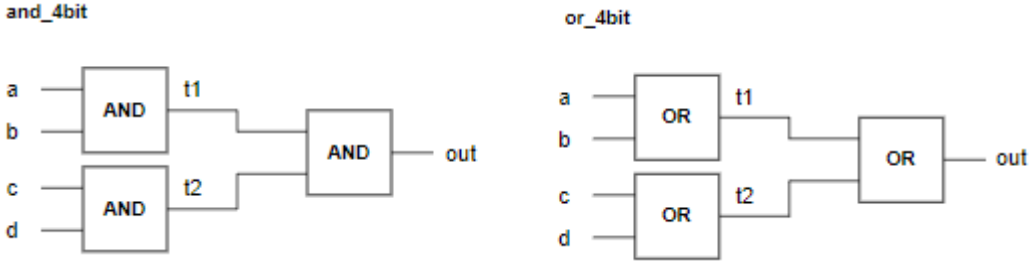
最後，根據 sel 的值，我們要給出相對應的 output。在得到各個 function 的 output 之後，我們使用了 4 個 8 to 1 的 MUX (1 bit)，分別得到 output 的每個 bit 的值，circuit 如下：



而 8 to 1 的 MUX (1 bit)，其 circuit 如下：

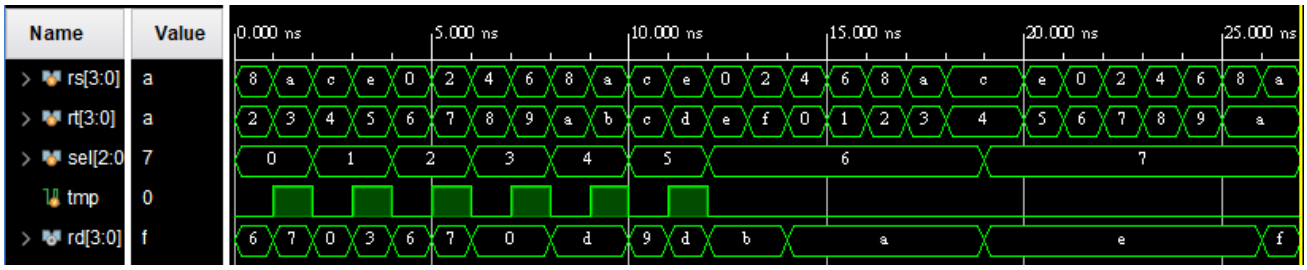


其中使用到了 4 bit 的 AND 以及 4 bit 的 OR，circuit 分別如下：

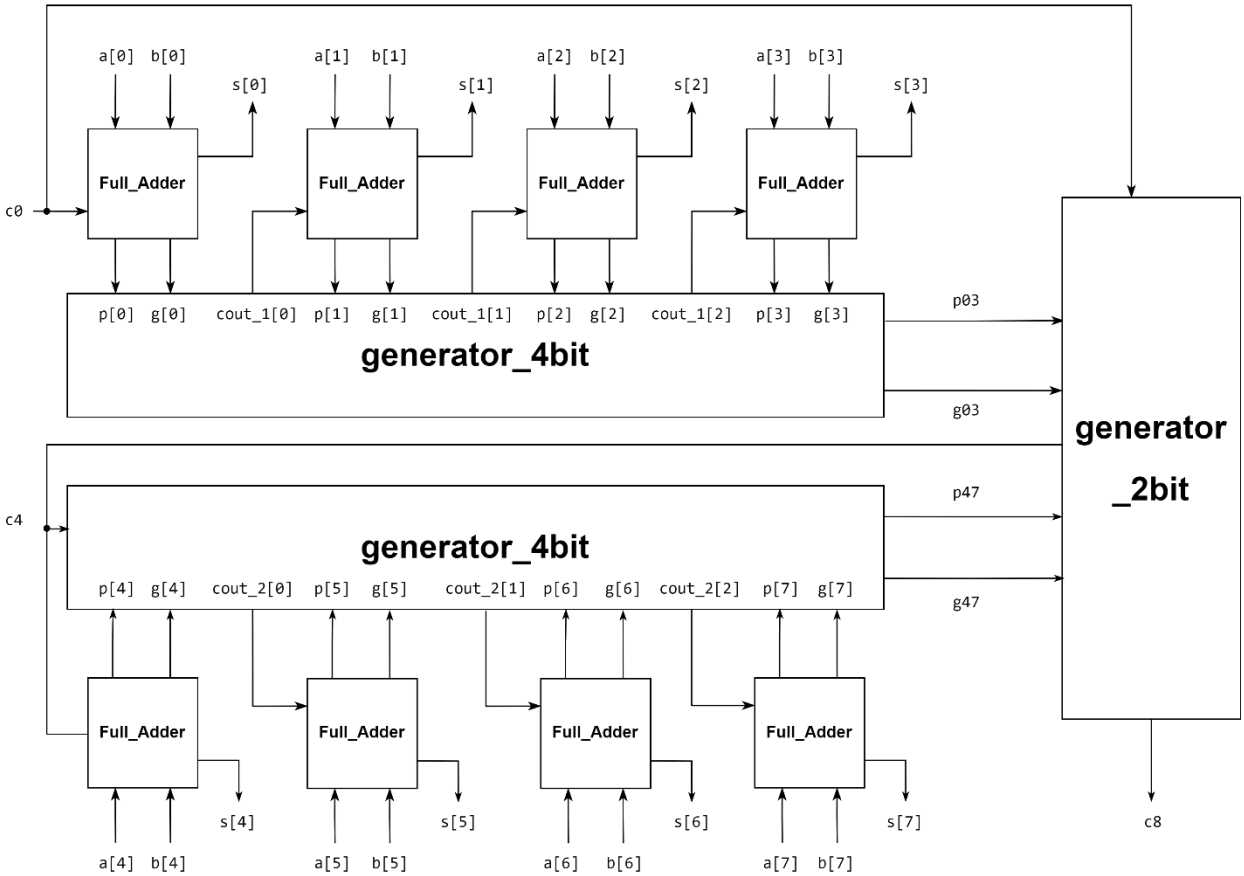


Testbench 的部分，對於前 6 個 function，我們讓 rs 的值每次加 2、讓 rt 的值每次加 1，並賦予不同的初始值下去確認每個 function 的 output 是否正確。而對 COMPARE LT 的部分，我們多跑了幾次，以確認在 $rs == rt$、$rs < rt$ 以及 $rs > rt$ 的情況下，

output 都會是正確的。COMPARE EQ 的部分，我們也特別 assign 了 rs 與 rt 的值，以確保在 $rs == rt$ 以及 $rs! = rt$ 的情況下，output 都是正確的。

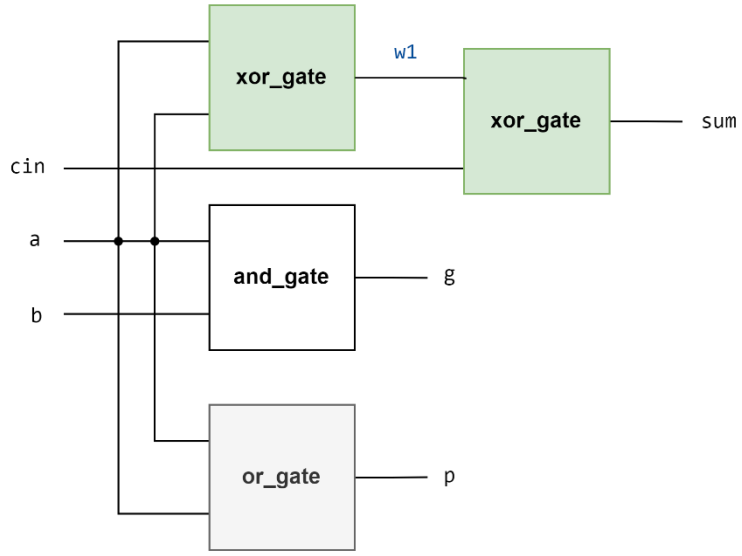


III. (Gate Level) 8-bit carry-lookahead (CLA) Adder

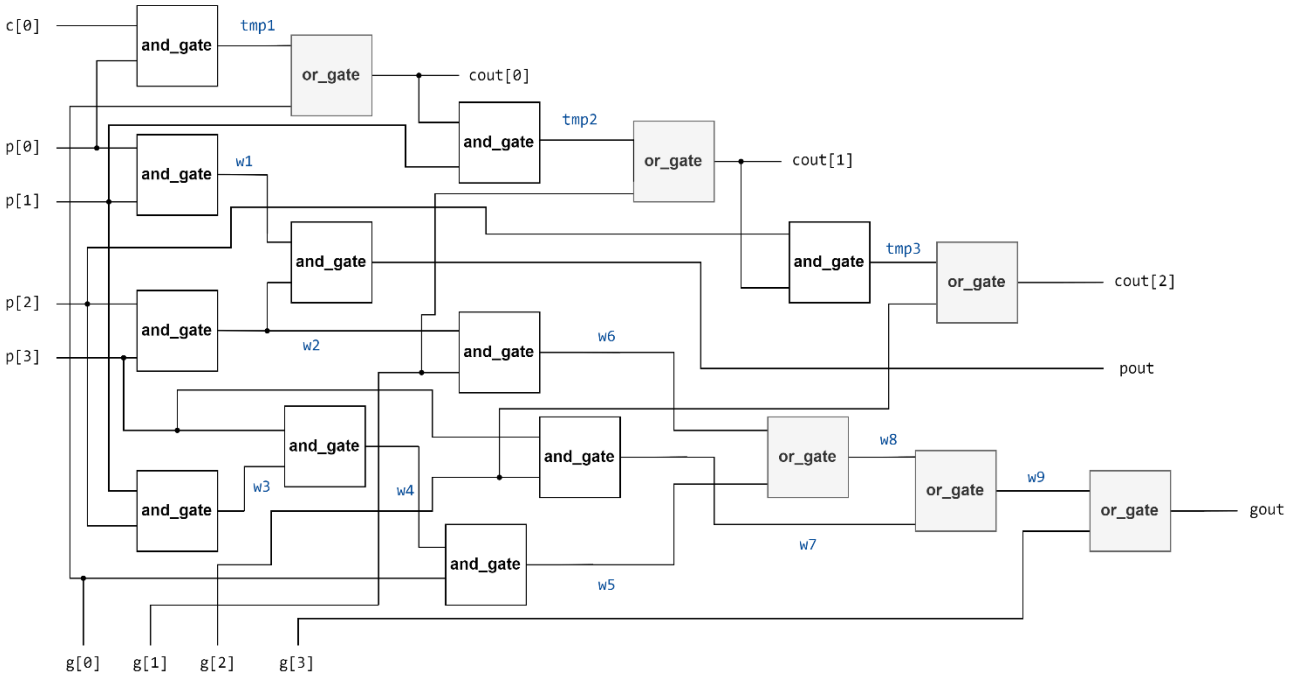


如上圖，這題我們需要八個 Full Adder、兩個 4-bit Carry-Look-Ahead Generator 與一個 2-bit Carry-Look-Ahead Generator。CLA 特別的地方在於它的 p, g 及 generator，讓它可以提前知道兩數相加時的 carry，且每個 bit 的 Full Adder 可以直接拿自己的 carry in 同時進行運算，而不用像 RCA 需要等待上一個 bit 做完之後才知道 carry out 的值，能夠減少延遲時間，也不須依賴前面的計算結果來獲得值，因此較為有效率，這也能夠從電路圖中看出來，同樣是 8bits 的 adder，RCA 需要的 and gate 及 or gate 個數相較之下比 CLA 多，因此在運算上也會較為耗時。

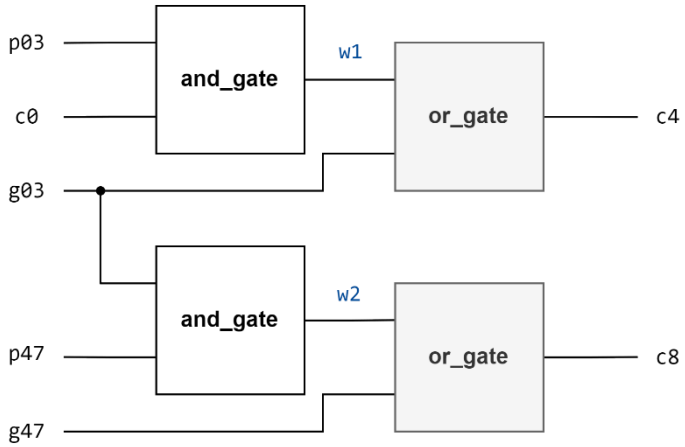
為了實作 generator，我們需要先做出 p 與 g 兩個訊號，p 代表的是 propagate，負責將兩個 bit 做 or，也就是說如果兩個 bit 中至少有一個 bit 為 1 就會是 1，此時當 carry in 是 1 時就會將此 carry 往下傳遞；g 代表的是 generate，負責將兩個 bit 做 and，意即當兩個 bit 都是 1 時必會出現 carry out。而在設計的部分我們將 input 丟進 Full Adder 中讓它 output, p, g, sum 再將 p, g 傳入 generator 中，這部分的 Full Adder 如下圖。



在計算進位的部分，它的 equation 為：$(((p[0] \cdot c0 + g[0]) \cdot p[1] + g[1]) \cdot p2 + g[2]) \cdot p[3] + g[3]$，經過計算可得到原式 $= p[0] \cdot p[1] \cdot p[2] \cdot p[3] \cdot c0 + p[1] \cdot p[2] \cdot p[3] \cdot g[0] + p[2] \cdot p[3] \cdot g[1] + p[3] \cdot g[2] + g[3]$，其中 $p[0] \cdot p[1] \cdot p[2] \cdot p[3]$ 即為 $p[0,3]$，而 $p[1] \cdot p[2] \cdot p[3] \cdot g[0] + p[2] \cdot p[3] \cdot g[1] + p[3] \cdot g[2] + g[3]$ 即為 $g[0,3]$，$p[4,7]$ 及 $g[4,7]$ 同理，於是我們可以知道 c4 即為 $p[0,3] \cdot c0 + g[0,3]$，而 c8 即為 $p[4,7] \cdot c4 + g[4,7]$。因此在 code 的部分，我們傳入 4bit 的 p 及 g，依照上面 equation 的框架做出 $p[i,i+3]$、$g[i,i+3]$ 及 cout，4bit generator 的 circuit diagram 如下。



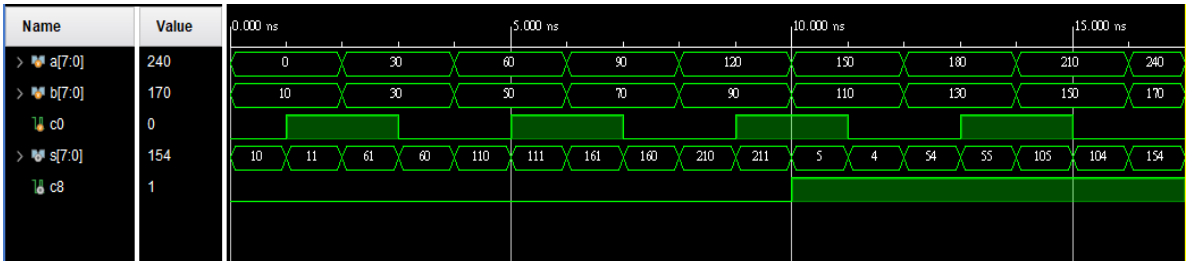
而後，4bit generator 的 pout 及 gout 會再傳入 2bit generator 做上述 c4 及 c8 的 equation 再 output 出 c4 及 c8，如下圖。



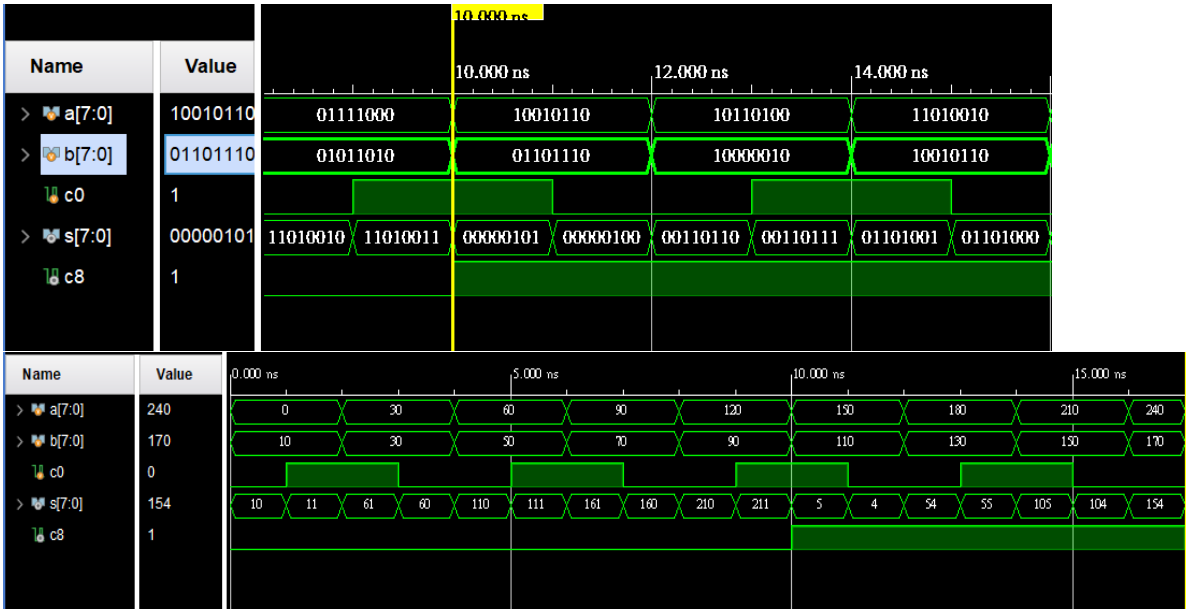
在 code 的部分，考慮到 c4 與其它 cout 接出來的地方不同，為了寫 code 上的方便，我們單獨處理 c4，並將 cout 分為 cout_1 及 cout_2 兩部分，各為 4bits，下圖為我們在 Carry_Look_Ahead_Adder_8bit 這個 module 中呼叫 Full Adder 及 generator 的部分。

```
14   // deal with FA
15   Full_Adder fa1(a[0], b[0], c0, p[0], g[0], s[0]);
16   Full_Adder fa2[3:1](a[3:1], b[3:1], cout_1[2:0], p[3:1], g[3:1], s[3:1]);
17   Full_Adder fa3(a[4], b[4], c4, p[4], g[4], s[4]);
18   Full_Adder fa4[7:5](a[7:5], b[7:5], cout_2[2:0], p[7:5], g[7:5], s[7:5]);
19
20   // deal with gen
21   generator_4bit genA(p[3:0], g[3:0], c0, p03, g03, cout_1);
22   generator_4bit genC(p[7:4], g[7:4], c4, p47, g47, cout_2);
23   generator_2bit genB(p03, g03, p47, g47, c0, c4, c8);
```

在 test 的部份我們用 clock 延遲的方式讓 c0 會在 a 與 b 沒有變動時出現 0 跟 1 兩種情況，以便確認 carry 這部分沒有問題，如下圖。



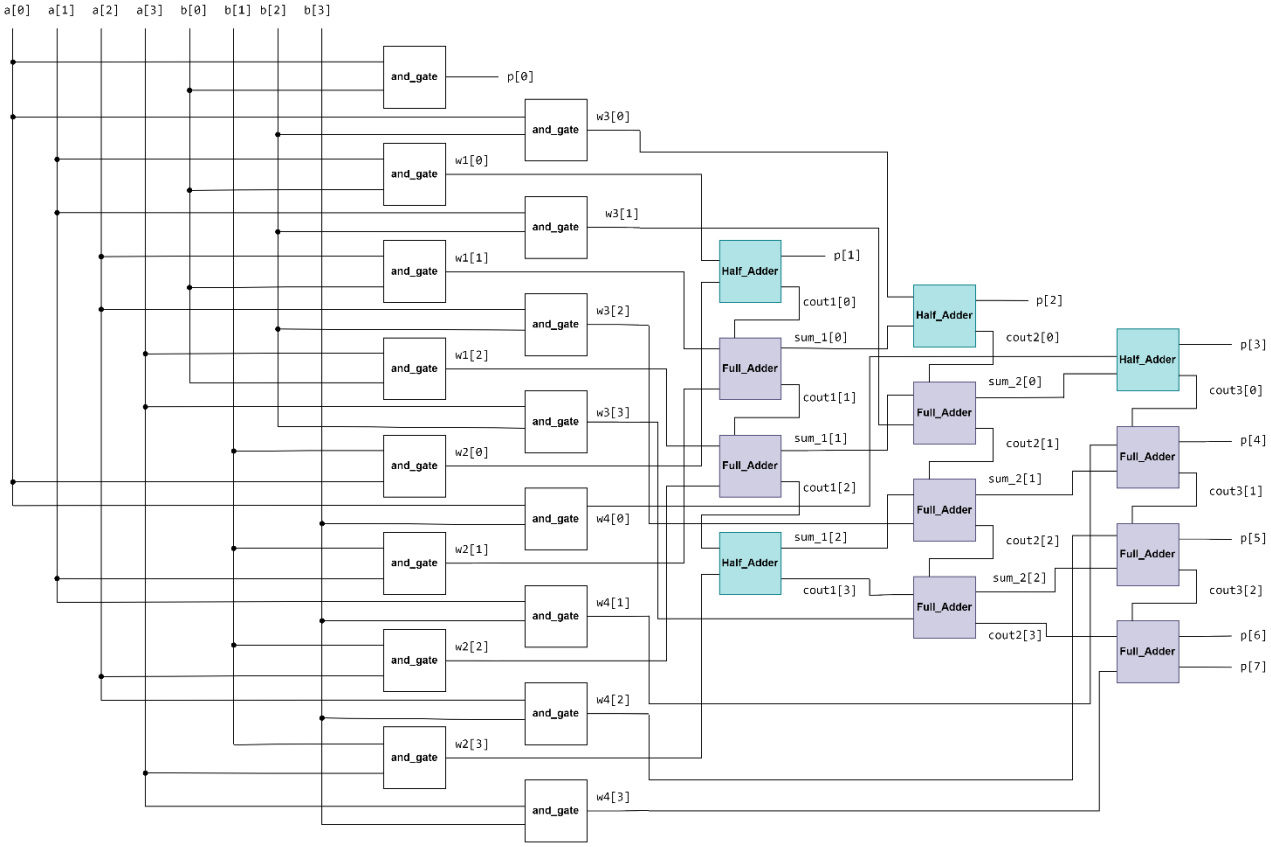
且為了觀察 c8 的狀況，我們這邊選擇讓 a 每次加 30，b 每次加 20，而當 s 產生 carry out，也就是 c8 為 1 時，可以發現 s 的數值無法完整呈現 a、b 相加該有的結果，這是因為 s 只有 8bits，因此最大值以十進位表示為 255，超過則會發生溢位的狀況，此時若以二進位觀察可以發現 s 是 a+b 的後面 8 個 bits，如下圖。若將 c8 與 s cascade 起來就會得到正確的結果。



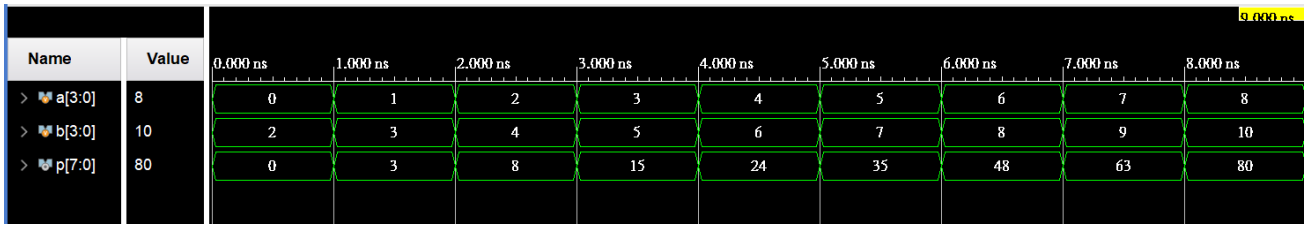
IV. (Gate Level) 4-bit multiplier

這題我們需要設計一個 4bit 的 multiplier，其中我們會用到前面時做過的 Full Adder 及 Half Adder，它的原理與直式乘法類似，透過計算每個 partial product 再將他們送至 adder 中做運算得到最後的值。如下圖，首先我們將 a 與 b 每個 bit 互相 and，其中，$a[0] \cdot b[0]$ 即為 p[0]，其他 bit 做完 and 的結果我們以 wire 接出來，由於我們這邊的作法是將 a 作為被乘數，b 作為乘數，因此我們將 $a[i] \cdot b[j]$ 對應到 w_{j+1}，接下來先將 w1 接出來的訊號送入圖中最左邊那一個 column 的 adder 中，這邊我們採用兩個 Half Adder 及兩個 Full Adder，因為該 column 最上方的 Half Adder 不會有 carry in，我們用設計上比較簡單的 Half Adder 進行運算，這個 Half Adder 的 output 即為 p[1]。而接下來兩個 bits 都會有 carry in，所以都採用 Full Adder；每個 Full Adder 會將 sum 送進右方

的 Adder 中，並將 carry out 往下傳，類似 Ripple Carry Adder；值得注意的是該 column 最下方的也是使用 Half Adder，其原因並不是沒有 carry in，而是將 carry in 視為 Half Adder input 的其中 1 bit，將此 carry in 與 partial product（也就是 w2[3]）相加。第二個 column 與前述類似，最上方的 Half Adder 即會算出 p[2]，而最下方我們使用的是 Full Adder，因為這邊要接左方的 cout1[3]、w3[2]及前一個 Full Adder 的 carry out（即 cout[2]）做運算。第三個 column 與左方相同，這邊的 sum 即為 p 的第 3 個 bit 到第 7 個 bit。



這邊 testbench 我們就簡單的讓 a 跟 b 每次加 1，觀察 p 的值是否符合 a 與 b 相乘的結果。



v. An exhaustive testbench design

這一題我們要 design 一個可以正確找出錯誤的 testbench，所以我們設計了一個 faulty 的 RCA 與題目要求的 testbench。

首先，在 testbench 的部分，由於題目要求要枚舉所有可能，參考 template 上的 example 後，我們用三層 repeat 來實作，首先固定 cin 為 0、b 為 0，讓 a 先從 0 跑到 15，依序對 b 為 1 到 15 的狀況讓 a 從 0 跑到 15，最後在對 cin 為 1 時做相同的事，這樣就會覆蓋到 a[3:0] + b[3:0]的所有可能情況。在最內層的 repeat 中，我們先延遲一個 clock，在將 error 設為 0，這樣做的用意是對每個要檢查的結果做 error 的初始化，且符合題目要求的，若檢查結果正確則在 input 進來 1ns 後將 error 設為 0。接下來我們用 if 對 RCA 的 output 進行檢查，這邊的想法是既然我們要確認 output 的正確性，那我們就先用 input 做出正確的結果，再檢查 output 與結果是否相等，一開始我們是想到 bitwise operation 可以做出正確結果，但要對每個 bit 做太麻煩了，後來想到其實直接如下圖將 a、b、cin 加起來就好，此時我們再將 cout 與 sum 做串接，並用嚴格不等於確保他們是逐位相等且只會回傳 0 與 1，在 if 條件成立即為偵測到錯誤，此時因為前面已經延遲過一個 clock，所以會馬上 raise error；if 條件判斷完後，我們讓他維持 4 個 clock 再改變 a 的值，因此加上下一次進 repeat 後即會有維持 5 個 clock 的 error 及 input 訊號。

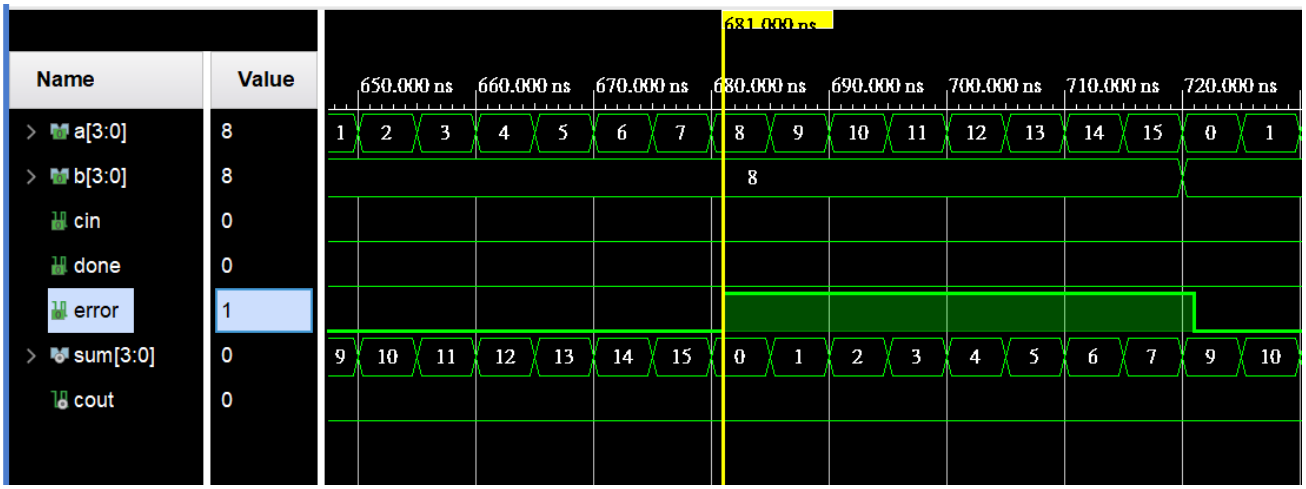
```
32  initial begin
33      repeat(2**1)begin
34          repeat(2**4) begin
35              repeat(2**4) begin
36                  #1;
37                  error = 1'b0;
38                  if({cout, sum[3:0]}!==a[3:0]+b[3:0]+cin) begin
39                      error = 1'b1;
40                  end
41                  #4;
42                  a = a+4'b0001;
43              end
44              b = b+4'b0001;
45          end
46          cin = cin +1'b1;
47      end
48      done = 1'b1;
49      #1 error = 1'b0;
50      #5 done = 1'b0;
51  end
```

為了檢察我們的 testbench 設計是否正確，我們先用正確的 RCA 確定它不會 raise error 也有正確的呈現 done 訊號，再用一個會產生錯誤結果的 RCA 進一步檢查，我們假設了兩種錯誤情況。

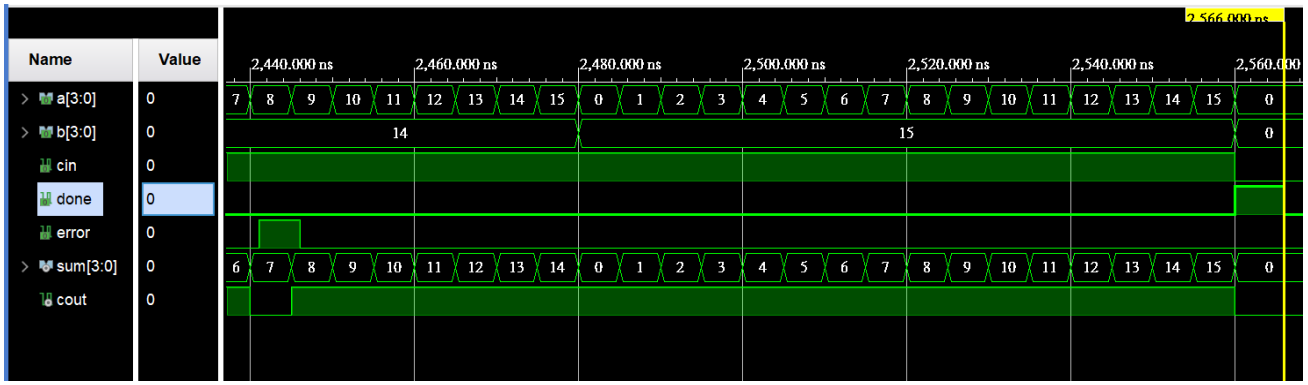
第一種是 cout 出錯，如下圖，我們讓這個 RCA 只在 c3 及 c4 都為 1 時才會產生 cout。

```
3   module Ripple_Carry_Adder(a, b, cin, cout, sum);
4   input [3:0] a, b;
5   input cin;
6   output cout;
7   output [3:0] sum;
8   wire c1, c2, c3;
9   wire c4;
10
11  Full_Adder fa1(a[0], b[0], cin, c1, sum[0]);
12  Full_Adder fa2(a[1], b[1], c1, c2, sum[1]);
13  Full_Adder fa3(a[2], b[2], c2, c3, sum[2]);
14  Full_Adder fa4(a[3], b[3], c3, c4, sum[3]);
15  and_gate and1(cout, c3, c4);
```

這個情況的 simulation 結果如下圖，在 b = 8，a = 8 到 a = 15 時 cout 沒有呈現 1，所以在偵測到錯誤 1ns 後，error 就會為 1，且由於他在這段區間內都是錯的，因此 error 會一直呈現 high 的狀態，一直到偵測到正確值 1ns 後才會變為 0。



這邊我們也能看到在 a = 8，b = 14 時，cout 輸出 0，因此會 raise 5 ns 的 error。在所有可能都跑過後，done 會在偵測到最後一個 input 5ns 後變為 1，並如同 spec 圖中在 5ns 後恢復成 0。

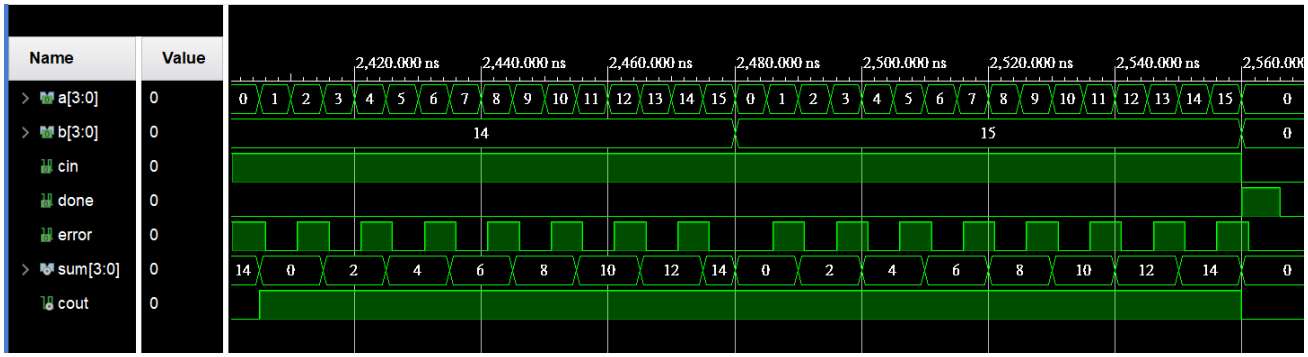


第二種情況是 sum 出現錯誤，如下圖，我們將原本 sum[0]的位子換成 c4，並將它與 0

做 and，造成它在應該輸出 1 時反而輸出 0。

```
3   module Ripple_Carry_Adder(a, b, cin, cout, sum);
4   input [3:0] a, b;
5   input cin;
6   output cout;
7   output [3:0] sum;
8   wire c1, c2, c3;
9   wire c4;
10
11  Full_Adder fa1(a[0], b[0], cin, c1, c4);
12  Full_Adder fa2(a[1], b[1], c1, c2, sum[1]);
13  Full_Adder fa3(a[2], b[2], c2, c3, sum[2]);
14  Full_Adder fa4(a[3], b[3], c3, cout, sum[3]);
15  and_gate and2(sum[0], 1'b0, c4);
```

這個情況的 simulation 結果如下圖。由於我們在枚舉時是每次加 1，而 raise error 與否決定在 sum[0]，因此 error 的狀況多數時候會交替出現，這個 RCA 的設計也是為了可以更清楚的看到在 correctness 部分的波型呈現。

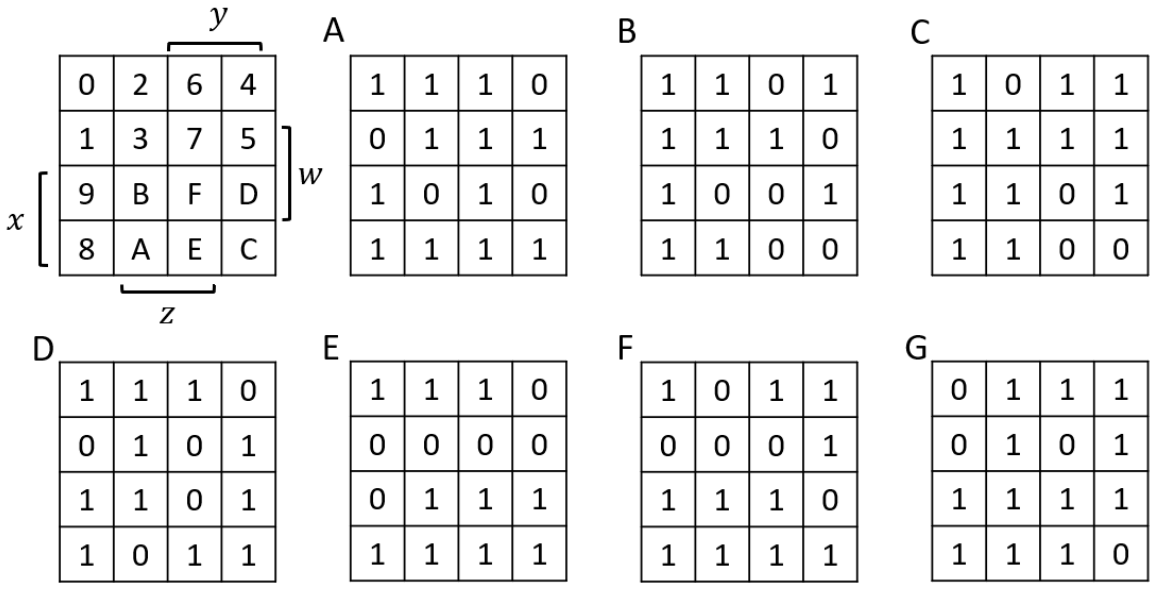


VI. FPGA: (Gate Level) Decode and execute

這題是要沿用 Advance question 2 的 module，並將結果顯示在 FPGA 板上，我們使用了 gate-level circuit 來完成這題。由於要用 7-segment display 來顯示我們的 output，首先我們要對每個數字及各個 segment 做對應。對於 0 ~ F 的每一個數字，它們對二進位以及 7-segment 的對應分別如下：

Hex	Binary	7-segment
0	0000	ABCDEF
1	0001	BC
2	0010	ABDEG
3	0011	ABCDG
4	0100	BCFG
5	0101	ACDFG
6	0110	ACDEFG
7	0111	ABC
8	1000	ABCDEFG
9	1001	ABCDFG
A	1010	ABCEFG
B	1011	CDEFG
C	1100	ADEF
D	1101	BCDEG
E	1110	ADEFG
F	1111	AEFG

我們對 7-segment 的每一個線段，去紀錄它們分別在 output 為哪些值時會亮，並使用 k-map 去寫出 function。下列的 k-map，1 表示要亮、0 表示不亮。x、y、z、w 分別代表 rd[3]、rd[2]、rd[1]、rd[0] 的值，A~G 則分別代表每一個 segment：



將每個 k-map 化簡之後，可以得到如下 sum of product 的算式：

A: $xw' + yz + x'z + x'y'w' + x'yw + xy'z'$

B: $y'z' + x'y' + x'z'w' + y'w' + xz'w + x'zw$

C: $y'z' + x'w + xy' + x'y + z'w$

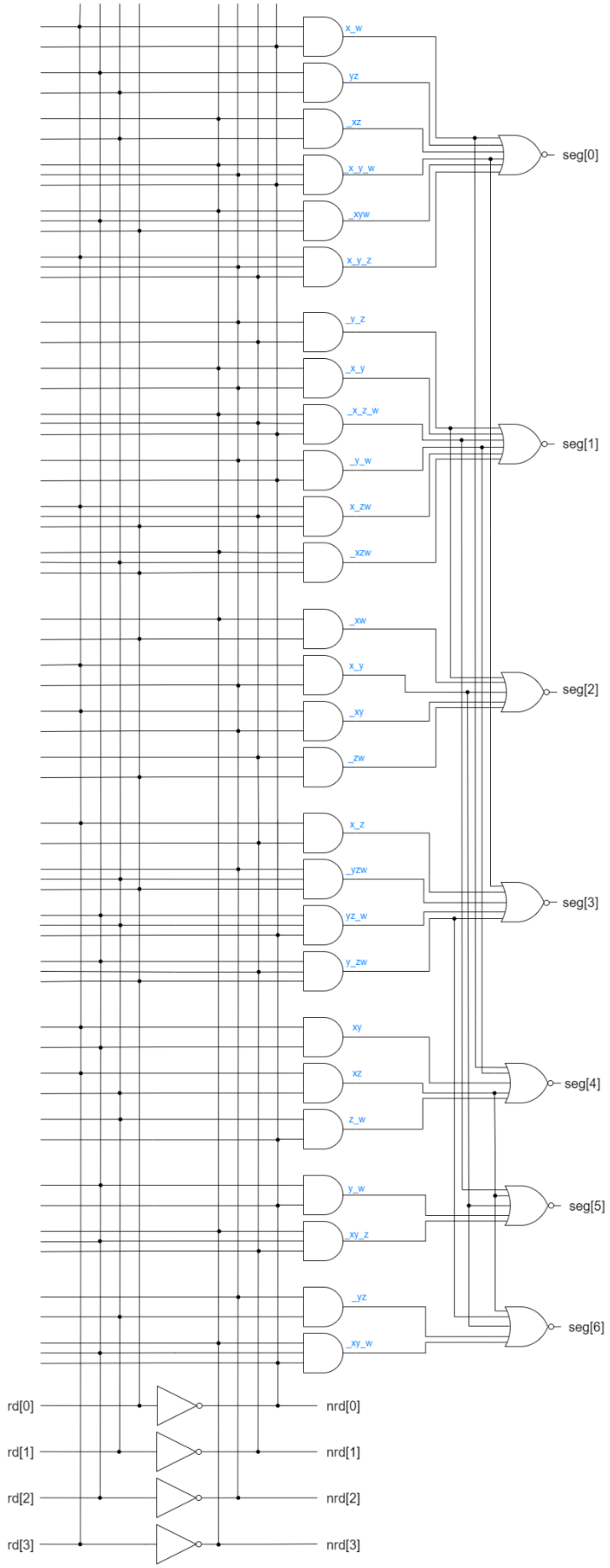
D: $xz' + x'y'w' + y'zw + yzw' + yz'w$

E: $xw' + xy + xz + zw' + y'w'$

F: $xy' + xz + yw' + x'yz' + x'z'w'$

G: $xy' + xz + y'z + x'yw' + yz'w$

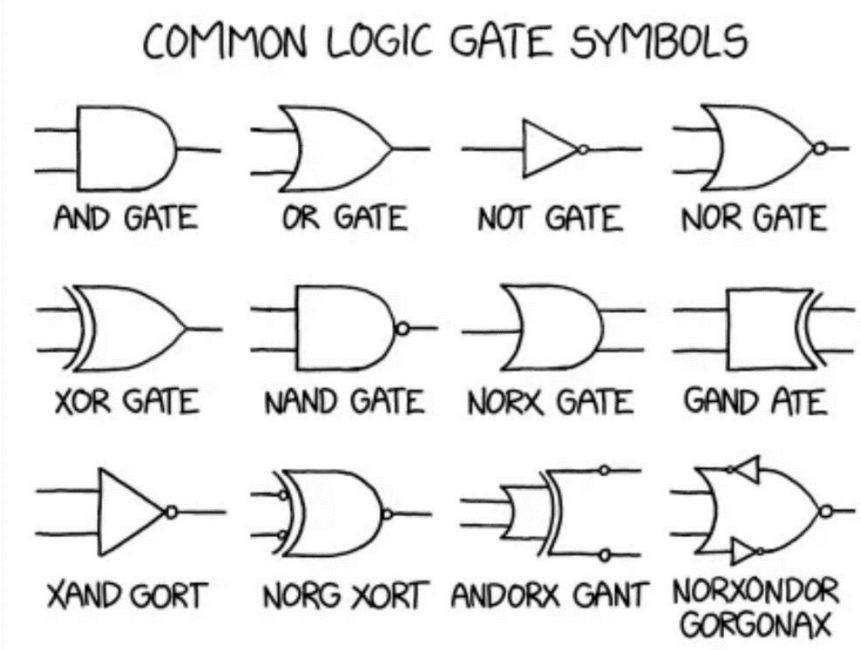
有了這些式子後，我們便可以用 logic gate 去實現它們，進而從 output 得知每個 segment 需要亮或不亮。由於前面使用 k-map 的時候，定義 1 表示要亮、0 表示不亮，因此每個 segment 最後的 output 要再接一個 NOT 才是正確的值，整體的 circuit 如下。


x_w
yz
_xz
x_y_w
_xyw
x_y_z
_y_z
_x_y
_x_z_w
_y_w
x_zw
_xzw
_xw
x_y
_xy
_zw
x_z
_yzw
yz_w
y_zw
xy
xz
z_w
y_w
_xy_z
_yz
_xy_w
seg[0]
seg[1]
seg[2]
seg[3]
seg[4]
seg[5]
seg[6]
rd[0]
rd[1]
rd[2]
rd[3]
nrd[0]
nrd[1]
nrd[2]
nrd[3]

VII. Summary

這次的 Lab 實作量比較大，也比上一次的 Lab 需要思考，在這次的 Lab 中，我們對不同的 adder 有了更多的了解，並能夠利用基本的 adder 去接出可以執行較複雜運算的 RCA、CLA 及 Multiplier，也了解到 FA 及 HA 的不同與應用；在 Decode and execute 中，我們運用 universal logic gate 組合出需要的 logic gate，學習如何用 logic gate 設計出需要的運算方式，除此之外也用到了 Mux 去做 select；在 FPGA 的部分也比上一次較為複雜，我們運用了在邏輯設計學過的 k-map，將其實際應用在 7-segment 上；在設計 testbench 的部分，我們需要設計出正確的時間延遲，並測試自己的 testbench 是否能成功判斷 Adder 設計的對或錯，對 testbench 的操作也有更進一步的認識。

比起上次基本上只要做接線及組合 logic gate 的動做就能得出結果，這次我們需要自己依照題目要求去設計，logic gate 的組合也更為複雜，所以多花了不少時間，但跟 logic gate 也有更緊密的接觸，如下圖。



VIII. Contributions

- **Code:**

  (Gate Level) 8-bit ripple carry adder (RCA) by 李品萱

  (Gate Level) Decode and execute by 李品萱

  (Gate Level) 8-bit carry-lookahead (CLA) Adder by 唐翊雯

  (Gate Level) 4-bit multiplier by 唐翊雯

  An exhaustive testbench design by 唐翊雯

  FPGA: (Gate Level) Decode and execute by 李品萱

- **Report:**
  兩人先描述各自在 code 部分負責的題目及畫電路圖，再由唐翊雯寫 summary。