LAB2

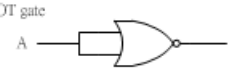
第七組

黎佑廷 105030009

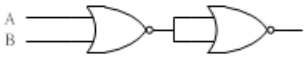
郭家瑋 105030015

BASIC NOR UNIVERSAL GATE

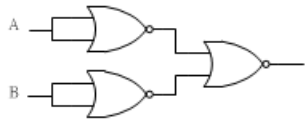
NOT gate



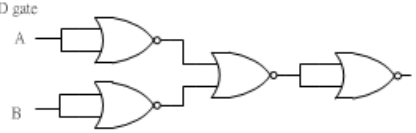
OR gate



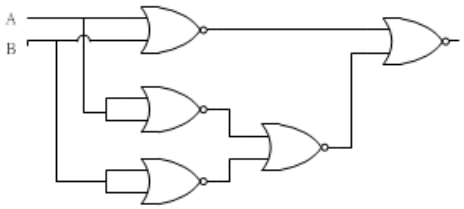
AND gate



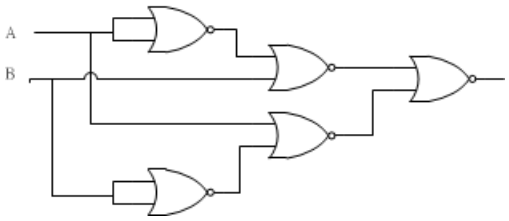
NAND gate



XOR gate



XNOR gate



VERILOG QUESTION 1:

(GATE LEVEL) BINARY CODE TO GREY CODE

1. Dout[0] 的 K-map

din[1], din[0]

din[3], din[2]	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

2. Dout[1] 的 K-map

din[1], din[0]

din[3], din[2]	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	0	0
10	0	0	1	1

3. Dout[2] 的 K-map

din[1], din[0]

din[3], din[2]	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

4. Dout[3] 的 K-map

din[1], din[0]

din[3], din[2]	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

由 K-map 的結果可以發現:

Dout[0] = (not(Din[1]) & Din[0]) | (Din[1] & not(Din[0]))

Dout[1] = (not(Din[1]) & Din[2]) | (Din[1] & not(Din[2]))

Dout[2] = (not(Din[3]) & Din[2]) | (Din[3] & not(Din[2]))

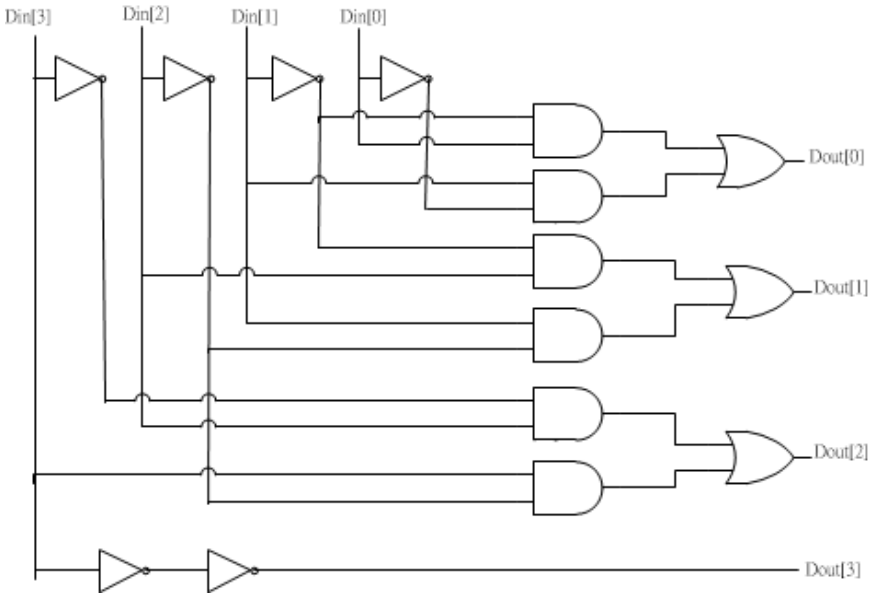
Dout[3] = Din[3]

它們所對應的 code 是這樣:

```
not not1(not_a, din[3]);
not not2(not_b, din[2]);
not not3(not_c, din[1]);
not not4(not_d, din[0]);
```

```
and and01(and_0_1, not_c, din[0]);
and and02(and_0_2, din[1], not_d);
or or01(dout[0], and_0_1, and_0_2);

and and11(and_1_1, din[2], not_c);
and and12(and_1_2, not_b, din[1]);
or or11(dout[1], and_1_1, and_1_2);

and and21(and_2_1, not_a, din[2]);
and and22(and_2_2, din[3], not_b);
or or21(dout[2], and_2_1, and_2_2);

not not31(dout[3], not_a);
```

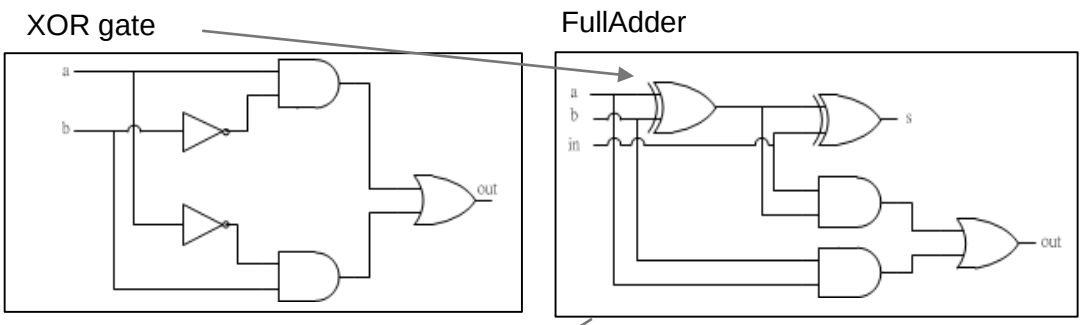
邏輯圖如下：



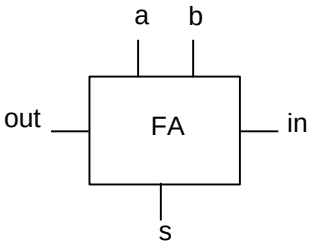
這題我所使用的驗證方法是將 input din 從 4’ b0000 加到 4’ b1111 以驗證所有的可能性。

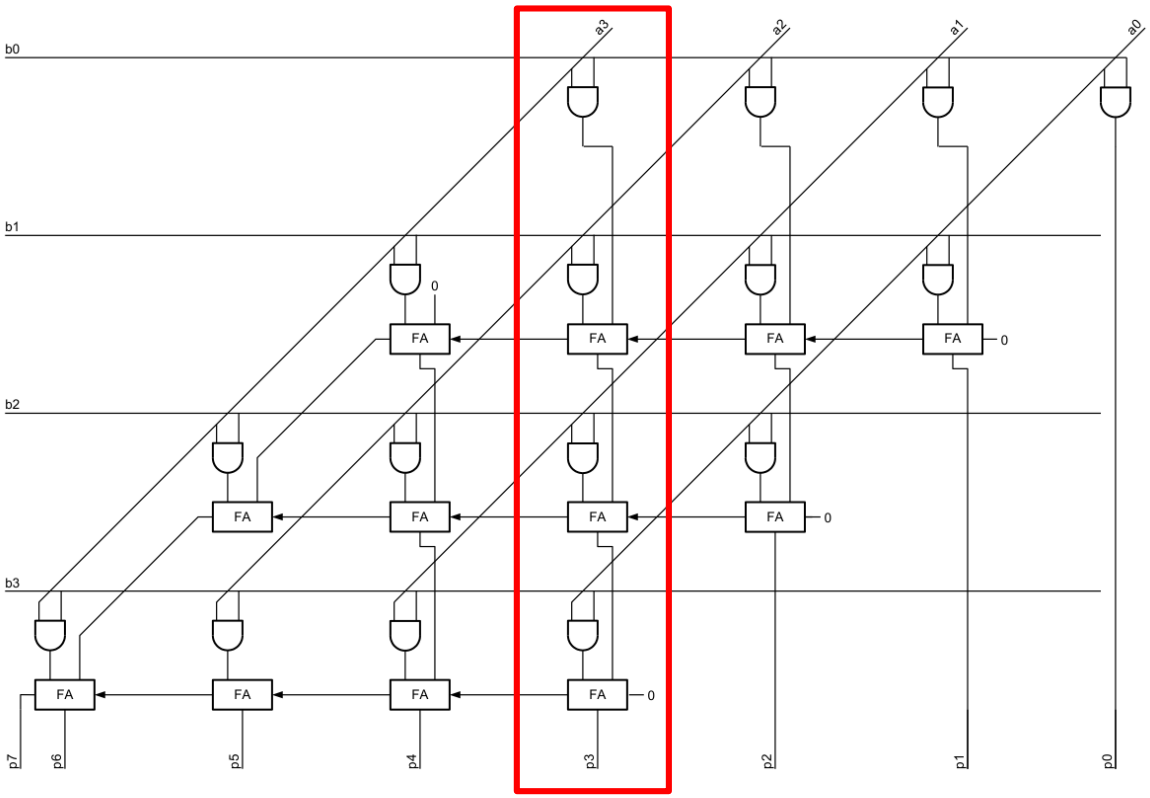
VERILOG QUESTION 2

(GATE LEVEL) MULTIPLIER



以下 FullAdder 腳位對應為





(圖片來自 W. J. Dally Digtal Design: A System Approach)

基本上，用 gate-level 寫出 4*4 的乘法器是跟數學直式完全相同。

首先，將 a[0]*b[0]放在最右 bit，並逐漸向左邊的 bit 移動，分別乘出 a[1]*b[0]、a[2]*b[0]、a[3]*b[0]。用 a[0]乘完一輪 b 的各 bit 後，要換成 a[1]乘 b 的各 bit，此時如同直式乘法，要整列向左移一 bit。每個 bit 相乘時，只有 1*1 才會得到 1，否則都是 0，所以用 AND-gate。

每個 bit 都乘完後，只需處理行的加法與進位問題。此時用上 1 bit Full Adder。因為 1 bit Full Adder 的 input 只有 a、b、cin，故一次只能處理兩個 1 bit 的相加。如圖上紅框處，若兩個 AND-gate 經第一個 FA 相加後，因為下方還有運算未結束，於是將 sum 作為下方 FA 的 input b。持續相同動作，直到一整行完成運算，將 sum 輸出成 p[3]。

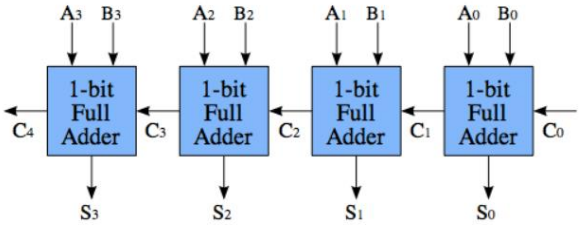
若某 FA 的 input a、b、cin 都為 1，此時除了 sum=1 還會有 cout=1，跟數學進位法相同，要將 cout 帶進左方 bit 的 FA 的 cin。在最右方的 FA 因為不會有 cin，所以直接用 0 代 cin。

Testbench 的寫法，是把全部的可能數字 a、b 都帶入乘法，驗證結果。

VERILOG QUESTION 3

(GATE LEVEL) 4-BIT CARRY-LOOKAHEAD (CLA) ADDER

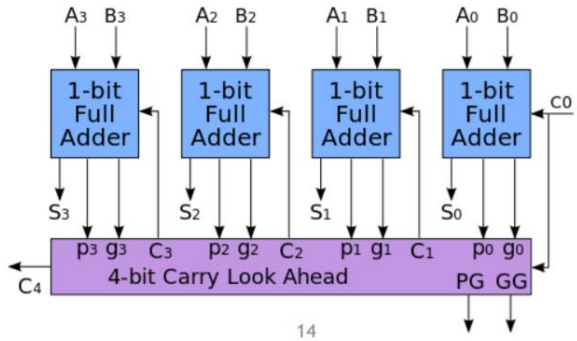
讓我們先從一般的 ripple 開始說起:



從右數來第二個 fulladder 必須等待第一個 fulladder 計算完並傳出 C1 後它才能正確計算出 C2 的值，

以此類推，C4 要等到 4 倍的 propagation delay 才能被正確的計算，而 carry-lookahead adder 解決了這樣的問題。

接下來我們來看看 carry-lookahead adder



其中 $S_i=A_i \oplus B_i \oplus C_i$

$C_{i+1} = A_i B_i \bar{C}_i + A_i B_i C_i + \bar{A}_i B_i C_i + A_i \bar{B}_i C_i$ $= A_i * B_i + (A_i \oplus B_i) * C_i$

$G_i = A_i * B_i$ (generate)

$P_i = A_i \oplus B_i$ (propagate)

所以 $C_{i+1} = G_i + P_i * C_i$ ，也就是說:

$C_1 = G_0 + P_0 * C_0$ ，

$C_2 = G_1 + P_1 * C_1$ ，

$C_3 = G_2 + P_2 * C_2$ ，

$C_4 = G_3 + P_3 * C_3$ ，

經過一些替代後，我們得到:

$C_0 = Cin$ ，

$C_1 = G_0 + P_0 * C_0$ ，

$C_2 = G_1 + G_0 * P_1 + C_0 * P_0 * P_1$ ，

$C_3 = G_2 + G_1 * P_2 + G_0 * P_1 * P_2 + C_0 * P_0 * P_1 * P_2$ ，

$C_4 = G_3 + G_2 * P_3 + G_1 * P_2 * P_3 + G_0 * P_1 * P_2 * P_3 + C_0 * P_0 * P_1 * P_2 * P_3 = Cout$ ，

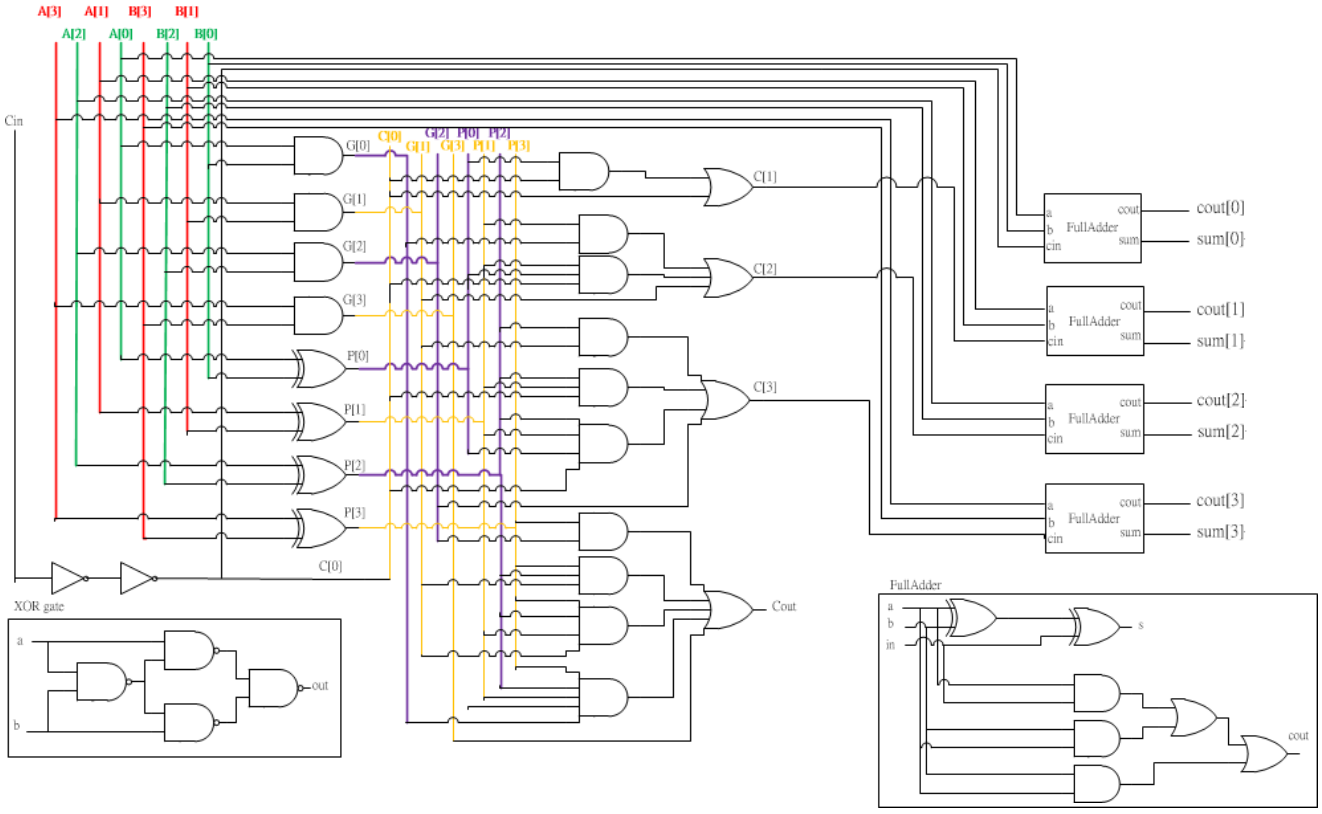
C0, C1, C2, C3, C4(cout) 所對應的 code 如下:

```
//c[0]
not not_1(not1, cin);
not not_2(C[0], not1);

//c[1]
and and_2(and2, P[0], C[0]);
or or_1(C[1], G[0], and2);

//c[2]
and and_3(and3, P[1], G[0]);
and and_4(and4, P[1], P[0], C[0]);
or or_2(C[2], G[1], and3, and4);

//c[3]
and and_5(and5, P[2], G[1]);
and and_6(and6, P[2], P[1], G[0]);
and and_7(and7, P[2], P[1], P[0], C[0]);
or or_3(C[3], G[2], and5, and6, and7);

//cout
and and_8(and8, P[3], G[2]);
and and_9(and9, P[3], P[2], G[1]);
and and_10(and10, P[3], P[2], P[1], G[0]);
and and_11(and11, P[3], P[2], P[1], P[0], C[0]);
or or_4(cout, G[3], and8, and9, and10, and11);
```

Sum 的部分應 slide 得要求，我使用了 lab1 的 fulladder module 來算出它，code 如下:

```
//sum
FullAdder f0(a[0], b[0], C[0], cout0, sum[0]);
FullAdder f1(a[1], b[1], C[1], cout1, sum[1]);
FullAdder f2(a[2], b[2], C[2], cout2, sum[2]);
FullAdder f3(a[3], b[3], C[3], cout3, sum[3]);
```

邏輯圖如下：



從這邊我們可以看出，每個位數所產生的進位都是獨立生成的，跟前面一個位元的運算沒有關聯，只和 input 以及 Cin 相關，因為這樣它減少了一般 ripple carry adder 進位所產生的延遲。

這題我所使用的驗證方法是賦予 input a, b, cin 不同的值來進行確認，但因為 a 跟 b 都有 16 種可能性, cin 有兩種可能性，所以有 512 種組合，要一一驗證會耗費大量時間與精力，因此我採用人工賦予 a, b 以及 cin 數值的方式，進行抽驗。

FPGA 的部分:

我將 AN[3] 、AN[2]、AN[1] 設成 SW[8] + not (SW[8]) 這樣一來不管 SW[8] 是多少它們都會等於 1

```
//AN
not not_3(not2, SW[8]);
or or_5(AN[3], SW[8], not2);
or or_6(AN[2], SW[8], not2);
or or_7(AN[1], SW[8], not2);
or or_8(or1, SW[8], not2);
not not_4(AN[0], or1);
```

至於 seg 我是用 Decoder 的方式將它從 sum 中 decode 出來，如表格:

Sum[3:0]	Seg[0:6]
0000	0000001
0001	1001111
0010	0010010
0011	0000110
0100	1001100
0101	0100100
0110	0100000
0111	0001111
1000	0000000
1001	0000100
1010	0001000
1011	1100000
1100	0110001
1101	1000010
1110	0110000
1111	0111000

畫完 k-map 後可以知道

Seg[0] = $\bar{a}\bar{b}\bar{c}d + \bar{a}b\bar{c}\bar{d} + ab\bar{c}d + a\bar{b}cd$

Seg[1] = $ab\bar{c}\bar{d} + \bar{a}b\bar{c}d + acd + abc + bc\bar{d}$

Seg[2] = $ab\bar{c}\bar{d} + \bar{a}\bar{b}c\bar{d} + abc$

Seg[3] = $\bar{a}b\bar{c}\bar{d} + \bar{a}\bar{b}\bar{c}d + bcd + a\bar{b}c\bar{d}$

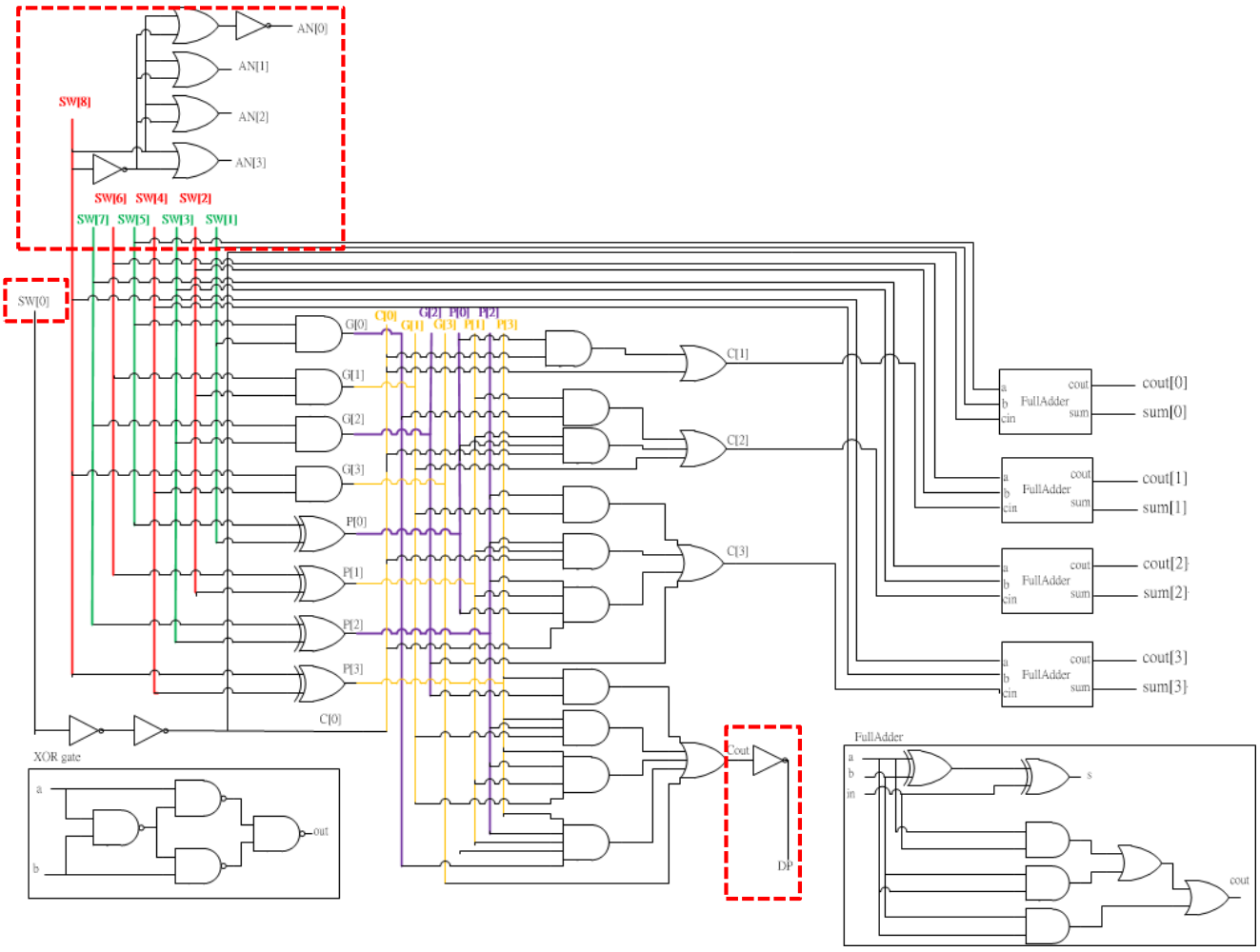
Seg[4] = $\bar{a}b\bar{c} + \bar{b}\bar{c}d + \bar{a}d$

Seg[5] = $ab\bar{c}d + \bar{a}\bar{b}d + \bar{a}cd + \bar{a}\bar{b}c$

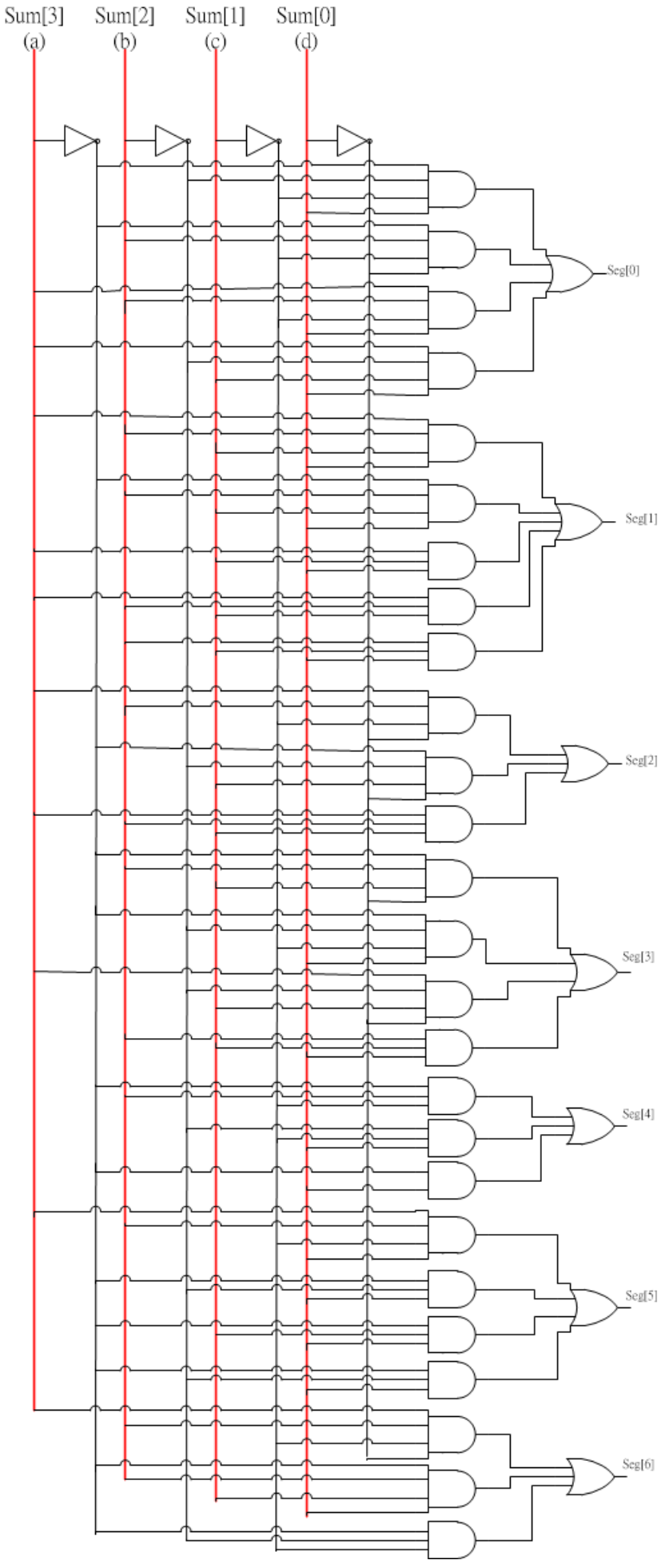
Seg[6] = $ab\bar{c}\bar{d} + \bar{a}bcd + \bar{a}\bar{b}\bar{c}$

DP 則是 Cout 經過 not 運算的結果。

FPGA 實作的邏輯圖與上圖類似，不過將 input 的 A、B 換成 SW。並且主要是多處理了七段顯示器的 decode。邏輯圖如下：



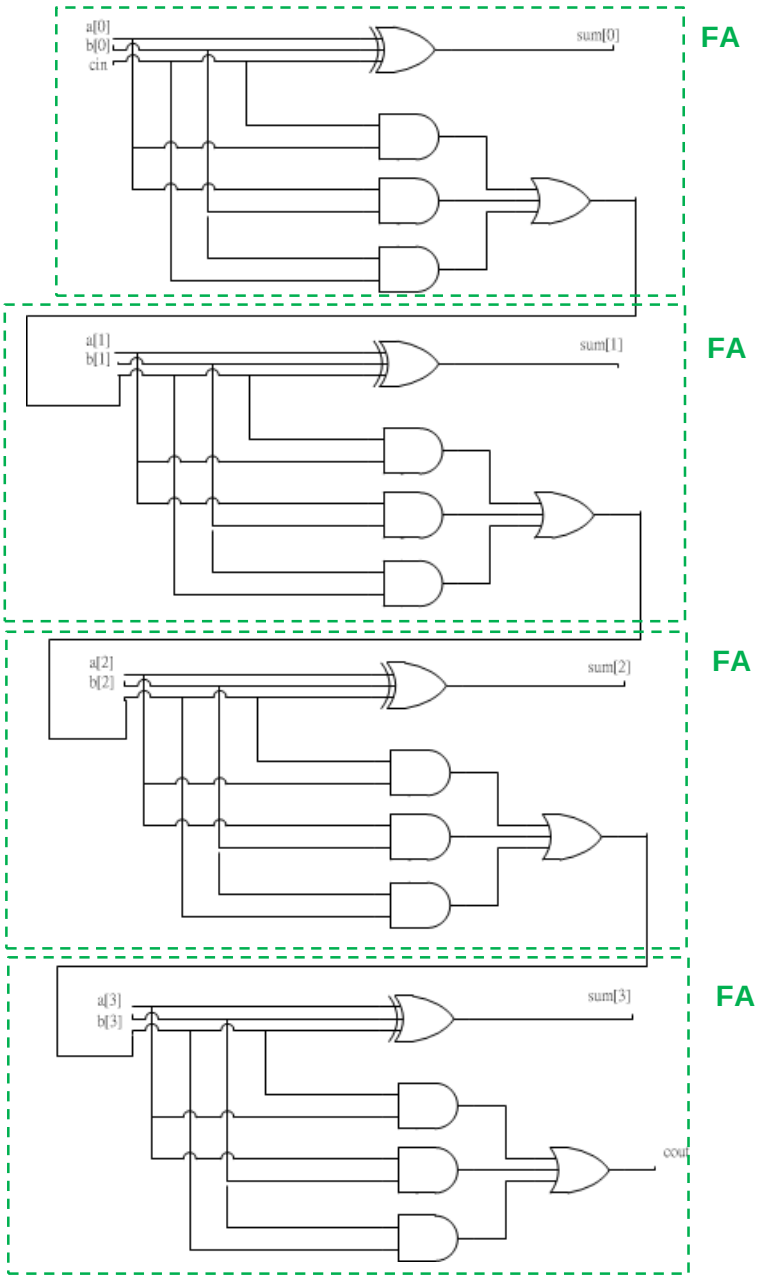
(篇幅原因，下圖接續上圖，處理 SEG 部分)

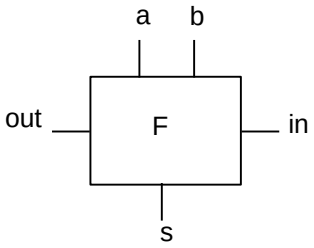

Sum[3]
(a)
Sum[2]
(b)
Sum[1]
(c)
Sum[0]
(d)
Seg[0]
Seg[1]
Seg[2]
Seg[3]
Seg[4]
Seg[5]
Seg[6]

VERILOG QUESTION 4

(GATE-LEVEL) 4-BIT RIPPLE-CARRY ADDER (RCA)

ADD MODULE





ADD 是由四個 FullAdder 串接，a[0]和 b[0]當 a、b，輸出 sum[0]與 cout，並且將 cout 連接至下一個 FullAdder (a、b input 是 a[1]和 b[1]) 的 cin。因為 a、b input 是 a[0]和 b[0]的 FullAdder 沒有 cin 故代 0。

SUB MODULE

SUB 的作法與 ADD 類似。a–b=a+(-b)，所以減法可以看成加一個負數。先把 b 用 NOT-gate 做二補數，再加 1’ b1 即是-b。因此把 b[0]、b[1]、b[2]、b[3]做 not，放在上圖中 b[0]、b[1]、b[2]、b[3]位置，再將 FA 的 LSB 的 cin 代入 1(作加法時代入 0)，代表加 1’ b1，即是答案。

INC MODULE

INC 的作法與 ADD 類似。只需把 b 設定成 4’ b0001，並且 LSB 的 FA 的 cin 還是維持代 0，即是答案。

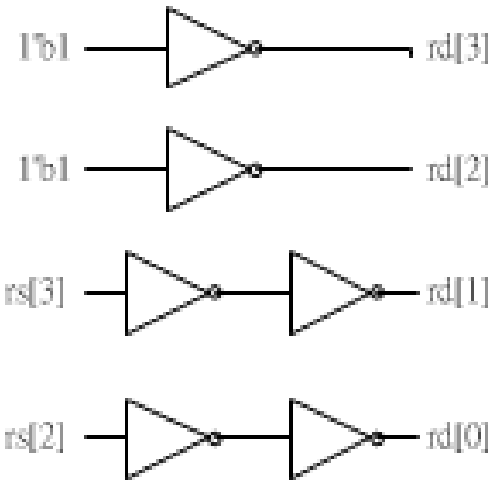
BITWISE NOR MODULE

BITWISE NOR 的作法即是把各個 bit 分開用 NOR-gate 做。

BITWISE NAND MODULE

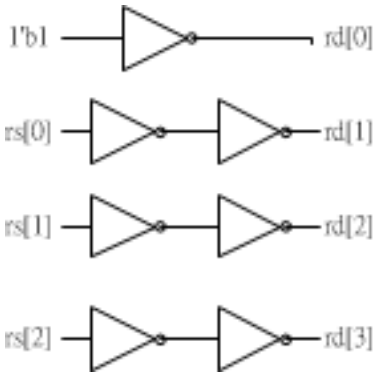
BITWISE NAND 的作法即是把各個 bit 分開用 NAND-gate 做。

RS DIV 4 MODULE



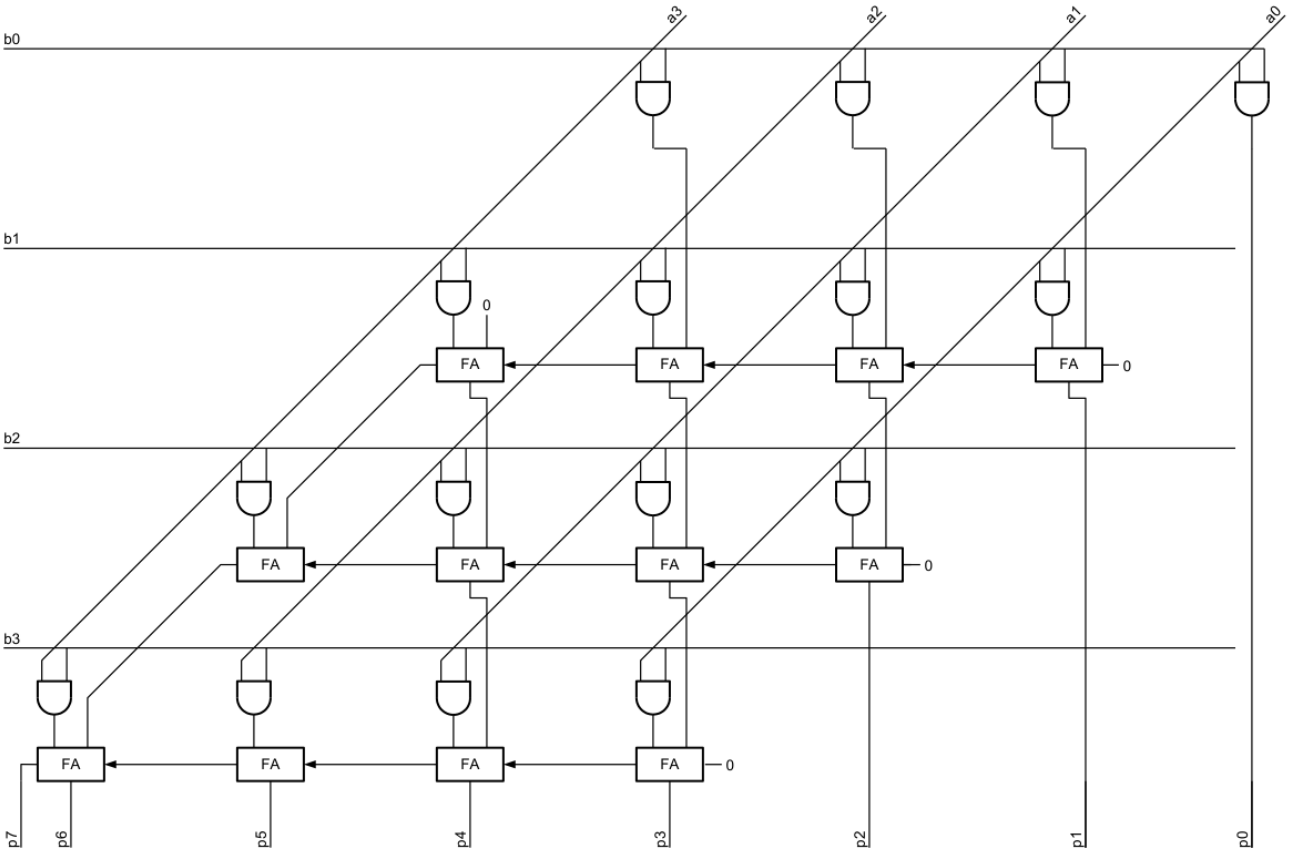
將 RS 除以 4 即是在二進位中，將 RS 的各個 bit 右移兩個 bit，並且因為題目中所說為〝>>〞，代表用 logic shift 就好，所以在右移兩 bit 後，在最大的兩個 bit 直接補 0。

RS_MUL_2 MODULE



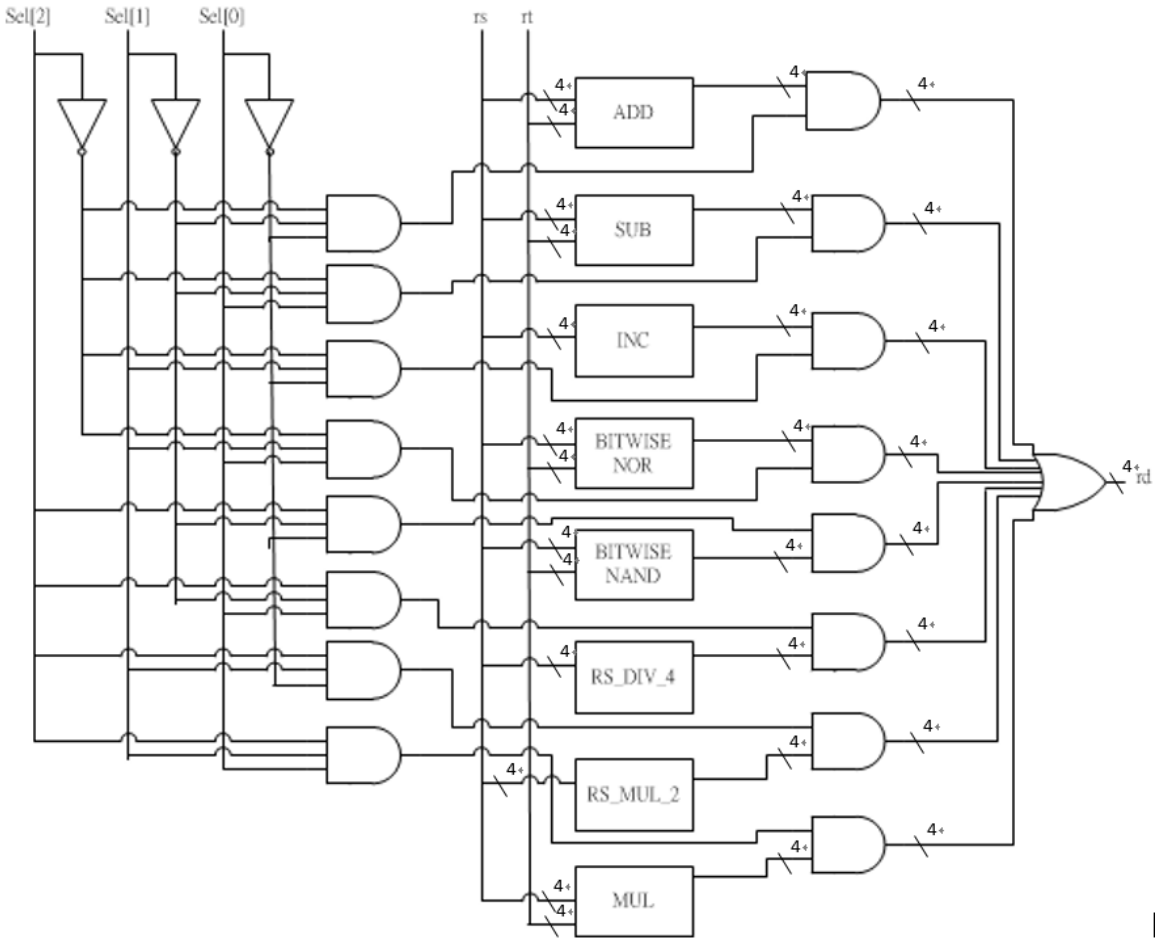
將 RS 乘以 2 即是在二進位中，將 RS 的各個 bit 左移一個 bit，並且因為題目中所說為〝<<〞，代表用 logic shift 就好(雖然左移時的 logic shift 和 asithmetic shift 相同)。

Multiplier MODULE



這與進階題第二題的 multiplier 完全相同，邏輯圖與說明見第二題，故不贅述。

Putting the modules altogether



此題的 testbench 寫法是將所有可能的數值代入 input，並檢查 output。

心得(BY 郭家偉)：

這次 lab 我覺得比較有挑戰性與趣味的是 Multiplier。如果用 behavior level 寫起來可能會很簡單，但用 gate level 寫起來就比較考驗邏輯了。不過用 gate level 寫起來也比較清楚程式實際的運作方式，而不是靠軟體去 synthesis，也比較不會出問題，但是寫起來也相當費時費力。

心得(BY 黎佑廷)：

我覺得這次 lab 第一題不難，只要畫畫 k-map 答案就呼之欲出了，必較麻煩的是第三題，在搞懂 carry lookahead adder 的過程就花了一些時間，至於 fpga 的部分，因為是第一次使用 7-Segment Display，所以也花了一些時間摸索。

成員:

黎佑廷: 負責第 1、3 題以及整理 report

郭家瑋: 負責第 2、4 題以及畫出 gate level circuit