

Lab2_Team1_Report

Basic Question1

Drawing of the design of Verilog Advanced Question 1

- Not_gate module

Not_gate

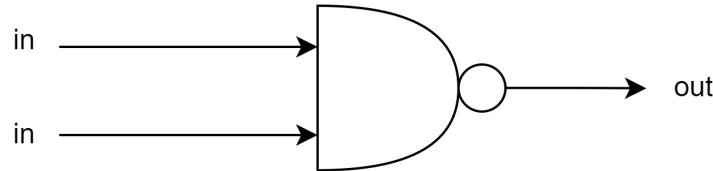


图1-1

$$- (in \cdot in)' = in'$$

- Or_gate module

Or_gate

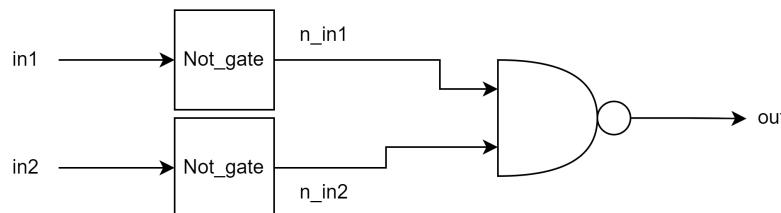


图1-2

$$- n_{in1} = in1'$$

$$- n_{in2} = in2'$$

$$- (n_{in1} \cdot n_{in2})' = (in1' \cdot in2')' = in1 + in2$$

- And_gate module

And_gate

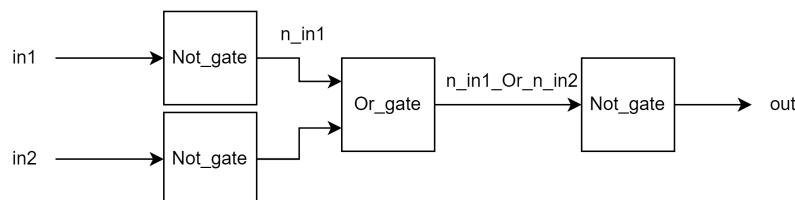


图1-3

$$- n_{in1} = in1'$$

$$- n_{in2} = in2'$$

$$- (n_{in1} + n_{in2})' = (in1' + in2')' = in1 \cdot in2$$

- Xor_gate module

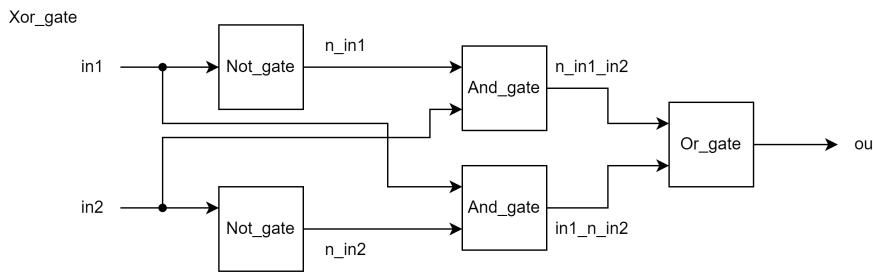


圖1-4

- $in1 \text{ xor } in2 = in1'in2 + in1in2'$
- $n_in1 = in1'$
- $n_in2 = in2'$
- $n_in1_in2 = n_in1 \cdot in2 = in1' \cdot in2$
- $in1_n_in2 = in1 \cdot n_in2 = in1 \cdot in2'$

- And_gate_3bit module

And_gate_3bit

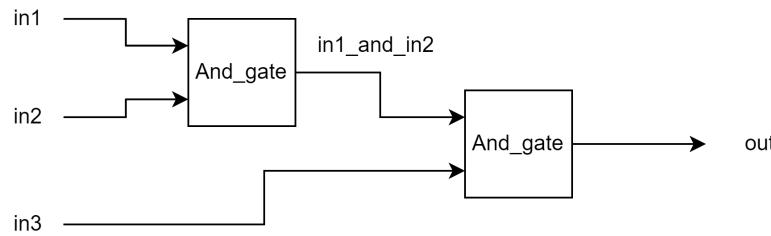


圖1-5

- 3個1bit input的and gate
- 由圖1-3所畫的And_gate module所實作出來

- Nand implement module

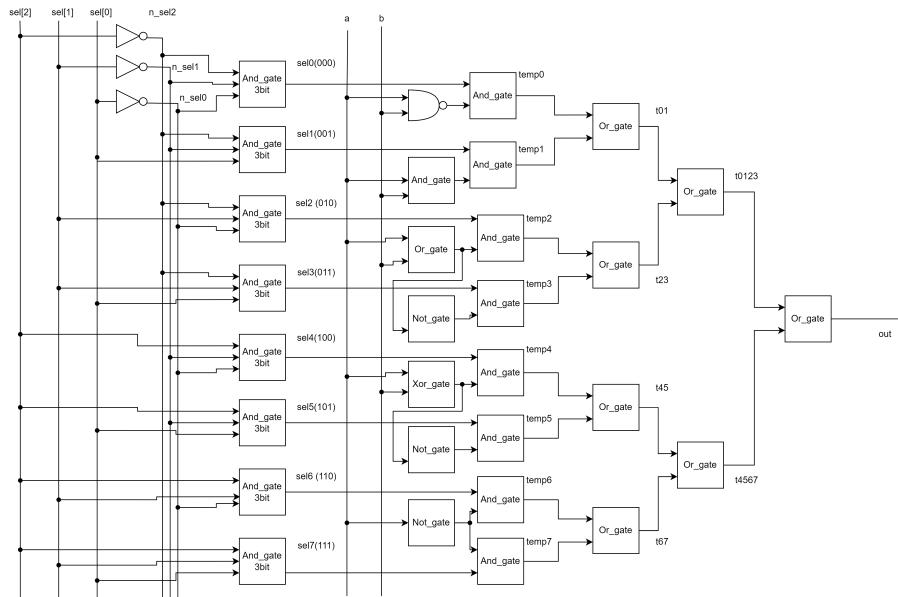


圖1-6

- 我們在用a,b以及其他modules實作出所要求的八種output之後，再把每一種結果跟相對應的sel接入And_gate module之後(e.g. sel0 = n_sel0 and n_sel1 and n_sel2，與sel = 3'b000時所應該得到的output做and的運算)，再把七種結果做Or的運算之後得到最後的結果。

Basic Question3**Explanation of the difference between full adder and half adder**

Half adder只有兩個input，而Full adder有三個。在所有的input當中，Full adder除了有Half adder有的a,b以外，還多了cin，代表Full adder是可以接收前面的進位的，例如 $2'b01 + 2'b01 = 2'b10$ ，此時再計算第二個bit的時候就接收了來自第一個bit的進位，而這樣的加法只有Full adder可以單獨實現，如果要用Half adder實現的話，則需要用兩個Half adder兜出一個Full adder才能得到我們所需要的結果（如下圖所示）。

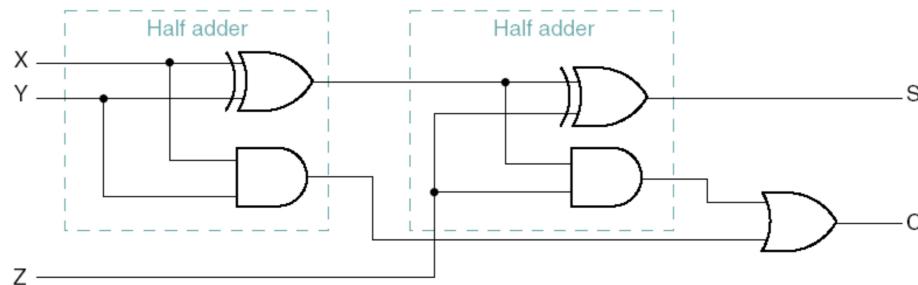
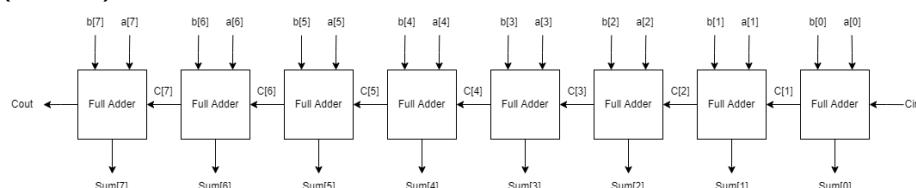


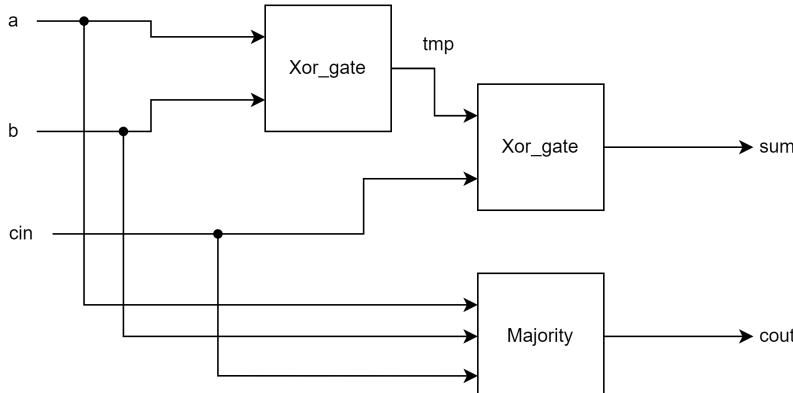
Fig. 3-27 Logic Diagram of Full Adder

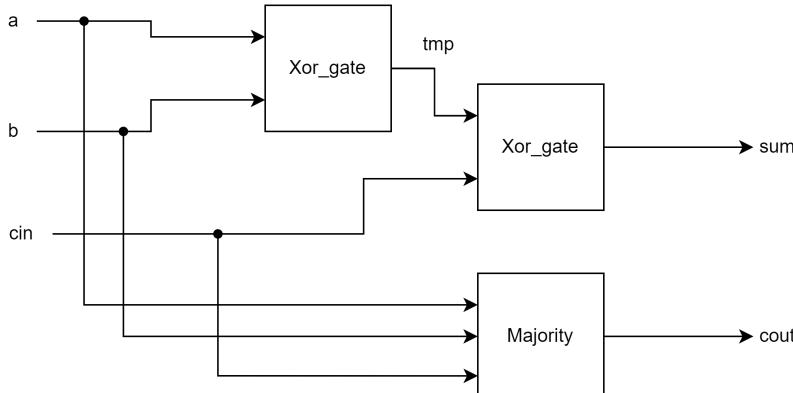
圖2-1(擷取自張世杰教授的邏輯設計講義)**Advanced Question1****Drawing of the design of Verilog Advanced Question 1**

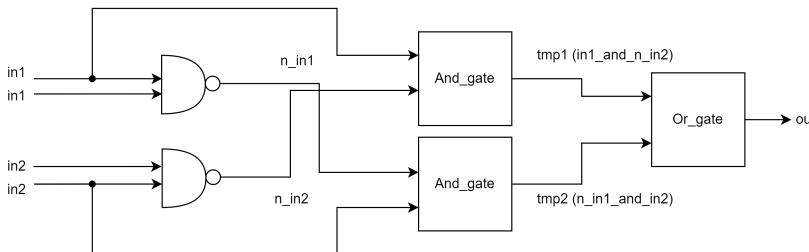
- Top module: Ripple Carry Adder (RCA)
(**圖3-1**)

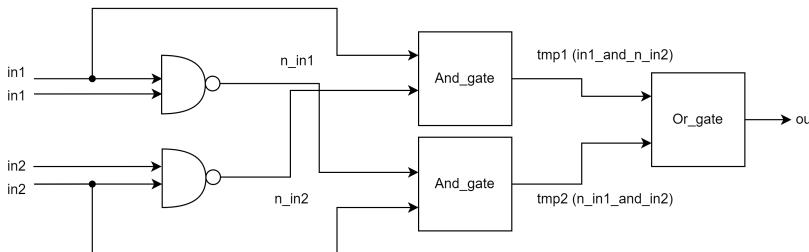


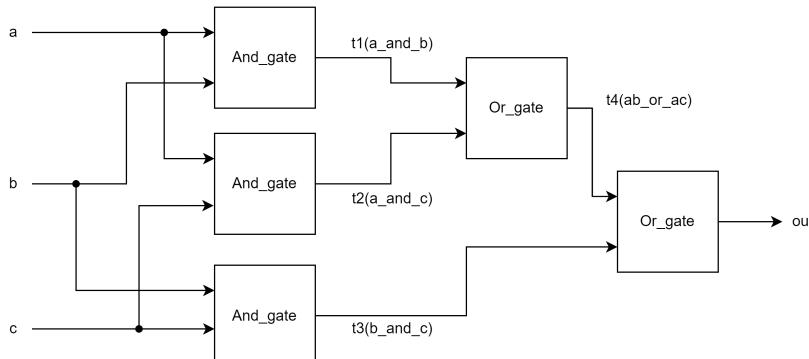
- module

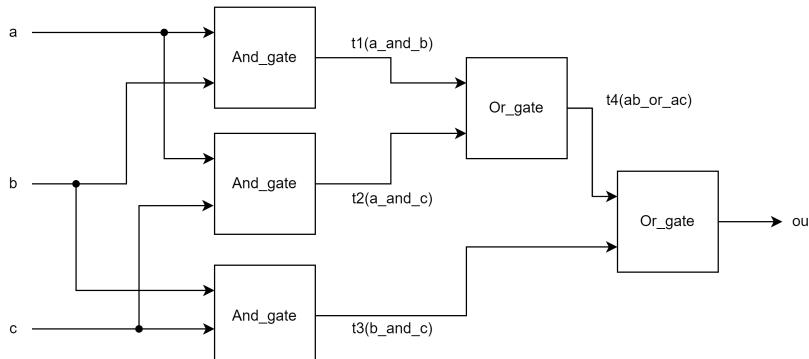
- Full Adder | in RCA (

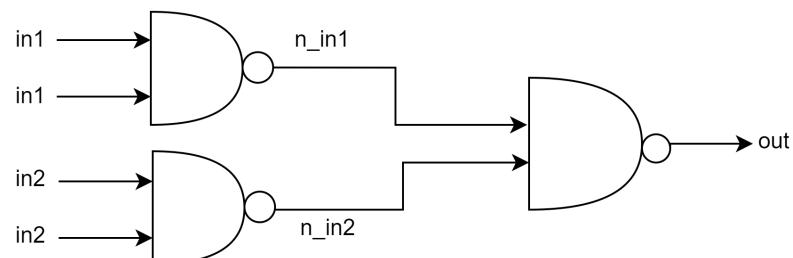


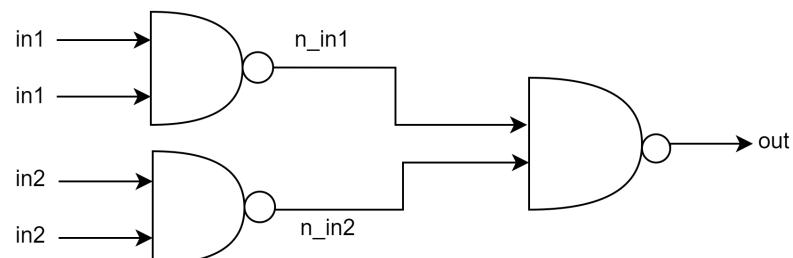
- Xor_gate | in Full Adder (

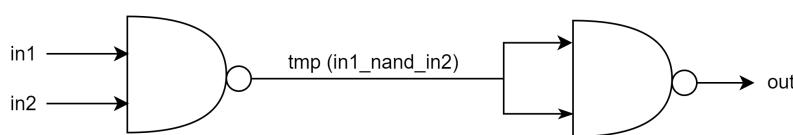


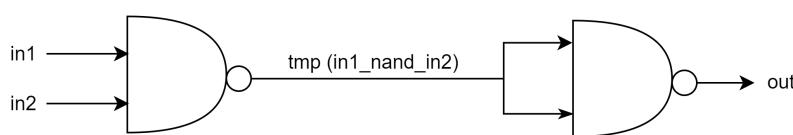
- Majority_gate | in Full Adder (



- Or_gate | in Xor_gate (



- And_gate | in Xor_gate (



Requirements

1. use NAND gate only

2. 依題目指示，會用 basic question3 實作出 1-bit Full Adder，延伸完成行波進位加法器(RCA)。

Design Explanation

1. 依照作業投影片上的題目需求來進行實作，使用 8 個 1-bit Full Adder 來完成 8-bit RCA 的任務，符合題目要求，依此畫出 circuits (圖3-1)。
2. 如同 Block Diagram 所示，Cin、a[0]、b[0]接入第一個 1-bit Full Adder，輸出 C[1]、Sum[0]，前者會繼續供應後者的 1-bit Full Adder 使用，後者輸出，以此類推...
 - a. Cin、a[0]、b[0]-> C[1]、Sum[0]
 - b. C[1]、a[1]、b[1]-> C[2]、Sum[1]
 - c. C[2]、a[2]、b[2]-> C[3]、Sum[2]
 - d. C[3]、a[3]、b[3]-> C[4]、Sum[3]
 - e. C[4]、a[4]、b[4]-> C[5]、Sum[4]
 - f. C[5]、a[5]、b[5]-> C[6]、Sum[5]
 - g. C[6]、a[6]、b[6]-> C[7]、Sum[6]
 - h. C[7]、a[7]、b[7]-> Cout、Sum[7]
3. 輸出 8 bit的 Sum 及 Cout。

Testbench Design & Result Explanation

1. 波形圖截圖

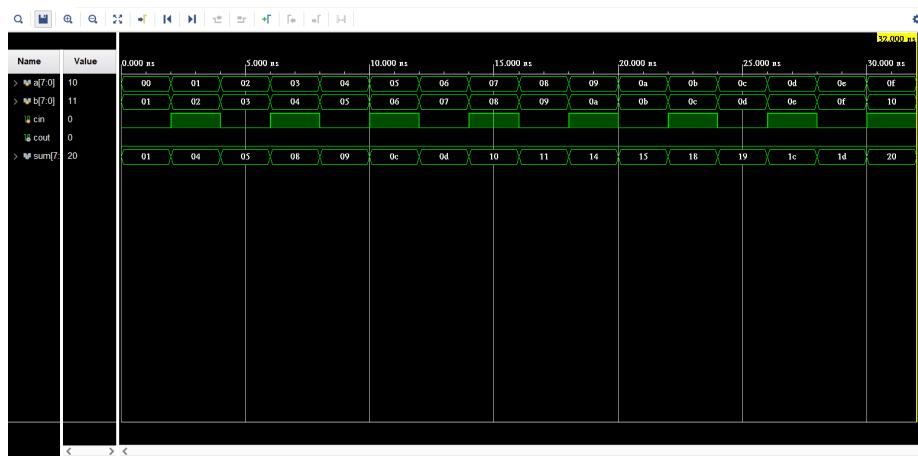


圖3-7

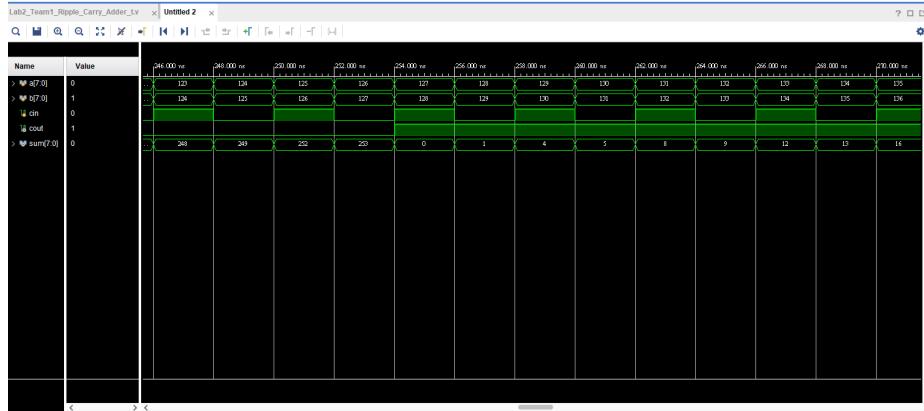


圖3-8

2. testbench設計說明

A. 初始化

- a 為 8'b00000000(0)
- b 為 8'b00000001(1)
- cin 為 1'b0。

B. 固定以下變化，每次循環(2ms)，並進行 2^8 次循環來看相對應跑出來的output是否正確

- a += 8'b00000001
- b += 8'b00000001
- cin += 1'b0

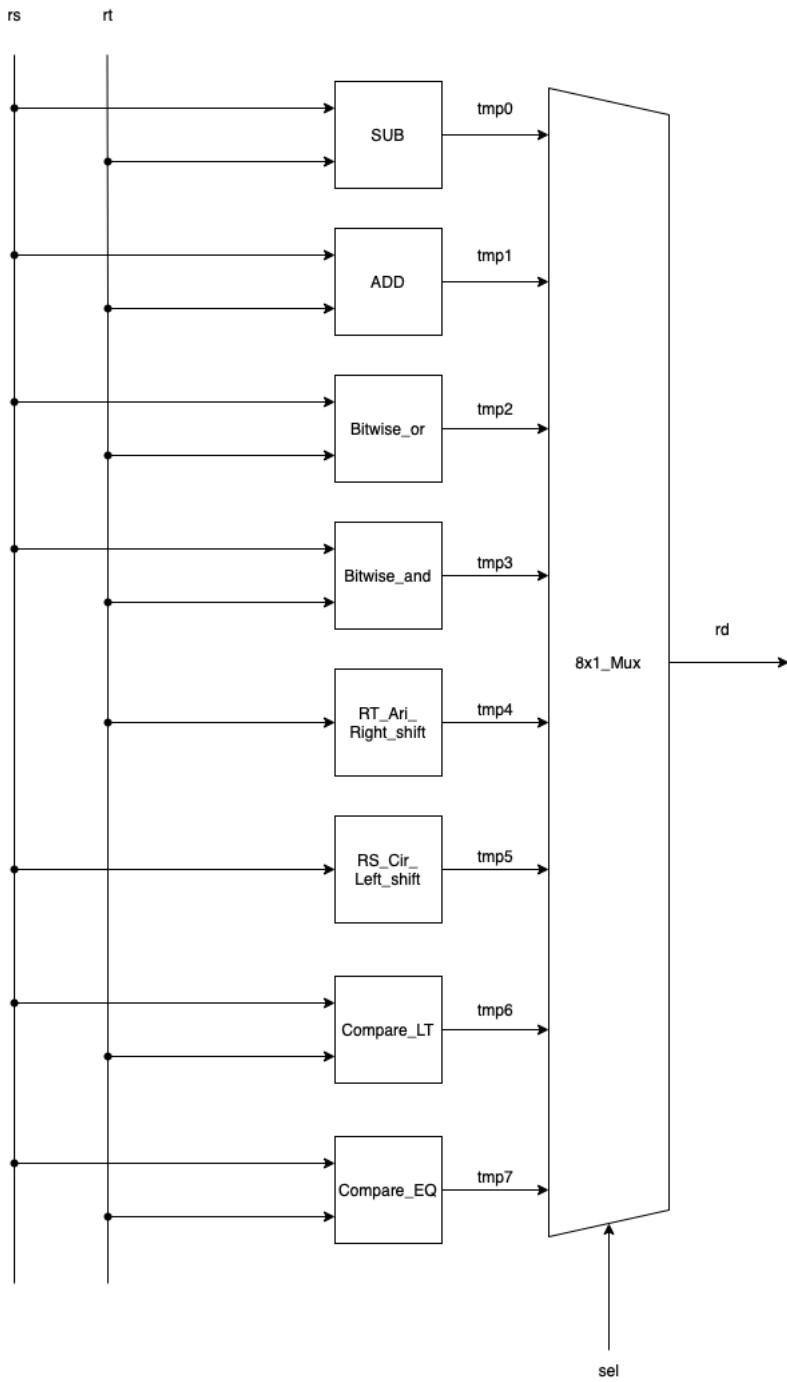
3. 波形圖結果說明

我們將波形圖與實際狀況做對照，8-bit RCA 的確有如預期輸入 a、b、cin 的運算而輸出，可以發現結果與我們所預期的結果相吻合。

Advanced Question2

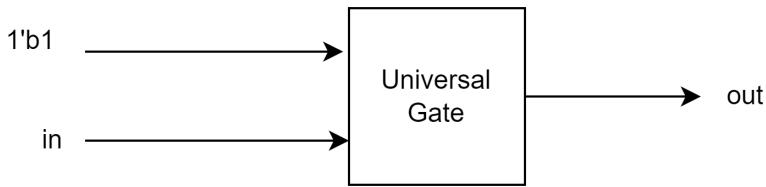
Drawing of the design of Verilog Advanced Question 2

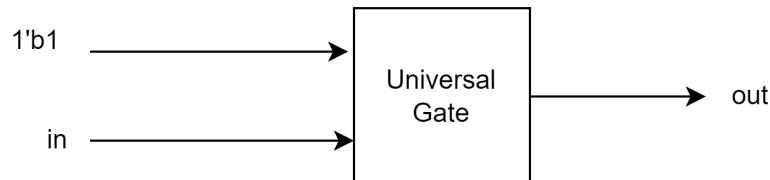
- Top module: Decode and Execute (圖4-20)



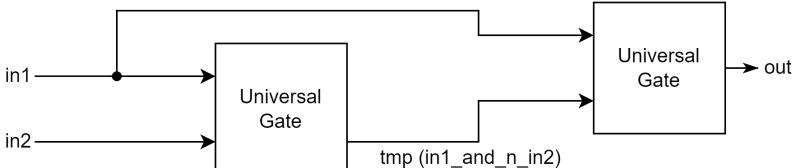
- module

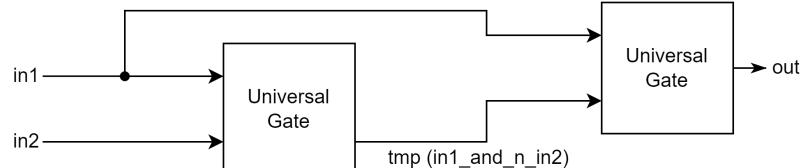
因為題目規定要使用 universal gate 做出 basic logic gates，因此並沒有使用到在 Q1 RCA 實作出的 modules，因是添加了以下規定格式的 modules:
(基礎或有用到 universal gate 的 module 會特別做介紹)

- Not_gate ()



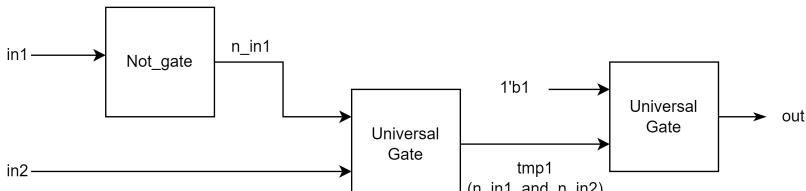
■ $1'b1 \cdot \sim(in) = \sim in$

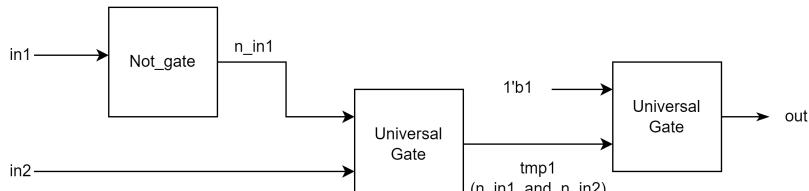
- And_gate ()



■ $\text{tmp} = \sim(in1) \cdot \sim(in2)$

■ $\text{out} = in1 \cdot \sim(in1 \cdot \sim(in2)) = in1 \cdot (\sim(\sim(in1)) + in2) = in1 \cdot in2$

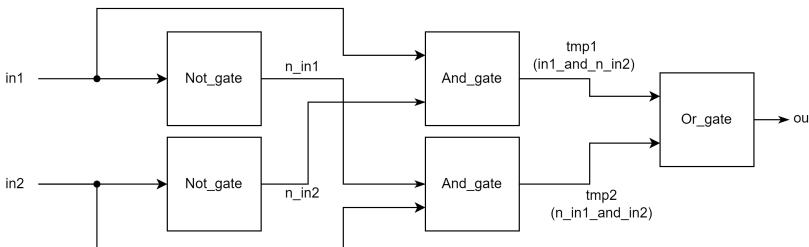
- Or_gate ()

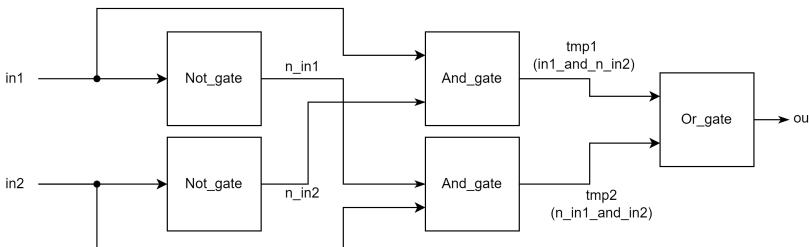


■ $n_in1 = \sim(in1)$

■ $\text{tmp1} = \sim(in1) \cdot \sim(in2)$

■ $\text{out} = 1'b1 \cdot \sim(\sim(in1) \cdot \sim(in2)) = in1 + in2$

- Xor_gate ()



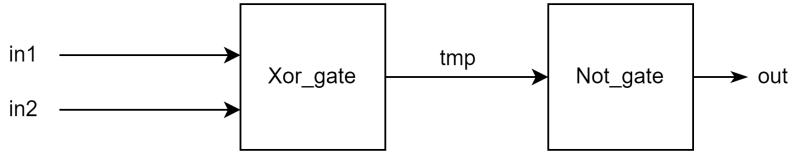
■ $n_in1 = \sim(in1)$

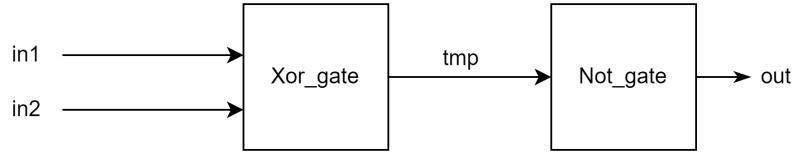
■ $n_in2 = \sim(in2)$

■ $\text{tmp1} = in1 \cdot \sim(in2)$

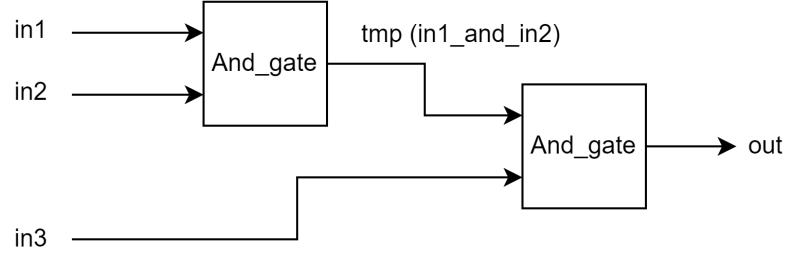
■ $\text{tmp2} = \sim(in1) \cdot in2$

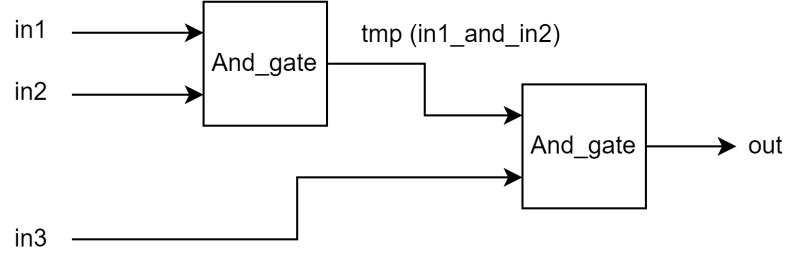
- $\text{out} = (\text{in1} \cdot \sim(\text{in2})) + (\sim(\text{in1}) \cdot \text{in2})$

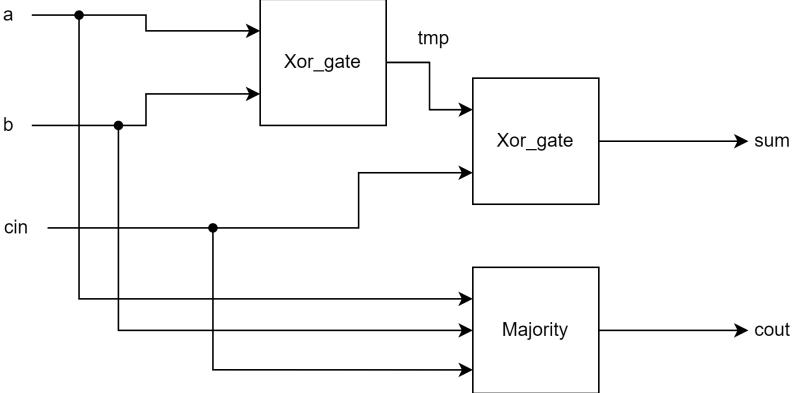
- Xnor_gate ()

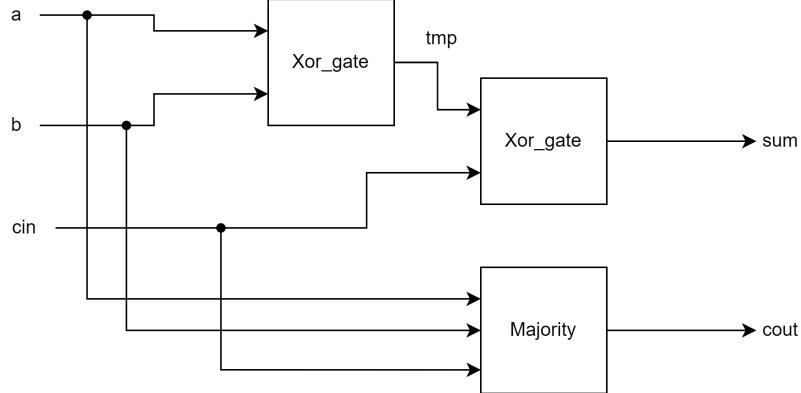


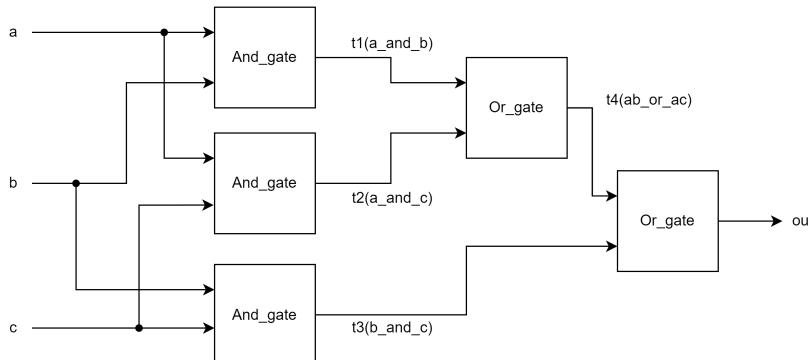
- $\text{tmp} = (\text{in1} \cdot \sim(\text{in2})) + (\sim(\text{in1}) \cdot \text{in2})$
- $\text{out} = \sim[(\text{in1} \cdot \sim(\text{in2})) + (\sim(\text{in1}) \cdot \text{in2})] = \text{in1} \cdot \text{in2}$
 $+ \sim(\text{in1}) \cdot \sim(\text{in2})$

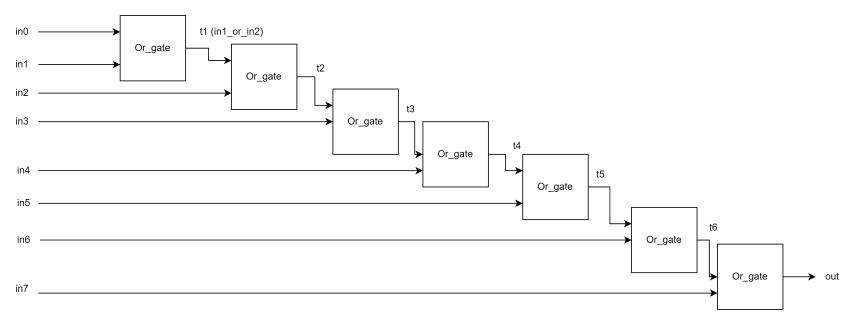
- And_gate_3bit ()



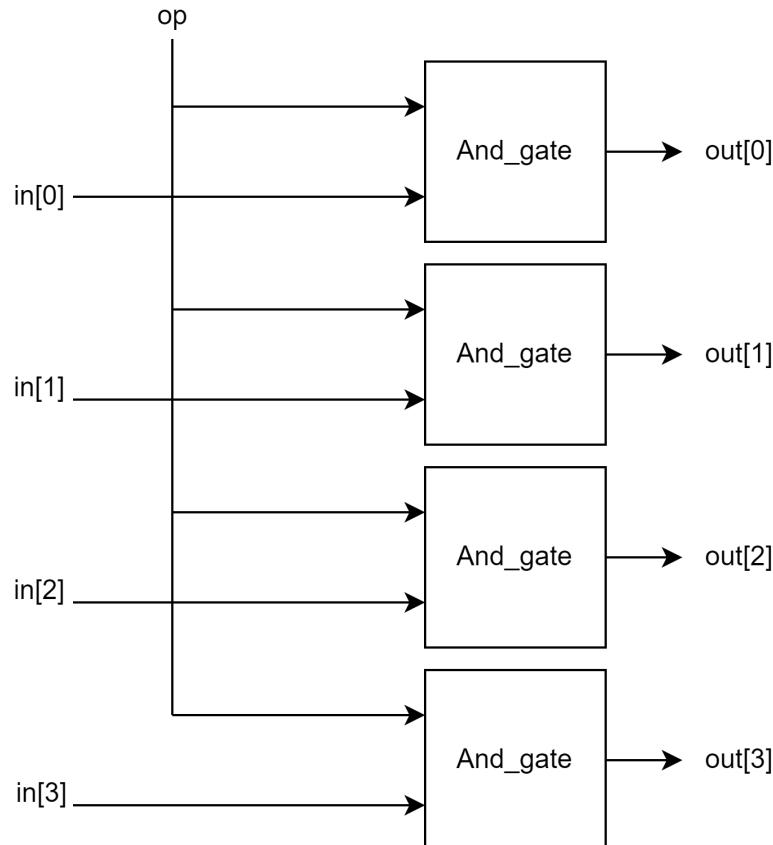
- Full adder ()



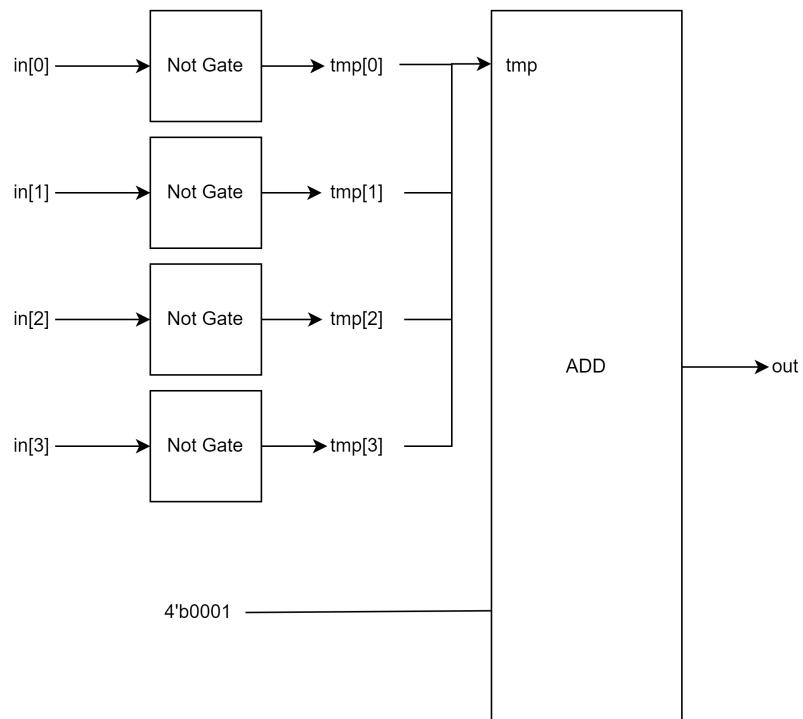
- Majority (Diagram 4-8: Majority logic circuit diagram. Three inputs a, b, and c enter three parallel AND gates. The outputs of these AND gates are labeled t1(a_and_b), t2(a_and_c), and t3(b_and_c). These three signals are fed into two OR gates. The first OR gate has inputs t1(a_and_b) and t2(a_and_c), with its output labeled t4(ab_or_ac). This output and signal t3(b_and_c) are fed into a final OR gate, which produces the output 'out'.

- Or_gate_8bit (Diagram 4-9: 8-bit OR gate circuit diagram. Eight inputs in0 through in7 enter a chain of seven OR gates. The first OR gate has inputs in0 and in1, with its output labeled t1 (in1_or_in2). This output and input in2 feed into the second OR gate. This pattern continues through six more OR gates, with intermediate outputs labeled t2, t3, t4, t5, and t6. The final output of the seventh OR gate is labeled 'out'.

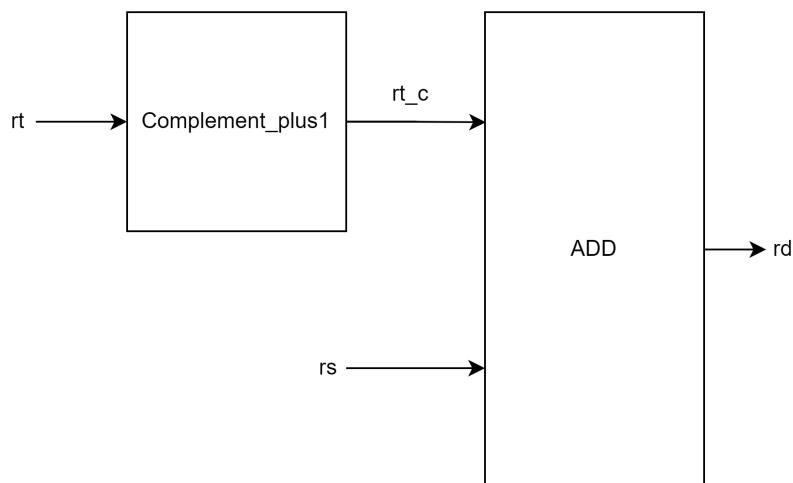
- filter



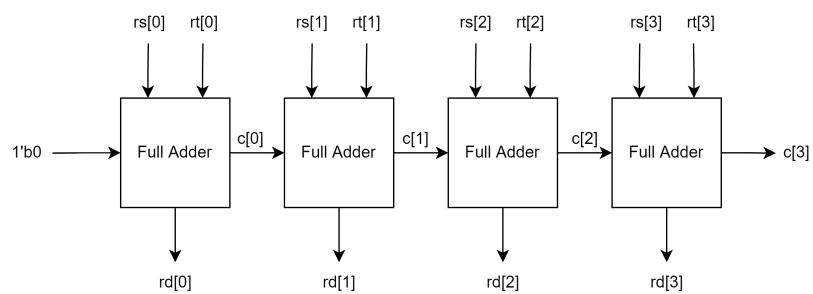
- Complement_plus1 (圖4-10)



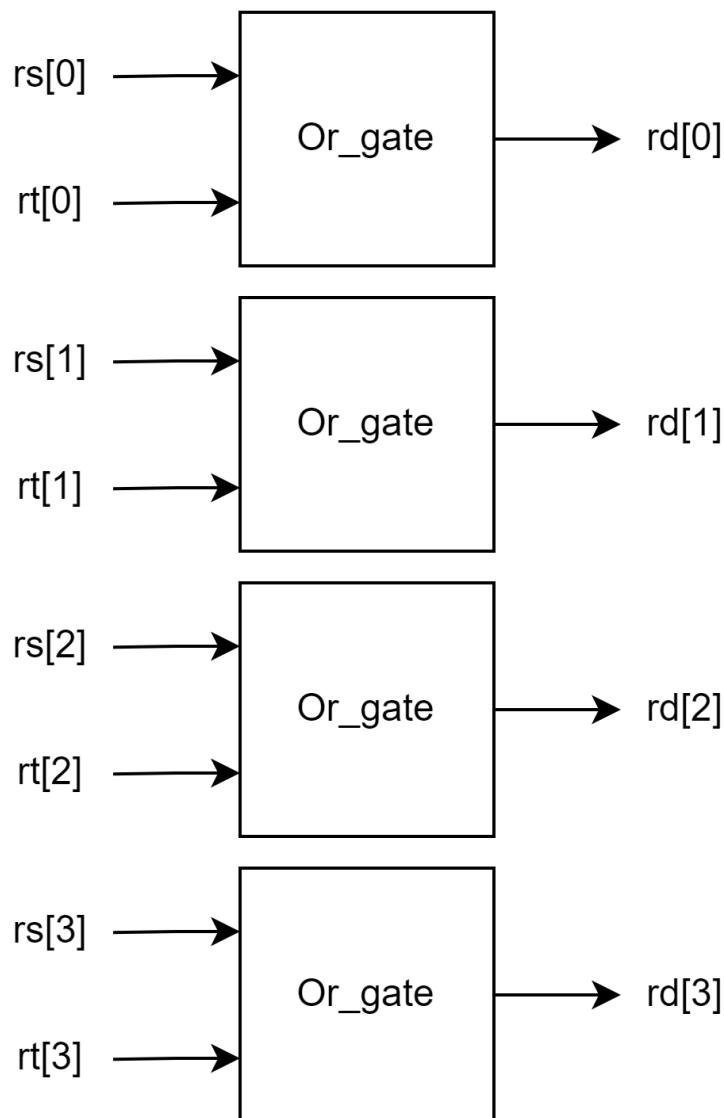
- function1: SUB for $[rd=rs-rt]$ (圖4-11)



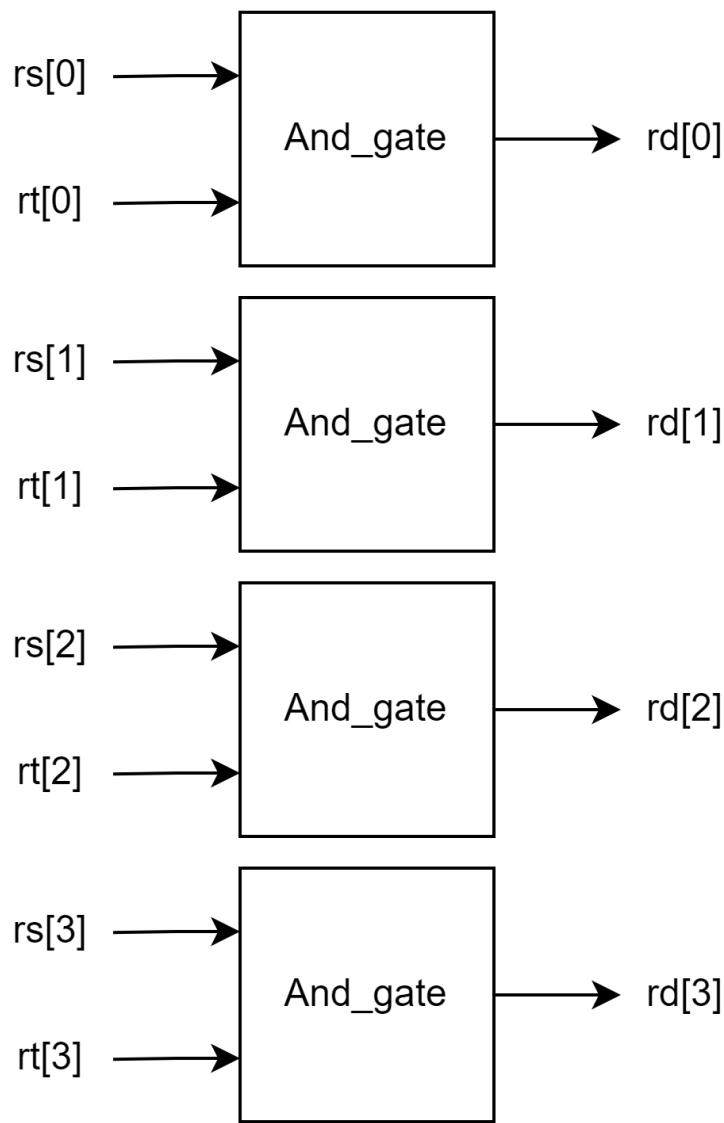
- function2: ADD for $[rd=rs+rt]$ (圖4-12)



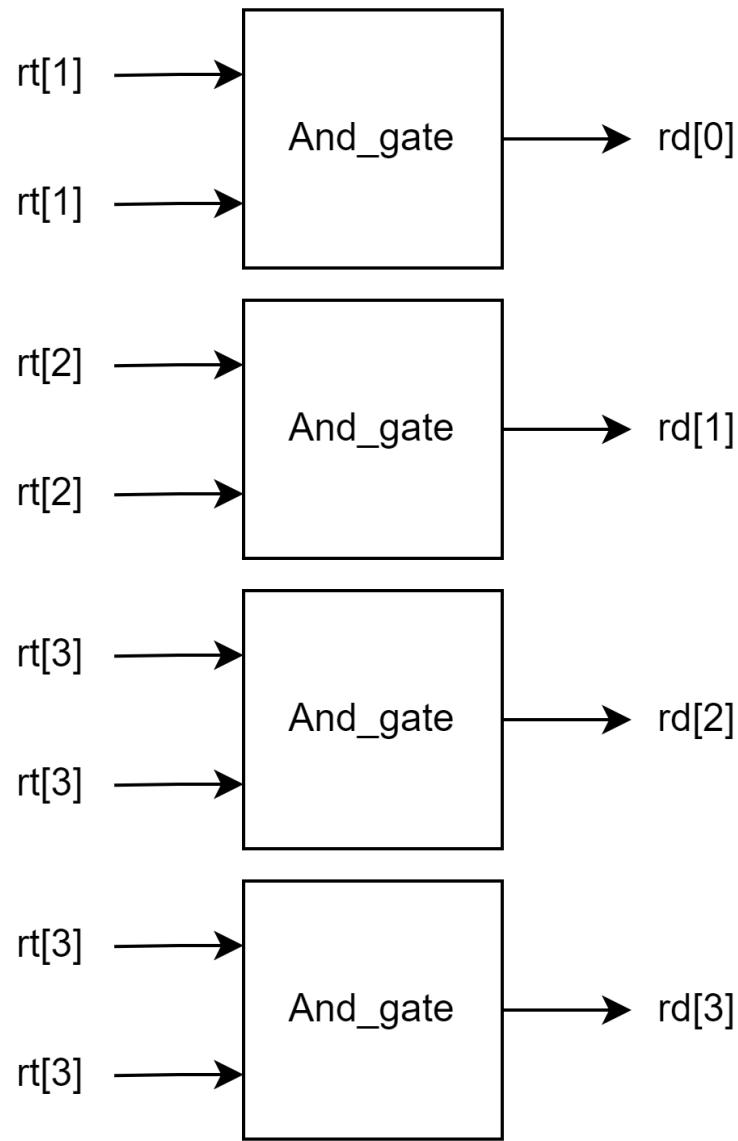
- function3: bitwise_or for [rd=rs(bitwise OR)rt] (圖4-13)



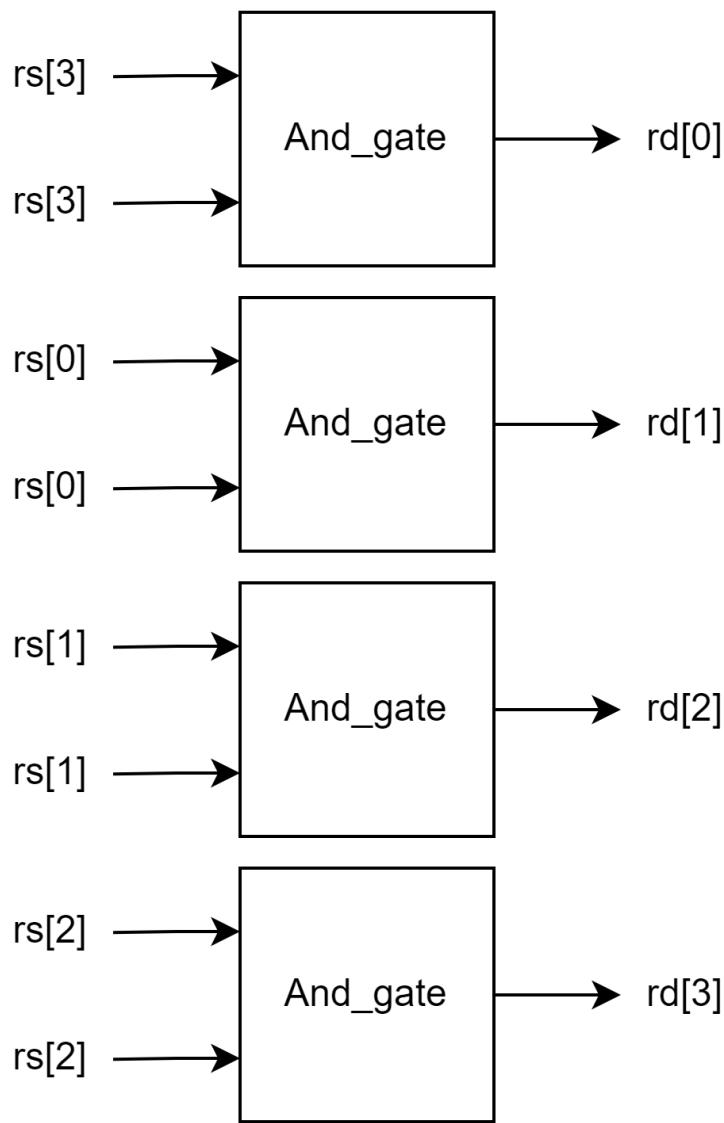
- function4: bitwise_and for [rd=rs(bitwise AND)rt] (圖4-14)



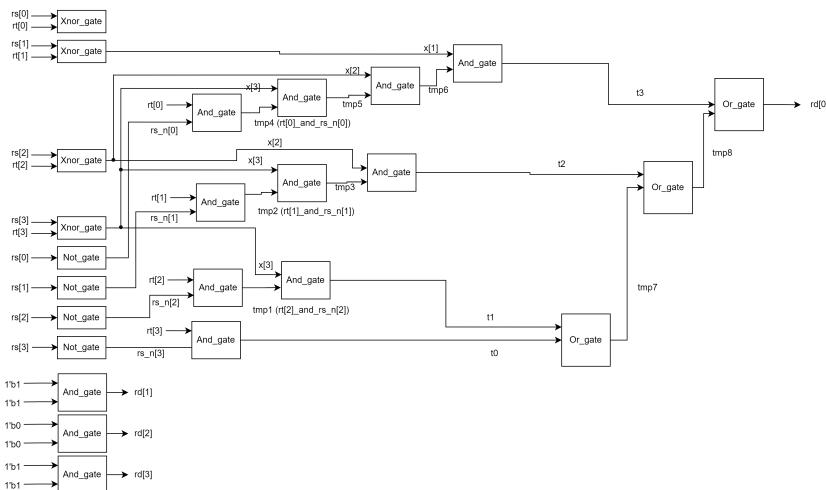
- function5: rt_ari_right_shift for [rd={rt[3], rt[3:1]}]
()



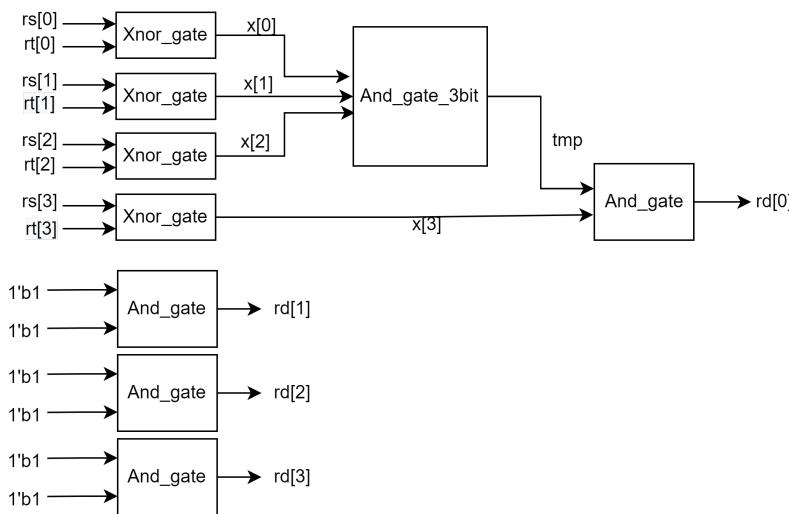
- function6: rs_cir_left_shift for [rd={rs[2:0], rs[3]}] (图 4-16)



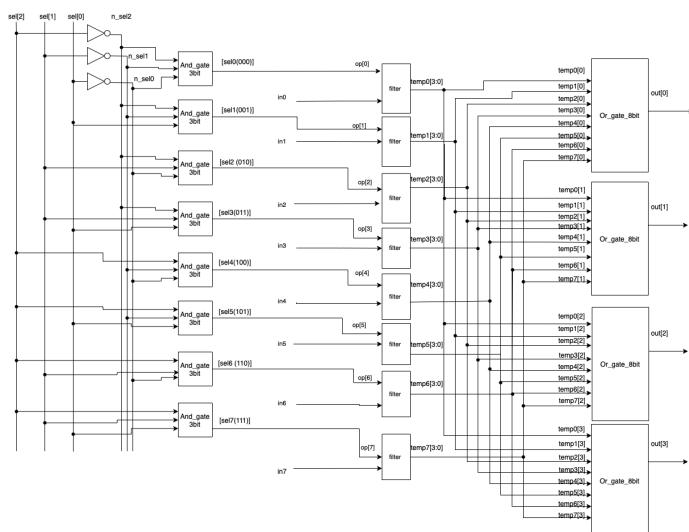
- function7: compare_lt for $[rd=\{3'b101, rs < rt\}]$ (图4-17)



- function8: compare_eq for $[rd=\{3'b111, rs == rt\}]$ (图4-18)



- 8x1 Mux(圖4-19)



Requirements

1. use universal gate only，並先在Universal_Gate(助教有給的.v檔)完成建構
 2. 依題目指示，會以此為基底 instantiate primitive gates(AND、OR、NOT...)，並完成 function*8，完成對指令解碼並執行的任務。

Design Explanation

- 用 rs, rt 以及其他 modules 實作出所要求的八種 function
 - function1:
我們在這個部分先將rt做complement並加上4'b0001之後，再與rs輸入function2(ADD)進行加法得到所求。

- function2:

我們在這個部分使用了4個full adder進行連加，相當於實作出一個4bit的ripple carry adder，而我們的output就是4bit的sum。

- function3:

我們在這個部分讓rs,rt的4個bit分別一起接入Or_gate module做or的運算(e.g. rd[0] = rs[0] or rt[0])並將output的4個bit分別接上stantiate的四個Or_gate module的output port。

- function4:

我們在這個部分讓rs,rt的4個bit分別一起接入And_gate module做and的運算(e.g. rd[0] = rs[0] and rt[0])並將output的4個bit分別接上stantiate的四個And_gate module的output port。

- function5:

我們在這個部分讓rt的4個bit依據所求(e.g. rd[3] = rt[3] = rt[3] and rt[3])分別接入4個And_gate module做and的運算，並將output的4個相對應的bit分別接上stantiate的四個And_gate module的output。

- function6:

我們在這個部分讓rs的4個bit依據所求(e.g. rd[0] = rs[3] = rs[3] and rs[3])分別接入4個And_gate module做and的運算，並將output的4個相對應的bit分別接上stantiate的四個And_gate module的output。

- function7:

在這個部分由於rd[3:1]無論如何都是3'b101，所以就依照所求分別將1'b1,1'b0接入And_gate module並得到所求(e.g. rd[3] = 1'b1 = 1'b1 and 1'b1)。

至於rs < rt的部分，我們已知(rs < rt) = rs[3]'rt[3] + x3 rs[2]'rt[2] + x3x2 rs[1]'rt[1] + x3x2x1 rs[0]' rt[0]

(x3 = rs[3]rt[3] + rs[3]'rt[3]',x2,x1以此類推)，於是我們將rs,rt的4個bit用Xnor_gate module做

xnor(equivalent)之後得到x3, x2, x1, x0之後再依據上面的式子使用And_gate, Or_gate modules做運算並得到rd[0]的值 (if rs < rt, rd[0] = 1'b1; else rd[0] = 1'b0)。

- function8:

在這個部分由於rd[3:1]無論如何都是3'b111，所以就

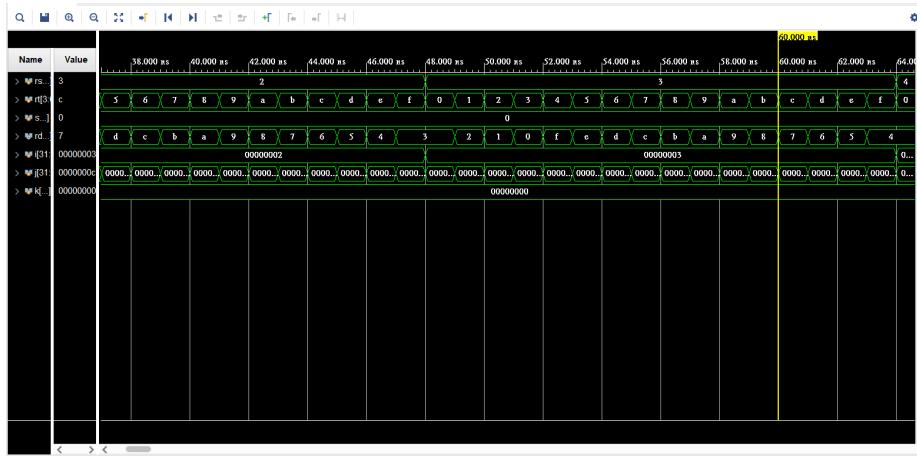
依照所求分別將 $1'b1, 1'b0$ 接入And_gate module並得到所求(e.g. $rd[3] = 1'b1 = 1'b1 \text{ and } 1'b1$)。

至於 $rs == rt$ 的部分，我們已知($rs == rt$) = $x3x2x1x0(x3 = rs[3]rt[3] + rs[3]'rt[3]')$ ，於是我們將 rs, rt 的4個bit用Xnor_gate module做xnor(equivalent)之後得到 $x3, x2, x1, x0$ 之後再依據上面的式子使用And_gate modules做運算並得到 $rd[0]$ 的值 (if $rs == rt$, $rd[0] = 1'b1$; else $rd[0] = 1'b0$)。

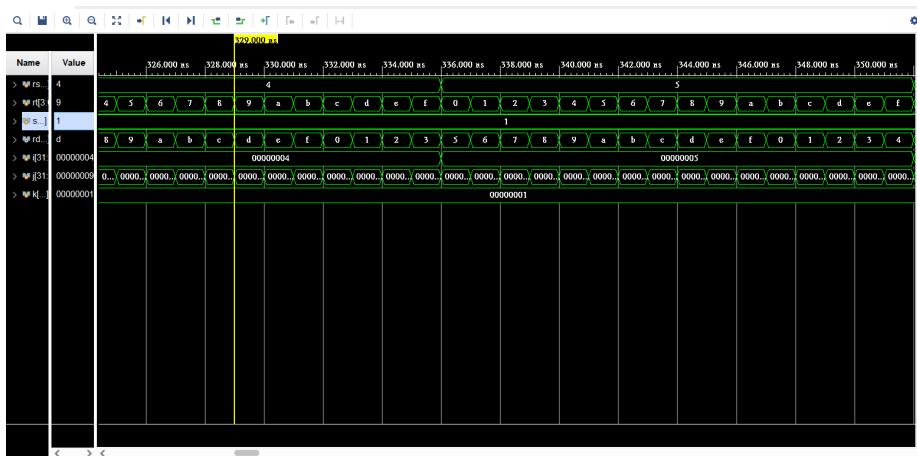
- 把 rs, rt 依照所求接入八個modules並將命名為 $tmp0 \sim tmp7$ 的4 bit wire分別接上8個modules的output port，再將 $tmp0 \sim tmp7$ 以及 sel 接入4bit 8x1 Mux來依據3bit sel中三個bit的訊號來從8個input($tmp0 \sim tmp7$)中選擇並輸出相對應的4bit訊號(例如 $sel=3'b000$ ，則輸出 $tmp0$)，最後再把 rd 接到4bit 8x1 Mux的output port。

Testbench Design & Result Explanation

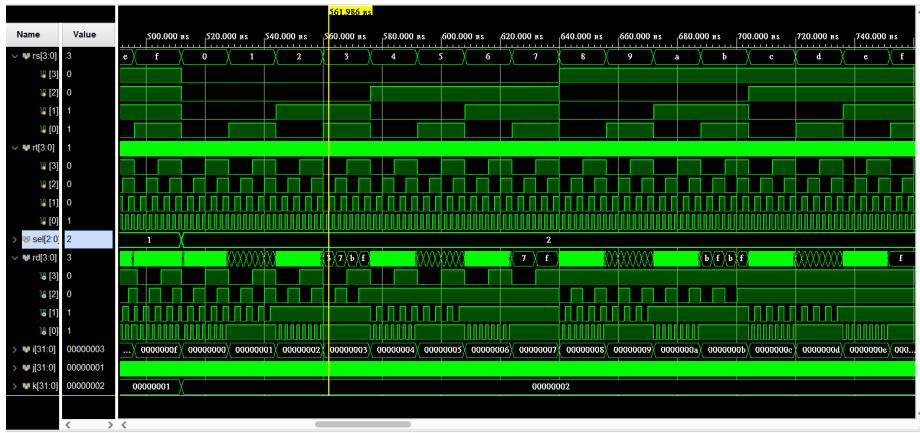
- function1: SUB 情境之呈現



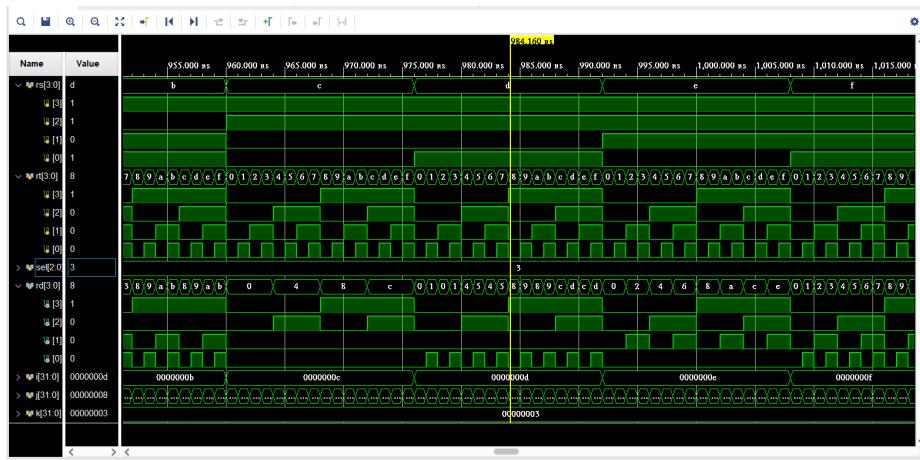
- function2: ADD 情境之呈現



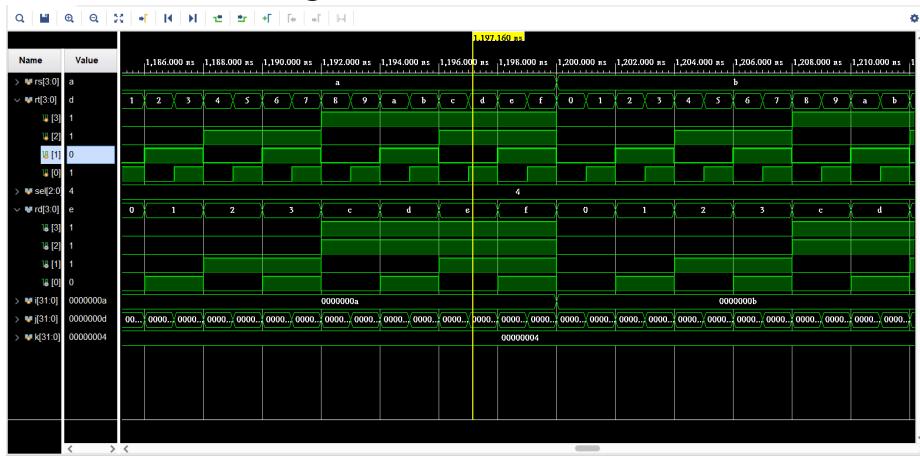
3. function3: bitwise_or 情境之呈現



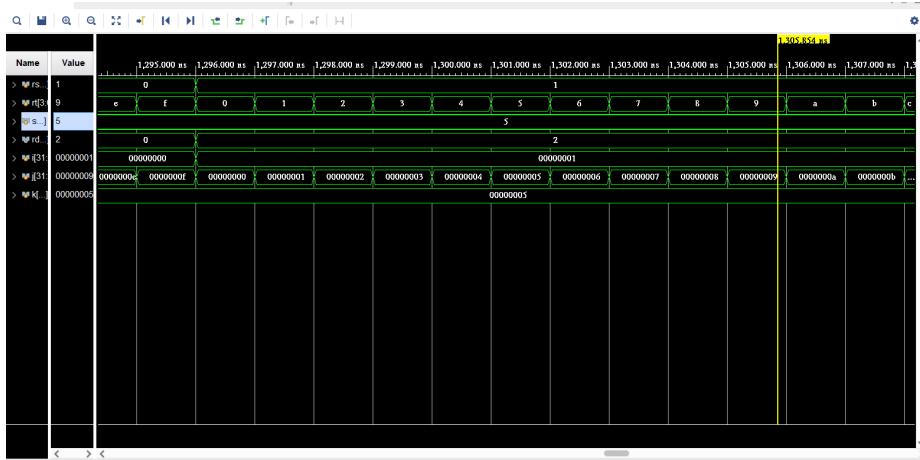
4. function4: bitwise_and 情境之呈現



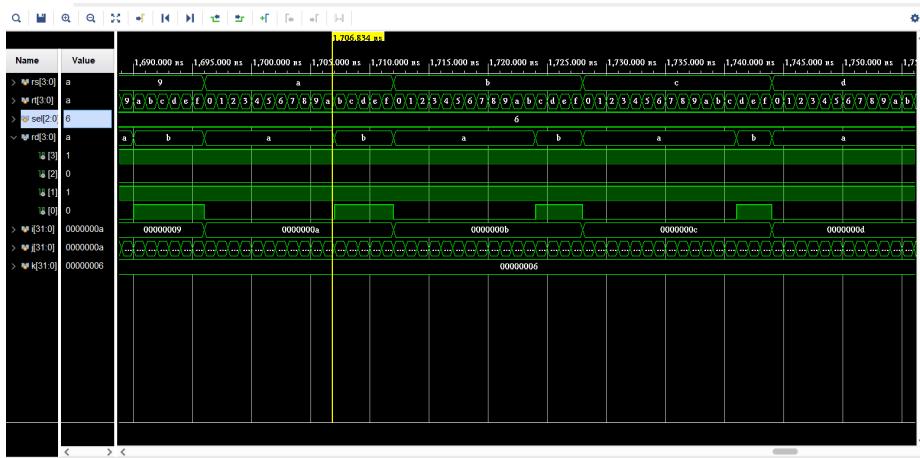
5. function5: rt_ari_right_shift 情境之呈現



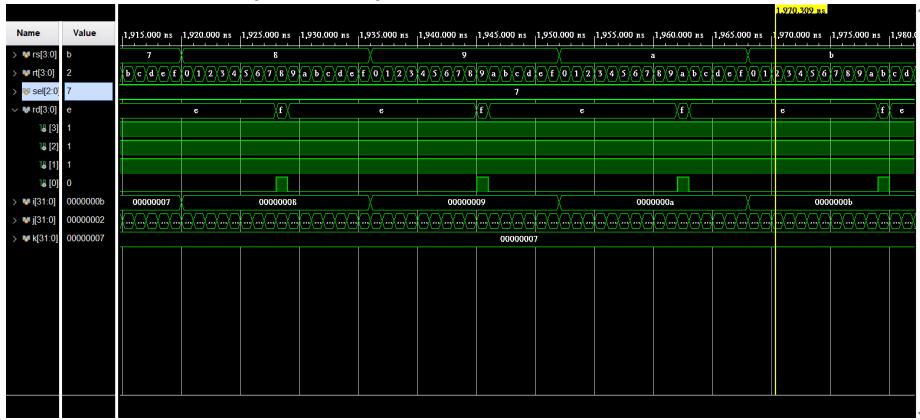
6. function6: rs_cir_left_shift 情境之呈現



7. function7: compare_lt 情境之呈現



8. function8: compare_eq 情境之呈現

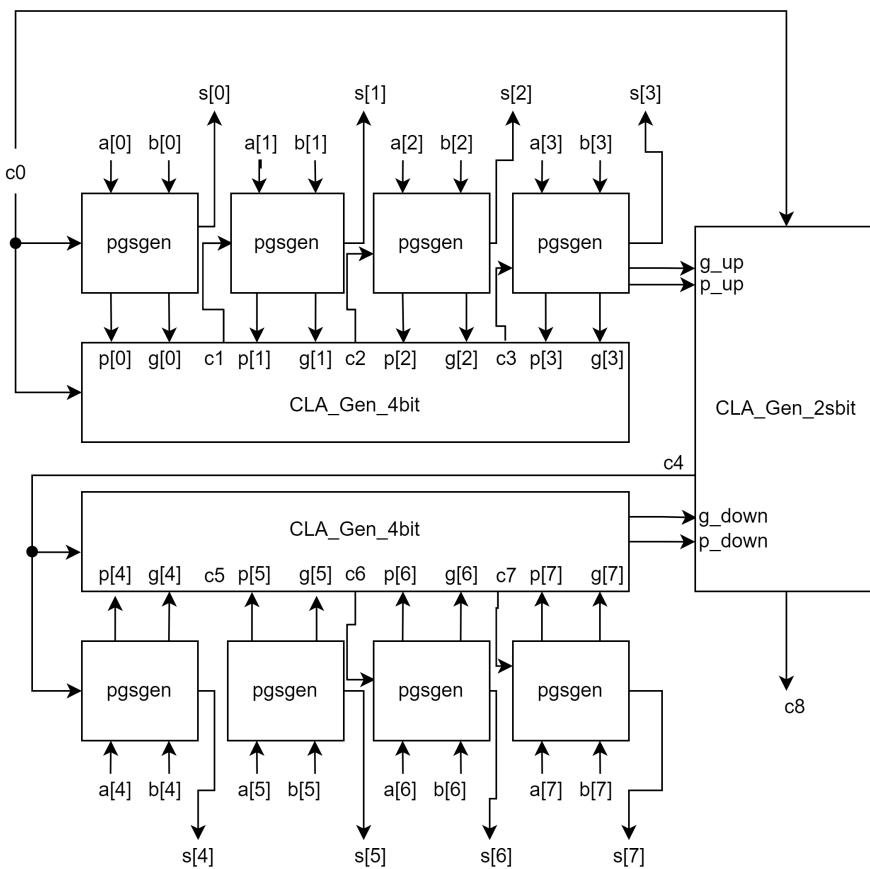


在這次的 test bench 設計中，我們使用窮舉的方式來測試所有可能產生的結果，而我們總共有 $8(\text{sel}:0\rightarrow 7)*16(\text{rs}:0\rightarrow f)*16(\text{rt}:0\rightarrow f)$ 種組合。在經過以上2048種測資分別對 8 種 function 細看進行驗算之後，可以發現輸出皆能得到相對應 4-bit rd 值，因此得以證實我們的設計是無誤的。

Advanced Question3

Drawing of the design of Verilog Advanced Question 3

- Top module: Carry_Look_Ahead_Adder(CLA) (圖5-7)



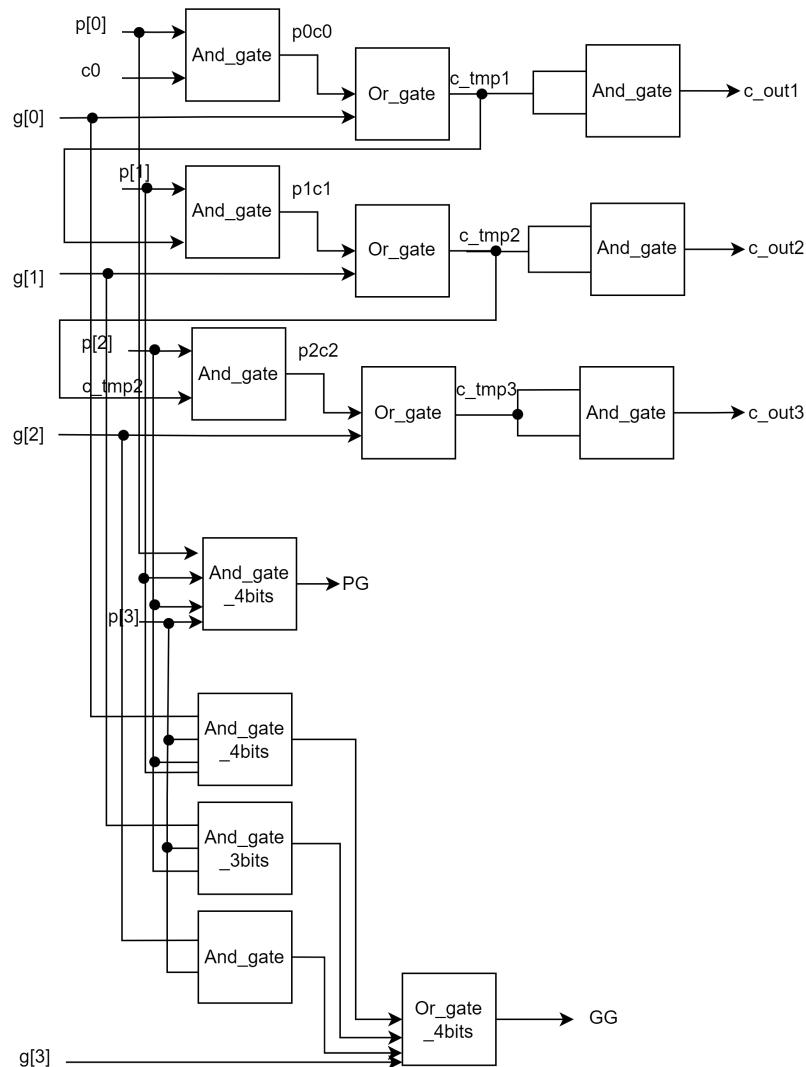
- module

我們使用到在 Q1 中以 nand 實作出的modules (如以下所列):

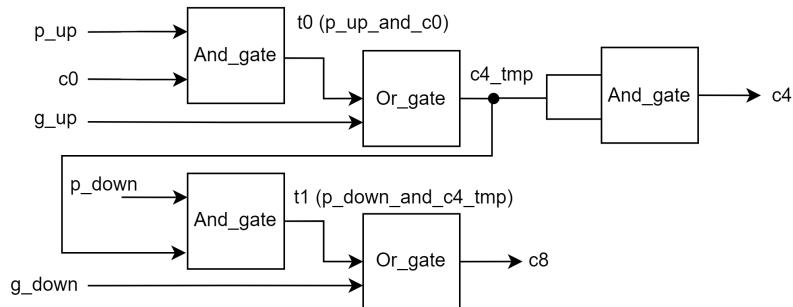
- Or_gate
- And_gate
- Xor_gate

除此之外，還加上 modules 方便使用:

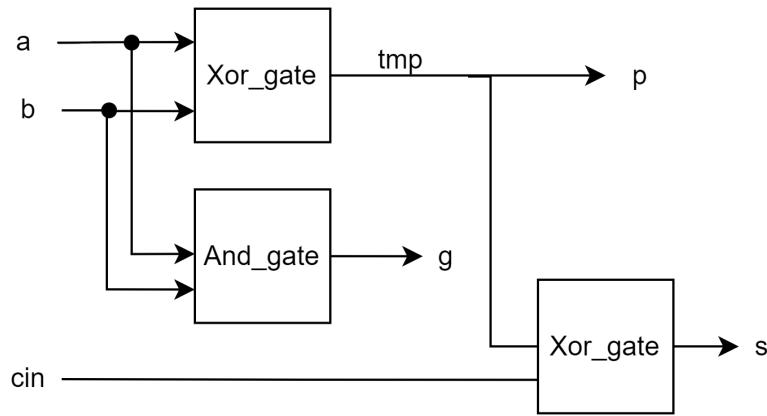
- CLA_Gen_4bit (图5-1)



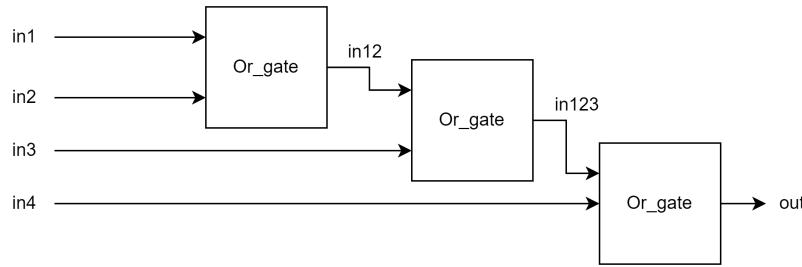
- CLA_Gen_2bit (图5-2)



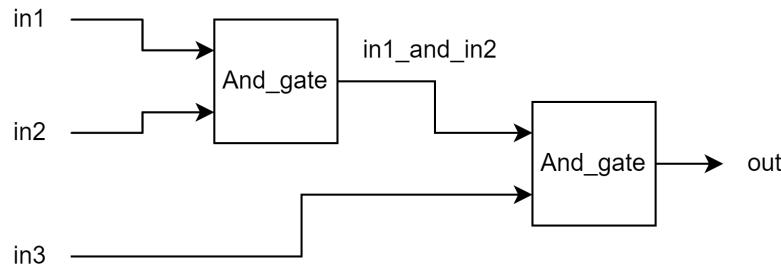
- pgsgen (圖5-3)



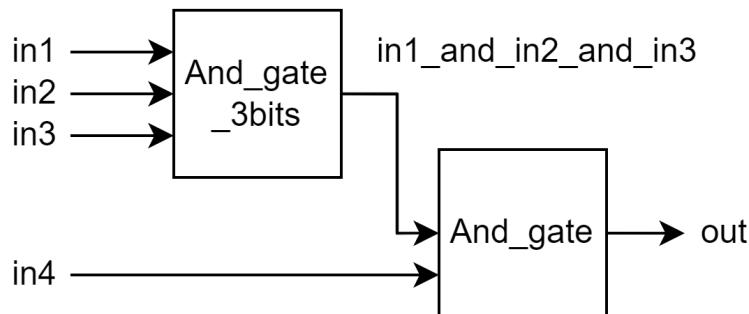
- Or_gate_4bits (圖5-4)



- And_gate_3bits (圖5-5)



- And_gate_4bits (圖5-6)



Requirements

1. use NAND gate only
2. 使用 hierarchical modules 的方式完成 CLA 設計

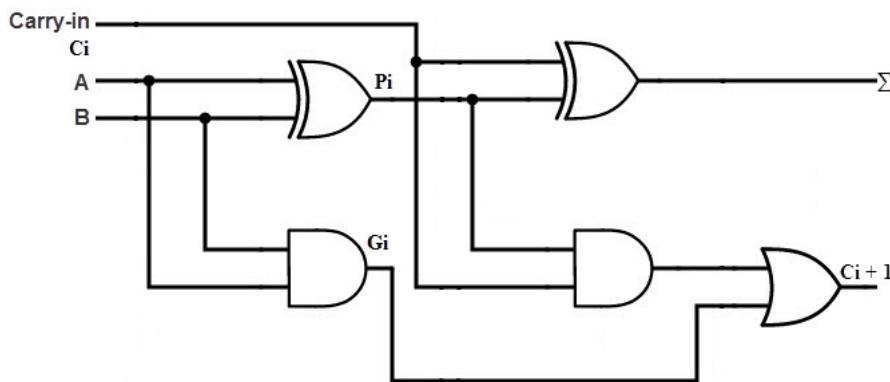
Design Explanation

1. 按照題目要求，在講述實踐步驟之前，先解釋我們對加法器的認知。超前進位加法器(CLA)是一個優化版的加法器，相較於簡潔設計的 RCA，計算複雜度相較低，因為高位運算不再需要等待低位的進位輸出信號，所有的數值在運算一開始就可以得到的數值。如此一來可避免每個 Full Adder 都要等前面一個把結果輸出才能開始做加法，如果我們能提前知道前面是否會進位，就能夠提前讓後面的 Full Adder 開始運作了，大幅加速。

[比較一下 RCA 及 CLA (4-bit)]

RCA	CLA
1 次進行 1 個位元的加法	1 次進行 4 個位元的加法
4 次加法才能得出結果	1 次加法就能得出結果
非常不具效率 (時間會依邏輯閘數量提升)	省時 (不需要等到最低兩位元加法得出的商再進行下一個位元的運算)

2. 依照投影片上的題目方法來進行實作，首先我們需要 8 個 pgsgen 完成基礎的 p、g、sum 生成，實作方法如同 basic question3 完成的 1-bit Full Adder。



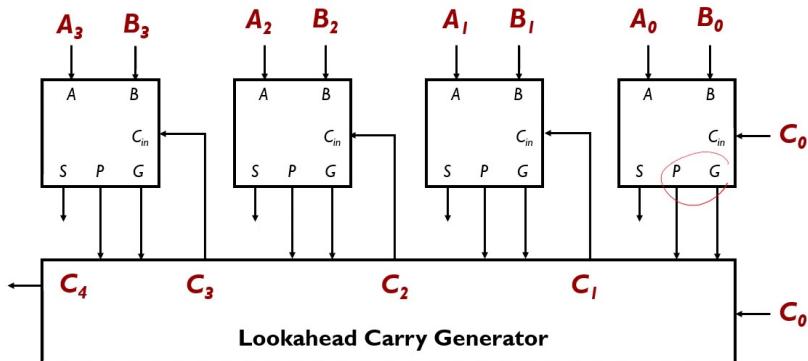
a. 按理推出以下式子

- p (propagate) = $a \oplus b$
- g (generate) = $a \cdot b$
- $s = cin \oplus (a \oplus b)$

b. 依據上面式子使用 And_gate, Or_gate modules 做出運算並得到 p, g 值輸出。

3. 再將 2 個 4-bit 的 CLA 連接成 8-bit 加法器。4-bit 的 CLA 實作方法如下：

4-bit CLA



a. 按理推出以下式子

- $c_{out1} = p[0]c0 + g[0]$
- $c_{out2} = p[1]c_{out1} + g[1] = p[1] (p[0]c0 + g[0]) + g[1] = p[1]p[0]c0 + p[1]g[0] + g[1]$
- $c_{out3} = p[2]c_{out2} + g[2] = p[2] (p[1]p[0]c0 + p[1]g[0] + g[1]) + g[2] = p[2]p[1]p[0]c0 + p[2]p[1]g[0] + p[2]g[1] + g[2]$
- $PG = p_0p_1p_2p_3$
- $GG = g_3 + g_2p_3 + g_1p_3p_2 + g_0p_3p_2p_1$

b. 依據上面式子使用 And_gate, Or_gate modules 做出運算並得到 c_{out1} , c_{out2} , c_{out3} , PG, GG 值輸出。

4. 將 2 個 4-bit 的 CLA 接入 2-bit 的 CLA 連接成完整的 8-bit 加法器。4-bit 的 CLA 實作方法如下(圖5-2)：

a. 按理推出以下式子

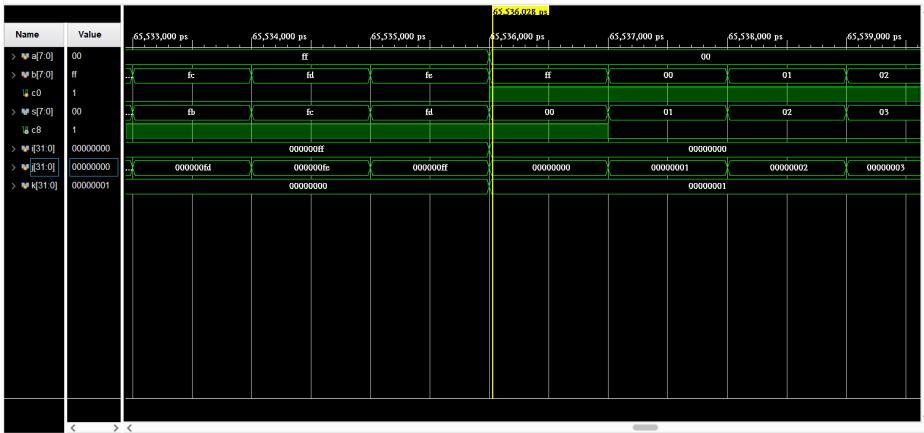
- $c4 = p_{up} \cdot c0 + g_{up}$
- $c8 = p_{down} \cdot c4 + g_{down}$

b. 依據上面式子使用 $c4$, $c8$ 值輸出。

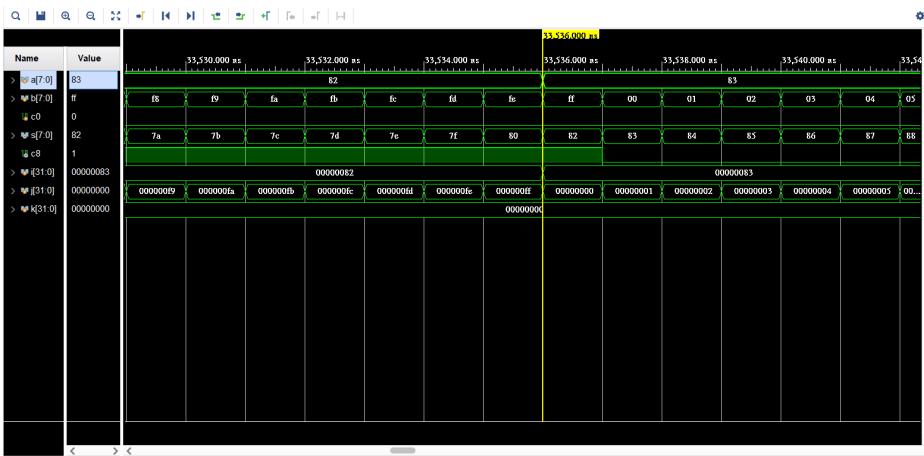
5. 以上，做出透過相較於 RCA，較快速版的 8-bit 加法器。

Testbench Design & Result Explanation

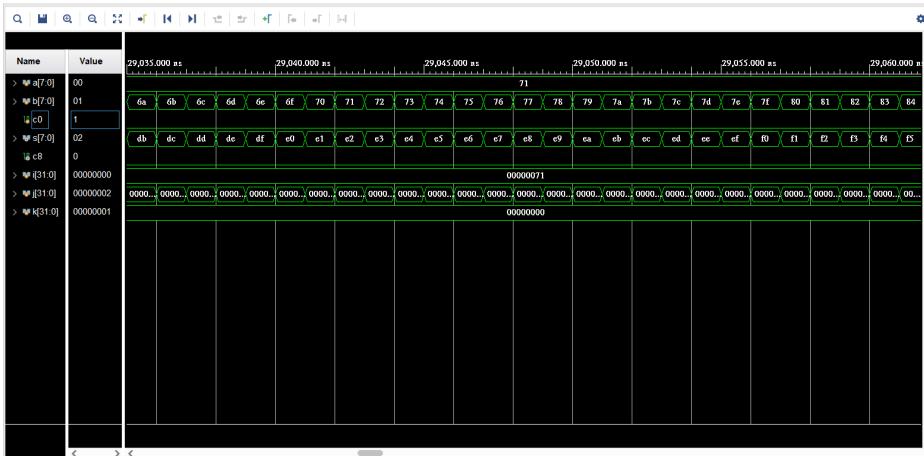
- c0: 0->1 變化 (圖5-7)



- a: 82->83 變化 (圖5-8)



- b: 6a->84 變化 (圖5-9)



在這次的 testbench 設計中，我們使用窮舉的方式來測試完所有可能產生的結果，而我們總共有 $2(c0:0\rightarrow1)*256(a:00\rightarrow ff)*256(b:00\rightarrow ff)$ 種組合。在選出幾個來運算後，發現我們的加法器也正確運算出目標值，因此驗證我們的設計是無誤的。

Advanced Question4

Drawing of the design of Verilog Advanced Question 4

- 我們使用在basic question1所實作出的modules(如以下所列):
 - Not_gate
 - Or_gate
 - And_gate
 - Xor_gate
- Majority

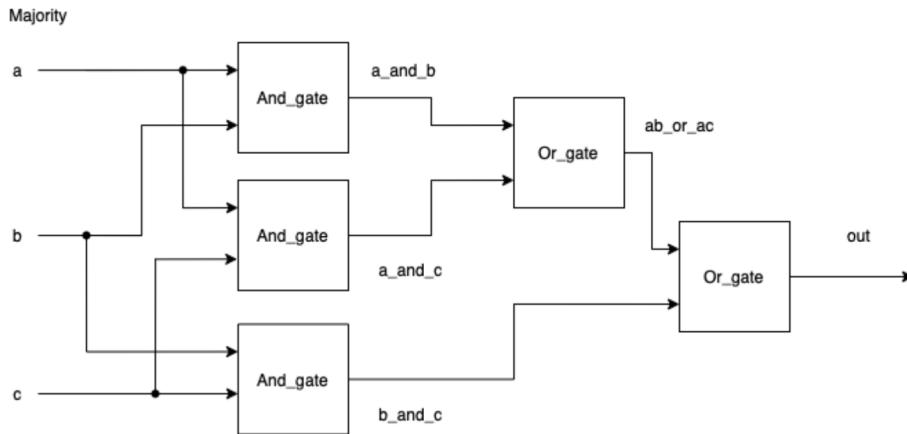


圖6-1

- Full adder

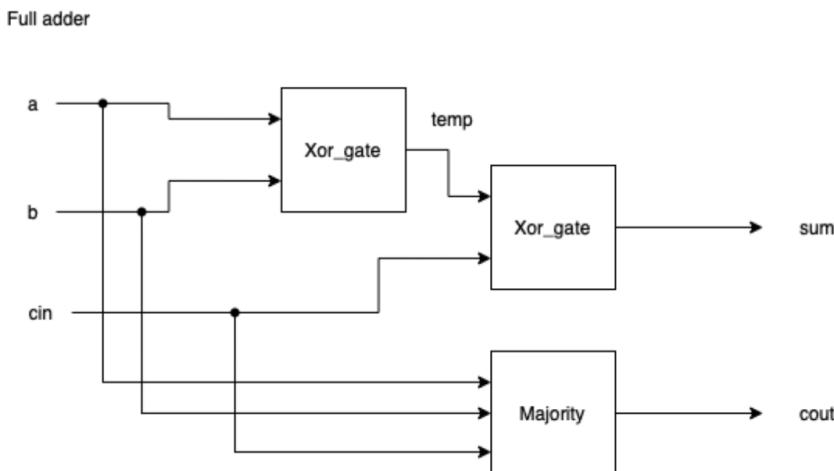


圖6-2

• Multiplexer

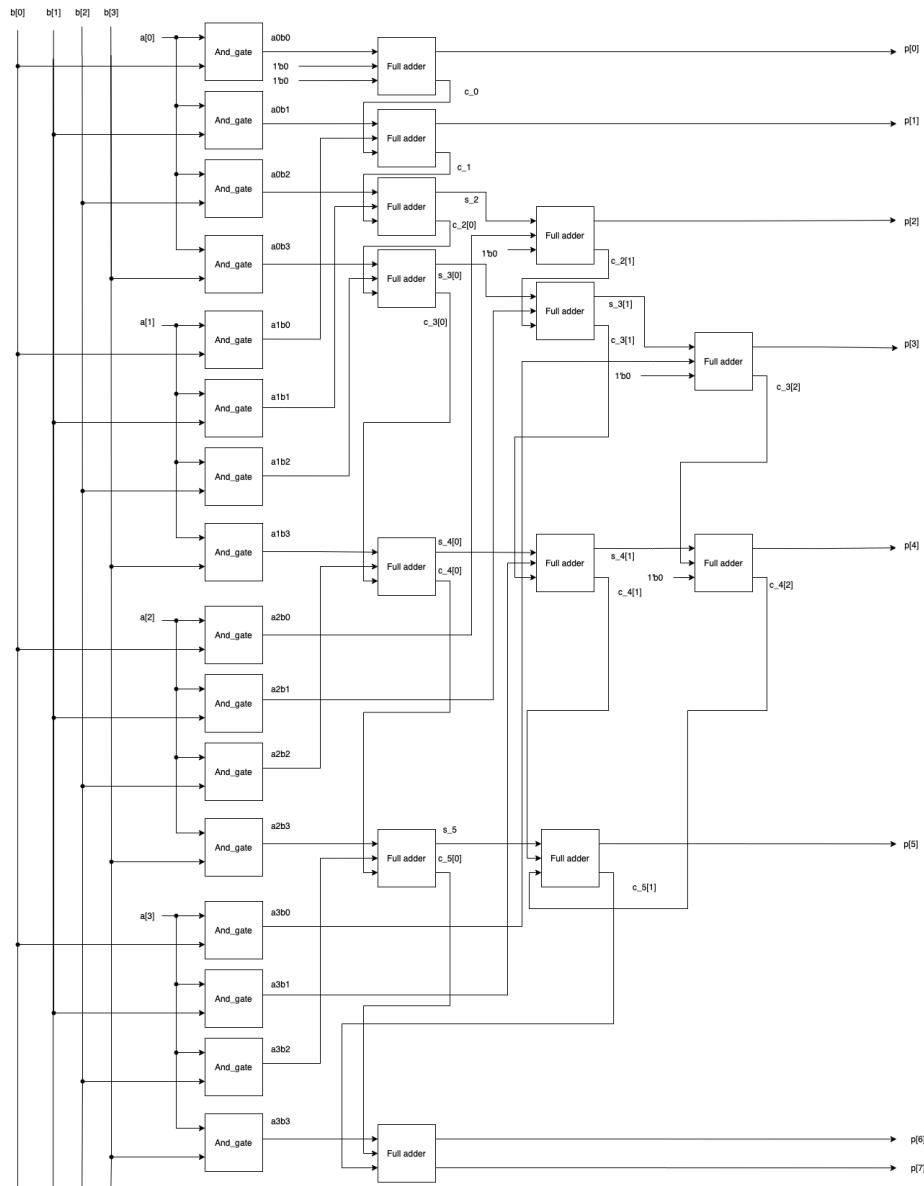


圖6-3

Requirements

1. use NAND gate only
2. 用basic question3所實作出的full adder以及half adder來設計一個乘法器

Design Explanation

1. 在本次設計中，為了方便起見所以我們一律使用full adder來實作，而在沒有進位的時候則輸入1'b0
2. 依照普通的直式乘法，我們可以看到以下結果
 - $p_0 = a_0b_0$
 - $p_1 = a_1b_0 + a_0b_1$
 - $p_2 = a_2b_0 + a_1b_1 + a_0b_2$

- $p3 = a3b0 + a2b1 + a1b2 + a0b3$
 - $p4 = a3b1 + a2b2 + a1b3$
 - $p5 = a3b2 + a2b3$
 - $p6 = a3b3$
 - $p7 = p6$ 的部分做完加法後的進位值
3. 而我們透過以上結果可以得知，有一些output是由不只三個數加起來的，而產生的進位值也不只一個，所以我們必須進行連加的動作並記錄計算每一個位數所得到的每一個進位，並在計算下一個位數的值的時候，把那些進位都加上。
4. 以上述的敘述來舉例：在進行 $p2$ 的運算時，我們已知 $p2 = a2b0 + a1b1 + a0b2 + c_1$ (來自 $p1$ 的進位)，而由於full adder一次只能對三個1bit的input進行加法，所以我們先將 $a0b2 + a1b1 + c_1$ 獲得 $\{c_2[0], s_2\}$ 之後，再將 $s_2 + a2b0 + 1'b0$ 獲得 $\{c_2[1], p[2]\}$ ，其中 $c_2[0], c_2[1]$ 為兩個進位值，將用來計算 $p3$ ，而 s_2 是第一次加法所得出的結果，然後我們要再將它與剩下沒有加完的值進行加法來完成連加的動作。
5. 如同上述的舉例所示，我們把計算第*i*個位數時所得到的進位值命名為 c_i ，而在連加的時候得到的sum則命名為 s_i
6. 而在經過12個加法器的運算之後，我們獲得了所有位數的值

Testbench Design & Result Explanation

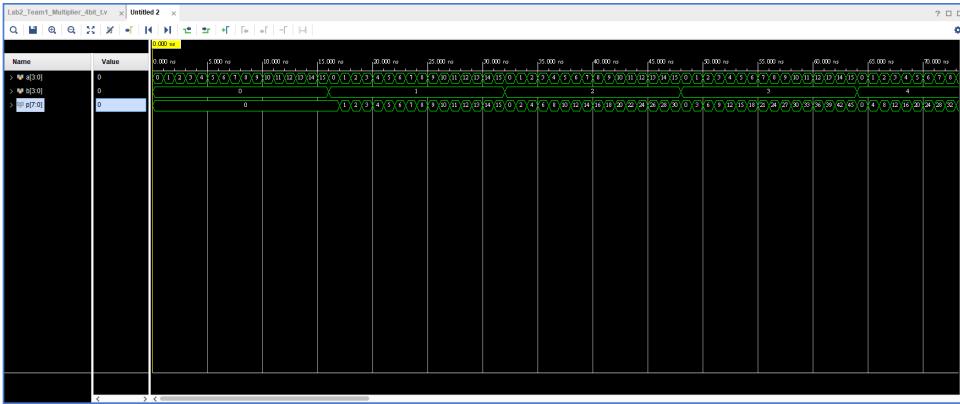


圖6-4

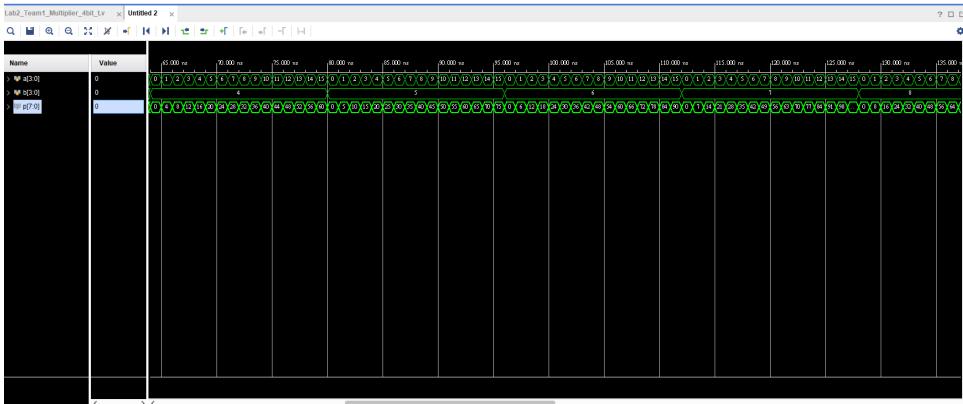


圖6-5



圖6-6

在這次的test bench的設計中，我們使用窮舉的方式來測試完所有可能產生的結果，而我們總共有 $16 \times 16 = 256$ 種組合。在經過對照之後可以發現我們的乘法器可以正確運算出 $0x0 \sim 15 \times 15$ 所得到的值，因此得以驗證我們的設計是無誤的。

Advanced Question5

Requirements

1. 設計一個測試4bit adder的testbench
2. testbench必須利用窮舉的方式將每一個測資都測到
3. 每5ns改變測資
4. error以及done初始化為1'b0

5. 在改變測資1ns之後，如果偵測到錯誤則將error升起，如果下次改變測資之後沒有發生錯誤則在改變測資1ns後將error降下來
6. 在所有測資測試完畢5ns之後，將done升起

Testbench Design & Result Explanation

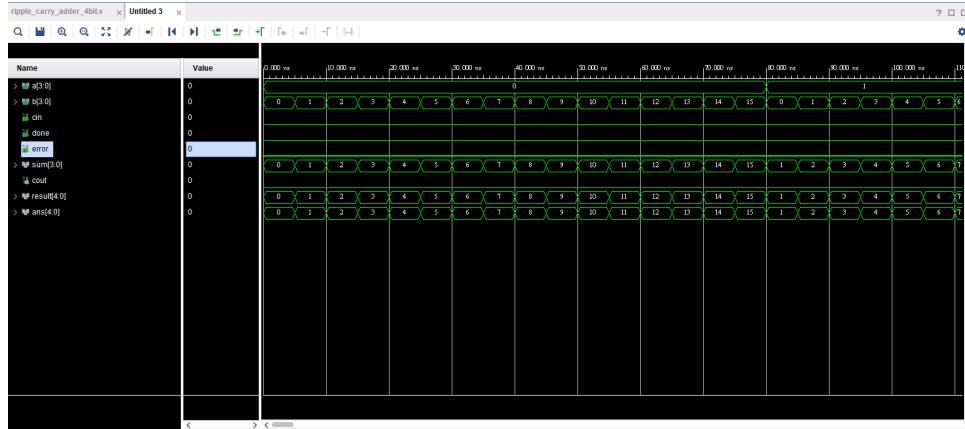


圖7-1

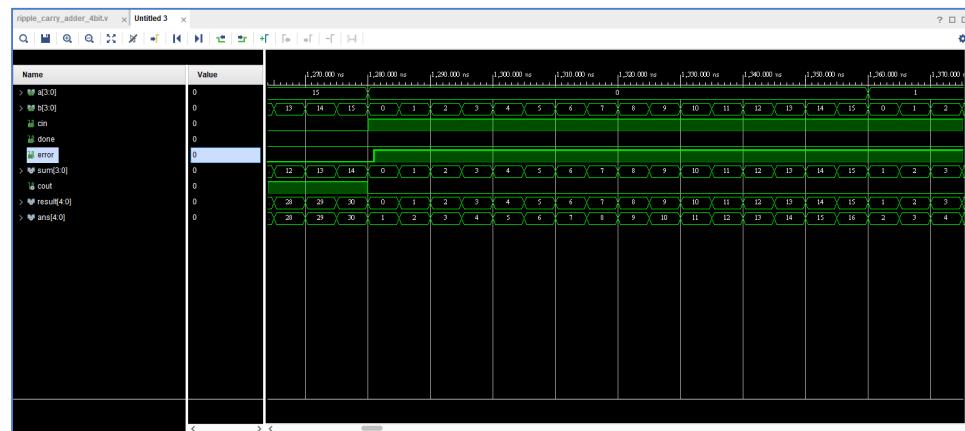


圖7-2

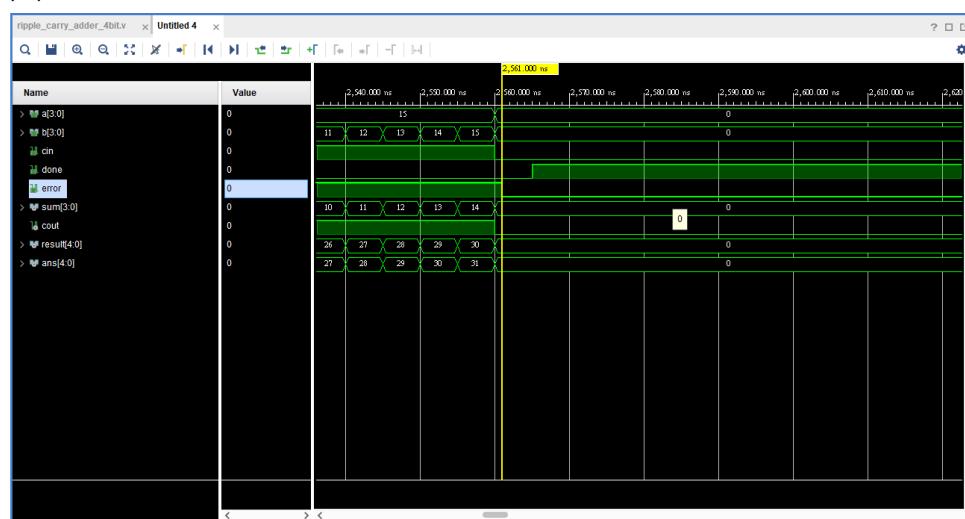
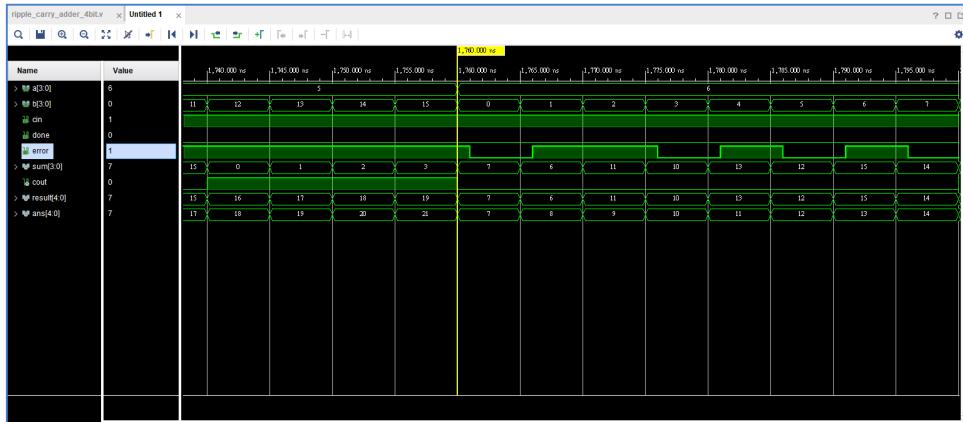


圖7-3**圖7-4**

1. 在本題的設計當中，我們使用三層repeat的迴圈來遍歷所有的情況
 - 第一層： $\text{cin} = 1'b0 \rightarrow \text{cin} = 1'b1$
 - 第二層： $a = 4'b0000 \rightarrow a = 4'b1111$
 - 第三層： $b = 4'b0000 \rightarrow b = 4'b1111$
2. 而在確認正確性的部分，我們用兩條5bit的
wire:
`result(result[4:0] = {cout, sum})`跟
`ans(ans[4:0] = a + b + cin)`來比較並驗證所設計的adder所得出的cout,sum是否為正確答案
3. 在上圖所得出的波形圖中，我們刻意將adder設計為錯的(原本該接入cin的port接入1'b0)，使得當 $\text{cin} = 1'b1$ 的時候，error就要在改變測資的1ns後升起來(如圖7-2)，直到遍歷完所有測資的1ns之後沒有error了才能在1ns後降下來(如圖7-3)
4. 而在最後遍歷完所有測資的5ns之後，done升起(如圖7-3)
5. 而由於圖7-3的結果只能測試到error在所有測資都跑完之後降下來的情況，所以我們將刻意設置錯的adder修改(計算右邊數來第二個bit的full adder在該接入來自第一個bit計算出來之後得到的進位c[0]改成接入c[1])，如此一來就有錯誤跟正確的答案穿插，也就能成功驗證error無論在遇到什麼樣的狀況都能在規定的時間內升起跟降下。

FPGA Demonstration 1

Drawing of the design of FPGA Demonstration 1

- 我們使用在advanced question2所實作出的modules(如以下所列)

- Not_gate
- Or_gate
- And_gate
- SUB
- ADD
- Bitwise_or
- Bitwise_and
- rt_ari_right_shift
- rs_cir_left_shift
- compare_lt
- compare_eq
- 8x1 mux

- Top module

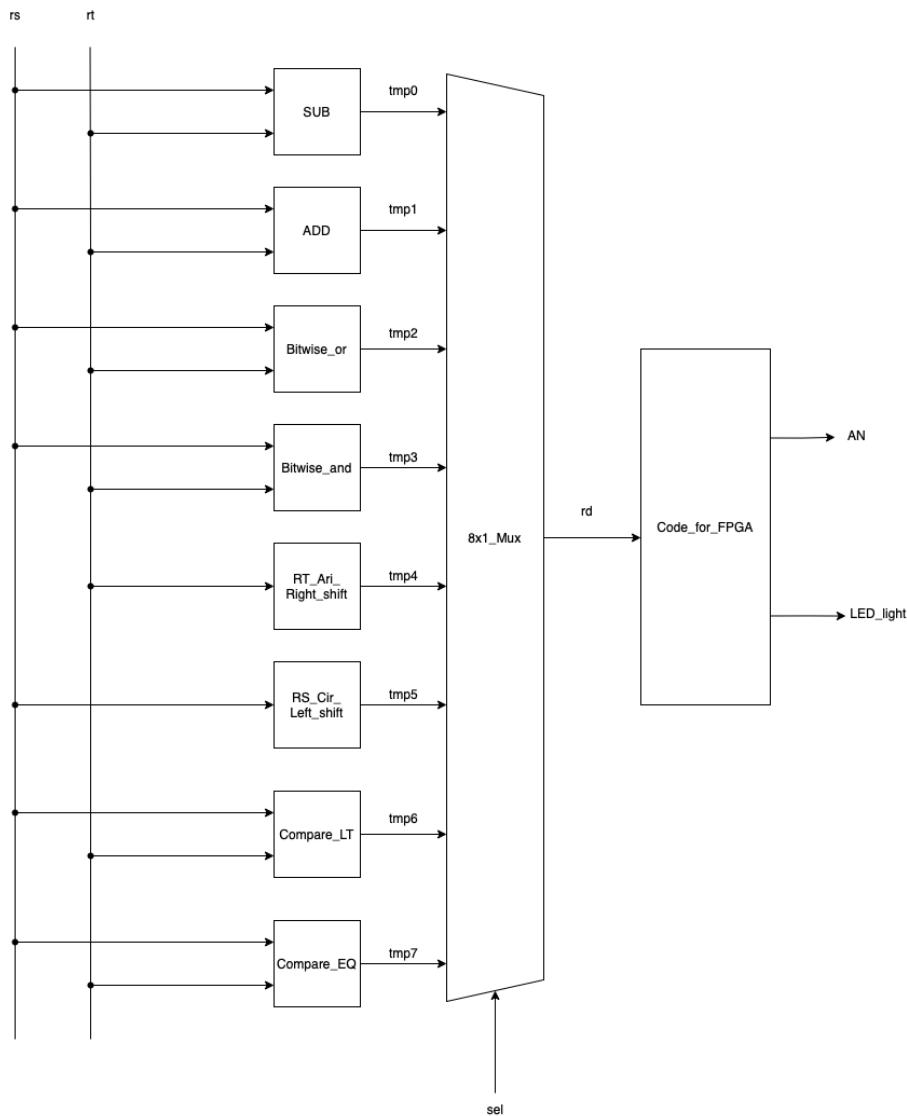


图8-1

- Code_for_FPGA module

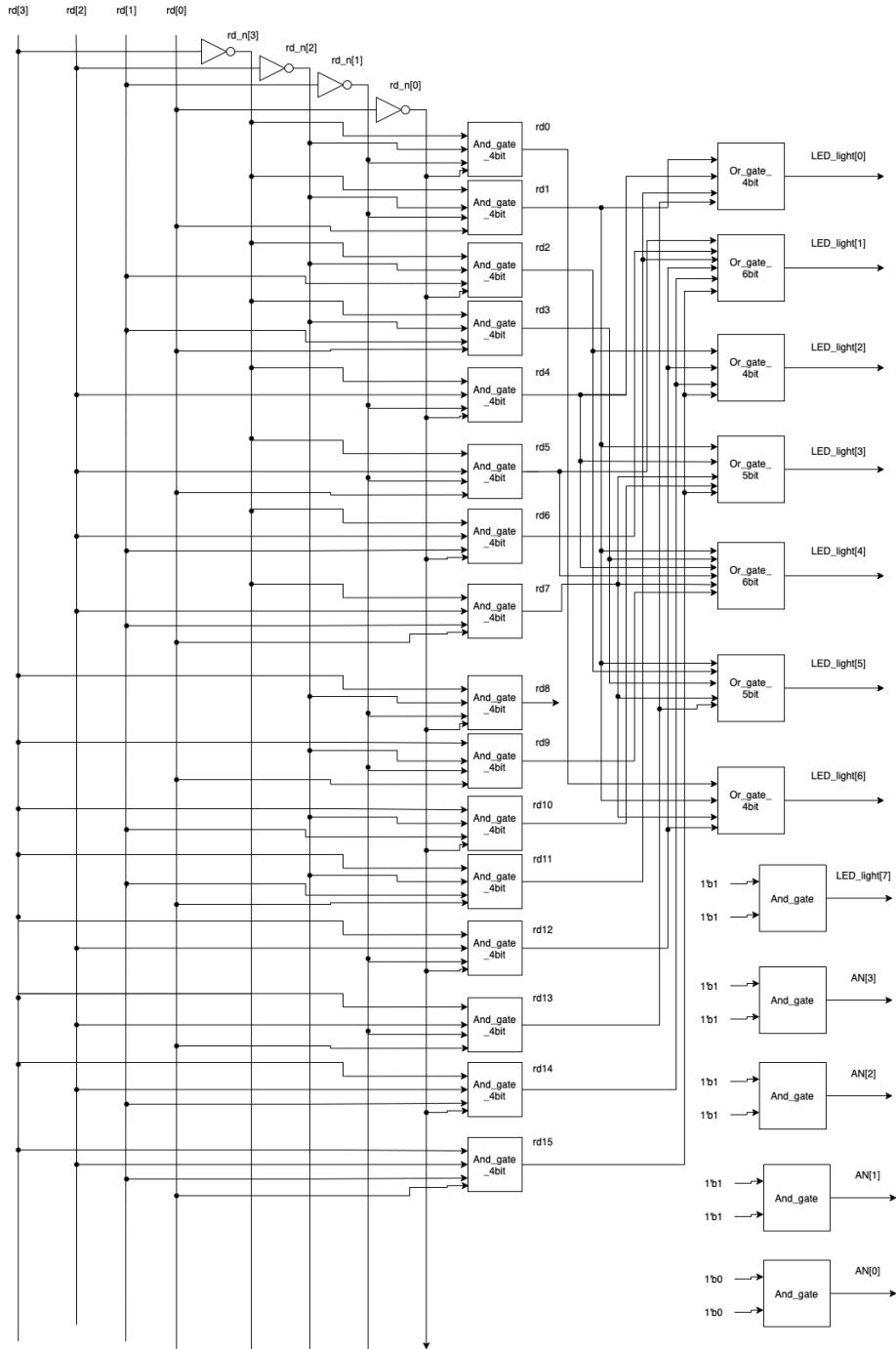


图8-2

- And_gate_4bit module

And_gate_4bit

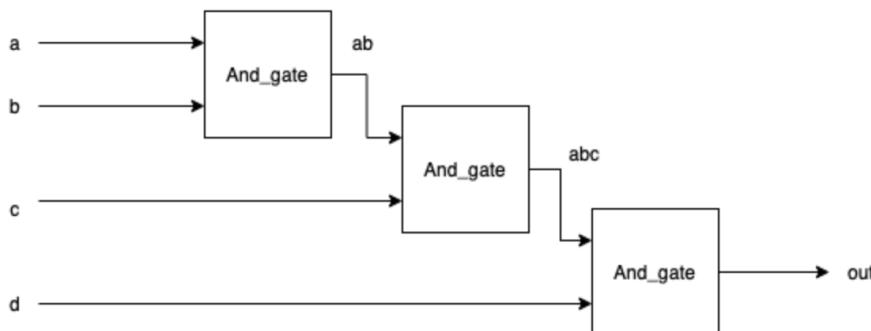


图8-3

- Or_gate_4bit module

Or_gate_4bit

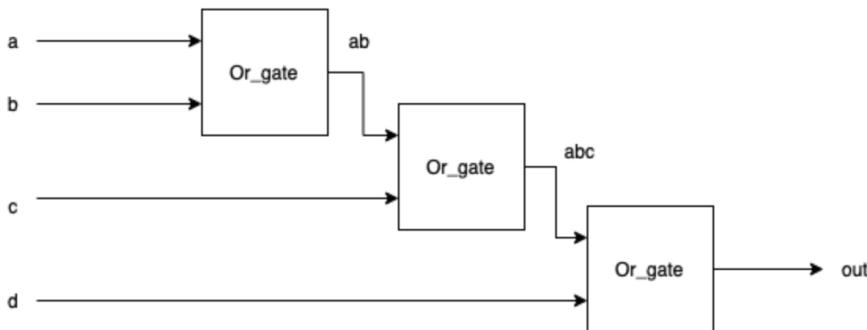


圖8-4

- Or_gate_5bit module

Or_gate_5bit

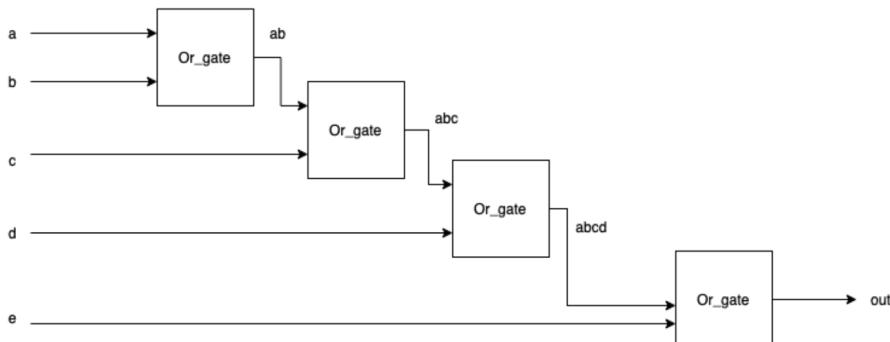


圖8-5

- Or_gate_6bit module

Or_gate_6bit

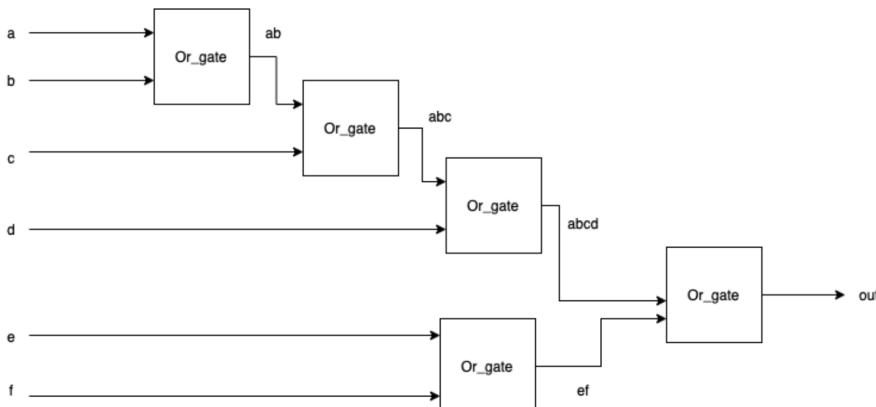


圖8-6

Design Explanation

1. 我們在LED燈的訊號的設計將符合以下真值表

rd[3]	rd[2]	rd[1]	rd[0]	DP	CG	CF	CE	CD	CC	CB	CA
0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	1
0	0	1	0	1	0	1	0	0	1	0	0
0	0	1	1	1	0	1	1	0	0	0	0
0	1	0	0	1	0	0	1	1	0	0	1
0	1	0	1	1	0	0	1	0	0	1	0
0	1	1	0	1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1	1	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0	1	0	0	0
1	0	1	1	1	0	0	0	0	0	1	1
1	1	0	0	1	1	0	0	0	0	1	0
1	1	0	1	1	0	1	0	0	0	0	1
1	1	1	0	1	0	0	0	0	1	1	0
1	1	1	1	1	0	0	0	1	1	1	0

2. 由上述真值表，我們可以得到以下結果

- $CA = m_1 + m_4 + m_{11} + m_{13}$
- $CB = m_5 + m_6 + m_{11} + m_{12} + m_{14} + m_{15}$
- $CC = m_2 + m_{12} + m_{14} + m_{15}$
- $CD = m_1 + m_4 + m_7 + m_{10} + m_{15}$
- $CE = m_1 + m_3 + m_4 + m_5 + m_7 + m_9$
- $CF = m_1 + m_2 + m_3 + m_7 + m_{13}$
- $CG = m_0 + m_1 + m_7 + m_{12}$
- $DP = 1$

3. 在本題的設計中，我們將advanced question2做了以下修改

- 將原本作為output的rd改為wire
- 新增[3:0]AN, [7:0]LED_light作為我們的output

- 新增 Code_for_fpga module來處理AN, LED_light的訊號
4. 此外，我們繼續沿用了第二題的and_gate以及or gate來進行設計，使得其餘的設計仍舊符合gate-level的要求。
 5. 我們將advanced question2的設計中的rd接入Code_for_fpga module的input port，並將AN, LED_light接到output port。
 6. 在AN的部分，由於我們只有要讓最右邊的digit亮起，所以在AN[3:1]三個bit的部分，我們用三個And_gate modules並輸入1'b1, 1'b1作為input，將AN[3], AN[2], AN[1]分別作為3個modules的output來使得這三個bit的值為1。而在AN[0]的部分，我們則輸入1'b0, 1'b0來使得輸出的值為0
 7. 而在LED_light的部分，我們額外設置了16條wire來表示現在rd的值，例如現在rd = 4'b1111，則rd15 = 1'b1，rd0~rd14 = 1'b0。並依據第二點中依照真值表所得到的結果來進行Or的運算，其中LED_light[0]的訊號代表CA的訊號，LED_light[1]的訊號代表CB，以此類推。相同的，rd1代表m1, rd2代表m2...rd15代表m15。

I/O pin assignment

Input	package pin
sel[0]	v17
sel[1]	v16
sel[2]	w16
rs[0]	w17
rs[1]	w15
rs[2]	v15
rs[3]	w14
rt[0]	w13
rt[1]	v2
rt[2]	t3
rt[3]	t2

output	package pin
AN[0]	u2
AN[1]	u4
AN[2]	v4
AN[3]	w4
LED_light[0]	w7
LED_light[1]	w6
LED_light[2]	u8
LED_light[3]	v8
LED_light[4]	u5
LED_light[5]	v5
LED_light[6]	u7
LED_light[7]	v7

Contribution

朱季葳：basic question1, 3;advanced question4,5;FPGA demonstration;advanced question2部分report
施泳瑜：advanced question 1 , 2, 3

What we have learned from Lab2

1. 我們在這次的lab中學會了不同種adders的區別以及用 gate-level 實作他們的方式。
2. 如何以gate-level的形式實作乘法器
3. 如何以gate-level的形式比較兩個input signal所代表的數值大小
4. 如何有效利用迴圈使得testbench得以窮舉完所有可能出現的input pattern
5. 如何讓output的結果對應到七算顯示器上