Lab2 Report

Team25

組長：蔡登瑞、組員：蔡政諺

Contents

1. Binary to Grey

這題寫了兩種 code，一開始我們想到的是先將 din 接到一個 3 to 8 的 Decoder，轉換成 one-hot 後再去用 OR gate 做出我們要的 dout。可是這樣一來，會使用到很多很多的 Logic Gate。後來我們想到可以運用 K-map，減少了很多的 Logic Gate。

Din[1:0]

Dout[3]	00	01	11	10
Din[3:2] 00	0	0	0	0
Din[3:2] 01	0	0	0	0
Din[3:2] 11	1	1	1	1
Din[3:2] 10	1	1	1	1

Dout[3] = Din[3]

Din[1:0]

Dout[2]	00	01	11	10
Din[3:2] 00	0	0	0	0
Din[3:2] 01	1	1	1	1
Din[3:2] 11	0	0	0	0
Din[3:2] 10	1	1	1	1

Dout[2] = Din[3]⊕Din[2]

Din[1:0]

Dout[1]	00	01	11	10
Din[3:2] 00	0	0	1	1
Din[3:2] 01	1	1	0	0
Din[3:2] 11	1	1	0	0
Din[3:2] 10	0	0	1	1

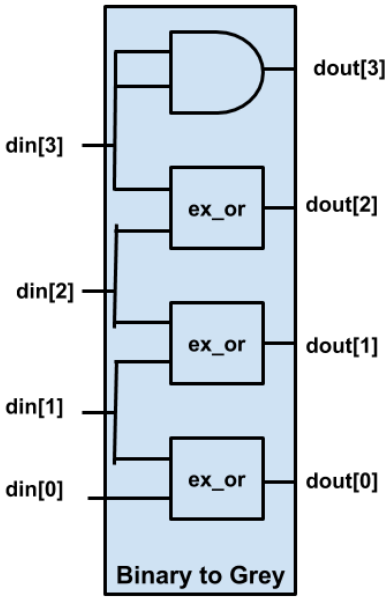
Dout[1] = Din[2]⊕Din[1]

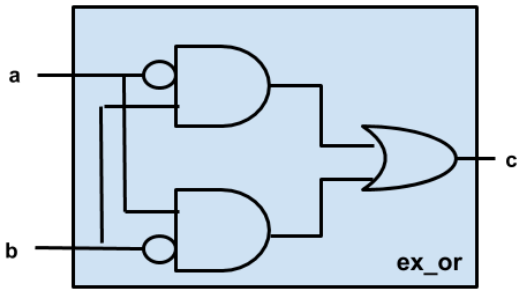
Din[1:0]

Dout[0]	00	01	11	10
Din[3:2] 00	0	1	0	1
Din[3:2] 01	0	1	0	1
Din[3:2] 11	0	1	0	1
Din[3:2] 10	0	1	0	1

Dout[0] = Din[1]⊕Din[0]

- **Gate-level diagram**





2. Multiplier

在寫這題的時候，我們有想過就直接把四個 bit 都相乘完後，再去做 Full Adder。可是這樣一來，就必須需要很多條 wire 把一個一個的 Full Adder 接起來，很容易就會把 wire 接錯。因此，我們採用另一種方法，將 1 個 4bit 相乘轉換成 4 個 2bit 相乘。將大的 Multiplier 分解成小的 Multiplier，一個大事化小，小事化無的概念。

然而，第二種方法最重要的就是把 4 個 2bit 相乘之後的結果加起來。

		Tmp_p1[3:2]	**Tmp_p1[1:0]**
	Tmp_p2[3:2]	Tmp_p2[1:0]	
	Tmp_p3[3:2]	Tmp_p3[1:0]	
Tmp_p4[3:2]	Tmp_p4[1:0]		
P[7:6]	**P[5:4]**	**P[3:2]**	**P[1;0]**

其中，都是利用 4bit 的 Full Adder 進行相加，最需要注意的地方是 tmp_p2 跟 tmp_p3 的地方，這兩個相加後有可能會變成 5bit，所以在處理相加的時候，必須用一個 5bit 的 wire 來接{cout,sum}。

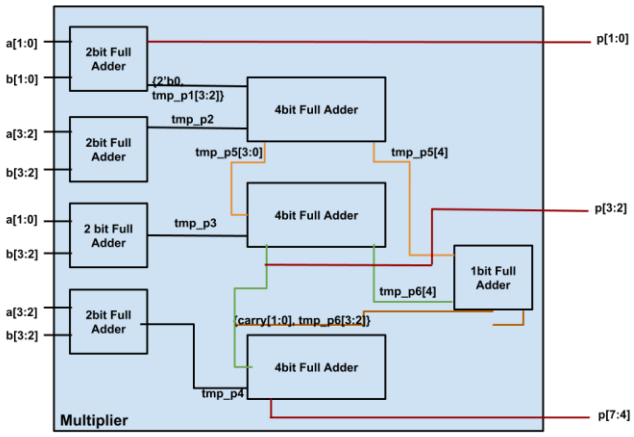
總共會用到 4 次的 2bit Multiplier、3 次的 4bit Full Adder

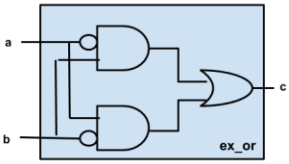
、1 次的 1bit Full Adder。

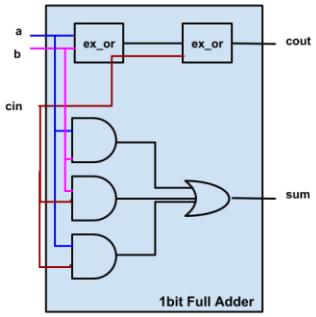
- a[1:0] x b[1:0] = { tmp_p1[3:2], **p[1:0]** }
- a[3:2] x b[1:0] = tmp_p2
- a[1:0] x b[3:2] = tmp_p3
- a[3:2] x b[3:2] = tmp_p4

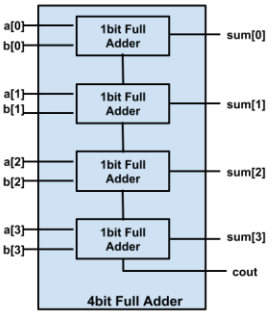
- { 2'b0, tmp_p1[3:2] } + tmp_p2 = tmp_p5[4:0]
- tmp_p5[3:0] + tmp_p3 = { tmp_p6[4:2], **p[3:2]** }
- tmp_p5[4] + tmp_p6[4] = carry[1:0]
- { carry[1:0], tmp_p6[3:2] } + tmp_p4 = **p[7:4]**

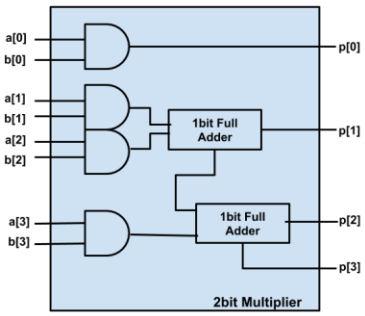
- **Gate-level diagram**











3. Carry Look Ahead Adder

在 Lab1 時我們寫過 Ripple Carry Adder，以進行 4 位數的二進位加法運算。但 Ripple Carry Adder 必須等待低位元的 Full Adder 將其 cout 運算出來後，才能輸入下一位元的 Full Adder 的 cin，依次向前傳遞。這種類型的加法器若需進行較大位數的運算，將會造成極大的延遲。

相較之下，Carry Look Ahead Adder(以下簡稱 CLA)就解決了這種問題。CLA 中每一位元的 Full Adder 都在 input 端分別獨立算出各自的 carry in，因此並沒有 Full Adder 之間需要依次等待的問題。

然而要如何分別算出各個位數的 carry in？首先我們定義兩個名詞：propagate(P)和 generate(G)。$P_i=A_i \oplus B_i$，$G_i=A_iB_i$。又 carry in $C_{i+1}=G_i \vee P_iC_i$，我們就可以藉由遞迴運算將 c1、c2、c3、c4(cout)運算出來。運算方式如下：

$$C_0=cin$$

$$C_1=G_0 \vee P_0C_0$$

$$C_2=G_1 \vee P_1C_1=G_1 \vee P_1(G_0 \vee P_0C_0)=G_1 \vee P_1G_0 \vee P_1P_0C_0$$

$$C_3=G_2 \vee P_2C_2=G_2 \vee P_2(G_1 \vee P_1C_1)=G_2 \vee P_2G_1 \vee P_2P_1C_1$$

$$=G_2 \vee P_2G_1 \vee P_2P_1(G_0 \vee P_0C_0)$$

$$=G_2 \vee P_2G_1 \vee P_2P_1G_0 \vee P_2P_1P_0C_0$$

$$C_4=G_3 \vee P_3C_3=G_3 \vee P_3(G_2 \vee P_2C_2)=G_3 \vee P_3G_2 \vee P_3P_2C_2$$

$$=G_3 \vee P_3G_2 \vee P_3P_2(G_1 \vee P_1C_1)=G_3 \vee P_3G_2 \vee P_3P_2G_1 \vee P_3P_2P_1C_1$$

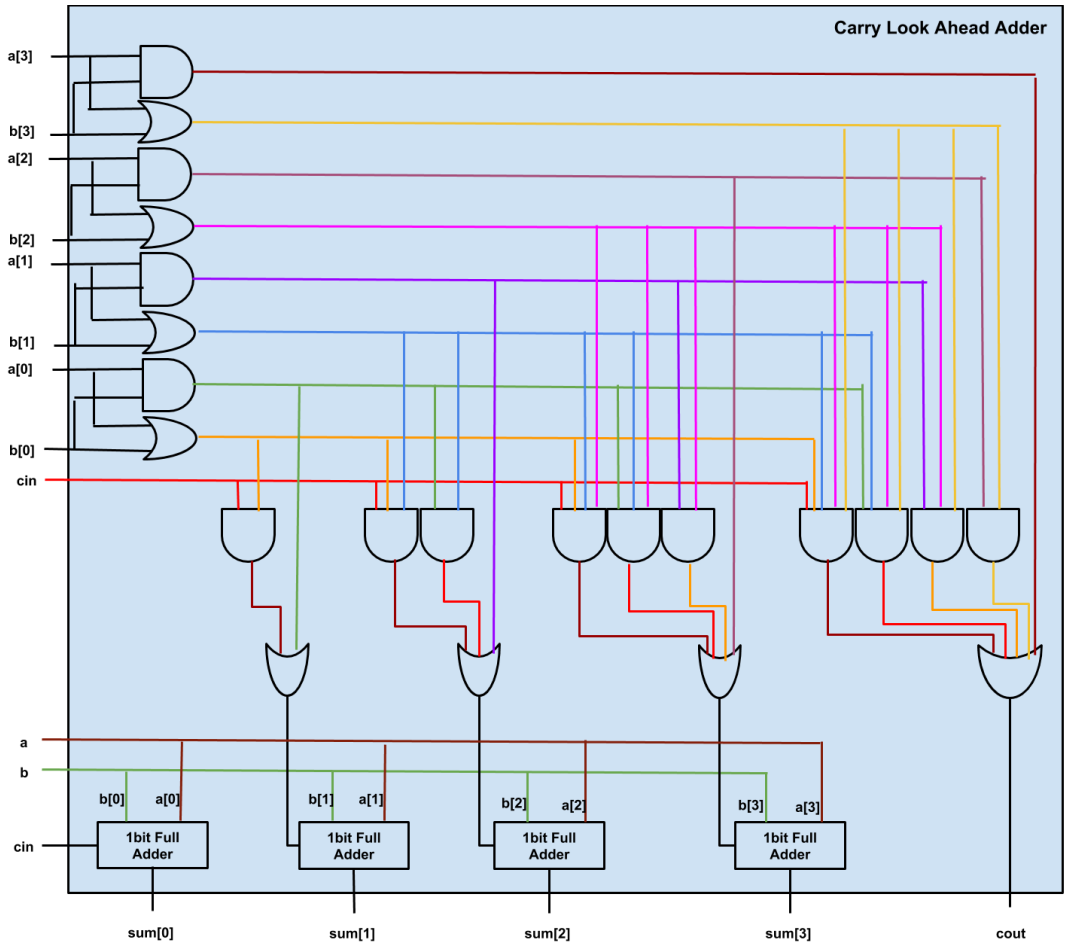
$$=G_3 \vee P_3G_2 \vee P_3P_2G_1 \vee P_3P_2P_1(G_0 \vee P_0C_0)$$

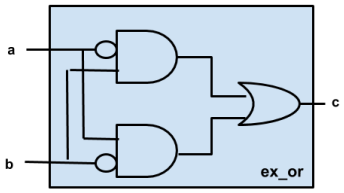
$$=G_3 \vee P_3G_2 \vee P_3P_2G_1 \vee P_3P_2P_1G_0 \vee P_3P_2P_1P_0C_0$$

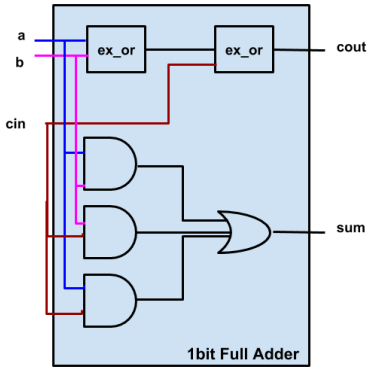
c1、c2、c3、c4(cout)都可以藉 input 的 A、B、cin 直接計算出，而不必通過任何 Full Adder。如此一來就可以減少 delay，進行平行的加法運算。

雖然 CLA 在運算速度上優於 Ripple Carry Adder，但製作同樣位數的加法器時，它相對需要更多的資源。在體積與成本方面，CLA 可能略遜於 Ripple Carry Adder。

- **Gate-level diagram**







4. Decode and Execute

首先寫一個 3 to 8 decoder，將 3-bit 的 op_code 轉成 8-bit 的 one-hot 編碼(這裡命名為 mux)。接著分別寫出 ADD、SUB、INC、NOR、NAND、RS DIV 4、RS MUL 2、MUL 的 module，與 mux 的每一個 bit 依序(低位至高位)接到 AND gate，以達到 multiplexer 的效果。

- ADD(Rd = Rs + Rt)
  先寫出 1-bit Full Adder，再以 Ripple Carry Adder 的概念接出 4-bit 的 Adder。最後再寫一個命名為 Add 的 module，省略 4-bit Adder 中 cout 這個 output。
- SUB(Rd = Rs - Rt)
  先對 rt 做 two's complement(用 NOT gate invert 之後用 4-bit Adder 加 1'b1)，再用 4-bit Adder 將 rs 與 rt 的 two's complement 相加(rs - rt = rs + 2's complement of rt)。
- INC(Rd = Rs + 1'b1)
  使用 4-bit Adder 將 rs 與 1'b1 相加。
- Bitwise NOR(Rd = Rs NOR Rt)
  將 rs、rt 的各個位數分別接到 NOR gate。
- Bitwise NAND(Rd = Rs NAND Rt)
  將 rs、rt 的各個位數分別接到 NAND gate。
- RS DIV 4(Rd = Rs >> 2)
  即向右 shift 兩位，詳見程式碼。概念如下：

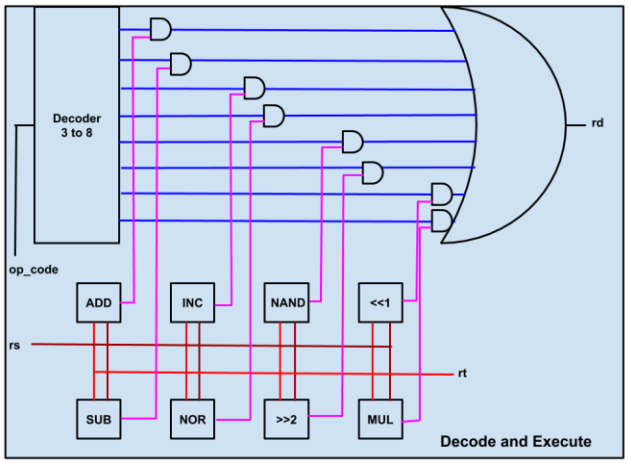
```
Rd[0] = Rs[2];
Rd[1] = Rs[3];
Rd[2] = 1'b0;
Rd[3] = 1'b0;
```

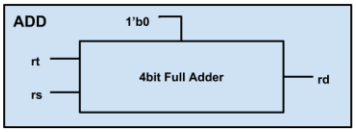
- RS MUL 2(Rd = Rs << 1)
  即向左 shift 一位，詳見程式碼。概念如下：

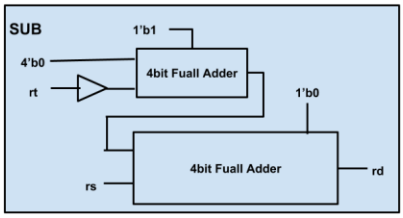
```
Rd[0] = 1'b0;
Rd[1] = Rs[0];
Rd[2] = Rs[1];
Rd[3] = Rs[2];
```

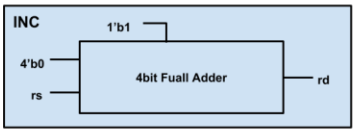
- MUL(Rd = Rs * Rt)
  直接沿用第二題的 code。

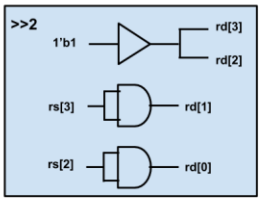
- **Gate-level diagram**

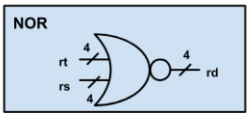


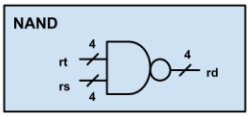


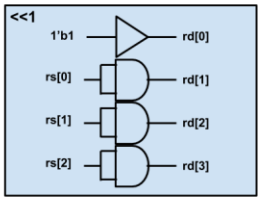


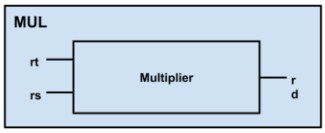












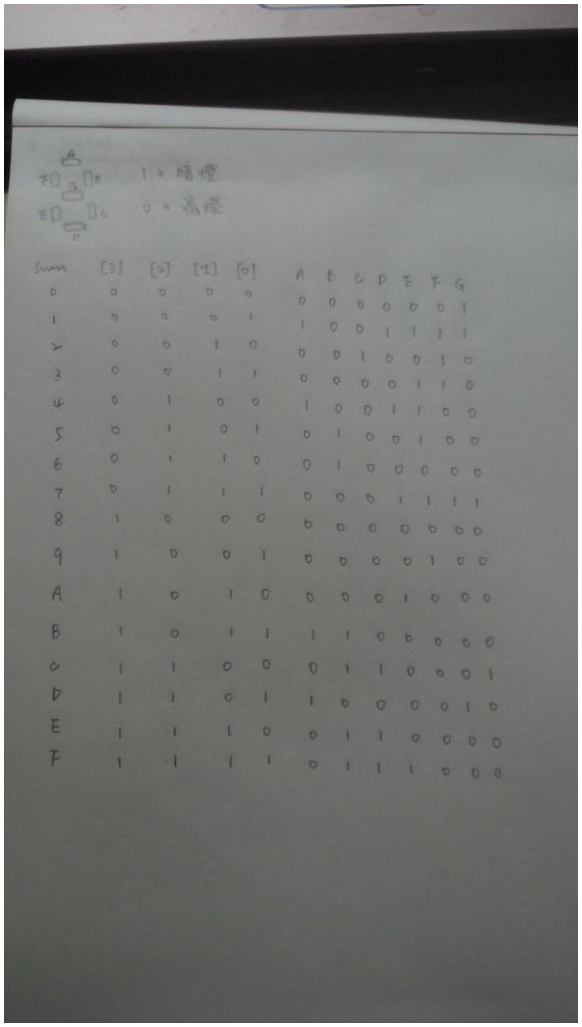
5. Carry Look Ahead Adder_FPGA

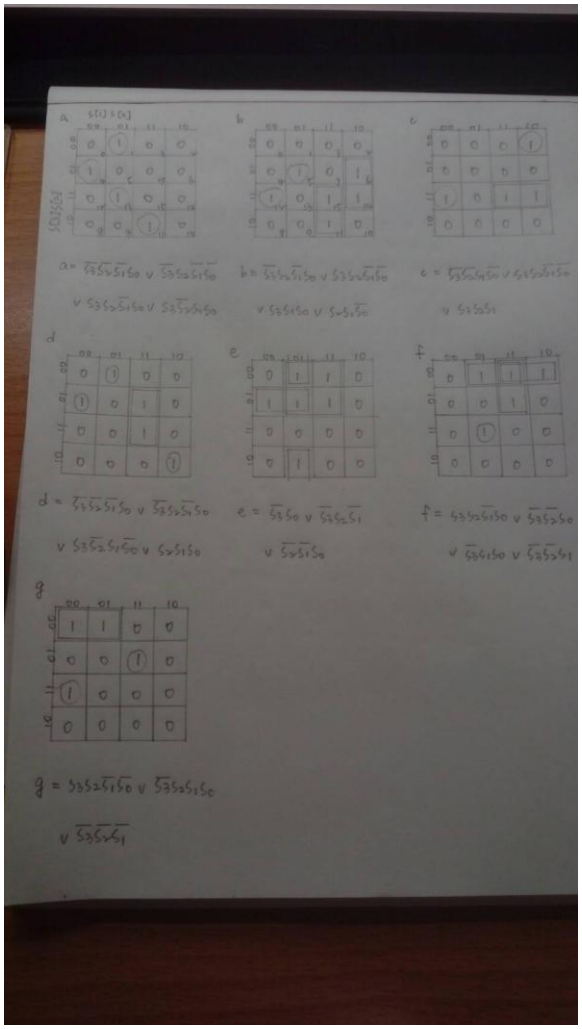
相較於原題，為了將 output 從 sum[3:0]轉換成 7-segment 的 output(a~g)，我先畫出以 4-bit 的 sum 為 input，7-bit 的 a~g 為 output 的 Truth Table(LEDs 亮燈時為 0，暗燈時為 1)，再畫出 a~g 的 K-map，就可以得到 a~g 的 Boolean equations。

但考量到只能使用 gate-level description，我後來改為用 sum-of-product 的形式寫 a~g 的 Boolean equations，在 verilog 上呈現出來會容易許多(當然是以 gate-level 表示)。而 cout 需要多接一個 NOT gate，以達到信號為 0 時暗燈，信號為 1 時亮燈的效果。

此外，我們還需要一個 AN[3:0]的 output，使 AN[3:1]固定輸出 1，AN[0]固定輸出 0，以指定只有最右邊的數字和點會顯示，否則 FPGA 上的四個數字都會亮。

最後只要將 I/O ports 接好，成功建立 XDC 檔就可以了。



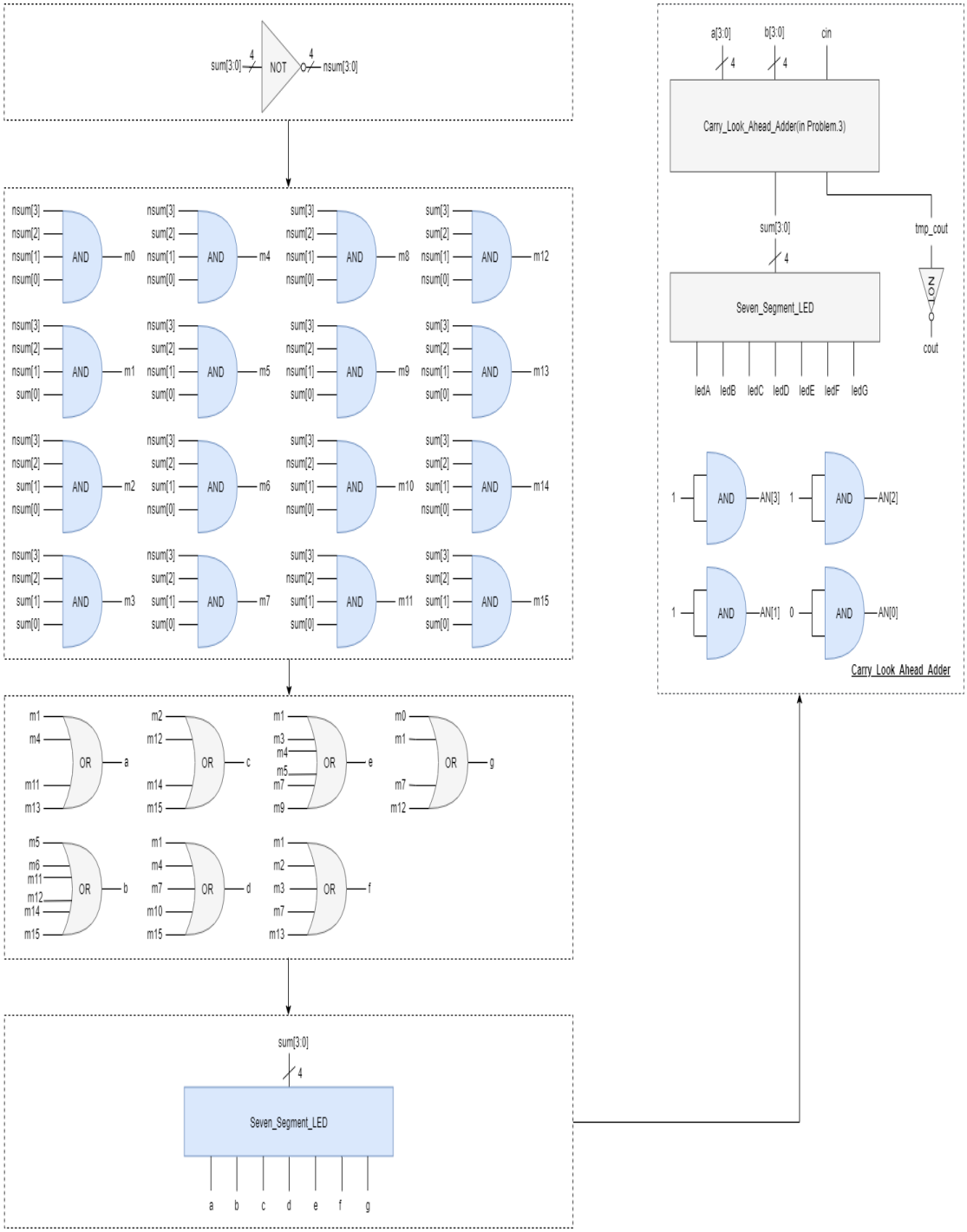


△　左圖為 Truth Table，右圖為 K-map

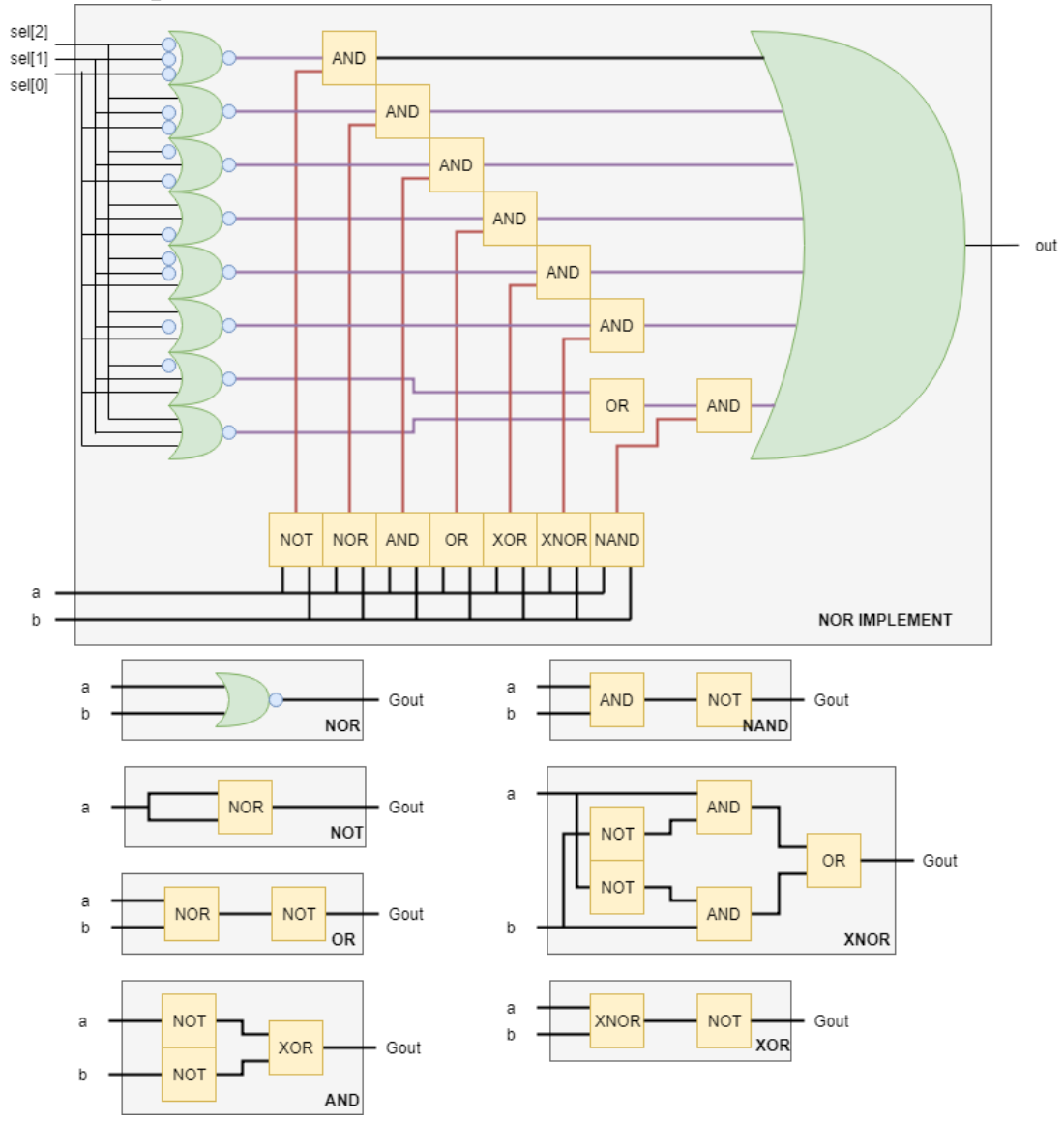
※以下標紅框處為 Carry_Look_Ahead_Adder_FPGA.v
相較於 Carry_Look_Ahead_Adder.v 新增的 code。

```
module Seven_Segment_LED (sum, a, b, c, d, e, f, g);
  input [4-1:0] sum;
  output a, b, c, d, e, f, g;

  wire [4-1:0] nsum;
  wire m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13, m14, m15;

  not n1 (nsum[0], sum[0]);
  not n2 (nsum[1], sum[1]);
  not n3 (nsum[2], sum[2]);
  not n4 (nsum[3], sum[3]);

  and a1 (m0, nsum[3], nsum[2], nsum[1], nsum[0]);
  and a2 (m1, nsum[3], nsum[2], nsum[1], sum[0]);
  and a3 (m2, nsum[3], nsum[2], sum[1], nsum[0]);
  and a4 (m3, nsum[3], nsum[2], sum[1], sum[0]);
  and a5 (m4, nsum[3], sum[2], nsum[1], nsum[0]);
  and a6 (m5, nsum[3], sum[2], nsum[1], sum[0]);
  and a7 (m6, nsum[3], sum[2], sum[1], nsum[0]);
  and a8 (m7, nsum[3], sum[2], sum[1], sum[0]);
  and a9 (m8, sum[3], nsum[2], nsum[1], nsum[0]);
  and a10 (m9, sum[3], nsum[2], nsum[1], sum[0]);
  and a11 (m10, sum[3], nsum[2], sum[1], nsum[0]);
  and a12 (m11, sum[3], nsum[2], sum[1], sum[0]);
  and a13 (m12, sum[3], sum[2], nsum[1], nsum[0]);
  and a14 (m13, sum[3], sum[2], nsum[1], sum[0]);
  and a15 (m14, sum[3], sum[2], sum[1], nsum[0]);
  and a16 (m15, sum[3], sum[2], sum[1], sum[0]);

  or o1 (a, m1, m4, m11, m13);
  or o2 (b, m5, m6, m11, m12, m14, m15);
  or o3 (c, m2, m12, m14, m15);
  or o4 (d, m1, m4, m7, m10, m15);
  or o5 (e, m1, m3, m4, m5, m7, m9);
  or o6 (f, m1, m2, m3, m7, m13);
  or o7 (g, m0, m1, m7, m12);
endmodule
```

```
module Carry_Look_Ahead_Adder (a, b, cin, cout, ledA, ledB, ledC, ledD, ledE, ledF, ledG, AN);
  input [4-1:0] a, b;
  input cin;
  output cout;
  wire [4-1:0] sum;
  output ledA, ledB, ledC, ledD, ledE, ledF, ledG;
  output [4-1:0] AN;

  wire [4-1:0] p;
  wire [4-1:0] g;
  wire c1, c2, c3;
  wire w1, w2, w3, w4, w5, w6, w7, w8, w9, w10;
  wire tmp_cout;

  and aa1 (AN[3], 1, 1);
  and aa2 (AN[2], 1, 1);
  and aa3 (AN[1], 1, 1);
  and aa4 (AN[0], 0, 0);

  ex_or eo1 (a[0], b[0], p[0]);
  ex_or eo2 (a[1], b[1], p[1]);
  ex_or eo3 (a[2], b[2], p[2]);
  ex_or eo4 (a[3], b[3], p[3]);

  and a1 (g[0], a[0], b[0]);
  and a2 (g[1], a[1], b[1]);
  and a3 (g[2], a[2], b[2]);
  and a4 (g[3], a[3], b[3]);

  and a5 (w1, p[0], cin);
  or o1 (c1, g[0], w1);
  and a6 (w2, p[1], g[0]);
  and a7 (w3, p[1], p[0], cin);
  or o2 (c2, g[1], w2, w3);
  and a8 (w4, p[2], g[1]);
  and a9 (w5, p[2], p[1], g[0]);
  and a10 (w6, p[2], p[1], p[0], cin);
  or o3 (c3, g[2], w4, w5, w6);
  and a11 (w7, p[3], g[2]);
  and a12 (w8, p[3], p[2], g[1]);
  and a13 (w9, p[3], p[2], p[1], g[0]);
  and a14 (w10, p[3], p[2], p[1], p[0], cin);
  or o4 (tmp_cout, g[3], w7, w8, w9, w10);
  not n1 (cout, tmp_cout);

  one_bit_full_adder fa1 (a[0], b[0], cin, sum[0]);
  one_bit_full_adder fa2 (a[1], b[1], c1, sum[1]);
  one_bit_full_adder fa3 (a[2], b[2], c2, sum[2]);
  one_bit_full_adder fa4 (a[3], b[3], c3, sum[3]);

  Seven_Segment_LED led1 (sum, ledA, ledB, ledC, ledD, ledE, ledF, ledG);
endmodule
```


sum[3:0]
4
NOT
4
nsum[3:0]
AND
m0
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
OR
a
b
c
d
e
f
g
Seven_Segment_LED
a
b
c
d
e
f
g
a[3:0]
b[3:0]
cin
Carry_Look_Ahead_Adder(in Problem.3)
tmp_cout
cout
ledA
ledB
ledC
ledD
ledE
ledF
ledG
AN[3]
AN[2]
AN[1]
AN[0]
Carry Look Ahead Adder

6. Nor Implement



● How we test our designs?

1. Binary to Grey

將 din 從 0000 每隔一個 delay 增加 1’b1 直到 1111。因為 grey code 很難以數學運算式表示，寫成 testbench 不會比直接觀察波形圖簡單，所以沒有特別寫 if 判斷式。

2. Multiplier

讓 a 跟 b 皆從 4’b0 開始，每隔一個 delay，{a, b}就增加 1’b1，直到 a 跟 b 都為 1111 為止。期間每一個 cycle 都用 if 判斷式檢測 Multiplier 的 output p 是否等於 a*b，若不是則輸出 wrong message，並使統計誤差總數的 integer error 加 1，最後於迴圈結束時輸出共出現幾次 error，以方便檢測。

3. Carry Look Ahead Adder

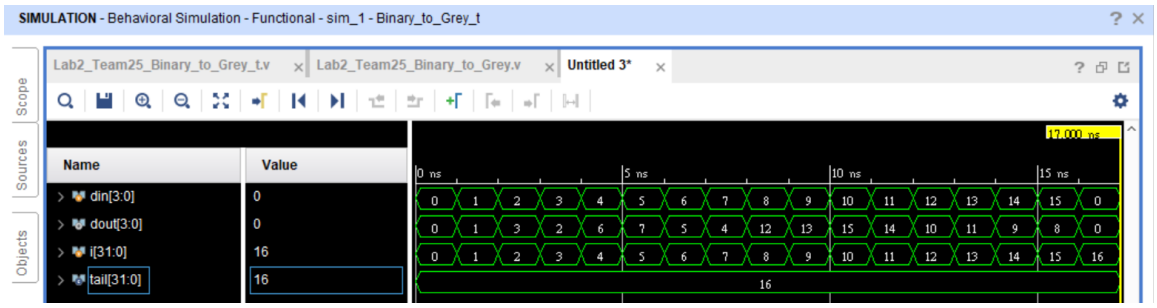
讓 a、b、cin 皆從 0 開始，每隔一個 delay，{a, b, cin}就增加 1’b1，直到 a、b、cin 的每個 bit 都為 1 為止。期間每一個 cycle 都用 if 判斷式檢測 Carry Look Ahead Adder 的 output {cout, sum[3:0]}(此處定義為 out[4:0])是否等於 a+b+cin，若不是則輸出 wrong message，並使統計誤差總數的 integer error 加 1，最後於迴圈結束時輸出共出現幾次 error，以方便檢測。

4. Decode and Execute

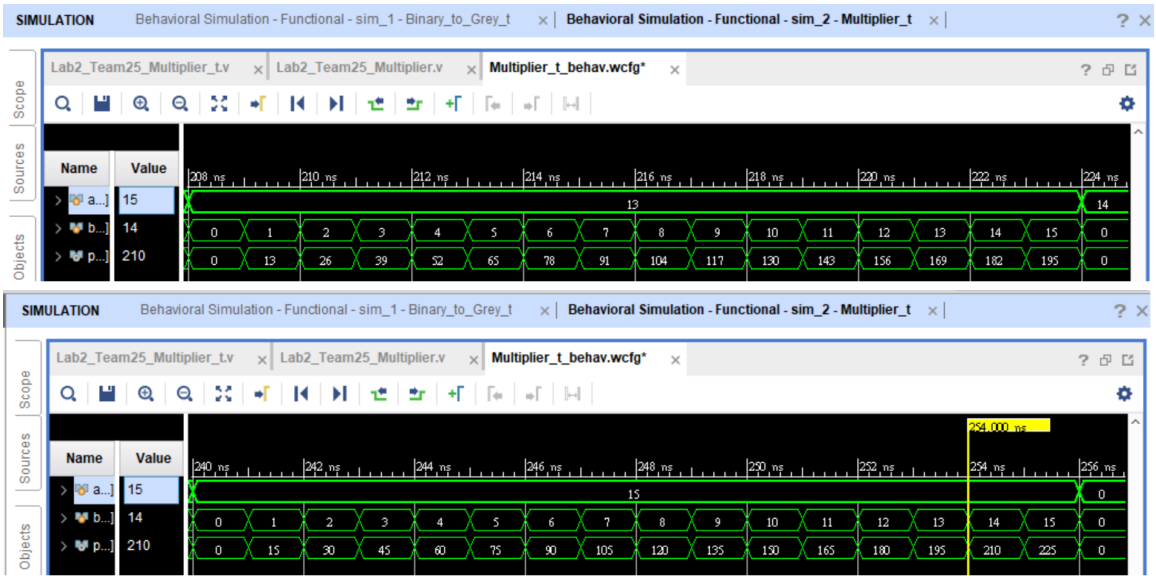
讓 rs、rt、op_code 皆從 0 開始，每隔一個 delay，{rs, rt, op_code}就增加 1’b1，直到 rs、rt、op_code 的每個 bit 都為 1 為止。期間每一個 cycle 都用 if 判斷式，先依據 op_code 的值進入不同的 case，再檢測 Decode and Execute 的 output rd 是否等於相對應的 function 的運算結果，若不是則輸出 wrong message，並使統計誤差總數的 integer error 加 1，最後於迴圈結束時輸出共出現幾次 error，以方便檢測。

- **Waveforms**

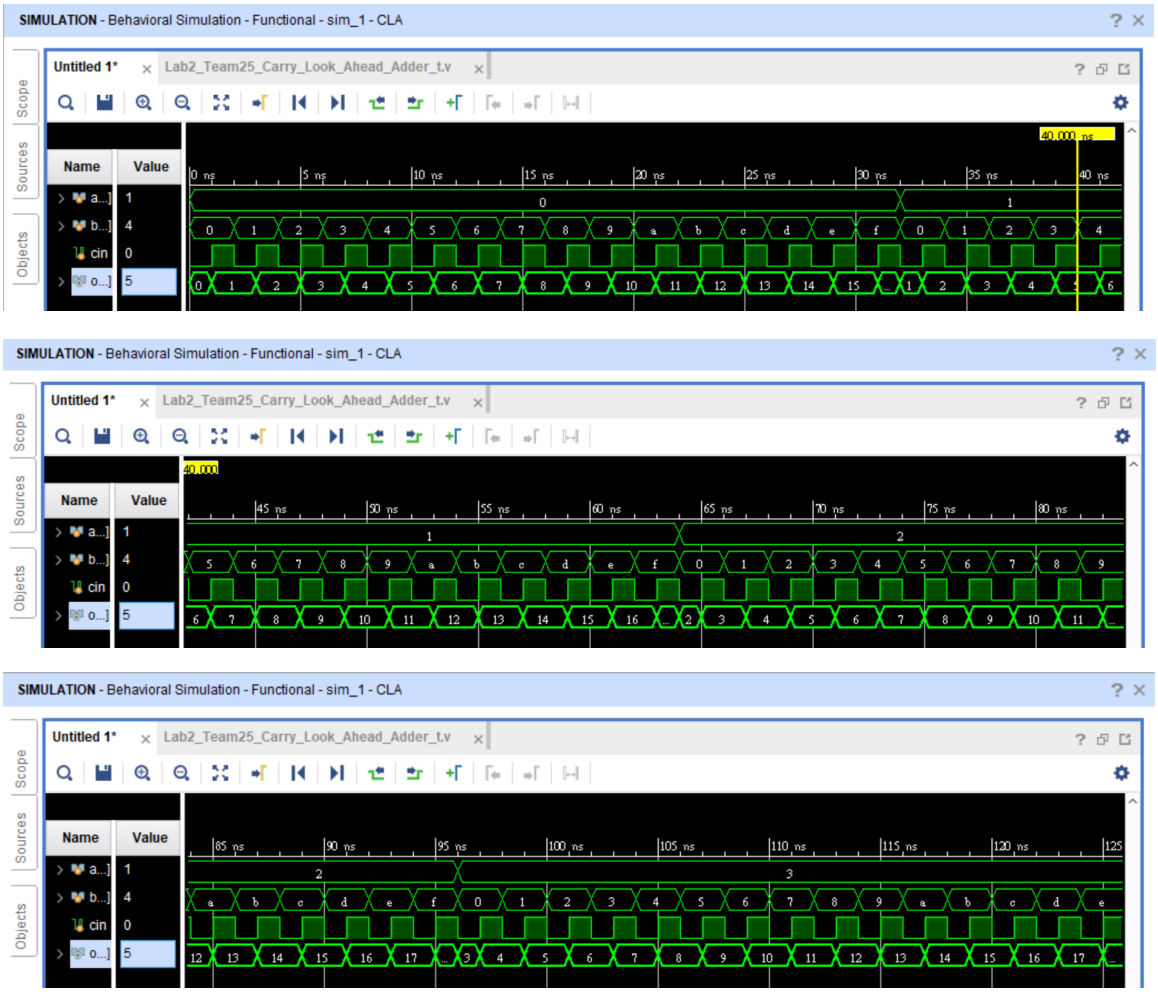
1. Binary to Grey



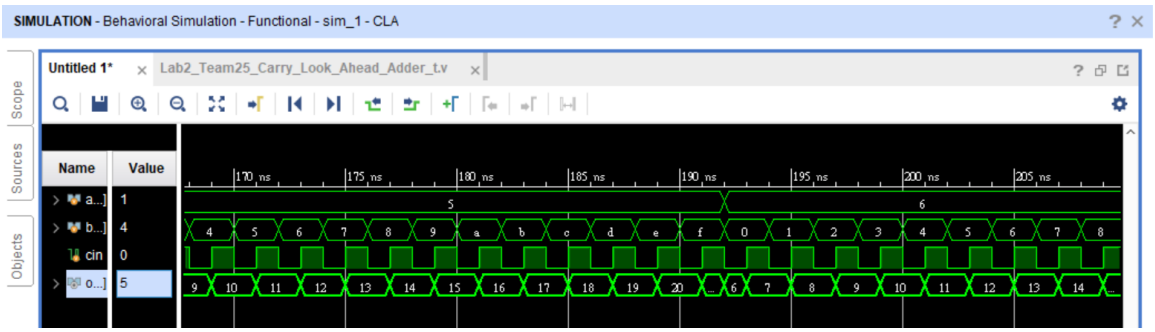
2. Multiplier

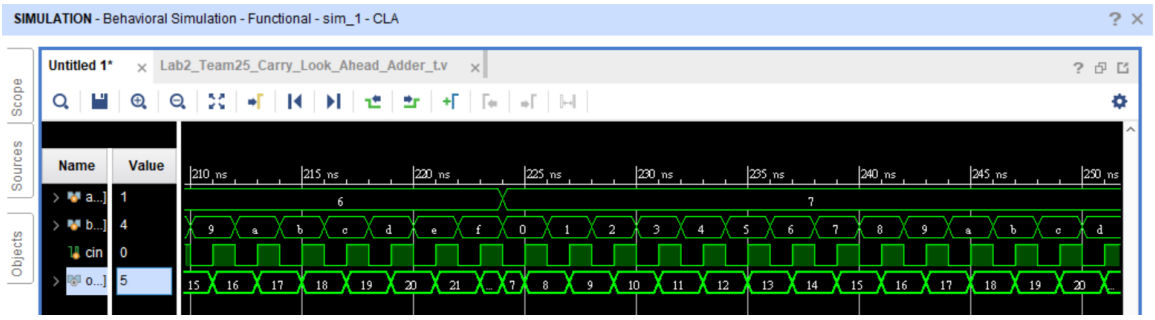


3. Carry Look Ahead Adder




SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Scope
Sources
Objects
Name
Value
a...]
b...]
cin
o...]
1
4
0
5
130 ns
135 ns
140 ns
145 ns
150 ns
155 ns
160 ns
165 ns


SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Scope
Sources
Objects
Name
Value
a...]
b...]
cin
o...]
1
4
0
5
170 ns
175 ns
180 ns
185 ns
190 ns
195 ns
200 ns
205 ns

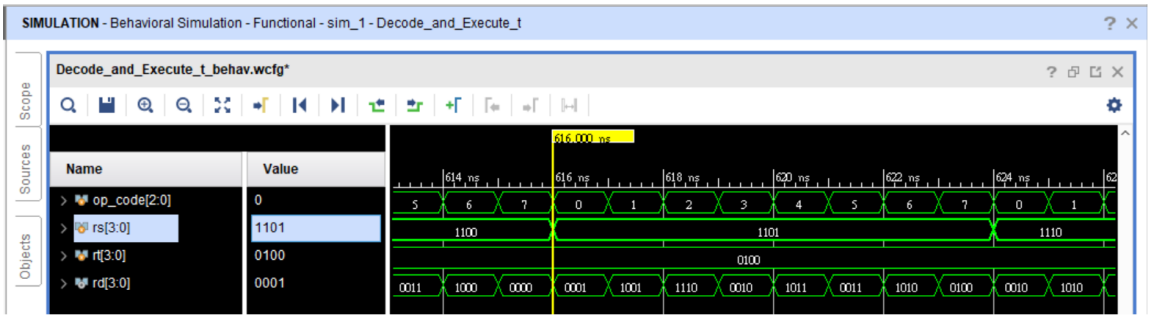

SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Scope
Sources
Objects
Name
Value
a...]
b...]
cin
o...]
1
4
0
5
210 ns
215 ns
220 ns
225 ns
230 ns
235 ns
240 ns
245 ns
250 ns

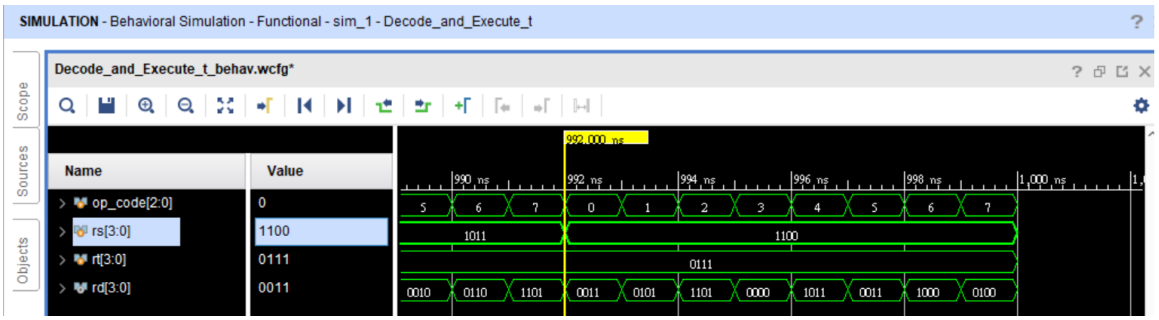

SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Scope
Sources
Objects
Name
Value
a...]
b...]
cin
o...]
1
4
0
5
255 ns
260 ns
265 ns
270 ns
275 ns
280 ns
285 ns
290 ns


SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Scope
Sources
Objects
Name
Value
a...]
b...]
cin
o...]
1
4
0
5
295 ns
300 ns
305 ns
310 ns
315 ns
320 ns
325 ns
330 ns


SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Scope
Sources
Objects
Name
Value
a...
b...
cin
o...
1
4
0
5
340 ns
345 ns
350 ns
355 ns
360 ns
365 ns
370 ns
375 ns
SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Name
Value
380 ns
385 ns
390 ns
395 ns
400 ns
405 ns
410 ns
415 ns
SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Name
Value
420 ns
425 ns
430 ns
435 ns
440 ns
445 ns
450 ns
455 ns
460 ns
SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Name
Value
465 ns
470 ns
475 ns
480 ns
485 ns
490 ns
495 ns
500 ns
SIMULATION - Behavioral Simulation - Functional - sim_1 - CLA
Untitled 1*
Lab2_Team25_Carry_Look_Ahead_Adder_t.v
Name
Value
475 ns
480 ns
485 ns
490 ns
495 ns
500 ns
505 ns
510 ns

4. Decode and Execute





- **What we have learned from Lab2?**

這次用到了很多以前數位邏輯設計的概念，如畫 Truth Table、使用 K-map、運算 two's complement 等等，複習了一些基礎但重要的知識。經過助教的 Lab1 review 之後，學會使用 google drawing 製圖，可以繪製更工整的電路圖，取代手繪。我們寫 testbench 也更加得心應手，這次在 testbench 裡新增 if-else block，用於檢測各題的 output，而不必以肉眼觀察波形圖是否有誤。

因為上一次 Lab 已經熟悉過一次操作 FPGA 的過程，這次比較沒什麼大問題，只是一開始寫 Truth Table 時沒有注意到 FPGA 上的七段顯示器會在信號為 0 時亮燈，信號為 1 時暗燈，差點釀成大禍。

Contribution List

- 蔡登瑞
  - 畫 gate-level diagram
- 蔡政諺
  - 實作 FPGA
- 共同完成
  - 寫 code、testbench
  - 寫 report