11010EECS207001Logic Design Lab

LAB2：Advanced Gate-Level Verilog

TEAM9

組長：108062213 顏浩昀

組員：106062304 黃鈺舒

Prof. Chun-Yi Lee

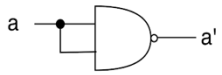
2021.10.14

11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

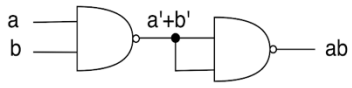
Basic Part

Nand Implement

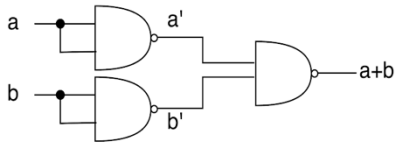
NAND_NOT



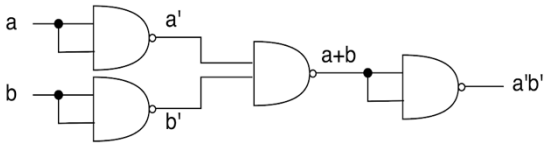
NAND_AND



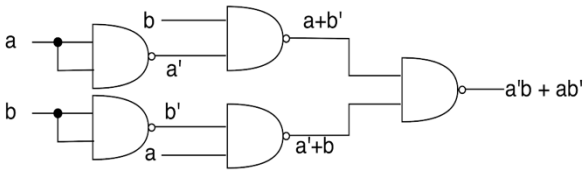
NAND_OR



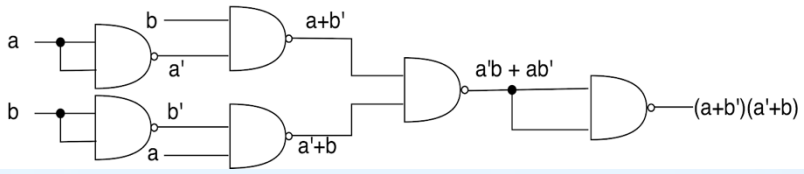
NAND_NOR



NAND_XOR

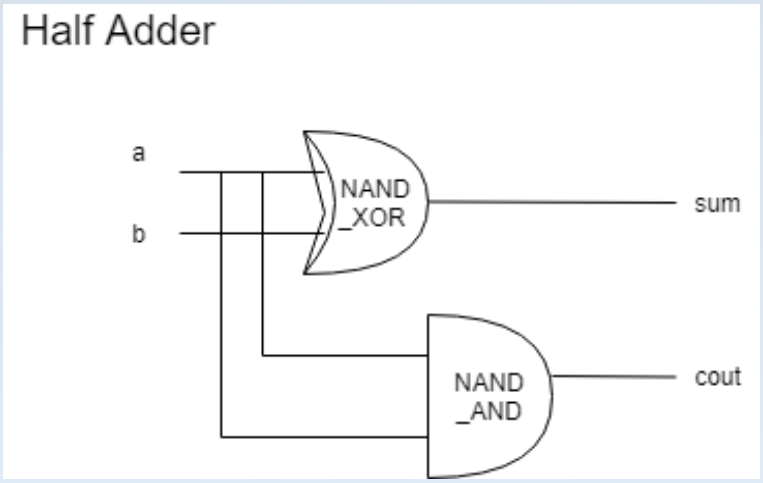


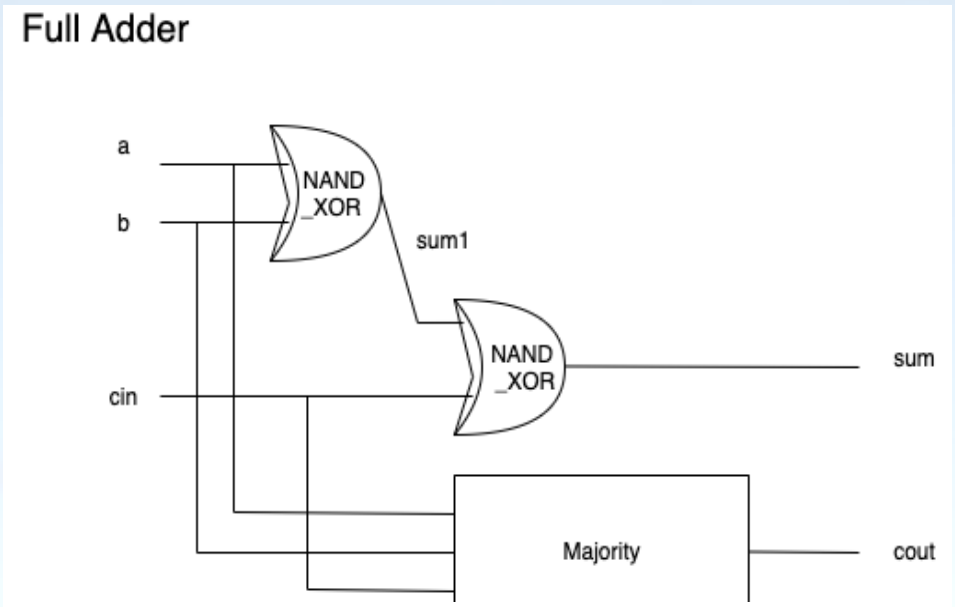
NAND_XNOR

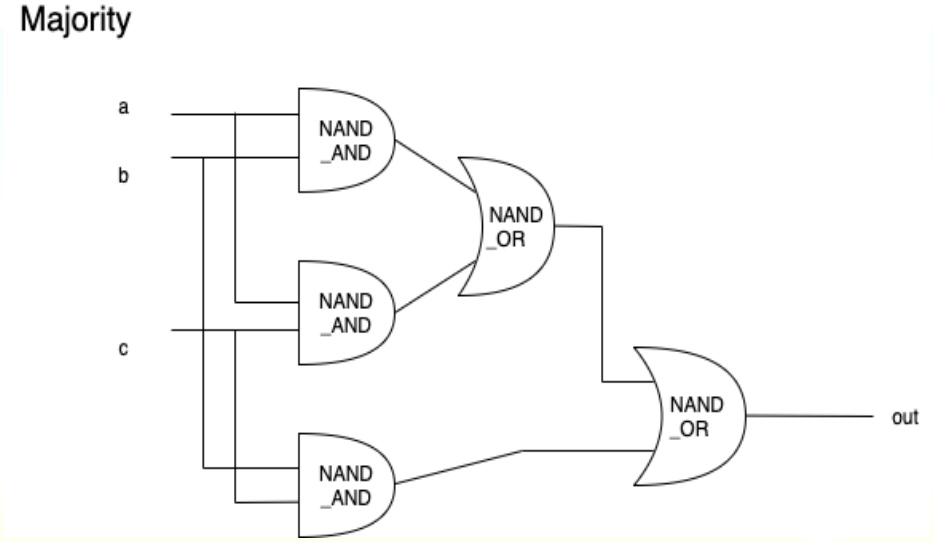


Adder

Half adder 並沒有 cin，只能算 1bit 簡單的運算。Full adder 則有 cin 輸入，可串接成加法器做多 bit 運算。







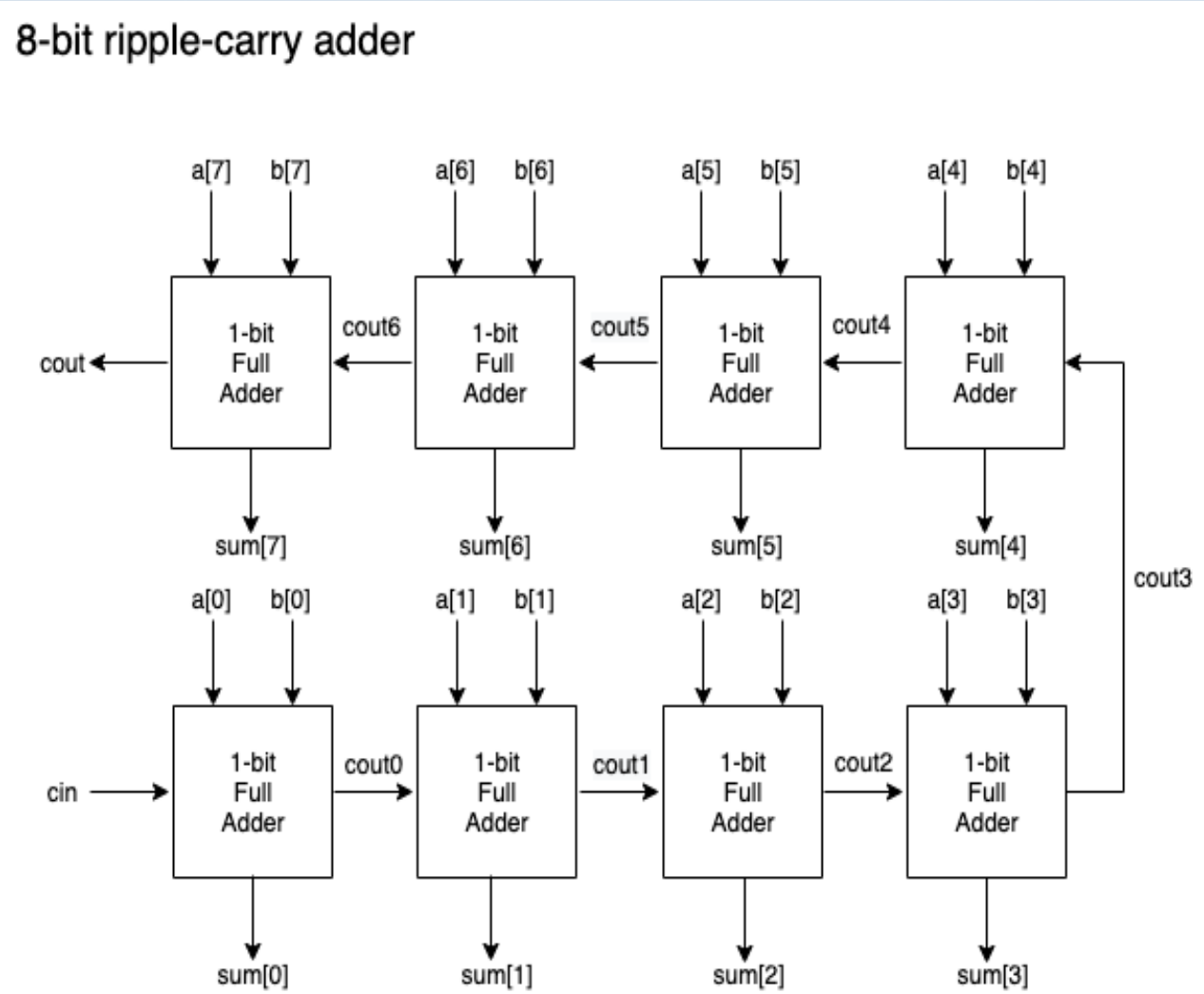
11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

Advance Part

11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

8-bit ripple-carry adder

I. Logic Design Diagram



Full Adder 實作請見 Basic Part

II. Explanation

將八個 Full adder 串接，每個 full addert 傳入 1bit a[i] b[i] cin 做運算，將 cout 進位傳入算下一個 bit 的 full adder 直到算完八個 bit 輸出每一位數的 sum 與最後的 carry out 進位。
（實作方式如下圖）

```
module Ripple_Carry_Adder(a, b, cin, cout, sum);
input [7:0] a, b;
input cin;
output cout;
output [7:0] sum;

wire cin1,cin2,cin3,cin4,cin5,cin6,cin7;
wire cout0,cout1,cout2,cout3,cout4,cout5,cout6;

Full_Adder a0(.a(a[0]), .b(b[0]), .cin(cin), .cout(cout0), .sum(sum[0]));
Full_Adder a1(.a(a[1]), .b(b[1]), .cin(cout0), .cout(cout1), .sum(sum[1]));
Full_Adder a2(.a(a[2]), .b(b[2]), .cin(cout1), .cout(cout2), .sum(sum[2]));
Full_Adder a3(.a(a[3]), .b(b[3]), .cin(cout2), .cout(cout3), .sum(sum[3]));
Full_Adder a4(.a(a[4]), .b(b[4]), .cin(cout3), .cout(cout4), .sum(sum[4]));
Full_Adder a5(.a(a[5]), .b(b[5]), .cin(cout4), .cout(cout5), .sum(sum[5]));
Full_Adder a6(.a(a[6]), .b(b[6]), .cin(cout5), .cout(cout6), .sum(sum[6]));
Full_Adder a7(.a(a[7]), .b(b[7]), .cin(cout6), .cout(cout), .sum(sum[7]));
endmodule
```

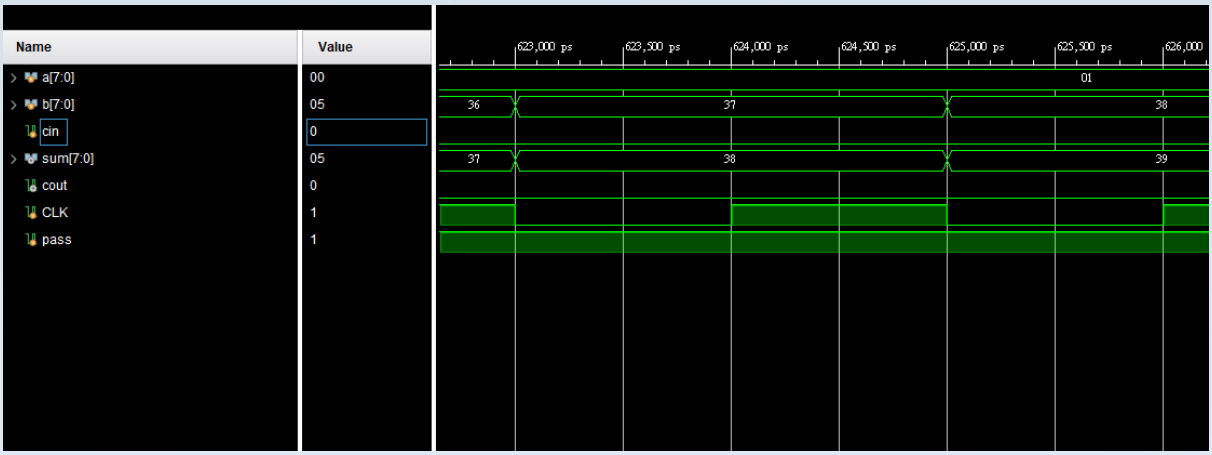
III. Testbench

這題用窮舉法，將 a,b,cin 的值全部跑過一次（在 negedge 賦予新值，本次所有 testbench 皆是在 negedge 給值，後面不再重述），然後透過 a+b+cin === {cout,sum} 的判斷式判斷結果是否正確（在 posedge 判斷，本次所有 testbench 皆是在 posedge 判斷正確性，後面不再重述），若正確 pass 訊號為 1，若錯誤則 pass 訊號為 0。（詳細實作方式請見下圖）

```
repeat (2**17) begin
    @(posedge CLK)
        simulate;
    @(negedge CLK)
        {cin,a,b} = {cin,a,b} + 1'b1;
end

task simulate;
    if(a+b+cin !== {cout, sum}) begin
        pass = 0;
    end
    else begin
        pass = 1;
    end
endtask
```

IIII. Wave


Name
Value
a[7:0]
00
b[7:0]
05
cin
0
sum[7:0]
05
cout
0
CLK
1
pass
1
623,000 ps
623,500 ps
624,000 ps
624,500 ps
625,000 ps
625,500 ps
626,000
01
36
37
38
37
38
39

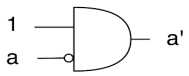
11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

Decode and execute

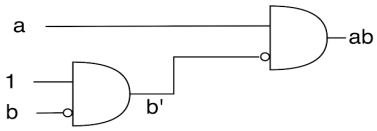
I. Logic Design Diagram

(i) Universal Gate 實作

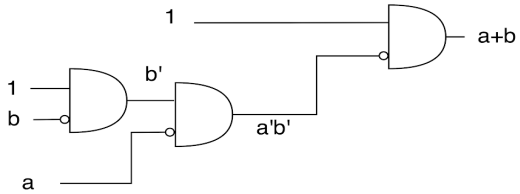
NOT



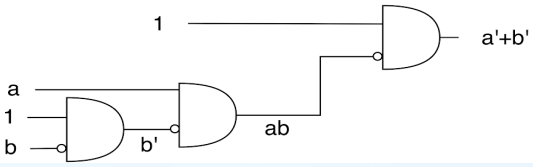
AND



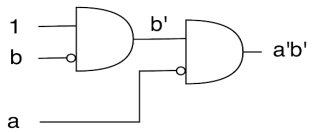
OR



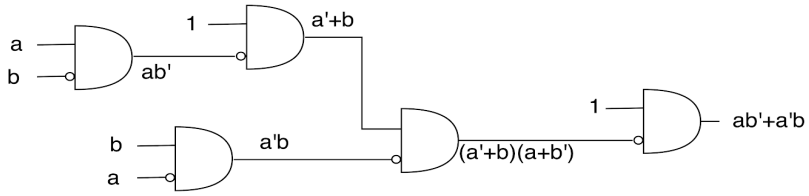
NAND



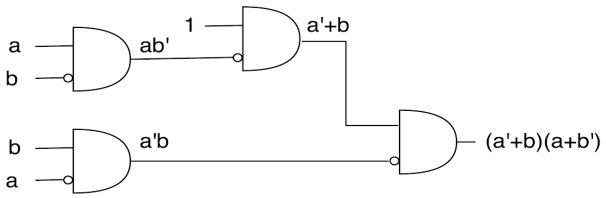
NOR



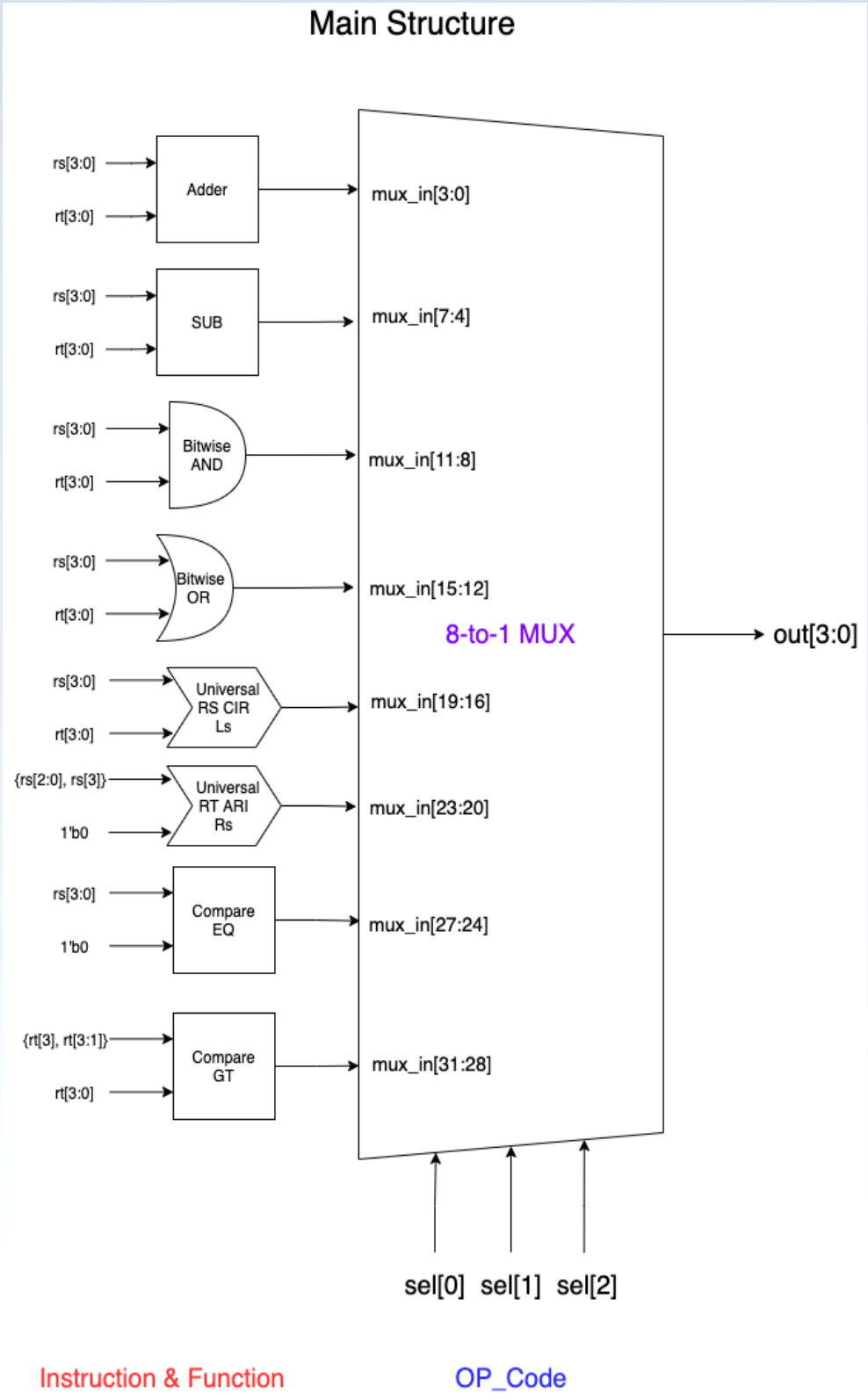
XOR



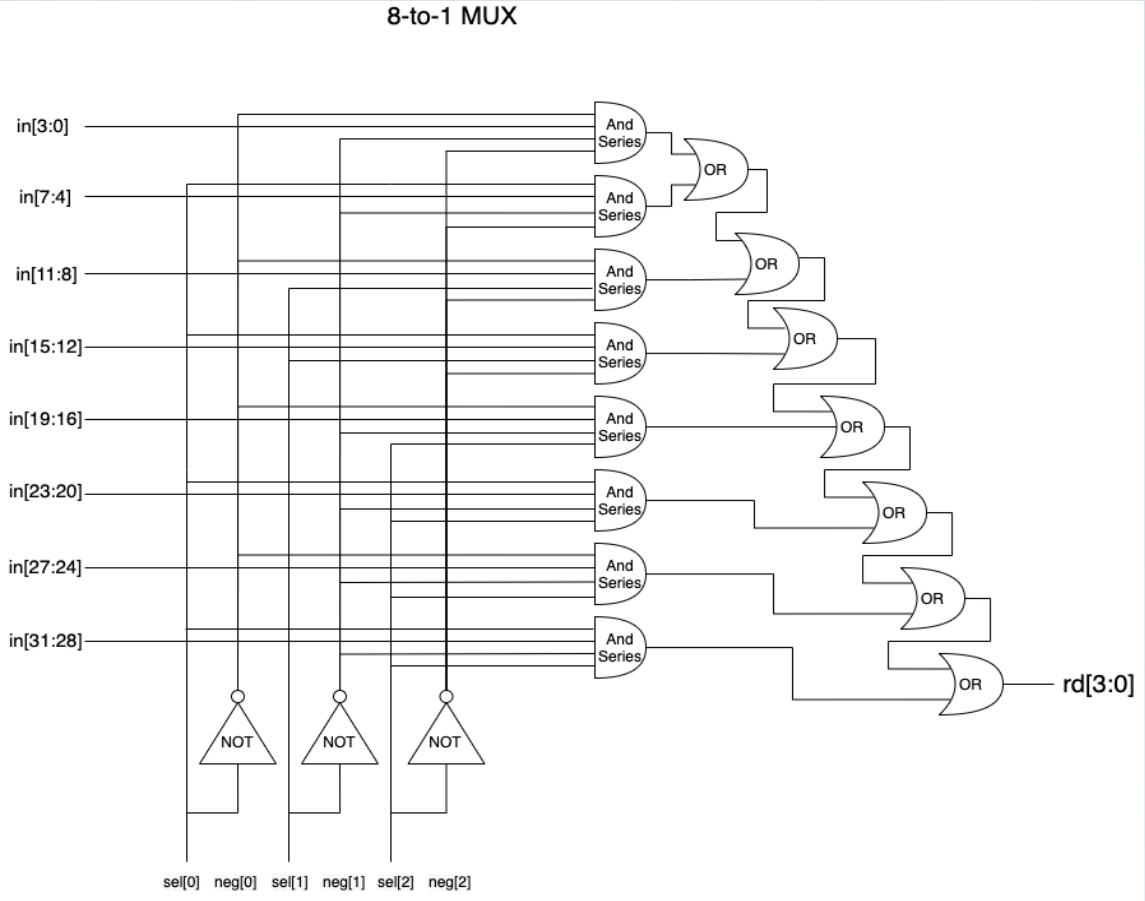
XNOR



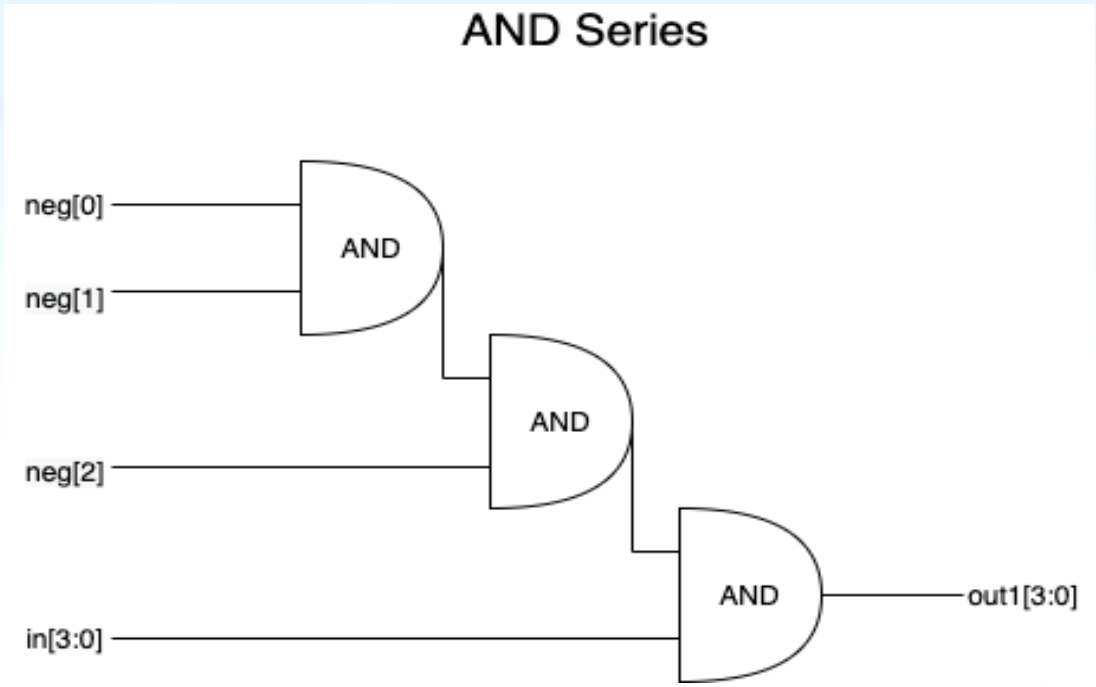
(11) 完整電路圖


Main Structure
rs[3:0]
rt[3:0]
Adder
mux_in[3:0]
rs[3:0]
rt[3:0]
SUB
mux_in[7:4]
rs[3:0]
rt[3:0]
Bitwise AND
mux_in[11:8]
rs[3:0]
rt[3:0]
Bitwise OR
mux_in[15:12]
8-to-1 MUX
out[3:0]
rs[3:0]
rt[3:0]
Universal RS CIR Ls
mux_in[19:16]
{rs[2:0], rs[3]}
1'b0
Universal RT ARI Rs
mux_in[23:20]
rs[3:0]
1'b0
Compare EQ
mux_in[27:24]
{rt[3], rt[3:1]}
rt[3:0]
Compare GT
mux_in[31:28]
sel[0] sel[1] sel[2]
Instruction & Function
OP_Code

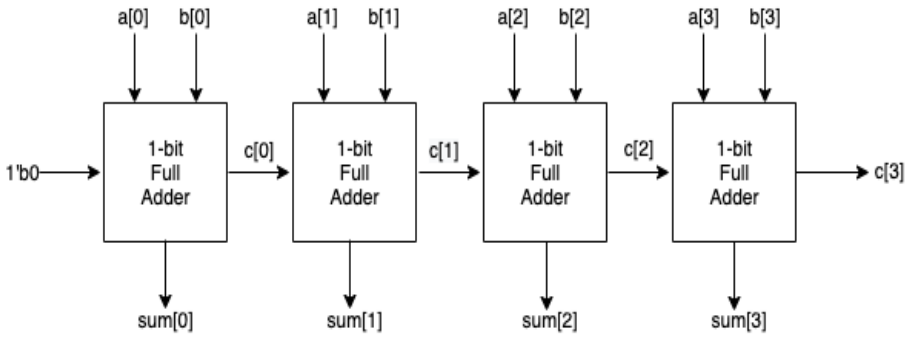
(ii) Mux



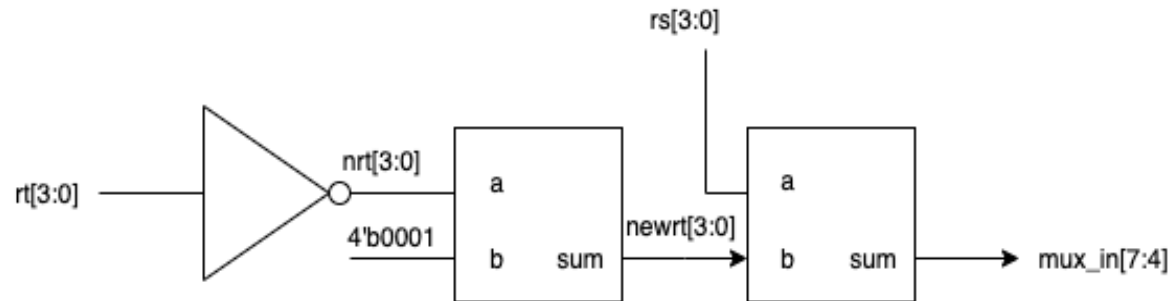
(iii) 各組元件 circuit



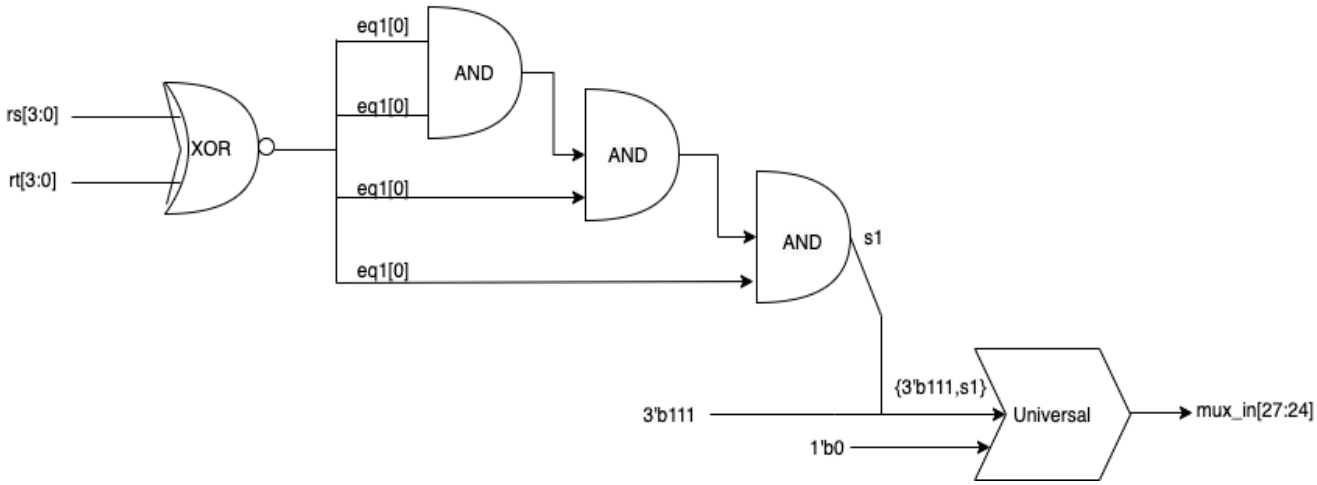
Adder



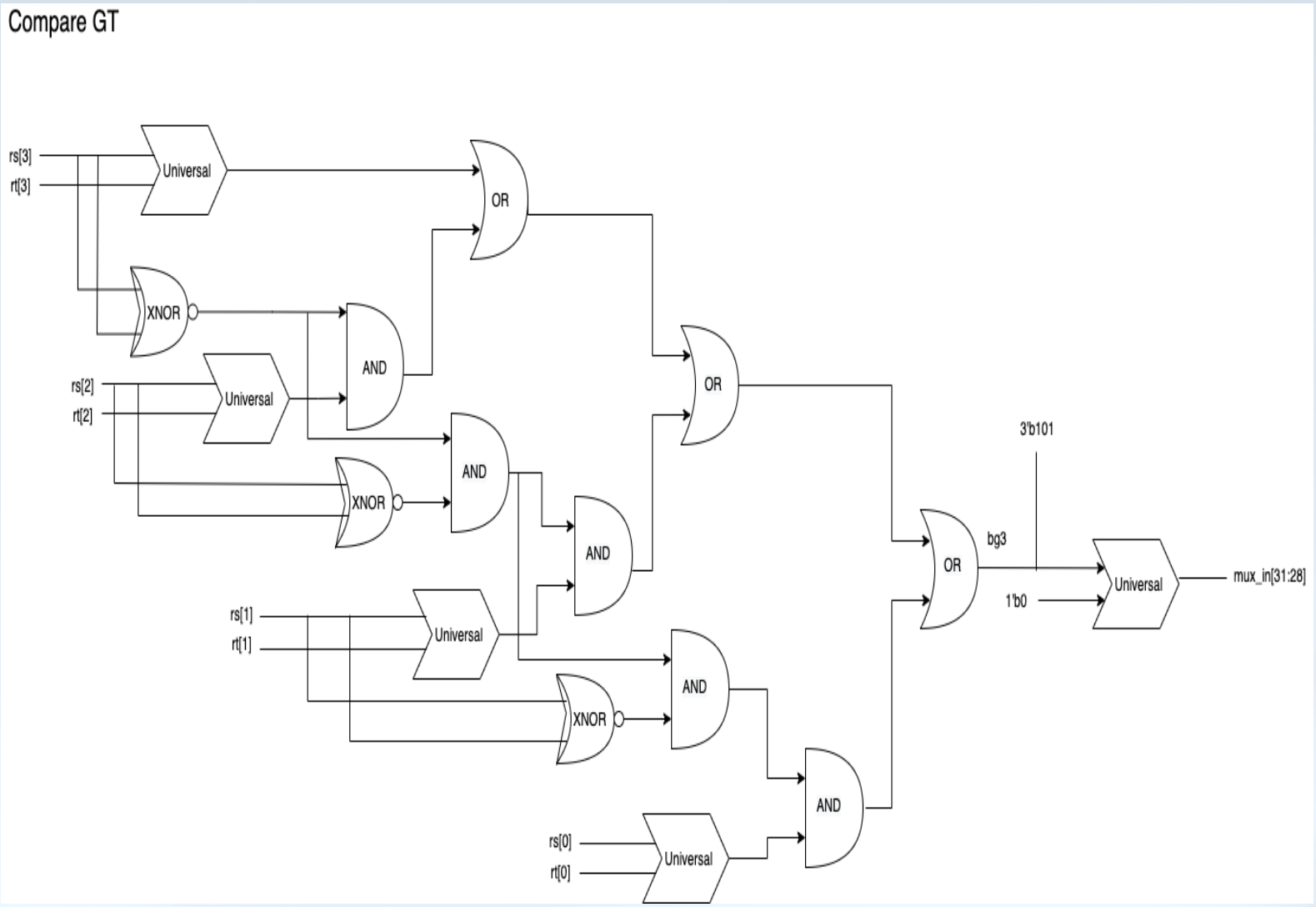
SUB



Compare EQ



Compare GT


rs[3]
rt[3]
Universal
OR
XNOR
AND
rs[2]
rt[2]
Universal
OR
3'b101
AND
XNOR
AND
bg3
OR
Universal
mux_in[31:28]
1'b0
rs[1]
rt[1]
Universal
AND
XNOR
AND
rs[0]
rt[0]
Universal

II. Explanation

這題主要架構是一個 8-bit-mux，最左邊輸入不同的 instruction，裡面實做不同的 function，每個運算的 block 如圖所示，傳入 rs 與 rt 算出八種結果，在進入 MUX 用作為 sel 訊號的 OPcode 選出指令的運算結果。（MUX 製作繁雜，僅取部分 code 為例）

```
module Mux_8in1(in, sel, out);
input [3-1:0] sel;
input [32-1:0] in;
output [4-1:0] out;

wire [4-1:0] out1, out2, out3, out4, out5, out6, out7, out8;
wire [3-1:0] neg;
wire and1_out, and2_out, and4_out, and5_out, and7_out, and8_out, and10_out, and11_out;
wire and13_out, and14_out, and16_out, and17_out, and19_out, and20_out, and22_out, and23_out;
wire [4-1:0] or1_out, or2_out, or3_out, or4_out, or5_out, or6_out;

// do negative select signal
NOT not0[2:0](neg, sel);

//000
AND and1(and1_out, neg[0], neg[1]);
AND and2(and2_out, and1_out, neg[2]);
AND and3[4-1:0](out1, {4{and2_out}}, in[3:0]);
///001
AND and4(and4_out, sel[0], neg[1]);
AND and5(and5_out, and4_out, neg[2]);
AND and6[4-1:0](out2, {4{and5_out}}, in[7:4]);
```

```
AND and19(and19_out, neg[0], sel[1]);
AND and20(and20_out, and19_out, sel[2]);
AND and21[4-1:0](out7, {4{and20_out}}, in[27:24]);
//111
AND and22(and22_out, sel[0], sel[1]);
AND and23(and23_out, and22_out, sel[2]);
AND and24[4-1:0](out8, {4{and23_out}}, in[31:28]);

OR or1[4-1:0](or1_out, out1, out2);
OR or2[4-1:0](or2_out, or1_out, out3);
OR or3[4-1:0](or3_out, or2_out, out4);
OR or4[4-1:0](or4_out, or3_out, out5);
OR or5[4-1:0](or5_out, or4_out, out6);
OR or6[4-1:0](or6_out, or5_out, out7);
OR or7[4-1:0](out, or6_out, out8);
endmodule
```

Adder：4-bit full adder
Sub：用 adder 做 2's complement，相當於取 NOT 再加 1'b0，再用 Adder 相加
Bitwise And：Universal And
Bitwise Or：Universal Or
RS CIR LS：用 Univeral Gate 將 bit 變成左移的組合
RT ARI RS：用 Univeral Gate 將 bit 變成右移的組合
Compare eq：用三個 Univeral and，如果相等就會持續傳出 1 訊號，最後加上 3'b111 用 Universal 跟 1'b0 做等價於 And 的運算組成 4bit 訊號輸入 MUX_IN。

Compare gt：依序檢查在 3,2,1,0bit 中 rs 是否大於 rt，如果有，universal 會輸出 1，否則代表兩個 bit 相等或小於，若為相等的情況:進入 xor = 1 到下一層繼續做比較，跟這層的 universal AND，如果 rs>rt 就會輸出 1，以此類推。若為 rs<rt 的情況:xor=0 的結果會輸入到下一層 And，再接到下一層 And...，最後輸出就會是 0。因此，只要從 3 到 0 任一 bit 檢查結果為 rs<rt，就輸出 0，再加上 3'b101，用 Universal 跟 1'b0 做等價於 And 的運算組成 4bit 訊號輸入 MUX_IN。

```
//000 ADD
Adder add(.a(rs), .b(rt), .sum(mux_in[3:0]));
//001 SUB two's compliment
NOT not0[4-1:0](nrt, rt);
Adder complement(.a(nrt), .b(4'b0001), .sum(newrt));
Adder sub(.a(rs), .b(newrt), .sum(mux_in[7:4]));
//010 bitwise AND
AND bitwise_and[4-1:0](.out(mux_in[11:8]), .a(rs), .b(rt));
//011 bitwise OR
OR bitwise_or[4-1:0](mux_in[15:12], rs, rt);
//100 rs left shift
Universal_Gate link1[4-1:0](.out(mux_in[19:16]), .a({rs[2:0], rs[3]}), .b(1'b0));
//101 rt right shift
Universal_Gate link2[4-1:0](.out(mux_in[23:20]), .a({rt[3], rt[3:1]}), .b(1'b0));
//110 compare rs == rt //4 bit magnitude comparator
XNOR xnor1 [3:0] (eq1, rs, rt);
AND and1(eq2, eq1[0], eq1[1]);
AND and2(eq3, eq2, eq1[2]);
AND and3(s1, eq3, eq1[3]);
Universal_Gate link_eq[3:0](.a({3'b111, s1}), .b(1'b0), .out(mux_in[27:24]));
```

```
//111 compare rs > rt//4 bit magnitude comparator
Universal_Gate bit3(.out(ab_3), .a(rs[3]), .b(rt[3])); //rs>rt in [3] when ab_3=1;

XNOR xnor_3(.out(continue_3), .a(rs[3]), .b(rt[3]));
Universal_Gate bit2(.out(ab_2), .a(rs[2]), .b(rt[2]));
AND and_2(.out(choose_in_2), .a(continue_3), .b(ab_2)); //rs>rt in [2] when choose_in_2=1;

XNOR xnor_2(.out(continue_2), .a(rs[2]), .b(rt[2]));
Universal_Gate bit1(.out(ab_1), .a(rs[1]), .b(rt[1]));
AND and_1_1(.out(choose_in_1_1), .a(continue_3), .b(continue_2));
AND and_1_2(.out(choose_in_1_2), .a(choose_in_1_1), .b(ab_1)); //rs>rt in [1] when choose_in_1_2=1;

XNOR xnor_1(.out(continue_1), .a(rs[1]), .b(rt[1]));
Universal_Gate bit0(.out(ab_0), .a(rs[0]), .b(rt[0]));
AND and_0_1(.out(choose_in_0_1), .a(continue_3), .b(continue_2));
AND and_0_2(.out(choose_in_0_2), .a(choose_in_0_1), .b(continue_1));
AND and_0_3(.out(choose_in_0_3), .a(choose_in_0_2), .b(ab_0)); //rs>rt in [0] when choose_in_0_3=1;

OR orbg1(.out(bg1), .a(ab_3), .b(choose_in_2));
OR orbg2(.out(bg2), .a(bg1), .b(choose_in_1_2));
OR orbg3(.out(bg3), .a(bg2), .b(choose_in_0_3));
```

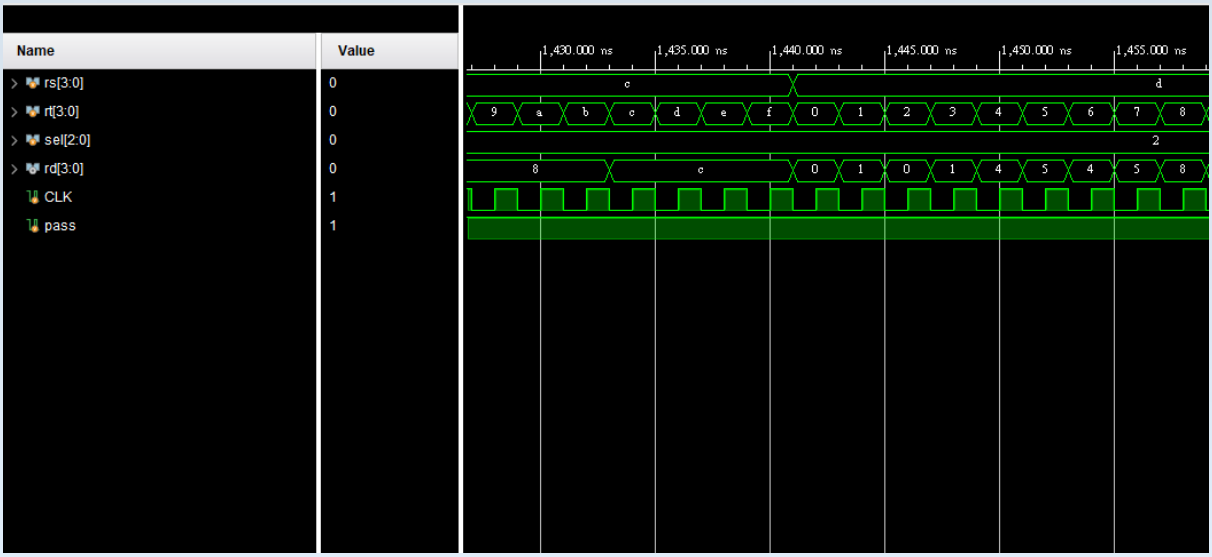
III. Testbench

本題使用窮舉法將 rs,rt,sel 的可能性全部跑過。這題比較特別的是根據不同 sel，會有不同的判斷方式，因此使用 case 去判斷要如何決定正確性。並且因為本題數量眾多，若只使用波形圖可能會不小心忽略錯誤，因此而外增加$display 的方式，若正確則輸出"Correct"，若錯誤則輸出"ERROR"並顯示該狀況的 rs,rt,rd,sel 及應該的正確解（正確解並非每個 case 都有使用）。另外在 sel=110 及 111 時，因為需要多層 if-else 才能判斷正確解，擔心有漏網之魚，因此將全部的 rs,rt,rd 都 display 一一比對。當然本題仍然有使用 pass 訊號快速並簡單的檢查錯誤。

```
3'b010:
    if((rs&rt) !== rd) begin
        pass = 0;
        $display("ERROR rs=%b, rt=%b, rd=%b, ans=%b sel=%b", rs, rt, rd, rs&rt, sel);
    end
    else begin
        pass = 1;
        $display("Correct");
    end
```

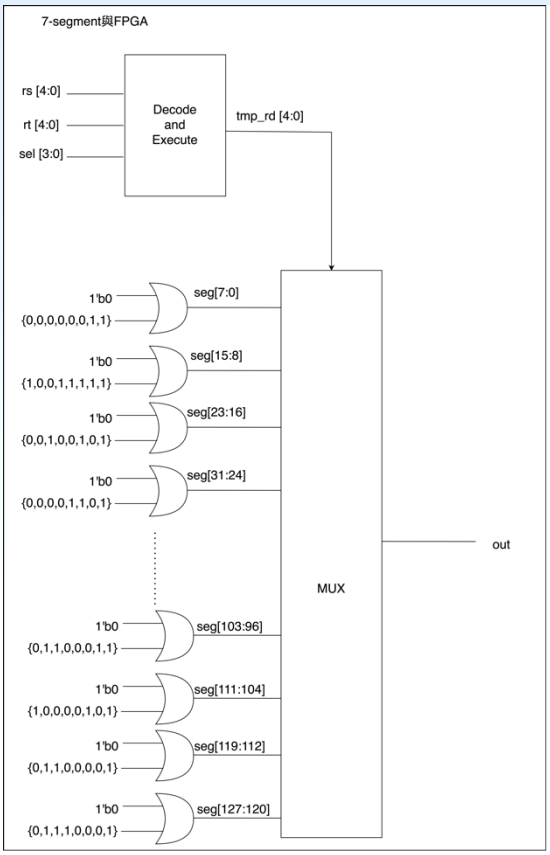
```
3'b110:
    if(rs === rt) begin
        pass = (rd===4'b1111)?1:0;
        $display("rs=%b, rt=%b, rd=%b, sel=%b", rs, rt, rd, sel);
    end
    else begin
        pass = (rd===4'b1110)?1:0;
        $display("rs=%b, rt=%b, rd=%b, sel=%b", rs, rt, rd, sel);
    end
3'b111:
    if(rs > rt) begin
        pass = (rd===4'b1011)?1:0;
        $display("rs=%b, rt=%b, rd=%b, sel=%b", rs, rt, rd, sel);
    end
    else begin
        pass = (rd===4'b1010)?1:0;
        $display("rs=%b, rt=%b, rd=%b, sel=%b", rs, rt, rd, sel);
    end
default: pass = 0;
```

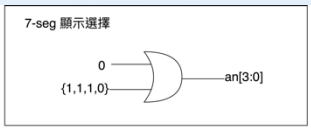
IIII. Wave

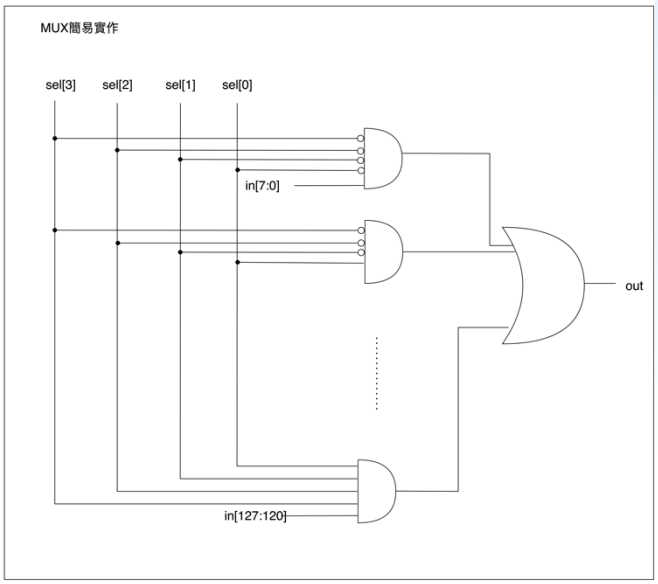


V.FPGA 實作

(i) FPGA diagram







(ii) Explanation

首先透過 Decoder and Execute 取得應該要輸出的 rd 值，這邊以 tmp_rd,代表 out1。接著運用 or gate 實作 7 segment 的數字顯示，然後作為 MUX 的 input，而剛剛的 tmp_rd 則為 MUX 的 sel signal，接著透過 MUX 選出需要的數字做為最後的 output。在 7 segment 的部分，還有透過 an[3:0]選擇需要顯示的位置（詳細接法見下圖程式碼）。

```
or num0[7:0](segment_control[7:0],     {8{1'b0}}, {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b1});
or num1[7:0](segment_control[15:8],    {8{1'b0}}, {1'b1, 1'b0, 1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1});
or num2[7:0](segment_control[23:16],   {8{1'b0}}, {1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 1'b1});
or num3[7:0](segment_control[31:24],   {8{1'b0}}, {1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b1, 1'b0, 1'b1});
or num4[7:0](segment_control[39:32],   {8{1'b0}}, {1'b1, 1'b0, 1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b1});
or num5[7:0](segment_control[47:40],   {8{1'b0}}, {1'b0, 1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1});
or num6[7:0](segment_control[55:48],   {8{1'b0}}, {1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1});
or num7[7:0](segment_control[63:56],   {8{1'b0}}, {1'b0, 1'b0, 1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1});
or num8[7:0](segment_control[71:64],   {8{1'b0}}, {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1});
or num9[7:0](segment_control[79:72],   {8{1'b0}}, {1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1});
or numA[7:0](segment_control[87:80],   {8{1'b0}}, {1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1});
or numB[7:0](segment_control[95:88],   {8{1'b0}}, {1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1});
or numC[7:0](segment_control[103:96],  {8{1'b0}}, {1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 1'b1});
or numD[7:0](segment_control[111:104], {8{1'b0}}, {1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 1'b1});
or numE[7:0](segment_control[119:112], {8{1'b0}}, {1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1});
or numF[7:0](segment_control[127:120], {8{1'b0}}, {1'b0, 1'b1, 1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1});
```

（圖：7-segment 實作）

```
not not1[3:0](neg, binary_num);

and and0[7:0](out0, in[7:0],     {8{neg[3]}},        {8{neg[2]}},        {8{neg[1]}},        {8{neg[0]}});
and and1[7:0](out1, in[15:8],    {8{neg[3]}},        {8{neg[2]}},        {8{neg[1]}},        {8{binary_num[0]}});
and and2[7:0](out2, in[23:16],   {8{neg[3]}},        {8{neg[2]}},        {8{binary_num[1]}}, {8{neg[0]}});
and and3[7:0](out3, in[31:24],   {8{neg[3]}},        {8{neg[2]}},        {8{binary_num[1]}}, {8{binary_num[0]}});
and and4[7:0](out4, in[39:32],   {8{neg[3]}},        {8{binary_num[2]}}, {8{neg[1]}},        {8{neg[0]}});
and and5[7:0](out5, in[47:40],   {8{neg[3]}},        {8{binary_num[2]}}, {8{neg[1]}},        {8{binary_num[0]}});
and and6[7:0](out6, in[55:48],   {8{neg[3]}},        {8{binary_num[2]}}, {8{binary_num[1]}}, {8{neg[0]}});
and and7[7:0](out7, in[63:56],   {8{neg[3]}},        {8{binary_num[2]}}, {8{binary_num[1]}}, {8{binary_num[0]}});
and and8[7:0](out8, in[71:64],   {8{binary_num[3]}}, {8{neg[2]}},        {8{neg[1]}},        {8{neg[0]}});
and and9[7:0](out9, in[79:72],   {8{binary_num[3]}}, {8{neg[2]}},        {8{neg[1]}},        {8{binary_num[0]}});
and andA[7:0](outA, in[87:80],   {8{binary_num[3]}}, {8{neg[2]}},        {8{binary_num[1]}}, {8{neg[0]}});
and andB[7:0](outB, in[95:88],   {8{binary_num[3]}}, {8{neg[2]}},        {8{binary_num[1]}}, {8{binary_num[0]}});
and andC[7:0](outC, in[103:96],  {8{binary_num[3]}}, {8{binary_num[2]}}, {8{neg[1]}},        {8{neg[0]}});
and andD[7:0](outD, in[111:104], {8{binary_num[3]}}, {8{binary_num[2]}}, {8{neg[1]}},        {8{binary_num[0]}});
and andE[7:0](outE, in[119:112], {8{binary_num[3]}}, {8{binary_num[2]}}, {8{binary_num[1]}}, {8{neg[0]}});
and andF[7:0](outF, in[127:120], {8{binary_num[3]}}, {8{binary_num[2]}}, {8{binary_num[1]}}, {8{binary_num[0]}});

or out1[7:0](out, out0, out1, out2, out3, out4, out5, out6, out7, out8, out9, outA, outB, outC, outD, outE, outF);
```

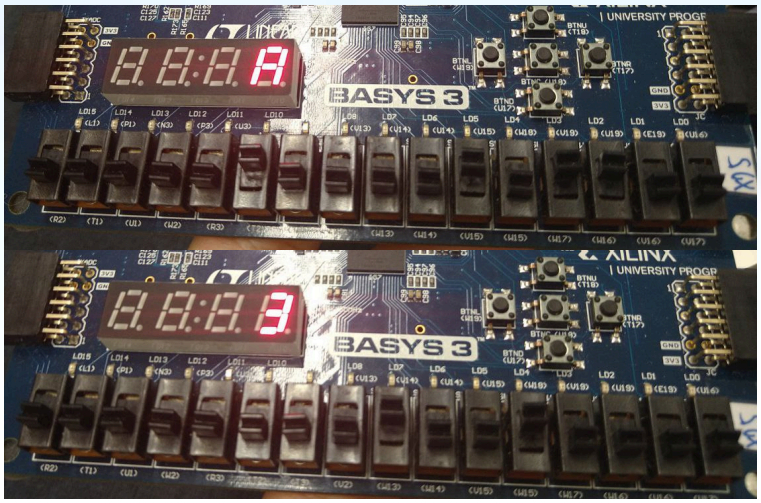
（圖：MUX 實作）

附上 I/O Port 圖檔表示 FPGA 上 input 與 output 實作。

an (4)	OUT			☑	34	LVCMOS33*
an[3]	OUT		W4	☑	34	LVCMOS33*
an[2]	OUT		V4	☑	34	LVCMOS33*
an[1]	OUT		U4	☑	34	LVCMOS33*
an[0]	OUT		U2	☑	34	LVCMOS33*
out (8)	OUT			☑	34	LVCMOS33*
out[7]	OUT		W7	☑	34	LVCMOS33*
out[6]	OUT		W6	☑	34	LVCMOS33*
out[5]	OUT		U8	☑	34	LVCMOS33*
out[4]	OUT		V8	☑	34	LVCMOS33*
out[3]	OUT		U5	☑	34	LVCMOS33*
out[2]	OUT		V5	☑	34	LVCMOS33*
out[1]	OUT		U7	☑	34	LVCMOS33*
out[0]	OUT		V7	☑	34	LVCMOS33*

rs (4)	IN			☑	14	LVCMOS33*
rs[3]	IN		W14	☑	14	LVCMOS33*
rs[2]	IN		V15	☑	14	LVCMOS33*
rs[1]	IN		W15	☑	14	LVCMOS33*
rs[0]	IN		W17	☑	14	LVCMOS33*
rt (4)	IN			☑	(Multiple)	LVCMOS33*
rt[3]	IN		T2	☑	34	LVCMOS33*
rt[2]	IN		T3	☑	34	LVCMOS33*
rt[1]	IN		V2	☑	34	LVCMOS33*
rt[0]	IN		W13	☑	14	LVCMOS33*
sel (3)	IN			☑	14	LVCMOS33*
sel[2]	IN		W16	☑	14	LVCMOS33*
sel[1]	IN		V16	☑	14	LVCMOS33*
sel[0]	IN		V17	☑	14	LVCMOS33*

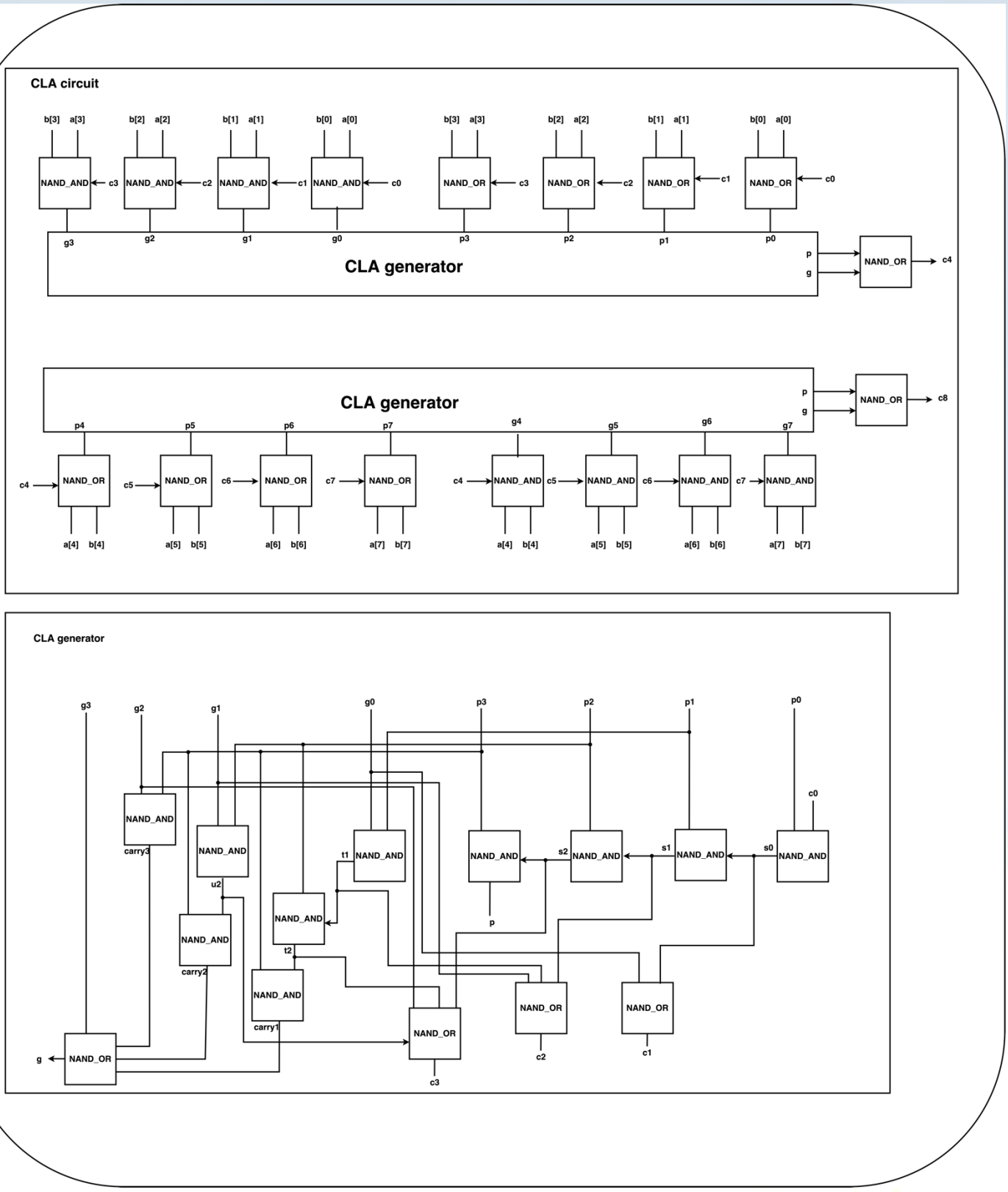
(iii)實作成果展示：分別展示(sel=100, rs=0101, rt=1000)及(sel=000, rs=0010,rt=0001)



11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

8-bit carry-lookahead (CLA) adder

I. Logic Design Diagram



II. Explanation

generate：gi = ai*bi

Praprogate：pi =ai+bi

carry in：ci+1 = gi+pi*ci

也就是說，以 g 來說若 ai,bi 都為 1 可以直接進位，是**我們一開始就可以得到的值，p 則是只要知道 c0 跟各層 ai,bi 也能直接算出來，都不需要特別等到前一個 Cin 來決定 Cout 是否為 1，比 RCA 節省了許多 Clock cycle**。

所以進入 4 bit CLA gen 做運算時，根據以上原則，我們有:

g(0,3)=g3+g2p3+g1p2p3+g0p1p2p3

p(0,3)=p3p2p1p0c0

這兩個 bit 再輸入 2 bit CLA gen 做 Universal 的 Or，只要有一個是 1 就代表會進位，輸出的 c4 再進入下一個計算 a[4:7]+b[4:7]的 4 bit CLA gen，算出 g,p(4,7)，用 2 bit CLA gen 得到最後的 cout。

（由於 code 繁雜僅取球出 p,g 時的 code 為例）

```
wire v0,v1,v2;
//NAND_OR or4(.a(carry0), .b(carry1), .out(v0));
NAND_OR or5(.a(carry1), .b(carry2), .out(v1));
NAND_OR or6(.a(v1), .b(carry3), .out(v2));
NAND_OR or7(.a(v2), .b(g3), .out(g));
```

```
NAND_AND and4(.a(p0), .b(c0), .out(s0));
NAND_AND and5(.a(s0), .b(p1), .out(s1));
NAND_AND and6(.a(s1), .b(p2), .out(s2));
NAND_AND and7(.a(s2), .b(p3), .out(p));
```

III. Testbench

本題使用窮舉法將所有 a,b,c0 跑過，透過 a+b+c0 === {c8,sum}的的判斷式判斷結果是否正確，若正確 pass 訊號為 1，若錯誤則 pass 訊號為 0。

```
repeat (2**17) begin
    @(posedge CLK)
        simulate;
    @(negedge CLK)
        {c0,a,b} = {c0,a,b}+1'b1;
end
```

```
task simulate;
    if(a+b+c0 !== {c8,s}) begin
        pass = 0;
    end
    else begin
        pass = 1;
    end
```

IIII. Wave

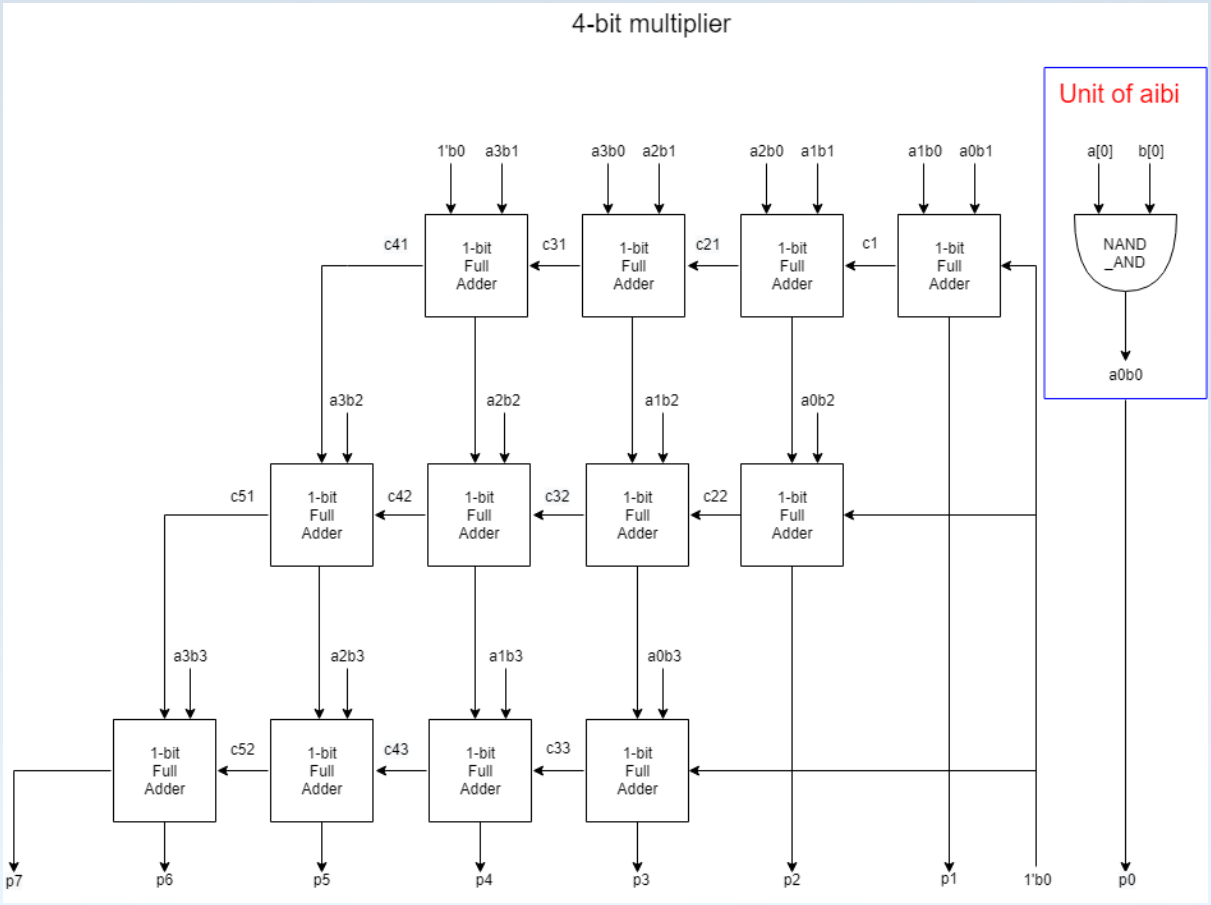

Name
Value
a[7:0]
04
b[7:0]
c4
c0
0
s[7:0]
c8
c8
0
CLK
1
pass
1
2,442.290 ns
2,415.000 ns
2,420.000 ns
2,425.000 ns
2,430.000 ns
2,435.000 ns
2,440.000 ns
04
b5 b6 b7 b8 b9 ba bb bc bd be bf c0 c1 c2 c3 c4 c5 c6
b9 ba bb bc bd be bf c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca

11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

4-bit multiplier

I. Logic Design Diagram

(1) 完整電路圖



II. Explanation

圖中每一個 aibi 代表一個 Cell，等於 a[i],b[i]做 Nand_And 的輸出。

乘法器如同直式算法，第一層將 a[3:0]分別與 b[0]And，第二層與 b[1]，第三層與 b[2]，做完所有組合。And 之後的 bit 中，垂直兩兩進入 full adder 相加，carry in 與 RCA 類似會傳入同層下一位數作為 cout 進位，當沒有下一位時，進入下一層的下一位跟那個 bit 做相加，值得注意的是最上層的最後一位的輸入 bit 因為這層已經做完也沒有上一層的 carry in 所以要輸入 1'b0。除了 p0 是 a[0]and b[0]，其他 p1~p6 都需要上層 sun 算完後作為 b 傳入跟此層相加，算出答案與 carry out。

（code 繁多，以其中一階層為例）

```
wire a3b1,a2b2,a1b3,c41,c42,c43,s41,s42;
NAND_AND and10(.a(a[3]), .b(b[1]), .out(a3b1));
NAND_AND and11(.a(a[2]), .b(b[2]), .out(a2b2));
NAND_AND and12(.a(a[1]), .b(b[3]), .out(a1b3));
Full_Adder a41(.a(r), .b(a3b1), .cin(c31), .cout(c41), .sum(s41));//
Full_Adder a42(.a(s41), .b(a2b2), .cin(c32), .cout(c42), .sum(s42));
Full_Adder a43(.a(s42), .b(a1b3), .cin(c33), .cout(c43), .sum(p[4]));
```

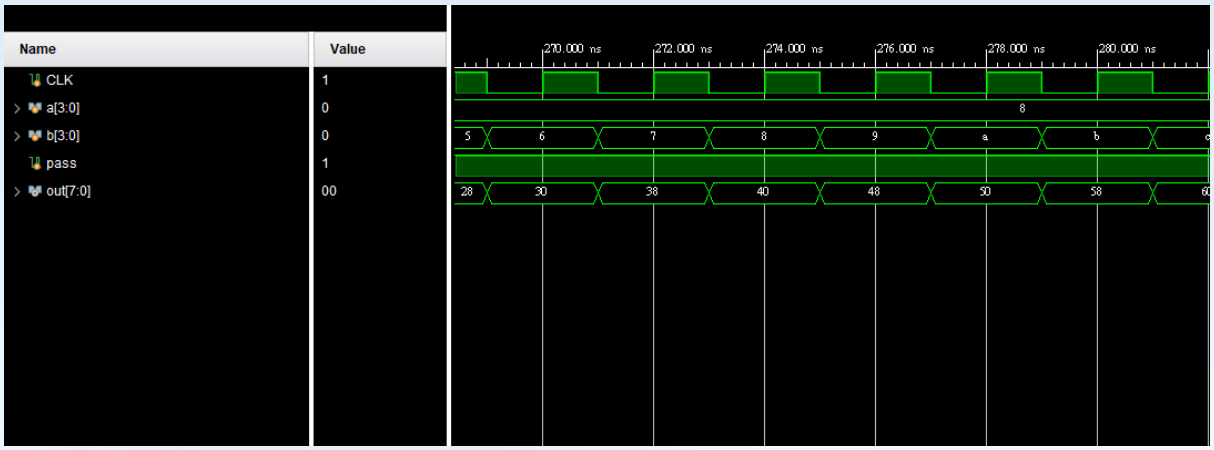
III. Test bench

本題使用窮舉法將所有 a,b 跑過，透過 a*b === out 的的判斷式判斷結果是否正確，若正確 pass 訊號為 1，若錯誤則 pass 訊號為 0。

```
repeat(2**8) begin
    @(posedge CLK)
        simulate;
    @(negedge CLK)
        {a,b} = {a,b}+1'b1;
end
```

```
task simulate;
    if(a*b !== out) begin
        pass = 0;
    end
    else begin
        pass = 1;
    end
endtask
```

IIII. Wave



11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

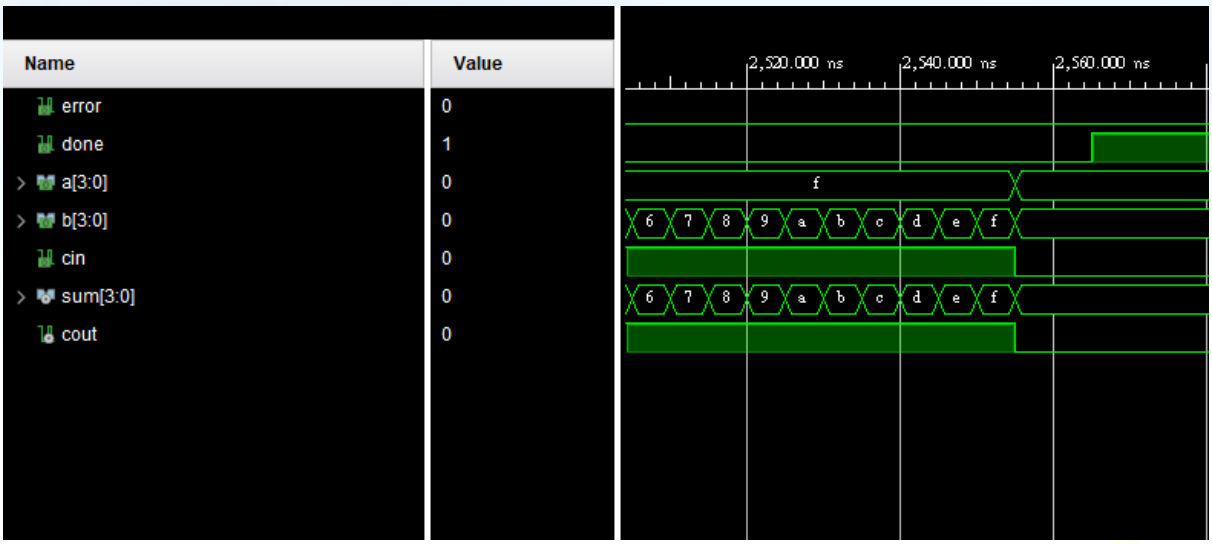
Exhausted Test

這個 testbench 同樣使用窮舉法，將 a,b 的值全數跑過。與前面不同的是題目要求判斷正確性的 error signal 與全部完成後的 done signal 有特定的對應時間。首先因為每次給予新值的間隔為 5 nanosconds，因此在 task test 當中共有(1+4)nanoseconds 的 delay，然後就會給予新的值。而當中的 1 nanosecond delay 就是 error signal 在輸入值後變化的時間點。詳細實作可看下圖

```
repeat (2**8) begin
        Test;
        {a,b} = {a,b}+1'b1;
end
done = 1'b1;

#5 $finish;
```

```
task Test;
    if({cout, sum} !== (a+b+cin)) begin
        #1
        error = 1'b1;
        #4
    end
    else begin
        #1
        error = 1'b0;
        #4
    end
endtask
```



11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

What do we learn in this Lab

再測第五題的時後發現原來 testbench 的 if else 裡不能放時間延遲，會出錯，改好後就對了。

decoder 那堤用 universal 實作 gate 也讓我開始會思考這些 gate 的意義，以及選擇他來實作這個邏輯的理由，compare 的 gt 就像是個用邏輯寫成的流程圖，其實看懂意義想通之後便可以一目瞭然。還有該怎麼實作一個好的 testbench，以及 coding style 的重要性，其實可以省下許多 debug 的時間。

11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

Cooperation

（依學號排列）

106062304 黃鈺舒：

Basic Q3 full,half adder,majority 電路圖

8-bit ripple-carry adder (RCA) 實作，

8-bit ripple-carry adder (RCA) report explanation，

8-bit ripple-carry adder (RCA) 電路圖

Decode and execute 實作，

Decode and execute report explanation,

Decode and execute structure, mux,

Universal full adder, majority, adder, sub, compare_eq, compare_gt 電路圖

8-bit carry-lookahead (CLA) adder 實作，

8-bit carry-lookahead (CLA) adder report explanation，

4-bit multiplier 實作

4-bit multiplier report explanation，

4-bit multiplier 電路圖

108062213 顏浩昀：

Basic Q1 Nand implement 電路圖

Nand Gate,And,Or,Not,Nor,Xor,Xnor 實作

8-bit ripple-carry adder (RCA) testbench，

8-bit ripple-carry adder (RCA) testbench explanation，

Decode and execute 實作，

Decode and execute testbench,

Decode and execute testbench explanation

Decode and execute Universal Gate,And,Or,Not,Nor,Xor,Xnor 電路圖

8-bit carry-lookahead (CLA) adder testbench

8-bit carry-lookahead (CLA) adder testbench explanation

8-bit carry-lookahead (CLA) adder 電路圖

11010EECS207001
Logic Design Lab
TEAM5 LAB1
2020.09.30

4-bit multiplier testbench

4-bit multiplier testbench explanation

Exhausted_Testing 製作

FPGA 電路圖

FPGA 板子燒錄