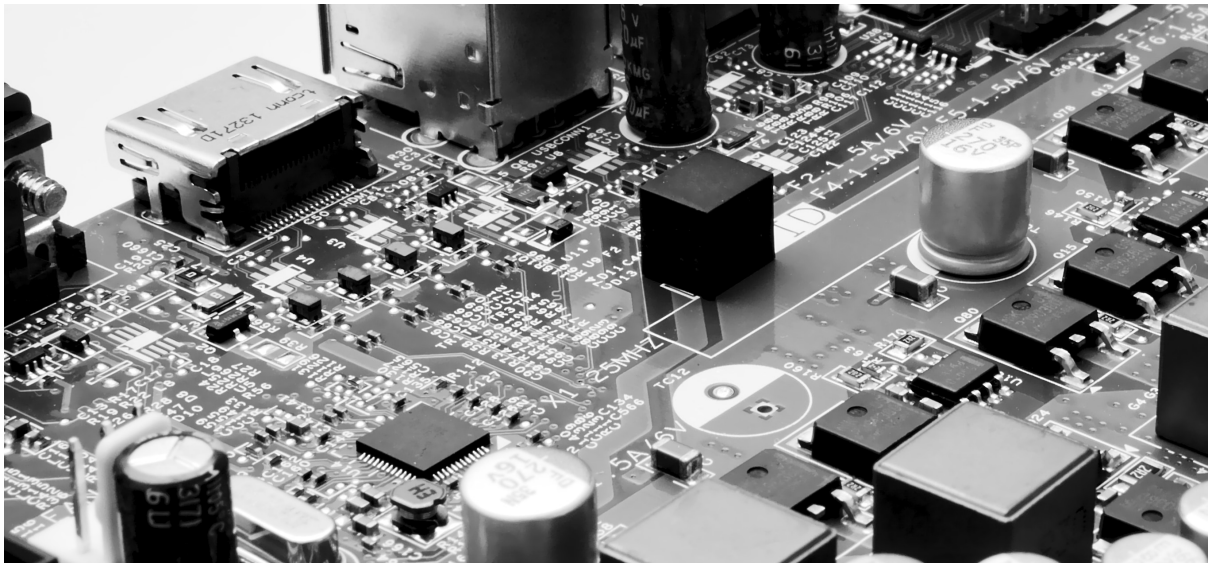


Lab1 Report Team10

Course : Logic Design Laboratory

Date : 2020/9/24

108062125 高敦晉
108062229 陳皇佑

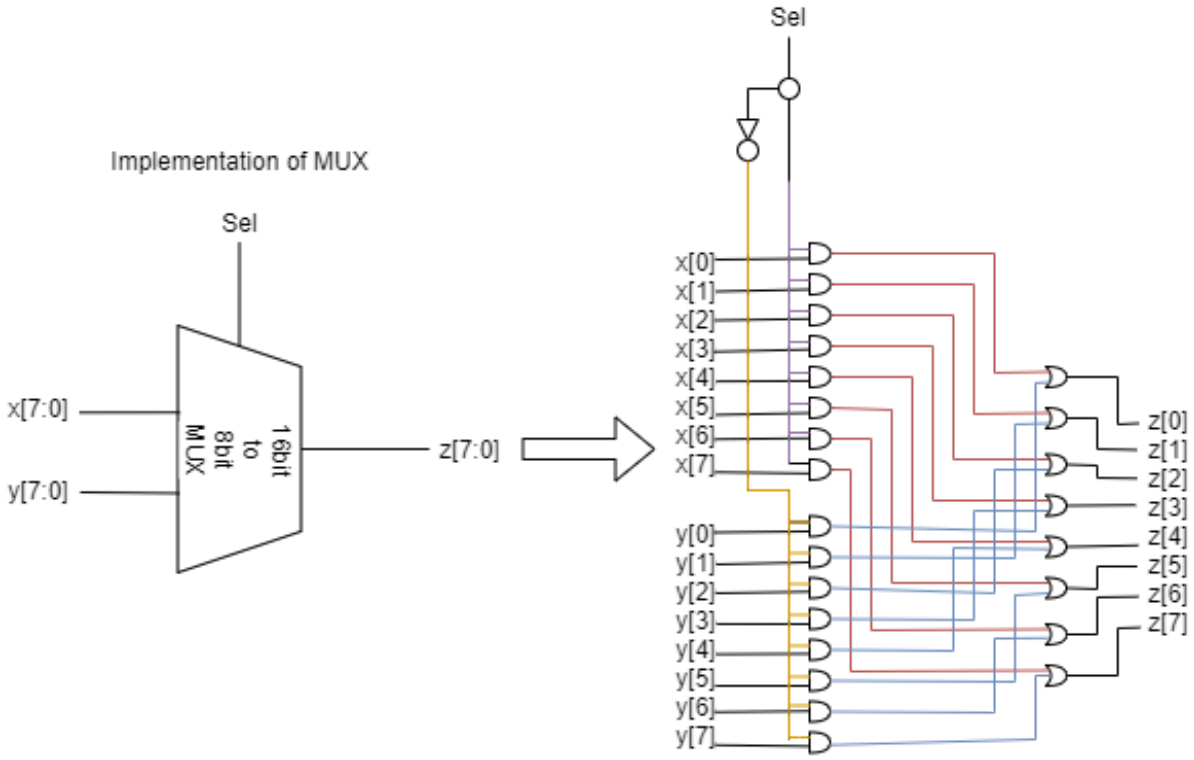
Verilog Question 1

Verilog Code

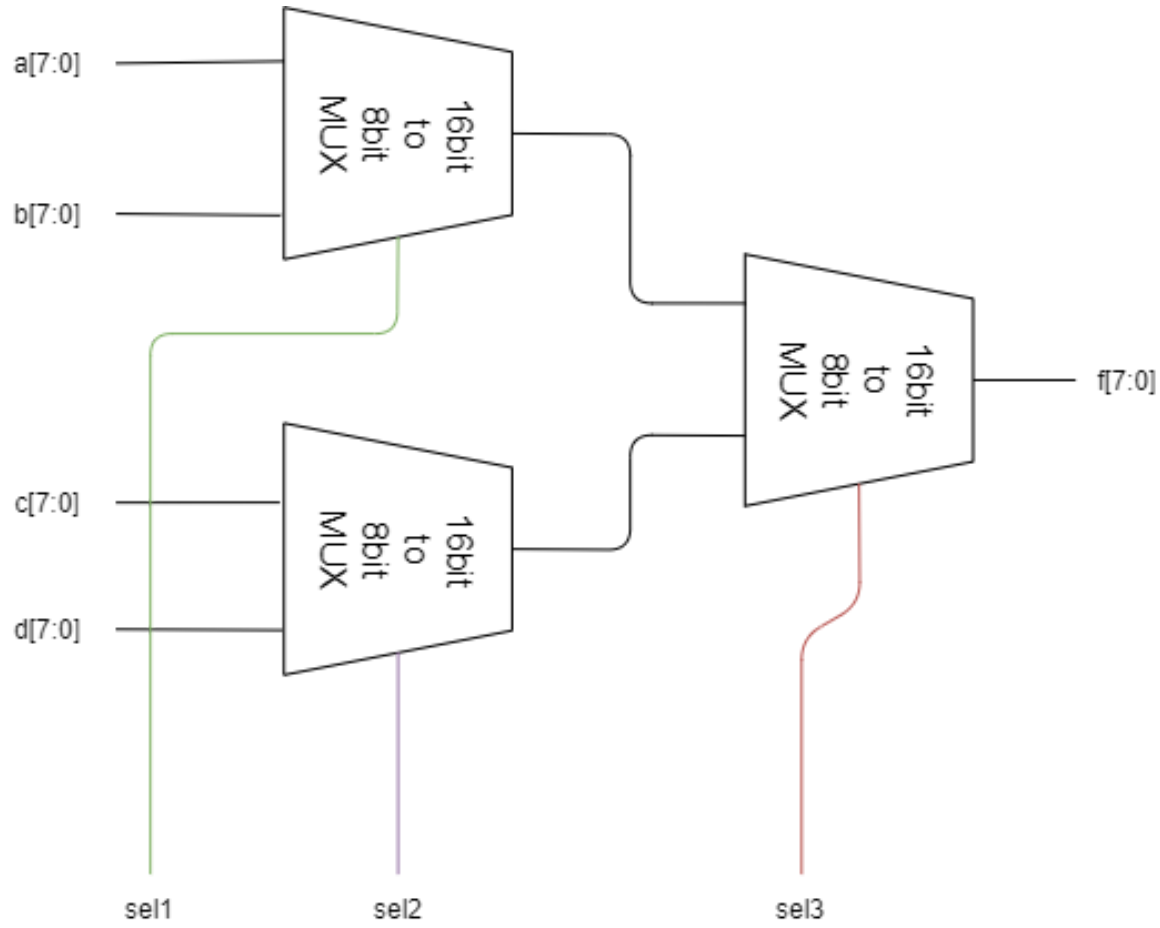
```
`timescale 1ns/1ps
module Mux_8bits (input [8-1:0]a, input [8-1:0]b, input [8-1:0]c,
input [8-1:0]d, input sel1,input sel2,input sel3, output [8-1:0]f);

wire n_sel1, n_sel2, n_sel3;
wire [8-1:0] a_and_sel, b_and_n_sel, c_and_sel, d_and_n_sel;
wire [8-1:0] a_second, b_second;
wire [8-1:0] a_second_and_n_sel, b_second_and_n_sel;

not (n_sel1, sel1);
not (n_sel2, sel2);
not (n_sel3, sel3);

and ( a_and_sel[0], a[0], sel1);
and ( a_and_sel[1], a[1], sel1);
and ( a_and_sel[2], a[2], sel1);
and ( a_and_sel[3], a[3], sel1);
and ( a_and_sel[4], a[4], sel1);
and ( a_and_sel[5], a[5], sel1);
and ( a_and_sel[6], a[6], sel1);
and ( a_and_sel[7], a[7], sel1);

and( b_and_n_sel[0], b[0], n_sel1);
and( b_and_n_sel[1], b[1], n_sel1);
and( b_and_n_sel[2], b[2], n_sel1);
and( b_and_n_sel[3], b[3], n_sel1);
and( b_and_n_sel[4], b[4], n_sel1);
and( b_and_n_sel[5], b[5], n_sel1);
and( b_and_n_sel[6], b[6], n_sel1);
and( b_and_n_sel[7], b[7], n_sel1);

and( c_and_sel[0], c[0], sel2);
and( c_and_sel[1], c[1], sel2);
and( c_and_sel[2], c[2], sel2);
and( c_and_sel[3], c[3], sel2);
and( c_and_sel[4], c[4], sel2);
and( c_and_sel[5], c[5], sel2);
and( c_and_sel[6], c[6], sel2);
and( c_and_sel[7], c[7], sel2);

and( d_and_n_sel[0], d[0], n_sel2);
and( d_and_n_sel[1], d[1], n_sel2);
and( d_and_n_sel[2], d[2], n_sel2);
and( d_and_n_sel[3], d[3], n_sel2);
and( d_and_n_sel[4], d[4], n_sel2);
and( d_and_n_sel[5], d[5], n_sel2);
and( d_and_n_sel[6], d[6], n_sel2);
and( d_and_n_sel[7], d[7], n_sel2);

or Or_Array_a_b_n[7:0] (a_second, a_and_sel, b_and_n_sel);
```

```
or Or_Array_c_d [7:0] (b_second, c_and_sel, d_and_n_sel);

and(a_second_and_n_sel[0], a_second[0], sel3);
and(a_second_and_n_sel[1], a_second[1], sel3);
and(a_second_and_n_sel[2], a_second[2], sel3);
and(a_second_and_n_sel[3], a_second[3], sel3);
and(a_second_and_n_sel[4], a_second[4], sel3);
and(a_second_and_n_sel[5], a_second[5], sel3);
and(a_second_and_n_sel[6], a_second[6], sel3);
and(a_second_and_n_sel[7], a_second[7], sel3);

and(b_second_and_n_sel[0], b_second[0], n_sel3);
and(b_second_and_n_sel[1], b_second[1], n_sel3);
and(b_second_and_n_sel[2], b_second[2], n_sel3);
and(b_second_and_n_sel[3], b_second[3], n_sel3);
and(b_second_and_n_sel[4], b_second[4], n_sel3);
and(b_second_and_n_sel[5], b_second[5], n_sel3);
and(b_second_and_n_sel[6], b_second[6], n_sel3);
and(b_second_and_n_sel[7], b_second[7], n_sel3);

or OR_ARR[7:0] (f, a_second_and_n_sel, b_second_and_n_sel);

endmodule
```

Implementation of 8 bit cascaded MUX



Implementation of 8 bit cascaded MUX



設計構想：

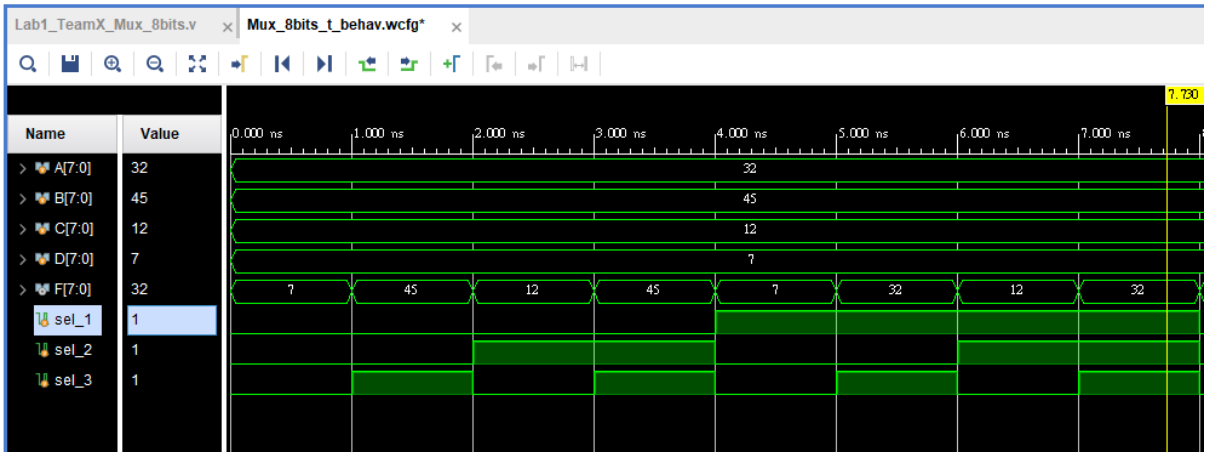
簡單來說就是做三個 8bit MUX，然後將sel訊號線接好就完成了，只是在 8bit MUX 實作上面，一開始忘記可以先把8bit MUX做成一個module來保持code的整潔，但是整體邏輯是一樣的，就只是把Code拆開了而已。

Testbench Code

```
`timescale 1ns / 1ps

module Mux_8bits_t;
reg [8-1:0]A, B, C, D;
wire [8-1:0] F;
reg sel_1 = 0, sel_2 = 0, sel_3 = 0;
Mux_8bits M8(.a(A), .b(B), .c(C), .d(D),
 .sel1(sel_1), .sel2(sel_2), .sel3(sel_3), .f(F));
initial begin
    A[8-1:0] = 8'd32;
    B[8-1:0] = 8'd45;
    C[8-1:0] = 8'd12;
    D[8-1:0] = 8'd7;

    repeat (2 ** 3) begin
         #1 {sel_1, sel_2, sel_3} = {sel_1, sel_2, sel_3} + 1'b1;
```

```
      end

end

endmodule
```

Wave of 8bit cascaded MUX



Testbench設計構想：

我覺得MUX的重點就是sel的排列組合有沒有確實的選擇到正確的input。所以在testbench的測試方面A,B,C,D的input值就沒有特別去改，簡單手打幾個不同的值，確定沒有接線錯就好。而三個sel的方面則是用repeat然後每次+1的方式來遍歷所有可能值，確定無誤即可。

開發過程中的問題/學習：

```
or OR_ARR[7:0] (f, a_second_and_n_sel, b_second_and_n_sel);
```

首先就是在下面整個verilog 中只要是類似上面這行Code的 gate array

本來是真的寫了 8 個 gate array 來實作，但是當下覺得應該有更好簡潔的寫法，查找了很多資料後也的確有找到，週二上課時教授也有直接點出gate array的寫法。

只是對於更general的狀況，其中只有1~n-1個是wire array的時候，也就是需要變的參數並非每個都有時，就不能使用上方的gate array 語法，例如以下

```
and(b_second_and_n_sel[0], b_second[0], n_sel3);
and(b_second_and_n_sel[1], b_second[1], n_sel3);
and(b_second_and_n_sel[2], b_second[2], n_sel3);
and(b_second_and_n_sel[3], b_second[3], n_sel3);
and(b_second_and_n_sel[4], b_second[4], n_sel3);
```

```
and(b_second_and_n_sel[5], b_second[5], n_sel3);
and(b_second_and_n_sel[6], b_second[6], n_sel3);
and(b_second_and_n_sel[7], b_second[7], n_sel3);
```

目前所找到的只有看到 genvar搭配 for迴圈的語法，比較能夠達到這種變數有限的狀況的想法，如下

```
genvar i;
generate
    for(i=0; i<8; i=i+1) begin:BLOCK1
        and(b_second_and_n_sel[i], b_second[i], n_sel3);
    end
endgenerate
```

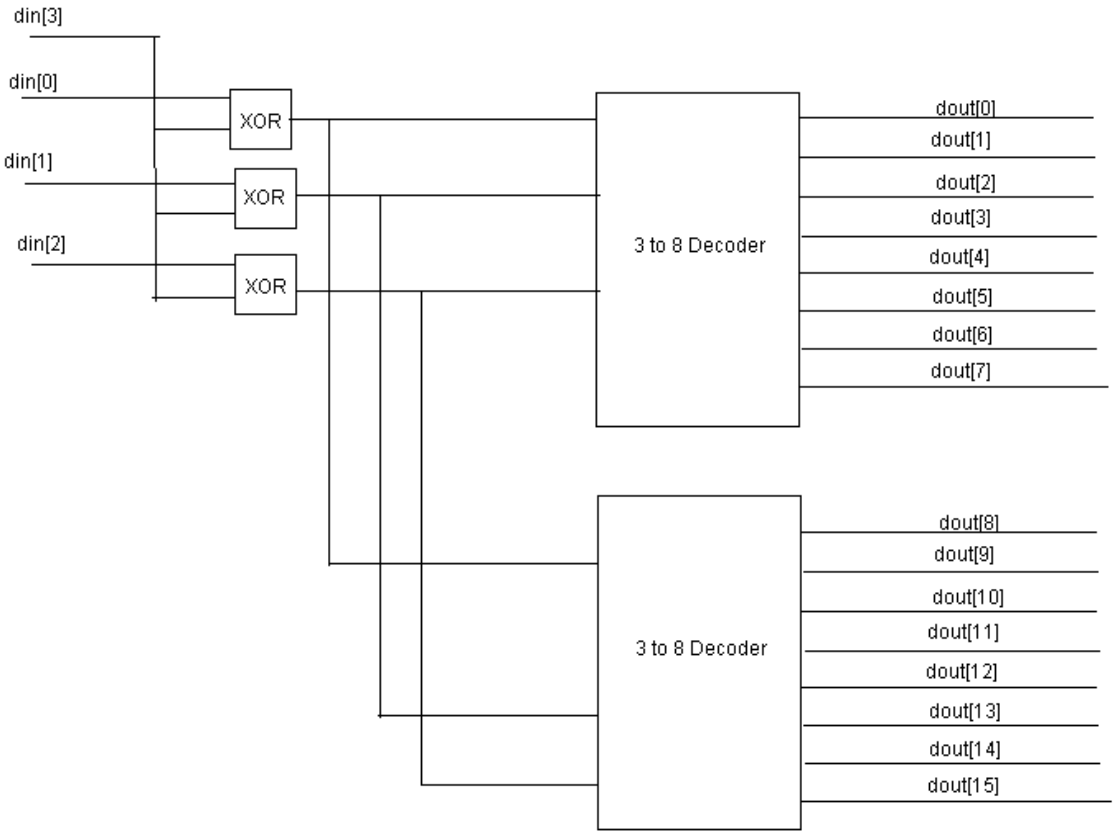
但是使用之前有在iLMS上面發問，教授的回應是迴圈語法有可能是無法合成的電路，意即使用迴圈會造成電路合成不出來的風險，於是就先棄用了。至於實際上真正的原因，可能還待之後課程慢慢了解。就我的猜測，verilog在解析generate語法時，其實應該也就是把它展開成n個gate level的描述，然後一個一個實作。

Verilog Question 2

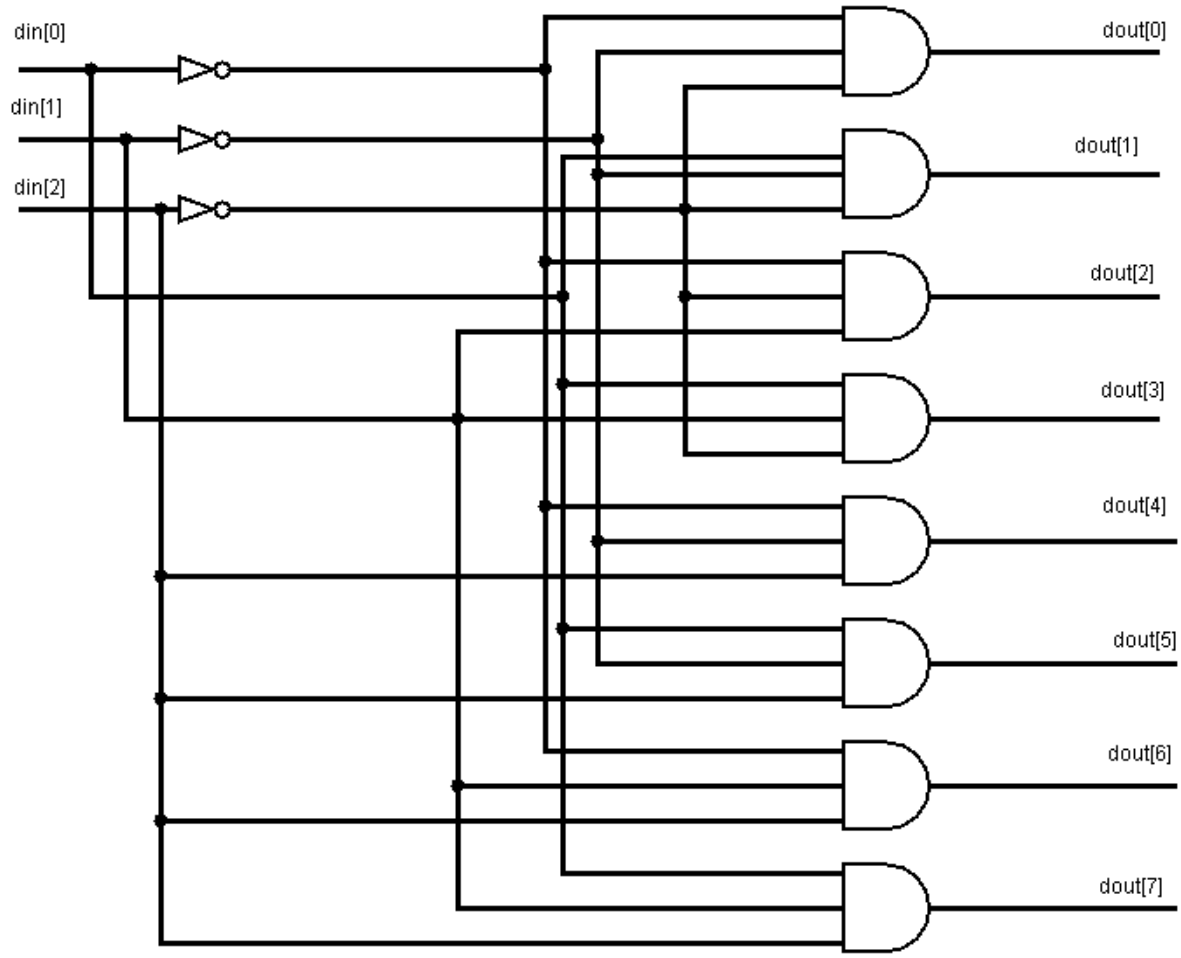
Verilog Code

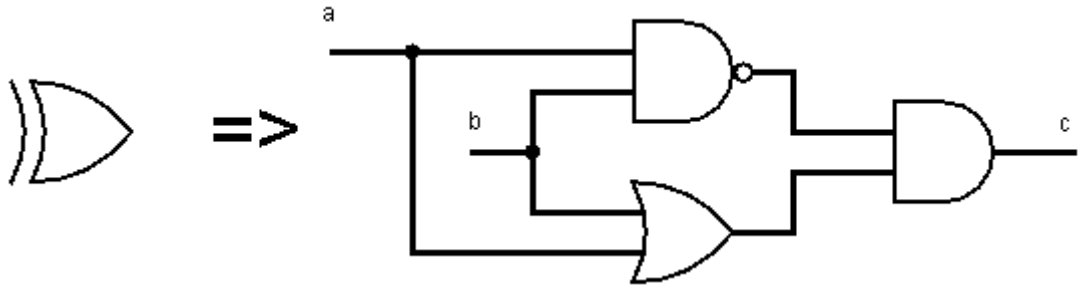
```
`timescale 1ns/1ps

module Decoder (din, dout);
input [4-1:0] din;
output [16-1:0] dout;
wire [2:0] din_n;
_xor x1(din[0], din[3], din_n[0]);
_xor x2(din[1], din[3], din_n[1]);
_xor x3(din[2], din[3], din_n[2]);
dec_3to8 d1(din_n, dout[15:8]);
dec_3to8 d2(din_n, dout[7:0]);


endmodule

module dec_3to8 (din, dout);
input [2:0] din;
output [7:0] dout;

wire [2:0] din_b;

not n1(din_b[0], din[0]);
not n2(din_b[1], din[1]);
not n3(din_b[2], din[2]);
```

```
and a1(dout[0], din_b[2], din_b[1], din_b[0]);
and a2(dout[1], din_b[2], din_b[1], din[0]);
and a3(dout[2], din_b[2], din[1], din_b[0]);
and a4(dout[3], din_b[2], din[1], din[0]);
and a5(dout[4], din[2], din_b[1], din_b[0]);
and a6(dout[5], din[2], din_b[1], din[0]);
and a7(dout[6], din[2], din[1], din_b[0]);
and a8(dout[7], din[2], din[1], din[0]);

endmodule

module _xor(a, b, c);
input a, b;
output c;
wire x1, x2, x3;
and a1(x1, a, b);
not n1(x2, x1);
or o1(x3, a, b);
and (c, x2, x3);

endmodule
```



3 to 8 Decoder





設計構想

觀察input and output可以發現output 分成兩組相同的訊號，

```
0000_0001_0000_0001
0000_0010_0000_0010
0000_0100_0000_0100
0000_1000_0000_1000
```

```
0001_0000_0001_0000
0010_0000_0010_0000
0100_0000_0100_0000
1000_0000_1000_0000
```

且若只看後3個bit分別為0到7及7到0對應0000_0001 X 2 ~ 1000_0000 X 2，而排列是逆序或順序則由第1個bit決定。於是先用第1個bit XOR 其他3個bit(0 xor a = a ; 1 xor a = a_bar)，再用兩個　3to8Decoder輸出output。

```
0000 1111 => 0000_0001
0001 1110 => 0000_0010
0010 1101 => 0000_0100
0011 1100 => 0000_1000
0100 1011 => 0001_0000
0101 1010 => 0010_0000
0110 1001 => 0100_0000
0111 1000 => 1000_0000
```

Testbench Code

```
`timescale 1ns / 1ps

module Dec_testbench;
reg [3:0]din;
wire [15:0]dout;
Decoder d(din, dout);
initial
begin
    din = 4'b0000;
    repeat (16)
    begin
        #10 din = din + 4'b0001;
    end
    $finish;
end

endmodule
```

repeat16次讓din從0000跑到1111測試所有可能input

開發過程中的問題/學習

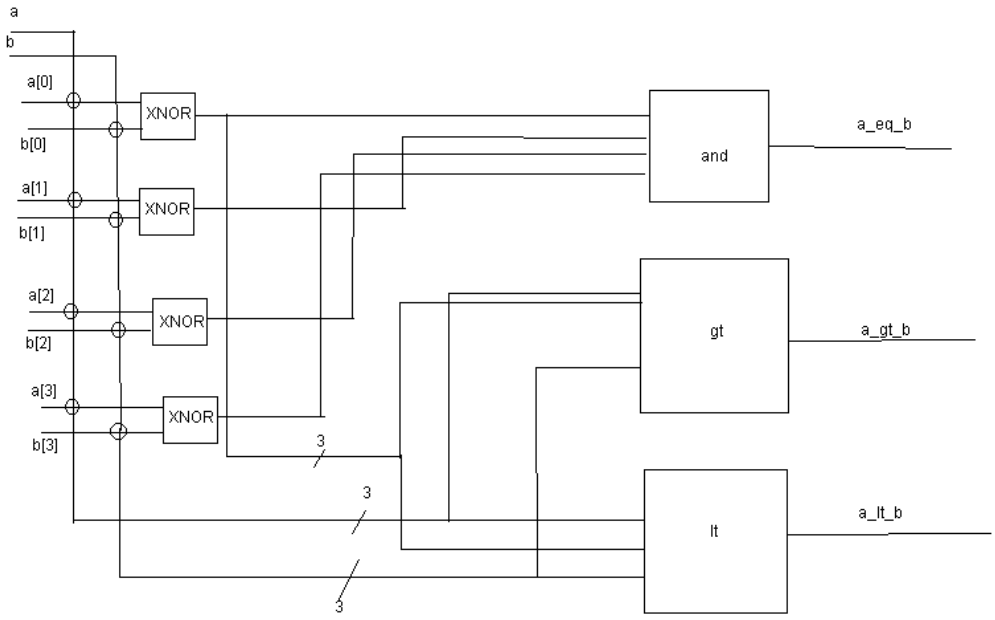
一開始以為要用K-map直接暴力解開，但覺得這樣太繁瑣，仔細觀察才發現input 和 output 的規律，如果拆成更小的問題，就只需要寫那些子問題的module，再把那些module串起來就好，大大增加可讀性，debug也比較容易。

Verilog Question 3

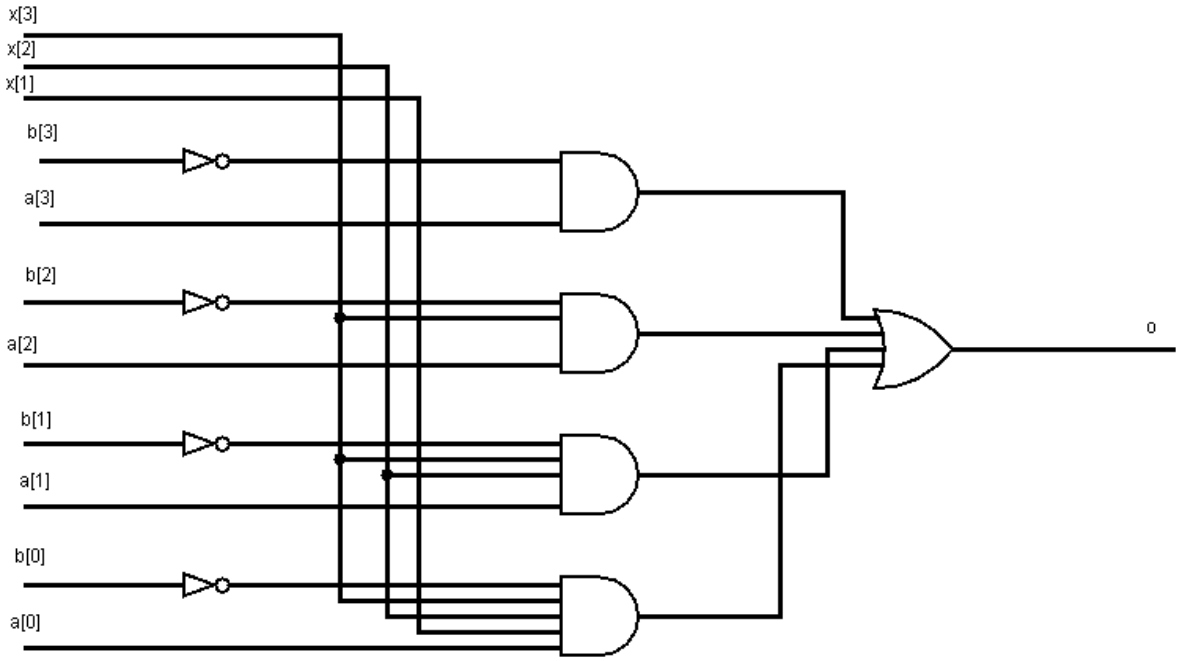
Verilog Code

```
`timescale 1ns/1ps

module Comparator_4bits (a, b, a_lt_b, a_gt_b, a_eq_b);
input [4-1:0] a, b;
output a_lt_b, a_gt_b, a_eq_b;
wire [3:0] x_res;

_xnor x1(a[0], b[0], x_res[0]);
_xnor x2(a[1], b[1], x_res[1]);
_xnor x3(a[2], b[2], x_res[2]);
_xnor x4(a[3], b[3], x_res[3]);

and and1(a_eq_b, x_res[0], x_res[1], x_res[2], x_res[3]);

gt gt1(a, b, x_res, a_gt_b);

lt lt1(a, b, x_res, a_lt_b);



endmodule

module gt(a, b, x, o);
input [3:0] a, b, x;
output o;
wire a_0, a_1, a_2, a_3, b3_b, b2_b, b1_b, b0_b;
not n1(b3_b, b[3]);
not n2(b2_b, b[2]);
not n3(b1_b, b[1]);
not n4(b0_b, b[0]);

and a1(a_3, b3_b, a[3]);
and a2(a_2, b2_b, a[2], x[3]);
and a3(a_1, b1_b, a[1], x[3], x[2]);
and a4(a_0, b0_b, a[0], x[3], x[2], x[1]);

or o1(o, a_3, a_2, a_1, a_0);

endmodule

module lt(a, b, x, o);
input [3:0] a, b, x;
output o;
wire a_0, a_1, a_2, a_3, a0_b, a1_b, a2_b, a3_b;

not not1(a3_b, a[3]);
not not2(a2_b, a[2]);
not not3(a1_b, a[1]);
not not4(a0_b, a[0]);
```

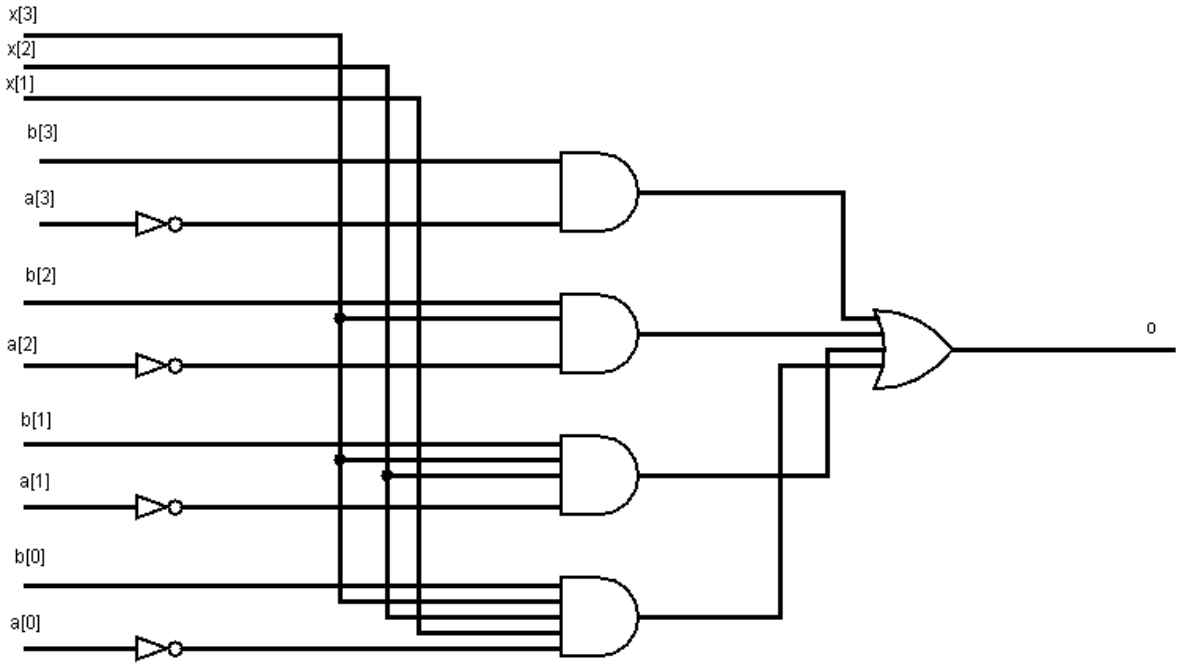
```
and a1(a_3, a3_b, b[3]);
and a2(a_2, a2_b, b[2], x[3]);
and a3(a_1, a1_b, b[1], x[3], x[2]);
and a4(a_0, a0_b, b[0], x[3], x[2], x[1]);

or o1(o, a_0, a_1, a_2, a_3);

endmodule

module _xnor(a, b, c);
input a, b;
output c;
wire x1, x2, x3, c_b;
and a1(x1, a, b);
not n1(x2, x1);
or o1(x3, a, b);
and (c_b, x2, x3);
not (c, c_b);

endmodule
```

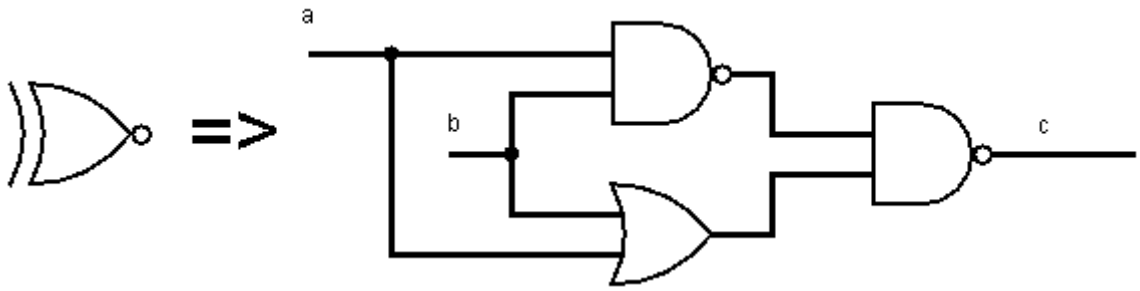


gt module



lt module





設計構想

先XNOR a, b的每一位bit 得出各個bit是否相同存為x_res，AND x_res[0 ~ 3] 得出a_eq_b。

gt module 是從i = 3到0逐個bit檢查a[i]是否>b[i]且a[k] == b[k](3 > k > i)，因為若前面不相等，則兩數大小關係已經在之前的過程中確立，最後用OR是因為只有從高往低的第一個不相同的bit會被檢測

大小關係。而lt module 和 gt module 只差在是檢查a[i]是否 < b[i]。

Testbench Code

```
`timescale 1ns / 1ps

module comp_testbench;
reg [3:0] a, b;
wire a_lt_b, a_gt_b, a_eq_b;
Comparator_4bits c(a, b, a_lt_b, a_gt_b, a_eq_b);
initial
begin
    a = 4'b0000;
    b = 4'b0000;
    repeat (16)
    begin
        repeat (16)
        begin
            #1 a = a + 4'b0001;
        end
        #1 b = b + 4'b0001;
    end
    $finish;
end

endmodule
```

跑16X16遍，測試a 0000 ~ 1111 及 b 0000 ~ 1111 的所有組合

開發過程中的問題/學習

原本是沒有gt module 和 lt module ，但是畫 circuit 的時候發現線會牽得太亂，於是改成多module的方式，再次體會到多module 設計方式的優點。

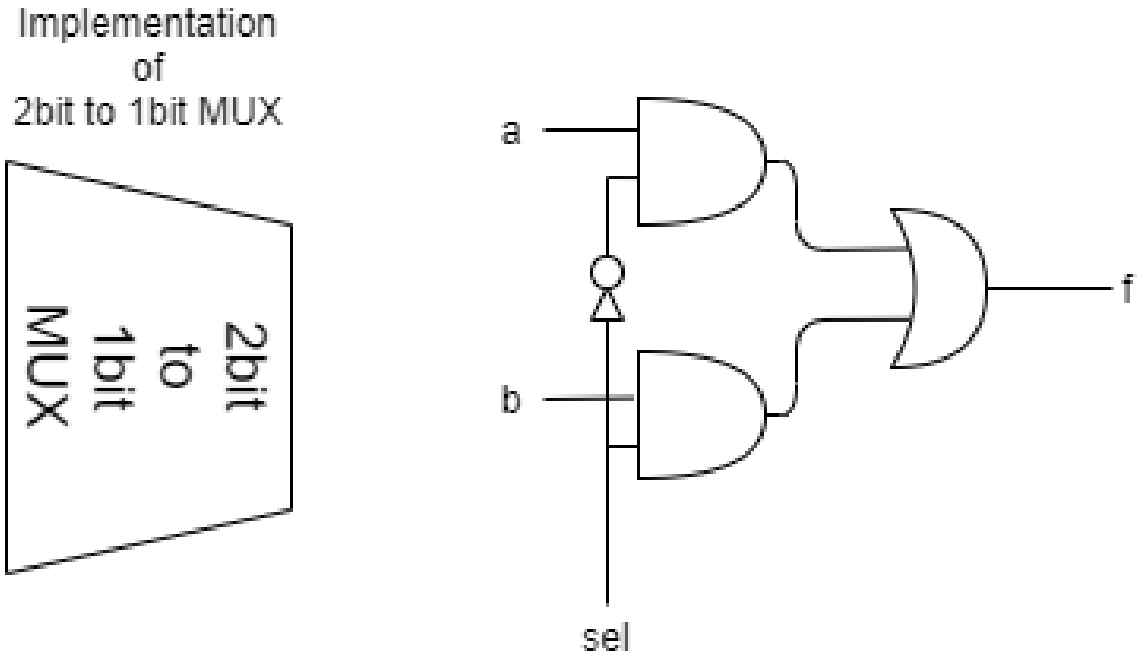
Verilog Question 4

Verilog Code

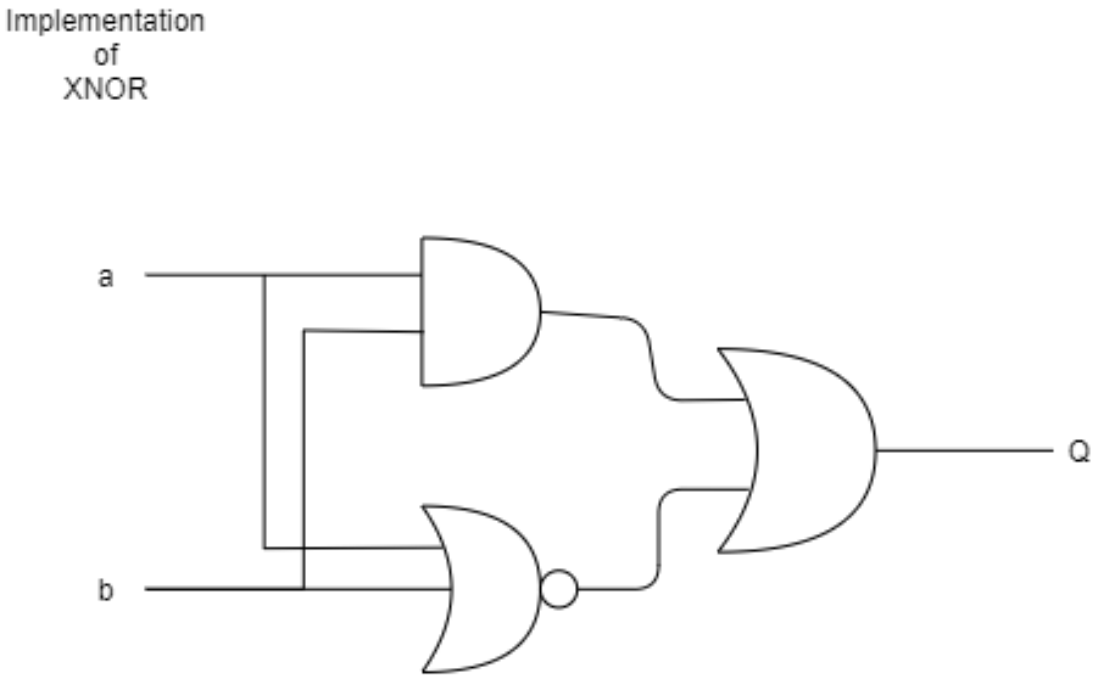
```
`timescale 1ns/1ps

module RippleCarryAdder (input [8-1:0] a, input [8-1:0]b,
input cin, input cout, output [8-1:0] sum);
wire [7-1:0] c;

    FullAdder A0(a[0], b[0], cin, c[0], sum[0]);
    FullAdder A1(a[1], b[1], c[0], c[1], sum[1]);
    FullAdder A2(a[2], b[2], c[1], c[2], sum[2]);
    FullAdder A3(a[3], b[3], c[2], c[3], sum[3]);
    FullAdder A4(a[4], b[4], c[3], c[4], sum[4]);
    FullAdder A5(a[5], b[5], c[4], c[5], sum[5]);
    FullAdder A6(a[6], b[6], c[5], c[6], sum[6]);
    FullAdder A7(a[7], b[7], c[6], cout, sum[7]);

endmodule

module FullAdder (a, b, cin, cout, sum);
input a, b, cin;
output sum;
output cout;
wire a, b, cin;
wire sum, cout;
wire XNOR_a_b;
    XNOR xnor_a_b(a, b, XNOR_a_b);
    XNOR xnor_cin_xnorab(cin, XNOR_a_b, sum);
    Mux_1bit gecout(a, cin, XNOR_a_b, cout);
endmodule

module XNOR (a, b, Q);
    input a, b;
    output Q;
    wire a, b;
    wire Q;
    wire nor_a_b, and_a_b;
    nor NOR_(nor_a_b, a, b);
    and AND_(and_a_b, a, b);
    or OR_(Q, and_a_b, nor_a_b);
endmodule

module Mux_1bit (a, b, sel, f);
input a, b;
input sel;
output f;
```

```
    wire a, b, sel;
    wire f;
    wire and_a_sel, and_b_sel;
    wire notsel;
    not NOT_1(notsel, sel);
    and AND_(and_a_sel, a, sel);
    and AND_2(and_b_sel, b, notsel);
    or OR_(f, and_a_sel, and_b_sel);
endmodule
```

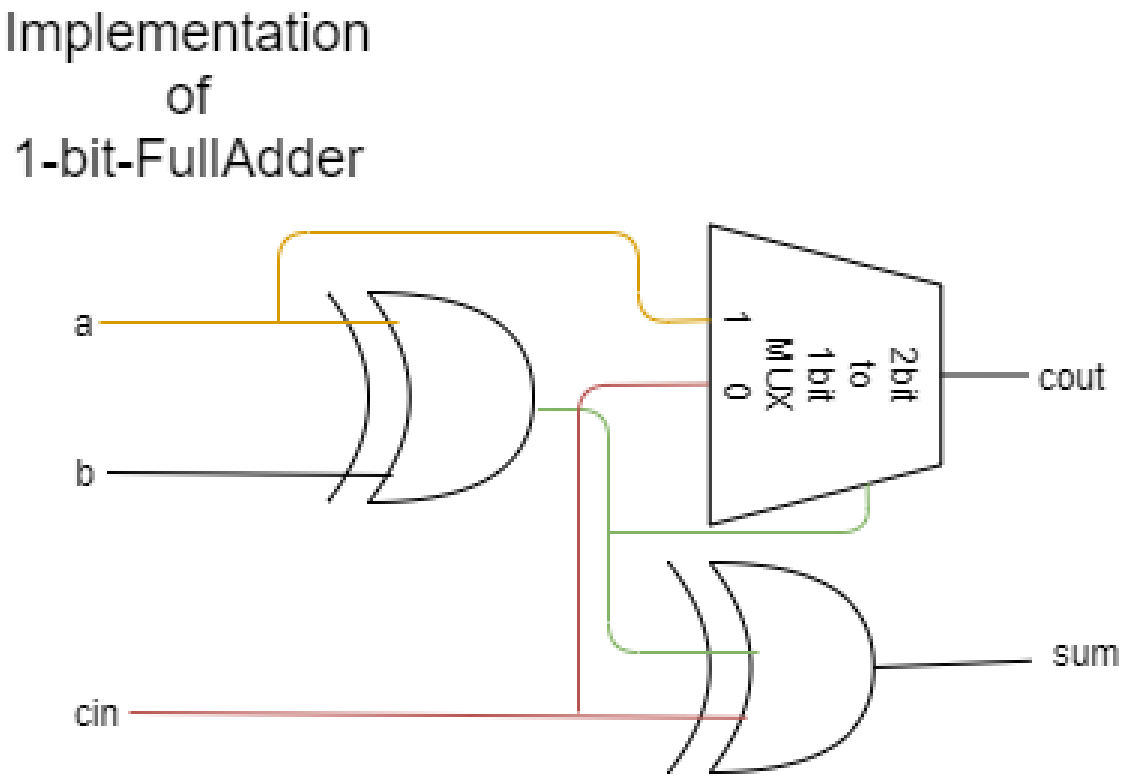
Implementation of 1bit MUX



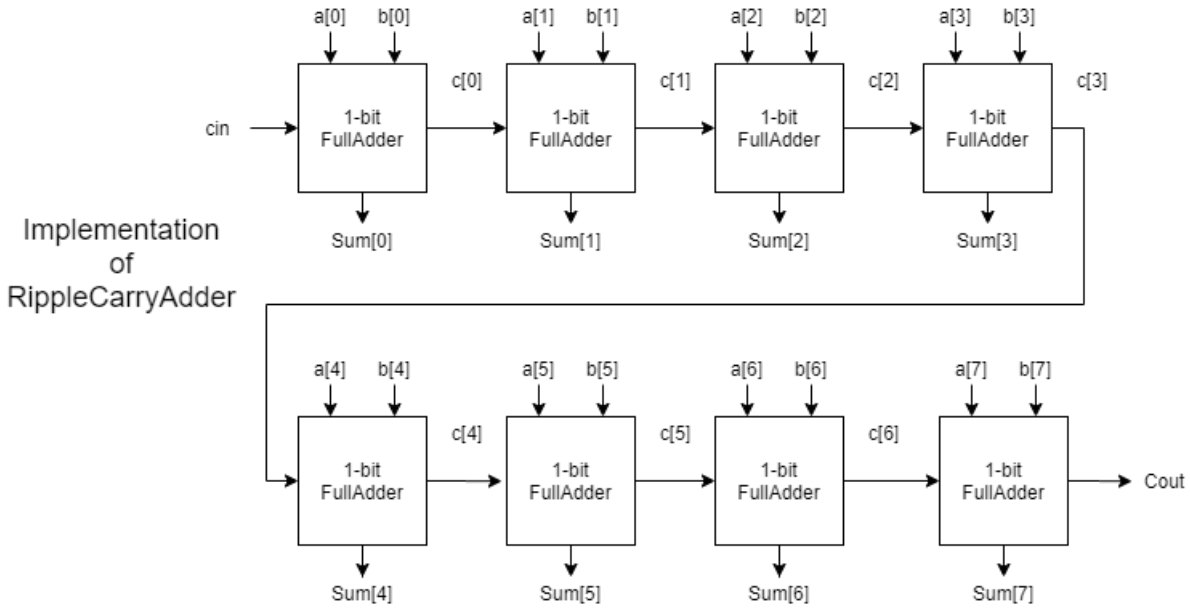
Implementation of XNOR



Implementation of 1-bit-FullAdder



Implementation of RippleCarryAdder



設計構想：

1-bit FullAdder的結構基本上跟basic Question的寫法一模一樣，是沿用那邊的code過來的。
先做出一個XNOR，再用XNOR做出1bit-FullAdder，再將每個1bit-FullAdder用c[6:0]接起來。

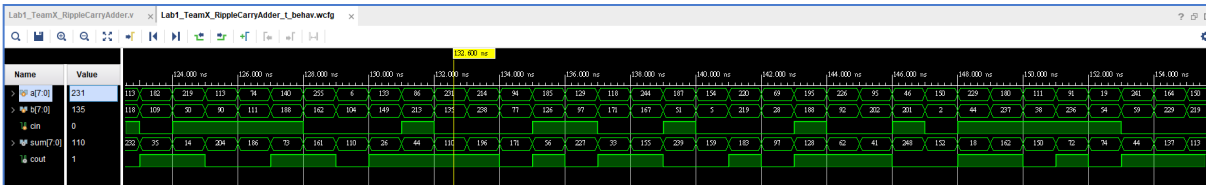
接著就在RipleCarrayAdder裏頭用c[6:0]的wire 當作每一位進位的bit，接到下一個FullAdder的cin位置，每個FullAdder的輸出接到Sum[7:0]上面。

(這邊為了明確表達進位的方向，改用有箭頭的邊。)

Testbench Code

```
`timescale 1ns / 1ps

module Lab1_TeamX_RippleCarryAdder_t;
reg [7:0] a, b;
reg cin;
wire [7:0] sum;
wire cout;
RippleCarryAdder RCA(a, b, cin, cout, sum);


initial begin
    {a[7:0], b[7:0], cin} = 17'b0;
    repeat (2 ** 30) begin
        #1 {a[7:0], b[7:0], cin} =  $random;
    end
    #10 $finish;
end
```

```
endmodule
```



testbench設計構想：

這邊為了確定FullAdder是正常運作，我用repeat了2^30次(這邊僅擷取部分)，每次都將 a[7:0],b[7;0],cin 換成一個random的值

```
#1 {a[7:0], b[7:0], cin} =  $random;
```

構造出許多不同的值來測試，只要sum + cout * 256 = a + b，就表RippleCarryAdder是正常運作的。

開發過程中的問題/學習：

學到了verilog 竟然有 $random這麼好用的東西，甚至不用include函式庫之類的，不過看起來random放進來也是需要另外設置seed，這次作業就沒有加入seed了，可能下次會再學習後加入。

分工合作：

高敦晉：

負責 Advanced Question 1 & Question 4，Question 3 燒進BASYS 3 的fanout 改寫，Q1 report、Q4 report。

校對 Question 2、Question 3 Code

陳皇佑：

負責 Advanced Question 2 & Question 3，Q3 report、Q4 report。

校對 Question 1、Question 4 Code