Advanced Question 1

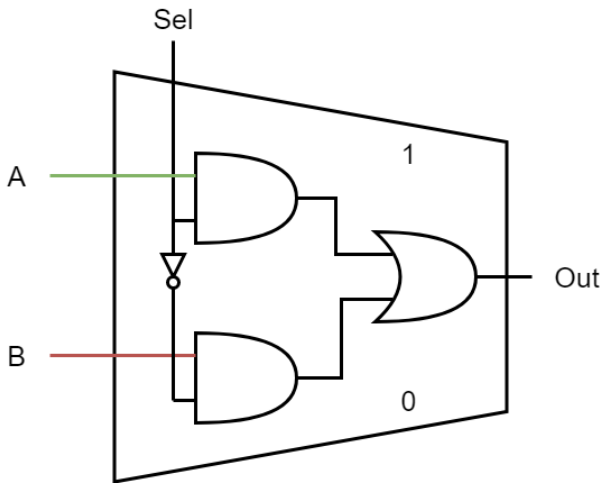
使用 8-bit Multiplexer 實作切換輸出的效果，邏輯圖如下：

Sel 3	Sel 1, Sel 2 = 00	01	11	10
1	B[7:0]	B[7:0]	A[7:0]	A[7:0]
0	D[7:0]	D[7:0]	C[7:0]	C[7:0]

而其中需要注意的是，Sel1 與 Sel2 同時只會有一個有實際上的作用（取決於 Sel3）。

Step 1：

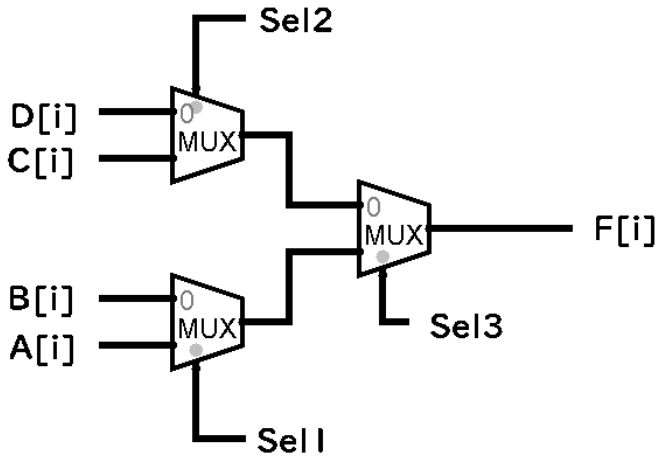
實作 2-to-1 Multiplexer



for i = 0, 1, ... , 7

Step 2：

對於 A, B, C, D, F 的每個位元，做出 4-to-1 Multiplexer



Question 1 testbench

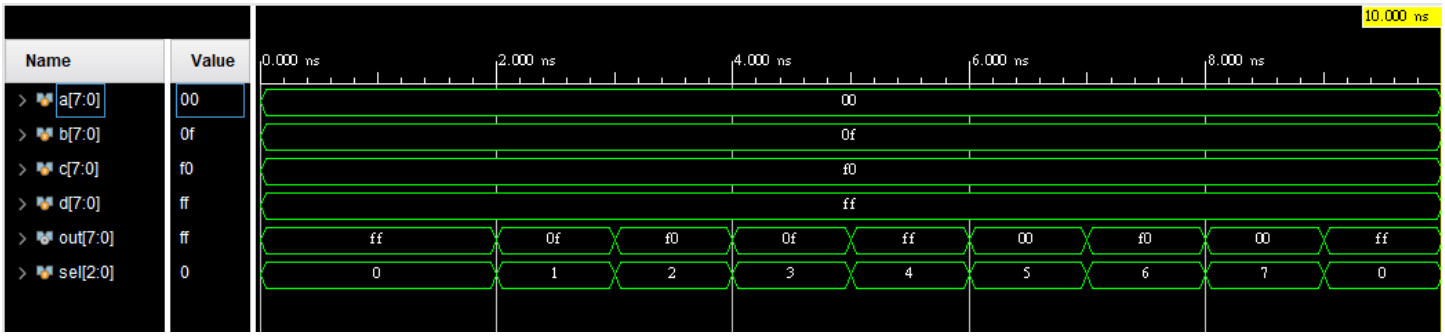
```
initial begin
    a = 8'b00000000;
    b = 8'b00001111;
    c = 8'b11110000;
    d = 8'b11111111;
    sel = 3'b000;
end
```

這題 testbench 的重點在於 Sel 的切換，與 A, B, C, D 的 value 數值沒有多大的關係。因此我把 A, B, C, D 放入不同的數值

```
initial begin
    #1
    repeat (2 ** 3) begin

        #1 sel = sel + 1'b1;

    end

    #1 $finish;
end
```

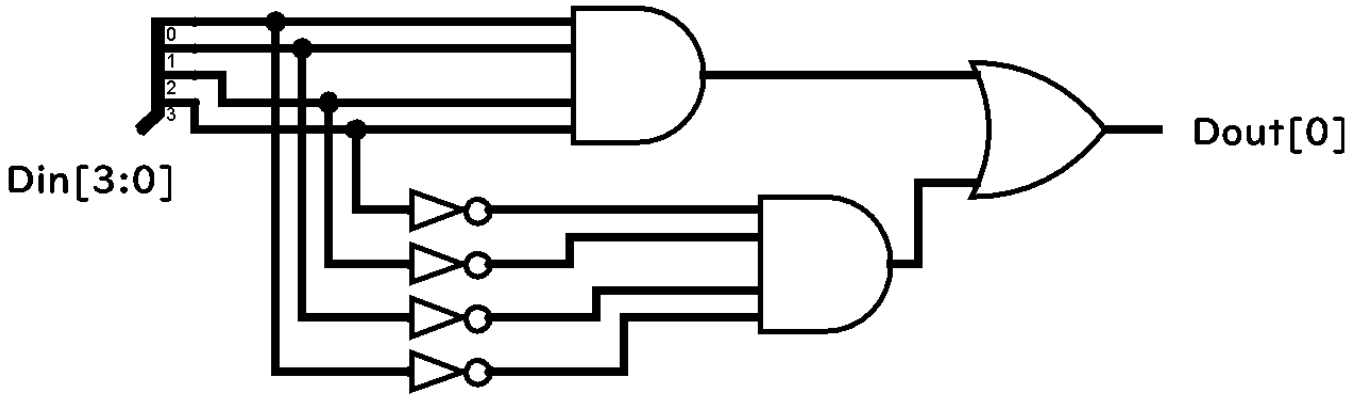
然後在 repeat 的過程中，觀察 F 的數值是否會隨著 Sel 1, 2, 3 而變化。

下圖為模擬結果。



Advanced Question 2

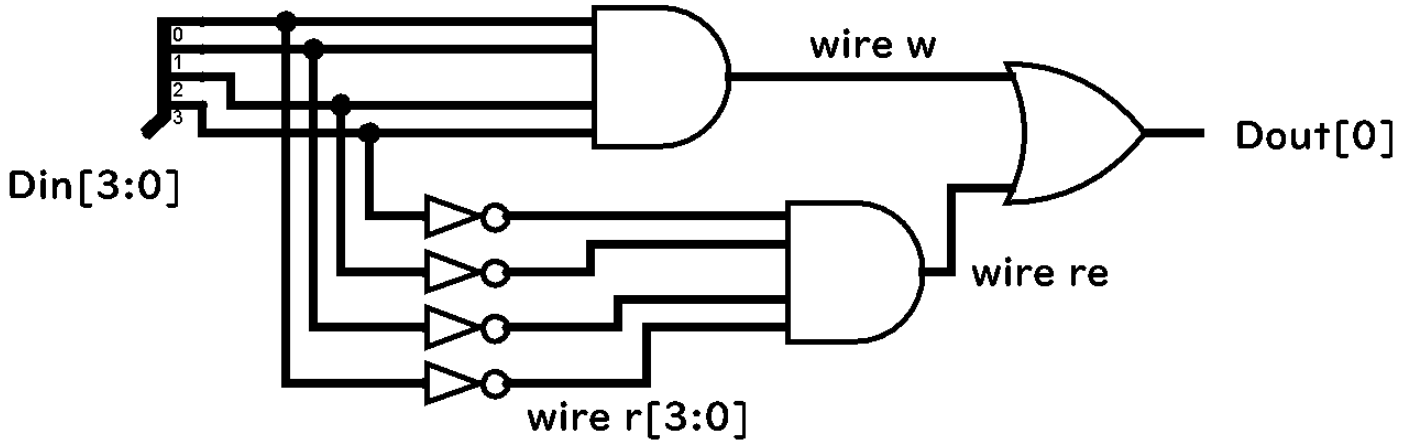
從觀察可以發現，0~7 與 8~15 是一樣的規律，因此針對 0~7 寫邏輯式，就可以解出這題。拿 Dout[0]來舉例，可以發現在 1111 與 0000 兩種情況的結果會是 1，因此可以寫出下圖這種電路。



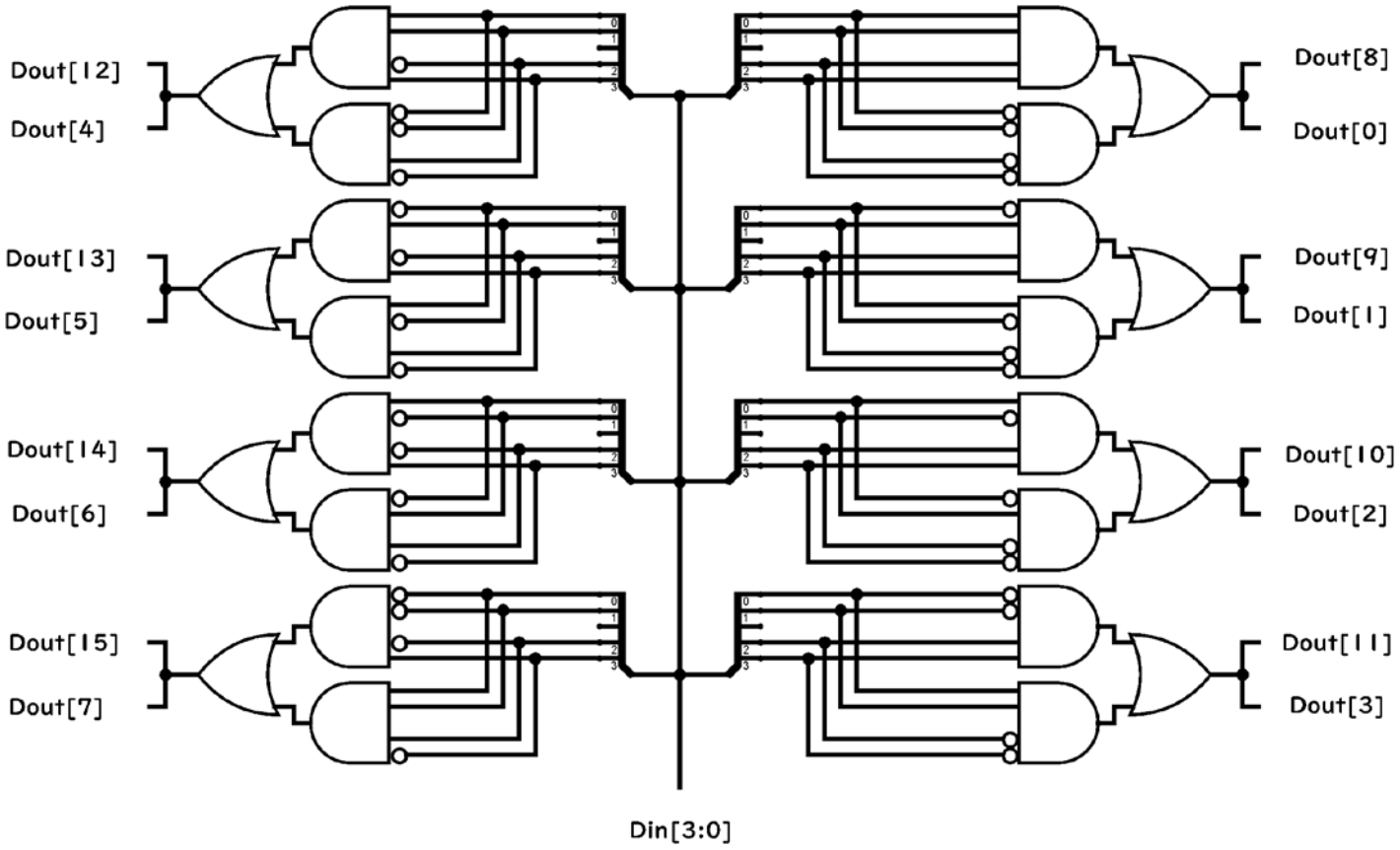
Dout[7:1]也是一樣的方法，寫成 verilog 的程式碼如下圖：

```
module Decoder (din, dout);

    input [4-1:0] din;
    output [16-1:0] dout;

    wire [3:0] r;
    not not1(r[3], din[3]);
    not not2(r[2], din[2]);
    not not3(r[1], din[1]);
    not not4(r[0], din[0]);

    wire [7:0] w;
    wire [7:0] re;

    and and1(w[7], r[3], din[2], din[1], din[0]);
    and and2(re[7], din[3], r[2], r[1], r[0]);
    or or1(dout[15], w[7], re[7]);
    or or1_2(dout[7], w[7], re[7]);

    and and3(w[6], r[3], din[2], din[1], r[0]);
    and and4(re[6], din[3], r[2], r[1], din[0]);
    or or2(dout[14], w[6], re[6]);
    or or2_2(dout[6], w[6], re[6]);
```

可以參照下圖更了解各個 wire 的意義！



最後的邏輯圖如下：

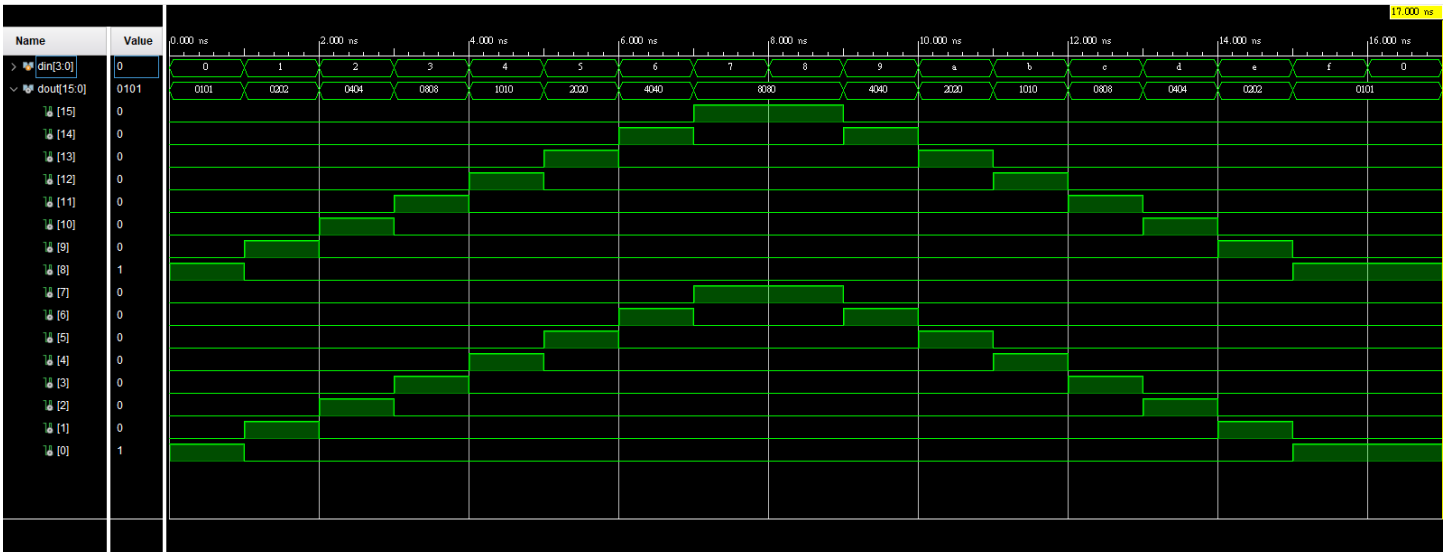


Question 2 testbench

```
module Decoder_t;
reg [3:0] din = 4'b0;
wire [15:0] dout;

Decoder d1( .din(din), .dout(dout));

initial begin
    repeat (2**4) begin

        #1 din = din + 1'b1;

    end

    #1 $finish;
end
endmodule
```

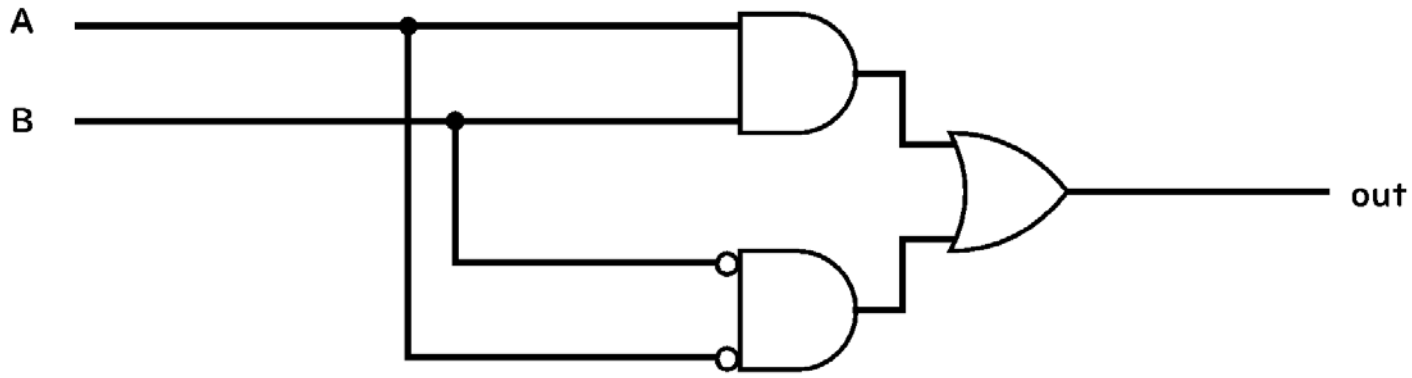
Q2 的 testbench 稍微簡單了一點，只要使用 repeat 跑過 Din 所有的可能性（16 種），觀察 Dout 的結果是否正確即可。

下圖為模擬結果。



Advanced Question 3

Q3 需要我們比較兩個 unsigned 4-bit 的數字，先從 A_eq_B 來思考，當 A 跟 B 一樣的時候，每個 bit 必然會一樣，因此，我們可以實作 XNOR 來判斷 bit 的值是不是一樣：



當 A、B 都是 1 或都是 0 時（i.e. A == B），out 為 1，否則為 0。

這樣子我們就可以接出 A_eq_B 的電路，只要判斷 A 與 B 的每個位元是不是 XNOR 的結果都是 1 即可。

再來實作 A_gt_B，從直觀上來想 A > B 的條件是：

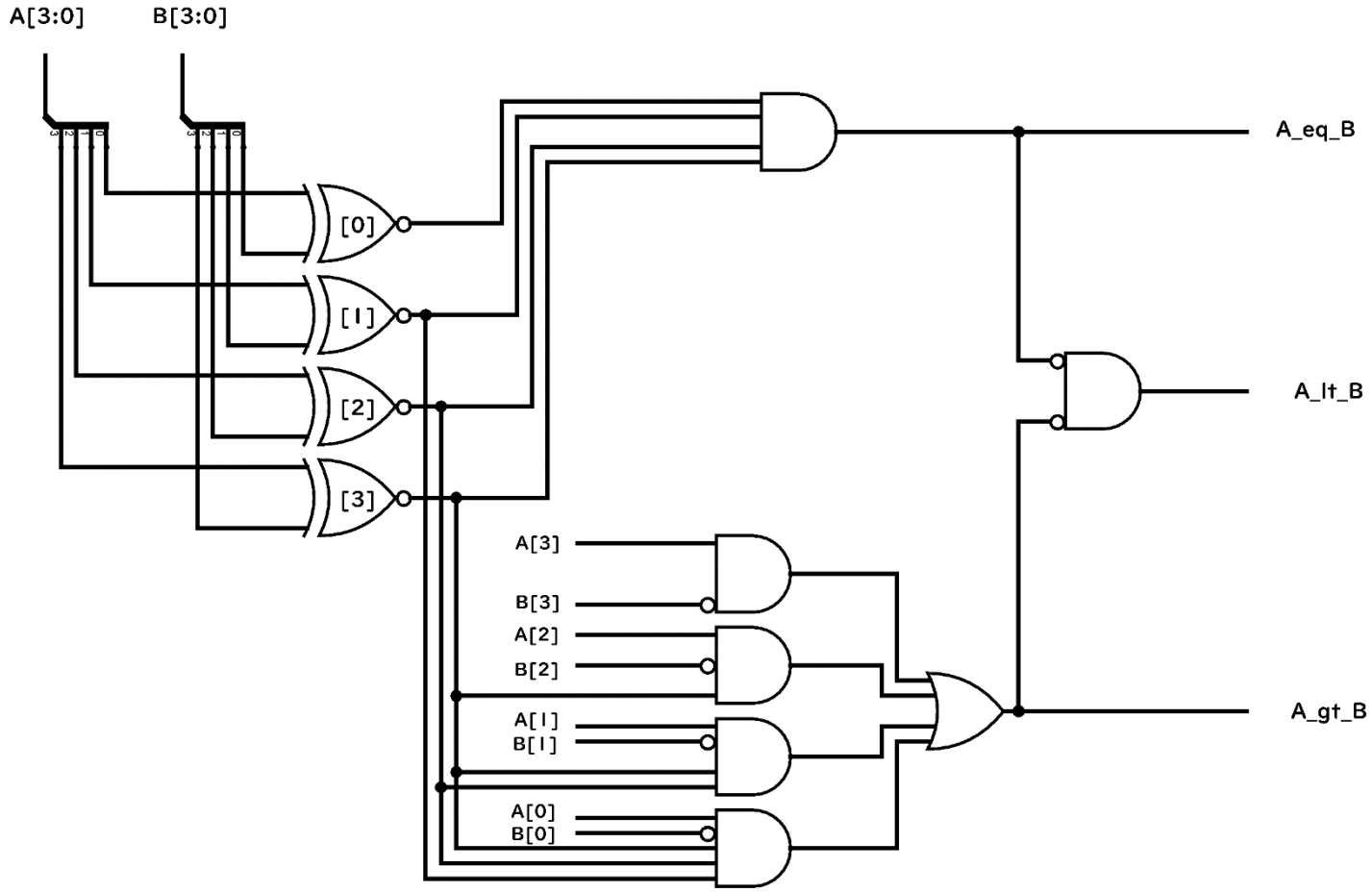
1. 如果 A[3] > B[3]，就可以得到 A > B 的結論。
2. 如果 A[3] = B[3]的話，則再往下一個 bit（A[2]與 B[2]）做比較。
3. 往下如此反覆，直到比較完 A[0]與 B[0]為止。

所以，A_gt_B 的 result 為 4 個情況通過 or gate 所得到的結果：

1. A[3] > B[3]
2. A[3] == B[3] && A[2] > B[2]
3. A[3:2] == B[3:2] && A[1] > B[1]
4. A[3:1] == B[3:1] && A[0] > B[0]

最後的 A_lt_B，可以將 A_gt_B 反過來做，我這裡的寫法是當 A_eq_B 與 A_gt_B 都為 0 時，A_lt_B 即為 1（不等於也不大於，那就是小於嘛）。

最後的 circuit 實作如下：



Question 3 testbench

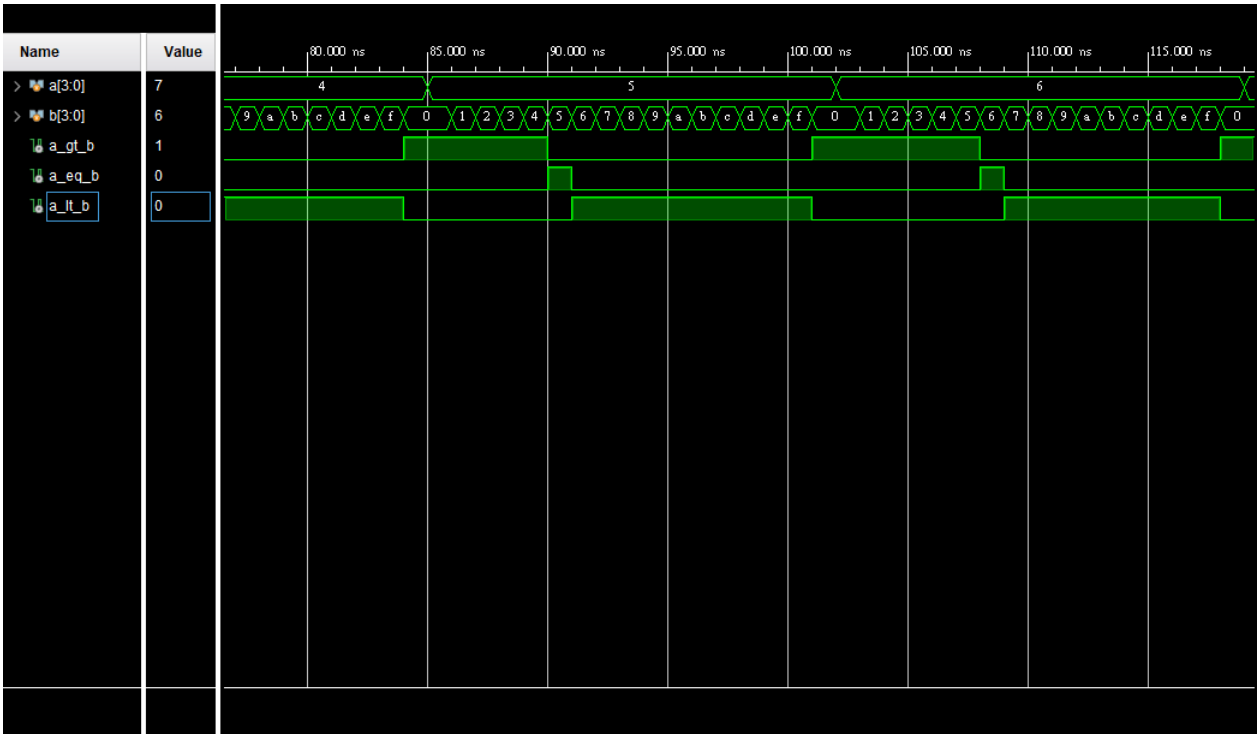
因為 A 與 B 都是 4-bit unsigned integer，所以所有的可能性只有 256 種（16*16），全部列出來看結果正不正確也是可行的方法。

Testbench 如下：

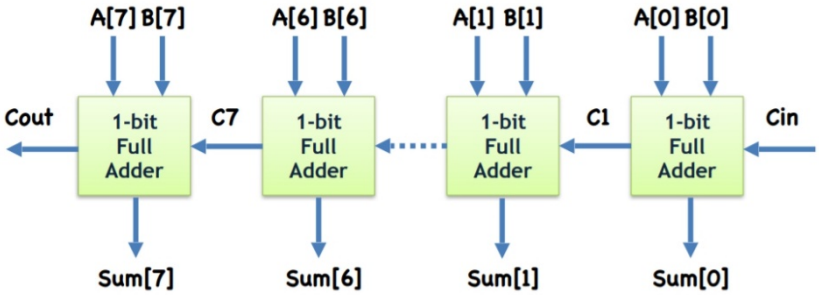
```
reg [3:0] a = 4'b0;
reg [3:0] b = 4'b0;
wire a_eq_b;
wire a_gt_b;
wire a_lt_b;
```

```
initial begin
    repeat (2**4) begin

        repeat (2**4)begin

            #1 b = b+1;

        end

        #1 a = a+1;
    end

    #1 $finish;
end
```

下圖為擷取的部分模擬結果：

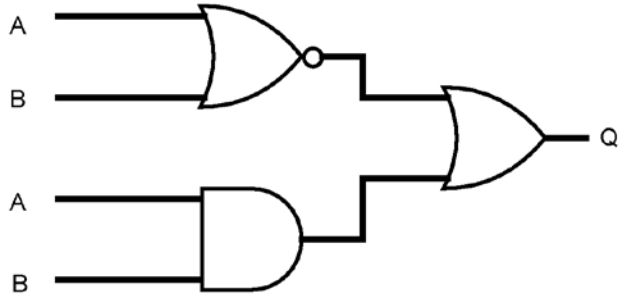


Advanced Question 4

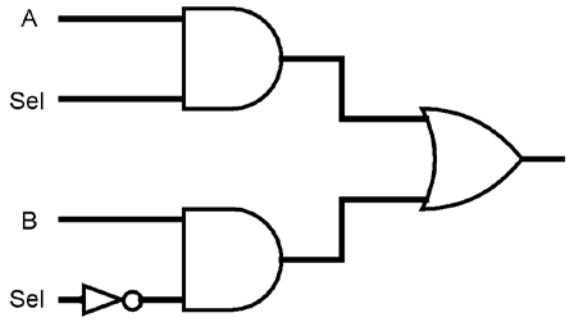


8-bit ripple-carry adder 由 8 個 1-bit 的 full adder 接在一起

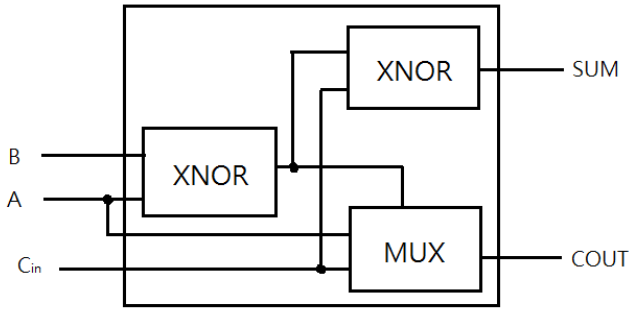
那麼就先把一個 1-bit 的 full adder 給做出來，用兩個 XNOR 跟一個 MUX



XNOR 的部分



MUX 的部分



1-bit full adder

```
module XNOR(a,b,cout);
input a,b;
output cout;

wire aandb,notab,nota,notb;

not n1(nota,a),n2(notb,b);
and a1(aandb,a,b),a2(notab,nota,notb);
or o1(cout,aandb,notab);

endmodule
```

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

wire nots,mix1,mix2;

not n1(nots,sel);
and a1(mix1,a,sel),a2(mix2,b,nots);
or o1(f,mix1,mix2);

endmodule
```

把 XNOR 的部分以及 MUX 的部分額外做出 module 來，這樣比較容易操作

然後因為 8-bit ripple-carry adder 是 8 個 1-bit 的 full adder 串接在一起，所以這裡先用出一個 1-bit full adder 的 module，這樣到時候接在一起就很快

```
module one_bit_fulladder(a,b,cin,cout,sum);
input a,b;
input cin;
output cout;
output sum;

wire q1;
XNOR x1(a,b,q1);
XNOR x2(q1,cin,sum);
MUX m1(a,cin,q1,cout);

endmodule
```

8-bit ripple-carry adder 的每一個 cin 都是前一個 full adder 產生的 cout(除了第一個以外)，所以這裡有 cin2,cin3,cin4,cin5,cin6,cin7,cin8, 這些 wire 來當作前一個 full adder 產生的 cout，然後就可以做為下一個 full adder 的 cin，這樣就完成 8-bit ripple-carry adder

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

wire cin2,cin3,cin4,cin5,cin6,cin7,cin8;

one_bit_fulladder r0(a[0],b[0],cin,cin2,sum[0]);
one_bit_fulladder r1(a[1],b[1],cin2,cin3,sum[1]);
one_bit_fulladder r2(a[2],b[2],cin3,cin4,sum[2]);
one_bit_fulladder r3(a[3],b[3],cin4,cin5,sum[3]);
one_bit_fulladder r4(a[4],b[4],cin5,cin6,sum[4]);
one_bit_fulladder r5(a[5],b[5],cin6,cin7,sum[5]);
one_bit_fulladder r6(a[6],b[6],cin7,cin8,sum[6]);
one_bit_fulladder r7(a[7],b[7],cin8,cout,sum[7]);

endmodule
```

Question 4 testbench

拿 A、B 來測試是否正確，總共做 32 次，每一次 A + 8'd5，B + 8'd11，並且 cin 在 1 跟 0 之間轉換，以此觀察結果是否正確。

```
initial begin

    a = 8'd0;
    b = 8'd0;
    cin = 1'd0;

    repeat (2 ** 5) begin
        #1

        a = a + 8'd5;
        b = b + 8'd11;
        cin = ~cin;

    end

    #1 $finish;
end
```

Teamwork：

	Q1	Q2	Q3	Q4
Module	莊景堯	莊景堯	莊景堯	林諭震
Testbench	莊景堯	林諭震	林諭震	莊景堯

What we learn：

1. 重新複習了一次大一下所學到的各種 Verilog 語法。
2. 課堂上學到了 Gate array 的寫法！
3. 如何使用 Vivado，以及如何燒錄到 FPGA 內。
4. 如何分配時間與分工，為了能夠準時交作業……