Lab 2 Report Team 10

Course : Logic Design Laboratory
Date : 2020/10/7

108062125 高敦晉 108062229 陳皇佑

Basic Questions 1

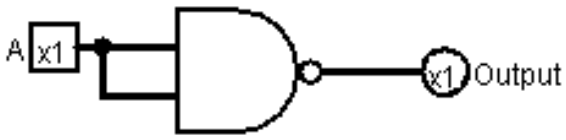
Implement NOT gate by NAND gate

- Verilog Code

```
module NOT_by_NAND(out, a);
input a;
output out;

nand nand_a_a(out, a, a);

endmodule
```

- Design Graph

NOT gate



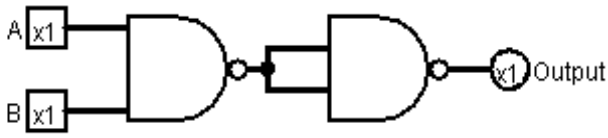
Implement AND gate by NAND gate

- Verilog Code

```
module AND_by_NAND(out, a, b);
input a, b;
output out;
wire w_nand_a_b;

nand nand_a_b(w_nand_a_b, a, b);
nand nand_to_output(out, w_nand_a_b,
 w_nand_a_b);

endmodule
```

- Design Graph

AND gate

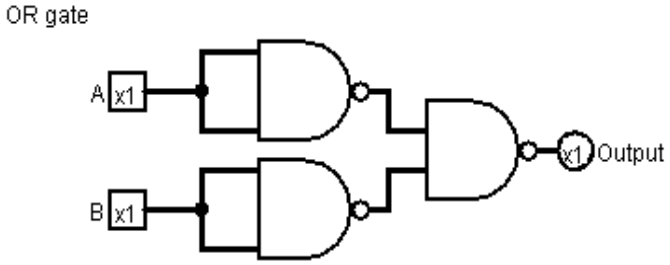


Implement OR gate by NAND gate

- Verilog Code

```
module OR_by_NAND(out, a, b);
input a, b;
output out;
wire nand_a, nand_b;

nand a_itself(nand_a, a, a);
nand b_itself(nand_b, b, b);
nand gene_output(out, nand_a, nand_b
);

endmodule
```

- Design Graph

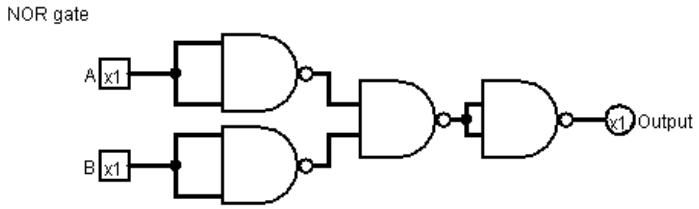


Implement NOR gate by NAND gate

- Verilog Code

```
module NOR_by_NAND(out, a, b);
input a, b;
output out;
wire or_a_b;

OR_by_NAND gene_OR(or_a_b, a, b);
NOT_by_NAND gene_out(out, or_a_b);
Endmodule
```

- Design Graph

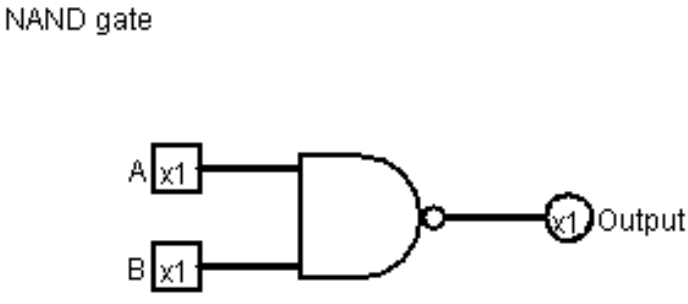


Implement NAND gate by NAND gate

- Verilog Code

```
module NAND_by_NAND(out, a, b);
input a, b;
output out;

nand simple(out, a, b);

endmodule
```

- Design Graph

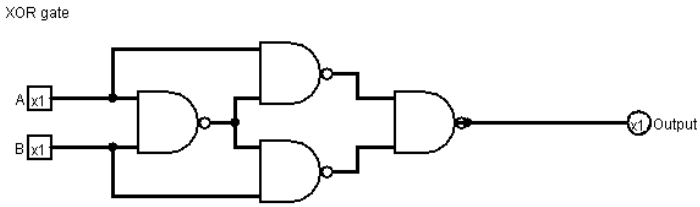


Implement XOR gate by NAND gate

- Verilog Code

```
module XOR_by_NAND(out, a, b);
input a, b;
output out;
wire a_NAND_b, a_nn, b_nn;

nand gene_a_NAND_b(a_NAND_b, a, b);
nand gene_a_nn(a_nn, a, a_NAND_b);
nand gene_b_nn(b_nn, b, a_NAND_b);
nand gene_out(out, a_nn, b_nn);

endmodule
```

- Design Graph

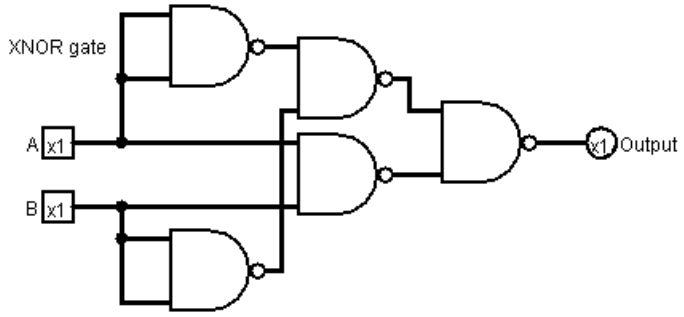


Implement XNOR gate by NAND gate

- Verilog Code

```
module XNOR_by_NAND(out, a, b);
input a, b;
output out;
wire not_a, not_b;
wire not_a_NAND_not_b, a_NAND_b;

NOT_by_NAND gene_not_a(not_a, a);
NOT_by_NAND gene_not_b(not_b, b);
nand gene_nand1(not_a_NAND_not_b, no
t_a, not_b);
nand gene_nand2(a_NAND_b, a, b);
nand gene_out(out, not_a_NAND_not_b,
 a_NAND_b);

endmodule
```

- Design Graph

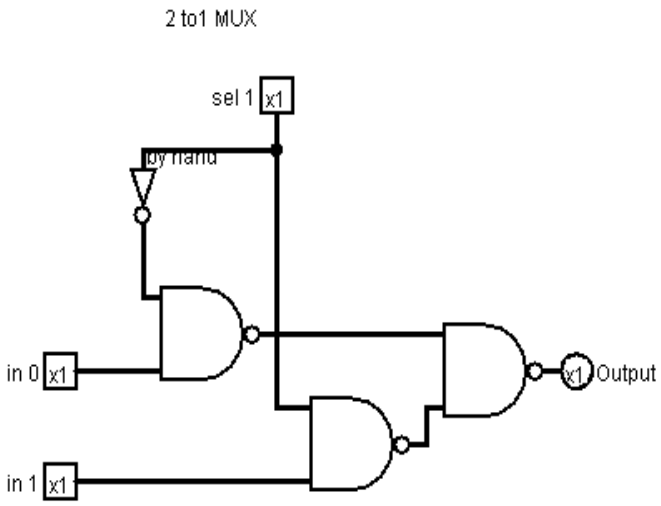


Implement 2 to 1 MUX by NAND gate

- Verilog Code

```
module MUX_2_to_1_by_NAND (
    in, sel, out
);
input [1:0] in;
input sel;
output out;
wire n_sel;
wire in_0_and, in_1_and;

NOT_by_NAND gene_n_sel(
n_sel, sel);
nand gene_in_0(
in_0_and, in[0], n_sel);
nand gene_in_1(
in_1_and, in[1], sel);
nand gene_out(
out, in_0_and, in_1_and);
endmodule
```

- Design Graph

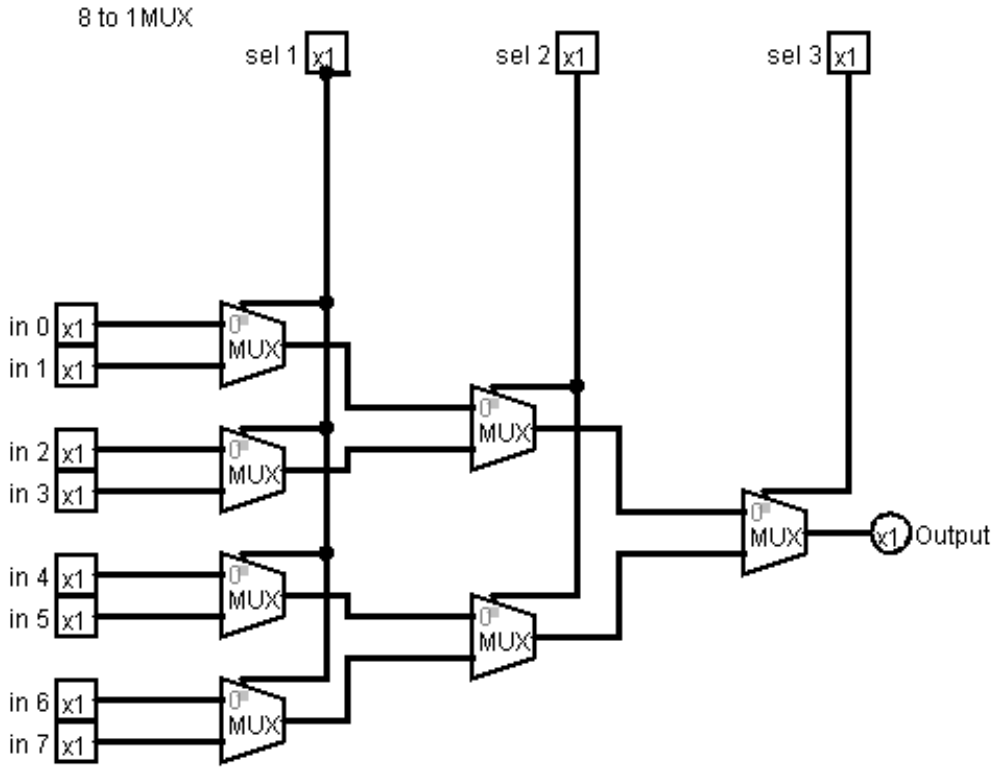


Implement 8 to 1 MUX by NAND gate

- Verilog Code

```
module MUX_8_to_1_by_NAND(in, sel, out);
input [7:0]in;
input [2:0] sel;
output out;
wire [3:0] layer1;
wire [1:0] layer2;

MUX_2_to_1_by_NAND MUX_1_0(in[1:0], sel[0], layer1[0]);
MUX_2_to_1_by_NAND MUX_3_2(in[3:2], sel[0], layer1[1]);
MUX_2_to_1_by_NAND MUX_5_4(in[5:4], sel[0], layer1[2]);
MUX_2_to_1_by_NAND MUX_7_6(in[7:6], sel[0], layer1[3]);

MUX_2_to_1_by_NAND MUX_layer2_0(layer1[1:0], sel[1], layer2[0]);
MUX_2_to_1_by_NAND MUX_layer2_1(layer1[3:2], sel[1], layer2[1]);

MUX_2_to_1_by_NAND gene_out(layer2[1:0], sel[2], out);
endmodules
```

- Design Graph

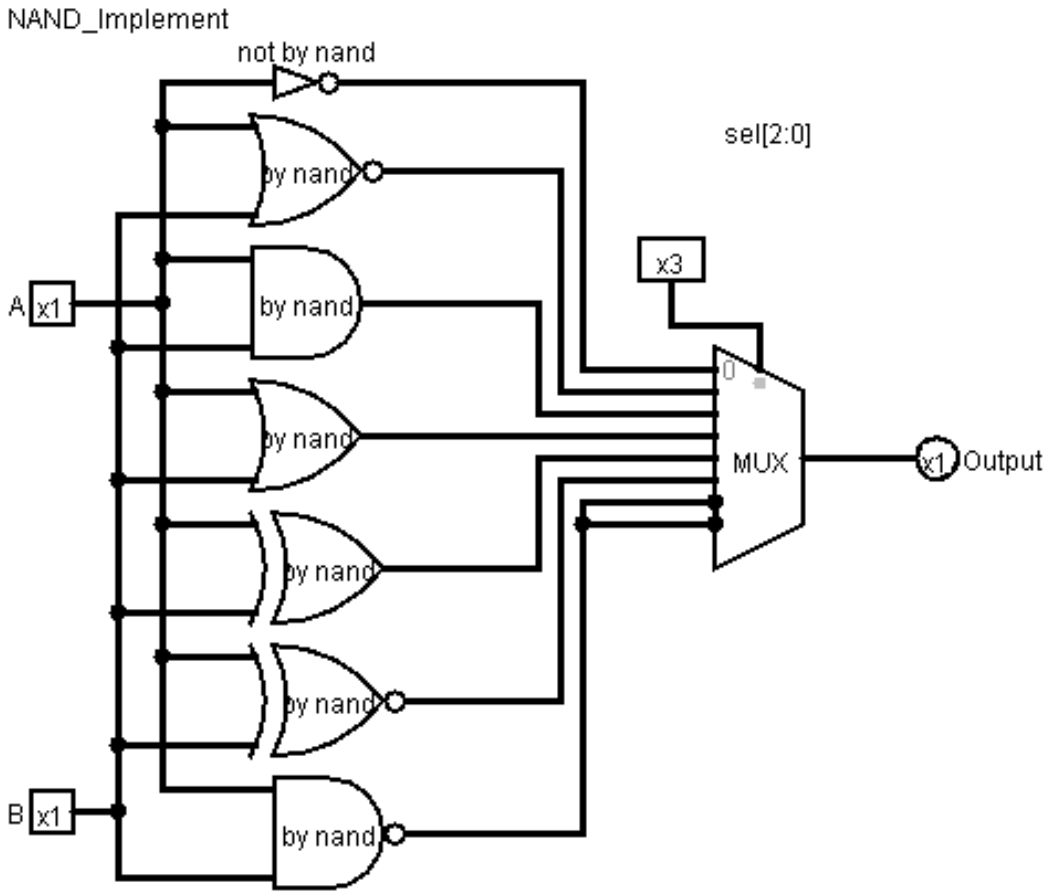


Implement the MAIN circuit

- Verilog Code

```
module NAND_Implement (a, b, sel, out);
input a, b;
input [3-1:0] sel;
output out;
wire [7:0] dataflow;

NOT_by_NAND gene_df_0 (dataflow[0], a);
NOR_by_NAND gene_df_1 (dataflow[1], a, b);
AND_by_NAND gene_df_2 (dataflow[2], a, b);
OR_by_NAND  gene_df_3 (dataflow[3], a, b);
XOR_by_NAND gene_df_4 (dataflow[4], a, b);
XNOR_by_NAND gene_df_5 (dataflow[5], a, b);
NAND_by_NAND gene_df_6 (dataflow[6], a, b);
NAND_by_NAND gene_df_7 (dataflow[7], a, b);
MUX_8_to_1_by_NAND gene_out(dataflow[7:0], sel[2:0], out);

endmodule
```

- Design Graph



設計構想：

首先將 not、nor、and、or、xor、xnor、nand gates，都使用 nand gate 給做出來，寫成一個個的 module。

接著使用 3 個 nand gate 以及剛剛做出來的 not gate 組成 2 to 1 的 MUX。再以 2 to 1 MUX 的 module 組合成 3 個 selector 的 8 to 1MUX。用來將題目所敘述的八個以 nand 做成的 input gate 接上。

開發過程中的問題/學習：

在用 nand gate 構造出其他 gate 的過程中想了很久，後來還有上維基百科確認最簡潔的寫法，XOR 一開始自己寫用了五個 nand gate，是 wiki 上提供的第二種寫法，沒想到有竟然有只用 4 個 nand 的作法。

2 to 1 MUX 本來是照著 lab 1 的 module 然後以 nand gate 做出來的 and 以及 or 來做而已，後來實際自己推一下，在 or 前面加兩個 inverter，再將 inverter 往後送，or 變成 and，inverter 都接到 and 前面，就變成了用 4 個 nand 做出來的版本(not by nand 只用一個 nand)，不亦樂乎。

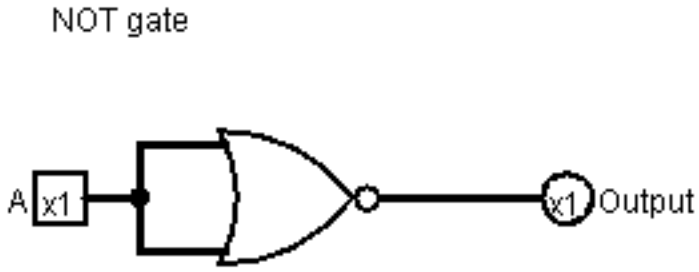
Basic Questions 2

Implement NOT gate by NOR gate

- Verilog Code

```
module NOT_by_NOR(out, a);
input a;
output out;

nor gene_out(out, a, a);

endmodule
```

Design Graph

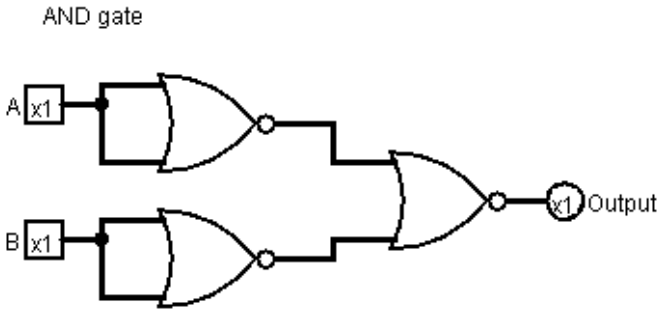


Implement AND gate by NOR gate

- Verilog Code

```
module AND_by_NOR(out, a, b);
input a, b;
output out;
wire n_a, n_b;

nor gene_n_a(n_a, a);
nor gene_n_b(n_b, b);
nor gene_out(out, n_a, n_b);

endmodule
```

- Design Graph

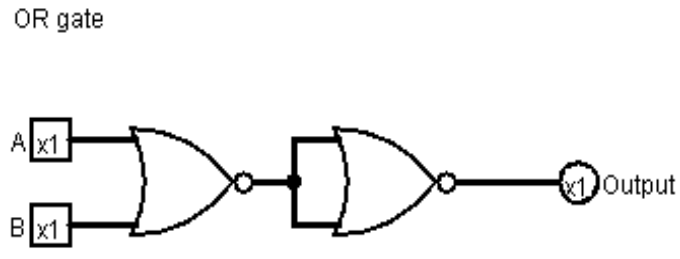


Implement OR gate by NOR gate

- Verilog Code

```
module OR_by_NOR(out, a, b);
input a, b;
output out;
wire a_nor_b;

nor gene_a_nor_b (a_nor_b, a, b);
nor gene_out(out, a_nor_b);

endmodule
```

- Design Graph

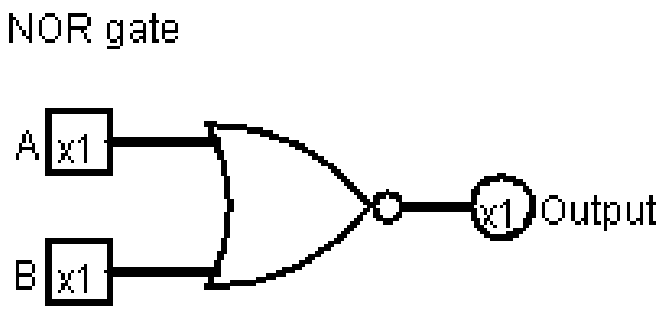


Implement NOR gate by NOR gate

- Verilog Code

```
module NOR_by_NOR(out, a, b);
input a, b;
output out;

nor gene_out(out, a, b);

endmodule
```

- Design Graph

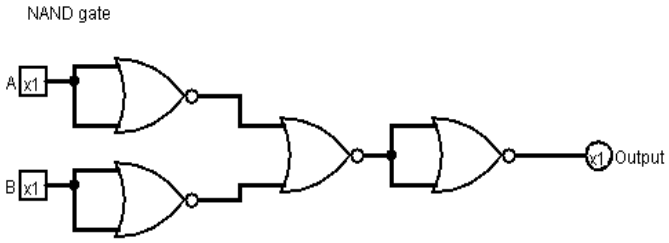


Implement NAND gate by NOR gate

- Verilog Code

```
module NAND_by_NOR(out, a, b);
input a, b;
output out;
wire a_and_b;
AND_by_NOR gene_a_and_b(a_and_b, a,
b);
NOT_by_NOR gene_out(out, a_and_b);
endmodule
```

- Design Graph

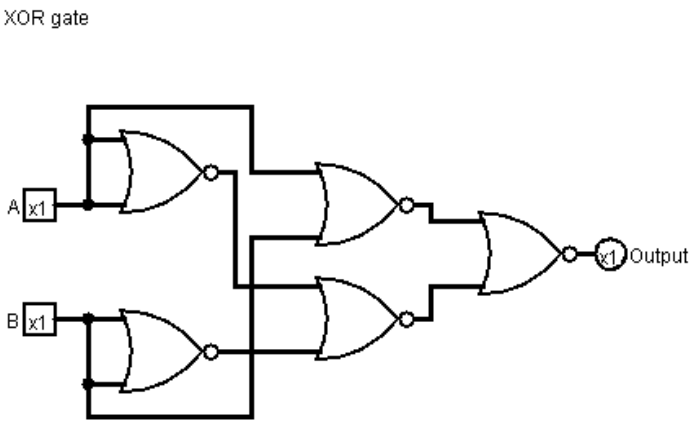


Implement XOR gate by NOR gate

- Verilog Code

```
module XOR_by_NOR(out, a, b);
input a, b;
output out;
wire n_a, n_b;
wire not_a_NOR_not_b, a_NOR_b;

NOT_by_NOR gene_not_a(not_a, a);
NOT_by_NOR gene_not_b(not_b, b);
nor gene_nor1(not_a_NOR_not_b, not_a
, not_b);
nor gene_nor2(a_NOR_b, a, b);
nor gene_out(out, not_a_NOR_not_b, a
_NOR_b);
endmodule
```

- Design Graph

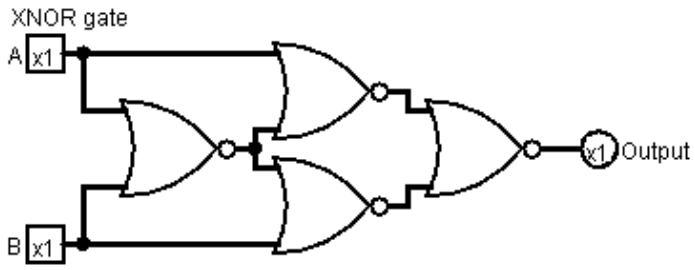


Implement XNOR gate by NOR gate

- Verilog Code

```
module XNOR_by_NOR(out, a, b);
input a, b;
output out;
wire a_NOR_b;
wire a_nor_norab, b_nor_norab;

nor gene_a_NOR_b(a_NOR_b, a, b);
nor gene_a_nor_norab(a_nor_norab, a,
 a_NOR_b);
nor gene_b_nor_norab(b_nor_norab, b,
 a_NOR_b);
nor gene_out(out, a_nor_norab, b_nor
_norab);

endmodule
```

- Design Graph

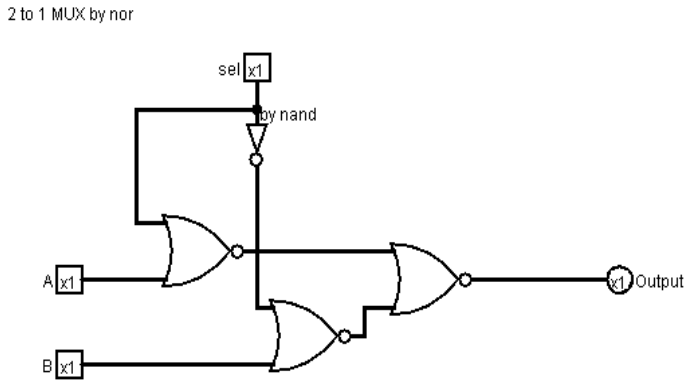


Implement 2 to 1 MUX by NOR gate

- Verilog Code

```
module MUX_2_to_1_by_NOR (
    in, sel, out
);
input [1:0] in;
input sel;
output out;
wire n_sel;
wire in_0_and, in_1_and;

NOT_by_NOR gene_n_sel(n_sel, se
l);
AND_by_NOR gene_in_0(in_0_and,
in[0], n_sel);
AND_by_NOR gene_in_1(in_1_and,
in[1], sel);
OR_by_NOR gene_out(out, in_0_an
d, in_1_and);
endmodule
```

- Design Graph

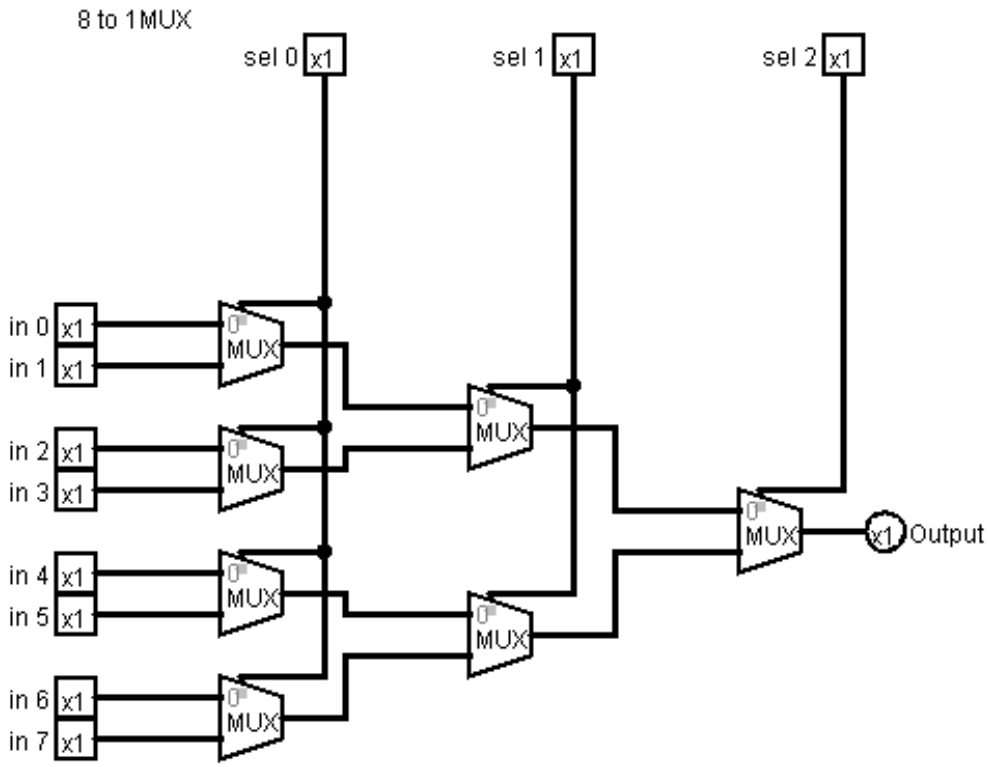


Implement 8 to 1 MUX by NOR gate

- Verilog Code

```
module MUX_8_to_1_by_NOR(in, sel, out);
input [7:0]in;
input [2:0] sel;
output out;

wire [3:0] layer1;
wire [1:0] layer2;

MUX_2_to_1_by_NOR MUX_1_0(in[1:0], sel[0], layer1[0]);
MUX_2_to_1_by_NOR MUX_3_2(in[3:2], sel[0], layer1[1]);
MUX_2_to_1_by_NOR MUX_5_4(in[5:4], sel[0], layer1[2]);
MUX_2_to_1_by_NOR MUX_7_6(in[7:6], sel[0], layer1[3]);

MUX_2_to_1_by_NOR MUX_layer2_0(layer1[1:0], sel[1], layer2[0]);
MUX_2_to_1_by_NOR MUX_layer2_1(layer1[3:2], sel[1], layer2[1]);

MUX_2_to_1_by_NOR gene_out(layer2[1:0], sel[2], out);
endmodule
```

- Design Graph

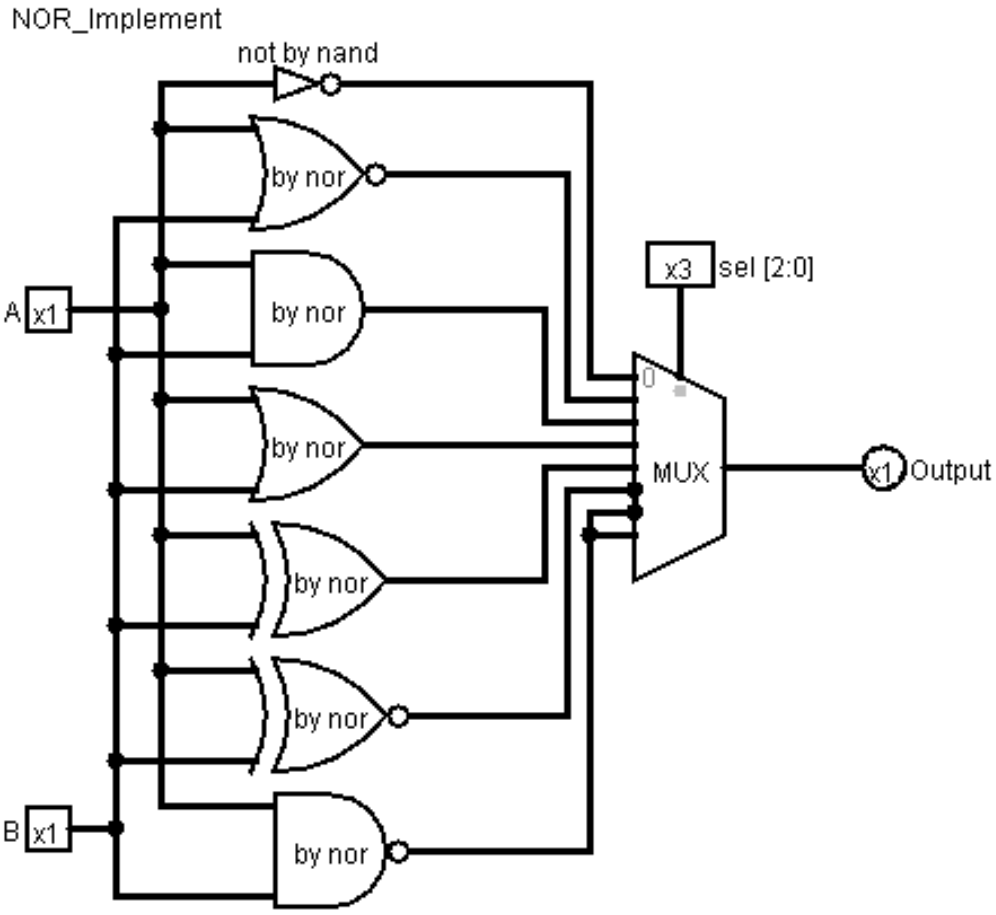


Implement the main circuit

- Verilog Code

```
module NAND_Implement (a, b, sel, out);
input a, b;
input [3-1:0] sel;
output out;
wire [7:0] dataflow;

NOT_by_NAND gene_df_0 (dataflow[0], a);
NOR_by_NAND gene_df_1 (dataflow[1], a, b);
AND_by_NAND gene_df_2 (dataflow[2], a, b);
OR_by_NAND  gene_df_3 (dataflow[3], a, b);
XOR_by_NAND gene_df_4 (dataflow[4], a, b);
XNOR_by_NAND gene_df_5 (dataflow[5], a, b);
NAND_by_NAND gene_df_6 (dataflow[6], a, b);
NAND_by_NAND gene_df_7 (dataflow[7], a, b);
MUX_8_to_1_by_NAND gene_out(dataflow[7:0], sel[2:0], out);

endmodule
```

- Design Graph



設計構想：

首先將 not、nor、and、or、xor、xnor、nand gates，都使用 nor gate 給做出來，寫成一個個的 module，。接著用 3 個 nor gate 以及剛剛做出來的 not gate 組成 2 to 1 的 MUX，再以 2 to 1 MUX 的 module 組合成 3 個 selector 的 8 to 1MUX。用來將題目所敘述的八個以 nor 做成的 input gate 接上。大致上就只是將 BQ1 的 gate 改成用 nor 坐而已，除了 MUX 外基本架構都一樣。

開發過程中的問題/學習：

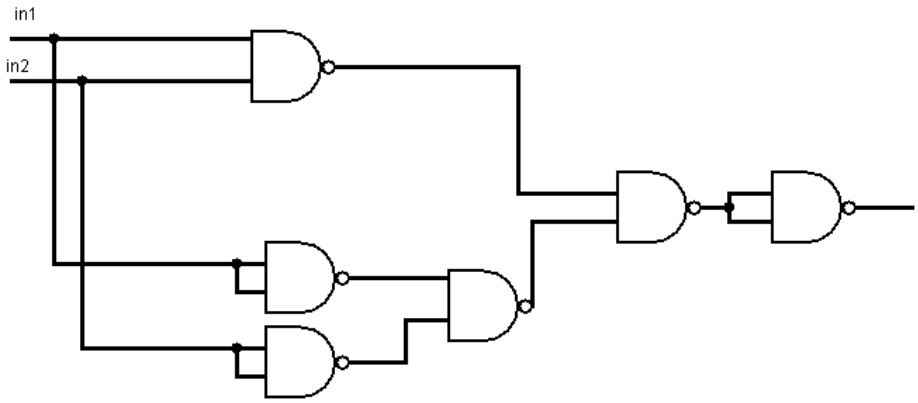
在用 nor gate 構造出其他 gate 的過程中想了更久，在後來想 XNOR 以及 XOR 的作法時，逐漸發現到 nand 跟 nor 之間微妙的關係，像是 XNOR 跟 XOR 基本上在 nor 跟 nand 的作法是整個對照過來的。

2 to 1 MUX 本來是照著 lab 1 的 module 然後以 nor gate 做出來的 and 以及 or 來做而已，不像是 BQ1 做出來後很明顯發現可以用 4 個 nand 改寫。反而是在 BQ1 的 MUX 成功改寫後，又經歷了上面那些 XOR、XNOR 以及 nor、nand 之間微妙的關係後。下意識認為應該是有辦法也只用 4 個 nor 做出 2 to 1 MUX 才

對，所以在寫 report 時臨時用 logism 的測試功能嘗試改做一下，就成為了現在這個 4 個 nor 的版本(not by nor 只用一個 nor gate)。

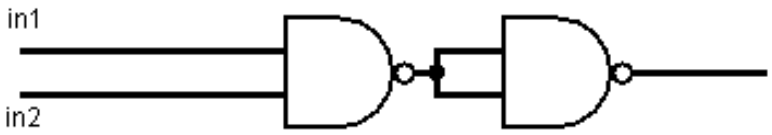
讀到這裡真的是忍不住心中的激動，nand 跟 nor 之間的關係也太過巧合，這中間應該有很好的離散解釋方法以及證明，之後會再去研讀看看。

AQ1.



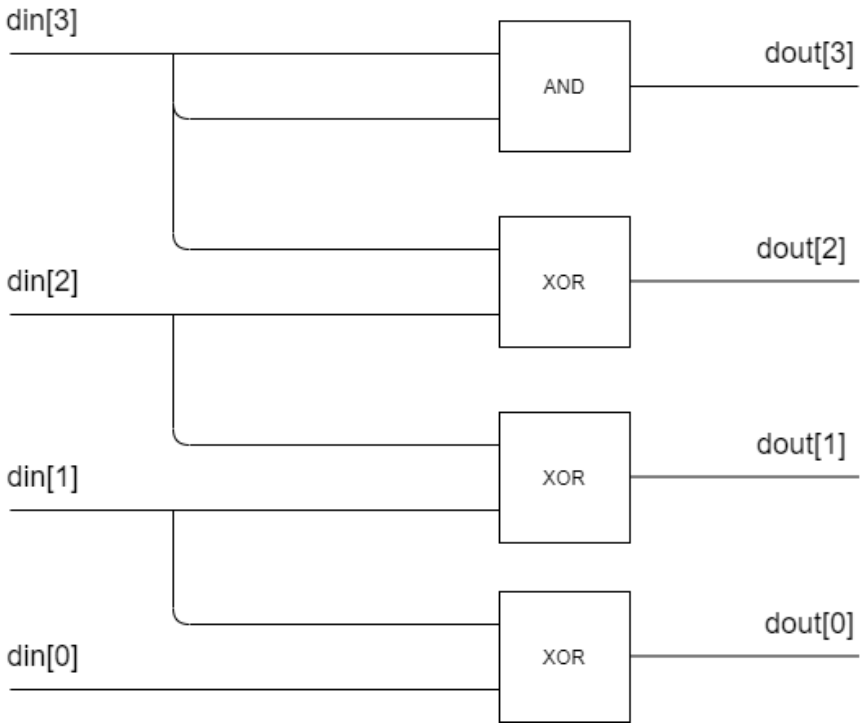

```
module XOR (out, in1, in2);
input in1, in2;
output out;
wire a_bar, b_bar, o_na1, o_na2, o_na3;
nand na_a(a_bar, in1, in1);
nand na_b(b_bar, in2, in2);
nand na1(o_na1, in1, in2);
nand na2(o_na2, a_bar, b_bar);
nand na3(o_na3, o_na2, o_na1);
nand na4(out, o_na3, o_na3);

endmodule
```

XOR




```
module AND (out, in1, in2);
input in1, in2;
output out;
wire o_na1;
nand na1(o_na1, in1, in2);
nand na2(out, o_na1, o_na1);

endmodule
```

AND




```
AND a1(dout[3], din[3], din[3]);
XOR x1(dout[2], din[3], din[2]);
XOR x2(dout[1], din[2], din[1]);
XOR x3(dout[0], din[1], din[0]);
```

Binary_to_Grey

設計構想：

先用一般的 gate – level 實作出 Binary_to_Grey 的 module，看需要用到哪些 Gate，再用 NAND 實作它們。而在實作 Binary_to_Grey 時，可以發現如果將 output，拆成 4 個 bit 來看，可以觀察出其與 input 某些 bit 的關聯。由最高位 bit 開始往下看，發現 din[3]與 dout[3]相等，dout[2]呈現`0000 1111 1111 0000`

的規律，dout[1]呈現`00 11 11 00`X2 的規律，dout[0]呈現

`0 1 1 0`X4 的規律，分別與{ din[3], din[2] }、{ din[2], din[1] }、

{ din[1], din[0] }的關係為XOR。

Testbench 設計構想：

```
initial begin
  repeat (16) begin
    #1 din = din + 1'b1;
  end
  #1 $finish;
end
```

將 din 窮舉 16 次，測試所有可能。

開發過程中的問題/學習：

仔細觀察 input 和 output 的規律及關係，可以大幅減低程式的長

度，也不需要使用 k – map。

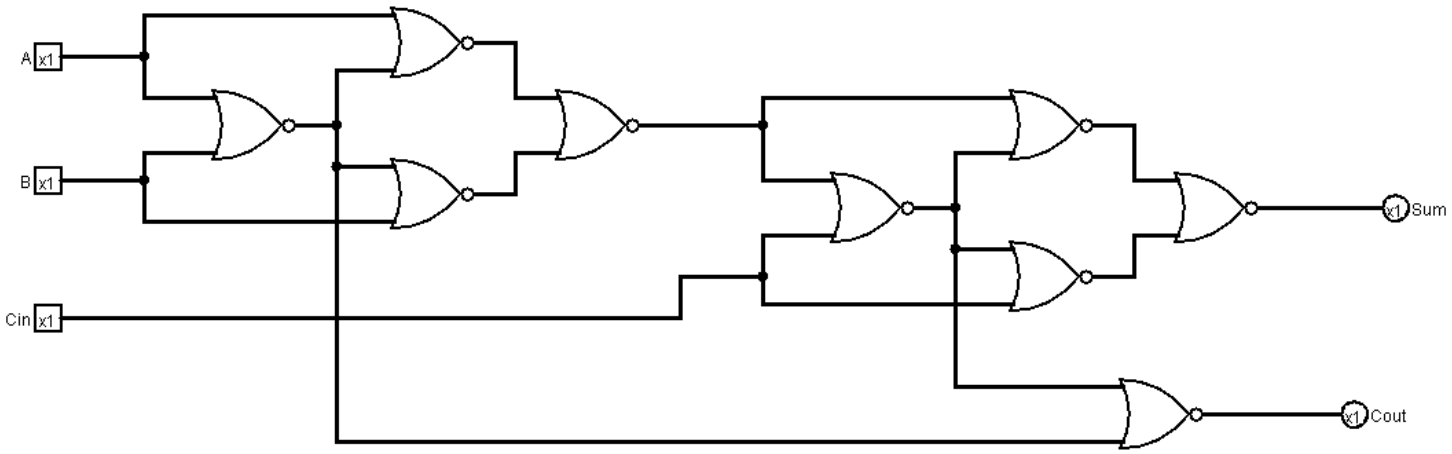
Advanced Question 2

Implement 1 bit Full Adder by NOR gate

- Verilog Code

```
module FullAdder_1bit(a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;
wire a_nor_b;
wire a_nor_a_nor_b, a_nor_b_nor_b;
wire layer1;
wire layer1_nor_cin, layer1_nor_layer1_nor_cin, layer1_nor_cin_nor_cin;

nor gene_a_nor_b(a_nor_b, a, b);
nor gene_ananb(a_nor_a_nor_b, a, a_nor_b);
nor gene_anbnb(a_nor_b_nor_b, b, a_nor_b);
nor gene_layer1(layer1, a_nor_a_nor_b, a_nor_b_nor_b);

nor gene_lnc(layer1_nor_cin, layer1, cin);
nor gene_lnlnc(layer1_nor_layer1_nor_cin, layer1, layer1_nor_cin);
nor gene_lncnc(layer1_nor_cin_nor_cin, cin, layer1_nor_cin);

nor gene_sum(sum, layer1_nor_cin_nor_cin, layer1_nor_layer1_nor_cin);
nor gene_cout(cout, layer1_nor_cin, a_nor_b);

endmodule
```

- Design Graph

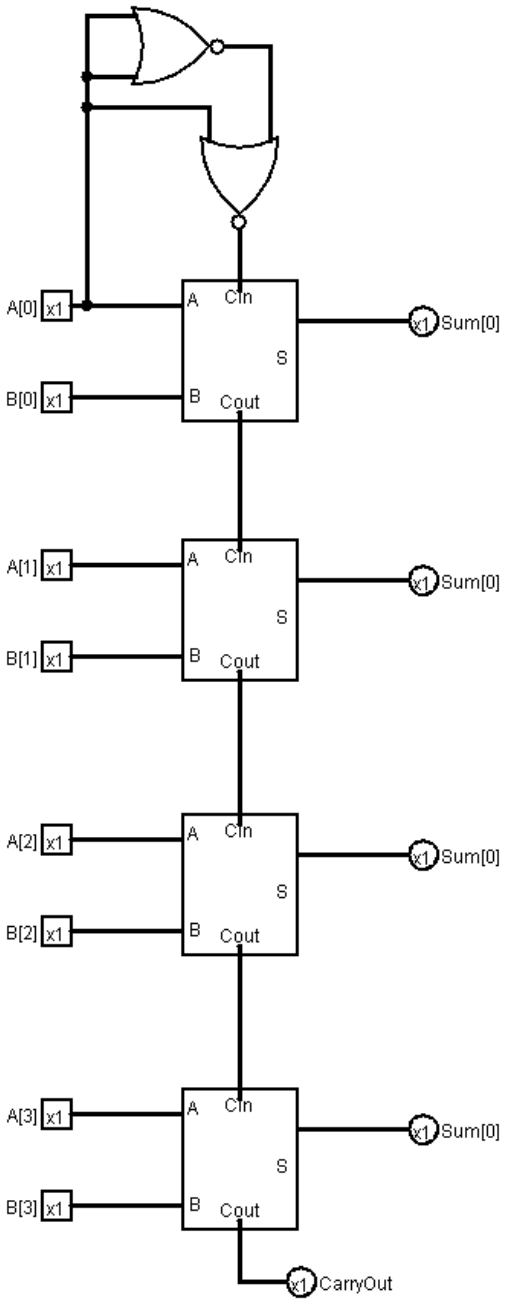


Implement 4 bit Ripple Carry Adder by NOR gate

- Verilog Code

```
module RCA_4bit(a, b, sum, cout);
input [3:0] a, b;
output [3:0] sum;
output cout;
wire [2:0] carry;
wire alwaysZero,  n_a_0;

nor gene_n_a_0(n_a_0, a[0], a[0]);
nor gene_Zero(alwaysZero, a[0], n_a_
0);

FullAdder_1bit FA_1(a[0], b[0], alwa
ysZero, sum[0], carry[0]);
FullAdder_1bit FA_2(a[1], b[1], carr
y[0], sum[1], carry[1]);
FullAdder_1bit FA_3(a[2], b[2], carr
y[1], sum[2], carry[2]);
FullAdder_1bit FA_4(a[3], b[3], carr
y[2], sum[3], cout);

endmodule
FullAdder_1bit FA_3(a[2], b[2], carr
y[1],
sum[2], carry[2]);
FullAdder_1bit FA_4(a[3], b[3], carr
y[2],
sum[3], cout);

endmodule
```

- Design Graph

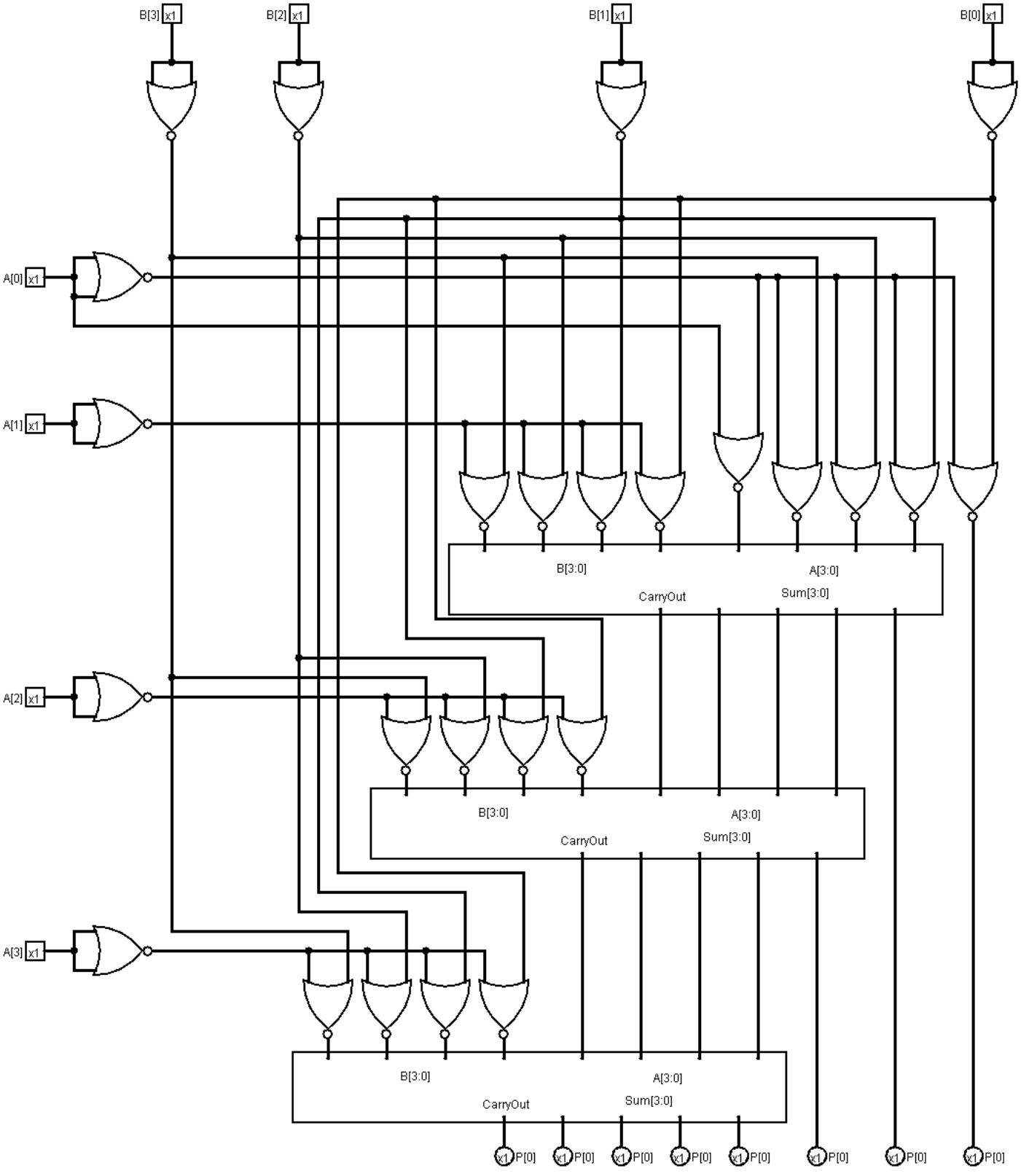


Implement Multiplier by NOR gate

- Verilog Code

```
module Multiplier (a, b, p);
input [4-1:0] a, b;
output [8-1:0] p;
wire [4-1:0] n_a, n_b;
wire [4-1:0] ab [4-1:0];
nor gene_n_a [4-1:0] (n_a, a, a);
nor gene_n_b [4-1:0] (n_b, b, b);
wire [4-1:0] layer[1:0];
wire alwaysZero;

nor gene_Zero(alwaysZero, n_a[0], a[0]);

nor gene_0_0(p[0], n_a[0], n_b[0]);
nor gene_0_1(ab[0][1], n_a[0], n_b[1]);
nor gene_0_2(ab[0][2], n_a[0], n_b[2]);
nor gene_0_3(ab[0][3], n_a[0], n_b[3]);

nor gene_1_0(ab[1][0], n_a[1], n_b[0]);
nor gene_1_1(ab[1][1], n_a[1], n_b[1]);
nor gene_1_2(ab[1][2], n_a[1], n_b[2]);
nor gene_1_3(ab[1][3], n_a[1], n_b[3]);

nor gene_2_0(ab[2][0], n_a[2], n_b[0]);
nor gene_2_1(ab[2][1], n_a[2], n_b[1]);
nor gene_2_2(ab[2][2], n_a[2], n_b[2]);
nor gene_2_3(ab[2][3], n_a[2], n_b[3]);

nor gene_3_0(ab[3][0], n_a[3], n_b[0]);
nor gene_3_1(ab[3][1], n_a[3], n_b[1]);
nor gene_3_2(ab[3][2], n_a[3], n_b[2]);
nor gene_3_3(ab[3][3], n_a[3], n_b[3]);

RCA_4bit RCA_1({alwaysZero, ab[0][3:1]}, ab[1], {layer[0][2:0], p[1]}, layer[0]
[3]);
RCA_4bit RCA_2(layer[0], ab[2], {layer[1][2:0], p[2]}, layer[1][3]);
RCA_4bit RCA_3(layer[1], ab[3], p[6:3], p[7]);

endmodule
```

- Design Graph



Testbench：

```
`timescale 1ps / 1ps
module testbenchAQ2;
reg [3:0] a, b;
wire [7:0] result;
Multiplier CLAA(a, b, result);
task Test;
begin
    if(result !== (a*b))begin
        $display("ERROR!!!!");
        $write("debug a:%d ", a);
        $write("debug b:%d ", b);
        $write("debug result:%d ", result);
    end
end
endtask
initial begin
    {a[3:0], b[3:0]} = 8'b0;
    repeat (2 ** 20) begin
        #1 {a[3:0], b[3:0]} =  {a[3:0], b[3:0]} + 1'b1;
        #1 Test;
    end
    #10 $finish;
end
endmodule
```

所有 case 都測過一次。並且加入了 task 作為測試。

Wow it Works：

首先我們可以看出其實就是 A[3] ~ A[0] 跟 B[3] ~ B[0]做 and，然後有四次 4 bit 的相加，每次相加就往右 shift，原先的 LSB 進 output，最後一層的 Sum 以及 carry 也都進 output。

概念上就是把一個 4bit*4bit 的乘法轉成 4bit * 1 bit 的四次乘法然後對要繼續累加的位數做相加而已。

設計構想：

首先將用 nor 做出 full adder，接著將四個 full adder 串接成 Ripple Carry Adder，由於等等在 multiplier 的應用上不需要 Cin，因此這邊就用兩個 nor 來造出 alwaysZero 永遠為 0 的 wire，作為虛擬的 Cin。在 Multiplier 的設計上，參考了上學期邏輯設計教的 Multiplier，並做了些變化，明顯的是在 a 跟 b 做 and 來當 input 的部分，利用簡單的回推全部換成 nor。還有就是增加一層的 RCA。第一層 RCA 的 A[3]input 放了一個利用兩個 nor 製造出的 1’ b0，用來填補不存在的進位

開發過程中的問題/學習：

這題基本上沒有遇到太大的困難。有趣的部分在於用 nor 做出 full adder，由於我是先做 Advanced Question 3，在處裡 AQ3 的問題中，找到的一篇 reference(註 1)描述了只用 nand 做出 Full Adder 的證明流程。然後套用在 Basic Question 中得到的 nand 與 nor 之間微妙的關係，所以我就猜想 full adder 應該也是能夠純用 nor gate 做出來，試了幾次後用 nand 直接改過來的狀況是 input 後面以及 output 都會多一個 not，嘗試把 not 拿掉的結果仍是正確。但目前還找不到相關的 reference 來證明是合理的離散推斷。也許等有空時自己來推斷看看。

註 1：https://www.eeweb.com/full-adder-nand-equivalent/

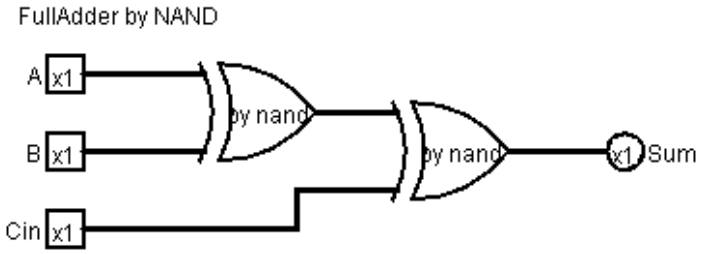
Advanced Question 3

Implement Full Adder gate by NAND gate

- Verilog Code

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

wire XNOR_a_b;
XNOR_by_NAND gene_XNOR_a_b(
XNOR_a_b, a, b);
XNOR_by_NAND gene_out(
sum, XNOR_a_b, cin);

endmodule
```

- Design Graph

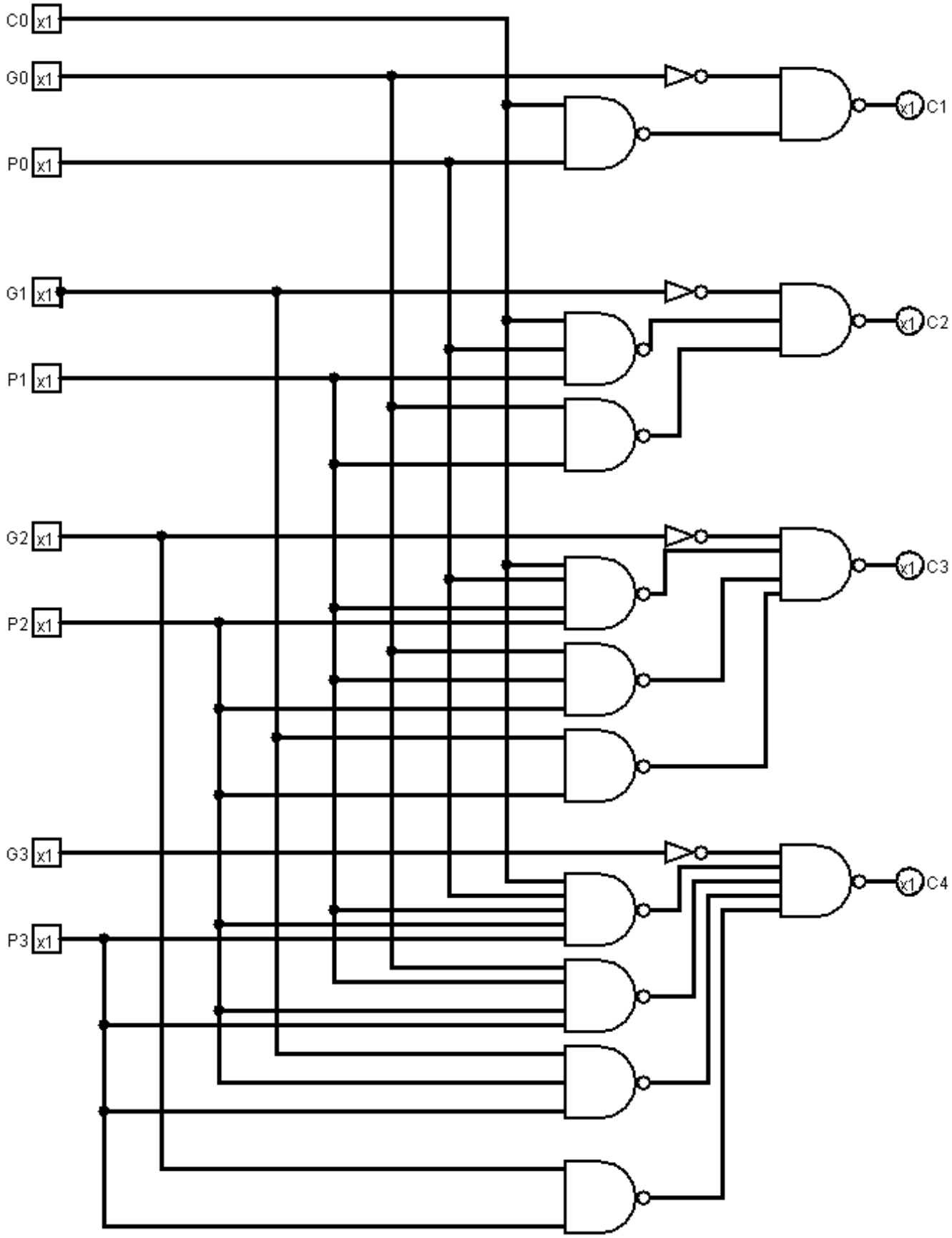


Implement 4bit Look Ahead gate by NOR gate

- Verilog Code

```
m module Look_4bit_Ahead(cin, G, P, Cout);
input cin;
input [3:0] G, P;
output [3:0] Cout;
wire [3:0] n_G;
wire layer1;
wire [1:0] layer2;
wire [2:0] layer3;
wire [3:0] layer4;
// init n_G
NOT_by_NAND gene_n_G [3:0](n_G, G);
// layer1
nand gene_layer1(layer1, P[0], cin);
// layer2
nand gene_layer2_0(layer2[0], P[1], G[0]);
nand gene_layer2_1(layer2[1], P[1], P[0], cin);
// layer3
nand gene_layer3_0(layer3[0], P[2], G[1]);
nand gene_layer3_1(layer3[1], P[2], P[1], G[0]);
nand gene_layer3_2(layer3[2], P[2], P[1], P[0], cin);
// layer4
nand gene_layer4_0(layer4[0], P[3], G[2]);
nand gene_layer4_1(layer4[1], P[3], P[2], G[1]);
nand gene_layer4_2(layer4[2], P[3], P[2], P[1], G[0]);
nand gene_layer4_3(layer4[3], P[3], P[2], P[1], P[0], cin);
// output
nand gene_cout_0(Cout[0], n_G[0], layer1);
nand gene_cout_1(Cout[1], n_G[1], layer2[0], layer2[1]);
nand gene_cout_2(Cout[2], n_G[2], layer3[0], layer3[1], layer3[2]);
nand gene_cout_3(Cout[3], n_G[3], layer4[0], layer4[1], layer4[2], layer4[3]);

endmodule
```

- Design Graph

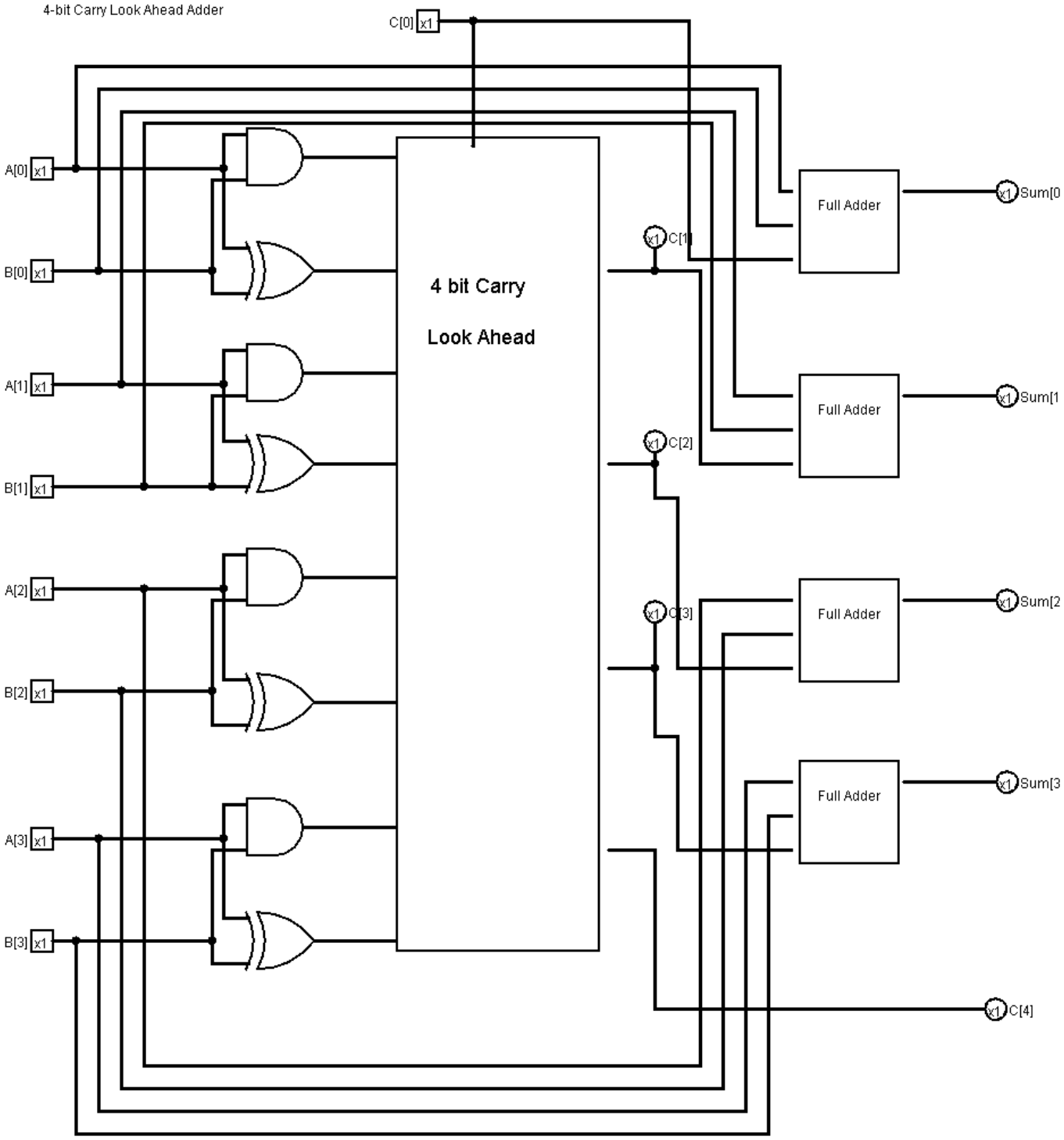


Implement Carry Lok Ahead Adder by NOR gate

- Verilog Code

```
module Carry_Look_Ahead_Adder (a, b, cin, cout, sum);
input [4-1:0] a, b;
input cin;
output cout;
output [4-1:0] sum;
wire [2:0] CarryOut;
wire [3:0] G, P;

AND_by_NAND gene_G[3:0](G, a, b);
XOR_by_NAND gene_P[3:0](P, a, b);

Look_4bit_Ahead L4A(cin, G, P, {cout, CarryOut[2:0]});
FullAdder gene_sum0(a[0], b[0], cin, sum[0]);
FullAdder gene_sum1(a[1], b[1], CarryOut[0], sum[1]);
FullAdder gene_sum2(a[2], b[2], CarryOut[1], sum[2]);
FullAdder gene_sum3(a[3], b[3], CarryOut[2], sum[3]);

endmodule
```

- Design Graph



Testbench：

```
`timescale 1ns / 1ps
module Lab2_TeamX_Carry_Look_Ahead_t;
reg [3:0] a, b;
reg cin;
wire [3:0] sum;
wire cout;
Carry_Look_Ahead_Adder CLAA(a, b, cin, cout, sum);
task Test;
begin
    if({cout, sum} !== (a + b + cin))begin
        $display("ERROR!!!!");
        $write("debug a:%d ", a);
        $write("debug b:%d ", b);
        $write("debug cin:%d ", cin);
        $write("debug cout:%d ", cout);
        $write("debug su:%d ", sum);
    end
end
endtask
initial begin
    {a[3:0], b[3:0], cin} = 17'b0;
    repeat (2 ** 30) begin
        #1 {a[3:0], b[3:0], cin} =  {a[3:0], b[3:0], cin} + 1'b1;
        #1 Test;
    end
    #10 $finish;
end
endmodule
```

所有 case 都測過一次。與之前的 testbench 不同，這次加入了 task 作為偵測的幫助。

How it works：

定義 Generation function 的以及 Propagation function，透過一些邏輯運算後遞迴寫出等式，然後就能直接判斷需不需要輸出 Cout。

好處就是等待的 gate 層數少，只會有兩層。相較於 ripple carry Adder，每一層都要等待上一層的 adder gate 跑完，才能得到 cin 繼續下一個 bit 的運算，CLA 能夠只用兩層 gate 得到 Cin，平行地執行 Full Adder 的運算，大幅提升速度。缺點是須利用較多的 gate(CLA)。

設計構想：

首先用 nand 做出 full adder，雖然說是 full adder，但是我並沒有實作 Cin，由於 Carry Look Ahead 是直接判斷出有無進位，因此不需要 cin 的部分，Full Adder 就以兩個由 nand 做成的 xor 組成。

4 bit Carry Look Ahead 參考了上學期邏輯設計學的 Generation and Propagation function，將每一層的 and、or gate 先連接好，在透過從 or 前面加兩個 inverter 並將 inverter 向後傳的簡單方式將其轉換成全由 nand gate 組成的 CLA(not gate 也用 nand 實作，方式如同 BQ1)。

最後用 nand gate 所組成的 and gate、xor gate，作為 Generation、Propagation functions 用來 input 進 4 bit Carry Look Ahead Module。

最後在每個 CLA Module 生出來的 Cout，當作每個 FullAdder 的 Cin 就大致上完成了。

開發過程中的問題/學習：

這題真的是好好訓練到確實把 CLA 的公式 implement 成實際的邏輯

閘，這張 CLA 的圖我大概重畫四五次。最後才生產出來比較簡潔的版本。

再來就是第一次在 verilog 中使用{}來連接 wire，還有在 AQ2 以及 AQ3 中都第一次的使用了類似 and test[3:0] (out, a ,b)的陣列 gate 寫法。

AQ2、3 總結心得：

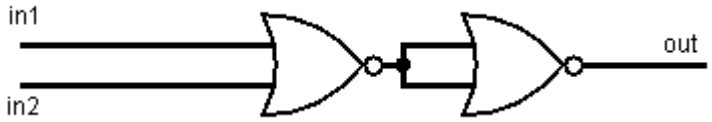
這次的 lab 我從 Draw.io 跳槽到了 Logism，上手時間花了大概三天在做白工，不過熟悉了之後真的是一個非常好用的工具，能夠每個 module 都做成一個可重複利用的模組，並套用在更大的 design 當中。還能夠及時測試 input/output 的 true table，能夠在設計構想時期就處理掉多數的思考錯誤，開發速度增快許多。

在 verilog 上面有記取上次畫圖的慘狀，本次作業採用先畫圖再寫 verilog 的流程，並在畫圖過程中將能夠包裝的 module 的 code 都包好，寫 code 時思緒清晰許多。也在 testbench 中加入了 task 來做測試，透過看 log 訊息的方式，能夠更快看到出錯的測資，加快作業速度，爭取睡眠時間。

AQ4.

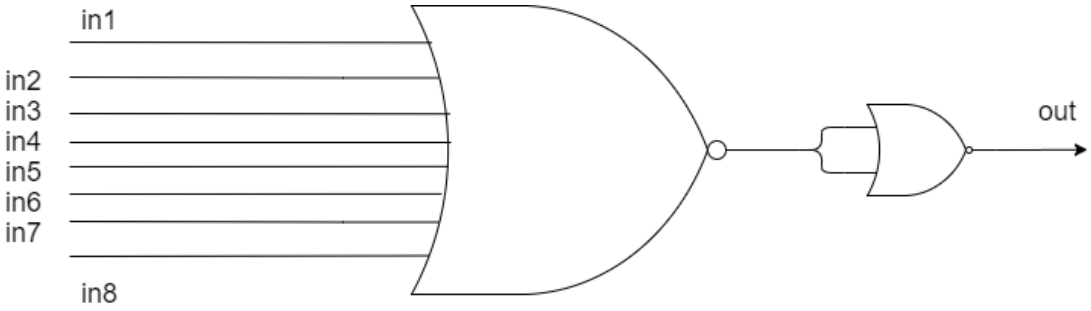

```
nor no1(out, in, in);
```

NOT



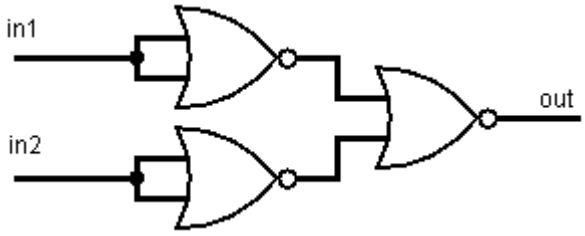

```
nor no1(o_no1, in1, in2);
nor no2(out, o_no1, o_no1);
```

OR



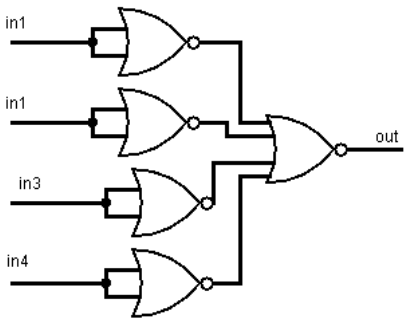

```
nor no1(o_no1, in1, in2, in3, in4, in5, in6, in7, in8);
nor no2(out, o_no1, o_no1);
```

OR_8



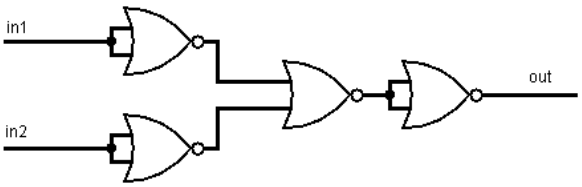

```
nor no1(o_no1, in1, in1);
nor no2(o_no2, in2, in2);
nor no3(out, o_no1, o_no2);
```

AND



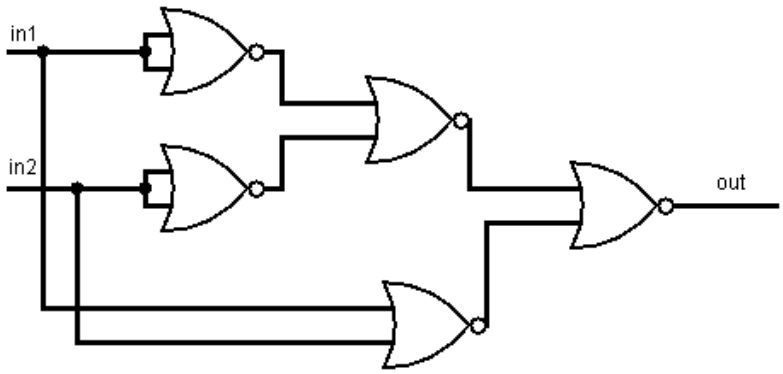

```
nor no1(o_no1, in1, in1);
nor no2(o_no2, in2, in2);
nor no3(o_no3, in3, in3);
nor no4(o_no4, in4, in4);
nor noo(out, o_no1, o_no2, o_no3, o_no4);
```

AND_4



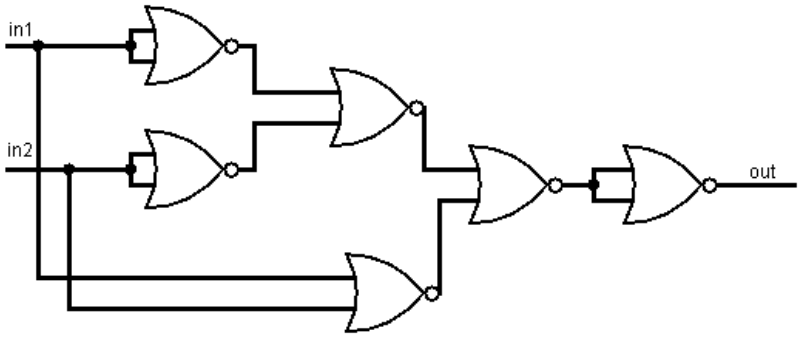

```
nor no1(o_no1, in1, in1);
nor no2(o_no2, in2, in2);
nor no3(o_no3, o_no1, o_no2);
nor no4(out, o_no3, o_no3);
```

NAND



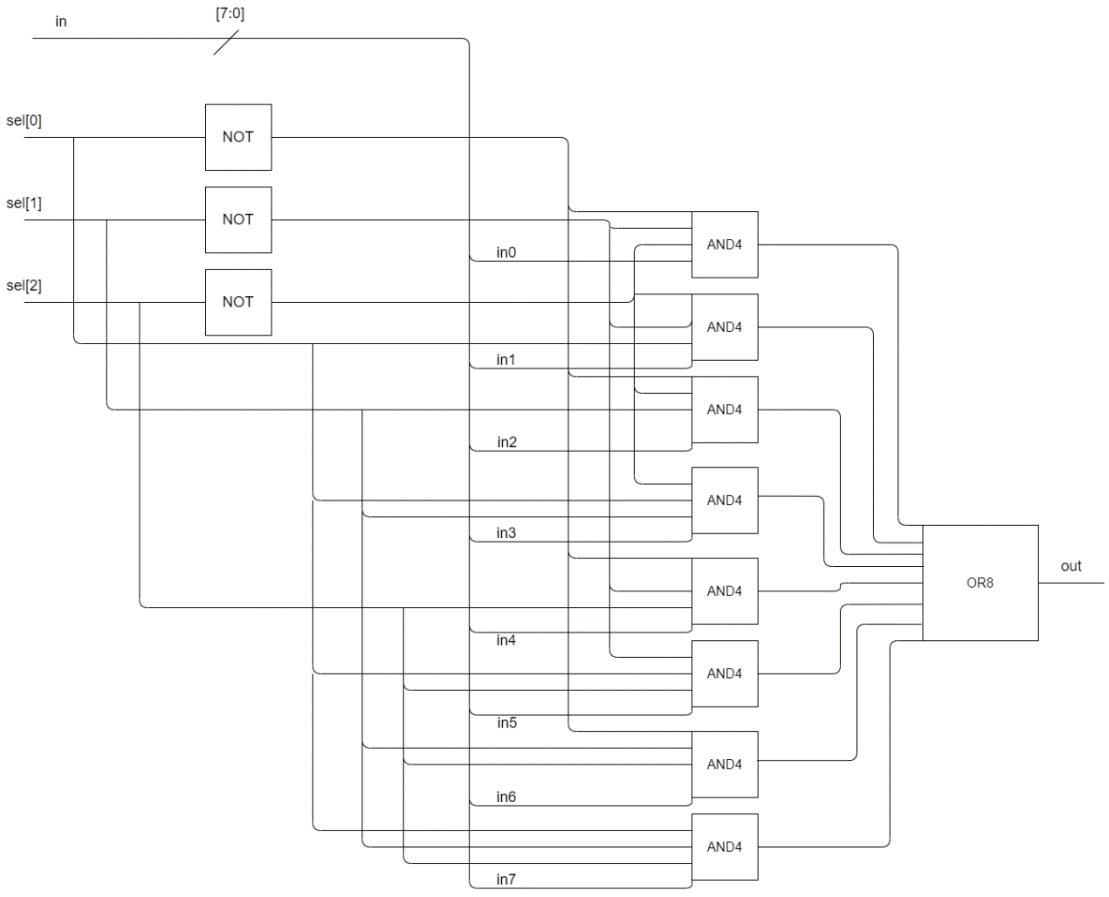

```
nor no_a(a_bar, in1, in1);
nor no_b(b_bar, in2, in2);
nor no1(o_no1, in1, in2);
nor no2(o_no2, a_bar, b_bar);
nor no3(out, o_no2, o_no1);
```

XOR

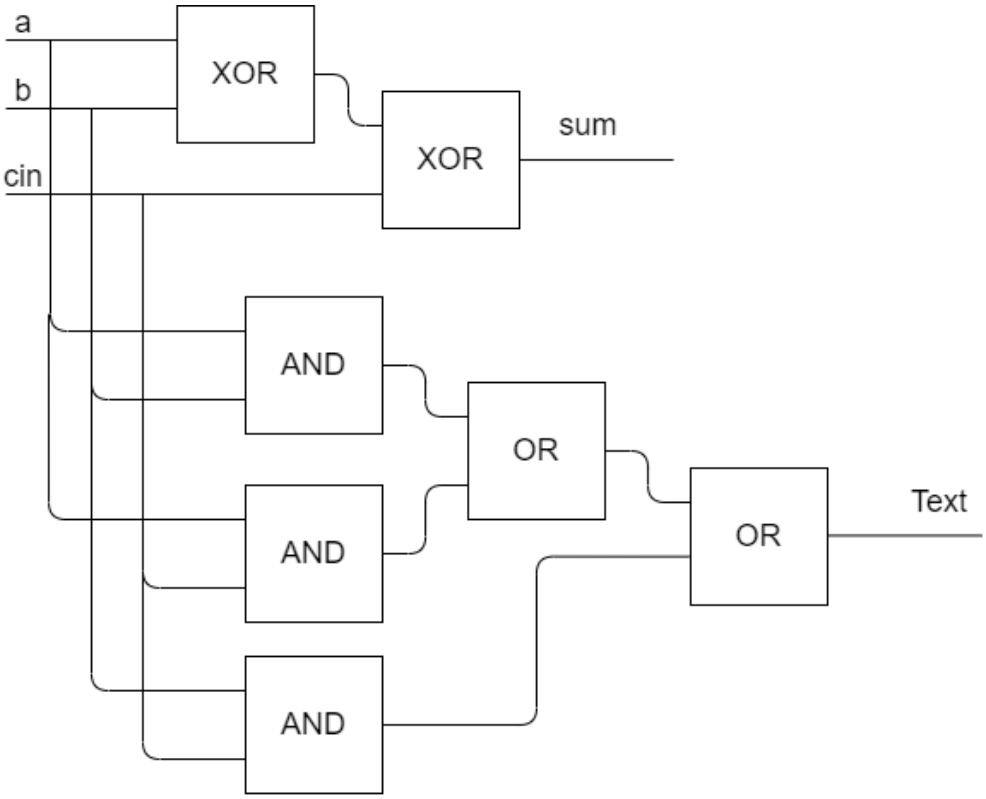



```
nor no_a(a_bar, in1, in1);
nor no_b(b_bar, in2, in2);
nor no1(o_no1, in1, in2);
nor no2(o_no2, a_bar, b_bar);
nor no3(o_no3, o_no2, o_no1);
nor no4(out, o_no3, o_no3);
```

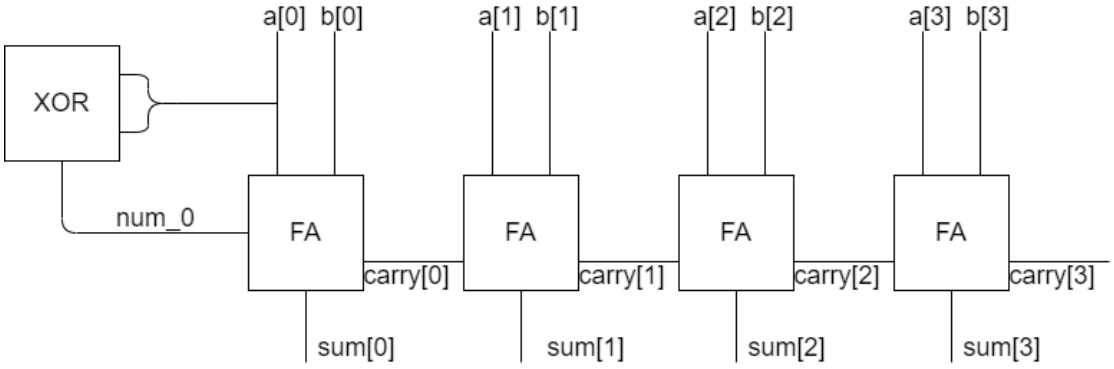
XNOR



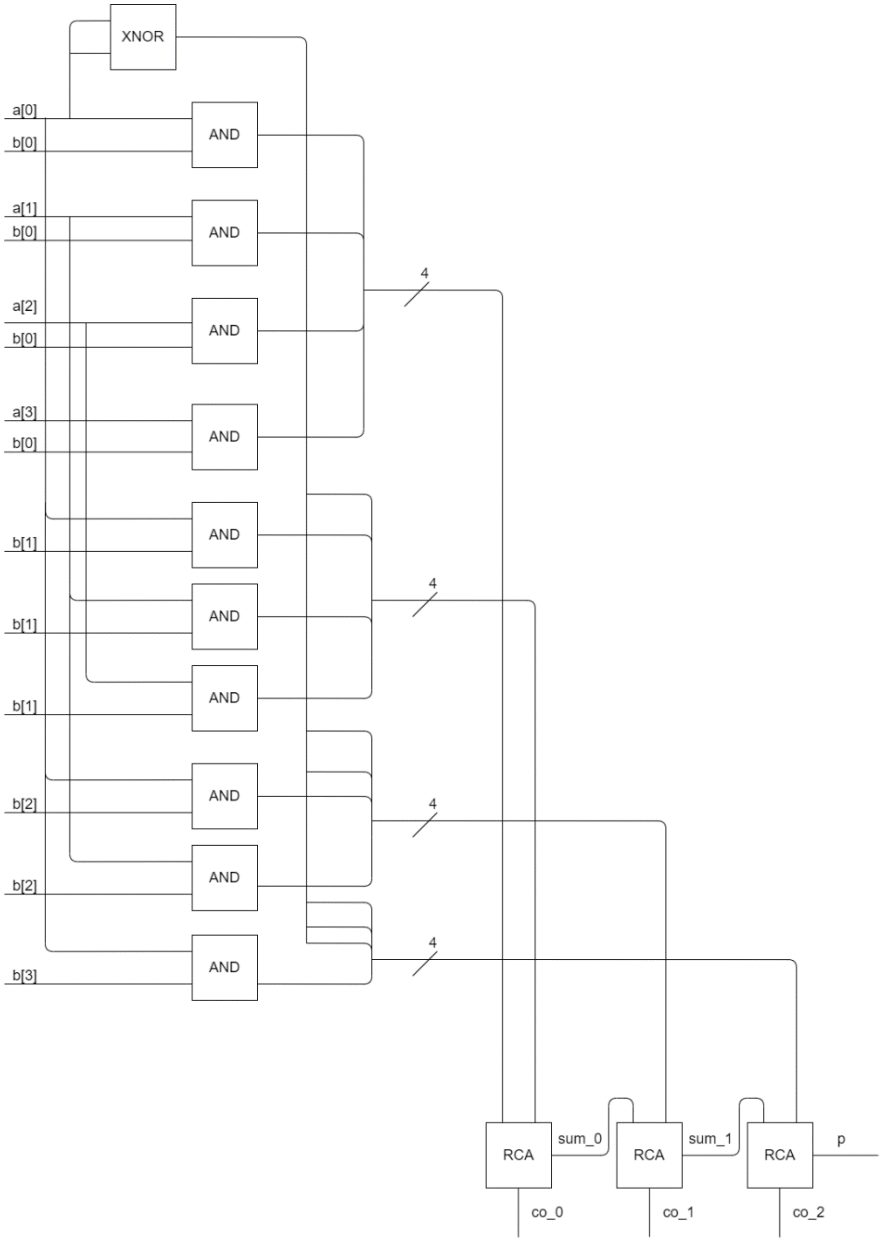
MUX 8bit



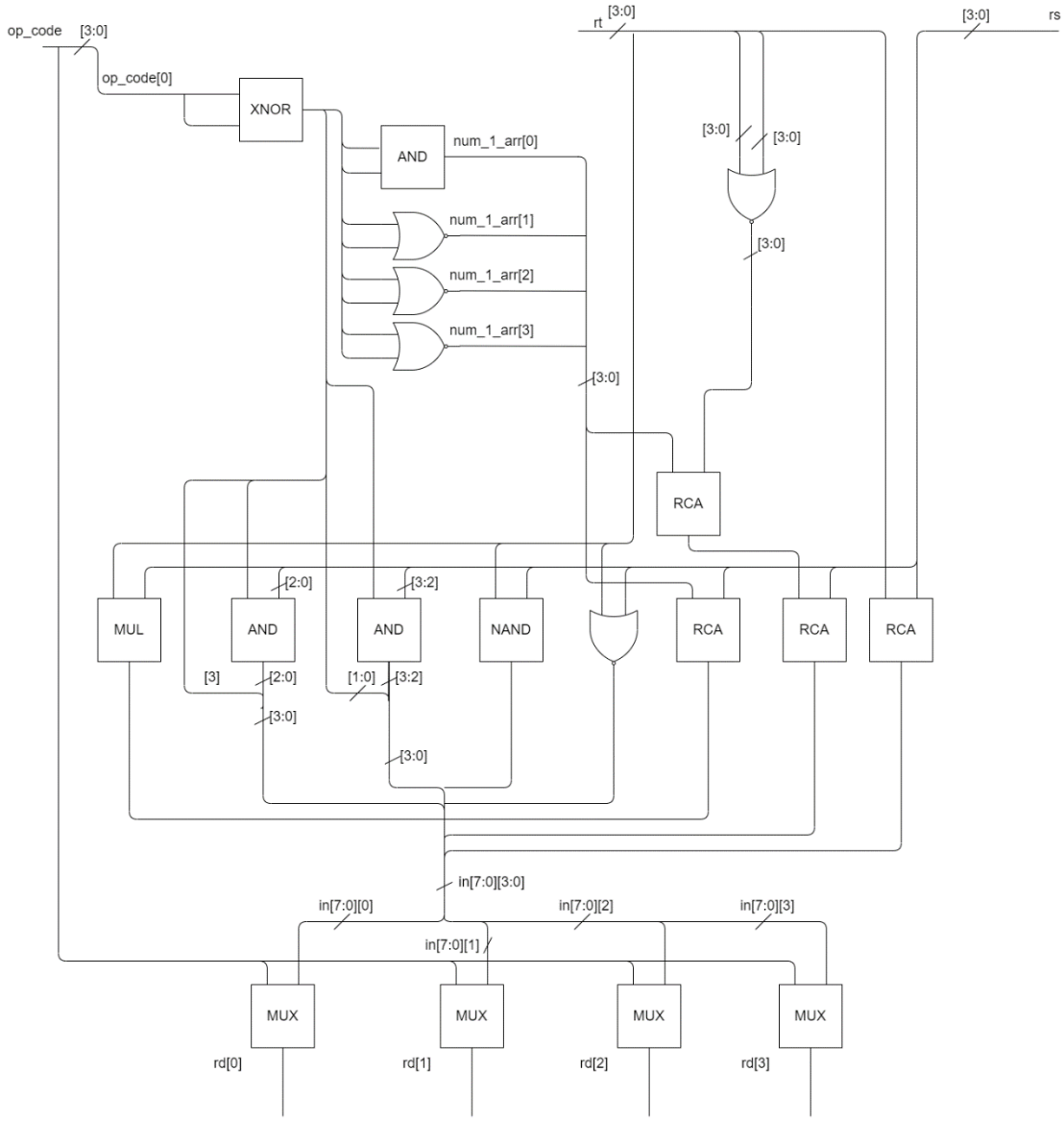
Full Adder



RippleCarryAdder 4bit


XNOR
a[0]
b[0]
AND
a[1]
b[0]
AND
a[2]
b[0]
AND
a[3]
b[0]
AND
4
b[1]
AND
b[1]
AND
b[1]
AND
4
b[2]
AND
b[2]
AND
4
b[3]
AND
RCA
sum_0
RCA
sum_1
RCA
p
co_0
co_1
co_2

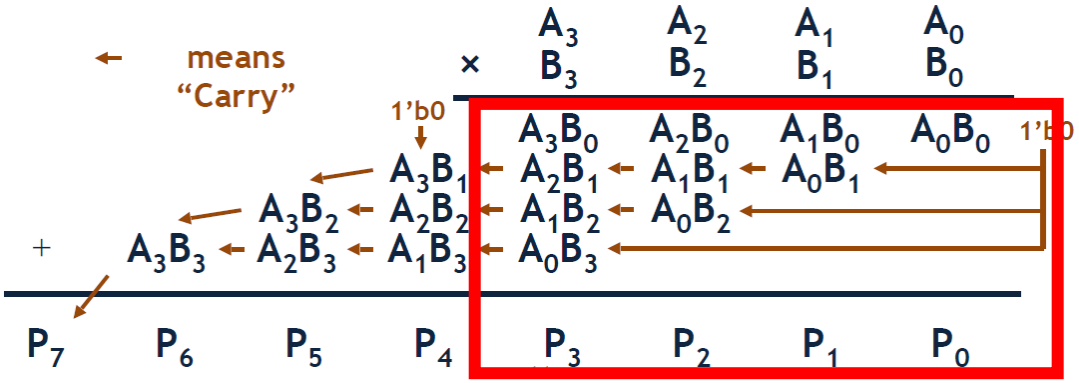
Multiplier



設計構想：

用前面題目實作過的基本 gate，和 ADDER、MULTIPLIER，照題目給的順序做連接，最後再把 OP_CODE 和操作完的 output 丟進 MUX 裡，就結束了。RippleCarryAdder 是用 4 個 FullAdder 串起來做的。FA 則是因為 sum3 個 bit 中有奇數個 1 就是 1 否則為 0，所以用 XOR 接上 3 個 input。而 carry_out 則是只要超過 2 個

bit 為 1 就為 1 否則為 0，所以就兩兩接上 AND，再用 OR 接起所有 AND 完的結果。Multiplier 因為不需要考慮超過 output[3]的 bit，



所以只需要將框起來的那 4 個 4bit 數加起來就知道 output 了，可以省下許多不必要的 gate。

Testbench 設計構想：

```
repeat (2**11) begin
  #1 {rs, rt, op_code} = {rs, rt, op_code} + 1'b1;
end
#1 $finish;
```

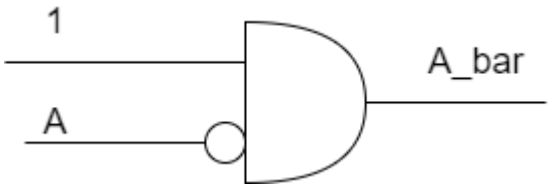
把 rs、rt、op_code，照順序排好，再重複加一 2^11 次，就會是 rs、rt 從 0000 0000 到 1111 1111 的組合，然後每個(rs, rt)組合都會跑過 8 種 op_code。

開發過程中的問題/學習：

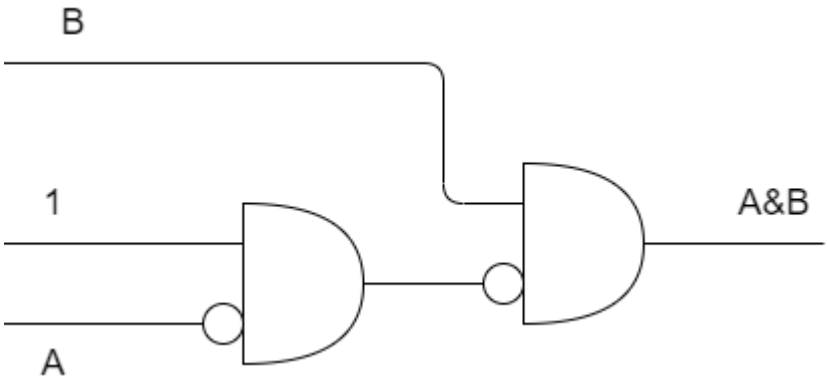
先寫出各個最小組成單元的 module 然後依次用小單位組成大單位，當結果出現問題的時候，就可以一個一個把小 module 的

input 和 output 抓出來看，debug 時輕鬆許多，整份 code 也變得比較容易讀懂。在跑 testbench 時，一開始跑不完所有迴圈，發現要把預設的 1ns/1ps 改成 1ps/1ps 才能跑完所有可能。

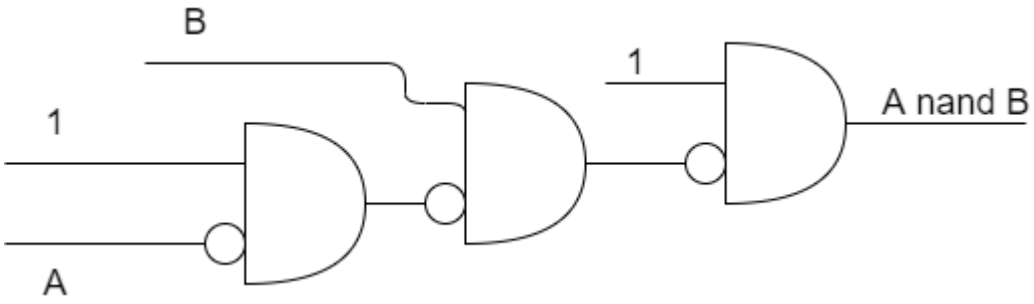
Additional Question



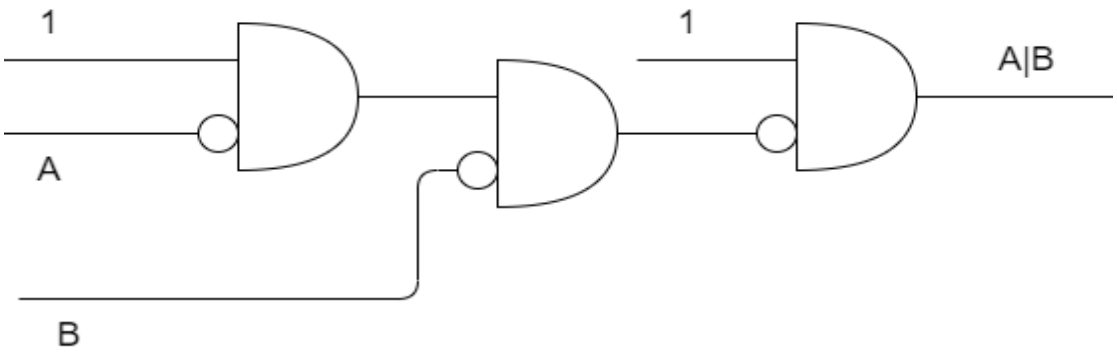
not 是 $(\overline{A} \& 1) = \overline{A}$



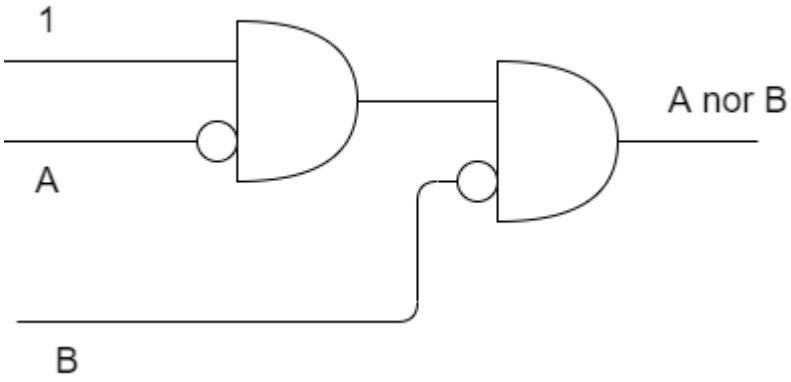
and 是$(\overline{A})$_bar = A，所以 A & B



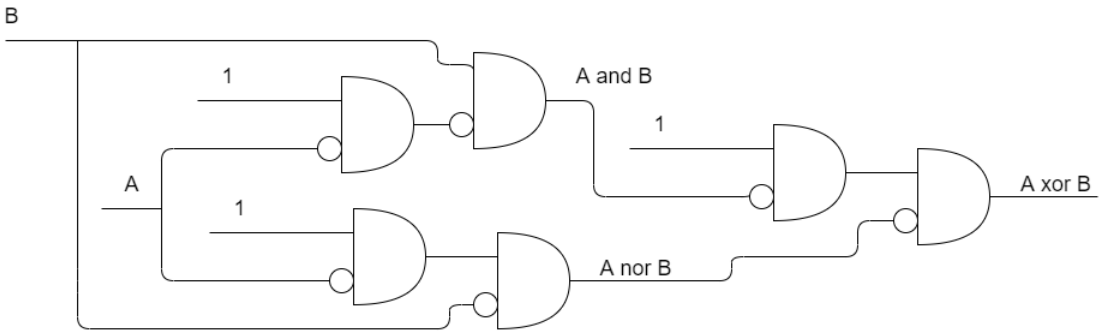
nand 是 not ((a nand b), (a&b))



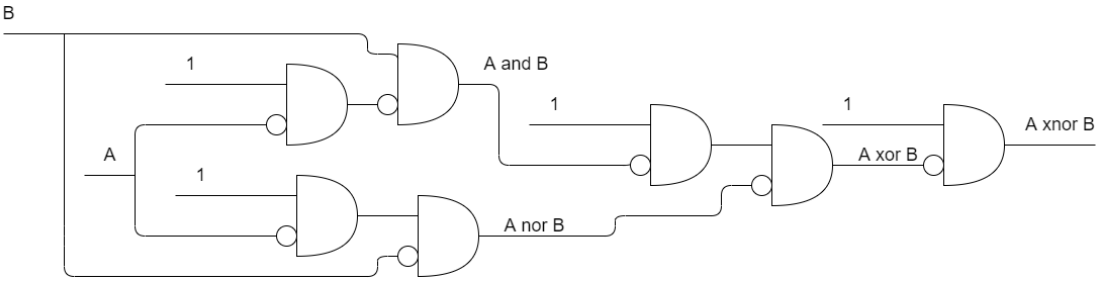
or 是因為$(\overline{A} \& \overline{B})$ = A nor B，所以(A nor B)_bar = A | B



nor 是因为($\overline{A}$ & $\overline{B}$) = A nor B



A xor B = (A nand B) & (A | B) = (A & B) nor (A nor B)



A xnor B = (A xor B)_bar