

**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);

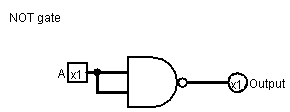
input a;

output out;

nand nand\_a\_a(out, a, a);

endmodule

* Design Graph



**Implement AND gate by NAND gate**

* Verilog Code

module AND\_by\_NAND(out, a, b);

input a, b;

output out;

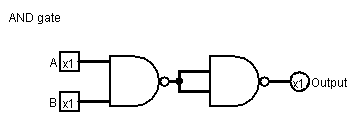
wire w\_nand\_a\_b;

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



**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;

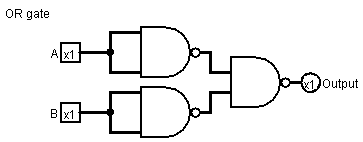
nand a\_itself(nand\_a, a, a);

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;

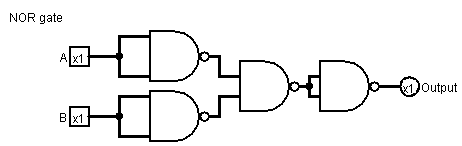
wire or\_a\_b;

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);

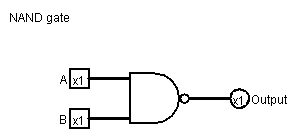
input a, b;

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);

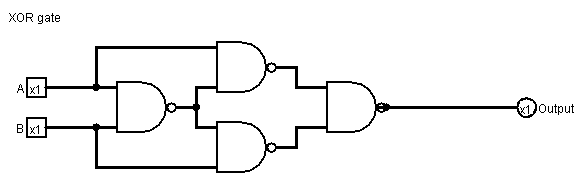
nand gene\_a\_nn(a\_nn, a, a\_NAND\_b);

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);

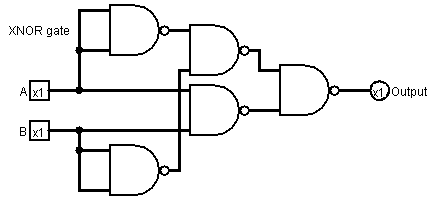
nand gene\_nand1(not\_a\_NAND\_not\_b, not\_a, not\_b);

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(

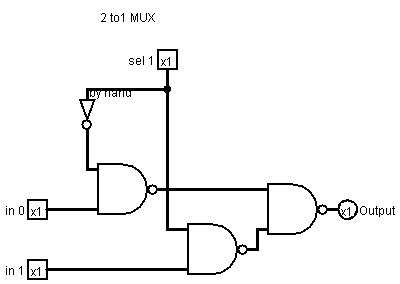
in\_1\_and, in[1], sel);

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]);

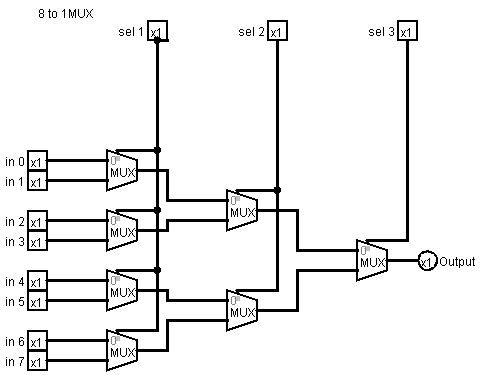
MUX\_2\_to\_1\_by\_NAND MUX\_layer2\_0(layer1[1:0], sel[1], layer2[0]);

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);

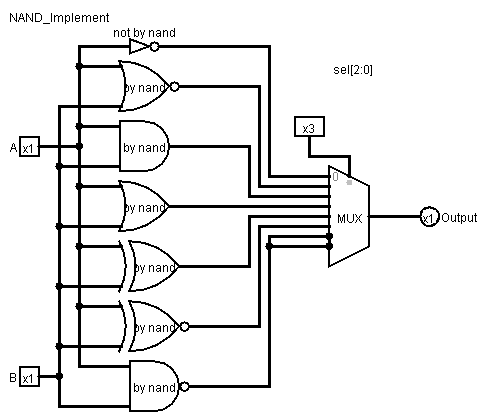
NAND\_by\_NAND gene\_df\_6 (dataflow[6], a, b);

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);

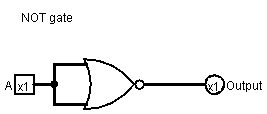
input a;

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;

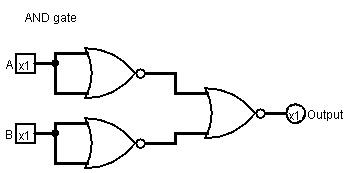
nor gene\_n\_a(n\_a, a);

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;

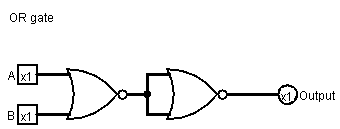
wire a\_nor\_b;

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);

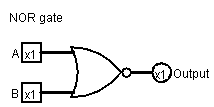
input a, b;

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;

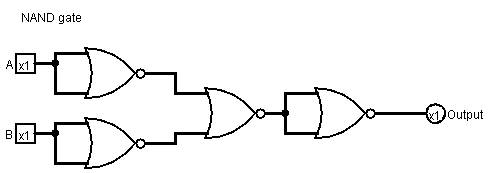
wire a\_and\_b;

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);

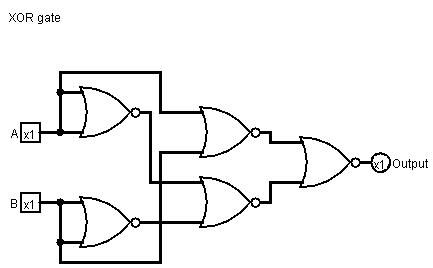
nor gene\_nor1(not\_a\_NOR\_not\_b, not\_a, not\_b);

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);

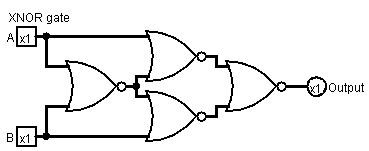
nor gene\_a\_nor\_norab(a\_nor\_norab, a, a\_NOR\_b);

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, sel);

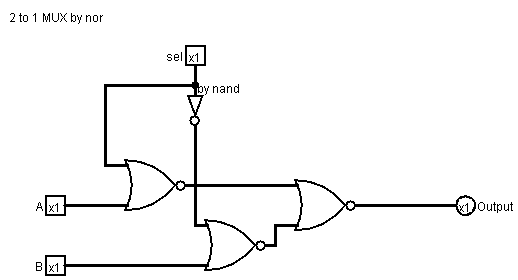
AND\_by\_NOR gene\_in\_0(in\_0\_and, in[0], n\_sel);

AND\_by\_NOR gene\_in\_1(in\_1\_and, in[1], sel);

OR\_by\_NOR gene\_out(out, in\_0\_and, 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]);

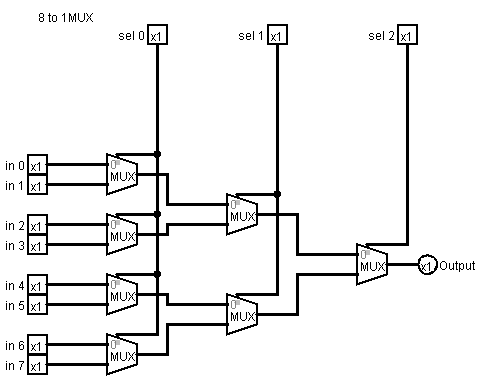
MUX\_2\_to\_1\_by\_NOR MUX\_layer2\_0(layer1[1:0], sel[1], layer2[0]);

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);

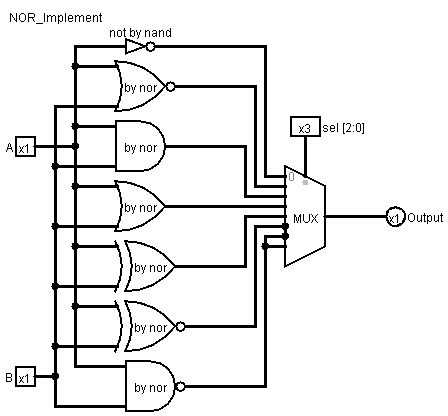
NAND\_by\_NAND gene\_df\_6 (dataflow[6], a, b);

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。

**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);

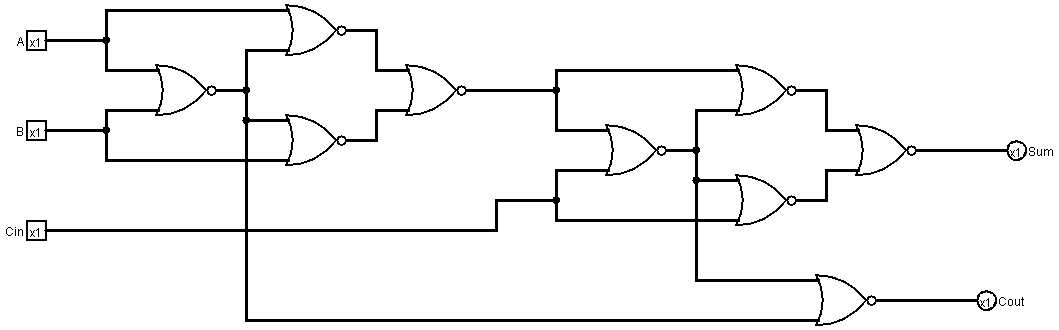
nor gene\_lncnc(layer1\_nor\_cin\_nor\_cin, cin, layer1\_nor\_cin);

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], alwaysZero, sum[0], carry[0]);

FullAdder\_1bit FA\_2(a[1], b[1], carry[0], sum[1], carry[1]);

FullAdder\_1bit FA\_3(a[2], b[2], carry[1], sum[2], carry[2]);

FullAdder\_1bit FA\_4(a[3], b[3], carry[2], sum[3], cout);

endmodule

FullAdder\_1bit FA\_3(a[2], b[2], carry[1],

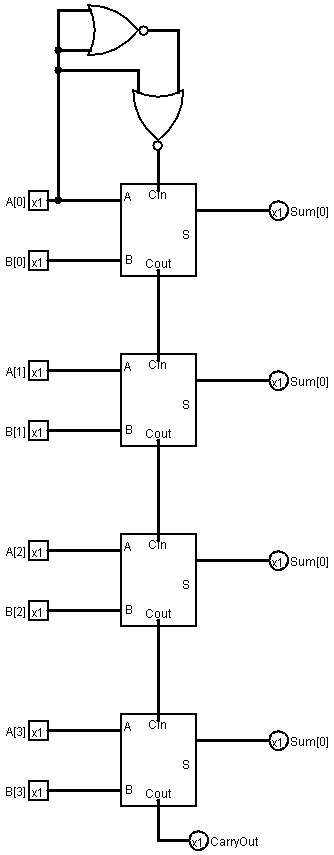
sum[2], carry[2]);

FullAdder\_1bit FA\_4(a[3], b[3], carry[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]);

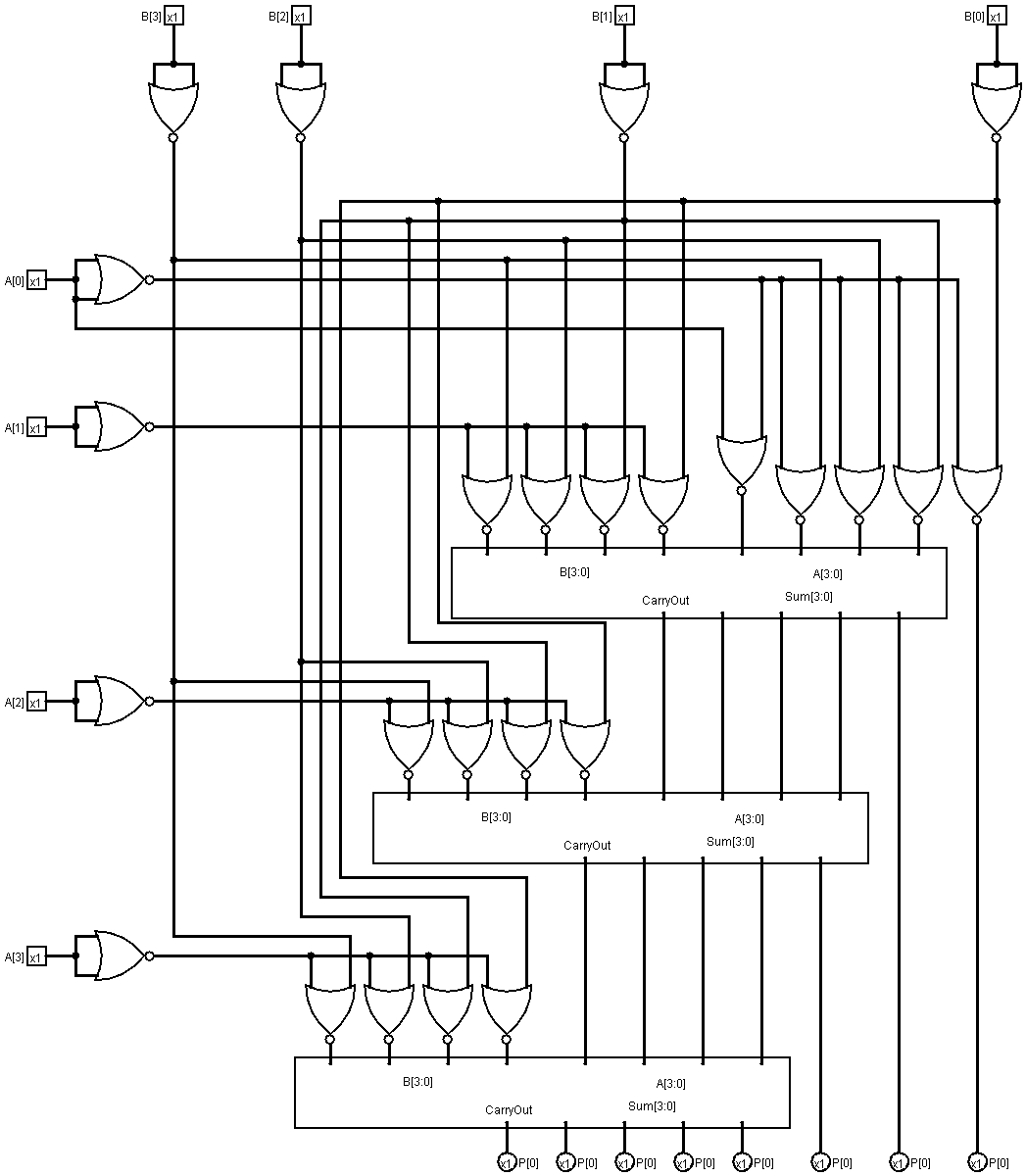
RCA\_4bit RCA\_1({alwaysZero, ab[0][3:1]}, ab[1], {layer[0][2:0], p[1]}, layer[0][3]);

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(

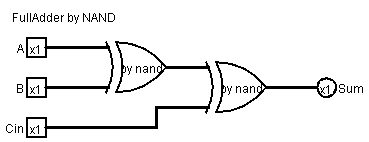
XNOR\_a\_b, a, b);

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]);

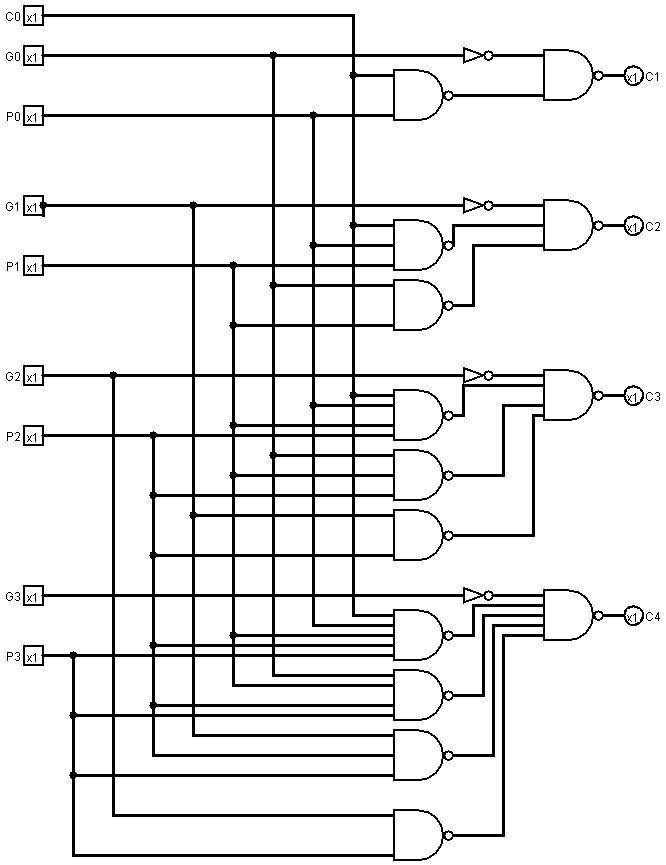
nand gene\_cout\_2(Cout[2], n\_G[2], layer3[0], layer3[1], layer3[2]);

nand gene\_cout\_3(Cout[3], n\_G[3], layer4[0], layer4[1], layer4[2], layer4[3]);

endmodule

Design Graph

* 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]);

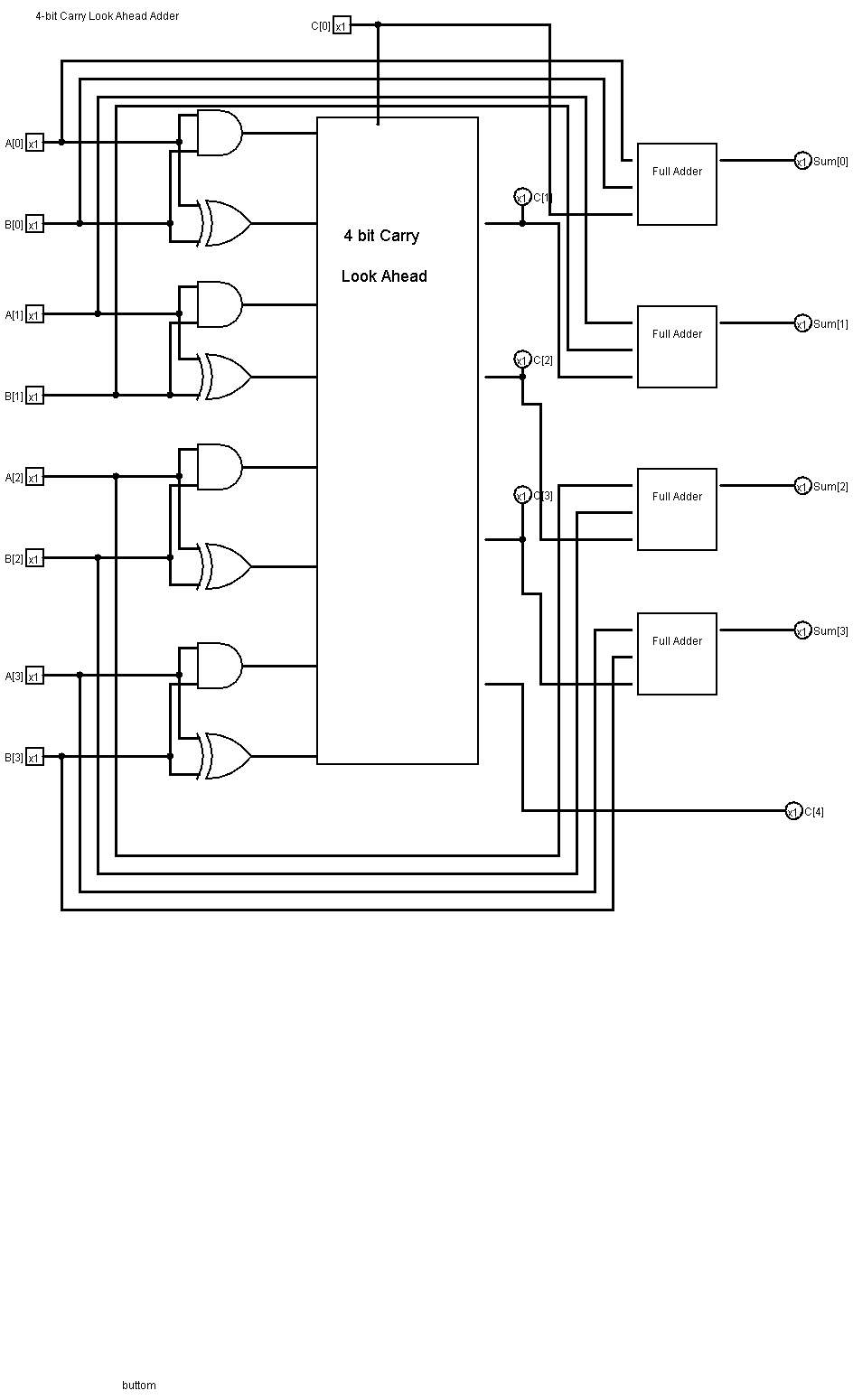
FullAdder gene\_sum1(a[1], b[1], CarryOut[0], sum[1]);

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。

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。