LAB 1

Team 6

鄭聰明 **Jason Theodorus Pratama (109006236)**

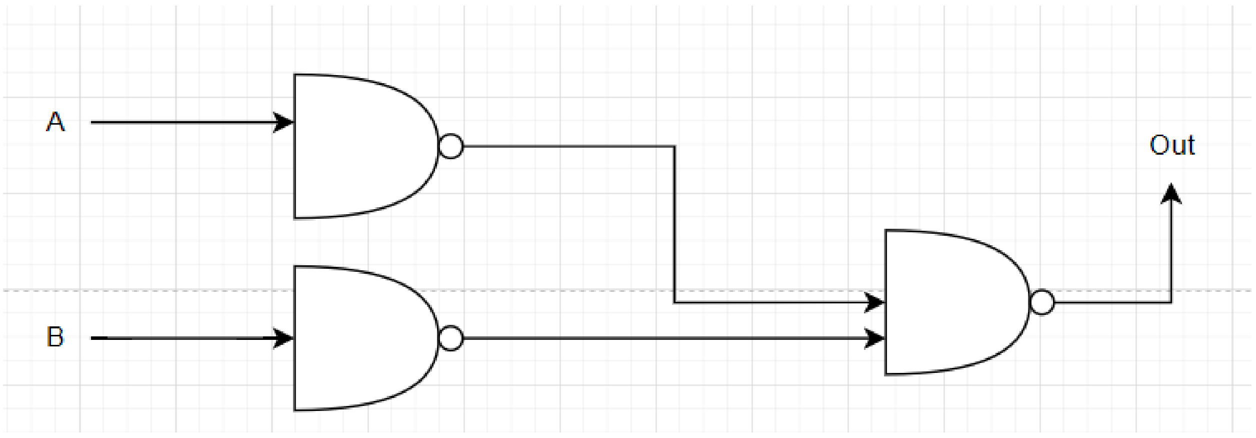
鐵特德 **Tuguldur Tserenbaljir (109006271)**

1. 8-bit Ripple Carry Adder (RCA)

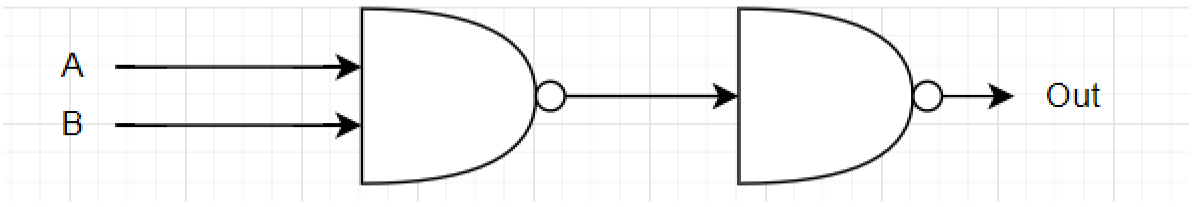
a. Design

For the 1bit Full Adder, we are told to use Majority gates as our implementation, so we made it as below, and to implement the Majority Gate from the given circuit, we use NAND Gates where the circuit is as follows.

OR using NAND

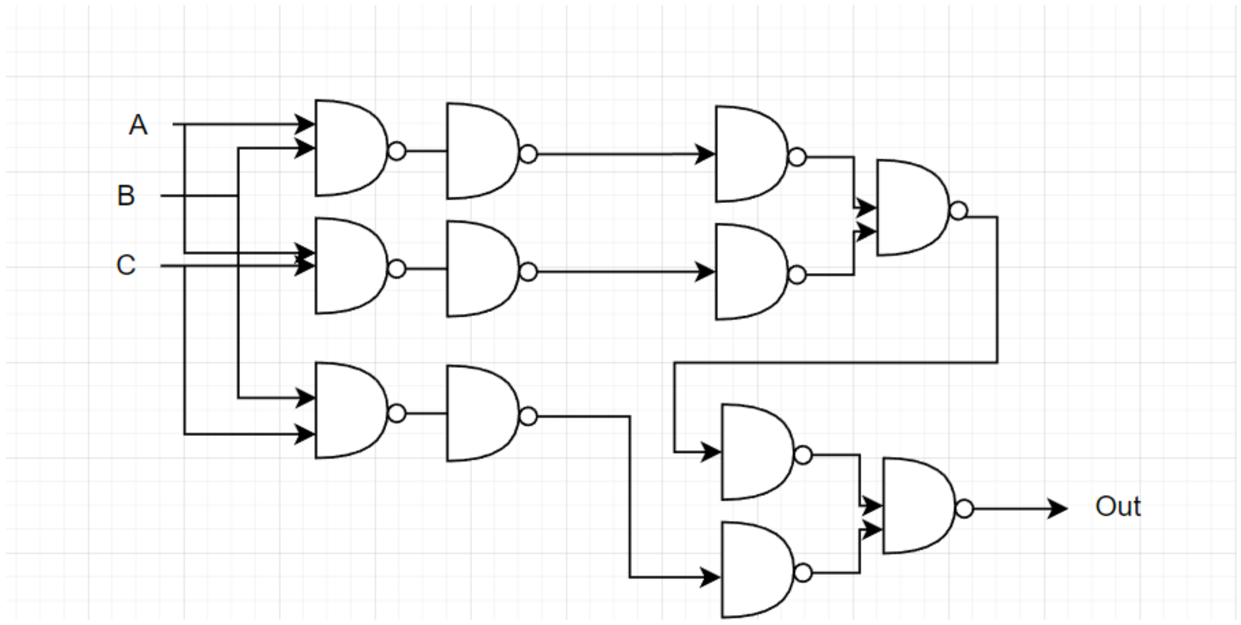


AND using NAND

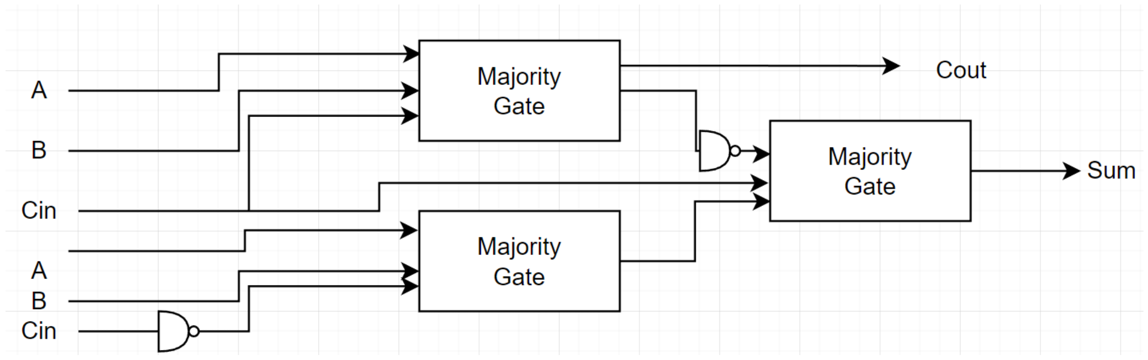


With the following circuit we can receive the same output as AND and OR with the NAND Gates.

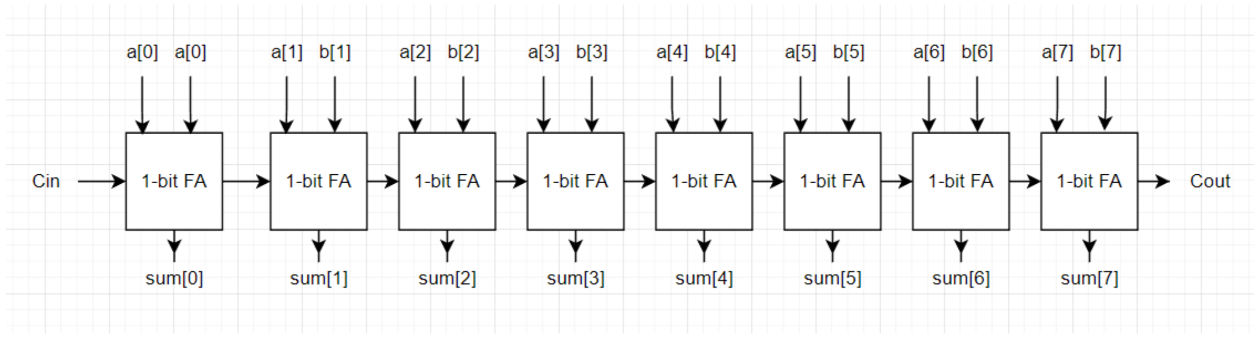
Majority Gate with Nand Gates



1bit Full Adder with Majority Gates



8bit Ripple Carry Adder with 1bit Full Adders



A Ripple Carry Adder is just like its name, where it is an adder, it is a combinational logic circuit, and it is used for the purpose of adding two n-bit binary numbers, it requires n full adders in its circuits for adding two n-bit binary numbers. In our assignment's case the n=8, because it is an 8-bit Ripple Carry Adder. Ripple Carry Adder is also known as n-bit parallel adder. In a nutshell, our Ripple Carry Adder uses 8 Full-Adders that we implemented using NAND Gates. And made with hierarchical design as below. Where each Full Adder have 3 inputs a and b, also the Cin , and output the Cout, and sum.

b. Circuit

```
`timescale 1ns/1ps

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

   wire c1,c2,c3,c4,c5,c6,c7;

   Full_Adder FADD0(a[0], b[0], cin, c1, sum[0]);
   Full_Adder FADD1(a[1], b[1], c1, c2, sum[1]);
   Full_Adder FADD2(a[2], b[2], c2, c3, sum[2]);
   Full_Adder FADD3(a[3], b[3], c3, c4, sum[3]);
   Full_Adder FADD4(a[4], b[4], c4, c5, sum[4]);
   Full_Adder FADD5(a[5], b[5], c5, c6, sum[5]);
   Full_Adder FADD6(a[6], b[6], c6, c7, sum[6]);
   Full_Adder FADD7(a[7], b[7], c7, cout, sum[7]);

endmodule

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

   wire notc,notcout, w1;

   nand nand1(notcin, cin);
   nand nand2(notcout,cout);

   Majority M1(a, b, cin, cout);
   Majority M2(notcin, b, a, w1);
   Majority M3(notcout, cin, w1, sum);

endmodule

module Majority(a, b, c, out);
   input a, b, c;
   output out;

   wire w1,w2,w3,w4;
   AND and1(a,b, w1);
   AND and2(a, c , w2);
   AND and3(b, c ,w3);

   OR or1(w1, w2, w4);
   OR or2(w4,w3, out);

endmodule

module AND (a, b , out);
   input a,b;
   output out;

   wire w1;

   nand nand1(w1, a, b);
   nand nand2(out, w1);

endmodule

module OR (a, b, out);
```

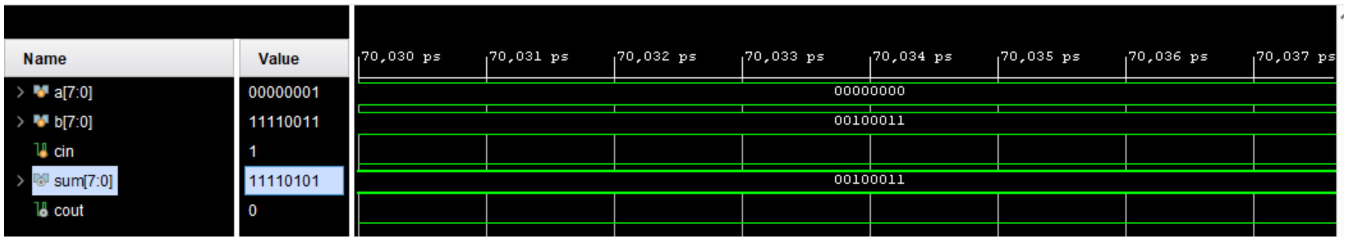
```
    input a,b;
    output out;

    wire w1,w2;

    nand nand1(w1, a);
    nand nand2(w2, b);
    nand nand3(out, w1, w2);

endmodule
```

c. Testbench

```
`timescale 1ns/1ps

module Ripple_Carry_Adder_t;
    reg [7:0] a, b;
    reg cin;
    wire [7:0] sum;
    wire cout;

    Ripple_Carry_Adder RCA(
        .a(a),
        .b(b),
        .cin(cin),
        .cout(cout),
        .sum(sum)
    );

    initial begin
        cin = 1'b0;
        a = 8'b0;
        b = 8'b0;
        repeat (2 ** 8) begin
            repeat (2 ** 8) begin
                repeat (2) begin
                    #1 cin = cin + 1'b1;
                end
                b = b + 1'b1;
                a = a + 1'b1;
            end
        end
        #1 $finish;
    end

endmodule
```

d. Waveform

The design is tested correct with the testbench and the proof is shown in the waveform where the 8bit binary 0000000 + 00100011 becomes 00100011, as well as the other inputs.

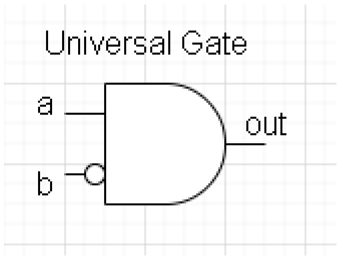


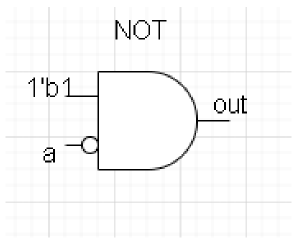
Name	Value
a[7:0]	00000001
b[7:0]	11110011
cin	1
sum[7:0]	11110101
cout	0

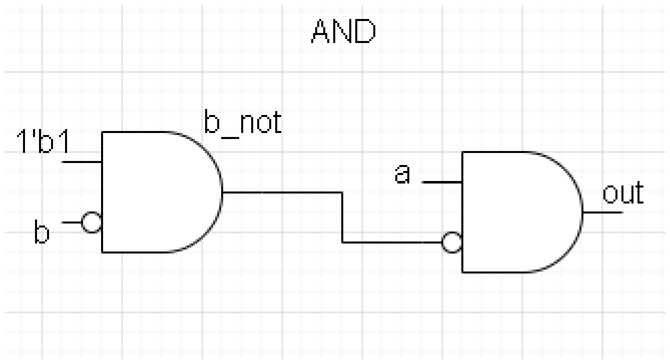
2. Decode and Execute

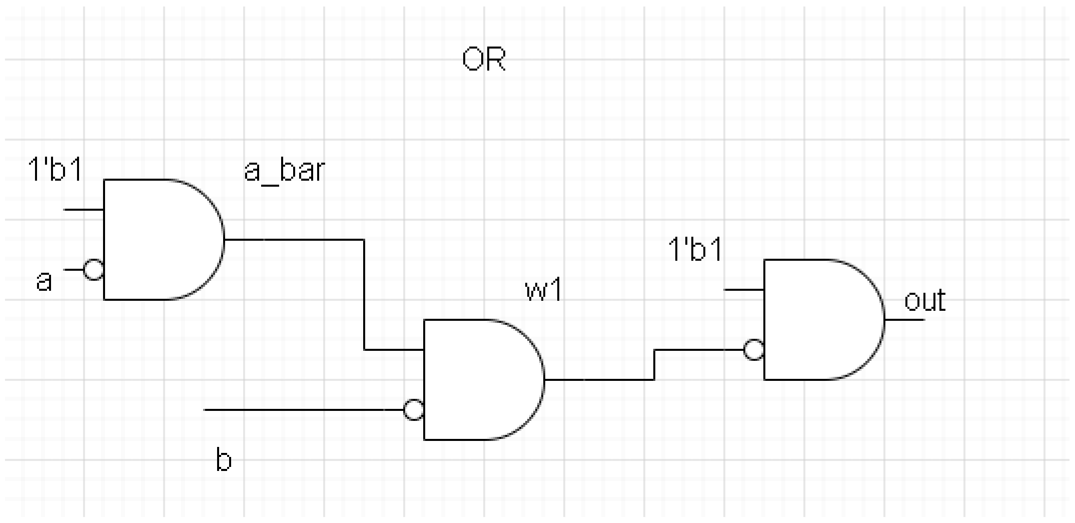
a. Design

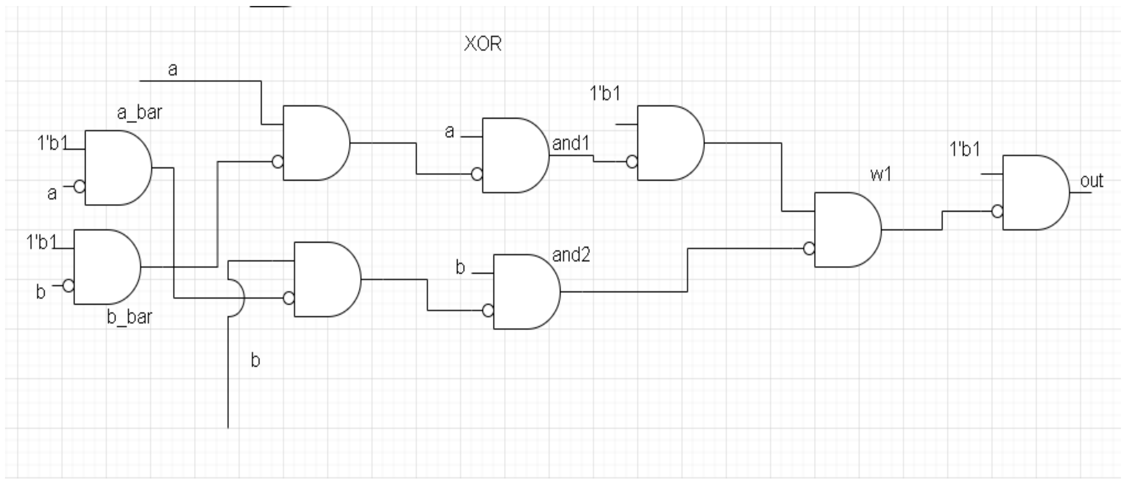
For the following question, we first used the universal gate to implement all of the basic gates NOT, AND, OR,XOR and XNOR. Our NOT gate module will only have one input and 1’b1 because of nod in the Universal Gate. Thus, we use them instead of original gates to design our blocks in the next coming steps.

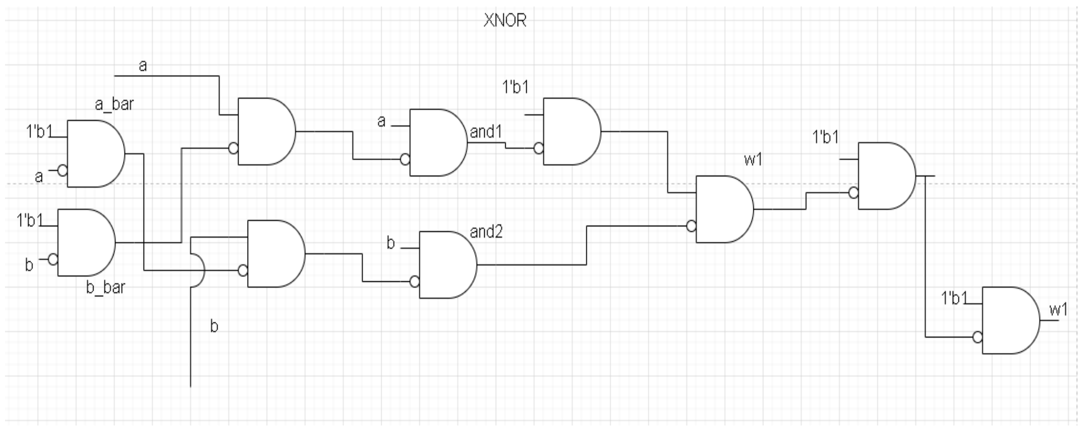




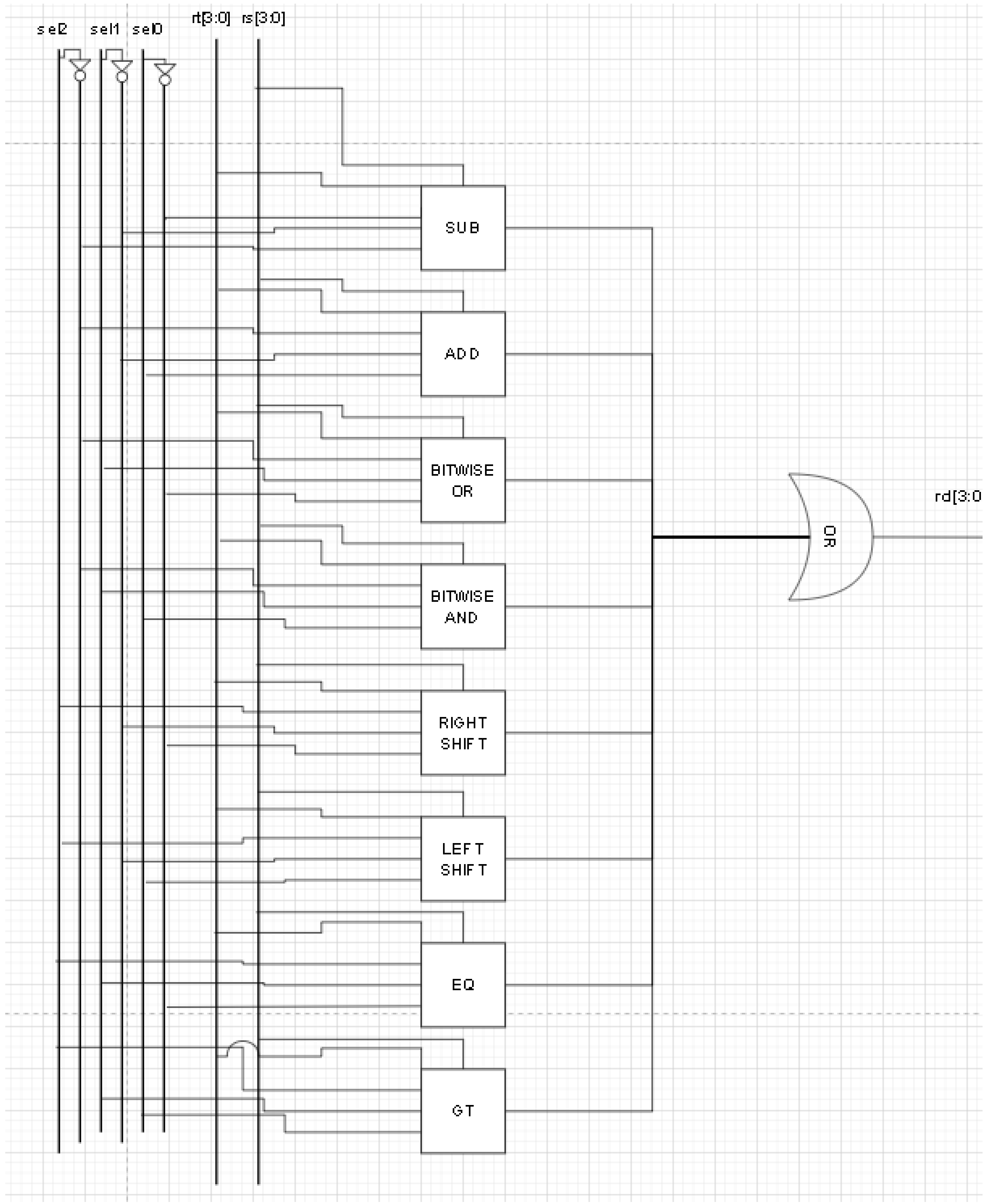








Since we have 8 cases to implement and 3 selection lines for our design, we created a 8-to-1 MUX, so that each case can be tested for our design. Starting with the value 000 will be the SUB function and until 111 will be the GT function for our module. After all of the blocks are connected to their respective selection lines their output or wires will all go into an OR which is designed from the universal gate from above and will give out the final output for our design.



Depending on the chosen selection line the 8 blocks will calculate. For example, the add blocks uses a full adder for every bit while the SUB block will take the 2's complement and also use the full adder. Bitwise and block and or block will calculate evaluate every bit for the output. Since our inputs are 4 bits the AND and OR block will calculate for each and every bit. Similarly to the bitwise AND and OR block, it goes the same for all of the other blocks as well. In which, our output will definitely be 4-bit as well.

b. Circuit

```
`timescale 1ns/1ps

module Decode_And_Execute(rs, rt, sel, rd);
   input [3:0] rs, rt;
   input [2:0] sel;
   output [3:0] rd;

   wire [3:0] case0,case1,case2,case3,case4,case5,case6,case7;
   wire [3:0] w1;


   //SUB
   COMPLEMENT C1(rt, w1);
   ADD Sub(rs, w1, case0);

   //RCA_4bit
   ADD A1(rs, rt, case1);

   //BITWISE OR
   BIT_OR OR1(rs, rt, case2);

   //BITWISE AND
   BIT_AND AND1(rs, rt, case3);


   //RIGHT SHIFT
   shift R0(rt[1], case4[0]);
   shift R1(rt[2], case4[1]);
   shift R2(rt[3], case4[2]);
   shift R3(rt[3], case4[3]);

   //LEFT SHIFT
   shift L0(rs[3], case5[0]);
```

```
    shift L1(rs[0], case5[1]);
    shift L2(rs[1], case5[2]);
    shift L3(rs[2], case5[3]);

    //EQ and GT
    COMPARATOR Comp(rs, rt, case7, case6);

    //MUX
    MUX_8X1_4bit M1(case0,case1,case2,case3,case4,case5,case6,case7,sel, rd);

endmodule

module shift (in, out);
    input in;
    output out;

    wire w;

    NOT_GATE SH0(w, in);
    NOT_GATE SH1(out, w);

endmodule

module MUX_2x1_4bit(a, b, sel, out);
    input [3:0] a, b;
    input sel;
    output [3:0] out;

    wire not_sel;

    wire w1, w2, w3, w4,w5, w6, w7, w8;

    NOT_GATE N(not_sel, sel);

    AND_GATE M2_A0(w1, a[0], not_sel);
    AND_GATE M2_A1(w2, a[1], not_sel);
    AND_GATE M2_A2(w3, a[2], not_sel);
    AND_GATE M2_A3(w4, a[3], not_sel);

    AND_GATE M2_B0(w5, b[0], sel);
    AND_GATE M2_B1(w6, b[1], sel);
    AND_GATE M2_B2(w7, b[2], sel);
    AND_GATE M2_B3(w8, b[3], sel);

    OR_GATE M2_O0(out[0], w1, w5);
    OR_GATE M2_O1(out[1], w2, w6);
```

```
   OR_GATE M2_O2(out[2], w3, w7);
   OR_GATE M2_O3(out[3], w4, w8);

endmodule

module MUX_8X1_4bit
(case0,case1,case2,case3,case4,case5,case6,case7,sel,out);
   input [3:0] case0,case1,case2,case3,case4,case5,case6,case7;
   input [2:0] sel;
   output [3:0] out;

   wire [3:0] w1,w2,w3,w4,w12,w34;

   MUX_2x1_4bit u1(case0, case1, sel[0], w1);
   MUX_2x1_4bit u2(case2, case3, sel[0], w2);
   MUX_2x1_4bit u3(case4, case5, sel[0], w3);
   MUX_2x1_4bit u4(case6, case7, sel[0], w4);

   MUX_2x1_4bit u5(w1, w2, sel[1], w12);
   MUX_2x1_4bit u6(w3, w4, sel[1], w34);

   MUX_2x1_4bit u7(w12, w34, sel[2], out);

endmodule

module COMPARATOR(a, b, eq, gt);
   input [3:0] a, b;
   output [3:0] eq, gt;

   wire E, G;

   //eq
   wire XNOR_0, XNOR_1, XNOR_2, XNOR_3;
   wire XNOR_23, XNOR_012;

   wire BB_0, BB_1, BB_2, BB_3;
   wire AB_0, AB_1;
   wire AB_2, AB_3;
   wire AB_01, AB_012, AB_0123;
   wire ACC01, ACC012;



   XNOR_GATE X0(XNOR_0, a[0], b[0]);
   XNOR_GATE X1(XNOR_1, a[1], b[1]);
   XNOR_GATE X2(XNOR_2, a[2], b[2]);
```

```
  XNOR_GATE X3(XNOR_3, a[3], b[3]);


  AND_GATE EQ01(XNOR_23, XNOR_2, XNOR_3);
  AND_GATE EQ012(XNOR_123, XNOR_1, XNOR_23);
  AND_GATE EQ0123(E, XNOR_0, XNOR_123);

  //gt


  NOT_GATE N0(BB_0, a[0]);
  NOT_GATE N1(BB_1, a[1]);
  NOT_GATE N2(BB_2, a[2]);
  NOT_GATE N3(BB_3, a[3]);


  AND_GATE A0(AB_0, b[0], BB_0);
  AND_GATE A1(AB_1, b[1], BB_1);
  AND_GATE A2(AB_2, b[2], BB_2);
  AND_GATE A3(AB_3, b[3], BB_3);

  AND_GATE GT1(AB_01, AB_2, XNOR_3);
  AND_GATE GT2(AB_012, AB_1, XNOR_23);
  AND_GATE GT3(AB_0123, AB_0, XNOR_123);


  OR_GATE O0(ACC01, AB_3, AB_01);
  OR_GATE O1(ACC012, ACC01, AB_012);
  OR_GATE O2(G, ACC012, AB_0123);

  //Value
  shift E0(E, eq[0]);
  shift E1(1'b1, eq[1]);
  shift E2(1'b1, eq[2]);
  shift E3(1'b1, eq[3]);

  shift G0(G, gt[0]);
  shift G1(1'b1, gt[1]);
  shift G2(1'b0, gt[2]);
  shift G3(1'b1, gt[3]);

endmodule

module BIT_AND (a, b, out);
  input [3:0] a, b;
  output [3:0] out;
```

```
    AND_GATE A0(out[0], a[0], b[0]);
    AND_GATE A1(out[1], a[1], b[1]);
    AND_GATE A2(out[2], a[2], b[2]);
    AND_GATE A3(out[3], a[3], b[3]);

endmodule

module BIT_OR(a, b, out);
    input [3:0] a, b;
    output [3:0] out;

    OR_GATE O0(out[0], a[0], b[0]);
    OR_GATE O1(out[1], a[1], b[1]);
    OR_GATE O2(out[2], a[2], b[2]);
    OR_GATE O3(out[3], a[3], b[3]);

endmodule

module COMPLEMENT(a, out);
    input [3:0] a;
    output [3:0] out;

    wire [3:0] f;

    NOT_GATE N0(f[0], a[0]);
    NOT_GATE N1(f[1], a[1]);
    NOT_GATE N2(f[2], a[2]);
    NOT_GATE N3(f[3], a[3]);

    ADD COMP1(f, 4'b0001, out);

endmodule

module ADD(a, b, s);
    input [3:0] a, b;
    output [3:0] s;

    wire f0, f1, f2, f3;

    Full_Adder A0(a[0], b[0], 1'b0, f0, s[0]);
    Full_Adder A1(a[1], b[1], f0, f1, s[1]);
    Full_Adder A2(a[2], b[2], f1, f2, s[2]);
    Full_Adder A3(a[3], b[3], f2, f3, s[3]);

endmodule
```

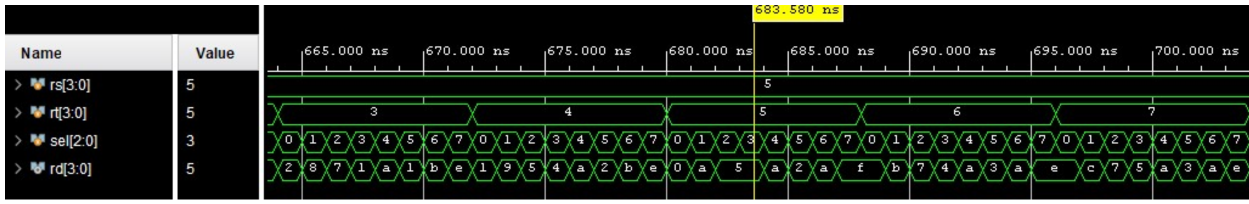
```
module Full_Adder(a, b, cin, out, sum);
    input a, b, cin;
    output out, sum;

    wire XOR_AB, AND_AB, AND_CX;

    XOR_GATE FA1(XOR_AB, a, b);
    XOR_GATE FA2(sum, XOR_AB, cin);

    AND_GATE A1(AND_AB, a, b);
    AND_GATE A2(AND_CX, XOR_AB, cin);

    OR_GATE O(out, AND_AB, AND_CX);

endmodule

module AND_GATE(out, a, b);
    input a, b;
    output out;

    wire not_b;

    NOT_GATE A1(not_b, b);

    Universal_Gate A2(out, a, not_b);

endmodule

module OR_GATE(out, a, b);
    input a, b;
    output out;

    wire a_bar, w1;

    NOT_GATE OR1(a_bar, a);
    NOT_GATE OR2(out, w1);

    Universal_Gate OR3(w1, a_bar, b);

endmodule

module NOT_GATE(out, a);
    input a;
    output out;
    Universal_Gate N(out, 1'b1, a);

endmodule

module XOR_GATE (out, a, b);
    input a, b;
    output out;

    wire a_bar, b_bar;
    wire and1, and2;

    NOT_GATE NOT1(a_bar, a);
    NOT_GATE NOT2(b_bar, b);

    AND_GATE NOT3(and1, a, b_bar);
    AND_GATE NOT4(and2, a_bar, b);

    OR_GATE NOT5(out, and1, and2);

endmodule

module XNOR_GATE(out, a, b);
    input a, b;
    output out;

    wire w1;

    XOR_GATE XNOR1(w1, a, b);

    NOT_GATE XNOR2(out, w1);

endmodule
```

c. **Testbench**

```
`timescale 1ns / 1ps

module Decode_and_Execute_t;
    reg [3:0] rs, rt;
    reg [2:0] sel;
    wire [3:0] rd;

    Decode_And_Execute DE(
        .rs(rs),
        .rt(rt),
        .sel(sel),
        .rd(rd)
    );

    initial begin
        sel = 3'd0;
        rs = 4'd0;
        rt = 4'd0;
        repeat(2 ** 4) begin
            repeat(2 ** 4) begin
                repeat(2 ** 3) begin
                    #1 sel = sel + 1'b1;
                end
                rt = rt + 1'b1;
            end
            rs = rs + 1'b1;
        end
        #1 $finish;
    end

endmodule
```

d. Waveform

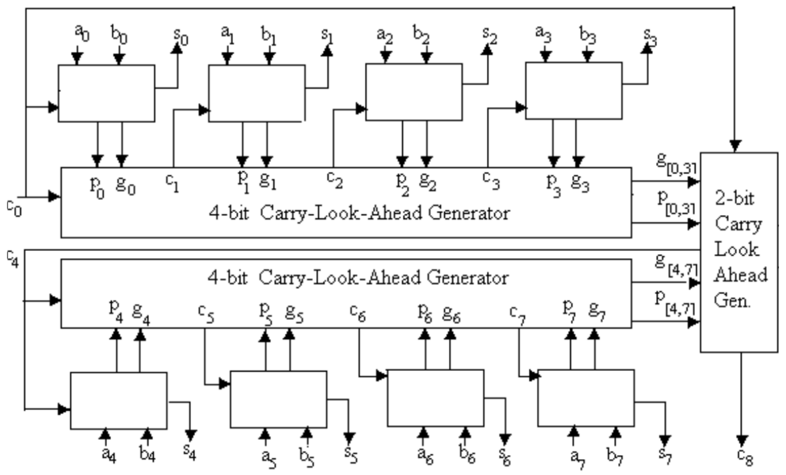
We test our code by the waveform which we check every input, selection line and outputs respectively.



3. 8-bit CLA

a. Design

For following 8-bit Carry-Look-Ahead Adder, it will consist of two 4-bit CLA, one 2-bit CLA and 8 Prop-and-Gen blocks which all of it will be designed by nand gates (Figure 3.1).



a0 b0 s0 a1 b1 s1 a2 b2 s2 a3 b3 s3
p0 g0 c1 p1 g1 c2 p2 g2 c3 p3 g3
c0
4-bit Carry-Look-Ahead Generator
g[0,3]
p[0,3]
2-bit Carry Look Ahead Gen.
c4
4-bit Carry-Look-Ahead Generator
g[4,7]
p[4,7]
p4 g4 c5 p5 g5 c6 p6 g6 c7 p7 g7
a4 b4 s4 a5 b5 s5 a6 b6 s6 a7 b7 s7
c8

(Figure 3.1)

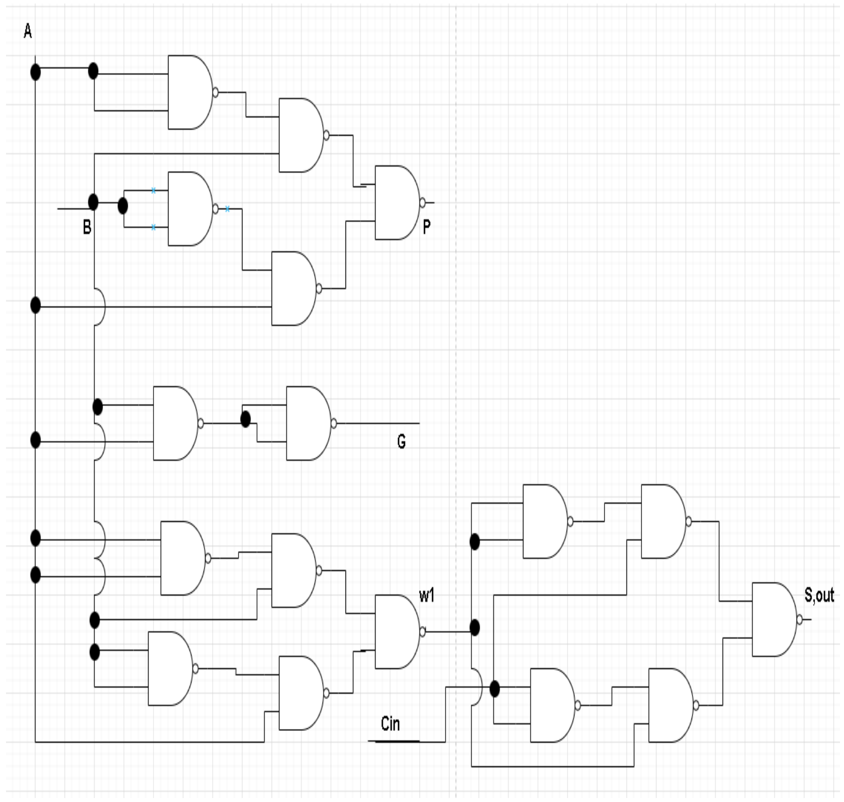
Following the given picture in the assignment, we will first create 8 Prop-and-Gen blocks where each 4-bit CLA will be given 4 each. Given their respective input a_i, b_i and c_i to output their s_i (Figure 3.2). The formula we used for Prop-and-Gen is

$$P = a \oplus b$$

$$G = a + b$$

$$S = (a \oplus b) \oplus cin$$

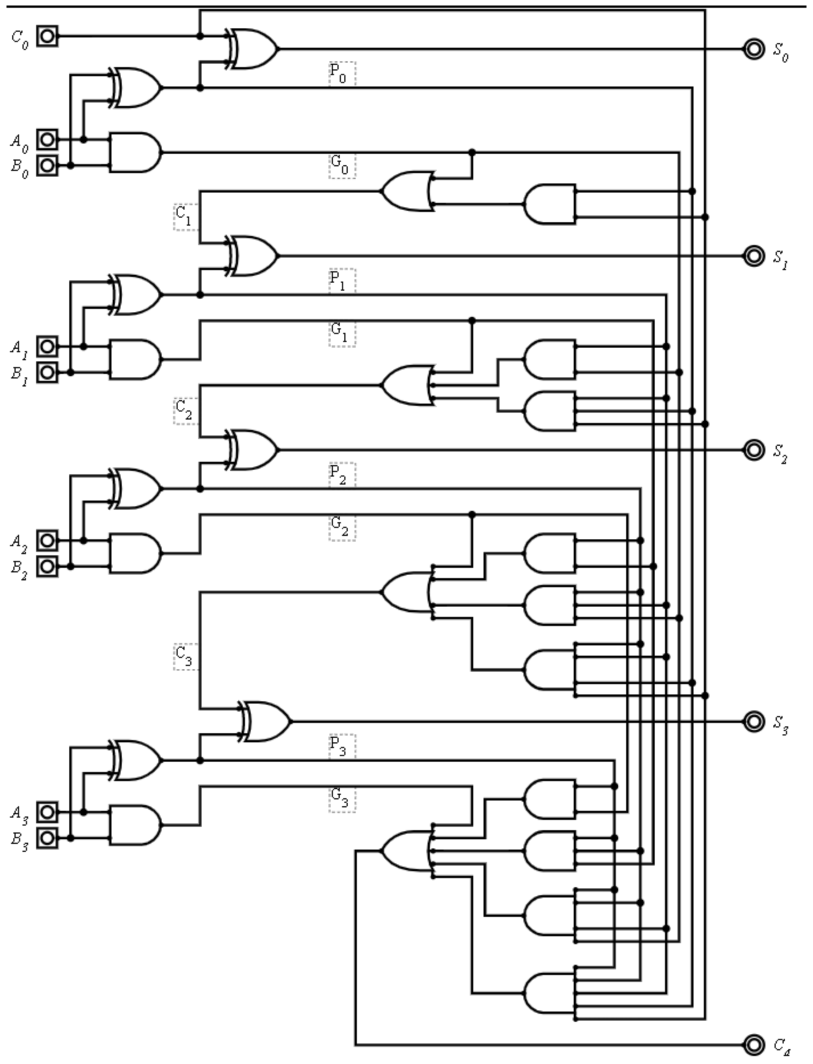
for all eight Prop-and-gen blocks.



(Figure 3.2)

Each P and G will connect to their respective CLA block.

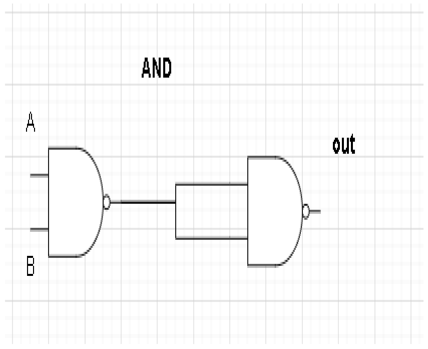
For the next part, we designed two 4-bit CLA where each one will be responsible for the first and last four Prop-and-Gen blocks. For simplicity reason we drew our 4-bit CLA as follows.

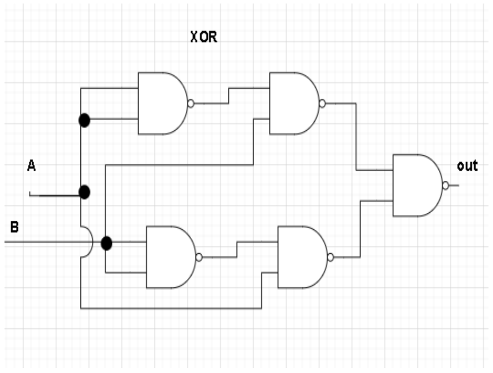


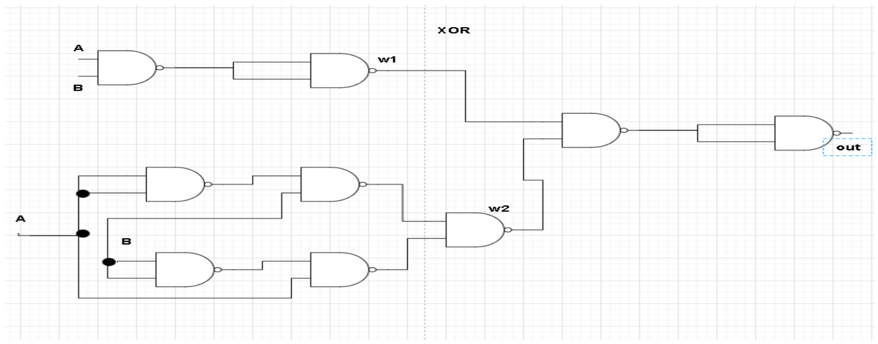
Thus, we followed the following formula to calculate the C.

$$C_i = G_i + P_i C_{i-1}$$

However, for our gates, we used nand gates. In addition, we add hierarchy from 2-bit until 5-bit for or, and gates because to calculate for easily. For example, for the 3-bit AND gate, we implement by using the previous module AND which is 2-bit. After it calculates the first two bit a and b, we wire it was wire1 and continue by calling it one more time and calculate the AND of wire1 and c bit (the third bit). Thus, this functionality goes until 5-bits of AND gates implemented by NAND gates. In addition, we also use the same logic for OR gates until 5-bit.







For the last part for our module, we have the 2-bit CLA. It will take all of the P and G values from the two 4-bit CLA. The 2-bit CLA will calculate the combined final value, c0,c4 and c8.

In short summary, CLA is beneficial for it’s speed where it predict the carry of the circuit. Thus, for a 8-bit CLA, we will need two 4-bit CLA, 2-bit CLA and 8 Prop-and-Gen (for each bit).

b. Circuit

```
`timescale 1ns/1ps

module Carry_Look_Ahead_Adder_8bit(a, b, c0, s, c8);
   input [7:0] a, b;
   input c0;
   output [7:0] s;
   output c8;

   wire c1, c2, c3, c4, c5, c6, c7;
   wire P0, P1, P2, P3, P4, P5, P6, P7;
   wire G0, G1, G2, G3, G4, G5, G6, G7;
   wire PG0123, PG4567;
   wire GG0123, GG4567;


   Prop_and_Gen CLA0(P0, G0, s[0], a[0], b[0], c0);
   Prop_and_Gen CLA1(P1, G1, s[1], a[1], b[1], c1);
   Prop_and_Gen CLA2(P2, G2, s[2], a[2], b[2], c2);
   Prop_and_Gen CLA3(P3, G3, s[3], a[3], b[3], c3);
   Prop_and_Gen CLA4(P4, G4, s[4], a[4], b[4], c4);
   Prop_and_Gen CLA5(P5, G5, s[5], a[5], b[5], c5);
   Prop_and_Gen CLA6(P6, G6, s[6], a[6], b[6], c6);
   Prop_and_Gen CLA7(P7, G7, s[7], a[7], b[7], c7);

   CLA_4bit CLA_0123(P0, G0,P1, G1,P2, G2,P3, G3,c0,c1, c2, c3,PG0123,
GG0123);
   CLA_4bit CLA_4567(P4, G4,P5, G5,P6, G6,P7, G7,c4,c5, c6, c7,PG4567,
GG4567);

   CLA_2bit CLA_ALL(PG0123, GG0123,PG4567, GG4567,c0, c4, c8);

endmodule

module Prop_and_Gen(P, G, out, a, b, cin);
   input a, b, cin;
   output P, G, out;

   wire w1;

   NAND_XOR Prop(P, a, b);

   NAND_AND Gen(G, a, b);

   NAND_XOR PG1(w1, a, b);
   NAND_XOR PG2(out, w1, cin);

endmodule

module CLA_4bit(P0,G0,P1,G1,P2,G2,P3,G3,cin,CA,CB,CC,PG,GG);
   input P0,G0,P1,G1,P2,G2,P3,G3;
   input cin;
   output CA, CB, CC;
   output PG, GG;

   wire P0cin;
   wire P1G0, P1P0cin;
   wire P2G1, P2P1G0, P2P1P0cin;
   wire P3G2, P3P2G1, P3P2P1G0;
```

```
    //CA
    NAND_AND CA1(P0cin, P0, cin);

    NAND_OR CA2(CA, P0cin, G0);

    //CB
    NAND_AND CB1(P1G0, P1, G0);

    NAND_AND_3bit CB2(P1P0cin, P1, P0, cin);

    NAND_OR_3bit CB3(CB, G1, P1G0, P1P0cin);

    //CC
    NAND_AND CC1(P2G1, P2, G1);

    NAND_AND_3bit CC2(P2P1G0, P2, P1, G0);

    NAND_AND_4bit CC3(P2P1P0cin, P2, P1, P0, cin);

    NAND_OR_4bit CC4(CC, G2, P2G1, P2P1G0, P2P1P0cin);

    //Prop and Gen Carry
    NAND_AND PGC1(P3G2, P3, G2);

    NAND_AND_3bit PGC2(P3P2G1, P3, P2, G1);

    NAND_AND_4bit PGC3(P3P2P1G0, P3, P2, P1, G0);

    //PG
    NAND_AND_4bit PG1(PG, P0, P1, P2, P3);

    //GG
    NAND_OR_4bit GG1(GG, G3, P3G2, P3P2G1, P3P2P1G0);

endmodule

module CLA_2bit(P0,G0,P1,G1,cin,carry,cout);
    input P0, G0,P1, G1,cin;
    output carry, cout;

    wire P0_cin, P1_G0, P1P0_cin;

    NAND_AND CLA1(P0_cin, P0, cin);

    NAND_OR CL2(carry, P0_cin, G0);
```

```
   NAND_AND CLA3(P1_G0, P1, G0);

   NAND_AND_3bit CLA4(P1P0_cin, P1, P0, cin);

   NAND_OR_3bit CLA5(cout, G1, P1_G0, P1P0_cin);

endmodule

module NAND_AND(out,a,b);
input a,b;
output out;
wire w1;
nand N1(w,a,b);
nand N2(out,w,w);
endmodule

module NAND_AND_3bit(out,a,b,c);
input a,b,c;
output out;
wire nand1;
NAND_AND A1(nand1,a,b);
NAND_AND A2(out,nand1,c);
endmodule

module NAND_AND_4bit(out,a,b,c,d);
input a,b,c,d;
output out;
wire nand1;
NAND_AND_3bit A1(nand1,a,b,c);
NAND_AND A2(out,nand1,d);
endmodule

module NAND_AND_5bit(out,a,b,c,d,e);
input a,b,c,d,e;
output out;
wire nand1;
NAND_AND_4bit A1(nand1,a,b,c,d);
NAND_AND A2(out,nand1,e);
endmodule

module NAND_OR(out,a,b);
input a,b;
output out;
wire w1,w2;
nand O1(w1,a,a);
nand O2(w2,b,b);
nand O3(out,w1,w2);
endmodule

module NAND_OR_3bit(out,a,b,c);
input a,b,c;
output out;
wire w1;
NAND_OR O1(w1,a,b);
NAND_OR O2(out,w1,c);
endmodule

module NAND_OR_4bit(out,a,b,c,d);
input a,b,c,d;
output out;
wire w1;
NAND_OR_3bit O1(w1,a,b,c);
NAND_OR O2(out,w1,d);
endmodule

module NAND_OR_5bit(out,a,b,c,d,e);
input a,b,c,d,e;
output out;
wire w1;
NAND_OR_4bit O1(w1,a,b,c,d);
NAND_OR O2(out,w1,e);
endmodule

module NAND_XOR(out,a,b);
input a,b;
output out;
wire w1,w2;
nand X1(w1,a,b);
NAND_OR X2(w2,a,b);
NAND_AND X3(out,w1,w2);
endmodule
```

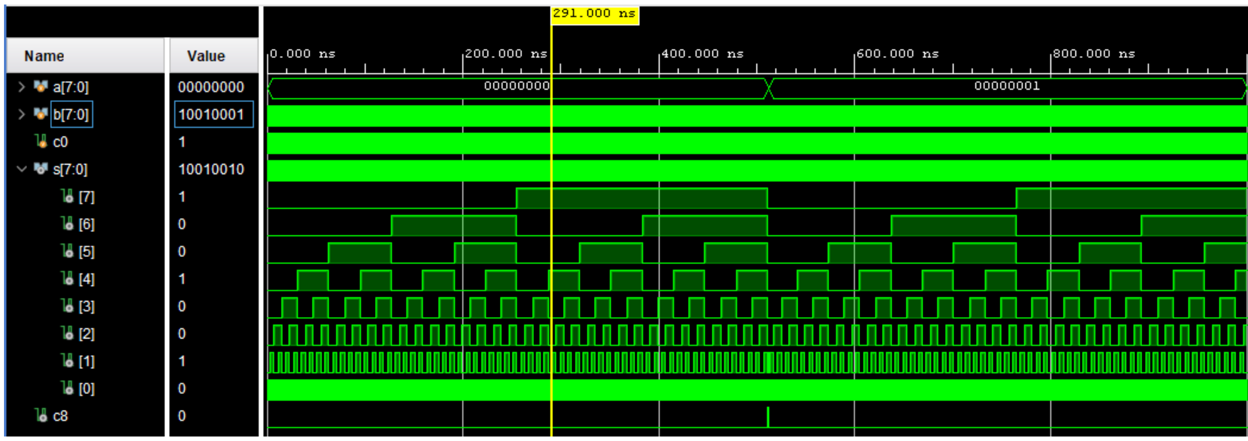
c. Testbench

```
`timescale 1ns / 1ps

module CLA_t;
   reg [7:0] a, b;
   reg c0;
   wire [7:0] s;
```

```
  wire c8;

  Carry_Look_Ahead_Adder_8bit CLA(
     .a(a),
     .b(b),
     .c0(c0),
     .s(s),
     .c8(c8)
  );

  initial begin
     c0 = 1'b0;
     a = 8'd0;
     b = 8'd0;
     repeat (2 ** 8) begin
        repeat (2 ** 8) begin
           repeat (2) begin
              #1 c0 = c0 + 1'b1;
           end
           b = b + 1'b1;
        end
        a = a + 1'b1;
     end
     #1 $finish;
  end

endmodule
```

d. Waveform

Using the testbench, we can see that our code works functionally to our design from above for every bit input. As an example, we can see the addition of a (00000000) and b(10010001) will be s(10010010) and there is no carry c8. From the wave highs and lows we can see that it works accordingly.

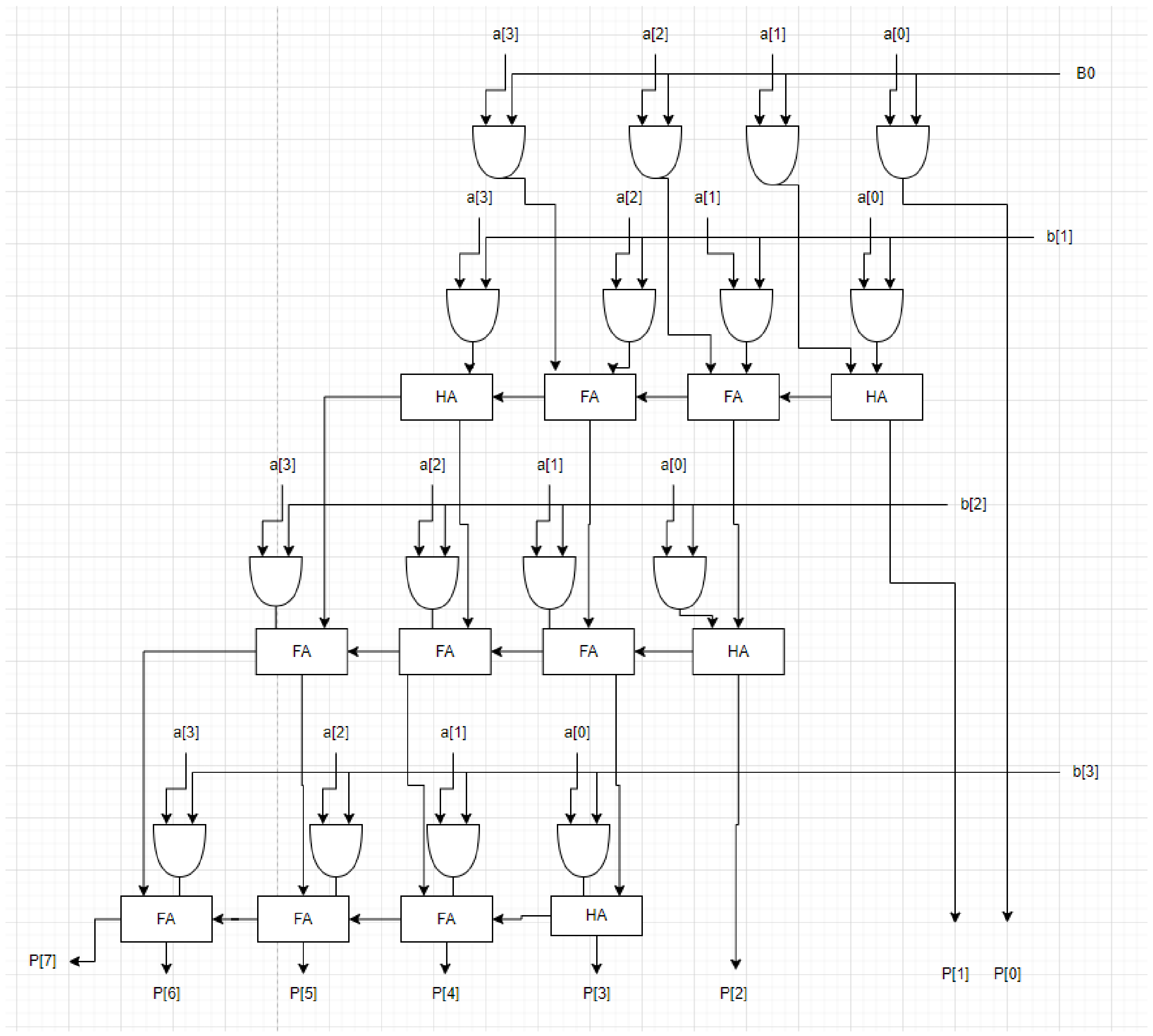


4. 4-bit Multiplier

a. Design

A 4-bit multiplier just like its name is to multiply 4 numbers with another 4 numbers. In the circuit, we use 2 NAND Gates for the replacement of the AND Gates. As the digit for A&B becomes the input for the Adders.

4 bit multiplier consists of 4 Half-Adders and 9 Full-Adders, and in our circuit it is divided into 3 levels. Where the 1st half adder on the right side is the sign of a level. Where the last adder of each level outputs the carry to the last adder of the next level. And the adders in the same level outputs the carry to the next level. And each adder get the input a,b,cin according to the circuit.



b. Circuit

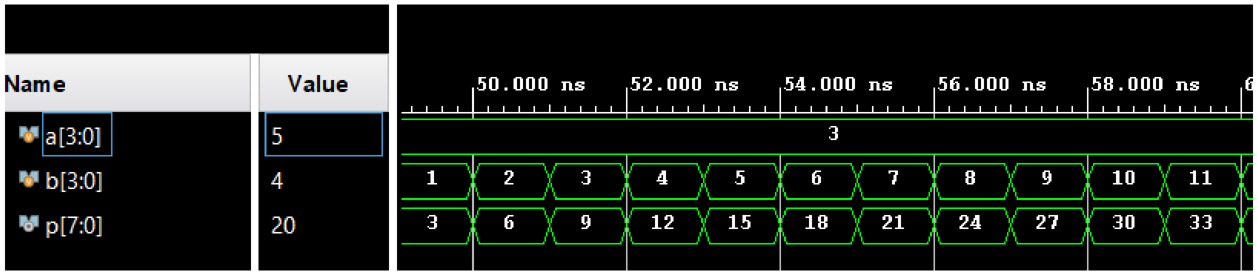
```
module Multiplier_4bit(a, b, p);
   input [4-1:0] a, b;
   output [8-1:0] p;
   // Check Circuit for Logic

//module FA(sout,cout,a,b,cin);
   wire and1,and2,and3,and4,and5,and6,and7,and8,and9,and10,and11,and12;



   AND ANDG1(a[0], b[1], and1);
   AND ANDG2(a[1], b[0], and2);
   AND ANDG3(a[0], b[2], and3);
   AND ANDG4(a[1], b[1], and4);
   AND ANDG5(a[1], b[2], and5);
   AND ANDG6(a[0], b[3], and6);
   AND ANDG7(a[1], b[3], and7);
   AND ANDG8(a[2], b[0], and8);
   AND ANDG9(a[2], b[1], and9);
   AND ANDG10(a[2], b[2], and10);
   AND ANDG11(a[2], b[3], and11);
   AND ANDG12(a[2], b[3], and12);
   AND ANDG13(a[3], b[0], and13);
   AND ANDG14(a[3], b[1], and14);
   AND ANDG15(a[3], b[2], and15);
   AND ANDG16(a[3], b[3], and16);

   wire x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16;

   AND ANDG0(a[0], b[0], p[0]);
   //Level 1
   Half_Adder HA1(and2, and1, x1, p[1]);
   Full_Adder FA1(and4,and3,x1,x3,x2);
   Full_Adder FA2(and5,and6,x3,x5,x4);
   Half_Adder HA2(and7,x5,x7,x6);

   Half_Adder HA3(x2,and8,x15,p[2]);
   Full_Adder FA5(x4,and9,x15,x16,x14);
   Full_Adder FA4(x6,and10,x16,x17,x13);
   Full_Adder FA3(x7,and12,x17, x8,x9);

   Half_Adder HA4(x14, and13, x12, p[3]);
   Full_Adder FA8(x13, and14, x12, x11, p[4]);
   Full_Adder FA7(x9,and15,x11,x10,p[5]);
   Full_Adder FA6(x8,and16,x10,p[7],p[6]);
endmodule


module Half_Adder(a, b, cout, sum);
   input a, b;
   output cout, sum;

   wire w1,w2,w3;

   nand nand1(w1,a,b);
   nand nand2(w2,a,w1);
   nand nand3(w3,b,w1);
   nand SumOut(sum, w2, w3);
   nand CarryOut(cout, w1);

endmodule

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

   wire notc,notcout, w1;

   nand nand1(notcin, cin);
   nand nand2(notcout,cout);

   Majority M1(a, b, cin, cout);
   Majority M2(notcin, b, a, w1);
   Majority M3(notcout, cin, w1, sum);

endmodule

module Majority(a, b, c, out);
   input a, b, c;
   output out;

   wire w1,w2,w3,w4;
   AND and1(a,b, w1);
   AND and2(a, c , w2);
   AND and3(b, c ,w3);

   OR or1(w1, w2, w4);
   OR or2(w4,w3, out);


endmodule

module AND (a, b , out);
   input a,b;
   output out;

   wire w1;

   nand nand1(w1, a, b);
   nand nand2(out, w1);

endmodule
```

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

    wire w1,w2;

    nand nand1(w1, a);
    nand nand2(w2, b);
    nand nand3(out, w1, w2);

endmodule
```

c. Testbench

```
`timescale 1ns / 1ps

module Multiplier_4bit_t;
    reg [3:0] a, b;
    wire[7:0] p;

    Multiplier_4bit mult(
        .a(a),
        .b(b),
        .p(p)
    );

    initial begin
        a = 4'b0000;
        b = 4'b0000;
        repeat (2 ** 4) begin
            repeat (2 ** 4) begin
                #1 b = b + 1'b1;
            end
            a = a + 1'b1;
        end
        #1 $finish;
    end
endmodule
```

d. Waveform



5. Exhaustive Testbench Design

a. Design

In our design, to test a 4-bit Ripple Carry Adder, where we make a check module , and inside the testbench we put the correct Ripple Carry Adder, as an anchor so we can check the false inputs if given, if there is no false input we don't raise the error flag. In the check module we use XNOR and AND Gates, XNOR is to see whether the sum and cout is the same as sum_correct and cout_correct. As when the outputs for the correct one and the false one get AND'd it will raise the error flag as the waveform below

XNOR		**=**	**AND**		**=**
0	0	1	0	0	0
1	0	0	1	0	0
0	1	0	0	1	0
1	1	1	1	1	1

b. Circuit

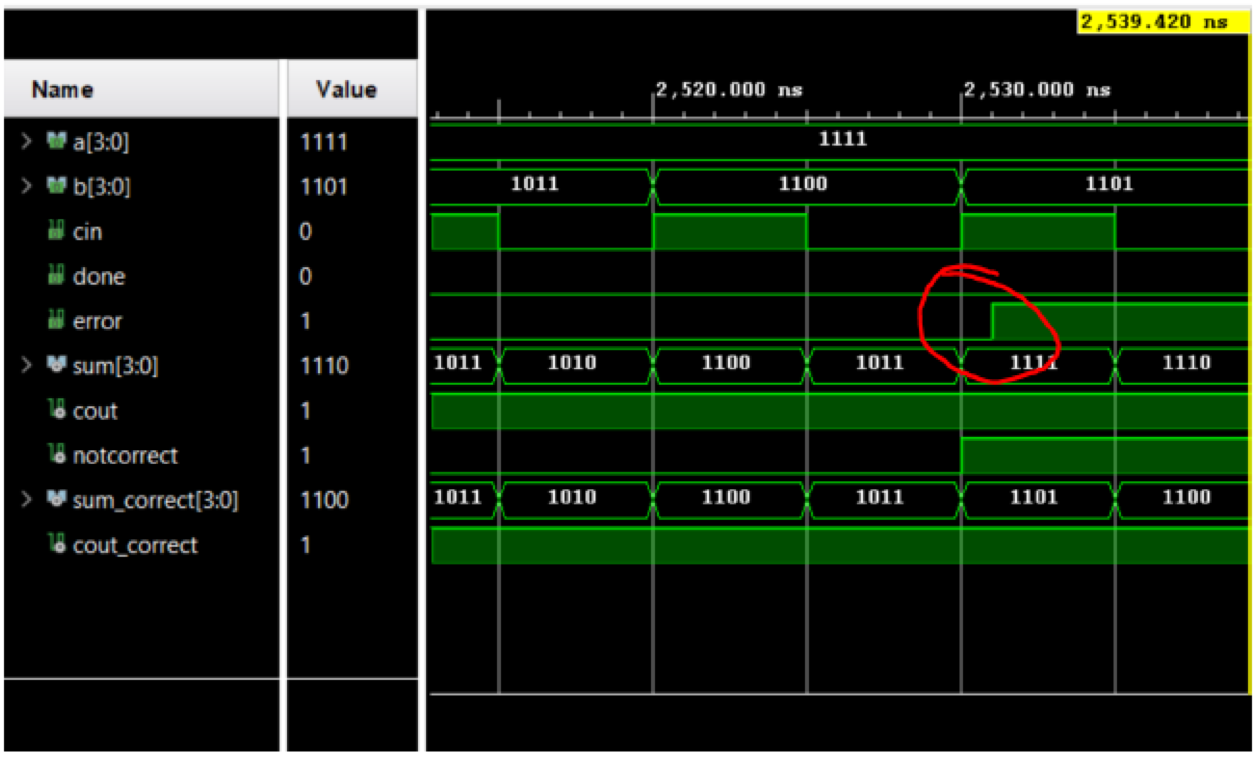
```
`timescale 1ns/1ps

module Exhausted_Testing(a, b, cin, error, done);
output [4-1:0] a, b;
output cin;
output error;
output done;

// input signal to the test instance.
reg [4-1:0] a = 4'b0000;
reg [4-1:0] b = 4'b0000;
reg cin = 1'b0;

// initial value for the done and error indicator: not done, no error
reg done = 1'b0;
reg error = 1'b0;

// output from the test instance.
wire [4-1:0] sum;
wire cout;

// Inidicator Correct
wire notcorrect;
wire[4-1:0]sum_correct;
wire cout_correct;

Ripple_Carry_Adder1 R1(
    .a (a),
    .b (b),
    .cin (cin),
    .cout_correct (cout_correct),
    .sum_correct (sum_correct)
    );
// instantiate the test instance. DONT CHANGE
Ripple_Carry_Adder rca(
    .a (a),
    .b (b),
    .cin (cin),
    .cout (cout),
    .sum (sum)
);

CHECK chk(
    .notcorrect(notcorrect),
    .cout_correct(cout_correct),
    .sum_correct(sum_correct),
    .sum(sum),
    .cout(cout)
);

initial begin
        repeat (2 ** 4) begin
            repeat (2 ** 4) begin
                repeat(2 ** 1) begin
                    cin = cin + 1'b1;
                    #1 error = notcorrect;
                    #4;
            end
            b = b + 1'b1;
         end
         a = a + 1'b1;
      end
      done = 1'b1;
      #5 done = 1'b0;
end

endmodule

module CHECK(notcorrect,cout_correct,sum_correct, sum, cout);
   input[3:0] sum , sum_correct;
   input cout, cout_correct;
   output notcorrect;

   wire w1,w2,w3,w4,w5;
   wire a1,a2,a3,a4;
   wire iscorrect;

   XNOR1 xnor1(sum_correct[0],sum[0],w1);
   XNOR1 xnor2(sum_correct[1],sum[1],w2);
   XNOR1 xnor3(sum_correct[2],sum[2],w3);
   XNOR1 xnor4(sum_correct[3],sum[3],w4);
   XNOR1 xnor5(cout,cout_correct,w5);

   AND1 and1(w1, w2, a1);
   AND1 and2(w3, w4, a2);
   AND1 and3(a1,a2, a3);
   AND1 and4(a3, w5, iscorrect);
   NOT1 not1(iscorrect, notcorrect);
endmodule

module Ripple_Carry_Adder1(a, b, cin, cout_correct, sum_correct);
   input [3:0] a, b;
   input cin;
   output cout_correct;
   output [3:0] sum_correct;

   wire c1,c2,c3,c4,c5,c6,c7;

   Full_Adder FADD0(a[0], b[0], cin, c1, sum_correct[0]);
   Full_Adder FADD1(a[1], b[1], c1, c2, sum_correct[1]);
   Full_Adder FADD2(a[2], b[2], c2, c3, sum_correct[2]);
   Full_Adder FADD3(a[3], b[3], c3, cout_correct, sum_correct[3]);
```

```
endmodule

//XNOR
module XNOR1(a, b, out);

input a, b;
output out;

wire w1, w2, w3, w4;

nand nand1(w1, a, b);
nand nand2(w2, a, w1);
nand nand3(w3, w1, b);
nand nand4(w4, w2, w3);
nand nand5(out, w4, w4);

endmodule

module Full_Adder1 (a, b, cin, cout, sum);
   input a, b, cin;
   output cout, sum;

   wire notc,notcout, w1;

   nand nand1(notcin, cin);
   nand nand2(notcout,cout);

   Majority1 M1(a, b, cin, cout);
   Majority1 M2(notcin, b, a, w1);
   Majority1 M3(notcout, cin, w1, sum);

endmodule

module Majority1(a, b, c, out);
   input a, b, c;
   output out;

   wire w1,w2,w3,w4;
   AND1 and1(a,b, w1);
   AND1 and2(a, c , w2);
   AND1 and3(b, c ,w3);

   OR1 or1(w1, w2, w4);
   OR1 or2(w4,w3, out);

endmodule
```

```
module AND1 (a, b , out);
   input a,b;
   output out;

   wire w1;

   nand nand1(w1, a, b);
   nand nand2(out, w1);

endmodule

module OR1 (a, b, out);
   input a,b;
   output out;

   wire w1,w2;

   nand nand1(w1, a);
   nand nand2(w2, b);
   nand nand3(out, w1, w2);

endmodule
module NOT1(a ,out);
   input a;
   output out;

   nand nand1 (out, a, a);

endmodule
```

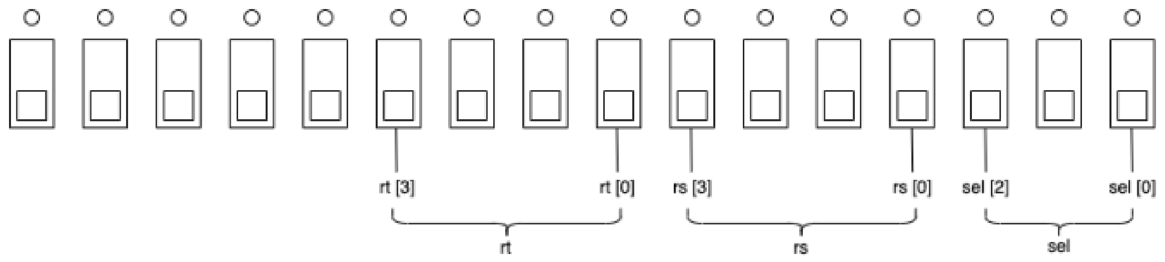
c. Waveform

Waveform where the sum is not the same as the sum_correct, it will raise the error flag 1ns after the notcorrect flag is raised

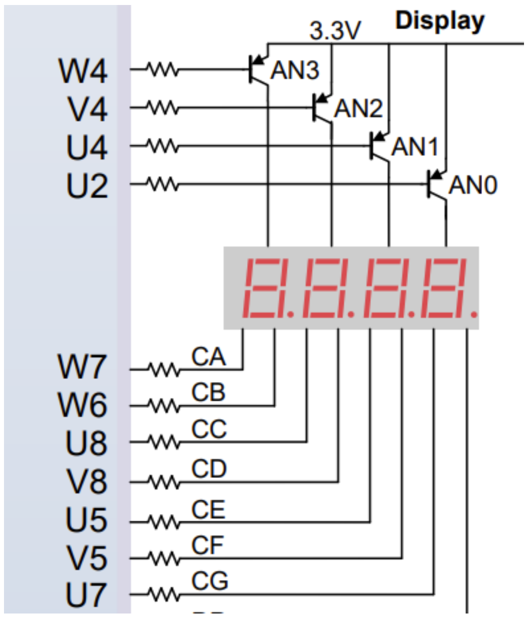

2,539.420 ns
2,520.000 ns
2,530.000 ns
Name
Value
a[3:0]
b[3:0]
cin
done
error
sum[3:0]
cout
notcorrect
sum_correct[3:0]
cout_correct
1111
1101
0
0
1
1110
1
1
1100
1
1111
1011
1100
1101
1011
1010
1100
1011
1111
1110
1011
1010
1100
1011
1101
1100

6. FPGA Implementation

Assigning the following I/O Ports for rs, rt and sel with the switches on the Basys-3 board.



And the following I/O ports for the digit display, with a reference from the internet



a. Code

We took reference from a website to convert, 4bit to a 7bit, where the table is as follows. And with this table we can conclude with the logic gate in the following table. For the cathode we just check for every bit and for every case when there is “1” we realize it from the Input using Logic Gates like the next table.

Number	Input	Cathode [6:0]
0	0000	0000001
1	0001	1001111
2	0010	0010010
3	0011	0000110
4	0100	1001100
5	0101	0100100
6	0110	0100000
7	0111	0001111
8	1000	0000000
9	1001	0000100
a	1010	0000010
b	1011	1100000
c	1100	0110001
d	1101	1000010
e	1110	0110000
f	1111	0111000

Where W = rd[3] X = rd[2] Y = rd[1] Z = rd[0].

Table of Logic Gate for the 4bit to 7bit

Logic Gate from realizing 4 bit to cathode of “1” .

out[6]	W’X’Y’Z + W’XY’Z’ + WX’YZ + WXY’Z
out[5]	W’XY’Z + W’XYZ’ + WX’YZ + WXY’Z’ + WXYZ’ + WXYZ
out[4]	W’X’YZ’ + WXY’Z’ + WXYZ’ + WXYZ
out[3]	W’X’Y’Z + W’XY’Z’ + W’XYZ + WXYZ
out[2]	W’X’Y’Z + W’X’YZ + W’XY’Z’ +W’XY’Z + W’XYZ + WX’Y’Z
out[1]	W’X’Y’Z + W’X’YZ” + W’X’YZ + W’XYZ + WX’YZ’ + WXY’Z
out[0]	W’X’Y’Z’ + W’X’Y’Z + W’XYZ + WXY’Z’

And when implemented into AND and OR Gates using the Universal gate it becomes like so, and we to assign out1 to turn off the LED Lights, because we only need to use the 4th LED Light. Since the out varies in their inputs, we make multiple modules of 4bit AND , 4bit OR, 6bit OR in order to make our work more simple and easy.

```
`timescale 1ns/1ps

module Decode_And_Execute(rs, rt, sel, out1, out2);
   input [3:0] rs, rt;
   input [2:0] sel;
   output [3:0] out1;
   output [6:0] out2;
   wire [3:0] rd;

   wire [3:0] case0,case1,case2,case3,case4,case5,case6,case7;
   wire [3:0] w1;

    ASSIGN_GATE ass1(out1[3], 1'b1);
    ASSIGN_GATE ass2(out1[2], 1'b1);
    ASSIGN_GATE ass3(out1[1], 1'b1);
    ASSIGN_GATE ass4(out1[0], 1'b0);

   //SUB
   COMPLEMENT C1(rt, w1);
   ADD Sub(rs, w1, case0);

   //RCA_4bit
   ADD A1(rs, rt, case1);

   //BITWISE OR
   BIT_OR OR1(rs, rt, case2);

   //BITWISE AND
   BIT_AND AND1(rs, rt, case3);


   //RIGHT SHIFT
   shift R0(rt[1], case4[0]);
   shift R1(rt[2], case4[1]);
   shift R2(rt[3], case4[2]);
   shift R3(rt[3], case4[3]);

   //LEFT SHIFT
   shift L0(rs[3], case5[0]);
   shift L1(rs[0], case5[1]);
   shift L2(rs[1], case5[2]);
   shift L3(rs[2], case5[3]);

   //EQ and GT
   COMPARATOR Comp(rs, rt, case7, case6);

   //MUX
   MUX_8X1_4bit M1(case0,case1,case2,case3,case4,case5,case6,case7,sel, rd);

   Conv4to7bit(rd,out2);
endmodule


module Conv4to7bit(rd,out);
   input [3:0] rd;
   output [7-1:0] out;

   wire notrd0,notrd1,notrd2,notrd3;
   NOT_GATE notd0(notrd0,rd[0]);
   NOT_GATE notd1(notrd1,rd[1]);
   NOT_GATE notd2(notrd2,rd[2]);
   NOT_GATE notd3(notrd3,rd[3]);

   wire
and0000,and0001,and0010,and0011,and0100,and0101,and0110,and0111,and1000,and10
01,and1010,and1011,and1100,and1101,and1110,and1111;
   AND_4bit A4bit1(notrd3,notrd2,notrd1,notrd0,and0000);
   AND_4bit A4bit2(notrd3,notrd2,notrd1,rd[0],and0001);
   AND_4bit A4bit3(notrd3,notrd2,rd[1],notrd0,and0010);
   AND_4bit A4bit4(notrd3,notrd2,rd[1],rd[0],and0011);
   AND_4bit A4bit5(notrd3,rd[2],notrd1,notrd0,and0100);
   AND_4bit A4bit6(notrd3,rd[2],notrd1,rd[0],and0101);
   AND_4bit A4bit7(notrd3,rd[2],rd[1],notrd0,and0110);
   AND_4bit A4bit8(notrd3,rd[2],rd[1],rd[0],and0111);
   AND_4bit A4bit9(rd[3],notrd2,notrd1,notrd0,and1000);
   AND_4bit A4bit10(rd[3],notrd2,notrd1,rd[0],and1001);
   AND_4bit A4bit11(rd[3],notrd2,rd[1],notrd0,and1010);
   AND_4bit A4bit12(rd[3],notrd2,rd[1],rd[0],and1011);
   AND_4bit A4bit13(rd[3],rd[2],notrd1,notrd0,and1100);
   AND_4bit A4bit14(rd[3],rd[2],notrd1,rd[0],and1101);
   AND_4bit A4bit15(rd[3],rd[2],rd[1],notrd0,and1110);
   AND_4bit A4bit16(rd[3],rd[2],rd[1],rd[0],and1111);

   OR_4bit O4bit1(and0001,and0100,and1011,and1101,out[6]);
   OR_6bit OR6bit1(and0101,and0110,and1011,and1100,and1110,and1111,out[5]);
   OR_4bit O4bit2(and0010,and1100,and1110,and1111,out[4]);
   OR_4bit O4bit3(and0001,and0100,and0111,and1111,out[3]);
   OR_6bit O6bit2(and0001,and0011,and0100,and0101,and0111,and1001,out[2]);
   OR_6bit O6bit3(and0001,and0010,and0011,and0111,and1010,and1101,out[1]);
   OR_4bit O4bit4(and0000,and0001,and0111,and1100,out[0]);
endmodule

module AND_4bit(a,b,c,d,out);
   input a,b,c,d;
   output out;
   wire w1;

   nand NAND4bit1 (w1,a,b,c,d);
   nand NAND4bit2(out,w1);
endmodule

module OR_4bit(a,b,c,d,out);
   input a,b,c,d;
   output out;
   wire w1,w2;
   OR_GATE OR_4bit1(w1,a,b);
   OR_GATE OR_4bit2(w2,c,d);

   OR_GATE OR_4bit3(out,w1,w2);

endmodule

module OR_5bit(a,b,c,d,e,out);
   input a,b,c,d,e;
   output out;

   wire w1,w2,w3;
   OR_GATE OR_5bit1(w1,a,b);
   OR_GATE OR_5bit2(w2,c,d);
   OR_GATE OR_5bit3(w3,w1,w2);

   OR_GATE OR_5bit4(out,w3,e);

endmodule

module OR_6bit(a,b,c,d,e,f,out);
   input a,b,c,d,e,f;
   output out;

   wire w1,w2,w3,w4;
   OR_GATE OR_6bit1(w1,a,b);
   OR_GATE OR_6bit2(w2,c,d);
   OR_GATE OR_6bit3(w3,w1,w2);
   OR_GATE OR_6bit4(w4,e,f);

   OR_GATE OR_6bit5(out,w3,w4);
endmodule

module shift (in, out);
   input in;
   output out;

   wire w;

   NOT_GATE SH0(w, in);
   NOT_GATE SH1(out, w);

endmodule

module MUX_2x1_4bit(a, b, sel, out);
```

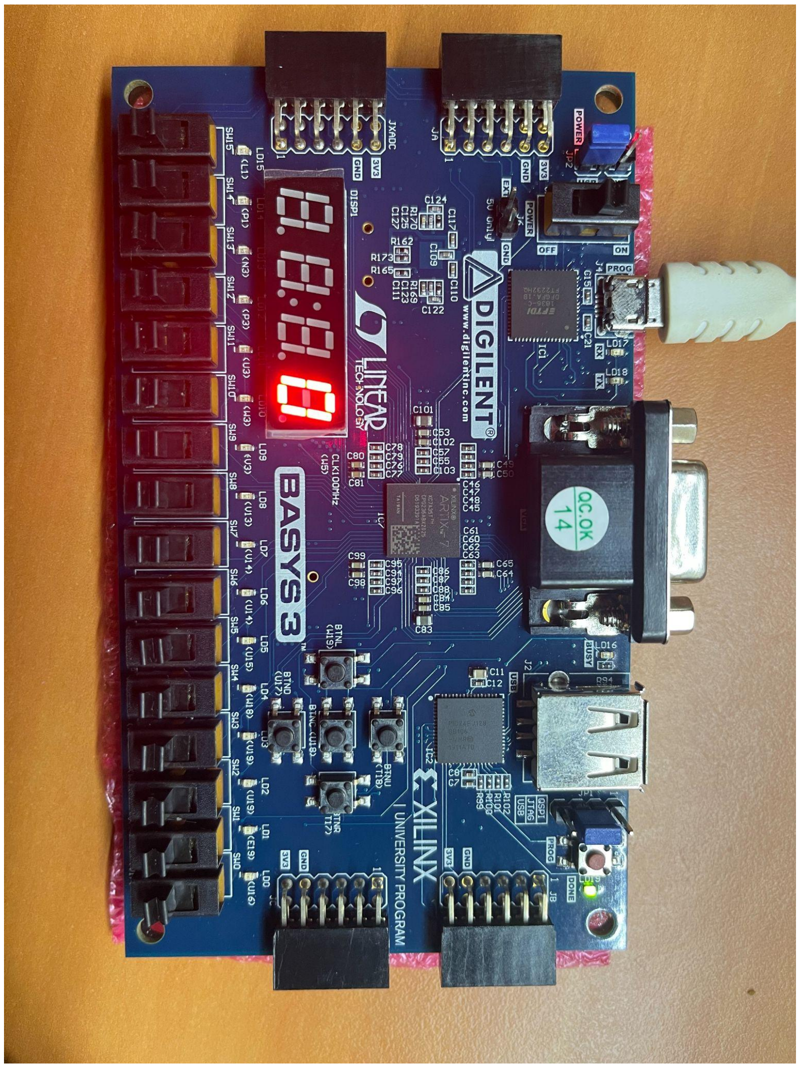
```
   input [3:0] a, b;
   input sel;
   output [3:0] out;

   wire not_sel;

   wire w0, w1, w2, w3,w4, w5, w6, w7;

   NOT_GATE N(not_sel, sel);

   AND_GATE A0(w0, a[0], not_sel);
   AND_GATE A1(w1, a[1], not_sel);
   AND_GATE A2(w2, a[2], not_sel);
   AND_GATE A3(w3, a[3], not_sel);

   AND_GATE B0(w4, b[0], sel);
   AND_GATE B1(w5, b[1], sel);
   AND_GATE B2(w6, b[2], sel);
   AND_GATE B3(w7, b[3], sel);

   OR_GATE O0(out[0], w0, w4);
   OR_GATE O1(out[1], w1, w5);
   OR_GATE O2(out[2], w2, w6);
   OR_GATE O3(out[3], w3, w7);

endmodule

module MUX_8X1_4bit (case0,case1,case2,case3,case4,case5,case6,case7,sel,out);
   input [3:0] case0,case1,case2,case3,case4,case5,case6,case7;
   input [2:0] sel;
   output [3:0] out;

   wire [3:0] w1,w2,w3,w4,w12,w34;

   MUX_2x1_4bit u1(case0, case1, sel[0], w1);
   MUX_2x1_4bit u2(case2, case3, sel[0], w2);
   MUX_2x1_4bit u3(case4, case5, sel[0], w3);
   MUX_2x1_4bit u4(case6, case7, sel[0], w4);

   MUX_2x1_4bit u5(w1, w2, sel[1], w12);
   MUX_2x1_4bit u6(w3, w4, sel[1], w34);

   MUX_2x1_4bit u7(w12, w34, sel[2], out);

endmodule
```

```
module COMPARATOR(a, b, eq, gt);
   input [3:0] a, b;
   output [3:0] eq, gt;

   wire E_confirm, G_confirm;

   //eq
   wire
   XNOR_0, XNOR_1, XNOR_2, XNOR3,
   XNOR_23, XNOR_012;

 wire
   BB_0, BB_1, BB_2, BB_3,
   GT_AB_0, GT_AB_1,
   GT_AB_2, GT_AB_3,
   GT_AB_01, GT_AB_012, GT_AB_0123,
   ACC01, ACC012;



   XNOR_GATE X0(XNOR_0, a[0], b[0]);
   XNOR_GATE X1(XNOR_1, a[1], b[1]);
   XNOR_GATE X2(XNOR_2, a[2], b[2]);
   XNOR_GATE X3(XNOR_3, a[3], b[3]);


   AND_GATE EQ01(XNOR_23, XNOR_2, XNOR_3);
   AND_GATE EQ012(XNOR_123, XNOR_1, XNOR_23);
   AND_GATE EQ0123(E_confirm, XNOR_0, XNOR_123);

   //gt


   NOT_GATE N0(BB_0, b[0]);
   NOT_GATE N1(BB_1, b[1]);
   NOT_GATE N2(BB_2, b[2]);
   NOT_GATE N3(BB_3, b[3]);


   AND_GATE A0(GT_AB_0, a[0], BB_0);
   AND_GATE A1(GT_AB_1, a[1], BB_1);
   AND_GATE A2(GT_AB_2, a[2], BB_2);
   AND_GATE A3(GT_AB_3, a[3], BB_3);

   AND_GATE GT1(GT_AB_01, GT_AB_2, XNOR_3);
   AND_GATE GT2(GT_AB_012, GT_AB_1, XNOR_23);
```

```
    AND_GATE GT3(GT_AB_0123, GT_AB_0, XNOR_123);


    OR_GATE O0(ACC01, GT_AB_3, GT_AB_01);
    OR_GATE O1(ACC012, ACC01, GT_AB_012);
    OR_GATE O2(G_confirm, ACC012, GT_AB_0123);

    //Value
    shift E_Value0(E_confirm, eq[0]);
    shift E_Value1(1'b1, eq[1]);
    shift E_Value2(1'b1, eq[2]);
    shift E_Value3(1'b1, eq[3]);

    shift G_Value0(G_confirm, gt[0]);
    shift G_Value1(1'b1, gt[1]);
    shift G_Value2(1'b0, gt[2]);
    shift G_Value3(1'b1, gt[3]);

endmodule

module BIT_AND (a, b, out);
    input [3:0] a, b;
    output [3:0] out;

    AND_GATE A0(out[0], a[0], b[0]);
    AND_GATE A1(out[1], a[1], b[1]);
    AND_GATE A2(out[2], a[2], b[2]);
    AND_GATE A3(out[3], a[3], b[3]);

endmodule

module BIT_OR(a, b, out);
    input [3:0] a, b;
    output [3:0] out;

    OR_GATE O0(out[0], a[0], b[0]);
    OR_GATE O1(out[1], a[1], b[1]);
    OR_GATE O2(out[2], a[2], b[2]);
    OR_GATE O3(out[3], a[3], b[3]);

endmodule

module COMPLEMENT(in, out);
    input [3:0] in;
    output [3:0] out;
    wire [3:0] f;

    NOT_GATE N0(f[0], in[0]);
    NOT_GATE N1(f[1], in[1]);
    NOT_GATE N2(f[2], in[2]);
    NOT_GATE N3(f[3], in[3]);

    ADD COMP1(f, 4'b0001, out);

endmodule

module ADD(a, b, s);
    input [3:0] a, b;
    output [3:0] s;

    wire f0, f1, f2, f3;

    Full_Adder C0(a[0], b[0], 1'b0, f0, s[0]);
    Full_Adder C1(a[1], b[1], f0, f1, s[1]);
    Full_Adder C2(a[2], b[2], f1, f2, s[2]);
    Full_Adder C3(a[3], b[3], f2, f3, s[3]);

endmodule

module Full_Adder(a, b, cin, out, sum);
    input a, b, cin;
    output out, sum;

    wire XOR_AB, AND_AB, AND_CX;

    XOR_GATE FA1(XOR_AB, a, b);
    XOR_GATE FA2(sum, XOR_AB, cin);

    AND_GATE A1(AND_AB, a, b);
    AND_GATE A2(AND_CX, XOR_AB, cin);

    OR_GATE O(out, AND_AB, AND_CX);

endmodule

module ASSIGN_GATE(out, in);

input in;
output out;
wire w1;
NOT_GATE no1(w1, in);
NOT_GATE no2(out, w1);

endmodule

module AND_GATE(out, a, b);
    input a, b;
    output out;

    wire not_b;

    NOT_GATE A1(not_b, b);

    Universal_Gate A2(out, a, not_b);

endmodule

module OR_GATE(out, a, b);
    input a, b;
    output out;

    wire a_bar, w1;

    NOT_GATE OR1(a_bar, a);
    NOT_GATE OR2(out, w1);

    Universal_Gate OR3(w1, a_bar, b);

endmodule

module NOT_GATE(out, a);
    input a;
    output out;

    Universal_Gate N(out, 1'b1, a);

endmodule

module XOR_GATE (out, a, b);
    input a, b;
    output out;

    wire a_bar, b_bar;
    wire and1, and2;

    NOT_GATE NOT1(a_bar, a);
    NOT_GATE NOT2(b_bar, b);
```

```
    AND_GATE NOT3(and1, a, b_bar);
    AND_GATE NOT4(and2, a_bar, b);

    OR_GATE NOT5(out, and1, and2);

endmodule

module XNOR_GATE(out, a, b);
    input a, b;
    output out;

    wire w1;

    XOR_GATE XNOR1(w1, a, b);

    NOT_GATE XNOR2(out, w1);

endmodule

module Universal_Gate(out, a, b);
    input a, b;
    output out;

    wire not_b;

    not N(not_b, b);

    and A(out, a, not_b);

endmodule
```

b.Picture

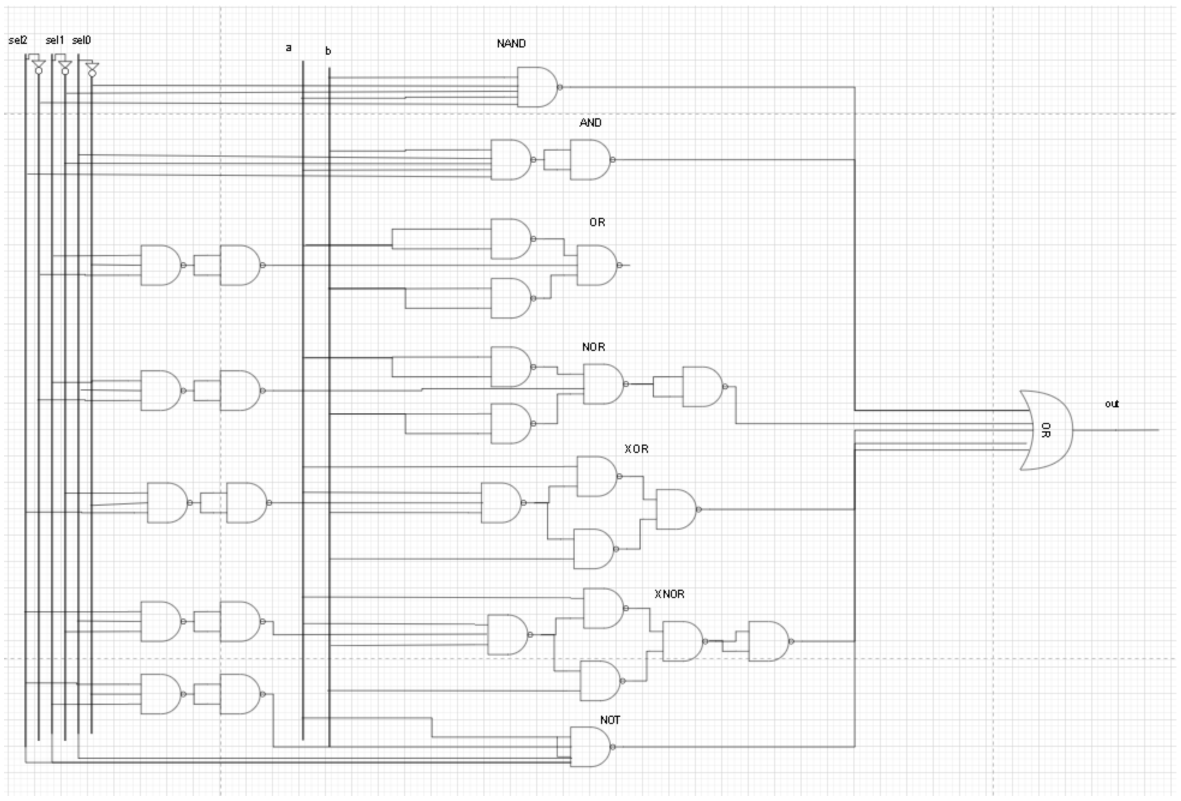


7. Extra Questions

a. Difference between Full Adder and Half Adder

Half Adder	**Full Adder**
Adds 2 of 1-bit Binary Digits, which generates carry and sum of inputs	Adds Three of 1-bit binary digits for performing addition (+), which generates sum of all three inputs and carry value
Does not add carry from previous addition to the next one	Along with the current inputs, also adds the previous carry
consist of 1 AND Gate and 1 XOR Gate	Consists of one OR Gate and 2 XOR Gate and 2 AND Gates
Two inputs (A and B)	Three Inputs (A, B, Cin)
It is good for Digital Measuring Devices,computers,calculators.	Good for various digital processors, addition of multiple bits.

b. Basic question 2, all modules made from NAND gate.



8. What we have learned from LAB2

We have learned how to use Universal Gates as our main gate and implement our original gates AND, OR, NOT and so on. Universal Gates can be used to define every logic variation. In addition to learning more about the Universal Gates, we also learned how to code it in verilog and also use hierarchical modules to implement the new ones for every bit.

In contrast, we learned a lot of new things about adders and how to code them in our Vivado Verilog. From the adders, the CLA was most confusing one to implement as it is considered to be a higher class super adder with respect to the others.

Finally, for the question two, we learned to use 8-to-1 mux to implement different functions within the three selected lines. With each selection line giving certain function to activate. It was one of the challenging ones, right after the FPGA implementation.

The most challenging part was the FPGA Implementation to use Gate Level to convert the 4bit to 7bit, where we needed to count each cathode output, even though it was giving us headaches and countless hours to work on it, we finally did it. There was a couple misunderstandings on the bit 110 but we were able to fix the bug.

9.Our sources

https://www.fpga4student.com/

https://app.diagrams.net/

https://en.wikipedia.org/wiki/Carry-lookahead_adder

https://www.youtube.com/watch?v=9lyqSVKbyz8

https://www.youtube.com/watch?v=1Z0i9lycXb4

https://www.researchgate.net/figure/Carry-Look-Ahead-Adder-8-bit_fig3_228866487

10. Our contributions

As there were a lot of tasks, we assigned each question one by one. If Jason was doing question one, Tuguldur would start working one question two. If any of us could not or can not understand, we google and searached for various educational vidoes to help us understand better. If one of us finished first, we would help the other finish faster, so that we can do the next two questions following the little algorithm we created for our group. Jason was charge of questions one, four, five as Tuguldur was in charge of questions two and three. As for FPGA design, we worked on it together, but on the Demo there were an issue with . However, we both chipped inorder to meet the deadlines for each questions that we set for each other.