Hardware Design

Lab 2 Report

Advanced Gate-Level Verilog

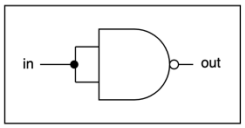
Team 01

112062144 謝佳晉　112062144 范升維

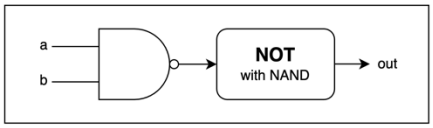
Table of Contents:

Basic Q1. Nand Implement

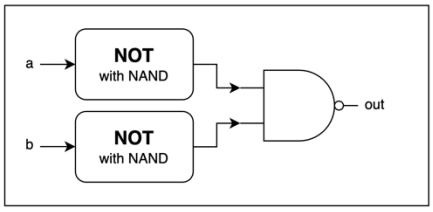
NOT_w_NAND



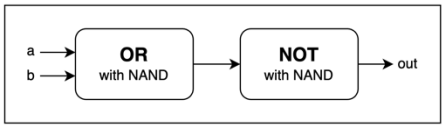
AND_w_NAND



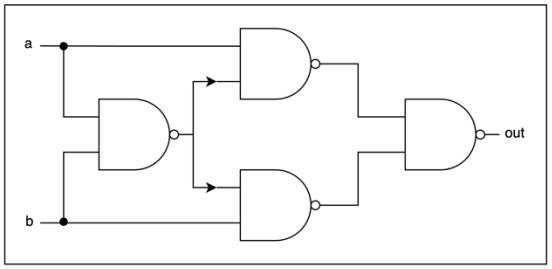
OR_w_NAND



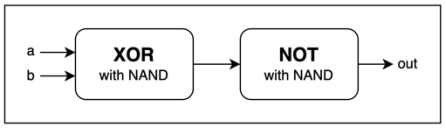
NOR_w_NAND



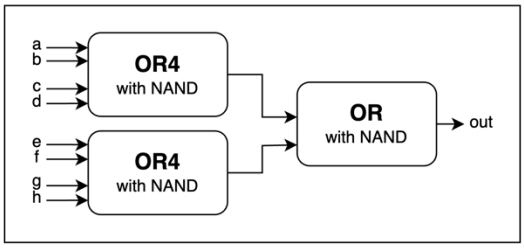
XOR_w_NAND



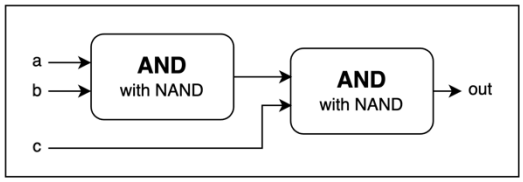
XNOR_w_NAND



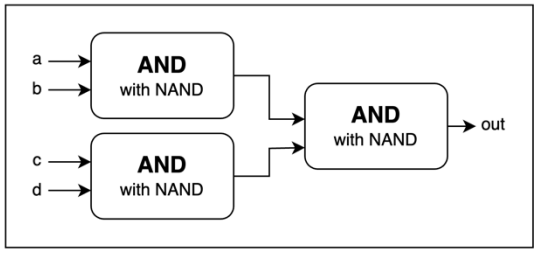
OR8_w_NAND



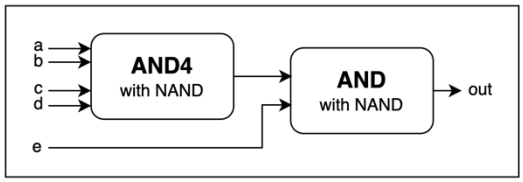
AND3_w_NAND



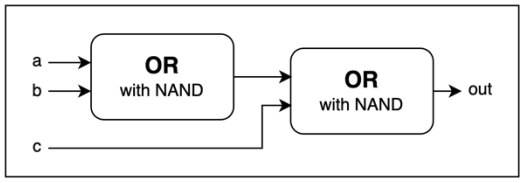
AND4_w_NAND



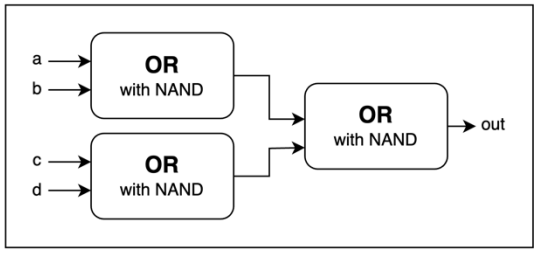
AND5_w_NAND



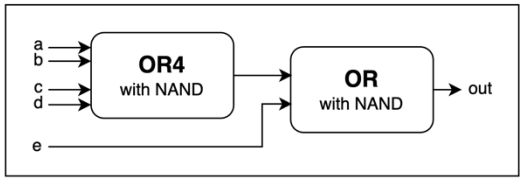
OR3_w_NAND

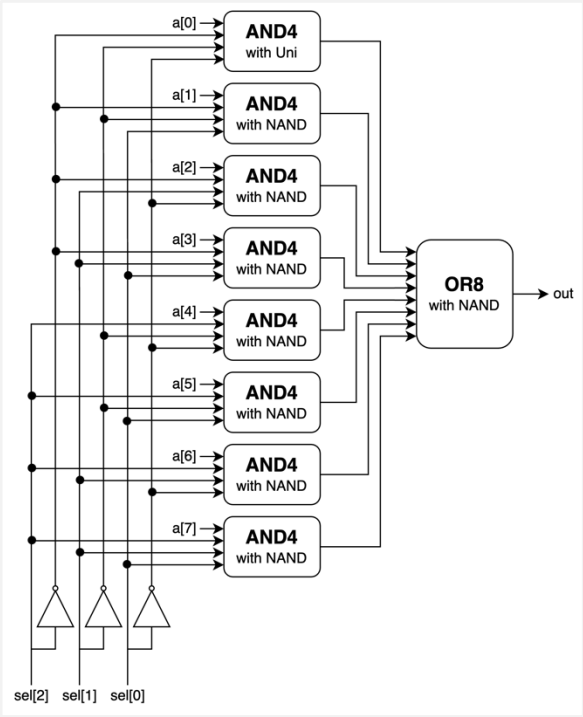


OR4_w_NAND

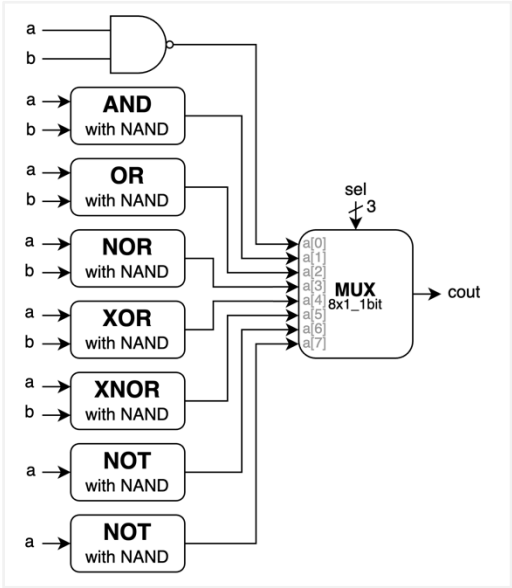


OR5_w_NAND





▲ Figure 0.1: 8x1_1bit MUX



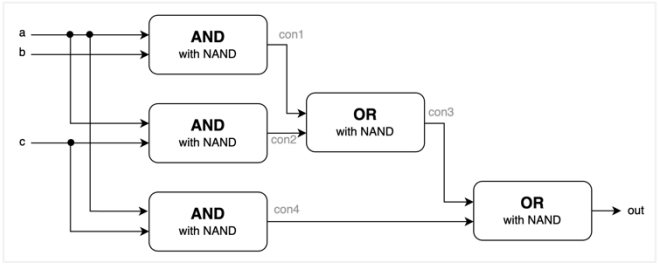
▲ Figure 0.2: NAND Implement

This problem requires us to utilize the NAND gate, which is a universal gate, to construct other basic gates. On the right side of the previous page, we have additionally implemented modules **AND3**, **AND4**, **AND5**, **OR3**, **OR4**, and **OR5**. These were prepared in advance for later use in the advanced Q3 section. Ultimately, we implemented an 8x1 1-bit multiplexer (**MUX**) to return the results of various basic gates in response to different input *sel* signals.

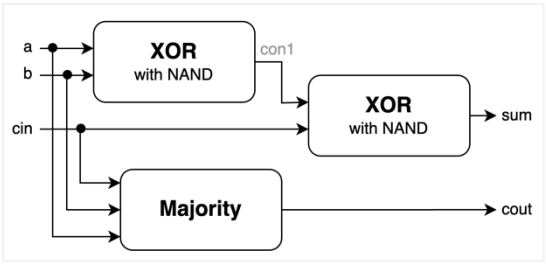
Basic Q3. Difference between Half Adder and Full Adder

Half adders and full adders differ primarily in their input complexity and arithmetic capabilities. A half adder processes two single-bit inputs, generating a sum and carry output. In contrast, a full adder accommodates **three single-bit inputs**, including a carry-in, producing a sum and carry-out. This additional input enables full adders to **integrate previous calculation results**, making them suitable for complex arithmetic operations and multi-bit additions.

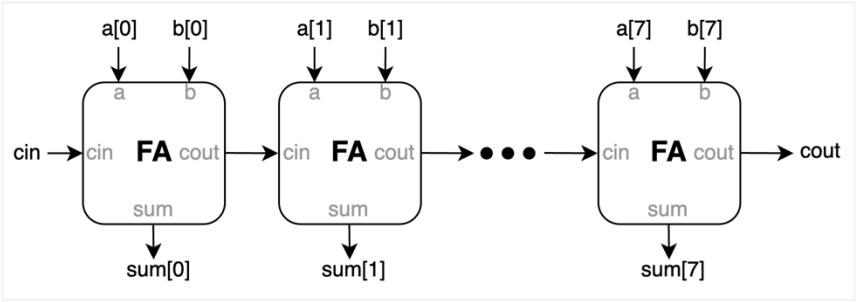
Advanced Q1. 8-bit Ripple Carry Adder (RCA)



▲ Figure 1.1: Majority



▲ Figure 1.2: Full Adder

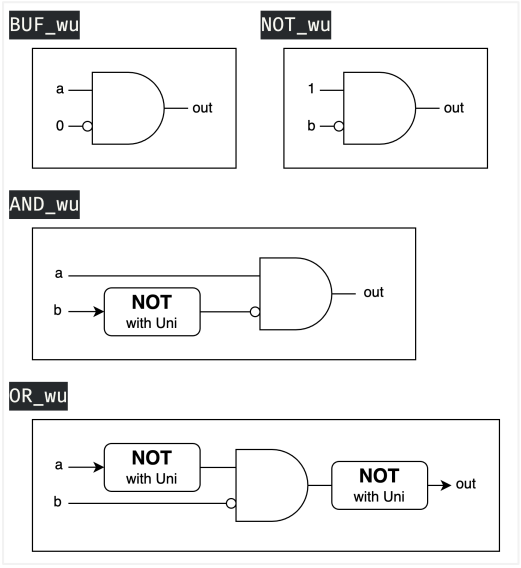


▲ Figure 1.3: Ripple Carry Adder

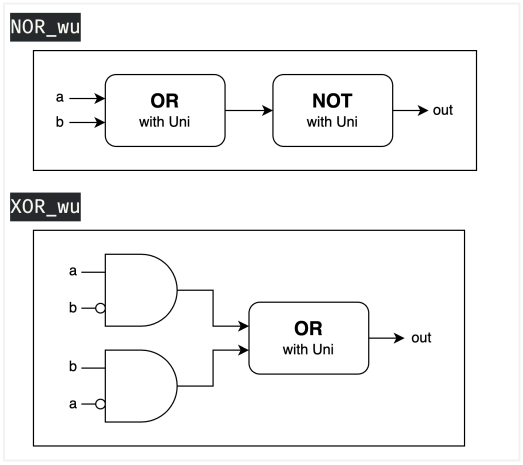
- 3 inputs: *a[7:0], b[7:0], cin*
- 2 outputs: *sum[7:0], cout*

To construct a **8-bit RCA**, we use eight Full Adders connecting each other to calculate each sum and carry. The first(rightmost), second, third, …, seventh Full Adders produce carry, for the next Full Adder to take as *cin*, and the eighth Full Adder produce the overall carry of the final answer. Each bit of sum is calculated by each Full Adder respectively. Additionally, due to the demand that we could only use NAND gates, so all the basic gates we are supposed to use are replaced by our hand-made modules, and so does following advanced question 3 and 4.

Advanced Q2. Decode and execute

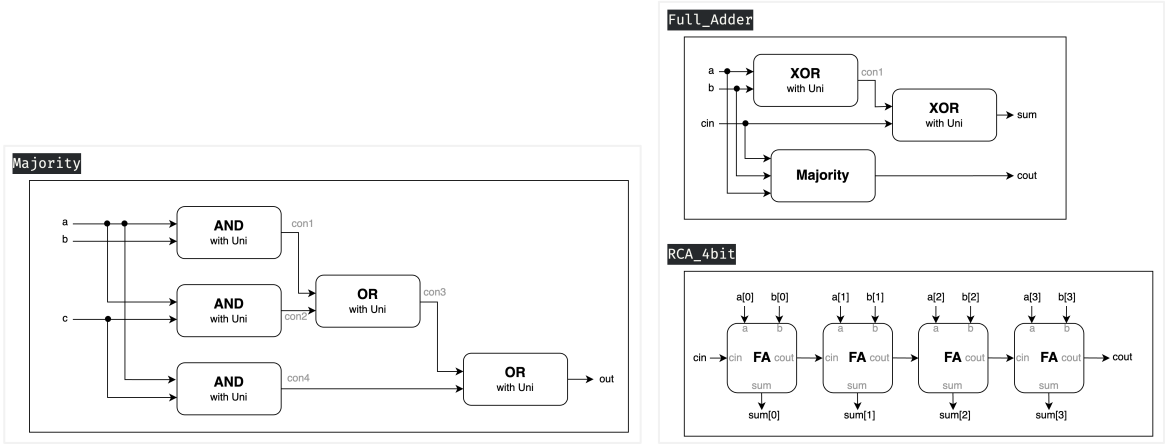


▲ Figure 2.1: Basic Gates



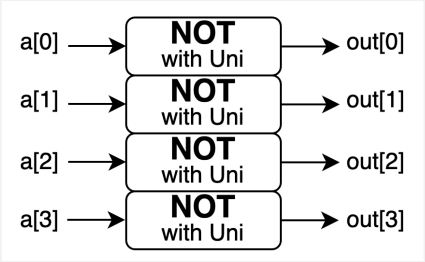
▲ Figure 2.2: Basic Gates

By the requirement of the question that we could only use the specific universal gate, we first built some fundamental gates shown above for our later use.

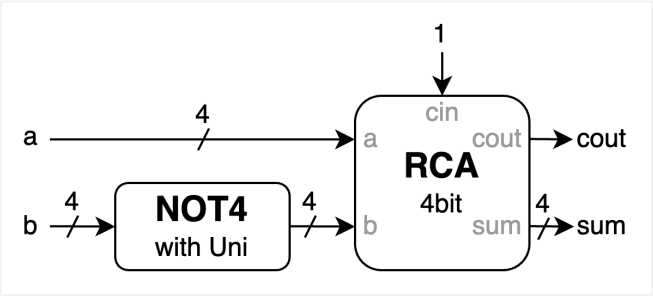


▲ Figure 2.3: 4bit Ripple Carry Adder

Then, we constructed 4bit RCA with Full Adder, the structure of which are alike what we built in Q1, but all with specific universal gate modules.

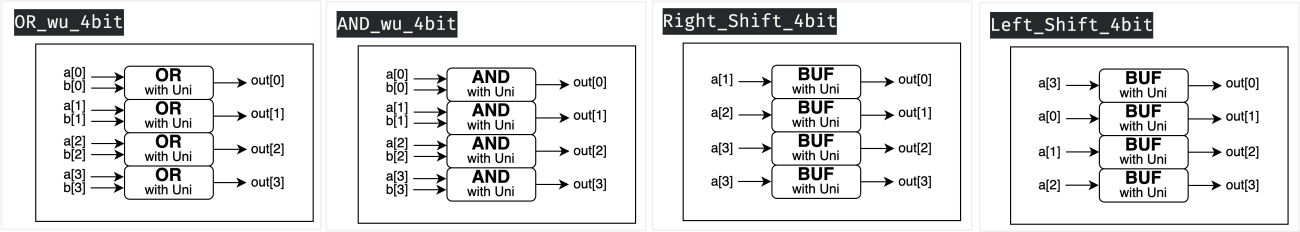


▲ Figure 2.4: 4bit NOT

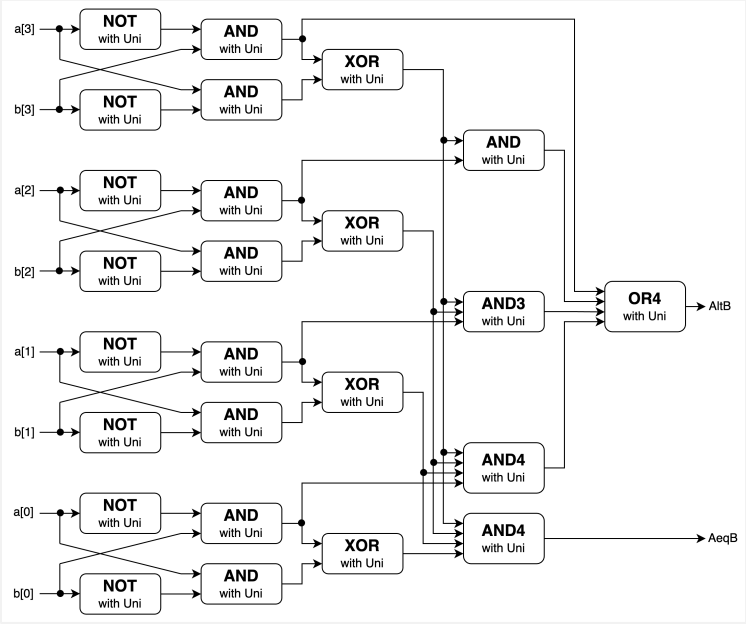


▲ Figure 2.5: 4bit Ripple Carry Adder - Minus

Next, to be able to easily manipulate the operation of "minus", we constructed a "**Minus version of RCA**" shown in Figure 2.5, using the mechanism of **Two's complement**, which is making the subtrahend number be negative by flipping all the bit of it and adding one to it.

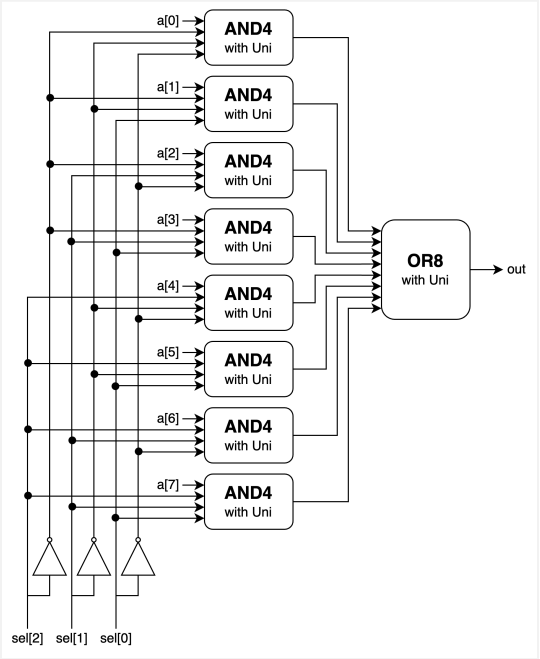


After that, we needed to build bitwise OR, bitwise AND, Arithmetic Right Shift and Circular Left Shift, so we accomplished it by easily using the method shown on the above figures.

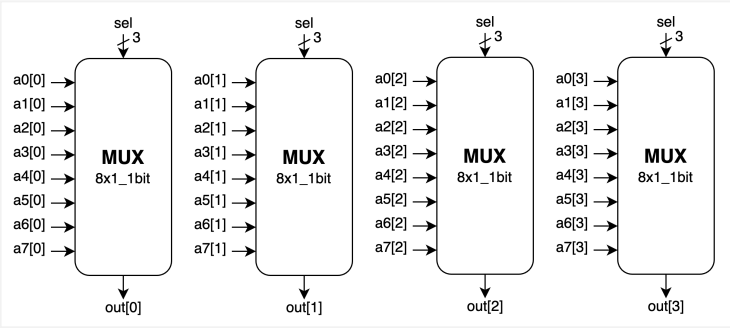


▲ Figure 2.5: Compare

Subsequently, we had to build the compare module, to distinguish two 4-bit number whether the first number is smaller than the second, or both of them are equal, the mechanism of which is that if their most significant bit is different (one is 1's, and the other is 0's), then the inequality can be seen, but if their MSB are the same, then check the second significant bit, and so on and so forth. After a series of comparison, if they are all the same, the $AeqB$ signal will be 1's.

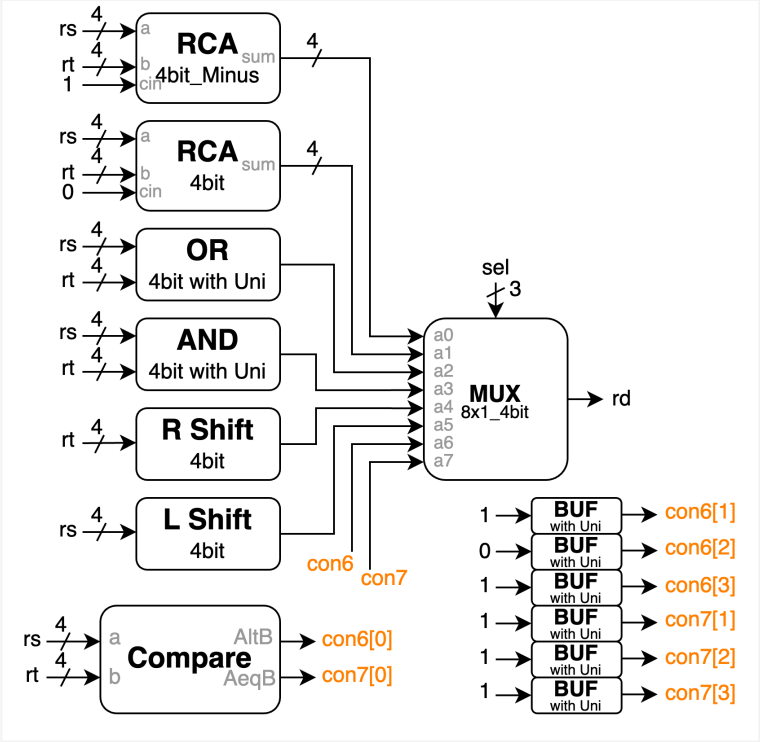


▲ Figure 2.6: 8x1 1bit MUX



▲ Figure 2.7: 8x1 4bit MUX

In the penultimate step, we constructed an 8x1 4-bit MUX by utilizing four 8x1 1-bit MUX. This approach enabled our final module to operate efficiently and concisely.



▲ Figure 2.8: Final Module(Decode and Execute)

- 2 inputs: *rs[3:0], rt[3:0]*
- 1 control signal: *sel[2:0]*
- 1 output: *rd[3:0]*

Finally, we reached the last step. We utilized the recently constructed 8x1 4-bit MUX along with all previously created modules to invoke the appropriate module based on the value of the *sel* signal, thereby generating the desired output. Of particular note are the last two compare modules, which only produce a single-bit output. As specified in the problem statement, the other three bits of these modules were assigned **predetermined** values. To accomplish this, we directly employed buffers (**BUF**) to assign these values.

Advanced Q3. 8-bit CLA

#TODO

Advanced Q4. 4-bit Multiplier

#TODO

Advanced Q5. Exhaustive testbench design

```
initial begin
    repeat(2) begin
        repeat (2**4) begin
            repeat (2**4-1) begin
                #0.001
                error = (sum != a+b);
                #0.004
                b = b + 4'b1;
            end
            #0.001
            error = (sum != a+b);
            #0.004
            a = a + 4'b1;
            b = 4'b0;
        end
        cin = 4'b1;
    end
    done = 1'b1;
    #0.001 error = (sum != a+b);
    #0.005 done = 1'b0;
end
```

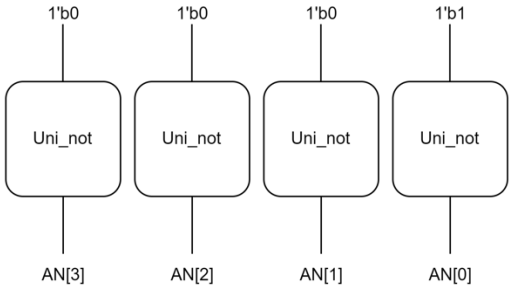
▲ Figure 5.1: Our testbench design

To design a proper exhaustive testbench, we use “Three-level nested loop” to run all the number combinations we will encounter when using a 4-bit RCA. When the sum produced by the 4-bit RCA isn’t equal to the actual value, the error will be set to 1’b1 one nanosecond later and last for five nanoseconds. After the nested loop run out, we set done to 1’b1 in

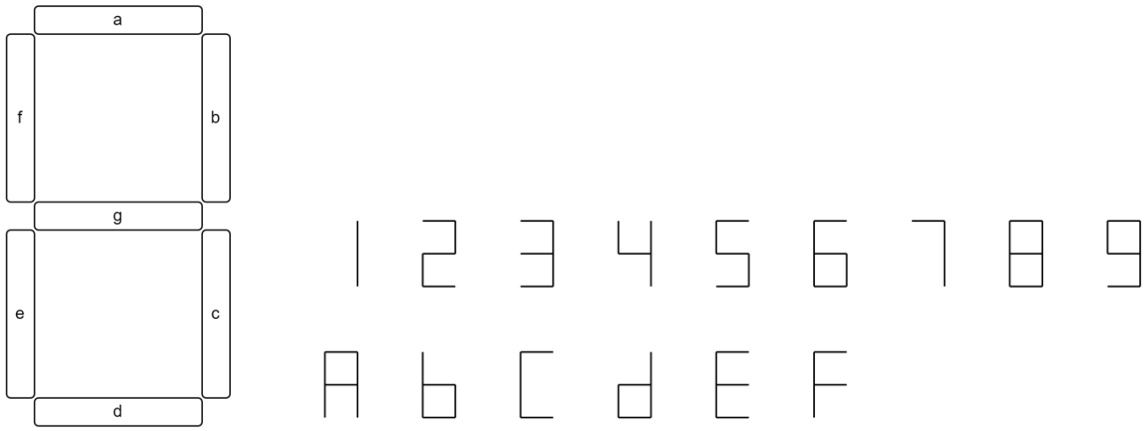
order to show that all of the combinations have been tested. Lastly, we reset done to 1’b0 after five nanoseconds.

FPGA: 7-Segment Display Control

In order to light the rightmost 7-segment display, we set AN[0] to 1’b0 and AN[3:1] to 1’b1.



In the 7-segment display, we have to set the value of each segment port respectively according to the value of rd[3:0]. Below is the design of number displays.



To set the value correctly, we have to draw K-map of each segment to get Boolean expressions. Take segment ‘a’ for example, we want to light up segment ‘**a**’ when rd[3:0] equals to 2, 3, 5, 6, 7, 8, 9, a, c, e, f, so we construct a K-map like this:

rd [1][0] / [3][2]	00	01	11	10
00	0	1	0	0
01	1	0	0	0
11	0	1	0	0
10	0	0	1	0

From this K-map, we can know the boolean expression of segment '**a**' is:

(rd[0]&(!rd[1])&(!rd[2])&(!rd[3]))|((!rd[0])&(!rd[1])&rd[2]&(!rd[3]))

|(rd[0]&(!rd[1])&rd[2]&rd[3])|(rd[0]&rd[1]&(!rd[2])&rd[3])

We can get other 6 segments' boolean expressions likewise:

b: (rd[0]&(!rd[1])&rd[2]&(!rd[3]))|((!rd[0]&rd[1]&rd[2])|(rd[0]&rd[1]&rd[3])|

((!rd[0]&rd[2]&rd[3])

c: ((!rd[0])&rd[1]&(!rd[2])&(!rd[3]))|(rd[1]&rd[2]&rd[3])|((!rd[0])&rd[2]&rd[3])

d: (rd[0]&(!rd[1])&(!rd[2])&(!rd[3]))|((!rd[0])&(!rd[1])&rd[2]&(!rd[3]))|

((!rd[0])&rd[1]&(!rd[2])&rd[3])|(rd[0]&rd[1]&rd[2])

e: (rd[0]&(!rd[3]))|(rd[0]&(!rd[1])&(!rd[2]))|((!rd[1])&rd[2]&(!rd[3]))

f: (rd[0]&(!rd[1])&rd[2]&rd[3])|(rd[0]&(!rd[2])&(!rd[3]))|(rd[1]&(!rd[2])&(!rd[3]))|

(rd[0]&rd[1]&(!rd[3]))

g: (rd[0]&rd[1]&rd[2]&(!rd[3]))|((!rd[0])&(!rd[1])&rd[2]&rd[3])|((!rd[1])&(!rd[2])&(!rd[3]))

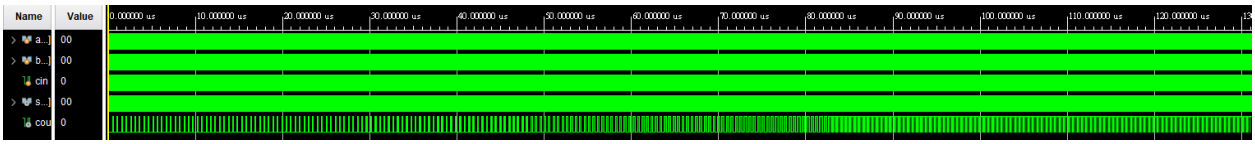
After we have all segments' boolean expressions, we can construct the FPGA module with primitive logic gates.

(output: light[3:0] □AN[3:0], display[6:0] □segment 'g', 'f', 'e', 'd', 'c', 'b', 'a')

Testbenches

A. 8-bit RCA

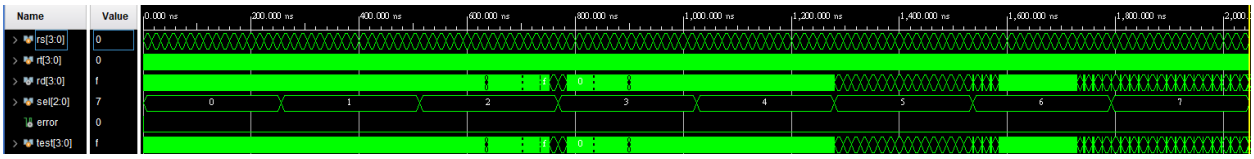
```
module Ripple_Carry_Adder_t();
    reg [7:0] a = 8'b0, b = 8'b0;
    reg cin = 1'b0;
    wire [7:0] sum;
    wire cout;
    Ripple_Carry_Adder R1(a, b, cin, cout, sum);
    always #1 cin = ~cin;
    initial begin
        repeat(2**8) begin
            repeat(2**8-1) begin
                #2 b = b + 8'b1;
            end
            #2
            a = a + 8'b1;
            b = 8'b0;
        end
        $finish;
    end
endmodule
```



We can test the module by using three level nested loop like advanced Q5 to run all the number combinations and see whether the output is correct.

B. Decode and execute

```
module Decode_And_Execute_t();
    reg unsigned [3:0] rs = 4'b0, rt = 4'b0;
    wire unsigned [3:0] rd;
    reg [2:0] sel = 3'b000;
    wire error;
    reg unsigned [3:0] test;
    Decode_And_Execute G1(rs, rt, sel, rd);
    assign error = !(rd == test);
    initial begin
        //SUB
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = rs - rt;
                #1
                rt = rt + 4'b1;
            end
            test = rs - rt;
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        rt = 4'b0;
        sel = sel + 3'b1;
        //ADD
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = rs + rt;
                #1
                rt = rt + 4'b1;
            end
            test = rs + rt;
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        rt = 4'b0;
        sel = sel + 3'b1;
        //bitwise OR
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = rs | rt;
                #1
                rt = rt + 4'b1;
            end
            test = rs | rt;
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        rt = 4'b0;
        sel = sel + 3'b1;
        //bitwise AND
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = rs & rt;
                #1
                rt = rt + 4'b1;
            end
            test = rs & rt;
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        rt = 4'b0;
        sel = sel + 3'b1;
        //Right Shift
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = {rt[3], rt[3], rt[2], rt[1]};
                #1
                rt = rt + 4'b1;
            end
            test = {rt[3], rt[3], rt[2], rt[1]};
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        rt = 4'b0;
        sel = sel + 3'b1;
        //Left Shift
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = {rs[2], rs[1], rs[0], rs[3]};
                #1
                rt = rt + 4'b1;
            end
            test = {rs[2], rs[1], rs[0], rs[3]};
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        rt = 4'b0;
        sel = sel + 3'b1;
        //Compare LT
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = {1'b1, 1'b0, 1'b1, rs < rt};
                #1
                rt = rt + 4'b1;
            end
            test = {1'b1, 1'b0, 1'b1, rs < rt};
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        rt = 4'b0;
        sel = sel + 3'b1;
        //Compare EQ
        repeat(2**4)begin
            repeat(2**4-1)begin
                test = {1'b1, 1'b1, 1'b1, rt == rs};
                #1
                rt = rt + 4'b1;
            end
            test = {1'b1, 1'b1, 1'b1, rt == rs};
            #1
            rs = rs + 4'b1;
            rt = 4'b0;
        end
        $finish;
    end
endmodule
```



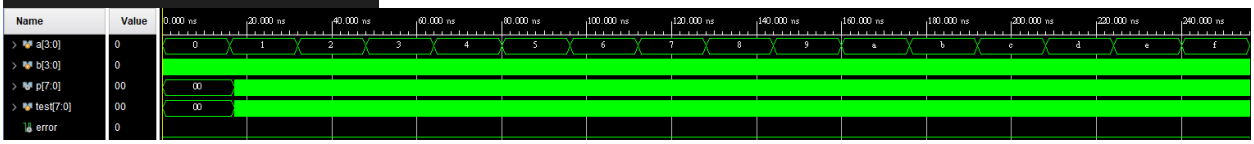
Steps:
Initialize→SUB→reset→ADD→reset→Bitwise OR→reset→Bitwise AND→reset→Right Shift→reset→Left Shift→reset→Compare LT→reset→Compare EQ→finish

C. 8-bit CLA

#TODO

D. 4-bit Multiplier

```
module Multiplier_4bit_t();
    reg [3:0] a = 4'b0, b = 4'b0;
    wire [7:0] p;
    reg [7:0] test;
    wire error;
    assign error = (test != p);
    Multiplier_4bit M1(a, b, p);
    initial begin
        repeat (2**4) begin
            repeat (2**4-1) begin
                test = a * b;
                #1
                b = b + 4'b1;
            end
            test = a * b;
            #1
            a = a + 4'b1;
            b = 4'b0;
        end
        $finish;
    end
endmodule
```



We can test the module by using two level nested loop to run all the number combinations and use signal ‘error’ to detect whether the output is correct.

What we have learned from Lab2?

#TODO

Contributions

謝佳晉：

wrote Q1~Q4 Verilog
wrote testbench Q1~Q4
performed Q1~Q4 simulation on Vivado
drew diagram Q3, Q4
constructed report foundation with images and descriptions
made FPGA demonstration

范升維：

wrote Q1~Q4 Verilog
wrote testbench Q1~Q4
performed Q1~Q4 simulation on Vivado
drew diagram Q1, Q2
revised diagram Q3, Q4
organized, beautified report, added more descriptions and revised images
implemented FPGA demonstration