Hardware Design Lab 2 Report

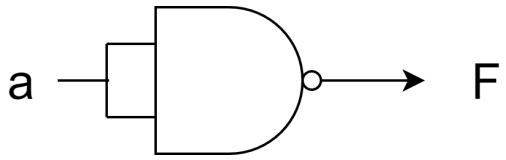
Group Number: 30

Members: 112062130 侯佑勳
112062326 孔祥光

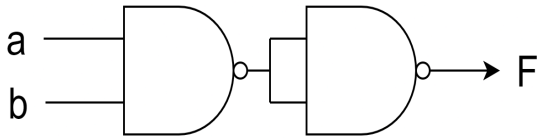
Contents:

NAND Implementations & Adders

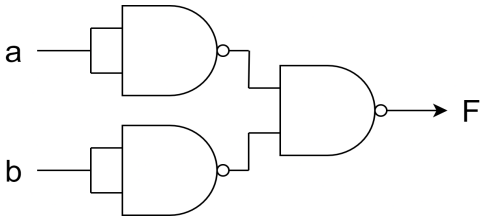
1) **Diagrams of logic functions (NOT, AND, OR, etc.) built with NAND gates, and the MUXes with the whole circuitry.**



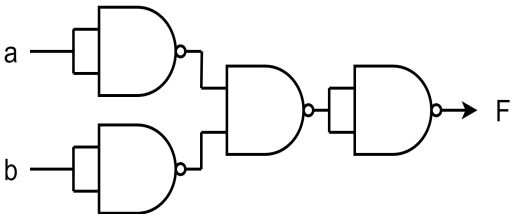
∆ **Figure 1-1** NOT (built with NAND)



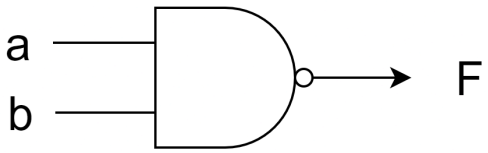
∆ **Figure 1-2** AND (built with NAND)



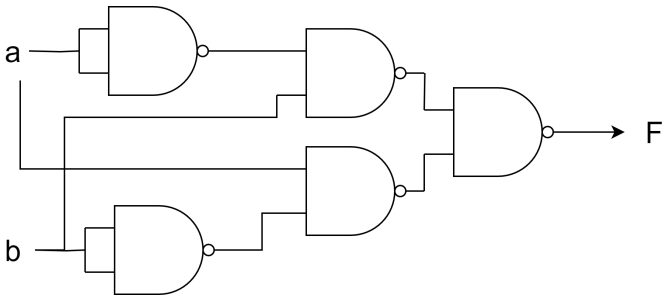
∆ **Figure 1-3** OR (built with NAND)



∆ **Figure 1-4** NOR (built with NAND)

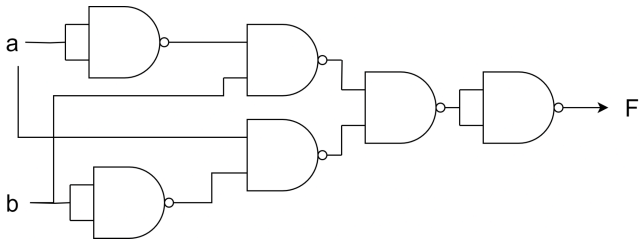


∆ **Figure 1-5** NAND (built with NAND)

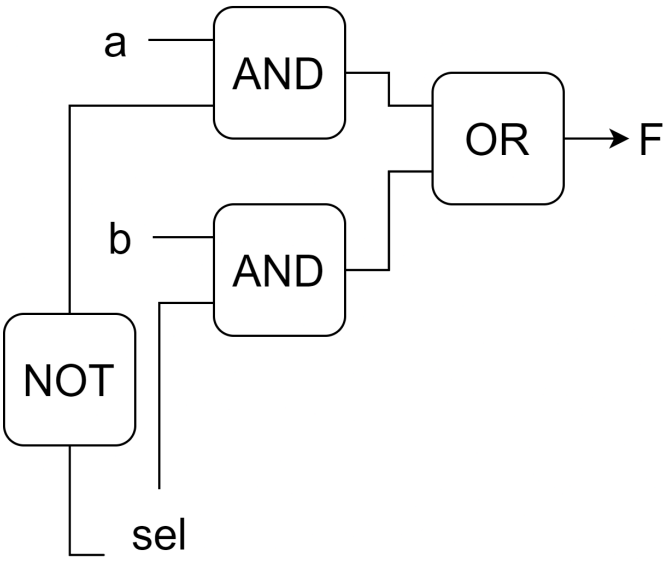


∆ **Figure 1-6** XOR (built with NAND)

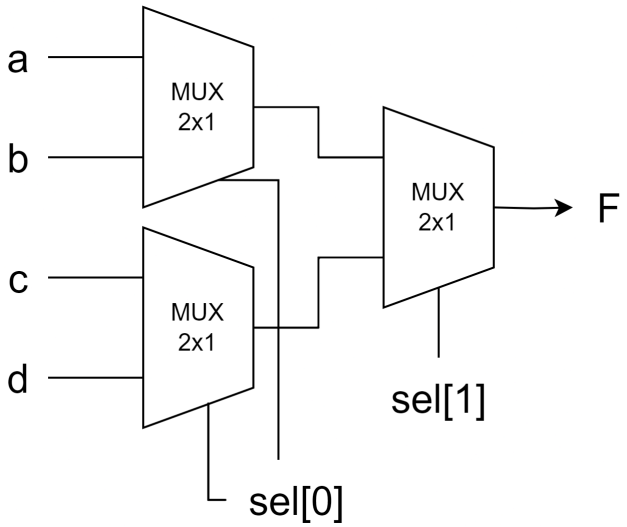
The intersection of signal a and b are not connected.



∆ **Figure 1-7** XNOR (built with NAND)

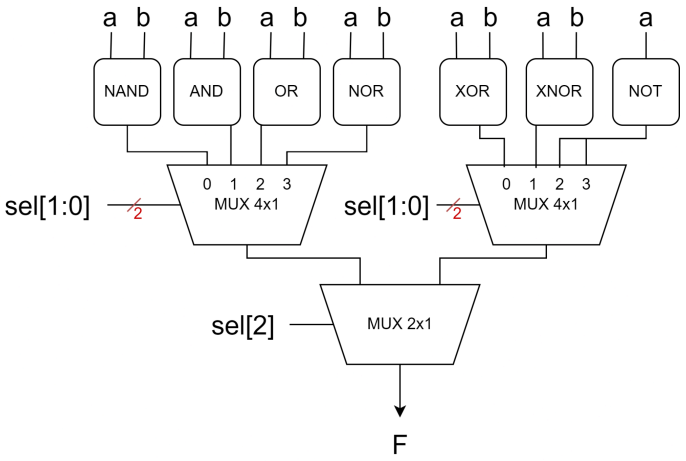


∆ **Figure 1-8** MUX_2x1_1-bit (built with NAND)



∆ **Figure 1-9** MUX_4x1_1-bit (built with NAND)

The intersection of signal sel_0 and output from MUX are not connected.



∆ **Figure 1-10** the whole circuitry of basic Q1

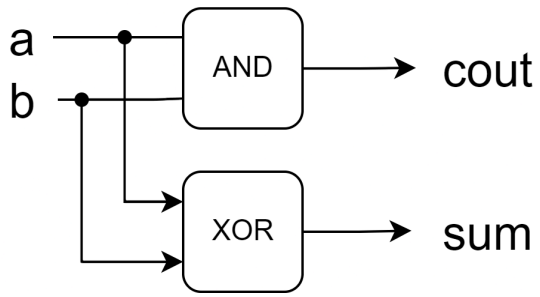
2) **Difference between half adder and full adder?**
Half-adder adds **two** 1-bit numbers, producing a sum and carry-out.Full-adder adds **three** 1-bit numbers (**with carry-in**), producing a sum and carry-out.

Gate-Level Designs, Explanation & Testing

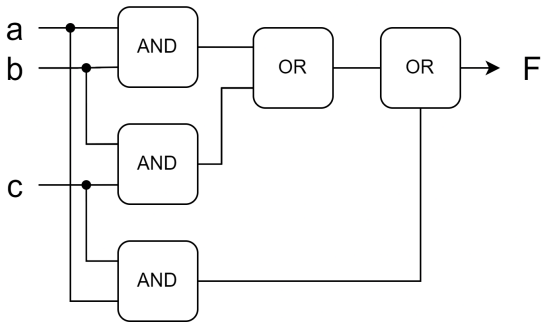
1) Ripple Carry Adder

All the primitive logic gates in use are implemented by NAND gates only.

First we build Half-adder, and a 3-bit Majority function. The Majority function asserts when there are more true inputs than false ones.

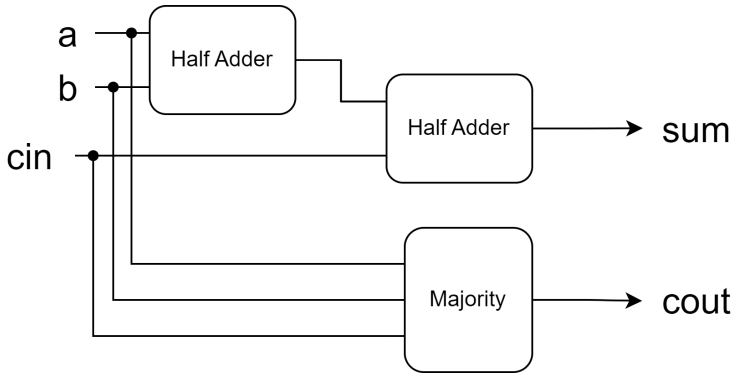


∆ **Figure 2-1** Half-adder



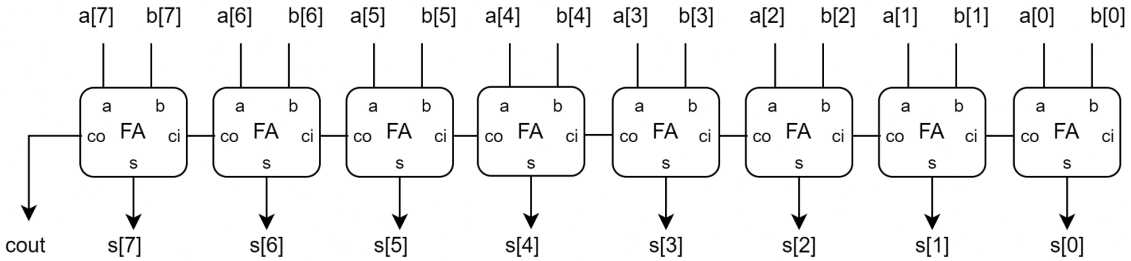
∆ **Figure 2-2** 3-bit Majority gate

With those we can construct a Full-adder. We use one Half-adder to sum up a and b, then use another one to add up the sum and carry-in. To determine whether to carry out, we'll check if 2 or more bits of a,b and cin are high, and the Majority function does exactly that.



∆ **Figure 2-3** Full Adder

By combining eight Full-adders, we're able to build an 8-bit Ripple Carry Adder. Notice how each FA has to wait for the carry-in signal from its predecessor's output, the carry-out signal, so the process of addition happens like a "ripple".



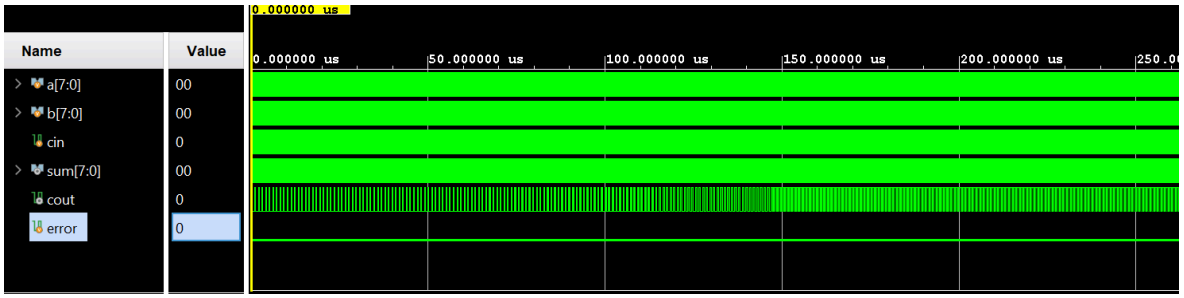
∆ **Figure 2-4** 8-bit RCA

For testing, we need to ensure that the module works under all case, which includes all the 2^{17} possible input patterns. And it's impossible to manually check such a vast waveform output by eye. So we'll introduce a new technique, **error raising**. (Similar to that of the Exhaustive Testbench problem.) An error will be thrown whenever the sum and carry-out doesn't check out.

```
// I/O signals
reg [8-1:0] a = 8'b0000_0000, b=8'b0000_0000;
reg cin = 1'b0;
wire [8-1:0] sum;
wire cout;
Ripple_Carry_Adder RCA(···
);
reg error = 1'b0;
initial begin
    //make sure the module works under all cases, bitwise.
    repeat (2**17) begin
        #1
        //raise error if cout and sum doesn't check out
        error = !( a+b+cin === {cout,sum});
        #1
        //next input pattern
        {a,b,cin} = {a,b,cin}+ 1'b1;
    end

    #1 $finish;
end
endmodule
```

∆ **Figure 2-5** 8-bit RCA testbench

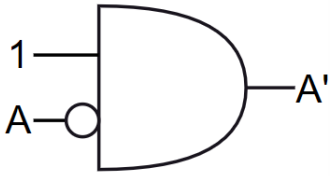
As we can tell from the waveform, the error, if occurred, would be spot rather easily, compared with checking sums case by case manually.



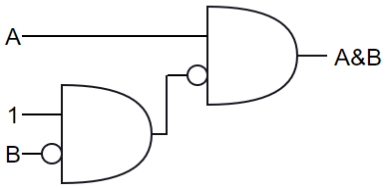
∆ **Figure 2-6** 8-bit RCA simulation waveform

2) Decode & Execute

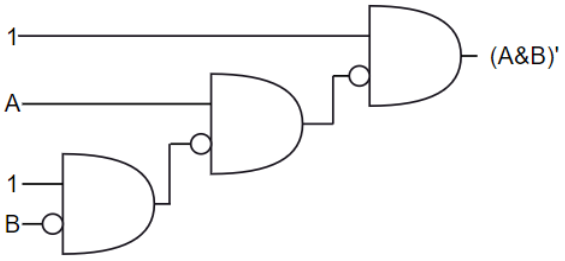
The schematics below show how the basic Boolean functions are implemented by the given Universal gate. (We'll abbreviate the universal gate as "UG" in the following section.)



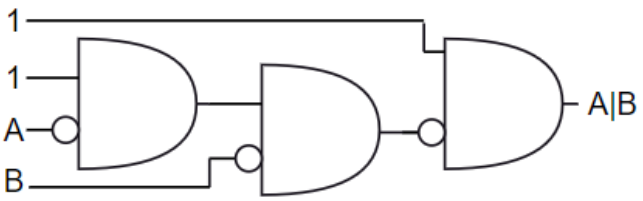
∆ **Figure 3-1** NOT(built with UG)



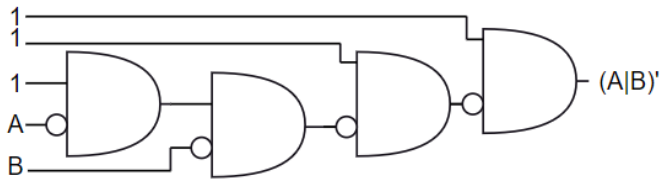
∆ **Figure 3-2** AND(built with UG)



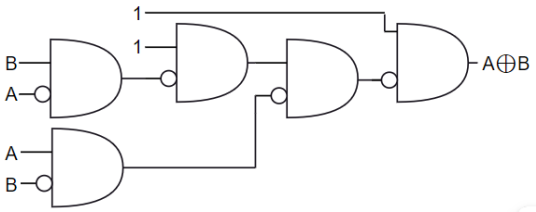
∆ **Figure 3-3** NAND(built with UG)



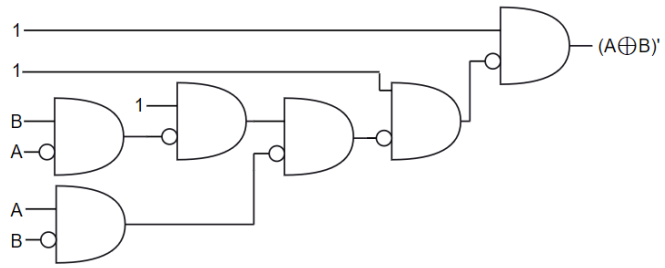
∆ **Figure 3-4** OR(built with UG)



Δ **Figure 3-5** NOR(built with UG)



Δ **Figure 3-6** XOR(built with UG)



Δ **Figure 3-7** XNOR(built with UG)

By using these basic gates, we can realize the following seven functions.

A. ADD

The adder in this question is basically the same as the **Ripple Carry Adder** covered in the previous section, with just a slight change from 8 bits to 4 bits.

B. SUB

We use 2's complement technique and the adder above to make a subtractor.

First, we need to **add a negative sign to the subtrahend**（by inverting each bit, and add 1 to the LSB）.

Second, we use the adder above to add minuend and the negative subtrahend together to achieve the subtract function.

C. BITWISE OR

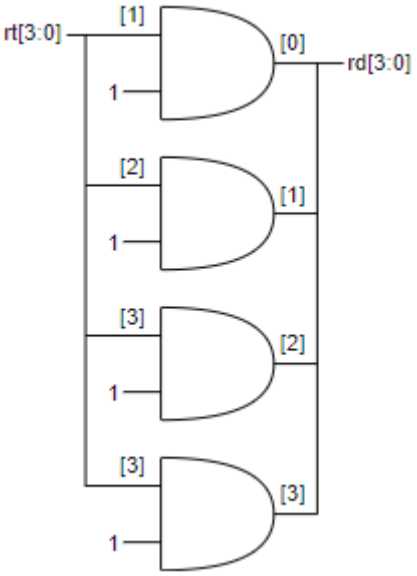
Just connect each bit of rs and rt to an OR gate (which have been defined above) respectively.

D. BITWISE AND

Just connect each bit of rs and rt to an AND gate (which have been defined above) respectively.

E. ARITHMETIC RIGHT SHIFT

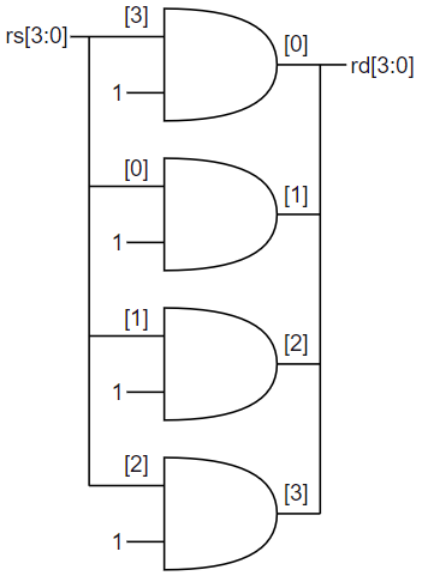
By using 4 AND Gates to shift the data to the right by one bit. However, because we need to implement an arithmetic right shift, we need to reserve the 4th bit (rt[3]) for the sign bit.



∆ **Figure 3-8** ARI.RIGHT SHIFT(rt)

F. ARITHMETIC LEFT SHIFT

By using 4 AND Gates to shift the whole data to the left by one bit.

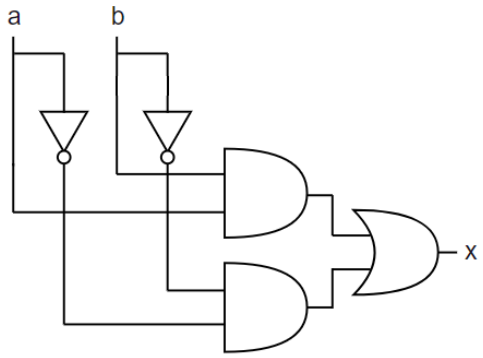


∆ **Figure 3-9** LEFT SHIFT(rs)

G. COMPARE(<)

To compare rs and rt, we must **compare each bit sequentially from the MSB to the LSB**. If the bits in the earlier positions have determined the result, the subsequent digits can be ignored.

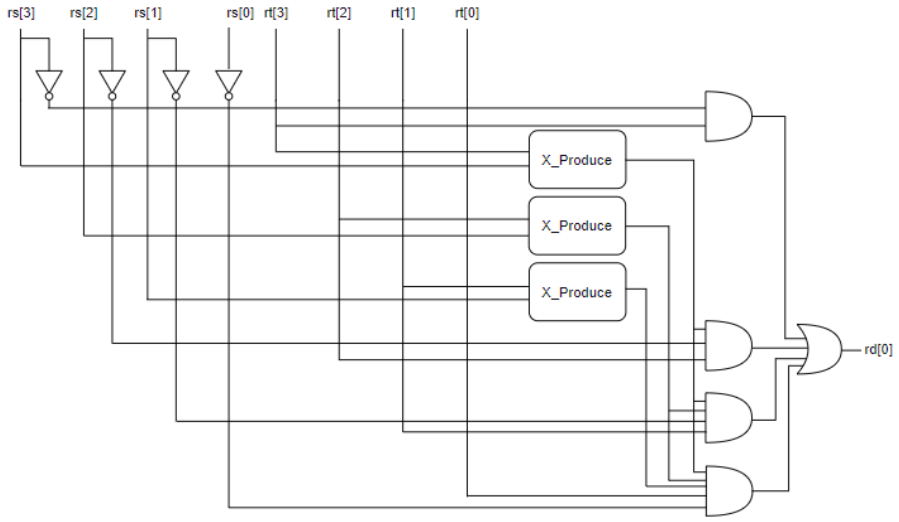
To realize this function, we designed the **X_Produce function**, which **output(x) = ab + a'b'**. If the output is 1, then we should continuously compare the subsequent bits. If the output is 0, then we can ignore the subsequent bits and get the result by testing whether **a'b(a<b) or ab'(a>b) is true**.



∆ **Figure 3-10** *X*_Produce Function

We use the logic expression below to achieve the 4 bits compare function. The comparison of digits will be effective if the more magnificent digits are equal.

$$(rs < rt) = (rs[3])'(rt[3]) + (x[3])(rs[2])'(rt[2]) + (x[3])(x[2])(rs[1])'(rt[1]) + (x[3])(x[2])(x[1])(rs[0])'(rt[0])$$



∆ **Figure 3-11** COMPARE(<)

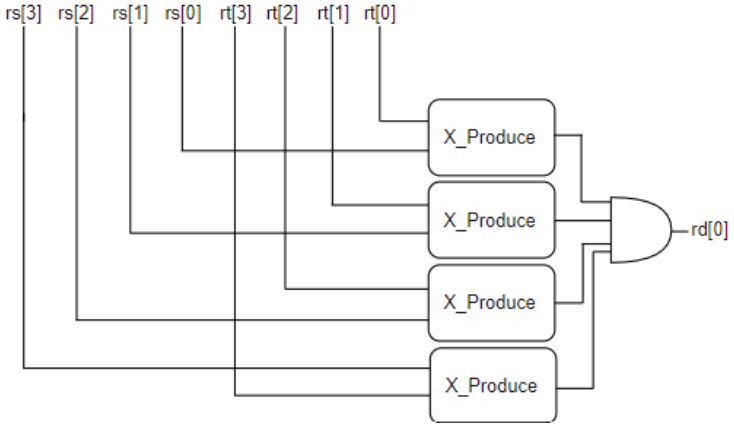
As shown in the figure above, if rs < rt, then rd[0] will be 1, and rd will be 1011.

H. COMPARE(==)

For this function, we utilize four X_Produce functions for each bit to help us determine whether each pair of bits are equal.

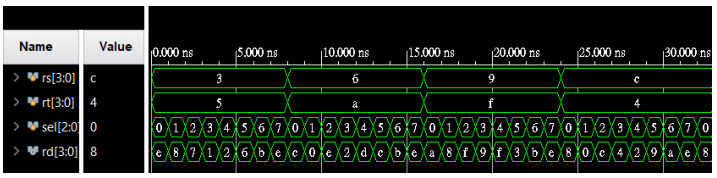
By using an AND gate, if the output is 1, then we can make sure each x must be one which means two inputs are equal.

The logic comparison and picture below show the whole function.
If rs == rt, then rd[0] will be 1, and rd = 1111.
(rs == rt) = (x[3])(x[2])(x[1])(x[0])



∆ **Figure 3-12** COMPARE(==)

In the testbench, we pick some arbitrary inputs, then run through all the op-codes and check if it's the desired result.



∆ **Figure 3-13** simulation waveform of D&E

3) Carry Lookahead Adder

All the primitive logic gates in use are implemented by NAND gates as NAND Implementations

While using the normal adder, we want to know the next digit's cout or sum, we must first know the previous digit's result. It takes a lot of time to wait for the previous digits' calculation if the number has many bits.

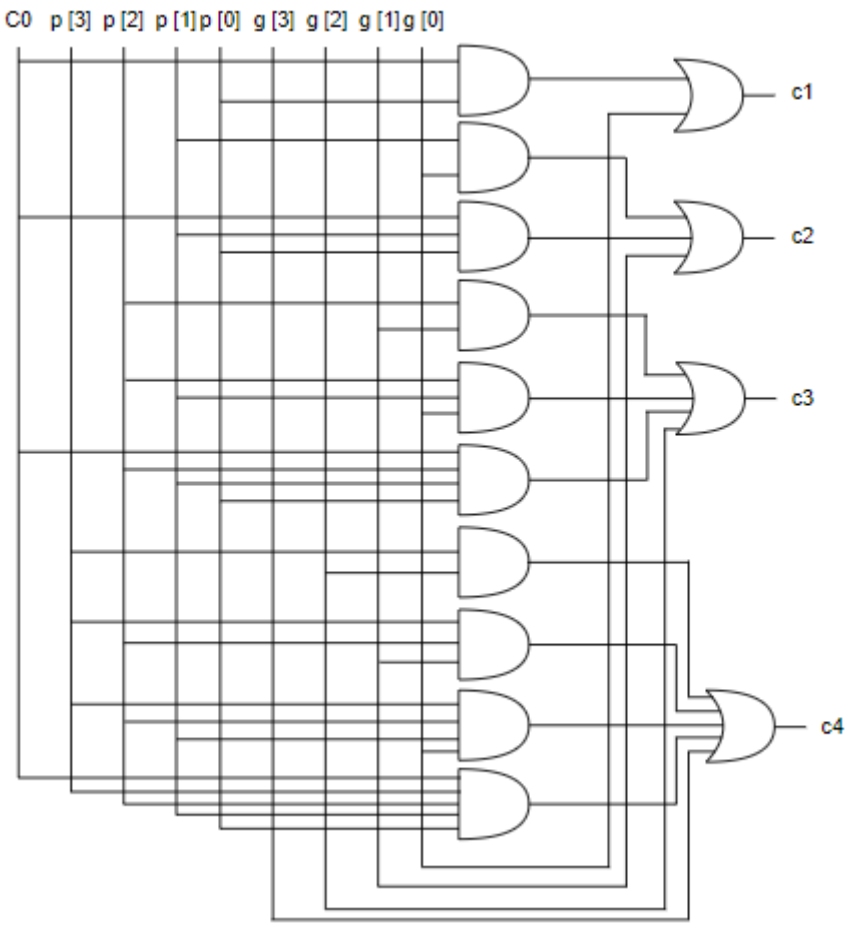
By using CLA, we can just utilize the numbers we want to add and the initial cin to calculate the cout of any digit! We can save a lot of time in that way.

The core knowledge of CLA is generating value and propagating value. Generate value's equation is g_i = $a_i b_i$, which determines whether there will be a cout by a and b themselves. Propagate value's equation is g_i = $a_i + b_i$, which determines whether the cin will generate the cout.

We can describe the i-th digit's cout as $c_i = g_{i-1} + p_{i-1} c_{i-1}$. Because c_{i-1} can also be represented by c_{i-2}, g_{i-2}, p_{i-2}, we can have an equation of any c_i by using c_1, a and b.

Below are equations and logic gate implement explain how $c_1 \sim c_4$ are produced.

$$c_1 = g_0 + c_0 p_0$$
$$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$$
$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$
$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$



∆ **Figure 4-1** 4-bit CLA generator

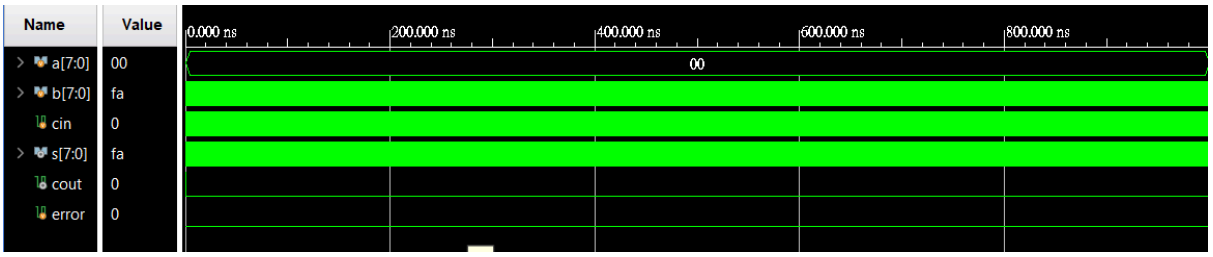
To design the 8 bit CLA, I think we need to use eight 1-bit full adders, two 4-bit CLA generators and two 2-bit CLA generators.
4-bit CLA generator can use the initial cin and the $p_i g_i$ to provide each full adder a proper cin. Those full adders can use these cins to calculate sums. A 2-bit CLA generator can use [3:0] propagate signal and generate signal(provided by 4-bit CLA) to calculate the next cout (the equations below show how the cout be generated, CG = next cout).
PG = $p_0 p_1 p_2 p_3$

GG = $g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3$

CG = GG + PG C_{in}

And we can use this cout to the 2nd 4-bit CLA generator and the other four full adders to complete the sums. Finally, by using a 2-bit CLA generator, we can get our last cout!

The testing for 8-bit CLA is near-identical to that of the 8-bit RCA, since their functionalities are essentially the same: 8-bit addition. Please refer to that section for detailed explanation.



∆ **Figure 4-2** 8-bit CLA simulation waveform

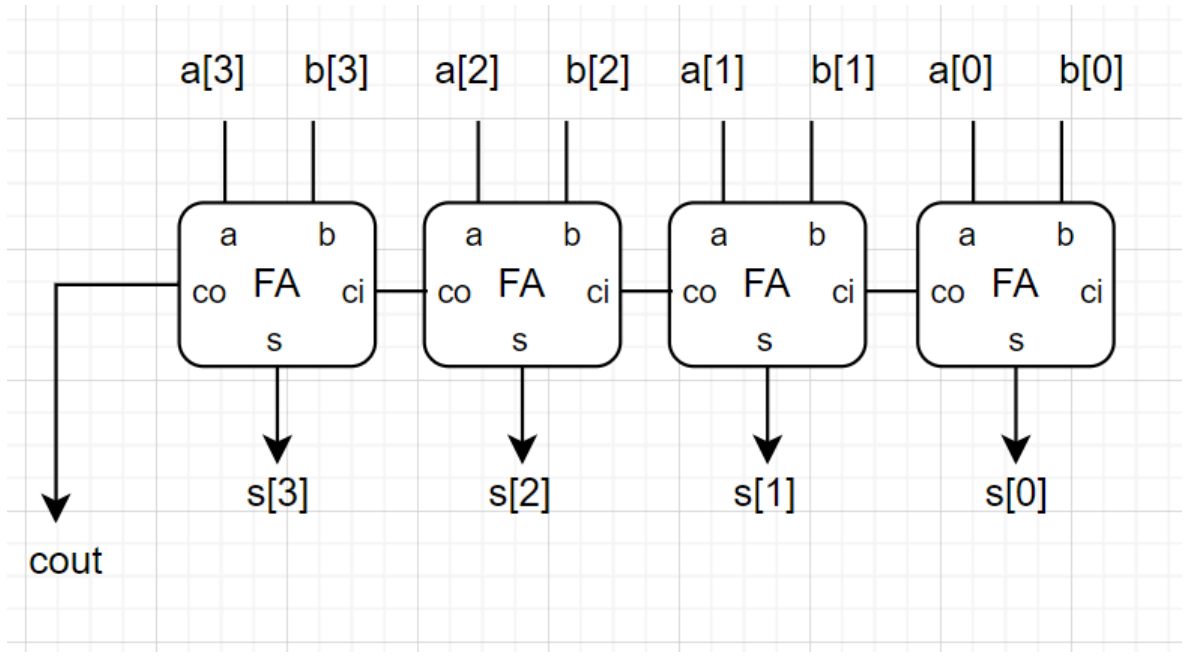
4) Multiplier

All the primitive logic gates in use are implemented by NAND gates only.

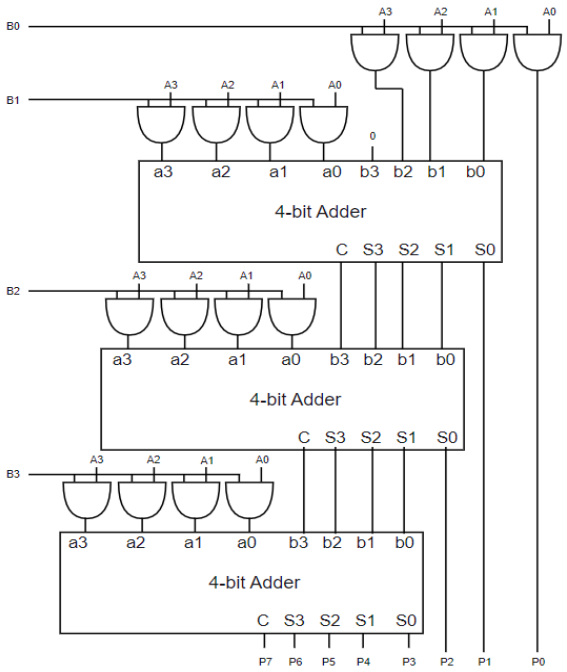
The multiplication process should look something like this:

				a_3	a_2	a_1	a_0
			$\times$	b_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
➕	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

So we'll simply calculate all the partial products, using AND, then sum them all up using 4-bit adders.



∆ **Figure 5-1** 4-bit Adder, no carry-in



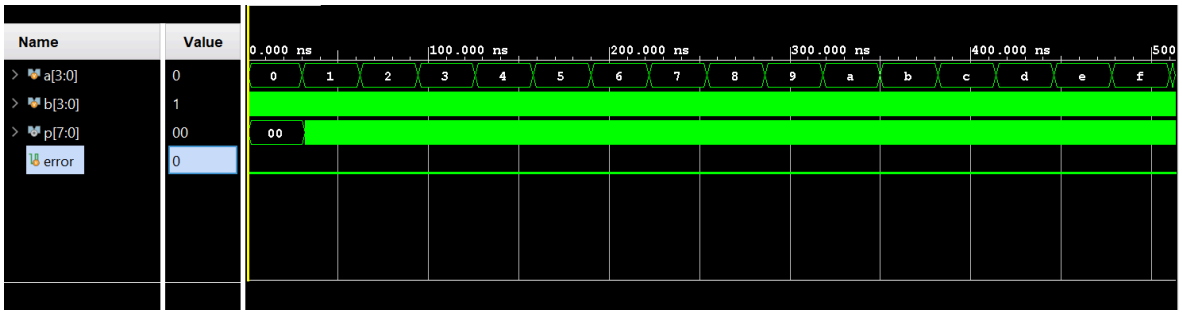
∆ **Figure 5-2** 4-bit * 4-bit Multiplier

As for testing, we iterate through all the possible cases and check if the product calculated by our module is correct.

```
module Multiplier_4bit_T();
//I/O signals
reg [4-1:0] a = 4'b0000, b = 4'b0001;
wire [8-1:0] p;
//test instance
Multiplier_4bit mul(.a(a), .b(b), .p(p));
reg error = 1'b0;
//run through all possible cases
initial begin

repeat (2**8) begin
    #1
    error = !(p===a*b);

    #1
    {a,b} = {a,b}+1'b1;
end

#1 $finish;
end
endmodule
```

∆ **Figure 5-3** Multiplier testbench

And here's the result of testing.



∆ **Figure 5-4** 4-bit * 4-bit Multiplier simulation waveform

5) Exhaustive Testing

Please see the comments, they're also exhaustive.

```
32  initial begin
33      //iterate through all 2^9 = 512 cases
34      //takes 512 * 5 = 2560ns
35      repeat (2**9) begin
36
37          //check output
38          //raise error if the cout or sum is incorrect
39          #1
40          error = ( {cout,sum} != a+b+cin )?1:0;
41
42          //set next input pattern
43          #4
44          {a,b,cin} = {a,b,cin} + 1'b1;
45      end
46      //set the done signal when done
47      done = 1'b1;
48
49      #1 error = 1'b0; //clear the last error (if any)
50      #4 done = 1'b0; //and finally, clear done signal
51  end
52
53  endmodule
```

∆ **Figure 6-1** Exhaustive Testing

FPGA for Decode & Execute

Design Description：

We can realize it by using the 2.(2) **Decode & Execute** module.

The 4 bits input are rs and rt, and the 3 bits input are “sel”, we can use switches as those inputs.

As for the output, it is divided into 2 parts：**AN signals and Segment signals**.

For AN signals, because we just want to use **the rightmost 7-segment display** to show the result, we only set **AN[0](i.e. PIN U2) to 0** to illuminate this bit. We set AN[3:1](i.e. PIN W4, V4, U4) to 1.

For Segment signals, I use a case to determine which number should we show, and use the list as in the below code to transform a hexadecimal number (output：rd) to an instruction for the 7 segments(0 -> illuminate, 1 -> not illuminate).

```
always @(*)
begin
    case(rd)
    4'b0000: LEDout = 7'b0000001; // "0"
    4'b0001: LEDout = 7'b1001111; // "1"
    4'b0010: LEDout = 7'b0010010; // "2"
    4'b0011: LEDout = 7'b0000110; // "3"
    4'b0100: LEDout = 7'b1001100; // "4"
    4'b0101: LEDout = 7'b0100100; // "5"
    4'b0110: LEDout = 7'b0100000; // "6"
    4'b0111: LEDout = 7'b0001111; // "7"
    4'b1000: LEDout = 7'b0000000; // "8"
    4'b1001: LEDout = 7'b0000100; // "9"
    4'b1010: LEDout = 7'b0001000; // "A"
    4'b1011: LEDout = 7'b1100000; // "b"
    4'b1100: LEDout = 7'b1110010; // "c"
    4'b1101: LEDout = 7'b1000010; // "d"
    4'b1110: LEDout = 7'b0110000; // "E"
    4'b1111: LEDout = 7'b0111000; // "F"
    default: LEDout = 7'b0000001; // "0"
    endcase
end
```

∆ **Figure 7-1** FPGA for Decode & Execute

What have we learned?

112062130 侯佑勳:
In this lab, I deeply understand how to realize the CLA and lots of clever equations to make the adder more effective. The most impressive things I have learned are 7 segment displays and how to compare two n-bits numbers in gate level. I hope to know more about the physics and electronic design of the 7 segment display! Thanks to my teammate, he developed a great testbench template for our future!

112062326 孔祥光:
In this lab we've covered some advanced gate-level design techniques, such as using universal gates (NAND, NOR or other custom made ones). We've also gained hands-on experience by building some basic combinational circuits and arithmetic units. Though the workload and exhaustive testing of modules was quite tiresome, this was overall fruitful.

Contribution

112062130 侯佑勳:
Decode & Execute
Carry Lookahead Adder
Decode & Execute on FPGA

112062326 孔祥光:
Ripple Carry Adder (+testbench)
Decode & Execute testbench
Carry Lookahead Adder testbench
Multiplier (+testbench)
Exhaustive Testing