Hardware Design

Lab 2 Report

Advanced Gate-Level Verilog

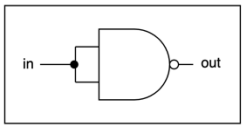
Team 01

112062122 謝佳晉　112062144 范升維

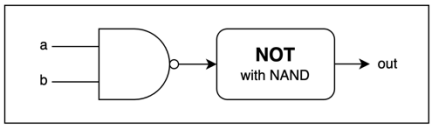
Table of Contents:

Basic Q1. Nand Implement

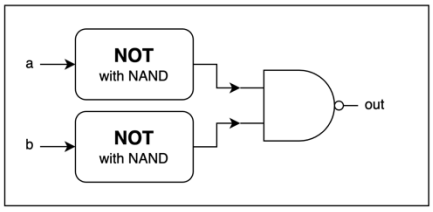
NOT_w_NAND



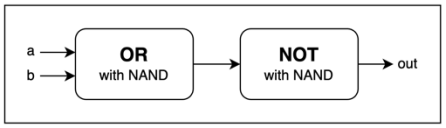
AND_w_NAND



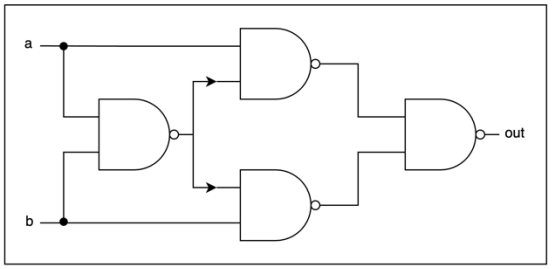
OR_w_NAND



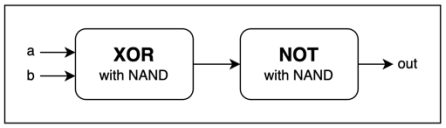
NOR_w_NAND



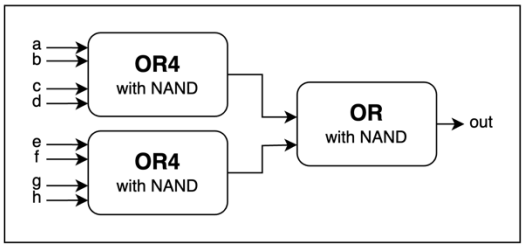
XOR_w_NAND



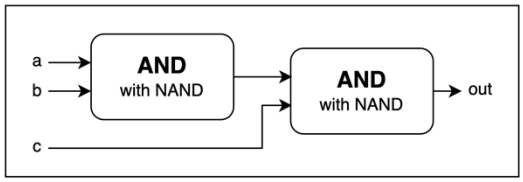
XNOR_w_NAND



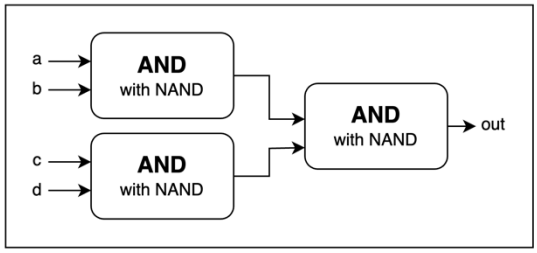
OR8_w_NAND



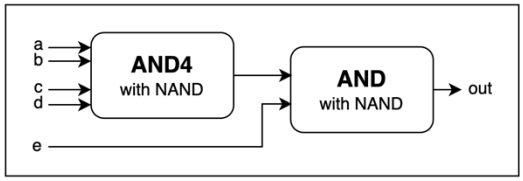
AND3_w_NAND



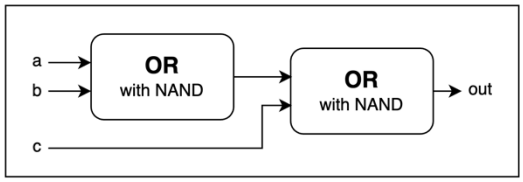
AND4_w_NAND



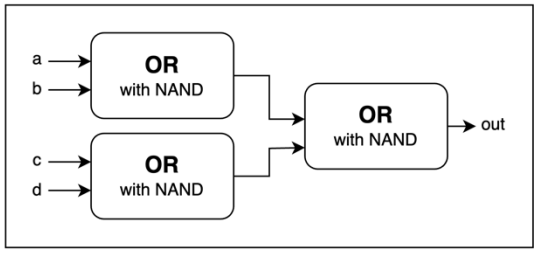
AND5_w_NAND



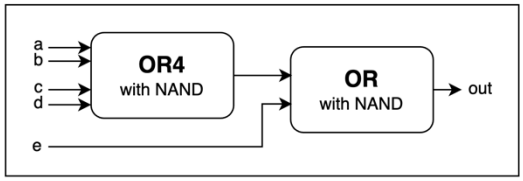
OR3_w_NAND

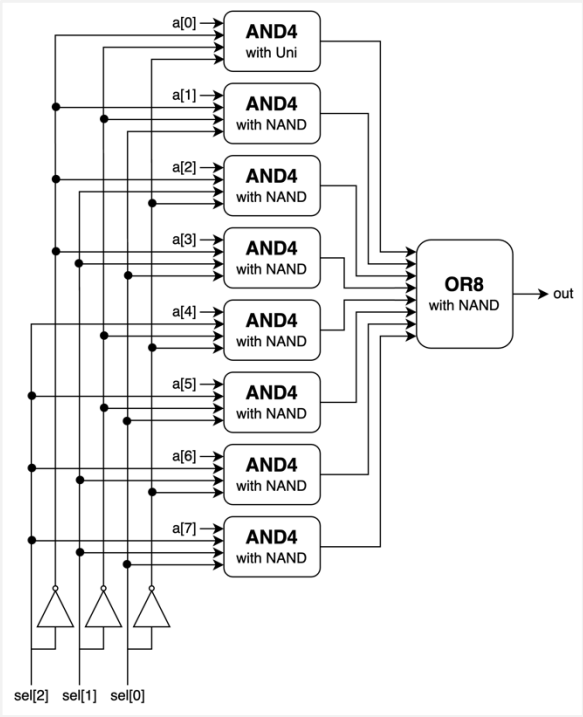


OR4_w_NAND

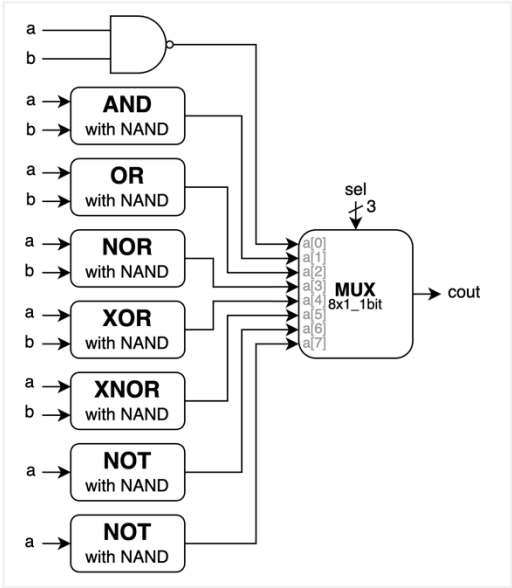


OR5_w_NAND





▲ Figure 0.1: 8x1_1bit MUX



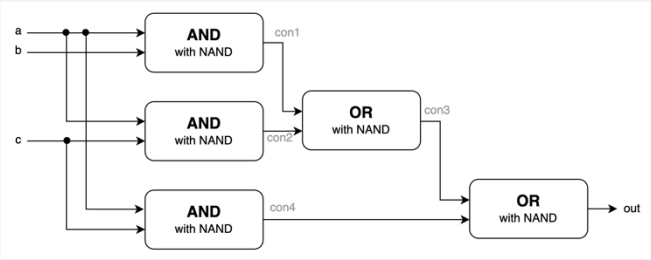
▲ Figure 0.2: NAND Implement

This problem requires us to utilize NAND gates, which is a universal gate, to construct other basic gates. On the right side of the previous page, we had 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.

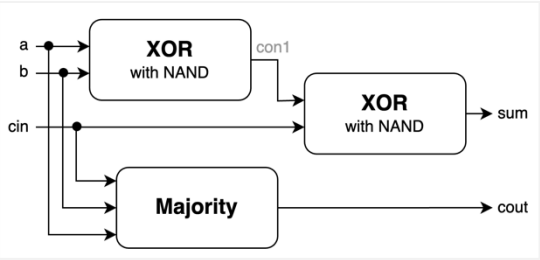
Basic Q3. Difference between Half Adder and Full Adder

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.

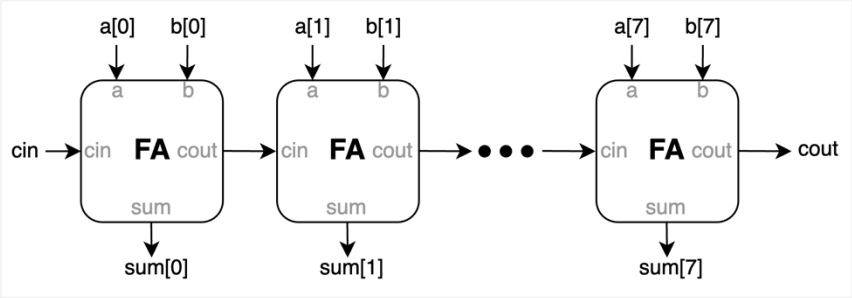
Advanced Q1. 8-bit Ripple Carry Adder (RCA)



▲ Figure 1.1: Majority



▲ Figure 1.2: Full Adder

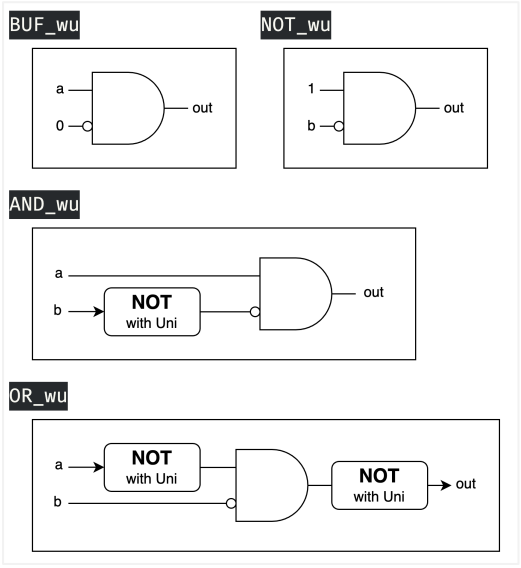


▲ Figure 1.3: Ripple Carry Adder

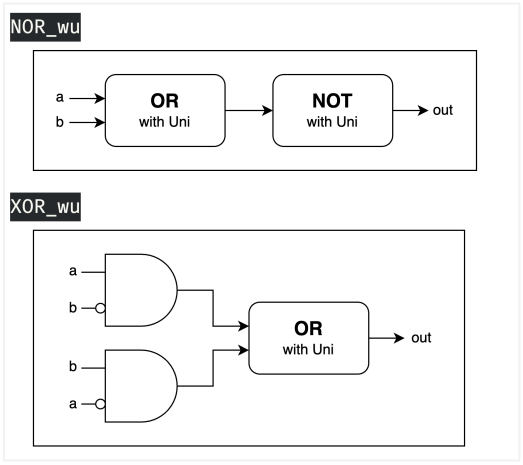
- 3 inputs: *a[7:0], b[7:0], cin*
- 2 outputs: *sum[7:0], cout*

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

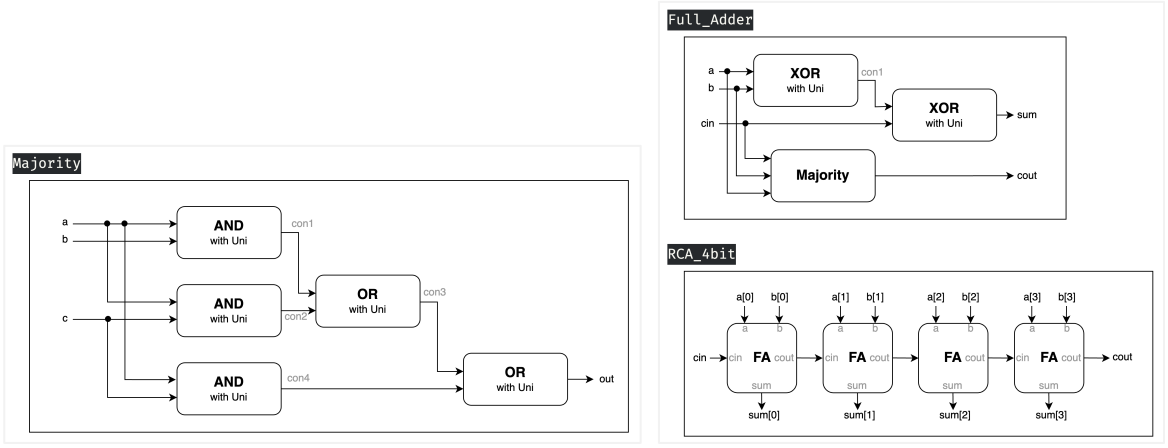


▲ Figure 2.1: Basic Gates



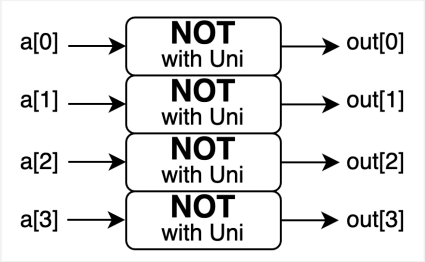
▲ Figure 2.2: Basic Gates

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.

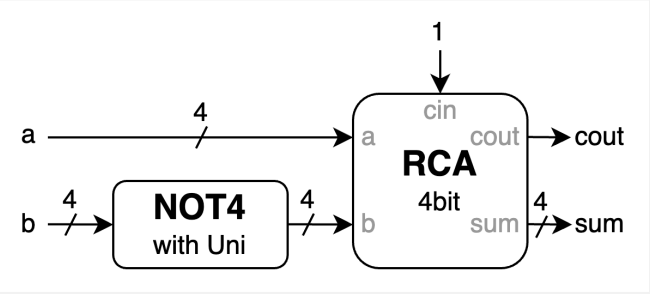


▲ Figure 2.3: 4bit Ripple Carry Adder

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.

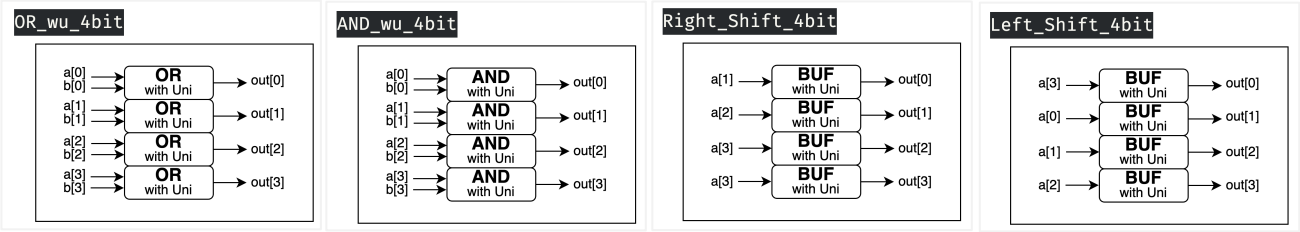


▲ Figure 2.4: 4bit NOT

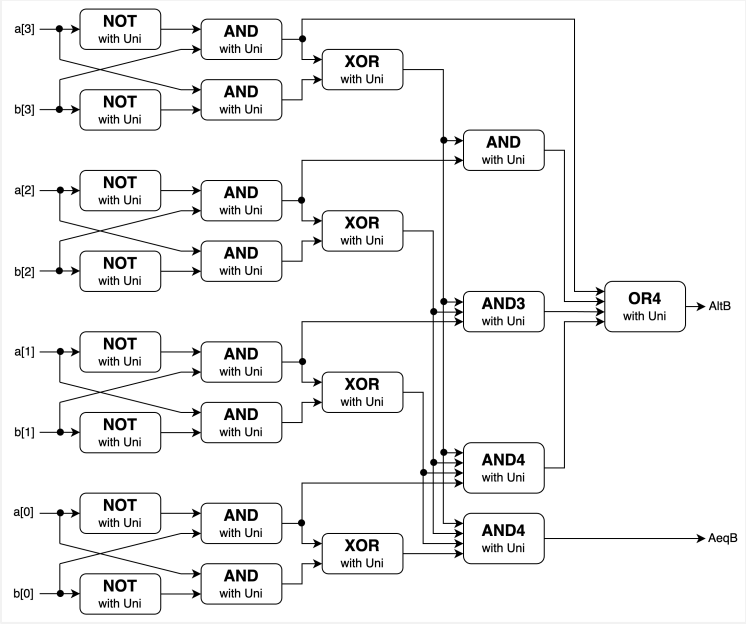


▲ Figure 2.5: 4bit Ripple Carry Adder - Minus

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.

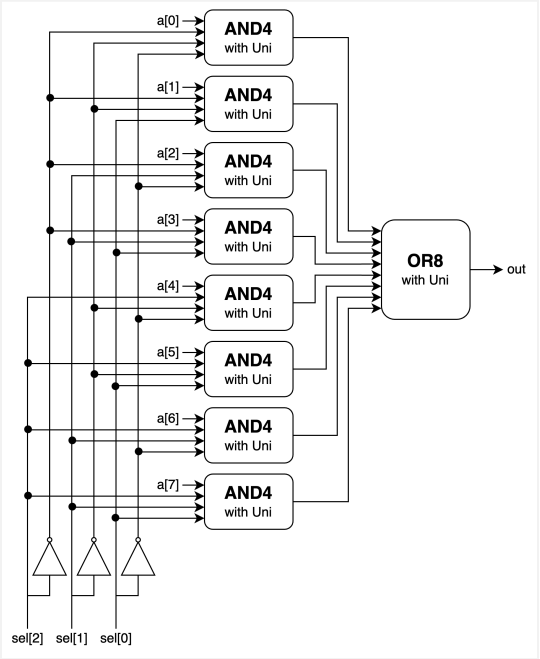


After that, we needed to build bitwise OR, bitwise AND, Arithmetic Right Shift and Circular Left Shift, so we accomplished it by easily using the method shown on the above figures.

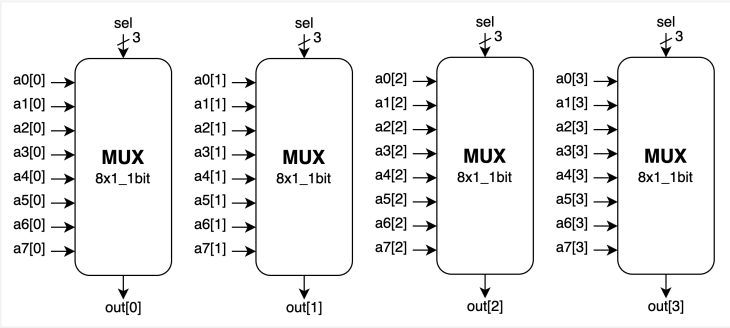


▲ Figure 2.5: Compare

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'b1, and the other is 1'b0), 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'b1.

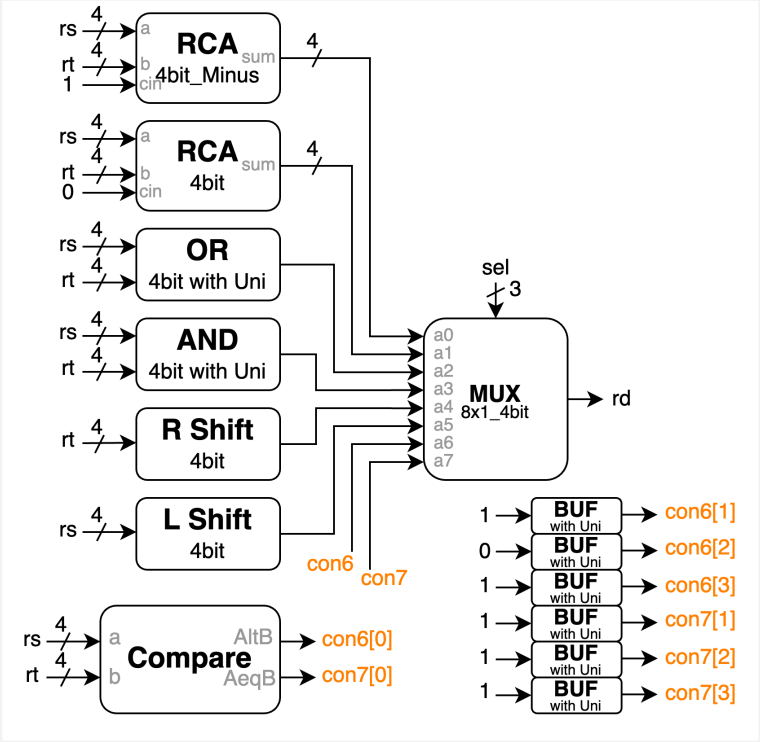


▲ Figure 2.6: 8x1 1bit MUX



▲ 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.



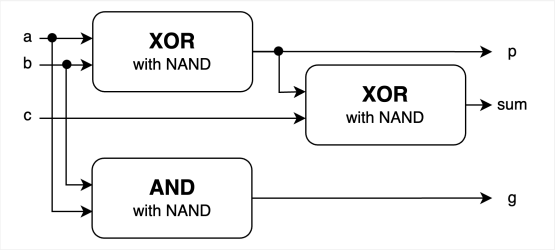
▲ Figure 2.8: Final Module(Decode and Execute)

- 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 Carry Look-ahead Adder (CLA)

The Ripple Carry Adder (RCA) requires each bit's addition operation to wait for the carry-out from the previous bit to be calculated as its carry-in before proceeding. Consequently, as the number of bits increases, delay becomes a significant issue. This limitation led to the development of the Carry Look-ahead Adder (CLA).

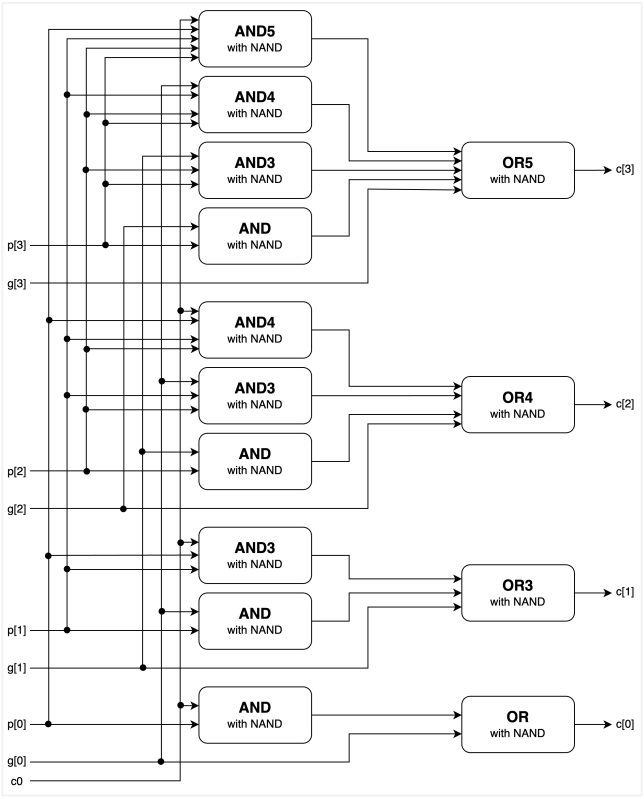


▲ Figure 3.1: 4bit CLA Generator

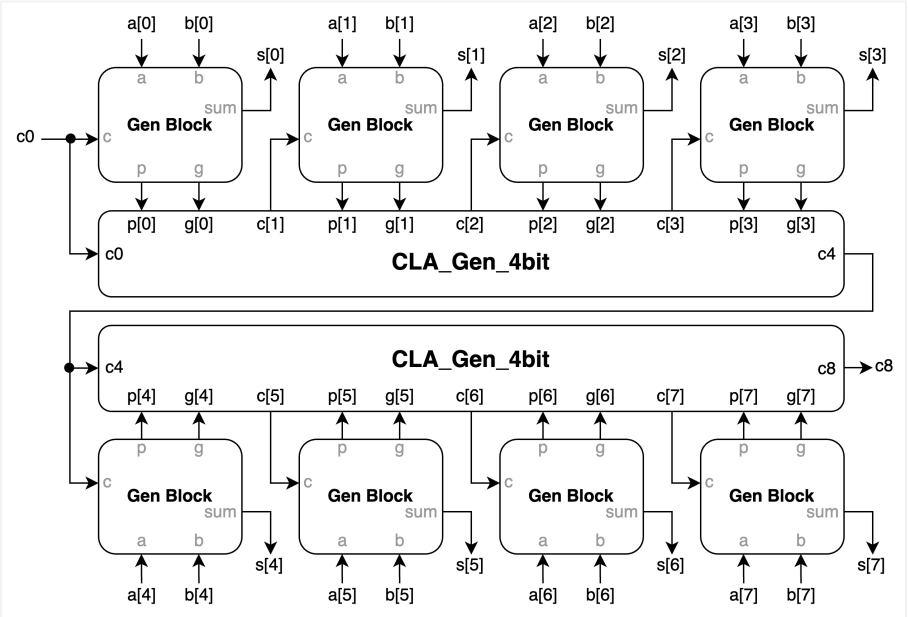
In this implementation, a 4-bit CLA utilizes a 4-bit CLA Generator to precompute all carry bits ($c1$ through $c4$) simultaneously. This approach allows the addition operation to be executed concurrently for all four bits. The CLA Generator employs a circuit we refer to as the Gen Block (as illustrated in Figure 3.1 above). Within this block, the input bits a and b undergo an **AND** operation to produce g (**generate**), indicating whether both numbers are 1'b1 and thus necessitate a carry to the next bit. Additionally, a and b are **XOR**ed to yield p (**propagate**). When p is 1, it signifies that if the previous bit's g is 1, or if the previous bit's p is 1 and the bit before that has a g of 1 (and so on recursively), then this bit will also receive a carry.

The CLA Generator utilizes a Gen Block for each bit position to generate the corresponding p and g values. Subsequently, it implements the following Boolean operations, the concept of which have been elucidated the previous paragraph, in a circuit diagram (illustrated in Figure 3.2 on the following page):

$$c[1] = g[0] + p[0]\cdot c0$$
$$c[2] = g[1] + p[1]\cdot g[0] + p[1]\cdot p[0]\cdot c0$$
$$c[3] = g[2] + p[2]\cdot g[1] + p[2]\cdot p[1]\cdot g[0] + p[2]\cdot p[1]\cdot p[0]\cdot c0$$
$$c[4] = g[3] + p[3]\cdot g[2] + p[3]\cdot p[2]\cdot g[1] + p[3]\cdot p[2]\cdot p[1]\cdot g[0] + p[3]\cdot p[2]\cdot p[1]\cdot p[0]\cdot c0$$



▲ Figure 3.2: 4bit CLA Generator

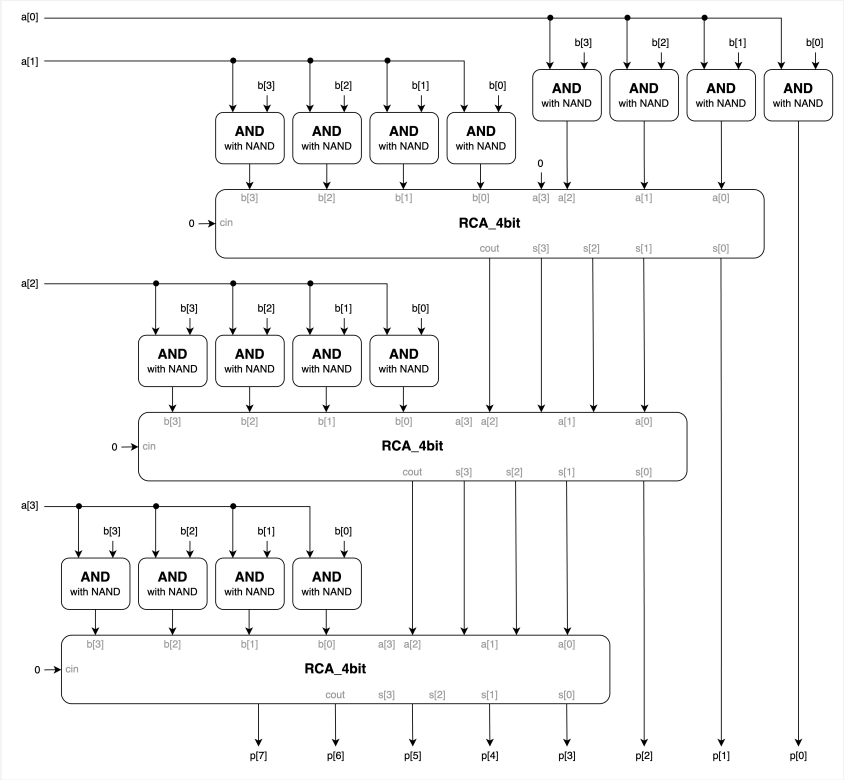


▲ Figure 3.3: 8bit Carry Look-ahead Adder (CLA)

- 3 inputs: *a[7:0], b[7:0], c0*
- 2 outputs: *s[7:0], c8*

Having developed the 4-bit CLA, we ultimately synthesized an 8-bit CLA by combining two 4-bit CLAs. This configuration, as illustrated in Figure 3.3 above, incorporates eight Gen Blocks along with two 4-bit CLAs. We adhered strictly to the specified diagram, with one notable exception: we did not implement an additional CLA Generator that takes eight p and eight g values as inputs and produces a 2-bit output. The rationale for this omission is that such a component's primary function would be to generate *c[8]* and *c[4]* simultaneously. However, the remaining outputs (*s[4]* through *s[7]*) must still await the computation of *c[4]*, which serves as an input to the lower 4-bit CLA Generator. Moreover, the process of generating *c[4]* using *g[0]* through *g[3]* and *p[0]* through *p[3]* as inputs is identical to the method employed by the upper 4-bit CLA Generator. Consequently, we deemed the inclusion of this additional component to be of limited necessity and opted for a more streamlined design by omitting it.

Advanced Q4. 4-bit Multiplier



▲ Figure 4.1: 4bit Multiplier

- 2 inputs: *a[3:0], b[3:0]*
- 1 output: *p[7:0]*

This problem differs from previous ones, where numerous modules needed to be constructed to achieve completion. In comparison, this task is considerably more concise, utilizing only AND gates and the previously constructed 4-bit RCA.

The principle of this 4-bit multiplier is actually quite simple and easy to understand, much like how we perform multiplication, but in binary form. First, the last digit of the multiplier is multiplied by each digit of the multiplicand, and the results are recorded in the first row. Then, the second digit of the multiplier is multiplied by each digit of the multiplicand, and the results are recorded in the second row, shifted one position to the left. This process is repeated, and the rows are then added together to obtain the final product of the multiplication.

Advanced Q5. Exhaustive testbench design

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

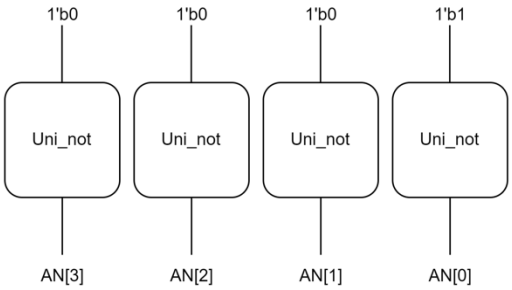
▲ Figure 5.1: Our Testbench Design

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. Initially, we conducted tests with the carry-in (*cin*) set to zero, followed by tests with *cin* set to one. Within each of both scenarios, we systematically examined all possible combinations of a and b additions, encompassing a total of 256 cases (16 times 16).

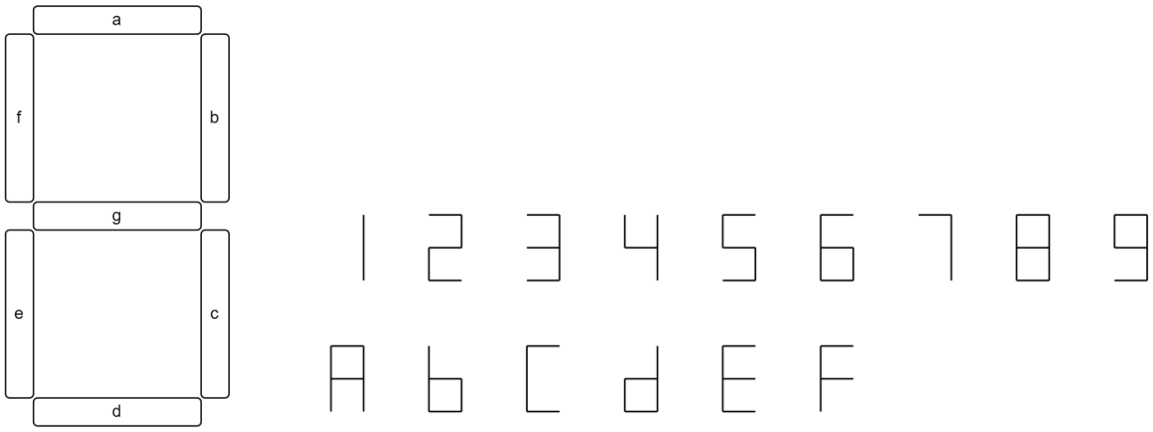
When the sum produced by the 4-bit RCA isn't equal to the actual value, the error will be set to 1'b1 one nanosecond later and last for five nanoseconds. After the nested loop run out, we set done to 1'b1 in order to show that all of the combinations have been tested. Lastly, we reset done to 1'b0 after five nanoseconds.

FPGA: 7-Segment Display Control

In order to light only 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

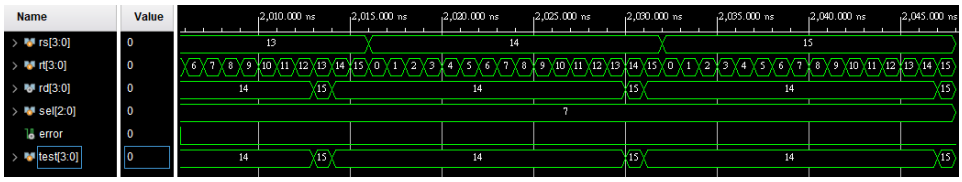
A. 8-bit Ripple Carry Adder (RCA)

```
module Ripple_Carry_Adder_t();
    reg [8-1:0] a = 8'b0;
    reg [8-1:0] b = 8'b0;
    reg cin = 1'b0;

    wire cout;
    wire [8-1:0] sum;

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

    always #1 cin = ~cin;
    initial begin
        repeat(2**8-1) 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. In the inner loop, we accumulate b to its extreme value. In the outer loop, we accumulate a to its extreme value and reset b to zero in order to properly initialize the next inner loop. Outside of the initial block, we use always block to change cin value periodically. In this way, we can monitor the change of output much easier.

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 →test 'SUB' → reset → test 'ADD' → reset → test 'Bitwise OR' → reset → test 'Bitwise AND' → reset → test 'Right Shift' → reset → test 'Left Shift' → reset → test 'Compare LT' → reset → test 'Compare EQ' → finish.

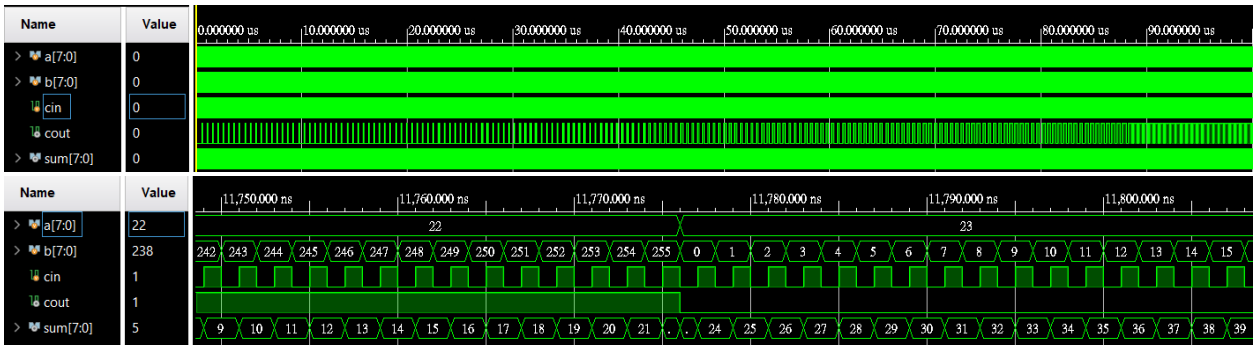
Whenever we test an operation out, we update the value of *sel[2:0]* to test the next operation. Besides, we have *test[3:0]* be the answer to the operation output.

To test 'SUB', 'ADD', 'Bitwise OR', 'Bitwise AND', 'Compare LT', and 'Compare EQ', we use two level nested loops to run all the 4-bit number combinations. If the output is not equal to *test[3:0]*, the error will be set to 1'b1.

To test 'Right Shift' and 'Left Shift', we still use the two levels nested loops because we want to test whether the outputs change with the unwanted input. Same, If the output is not equal to *test[3:0]*, the *error* will be set to 1'b1.

C. 8-bit Carry Look-ahead Adder (CLA)

```
module Carry_Look_Ahead_Adder_8bit_t();
    reg [8-1:0] a = 8'b0;
    reg [8-1:0] b = 8'b0;
    reg cin = 1'b0;

    wire cout;
    wire [8-1:0] sum;

    Carry_Look_Ahead_Adder_8bit CLA8bit(a, b, cin, sum, cout);

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



Given the near-identical structure of this testbench to that of Q1, further elaboration is deemed unnecessary.

D. 4-bit Multiplier

```
module Multiplier_4bit_t();
    reg [4-1:0] a = 4'b0;
    reg [4-1:0] b = 4'b0;
    wire [7:0] p;
    wire error;
    reg [7:0] test;

    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
```

In order to test out all the possible output, we use two level nested loops to run all the number combinations and use signal *error* to detect whether the output is correct. When the value of *p[7:0]* is not equal to *test[7:0]*, the *error* will be set to 1'b1 to show that the output is wrong.

What we have learned from Lab2?

This lab exercise proved to be exceptionally time-consuming. It required us to construct various basic gates using NAND gates and a specialized universal gate, followed by the implementation of half adders and full adders. We then progressed to creating RCA and CLA. The final step of this process involved utilizing these adders to construct a multiplier through iterative combination. In addition, Question 5 introduced us to an efficient method of writing testbenches.

While the majority of these concepts were covered in the previous semester's Logic Design course, the experience of implementing them personally and employing components such as multiplexers to observe various outcomes was particularly enlightening. The final FPGA implementation, which allowed us to observe physical numerical displays, was especially gratifying.

However, the process of drawing K maps (Karnaugh maps) and deriving their corresponding Boolean expressions proved to be both intricate and time-intensive. Undoubtedly, the most challenging aspect was crafting circuit diagrams for all modules. Nevertheless, this laboratory exercise significantly enhanced our proficiency in gate-level operations. Inevitably, it also honed our skills in circuit diagram construction.

Contributions

謝佳晉：

wrote Verilog modules: Q1, Q2, Q4
wrote testbenches: Q1, Q2, Q4, Q5
performed simulation: Q1, Q2, Q4, Q5
drew diagram: basic Q1, Q2
wrote report: basic Q3, Q5, tbQ1, tbQ2, tbQ4
made FPGA demonstration

范升維：

wrote Verilog modules: Q1, Q2, Q3, Q4
wrote testbench: Q3
performed simulation: Q1, Q2, Q3, Q4 simulation on Vivado
drew diagram: ALL (basic Q1, Q1, Q2, Q3, Q4)
wrote report: basic Q1, basic Q3, Q1, Q2, Q3, Q4, tbQ3, what we learned
organized whole report

	b-Q1	b-Q3	Q1	Q2	Q3	Q4	Q5	FPGA	What we learned
wrote Verilog modules									
wrote testbenches									
performed simulation									
drew diagram									
wrote report									
wrote report tb									

Both
謝佳晉
范升維