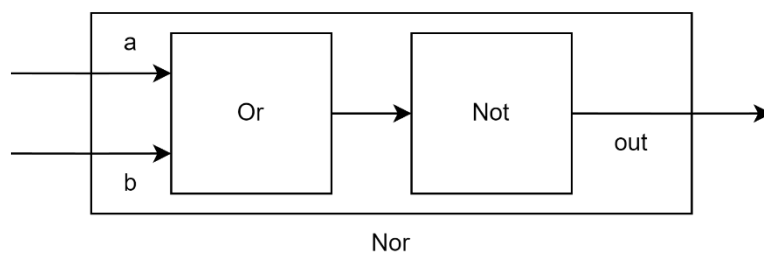
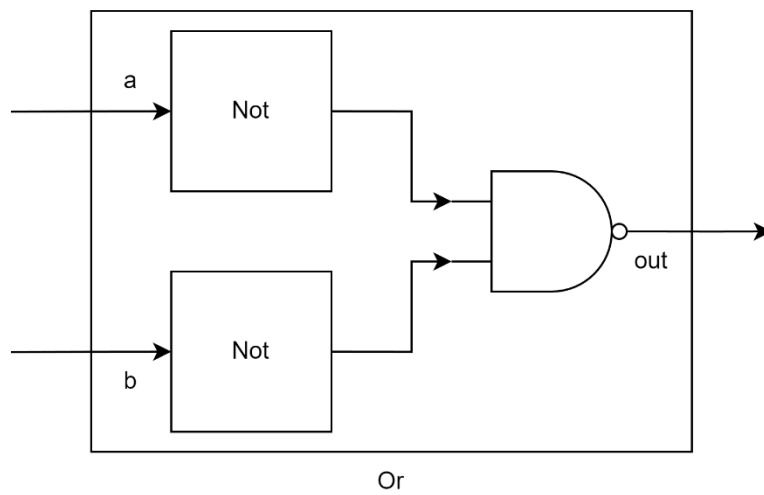
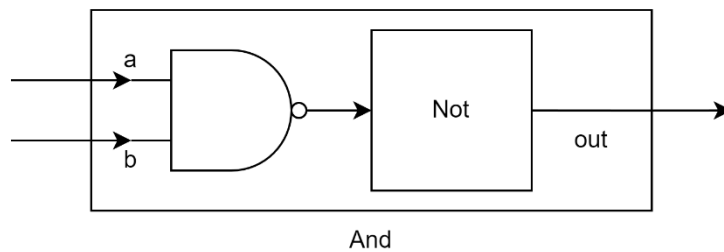
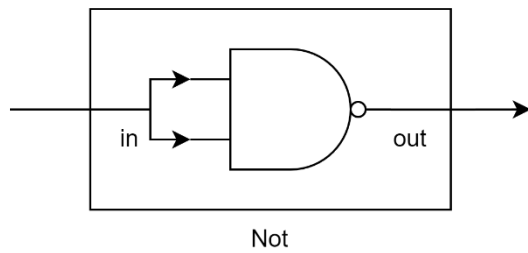


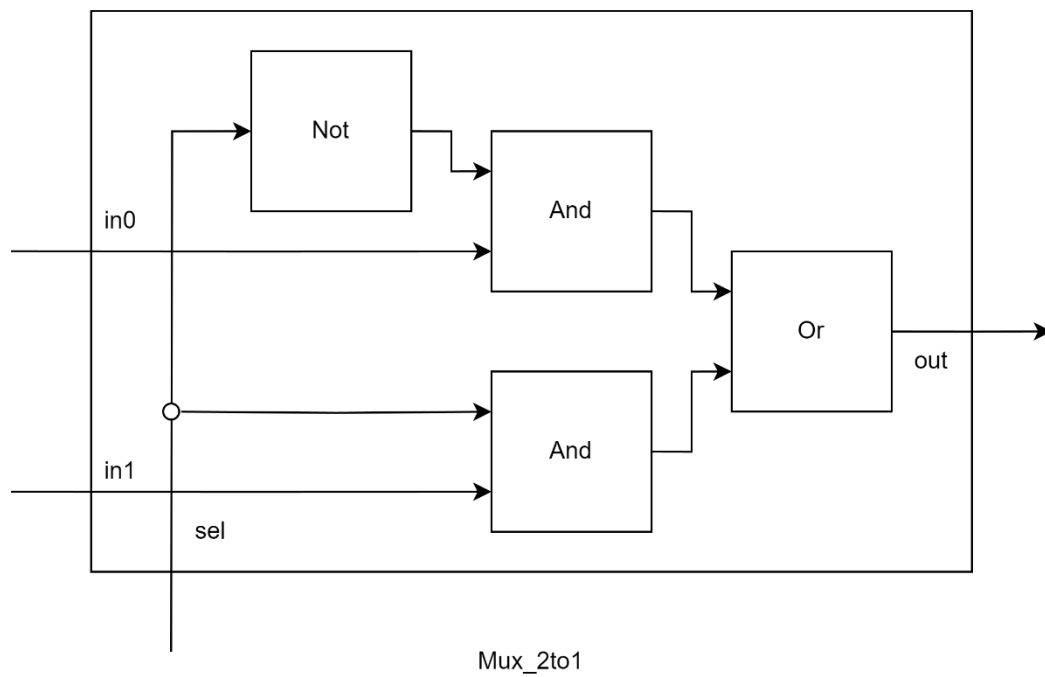
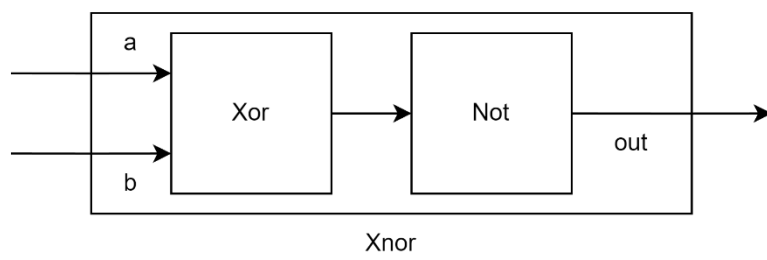
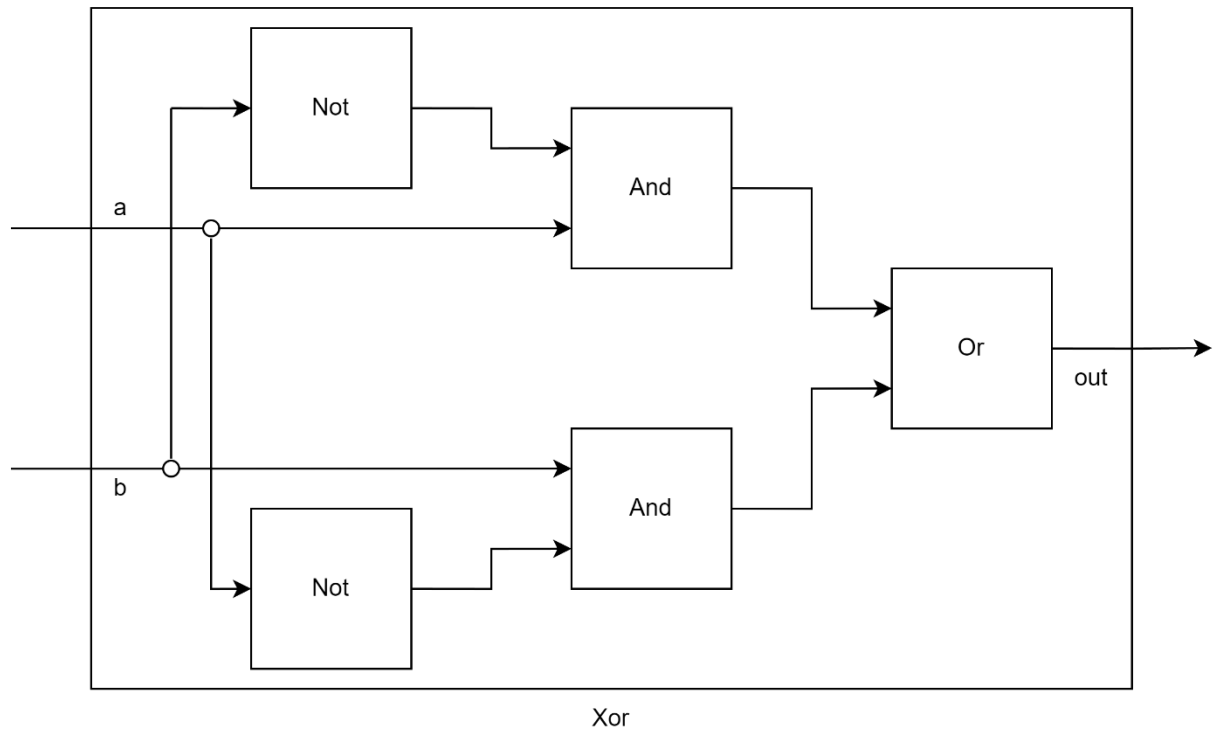
Hardware Design and Lab: Lab1

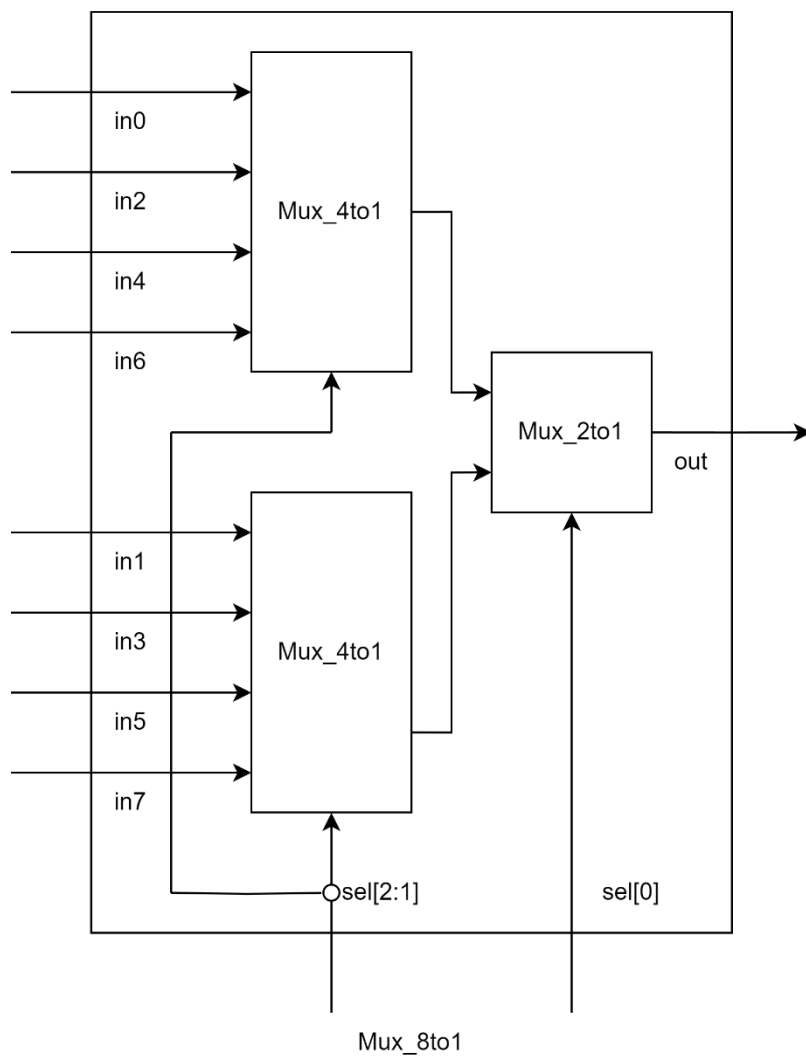
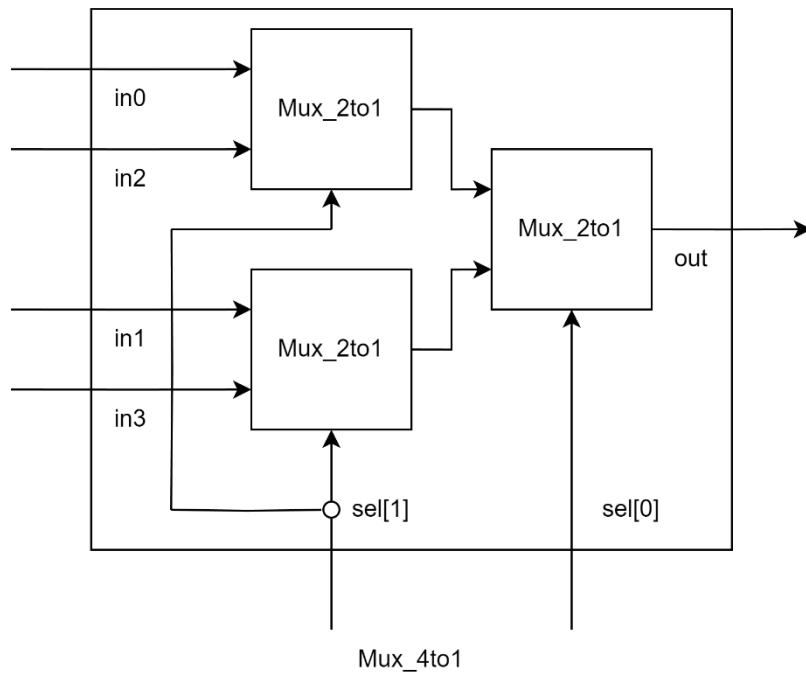
111060013 EECS 26' 劉祐廷

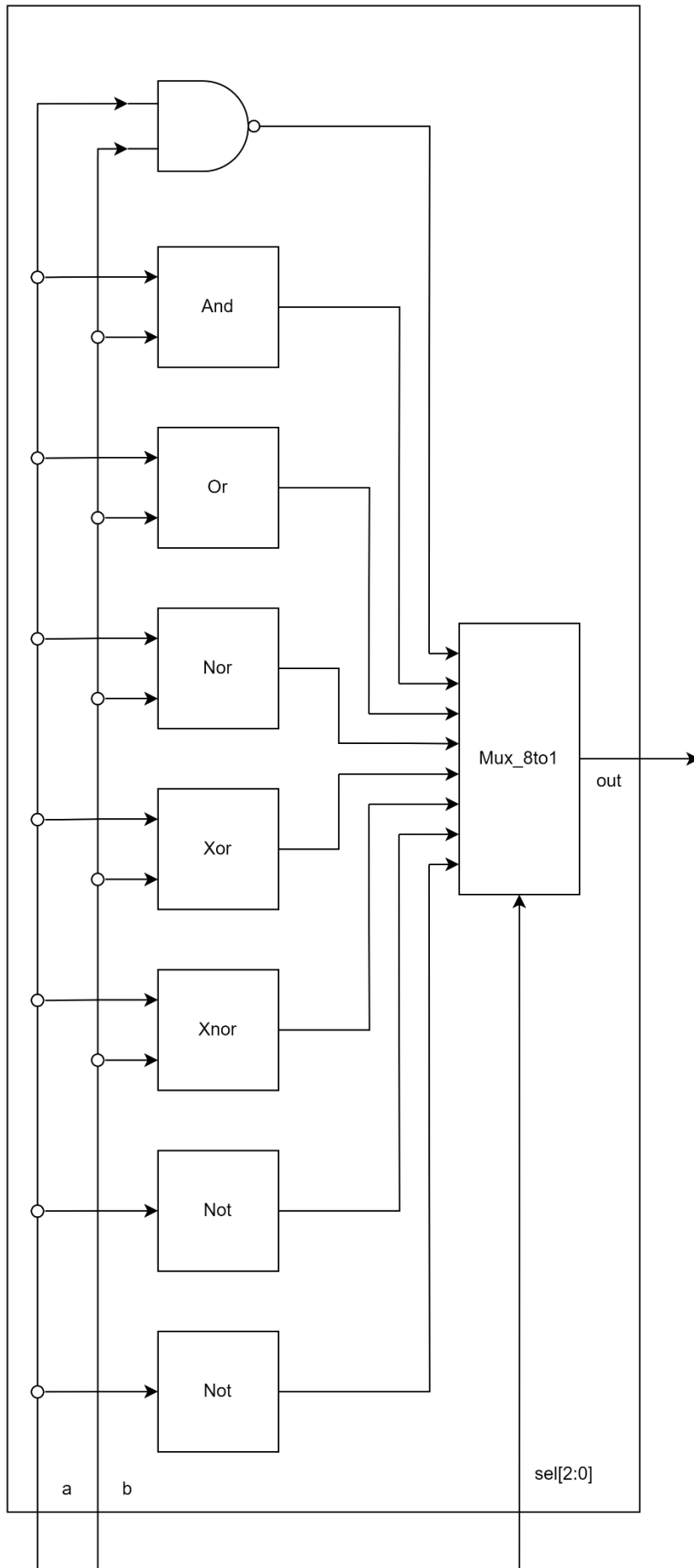
1. Basic

A. Basic Question 1: Block Diagram







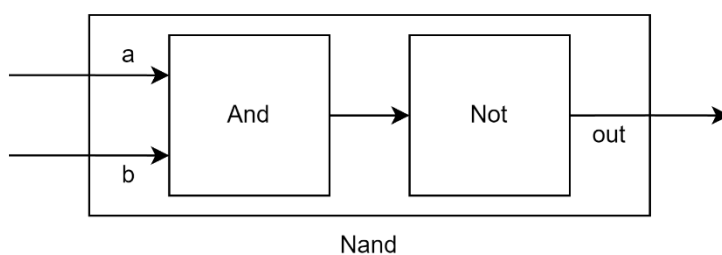
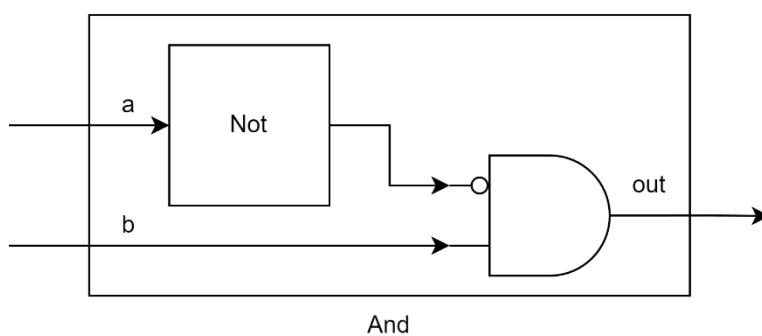
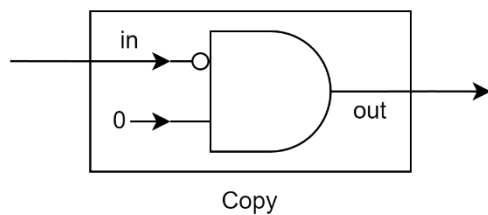
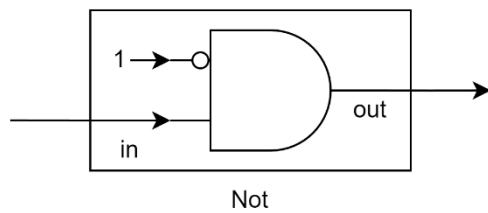


B. Basic Question 3: The Difference Between Full Adder and Half Adder

The most significant difference between them is that a half adder can only deal with the situation without carry in; however, a full adder can handle the situation with carry in.

2. Advanced: Decode and execute

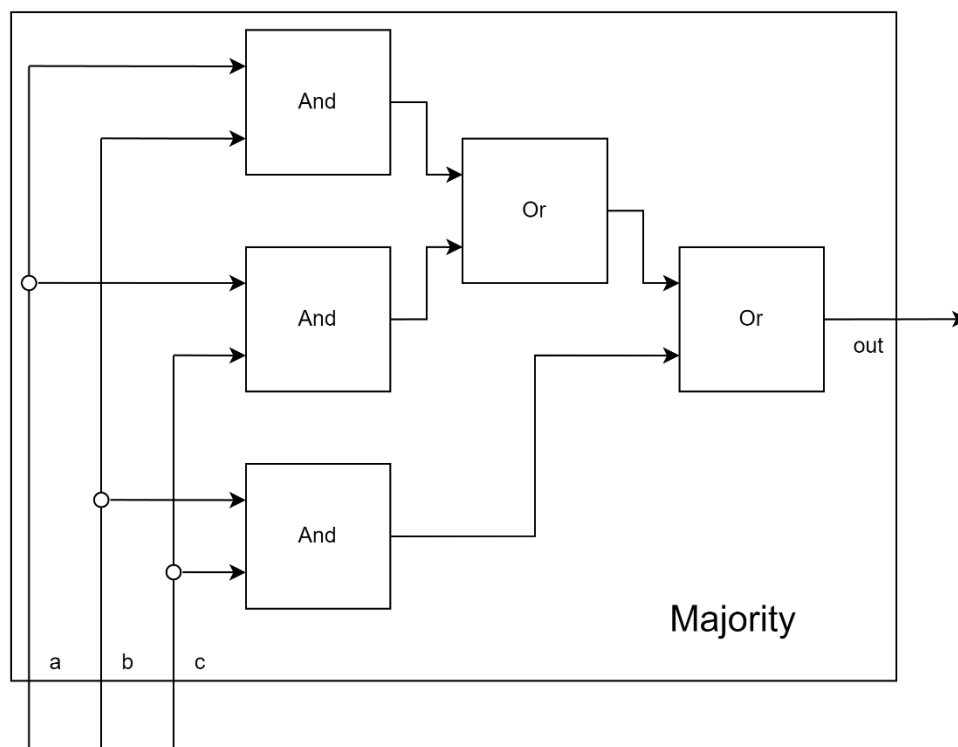
A. Block Diagram: Basic Modules

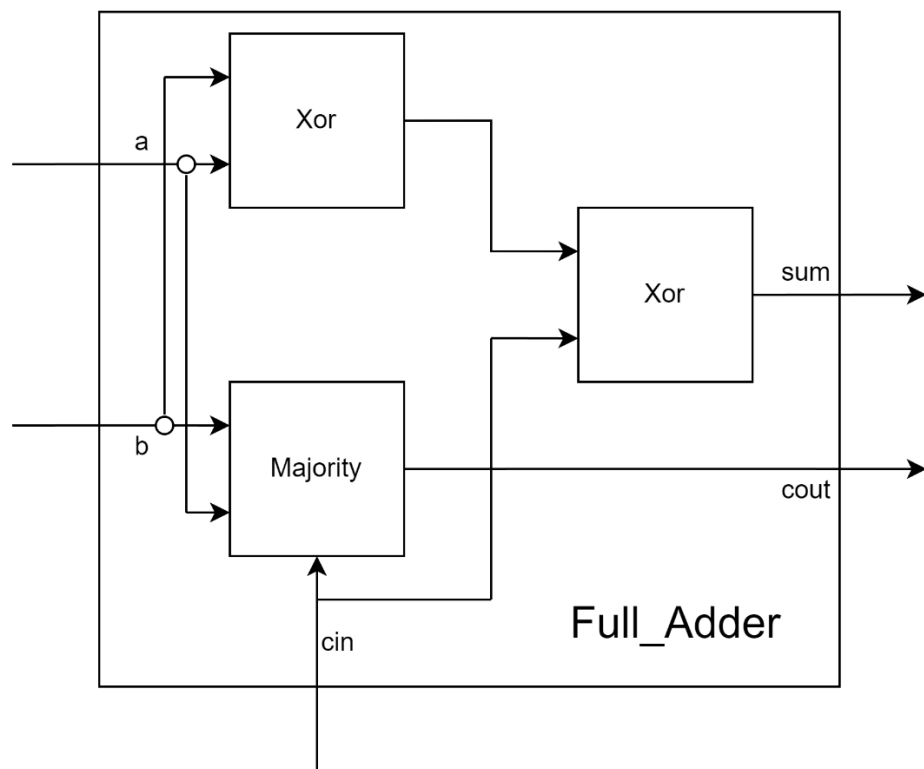
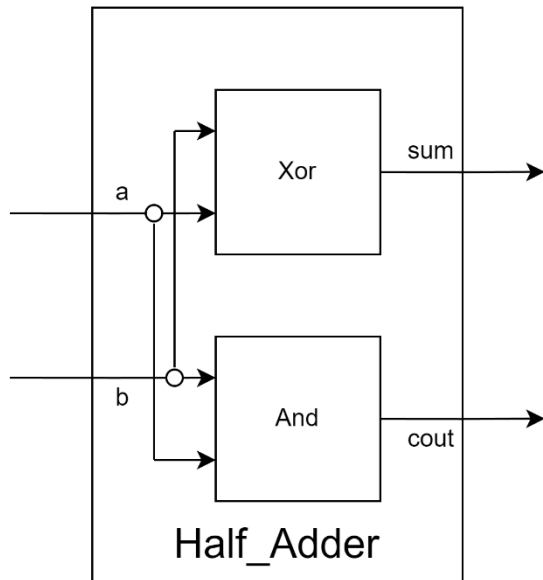


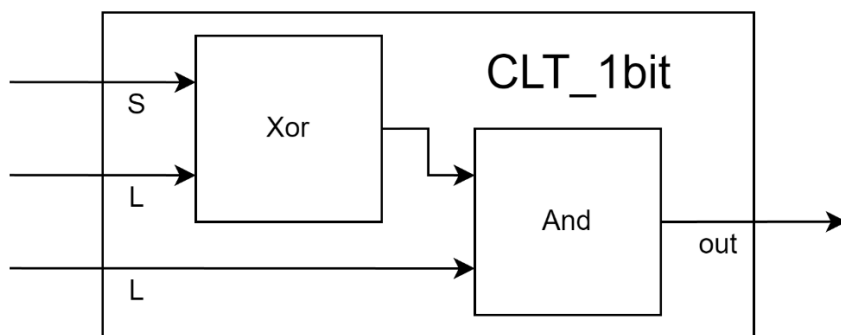
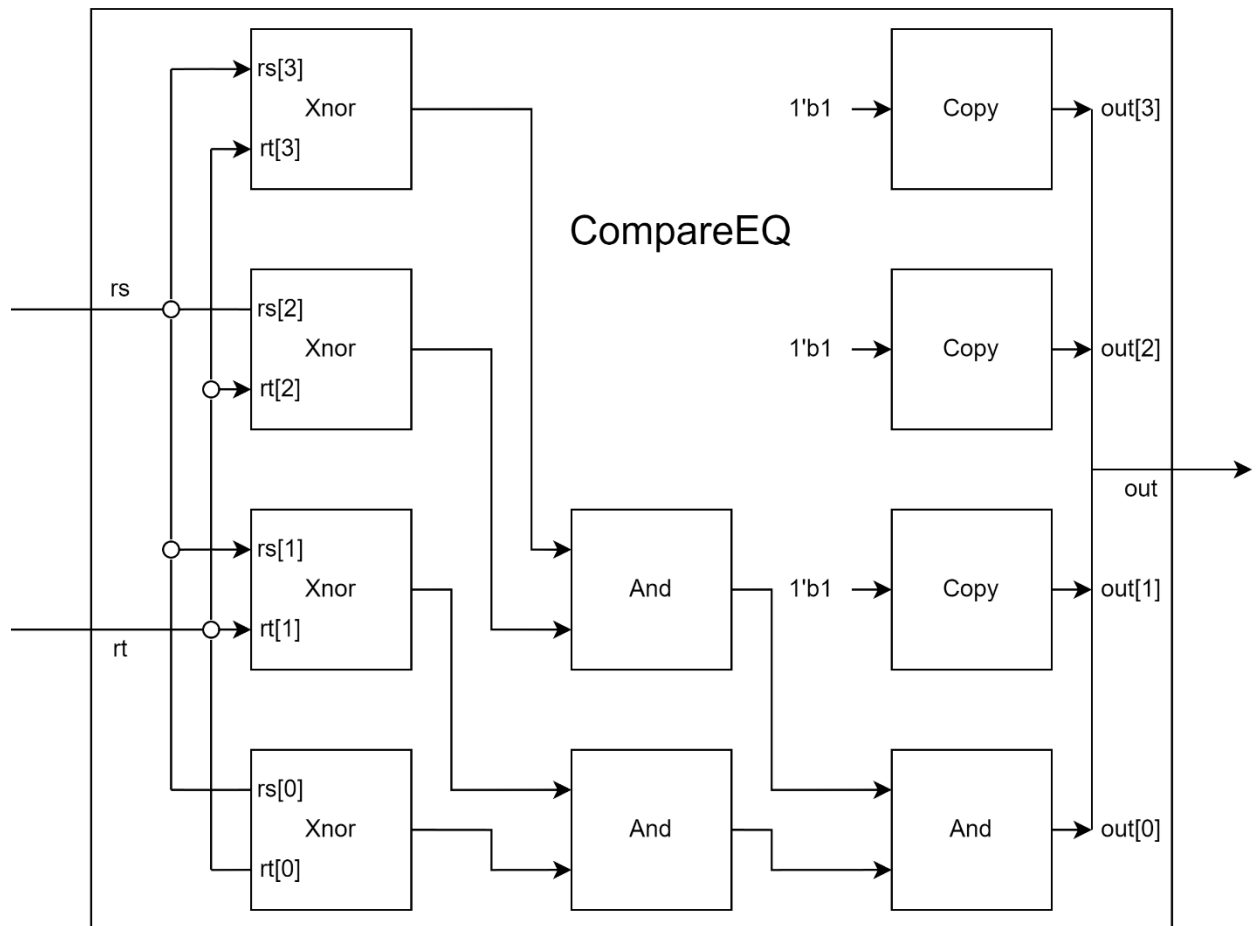
B. Explanation: Basic Modules

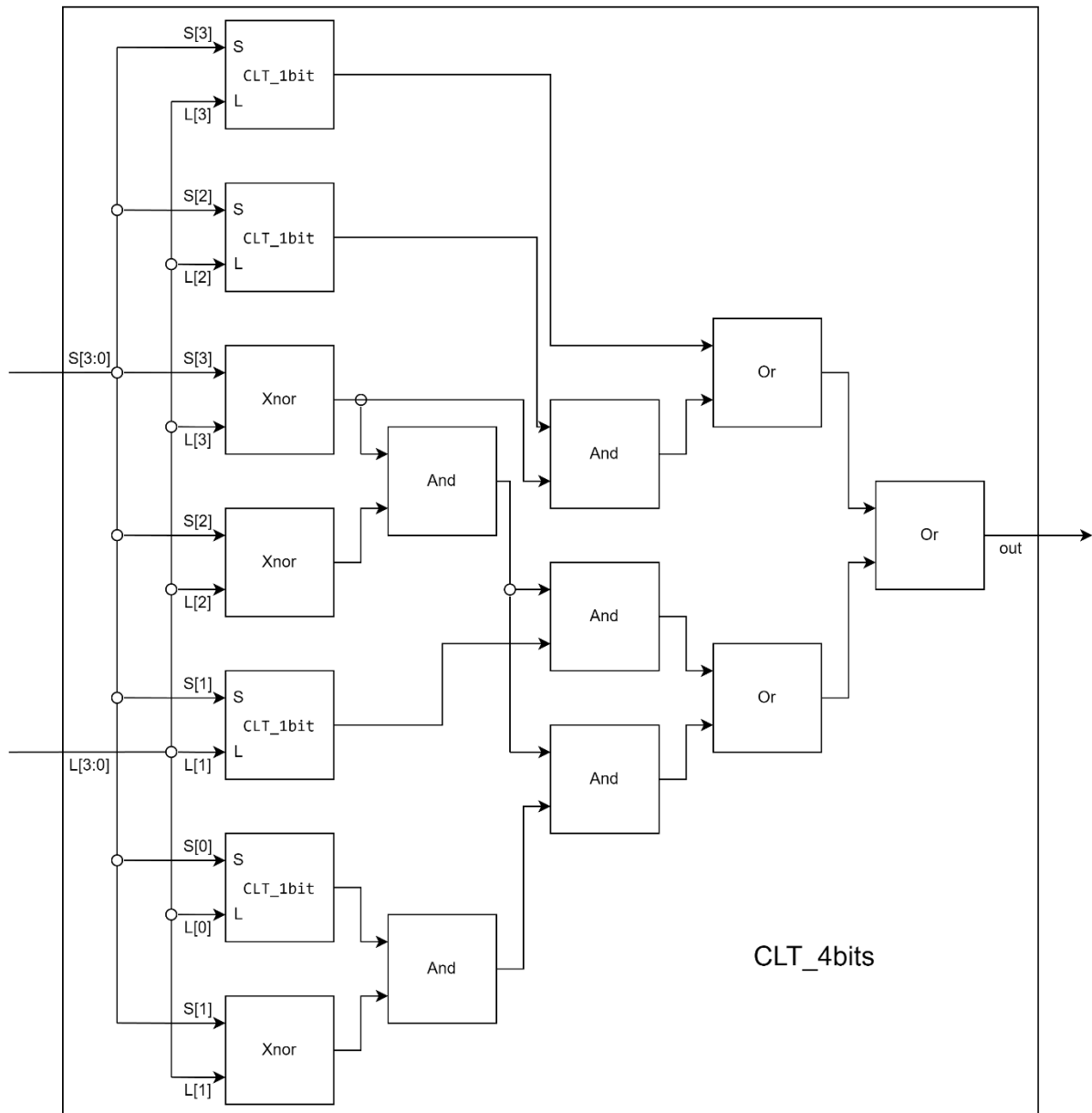
First of all, I drew the truth table of **Universal Gate** to design my own **Not** module and **Copy** module. And then I combined a **Not** module and an **Universal Gate** to create **And** module. After that, I make a **Nand** module with an **And** module and a **Not** module. The reason why I design **Nand** module before designing other modules (ex: **Or**, **Xor**) is that I have designed several modules consist with only **nand gates** in **Basic Question 1**. By designing out the **Nand** module first, I could design other modules more easily by only replacing all nand gates with **Nand** modules.

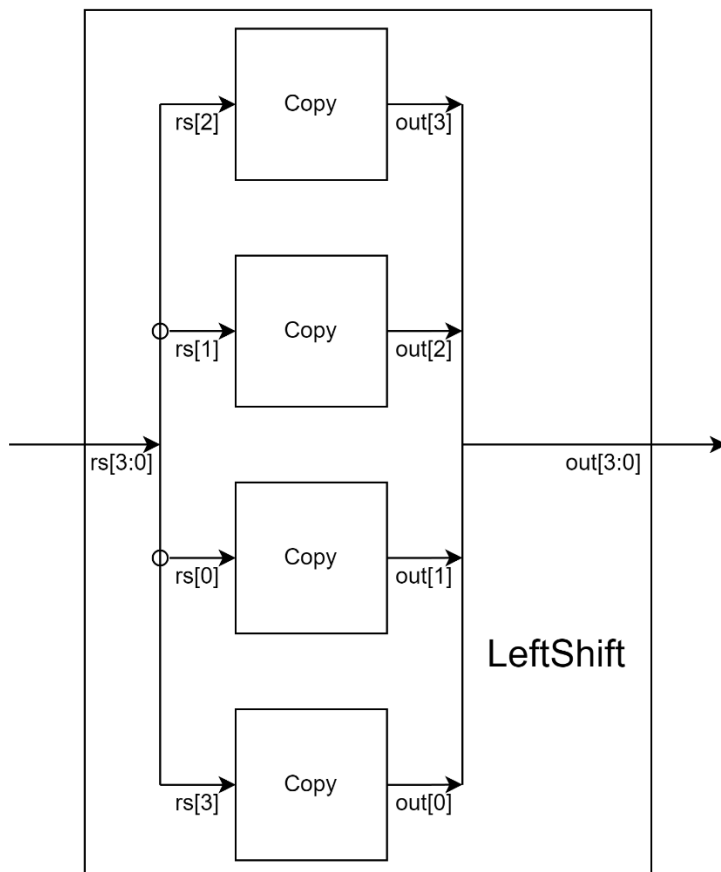
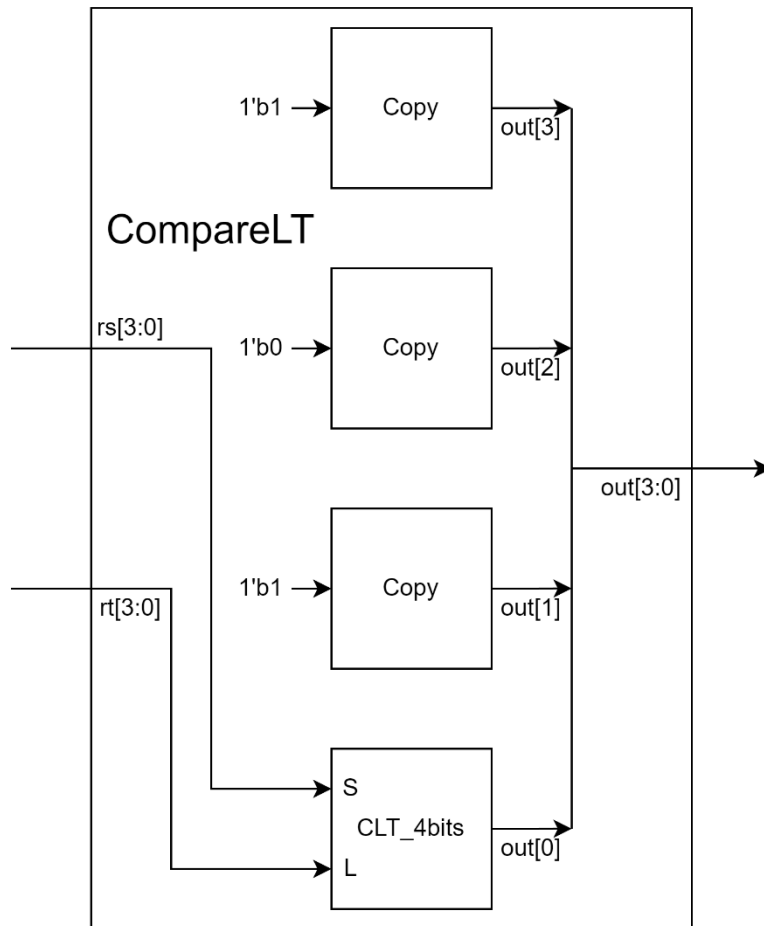
C. Block Diagram: Advanced Modules

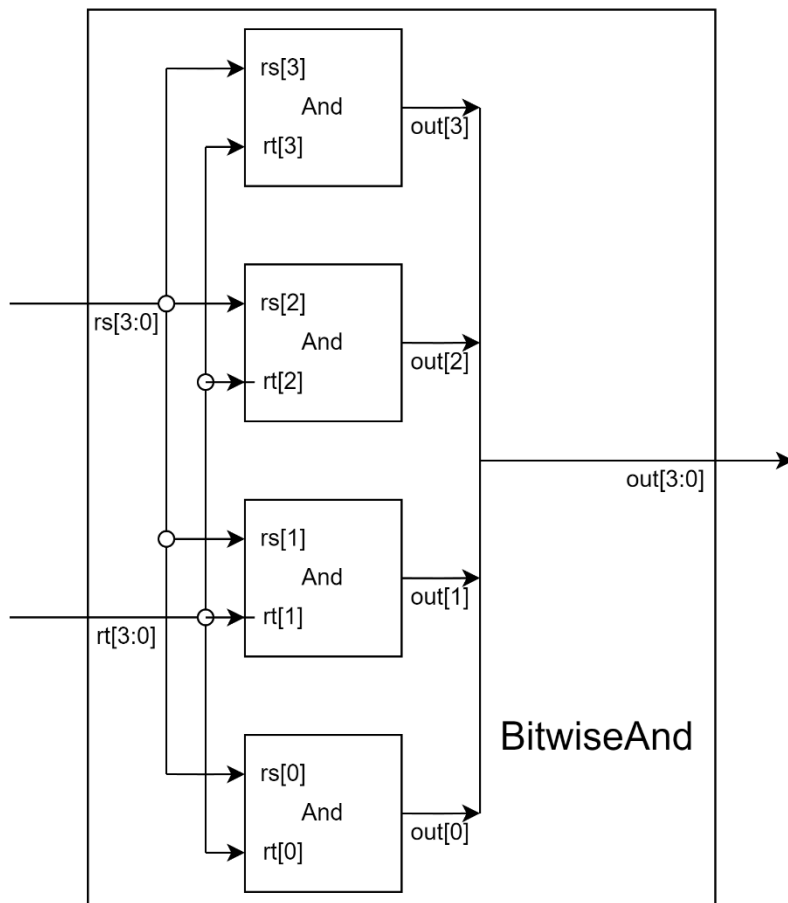
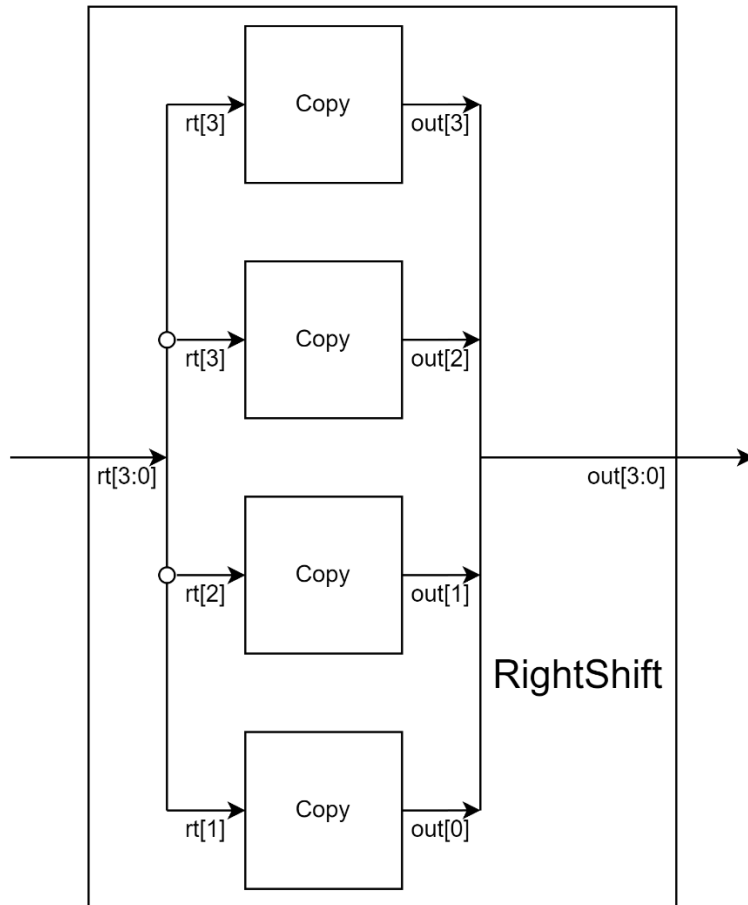


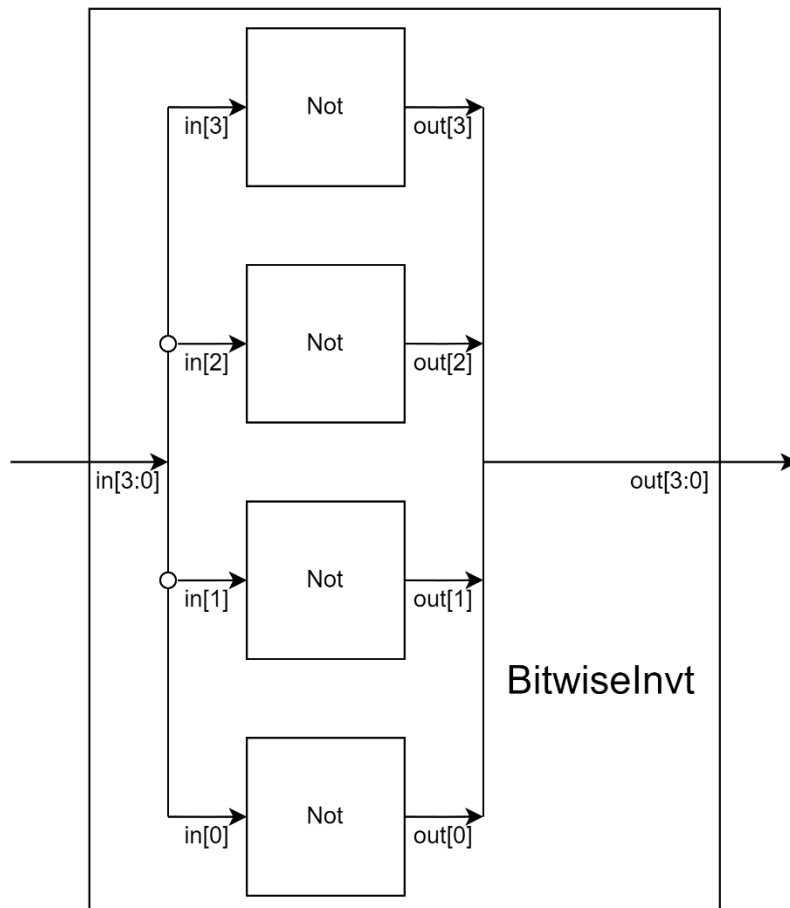
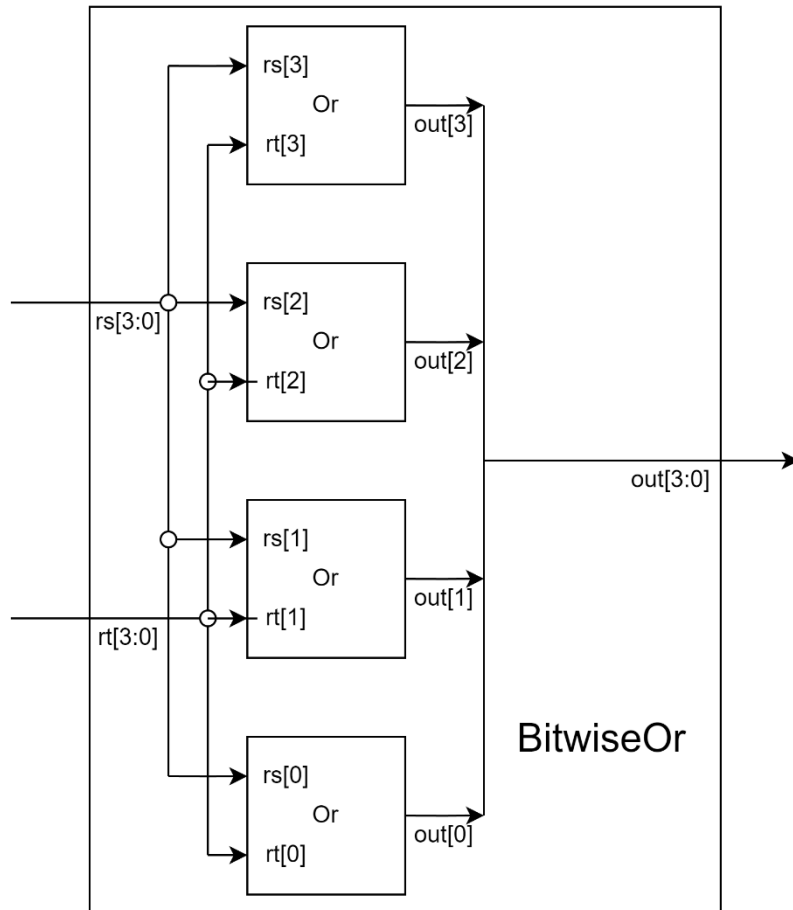


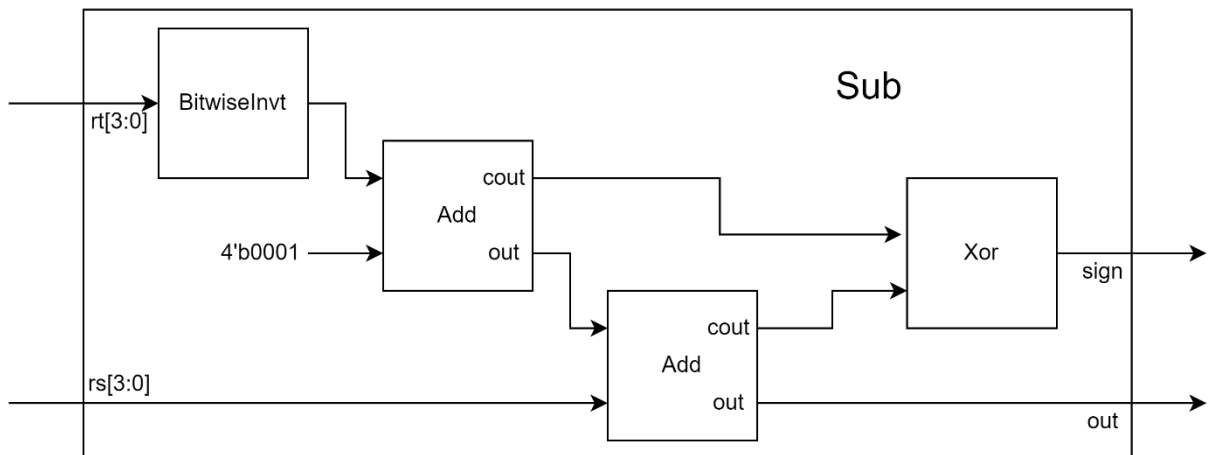
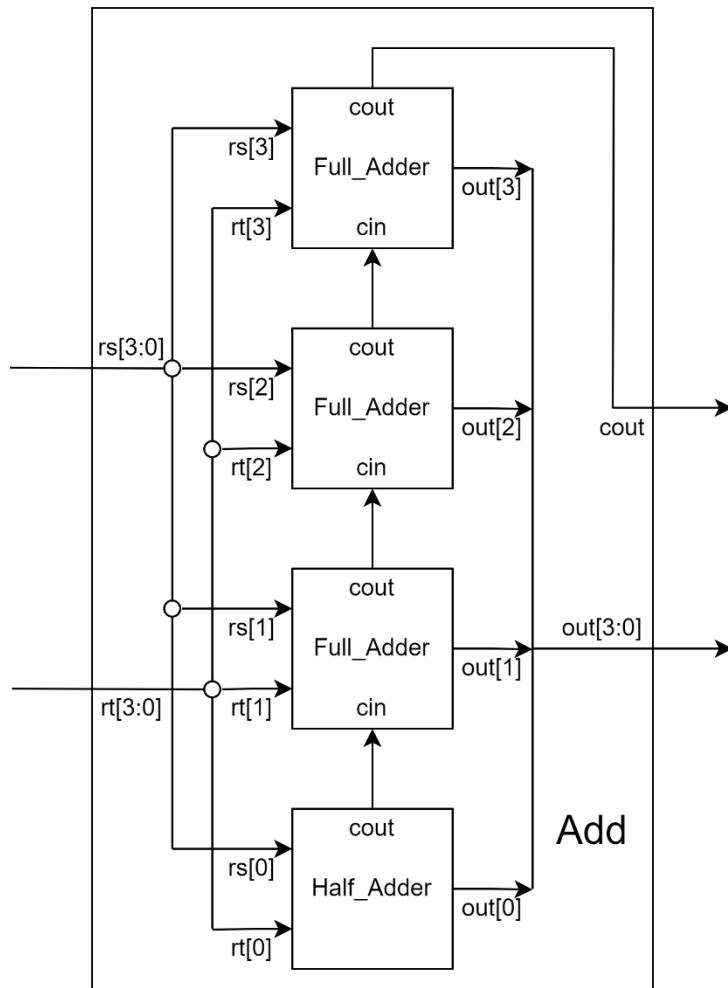


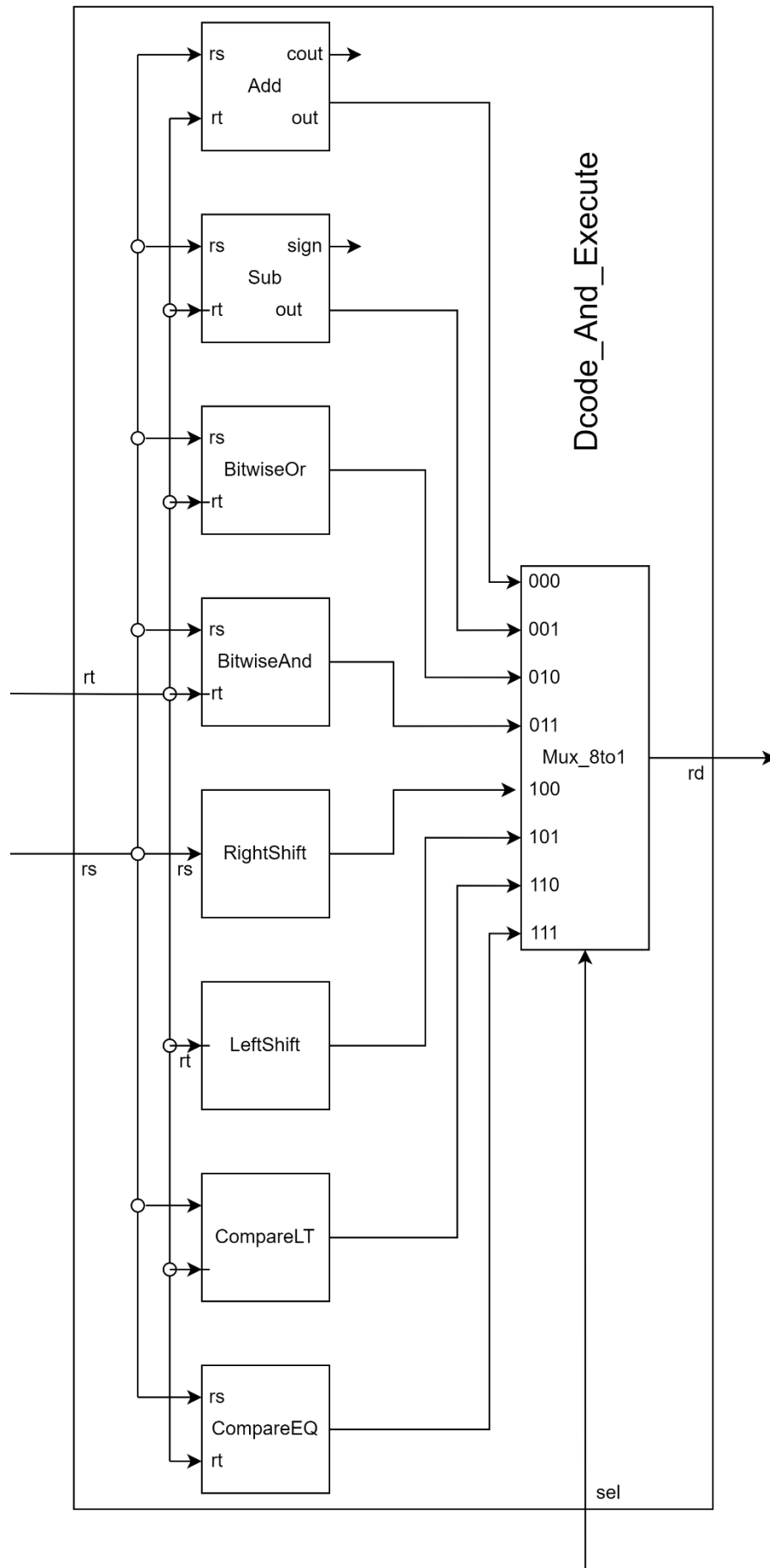












D. Explanation: Some Useful Modules

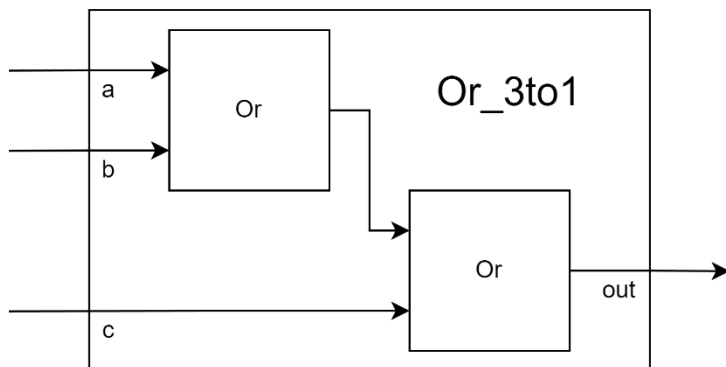
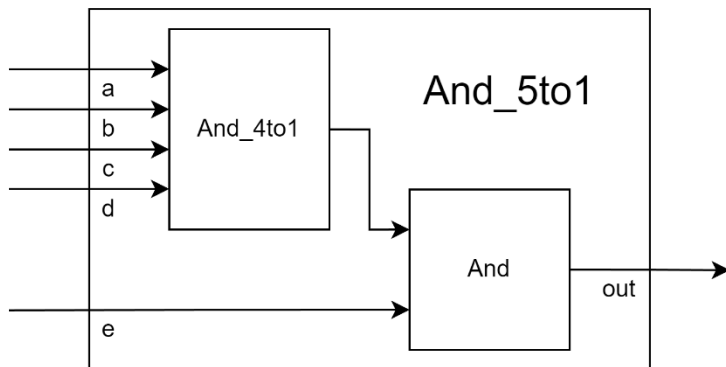
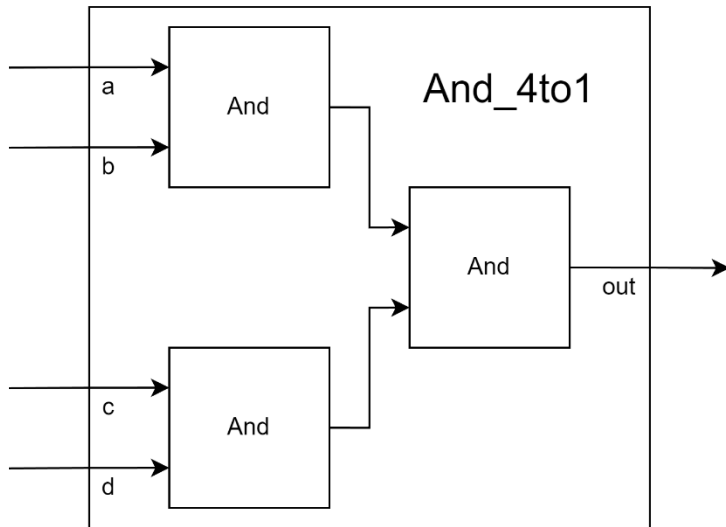
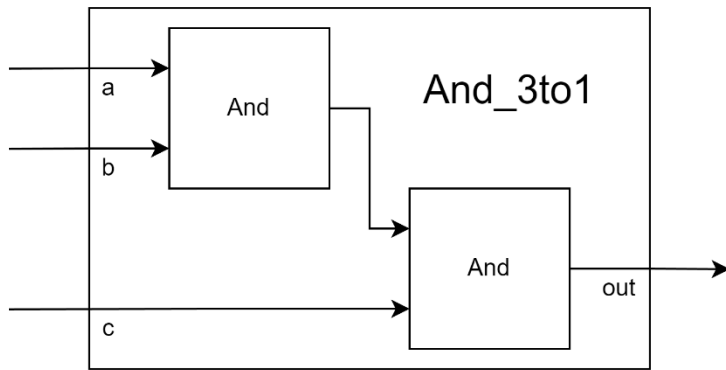
- a. **Majority:** Count the carry out
- b. **Half_adder:** Deal with 1-bit addition without carry in
- c. **Full_Adder:** Deal with 1-bit addition with carry in
- d. **CompareEQ:** Compare two 4-bit data whether they are equal or not and output **rd**
- e. **CLT_1bit:** Compare two 1-bit data **rs** and **rt** and output 1 if **rs < rt**
- f. **CLT_4bits:** Compare two 4-bit data **rs** and **rt** and output 1 if **rs < rt**
- g. **CompareLT:** Combine the result from **CLT_4bits** and output **rd**
- h. **LeftShift:** Use four **Copy** modules to shift **rs**
- i. **RightShift:** Use four **Copy** modules to shift **rt**
- j. **BitwiseAnd:** Use four **And** modules to realize it
- k. **BitwiseOr:** Use four **Or** modules to realize it
- l. **BitwiseInvt:** Use four **Not** modules to realize it
- m. **Add:** Use a **Half_Adder** module and three **Full_Adder** modules to make this ripple carry adder
- n. **Sub:** Use a **BitwiseInvt** module and a **Add** module to get the negation of **rt** and then use a **Add** module to get **rd = rs + (-rt)**
- o. **Decode_And_Execute:** Use the modules mentioned above to deal with eight situation and then use a **Mux_8to1** module to choose which result should be output

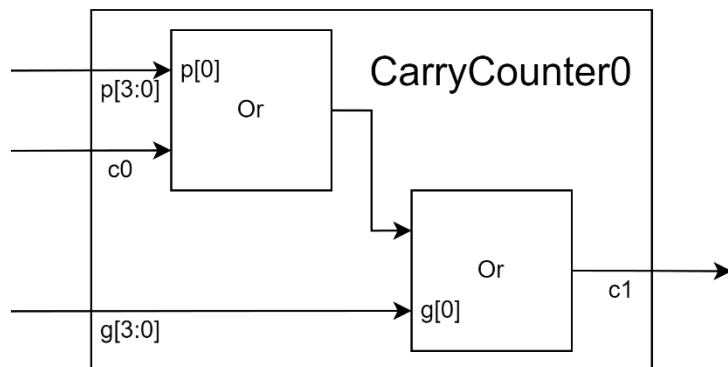
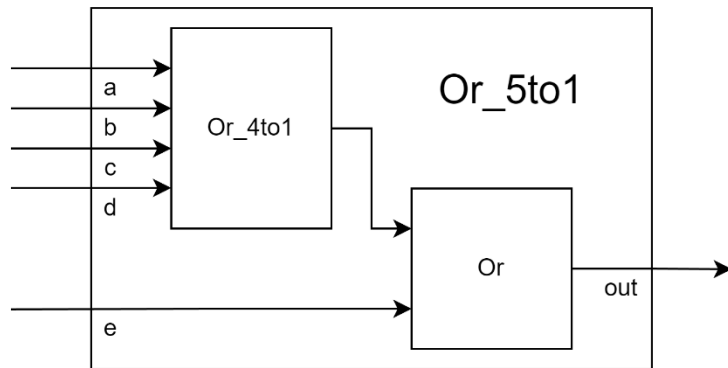
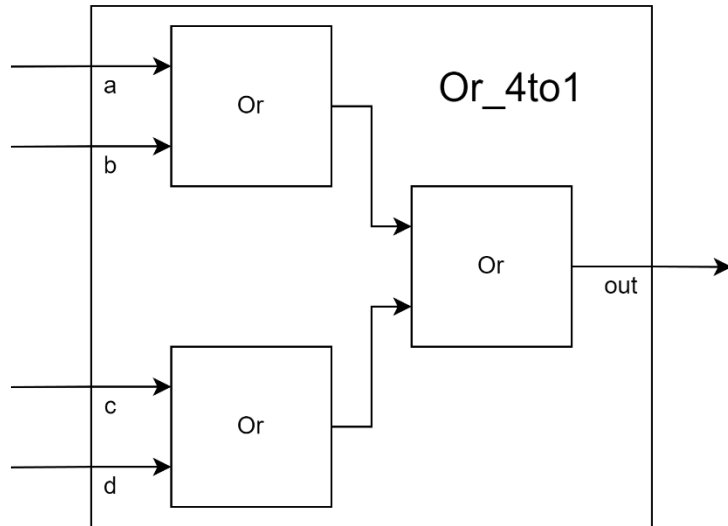
E. Testbench

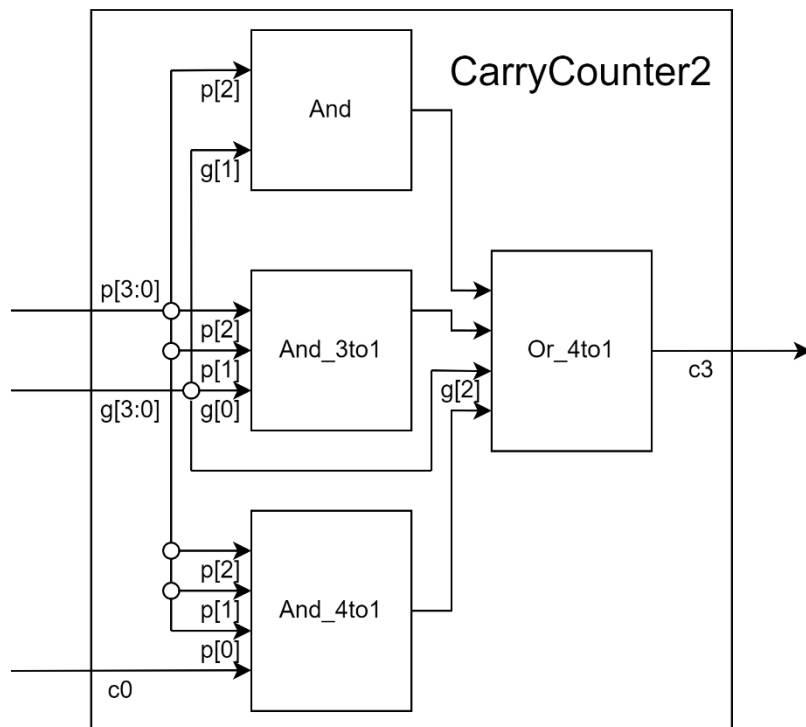
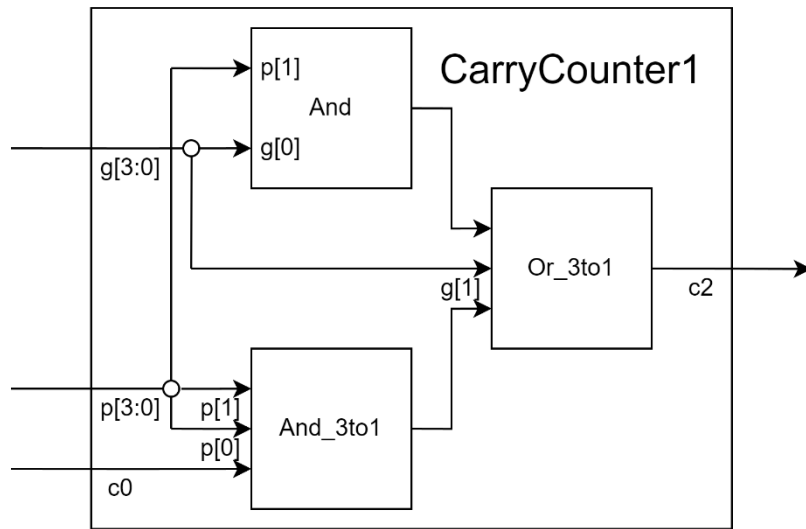
I set the simulation time to 1,0000,0000 ns and use three layers of loop (for **rs**, **rt**, **sel**) to go through every input pattern. And I also define a register called **err**. The testbench will check every result by my behavioral-level code. If the result from the gate-level circuit is different from the result which is counted by the behavioral code, **err** will be pulled up as 1'b1. Otherwise, it will remain 1'b0.

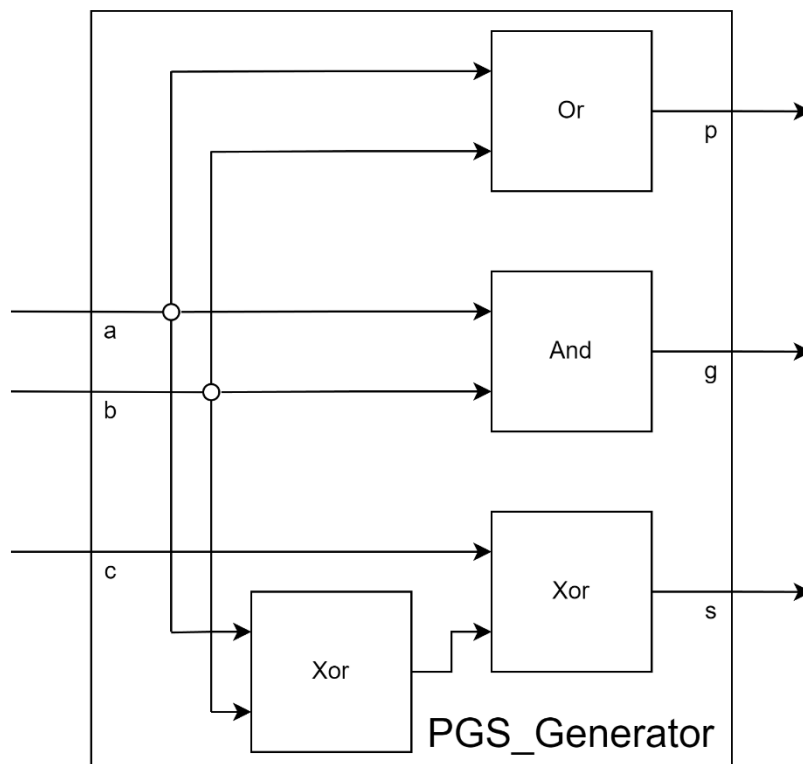
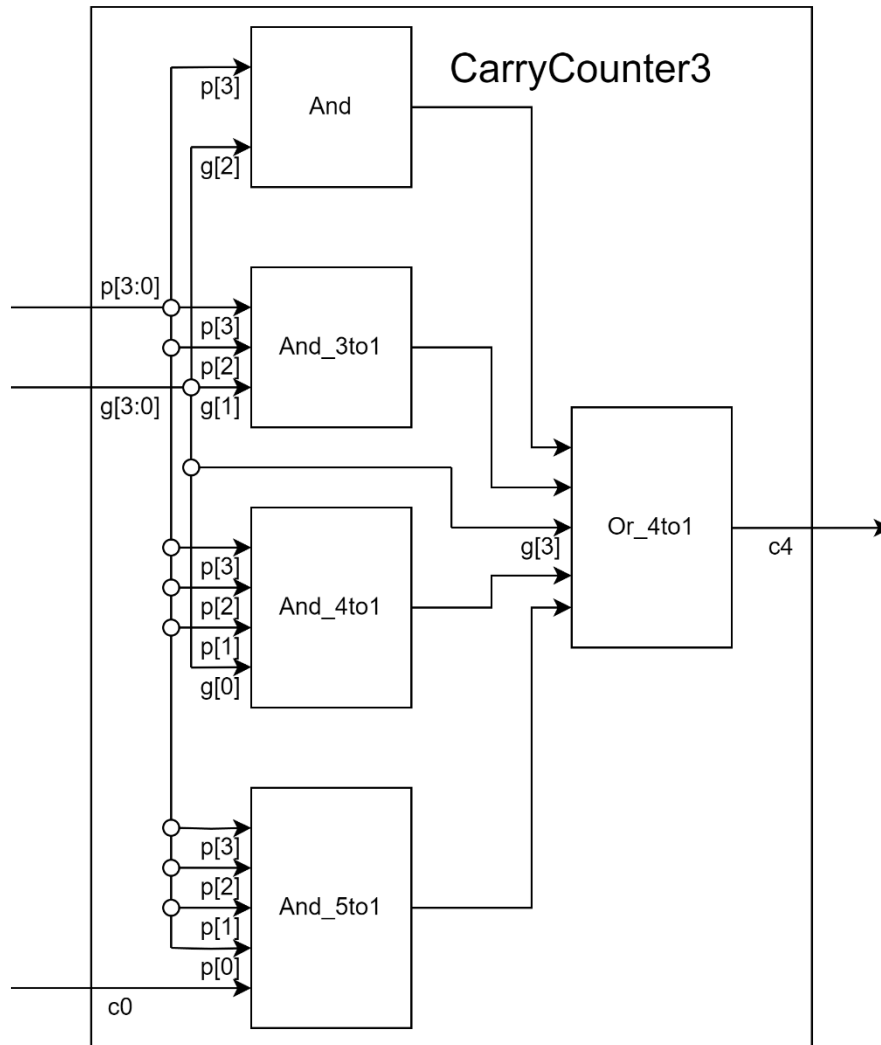
3. Advanced: 8-bit carry-lookahead (CLA) Adder

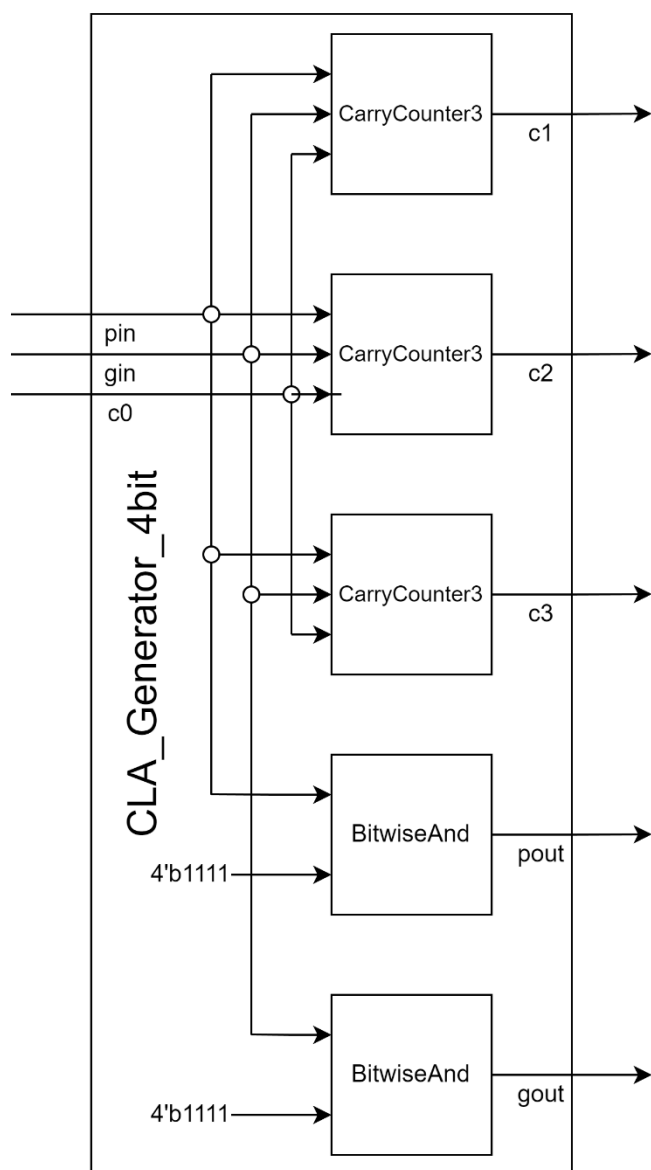
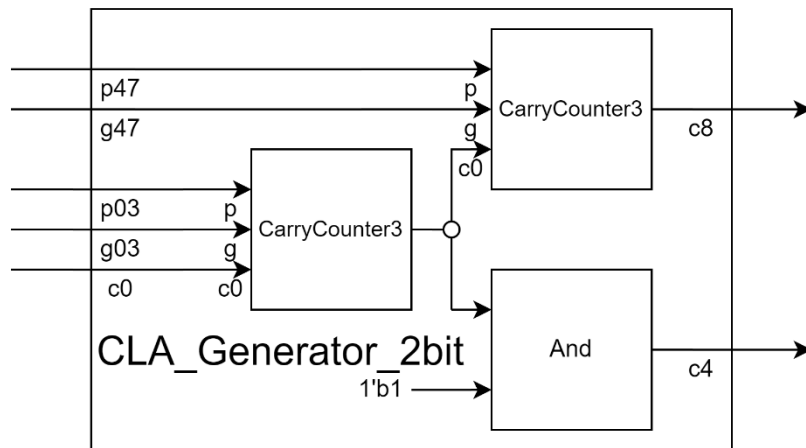
A. Block Diagram

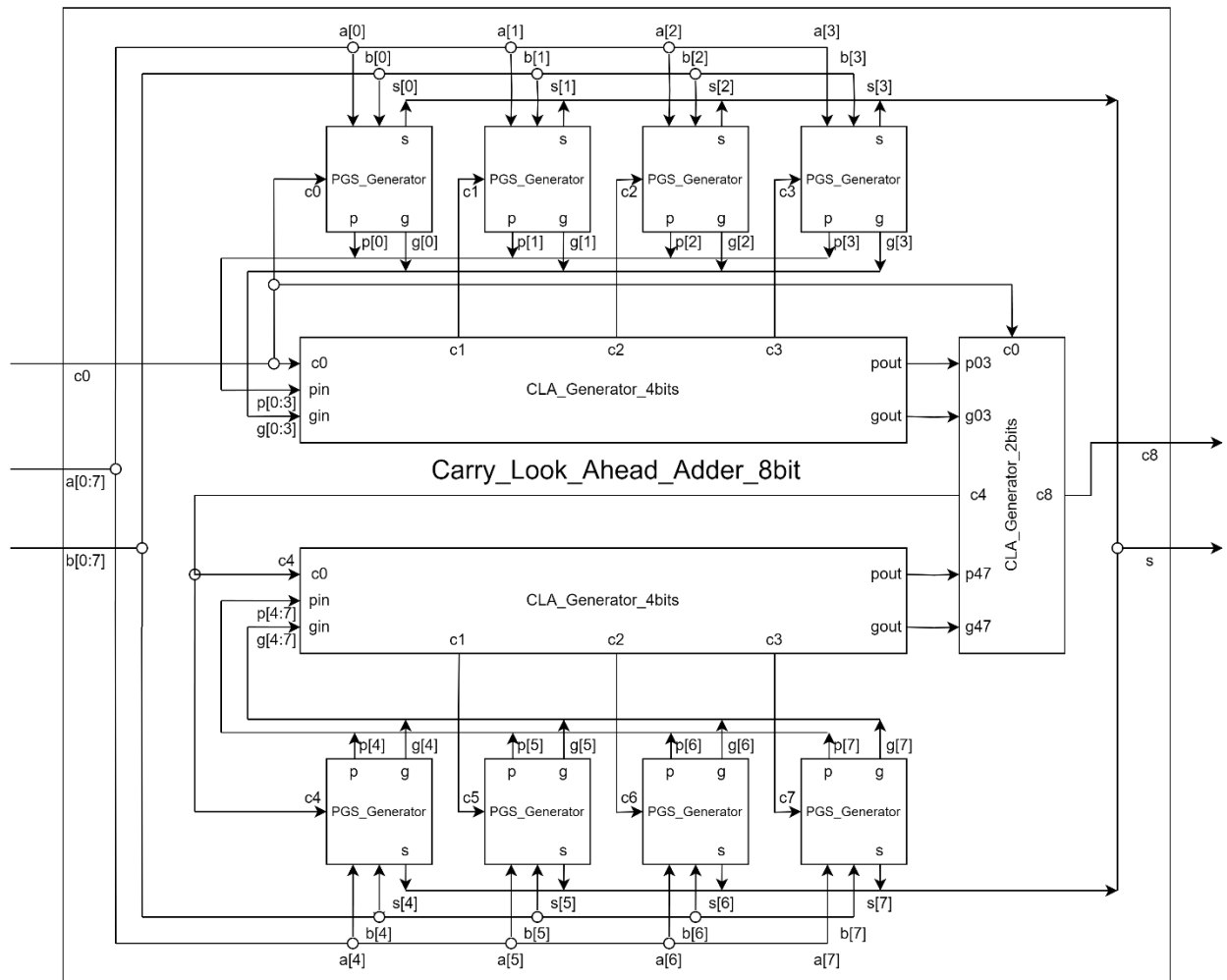












B. Explanation

- $Cin_1 = g_0 + (p_0 * Cin_0)$
- $Cin_2 = g_1 + (p_1 * g_0) + (p_1 * p_0 * Cin_0)$
- $Cin_3 = g_2 + (p_2 * g_1) + (p_2 * p_1 * g_0) + (p_2 * p_1 * p_0 * Cin_0)$
- $Cin_4 = g_3 + (p_3 * g_2) + (p_3 * p_2 * g_1) + (p_3 * p_2 * p_1 * g_0) + (p_3 * p_2 * p_1 * p_0 * Cin_0)$

▲ source: https://chi_gitbook.gitbooks.io/personal-note/content/addition.html

CarryCounter0: count **c1** by **c0**, **p** and **g**

CarryCounter1: count **c2** by **c0**, **p** and **g**

CarryCounter2: count **c3** by **c0**, **p** and **g**

CarryCounter3: count **c4** by **c0**, **p** and **g**

PGS_Generator: count **p**, **g** and **s** (sum) by **a**, **b** and **c** (carry in)

CLA_Generator_2bit: Combine the data generated by **CLA_Generator_4bit**

CLA_Generator_4bit: generate carry for each bit follow the picture above

simultaneously

Carry_Look_Ahead_Adder_8bit: add two 8-bit data by carry-look-ahead method

C. Testbench

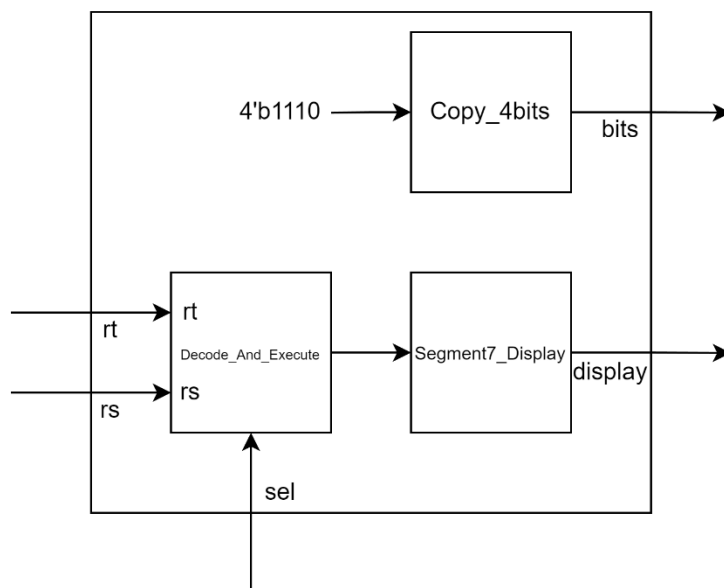
I set the simulation time to 1,0000,0000 ns and use three layers of loop (for **a**, **b**, **c0**) to go through every input pattern. And I also define a register called **err**. The testbench will check every result by my behavioral-level code. If the result from the gate-level circuit is different from the result which is counted by the behavioral code, **err** will be pulled up as 1'b1. Otherwise, it will remain 1'b0.

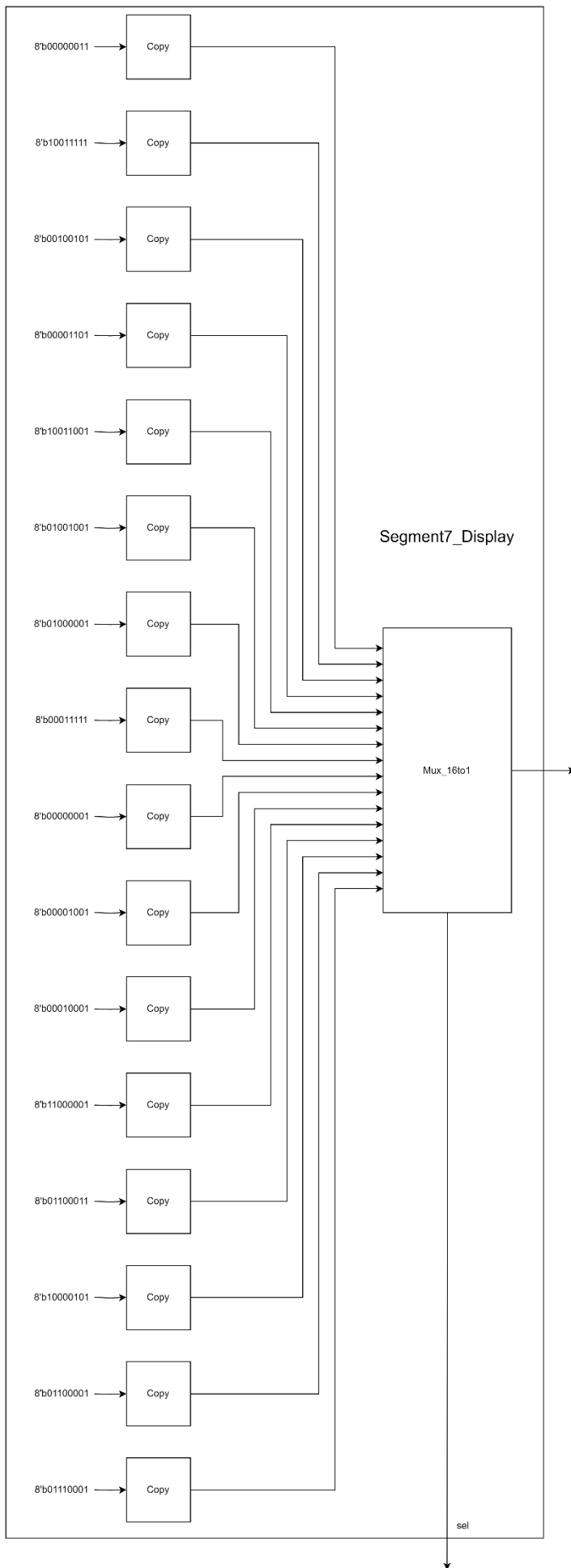
4. An exhaustive testbench design

After finishing **DAE** and **CLA**, I learned that it is inefficient to check wave form through my eyes. Therefore, I use behavioral-level code to save the correct answer in some registers and compare it to the result from the circuit. If they are not the same, **err** will be pulled up. Otherwise, **err** will be pulled down. In this way, I can only check **err** to know that if there is a bug or not.

5. Decode and execute (FPGA)

A. Block Diagram





B. Explanation

After counting **rd** by the **Decode_And_Execute** module, **rd** would be passed into an encode module called **Segment7_Display** to transform to correct value to control 7-segment display on FPGA board. **bits** is for controlling which bit on the FPGA board should be active. **display** is for controlling which segment should be active in a bit. Note that 7-segment display is low-active.

6. What I Have Learned?

In the past, I used to write code without any plan. However, in this lab, I've learned the importance of planning before writing. The profit of planning before writing is that I can design my modules more efficiently and more precisely. Also, I can design some useful modules and reuse them to concise my design, which can make me debug more easily. In addition, from **Exhausted Testbench**, I learned that the simulation time limit can be set by myself and it is more conveniently that defining a register to show if there is something wrong.