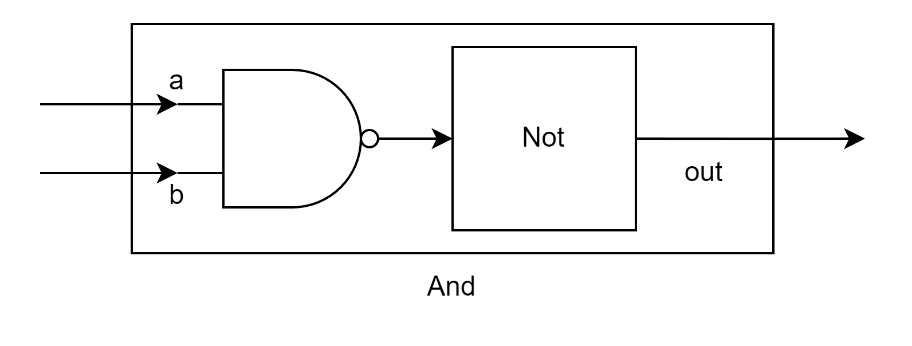
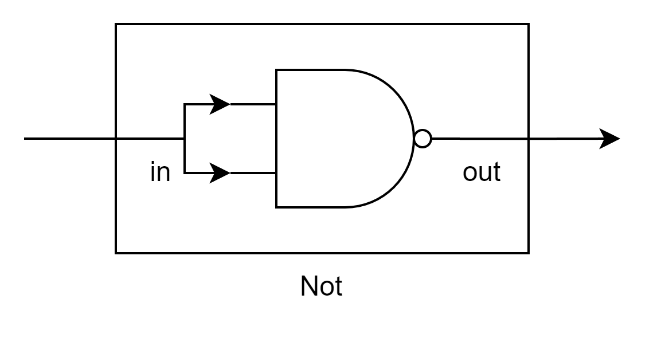
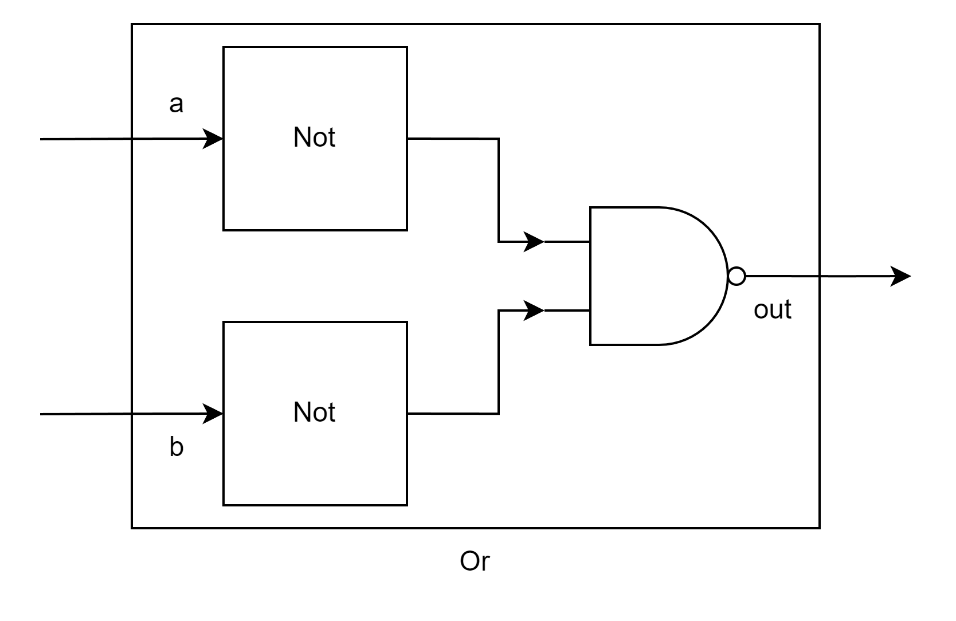
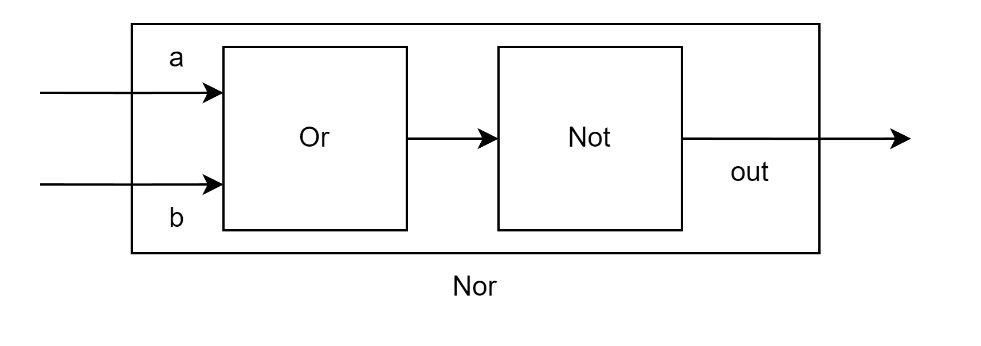
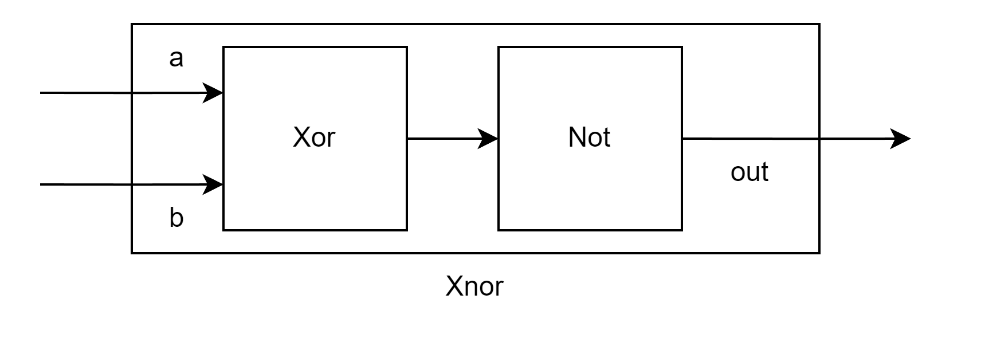
**Hardware Design and Lab: Lab1**

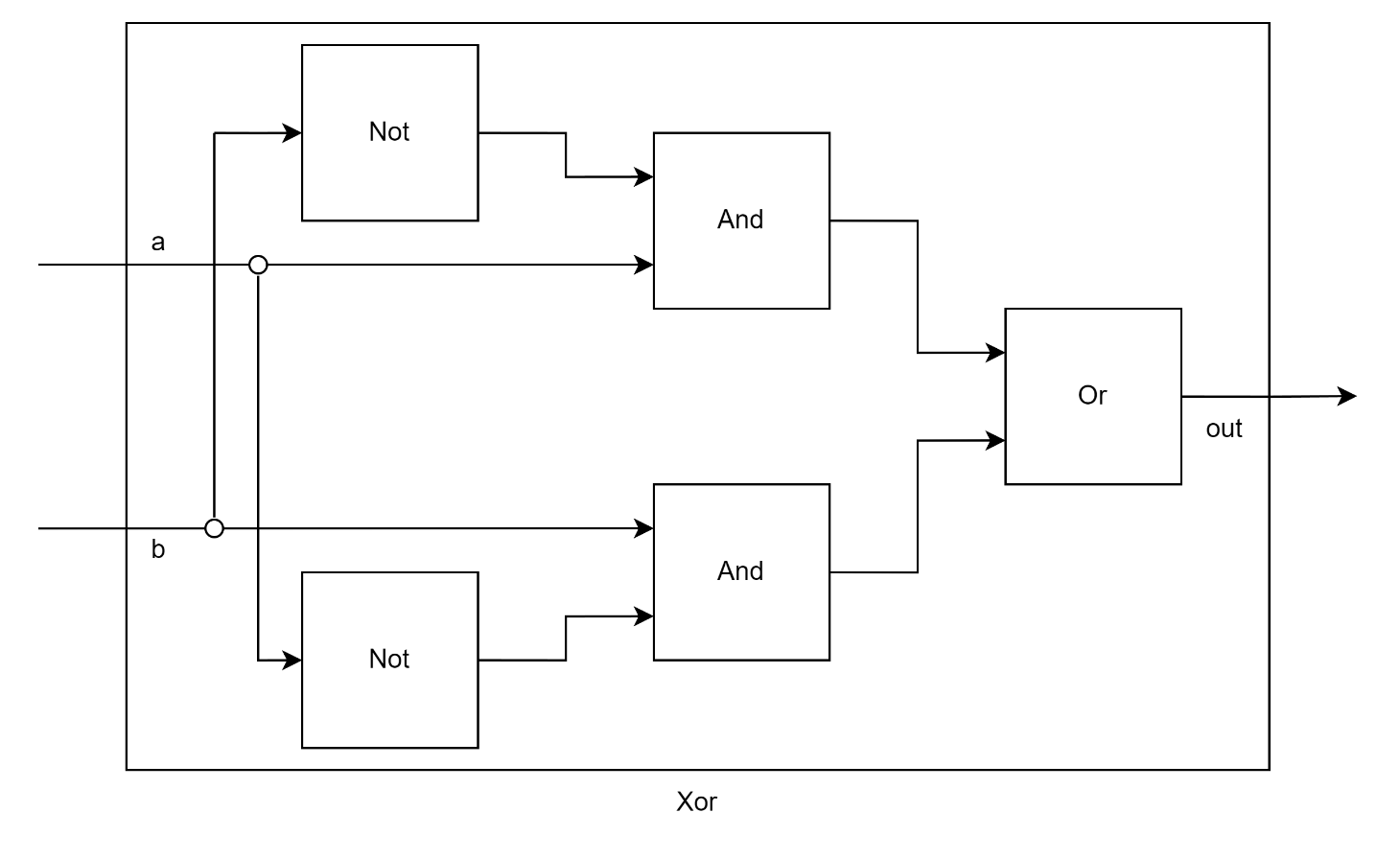
**111060013 EECS 26' 劉祐廷**

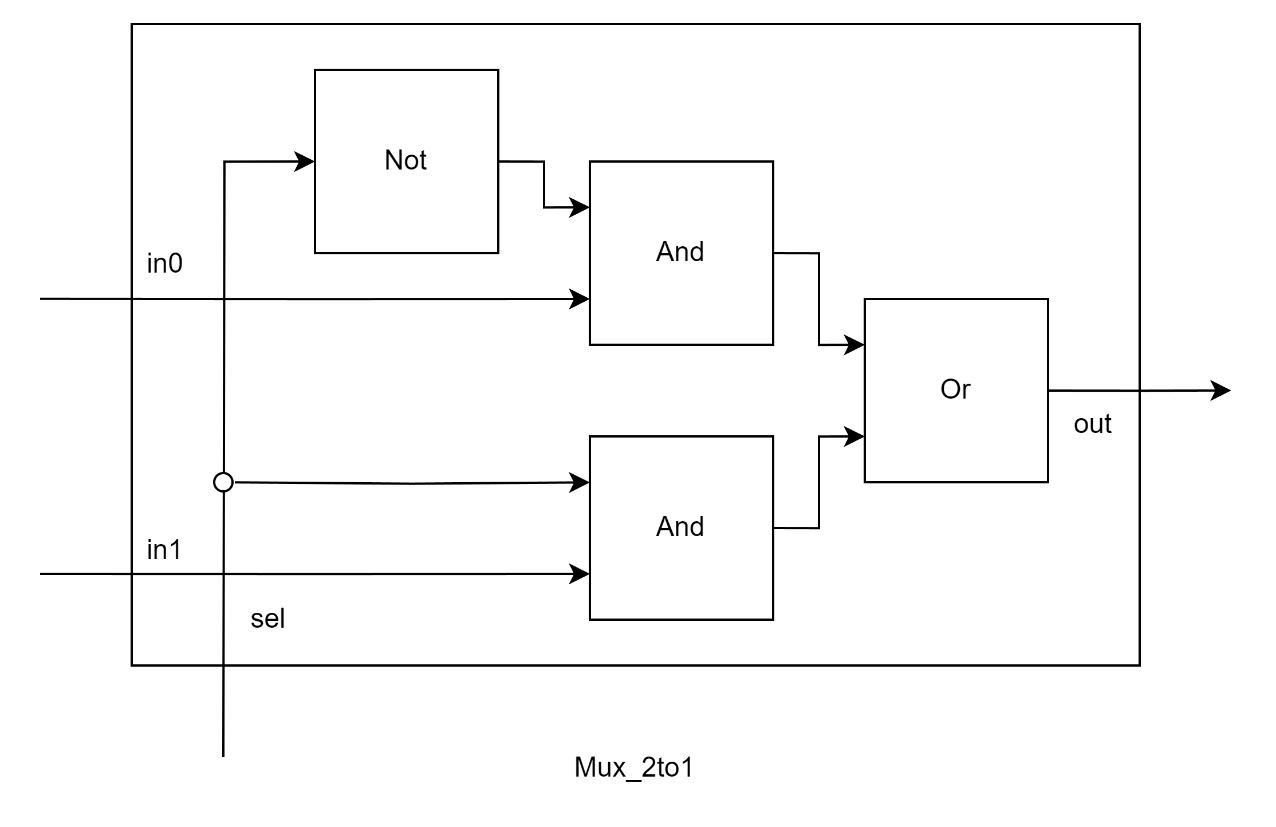
1. **Basic**
2. **Basic Question 1: Block Diagram**

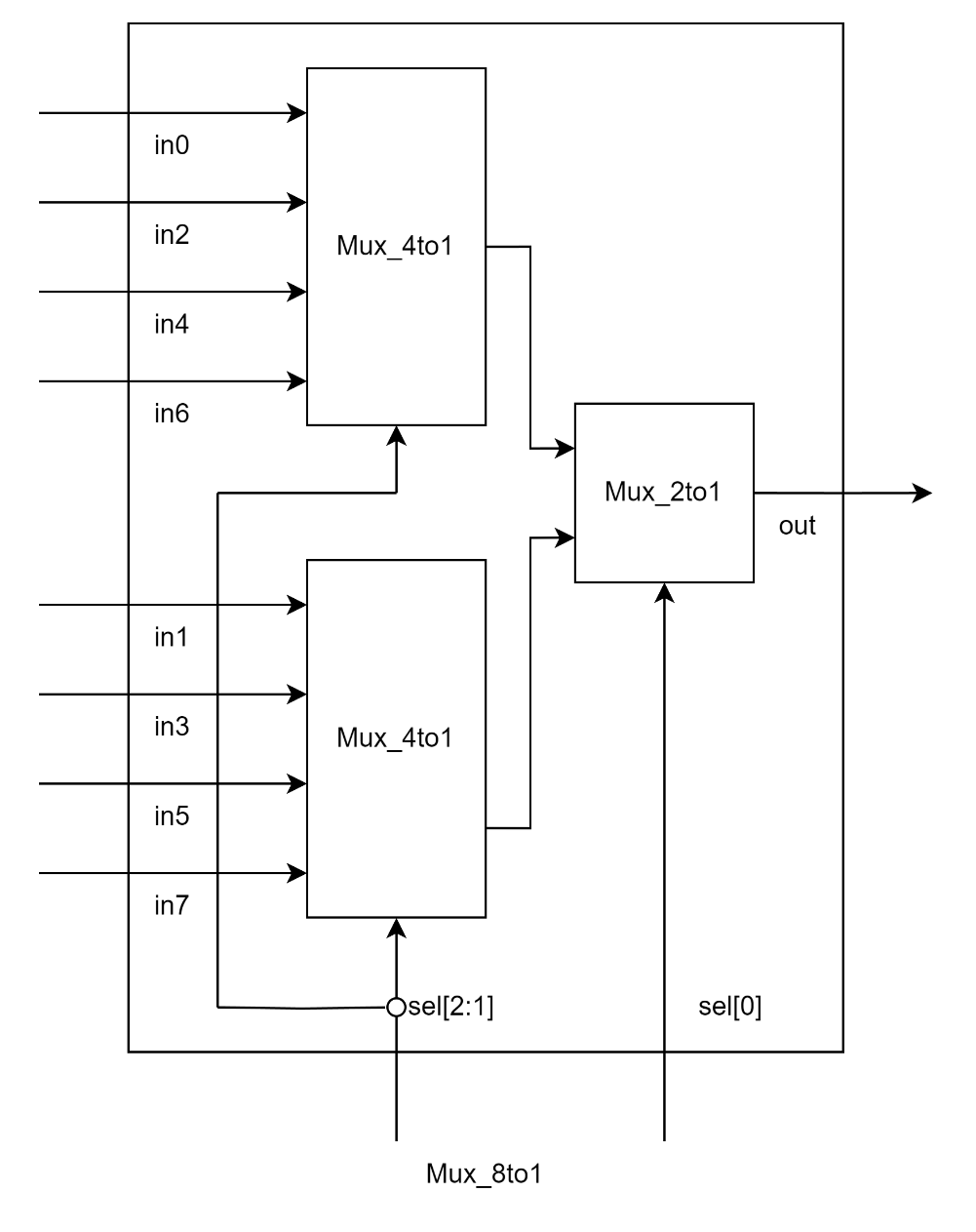
****

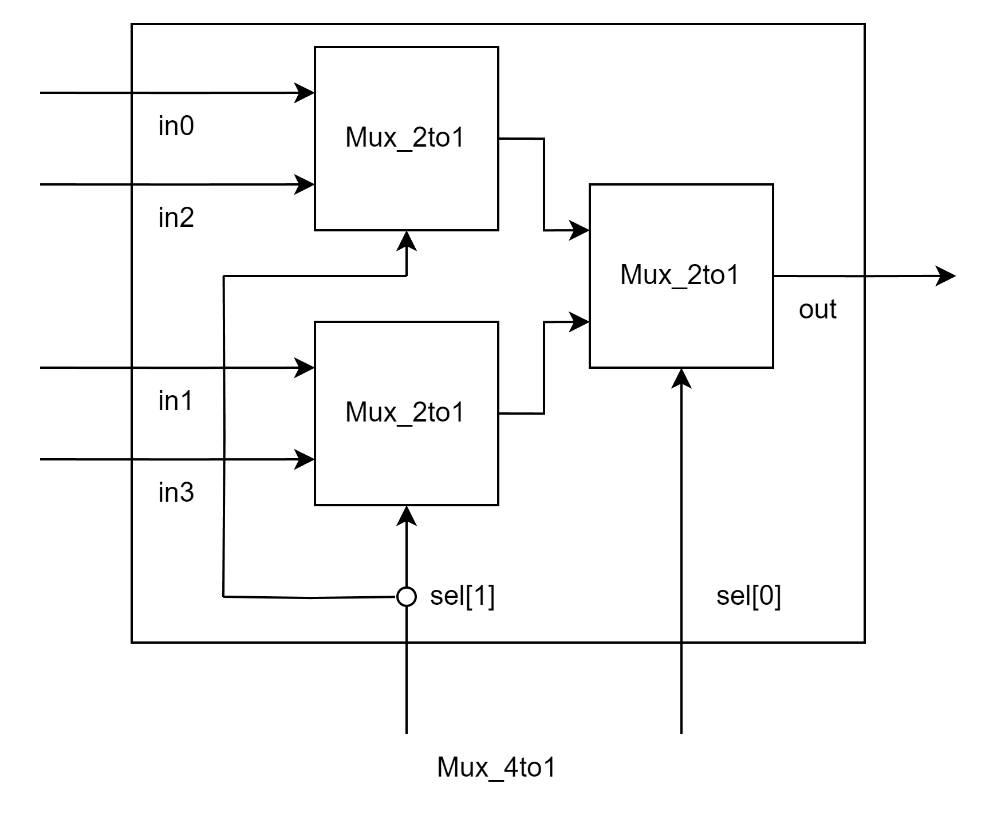
****

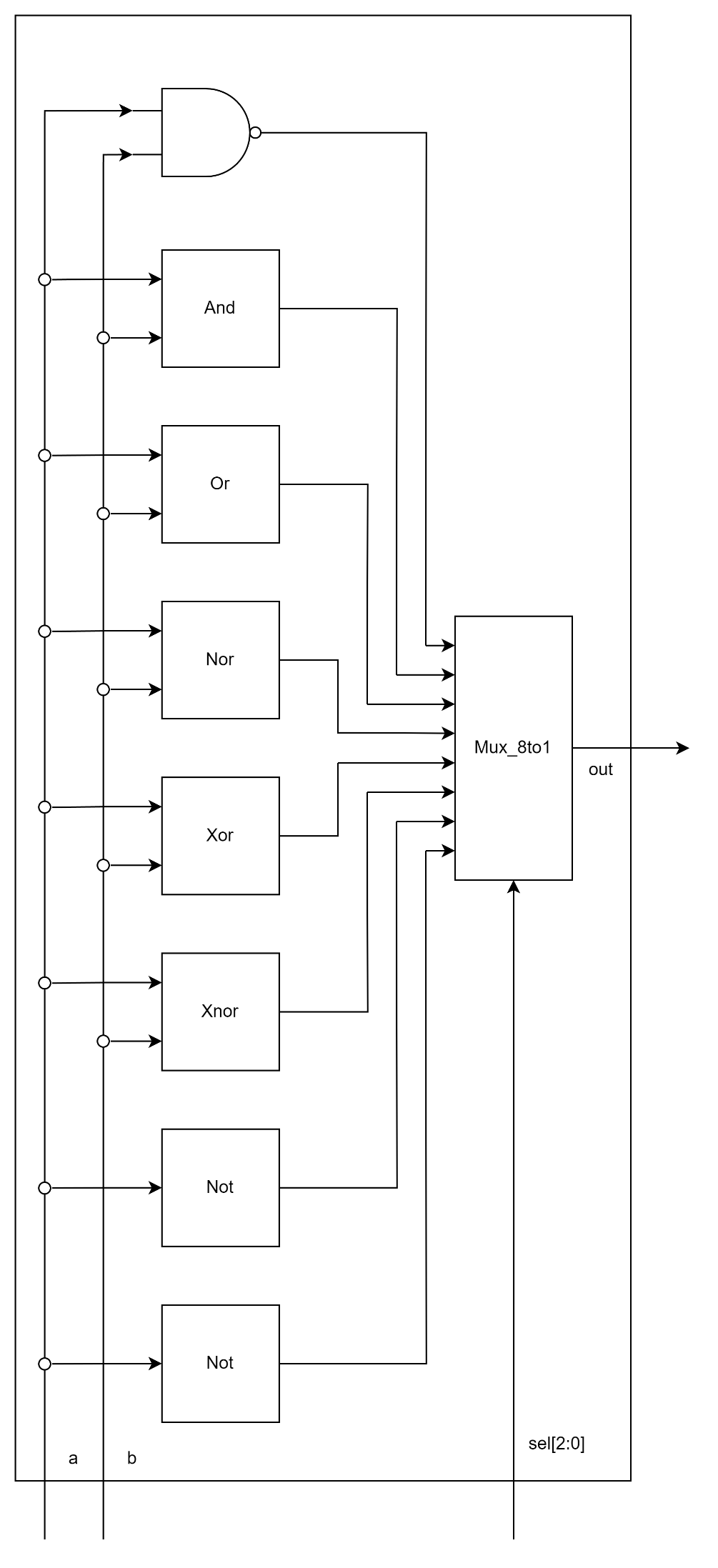
****

****

****

****

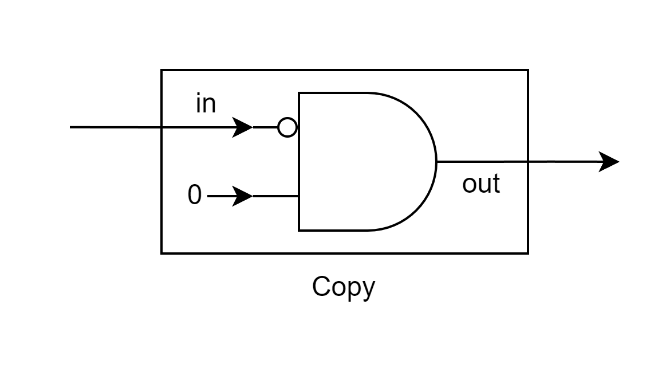
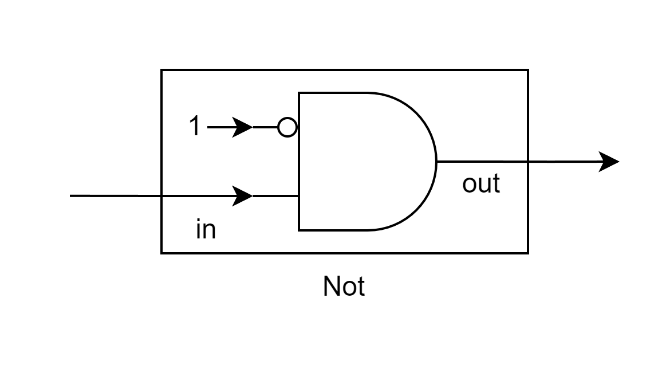
****

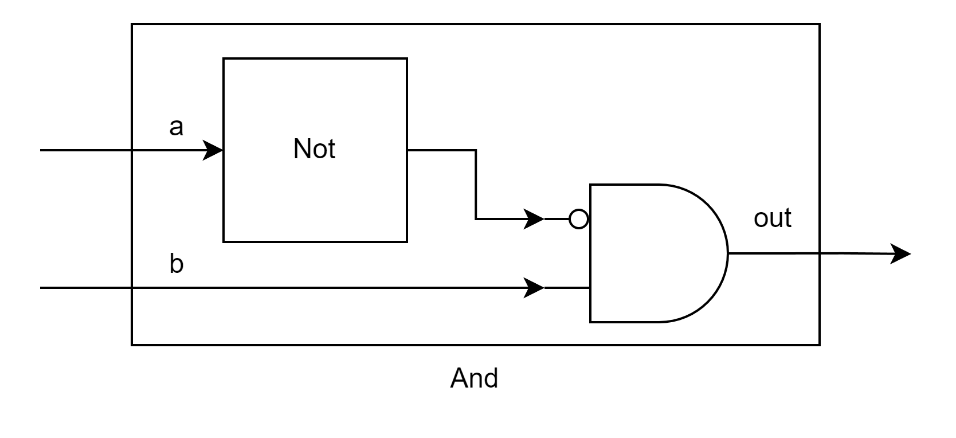
****

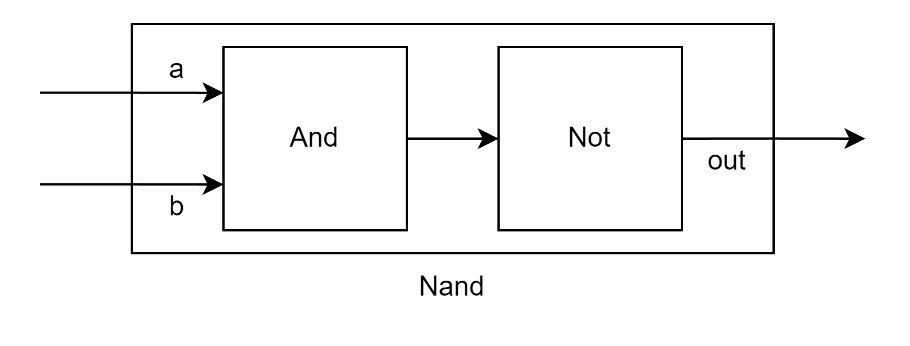
1. **Basic Question 3: The Difference Between Full Adder and Half Adder**

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.

1. **Advanced: Decode and execute**
2. **Block Diagram: Basic Modules**

****

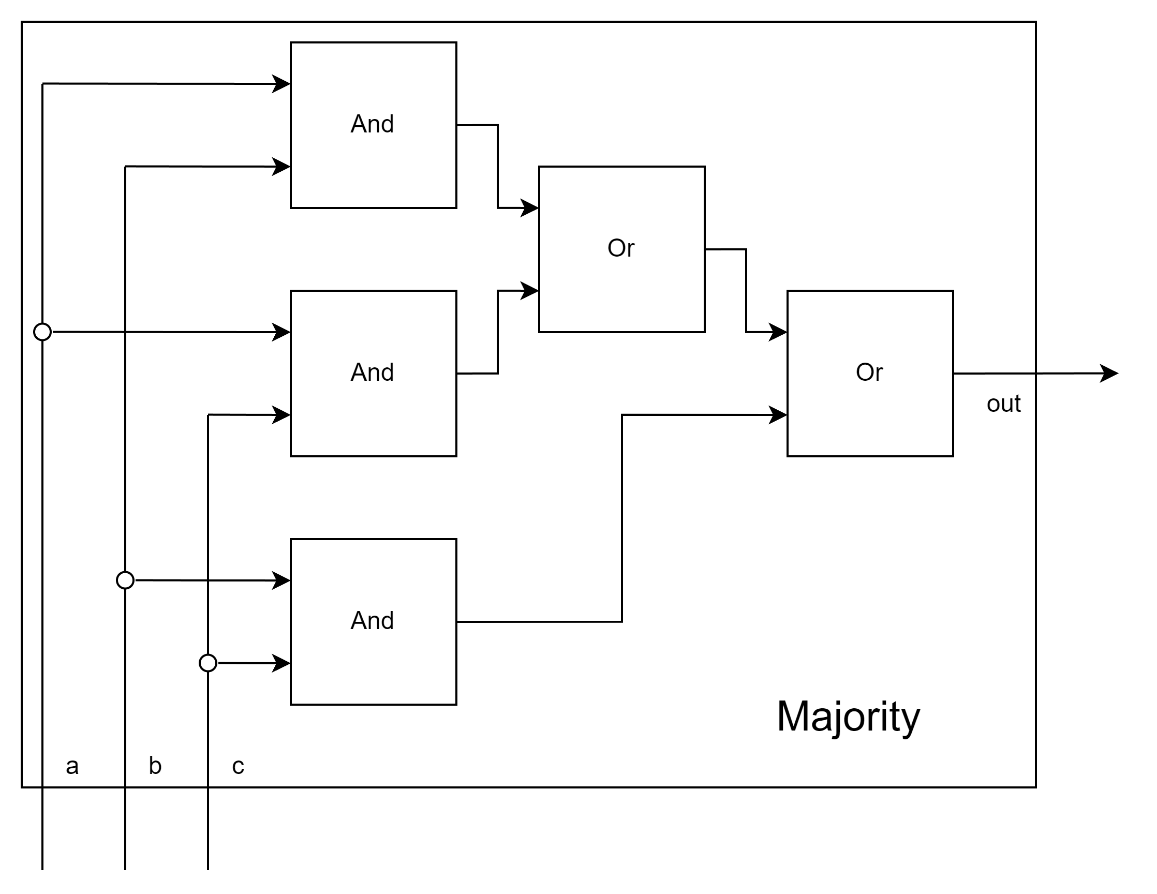
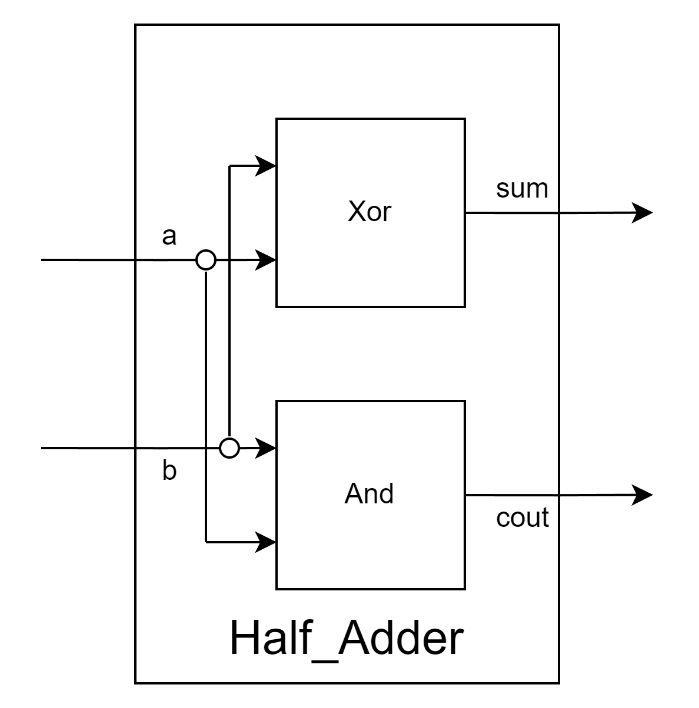
****

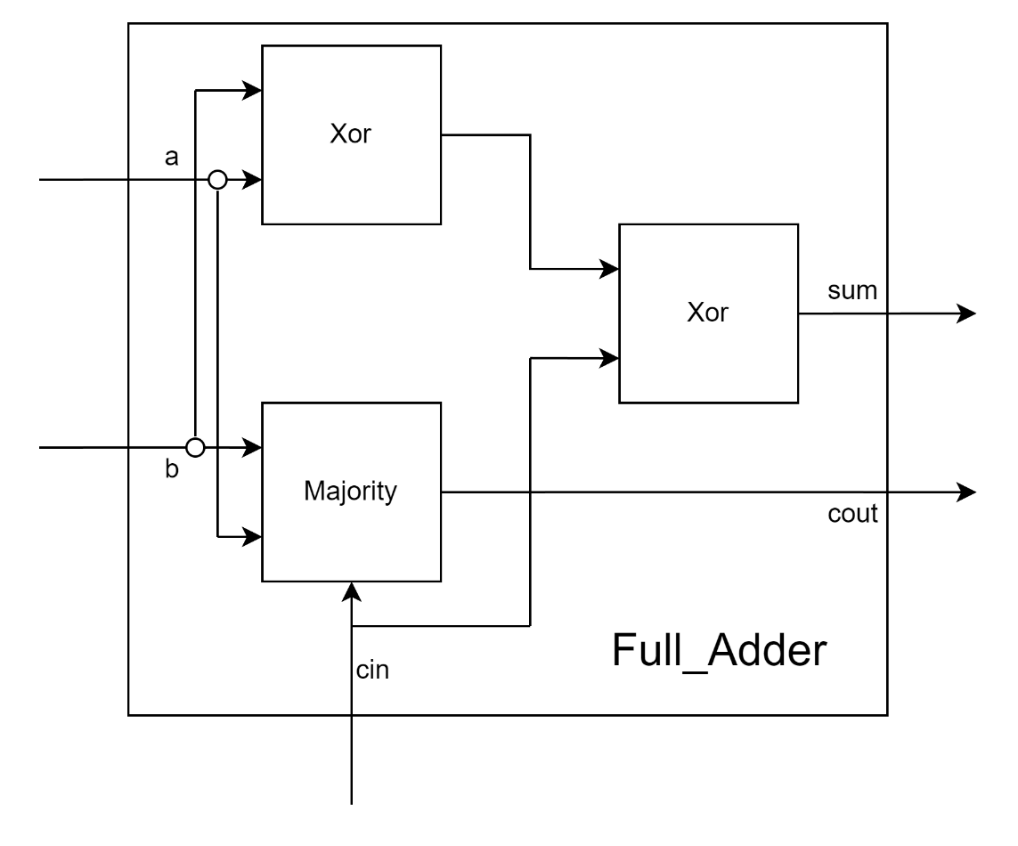
****

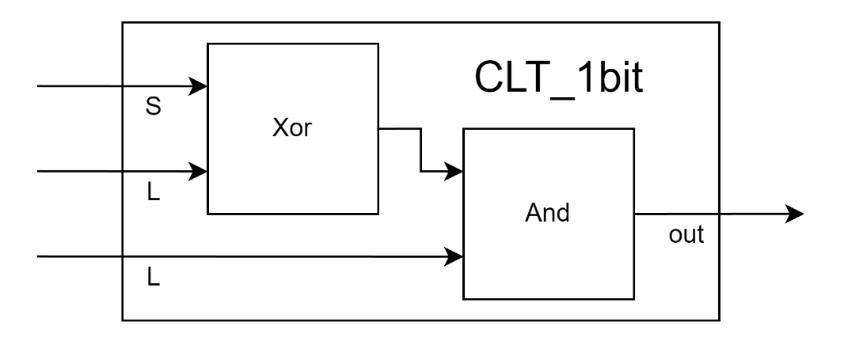
1. **Explanation: Basic Modules**

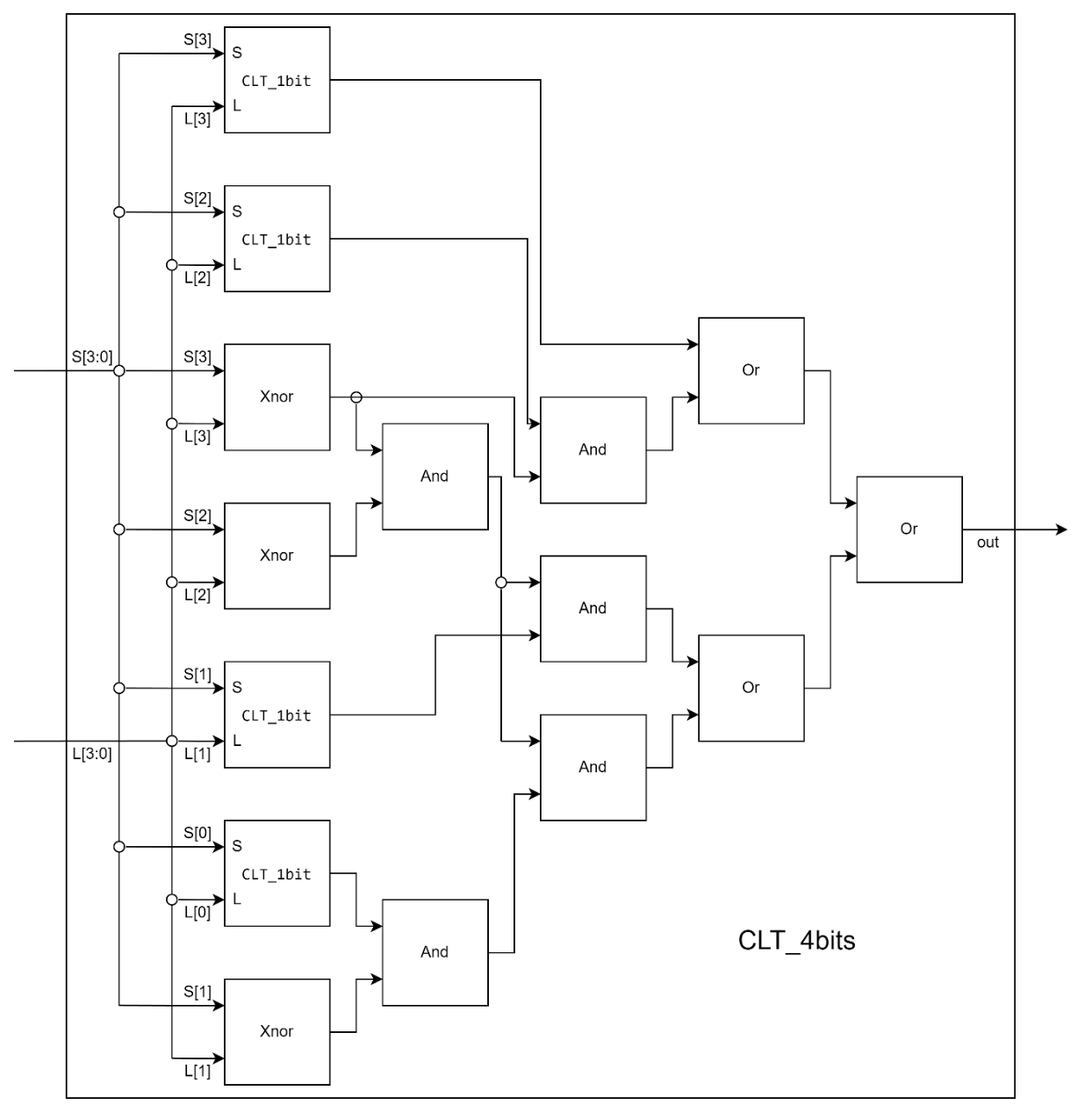
First of all, I drew the truth table of **Universal Gate** to design my own **Not** module and **Copy** module. And then I combined a **Not** module and an **Universal Gate** to create **And** module. After that, I make a **Nand** module with an **And** module and a **Not** module. The reason why I design **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.

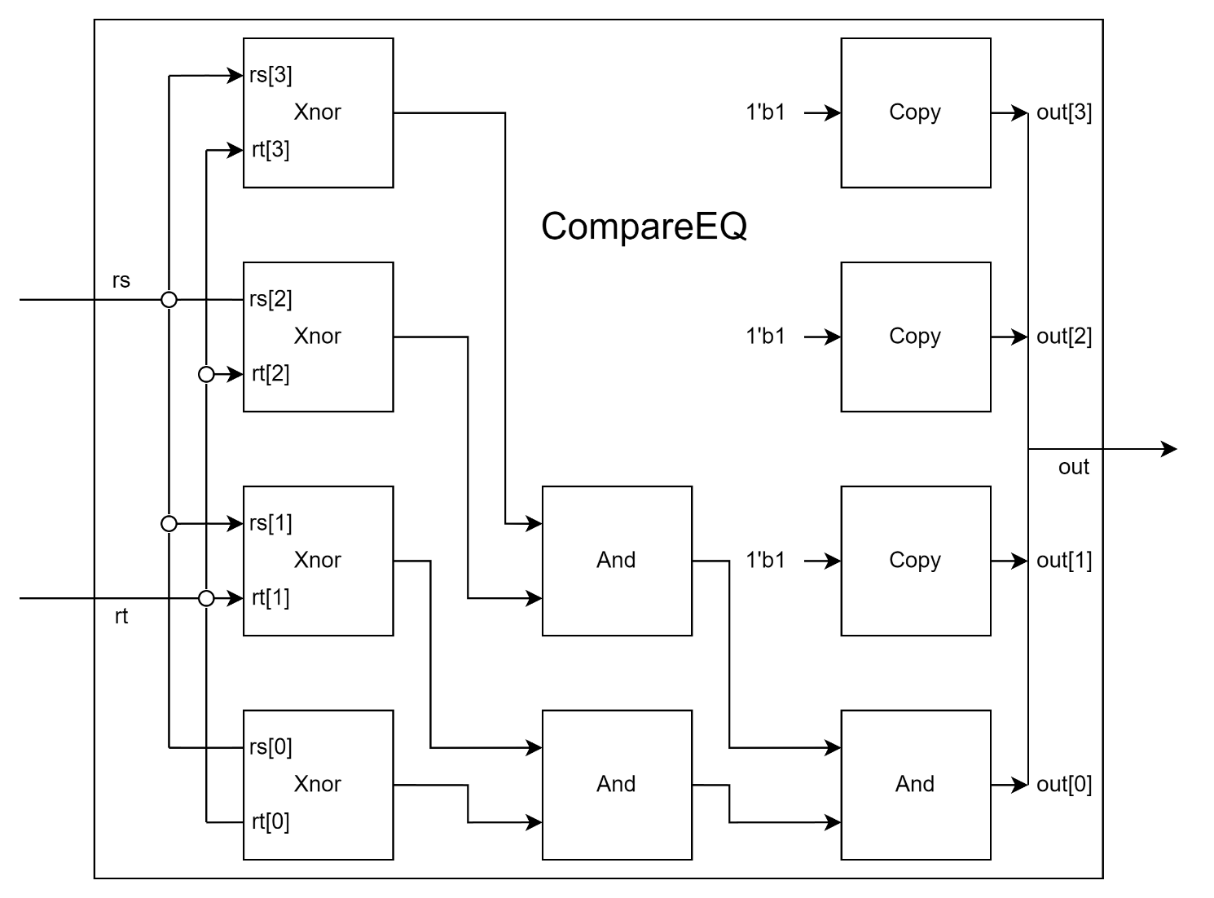
1. **Block Diagram: Advanced Modules**

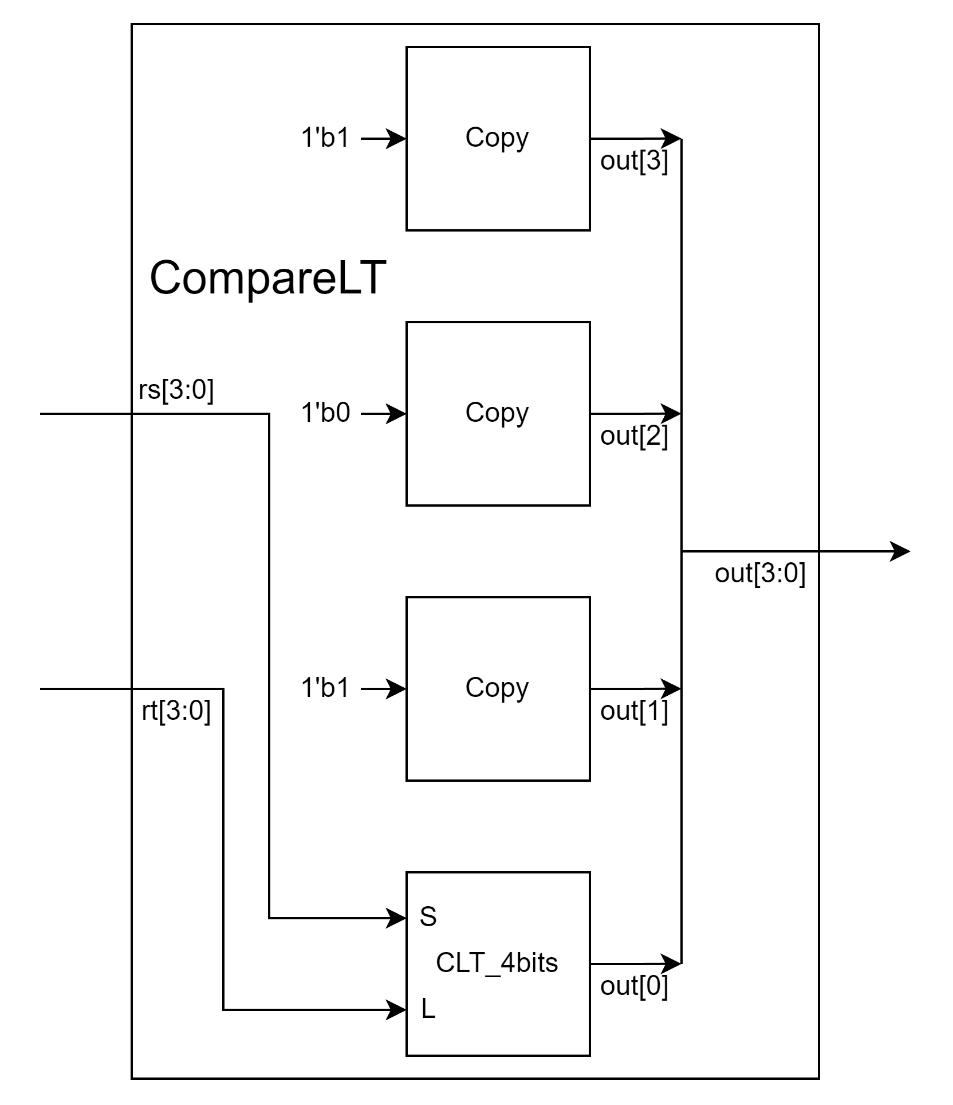
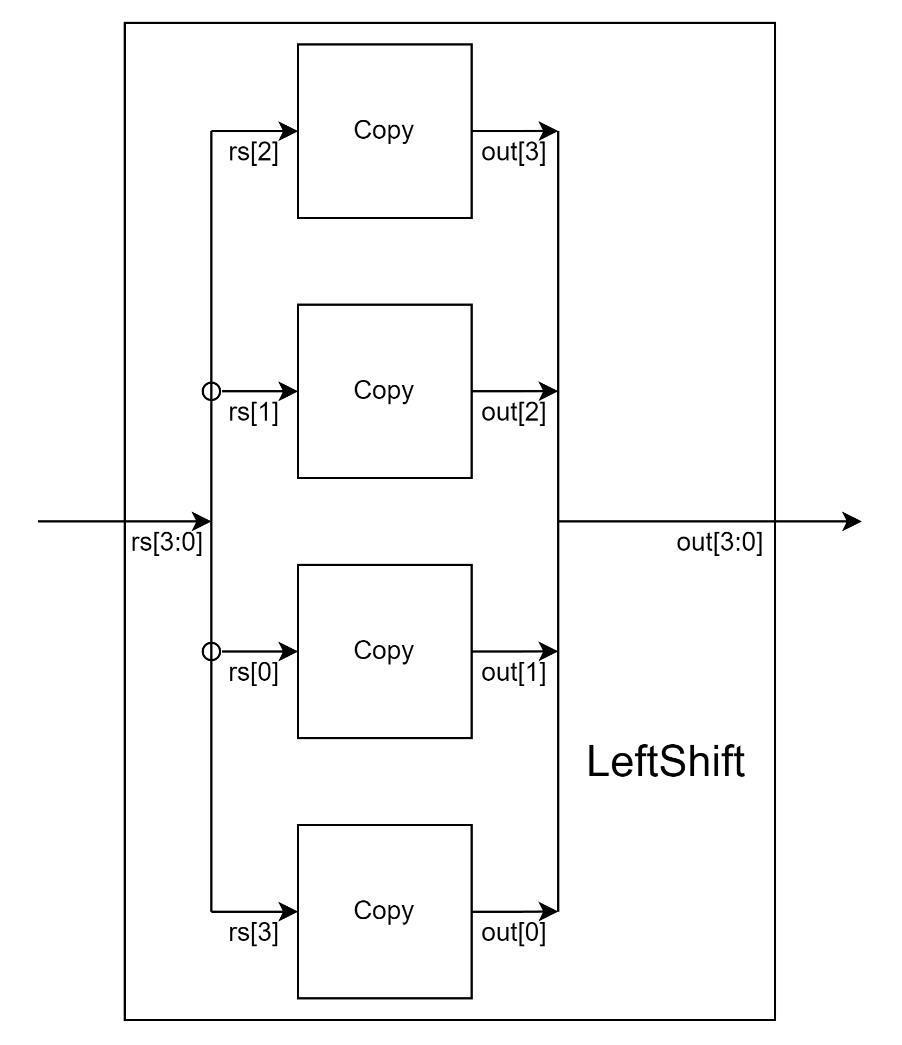
****

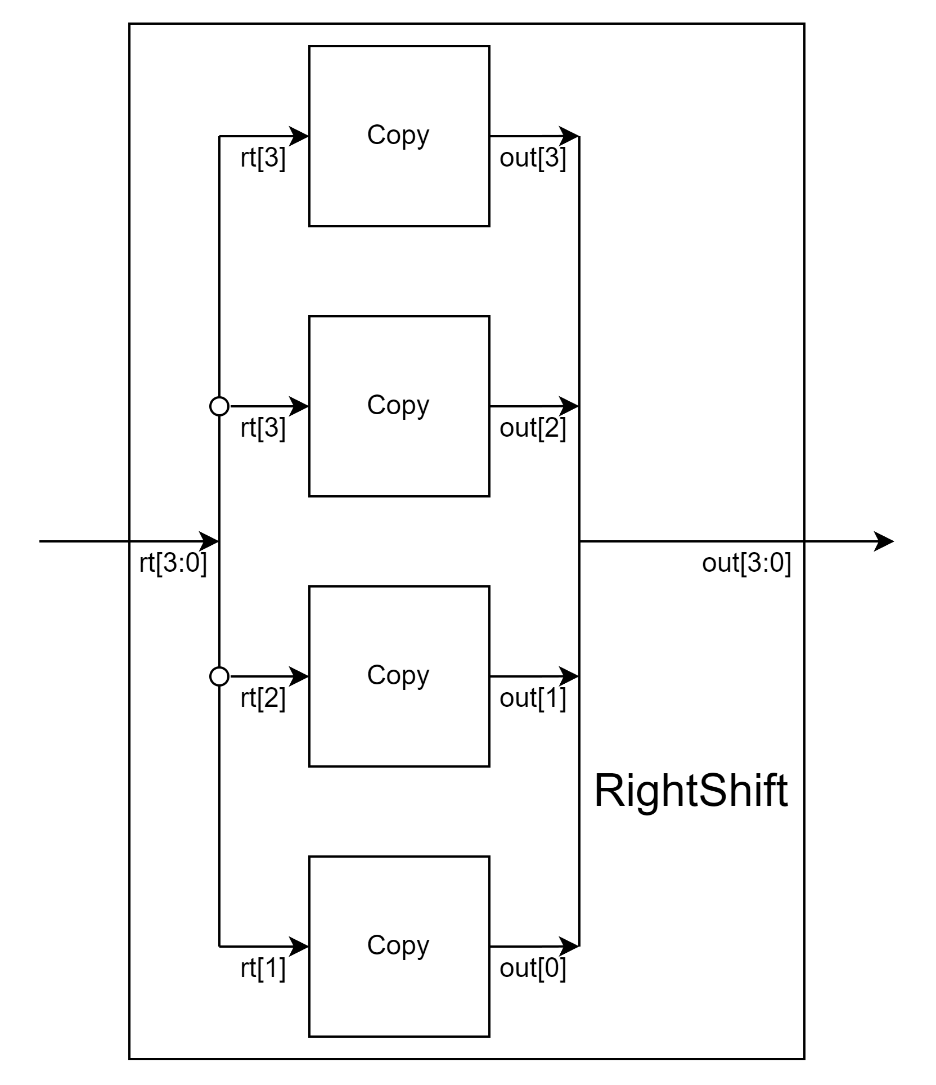
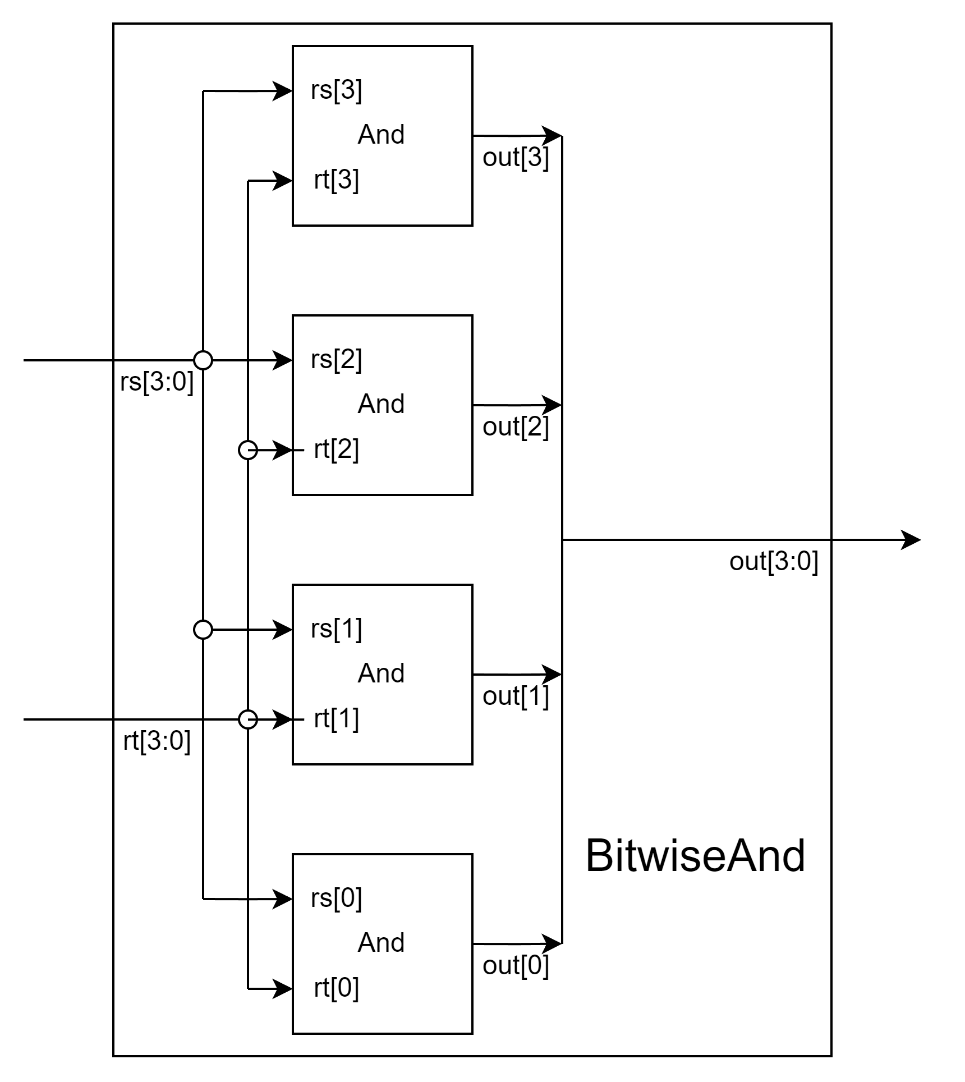
****

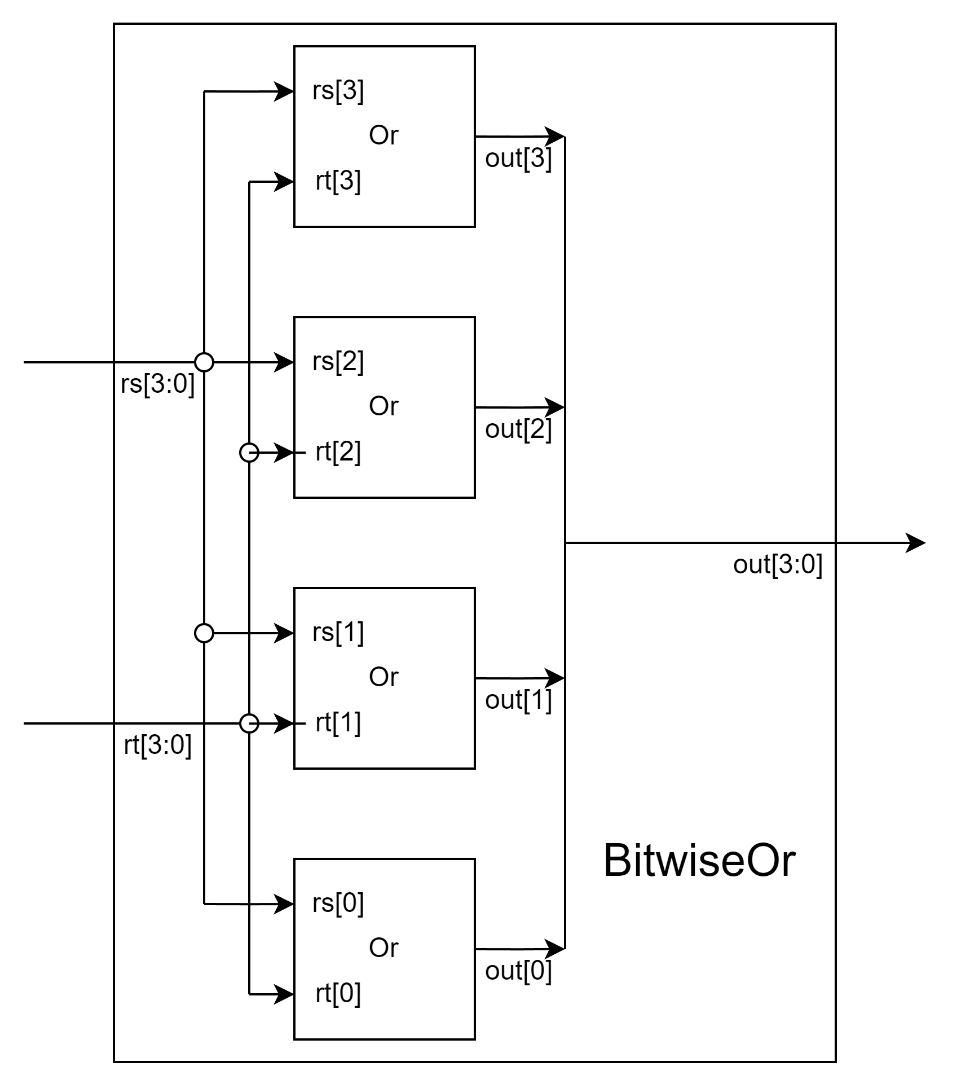
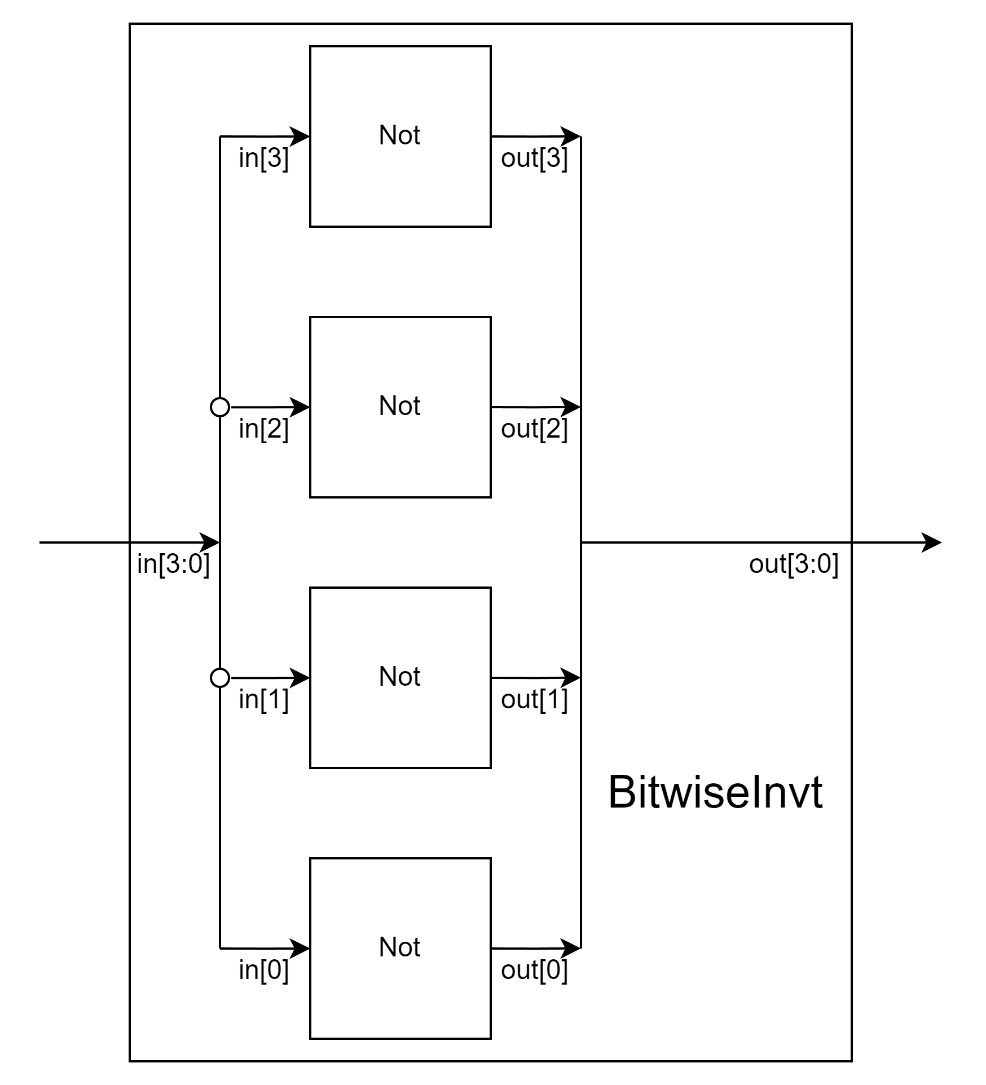
****

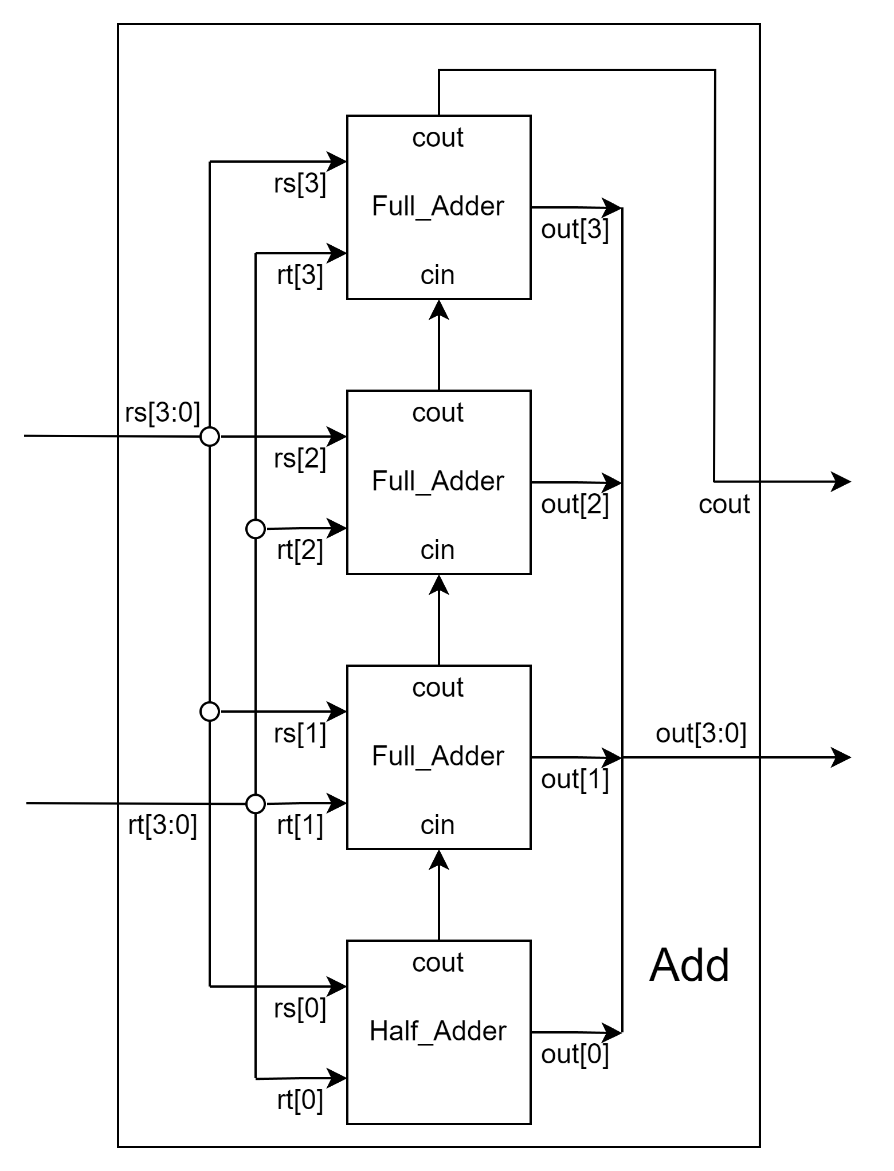
****

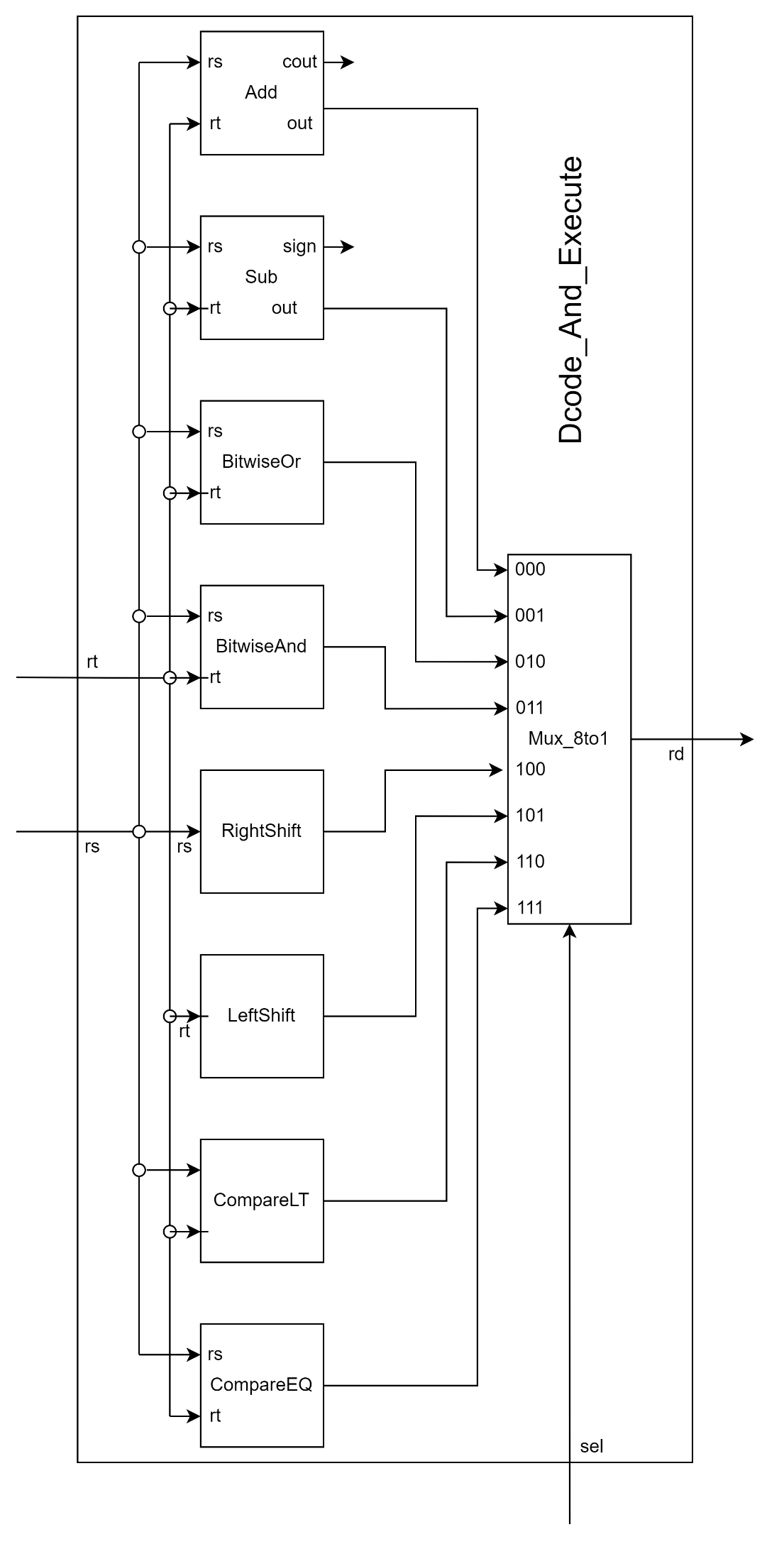
****

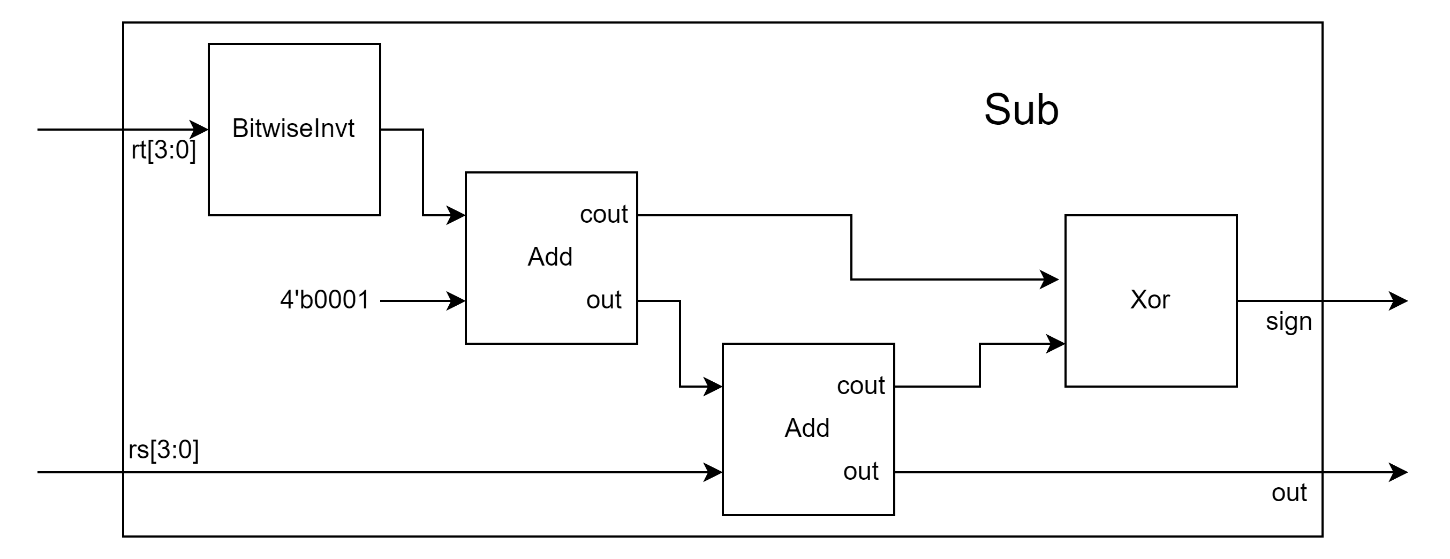
****

****

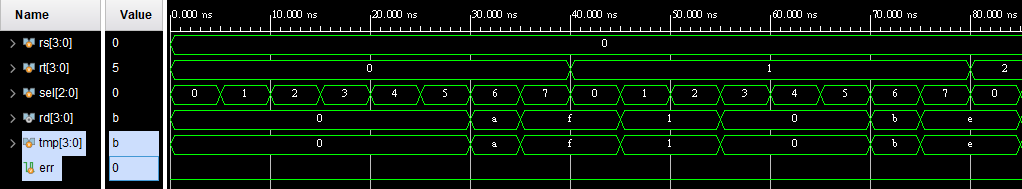
****

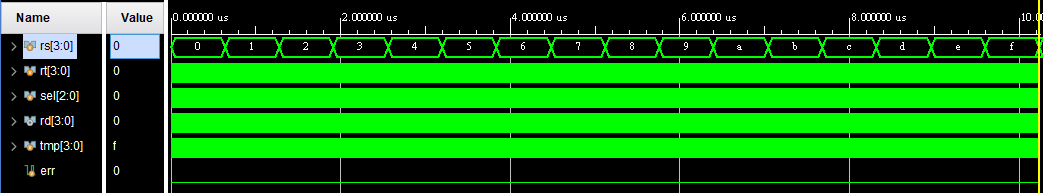
****

****

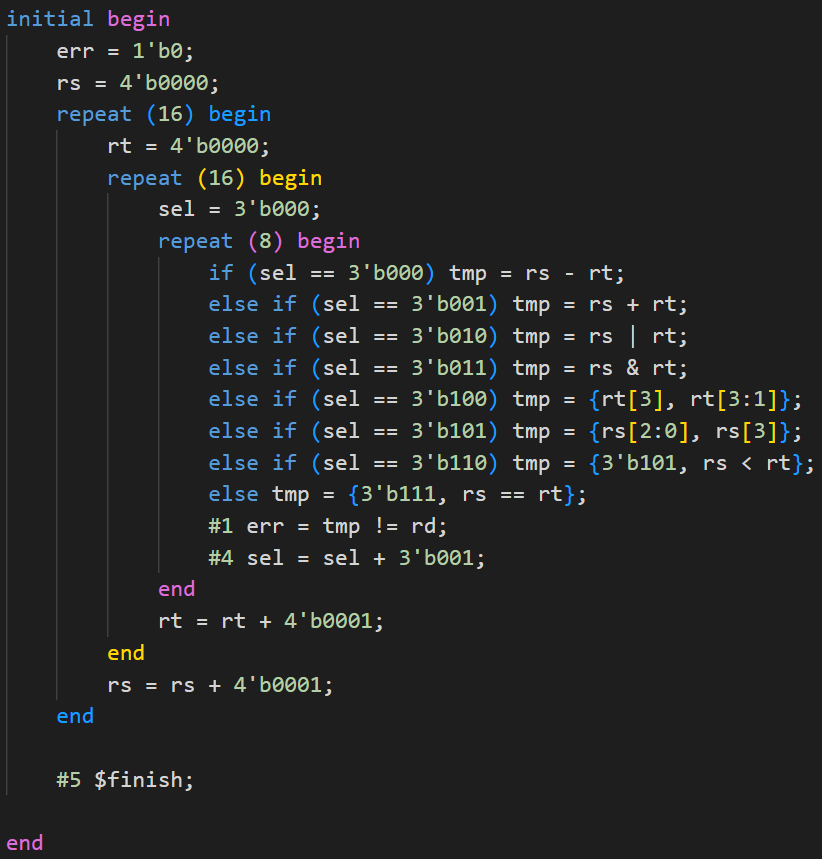
****

1. **Explanation: Some Useful Modules**
2. **Majority:** Count the carry out
3. **Half\_adder:** Deal with 1-bit addition without carry in
4. **Full\_Adder:** Deal with 1-bit addition with carry in
5. **CompareEQ:** Compare two 4-bit data whether they are equal or not and output **rd**
6. **CLT\_1bit:** Compare two 1-bit data **rs** and **rt** and output 1 if **rs** < **rt**
7. **CLT\_4bits:** Compare two 4-bit data **rs** and **rt** and output 1 if **rs** < **rt**
8. **CompareLT:** Combine the result from **CLT\_4bits** and output **rd**
9. **LeftShift:** Use four **Copy** modules to shift **rs**
10. **RightShift:** Use four **Copy** modules to shift **rt**
11. **BitwiseAnd:** Use four **And** modules to realize it
12. **BitwiseOr:** Use four **Or** modules to realize it
13. **BitwiseInvt:** Use four **Not** modules to realize it
14. **Add:** Use a **Half\_Adder** module and three **Full\_Adder** modules to make this ripple carry adder
15. **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**)
16. **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
17. **Testbench**

****

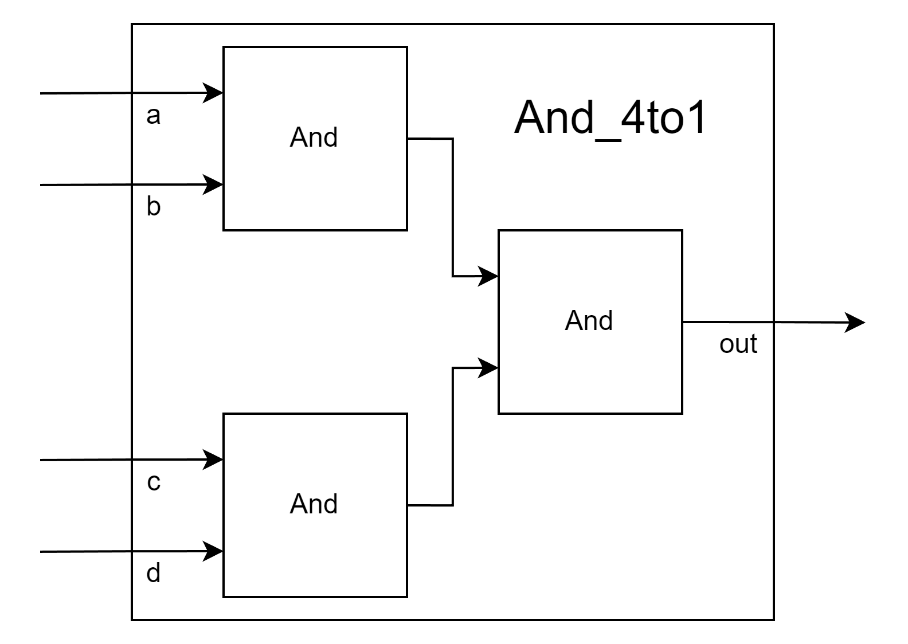
****

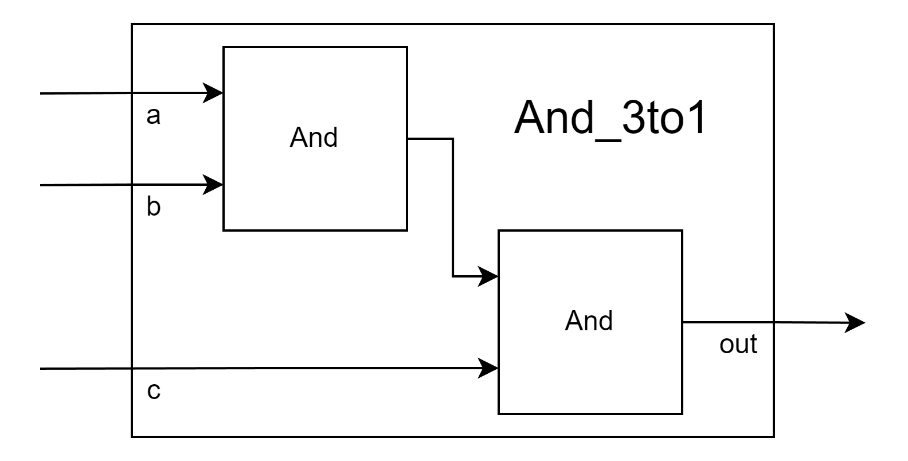
I set the simulation time to 1,0000,0000 ns and use three layers of loop (for **rs**, **rt**, **sel**) to go through every input pattern. And I also define a register called **err**. The testbench will check every result by my behavioral-level code. If the result from the gate-level circuit is different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

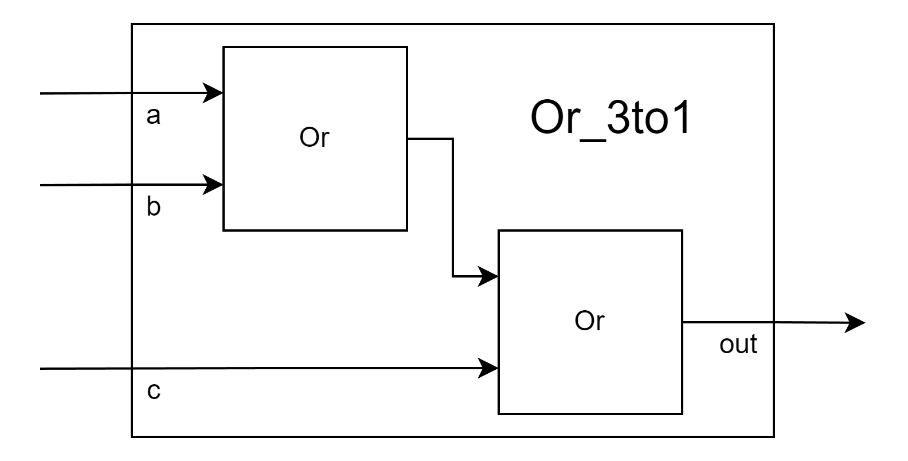
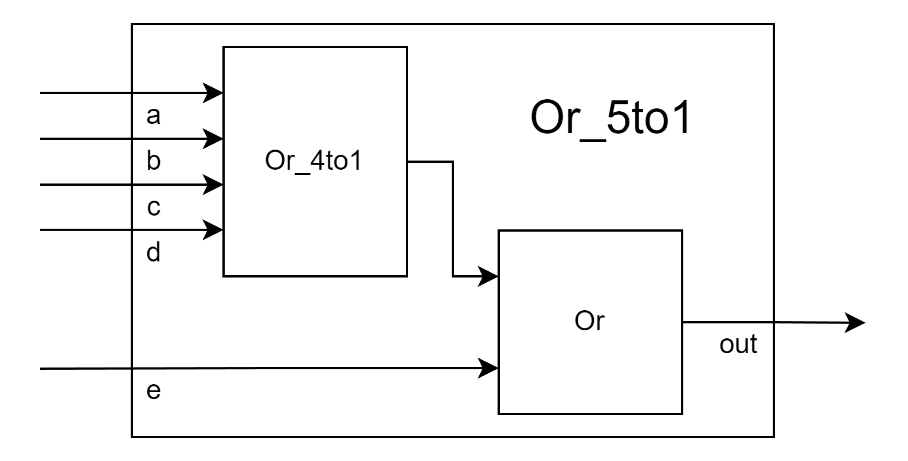
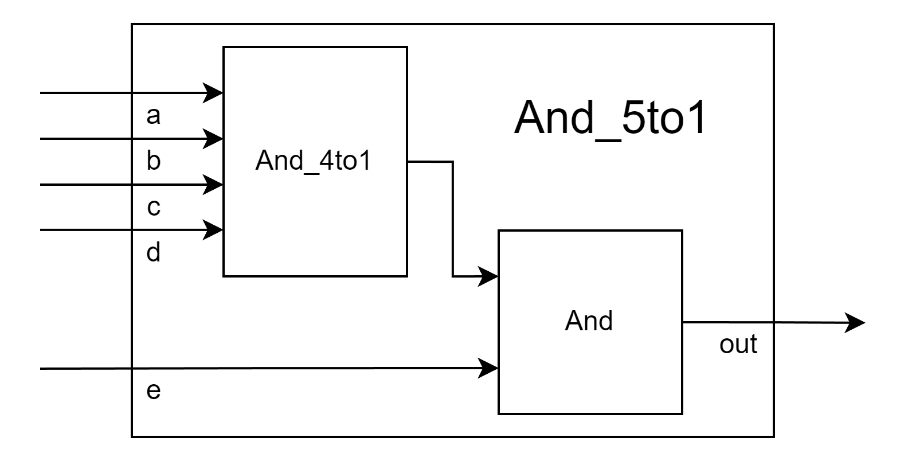
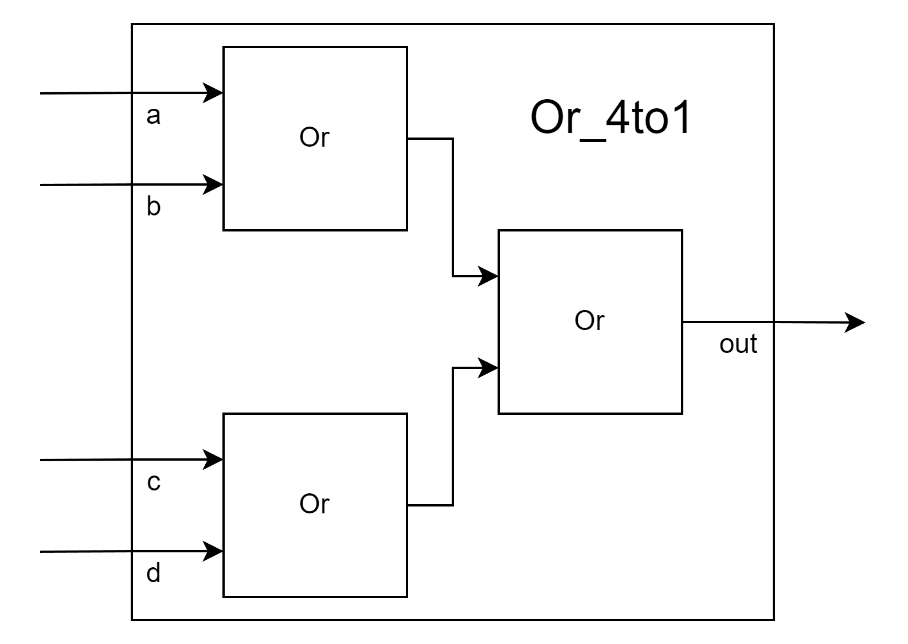
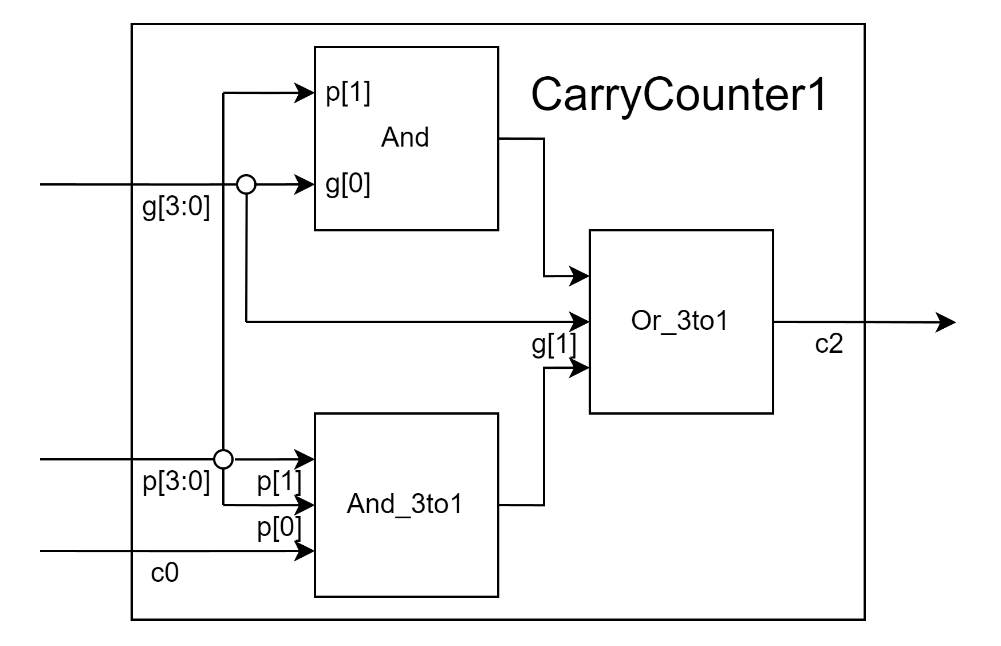
****

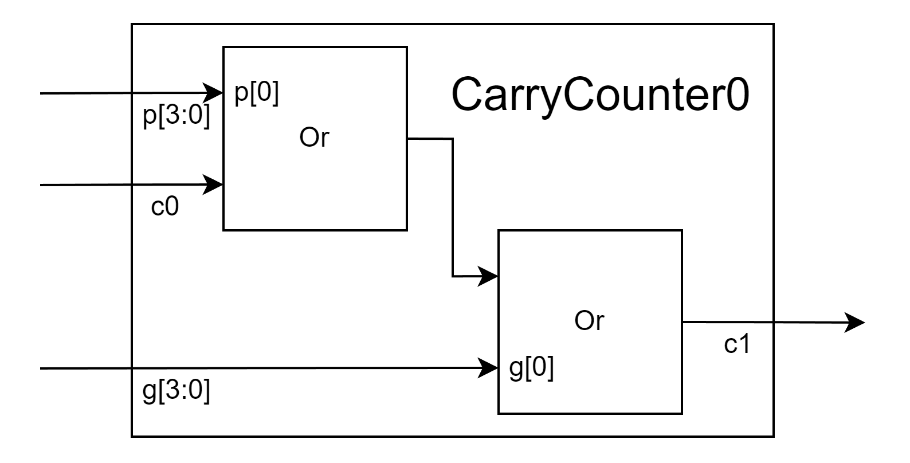
Using behavioral-level code to help me check the result

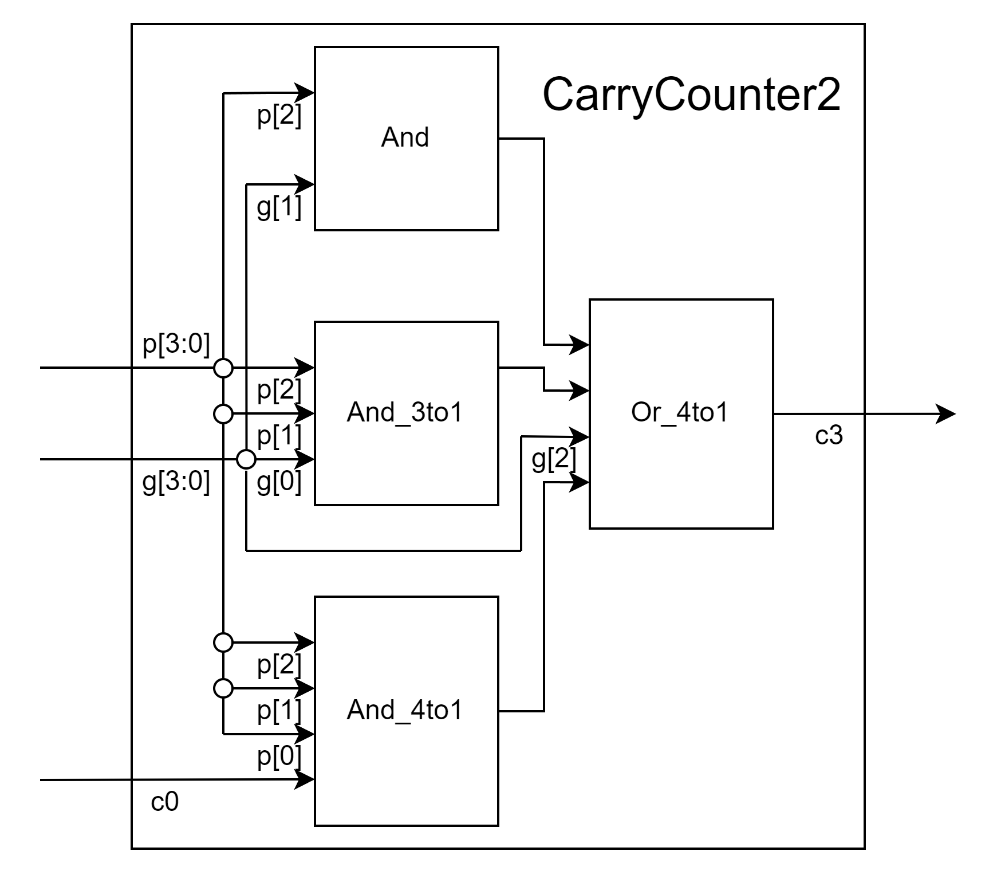
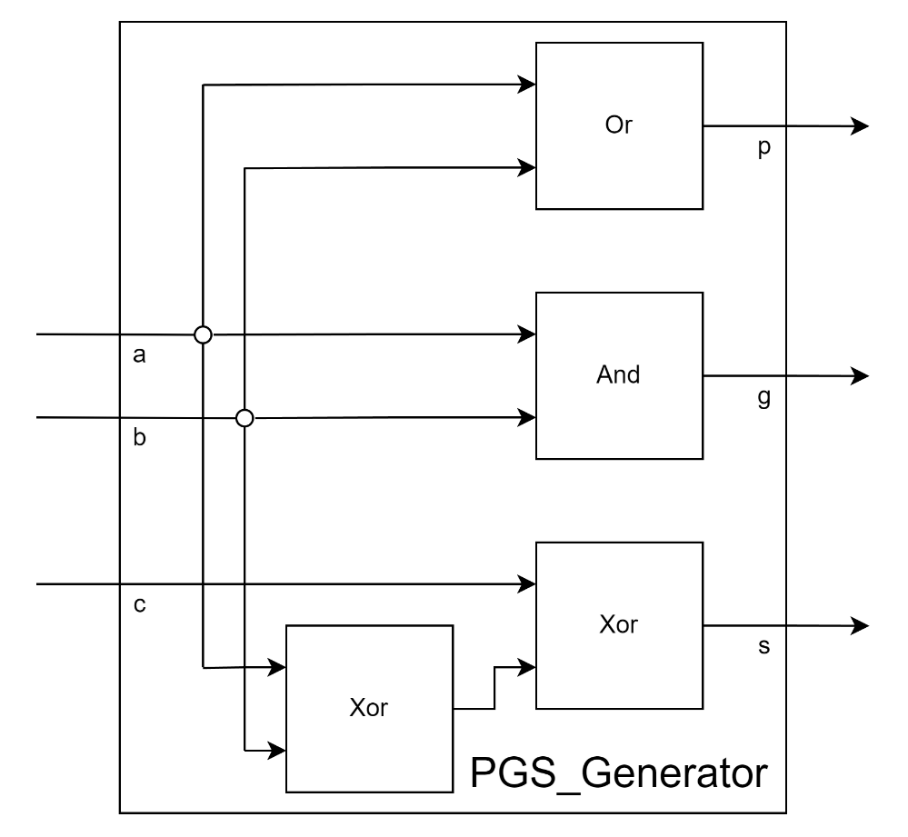
1. **Advanced: 8-bit carry-lookahead (CLA) Adder**
2. **Block Diagram**

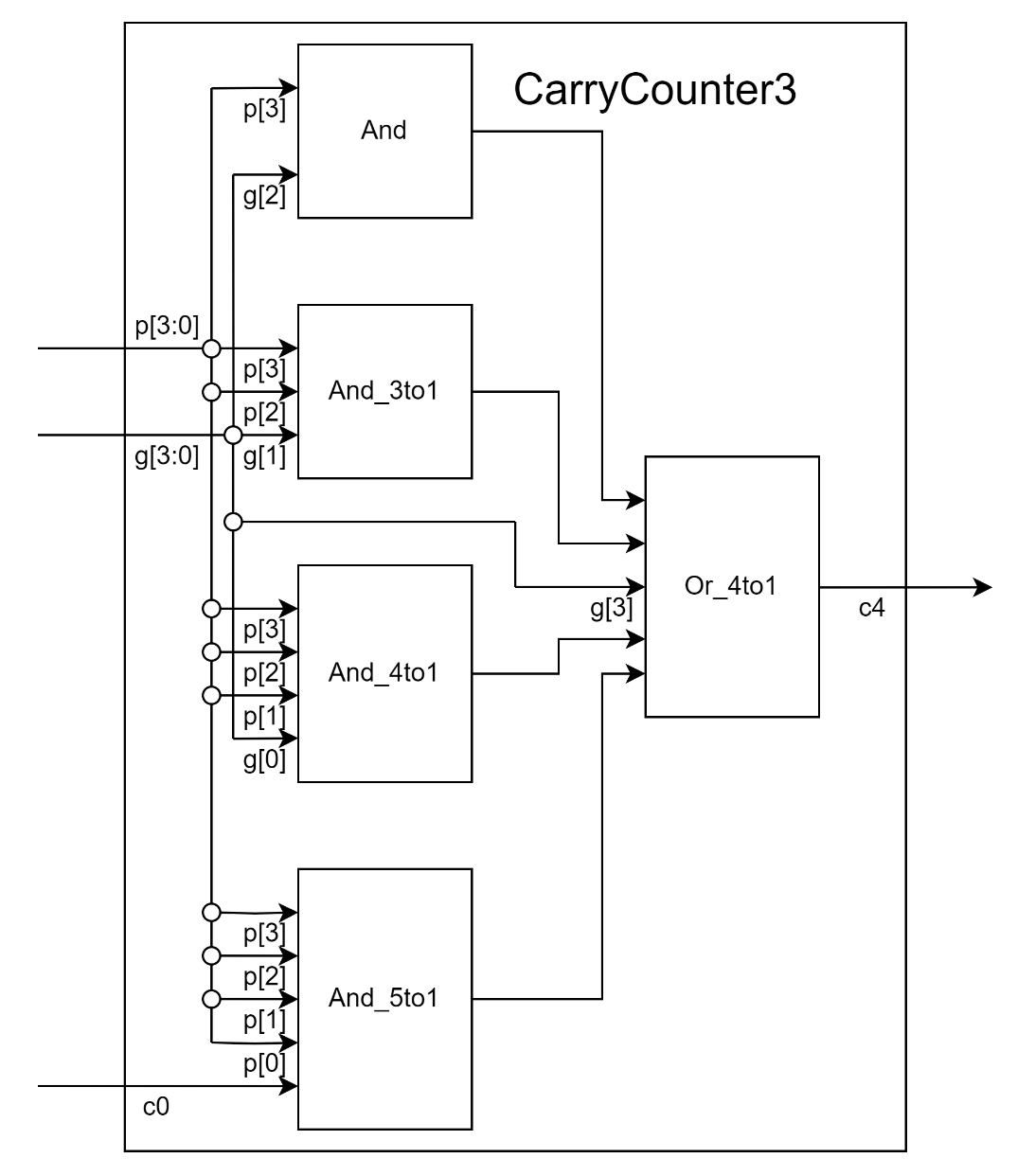
****

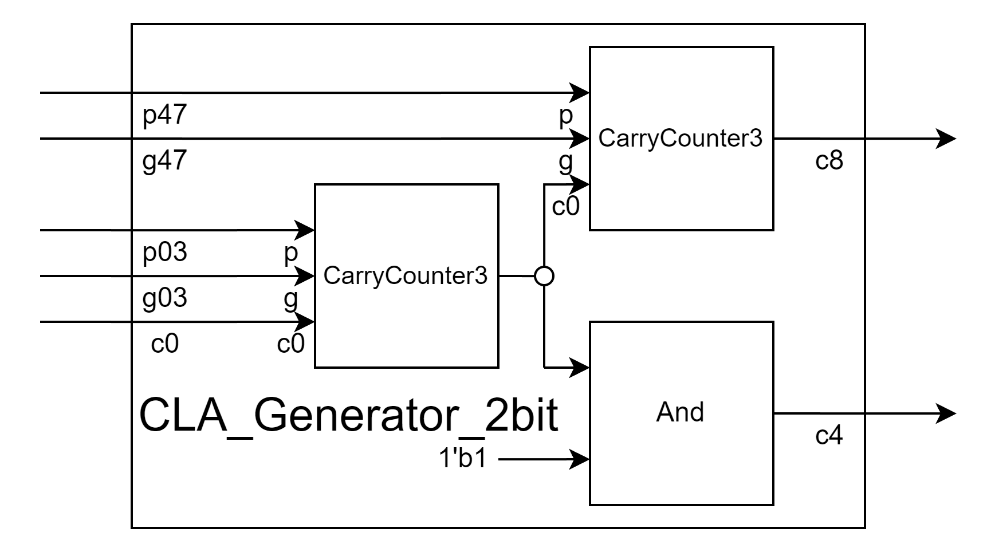


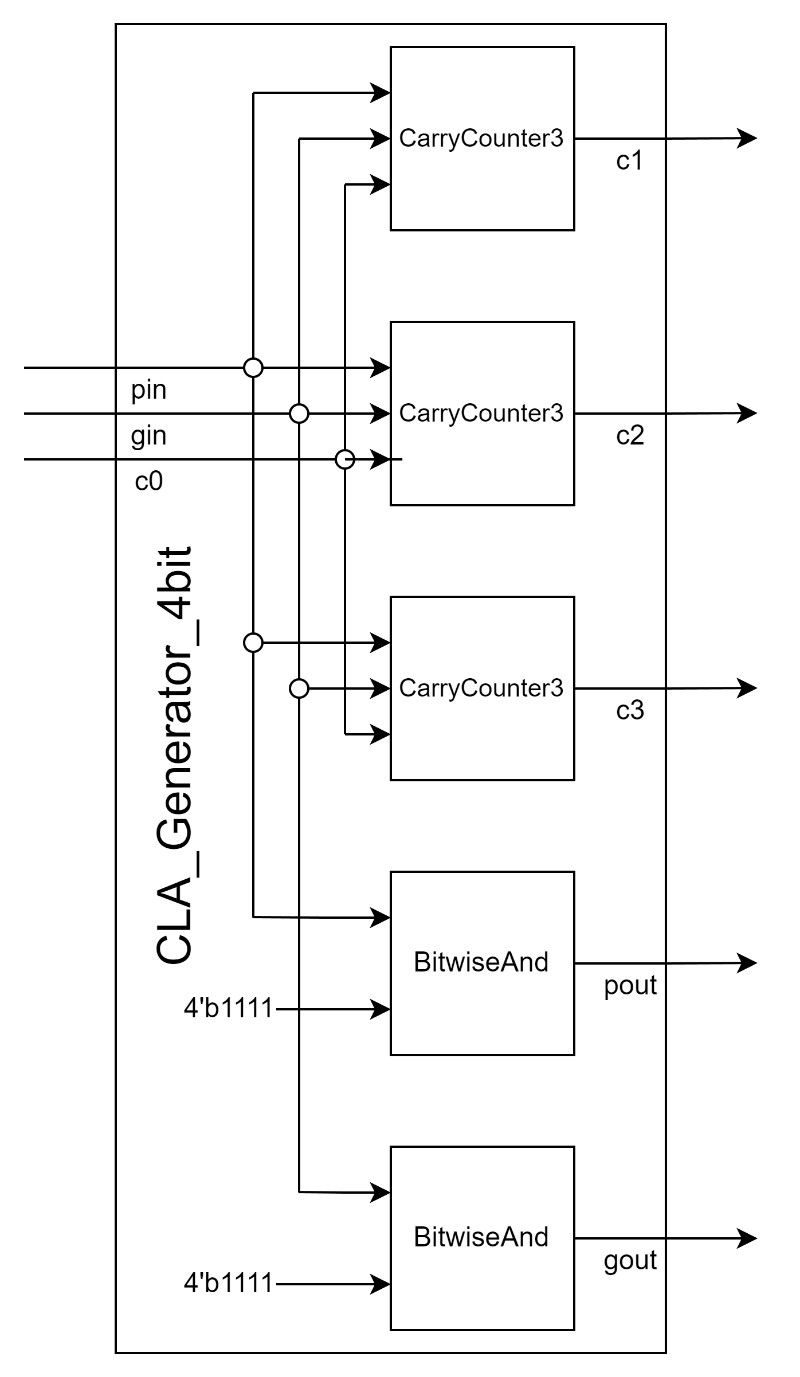
****

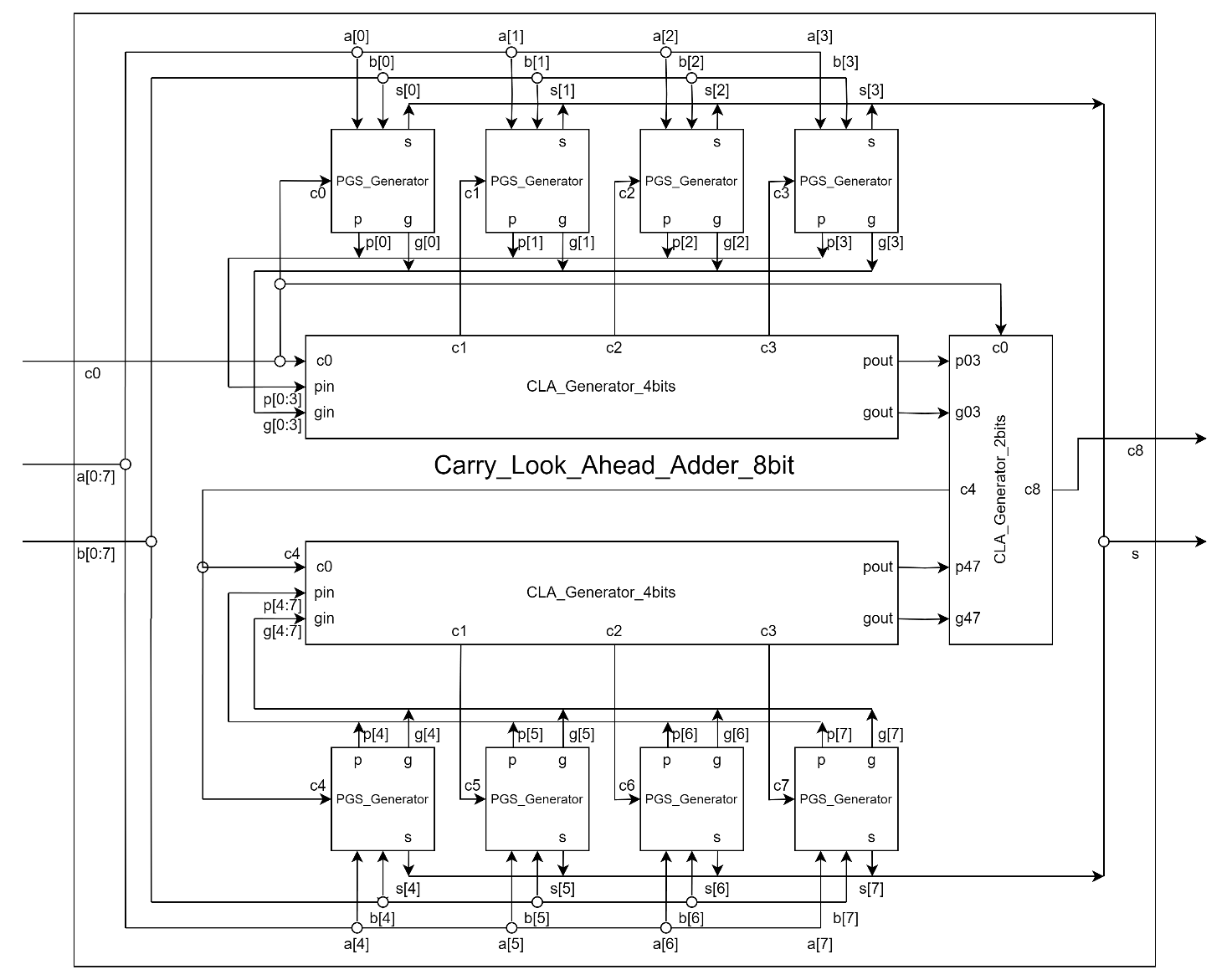
****

****

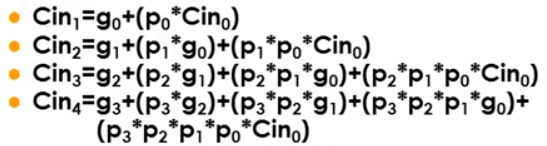
****

****

****

****

1. **Explanation**

****

* + sourse: <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

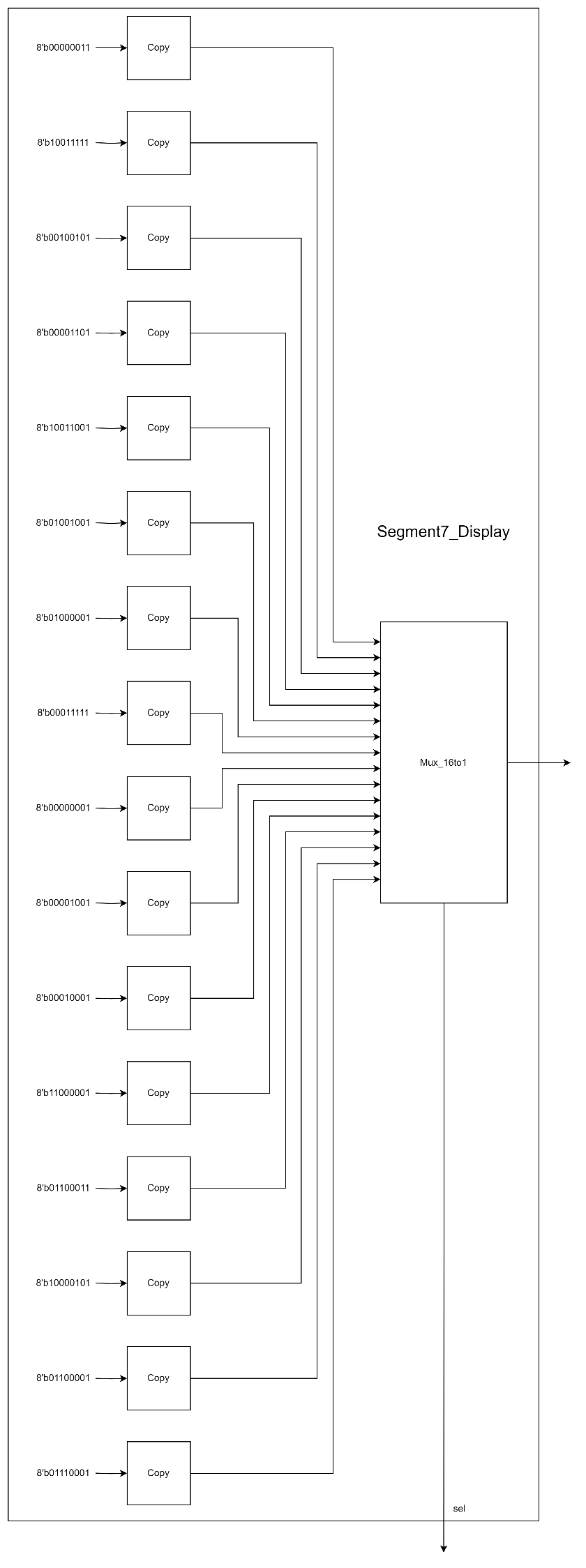
**Carry\_Look\_Ahead\_Adder\_8bit:** add two 8-bit data by carry-look-ahead method

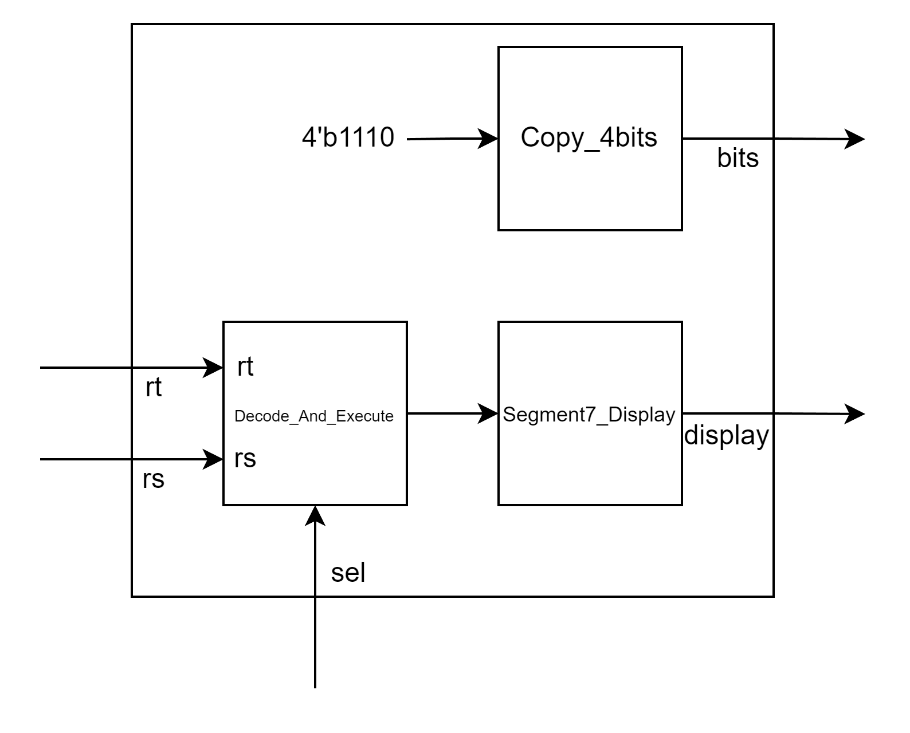
1. **Testbench**

As **DAE Problem**, 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.

1. **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.

1. **Decode and execute (FPGA)**
2. **Block Diagram**

****

1. **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.

1. **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. By the way, the tragedy that happened on National holiday reminded me that it is important to back up data.