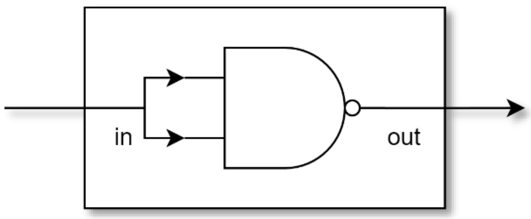
Hardware Design and Lab: Lab1

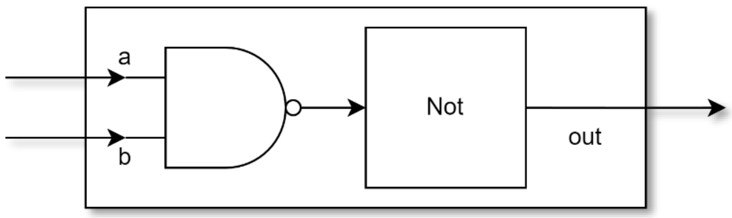
111060013 EECS 26' 劉祐廷

1. Basic

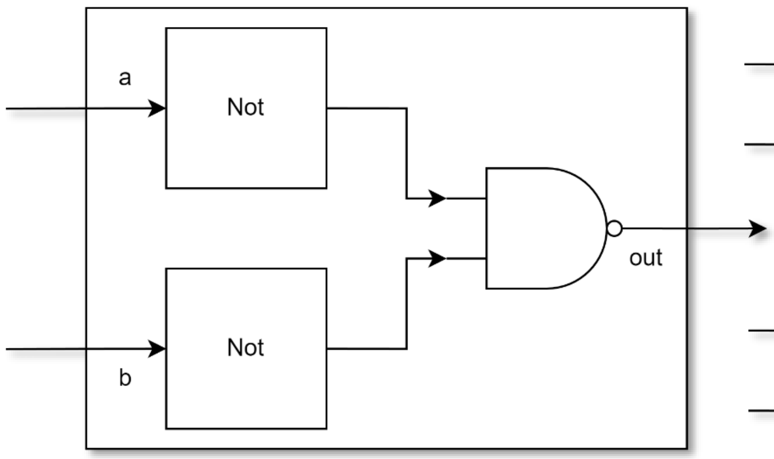
A. Basic Question 1: Block Diagram



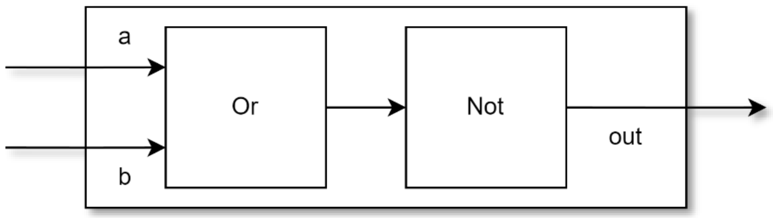
Not



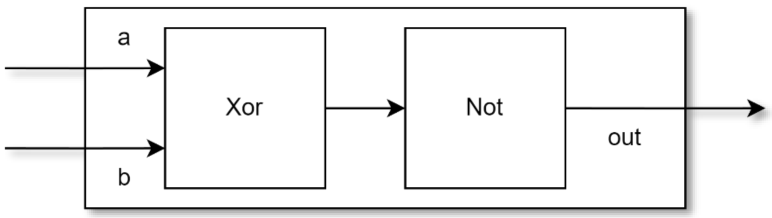
And



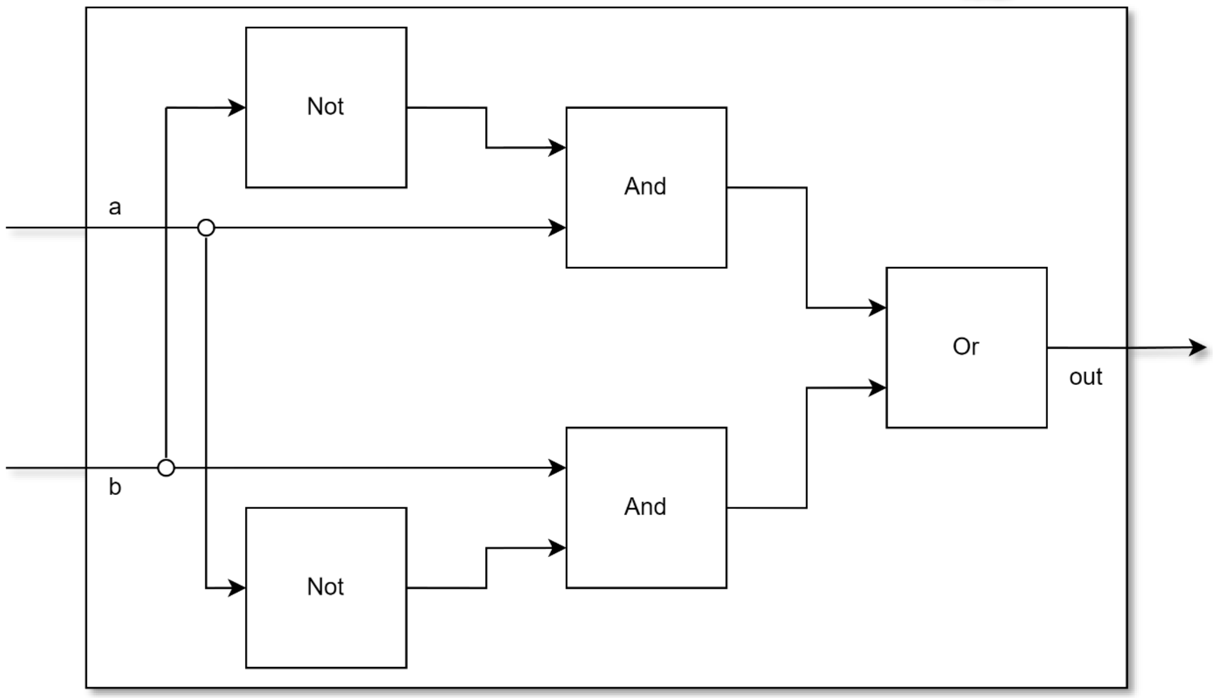
Or



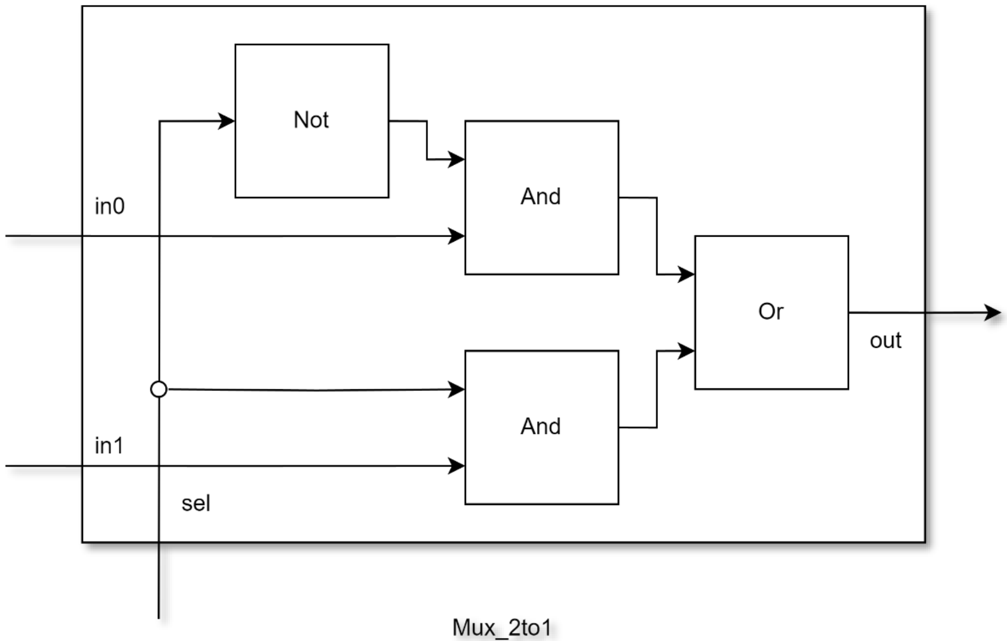
Nor



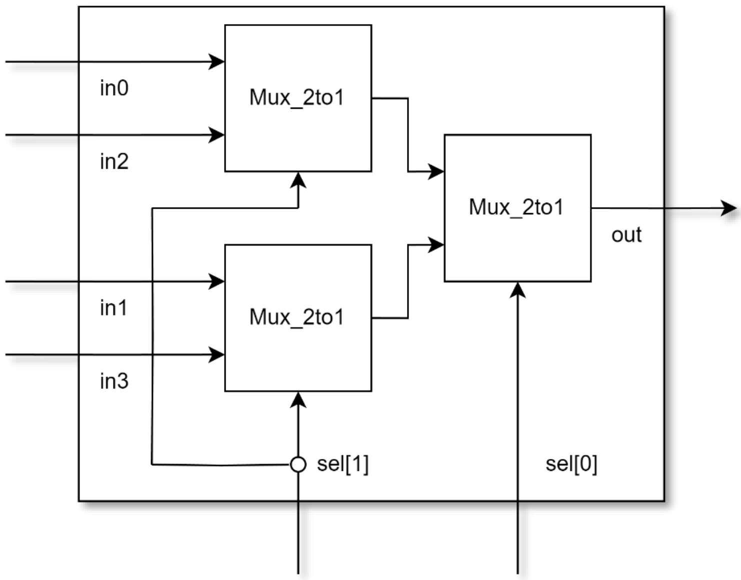
Xnor



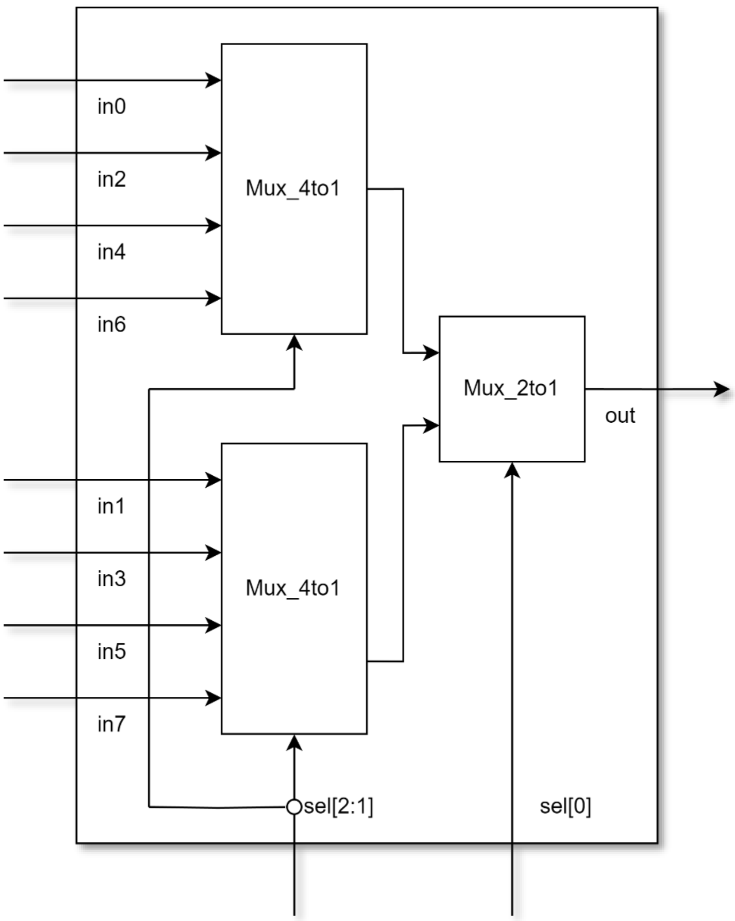
Xor



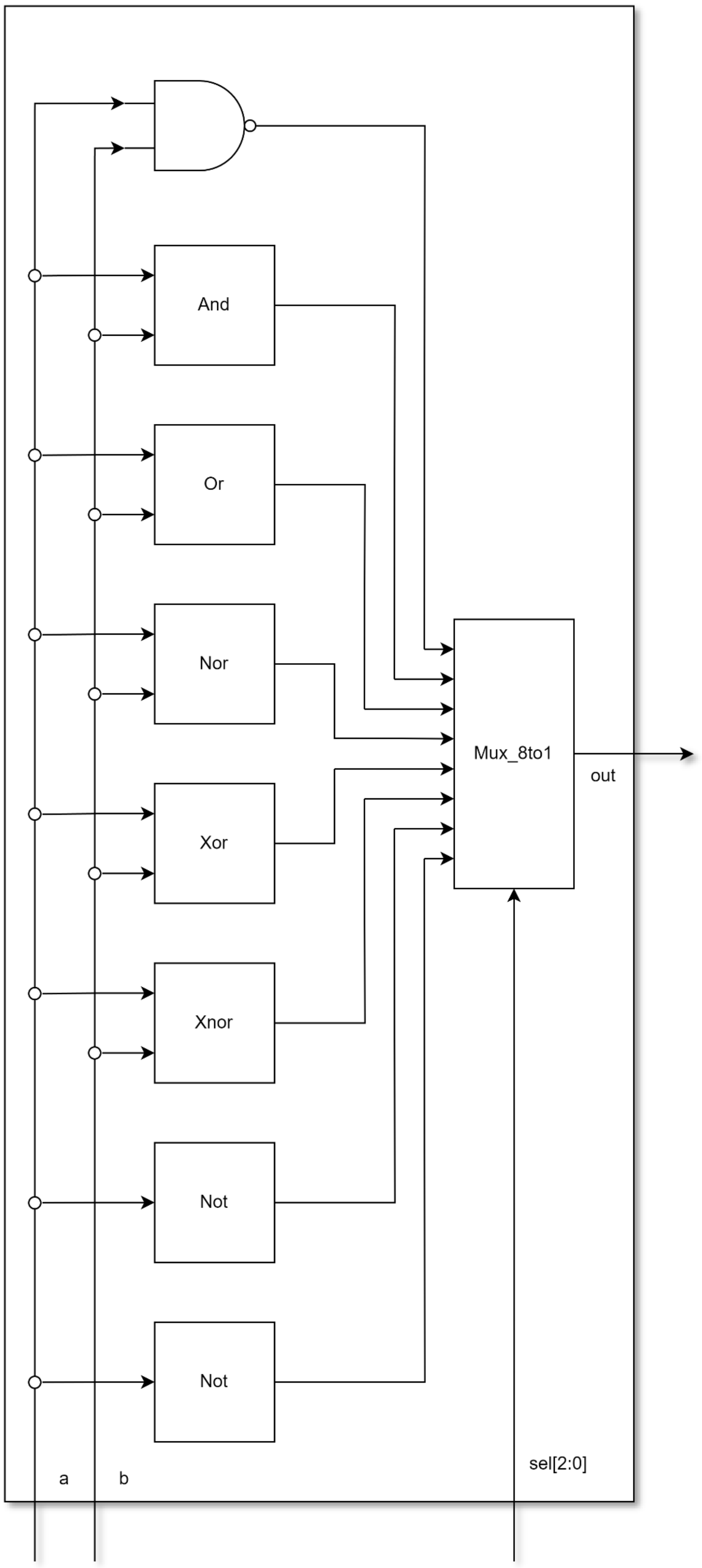
Mux_2to1



Mux_4to1



Mux_8to1

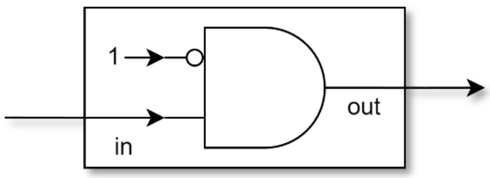

And
Or
Nor
Xor
Xnor
Not
Not
Mux_8to1
out
sel[2:0]
a
b

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

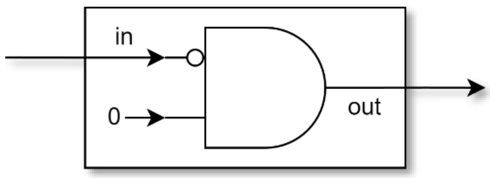
The most significant difference between them is that a half adder can only deal with the situation without carry in; however, a full adder can handle the situation with carry in.

2. Advanced: Decode and execute

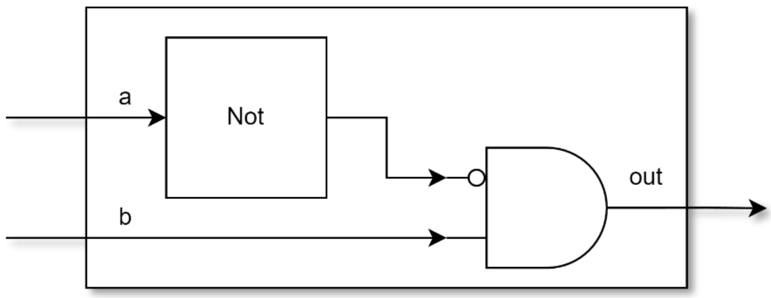
A. Block Diagram: Basic Modules



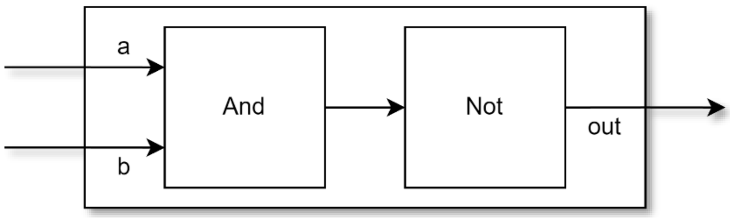
Not



Copy



And

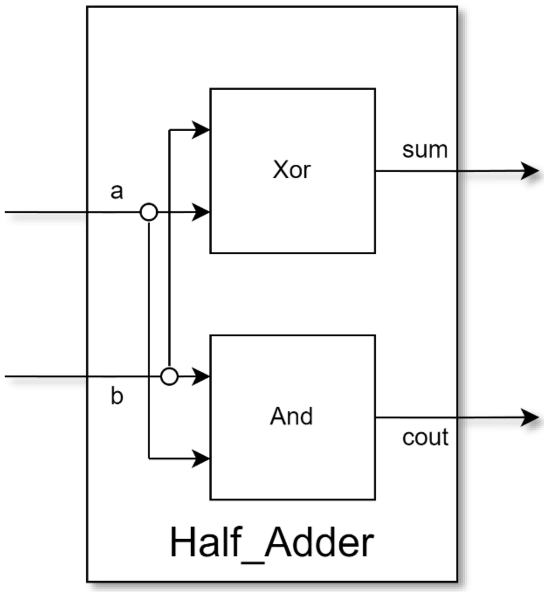


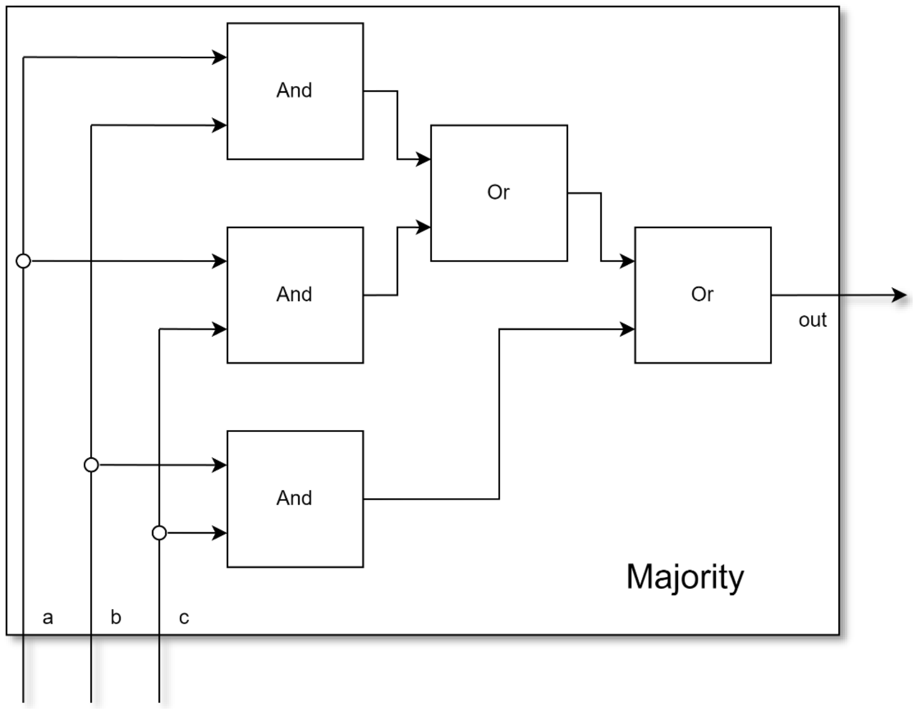
Nand

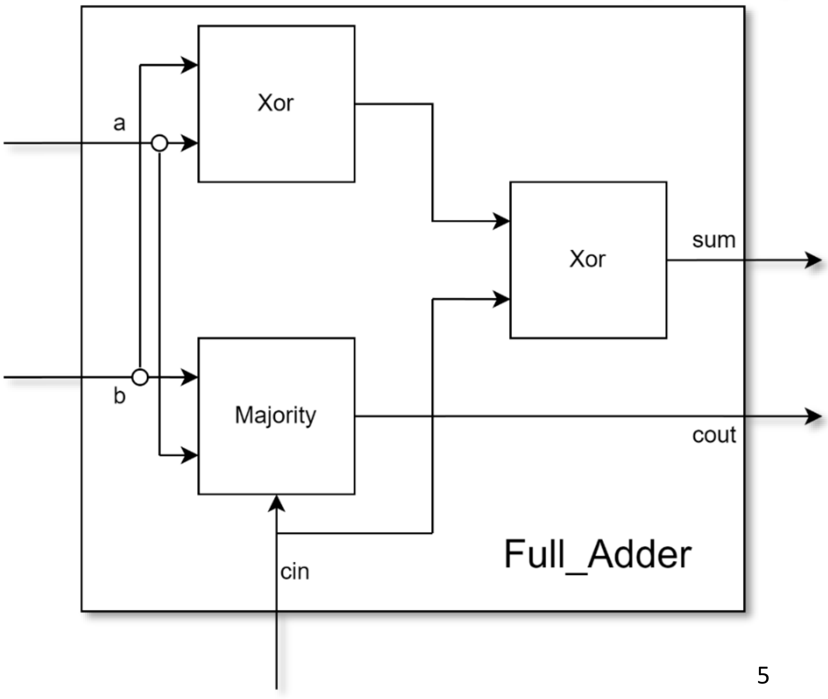
B. Explanation: Basic Modules

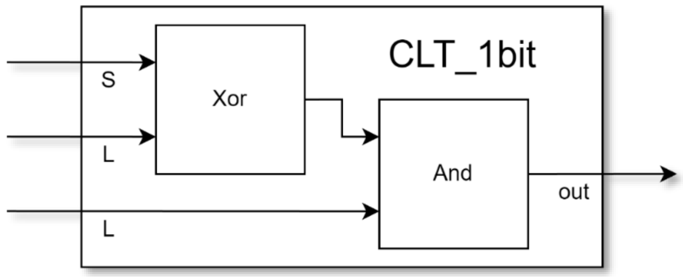
First of all, I drew the truth table of **Universal Gate** to design my own **Not** module and **Copy** module. And then I combined a **Not** module and an **Universal Gate** to create **And** module. After that, I make a **Nand** module with an **And** module and a **Not** module. The reason why I design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

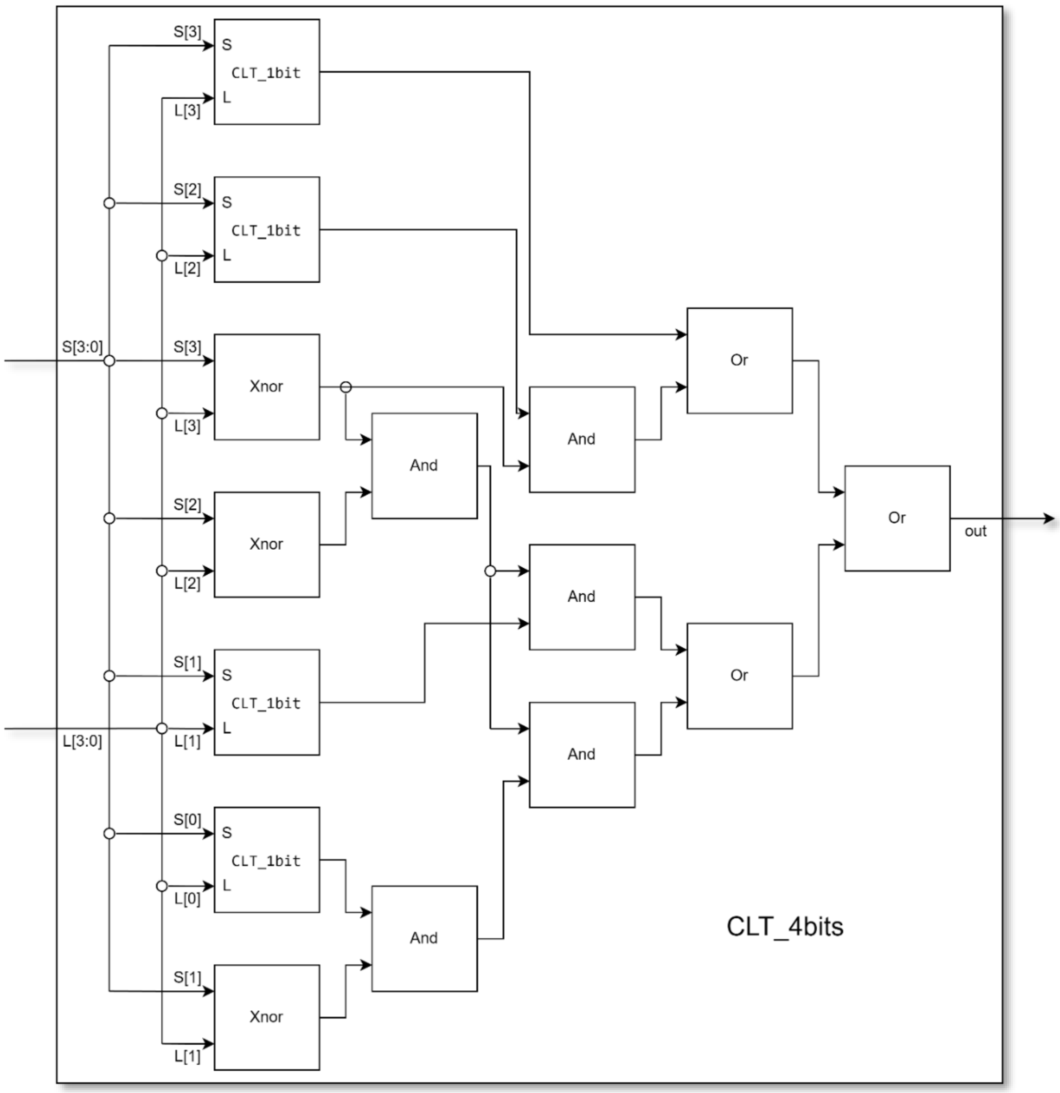
C. Block Diagram: Advanced Modules

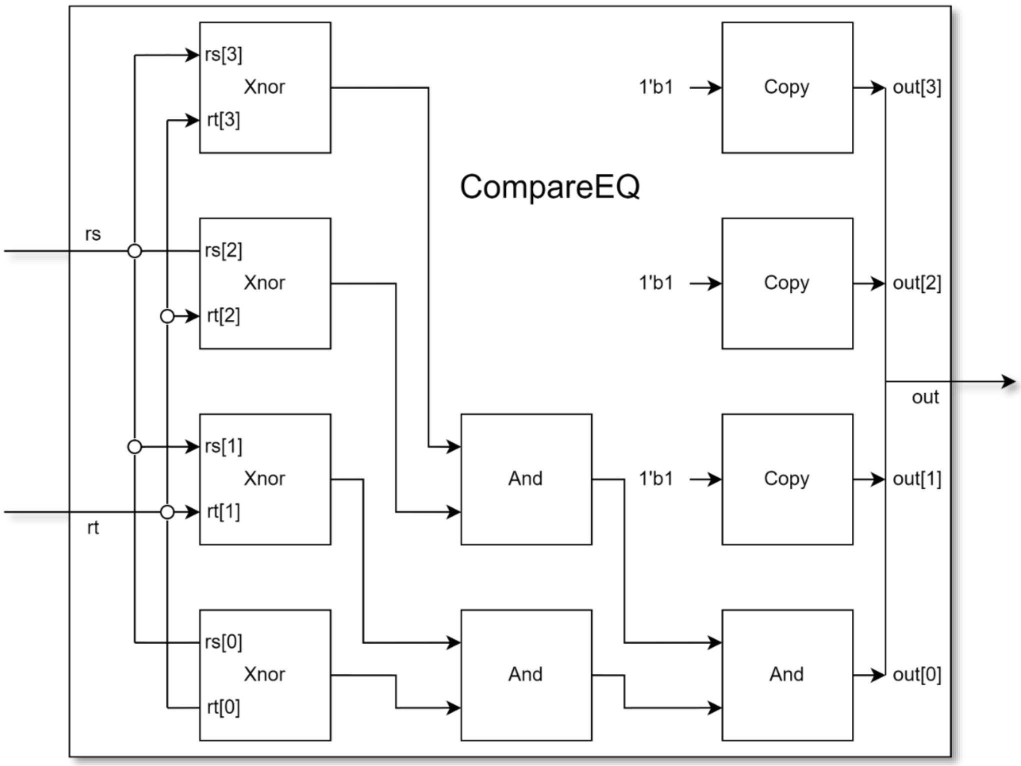


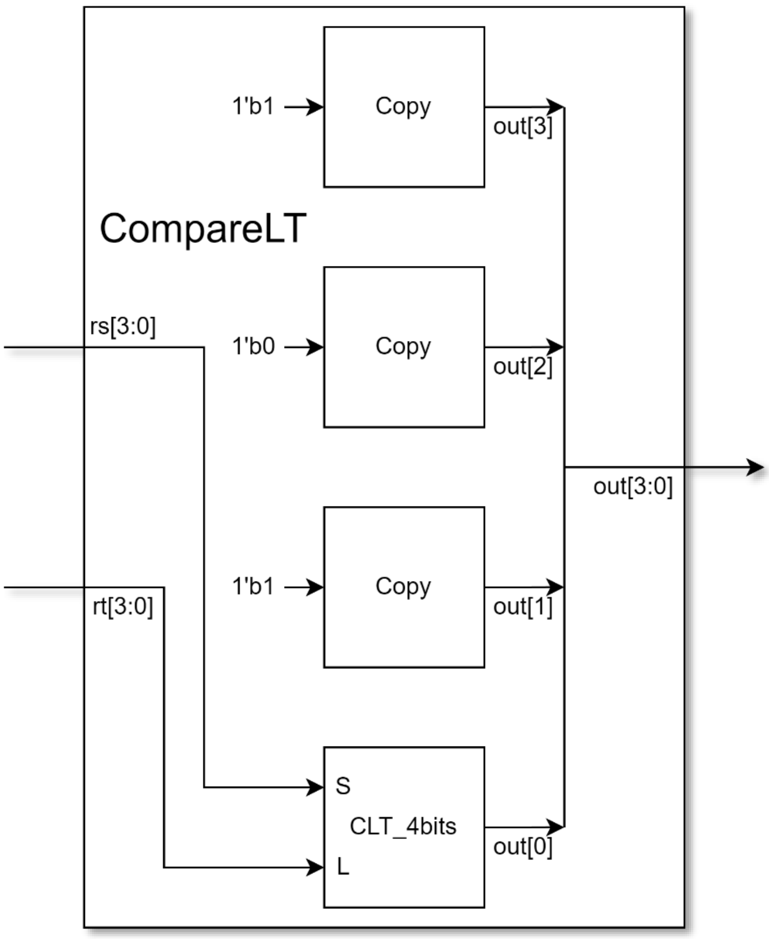


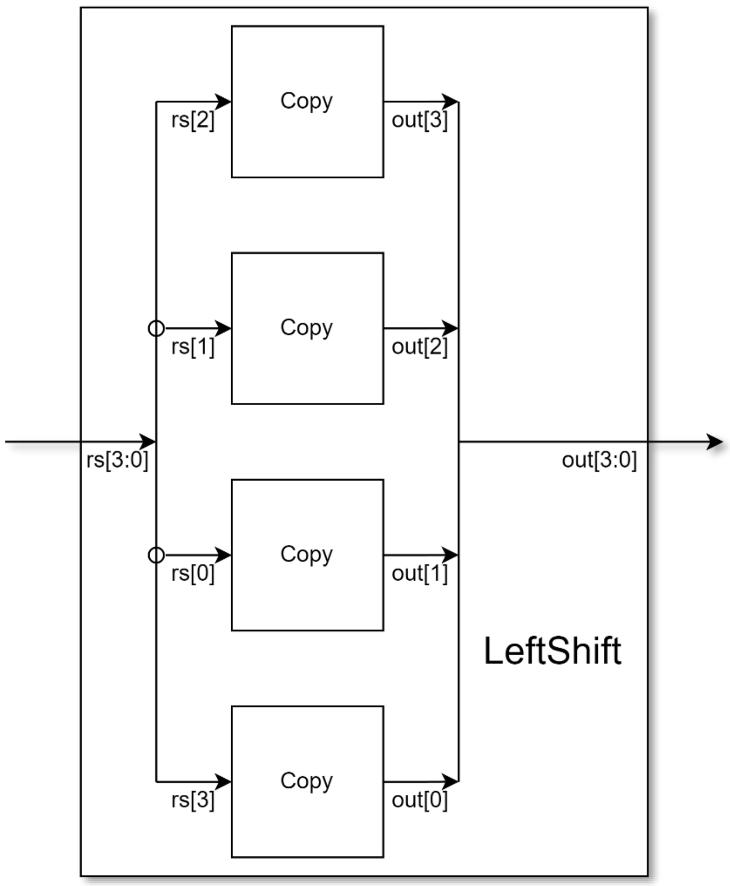


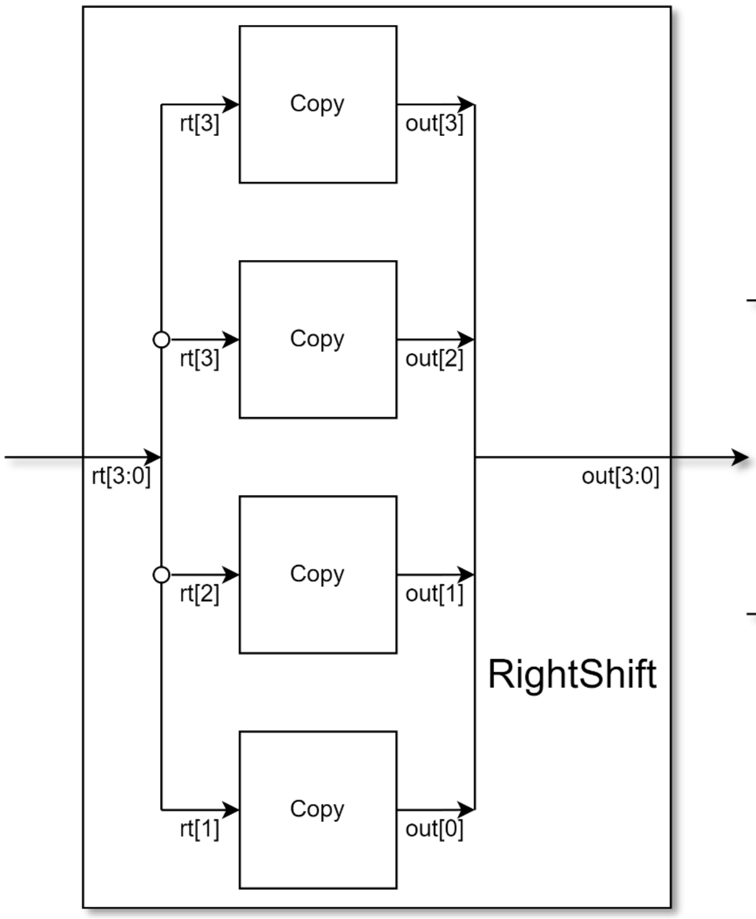


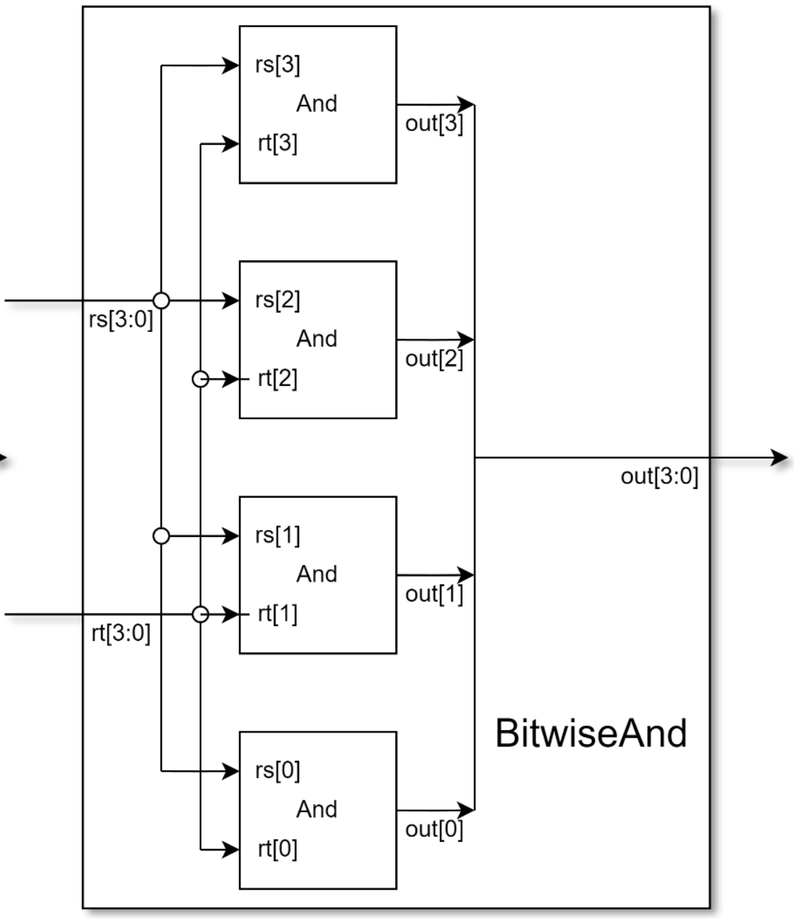

S[3:0]
L[3:0]
S[3]
L[3]
S[2]
L[2]
S[1]
L[1]
S[0]
L[0]
S
L
CLT_1bit
Xnor
And
Or
out
CLT_4bits

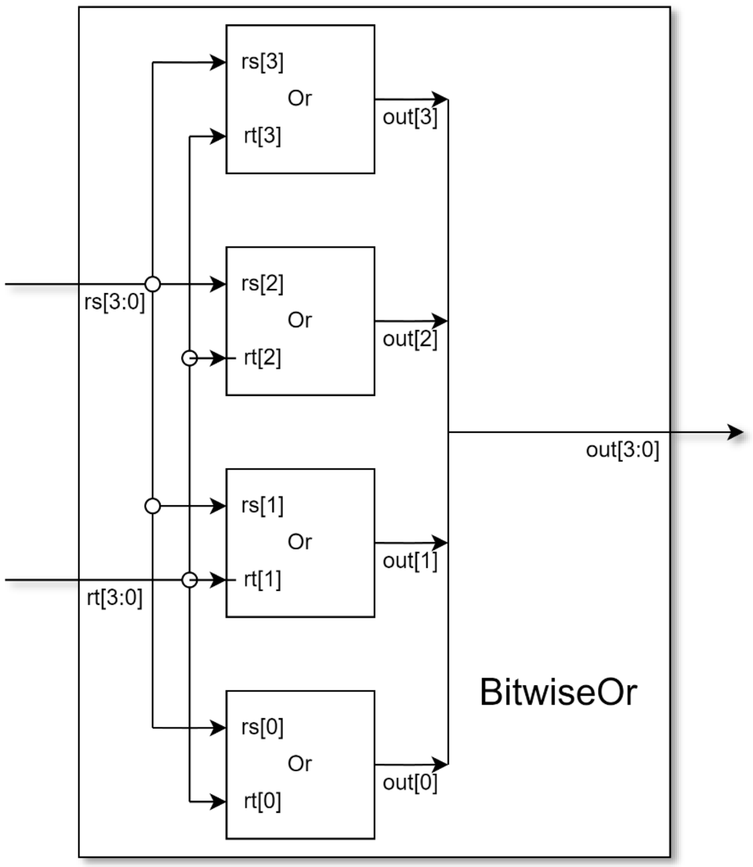

rs
rt
rs[3]
rt[3]
rs[2]
rt[2]
rs[1]
rt[1]
rs[0]
rt[0]
Xnor
And
1'b1
Copy
CompareEQ
out[3]
out[2]
out[1]
out[0]
out

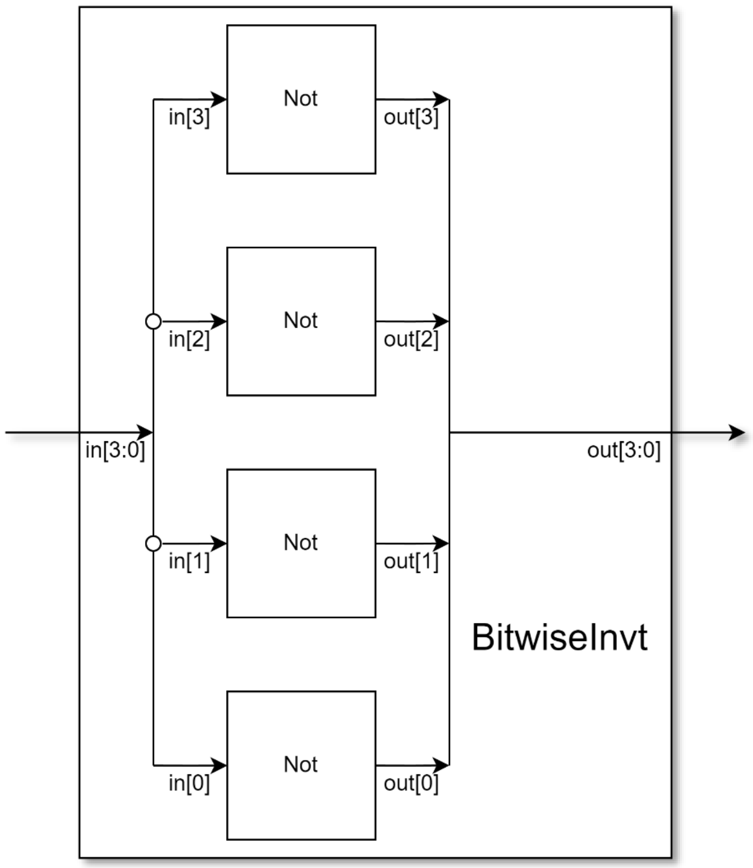

CompareLT
1'b1
Copy
out[3]
1'b0
Copy
out[2]
1'b1
Copy
out[1]
rs[3:0]
rt[3:0]
S
CLT_4bits
L
out[0]
out[3:0]

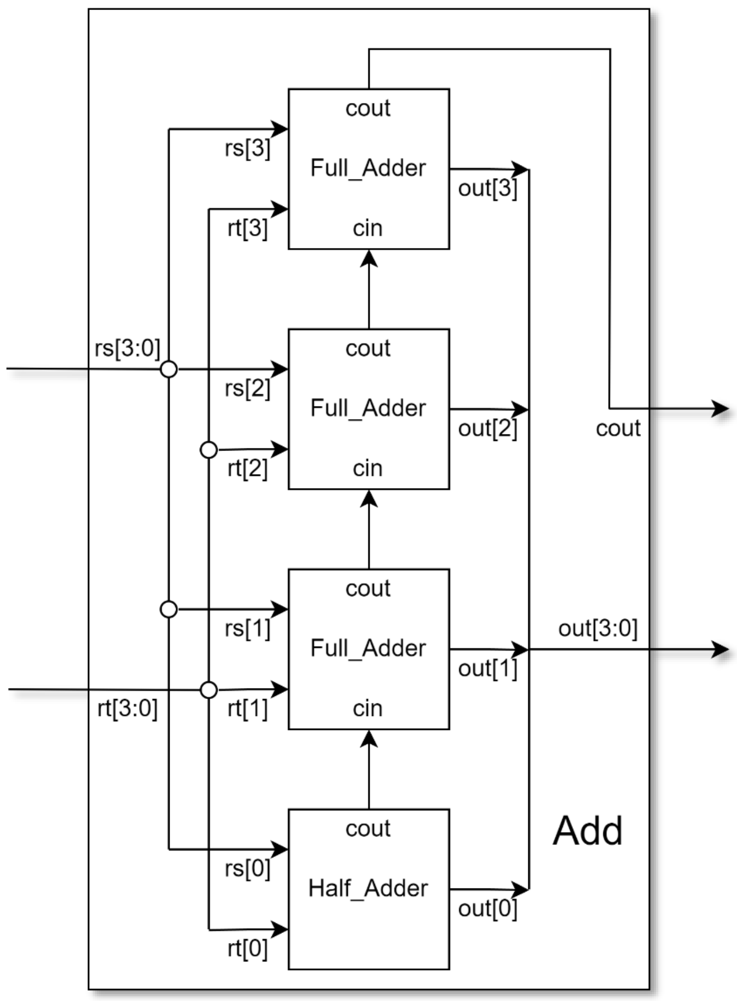

rs[2]
Copy
out[3]
rs[1]
Copy
out[2]
rs[3:0]
out[3:0]
rs[0]
Copy
out[1]
LeftShift
rs[3]
Copy
out[0]

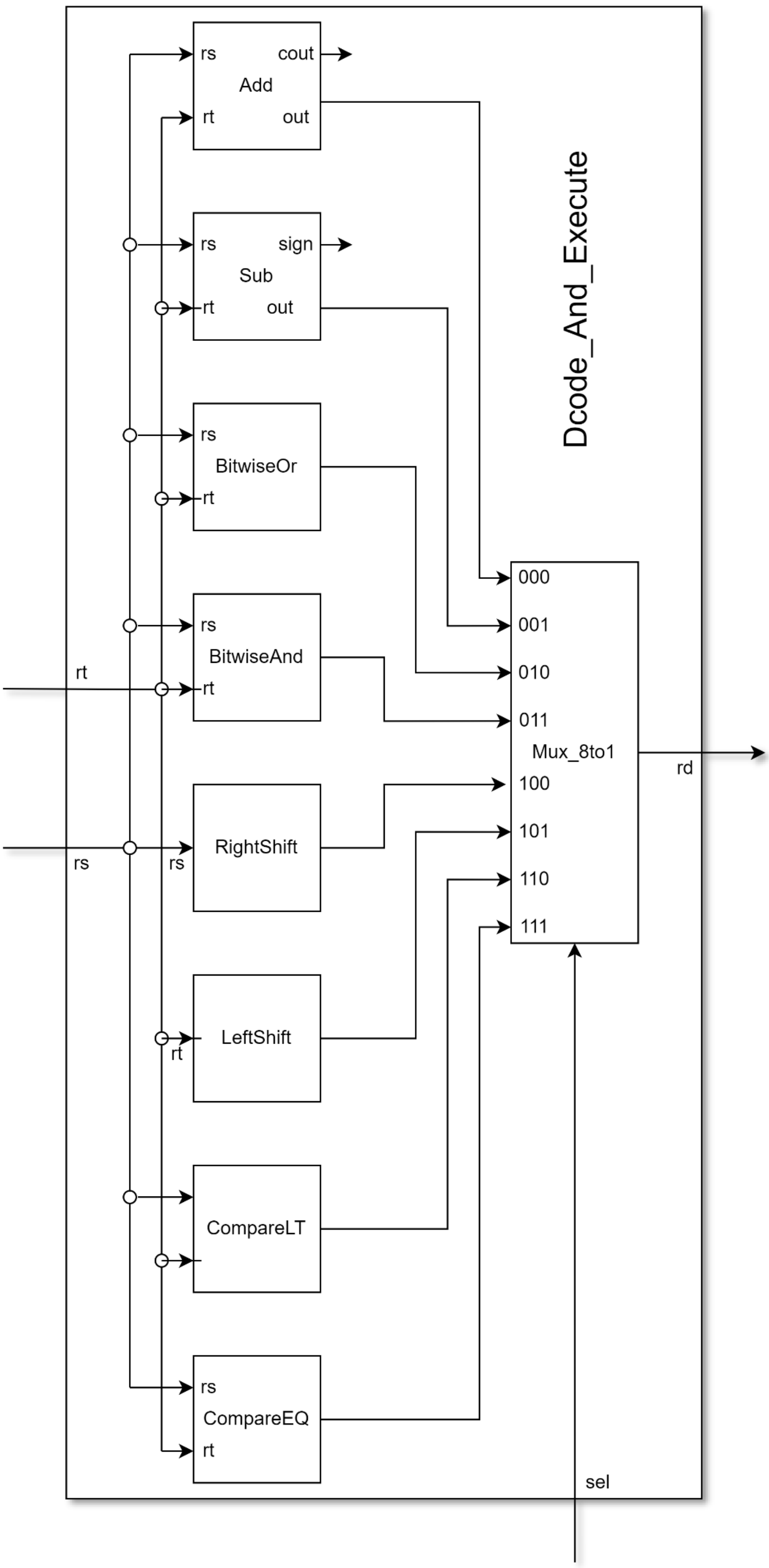

rt[3]
Copy
out[3]
rt[3]
Copy
out[2]
rt[3:0]
out[3:0]
rt[2]
Copy
out[1]
RightShift
rt[1]
Copy
out[0]

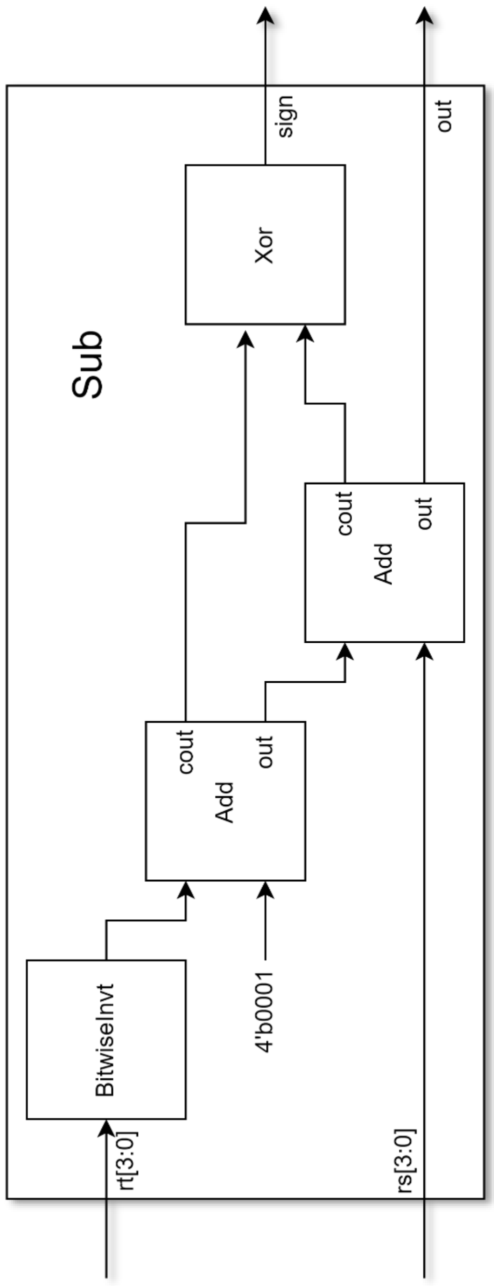

rs[3]
And
rt[3]
out[3]
rs[3:0]
rs[2]
And
rt[2]
out[2]
out[3:0]
rs[1]
And
rt[1]
out[1]
rt[3:0]
BitwiseAnd
rs[0]
And
rt[0]
out[0]


rs[3]
Or
rt[3]
out[3]
rs[2]
Or
rt[2]
out[2]
rs[3:0]
rs[1]
Or
rt[1]
out[1]
rt[3:0]
out[3:0]
rs[0]
Or
rt[0]
out[0]
BitwiseOr


Not
in[3]
out[3]
Not
in[2]
out[2]
in[3:0]
out[3:0]
Not
in[1]
out[1]
BitwiseInvt
Not
in[0]
out[0]


cout
Full_Adder
rs[3]
rt[3]
cin
out[3]
rs[3:0]
cout
Full_Adder
rs[2]
rt[2]
cin
out[2]
cout
cout
Full_Adder
rs[1]
rt[1]
cin
out[1]
rt[3:0]
out[3:0]
cout
Half_Adder
rs[0]
rt[0]
out[0]
Add

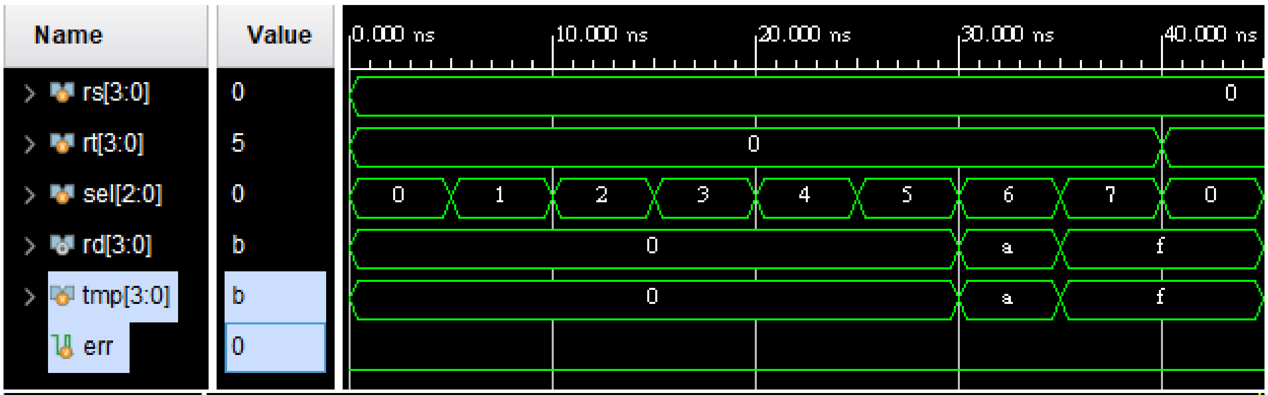

Dcode_And_Execute
rs
cout
Add
rt
out
rs
sign
Sub
rt
out
rs
BitwiseOr
rt
rs
BitwiseAnd
rt
rt
RightShift
rs
rs
LeftShift
rt
CompareLT
rs
CompareEQ
rt
000
001
010
011
Mux_8to1
100
101
110
111
rd
sel

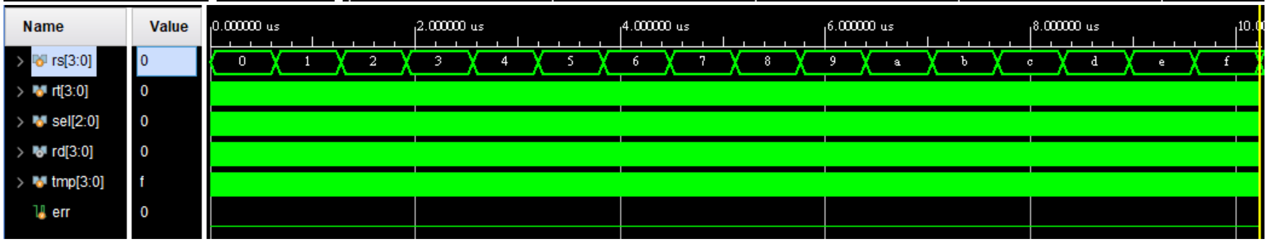

Sub
sign
out
Xor
cout
out
Add
cout
out
Add
BitwiseInvt
4'b0001
rt[3:0]
rs[3:0]

D. Explanation: Some Useful Modules

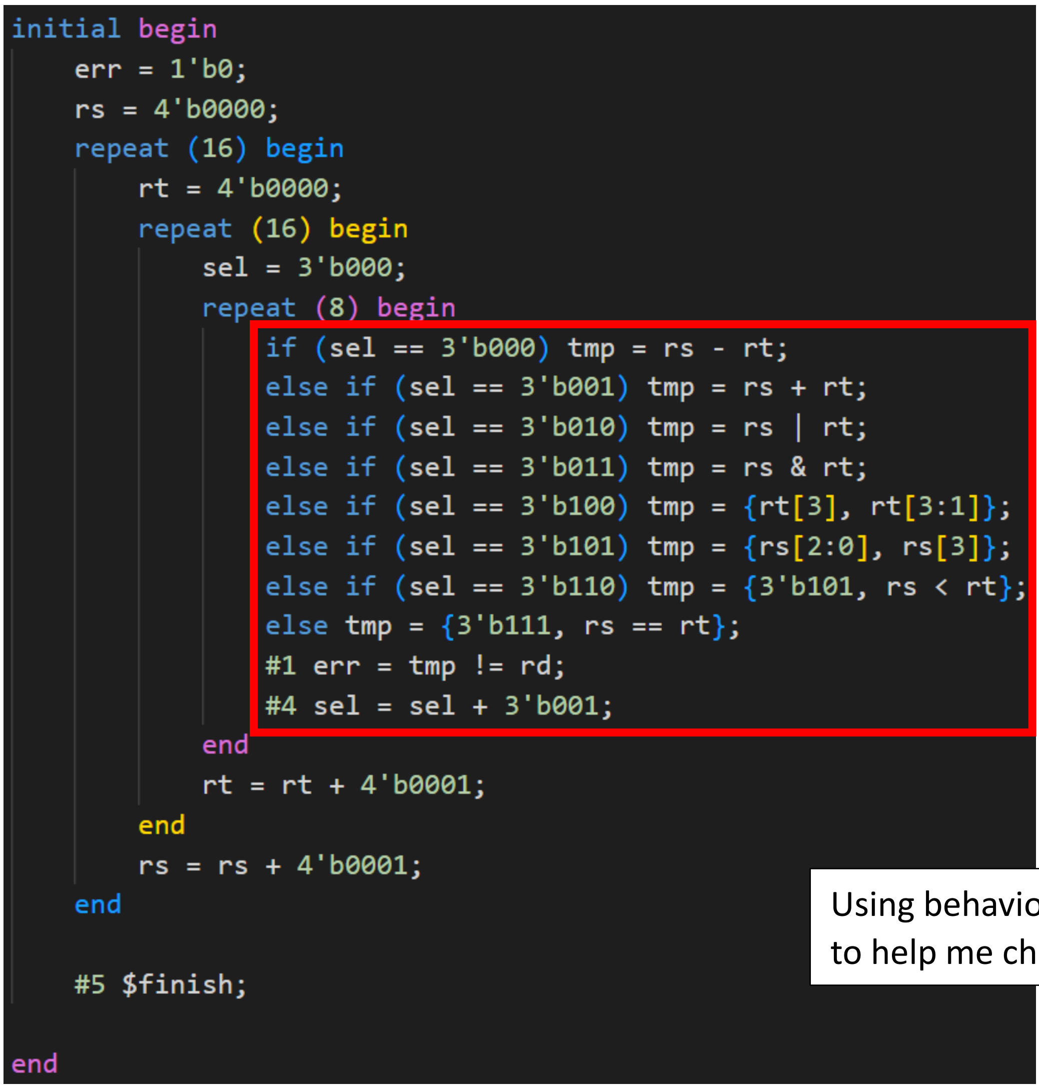
a. **Majority:** Count the carry out
b. **Half_adder:** Deal with 1-bit addition without carry in
c. **Full_Adder:** Deal with 1-bit addition with carry in
d. **CompareEQ:** Compare two 4-bit data whether they are equal or not and output **rd**
e. **CLT_1bit:** Compare two 1-bit data **rs** and **rt** 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.

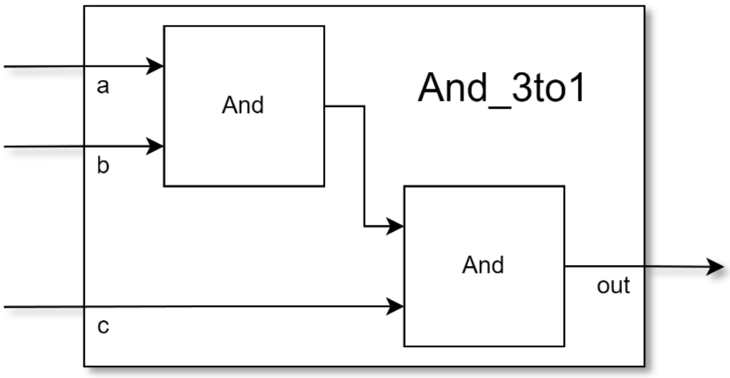


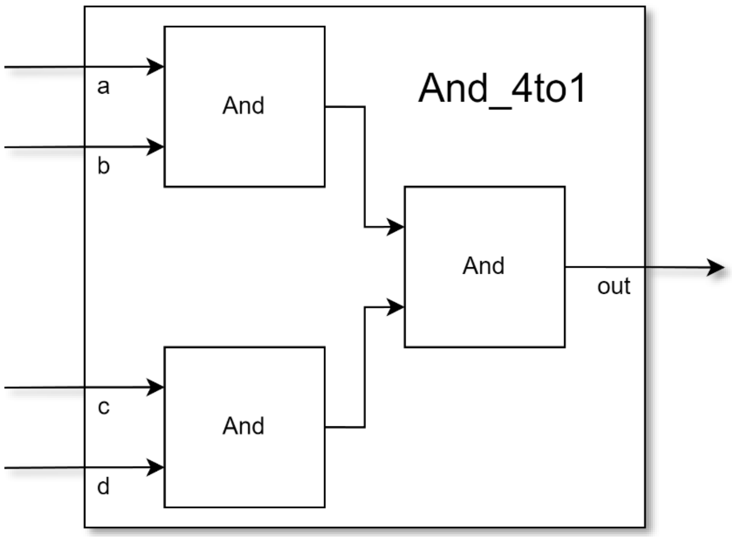
```
initial begin
    err = 1'b0;
    rs = 4'b0000;
    repeat (16) begin
        rt = 4'b0000;
        repeat (16) begin
            sel = 3'b000;
            repeat (8) begin
                if (sel == 3'b000) tmp = rs - rt;
                else if (sel == 3'b001) tmp = rs + rt;
                else if (sel == 3'b010) tmp = rs | rt;
                else if (sel == 3'b011) tmp = rs & rt;
                else if (sel == 3'b100) tmp = {rt[3], rt[3:1]};
                else if (sel == 3'b101) tmp = {rs[2:0], rs[3]};
                else if (sel == 3'b110) tmp = {3'b101, rs < rt};
                else tmp = {3'b111, rs == rt};
                #1 err = tmp != rd;
                #4 sel = sel + 3'b001;
            end
            rt = rt + 4'b0001;
        end
        rs = rs + 4'b0001;
    end

    #5 $finish;

end
```

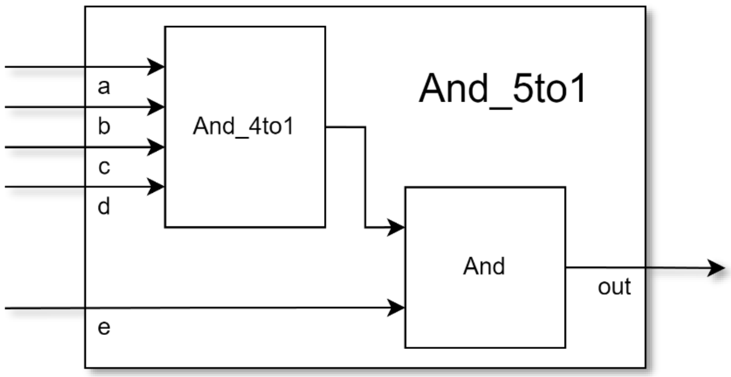
Using behavioral-level code to help me check the result

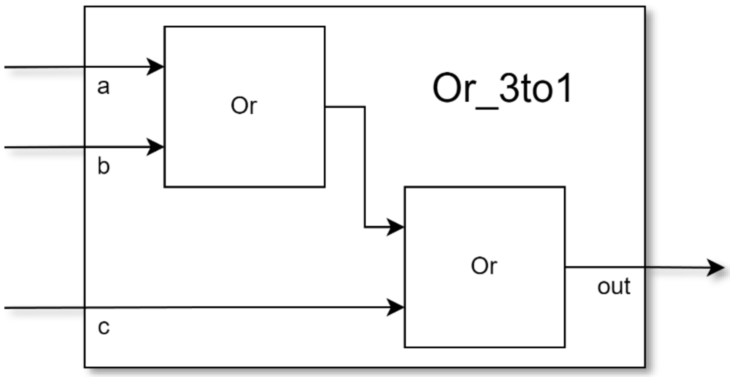
3. Advanced: 8-bit carry-lookahead (CLA) Adder

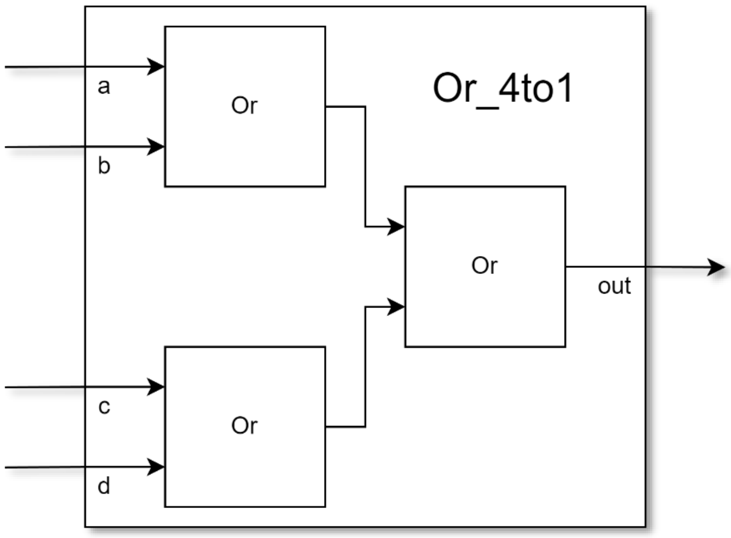
A. Block Diagram

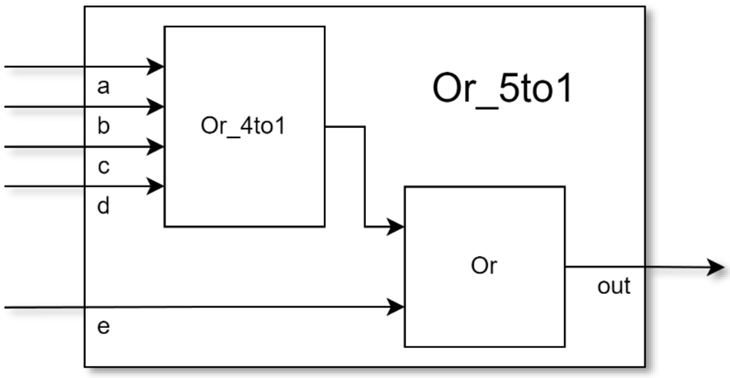


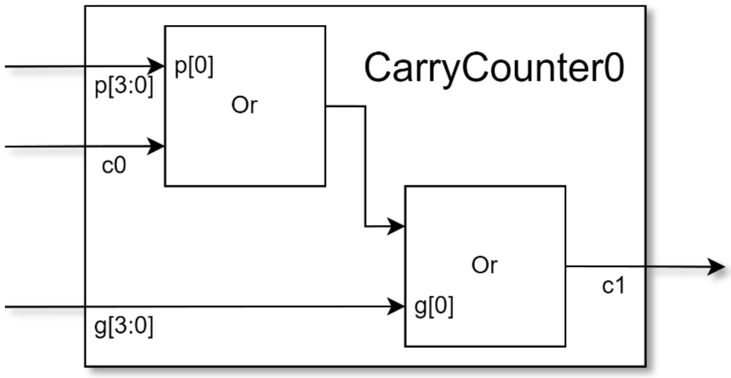


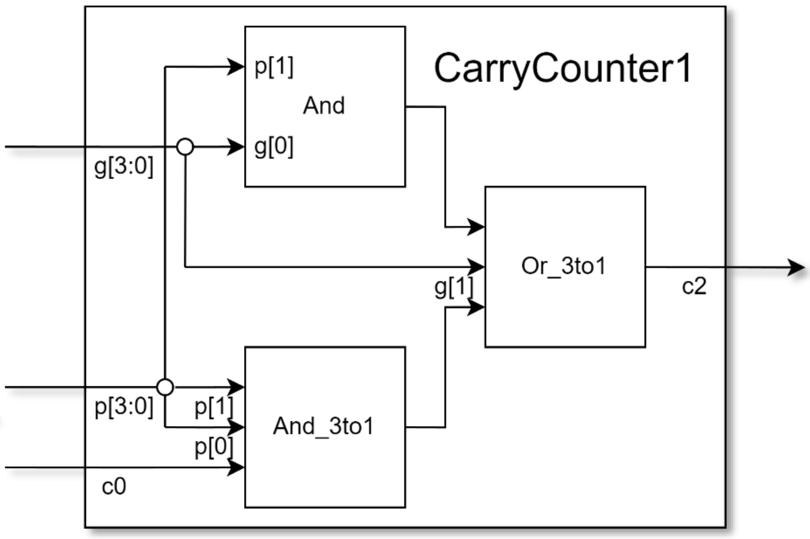

And_5to1
a
b
c
d
e
And_4to1
And
out

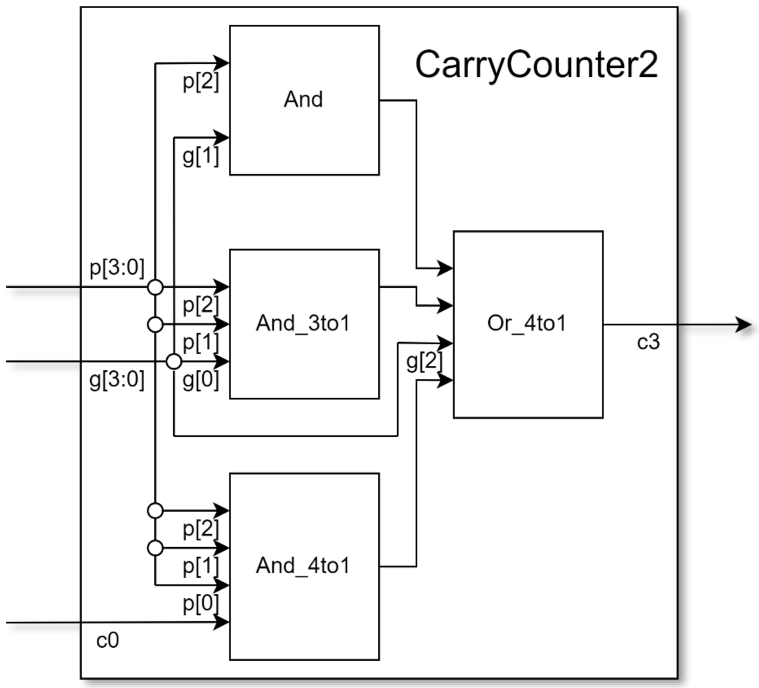

Or_3to1
a
b
c
Or
Or
out

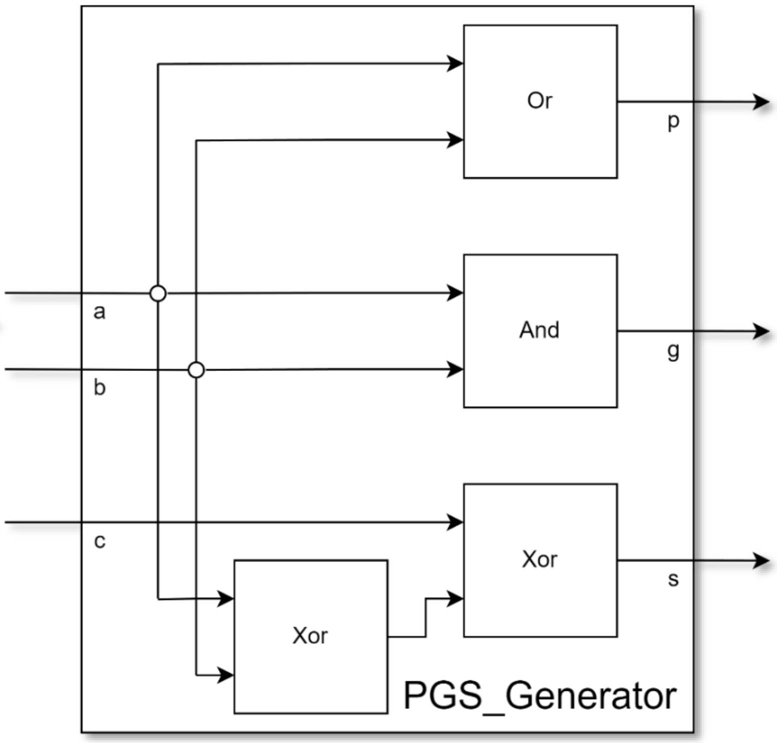

Or_4to1
a
b
c
d
Or
Or
Or
out

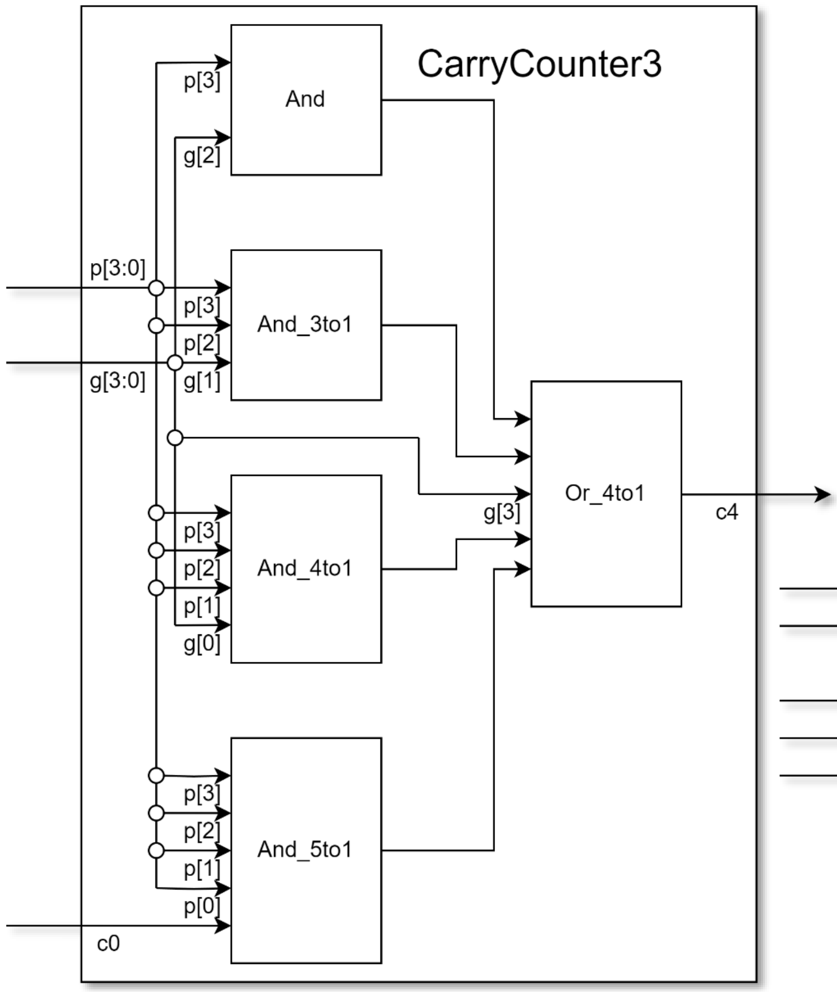

Or_5to1
a
b
c
d
e
Or_4to1
Or
out

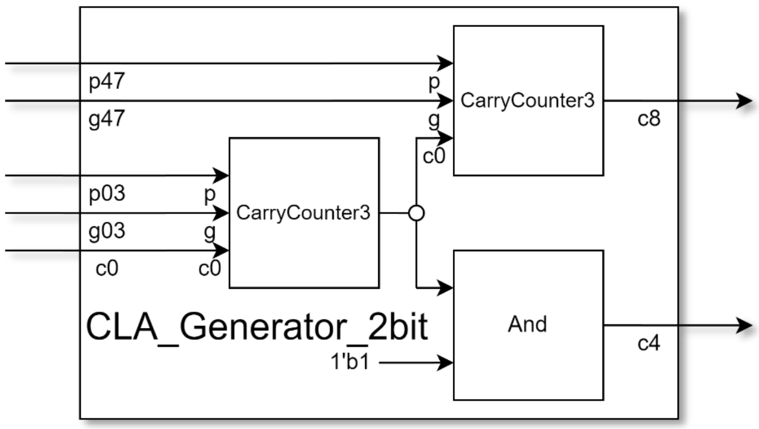

CarryCounter0
p[3:0]
p[0]
Or
c0
Or
g[0]
g[3:0]
c1

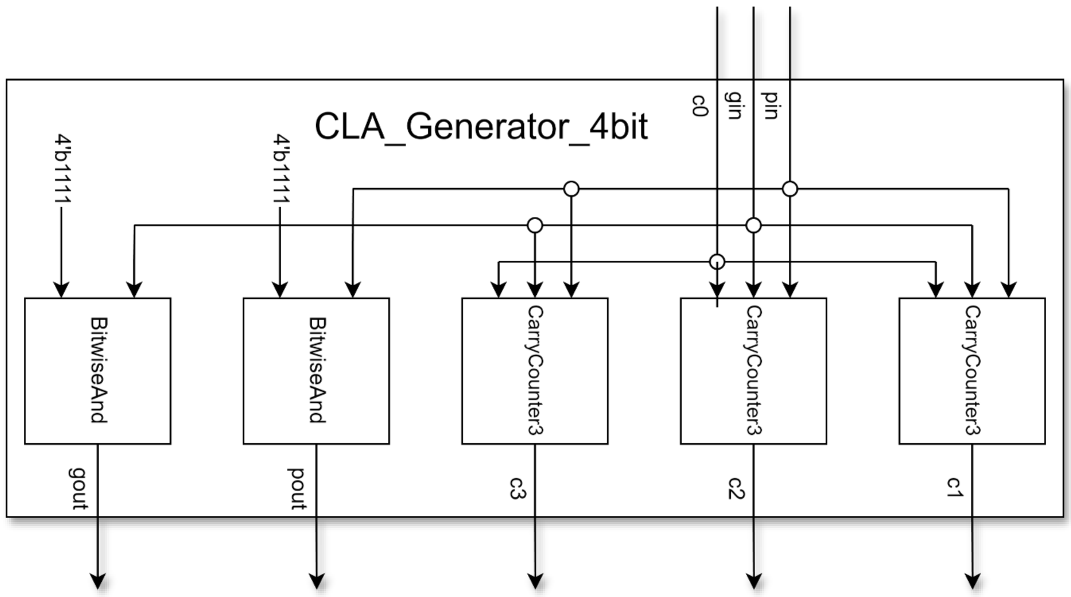

CarryCounter1
p[1]
And
g[0]
g[3:0]
Or_3to1
g[1]
c2
p[3:0]
p[1]
p[0]
And_3to1
c0

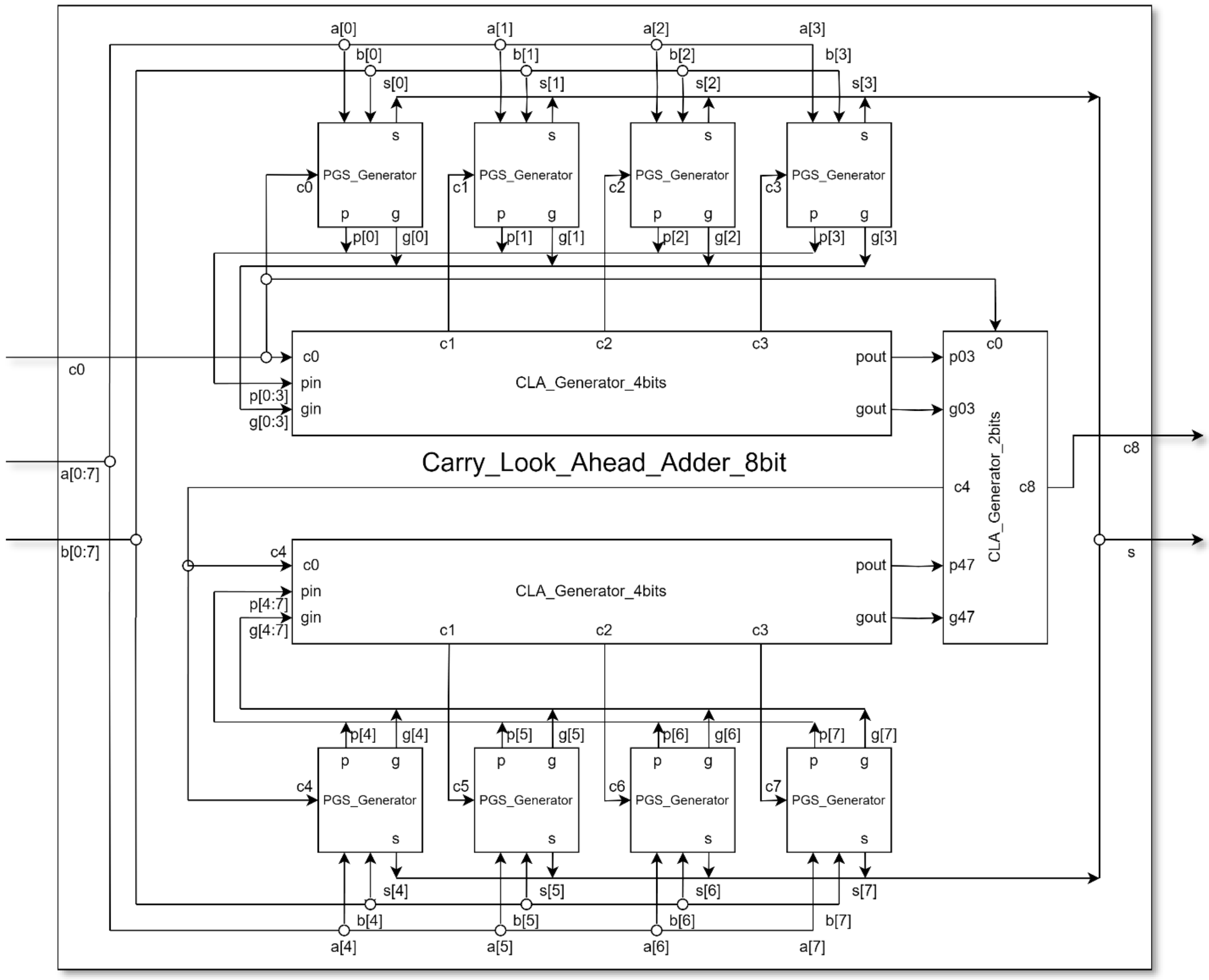

CarryCounter2
p[2]
g[1]
And
p[3:0]
p[2]
p[1]
g[3:0]
g[0]
And_3to1
g[2]
Or_4to1
c3
p[2]
p[1]
p[0]
And_4to1
c0


Or
p
a
And
g
b
c
Xor
Xor
s
PGS_Generator

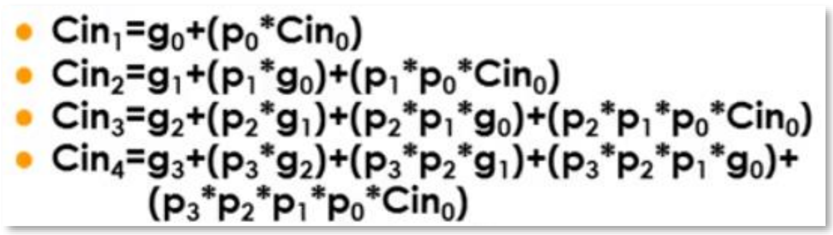

CarryCounter3
p[3]
g[2]
And
p[3:0]
p[3]
p[2]
g[3:0]
g[1]
And_3to1
g[3]
Or_4to1
c4
p[3]
p[2]
p[1]
g[0]
And_4to1
p[3]
p[2]
p[1]
p[0]
And_5to1
c0


p47
g47
p
g
CarryCounter3
c0
c8
p03
g03
c0
p
g
c0
CarryCounter3
And
c4
CLA_Generator_2bit
1'b1


CLA_Generator_4bit
c0
gin
pin
4'b1111
4'b1111
BitwiseAnd
BitwiseAnd
CarryCounter3
CarryCounter3
CarryCounter3
gout
pout
c3
c2
c1


Carry_Look_Ahead_Adder_8bit
a[0] b[0] s[0]
a[1] b[1] s[1]
a[2] b[2] s[2]
a[3] b[3] s[3]
PGS_Generator
p[0] g[0] p[1] g[1] p[2] g[2] p[3] g[3]
c0 c1 c2 c3
CLA_Generator_4bits
c0 pin gin p[0:3] g[0:3]
pout gout
p03 g03
CLA_Generator_2bits
c4 c8
p47 g47
c8
s
a[0:7]
b[0:7]
c4 p[4:7] g[4:7]
p[4] g[4] p[5] g[5] p[6] g[6] p[7] g[7]
c4 c5 c6 c7
s[4] s[5] s[6] s[7]
b[4] b[5] b[6] b[7]
a[4] a[5] a[6] a[7]

B. Explanation



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

▲ 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

C. Testbench

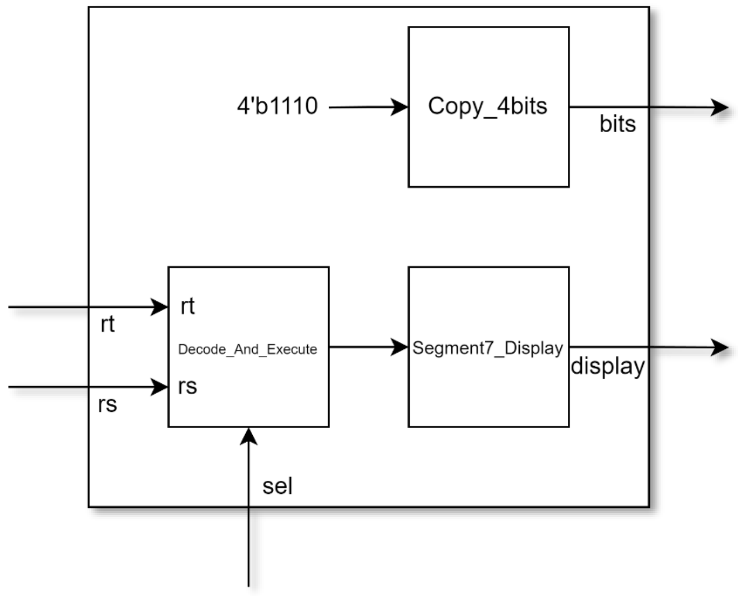
As **DAE Problem**, I set the simulation time to 1,0000,0000 ns and use three layers of loop (for **a**, **b**, **c0**) to go through every input pattern. And I also define a register called **err**. The testbench will check every result by my behavioral-level code. If the result from the gate-level 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.

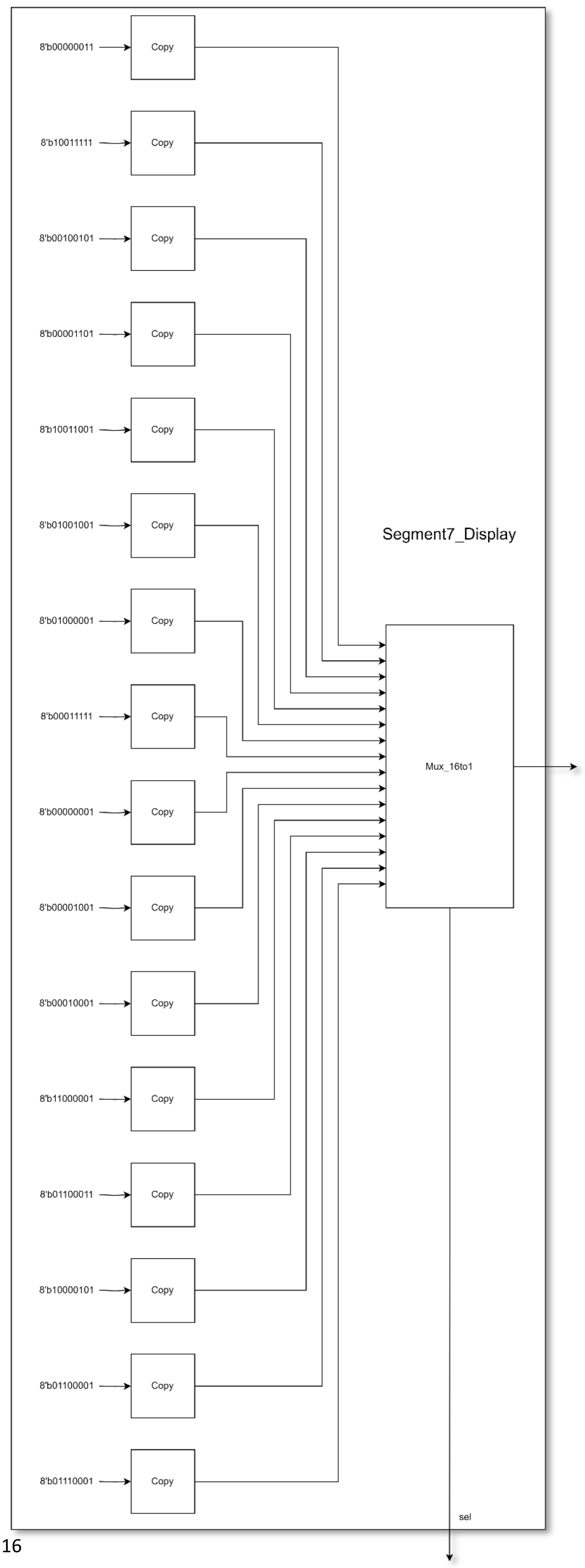
4. An exhaustive testbench design

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