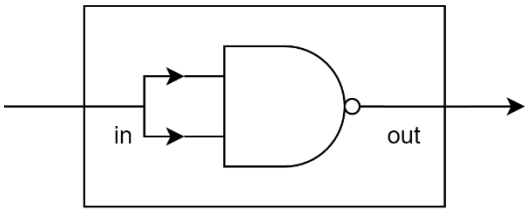
Hardware Design and Lab: Lab1

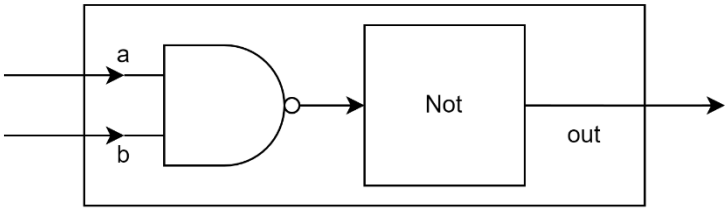
111060013 EECS 26' 劉祐廷

1. Basic

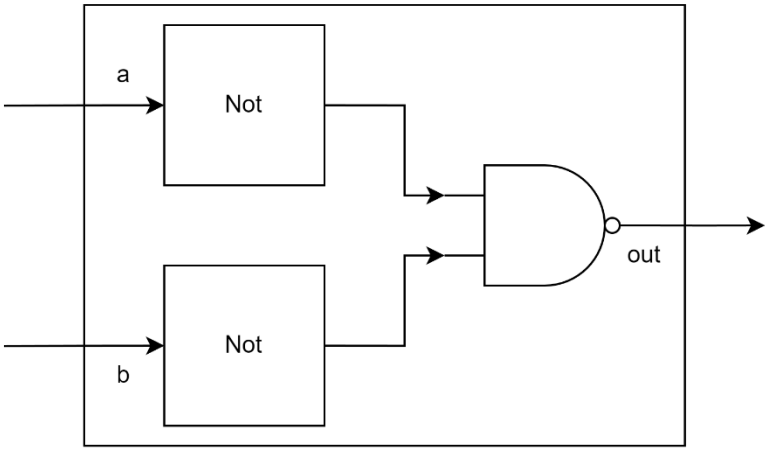
A. Basic Question 1: Block Diagram



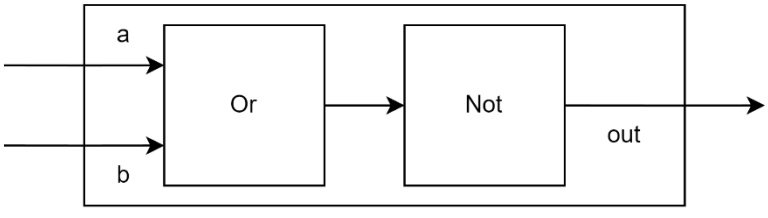
Not



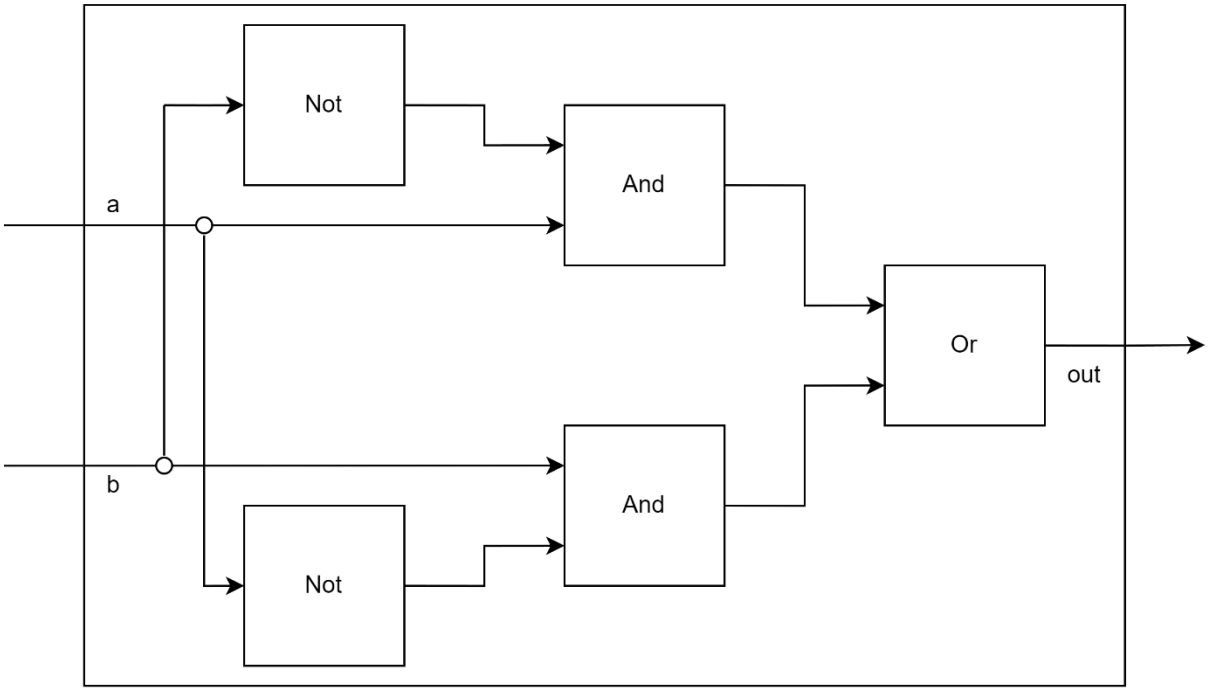
And



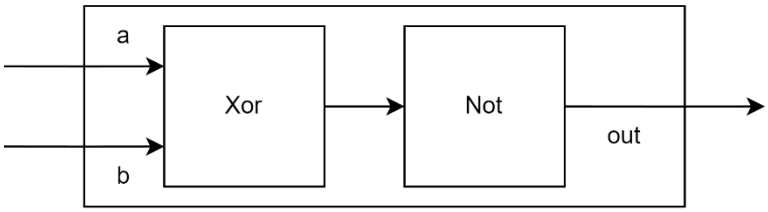
Or



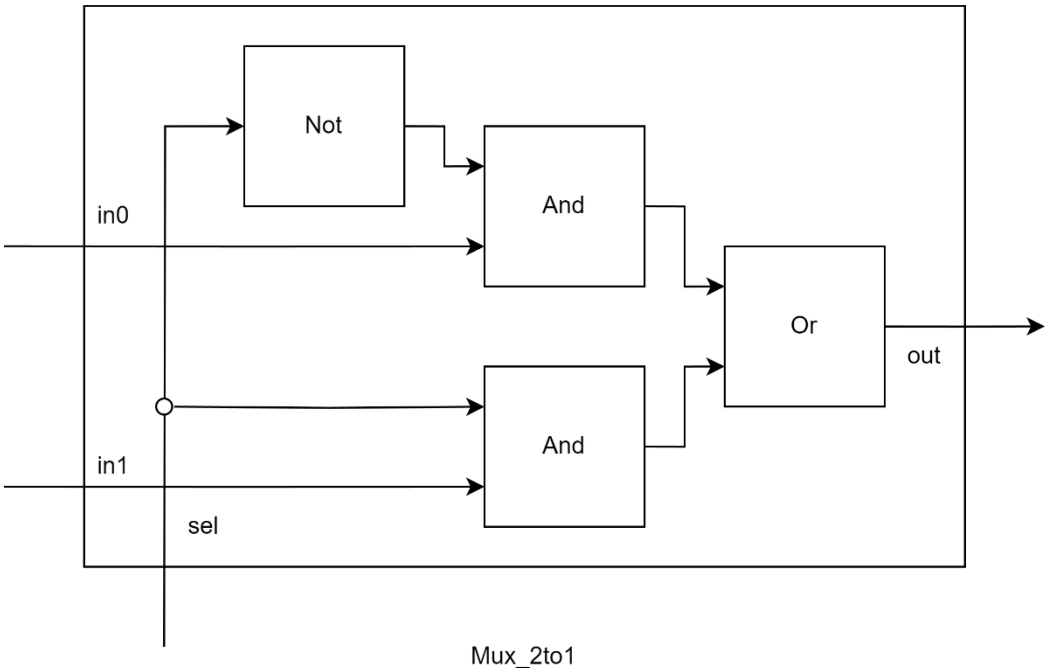
Nor



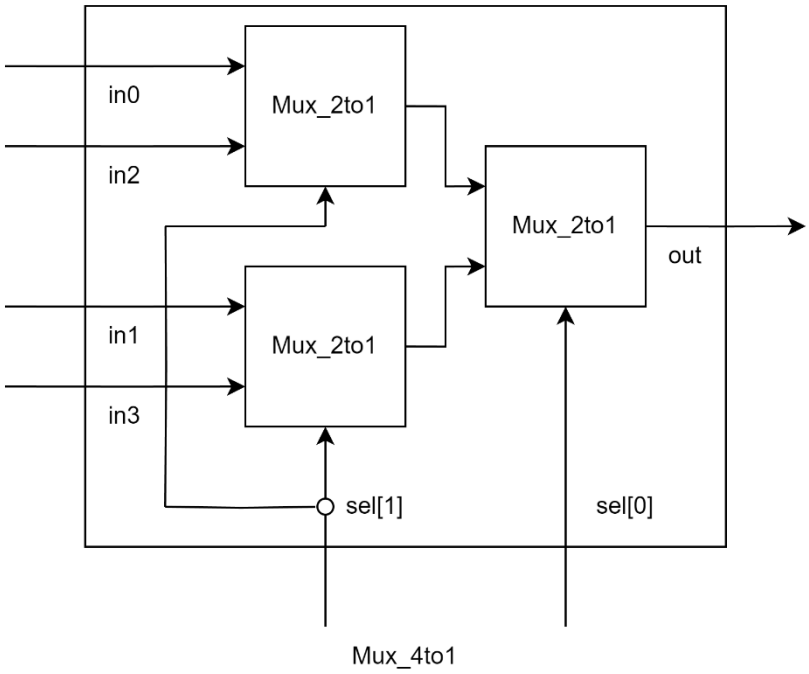
Xor



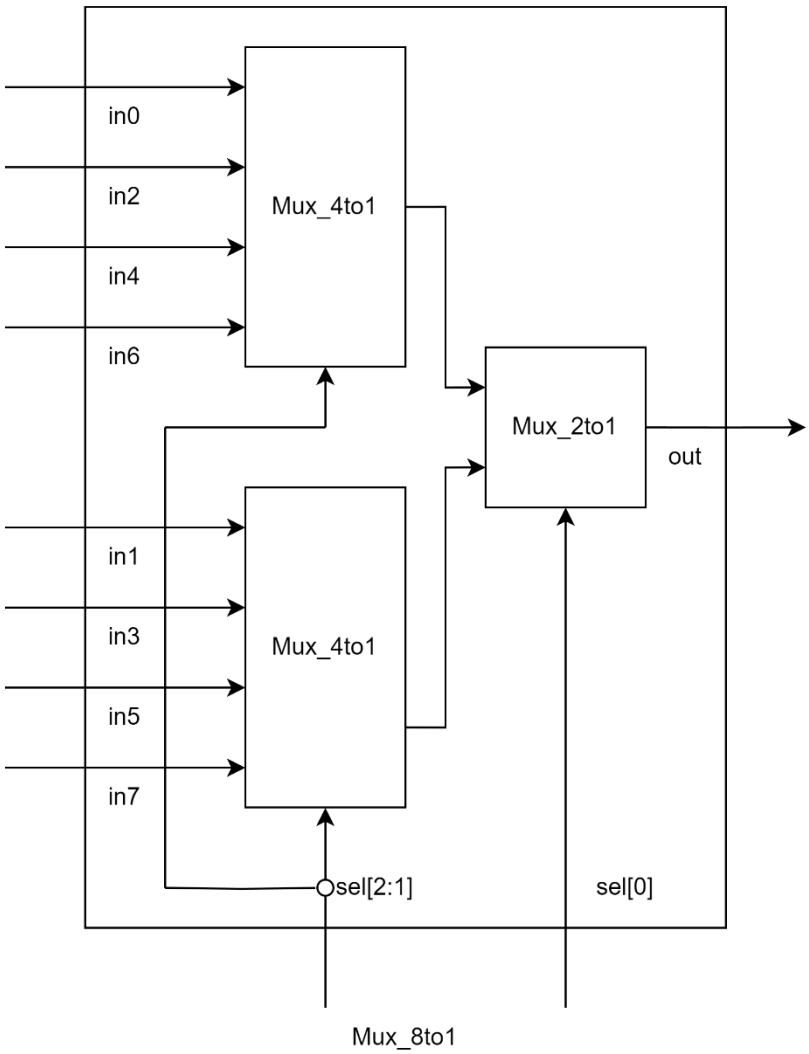
Xnor



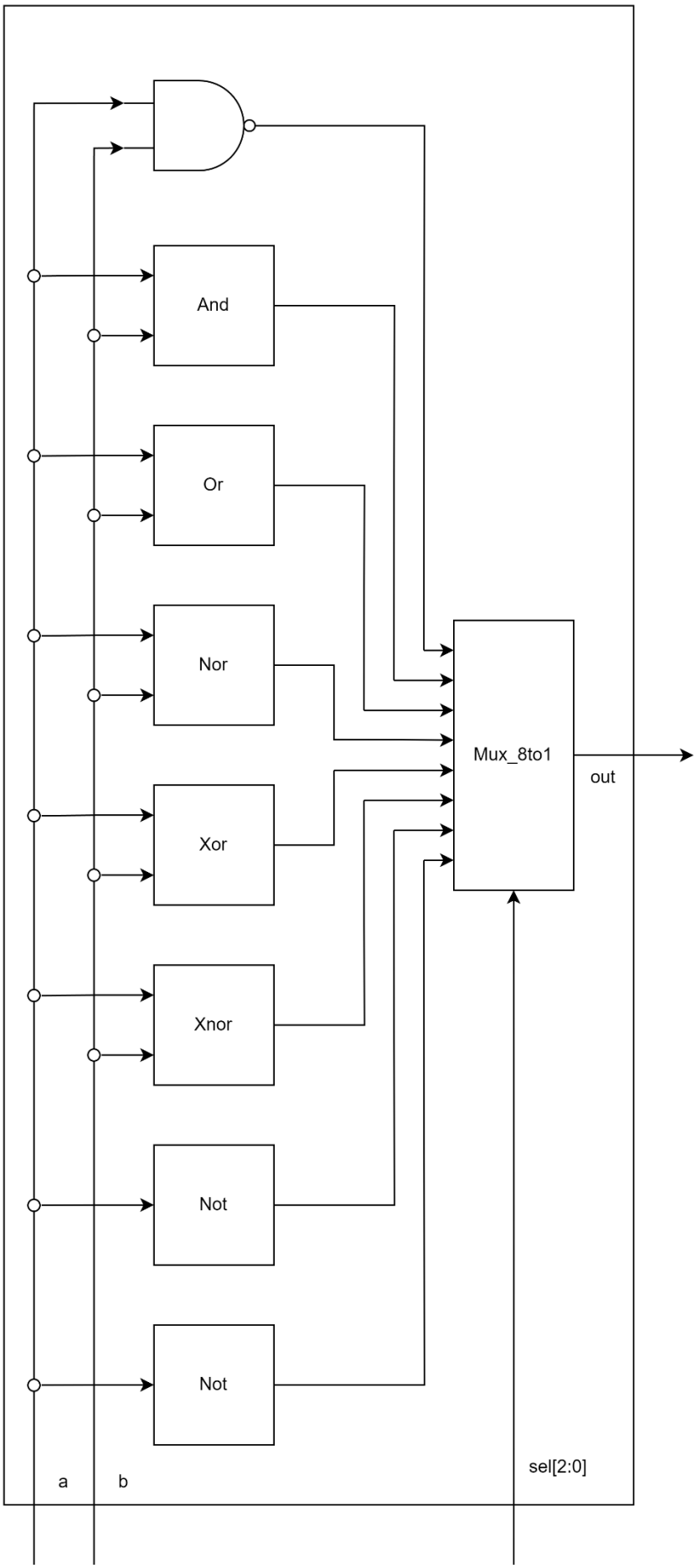
Mux_2to1



Mux_4to1



Mux_8to1

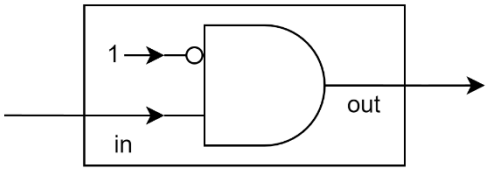

And
Or
Nor
Xor
Xnor
Not
Not
Mux_8to1
out
a
b
sel[2:0]

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

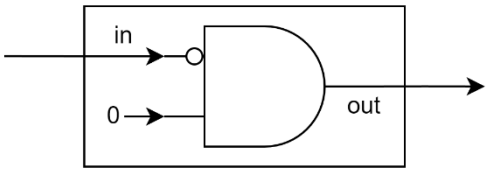
The most significant difference between them is that a half adder can only deal with the situation without carry in; however, a full adder can handle the situation with carry in.

2. Advanced: Decode and execute

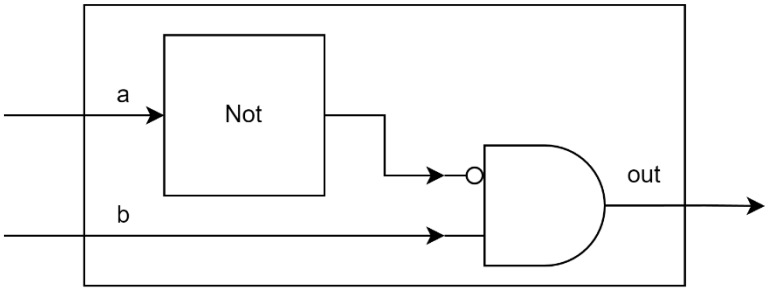
A. Block Diagram: Basic Modules



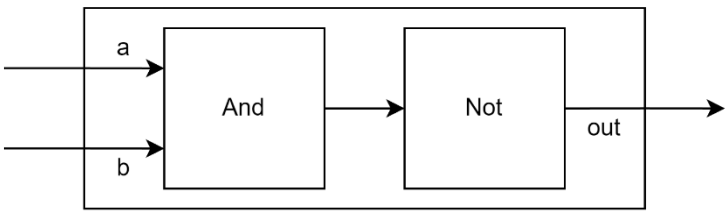
Not



Copy



And

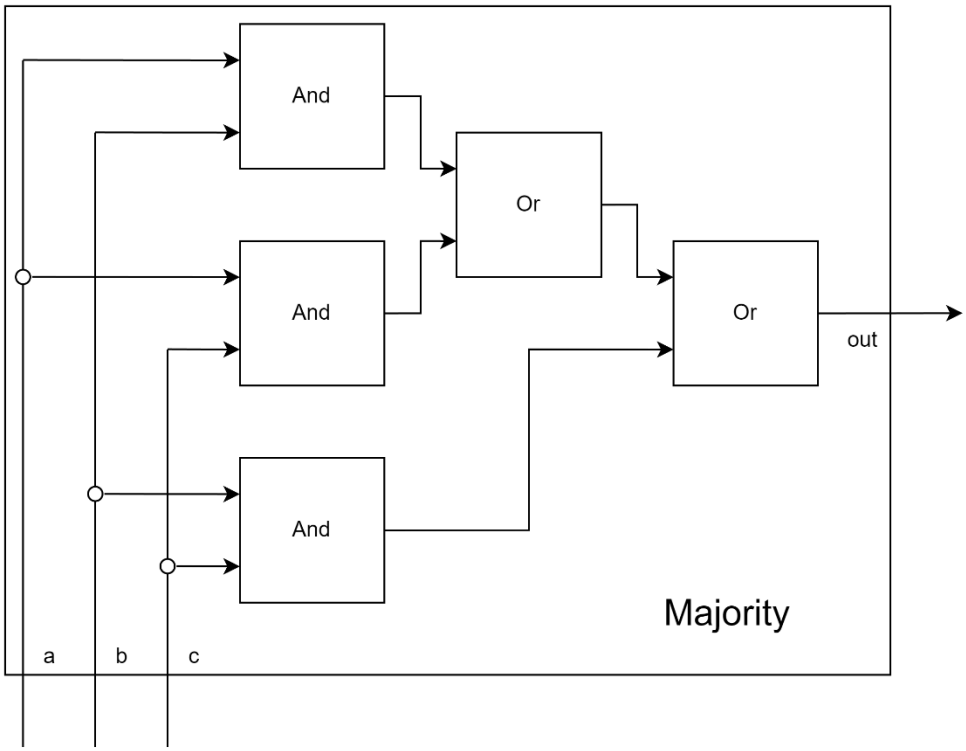


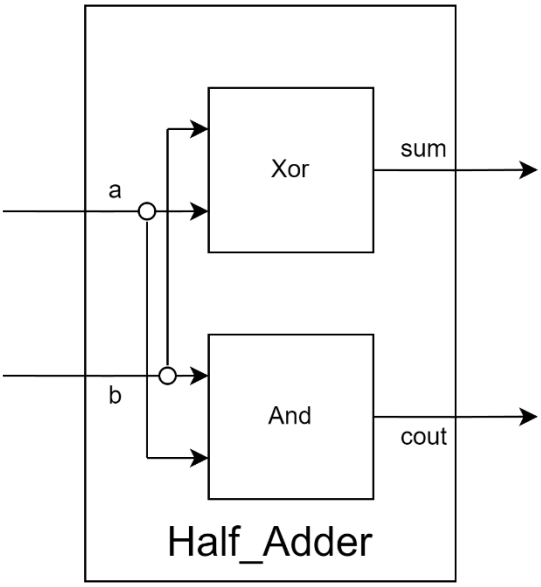
Nand

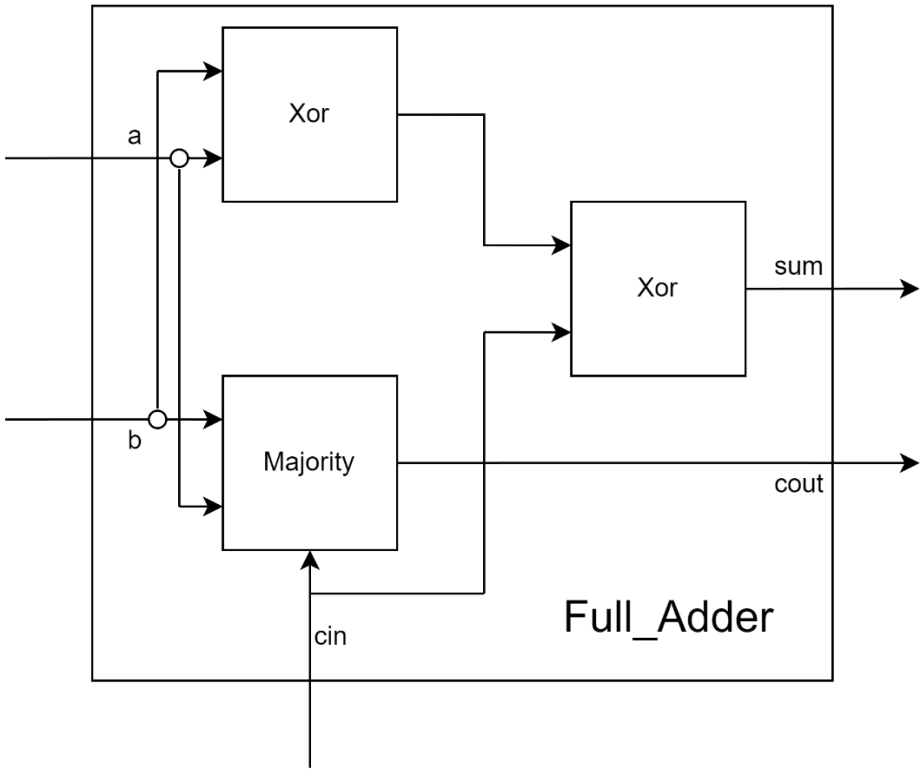
B. Explanation: Basic Modules

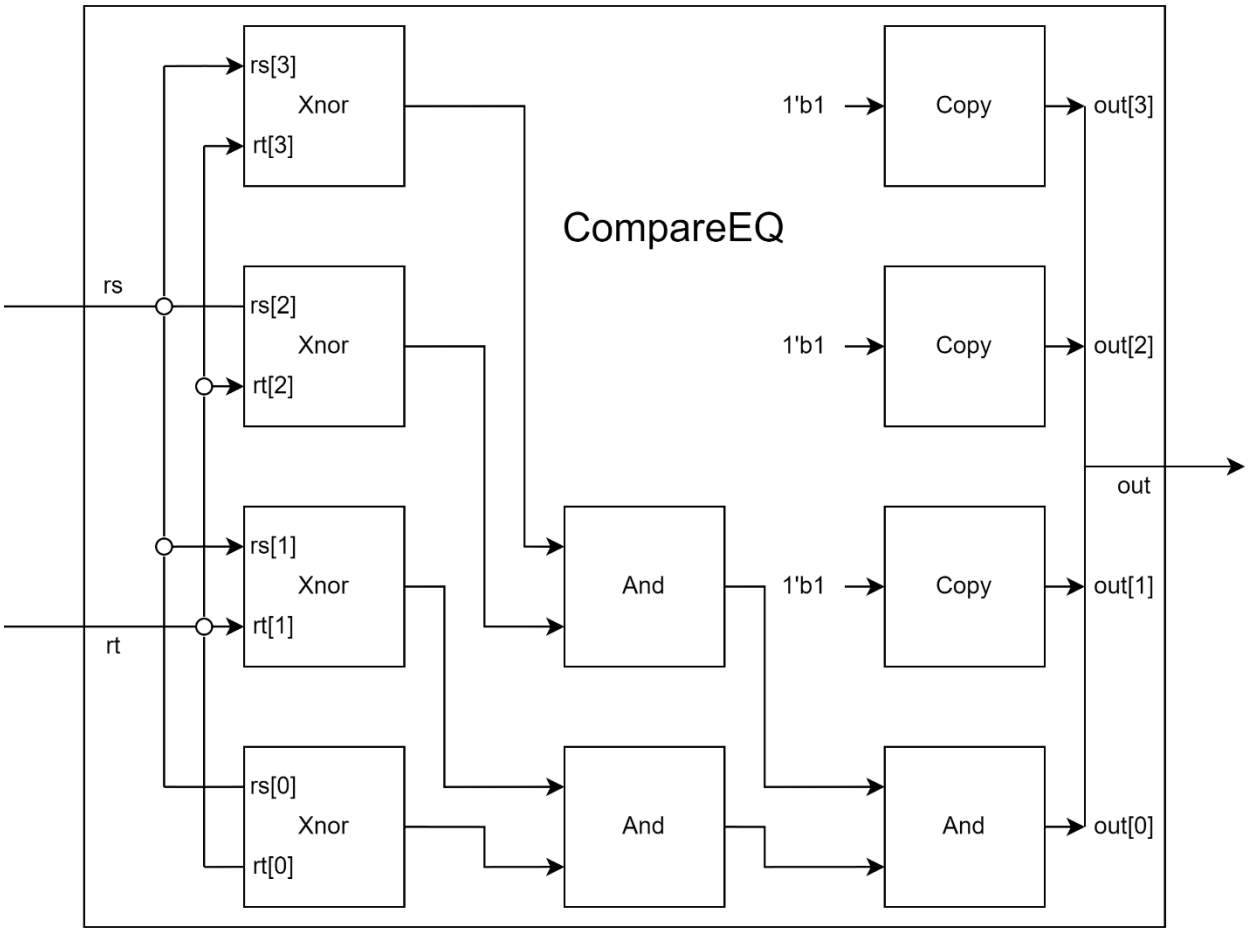
First of all, I drew the truth table of **Universal Gate** to design my own **Not** module and **Copy** module. And then I combined a **Not** module and an **Universal Gate** to create **And** module. After that, I make a **Nand** module with an **And** module and a **Not** module. The reason why I design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

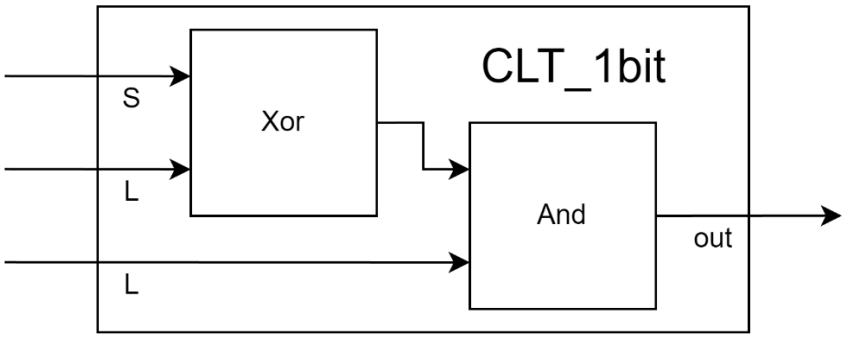
C. Block Diagram: Advanced Modules

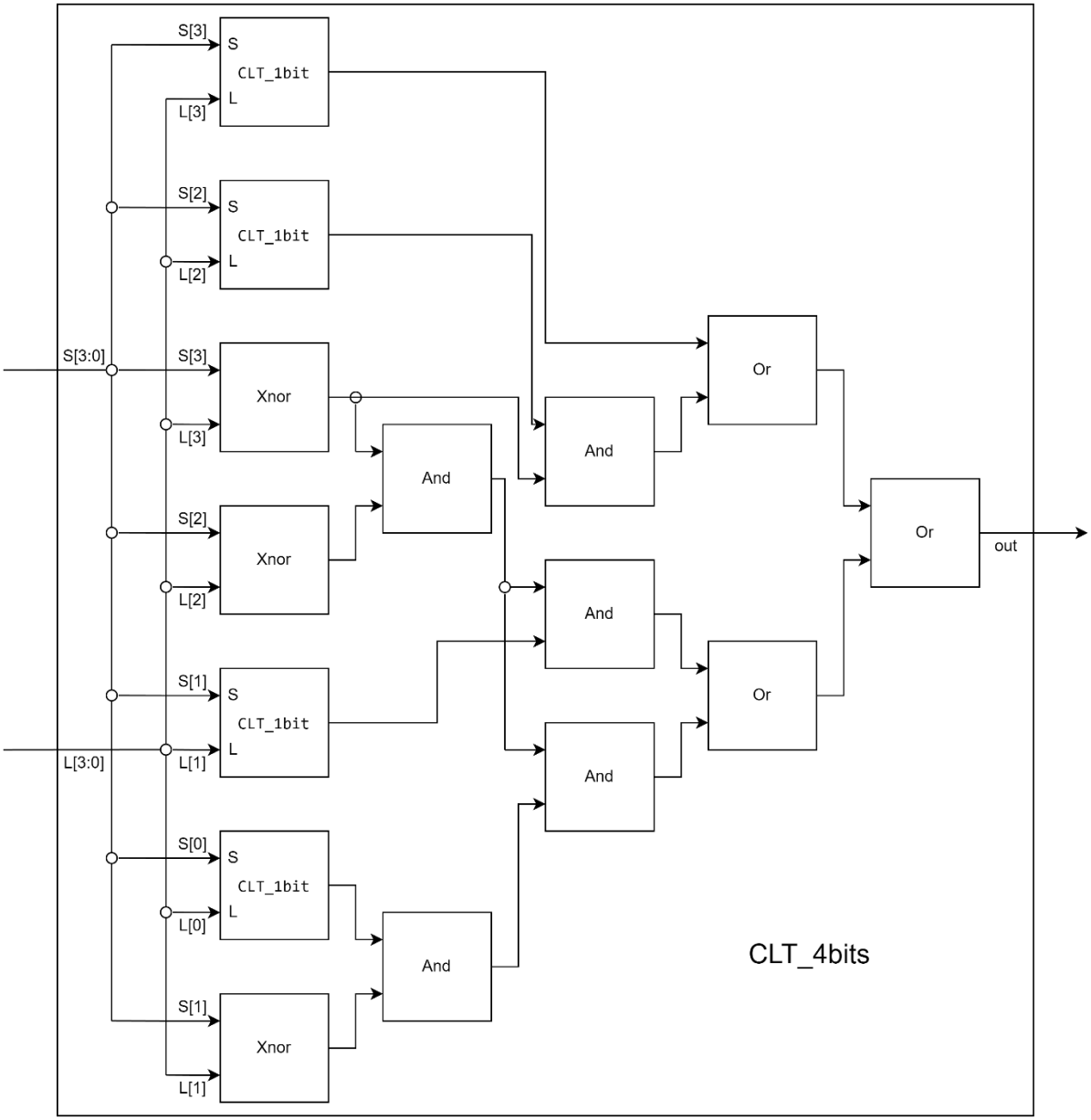


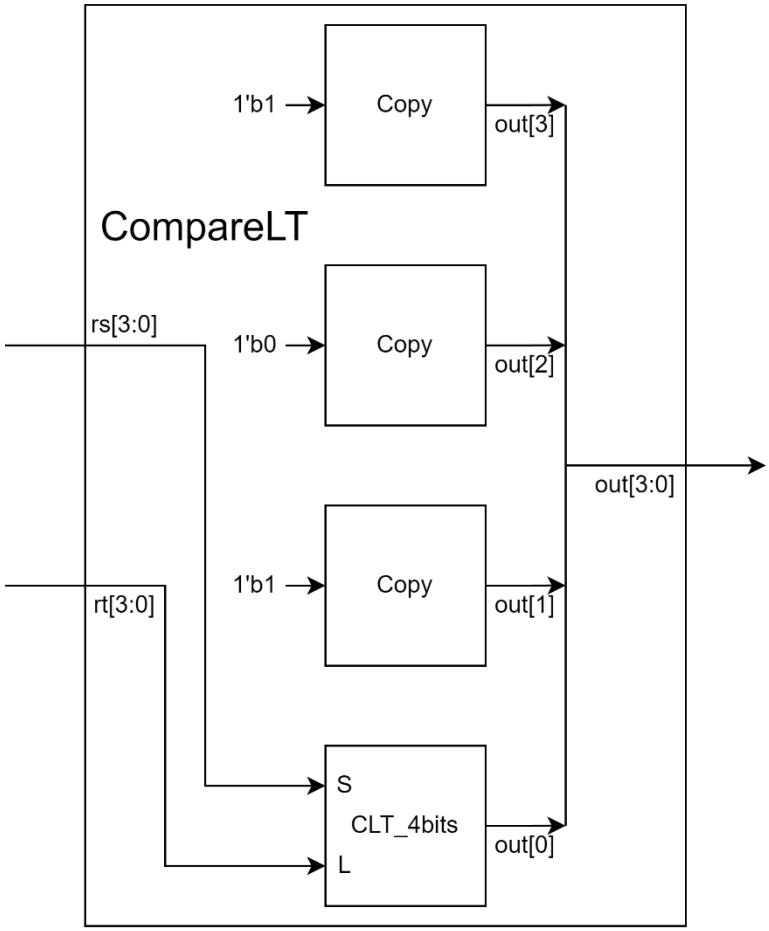

Xor
sum
a
b
And
cout
Half_Adder

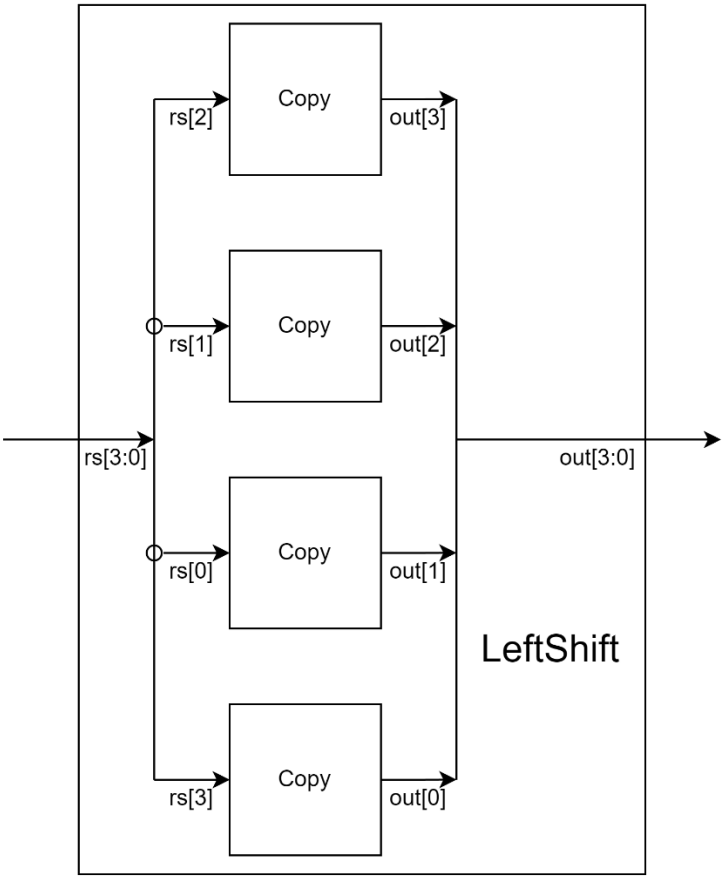

Xor
a
Xor
sum
b
Majority
cout
cin
Full_Adder

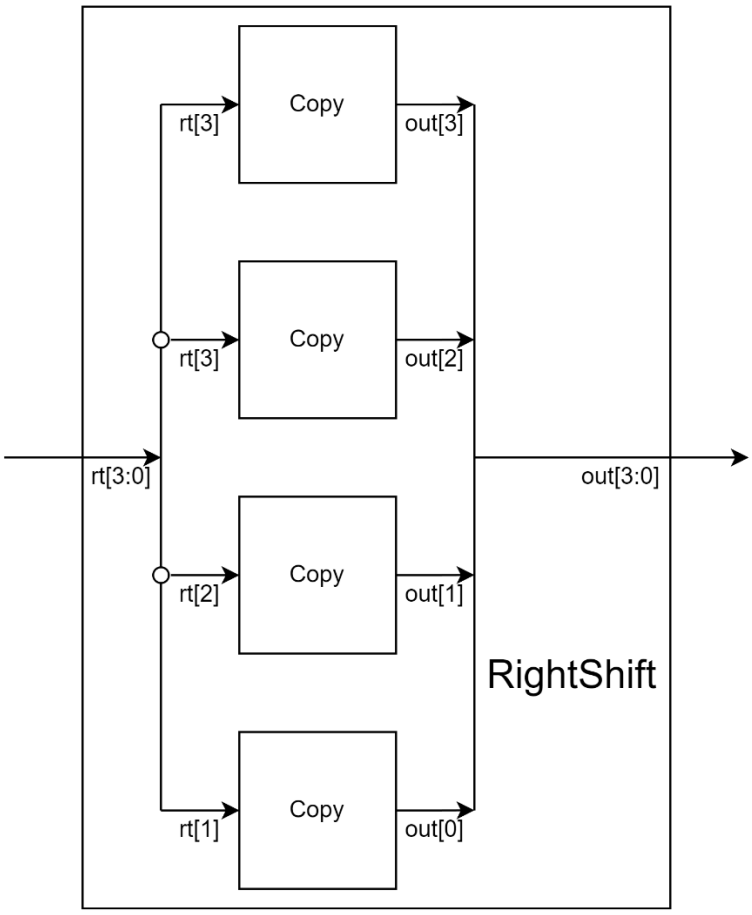

CompareEQ
rs
rt
rs[3]
Xnor
rt[3]
rs[2]
Xnor
rt[2]
rs[1]
Xnor
rt[1]
rs[0]
Xnor
rt[0]
1'b1
Copy
out[3]
1'b1
Copy
out[2]
out
And
1'b1
Copy
out[1]
And
And
out[0]

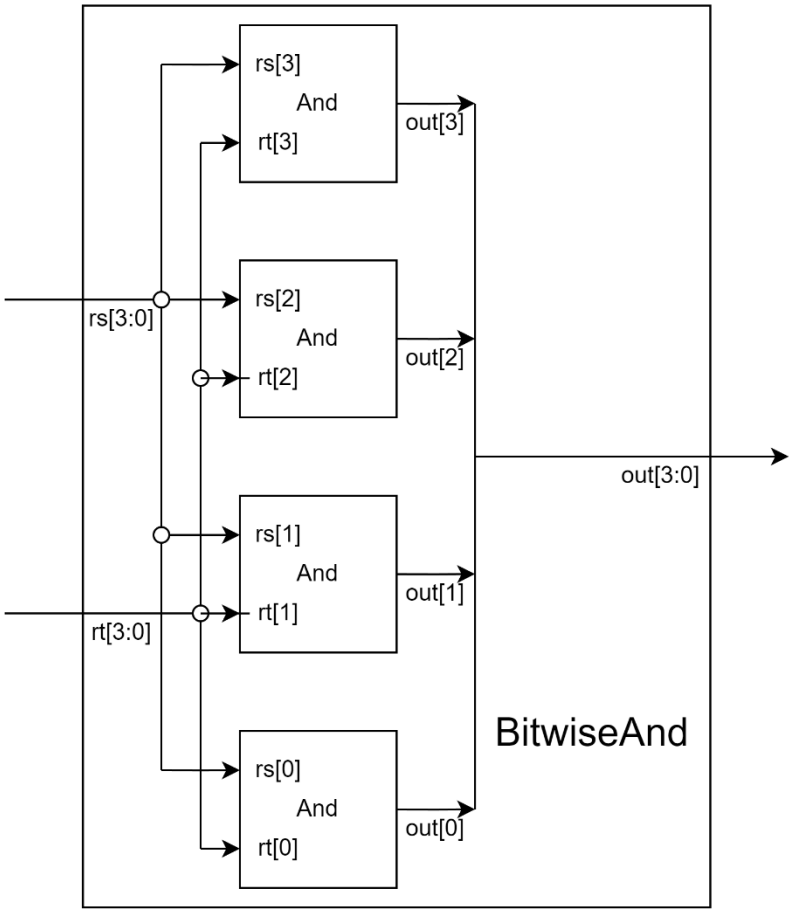

CLT_1bit
S
L
L
Xor
And
out

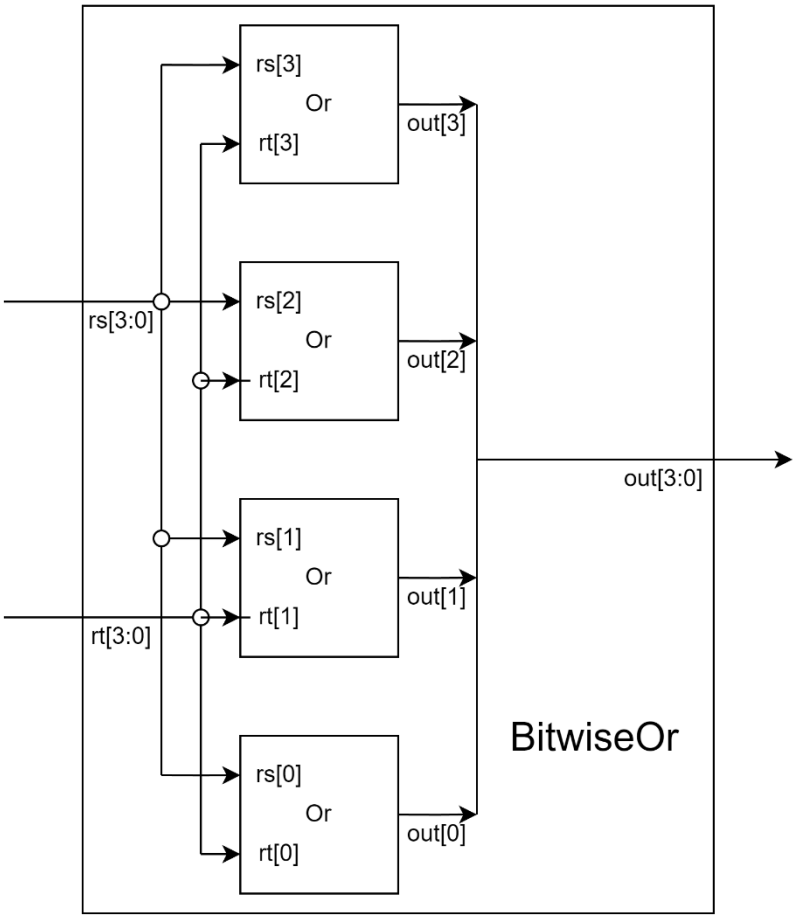

S[3]
S
CLT_1bit
L
L[3]
S[2]
S
CLT_1bit
L
L[2]
S[3:0]
S[3]
Xnor
L[3]
And
Or
And
S[2]
Xnor
L[2]
Or
out
And
S[1]
S
CLT_1bit
L
L[3:0]
L[1]
Or
And
S[0]
S
CLT_1bit
L
L[0]
And
S[1]
Xnor
L[1]
CLT_4bits

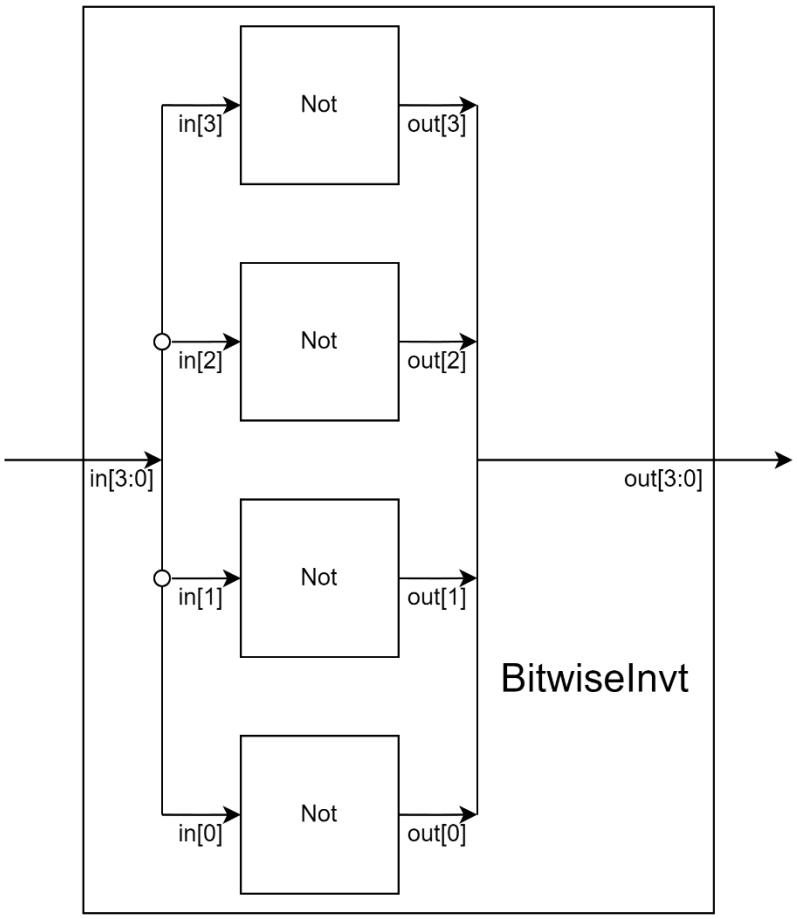

1'b1
Copy
out[3]
CompareLT
rs[3:0]
1'b0
Copy
out[2]
out[3:0]
1'b1
Copy
out[1]
rt[3:0]
S
CLT_4bits
L
out[0]

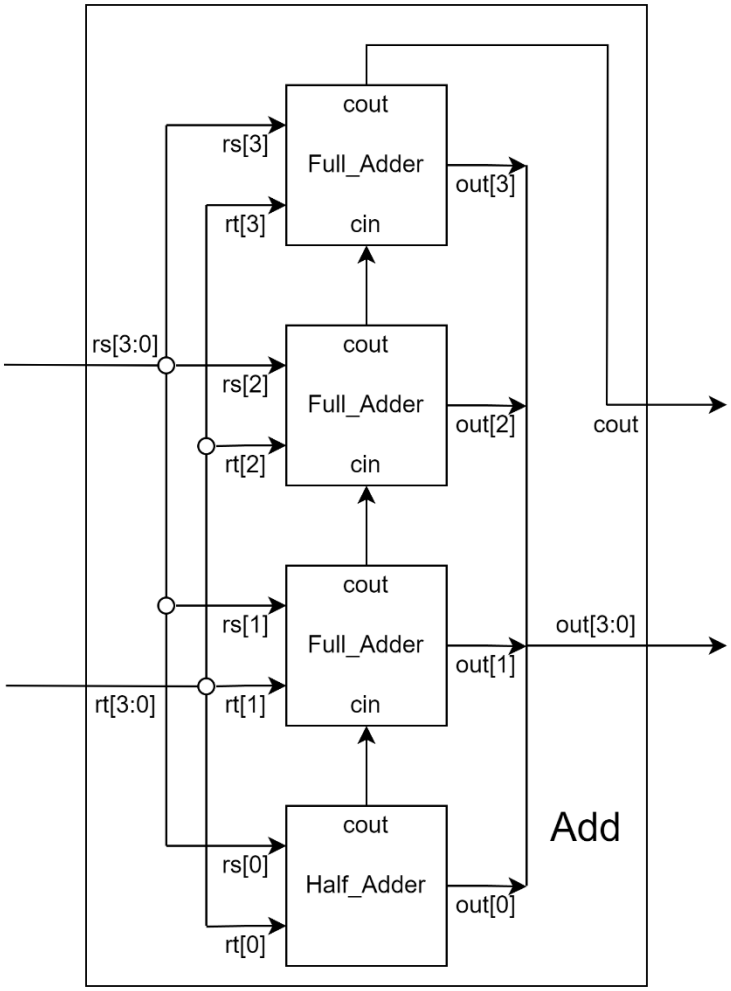

Copy
rs[2]
out[3]
Copy
rs[1]
out[2]
rs[3:0]
out[3:0]
Copy
rs[0]
out[1]
LeftShift
Copy
rs[3]
out[0]

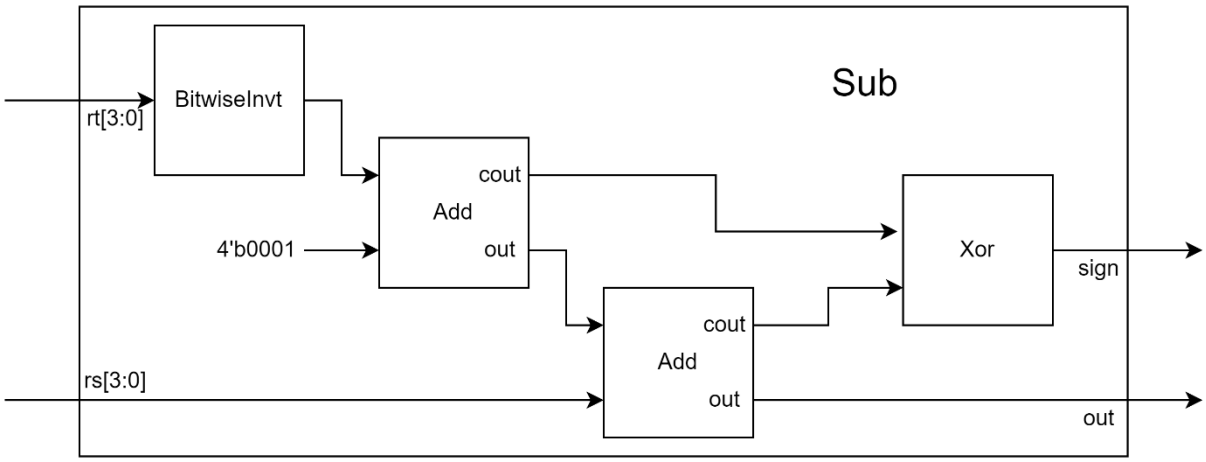

Copy
rt[3]
out[3]
Copy
rt[3]
out[2]
rt[3:0]
out[3:0]
Copy
rt[2]
out[1]
RightShift
Copy
rt[1]
out[0]

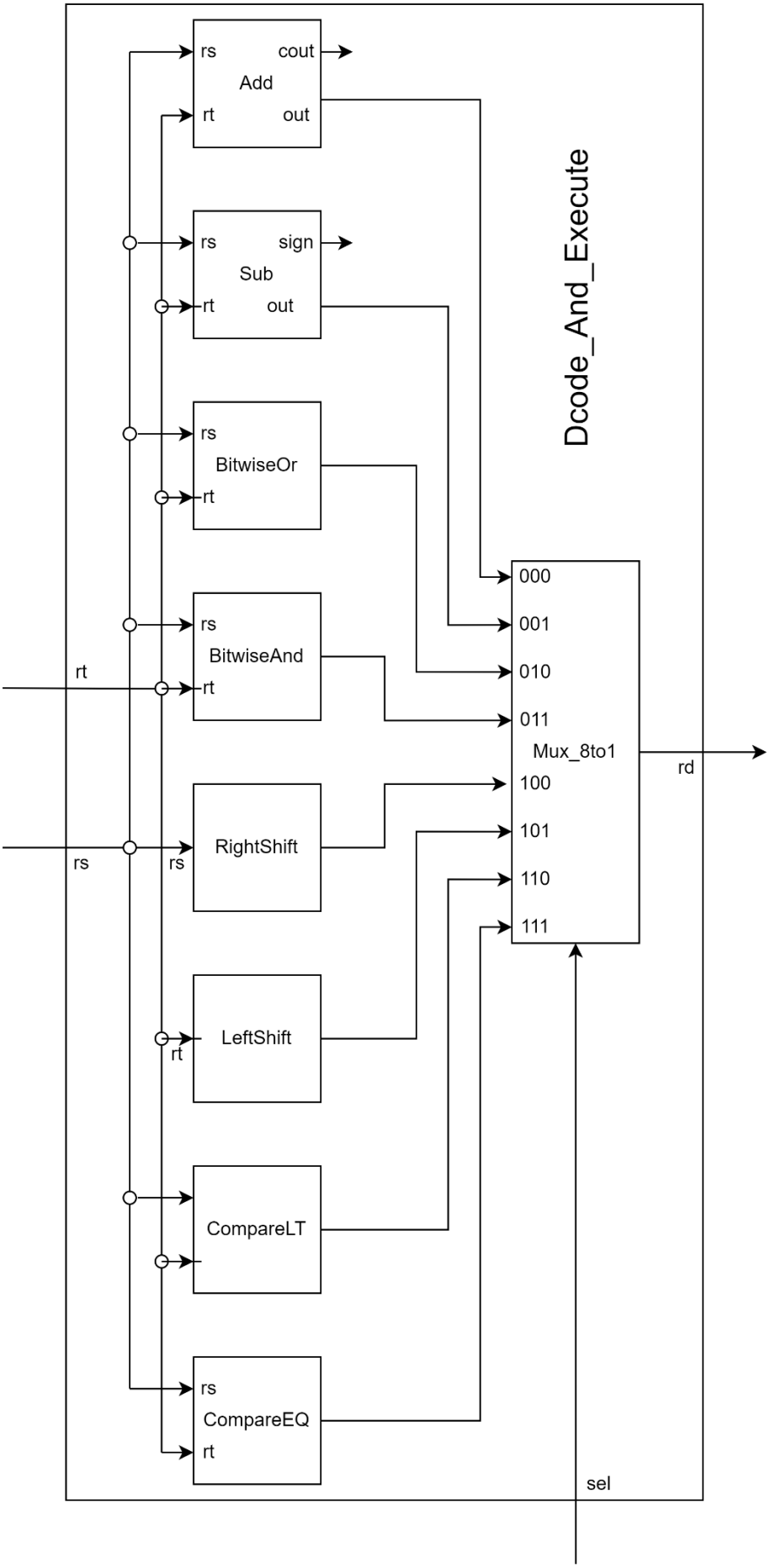

rs[3]
And
rt[3]
out[3]
rs[2]
And
rt[2]
out[2]
rs[3:0]
out[3:0]
rs[1]
And
rt[1]
out[1]
rt[3:0]
BitwiseAnd
rs[0]
And
rt[0]
out[0]


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


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


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


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

D. Explanation: Some Useful Modules

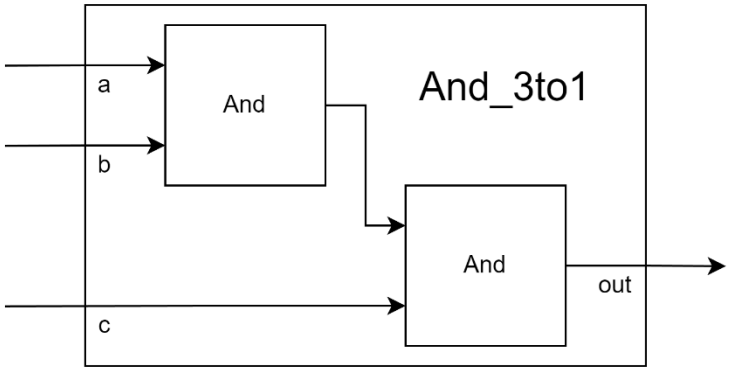
a. **Majority:** Count the carry out
b. **Half_adder:** Deal with 1-bit addition without carry in
c. **Full_Adder:** Deal with 1-bit addition with carry in
d. **CompareEQ:** Compare two 4-bit data whether they are equal or not and output **rd**
e. **CLT_1bit:** Compare two 1-bit data **rs** and rt and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

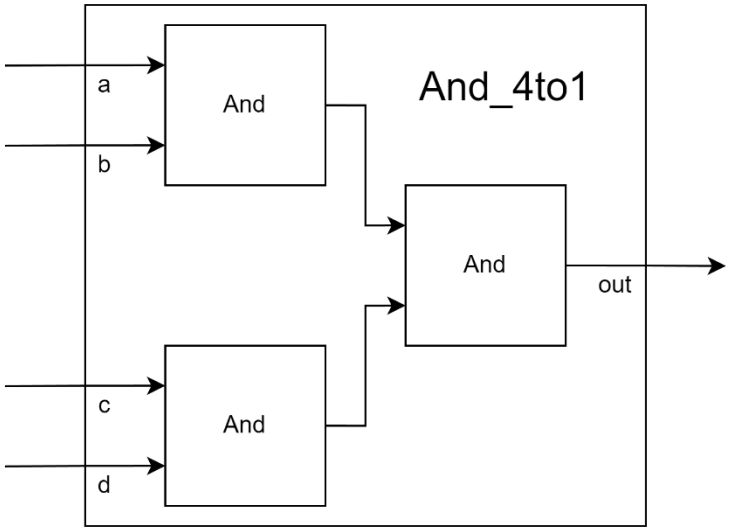
E. Testbench

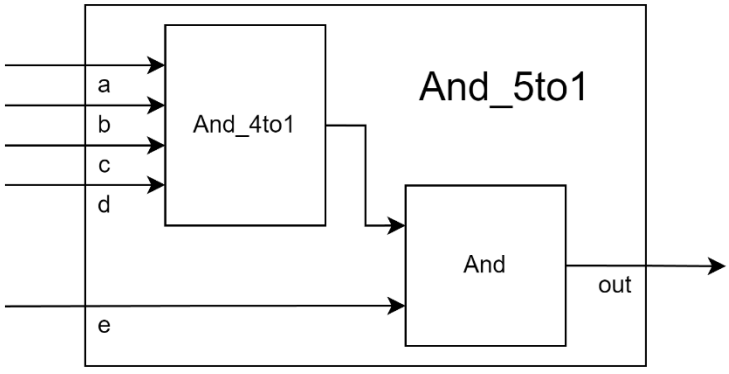
I set the simulation time to 1,0000,0000 ns and use three layers of loop (for **rs**, **rt**, **sel**) to go through every input pattern. And I also define a register called **err**. The testbench will check every result by my behavioral-level code. If the result from the gate-level circuit is different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

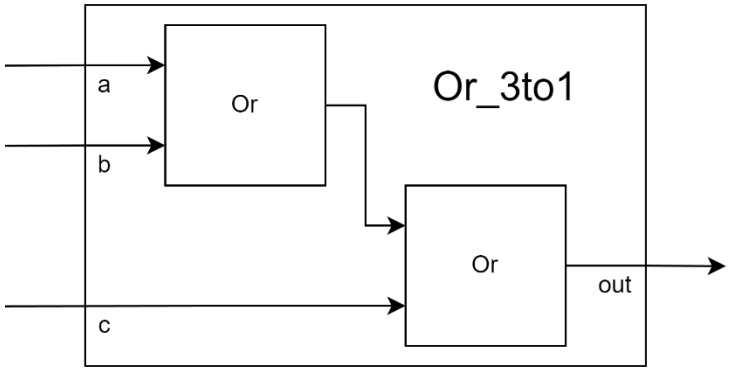
3. Advanced: 8-bit carry-lookahead (CLA) Adder

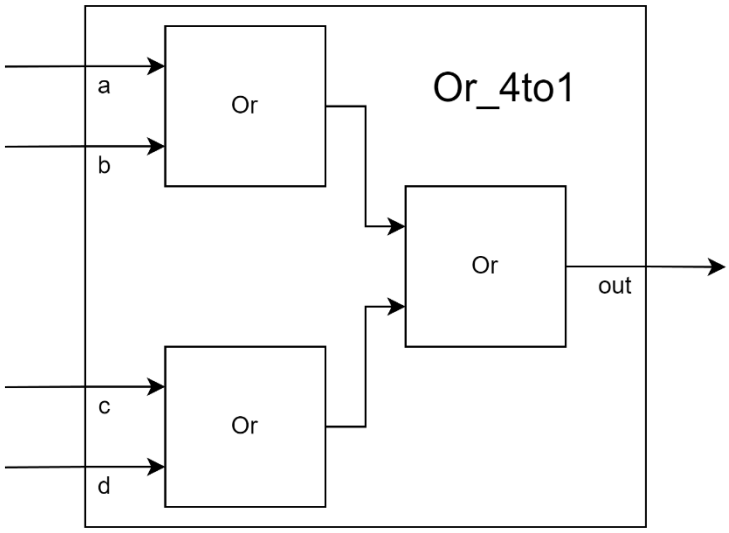
A. Block Diagram

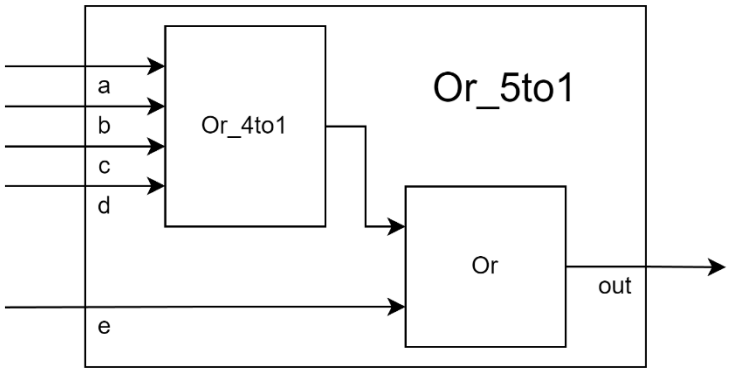

And_3to1
a
b
c
And
And
out

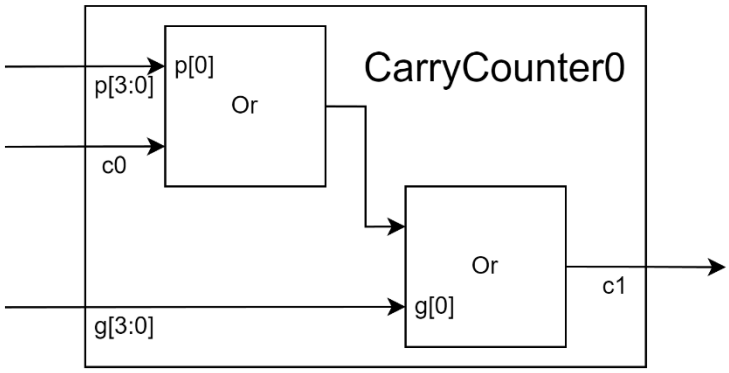

And_4to1
a
b
c
d
And
And
And
out

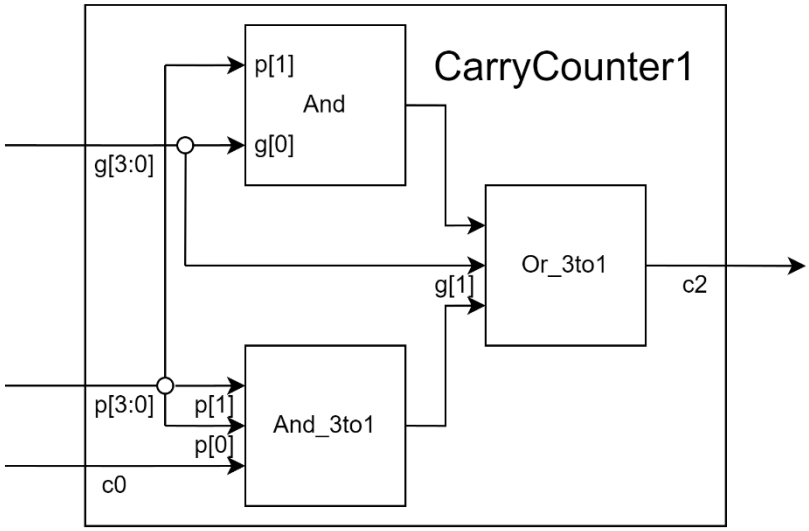

And_5to1
a
b
c
d
e
And_4to1
And
out

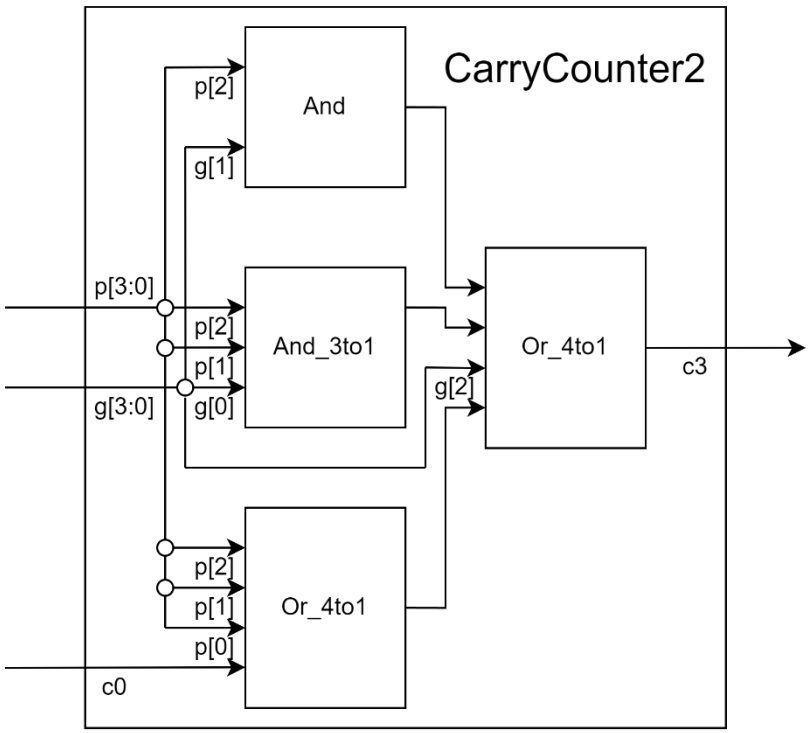

Or_3to1
a
b
c
Or
Or
out

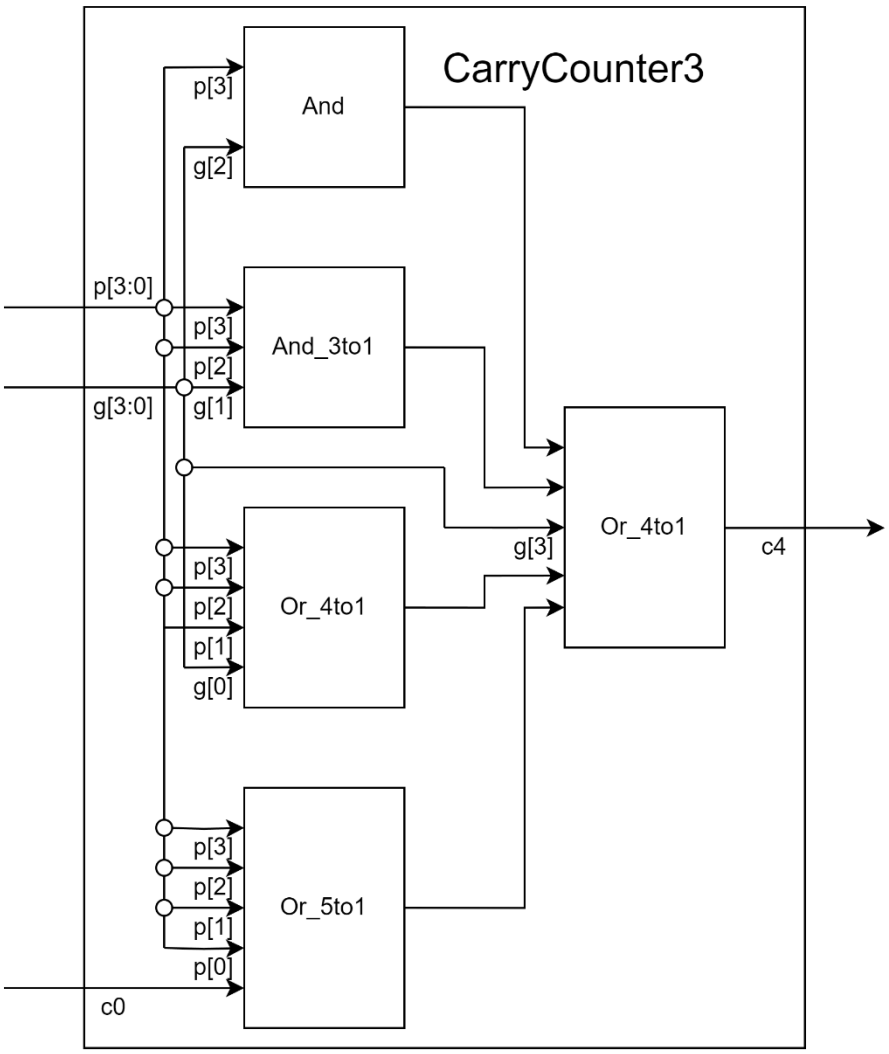

Or_4to1
a
b
c
d
Or
Or
out

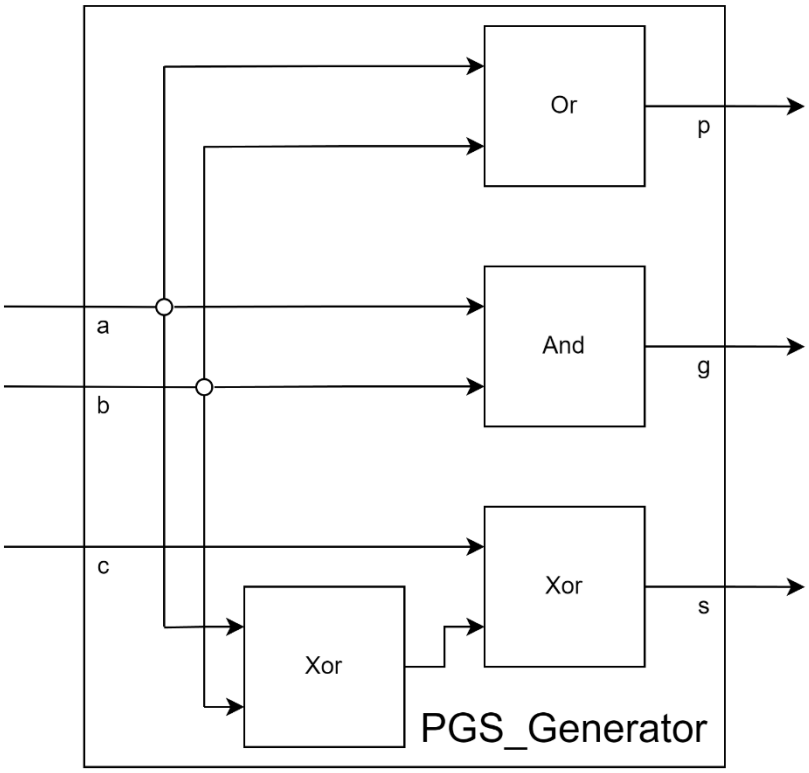

Or_5to1
a
b
c
d
e
Or_4to1
Or
out

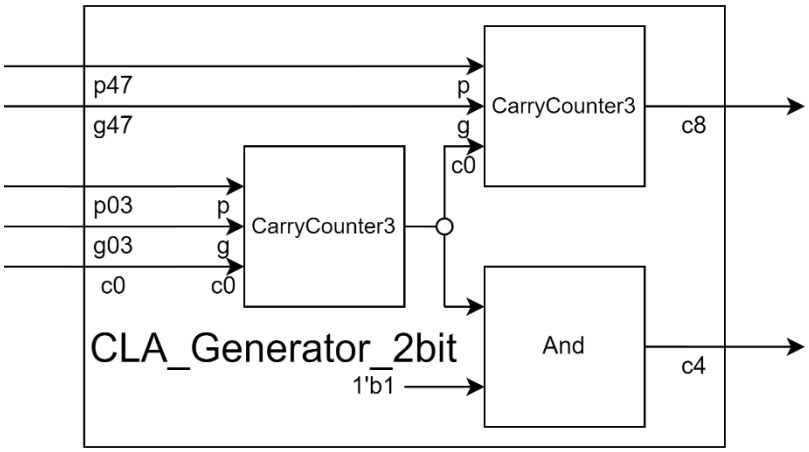

CarryCounter0
p[3:0]
c0
g[3:0]
p[0]
Or
Or
g[0]
c1

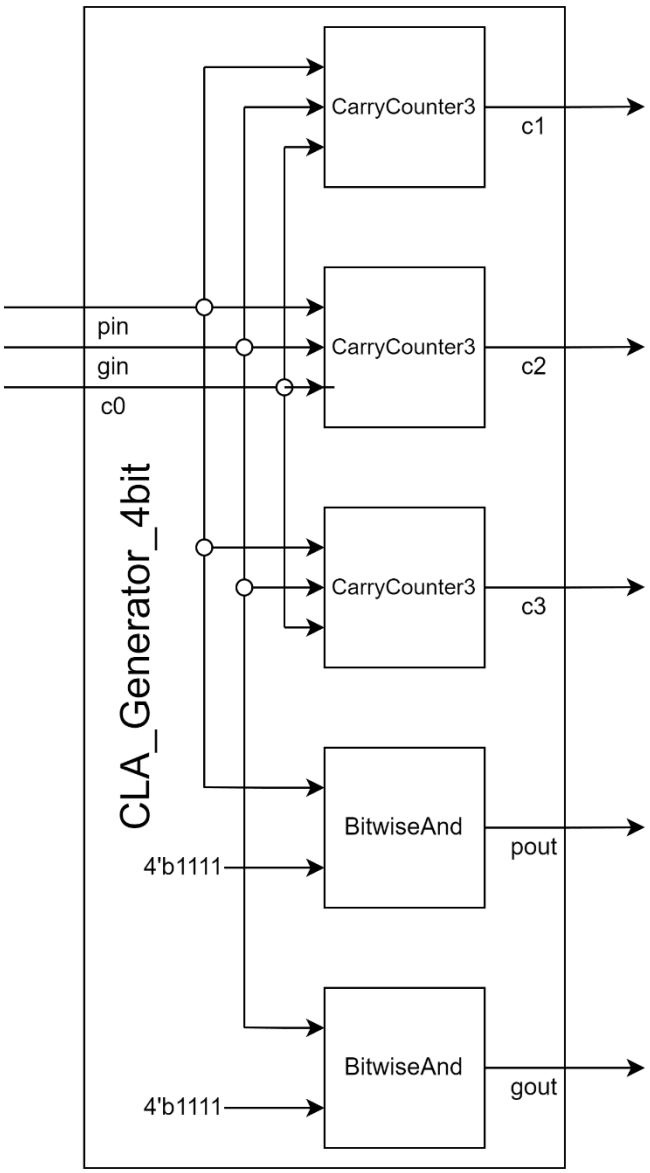

CarryCounter1
p[1]
And
g[0]
g[3:0]
Or_3to1
g[1]
c2
p[3:0]
p[1]
And_3to1
p[0]
c0

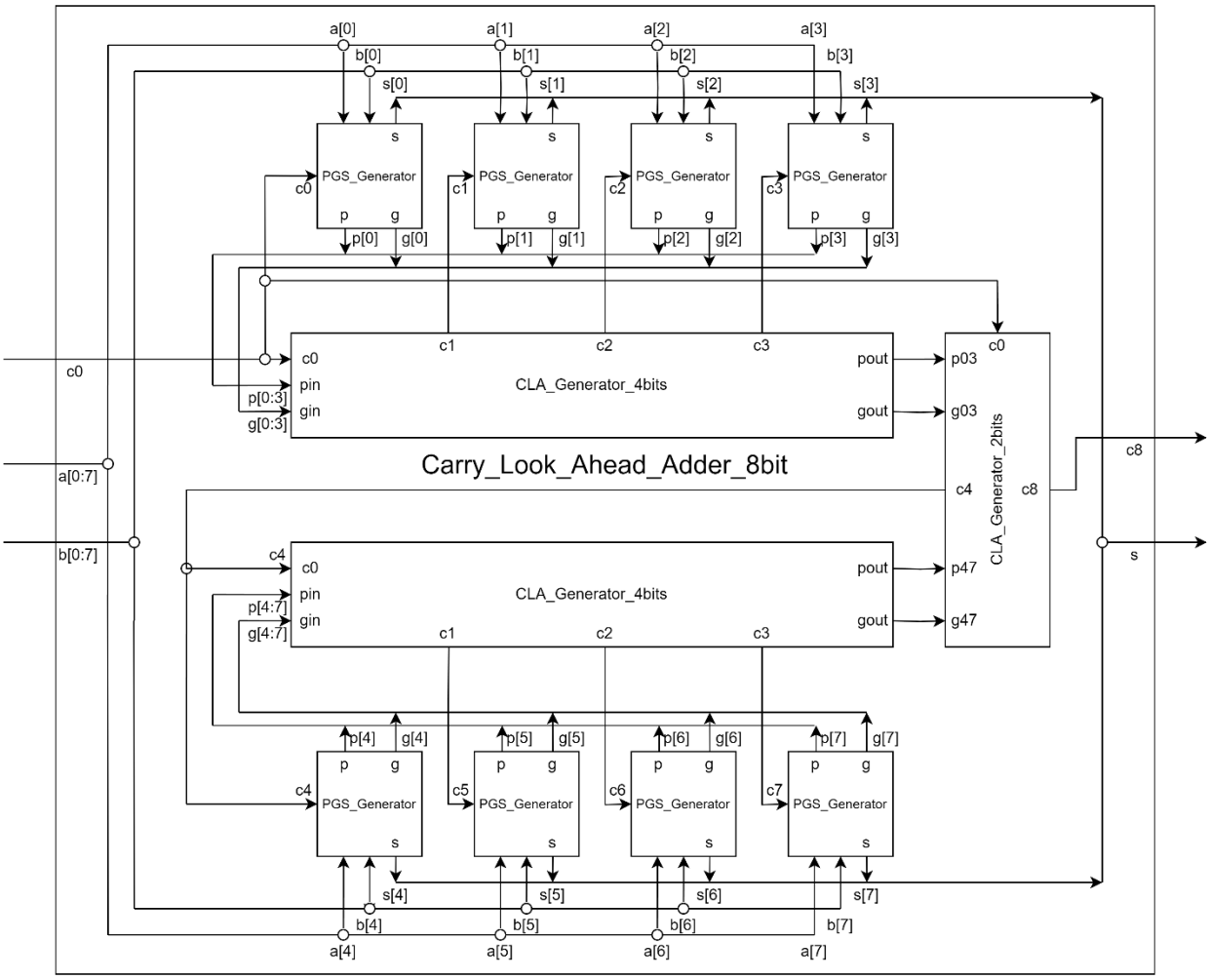

CarryCounter2
p[2]
And
g[1]
p[3:0]
And_3to1
Or_4to1
c3
p[2]
p[1]
g[2]
g[3:0]
g[0]
Or_4to1
p[2]
p[1]
p[0]
c0


CarryCounter3
And
p[3]
g[2]
p[3:0]
g[3:0]
And_3to1
p[3]
p[2]
g[1]
Or_4to1
g[3]
c4
Or_4to1
p[3]
p[2]
p[1]
g[0]
Or_5to1
p[3]
p[2]
p[1]
p[0]
c0


Or
p
And
g
a
b
Xor
s
c
Xor
PGS_Generator


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


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



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

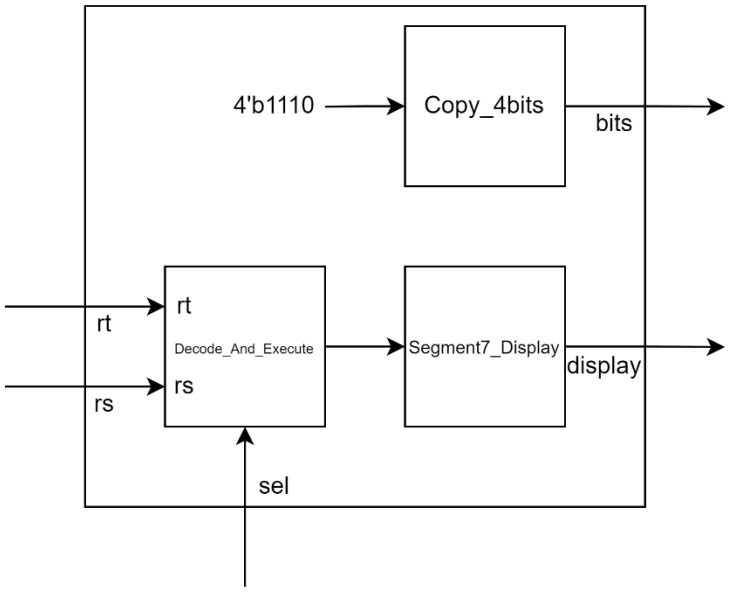
I set the simulation time to 1,0000,0000 ns and use three layers of loop (for **a**, **b**, **c0**) to go through every input pattern. And I also define a register called **err**. The testbench will check every result by my behavioral-level code. If the result from the gate-level circuit is different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

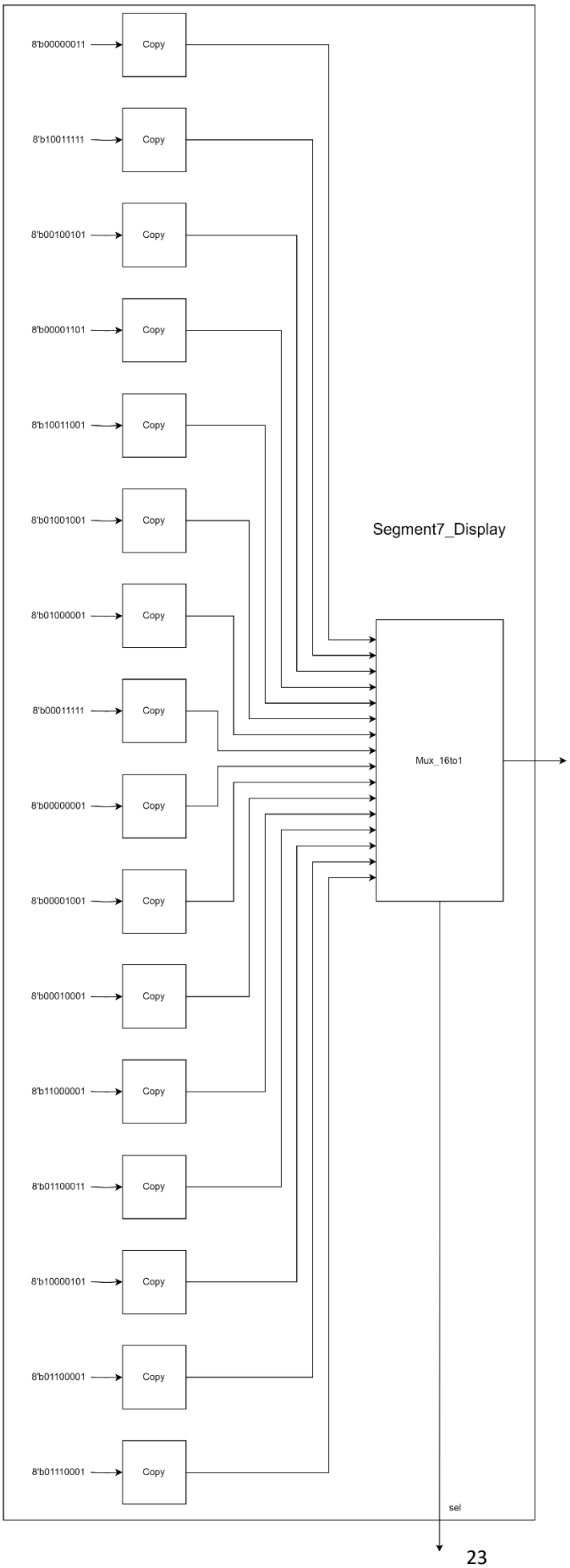
4. An exhaustive testbench design

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




8'b00000011
Copy
8'b10011111
Copy
8'b00100101
Copy
8'b00001101
Copy
8'b10011001
Copy
8'b01001001
Copy
8'b01000001
Copy
8'b00011111
Copy
8'b00000001
Copy
8'b00001001
Copy
8'b00010001
Copy
8'b11000001
Copy
8'b01100011
Copy
8'b10000101
Copy
8'b01100001
Copy
8'b01110001
Copy
Segment7_Display
Mux_16to1
sel

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.