Digital systems ENCS2340

Project Report

Design and Implementation of

a Simple ALU using Verilog HDL

1st semester 2023/2024

Instructor: Dr. Yazan Abo Farha

Prepared by: Qusay Bider

ID: 1220649

Table of Contents

Introduction:

In this project, you will design a simple Arithmetic Logic Unit (ALU) using Verilog Hardware Description Language (HDL). The ALU should be capable of performing four basic arithmetic and logic operations: addition, subtraction, bitwise AND, and bitwise OR.

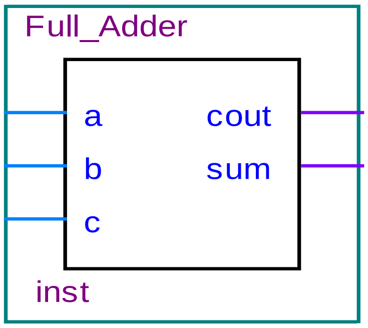
ALU:

An Arithmetic Logic Unit (ALU) is a digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers. It differs from a Floating-Point Unit (FPU), which handles floating-point numbers. The ALU is a crucial component in various computing circuits, including the Central Processing Unit (CPU) in computers, Floating-Point Units (FPUs), and Graphics Processing Units (GPUs). It's important to note that a single CPU, FPU, or GPU may contain multiple ALUs.

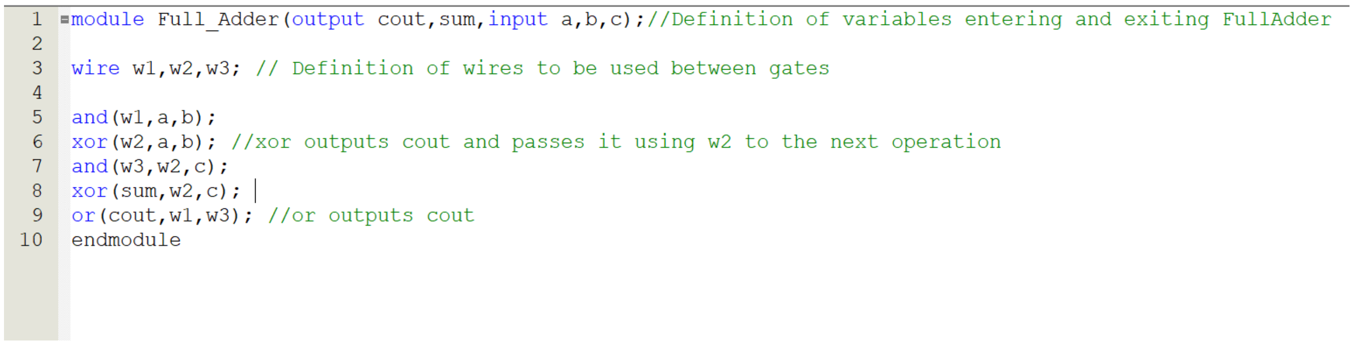
Used components:

Full Adder:

Full _Adder circuit has two inputs: X and Y it calculates the summation of them and set it as output (sum).

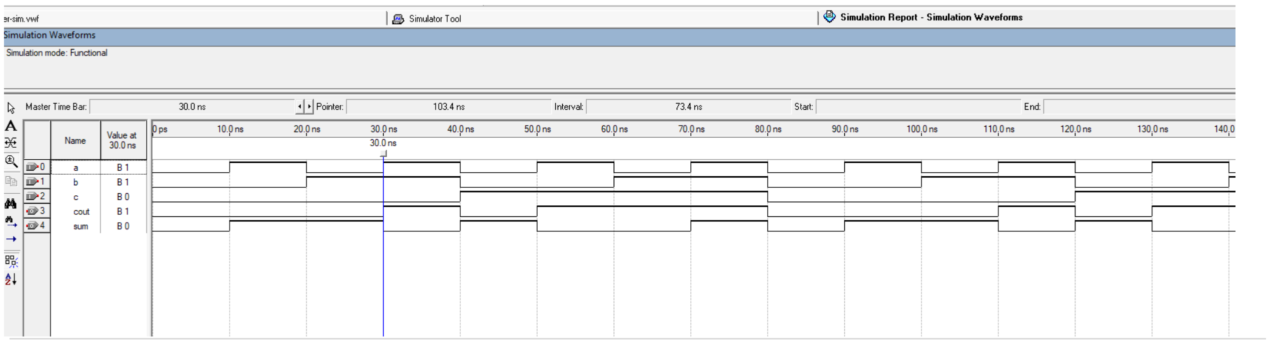


Code:



```
module Full_Adder(output cout,sum,input a,b,c);//Definition of variables entering and exiting FullAdder

wire w1,w2,w3; // Definition of wires to be used between gates

and(w1,a,b);
xor(w2,a,b); //xor outputs cout and passes it using w2 to the next operation
and(w3,w2,c);
xor(sum,w2,c);
or(cout,w1,w3); //or outputs cout
endmodule
```

Simulation:



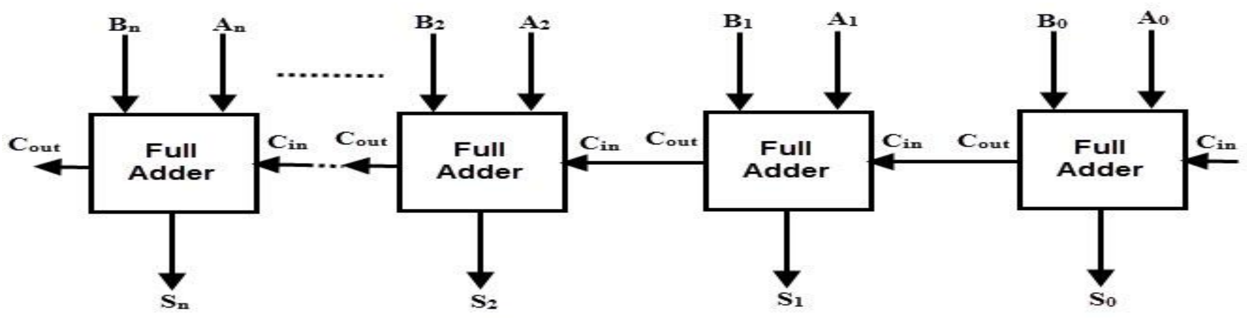
Simulation mode: Functional

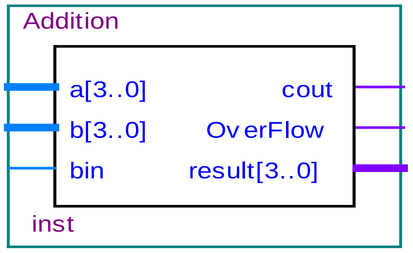
Master Time Bar: 30.0 ns Pointer: 103.4 ns Interval: 73.4 ns Start: End:

Name	Value at 30.0 ns
a	B 1
b	B 1
c	B 0
cout	B 1
sum	B 0

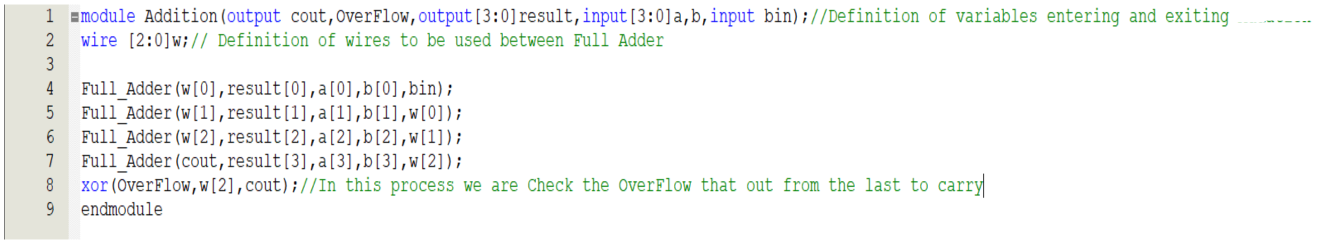
Addition:

Binary adders are implemented to add two binary numbers. So, in order to add two 4-bit binary numbers, we will need to use 4 full-adders. The connection of full-adders to create binary adder circuit is discussed in block diagram below.



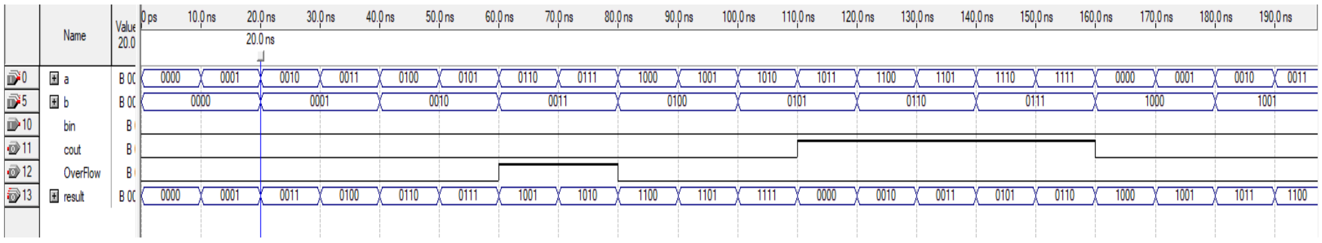


Code:



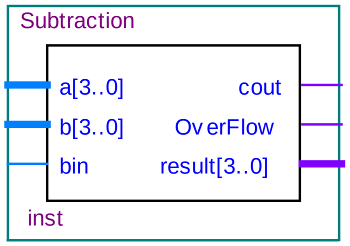
```
module Addition(output cout,OverFlow,output[3:0]result,input[3:0]a,b,input bin);//Definition of variables entering and exiting
wire [2:0]w;// Definition of wires to be used between Full Adder

Full_Adder(w[0],result[0],a[0],b[0],bin);
Full_Adder(w[1],result[1],a[1],b[1],w[0]);
Full_Adder(w[2],result[2],a[2],b[2],w[1]);
Full_Adder(cout,result[3],a[3],b[3],w[2]);
xor(OverFlow,w[2],cout);//In this process we are Check the OverFlow that out from the last to carry
endmodule
```

Simulation:



Subtraction:

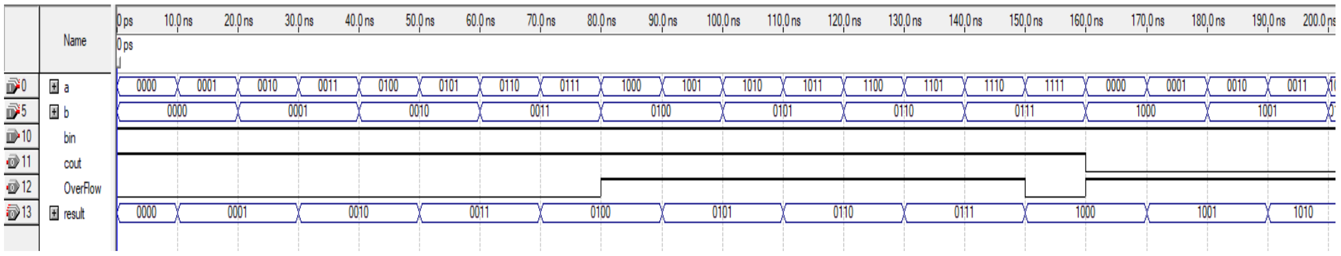
Subtractor circuit has three inputs: X & Y &bin it calculates the difference between them as (X – Y) and sets it as output (result) and put the carry in Cout and also check the Overflow .



Code:

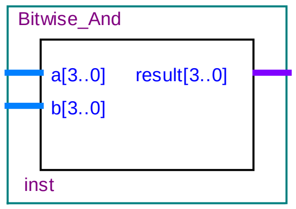
```
module Subtraction(output cout,OverFlow,output[3:0]result,input[3:0]a,b,input bin); //Definition of variables entering and Subtraction

wire [2:0]w;//Definition the wire that used between the The Full_Adder

Full_Adder(w[0],result[0],a[0],~b[0],bin);// bin is the first carray that take the value 1 to get the secound
Full_Adder(w[1],result[1],a[1],~b[1],w[0]);// (~)its used to get the one complemnt for the scound input .
Full_Adder(w[2],result[2],a[2],~b[2],w[1]);
Full_Adder(cout,result[3],a[3],~b[3],w[2]);
xor(OverFlow,w[2],cout);//In this process we are Check the OverFlow that out from the last to carry
endmodule
```

Simulation:

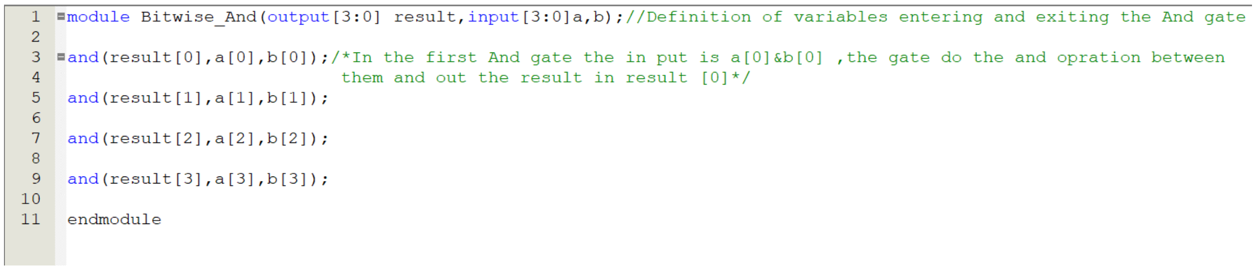


Bitwise AND:

This block has two n-bit inputs (a and b), and one output (result), The bitwise And circuit check a and b bit by bit, if they are both one, the result of this bit in the f output will be one, otherwise, it will be zero.

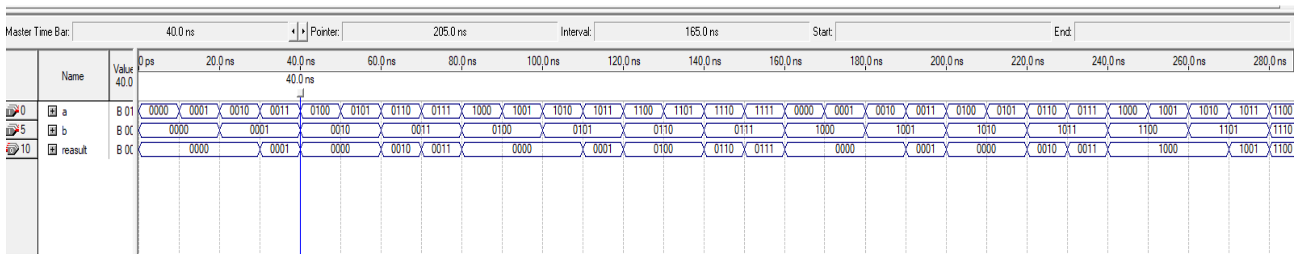


Code:



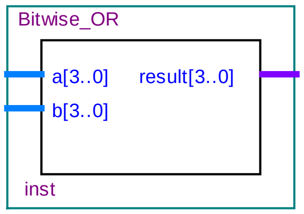
```
module Bitwise_And(output[3:0] result,input[3:0]a,b);//Definition of variables entering and exiting the And gate

and(result[0],a[0],b[0]);/*In the first And gate the in put is a[0]&b[0] ,the gate do the and opration between
                           them and out the result in result [0]*/
and(result[1],a[1],b[1]);

and(result[2],a[2],b[2]);

and(result[3],a[3],b[3]);

endmodule
```

Simulation:


Master Time Bar: 40.0 ns
Pointer: 205.0 ns
Interval: 165.0 ns
Start:
End:
Name
Value 40.0
0 ps
20.0 ns
40.0 ns
60.0 ns
80.0 ns
100.0 ns
120.0 ns
140.0 ns
160.0 ns
180.0 ns
200.0 ns
220.0 ns
240.0 ns
260.0 ns
280.0 ns
40.0 ns
0 a B 01 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100
5 b B 00 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110
10 reasult B 00 0000 0001 0000 0010 0011 0000 0001 0100 0110 0111 0000 0001 0000 0010 0011 1000 1001 1100

Bitwise OR:

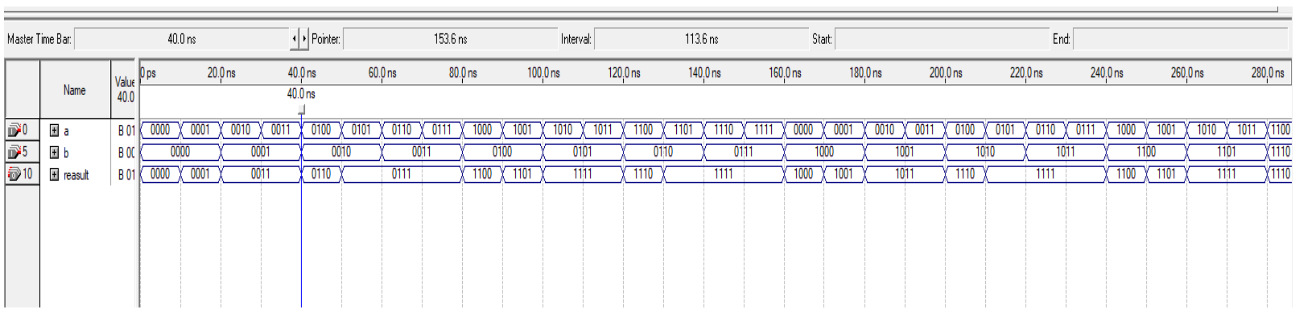
This block has two n-bit inputs (a and b), and one output (result), The bitwise OR circuit check a and b bit by bit, if they are both zero, the result of this bit in the f output will be zero, otherwise, it will be one.



Code:

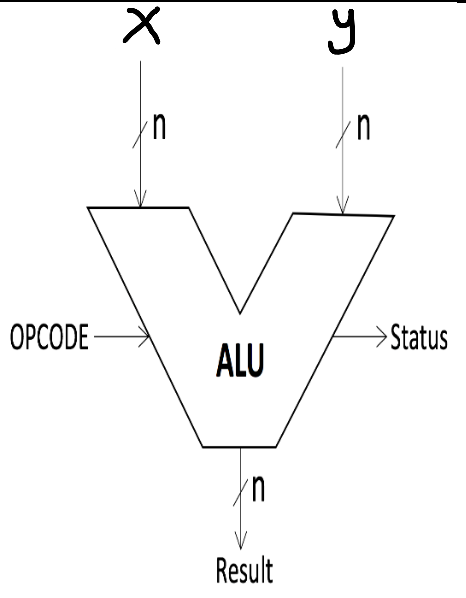
```
module Bitwise_OR(output[3:0] result,input[3:0]a,b);//Definition of variables entering and exiting the OR gate

or(result[0],a[0],b[0]);/*In the first or gate the input is a[0]&b[0] ,the gate do the OR opration between
                          them and out the result in result [0]*/
or(result[1],a[1],b[1]);

or(result[2],a[2],b[2]);

or(result[3],a[3],b[3]);

endmodule
```

Simulation:


Master Time Bar: 40.0 ns
Pointer: 153.6 ns
Interval: 113.6 ns
Start:
End:
Name
Value 40.0
0 ps
20.0 ns
40.0 ns
60.0 ns
80.0 ns
100.0 ns
120.0 ns
140.0 ns
160.0 ns
180.0 ns
200.0 ns
220.0 ns
240.0 ns
260.0 ns
280.0 ns
40.0 ns
0 a B 01
5 b B 00
10 reasult B 01
a: 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100
b: 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110
reasult: 0000 0001 0011 0110 0111 1100 1101 1111 1110 1111 1000 1001 1011 1110 1111 1100 1101 1111 1110

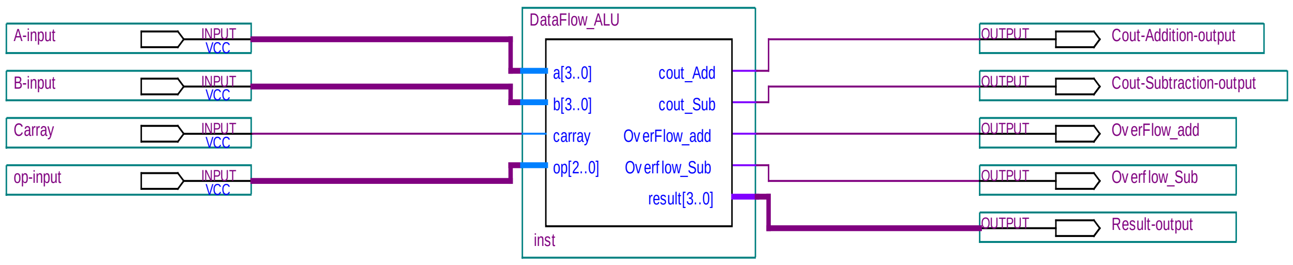
ALU Implementation:

This project is designed to do multiple tasks on n-bits binary numbers as shown in the table below.

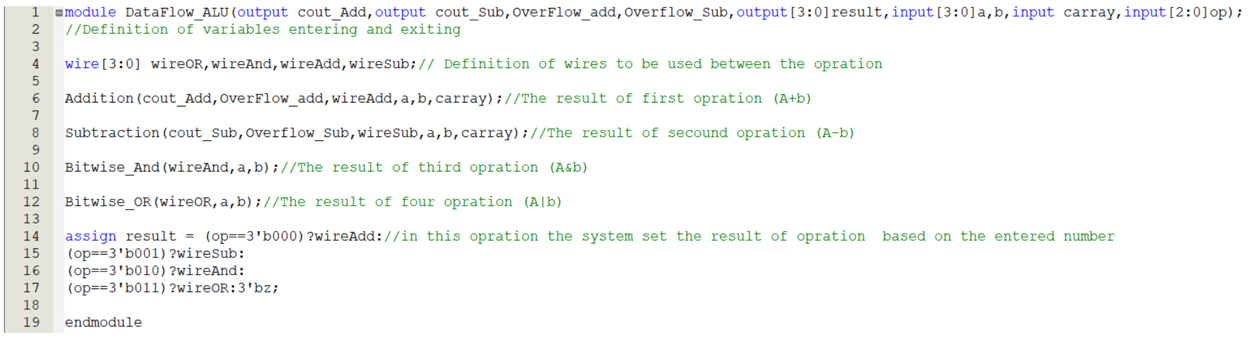
ALU Function code	ALU Output (Result)	ALU Symbol
000	(X + Y)	
001	(X – Y)	
010	(X & Y)	
011	(X | Y)	
101	NO Output	
110	NO Output	
111	NO Output	

Data Flow ALU:

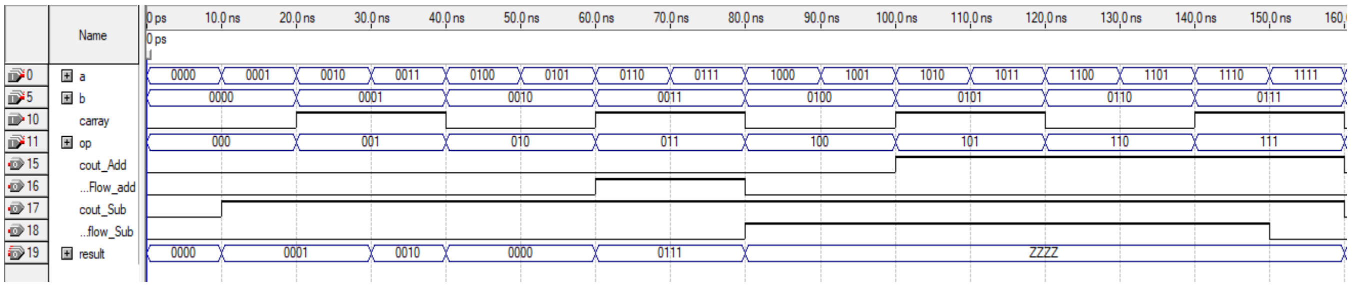
The ALU is the combinational circuit responsible for all the arithmetic operations in a digital computer, containing the half adder, subtractor, and multiplier. The ALU designed per this project is a limited 4-bit.



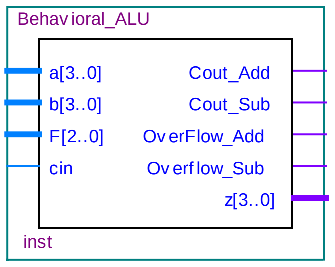
Code:


```
module DataFlow_ALU(output cout_Add,output cout_Sub,OverFlow_add,Overflow_Sub,output[3:0]result,input[3:0]a,b,input carray,input[2:0]op);
//Definition of variables entering and exiting

wire[3:0] wireOR,wireAnd,wireAdd,wireSub;// Definition of wires to be used between the opration

Addition(cout_Add,OverFlow_add,wireAdd,a,b,carray);//The result of first opration (A+b)

Subtraction(cout_Sub,Overflow_Sub,wireSub,a,b,carray);//The result of secound opration (A-b)

Bitwise_And(wireAnd,a,b);//The result of third opration (A&b)

Bitwise_OR(wireOR,a,b);//The result of four opration (A|b)

assign result = (op==3'b000)?wireAdd://in this opration the system set the result of opration  based on the entered number
(op==3'b001)?wireSub:
(op==3'b010)?wireAnd:
(op==3'b011)?wireOR:3'bz;

endmodule
```

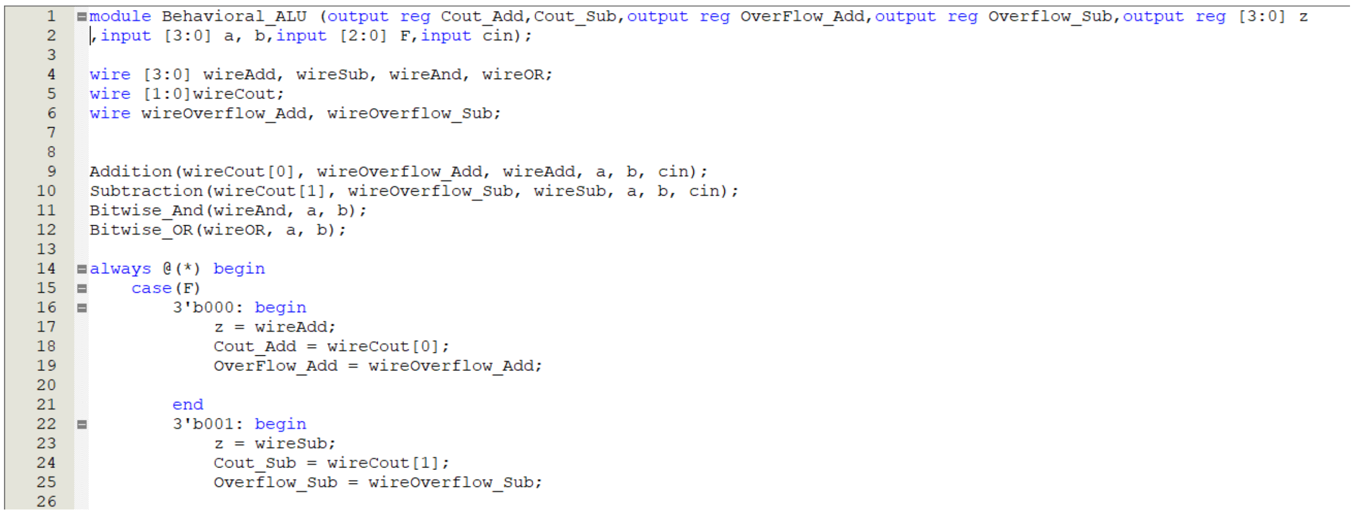
Simulation:

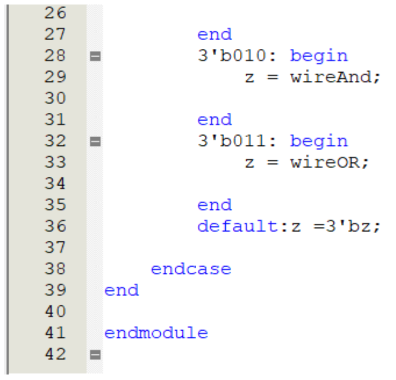


Behavioral ALU:

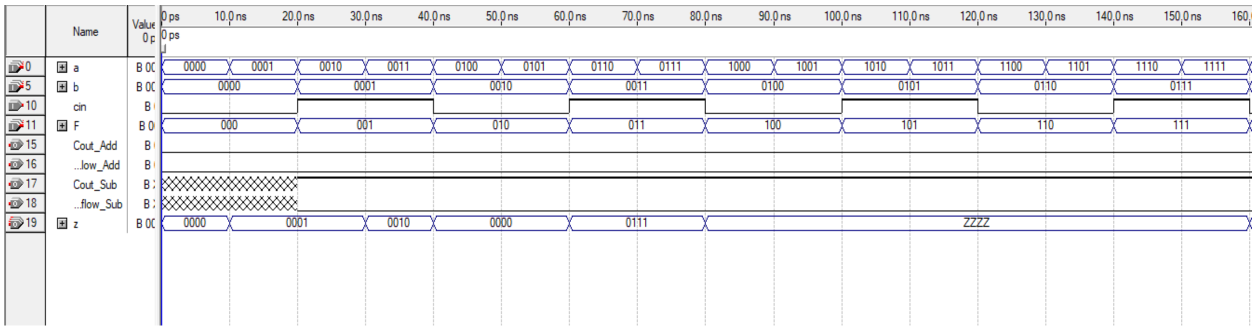


Code:


```
module Behavioral_ALU (output reg Cout_Add,Cout_Sub,output reg OverFlow_Add,output reg Overflow_Sub,output reg [3:0] z
,input [3:0] a, b,input [2:0] F,input cin);

wire [3:0] wireAdd, wireSub, wireAnd, wireOR;
wire [1:0]wireCout;
wire wireOverflow_Add, wireOverflow_Sub;


Addition(wireCout[0], wireOverflow_Add, wireAdd, a, b, cin);
Subtraction(wireCout[1], wireOverflow_Sub, wireSub, a, b, cin);
Bitwise_And(wireAnd, a, b);
Bitwise_OR(wireOR, a, b);

always @(*) begin
    case(F)
        3'b000: begin
            z = wireAdd;
            Cout_Add = wireCout[0];
            OverFlow_Add = wireOverflow_Add;

        end
        3'b001: begin
            z = wireSub;
            Cout_Sub = wireCout[1];
            Overflow_Sub = wireOverflow_Sub;

```


```

        end
        3'b010: begin
            z = wireAnd;

        end
        3'b011: begin
            z = wireOR;

        end
        default:z =3'bz;

    endcase
end

endmodule

```

Simulation:


Name
Value
0 ps
10.0 ns
20.0 ns
30.0 ns
40.0 ns
50.0 ns
60.0 ns
70.0 ns
80.0 ns
90.0 ns
100.0 ns
110.0 ns
120.0 ns
130.0 ns
140.0 ns
150.0 ns
a 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
b 0000 0001 0010 0011 0100 0101 0110 0111
cin
F 000 001 010 011 100 101 110 111
Cout_Add
...low_Add
Cout_Sub
...flow_Sub
z 0000 0001 0010 0000 0111 ZZZZ