Faculty of Engineering Technology

Electrical & Computer Engineering Department

First Semester-2023/2024

ENCS2340, Digital Systems

Project Report

Prepared by: Abdulrahman Rateb Atyani

student ID: 1221808

Instructor: Dr. Ali Abdo

Section: 3

Contents

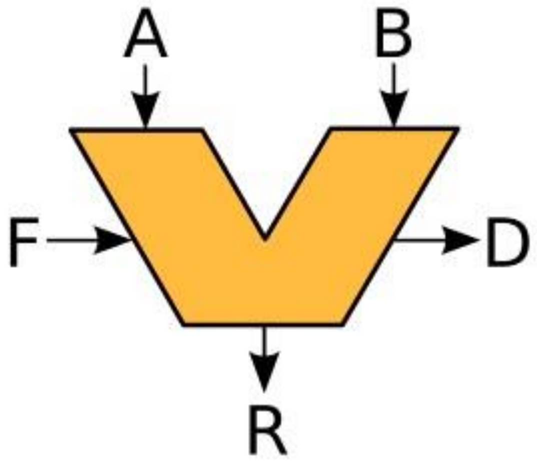
*introduction of Alu

It is a digital department located inside the central processing unit responsible for performing all arithmetic operations (such as addition and subtraction) on integers. The device treats them as binary numbers 0 and 1. It also performs comparison operations to find out the result of logical comparisons, which are: (greater than, less than, equal to, and not equal to) and derivatives. These comparisons also provide the ability to temporarily store information in addition to the ability to process information .

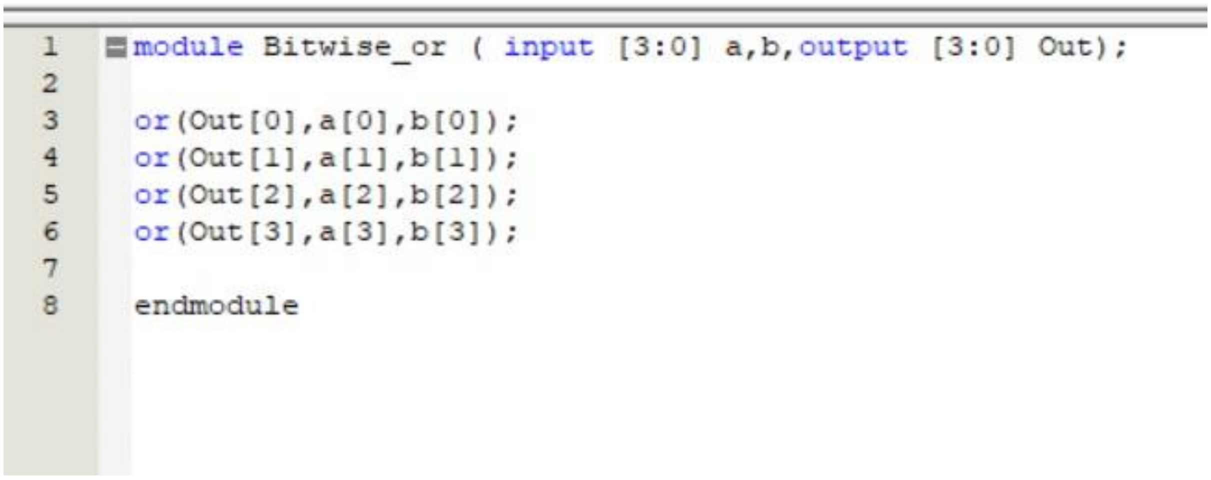
*This figure show the ALU Sturcture

Inputs:A[n-1:0],b[n-1:0],F[1:0].

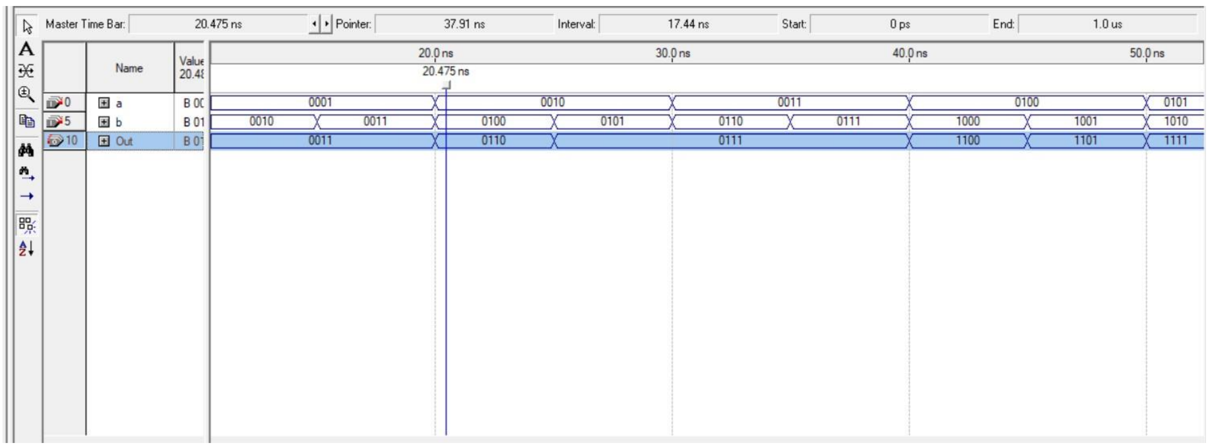
Outputs:D(carry out),R[n-1:0] (result).



Bitwise Or



```
module Bitwise_or ( input [3:0] a,b,output [3:0] Out);

or(Out[0],a[0],b[0]);
or(Out[1],a[1],b[1]);
or(Out[2],a[2],b[2]);
or(Out[3],a[3],b[3]);

endmodule
```

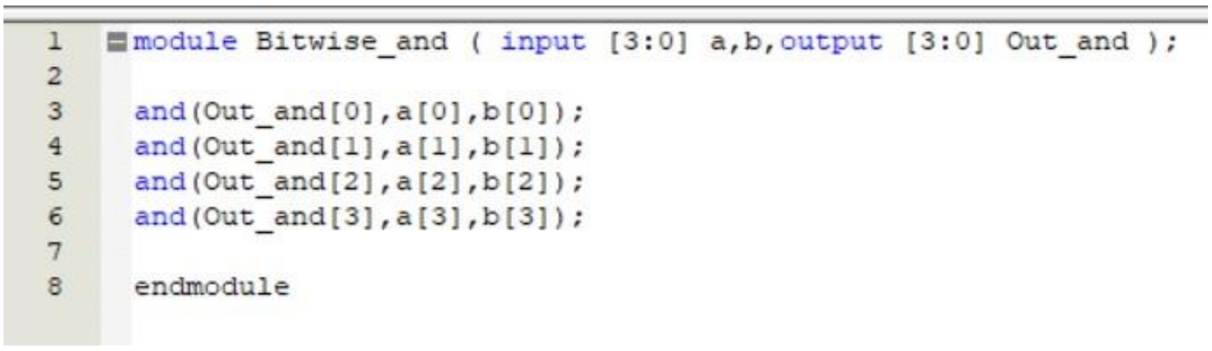


```
Master Time Bar: 20.475 ns   Pointer: 37.91 ns   Interval: 17.44 ns   Start: 0 ps   End: 1.0 us
```

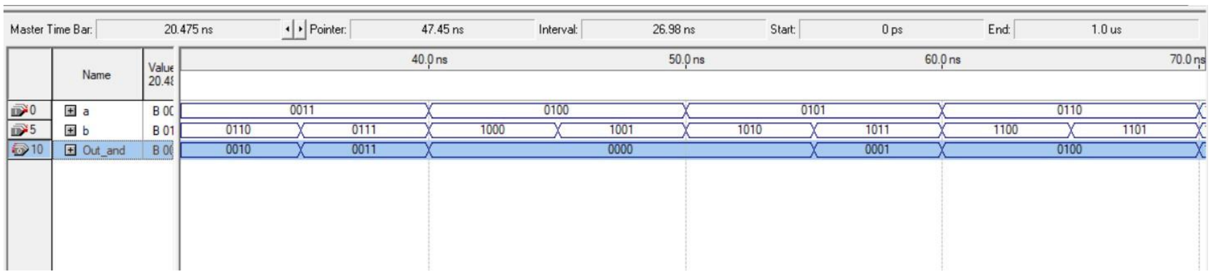
Name
Value 20.4
20.0 ns
20.475 ns
30.0 ns
40.0 ns
50.0 ns
0 a B 00 0001 0010 0011 0100 0101
5 b B 01 0010 0011 0100 0101 0110 0111 1000 1001 1010
10 Out B 01 0011 0110 0111 1100 1101 1111

It is clear from the above that the code pertains to the Or operation, as it works to take values in binary as a value separately and compares according to the principle of or, and thus outputs the value to us based on the above. As for the waveform, what I said is represented by it, and where I entered two inputs and output a value, and for example, it is clear by the first The clip says that when I entered the number 0 and then the number 1, it gave me the number 1, and this is a correct result,and the same applies to the rest .

Bitwise And



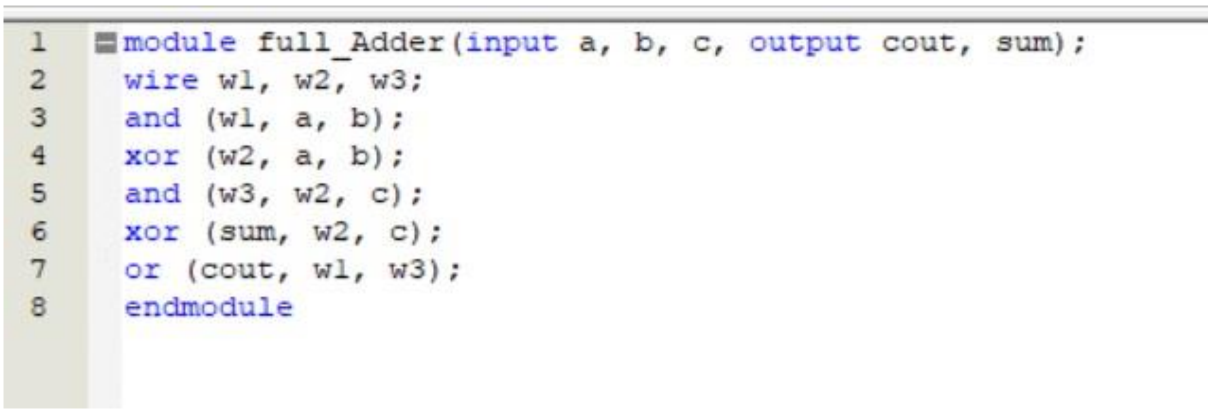
```
module Bitwise_and ( input [3:0] a,b,output [3:0] Out_and );

and(Out_and[0],a[0],b[0]);
and(Out_and[1],a[1],b[1]);
and(Out_and[2],a[2],b[2]);
and(Out_and[3],a[3],b[3]);

endmodule
```


Master Time Bar: 20.475 ns
Pointer: 47.45 ns
Interval: 26.98 ns
Start: 0 ps
End: 1.0 us
Name
Value 20.48
40.0 ns
50.0 ns
60.0 ns
70.0 ns
a B 00 0011 0100 0101 0110
b B 01 0110 0111 1000 1001 1010 1011 1100 1101
Out_and B 00 0010 0011 0000 0001 0100

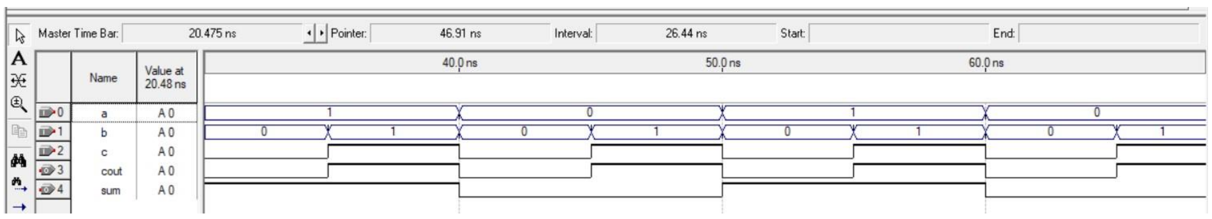
It is clear from the above that the code pertains to the and operation, as it works to take values in binary as a value separately and compares according to the principle of and, and thus outputs the value to us based on the above. As for the waveform, what I said is represented by it, and where I entered two inputs (each input contains 4 bit)and output a value(that contains 4 bit), and for example, it is clear by the second The clip says that when I entered the number 4 and then the number 8, it gave me the number 0, and this is a correct result,and the same applies to the rest .

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Full Adder

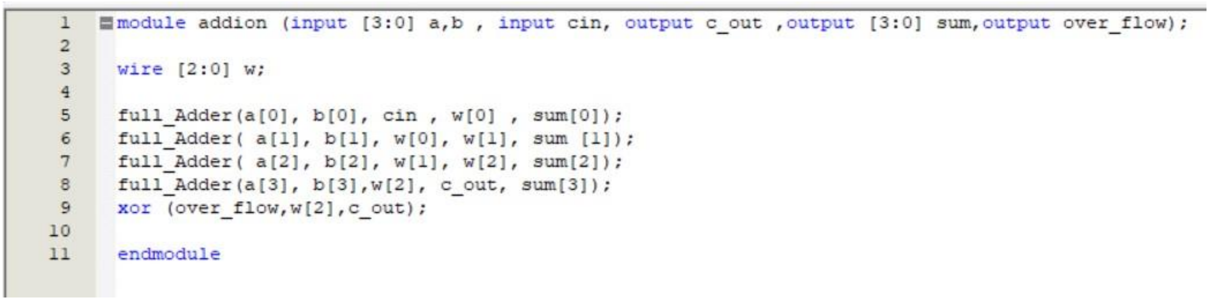


```
module full_Adder(input a, b, c, output cout, sum);
wire w1, w2, w3;
and (w1, a, b);
xor (w2, a, b);
and (w3, w2, c);
xor (sum, w2, c);
or (cout, w1, w3);
endmodule
```

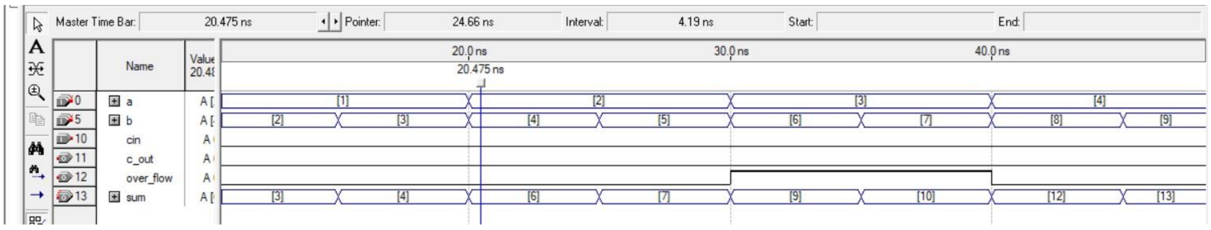


It is clear from the previous code how to build a full adder from a group of gates, and how it works by adding three bits together. The principle of its work is based on sum and carry, where it collects the bits and puts the result in sum, and what is more than the sum is put in carry, and what proves the truth of my words is what we see in Semulation, where, for example, in the first clip, the input values are as follows: a=1, b=1, c=1, and this results in sum = 1, and the value of crray = 1, and this applies to The rest of the clips are in simulation.

Addion Operation

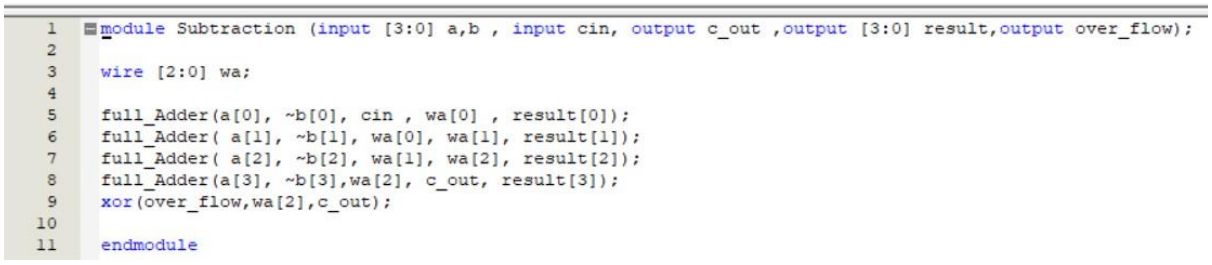


```
module addion (input [3:0] a,b , input cin, output c_out ,output [3:0] sum,output over_flow);

wire [2:0] w;

full_Adder(a[0], b[0], cin , w[0] , sum[0]);
full_Adder( a[1], b[1], w[0], w[1], sum [1]);
full_Adder( a[2], b[2], w[1], w[2], sum[2]);
full_Adder(a[3], b[3],w[2], c_out, sum[3]);
xor (over_flow,w[2],c_out);

endmodule
```

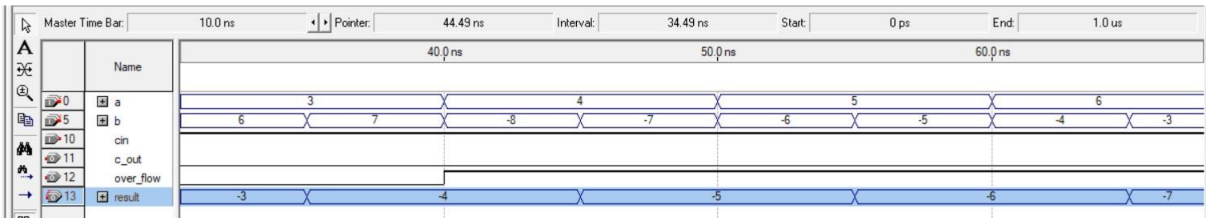


We see in this summation process that we have made a call to the module of full adder four times for each bit once in order to sum the inputs a and b. As for the third input, which is cin, its value is always zero. We also notice in simulation that there is a new output, which is over flow, which appears when it is last. two carry is different, or when adding two positive numbers and the result is negative, or two negative numbers and the result is positive, for example in the third section when adding 3 with 6, the result is 9, and there is also over flow in addition to the lack of carry, and this is what the addition process looks like.

Subtraction Operation

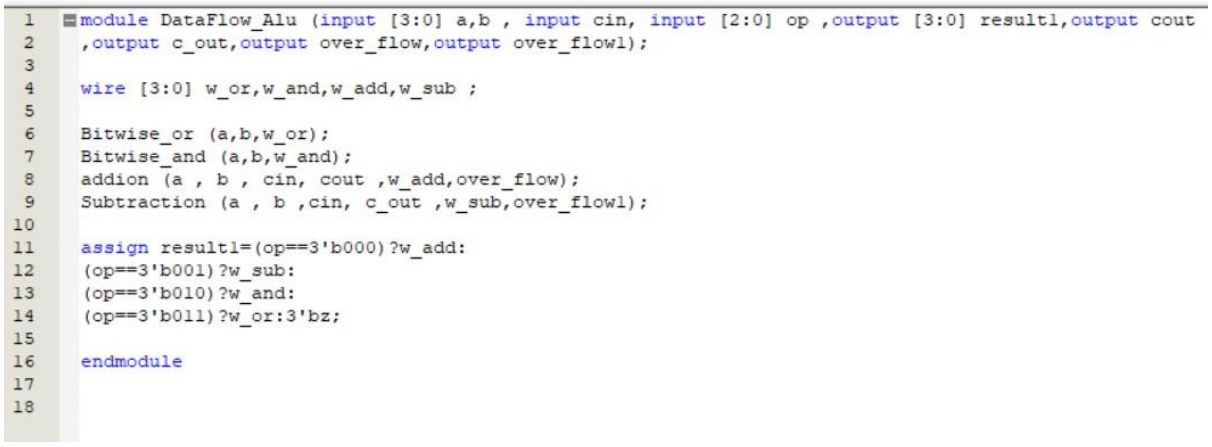


```
module Subtraction (input [3:0] a,b , input cin, output c_out ,output [3:0] result,output over_flow);

wire [2:0] wa;

full_Adder(a[0], ~b[0], cin , wa[0] , result[0]);
full_Adder( a[1], ~b[1], wa[0], wa[1], result[1]);
full_Adder( a[2], ~b[2], wa[1], wa[2], result[2]);
full_Adder(a[3], ~b[3],wa[2], c_out, result[3]);
xor(over_flow,wa[2],c_out);

endmodule
```

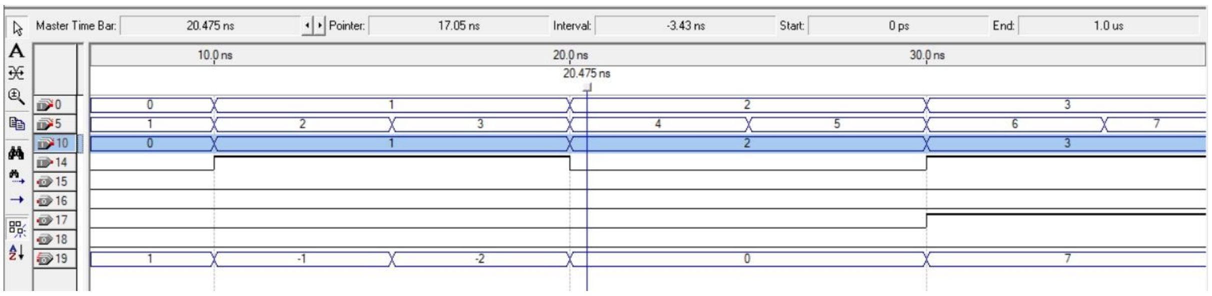


We see in this subtraction process that we did the same thing by calling full adder four times for each bit once in order to subtract the inputs a and b. As for the third input, which is cin, its value is always 1, as we notice in the code that we took the value of 1s complement of b, in order to add This value is combined with cin and appears as -b. We also notice in the simulation that there is a new output, which is over flow, which appears as it appears during the addition process. For example, in the second section, when 4 is subtracted with -8, the result is -4, and this is illogical, so over flow appears, and this picture that show the subtraction process.

Data Flow_Alu

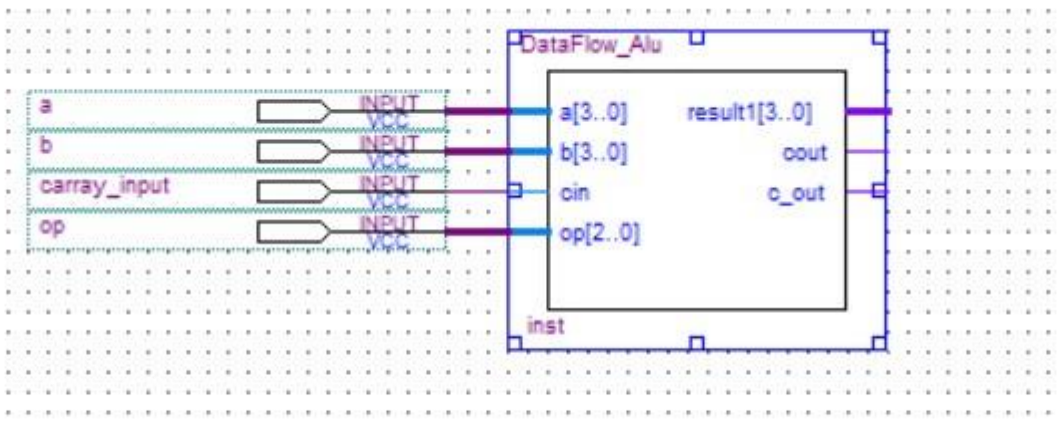


```
module DataFlow_Alu (input [3:0] a,b , input cin, input [2:0] op ,output [3:0] result1,output cout
,output c_out,output over_flow,output over_flow1);

wire [3:0] w_or,w_and,w_add,w_sub ;

Bitwise_or (a,b,w_or);
Bitwise_and (a,b,w_and);
addion (a , b , cin, cout ,w_add,over_flow);
Subtraction (a , b ,cin, c_out ,w_sub,over_flow1);

assign result1=(op==3'b000)?w_add:
(op==3'b001)?w_sub:
(op==3'b010)?w_and:
(op==3'b011)?w_or:3'bz;

endmodule
```

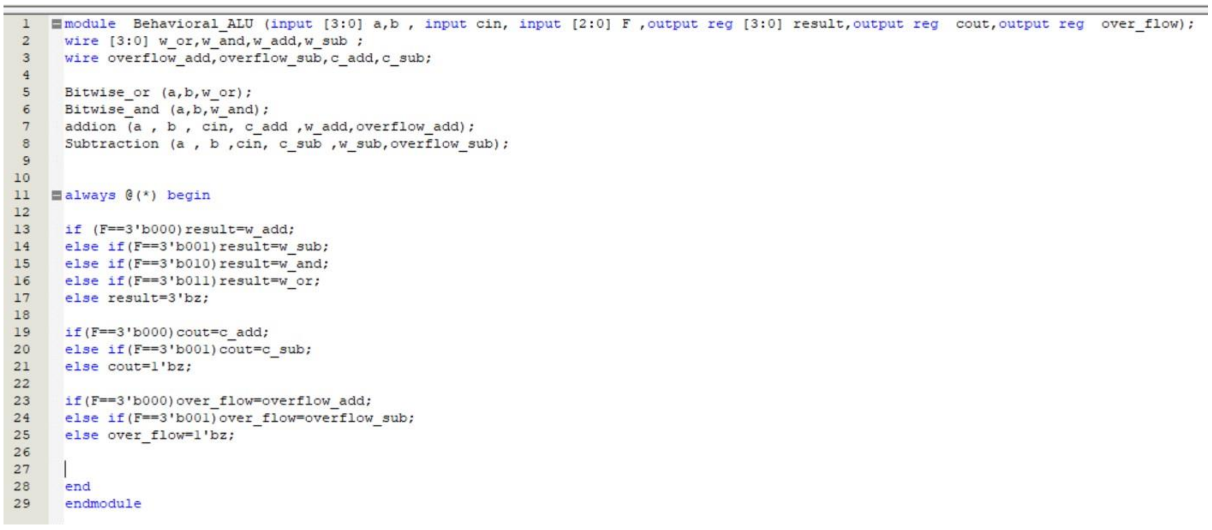


Master Time Bar: 20.475 ns Pointer: 17.05 ns Interval: -3.43 ns Start: 0 ps End: 1.0 us

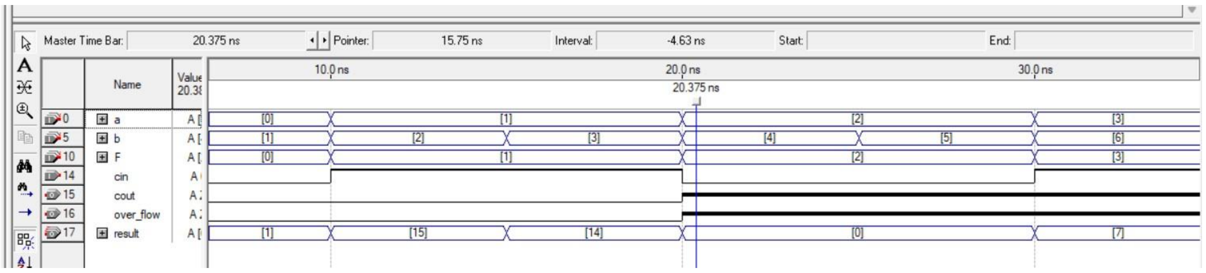
Here we see how to build Alu through the Data flow method, which is characterized by assign keyword, where we collected all the modules by calling them and placing them inside the shown module, where we added a new input, which is opcode, which determines the calculation process through the conditions inside the code, as is shown in semulation Each section represents a mathematical operation. The first, for example, represents the addition operation, where we entered 1 with 0 and the result was 1. In the second section, we performed the subtraction operation, the third section performed the and operation between inputs, and the last section we performed the or operation.


DataFlow_Alu
a
b
carray_input
op
INPUT
VCC
a[3..0]
b[3..0]
cin
op[2..0]
result1[3..0]
cout
c_out
inst

behaviroal_Alu

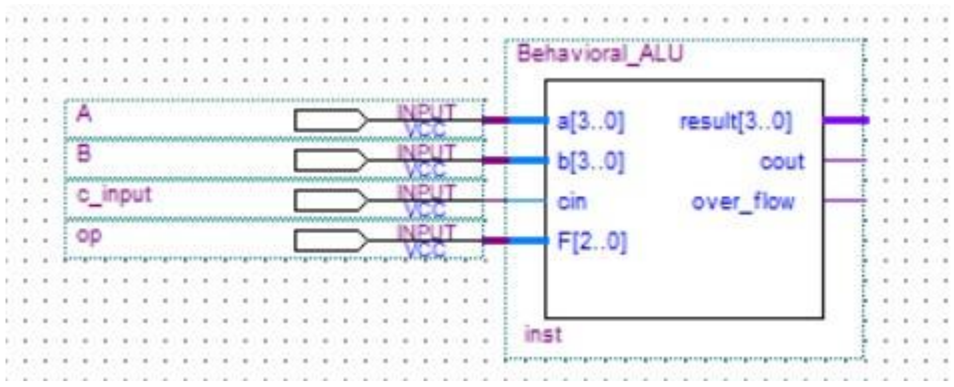


```verilog
module  Behavioral_ALU (input [3:0] a,b , input cin, input [2:0] F ,output reg [3:0] result,output reg  cout,output reg  over_flow);
 wire [3:0] w_or,w_and,w_add,w_sub ;
 wire overflow_add,overflow_sub,c_add,c_sub;

 Bitwise_or (a,b,w_or);
 Bitwise_and (a,b,w_and);
 addion (a , b , cin, c_add ,w_add,overflow_add);
 Subtraction (a , b ,cin, c_sub ,w_sub,overflow_sub);


always @(*) begin

 if  (F==3'b000)result=w_add;
 else if(F==3'b001)result=w_sub;
 else if(F==3'b010)result=w_and;
 else if(F==3'b011)result=w_or;
 else result=3'bz;

 if(F==3'b000)cout=c_add;
 else if(F==3'b001)cout=c_sub;
 else cout=1'bz;

 if(F==3'b000)over_flow=overflow_add;
 else if(F==3'b001)over_flow=overflow_sub;
 else over_flow=1'bz;


 end
 endmodule
```



Here we see how to build Alu through the behavioral method, which is characterized by the always keyword, where we collected all the modules by calling them and placing them inside the shown module, where we added the input itself that is present in the data flow, which is the opcode that determines the computational process through the conditions inside the code, As it becomes clear in semulation, each stanza represents a mathematical operation. The first, for example, represents the addition operation, where we entered 1 with 0 and the result was 1. In the second stanza we performed the subtraction operation, the third stanza we performed the and operation between inputs, and the last stanza we performed the or operation.

This is the block diagram of behavirol _Alu


Behavioral_ALU
A
B
c_input
op
INPUT
VCC
INPUT
VCC
INPUT
VCC
INPUT
VCC
a[3..0]
b[3..0]
cin
F[2..0]
result[3..0]
cout
over_flow
inst