



Project title :

Modeling a Multifunction ALU

Name : Osaid Hasan Nur

ID : 1210733

Instructor :

Dr. Mohammad Hussein

Section : 3

Date : 7/2/2023

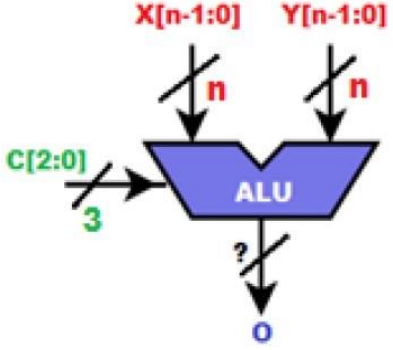
Contents

Title	Page
Introduction	3
ALU implementation	4
Adder – Subtractor	6
Multiplier	7
Mux	8
NAND	9
NOT	10
NOR	11
XOR	12
ALU (Structural)	13
Simulation for Structural	14
ALU (Behavioural)	16
Simulation for behavioural	17

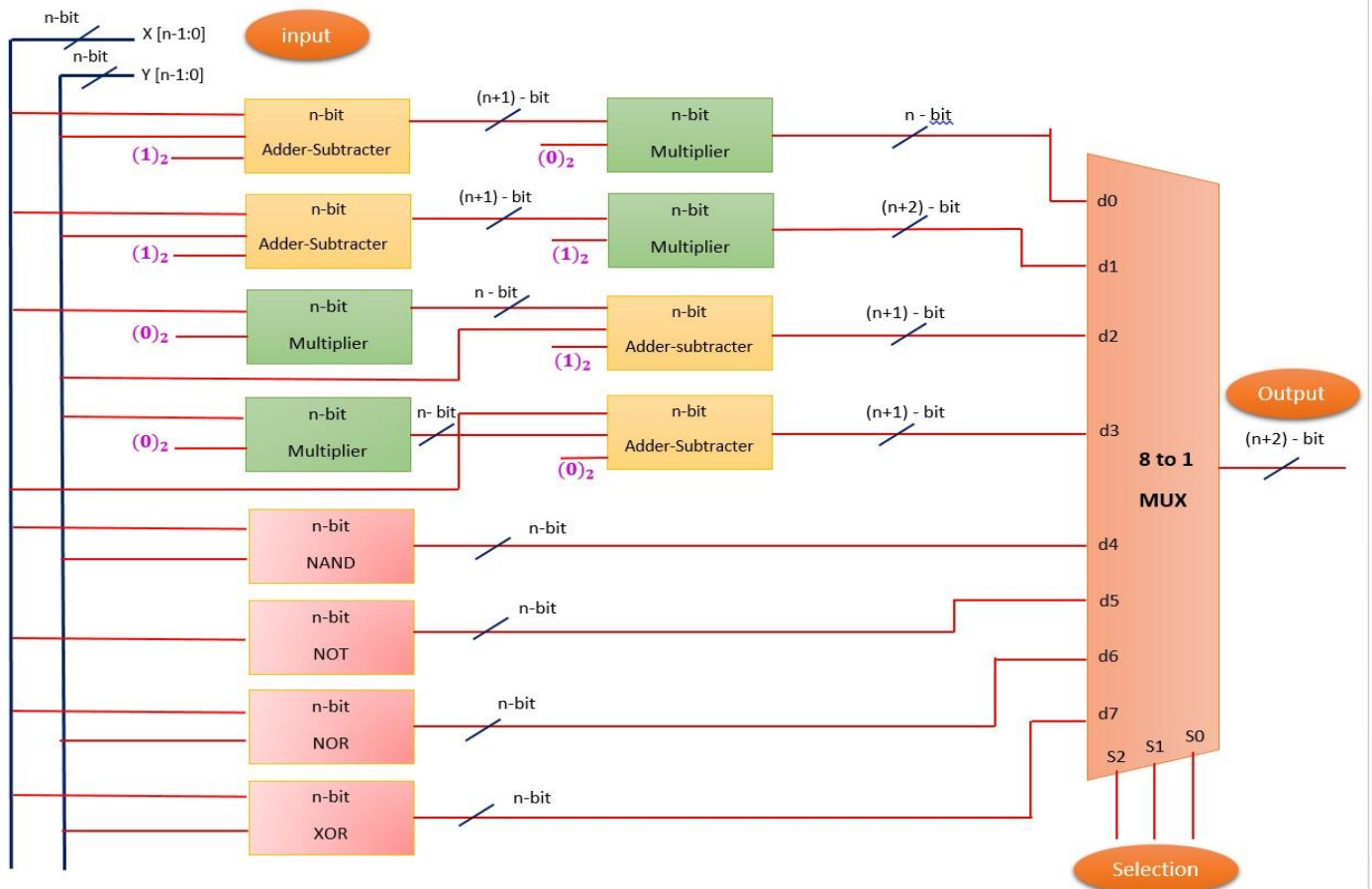
Introduction

In this project, I 'm going to design a multifunction arithmetic and logic unit (ALU) that do 8 operation at all, 4 arithmetic operations and 4 logic operations.

First , I will show ALU implementation using block diagram , then , I will show implementation for every component with simulation and using behavioural Verilog(HDL) , then , I will present the whole ALU implementation using structural method , after that I will make the same ALU but with behavioural method . At last, I will show simulation for the ALU in two methods.

ALU Function Code (C)	ALU Output (O)	ALU Symbol
000	$(X+Y)/2$	
001	$2*(X+Y)$	
010	$(X/2)+Y$	
011	$X-(Y/2)$	
100	X NAND Y	
101	NOT(X)	
110	X NOR Y	
111	X XOR Y	

ALU implementation



A) The size of output I used is $n+2$.

To explain why $n+2$, suppose I have an input of X, Y with signed magnitude represented in 2's complement, and suppose the value of n is 3, then the maximum value of x and y in our case is 101 which represented as -4,4 in decimal (using signed magnitude represented in 2's complement), "we found the maximum value(worst case)".

->>> when adding this two numbers, we have $-4 + -4 = -8$, and this number can't be represented in 3 bits (n) (overflow will occur), it needs 4 bits ($n+1$) to represent it, this implies that for adding process we need an output of $n+1$ bits. The subtraction process is almost the same and needs also $n+1$ bits to represent.

->>> When multiplying x and y , the result of multiplying is $-4 * -4 = 16$. This number can't be represented in 3 bits(n), or even in 4bits($n+1$)(maximum number can be presented in 4 bits is 15), so we need 5bits($n+2$) to represent the result of multiplying, a special case for this process is multiplying the number by a fractional number, in this case, after the multiplication, the number will get smaller, so we don't have to use any additional bit, so the output will need(n) bits only.

The logic operations (NAND, NOT, NOR, XOR) don't need any additional bit in the output, for example, if we take the values of X and Y in the first paragraph,

$$\text{NAND}(x, y) = \text{NAND}(-4, -4) = \text{NAND}((100)_2, (100)_2) = (011)_2$$

$$\text{NOT}(X) = \text{NOT}(100) = 001.$$

$$\text{NOR}(X, Y) = \text{NOR}(-4, 4) = \text{NOR}((100)_2, (100)_2) = (011)_2.$$

$$\text{XOR}(X, Y) = \text{XOR}(-4, -4) = \text{XOR}((100)_2, (100)_2) = (000)_2.$$

All of previous logical operation can be presented in 3 bits without any need of additional bits.

We notice that the maximum number of bits needed for all 8 operations is ($n+2$)at multiplication operation.

The output size needed for every operation:

Operation	Output Size
$(X+Y)/2$	N
$2*(X+y)$	$N+2$
$(X/2)+Y$	$N+1$
$X - (Y/2)$	$N+1$
$X \text{ NAND } Y$	N
$\text{NOT}(X)$	N
$X \text{ NOR } Y$	N
$X \text{ XOR } Y$	N

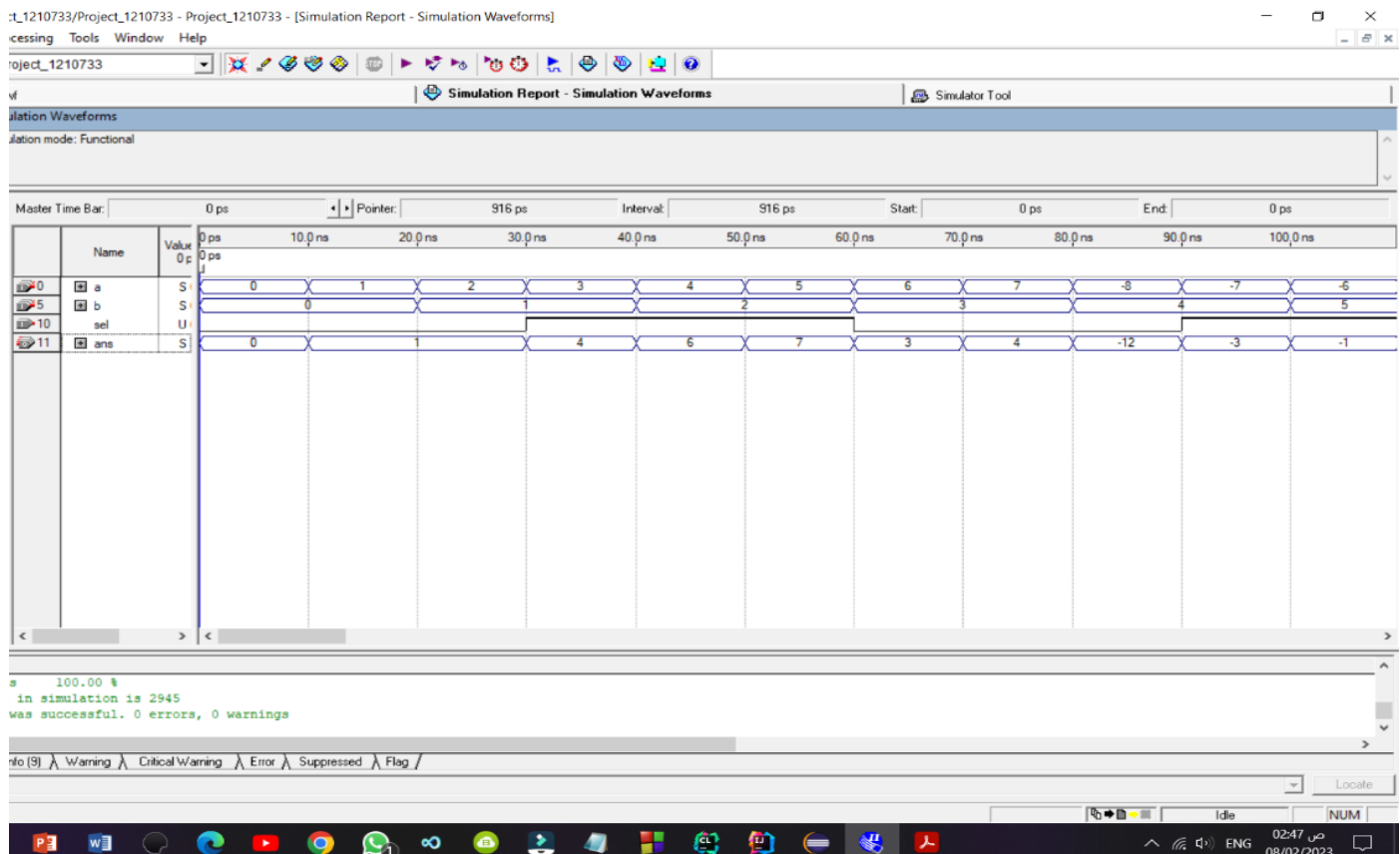
Adder - Subtractor

The adder- subtractor block I used is a block that make two things depending on the value of selection, this block is used for first 4 operations, the input size is n , and the output size is $n+1$ explained in page 5 .

Code :

```
1 module ADD_SUB_1210733#(parameter n=4) (a,b,sel,ans);
2   input signed [n-1:0]a,b ; // input size is n
3   input sel ; // 0 or 1
4   output reg signed [n:0] ans ; // output size is n+1
5   always@(a,b,sel)
6   begin
7       // if the selection is 0 , the block will work as subtracter
8       // and if the selection is 1 , the block will work as adder
9       case(sel)
10          1'b0 : ans = a-b ;
11          1'b1 : ans = a+b ;
12      endcase ;
13  end
14 endmodule
```

Simulation :



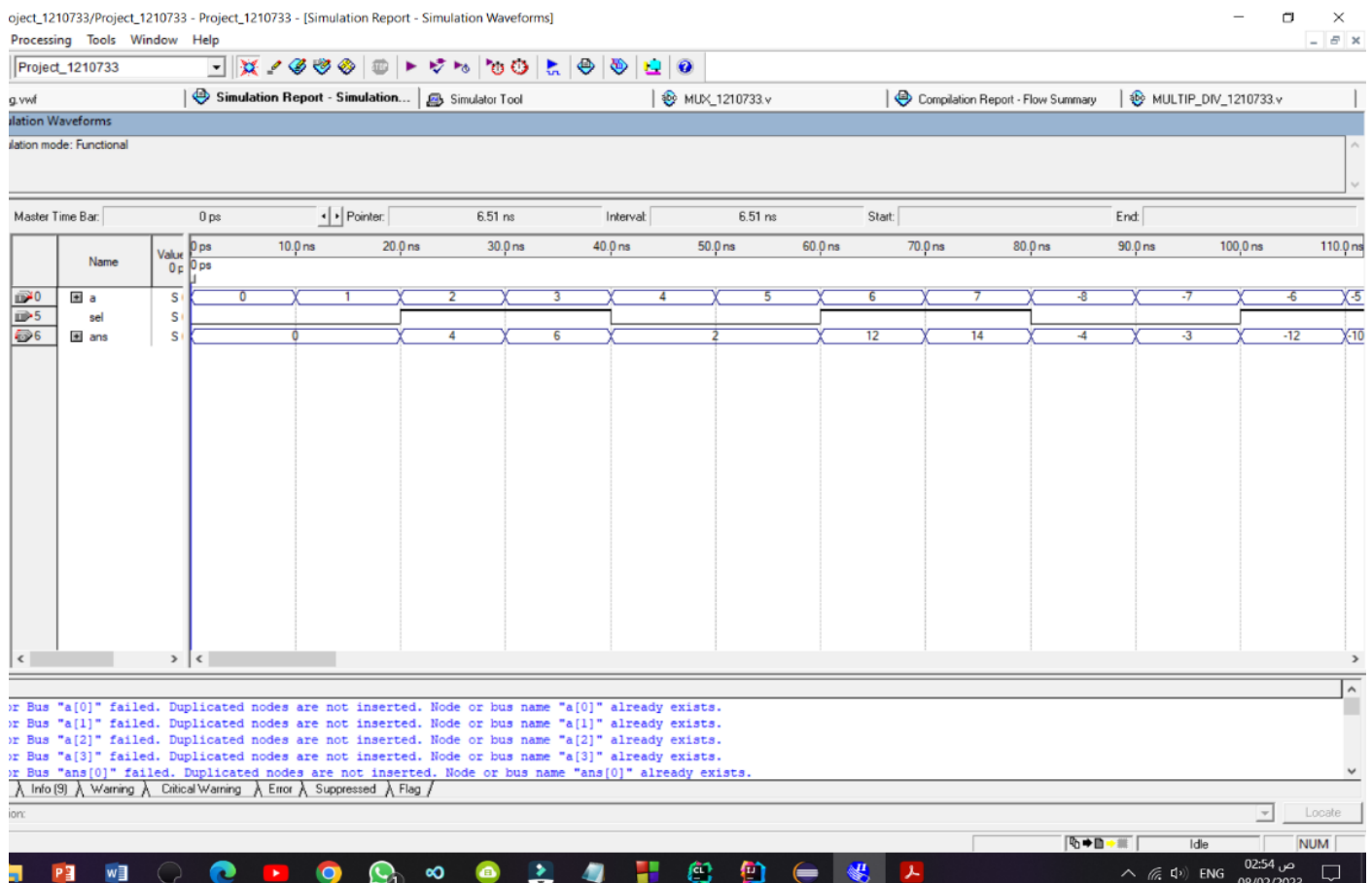
Multiplier - Divider

I used one block for the divide and multiply operation , the selection input decides to use the block either for multiplying or for dividing , this module multiply the number by two or divide the number by two.

Code :

```
1 module MULTIP_DIV_1210733#(parameter n=4) (a,sel,ans);
2   input signed [n-1:0] a ; // input size is n
3   input sel ; // 0 or 1
4   output reg signed[n+1:0] ans ; // output size is n+2
5   always@(a,sel)
6   begin
7     /* if the selection input is 0 , the block will
8        work as divider (right shift by one bit)
9        and if the selection input is 1 , the block will work as
10       multiplier (left shift by one bit)
11    */
12    case (sel)
13      1'b0 : ans = a/2 ;
14      1'b1 : ans = a*2 ;
15    endcase
16  end
17 endmodule
```

Simulation :



MUX

The mux I used is 8 to 1 mux , that take 8 inputs and give one output . The output depends on the selection value , the range for selection value is [0-7] .

The size of input in the mux is different depending on the operation , we have 5 input of size n , with operations 0,4,5,6,7 . and 2 input of size n+1 with operations 2 and 3, and one input of size n+2 with operation 2 .

Code :

```
1 module MUX_1210733#(parameter n=4)
2     (in0,in1,in2,in3 ,in4 , in5 ,in6 ,in7,SEL,res);
3 // the size of input to mux is different for each operation
4 input signed [n-1:0]in0 ,in4 , in5 ,in6 ,in7 ;
5 input signed [n:0] in2,in3 ;
6 input signed [n+1:0]in1;
7 input [2:0]SEL ;
8 // the output size is depending on the maximum size in input which is n+2
9 output reg signed [n+1:0] res ;
10 always@(*)
11 begin
12     case(SEL)
13         3'b000 : res <= in0;
14         3'b001 : res <= in1;
15         3'b010 : res <= in2;
16         3'b011 : res <= in3;
17         3'b100 : res <= in4;
18         3'b101 : res <= in5;
19         3'b110 : res <= in6;
20         3'b111 : res <= in7;
21     endcase ;
22 end
23 endmodule
```

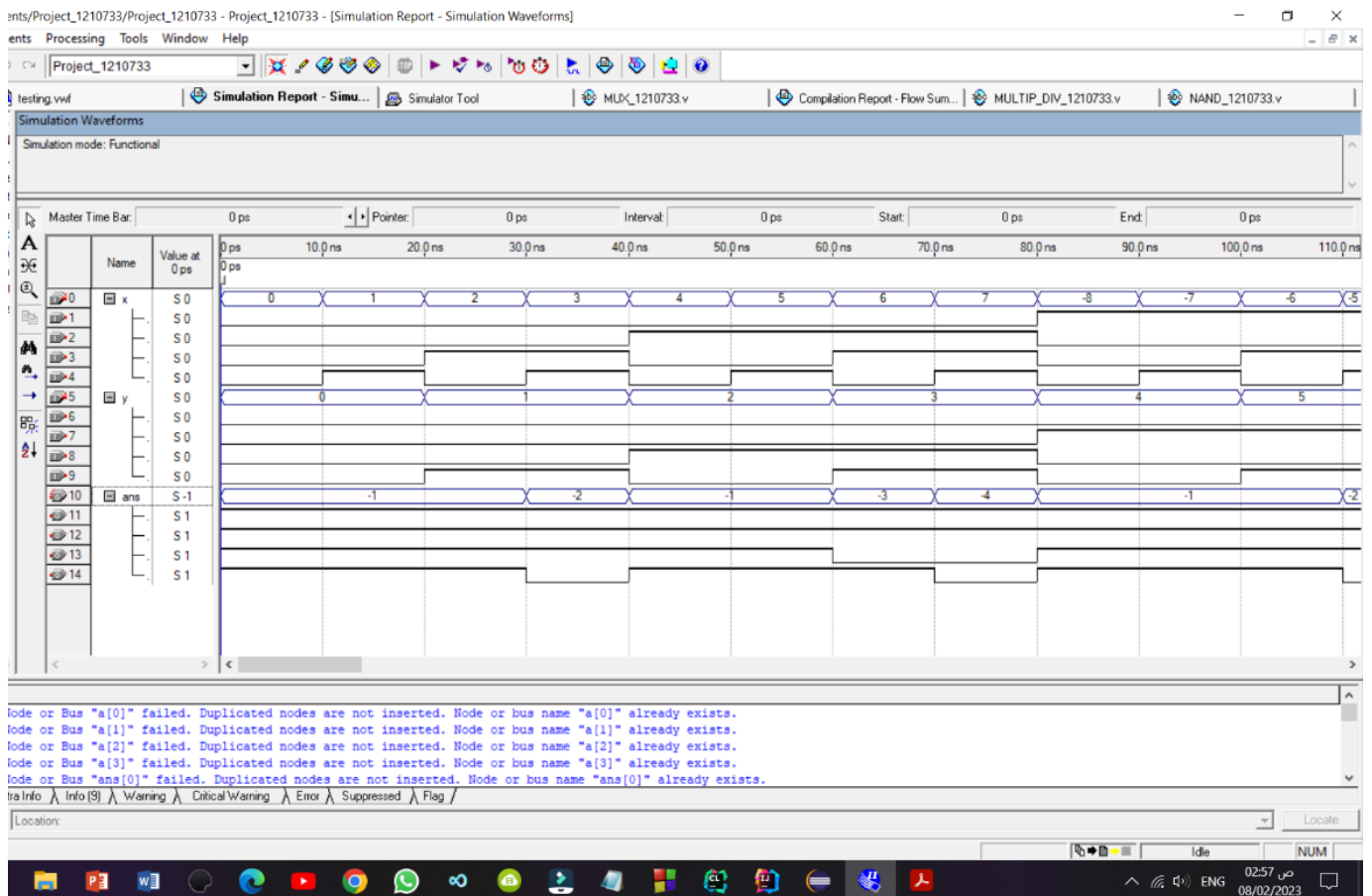
The simulation for this component is hard to represent , we have 8 input variables .

NAND

Code :

```
1 module NAND_1210733#(parameter n=4)(x,y,ans);
2   input signed [n-1:0] x,y ; // input size is n
3   output reg signed[n-1:0] ans ;// output size is n
4   always@(x,y)
5   =begin
6     ans = ~(x&y) ;
7   end
8 endmodule
```

Simulation :

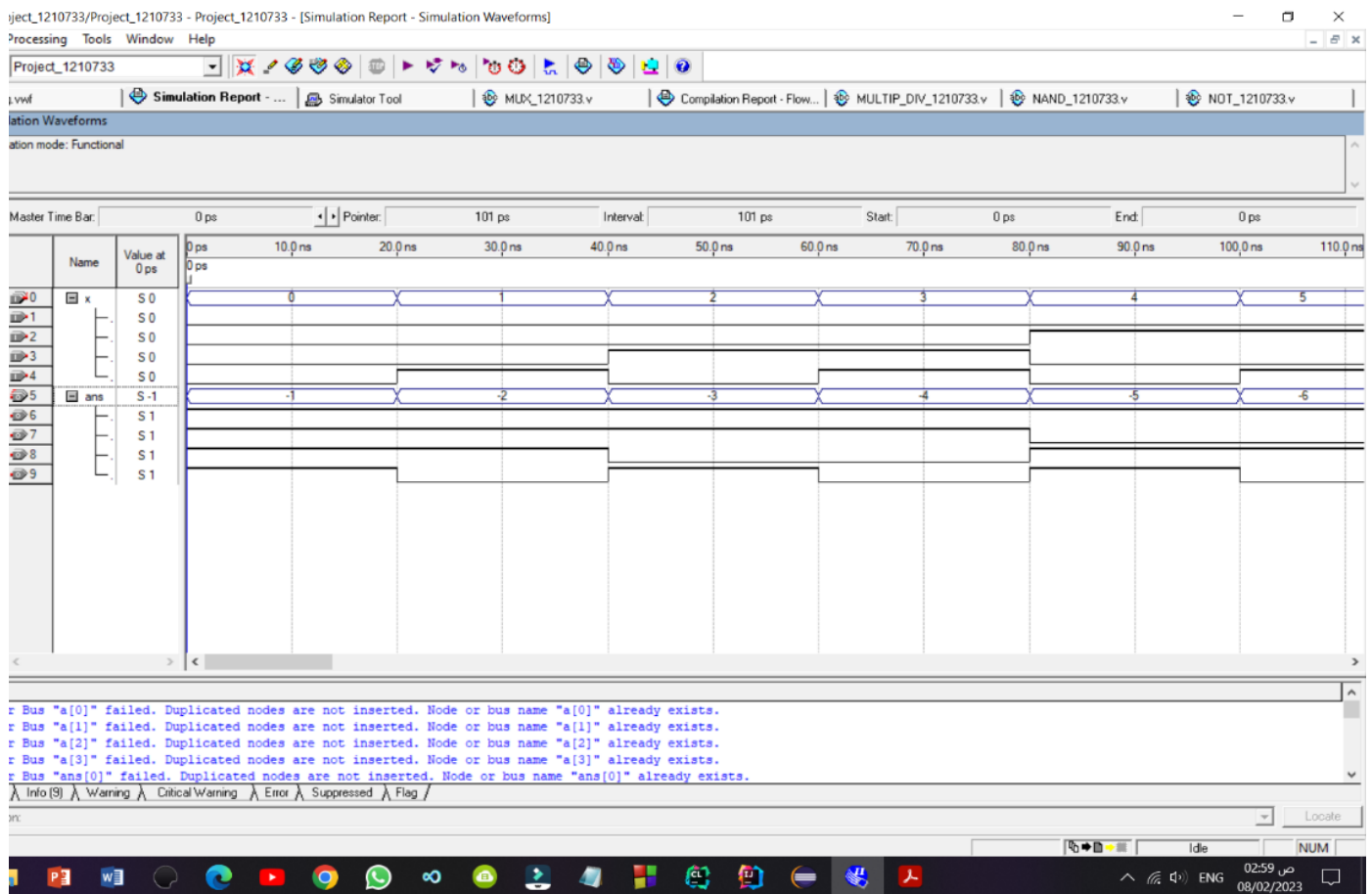


NOT

Code :

```
1 module NOT_1210733#(parameter n=4) (x,ans);
2   input signed [n-1:0] x ; // input size is n
3   output reg signed[n-1:0] ans ; // output size is n
4   always@(x)
5   begin
6     ans = ~ x ;
7   end
8 endmodule
```

Simulation :

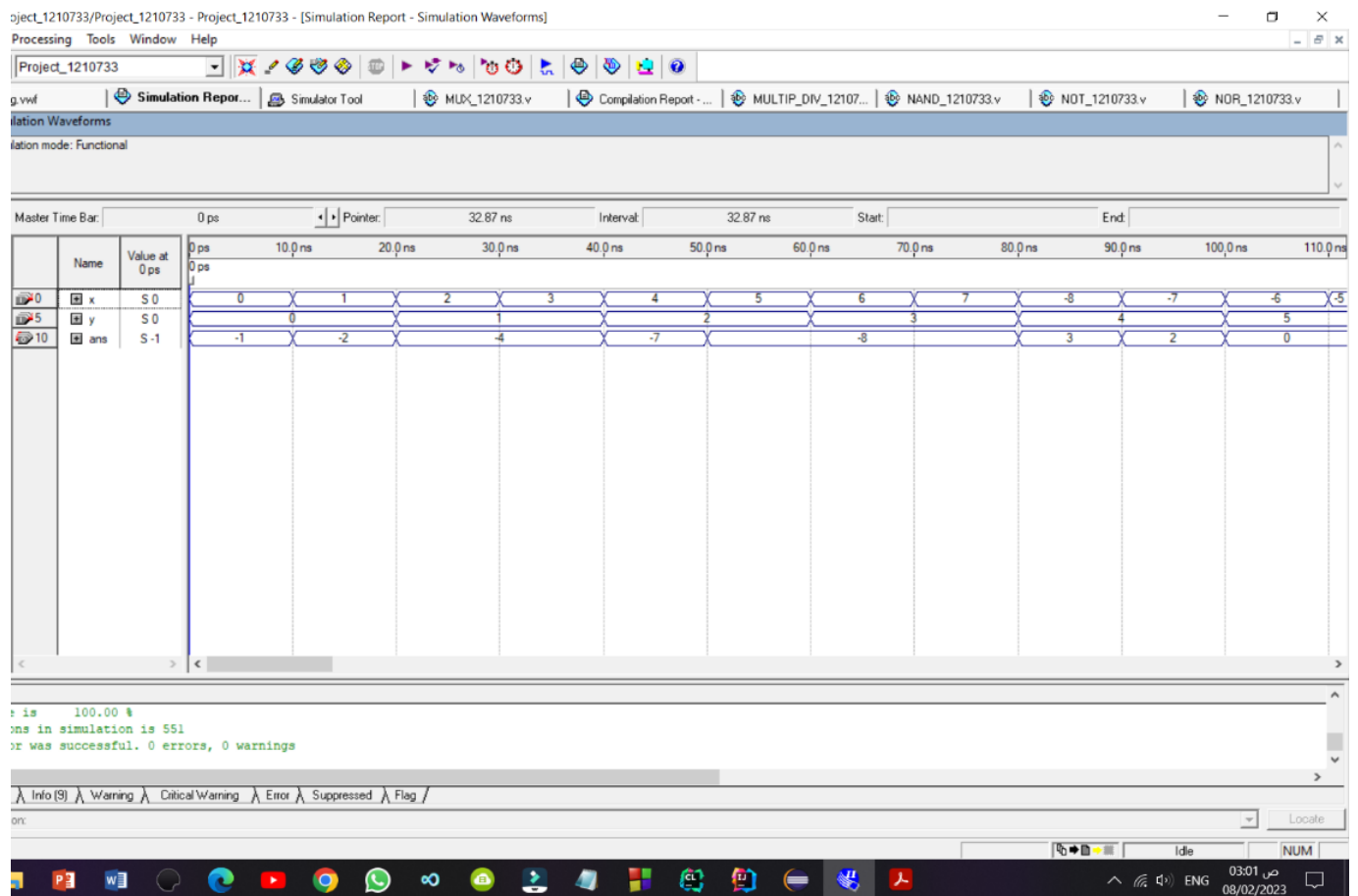


NOR

Code :

```
1 module NOR_1210733#(parameter n=4) (x,y,ans);
2   input signed [n-1:0] x,y ; // input size is n
3   output reg signed[n-1:0] ans ; // output size is n
4   always@(x,y)
5   begin
6     ans = ~(x|y) ;
7   end
8 endmodule
```

Simulation :

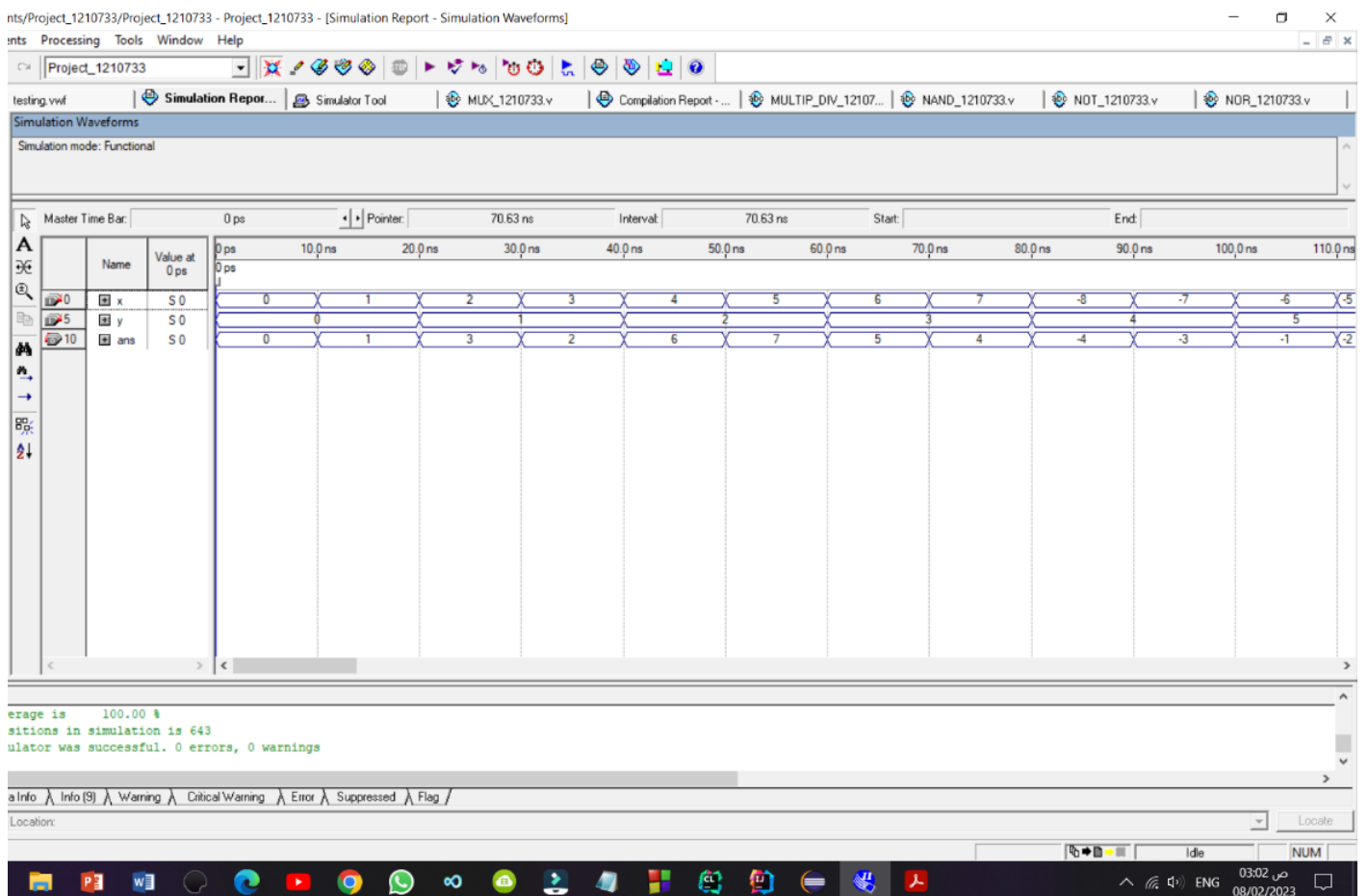


XOR

Code :

```
1 module XOR_1210733#(parameter n=4) (x,y,ans);
2   input signed [n-1:0] x,y ; // input size is n
3   output reg signed[n-1:0] ans ; // output size is n
4   always@(x,y)
5   begin
6     ans = x^y ;
7   end
8 endmodule
```

Simulation :



ALU implementation using Structural method

Code :

```
1 module ALU_structural_1210733 #(parameter n=4) (X , Y , SEL , OUT);
2   input signed [n-1:0] X ,Y ; // input size is n
3   input [2:0] SEL ;
4   output signed [n+1:0] OUT ; // output size is n+2
5
6   // wires to store the result of operations that needs n bits
7   wire [n-1:0] op0 ,op4 , op5 , op6 , op7 ;
8   // wires to store the result of operations that needs n+1 bits
9   wire [n:0] op2 , op3 ,temp1,temp2,temp3 ;
10  // wires to store the result of operations that needs n+2 bits
11  wire [n+1:0] op1 ;
12
13  // 1-- (x+y)/2
14  ADD_SUB_1210733 add1(X,Y,1,temp1); // store the value of x+y in temporarily wire called temp1
15  MULTIP_DIV_1210733 m1(temp1,0,op0);
16
17  // 2-- 2*(x+y)
18  MULTIP_DIV_1210733 m2(temp1,1,op1); // using the value stored in temp1 without add x,y again
19
20  // 3-- (x/2)+y
21  MULTIP_DIV_1210733 m3(X,0,temp2); // store the value of x/2 in temp2 wire
22  ADD_SUB_1210733 add2(temp2,Y,1,op2); // adding temp2 to y
23
24  // 4-- x-(y/2)
25  MULTIP_DIV_1210733 m4(Y,0,temp3); // store the value of y/2 in temp3 wire
26  ADD_SUB_1210733 add3(X,temp3,0,op3); // subtracting temp3 from x "> order is important <"
27
28  // 5- NAND
29  NAND_1210733 n1(X,Y,op4);
30
31  // 5- NOT
32  NOT_1210733 n2(X,op5);
33
34  // 5- NOR
35  NOR_1210733 n3(X,Y,op6);
36
37  // 5- XOR
38  XOR_1210733 n4(X,Y,op7);
39
40  // invoke all values calculated ,and selection value to the mux with respect to order
41
42  MUX_1210733 result(op0 , op1 , op2 , op3 , op4 , op5 , op6 , op7 , SEL , OUT);
43 endmodule
```

The idea of My solution is to store the operation on x,y in wires , and the wires also have different size , depending on the size of output needed for the operation that will be stored in it .

And to increase reusability, at the first two operation, I initialized a temp wire that stores the value of x+y , and I used it twice instead of call adder-subtracter block again

At the end, I invoked all values in the mux 8to1 with the selection, so that the output is presented depending of the selection value.

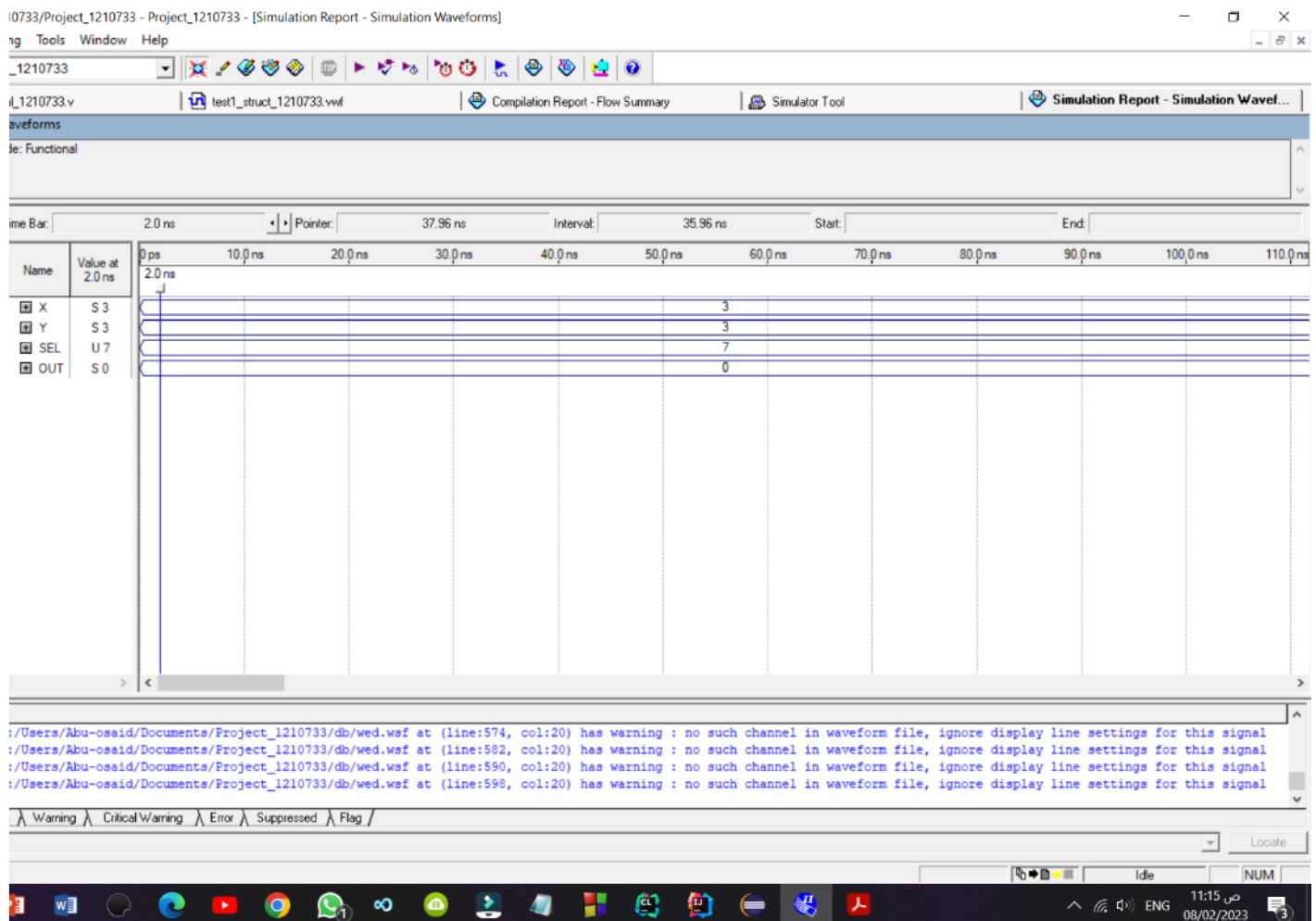
The next simulations are selected depending on the ID number as follows :

1	C2	Y2	X2	C1	Y1	X1
1	2	1	0	7	3	3

Simulation 1 :

X = 3 || Y = 3 || C = 7

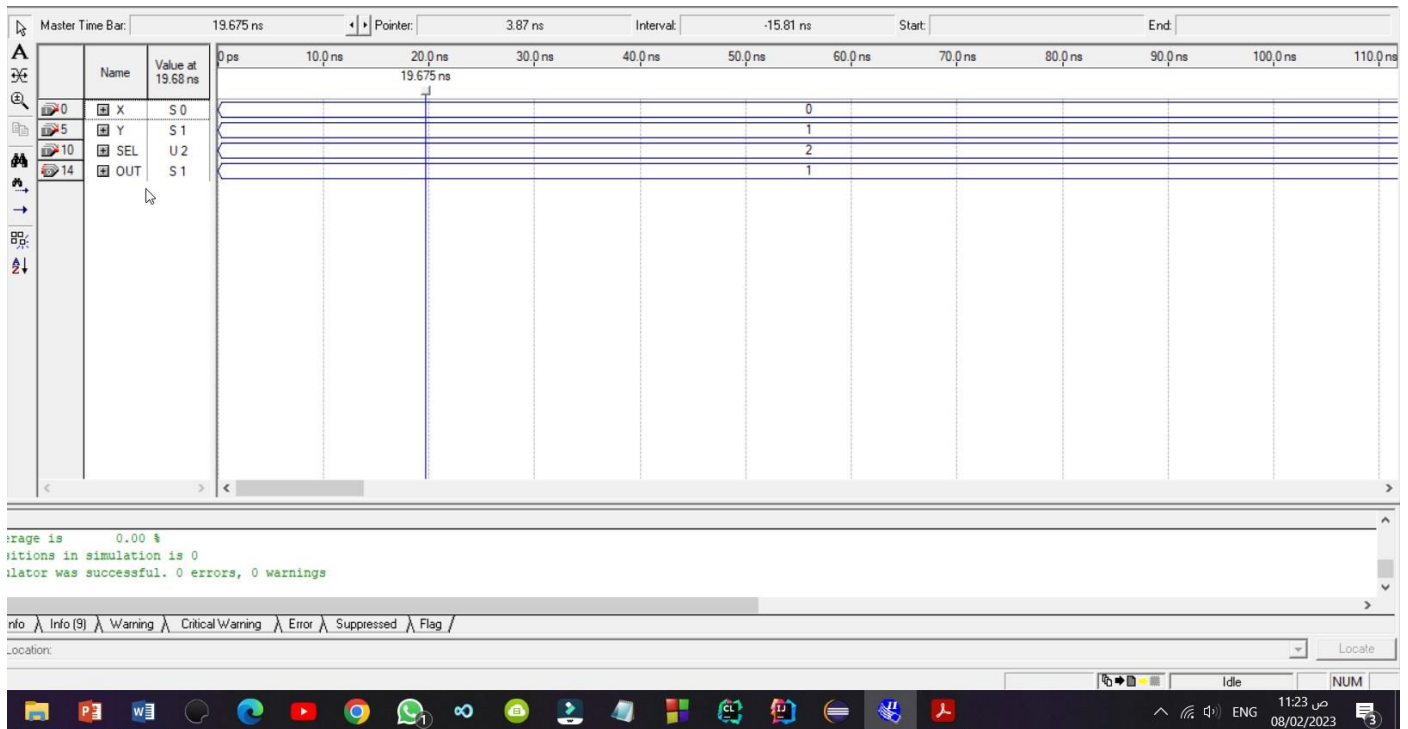
Operation 7 → 0011 xor 0011 = 0



Simulation 2 :

$X = 0 \quad || \quad Y = 1 \quad || \quad C = 2$

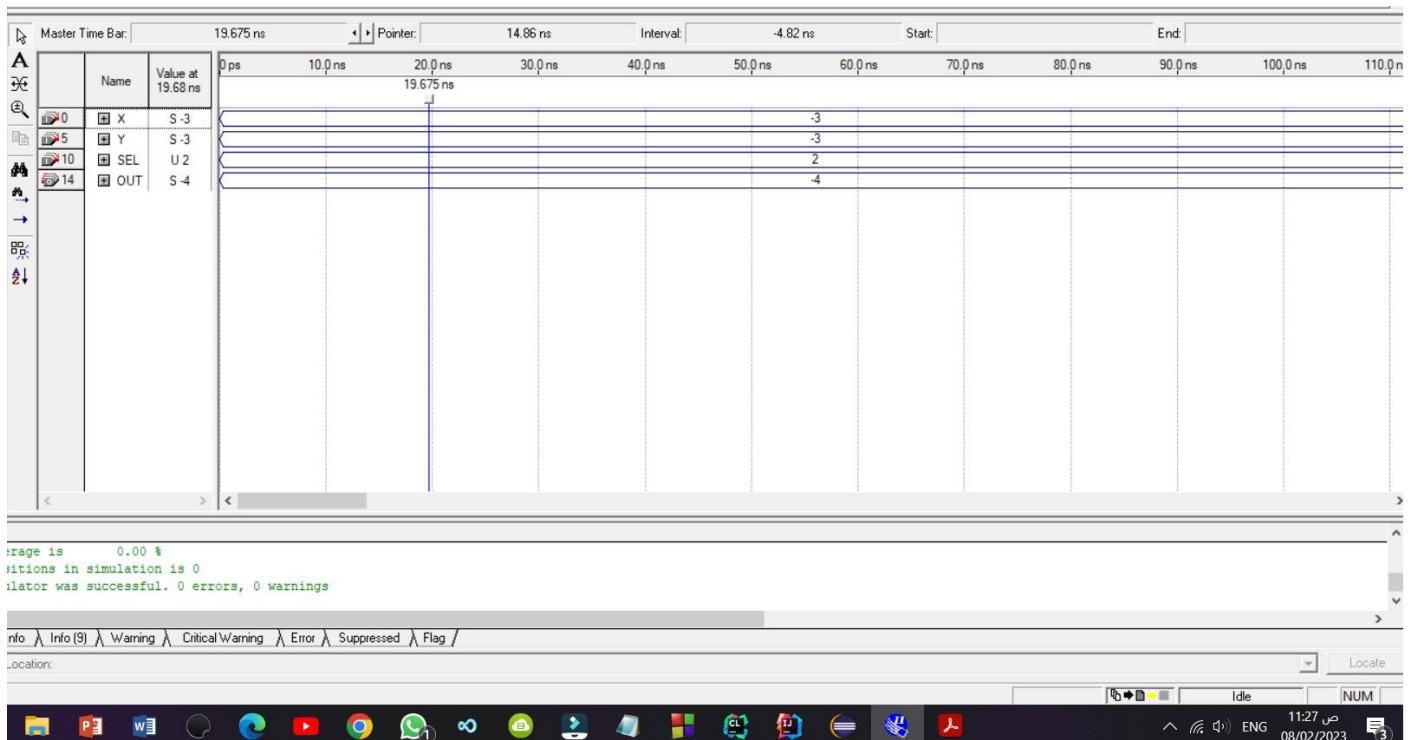
Operation 2 $\rightarrow (x/2)+y == (0/2)+1 = 1$



Simulation 3 :

$X = -3 \quad || \quad Y = -3 \quad || \quad C = 2$

Operation 2 $\rightarrow (x/2)+y == (-3/2)+-3 = -1 -3 = -4$



ALU implementation using behavioural method

This implementation actually, is just to use the usual symboles we know like + , - , ^ , & , |

Code :

```
1 module ALU_behavioral_1210733 #(parameter n=4) (X , Y , SEL , OUT) ;
2   input signed [n-1:0] X ,Y ; // input size is n
3   input [2:0] SEL ;
4   output reg signed [n+1:0] OUT ; // output size is n+2
5   // the output in this module depends on the selection value to do the
6   // desired operation using usual symboles without invoking any other module
7   always@(X,Y,SEL)
8   begin
9       case (SEL)
10          3'b000 : OUT<= (X+Y)/2;
11          3'b001 : OUT<= 2*(X+Y);
12          3'b010 : OUT<= (X/2)+Y;
13          3'b011 : OUT<= X-(Y/2);
14          3'b100 : OUT<= ~(X&Y);
15          3'b101 : OUT<= ~X ;
16          3'b110 : OUT<= ~(X|Y);
17          3'b111 : OUT<= X^Y;
18      endcase ;
19   end
20 endmodule
```

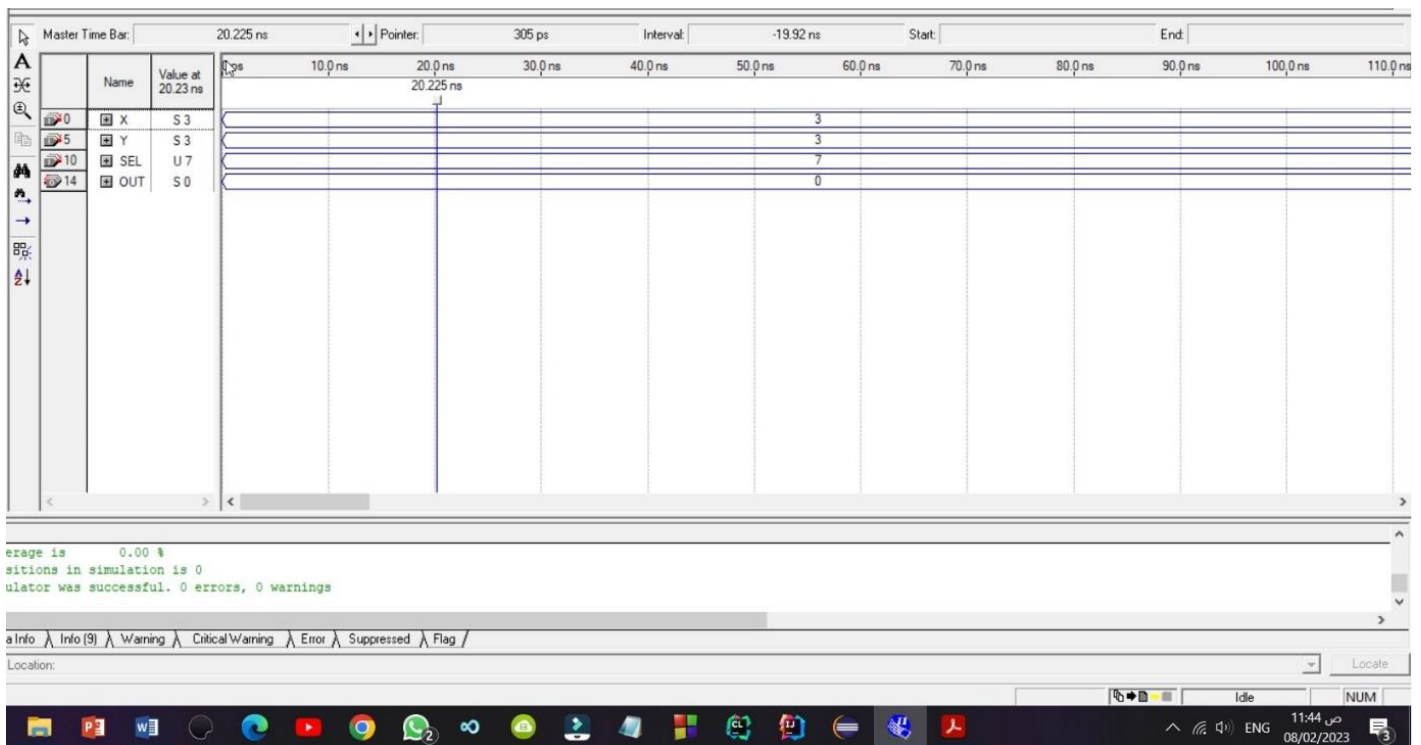
The next simulations are as same as structural , selected depending on the ID number as follows :

1	C2	Y2	X2	C1	Y1	X1
1	2	1	0	7	3	3

Simulation 1 :

$X = 3 \quad || \quad Y = 3 \quad || \quad C = 7$

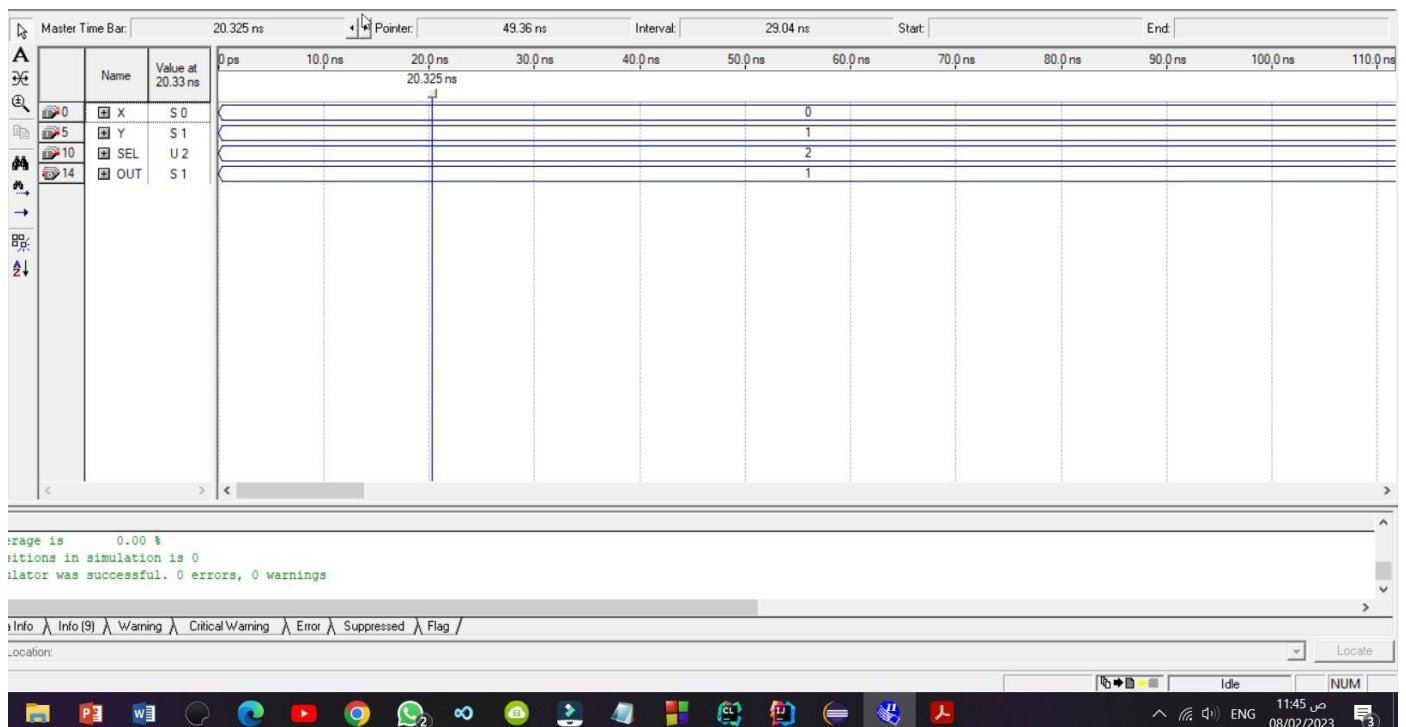
Operation 7 $\rightarrow 0011 \text{ xor } 0011 = 0$



Simulation 2 :

$X = 0 \quad || \quad Y = 1 \quad || \quad C = 2$

Operation 2 $\rightarrow (x/2)+y == (0/2)+1 = 1$



Simulation 3 :

$X = -3$ || $Y = -3$ || $C = 2$

Operation 2 $\rightarrow (x/2)+y == (-3/2)+-3 = -1 -3 = -4$

