Intro To Processor Architecture

Building Alu Using Verilog

Atharv Sujlegaonkar (2020102025)

Mitul Garg (2020102026)

ALU :

ALU is the combination of ADD, SUB, AND, and XOR operations. Depending on which operation the user wants, two 64 bit numbers are taken as input and after the operation is performed, its output is given.

The inputs for the operations are 0-ADD, 1-SUB, 2-AND, 3-XOR. After these inputs, we get two 32 bit numbers-A, B- inputs. The final answer is given as Y.

In this example we are giving two 32 bit numbers- A=-5, B=3- as inputs to the ALU. After every 50 nanoseconds, we have changed the operation, from 0 to 3. Its corresponding output is noticed at Y. The output values of A, B, and C are in signed decimals.

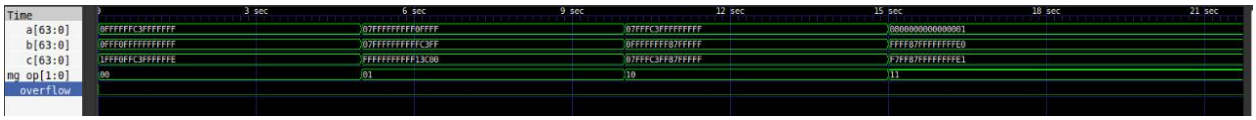
When C=0, the two numbers are added, giving Y = -2.

When C=1, the two numbers are subtracted, giving Y = -8.

When C=2, the AND of two numbers are performed giving, Y = 00000000000000000000000000000011 = 3.

When C=3, the XOR of two numbers are performed giving, Y = 11111111111111111111111111111000 = -8.

Following is the gtk wave output :



And following are some examples displaying the working of alu :

```
VCD info: dumpfile ALU_gtk.vcd opened for output.
Control = 0 --> ADD
Control = 1 --> SUB
Control = 2 --> AND
Control = 3 --> XOR

Control = 0
 a  =  1152921488500719615
 b  =  1152657621816180735
overflow = 0
 c  =  2305579110316900350

Control = 1
 a  =   576460752302440447
 b  =   576460752303408127
overflow = 0
 c  =              -967680

Control = 2
 a  =   576456629134819327
 b  =  1152921504481017855
overflow = 0
 c  =   576456629008990207

Control = 3
 a  =   576460752303423489
 b  =     -131941395333152
overflow = 0
 c  =  -576592693698756639
```

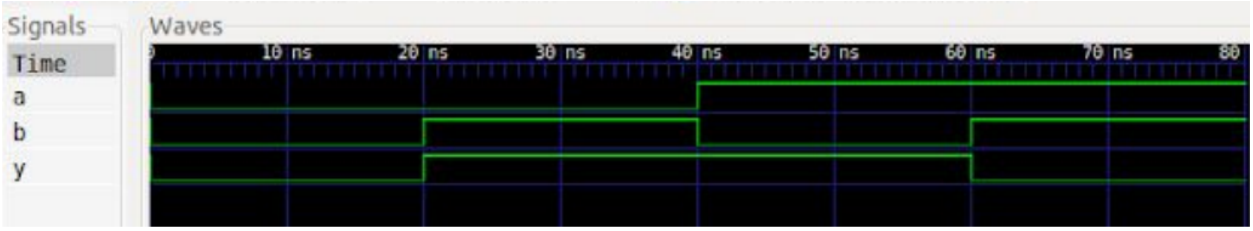
Each of these operations is seen in detail below.

XOR Operation :

To perform the operation of XOR, we have to know its truth table.

INPUT		OUTPUT
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

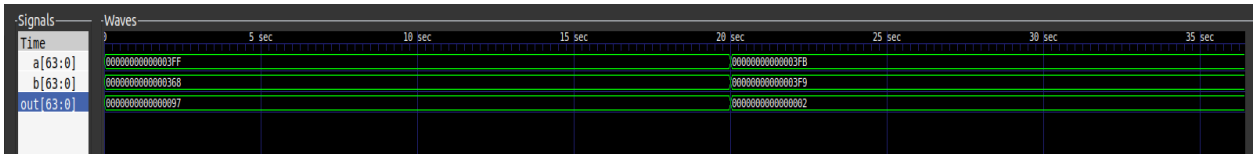
We apply the above inputs to see its GTKWAVE output.



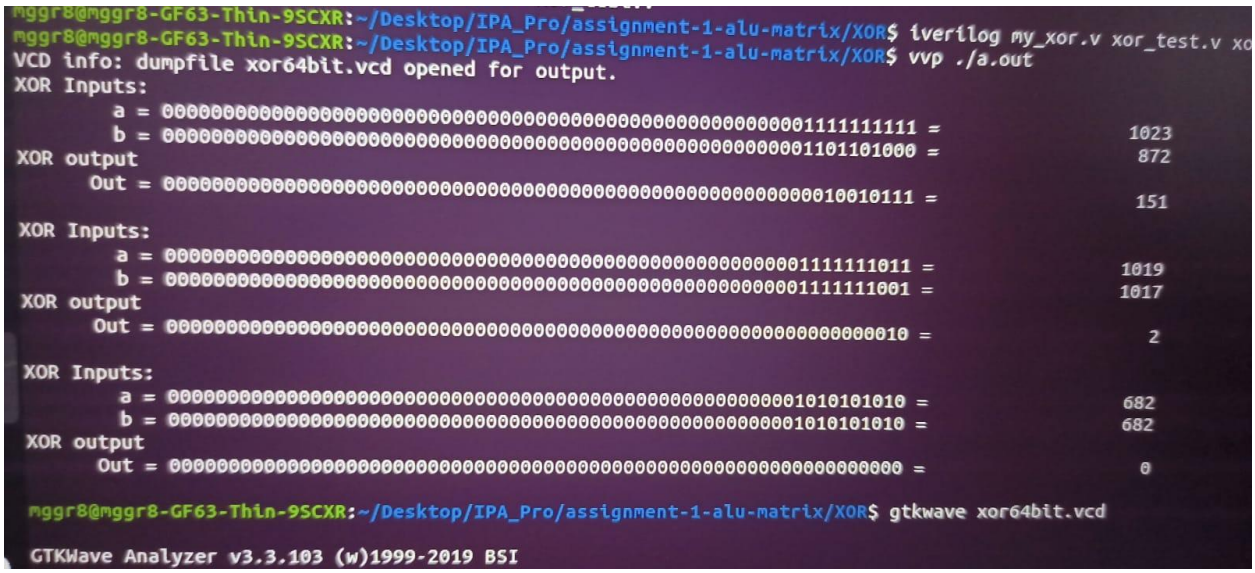
To compute the XOR function of two 64 bit numbers we have we can simply perform bitwise XOR operations. We have to define a module that does the XOR of 2 bits as in the truth table by simple if-else statements.

When we store 64-bit numbers in a bus, we can take each corresponding element (bit) of the bus and pass it to the module. By calling the modules for each bit, the process is complete. In this example, we have taken the inputs in binary format so that we can easily check bitwise XOR.

In this example we have taken the inputs in hexadecimal format :



Some examples are displayed in following image verifying the modul's function :

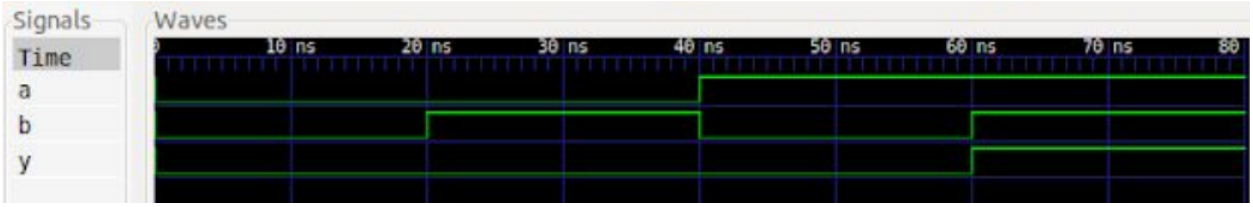


AND Operation :

To perform the operation of AND, we have to know its truth table.

INPUT		OUTPUT
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

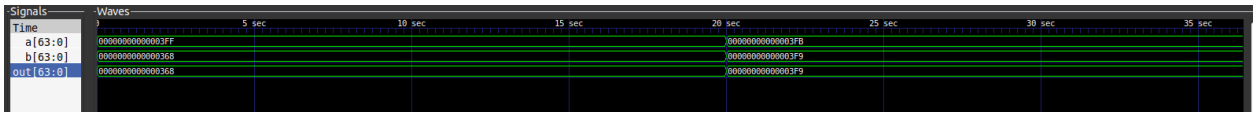
We apply the above inputs to see its GTKWAVE output:



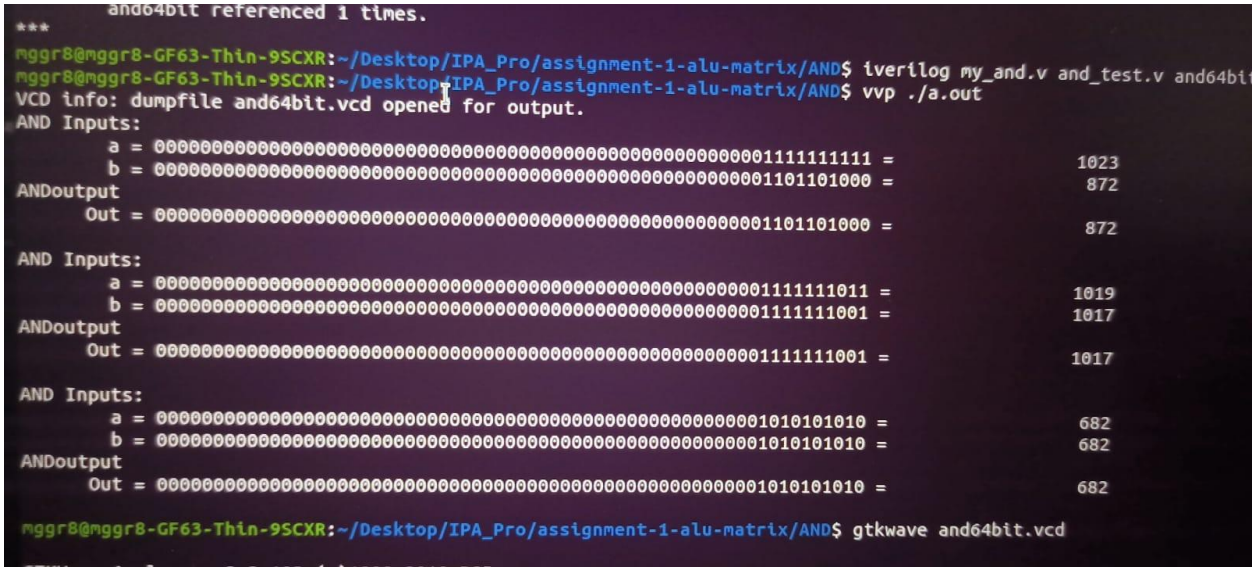
To compute the AND function of two 64 bit numbers we have we can simply perform bitwise AND operations. We have to define a module that does the AND of 2 bits as in the truth table by simple if-else statements.

When we store 64-bit numbers in a bus, we can take each corresponding element (bit) of the bus and pass it to the module. By calling the modules for each bit, the process is complete. In this example, we have taken the inputs in binary format so that we can easily check bitwise AND.

In this example we have taken inputs in hexadecimal formats :



Some examples are displayed in following image verifying the modul's function :

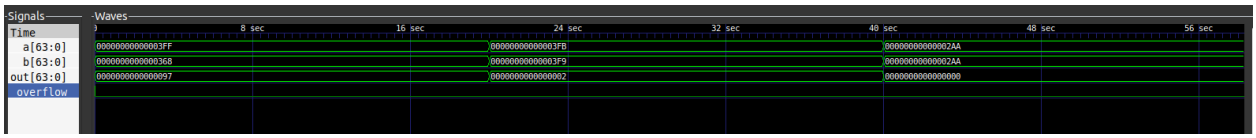


ADD OPERATION :

To perform the addition of two 32 bit numbers we have to understand the truth table of a full adder. When we add any 2 numbers, we have its 'sum'. But if there is an overflow, i.e., there is an extra digit, it's called a 'carry over'. In a similar manner, when we add 2 numbers of one bit, there is a possibility of an overflow/carry. This carry is then passed over for addition with the upcoming bits.

INPUT			OUTPUT	
C(in)	A	B	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This looks as if it is a 3-bit addition. Here, A and B are the actual input bits (the values at the indices of 64-bit numbers). The C(in), is the carry-over that has taken place in the previous ADD operation. To follow this operation, we first call a module, that simply adds the least significant bits of the two 32 bit numbers and returns their sum and carry. Then we call another module, which takes the carry of the previous operation (i.e., C(in)), and the corresponding bits of the 64- bit numbers (as A and B). The sum and carry values are assigned by simple if-else-if ladder statements. In the upcoming sample outputs, the inputs are in hexadecimal so as to show the working with negative numbers.

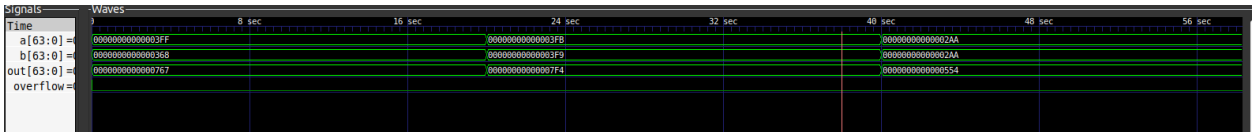


Some examples are displayed in the following image verifying the module's function :

SUB OPERATION :

To perform subtraction of two numbers, we have to know about the 2’s complement. The 2’s complement of a number is determined by inverting the bits (changing 0 to 1 and vice versa) of the number and then ADD 1 to it. Of the two 64 bit numbers- A and B- we have to determine the 2’s complement of B.

To perform the 2’s complement we have inverted all the bits of the given number and called the ADD module for the complement of B and 1. Then we simply call the ADD module of A and B's. In the upcoming sample outputs, the inputs are in hexadecimal so as to show the working with negative numbers.



Some examples are displayed in the following image verifying the modul's function :