INTRODUCTION TO PROCESSOR ARCHITECTURE

ASSIGNMENT 1

ALU

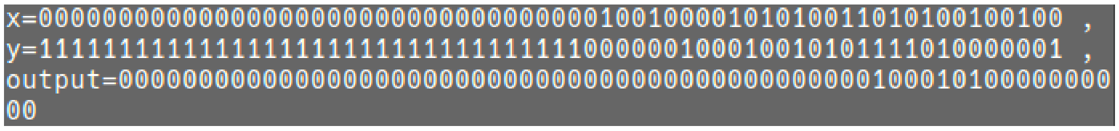
Nitin Shrinivas
2020102046
Merugu Nanditha
2020102061

AND operation

We have 2 64-bit signed binary inputs namely x and y and we have to perform the AND operation between these 2 inputs.

Our approach was to take the AND operation for each bit and assign each bit of the output to the OUT vector wherein it is of the array of binary numbers of the result from AND operation.

To check the module is correct, we have used a testbench wherein the inputs x and y are random 64-bit binary numbers, and the output is displayed on the monitor.



```
x=0000000000000000000000000000000000010010000101010011010100100100 ,
y=1111111111111111111111111111111111000000100010010101111010000001 ,
output=00000000000000000000000000000000000000000000000100010100000000
00
```

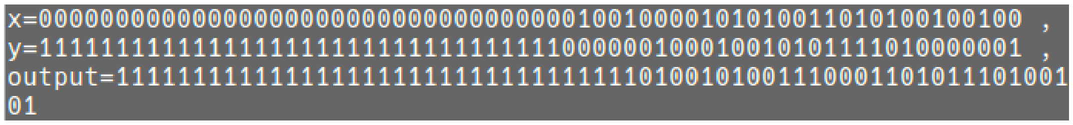
The output can be explained as, the positions of the 64-bit number input’s x and y are both one then the output is one else in that position the output is zero.

XOR operation

We have 2 64-bit signed binary inputs namely x and y and we have to perform the AND operation between these 2 inputs.

Our approach was to take the AND operation for each bit and assign each bit of the output to the OUT vector wherein it is of the array of binary numbers of the result from AND operation.

To check the module is correct, we have used a testbench wherein the inputs x and y are random 64-bit binary numbers, and the output is displayed on the monitor.



```
x=0000000000000000000000000000000000010010000101010011010100100100 ,
y=1111111111111111111111111111111111000000100010010101111010000001 ,
output=11111111111111111111111111111111110100101001110001101011101001
01
```

The output can be explained as, the positions of the 64-bit number input’s x and y are both same then the output is zero else in that position the output is one which is true from the truth table of XOR.

ADD operation

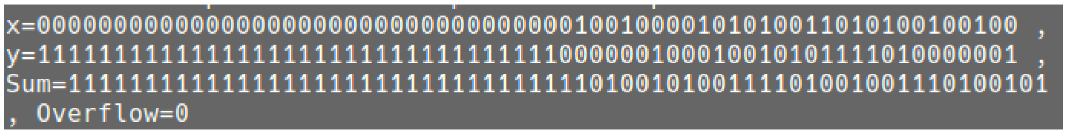
We have 2 64-bit signed binary inputs namely x and y and we have to perform the ADD operation between these 2 inputs.

Our approach was to find the sums and the carry of the signed numbers by considering that the initial carry as 0 and then finding the sum as
SUM(i) = x(i) (XOR) y(i) (XOR) CARRY(i)
CARRY(i+1) = x(i).y(i) + CARRY(i).y(i) + CARRY(i).x(i)

The sum and carry bit are found as follows and the overflow is the condition wherein the 64 is not enough for the sum to be displayed and hence shows some other output and we need to detect it.

The overflow can be detected if the xor bit of the 64th and the 65th bits gives an output of 1 then there is a carry and overflow must be reported else there will not be any overflow.

To check the module is correct, we have used a testbench wherein the inputs x and y are random 64-bit binary numbers, and the output is displayed on the monitor.



```
x=0000000000000000000000000000000000010010000101010011010100100100 ,
y=1111111111111111111111111111111111000000100010010101111010000001 ,
Sum=1111111111111111111111111111111111010010100111101001001110100101
, Overflow=0
```

The output can be explained as the 64-bit number from LSD to MSD input’s x and y we must find the carry and then add it to the next bit’s addition. Hence we will have the 64 bit sum and 65 bit carry.

SUB operation

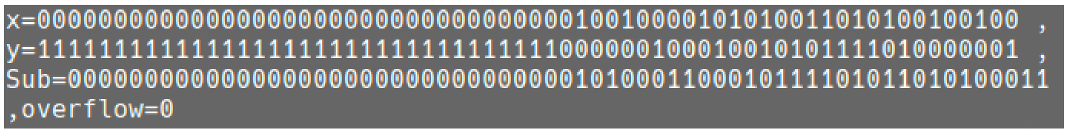
We have 2 64-bit signed binary inputs namely x and y and we have to perform the SUB operation between these 2 inputs.

Our approach was to convert the subtrahend into the 2’s complement that is first we need to convert it into its 1’s complement by using the not operation and then after that we will add 1 which is already have the module to find the addition of 2 binary numbers.
Now we use addition and add the 2 numbers and we get the result.
If the number has no extra carry bit then the number is negative and then we need to find the 2’s complement and attach a negative symbol but as the number is already signed and we need to represent in the signed number form and thus the result is kept in the same way and not altered.

The overflow is the condition wherein the 64 is not enough for the difference to be displayed and hence shows some other output and we need to detect it.

The overflow can be detected if the xor bit of the 64th and the 65th bits gives an output of 1 then there is a carry and overflow must be reported else there will not be any overflow.

To check the module is correct, we have used a testbench wherein the inputs x and y are random 64-bit binary numbers, and the output is displayed on the monitor.



```
x=0000000000000000000000000000000000010010000101010011010100100100 ,
y=1111111111111111111111111111111111000000100010010101111010000001 ,
Sub=0000000000000000000000000000000001010001100010111101011010100011
,overflow=0
```

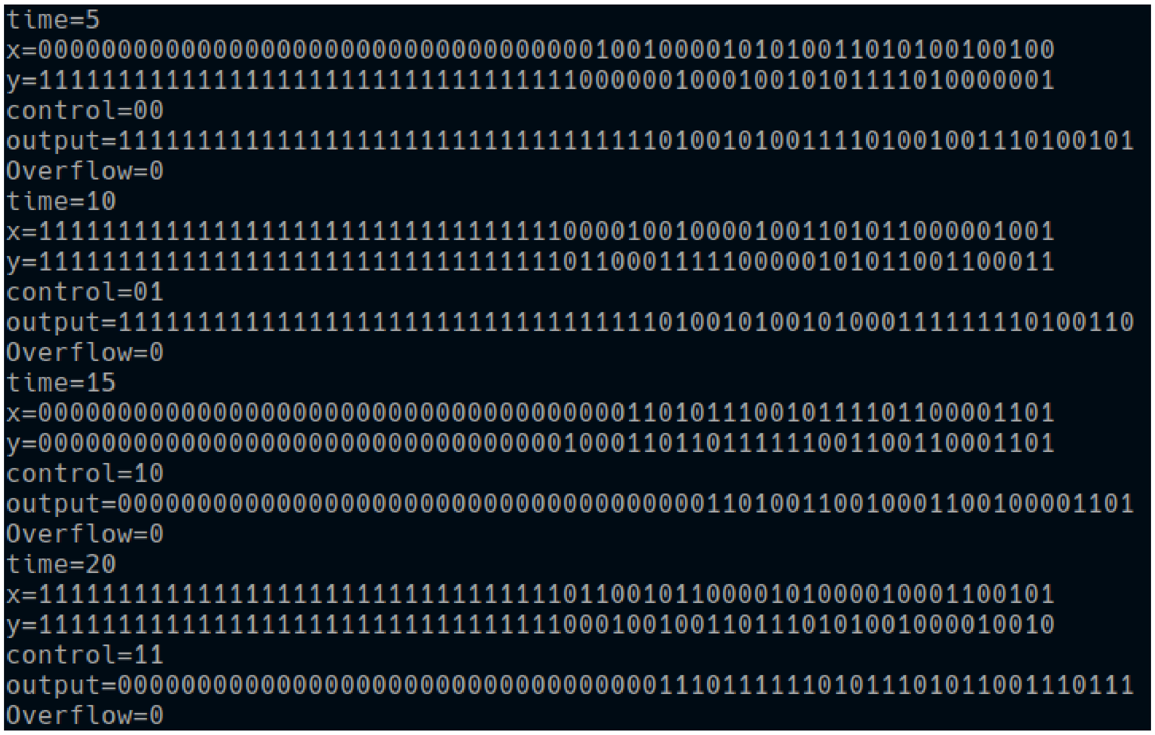
The output can be explained as the 64-bit number from LSD to MSD input’s x and y we must find the carry and then add it to the next bit’s addition. Hence we will have the 64 bit sum and 65 bit carry.

ALU file

This is the file alu.v where in it is the control file of the operation wherein the control will be with the user to which function or file he will call that is add, and, xor or subtract.

We have used the case functionality of the verilog wherein we used the condition statement from the input control and then after the function undergoes the required functionality the output is stored in OF.

Now to test the alu.v module we have created the test bench where in the 64 bit binary inputs x,y are mentioned and the control input is also provided.

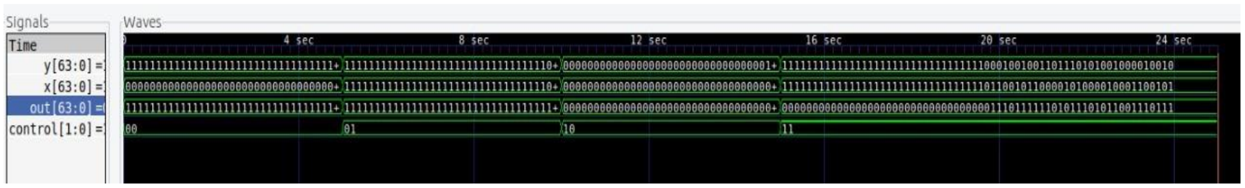


```
time=5
x=0000000000000000000000000000000001001000010101001101010100100100
y=1111111111111111111111111111111110000000100010010101111010000001
control=00
output=1111111111111111111111111111111111010010100111101001001110100101
Overflow=0
time=10
x=1111111111111111111111111111111110000100100001001101011000001001
y=1111111111111111111111111111111110110001111100000101011001100011
control=01
output=1111111111111111111111111111111111010010100101000111111110100110
Overflow=0
time=15
x=0000000000000000000000000000000000000110101110010111101100001101
y=0000000000000000000000000000000001000110110111111001100110001101
control=10
output=0000000000000000000000000000000000000110100110010001100100001101
Overflow=0
time=20
x=1111111111111111111111111111111110110010110000101000010001100101
y=1111111111111111111111111111111110001001001101110101001000010010
control=11
output=0000000000000000000000000000000000111011111101011101011001110111
Overflow=0
```

The control 0 is the add functionality
The control 1 is the Sub functionality
The control 2 is the and functionality
The control 3 is the xor functionality

As we can check the outputs from the previous cases separately and compare here the alu module is correct.

Waveform for the ALU.v



This is the waveform of the ALU wherein the 00 represents the waveform for the add function and 01 for sub, 10 for AND and finally we will have xor functionality for 11.
For the inputs we have the output waveform.