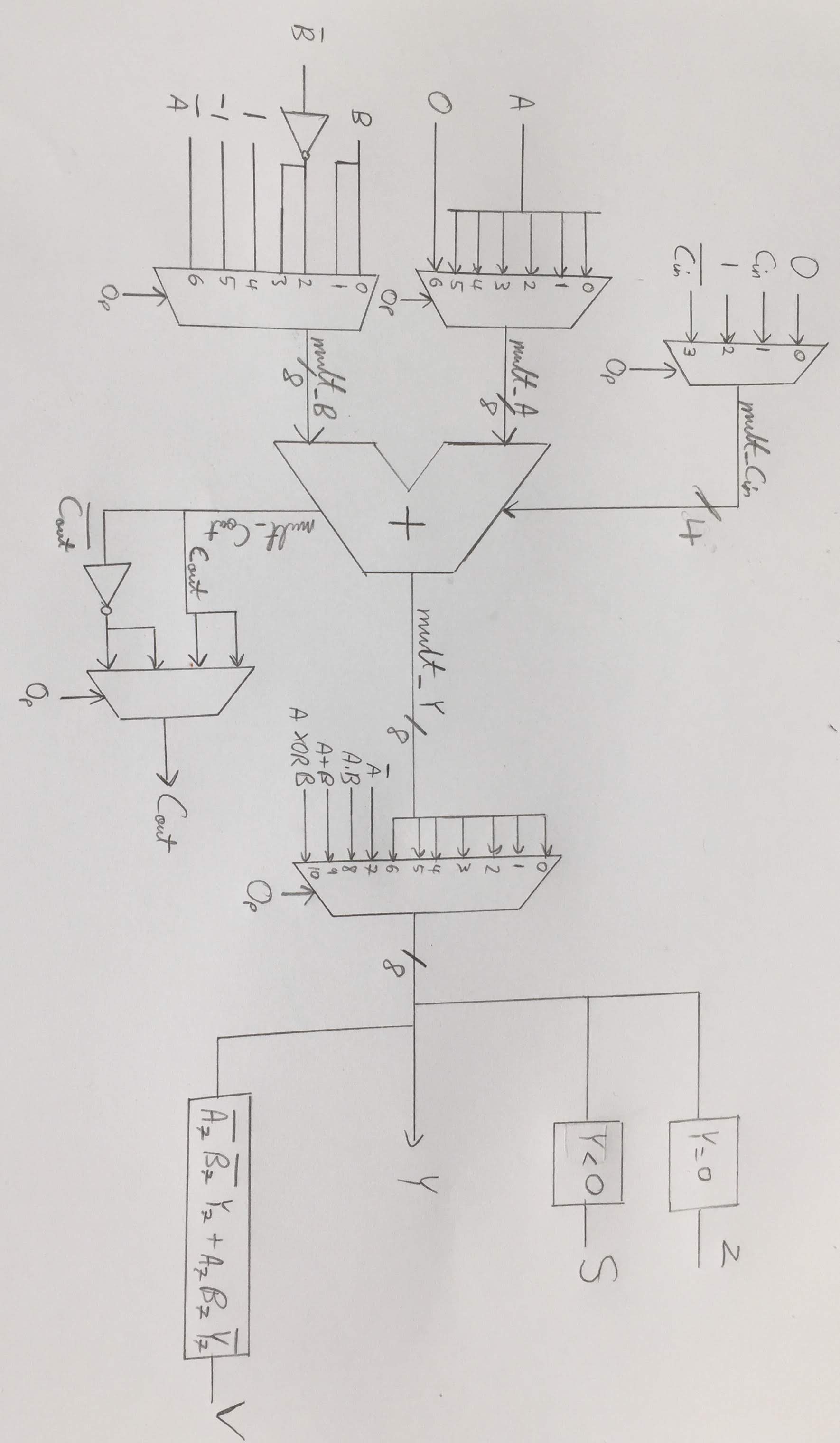
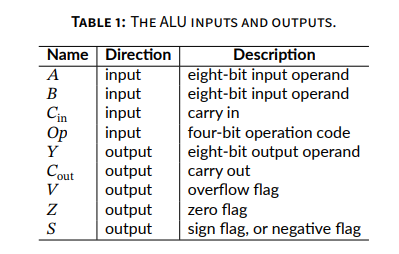
**Computer Logic – Practical 4**

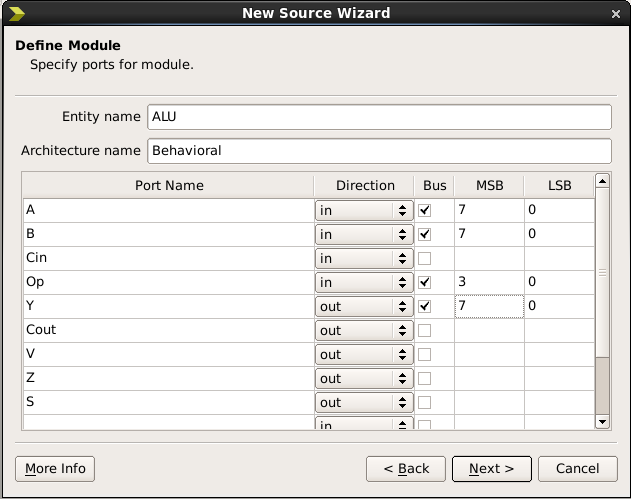
**Objective:**

To design and implement a simple ALU.

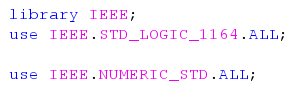
**Tasks:**

1. A sketch of the ALU implementation was drawn as can be seen:
2. A new project was created and a new VHDL module entitled ‘ALU’ was also initialized exactly after based on the table below. From the table it is clear that A and B are 2 8-bit input operands, Cin and Op3:0 are the control inputs, Y is the 8-bit output operand and Cout, V, Z and S are the control outputs.

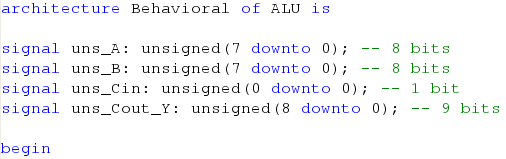




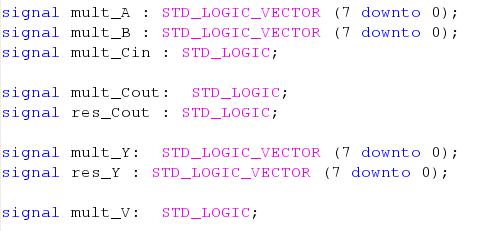
1. The following steps were followed for the implementation of the sketched ALU:
2. The package line *use IEEE.NUMERIC\_STD.ALL* was entered at the top of the file under the other IEEE declaration.



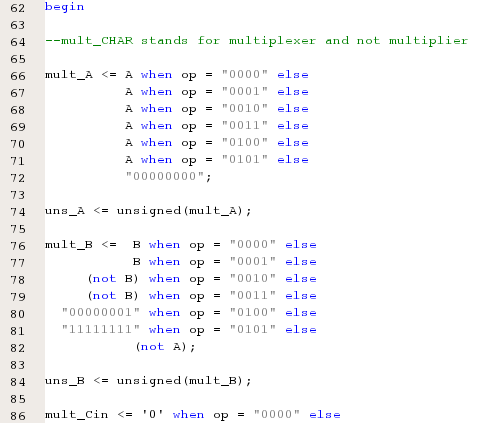
1. After entering the package line, signals of type unsigned were entered for the values A, B, Cin, Y and Cout in the architecture declaration part. To make the operation simpler while making use of less code, Y (8-bit output) and Cout (1-bit output) were combined into a single nine-bit signal. The declarations made can be seen below.

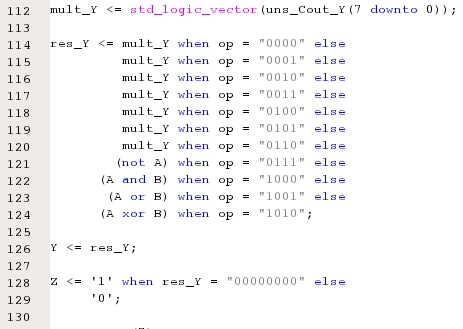
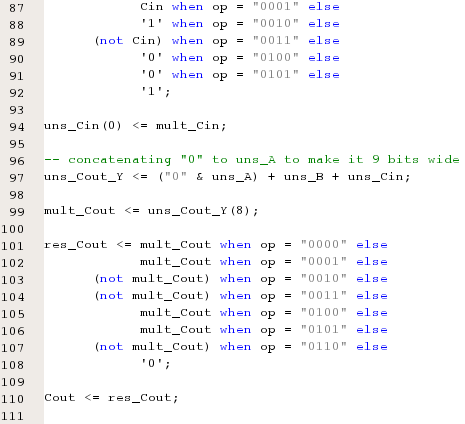


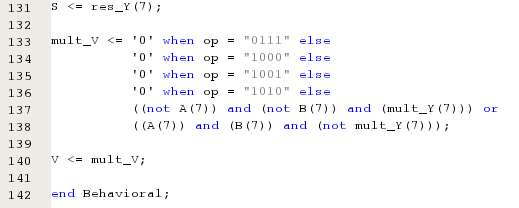
Following these declarations signals of type STD\_LOGIC\_VECTOR (7 downto 0) were also declared for the multiplexers mult\_A, mult\_B, mult\_Y and res\_Y. The signals of type STD\_LOGIC had to also be declared for the ALU to work. These included : mult\_Cin, mult\_Cout, res\_Cout and mult\_V.



1. After this was done, addition was implemented in the architecture body. This was a very challenging step as all the bits and pieces making up the ALU had to be explained and connected together and one small fault would easily cause compilation errors.







1. A test bench entitled *ALU\_tb* was then written to verify the ALUs operation and to test if it was implemented correctly. Care was taken to include once again the *USE ieee.numeric\_std.ALL;* package line and any clock-related functions were removed to be implemented as rest states directly in the stimulus processes during test cases. The 17 test cases created were carefully written to test as best as they can the ALUs functionality. All operations were tested at least once whereas some were tested more than once with one or more of the variables changed. Clock functions of 50ns intervals were implemented between one calculation and another and at the start making the duration of each the same. A short description is included at the start of every test case and the expected output is also written down as can be seen below:

-- Stimulus process

stim\_proc: process

begin

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #1 (ADD)

--Adding a + b only which should

--give a result of 2

a <= "00000001";

b <= "00000001";

cin <= '0';

op <= "0000";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #2 (ADD)

--This should also give a result of

--2 as Cin is not included in a

--normal ADD function

a <= "00000001";

b <= "00000001";

cin <= '1';

op <= "0000";

-- hold reset state for 50 ns.

wait for 50 ns;

wait for 50 ns;

--Test Case #2 (ADD)

--This should also give a result of

--2 as Cin is not included in a

--normal ADD function

a <= "00000001";

b <= "00000001";

cin <= '1';

op <= "0000";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #3 (ADC)

--Adds operand a+b+cin together which

--should give a result of 7

--Cin is 0 in this case

a <= "00000001";

b <= "00000110";

cin <= '0';

op <= "0001";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #4 (ADC)

--Testing the ADC function once again

--this time having cin = 1

--this should give an answer of 8

a <= "00000001";

b <= "00000110";

cin <= '1';

op <= "0001";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #5 (SUB)

--Subtracts b from a while ignoring cin

--should give a value of 55

--Test Case #4 (ADC)

--Testing the ADC function once again

--this time having cin = 1

--this should give an answer of 8

a <= "00000001";

b <= "00000110";

cin <= '1';

op <= "0001";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #5 (SUB)

--Subtracts b from a while ignoring cin

--should give a value of 55

a <= "01110001";

b <= "00111010";

cin <= '0';

op <= "0010";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #6 (SUB)

--Testing again the SUB function with

--different values for a, b and cin

--It's aim is to test if the operation

--returns back negative numbers and to

--check what happens to the result

--when cin = 1

--the answer should be -17

--It's aim is to test if the operation

--returns back negative numbers and to

--check what happens to the result

--when cin = 1

--the answer should be -17

a <= "00000001";

b <= "00010010";

cin <= '1';

op <= "0010";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #7 (SBB)

--This function subtracts b and cin from a

--This should give the answer of 70

a <= "01100001";

b <= "00011010";

cin <= '1';

op <= "0011";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #8 (INC)

--This adds 1 to the value of a and ignores

--the values of b and cin

--This adds 1 to the value of a and ignores

--the values of b and cin

--This should give an answer of 2

a <= "00000001";

b <= "00010111";

cin <= '1';

op <= "0100";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #9 (DEC)

--This subtracts 1 from the value of a and

--ignores the values of b and cin

--This should give an answer of 0

a <= "00000001";

b <= "00000110";

cin <= '0';

op <= "0101";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #10 (NEG)

--Negates the value stored in a only

--Should give an answer of -25

a <= "00011001";

b <= "00111111";

cin <= '1';

op <= "0110";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #11 (NOT)

--Turns the value of a into its bitwise

--complement and ignores b and cin

--The answer is -46

a <= "00101101";

b <= "00111110";

cin <= '0';

op <= "0111";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #12 (AND)

--Performs the logic operation a and b

--cin is ignored

--Its answer should be 0 as the values

--of a and b were carefully chosen

--to test what happens when 1 value is 1

--and the other is 0 for every n

--and the other is 0 for every n

a <= "10101001";

b <= "01010110";

cin <= '1';

op <= "1000";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #13 (AND)

--Testing further a and b, this time

--with different values for a and b

--answer should be 72

a <= "11101001";

b <= "01011110";

cin <= '0';

op <= "1000";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #14 (AND)

--The last test for and in which

--overflow is being tested by

--turning both a and b into all 1s

--Being 8-bit addition, the value

--turning both a and b into all 1s

--Being 8-bit addition, the value

--would be too large to be computed

--into 8-bits thus returning an

--incorrect value such as -1 instead

a <= "11111111";

b <= "11111111";

cin <= '1';

op <= "1000";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #15 (OR)

--Performs the logic operation a or b

--cin is ignored

--Its answer should be -1 as the values

--of a and b were carefully chosen

--to test what happens when 1 value is 1

--and the other is 0 for every n

a <= "01111101";

b <= "10000010";

cin <= '0';

op <= "1001";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #16 (OR)

--Testing a or b once more, this

--time with overflow

--As explained already, being 8-bit

--addition, the result would be too

--large to fit into just 8-bits

--and so an overflow occurs giving

--the value of -1 once again

a <= "11111111";

b <= "11111111";

cin <= '1';

op <= "1001";

-- hold reset state for 50 ns.

wait for 50 ns;

--Test Case #17 (XOR)

--Testing the XOR operation in

--which the answer should be

--39 when computed

a <= "01001101";

b <= "01101010";

cin <= '1';

op <= "1010";

--being the last operation, the

--value returned back from this

--operation is appended further

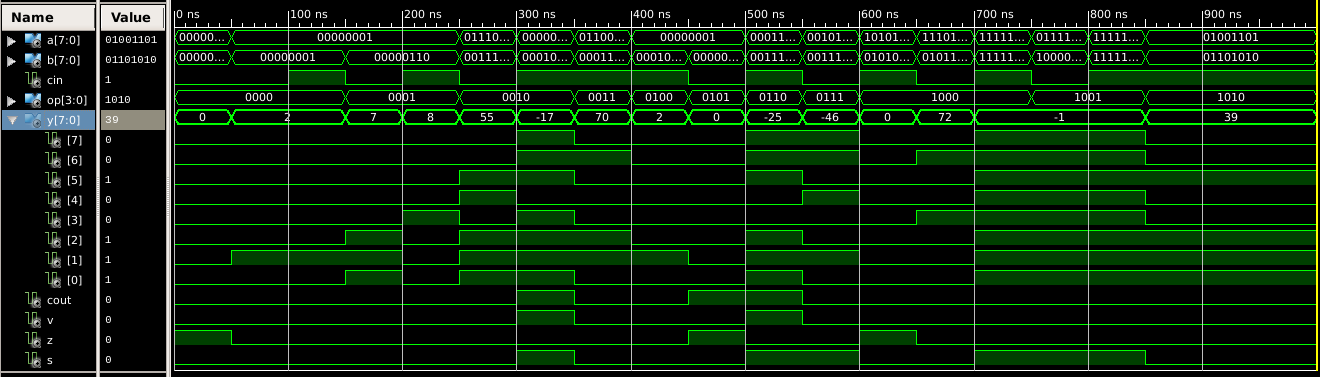
--until the end of the simulated

--model

wait;

end process;

END;

After having written all the test cases and corresponding comments, the *ALU\_tb* was simulated using Behavioral Simulation. The radix of y[7:0] was immediately changed to *Signed Decimal* to be able to compare the expected outputs to the actual ones. The output from the final test case was extended till the end of the stimulus process (i.e. till 1000ns). The Simulated Behavioral model is displayed below:

**Appendix: (Code)**

The code for both *ALU.vhd* and *ALU\_tb.vhd* is included in the 2 separate text files added to the submission folder as it was easier to do so rather than just copying and pasting the code here with the risks of changing its format.

**Conclusion:**

A simple ALU was successfully designed and simulated and the expected output matched the actual output.