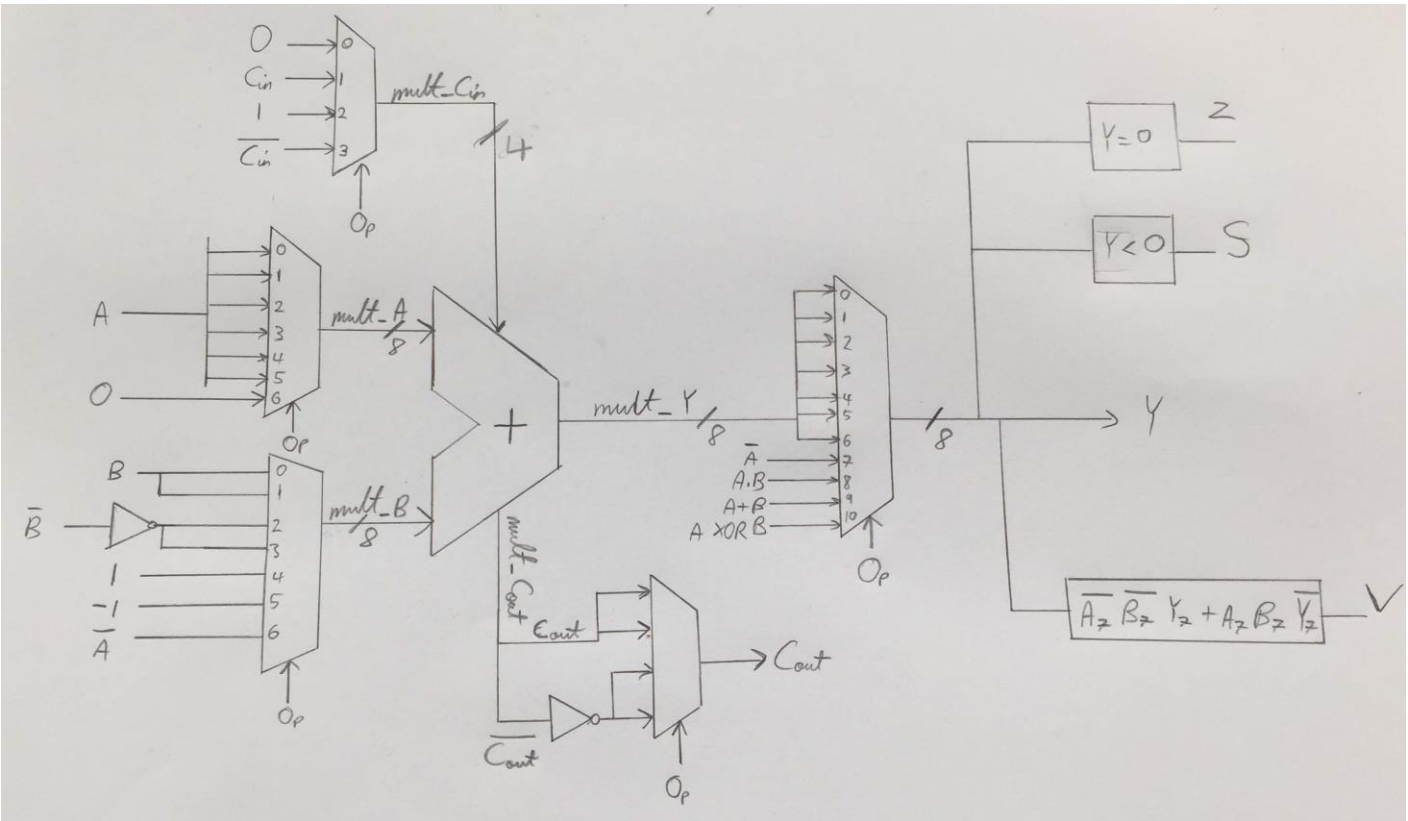
Computer Logic – Practical 4

Objective:

To design and implement a simple ALU.

Tasks:

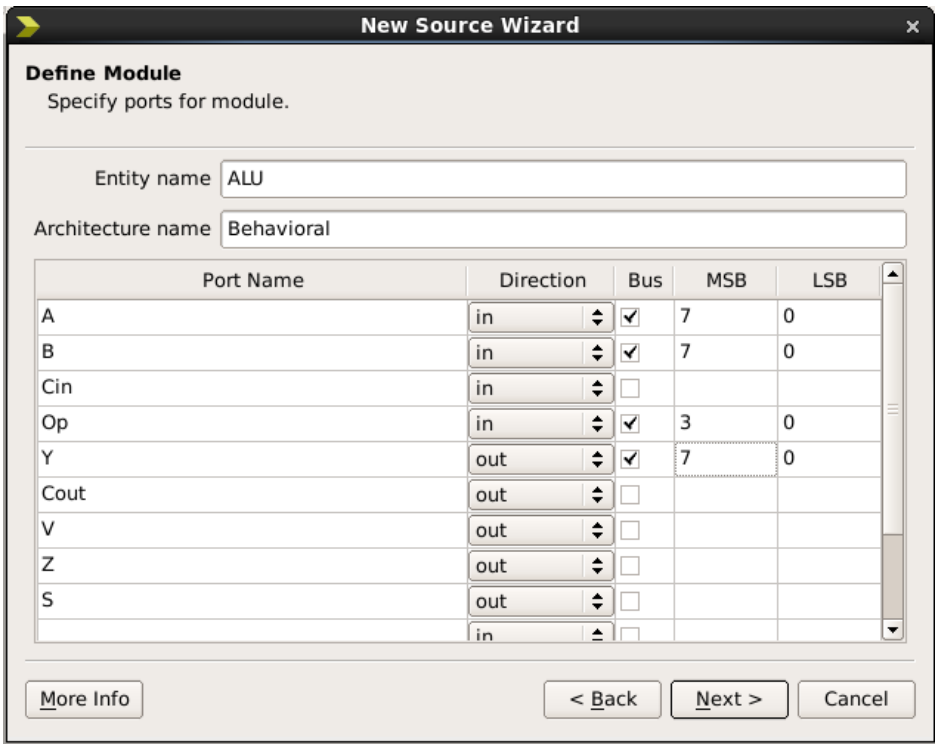
1) A sketch of the ALU implementation was drawn as can be seen:



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, C_{in} and $O_{p3:0}$ are the control inputs, Y is the 8-bit output operand and C_{out}, V, Z and S are the control outputs.

TABLE 1: THE ALU INPUTS AND OUTPUTS.

Name	Direction	Description
A	input	eight-bit input operand
B	input	eight-bit input operand
C_{in}	input	carry in
Op	input	four-bit operation code
Y	output	eight-bit output operand
C_{out}	output	carry out
V	output	overflow flag
Z	output	zero flag
S	output	sign flag, or negative flag



3) The following steps were followed for the implementation of the sketched ALU:

a) The package line *use IEEE.NUMERIC_STD.ALL* was entered at the top of the file under the other IEEE declaration.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;
```

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

```
architecture Behavioral of ALU is

signal uns_A: unsigned(7 downto 0); -- 8 bits
signal uns_B: unsigned(7 downto 0); -- 8 bits
signal uns_Cin: unsigned(0 downto 0); -- 1 bit
signal uns_Cout_Y: unsigned(8 downto 0); -- 9 bits

begin
```

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.

```
signal mult_A : STD_LOGIC_VECTOR (7 downto 0);
signal mult_B : STD_LOGIC_VECTOR (7 downto 0);
signal mult_Cin : STD_LOGIC;

signal mult_Cout:  STD_LOGIC;
signal res_Cout : STD_LOGIC;

signal mult_Y:  STD_LOGIC_VECTOR (7 downto 0);
signal res_Y : STD_LOGIC_VECTOR (7 downto 0);

signal mult_V:  STD_LOGIC;
```

c) 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.

```
62  begin
63
64  --mult_CHAR stands for multiplexer and not multiplier
65
66  mult_A <= A when op = "0000" else
67            A when op = "0001" else
68            A when op = "0010" else
69            A when op = "0011" else
70            A when op = "0100" else
71            A when op = "0101" else
72            "00000000";
73
74  uns_A <= unsigned(mult_A);
75
76  mult_B <=  B when op = "0000" else
77             B when op = "0001" else
78       (not B) when op = "0010" else
79       (not B) when op = "0011" else
80    "00000001" when op = "0100" else
81    "11111111" when op = "0101" else
82              (not A);
83
84  uns_B <= unsigned(mult_B);
85
86  mult_Cin <= '0' when op = "0000" else
```

```
87              Cin when op = "0001" else
88              '1' when op = "0010" else
89        (not Cin) when op = "0011" else
90              '0' when op = "0100" else
91              '0' when op = "0101" else
92              '1';
93
94  uns_Cin(0) <= mult_Cin;
95
96  -- concatenating "0" to uns_A to make it 9 bits wide
97  uns_Cout_Y <= ("0" & uns_A) + uns_B + uns_Cin;
98
99  mult_Cout <= uns_Cout_Y(8);
100
101 res_Cout <= mult_Cout when op = "0000" else
102             mult_Cout when op = "0001" else
103       (not mult_Cout) when op = "0010" else
104       (not mult_Cout) when op = "0011" else
105             mult_Cout when op = "0100" else
106             mult_Cout when op = "0101" else
107       (not mult_Cout) when op = "0110" else
108             '0';
109
110 Cout <= res_Cout;
111
112 mult_Y <= std_logic_vector(uns_Cout_Y(7 downto 0));
113
114 res_Y <= mult_Y when op = "0000" else
115          mult_Y when op = "0001" else
116          mult_Y when op = "0010" else
117          mult_Y when op = "0011" else
118          mult_Y when op = "0100" else
119          mult_Y when op = "0101" else
120          mult_Y when op = "0110" else
121        (not A) when op = "0111" else
122      (A and B) when op = "1000" else
123       (A or B) when op = "1001" else
124      (A xor B) when op = "1010";
125
126 Y <= res_Y;
127
128 Z <= '1' when res_Y = "00000000" else
129      '0';
130
131 S <= res_Y(7);
132
133 mult_V <= '0' when op = "0111" else
134           '0' when op = "1000" else
135           '0' when op = "1001" else
136           '0' when op = "1010" else
137           ((not A(7)) and (not B(7)) and (mult_Y(7))) or
138           ((A(7)) and (B(7)) and (not mult_Y(7)));
139
140 V <= mult_V;
141
142 end Behavioral;
```

4) 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;

                --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

                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

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 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
```

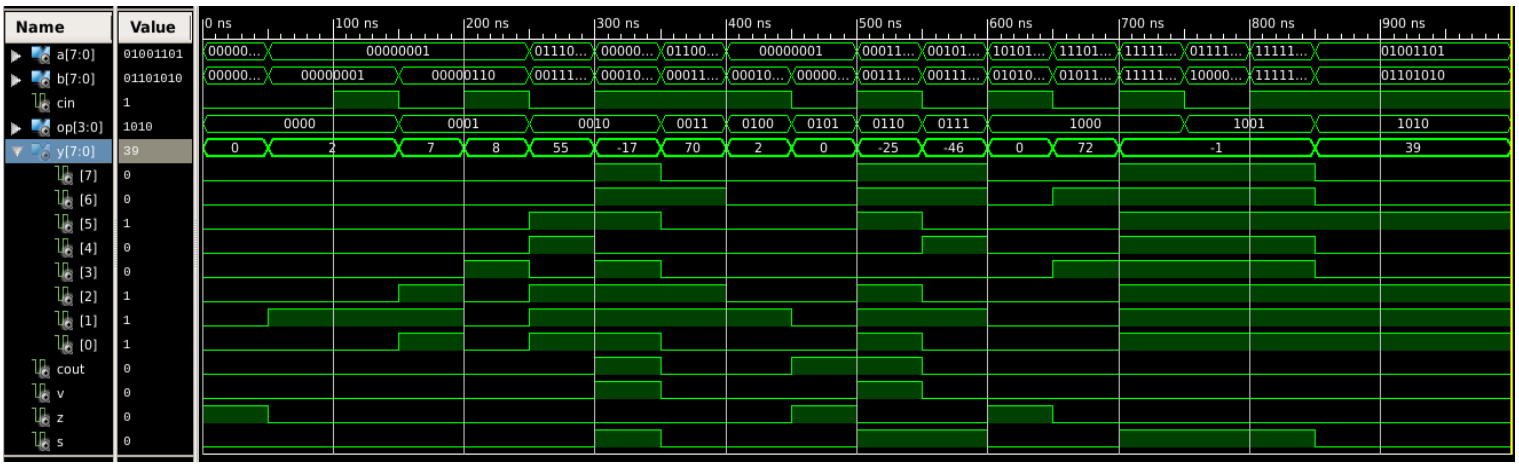
```
                --and the other is 0 for every n


                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
```

```
                --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.