Lab 4: Arithmetic Logic Unit

1) Purpose

Aim of this lab is to implement and design an arithmetic logic unit using VHDL and Basys 3. An ALU unit is basically a unit that generates some functions with given numbers according to a select signal. For example, it sums the given numbers when the select signal is “000” or subtracts them if the select signal is “001” or operates logic shift or bit wise logical operations. In this lab an ALU generating 8 functions is implemented. It functions addition when select signal is “000”; subtraction when select signal is “001”; increment, adds 1 to num1, when select signal is “010”; bit wise AND when select signal is “011”; one’s complement when select signal is “100”; bit wise XOR when select signal is “101”; left logic shift when select signal is “110”; left rotation shift when select signal is “111”. A multiplexer used for selecting the output of the signal also a overflow detection output is implemented.

2) Methodology

To implement a ALU using VHDL, a modular design manner is followed. To generate summation subtraction and increment functions a 4 bit adder is implemented. First a full adder is designed then using the full adder component a four bit adder is generated and at the top module generating the ALU this four bit adder is used, the other bit wise logic operations and shifts implemented in the top module. The output of the ALU is controlled by a 3 bit select input signal. A multiplexer is used to determine the output of the ALU regarding to select signal.

The output functions of the ALU regarding to select signal can be seen in the figure below.

Select signal	ALU output function
“000”	Addition
“001”	Subtraction
“010”	Increment
“011”	Bit wise AND
“100”	One’s complement
“101”	Bit wise XOR
“110”	Left logic shift
“111”	Left rotation shift

Figure 2.1 output functions of the ALU regarding to select signal.

Addition and subtraction functions adds or subtracts num1 and num2, increment increases num1 by adding 1, bit wise logic operations generate a output logic vector whose bits are the outputs of the logic operation of the given logic vector’s bits, left logic shift shifts each bit of the vector towards left (most significant digit) the most significant bit of the given vector disappears and a ‘0’ bit is concatenated to least significant bit of the resulting vector and left rotation shifts each bit to left as left logic shift but it concatenates most significant bit of the input vector to the least significant bit of the output vector. Addition, increment and subtraction functions are implemented using 1 four bit adder and 2 multiplexers. Regarding to the last two digits of the select signal a multiplexer connects the input carry to ‘0’ if the signals are increment or addition, otherwise assigned to ‘1’ to operate subtraction. Also another multiplexer controls the inputs of the four bit adder if the signal is addition inputs of the four bit adder are num1 and num2, if it is in subtraction mode they are num1 and not(num2) or if it is in increment mode they are num1 and “0001”. bit wise shift and logic functions implemented by assigning a signal’s digits to the intended outputs of the functions. At the and another multiplexer is used to determine the final output. Additionally, a internal overflow detection output signal is designed in the four bit adder module it operates by generating the function C_out(3) XOR C_out(2). By assigning xor value of 3rd and 2nd carry signals a overflow detection signal is generated. Since if the mode of ALU is not in the addition, subtraction or increment mode it’s overflow output will always be ‘0’ another multiplexer for the overflow detection signal is used. The design schematics can be seen in the figures bellow also the design source codes added to appendix.

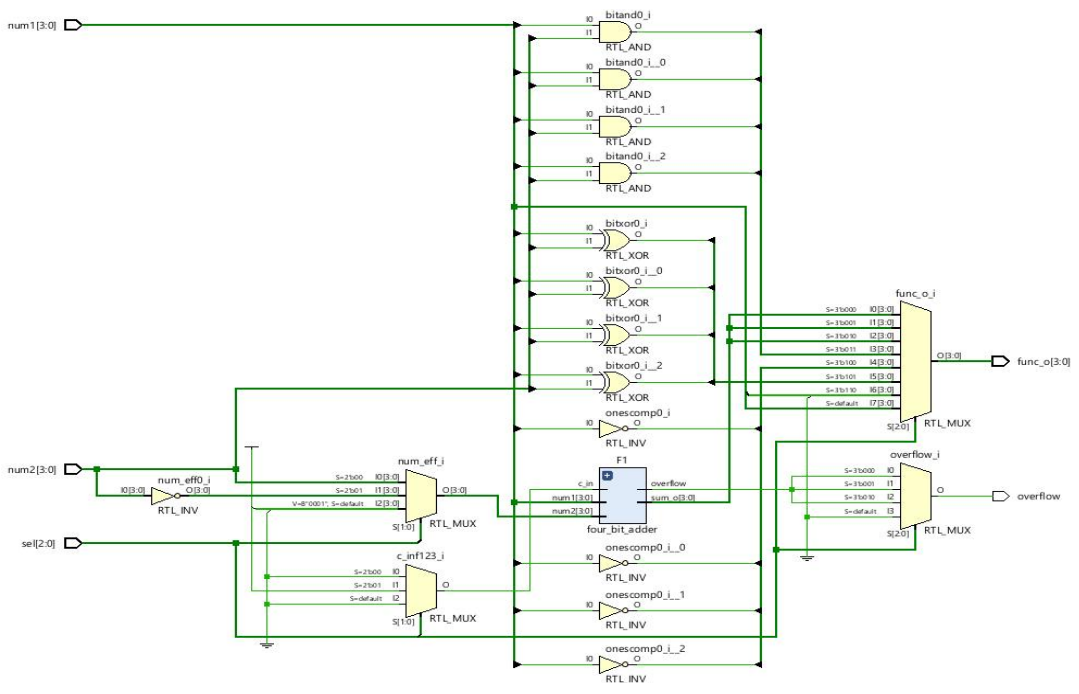

Figure 2.2 schematic description of the ALU design.

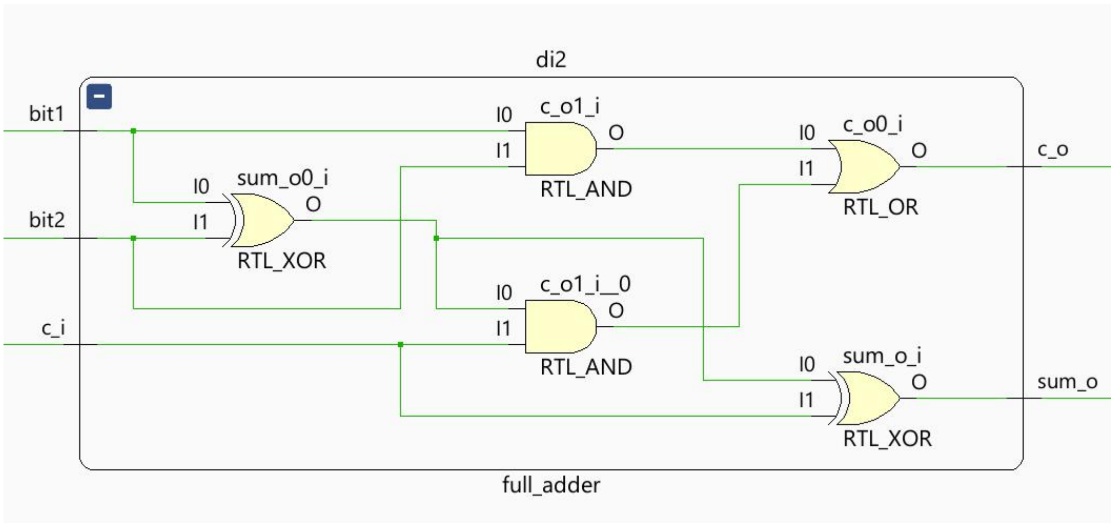

Figure 2.3 schematic description of the full adder component design.

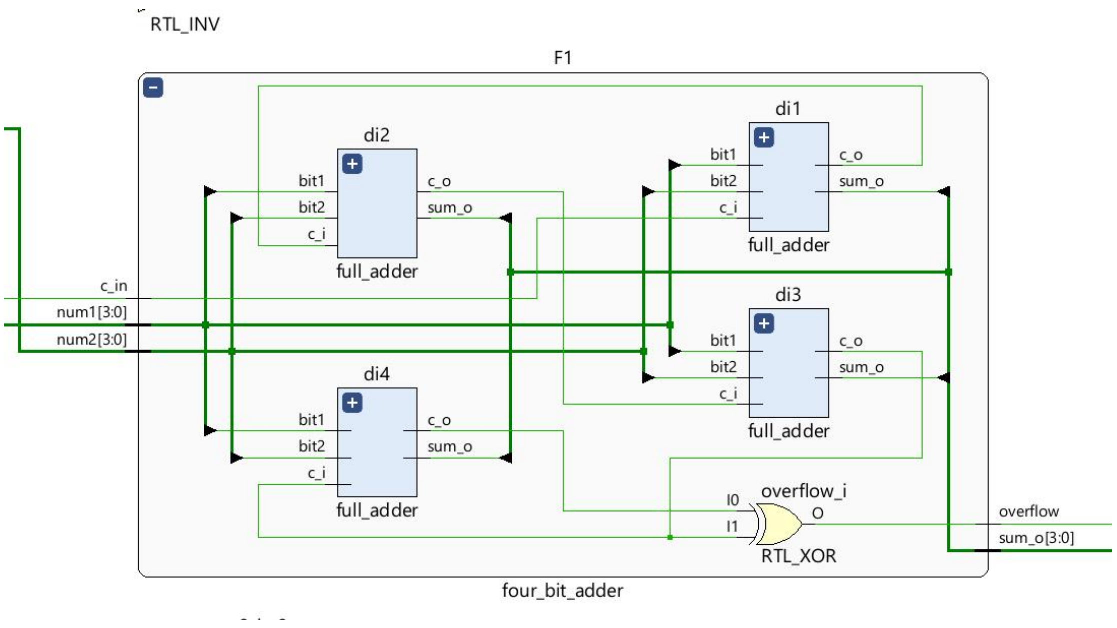

Figure 2.4 schematic description of the four bit adder component design.

3) Results

After completing the design a test bench code is written including 8 different functions with proper input combinations. The simulation result showed logic vectors as converted to decimal numbers to make it more readable but it used a as “1010”, b as “1011”, d as “1101”, e as “1110” and f as “1111”. The resulting test bench signal can be seen the figure bellow. Also the code for test bench is added to appendix.

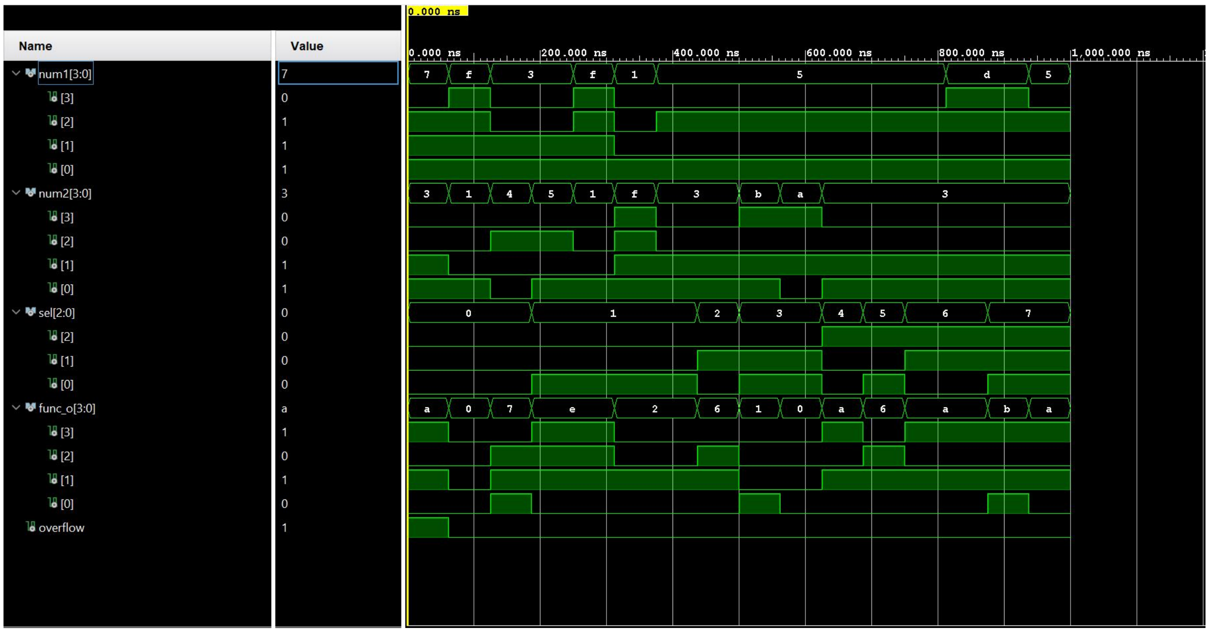

Figure 3.1

Result of behavioral simulation.

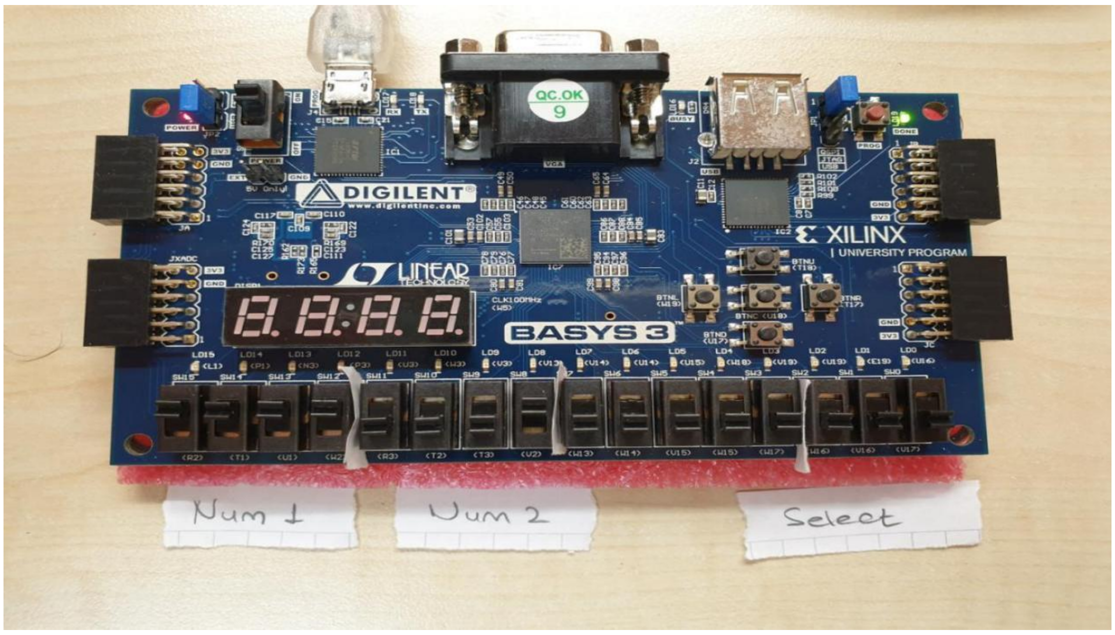

Figure 3.2 num1 <= 1111 (-1) ; num2 <= 0001 (1) ; sel 000 (add) ; result 0000 (0)

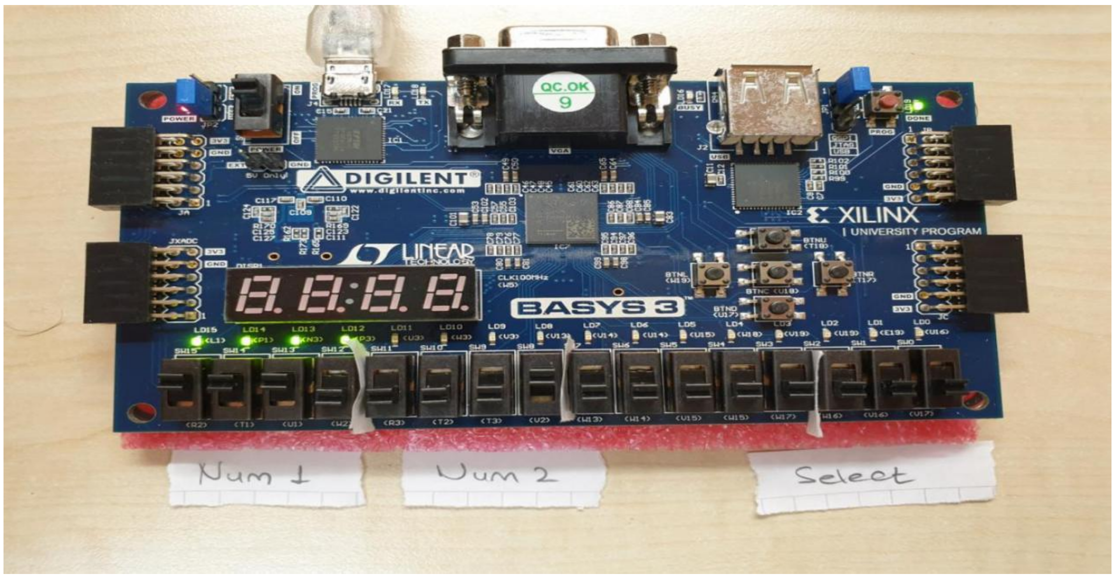

Figure 3.3 num1 <= 1110 (-2) ; num2 <= 0001 (1) ; sel 000 (add) ; result 1111 (-1)

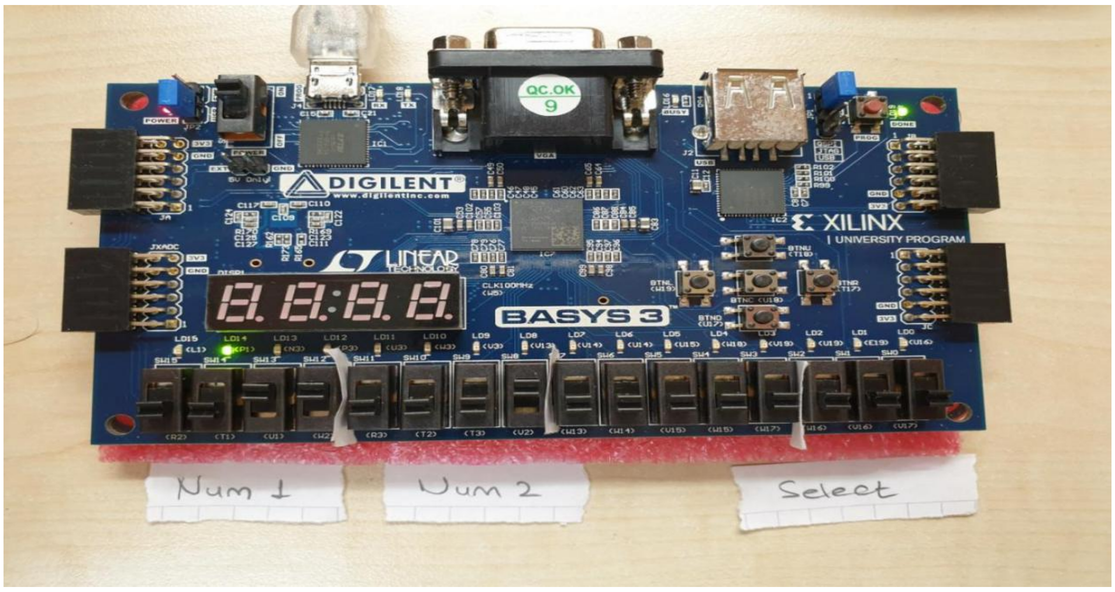

Figure 3.4 num1 <= 0011 (3) ; num2 <= 0001 (1) ; sel 000 (add) ; result 0100 (4)

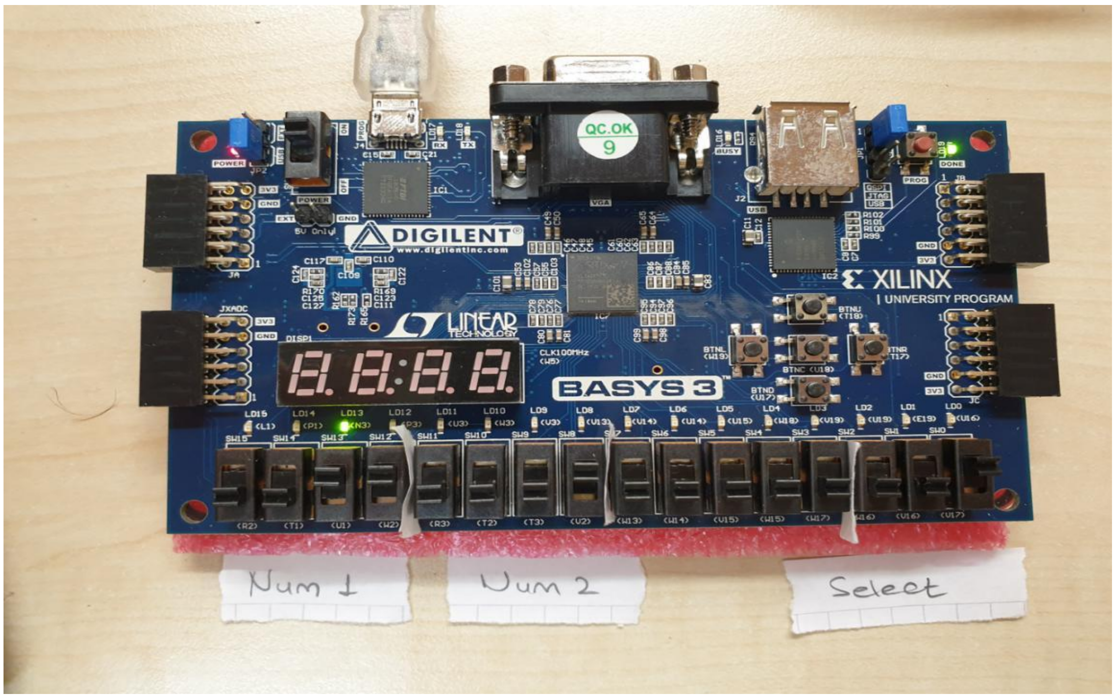

Figure 3.5 num1 <= 0011 (3) ; num2 <= 0001 (1) ; sel 001 (subtract) ; result 0010 (4)

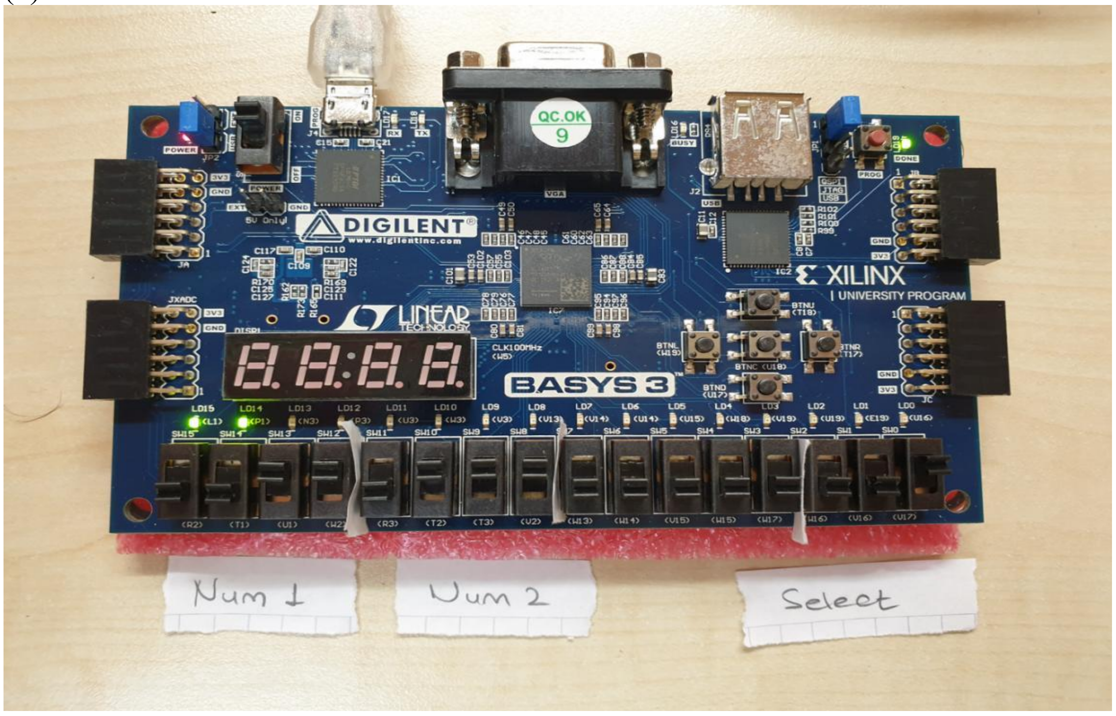

Figure 3.6 num1 <= 0011 (3) ; num2 <= 0111 (7) ; sel 001 (subtract) ; result 1100 (-4)

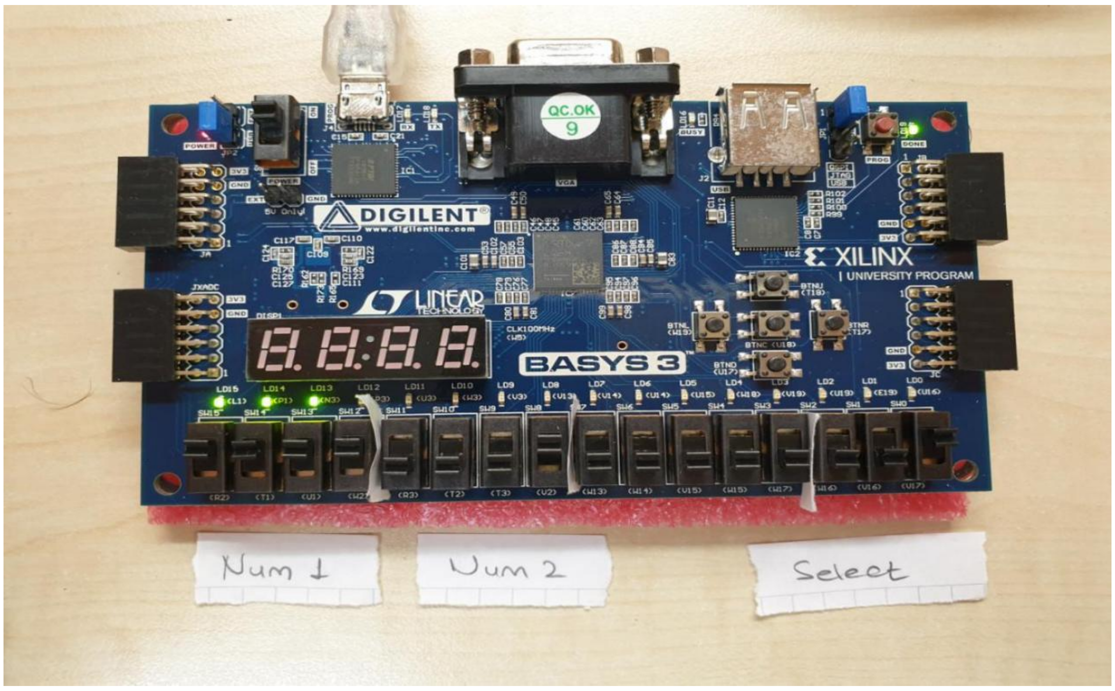

Figure 3.7 num1 <= 1111 (-1) ; num2 <= 0001 (1) ; sel 001 (subtract) ; result 1110 (-2)

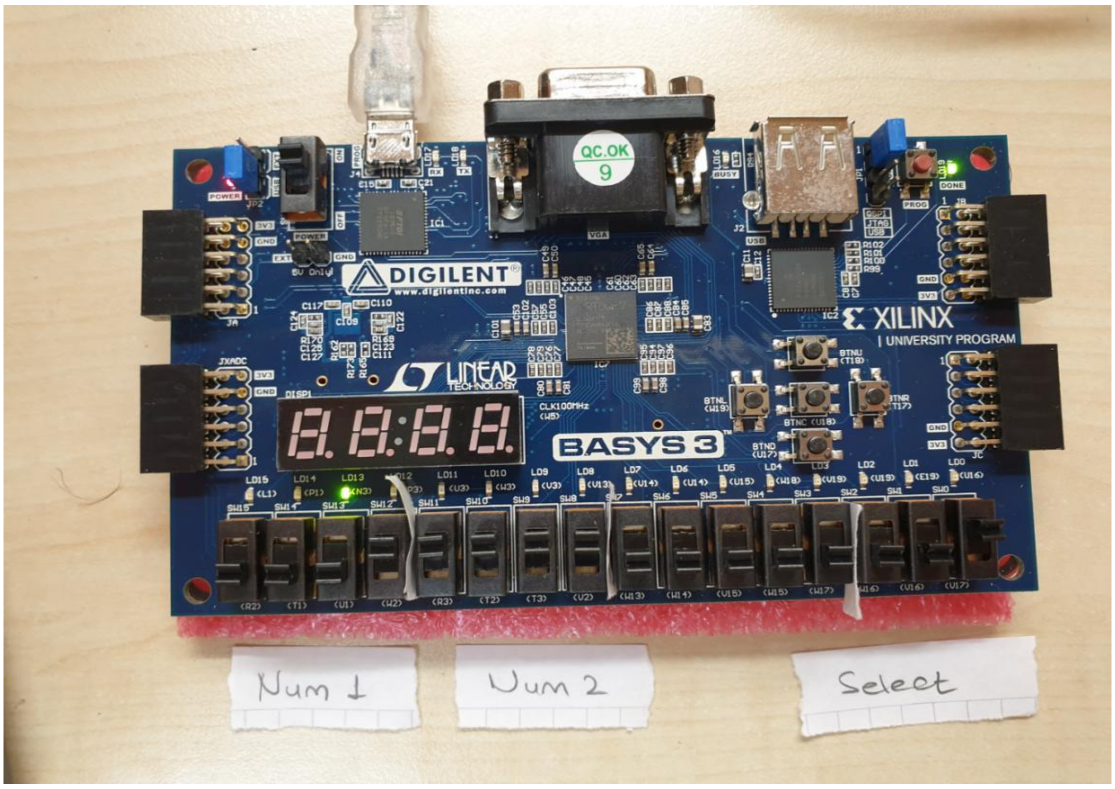

Figure 3.8 num1 <= 0001 (1) ; num2 <= 1111 (-1) ; sel 001 (subtract) ; result 0010 (2)

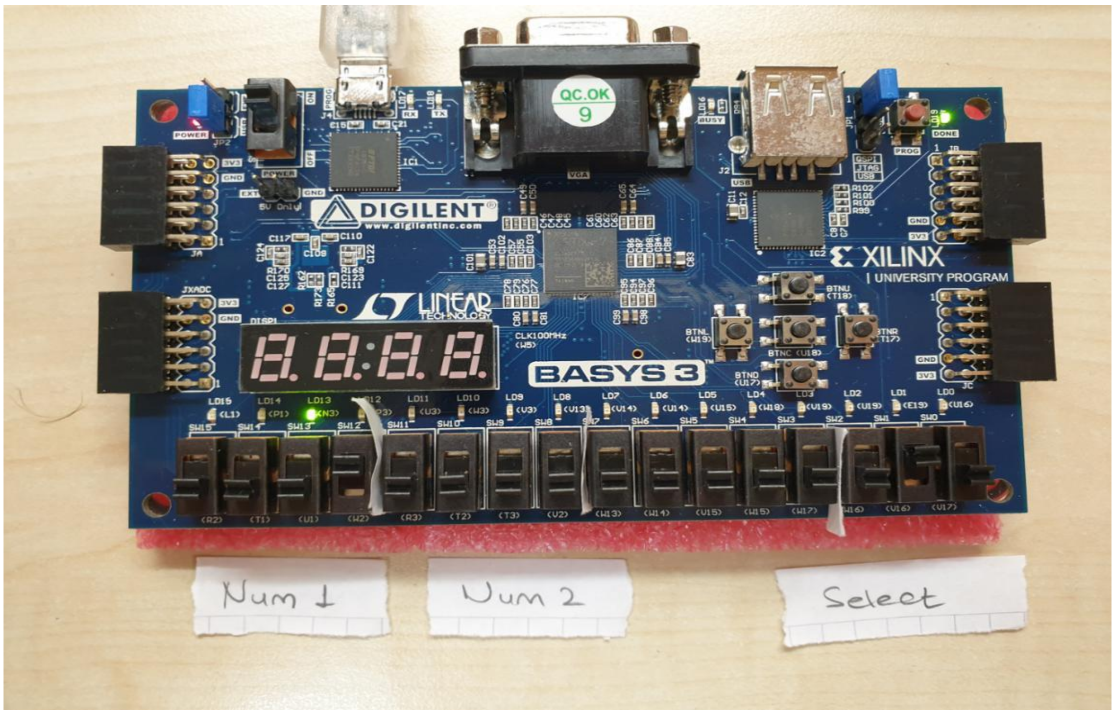

Figure 3.9 num1 <= 0001 (1) ; num2 <= 0000 (0) ; sel 010 (inc.) ; result 0010 (2)

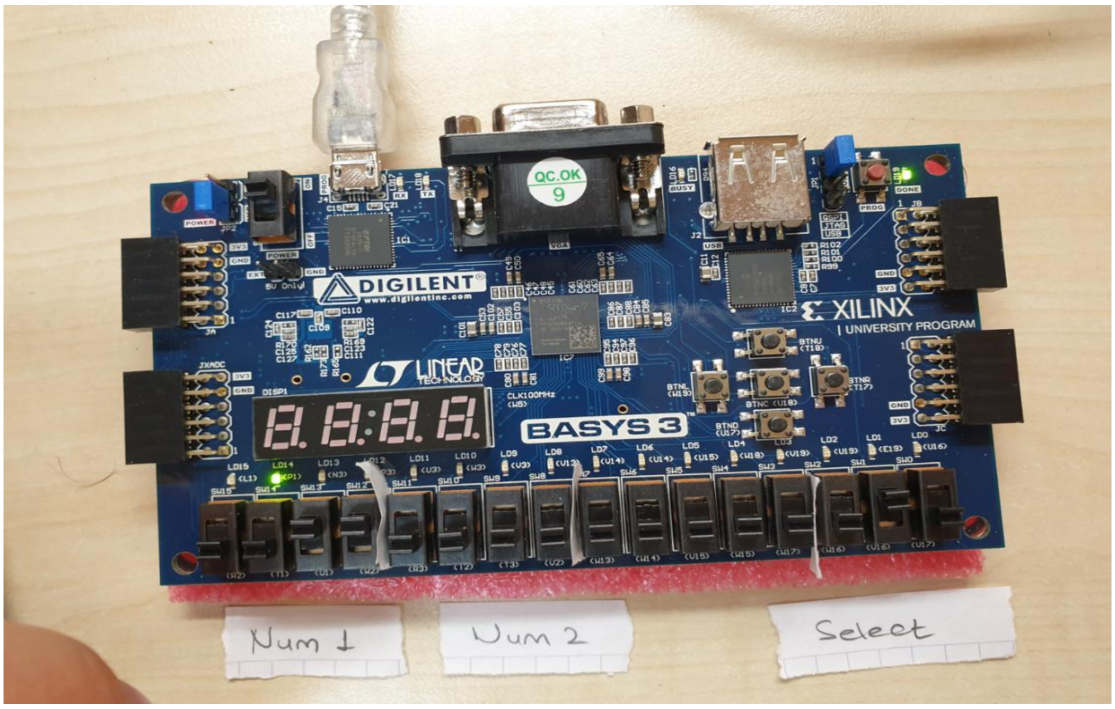

Figure 3.10 num1 <= 0011 (3) ; num2 <= 0000 (0) ; sel 010 (inc.) ; result 0100 (4)

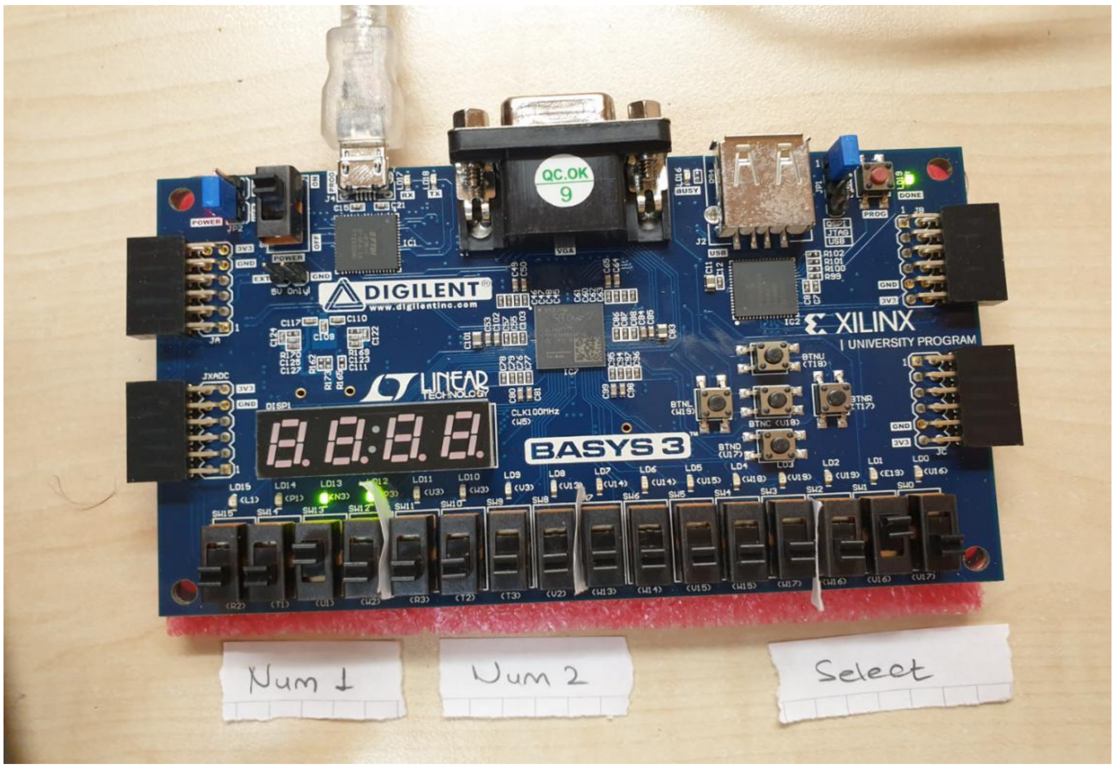

Figure 3.10 num1 <= 0010 (2) ; num2 <= 0111 (7) ; sel 010 (inc) ; result 0011 (3)

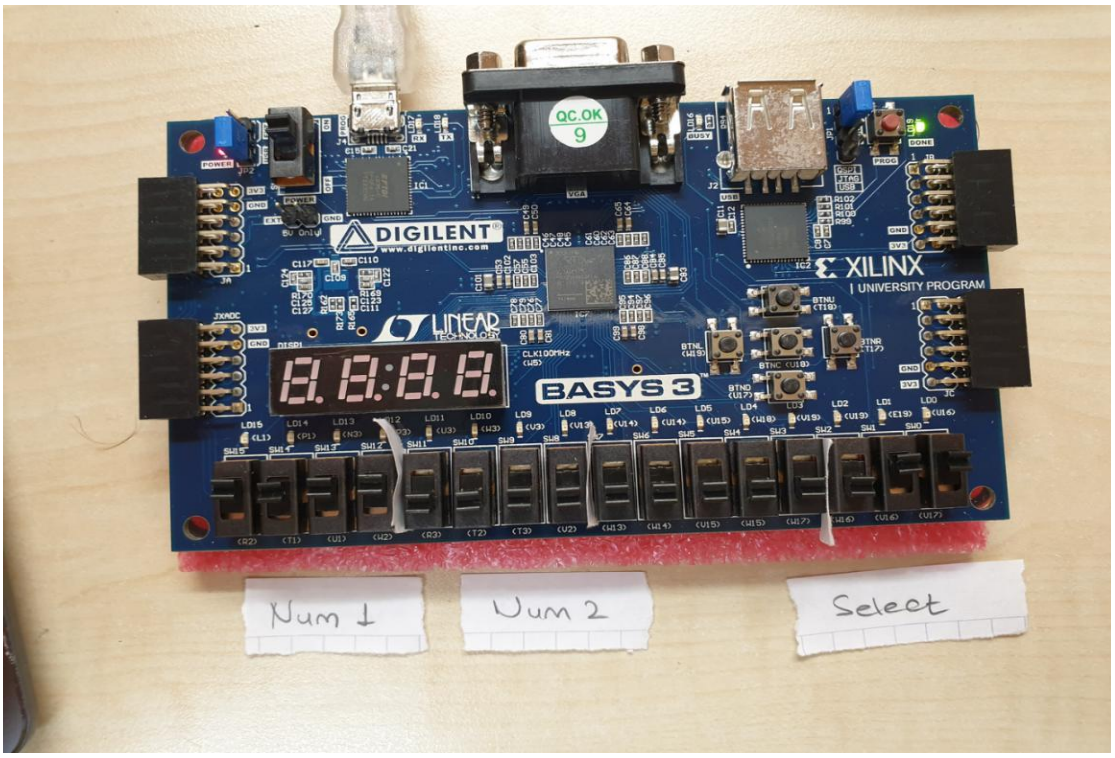

Figure 3.11 num1 <= 1111 ; num2 <= 0000 ; sel 011 (bitand) ; result 0000

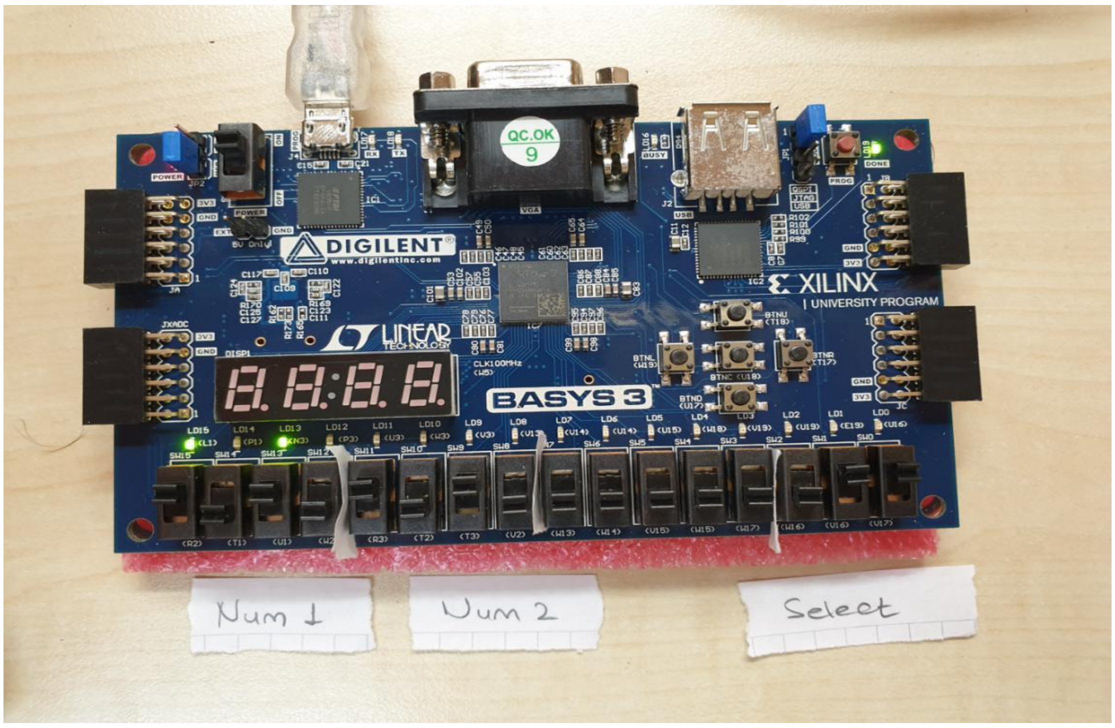

Figure 3.12 num1 <= 1010 ; num2 <= 1010 ; sel 011 (bitand) ; result 1010

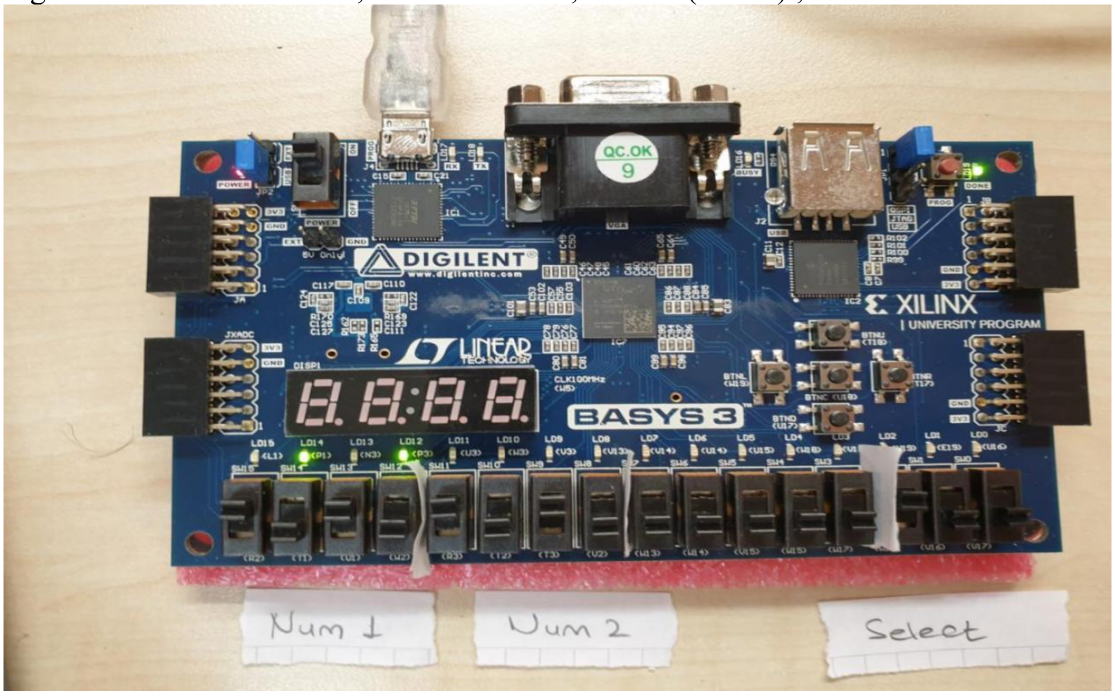

Figure 3.13 num1 <= 1010 ; num2 <= 1010 ; sel 100 (onescomp) ; result 0101

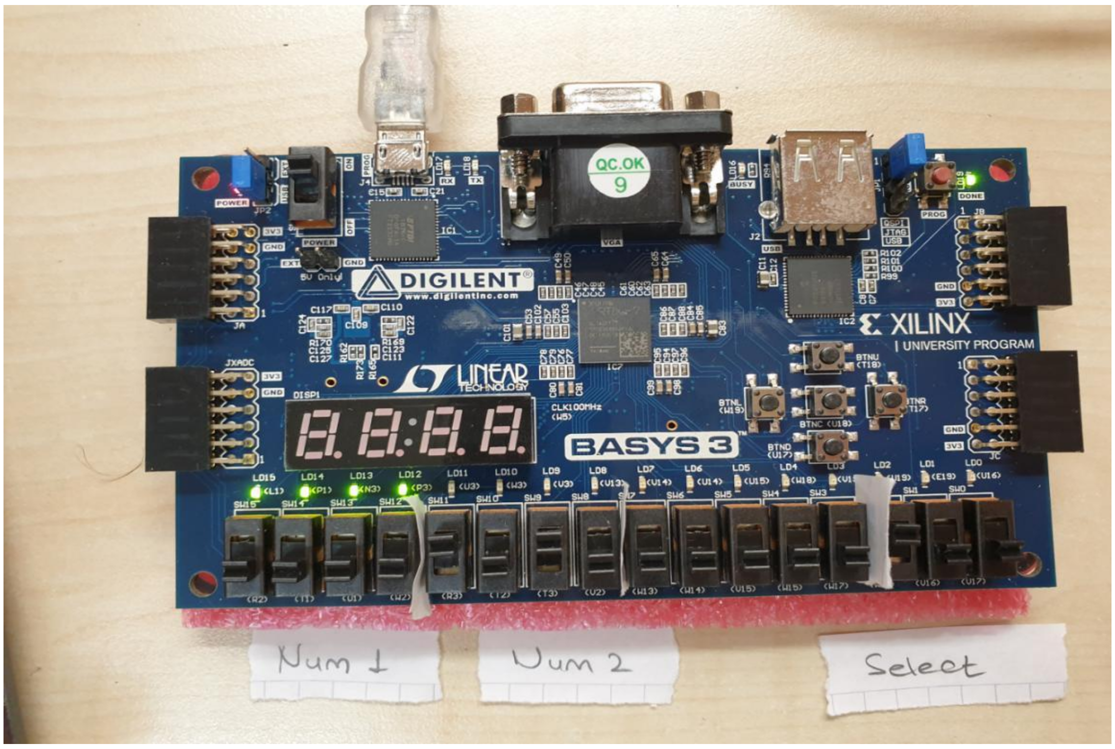

Figure 3.14 num1 <= 0000 ; num2 <= 1010 ; sel 100 (onscomp) ; result 1111

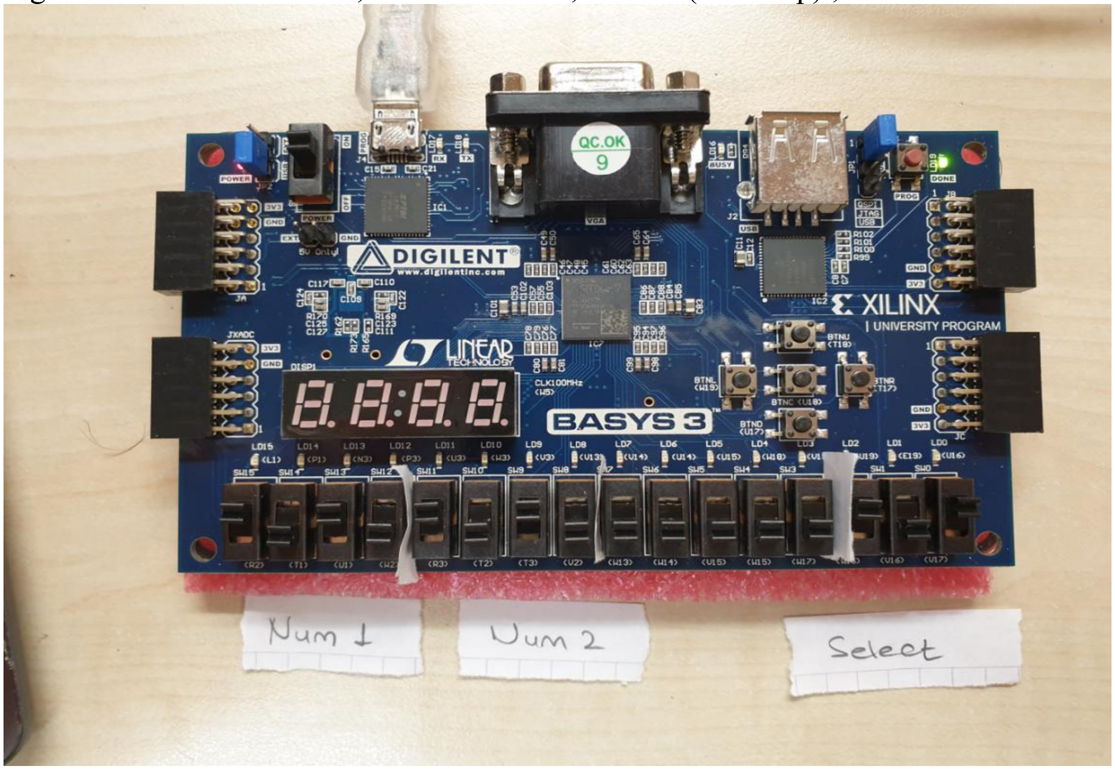

Figure 3.15 num1 <= 1010 ; num2 <= 1010 ; sel 101 (bitxor) ; result 0000

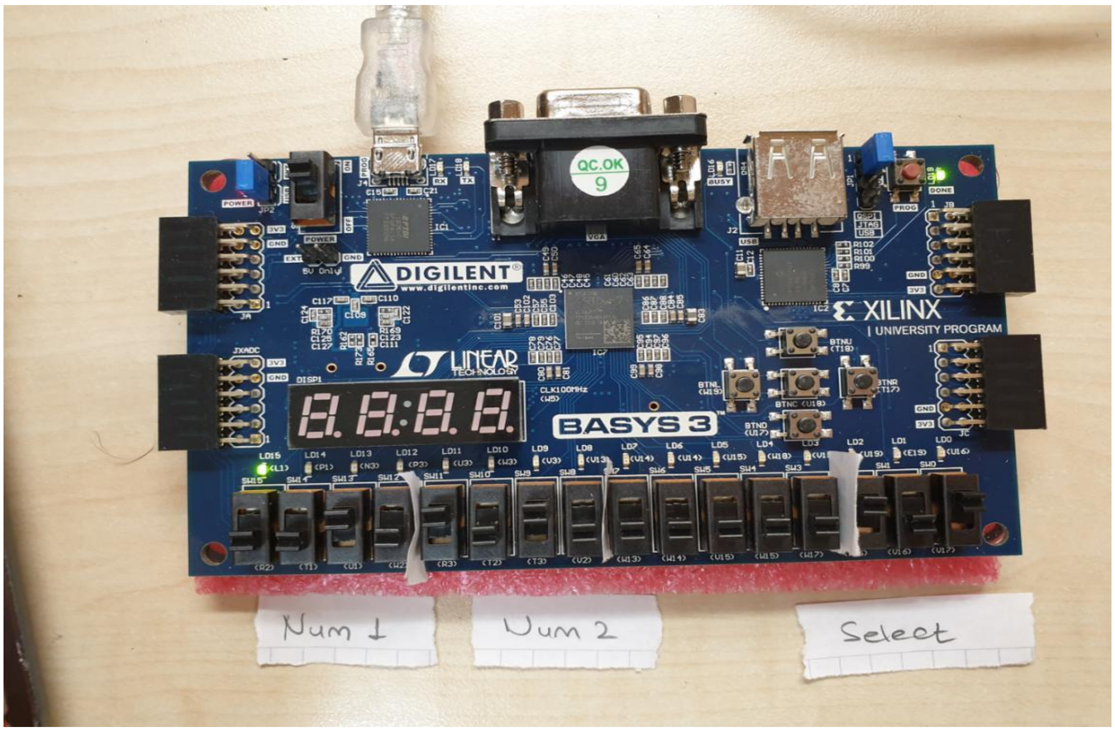

Figure 3.16 num1 <= 0010 ; num2 <= 1010 ; sel 101 (bitxor) ; result 1000

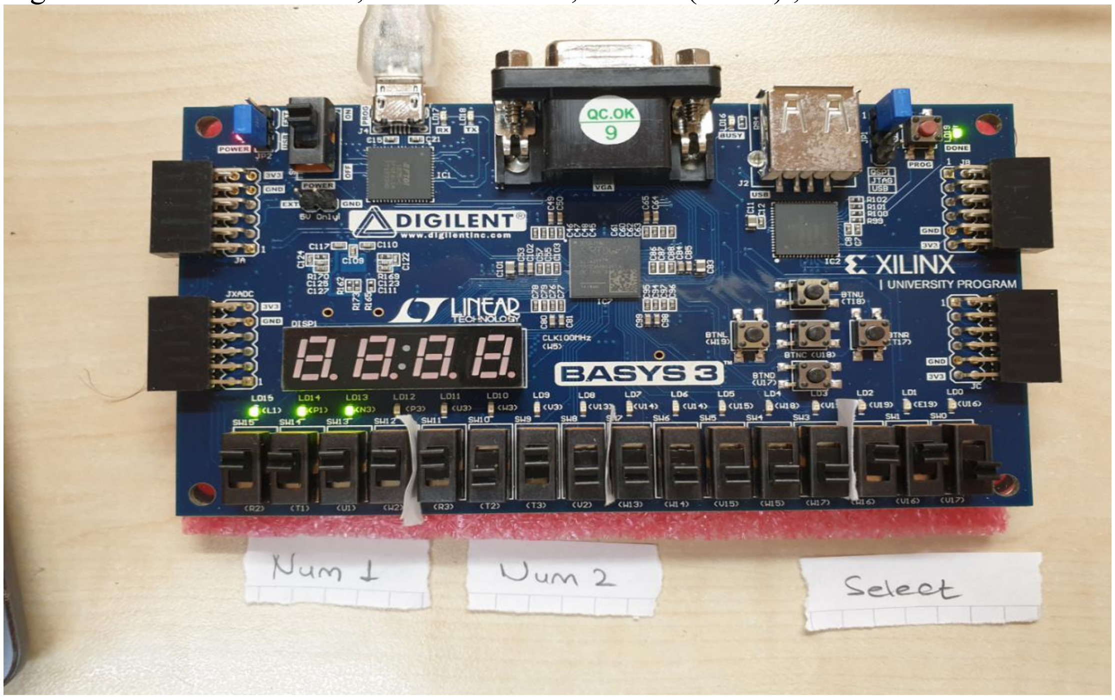

Figure 3.17 num1 <= 1111 ; num2 <= 1010 ; sel 110 (left logic shift) ; result 1110

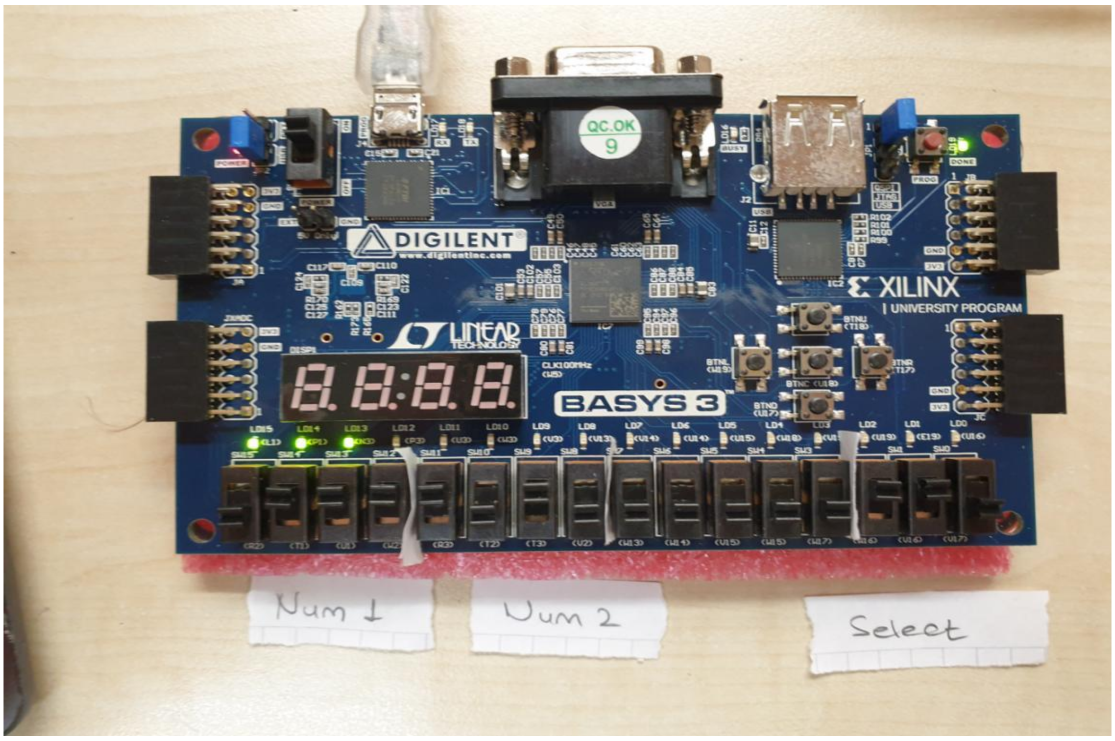

Figure 3.18 num1 <= 0111 ; num2 <= 1010 ; sel 110 (left logic shift) ; result 1110

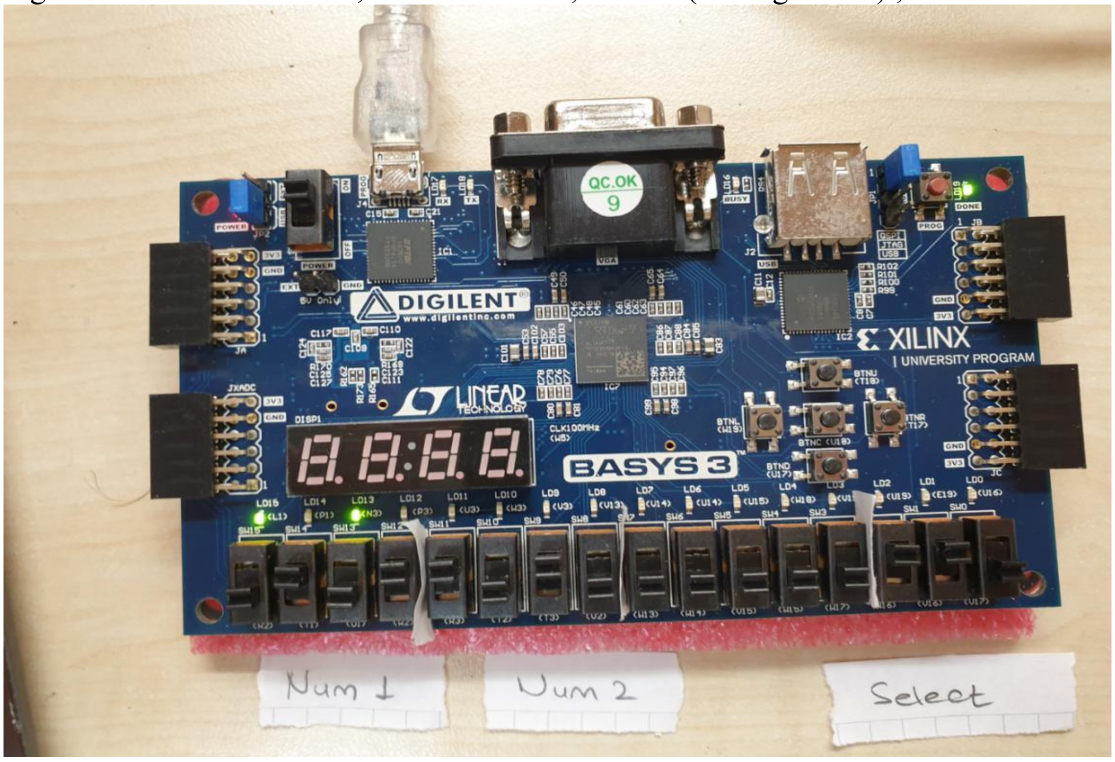

Figure 3.19 num1 <= 0101 ; num2 <= 1010 ; sel 110 (left logic shift) ; result 1010

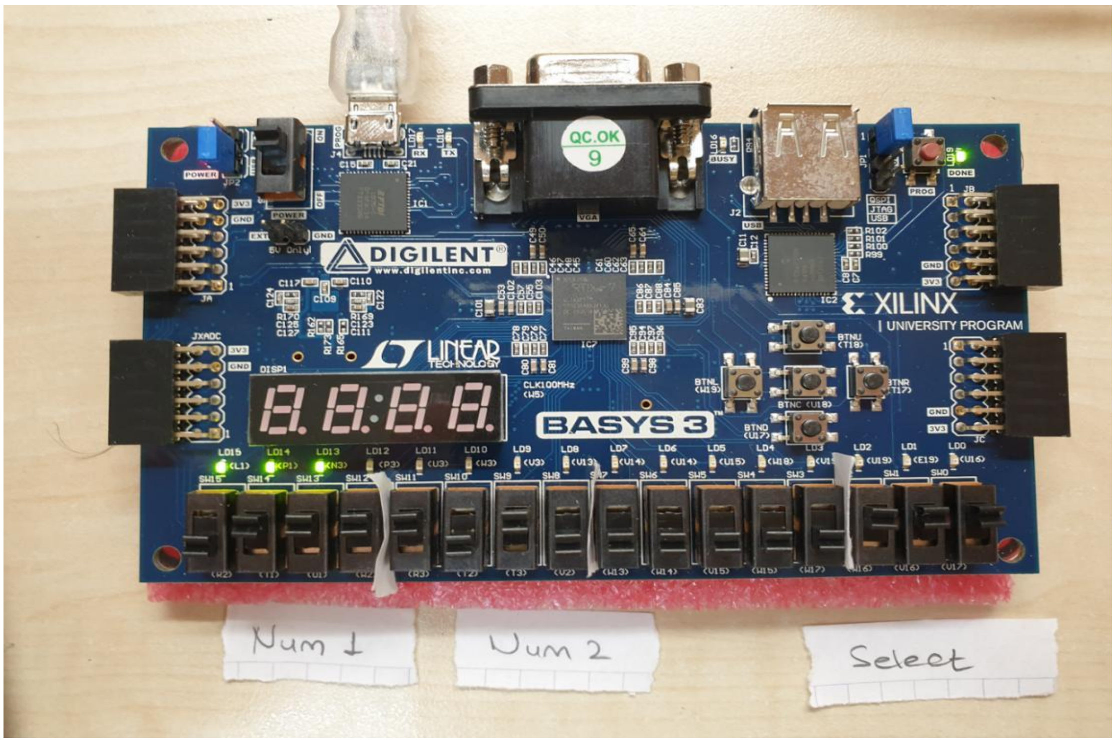

Figure 3.20 num1 <= 0111 ; num2 <= 1010 ; sel 111 (left rotation shift) ; result 1110

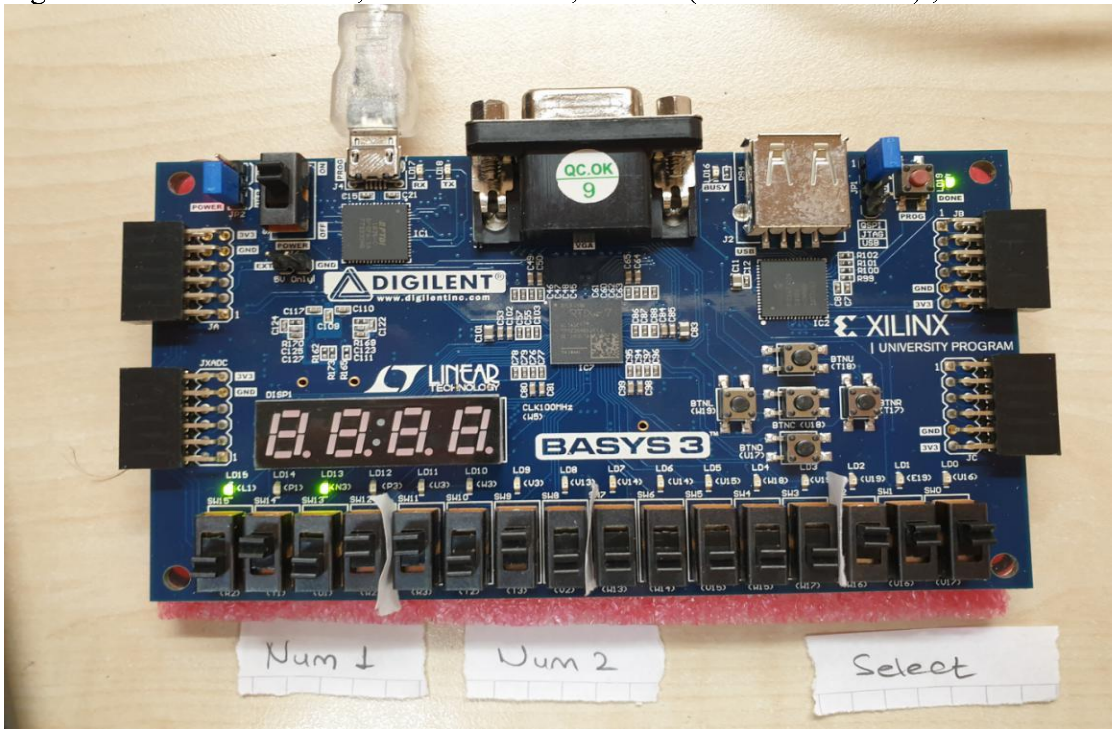

Figure 3.21 num1 <= 0101 ; num2 <= 1010 ; sel 111 (left rotation shift) ; result 1010

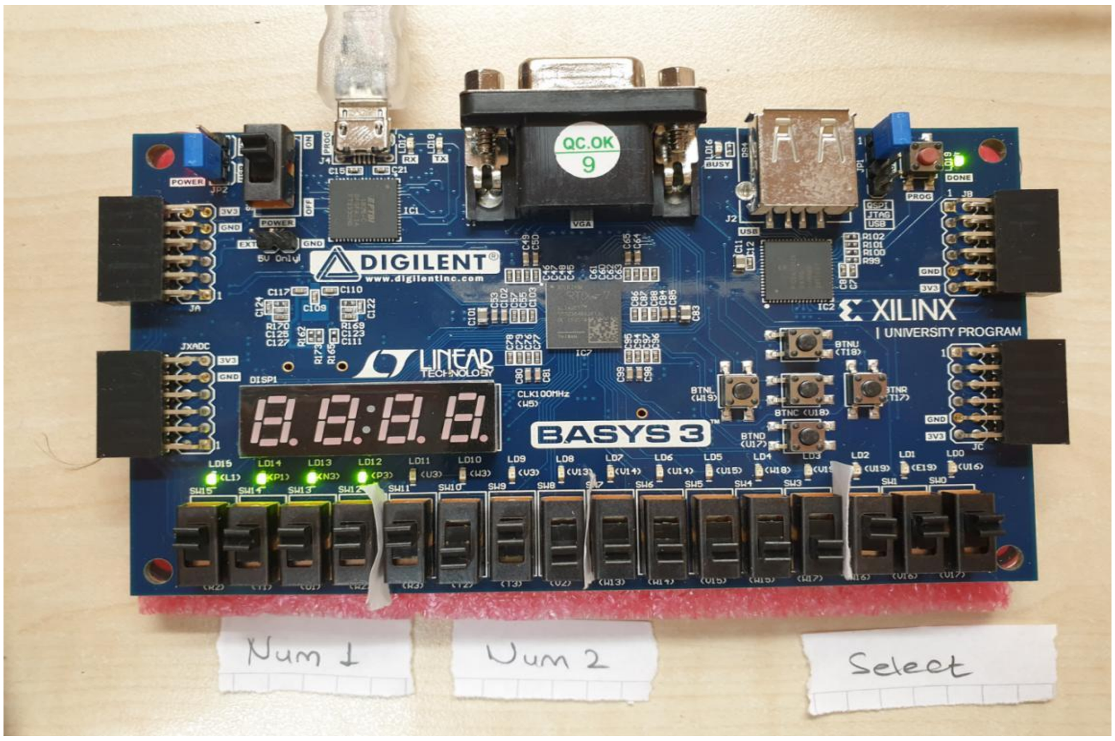

Figure 3.22 num1 <= 1111 ; num2 <= 1010 ; sel 111 (left rotation shift) ; result 1111

4) Conclusion

In conclusion, we learned how to implement a arithmetic logic unit with VHDL using Basys 3 fpga. Also we used multiplexers and employed modular design by implementing component declaration and instantiation. We learned basic functions that a ALU can do and learned it is used in CPU’s and GPU’s. We refreshed our VHDL and combinational circuit design knowledge.

4) Appendix

- Full adder

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


entity full_adder is

   Port ( bit1 : in STD_LOGIC;

          bit2 : in STD_LOGIC;

          c_i : in STD_LOGIC;

          sum_o : out STD_LOGIC;

          c_o : inout STD_LOGIC);

end full_adder;


architecture Behavioral of full_adder is


begin

sum_o <= ( bit1 xor  bit2) xor c_i;

c_o <= (bit1 and bit2) or ((bit1 xor bit2) and c_i);

end Behavioral;
```

- Four bit adder

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;


entity four_bit_adder is
   Port ( num1     : in std_logic_vector (3 downto 0);
          num2     : in std_logic_vector (3 downto 0);
          sum_o    : out std_logic_vector (3 downto 0);
          c_in     : in STD_LOGIC;
          overflow : out std_logic);
end four_bit_adder;


architecture Behavioral of four_bit_adder is
signal  c_out :  std_logic_vector (3 downto 0):="0000";
component full_adder is
   Port ( bit1 : in STD_LOGIC;
          bit2 : in STD_LOGIC;
          c_i : in STD_LOGIC;
          sum_o : out STD_LOGIC;
          c_o : inout STD_LOGIC);
end component full_adder;
begin
di1 : full_adder port map(bit1 => num1(0),bit2 => num2(0),c_i => c_in     ,sum_o=>
sum_o(0),c_o=> c_out(0));
di2 : full_adder port map(bit1 => num1(1),bit2 => num2(1),c_i => c_out(0),sum_o=>
sum_o(1),c_o=> c_out(1));
```

```
di3 : full_adder port map(bit1 => num1(2),bit2 => num2(2),c_i => c_out(1),sum_o=>
sum_o(2),c_o=> c_out(2));
di4 : full_adder port map(bit1 => num1(3),bit2 => num2(3),c_i => c_out(2),sum_o=>
sum_o(3),c_o=> c_out(3));
overflow <= c_out(3) xor c_out(2);
end Behavioral;
```

- ALU top module

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

entity alu_yit is
   Port ( num1    : in  std_logic_vector  (3 downto 0);
          num2    : in  std_logic_vector  (3 downto 0);
          sel     : in  std_logic_vector  (2 downto 0);
          func_o  : out std_logic_vector  (3 downto 0);
          overflow: out std_logic                   );
end alu_yit;

architecture Behavioral of alu_yit is
signal summation : std_logic_vector(3 downto 0) :="0000";
signal substr    : std_logic_vector(3 downto 0) :="0000";
signal increment : std_logic_vector(3 downto 0) :="0000";
signal bitand    : std_logic_vector(3 downto 0) :="0000";
signal onescomp  : std_logic_vector(3 downto 0) :="0000";
signal bitxor    : std_logic_vector(3 downto 0) :="0000";
signal llogshf   : std_logic_vector(3 downto 0) :="0000";
signal bitlrot   : std_logic_vector(3 downto 0) :="0000";
signal c_out     : std_logic_vector(3 downto 0) :="0000";

signal subsnum2  : std_logic_vector(3 downto 0) :="0000";
signal overflow1 : std_logic :='0';
signal c_inf123  : std_logic:='0';
signal num_eff   : std_logic_vector (3 downto 0):="0000";
signal mult_contr: std_logic_vector (1 downto 0):="00";
component four_bit_adder is
    Port ( num1     : in    std_logic_vector (3 downto 0);
           num2     : in    std_logic_vector (3 downto 0);
           sum_o    : out   std_logic_vector (3 downto 0);
           c_in     : in    std_logic                    ;
           overflow : out   std_logic);
end component four_bit_adder;
begin
mult_contr(0) <= sel(0);
mult_contr(1) <= sel(1);
with mult_contr select
 num_eff <= num2      when "00",
            not(num2) when "01",
            "0001"    when others;
with mult_contr select
 c_inf123 <= '0' when "00",
            '1' when "01",
            '0' when others;
F1 : four_bit_adder port map(num1 => num1, num2 => num_eff , c_in => c_inf123,
sum_o => summation , overflow=>overflow1);


bitand(0) <= num1(0) and num2(0);
bitand(1) <= num1(1) and num2(1);
bitand(2) <= num1(2) and num2(2);
bitand(3) <= num1(3) and num2(3);
onescomp(0) <= not(num1(0));
onescomp(1) <= not(num1(1));
onescomp(2) <= not(num1(2));
onescomp(3) <= not(num1(3));
bitxor(0)   <= num1(0) xor num2(0);
bitxor(1)   <= num1(1) xor num2(1);
bitxor(2)   <= num1(2) xor num2(2);
bitxor(3)   <= num1(3) xor num2(3);
llogshf(0)<= '0';
llogshf(1)<= num1(0);
llogshf(2)<= num1(1);
llogshf(3)<= num1(2);
bitlrot(0)<= num1(3);
bitlrot(1)<= num1(0);
bitlrot(2)<= num1(1);
bitlrot(3)<= num1(2);
```

```
substr    <= summation;
increment <= summation;
with sel select
func_o <= summation when "000",
          substr    when "001",
          increment when "010",
          bitand    when "011",
          onescomp  when "100",
          bitxor    when "101",
          llogshf   when "110",
          bitlrot   when others;
with sel select
overflow <= overflow1 when "000",
            overflow1 when "001",
            overflow1 when "010",
            '0'       when others ;
end Behavioral;
```

- Simulation, Test bench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;


entity simulation is
   --Port ( );
```

```
end simulation;


architecture Behavioral of simulation is
component alu_yit is
   Port ( num1    : in  std_logic_vector  (3 downto 0);
          num2    : in  std_logic_vector  (3 downto 0);
          sel     : in  std_logic_vector  (2 downto 0);
          func_o  : out std_logic_vector  (3 downto 0);
          overflow: out std_logic );
end component alu_yit;
signal num1 :std_logic_vector (3 downto 0):="0000";
signal num2 :std_logic_vector (3 downto 0):="0000";
signal sel  :std_logic_vector (2 downto 0):="000" ;
signal func_o :std_logic_vector (3 downto 0):="0000";
signal overflow :std_logic:='0' ;
begin
uut : alu_yit port map(num1 => num1, num2 => num2, sel =>
sel,func_o=>func_o,overflow=>overflow);
stim_proc : process
begin
num1 <= "0111";
num2 <= "0011";
sel<="000";
wait for 62.5ns;
num1 <= "1111";
```

```
num2 <= "0001";

sel<="000";

wait for 62.5ns;

num1 <= "0011";

num2 <= "0100";

sel<="000";

wait for 62.5ns;

num1 <= "0011";

num2 <= "0101";

sel<="001";

wait for 62.5ns;

num1 <= "1111";

num2 <= "0001";

sel<="001";

wait for 62.5ns;

num1 <= "0001";

num2 <= "1111";

sel<="001";

wait for 62.5ns;

num1 <= "0101";

num2 <= "0011";

sel<="001";

wait for 62.5ns;

num1 <= "0101";

num2 <= "0011";
```

```
sel<="010";
wait for 62.5ns;
num1 <= "0101";
num2 <= "1011";
sel<="011";
wait for 62.5ns;
num1 <= "0101";
num2 <= "1010";
sel<="011";
wait for 62.5ns;
num1 <= "0101";
num2 <= "0011";
sel<="100";
wait for 62.5ns;
num1 <= "0101";
num2 <= "0011";
sel<="101";
wait for 62.5ns;
num1 <= "0101";
num2 <= "0011";
sel<="110";
wait for 62.5ns;
num1 <= "1101";
num2 <= "0011";
sel<="110";
```

```
wait for 62.5ns;

num1 <= "1101";

num2 <= "0011";

sel<="111";

wait for 62.5ns;

num1 <= "0101";

num2 <= "0011";

sel<="111";

wait for 62.5ns;

end process;


end Behavioral;
```

- Constraint file

```
set_property PACKAGE_PIN V2 [get_ports {num2[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num2[0]}]

set_property PACKAGE_PIN T3 [get_ports {num2[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num2[1]}]

set_property PACKAGE_PIN T2 [get_ports {num2[2]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num2[2]}]

set_property PACKAGE_PIN R3 [get_ports {num2[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num2[3]}]

set_property PACKAGE_PIN W2 [get_ports {num1[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num1[0]}]

set_property PACKAGE_PIN U1 [get_ports {num1[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num1[1]}]
set_property PACKAGE_PIN T1 [get_ports {num1[2]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num1[2]}]

set_property PACKAGE_PIN R2 [get_ports {num1[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {num1[3]}]


set_property PACKAGE_PIN V17 [get_ports {sel[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {sel[0]}]

set_property PACKAGE_PIN V16 [get_ports {sel[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {sel[1]}]

set_property PACKAGE_PIN W16 [get_ports {sel[2]}]

set_property IOSTANDARD LVCMOS33 [get_ports {sel[2]}]


set_property PACKAGE_PIN P3 [get_ports {func_o[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {func_o[0]}]

set_property PACKAGE_PIN N3 [get_ports {func_o[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {func_o[1]}]

set_property PACKAGE_PIN P1 [get_ports {func_o[2]}]

set_property IOSTANDARD LVCMOS33 [get_ports {func_o[2]}]

set_property PACKAGE_PIN L1 [get_ports {func_o[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {func_o[3]}]


set_property PACKAGE_PIN U16 [get_ports {overflow}]

set_property IOSTANDARD LVCMOS33 [get_ports {overflow}]
```