MINISTERUL EDUCAȚIEI NAȚIONALE
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

# Digital signal filtering circuit

Structure of Computer Systems

**Horvath Andrea Anett**

Faculty of Automation and Computer Science
Computer Science Department
3'rd year, Group 30434

## *Contents*

# *I. Introduction*

a. Context &Objectives

The purpose of this project is to design, implement and test the digital signal filtering circuit. The circuit will be able to convert sequentially received signals into signals with different values based on a mathematical formula using constants and the previous signals.

This device can be used as a stand-alone project by people who want to process sequential signals or as a component in a more complex project where a filtering circuit is needed.

b. Specifications

The project will be implemented using VHDL and its functionality will be tested in the IDE provided by Vivado. Finally, the program will be run on a Basys 3 development board. The mathematical formula used to implement the filtering process is the next one:

$$Y(k) = X(k) * a1 + X(k-1) * a2 + X(k-2) * a3$$

where:

Y(k) - the output signal

X(k) - the last input signal

X(k-1), X(k-2) - the previous two input signals

a1, a2, a3 - constant values

Internally, the signals will be represented as 16 bit integers. The input will be received by setting on the switches the binary representation of the number and when the number is ready, a button will be pressed, the signal will be filtered and the output will be displayed on the seven segment displays. The system will also have a reset option, also mapped to a button, which will erase the content of the previously entered values.

## *II. Bibliographic study*

The circuit will implement the next mathematical formula, also presented in the previous chapter.

$$Y(k) = X(k) * a1 + X(k-1) * a2 + X(k-2) * a3$$

where:

Y(k) - the output signal

X(k) - the last input signal

X(k-1), X(k-2) - the previous two input signals

a1, a2, a3 - constant values

All the input signals will be represented internally as 8 bit binary numbers. The constants will be represented on 4 bit numbers t o make the computation easier. The input signals, i.e. X(...), will read their values from the development board, while the values of the constants will be declared in the program with no possibility of changing them from the board. The value of the output, i.e. Y(...), will be computed based on the formula.

It can be seen in the mathematical formula which performs the filtering process that the only two operations which will be used are the addition and the multiplication.

For the addition a generic *n* bit full adder, whose structure will be presented in a future chapter (Design).

For the multiplication, which is a little more complex operation, the chosen method for implementing it is the Shift-and-Add Multiplication technique described in the laboratory support documents. The algorithm is based on taking each digit of the multiplicand in turn and multiplying it by a single digit of the multiplier. Each intermediate product is placed in the appropriate positions, to the left of the earlier results. Finally, all the intermediate products are added together, to get the final result.

## *III. Analysis*

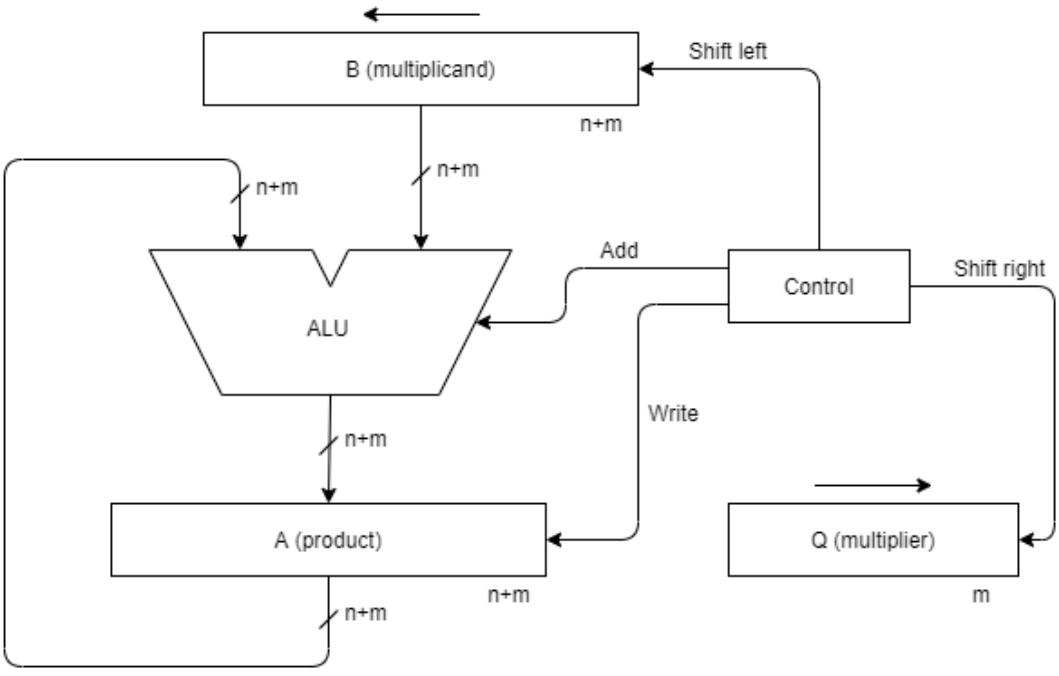
The operation worth analyzing which is performed in this project is the multiplication. The component which implements the multiplication is written as generic but for the explanation, numbers on 4 bits will be used.

From the multiple ways of computing the product of two numbers, the Shift-and-Add Multiplication technique was chosen. The method is similar to the multiplication performed on paper. This method adds the multiplicand X to itself Y times, where Y denotes the multiplier. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

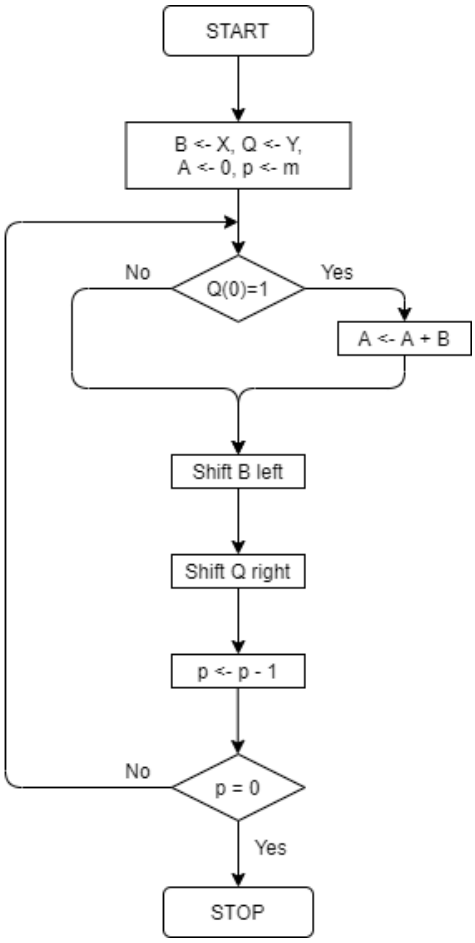
An example of multiplying two 4 bit numbers is the following:

```
  1001*
  1010
   0000
  1001
 0000
1010
1100010
```

The multiplier circuit, which implements the shift-and-add multiplication method for an n bit and an m bit number, is shown in figure below.



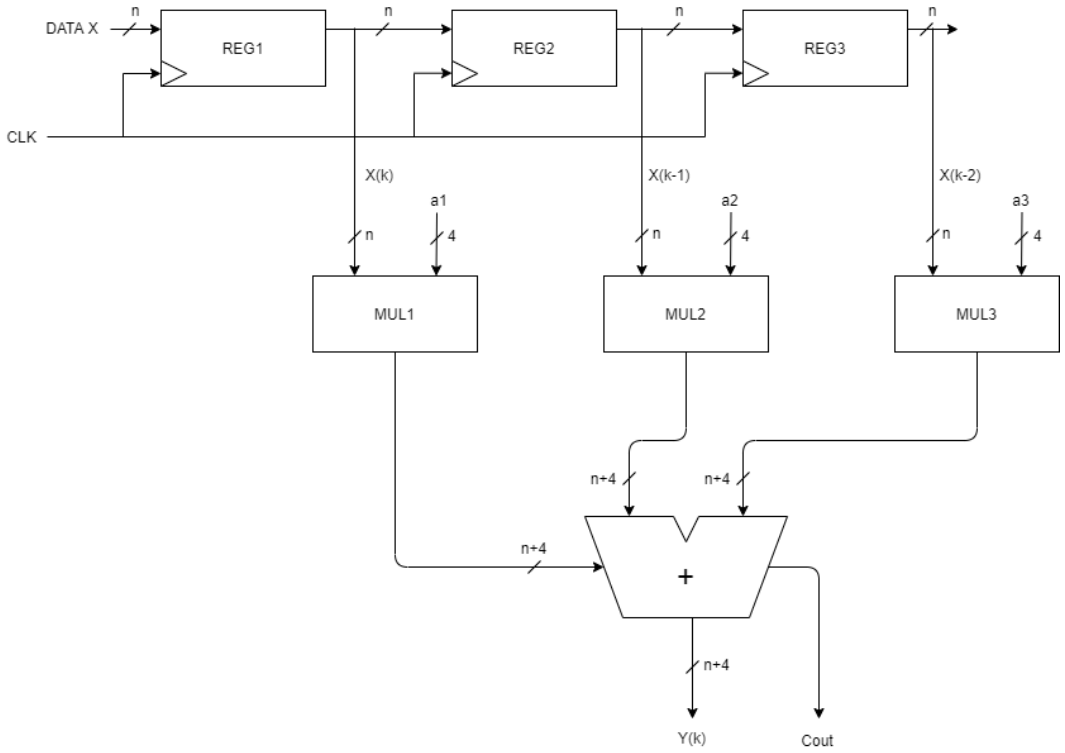
The n+m bit product register is initialized with 0. Since the basic algorithm shifts the multiplicand register left one position each step to align the multiplicand with the sum being accumulated in the product register, an n+m bit multiplicand register is used, placing in it the extended value of the multiplicand.




In the above figure the steps needed for the multiplication are sketched. The algorithm starts by loading the multiplicand into the B register, loading the multiplier into the Q register, and initializing the A register to 0. The counter p is initialized to m. The least significant bit of the multiplier register, i.e. Q(0) determines whether the multiplicand is added to the product register. The left shift of the multiplicand has the effect of shifting the intermediate products to the left, just as when multiplying by paper and pencil. The right shift of the multiplier prepares the next bit of the multiplier to examine in the following iteration.

## *IV. Design*

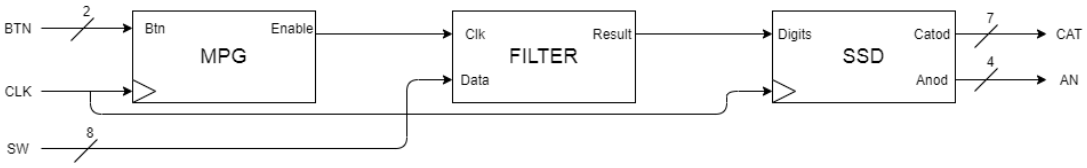
The overall structure of the system is visible in the next figure.



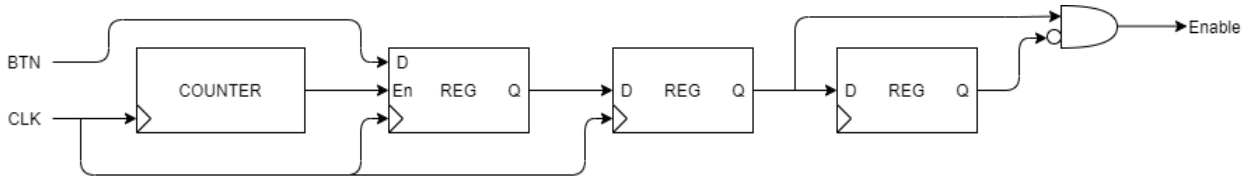

The circuit consists in three multipliers of shift-and-add type which were presented in the previous chapter, and adder and three data registers.

On the *DATA* input port the data flux enters sequentially on each clock cycle. In the three registers the last three values are stored and available for computing the result. The content of the registers is shifted right at each clock cycle from one register to the next one, after three clock periods the value is lost. Each of the last three entered numbers, i.e. *X(k)*, *X(k-1)*, *X(k-2)*, is multiplied with a constant, i.e. *a1*, *a2*, *a3* and the results are added together in the adder. The final result is the sum, i.e. *Y(k)* together with the carry out, i.e. *cout*.

What was presented until now represents strictly the computational part of the project, but its functionality will be tested on the development board so two more components were added in order to implement the reading of the input data and the display of the results. The two new components are a mono-pulse generator and the seven-segment display and the structure of the final project is visible in the next figure.



The MPG unit has the structure presented in the next figure. The registers together with the counter have the role of providing a delay necessary to de-bounce the buttons.



# *V. Implementation*

The system was designed in a structural manner, hence, the project will contain components implementing different functionalities: component for adder, component for multiplier and component for the whole system in which the previous two are mapped. Since the project will be tested on the development board, two additional components were needed: a mono-pulse generator and a seven-segment display.

a. Adder

```
entity adder is
    Generic (n: integer := 8);
    Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
           B : in STD_LOGIC_VECTOR (n-1 downto 0);
           C : in STD_LOGIC_VECTOR (n-1 downto 0);
           sum : out STD_LOGIC_VECTOR (n-1 downto 0);
           Cout : out STD_LOGIC);
end adder;

architecture Behavioral of adder is

signal result : STD_LOGIC_VECTOR (n downto 0);

begin

    process (A, B, C)
    begin
        result <= std_logic_vector(resize(signed(A + B + C), result'length));
    end process;

    Cout <= result (n);
    sum <= result (n-1 downto 0);

end Behavioral;
```

The component is written in a generic matter so it can be reused in case of future modification which will change the width of the data path.

b. Multiplier

```
entity multiplier is
    Generic (n: integer := 8;
             m : integer := 4);
    Port ( X : in STD_LOGIC_VECTOR (n-1 downto 0);
           Y : in STD_LOGIC_VECTOR (m-1 downto 0);
           result : out STD_LOGIC_VECTOR (n+m-1 downto 0));
end multiplier;

```

```
architecture Behavioral of multiplier is

begin
    process (X, Y)
    variable B : STD_LOGIC_VECTOR (n+m-1 downto 0);
    variable Q : STD_LOGIC_VECTOR (m-1 downto 0);
    variable A : STD_LOGIC_VECTOR (n+m-1 downto 0);
    variable p : integer;
    begin
        B := std_logic_vector(resize(signed(X), B'length));
        --Q := std_logic_vector(resize(signed(Y), Q'length));
        Q := Y;
        A := (others => '0');
        p := m;

        while (p /= 0) loop
            if (Q(0) = '1') then
                A := A + B;
            end if;
            B := B((B'length - 2) downto 0) & '0';
            Q := '0' & Q(m-1 downto 1);
            p := p - 1;
        end loop;
        result <= A;
    end process;

end Behavioral;
```

The multiplier is implemented according to the structure described in the Analysis chapter and it is also written in a generic way also to make a future change of data path width easier.

c. Filter

```
entity filter is
    Generic ( n : integer := 8);
    Port (data : in STD_LOGIC_VECTOR (n-1 downto 0);
          reset : in STD_LOGIC;
          result : out STD_LOGIC_VECTOR (n+3 downto 0); -- the length of the
          --result should be changed according to the formua n+m-1 where m
          --is the length of the constants
          cout : out STD_LOGIC;
          clk : in STD_LOGIC);
end filter;

architecture Behavioral of filter is

component multiplier is
    Generic (n: integer := 8;
             m : integer := 4);
    Port ( X : in STD_LOGIC_VECTOR (n-1 downto 0);
           Y : in STD_LOGIC_VECTOR (m-1 downto 0);
           result : out STD_LOGIC_VECTOR (n+m-1 downto 0));
end component multiplier;

component adder is
    Generic (n: integer := 8);
    Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
           B : in STD_LOGIC_VECTOR (n-1 downto 0);
           C : in STD_LOGIC_VECTOR (n-1 downto 0);
           sum : out STD_LOGIC_VECTOR (n-1 downto 0);
           Cout : out STD_LOGIC);
end component adder;


--the constants can be changed
constant a1 : STD_LOGIC_VECTOR (3 downto 0) := "0010"; --2
constant a2 : STD_LOGIC_VECTOR (3 downto 0) := "0011"; --3
constant a3 : STD_LOGIC_VECTOR (3 downto 0) := "0100"; --4
```

```
68
69  signal reg1 : STD_LOGIC_VECTOR (n-1 downto 0) := (others => '0');
70  signal reg2 : STD_LOGIC_VECTOR (n-1 downto 0) := (others => '0');
71  signal reg3 : STD_LOGIC_VECTOR (n-1 downto 0) := (others => '0');
72
73  signal mul1 : STD_LOGIC_VECTOR (n + a1'length - 1 downto 0);
74  signal mul2 : STD_LOGIC_VECTOR (n + a1'length - 1 downto 0);
75  signal mul3 : STD_LOGIC_VECTOR (n + a1'length - 1 downto 0);
76
77  begin
78
79      process (clk, reset)
80      begin
81          if (reset = '1') then
82              reg1 <= (others => '0');
83              reg2 <= (others => '0');
84              reg3 <= (others => '0');
85          elsif (rising_edge(clk)) then
86              reg1 <= data;
87              reg2 <= reg1;
88              reg3 <= reg2;
89          end if;
90      end process;
91
92      Multiplication1 : multiplier generic map (n, a1'length) port map (reg1, a1, mul1);
93      Multiplication2 : multiplier generic map (n, a2'length) port map (reg2, a2, mul2);
94      Multiplication3 : multiplier generic map (n, a3'length) port map (reg3, a3, mul3);
95
96      --in the generic mapping, the value should be n+m where m is the length of the constants
97      Addition : adder generic map (n+4) port map (mul1, mul2, mul3, result, cout);
98
99  end Behavioral;
```

The filter maps the adder and the multiplier according to the figure in the Design chapter and implements the process in which the data is shifted from one register to another. Also the reset functionality is implemented in this component.

d. Mono-pulse generator

```
36  entity mpg is
37      Port ( btn : in STD_LOGIC_VECTOR (4 downto 0);
38             clk : in STD_LOGIC;
39             step : out STD_LOGIC_VECTOR (4 downto 0));
40  end mpg;
41
42  architecture Behavioral of mpg is
43
44  signal q1 : std_logic_vector(4 downto 0);
45  signal q2 : std_logic_vector(4 downto 0);
46  signal cnt : std_logic_vector(15 downto 0);
47
48  begin
49
50      -- counter
51      process(clk)
52      begin
53          if rising_edge(clk) then
54              cnt <= cnt + 1;
55          end if;
56      end process;
57
```

```
    process(clk)
    begin
        if rising_edge(clk) then
            if cnt = x"FFFF" then
                q1 <= btn;
            end if;
        end if;
    end process;

    process(clk)
    begin
        if rising_edge(clk) then
            q2 <= q1;
        end if;
    end process;

    step <= q1 and not q2;

end Behavioral;
```

e. Seven-segment display

```
entity ssd is
    Port ( digit0 : in STD_LOGIC_VECTOR (3 downto 0);
           digit1 : in STD_LOGIC_VECTOR (3 downto 0);
           digit2 : in STD_LOGIC_VECTOR (3 downto 0);
           digit3 : in STD_LOGIC_VECTOR (3 downto 0);
           clk : in STD_LOGIC;
           cat : out STD_LOGIC_VECTOR (6 downto 0);
           an : out STD_LOGIC_VECTOR (3 downto 0));
end ssd;

architecture Behavioral of ssd is

signal cnt : std_logic_vector (15 downto 0);
signal digit : std_logic_vector (3 downto 0);

begin

    process (clk)
    begin
        if (rising_edge(clk)) then
            cnt <= cnt + 1;
        end if;
    end process;


    process (cnt(15 downto 14), digit0, digit1, digit2, digit3)
    begin
        case cnt(15 downto 14) is
            when "00" => digit <= digit0;
            when "01" => digit <= digit1;
            when "10" => digit <= digit2;
            when others => digit <= digit3;
        end case;
    end process;

    process (cnt(15 downto 14))
    begin
        case cnt(15 downto 14) is
            when "00" => an <= "1110";
            when "01" => an <= "1101";
            when "10" => an <= "1011";
            when others => an <= "0111";
        end case;
    end process;

```

```
    with digit select cat <=
          "1111001" when "0001",   --1
          "0100100" when "0010",   --2
          "0110000" when "0011",   --3
          "0011001" when "0100",   --4
          "0010010" when "0101",   --5
          "0000010" when "0110",   --6
          "1111000" when "0111",   --7
          "0000000" when "1000",   --8
          "0010000" when "1001",   --9
          "0001000" when "1010",   --A
          "0000011" when "1011",   --b
          "1000110" when "1100",   --C
          "0100001" when "1101",   --d
          "0000110" when "1110",   --E
          "0001110" when "1111",   --F
          "1000000" when others;   --0

end Behavioral;
```

f. Display

```
entity display is
    Port ( internal_clk : in STD_LOGIC;
           btn : in STD_LOGIC_VECTOR (4 downto 0);
           sw : in STD_LOGIC_VECTOR (15 downto 0);
           led : out STD_LOGIC_VECTOR (15 downto 0);
           an : out STD_LOGIC_VECTOR (3 downto 0);
           cat : out STD_LOGIC_VECTOR (6 downto 0));
end display;

architecture Behavioral of display is

component mpg is
Port ( btn : in STD_LOGIC_VECTOR (4 downto 0);
       clk : in STD_LOGIC;
       step : out STD_LOGIC_VECTOR (4 downto 0));
end component;

component ssd is
Port ( digit0 : in STD_LOGIC_VECTOR (3 downto 0);
       digit1 : in STD_LOGIC_VECTOR (3 downto 0);
       digit2 : in STD_LOGIC_VECTOR (3 downto 0);
       digit3 : in STD_LOGIC_VECTOR (3 downto 0);
       clk : in STD_LOGIC;
       cat : out STD_LOGIC_VECTOR (6 downto 0);
       an : out STD_LOGIC_VECTOR (3 downto 0));
end component;


component filter is
    Generic ( n : integer := 8);
    Port (data : in STD_LOGIC_VECTOR (n-1 downto 0);
          reset : in STD_LOGIC;
          result : out STD_LOGIC_VECTOR (n+3 downto 0);
          cout : out STD_LOGIC;
          clk : in STD_LOGIC);
end component filter;

signal step : std_logic_vector(4 downto 0);
signal digit : std_logic_vector(15 downto 0);

signal clk : std_logic;

signal data : STD_LOGIC_VECTOR (7 downto 0);
signal reset : STD_LOGIC;
signal result : STD_LOGIC_VECTOR (11 downto 0);
signal cout : STD_LOGIC;
```

```
79
80  begin
81
82      MPGU: mpg port map(btn, internal_clk, step);
83      SSDU: ssd port map(digit(3 downto 0), digit(7 downto 4), digit(11 downto 8), digit(15 downto 12), internal_clk, cat, an);
84      FILTER_UNIT : filter generic map (8) port map (data, reset, result, cout, clk);
85
86      reset <= step(0);
87      clk <= step(1);
88
89      data <= sw(7 downto 0);
90
91      digit(11 downto 0) <= result;
92      led(0) <= cout;
93
94
95  end Behavioral;
```

This component represents the top module, it maps the MPG, SSD and the filter together. Here the buttons, switches, seven-segment displays and a led are also assigned. Two buttons are used: one for the reset and one for the enable signal. For entering the data 8 switches are used, the width of the data path was chosen to be 8 bits for testing. The results are displayed on the seven-segment displays and the carry out is assigned to a led.

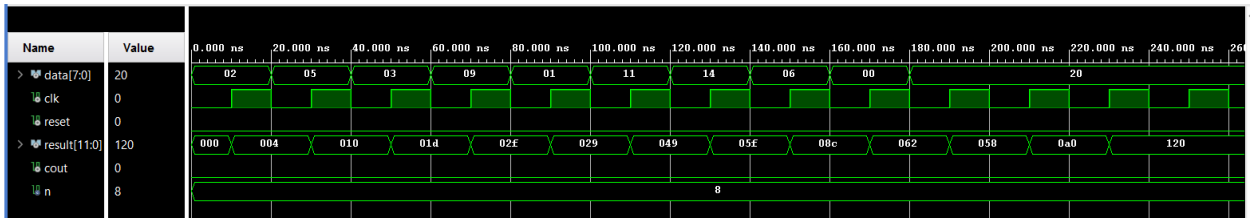
## *VI. Testing and Validation*

The testing was performed in two ways. The functionality was firstly tested in the simulator using a test bench.

```
34  entity test_bench is
35  --  Port ( );
36  end test_bench;
37
38  architecture Behavioral of test_bench is
39
40  component filter is
41      Generic ( n : integer := 8);
42      Port (data : in STD_LOGIC_VECTOR (n-1 downto 0);
43            reset : in STD_LOGIC;
44            result : out STD_LOGIC_VECTOR (n+3 downto 0);
45            cout : out STD_LOGIC;
46            clk : in STD_LOGIC);
47  end component filter;
48
49  constant n : integer := 8;
50  signal data : STD_LOGIC_VECTOR (7 downto 0);
51  signal clk : STD_LOGIC;
52  signal reset : STD_LOGIC;
53  signal result : STD_LOGIC_VECTOR (11 downto 0);
54  signal cout : STD_LOGIC;
55
56  begin
57      uut : filter generic map (8) port map (data, reset, result, cout, clk);
58      process
59      begin
60          clk <= '0';
61          wait for 10 ns;
62          clk <= '1';
63          wait for 10 ns;
64      end process;
65      process

66      begin
67          reset <= '0';
68          data <= x"02";
69          wait for 20 ns;
70          reset <= '0';
71          data <= x"05";
72          wait for 20 ns;
73          reset <= '0';
74          data <= x"03";
75          wait for 20 ns;
76          reset <= '0';
77          data <= x"09";
78          wait for 20 ns;
79          reset <= '0';
80          data <= x"01";
81          wait for 20 ns;
82          reset <= '0';
83          data <= x"11";
84          wait for 20 ns;
85          reset <= '0';
86          data <= x"14";
87          wait for 20 ns;
88          reset <= '0';
89          data <= x"06";
90          wait for 20 ns;
```

```
91          reset <= '0';
92          data <= x"00";
93          wait for 20 ns;
94          reset <= '0';
95          data <= x"20";
96          wait for 20 ns;
97          wait;
98      end process;
99
100 end Behavioral;
```




The formula after which the results are computed is: Y(k) = X(k) * a1 + X(k-1) * a2 + X(k-2) * a3.

a1 = 2;  a2 = 3;  a3 = 4  -> constants

Tracing the algorithm we obtain the following results (in hexadecimal):

Step 1:

X(k) = 2; X(k-1) = 0; X(k-2) = 0

Y(k) = 2 * 2 + 0 * 3 + 0 * 4 = 4

Step 2:

X(k) = 5; X(k-1) = 2; X(k-2) = 0

Y(k) = 5 * 2 + 2 * 3 + 0 * 4 = 10

Step 3:

X(k) = 3; X(k-1) = 5; X(k-2) = 2

Y(k) = 3 * 2 + 5 * 3 + 2 * 4 = 1d

Step 4:

X(k) = 9; X(k-1) = 3; X(k-2) = 5

Y(k) = 9 * 2 + 3 * 3 + 5 * 4 = 2f

Step 5:

X(k) = 1; X(k-1) = 9; X(k-2) = 3

Y(k) = 1 * 2 + 9 * 3 + 3 * 4 = 29

Step 6:

X(k) = 11; X(k-1) = 1; X(k-2) = 9

Y(k) = 11 * 2 + 1 * 3 + 9 * 4 = 49

Step 7:

X(k) = 14; X(k-1) = 11; X(k-2) = 1

Y(k) = 14 * 2 + 11 * 3 + 1 * 4 = 5f

Step 8:

X(k) = 6; X(k-1) = 14; X(k-2) = 11

Y(k) = 6 * 2 + 14 * 3 + 11 * 4 = 8c

Step 9:

X(k) = 0; X(k-1) = 6; X(k-2) = 14

Y(k) = 0 * 2 + 6 * 3 + 14 * 4 = 62

Step 10:

X(k) = 20; X(k-1) = 0; X(k-2) = 6

Y(k) = 20 * 2 + 0 * 3 + 6 * 4 = 58

Step 11:

X(k) = 20; X(k-1) = 20; X(k-2) = 0

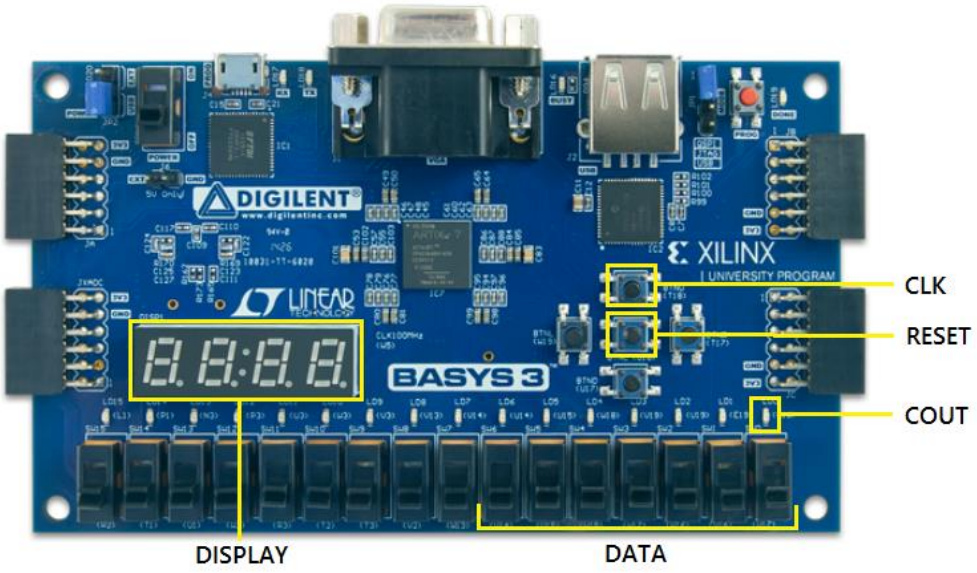
Y(k) = 20 * 2 + 20 * 3 + 0 * 4 = a0

Step 12:

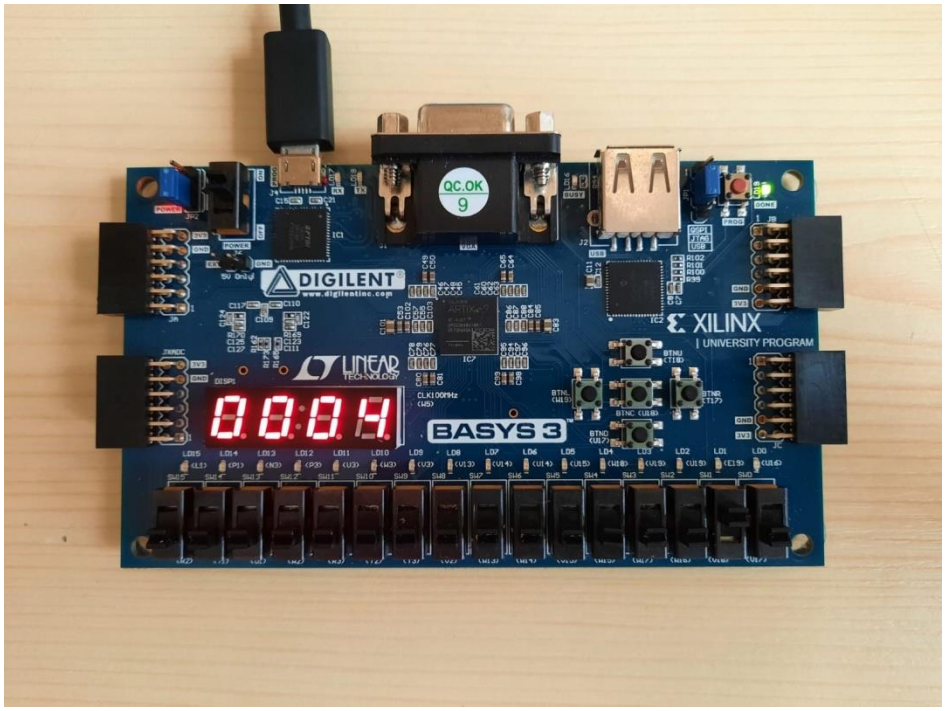
X(k) = 20; X(k-1) = 20; X(k-2) = 20

Y(k) = 20 * 2 + 20 * 3 + 20 * 4 = 288

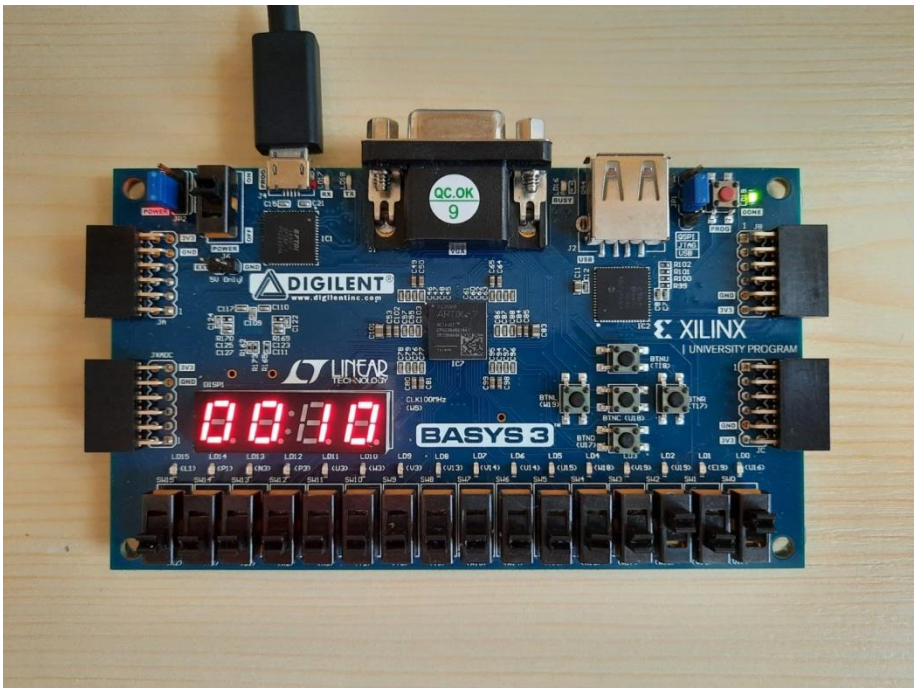
For testing the project with the board the same values as above will be used. The mapping of the buttons and switches is shown in the picture below.



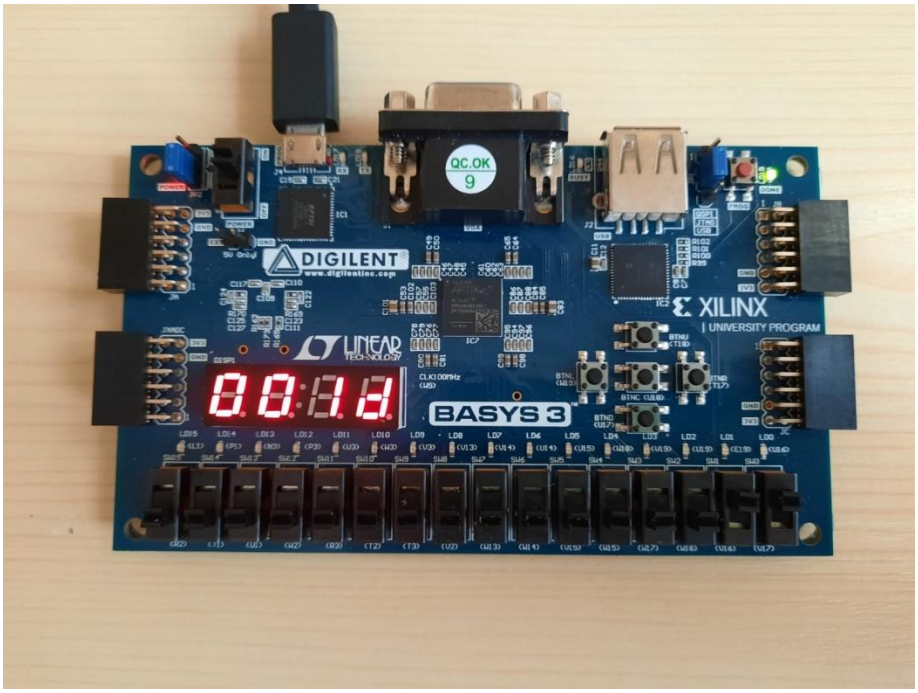

Step 1



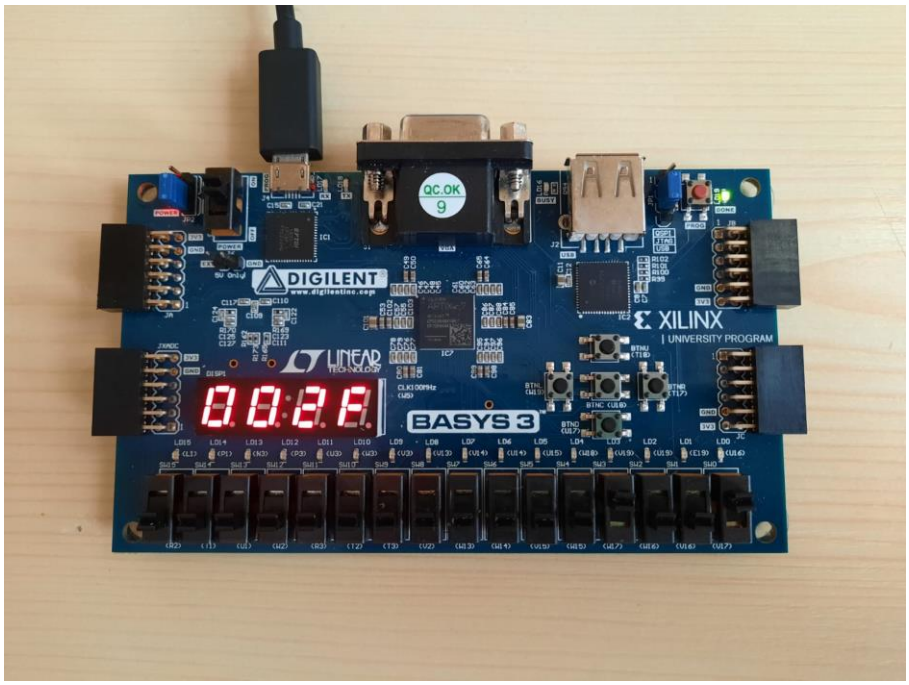
Step 2


QC.OK
9
DIGILENT
www.digilentinc.com
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

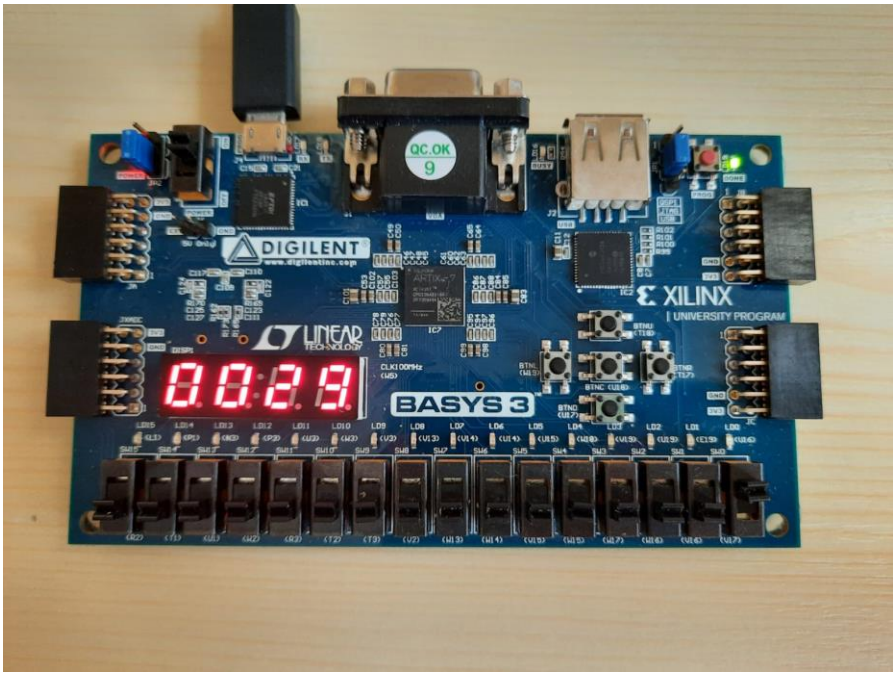
Step 3


QC.OK
9
DIGILENT
www.digilentinc.com
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

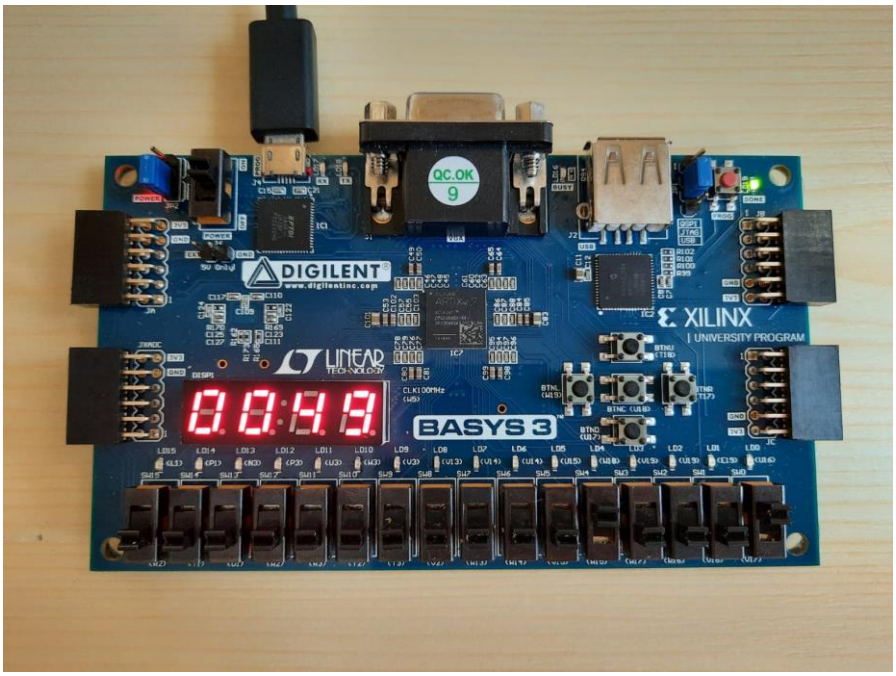
Step 4


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

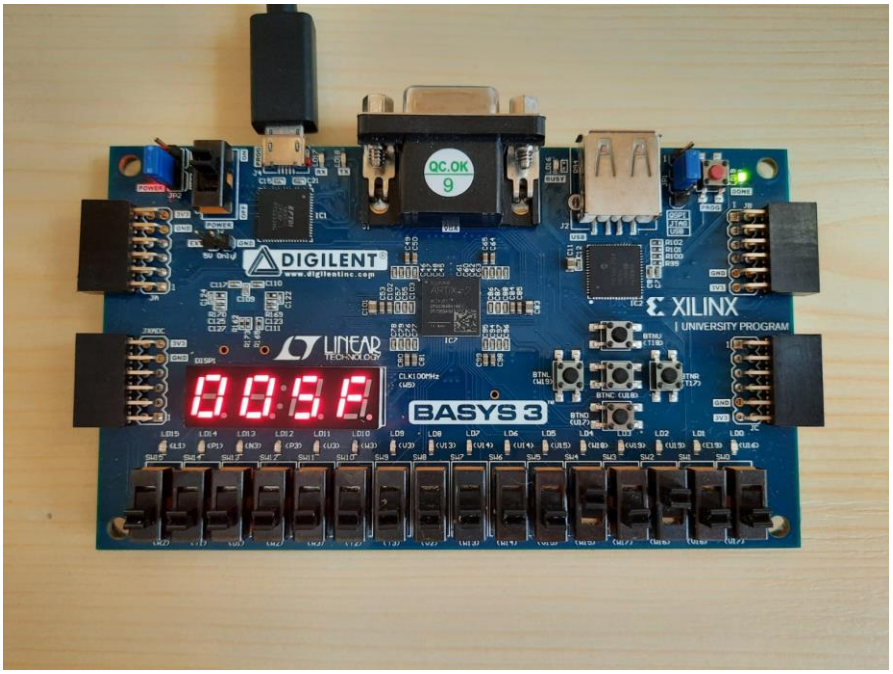
Step 5


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

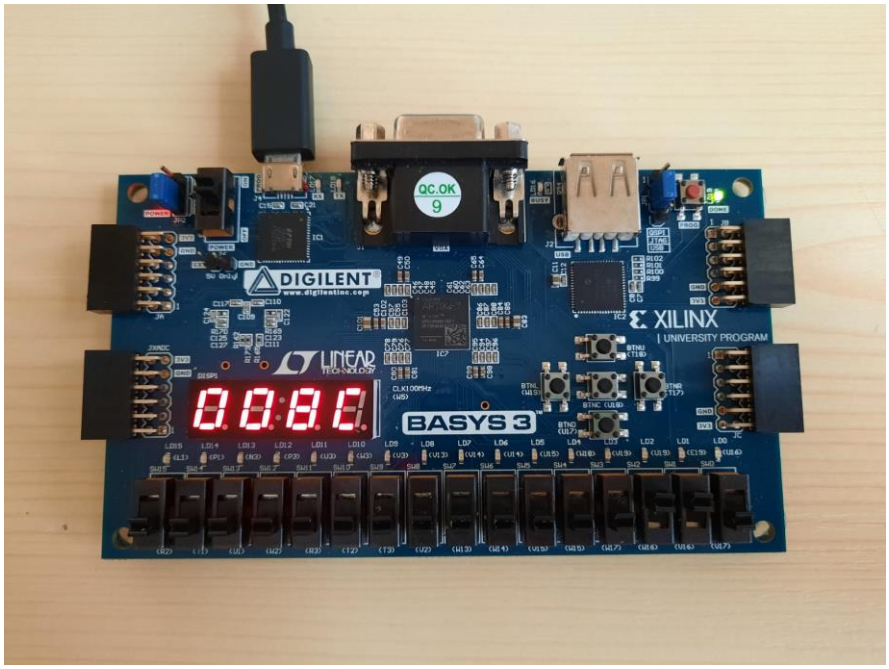
Step 6


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

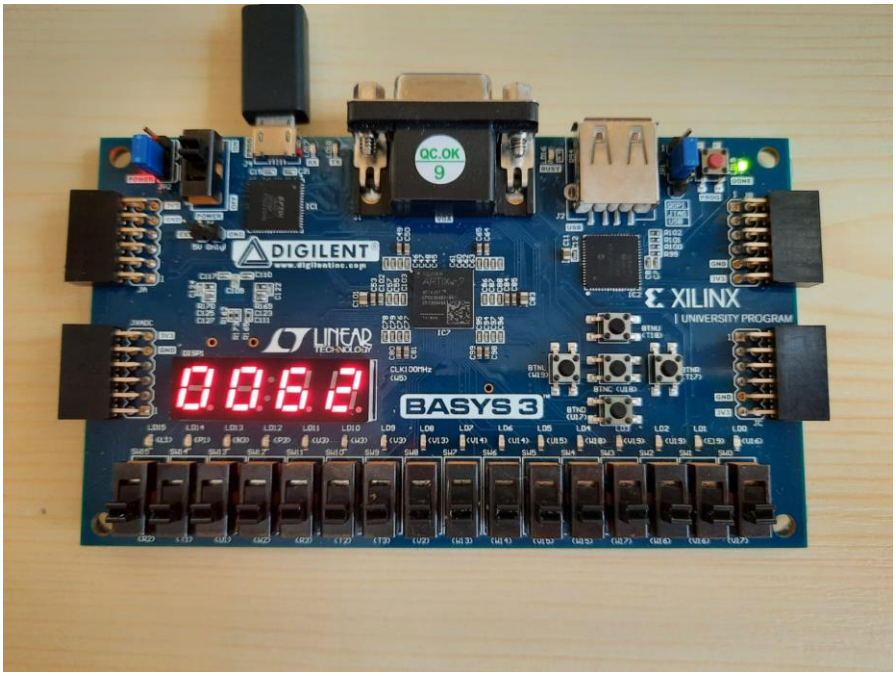
Step 7


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

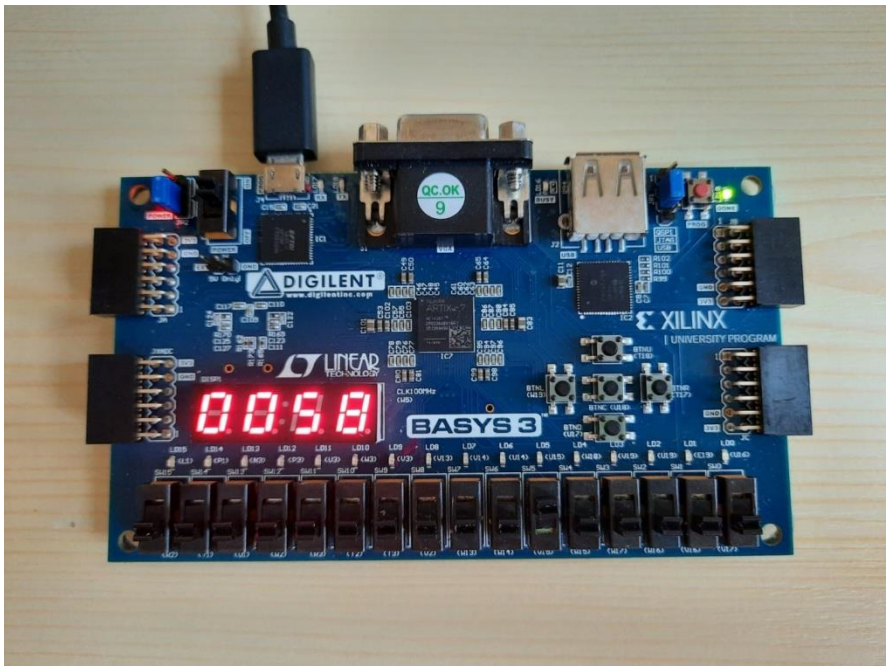
Step 8


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

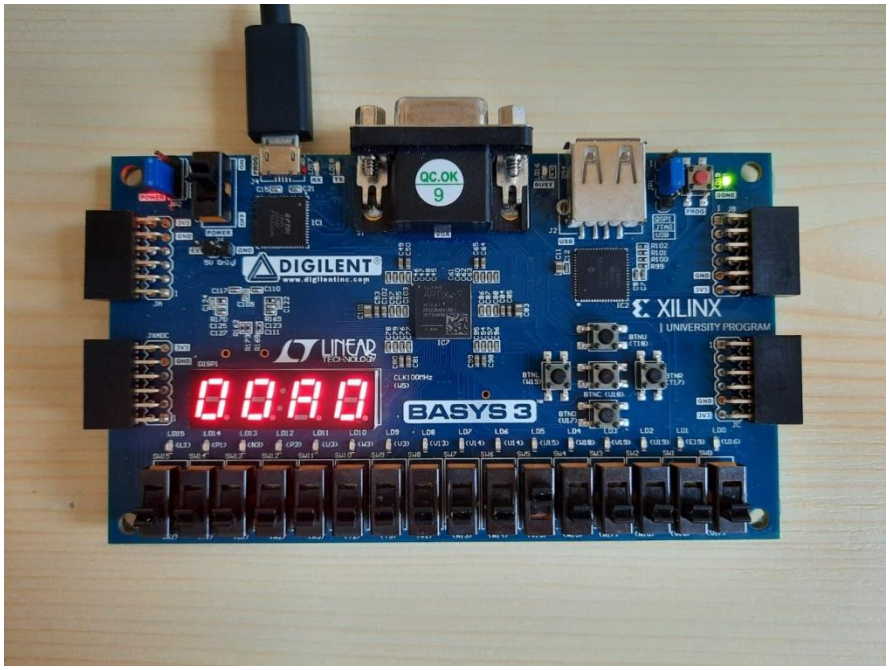
Step 9


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

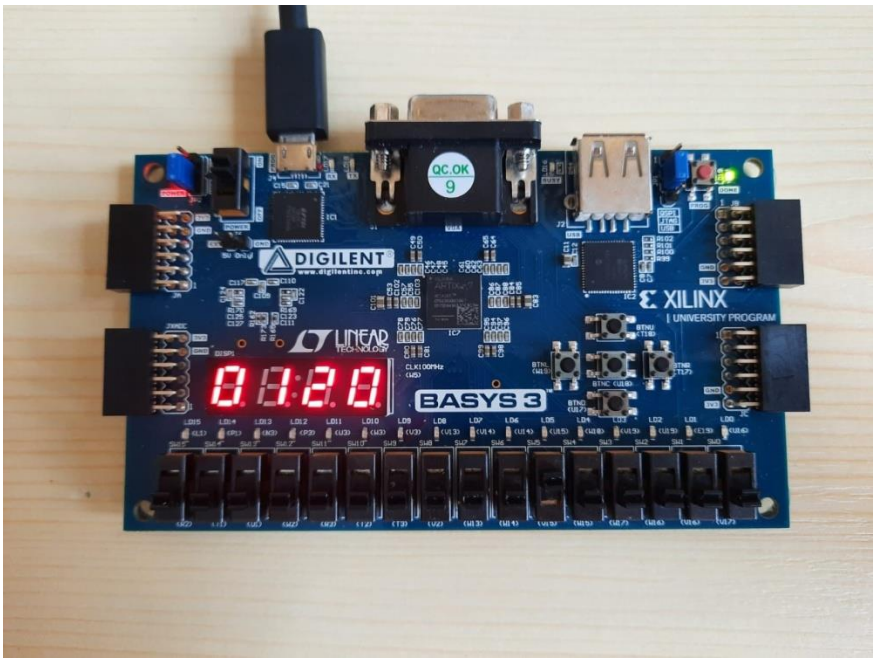
Step 10




Step 11



Step 12


QC.OK
9
DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

## *VII. Conclusions*

In conclusion, the project works according to our expectations, the testing and validation stage proved that. We obtained in the end a fully functional digital signal filtering circuit which can be used as a stand-alone project or as a component in a more complex project.

As a future improvement, extending the width of the data path can be considered. The code can be adapted easily since most of the components were designed in a generic manner. Unfortunately this can be done only using a different development board which has more seven-segment displays because the results will be considerably bigger so the display would be impossible on a Basys 3 board. Another modification which can be done is modifying the mathematical formula after which the input data is filtered.

## *VIII. Bibliography*

1. http://users.utcluj.ro/~negrum/index.php/computer-architecture/

2. https://www.ques10.com/p/19357/draw-and-explain-the-flowchart-of-add-and-shift-me/

3. Laboratory works on moodle: https://moodle.cs.utcluj.ro/course/view.php?id=321