Digital signal filtering circuit

Student: Cristian Casian-Cristi

Group: 30432

Technical University of Cluj-Napoca

Table of contents:

1. Introduction

1.1 Context

A circuit will be designed (Xilinx/VHDL) which transform a sequence of values of inputs X(i) based on a formula.

1.2 Objectives

The project will use VHDL as a programming language where I will do the entire design of the project. The result will be display on Basys3 board, which can be reprogrammed to desired applications or functionality requirements after manufacturing.

The data will be sent from the file. From the board I can select the filter from switches and then I should filter the data and send them on the computer.

I have chosen to use unsigned integers for my project, with 8-bit inputs and 16-bit outputs, based on my research, those choices simplify working with the data representation.

2. Bibliography study

2.1 Digital filter

In signal processing the function of a filter is to remove unwanted parts of the signal, such as random noise, or to extract useful parts of the signal, such as the components lying within a certain frequency range.

A digital filter uses a digital processor to perform numerical calculations on sampled values of the signal. The processor may be a general-purpose computer such as PC, or a specialised DSP (Digital Signal Processor) chip.

The analog input signal must first be sampled and digitised using an ADC (Analog Digital Converter). The resulting binary numbers, representing successive sampled values of the input signal, are transferred to the processor, which carries out numerical calculations on them. These calculations typically involve multiplying the input values by constants and adding the product together. If necessary, the result of these calculations, which now represent sampled values of the filtered signal, are output thought a DAC (Digital to Analog Converter) to convert the signal back to analog form.

Note that in a digital filter, the signal is represented by a sequence of numbers, rather than a voltage or current.

The advantages of using digital filter are: is programmable, its operation is determined by a program stored in the processor’s memory. This means the digital filter can easily be changed without affecting the circuitry. Digital filters are easily designed, tested, and implemented on a general-purpose computer or workstation.

2.2 Characterization

A digital filter is characterized by its transfer function, or equivalently, its difference equation. Mathematical analysis of the transfer function can describe how it will respond to any input. As such, designing a filter consists of developing specifications appropriate to the problem (for example, a second-order low pass filter with a specific cut-off frequency), and then producing a transfer function which meets the specifications.

This is the form for recursive filters, which typically leads to an infinite impulse response (IIR) behaviour, but if the denominator is made equal to unity, i.e. no feedback, then this becomes a finite impulse response (FIR) filter. There exists a lot of types of digital filters, for example: casual filter, time-invariant filter, stable filter, finite impulse response, infinite impulse response, linear filter.

2.3 FIR filters

In signal processing, a finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length) is of finite duration, because it settles to zero in finite time. This is contrast to infinite impulse response (IIR) filters, which may have interval feedback and may continue to respond indefinitely (usually decaying).

This impulse response (that is, the output in response to a Kroncker delta input) of an N^{th}- order discrete-time FIR filter lasts exactly N + 1 samples (from first nonzero element through last nonzero element) before it then settles to zero.

FIR filters can be discrete-time or continuous-time and digital or analog.

2.4 IIR filters

IIR is a property applying to many linear time-invariant systems that are distinguished by having an impulse response h(t) which does not become exactly zero past a certain point, but continues indefinitely. This is in contrast to a finite impulse response system in which the impulse response *does* become exactly zero at times t for some finite T, thus being of finite duration. Common examples of linear time-invariant systems are most electronic and digital filters. Systems with this property are known as *IIR systems* or *IIR filters*.

In practice, the impulse response, even of IIR systems, usually approaches zero and can be neglected past a certain point. However, the physical systems which give rise to IIR or FIR responses are dissimilar, and therein lies the importance of the distinction. For instance, analog electronic filters composed of resistors, capacitors, and/or inductors (and perhaps linear amplifiers) are generally IIR filters. On the other hand, discrete-time filters (usually digital filters) based on a tapped delay line employing no feedback are necessarily FIR filters. The capacitors (or inductors) in the analog filter have a "memory" and their internal state never completely relaxes following an impulse (assuming the classical model of capacitors and inductors where quantum effects are ignored). But in the latter case, after an impulse has reached the end of the tapped delay line, the system has no further memory of that impulse and has returned to its initial state; its impulse response beyond that point is exactly zero.

3. Analysis

First of all, we should make a method how to take the inputs. The inputs should be in a file and there with a buffer we take the data from there. Filter will work with those and after with another buffer we transmit the results the board.

After making research about many filters, I will implement 3 types of those: a custom one, Low-pass filter and high-pass filter. The block diagram for FIR filters is:

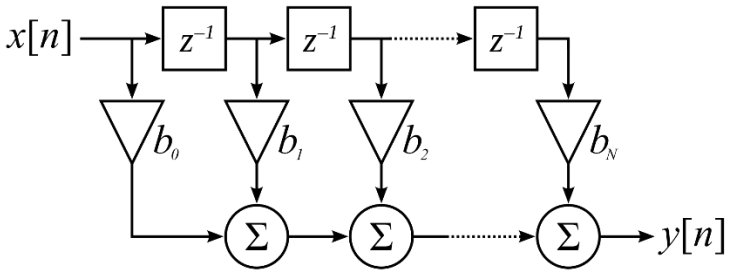

Fig.1: FIR filter

For a casual discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values: y[b]=b0*x[n] + b1*x[n-1] + … + bN*x[n-N] $=\sum_{i=0}^{N} bi * x[n-i]$, where: x[n] is the input signal y[n] is the output signal

N is the filter order; an Nth-order filter has N+1 terms on the right-hand side bi is the value of the impulse response at the ith instant for 0<=i<=N of an Nth-order FIR filter. If the filter is a direct form FIR filter then bi is also a coefficient of the filter.

This computation is also known as discrete convolution.

The x[n-1] in these terms are commonly referred to as *taps*, based on the structure of a tapped delay line that in many implementations or block diagrams provides the delayed inputs to the multiplication operations. One may speak of a 5th order/6-tap filter, for instance.

I will use UART to stream the data between the transmitter and the receiver. The UART transmits the least significant bit(LSB) of tx_data first. In the transaction shown, an even parity bit follows the data word. Once the logic high stop bit is sent and the transmission is complete, the tx_busy signal deasserts, indicating that the transmit circuitry is ready for the next transaction.

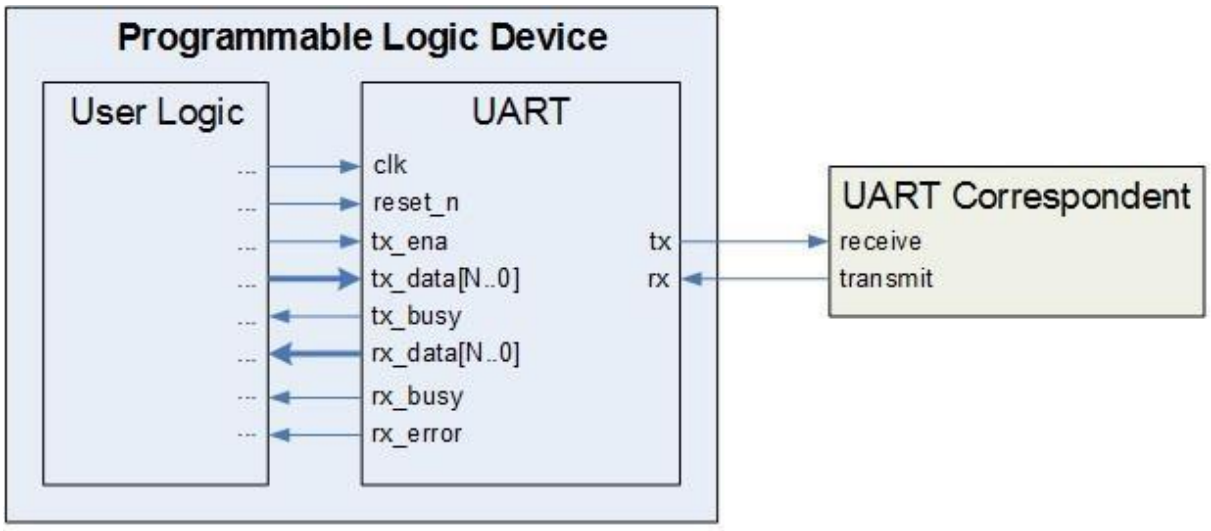

Fig. 2: UART

Fig. 3: Waveform UART

A low-pass filter(LPS) is a circuit that only passes signals below its cutoff frequency while attenuating all signals above it.

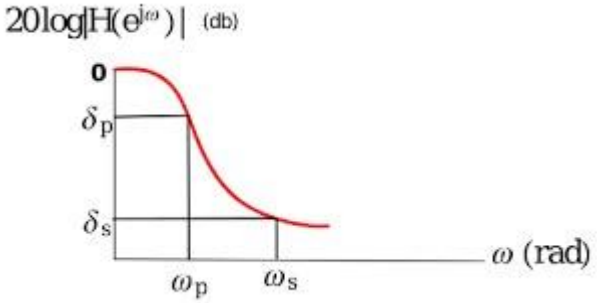

Fig. 4: Low-pass filter

ɯs is normalized cut-off frequency in the stopband

ɯp is normalized cut-off frequency in the passband

δs-minimum attenuation in the stopband δp-maximum

ripples in the passband

A high-pass filter is an EQ curve that is used to remove low-frequency sounds from an audio signal. It is called a high-pass filter because it allows high-frequency signals to pass through, while attenuating (reducing the amplitude of) lower-frequency signals.

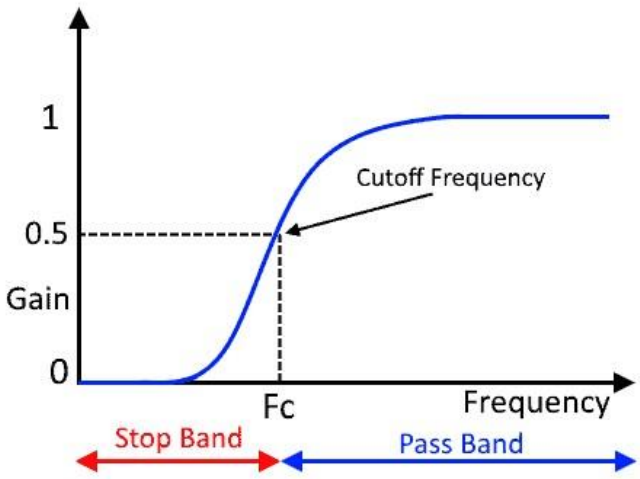

Fig. 5: High-pass filter

4. Design

First filter is the interval from which the inputs should be: I will choose the interval to be [5;30] and all the numbers which are between this interval will be displayed on the FPGA board.

Second filter, which is low-pass filter I will use this formula: y(k)=a*x(k)+b*x(k-1) +c*x(k-2).

Third filter, which is high-pass filter I will use this formula: y(k) = (a+b)*x(k) + c*x(k-2) + d*y(k-1).

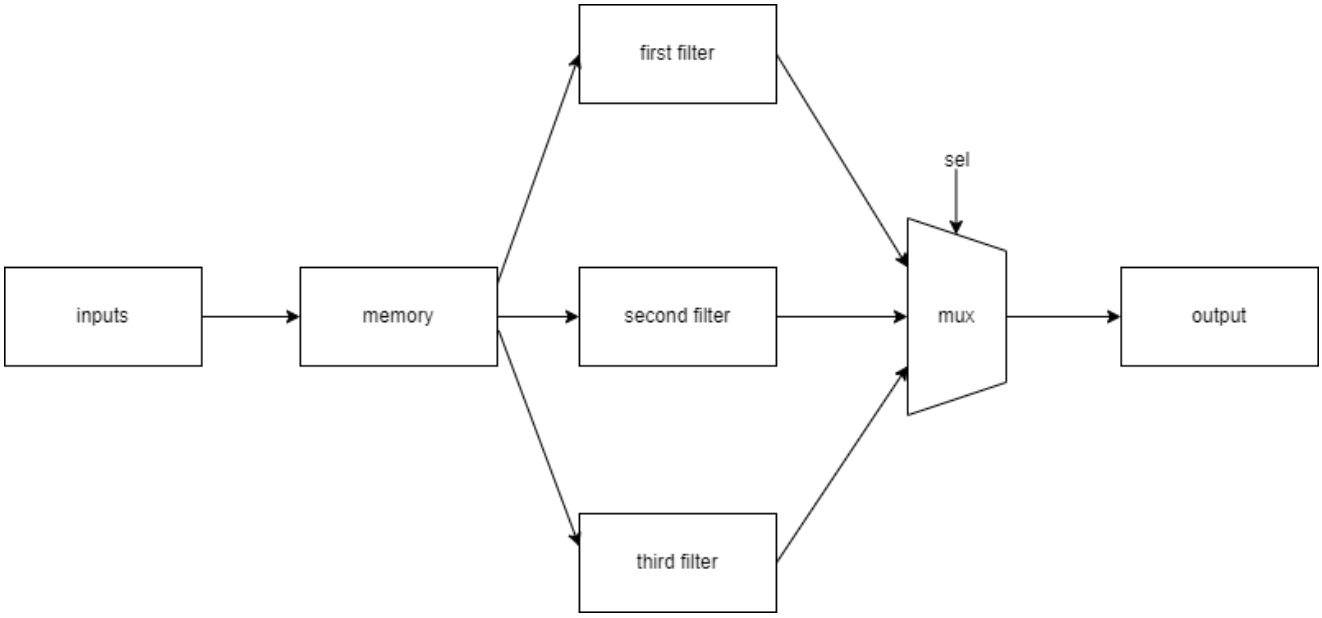

Fig. 6: Black box of the project

4.1 First filter component:

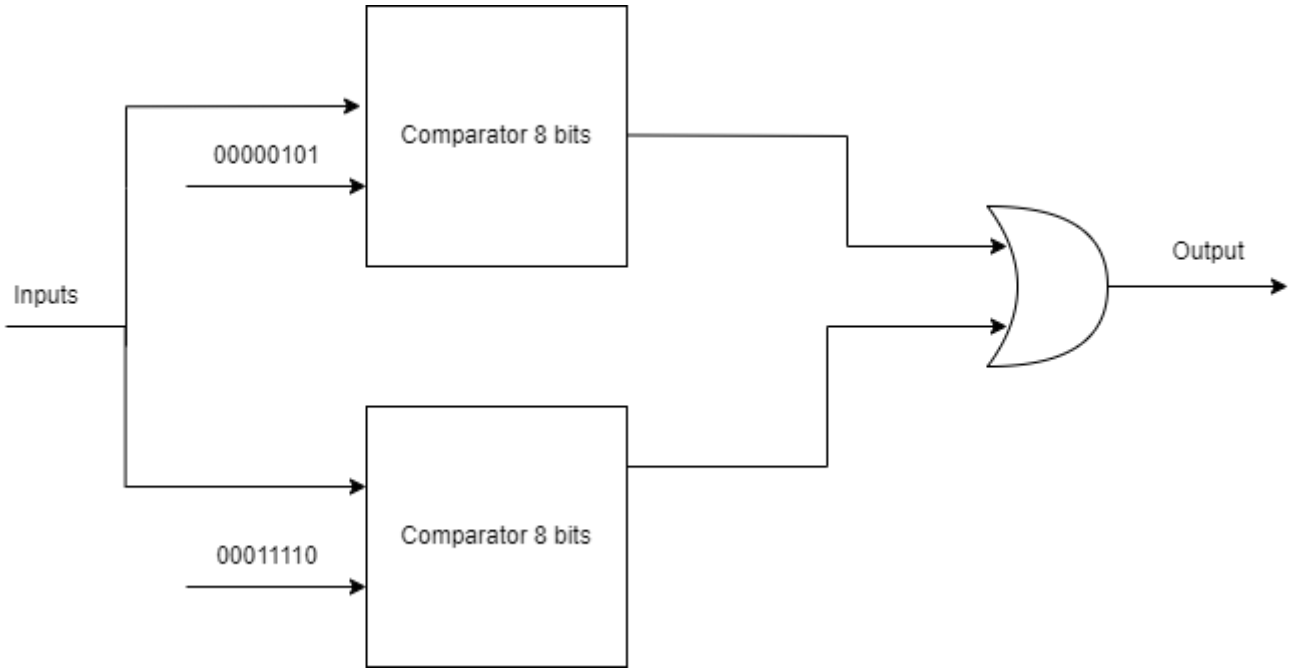

Fig. 7: Black box of the first filter

As I said before, the first filter is an interval between the inputs should be: 5 (00000101 – representation on 8-bits) and 30 (00011110 – representation on 8-bits). If the inputs are between these two numbers, they will be displayed on the FPGA board on 16 bits. To check if the inputs are right, I will use two comparators on 8-bits and their outputs on the branch A<B will be connected to a OR gate, having the final output. If this output is 1 that means the input is not in that interval, so we will ignore it. If the output of the OR gate is 0 that means the input is between 5 and 30, so I will display the number on the FPGA board.

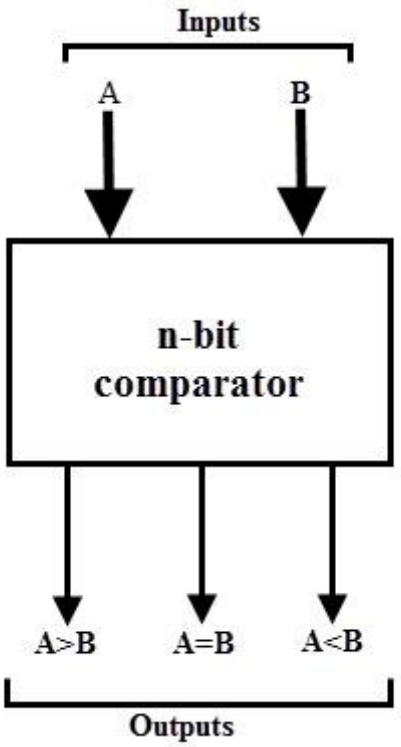

Fig. 8: Comparator on n-bits

4.2 Second filter component:

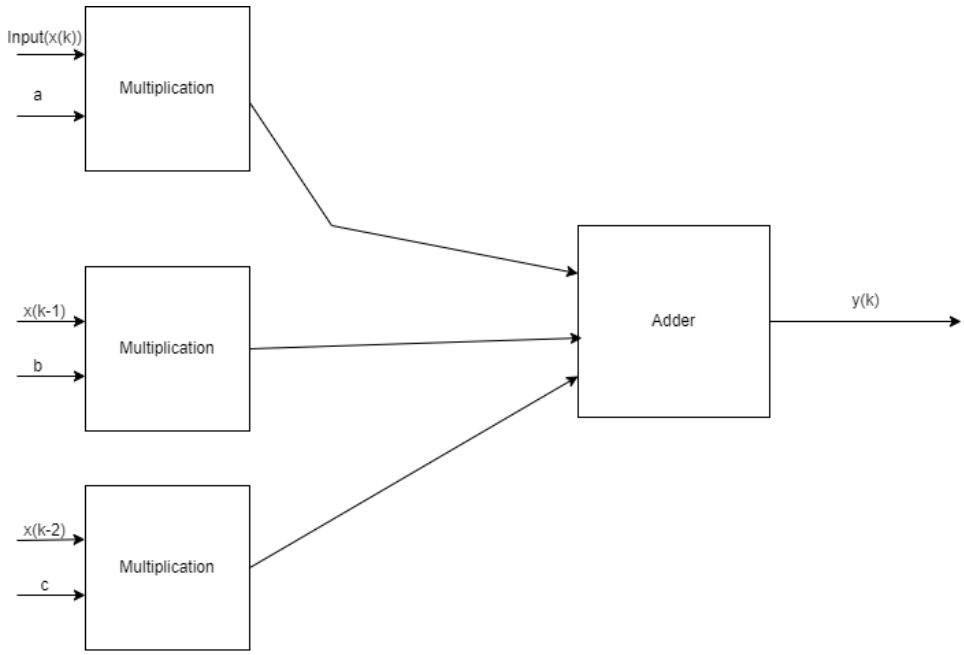

Fig. 9: Black box for the second filter

In order to realize the functionality of the second filter I will use three multiplication (shift and adder multiplication) and one Ripple Carry Adder in order to sum up all the results from the multiplications. a, b, and c are constants, while x(k), x(k-1) and x(k-2) are the inputs.

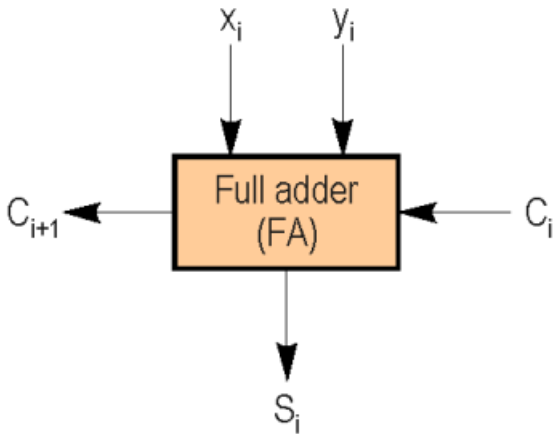

Fig. 10: Full adder

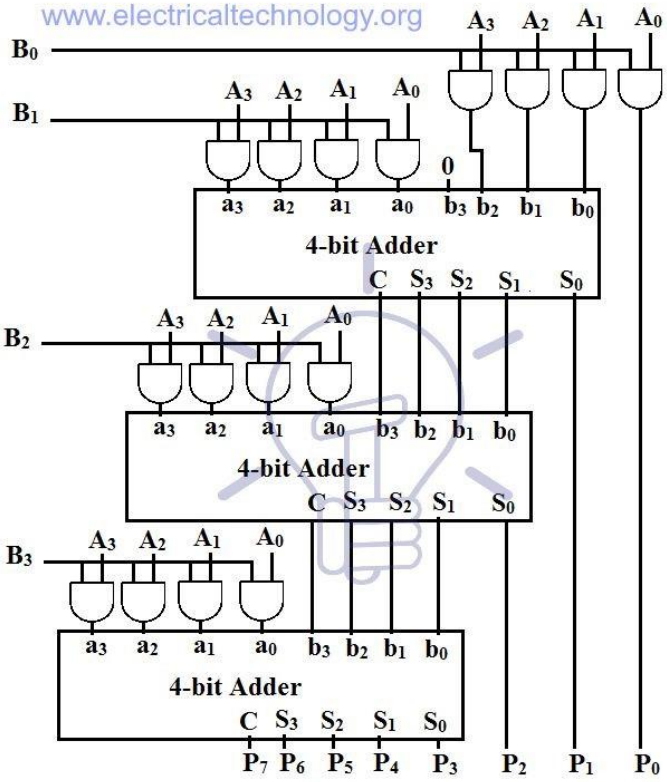

Fig. 11: Shift and adder multiplication on 4 bits

4.3 Third filter

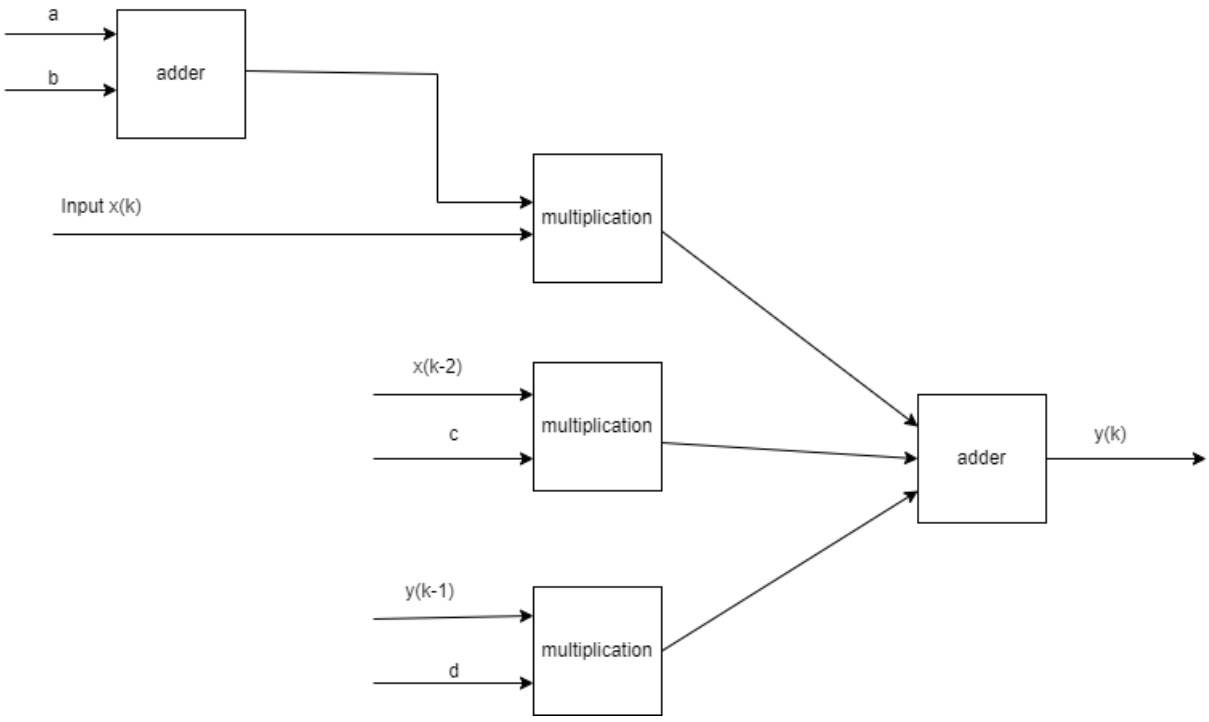

Fig. 11: Black box of the third filter

Third filter is like the second one, when I'm speaking about components, I'm using 3 shift and adder multiplication like before, but I have an extra adder which is for computing the sum between two constant values. Also like before, a, b, c, and d are constants. x(k), x(k-2) and y(k-1) are inputs, and y(k) is the output.

5.Implementation

First think what I did at the implementation was to make the first filter. For this, I make 2 comparators and their output from the first one, the "smaller" output and from the second one the "greater" output, I made an OR between them to realise the functionality of the first filter. "a" is the input on 8-bits, and the "output" is the out signal which guaranteed me if the number is between interval or not, 0 for false, 1 for true. If "output" is 1, that means the number is between the intervals and output which is represented on 16 bits will be displayed, otherwise will be 0.

```
entity firstfilter is
    Port ( input : in STD_LOGIC_VECTOR (7 downto 0);
           output : out STD_LOGIC_VECTOR(15 downto 0));
end firstfilter;

architecture Behavioral of firstfilter is
component comparator is
    Port ( a : in STD_LOGIC_VECTOR (7 downto 0);
           b : in STD_LOGIC_VECTOR (7 downto 0);
           equal : out STD_LOGIC;
           smaller : out STD_LOGIC;
           greater : out STD_LOGIC);
end component comparator;

signal smaller1, smaller2 :STD_LOGIC := '0';
signal greater1, greater2 : STD_LOGIC := '0';
signal equal1, equal2 : STD_LOGIC := '0';
signal output1 : STD_LOGIC := '0';
signal output2 : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000000";
begin

    firstcomparator : comparator port map(a => input, b => "00000101", equal => equal1, smaller => smaller1, greater => greater1);
    secondcomparator : comparator port map(a=>input, b => "00011110", equal => equal2, smaller => smaller2, greater => greater2);

    output1 <= not (smaller1 OR greater2);

    output <= "00000000" & input when output1 = '1' else "0000000000000000";


end Behavioral;
```

Fig. 13: First filter code in VHDL

Second filter contains three Shift and Adder Multiplication and one Ripple Carry Adder.

```
component multiplication is
    Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
           B : in STD_LOGIC_VECTOR (7 downto 0);
           P : out STD_LOGIC_VECTOR (15 downto 0));
end component multiplication;


constant a : std_logic_vector(7 downto 0) := "00000001"; --=01
constant b : std_logic_vector(7 downto 0) := "00011000"; --=18
constant c : std_logic_vector(7 downto 0) := "10101010"; --=AA

signal P1: std_logic_vector(15 downto 0);
signal P2: std_logic_vector(15 downto 0);
signal P3: std_logic_vector(15 downto 0);

signal S1 : std_logic_vector(15 downto 0);
signal carry_out_signal : std_logic;
signal final_carry_out : std_logic;
signal output : STD_LOGIC_VECTOR (15 downto 0):=(others => '0');

begin

    first_mul : multiplication port map(A => x1, B => a, P => P1);
    second_mul : multiplication port map(A => x2, B => b, P => P2);
    third_mul : multiplication port map(A => x3, B =>c, P => P3);

    first_adder : adder16bits port map(A => P1, B => P2, Cin => '0', S => S1, Cout => carry_out_signal);
    second_adder : adder16bits port map(A => S1, B=> P3, Cin => carry_out_signal, S => output, Cout => final_carry_out);
    y <= output when control = '1';
```

Fig. 14: Second filter code in VHDL

The first input is multiplied with the first constant, second input with the second constant and third input with the third constant and after this we sum up the results from all multiplications.

In order to make the third filter to work properly, I need the functionality of y to store the last value of the output and transmit it to the third filter when is activated.

For the third filter I used one extra adder to sum up the first two constants before multiplying it with the first input.

```
constant a : std_logic_vector(7 downto 0) := "00000001"; --=01
constant b : std_logic_vector(7 downto 0) := "00011000"; --=18
constant c : std_logic_vector(7 downto 0) := "10101010"; --=AA
constant d : std_logic_vector(7 downto 0) := "00011101"; --=1D

signal S1 : std_logic_vector(7 downto 0);
signal S2: std_logic_vector(15 downto 0);
signal cout1 : std_logic;
signal cout2: std_logic;
signal cout_final: std_logic;

signal P1 : std_logic_vector(15 downto 0);
signal P2 : std_logic_vector(15 downto 0);
signal P3 : std_logic_vector(31 downto 0);

signal d1 : std_logic_vector(15 downto 0);

begin

    adderBetweenConstants : adder8bits port map (A => a, B => b, Cin => '0', S => S1, Cout => cout1);

    first_mul : multiplication port map (A=> S1, B => x1, P => P1);
    second_mul : multiplication port map( A => x2, B => c, P => P2);

    concate : concatenare port map( input => d, output => d1);
    third_mul : multiplication16bits port map(A=>y1, B=>d1, P=> P3);

    first_adder : adder16bits port map(A=> P1, B=>P2, Cin => '0', S => S2, Cout => cout2);
    second_adder: adder16bits port map(A=>S2, B=> P3(15 downto 0), Cin => cout2, S=>y2, Cout => cout_final);


end Behavioral;
```

Fig. 15: Third filter code in VHDL

In order to make all the project to work I create a memory to make delay for inputs for having 3 different inputs for the last two filters. For choosing which filter to run I used a mux with 3 inputs which are the output from the all 3 filters and with the selection we choose which one to go further. “00” for first filter, “01” for second filter and “10” for third filter.

```
signal s_input0, s_input1, s_input2 : std_logic_vector(7 downto 0) := "00000000";
signal output_firstfilter : std_logic_vector(15 downto 0) := "0000000000000000";
signal output_secondfilter : std_logic_vector(15 downto 0) := "0000000000000000";
signal output_thirdfilter : std_logic_vector(15 downto 0) := "0000000000000000";
signal output_thirdfilter1 : std_logic_vector(15 downto 0) := "0000000000000000";
signal input_thirdfilter : std_logic_vector(15 downto 0) := "0000000000000000";
signal output_mux : std_logic_vector(15 downto 0) := "0000000000000000";
signal y_31 : std_logic_vector(15 downto 0) := "0000000000000000";
signal control : std_logic;

begin

    InputMemoty : first_memory port map(current_input => input, input0 => s_input0, input1 => s_input1, input2 => s_input2, clock => clock);
    FirstFilterz : firstfilter port map(input => s_input0, output => output_firstfilter);
    SecondFilterz : secondfilter port map(x1 => s_input0, x2 => s_input1, x3 => s_input2, control => control, y => output_secondfilter);
    control <= '1' when sel = "01" else '0';
    ThirdFilterz : thirdfilter port map(x1 => s_input0, x2 => s_input1, y1 => output_secondfilter , y2 => output_thirdfilter);
    MuxOutputz : mux_output port map(output_filter1 => output_firstfilter, output_filter2 => output_secondfilter, output_filter3 => output_thirdfilter, sel => sel, result

    output <= output_mux;

    display : seg port map (clk => clock, digit0 => output_mux(15 downto 12), digit1 => output_mux(11 downto 8), digit2 => output_mux(7 downto 4), digit3 => output_mux(3 d



end Behavioral;
```

Fig. 16: Project code in VHDL

For the simulation of the project, I used files where I put approximately 200 numbers and the output is also on the file.

```
    process
    file text_input : text open read_mode is "C:\Users\casia\OneDrive\Desktop\FACULTATE\3rd year - 1st semester\scs\proiect\project_2\project_2.srcs\sim_1\new\input.txt";
    variable in_line : line;

    variable inputs : std_logic_vector(7 downto 0);
    variable space : character;
    variable selector : std_logic_vector(1 downto 0);
    begin
       if not endfile(text_input) then
           readline(text_input, in_line);
           read(in_line, inputs);
           read(in_line,space);
           read(in_line, selector);
           wait for 10ns;
           input <= inputs;
           sel <= selector;
           wait for 10ns;
       else
           file_close(text_input);

   end if;
   end process;

   process
   file file_output : text open write_mode is "C:\Users\casia\OneDrive\Desktop\FACULTATE\3rd year - 1st semester\scs\proiect\project_2\project_2.srcs\sim_1\new\result.csv
   variable out_line : line;

   begin
    -- if wr_count < 5 then
       write(out_line, output);
       writeline(file_output, out_line);
     --  wr_count <= wr_count + 1;

       wait for 10ns;
```

Fig. 17: Testbench with files in VHDL

6. Testing

6.1 First filter:

For the first filter which is the interval where the numbers should be [5;30], I considered the following values for the input: FF, 0E, 01, 07. For FF and 01 the output must be 00 because these two numbers are outside the interval, for the 0E and 07 the output must be represented on 16bits.

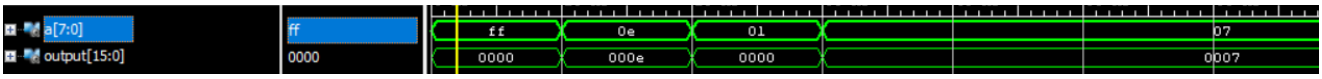

Fig. 18: Testbench for the first filter

6.2 Second filter:

For the second filter with the formula y(k)=a*x(k)+b*x(k-1) +c*x(k-2), I used as constants: a=01, b=18 and c=AA. I have three different inputs. First series of inputs are 55, 00, F8. So, 01*55 = 55, 18*00=00, AA*F8=A4B0. If we sum up the results, we obtain 55+00+A4B0=A505, exactly the output what we expected. Second series of inputs are: AA,01, DB. So, 01*AA=AA, 18*01=18, AA*DB=916E. If we sum up the results, we obtain AA+18+916E=9230, exactly the output we expected. Third series of inputs are 33, 02, 07. So, 01*33=33, 18*02=30, AA*07=4A6. If we sum up the results, we obtain 33+30+4A6=0509, exactly the output we expected. Control is always 1 in order to permit to write the output.

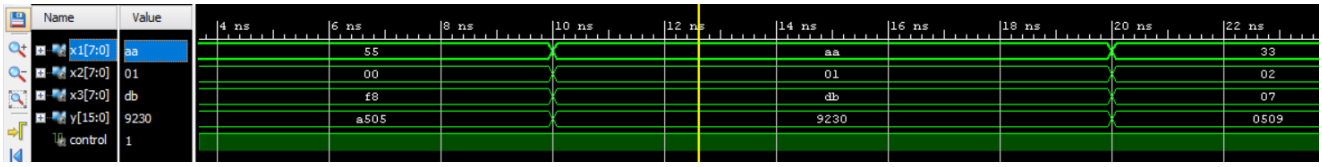

Fig. 19: Testbench for the second filter

6.3 Third filter:

For the third filter with the formula y(k) = (a+b)*x(k) + c*x(k-2) + d*y(k-1), I used as constants: a=01, b=18, c=AA, d=1D. I have three different inputs. First series of inputs are AA, 01, 0001. So we have first of all the sum of the first two constants a+b=01+18=19, so the first constant multiplied with the first input will be always 19. So 19*AA=109A, AA*01=AA, 1D*0001=1D. If we sum up the results, we obtain 109A+AA+1D=1161, exactly the output what we expected. Second series of inputs are: 55, FF, 000F. So 19*55=84D, AA*FF=A956, 1D*000F=1B3. If we sum up the results we obtain 84D+A956+1B3=B356. Third series of inputs are: 1C, 00, 0439. So we have 19*1C=2BC, AA*00=00, 1D*0439=7A75. If we sum up the results we obtain 2BC+00+7A75=7D31, exactly the output that we expected.

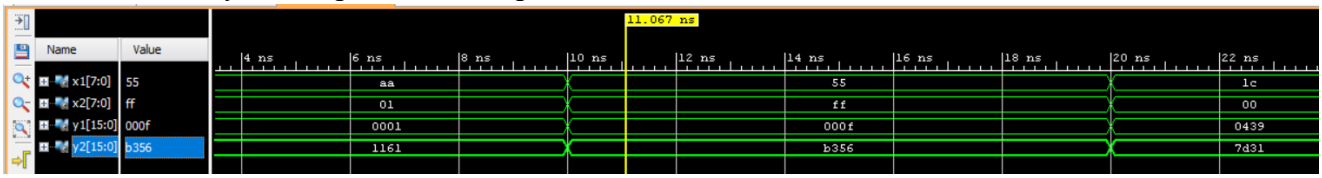

Fig. 20: Testbench for the third filter

6.4 Circuit:

For the entire circuit I have a lot of inputs that come at every 10ns. I will take examples from somewhere in the middle. For the selection “00” we have the first filter where the inputs are 09 and 0A, because the numbers are between the interval, the output will be the input on 16 bits: 0009 and 000A. After that we have selection “01” which is the second filter. Here first we have the inputs: 0B, 0A, 09. 01*0B=0B, 18*0A=F0, AA*09=5FA, after summing up the results we obtain 0B+F0+5FA=6F5 which is the expected output. Let’s take another example of the second filter, inputs are: 0C, 0B, 0A. 01*0C=0C, 18*0B=108, AA*0A=6A4, after summing up the results we obtain the expected output 0C+108+6A4=7B8. After this we have selection “10” which is the third filter, the third input is the last output that second filter had, in this example the last output was 87A, another two inputs are 0D and 0C. 19*0D=145, AA*0C=7F8, 1D*87A=F5D2. After summing up the result we obtain the expected output 145+7F8+F5D2=FF0F.

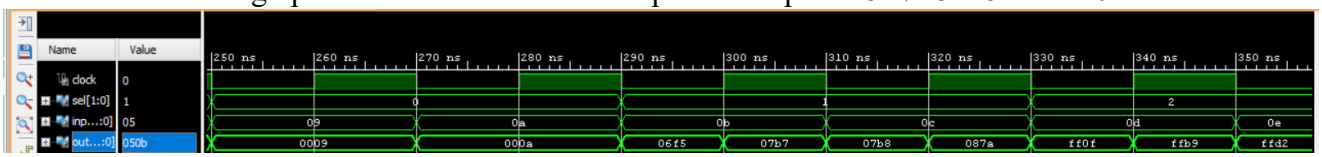

Fig. 21: Testbench for the entire circuit

7. Conclusion

At the end the project was a challenging one, where I learned new things like working with files in VHDL, to do a filter from the beginning, learning another types of adder and multiplication despite that ones that knew them before, how to save the last output and transmit it as input to the next filter. I had moments when I got stuck but I managed to go throw them after making some research on the internet. But overall I liked to work at this project.

8. Bibliography

https://en.wikipedia.org/wiki/Finite_impulse_response
https://123.physics.ucdavis.edu/week_5_files/filters/digital_filter.pdf
https://en.wikipedia.org/wiki/Digital_filter
https://en.wikipedia.org/wiki/Infinite_impulse_response https://surf-vhdl.com/read-from-file-in-vhdl-using-textio-library/ https://forum.digikey.com/t/uart-vhdl/12670
https://www.analog.com/en/design-center/glossary/low-passfilter.html#:~:text=A%20low%2Dpass%20filter%20(LPF,attenuating%20all%20signals%20above%20it.
https://www.izotope.com/en/learn/6-ways-to-use-a-high-pass-filter-whenmixing.html#:~:text=A%20high%2Dpass%20filter%20is,of)%20lower%2Dfrequency%20signals.

Table of figures: