UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galielo Galilei"

Master degree in Physics of Data

Course: Management and Analysis of Physics Datasets, module A

FIR filter implementation on FPGA

Group 2

Bergamin Eleonora
Boni Filippo
Campagnola Stefano
Santagata Luca

Abstract

The aim of the project is the implementation of a low-pass FIR filter on an FPGA board. In order to do this, VHDL and Python code is used. The objective then is to study and analyze the results obtained and compare them with those obtained through a Python simulation.

Contents

1 FIR filter **3**
1.1 Definition . . . 3
1.2 Analysis of filter behavior . . . 4

2 Implementation on FPGA **6**
2.1 Data representation . . . 6
2.2 VHDL implementation . . . 7

3 Analysis and results **12**
3.1 Signal pre processing . . . 12
3.2 Cutoff frequency and number of taps . . . 12
3.3 FPGA vs Python . . . 15

4 Conclusions **17**

5 Appendix **18**

1 FIR filter

1.1 Definition

The used definition of FIR filter is the following:

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

Mathematically, a series of input data $\{x_i\}_{i=1,..,M}$ is considered, and using a N order FIR filter the output sequence$\{y_i\}_{i=1,..,M}$ is given by:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] = \sum_{i=0}^{N} b_i x[n-i] \tag{1}$$

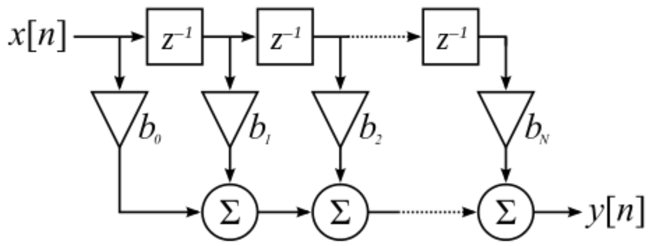


Figure 1: N^{th} order FIR filter.

- $x[n]$ and $y[n]$ are the n^{th} components of the input and output series;

- $\{b_i\}_{i=0,..,N}$ are the coefficients which characterize the response of the filter;

- N is the order of the filter: the larger N is, the more the filter behaves similar to an ideal one. Moreover the number of coefficients $N + 1$ is called number of taps.

It is to note that filter coefficients represent the weights of a linear combination: since the FIR filter is linear, the output can be interpreted as the linear combination of the input data values.

1.2 Analysis of filter behavior

In order to implement a N^{th} order filter the `scipy.signal.firwin` Python method is used. The function returns the filter's coefficients when the number of taps $N+1$ is given in input, the cutoff frequency f_c, the sampling frequency of signal f_s and the *filter type*.

In this case, the aim is designing a low-pass FIR filter, meaning that all the frequencies above the cutoff frequency will be attenuated.

Listing 1: `firwin` library

```
from scipy.signal import firwin

N_taps = 8    #number of taps
fc = 2000     #cutoff frequency
fs = 44100    #sampling frequency

coeffs = firwin(numtaps=N_taps,
               cutoff=2000,
               fs=fs,
               pass_zero="lowpass")
```

Given the coefficients, the filter response can be computed using the `scipy.signal.freqz` Python method. The output consists of a range of frequencies and the corresponding filter responses.

Listing 2: `freqz` method

```
from  scipy.signal import freqz

[w,h] = freqz(coeffs, worN = len(x))
```

It is to note that the most important parameters that define the filter's behavior are the cutoff frequency f_c and the number of taps $N+1$; for this reason an analysis of the response of the filter with respect to the number of taps is performed. To do this the cutoff frequency is set at $f_c = 2000$ Hz as explained in section 3.2.

Each plot of figure 2 shows the behavior of the ideal filter compared to the behavior of a real one with a fixed finite number of taps. Going from left to right, it is observed that when considering the fewest number of taps among those selected, a significant damping effect is only effective at frequencies much higher than that of the cutoff. As the number of taps is increased it can be seen that the behavior of the real filter gets closer to the behavior of the ideal one and the filtering action is also effective at frequencies closer to the cutoff frequency.

Suppose now that the aim is filtering a noise signal at a fixed frequency f_{noise}. To reach the objective two possible solutions can be analyzes:

- **Increase in the number of taps:** The cutoff frequency could be fixed, and the number of taps increased. The response of the real FIR filter then will be closer to the response of the ideal one. However, it is not practical to implement this type of solution on the FPGA.
- **Lower the cutoff frequency:** It is known that the attenuation of the signal becomes more and more relevant as the distance from the cutoff frequency increases. By choosing a fixed number of taps, the cutoff frequency can be lowered, in order

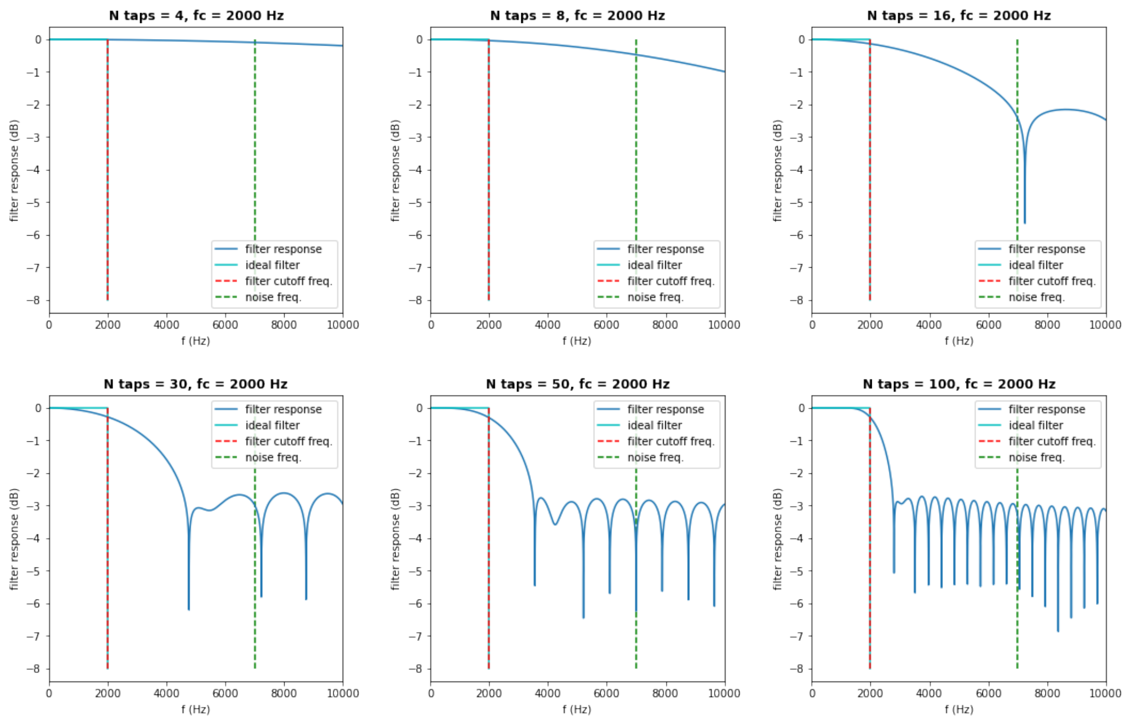


Figure 2: Ideal and real FIR filter response

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7
0.018	0.062	0.166	0.254	0.254	0.166	0.062	0.018

Table 1: Filter coefficients

to shift the frequency window in which the attenuation is unsatisfactory. However, this comes at the price of also partially attenuating signals that in principle should not be dampened.

From what has been said, it can be deduced that the cutoff frequency and the number of taps are in trade-offs. The analysis performed in section 3.2 allows to set the number of taps and the cutoff frequency to $N + 1 = 8$ and $f_c = 2000$ Hz.
This choice may seem not optimal due to the logarithmic scale with which the plots are made, but it guarantees an attenuation of the noise of about 70%. Starting from these values, the filter coefficients (table 1) are computed.

As can be seen in figure 3 the distribution of the values of the coefficients is regular, and symmetrical with respect to the indexes of the coefficients; furthermore the coefficients add up to one. From theory, it is known that this symmetry condition is satisfied by a linear phase FIR filter: this means that the delay of the filter is the same at all the frequencies, so there is not distortion. The signal then is subject to an overall delay.

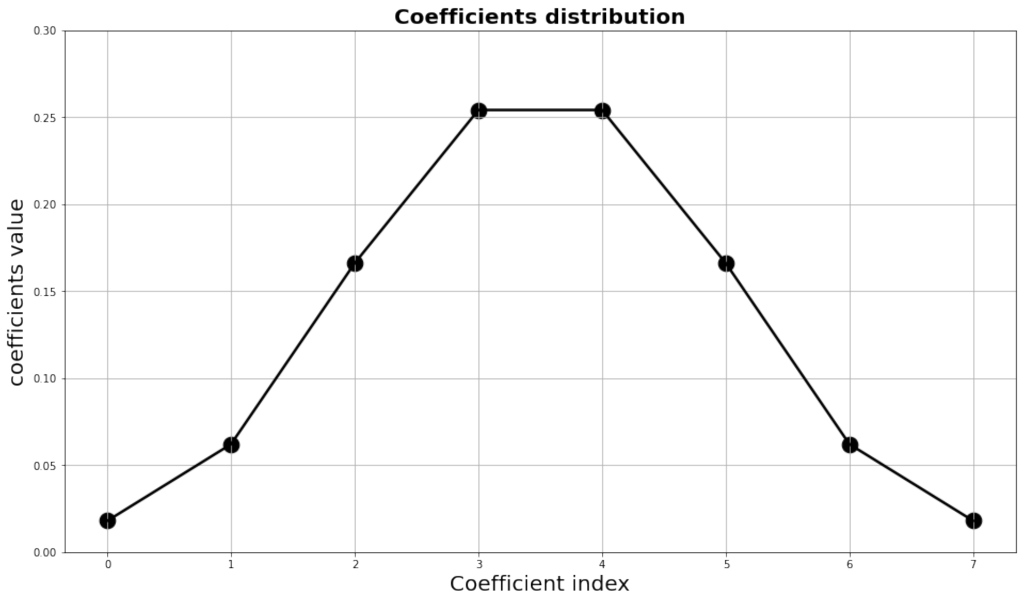


Figure 3: Coefficients distribution

2 Implementation on FPGA

2.1 Data representation

The input data the FPGA can deal with consists of 8 bits signed integers in binary representation. Using this format the most significant bit is the sign of the number, so the range of values that can be represented is [-128,127]. However the computed coefficients are float numbers in the range [-1,1], so they have to be properly rescaled. Also the audio signal data has the same initial amplitude range, so a similar rescaling must be performed.
The rescaling and the following cast to integer has to be done in such a way that the omitted fractional part retains as little information as possible. The largest possible coefficient allowing to fit a decimal number smaller than or equal to one (such as the filter coefficients) in an 8 bits number, is 2^7. It should be remembered that this type of decimal multiplication corresponds to a 7 bits shift to the left in binary representation. To proceed then, the coefficients are multiplied by 2^7, and then the FPGA will use the integer part of these rescaled coefficients.

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7
2	7	21	32	32	21	7	2

Table 2: Rescaled filter coefficients

It is known that a vertical rescaling of the signal amplitude holds no physical meaning: indeed it represents only a change of units in which the amplitude is measured, and this is always possible. The objective therefore is to transform the signal in order to get the highest possible precision, i.e. the integer part retains most of the information; it should be noted that the FPGA output obtained after the processing will be scaled by the same coefficient. However, the input signal must have values in $[-127, 128]$ range, so that its integer part can be represented using 8 bits. Since the signal amplitude has absolute value lower than 0.25 as can be seen in figure 8, the chosen multiplying coefficient is $k_{data} = 500$, that low enough to satisfy the representation constraint and high enough to ensure little loss of precision.

The consequence of the binary operations performed by the FIR filter is an increase in the data size. Indeed, when considering two binary numbers C, D of length N, M bits respectively:

- The sum $C + D$ has length $[max\{N, M\} + 1]$ bits;
- The product $C \cdot D$ has length $[N + M]$ bits;

In the process 8 bits signed integers are used, and an 8 taps FIR filter is considered, so after the filtering the output length will be 19 bits. In order to compare the FPGA output with Python simulation results, the output has to be rescaled back. Having rescaled the FIR coefficients by $k_{coeff} = 2^7$ allows to consider the 7 LSB of the FPGA output as its fractional part. This means that the other $19 - 7 = 12$ bits represent the integer part of the number. Keeping in mind that the output must be an 8 bits integer number, it is necessary to select a subsequence made of the 8 most meaningful bits, meaning that they contain most of the information. As shown below, the appropriate choice consists of the 8 LSB out of the 12 remaining ones, since this procedure does not imply any precision loss.

This selection corresponds to a back shift to the right of 7 positions.

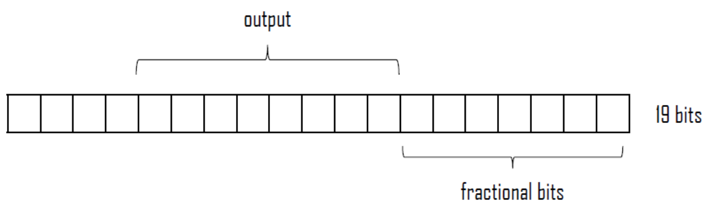


Figure 4: Binary FPGA output

2.2 VHDL implementation

The VHDL unit designed consists of three different components: *UART Receiver, FIR Filter*, and *UART Transmitter.*

Data are sent from the external device to the UART Receiver, which allows it to communicate with the FIR filter. The filter then processes the data, and the Uart Transmitter sends them back to the initial device. The three sub-unit share the same clock with a 100 MHz frequency. UART Transmitter and Receiver work with a frequency of 115200 pulses/s, meaning that they process 115200 bits per second: this frequency is called baudrate and corresponds to processing 1 bit in approximately 867 clock cycles. They are asynchronous devices: in fact, the transmission synchronization takes place through data itself.

The FIR filter unit is characterized by the following structure.

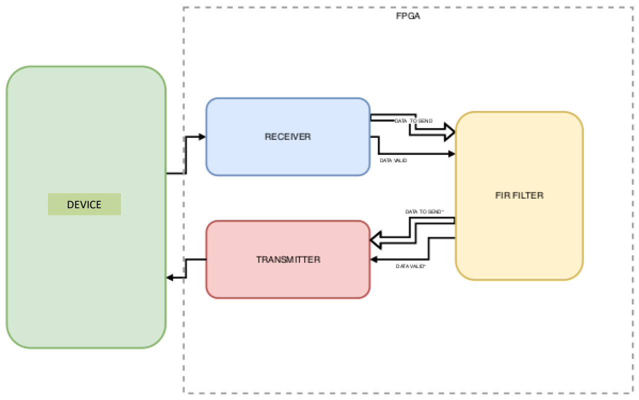


Figure 5: Device-FPGA interface

Listing 3: VHDL code for FIR filter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
---------------------------------------------------------------------
entity fir_filter is
port (
        clock   : in  std_logic;
        valid_in : in  std_logic; -- data valid in
        valid_out : out std_logic; -- data valid out
        -- coeff
        i_co_0 : in  std_logic_vector(7 downto 0);
        i_co_1 : in  std_logic_vector(7 downto 0);
        i_co_2 : in  std_logic_vector(7 downto 0);
        i_co_3 : in  std_logic_vector(7 downto 0);
        i_co_4 : in  std_logic_vector(7 downto 0);
        i_co_5 : in  std_logic_vector(7 downto 0);
        i_co_6 : in  std_logic_vector(7 downto 0);
        i_co_7 : in  std_logic_vector(7 downto 0);
        -- data i/o
        data_input : in  std_logic_vector(7 downto 0);
        data_output : out std_logic_vector(7 downto 0));
end fir_filter;

---------------------------------------------------------------------

architecture rtl of fir_filter is

        -- type declaration (they are all matrices)
        type buffer_t   is array (0 to 7) of signed(7    downto 0);
        type coeff_t    is array (0 to 7) of signed(7    downto 0);
        type mult_t     is array (0 to 7) of signed(15   downto 0);
        type add0_t    is array (0 to 3) of signed(15+1 downto 0);
        type add1_t    is array (0 to 1) of signed(15+2 downto 0);

        -- signals of various types initialized to '0'
        signal buffer_s : buffer_t := (others => (others => '0')); -- buffer collects temporary the input data
        signal coeff_s  : coeff_t  := (others => (others => '0')); -- coeff collects the fir coefficients
        signal mult_s   : mult_t   := (others => (others => '0')); -- mult collects the results of the multiplications
        signal add0_s  : add0_t  := (others => (others => '0')); -- add0 collects the results of the first sum
        signal add1_s  : add1_t  := (others => (others => '0')); -- add1 collects the results of the second sum
        signal add2_s  : signed(15+3 downto 0) := (others => '0'); -- add2 collects the results of the last sum

        type state_type is (Idle, Input, Sum_1, Mul, Sum_2, Sum_3, Sumf, Output); -- new type state_type with multiple states
        signal state : state_type := Idle; -- signal of type state_type

---------------------------------------------------------------------

    begin
        main : process (clock) is -- main process of the fir filter
        begin
            if rising_edge(clock) then


                case state is
                    when Idle =>
                        valid_out <= '0';
                    if valid_in = '1' then -- receiver has correctly received the data
                        state <= Input;
                    end if;
                when Input =>
                    buffer_s <= signed(data_input)&buffer_s(0 to buffer_s'length-2); -- & concatenate input data
                    coeff_s(0) <= signed(i_co_0); -- fill the coefficients with the ones given in the top
                    coeff_s(1) <= signed(i_co_1);
                    coeff_s(2) <= signed(i_co_2);
                    coeff_s(3) <= signed(i_co_3);
                    coeff_s(4) <= signed(i_co_4);
                    coeff_s(5) <= signed(i_co_5);
                    coeff_s(6) <= signed(i_co_6);
                    coeff_s(7) <= signed(i_co_7);

                    state <= Mul;

                when Mul =>
                    for i in 0 to 7 loop -- every coefficient is multiplied by the respective buffer element
                            mult_s(i) <= buffer_s(i)*coeff_s(i);
                            end loop;

                    state <= Sum_1;

                when Sum_1 =>

                    for i in 0 to 3 loop
                        add0_s(i) <= resize(mult_s(i*2),17) + resize(mult_s(i*2+1),17); -- combinations 01, 23, 45, 67
                    end loop;

                    state <= Sum_2;

                when Sum_2 =>

                    for i in 0 to 1 loop
                        add1_s(i) <= resize(add0_s(i*2),18) + resize(add0_s(i*2+1),18); -- combinations 01, 23
                    end loop;

                    state <= Sum_3;

                when Sum_3 =>

                    add2_s <= resize(add1_s(0),19) + resize(add1_s(1),19);

                   state <= Sumf;

                when Sumf =>

                    data_output <= std_logic_vector(add2_s(14 downto 7)); -- take only 8 bits for the output

                        state <= Output;

                    when Output =>

                        valid_out <= '1';

                        state <= Idle;

                when others => null;
            end case;

        end if;
    end process main;

end architecture rtl;
```

The main process of this entity is a state machine defined by the cases [*Idle*, *Input*, *Mul*, *Sum_1*, *Sum_2*, *Sum_3*, *Sumf*, *Output*]. In *Idle* the filter is not performing any operations, waiting for the validation command from the UART receiver. In *Input* the buffer vector is updated to collect a new 8-bit value as its first element, and the coefficient vector is synchronized with the one in the Top architecture. In *Mul* the first mathematical operation is performed multiplying each element of the buffer with the correct FIR coefficient. The sum of all these terms is split between *Sum_1*, *Sum_2* and *Sum_3*: each time the elements are added in pairs of two. This is not strictly necessary, but it helps to keep individual operations simpler and more separate. In *Sumf* only 8 of the bits (14 downto 7) are kept from the 19 bits of *add2_s* and assigned to the output, for the reason previously explained. At the end, the validation signal for the UART transmitter is emitted in *Output*.

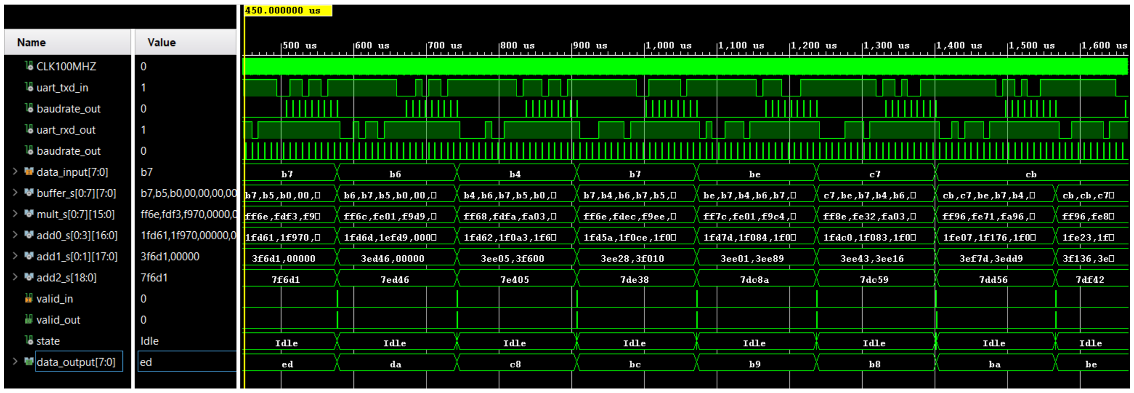
In order to connect all the components of the code a Top entity is needed. Its design is provided just below.

Listing 4: VHDL code for Top architecture

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity top is

  port (
    CLK100MHZ    : in  std_logic;
    uart_txd_in  : in  std_logic;
    uart_rxd_out : out std_logic);

end entity top;

architecture str of top is

  signal data_from_receiver       : std_logic_vector(7 downto 0);
  signal data_from_fir  : std_logic_vector(7 downto 0);
  signal validation_from_receiver : std_logic;
  signal validation_from_fir : std_logic;
  signal busy           : std_logic;

  -- constant signals: value of the coefficients computed as python_coef*2^7 and then kept the integer part
  signal i_co_0 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(2,8));
  signal i_co_1 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(7,8));
  signal i_co_2 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(21,8));
  signal i_co_3 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(32,8));
  signal i_co_4 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(32,8));
  signal i_co_5 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(21,8));
  signal i_co_6 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(7,8));
  signal i_co_7 : std_logic_vector(7 downto 0) := std_logic_vector(to_signed(2,8));


  component uart_transmitter is
    port (
      clock           : in  std_logic;
      data_valid      : in  std_logic;
      data_to_send    : in  std_logic_vector(7 downto 0);
      uart_tx       : out std_logic;
      busy          : out std_logic);
  end component uart_transmitter;

  component uart_receiver is
    port (
      clock           : in  std_logic;
      uart_rx       : in  std_logic;
      received_data : out std_logic_vector(7 downto 0);
      valid         : out std_logic);
  end component uart_receiver;

  component fir_filter is
    port
    (
        -- clock and validation i/o
        clock     : in std_logic;
        valid_in    : in std_logic;
        valid_out : out std_logic;

        i_co_0 : in std_logic_vector(7 downto 0);
        i_co_1 : in std_logic_vector(7 downto 0);
        i_co_2 : in std_logic_vector(7 downto 0);
        i_co_3 : in std_logic_vector(7 downto 0);
        i_co_4 : in std_logic_vector(7 downto 0);
        i_co_5 : in std_logic_vector(7 downto 0);
        i_co_6 : in std_logic_vector(7 downto 0);
        i_co_7 : in std_logic_vector(7 downto 0);

        -- data i/o
        data_input    : in std_logic_vector(7 downto 0);
        data_output    : out std_logic_vector(7 downto 0)
    );
  end component fir_filter;

begin

  receiver: uart_receiver
    port map (
      clock        => CLK100MHZ,
      uart_rx    => uart_txd_in,
      valid      => validation_from_receiver,
      received_data   => data_from_receiver);

  filter : fir_filter
    port map (
      clock => CLK100MHZ,
      valid_in  => validation_from_receiver,
      valid_out => validation_from_fir,
      i_co_0 => i_co_0,
      i_co_1 => i_co_1,
      i_co_2 => i_co_2,
      i_co_3 => i_co_3,
      i_co_4 => i_co_4,
      i_co_5 => i_co_5,
      i_co_6 => i_co_6,
      i_co_7 => i_co_7,
      data_input => data_from_receiver,
      data_output => data_from_fir);

  transmitter : uart_transmitter
    port map (
      clock           => CLK100MHZ,
      data_valid      => validation_from_fir,
      data_to_send    => data_from_fir,
      busy            => busy,
      uart_tx         => uart_rxd_out);

end architecture str;
```

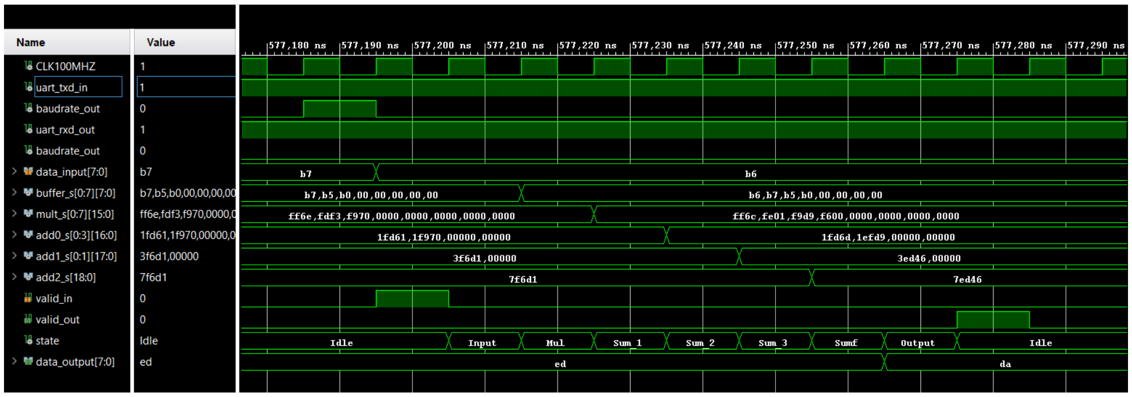
It's important to notice that FIR coefficients need to be set directly in their relative section and converted to signed *std_logic_vector*.

The correct functioning of the code was tested in a behavioural simulation of the Top architecture. As can be seen in figure 6 all signals and ports seem to change with the correct timing and assume the expected values (expressed in hexadecimal). More in detail, a simulation of the filter behaviour can be seen in figure 7, where all the states are properly modifying the buffer and the other arrays.



Name	Value
CLK100MHZ	0
uart_txd_in	1
baudrate_out	0
uart_rxd_out	1
baudrate_out	0
data_input[7:0]	b7
buffer_s[0:7][7:0]	b7,b5,b0,00,00,00,00
mult_s[0:7][15:0]	ff6e,fdf3,f970,0000,0
add0_s[0:3][16:0]	1fd61,1f970,00000,0
add1_s[0:1][17:0]	3f6d1,00000
add2_s[18:0]	7f6d1
valid_in	0
valid_out	0
state	Idle
data_output[7:0]	ed

Figure 6: Testbench of the top architecture



Name	Value
CLK100MHZ	1
uart_txd_in	1
baudrate_out	0
uart_rxd_out	1
baudrate_out	0
data_input[7:0]	b7
buffer_s[0:7][7:0]	b7,b5,b0,00,00,00,00
mult_s[0:7][15:0]	ff6e,fdf3,f970,0000,0
add0_s[0:3][16:0]	1fd61,1f970,00000,0
add1_s[0:1][17:0]	3f6d1,00000
add2_s[18:0]	7f6d1
valid_in	0
valid_out	0
state	Idle
data_output[7:0]	ed

Figure 7: Fir filter mechanism

Due to several problems with Python serial communication between the computer and the FPGA, it was not possible to obtain the filtered signal directly from the ARTY-A7 device. To bypass this kind of obstacle, the testbench code was set in an intelligent way. *TextIO* VHDL package allowed to read an external text file (*input_int.txt*) and at the same time to write to a new file (*output_results.txt*). Using this package, therefore, the input audio file to be filtered can be provided to the testbench in txt format, and the signal processed by the filter, then, can be recorded in a text file.

From now on, what is called FPGA output is actually the content of the text file obtained from the testbench.

3 Analysis and results

3.1 Signal pre processing

Now the focus is shifted to the audio signal and the FPGA output. The chosen input data for the FIR filter is a sample coming from a *Ludwig Van Beethoven's 5th Symphony in C Minor* audio file. The sampling frequency in this type of file (MP3, OGG, ...) is set to $f_s = 44100$ Hz.

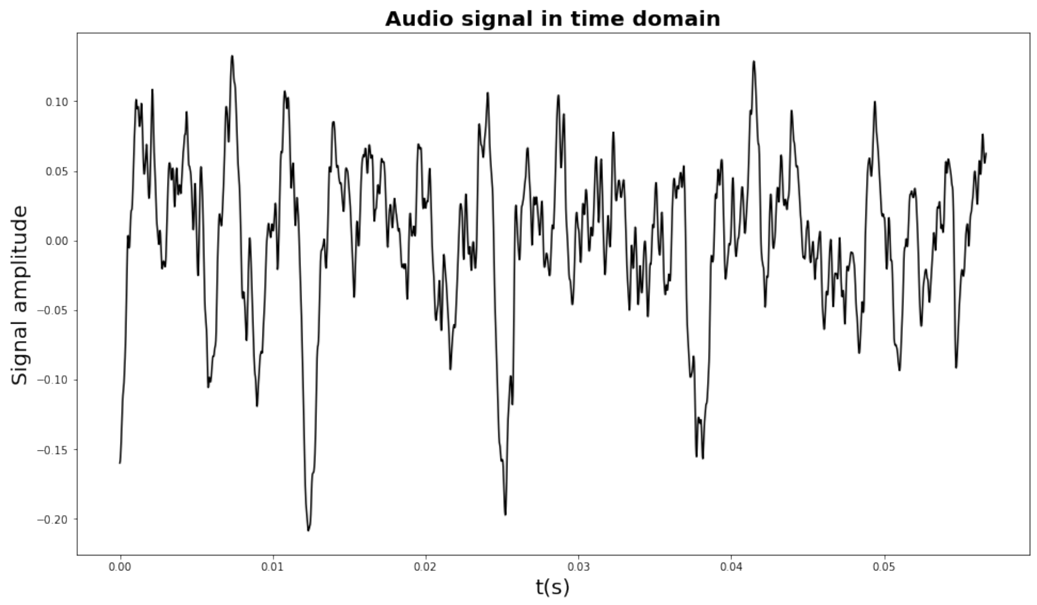


Figure 8: Audio signal in time domain

In figure 8 the waveform of the audio signal is shown in time domain. In figure 9 its frequency spectrum is plotted using a Fast Fourier Transform function provided by Python `Scipy` library.

It can be observed that the signal is the combination of a small number of frequencies mostly below 4000 Hz. In order to show the effect of FIR filter, the signal is altered by adding a sine wave with amplitude $A = 0.01$ and $f_{noise} = 7000$ Hz, so that in the Fourier spectrum appears a peak at the same frequency, as shown in figure 11.

Since the FIR filter is low-pass, and the cutoff frequency f_c is lower than noise frequency f_{noise}, the peak is expected to disappear after the processing.

3.2 Cutoff frequency and number of taps

As said in section 1.2 the number of taps $N + 1$ and the cutoff frequency f_c are in trade-offs, so an appropriate combination of them must be chosen to implement the

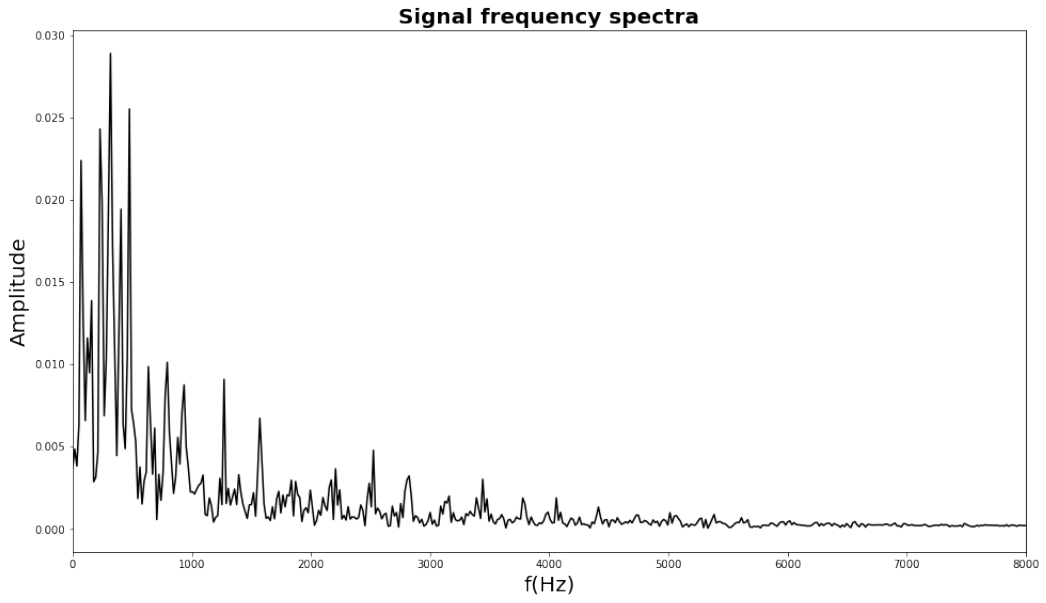


Figure 9: Frequency spectrum of the audio signal

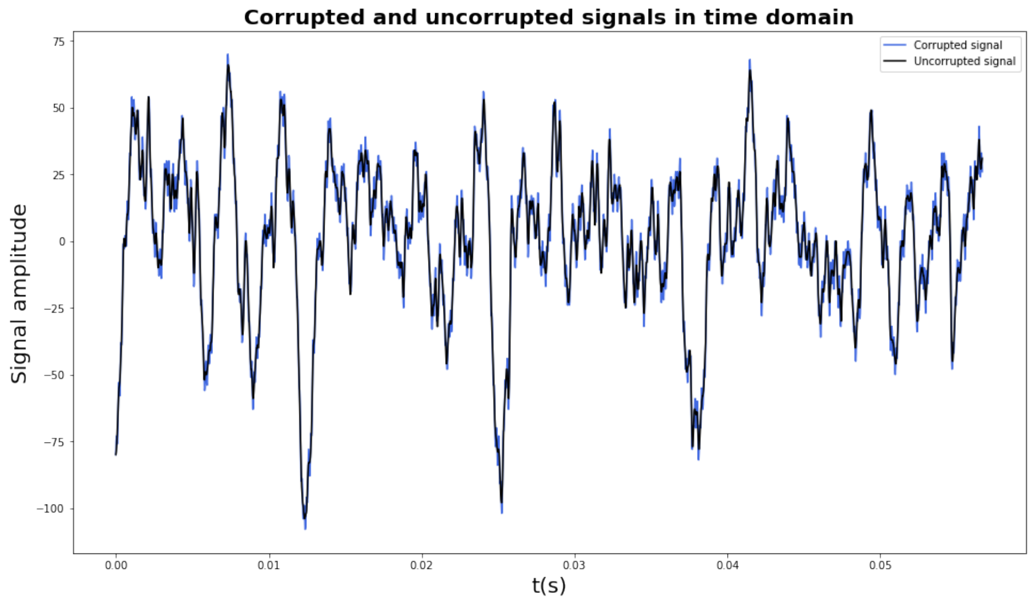


Figure 10: Audio signal before and after corruption

filter. It is decided to keep the number of taps small enough so as to keep the analysis simple: the selected number of taps is eight. In order to satisfy this constraint and to dampen the noise enough, the cutoff frequency must be sufficiently lower than the perturbation one. As a consequence, the selected range of possible cutoff frequencies is $[5000, 2000, 500]$ Hz.

As shown in figure 12 the filter effect on low frequencies (and therefore more significant) is similar for all the cutoff frequencies considered. However, it is noted that the attenuation of the noise increases as the frequency decreases to about 2000 Hz, then stabilizes. It is in fact observed that the behavior of the filter, having chosen a frequency cutoff lower than or equal to 2000 Hz, is always similar: there is no significant

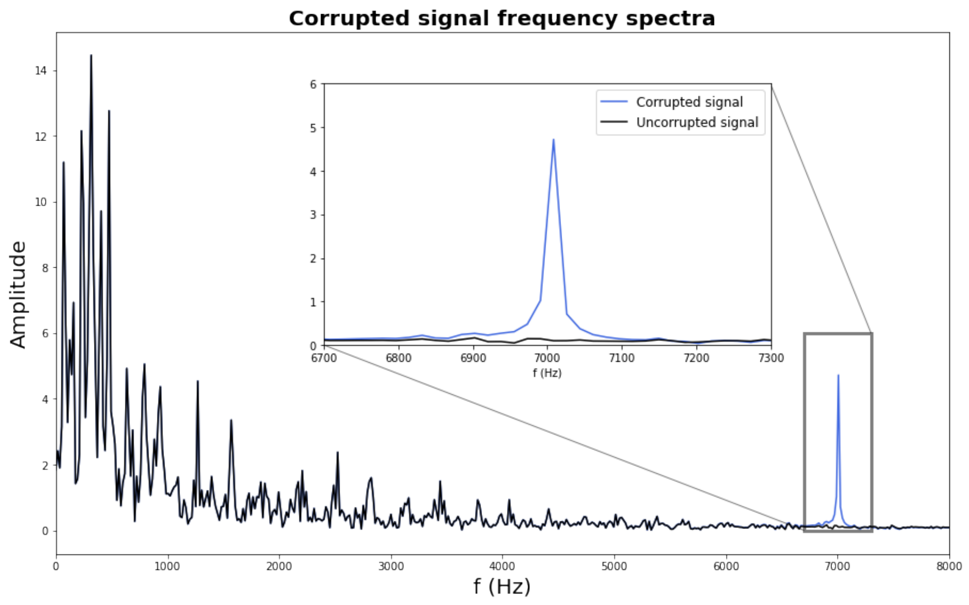


Figure 11: Frequency spectrum of the audio signal after corruption

improvement in the noise filtering compared to $f_c = 2000$ Hz and the attenuation of the actual signal remains the same. We have therefore chosen 2000 Hz as the cutoff frequency, which corresponds to the critical value of change in the filter behavior.

As a consequence of these considerations, the chosen value for the cutoff frequency is $f_c = 2000$ Hz. Using this cutoff frequency and eight taps, a noise attenuation of approximately 70% is calculated from the corrupted signal and FPGA output. This is consistent with what was found in section 1.2 through the transfer function.

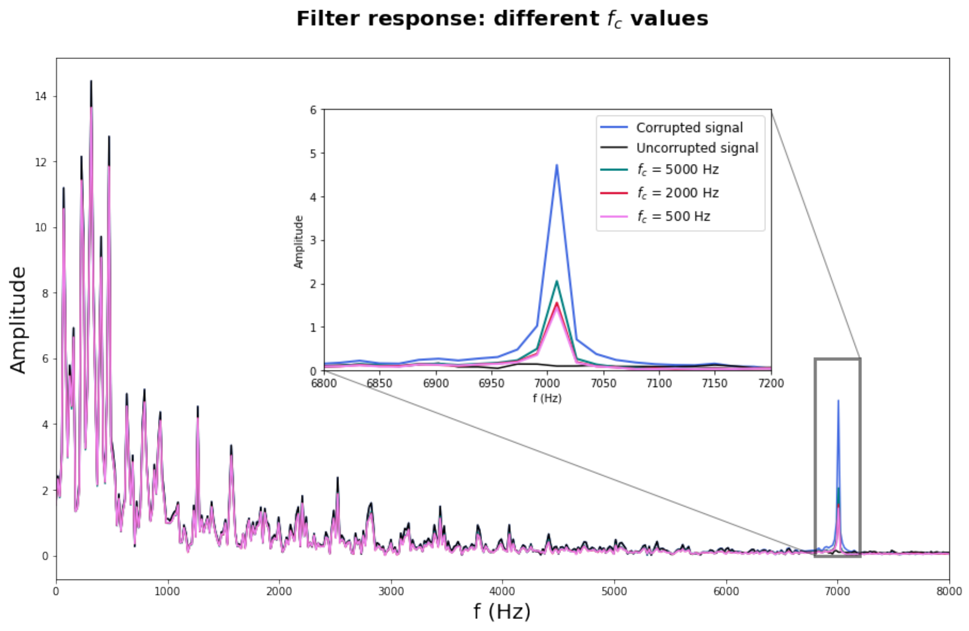


Figure 12: Frequency spectrum of the filter output for different cutoff frequencies

3.3 FPGA vs Python

Now the Python simulation results and the FPGA output are compared.
The code used to simulate the FIR filter behavior is the following:

Listing 5: FIR filter Python simulation

```
coeffs = firwin(numtaps=N_taps, cutoff=fc, fs=fs, pass_zero="lowpass") #compute filter coefficients
coeffs = (k_coeff*coeffs).astype(int) #FIR coefficients multiplication by k_coeff and integer conversion


#Filter convolution
def fir(input, coeffs, Nt) :
    output = np.zeros(input.shape[0])
    for n in range(7, len(output)) :
        for i in range(Nt) :
            output[n] += int(coeffs[i]*input[n-i])
    return output
```

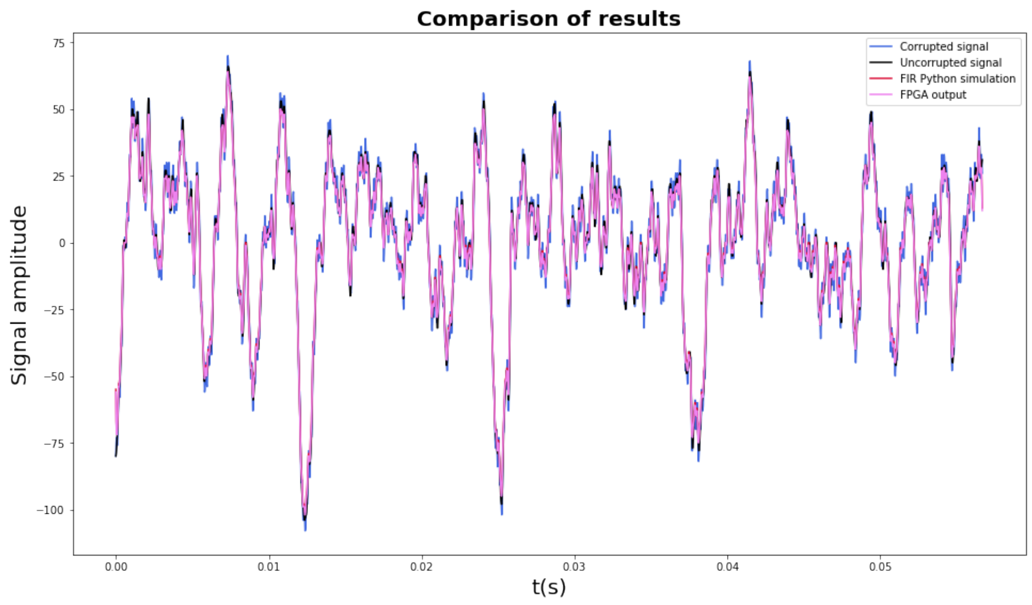


Figure 13: Comparison between Python simulated output and FPGA computed output.

Remembering that the corrupted signal (blue) is fed to the FPGA, the corresponding output is represented in figure 13 by the purple line. As can be seen, there is a good match between the FPGA output and the simulation results: the filter processed the data as expected. To quantify what has been said, the difference between the simulation and FPGA output is graphed in figure 15. It is observed that the difference in about half of the cases is different from zero, and is equal to -1.

This is because the FPGA processes unsigned integers and uses the 2's complement convention to represent negative numbers.
Using this representation, the range [-128, 127] is mapped to the range [0, 255], with the first half [0, 127] corresponding to positive numbers and the second [128, 255] corresponding to negative numbers.
When, at the end of the processing, the integer part of the 19-bit filter output sequence is taken, the analogue of a truncation operation is performed. For all numbers, positive and negative, this corresponds to a rounding down, therefore to the lower integer. In the simulation in Python instead this convention is not used, and in the end only the integer part of the obtained value is taken. However, for negative numbers this operation corresponds to a rounding to the higher integer number.

This explains the extent of the difference, which is at most 1, but also the sign: due to the different treatment, the negative number returned by the FPGA will be lower than that returned by the simulation.

Moreover, it can be observed that there is also a good correspondence between the FPGA output and the unaltered signal (black). This means that the filter has successfully dampened the noisy signal by reducing the amplitude of the oscillations, while keeping the part of the signal that contains most of the signal unchanged. To further confirm the analysis that has been carried out, the frequency spectra of the various signals considered are plotted.

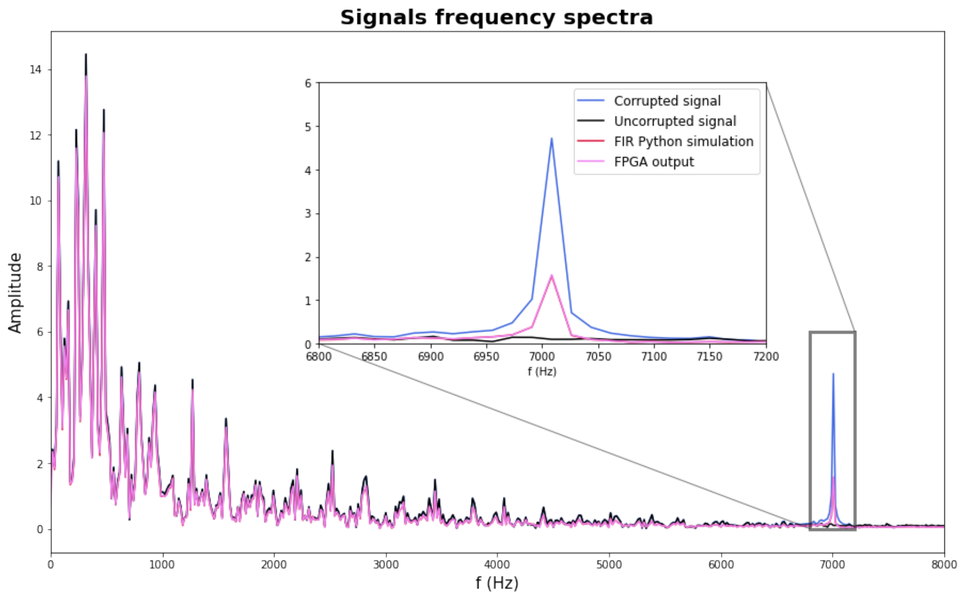


Figure 14: Comparison between frequency spectra of Python simulated output and FPGA computed output.

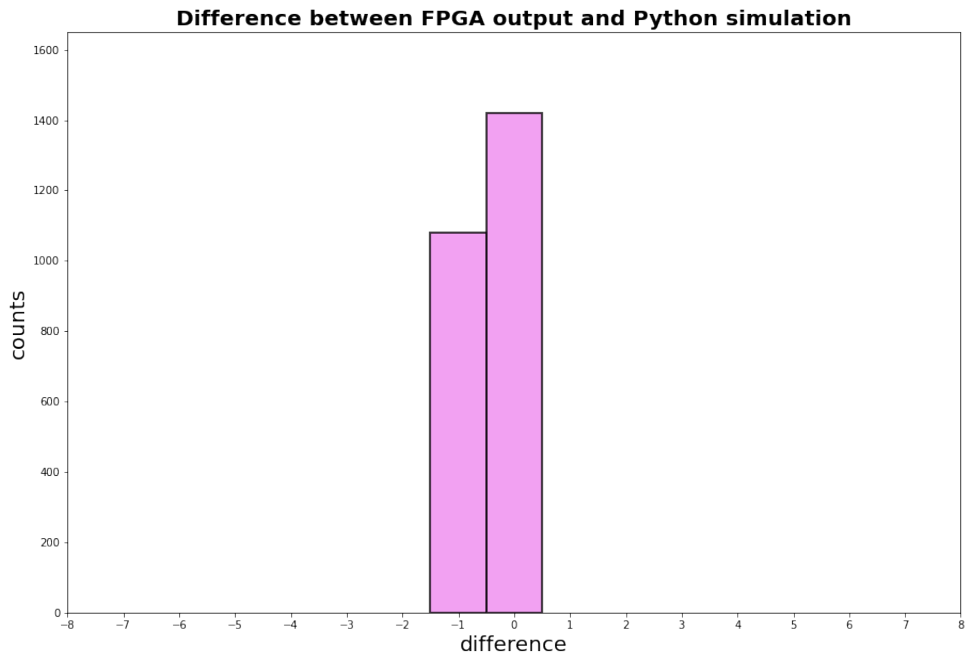


Figure 15: Histogram of the difference between FPGA output and Python simulation.

4 Conclusions

The aim of this project was to implement and study the behavior of a low-pass FIR filter on FPGA. A state machine was used to do this. The goodness of the results was verified by means of a Fourier analysis of the processed signal and by comparison with a computer simulation. The results confirm that the signal pre-processing has been carried out correctly and the behavior of the implemented filter is the expected one.

5 Appendix

Listing 6: Baudrate Generator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity baudrate_generator is

        port(
                        clk_in   : in std_logic;
                        clk_out  : out std_logic);

end entity baudrate_generator;
---------------------

architecture str of baudrate_generator is

        signal count          : unsigned(10 downto 0) := (others => '0');
        constant divider      : unsigned(10 downto 0) := to_unsigned(867, 11);
begin


        baudrate : process(clk_in) is


        begin

                if (rising_edge(clk_in)) then
                        count <= count + 1;
                        if count = divider then
                                clk_out <= '1';
                                count   <= (others => '0');
                        else
                                clk_out <= '0';
                        end if;

                end if;

        end process baudrate;

end architecture str;
```

Listing 7: Sample Generator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sampler_generator is

  port (
    clock        : in  std_logic;
    uart_rx      : in  std_logic;
    baudrate_out : out std_logic);

end entity sampler_generator;

architecture rtl of sampler_generator is

  type state_t is (idle_s, start_s, bit0_s, bit1_s, bit2_s, bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s, stop_s);
  signal state : state_t := idle_s;

  signal counter        : unsigned(10 downto 0) := (others => '0');
  signal delay_counter  : unsigned(10 downto 0) := (others => '0');
  constant divisor      : unsigned(10 downto 0) := to_unsigned(867, 11);
  constant half_divisor : unsigned(10 downto 0) := to_unsigned(433, 11);
  signal busy           : std_logic             := '0';
  signal pulse_out      : std_logic;
  signal enable_counter : std_logic             := '0';
  signal enable_delay   : std_logic             := '0';
begin

  --
  pulse_generator : process (clock) is
  begin  -- process main
    if rising_edge(clock) then          -- rising clock edge
      if enable_counter = '1' then
        counter <= counter + 1;
        if counter = divisor then
          pulse_out <= '1';
          counter   <= (others => '0');
        else
          pulse_out <= '0';
        end if;
      else
        counter <= (others => '0');
      end if;
    end if;
  end process pulse_generator;
  state_machine : process (clock) is
  begin  -- process state_machine
    if rising_edge(clock) then          -- rising clock edge
      case state is
        when idle_s =>
          enable_counter <= '0';
          if uart_rx = '0' then
            state <= start_s;
          end if;
        when start_s =>
          -- enable baudrate_generator
          enable_counter <= '1';
          if pulse_out = '1' then
            state <= bit0_s;
          end if;
        when bit0_s =>
          if pulse_out = '1' then
            state <= bit1_s;
          end if;
        when bit1_s =>
          if pulse_out = '1' then
            state <= bit2_s;
          end if;
        when bit2_s =>
          if pulse_out = '1' then
            state <= bit3_s;
          end if;
        when bit3_s =>
          if pulse_out = '1' then
            state <= bit4_s;
          end if;
        when bit4_s =>
          if pulse_out = '1' then
            state <= bit5_s;
          end if;
        when bit5_s =>
          if pulse_out = '1' then
            state <= bit6_s;
          end if;
        when bit6_s =>
          if pulse_out = '1' then
            state <= bit7_s;
          end if;
        when bit7_s =>
          if pulse_out = '1' then
            state <= idle_s;
          end if;
        when others => null;
      end case;
    end if;
  end process state_machine;

  delay_line : process (clock) is
  begin  -- process delay_line
    if rising_edge(clock) then          -- rising clock edge
      if pulse_out = '1' then
        --start_count
        enable_delay <= '1';
      end if;
      if delay_counter = half_divisor then
        enable_delay <= '0';
        baudrate_out <= '1';
      --end count
      else
        baudrate_out <= '0';
      end if;
      if enable_delay = '1' then
        delay_counter <= delay_counter + 1;
      else
        delay_counter <= (others => '0');
      end if;
    end if;
  end process delay_line;

end architecture rtl;
```

Listing 8: Uart Receiver

```
library ieee;
use ieee.std_logic_1164.all;


entity uart_receiver is

  port (
    clock         : in  std_logic;
    uart_rx       : in  std_logic;
    valid         : out std_logic;
    received_data : out std_logic_vector(7 downto 0));

end entity uart_receiver;
```

```
architecture str of uart_receiver is

  type state_t is (idle_s, start_s, bit0_s, bit1_s, bit2_s, bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s, stop_s);
  signal state : state_t := idle_s;

  signal baudrate_out : std_logic;
  signal received_data_s : std_logic_vector(7 downto 0);

  component sampler_generator is
    port (
      clock        : in  std_logic;
      uart_rx      : in  std_logic;
      baudrate_out : out std_logic);
  end component sampler_generator;


begin  -- architecture str

  sampler_generator_1 : sampler_generator
    port map (
      clock        => clock,
      uart_rx      => uart_rx,
      baudrate_out => baudrate_out);


  main : process (clock) is
  begin  -- process main
    if rising_edge(clock) then          -- rising clock edge
      case state is
        when idle_s =>
          received_data_s <= (others => '0');
          valid           <= '0';
          if uart_rx = '0' then
            state <= start_s;
          end if;
        when start_s =>
          if baudrate_out = '1' then
            received_data_s(0) <= uart_rx;
            state <= bit0_s;
          end if;
        when bit0_s =>
          if baudrate_out = '1' then
            received_data_s(1) <= uart_rx;
            state              <= bit1_s;
          end if;
        when bit1_s =>
          if baudrate_out = '1' then
            received_data_s(2) <= uart_rx;
            state              <= bit2_s;
          end if;
        when bit2_s =>
          if baudrate_out = '1' then
            received_data_s(3) <= uart_rx;
            state              <= bit3_s;
          end if;
        when bit3_s =>
          if baudrate_out = '1' then
            received_data_s(4) <= uart_rx;
            state              <= bit4_s;
          end if;
        when bit4_s =>
          if baudrate_out = '1' then
            received_data_s(5) <= uart_rx;
            state              <= bit5_s;
          end if;
        when bit5_s =>
          if baudrate_out = '1' then
            received_data_s(6) <= uart_rx;
            state              <= bit6_s;
          end if;
        when bit6_s =>
          if baudrate_out = '1' then
            received_data_s(7) <= uart_rx;
            state              <= bit7_s;
          end if;
        when bit7_s =>
          if baudrate_out = '1' then
            valid              <= '1';
            received_data <= received_data_s;
            state              <= idle_s;
          end if;
        when others => null;
      end case;
    end if;
  end process main;
end architecture str;
```

Listing 9: Uart Transmitter

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity uart_transmitter is

  port (
    clock        : in  std_logic;
    data_to_send : in  std_logic_vector(7 downto 0);
    data_valid   : in  std_logic;
    busy         : out std_logic;
    uart_tx      : out std_logic);

end entity uart_transmitter;


architecture rtl of uart_transmitter is


  component baudrate_generator is
    port (
      clock        : in  std_logic;
      baudrate_out : out std_logic);
  end component baudrate_generator;

  signal baudrate_out : std_logic;
-- state machine signals
  type state_t is (idle_s, data_valid_s, start_s, bit0_s, bit1_s, bit2_s, bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s, stop_s);

  signal state : state_t := idle_s;
begin  -- architecture rtl

  baudrate_generator_1 : baudrate_generator
    port map (
      clock        => clock,
      baudrate_out => baudrate_out);


  -- State Machine


  main_state_machine : process (clock) is
  begin  -- process main_state_machine
    if rising_edge(clock) then          -- rising clock edge
      case state is
        when idle_s =>
          busy    <= '0';
          uart_tx <= '1';
          if data_valid = '1' then
            state <= data_valid_s;
          end if;
        when data_valid_s =>
          busy <= '1';
          if baudrate_out = '1' then
            state <= start_s;
          end if;
        when start_s =>
          uart_tx <= '0';
          if baudrate_out = '1' then
            state <= bit0_s;
          end if;
        when bit0_s =>
          uart_tx <= data_to_send(0);
          if baudrate_out = '1' then
            state <= bit1_s;
          end if;
        when bit1_s =>
          uart_tx <= data_to_send(1);
          if baudrate_out = '1' then
            state <= bit2_s;
          end if;
        when bit2_s =>
          uart_tx <= data_to_send(2);
          if baudrate_out = '1' then
            state <= bit3_s;
          end if;
        when bit3_s =>
          uart_tx <= data_to_send(3);
          if baudrate_out = '1' then
            state <= bit4_s;
          end if;
        when bit4_s =>
          uart_tx <= data_to_send(4);
          if baudrate_out = '1' then
            state <= bit5_s;
          end if;
        when bit5_s =>
          uart_tx <= data_to_send(5);
          if baudrate_out = '1' then
            state <= bit6_s;
          end if;
        when bit6_s =>
          uart_tx <= data_to_send(6);
          if baudrate_out = '1' then
            state <= bit7_s;
          end if;
        when bit7_s =>
          uart_tx <= data_to_send(7);
          if baudrate_out = '1' then
            state <= stop_s;
          end if;
        when stop_s =>
          uart_tx <= '1';
          if baudrate_out = '1' then
            state <= idle_s;
          end if;
        when others => null;
      end case;
    end if;
  end process main_state_machine;

  end architecture rtl;]
```

Listing 10: Testbench

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use STD.textio.all; -- package for input-output of txt files

entity simulation is

end entity simulation;


architecture test of simulation is

    signal CLK100MHZ    : std_logic := '0'; -- master clock of the fpga
    signal uart_txd_in  : std_logic:= '0'; -- uart receiver input
    signal uart_rxd_out : std_logic:='0'; -- uart transmitter output
    file file_VECTORS   : text; -- txt input file
    file file_RESULTS   : text;  -- txt output file

begin

  DUT : entity work.top -- device under test
    port map (
      CLK100MHZ       => CLK100MHZ,
      uart_txd_in     => uart_txd_in,
      uart_rxd_out    => uart_rxd_out);

  -- generate clock
  CLK100MHZ <= not CLK100MHZ after 5 ns; --100Mhz -> 10 ns period

  -- waveform generation
  WaveGen_Proc : process
    variable v_ILINE        : line; -- line of txt input file
    variable v_OLINE        : line; -- line of txt output file
    variable i_data_integer : integer   := 0;
    variable o_data_integer : integer   := 0;
    variable i_data_slv     : std_logic_vector(7 downto 0) := (others => '0');
    variable o_data_slv     : std_logic_vector(7 downto 0) := (others => '0');
    variable count          : integer := 0;
    constant c_WIDTH        : natural   := 8;
    constant divisor        : natural   := 867; -- clock frequency/baudrate

  begin
    -- insert signal assignments here
    file_open(file_VECTORS, "input_int.txt", read_mode); -- open input file
    file_open(file_RESULTS, "output_results.txt", write_mode); -- open output file
    wait until rising_edge(CLK100MHZ);
    data : while not endfile(file_VECTORS) loop -- til the end of the file
      readline(file_VECTORS, v_ILINE); -- reading a line
      read(v_ILINE, i_data_integer); -- reading the first number of the line
      i_data_slv         := std_logic_vector(to_signed(i_data_integer, i_data_slv'length));
      -- slave means the same but to std logic vector
    wait until rising_edge(CLK100MHZ);
        uart_txd_in <= '0';

        while (count < divisor) loop -- waiting for one baudrate
                        wait until rising_edge(CLK100MHZ);
                        count := count + 1;
                end loop;
        count := 0;

        i_o : for i in 0 to 7 loop -- send the 8 bits to uart receiver
                while (count < divisor) loop
                        wait until rising_edge(CLK100MHZ);
                        uart_txd_in     <= std_logic(i_data_slv(i)); -- 1 bit at a time
                        count := count + 1;
                end loop;
        count := 0;
        end loop i_o;

        uart_txd_in <= '1'; -- send idle bit
        wait until rising_edge(CLK100MHZ);
    -- here fir filter is acting
        wait until uart_rxd_out = '0';


        while (count < divisor) loop -- waiting for one baudrate
                        wait until rising_edge(CLK100MHZ);
                        count := count + 1;
                end loop;
        count := 0;

        o_i : for k in 0 to 7 loop -- collect the 8 bits from uart transmitter
                while (count < divisor) loop
                        wait until rising_edge(CLK100MHZ);
                        o_data_slv(k)     := std_logic(uart_rxd_out); -- 1 bit at a time
                        count := count + 1;
                end loop;
        count := 0;
        end loop o_i;

      o_data_integer := to_integer(signed(o_data_slv));
      write(v_OLINE, o_data_integer, left, c_WIDTH); -- write on a line of the output file
      writeline(file_RESULTS, v_OLINE); -- insert the line
    end loop data;

    file_close(file_VECTORS); -- closing the two txt files
    file_close(file_RESULTS);
    wait;
  end process WaveGen_Proc;


end architecture test;
```

References

[1] Bryan Mealy, Fabrizio Tappero, Free Range VHDL, http://freerangefactory.org/pdf/df344hdh4h8kjfh3500ft2/free_range_vhdl.pdf (2018)