UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Master degree in Physics of Data

Course: Management and Analysis of Physics Datasets, module A

FIR filter implementation on FPGA

Group 7

Buriola Lorenzo 2021860

Campesan Giulia 2027592

Mellino Daniele 2013373

Anno Accademico 2020/2021

Github Repo https://github.com/LorenzoBuriola/MAPD_A

1 Aim

The goal of this lab project is the implementation of a simple FIR-filter co-processor on a FPGA. Our task is to write down the VHDL source code that will be synthetized in the hardware.
The top entity implemented is shown in Fig. 1. The filter block is in communication with the outside through a UART unit: the data are transmitted from the PC (through the python script) to the receiver, they are filtered by the FIR filter and then sent back to the PC by the transmitter. The data flow is synchronized by the master clock, set at 100 MHz, and the UART operates with a baudrate generator frequence of 115200 pulses/s.

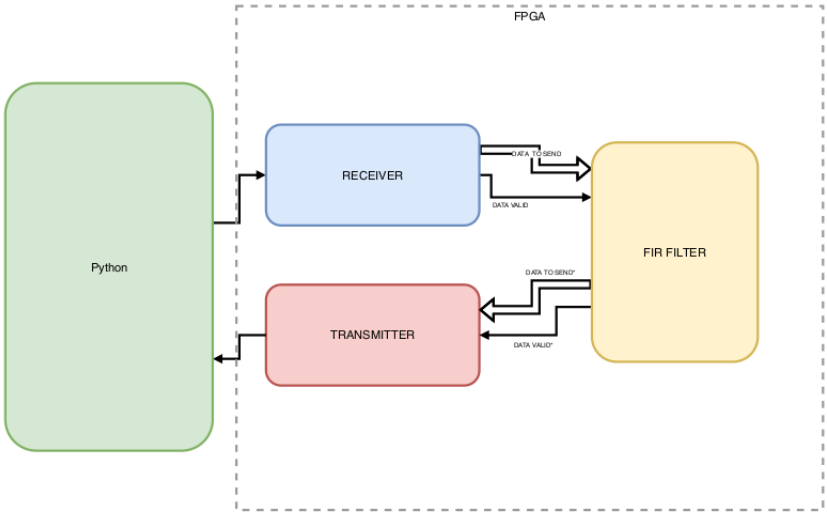


Figure 1: PC-FPGA interface

2 FIR filter: how it works and our architecture

The finite impulse response(FIR) filter is a filter whose impulse response is of finite duration. Given an input sequence of N data samples $\{x_i\}_{i=1,..,N}$, the output of a k-th order fir filter $\{y_i\}_{i=1,..,N}$ is given by:

$$y[n] = C_0 x[n] + C_1 x[n-1] + \cdots + C_{k-1} x[n-k+1] = \sum_{i=0}^{k-1} C_i \cdot x[n-i] \tag{1}$$

The coefficients C_i represents the impulse response at the i^{th} instant, they characterize the behavior of the filter. The typical structure of the filter is schematized in Fig. 2. Where we can see the presence of four registers that save the coefficients values and four data vectors.

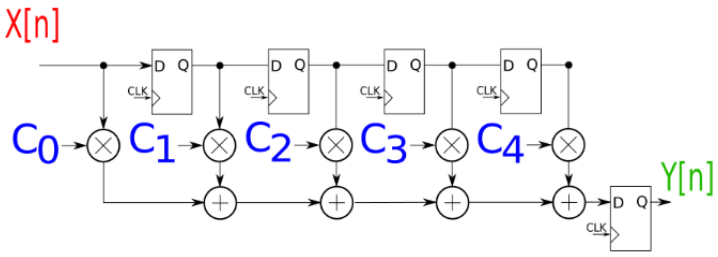


Figure 2: 4^{th} order FIR filter

We choose to implement a 4^{th} order low pass FIR filter with cutoff frequency of 25 Hz ($0.1 \cdot f_{sampling}/2$). In order to compute the relative filter coefficients, we use the **firwin** method of the **scipy.signal** library.

It takes in input the number of coefficients (5 in our case), the cutoff frequency($f_c = 25$ Hz) and the sampling frequency of the signal($f_{sampling} = 500$ Hz).

```
coeff = np.asarray(firwin(taps=5, fc=0.1/2, fs=500))
```

The obtained coefficients are:

C_0	C_1	C_2	C_3	C_4
0.034	0.240	0.452	0.240	0.034

Table 1: Values of coefficients

Note that coefficients are symmetric and they sum up to one.

We then performed, using the **freqz** function of the **scipy.signal** package, the response in the frequency domain:

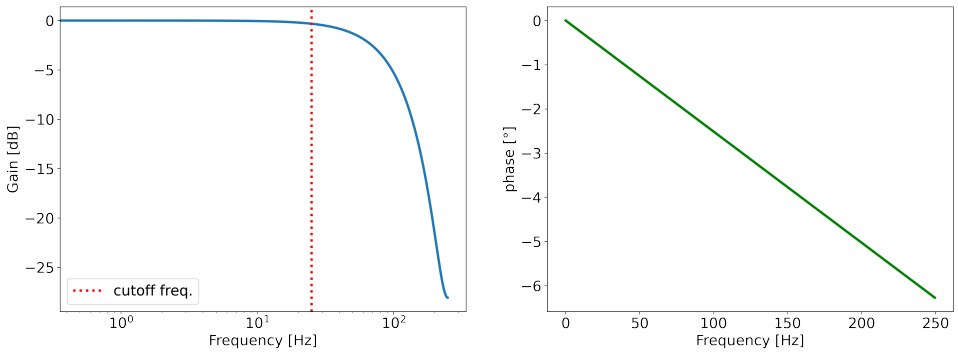


Figure 3: Frequency response of the filter

In Fig. 3 the amplitude and phase as functions of the frequency are reported. Looking at the gain trend, one can recognize the typical low-pass behaviour. Note that using only five taps the bandwidth seems to be wider than the one expected knowing the cutoff frequency. Adding more and more taps the filter behaviour starts to resemble the ideal behaviour as can be seen in Fig. 4. As expected the phase response is linear given the nature of the FIR filter. This means that each frequency is shifted in time such that the result is an overall delay of the signal. From the theory we know that the delay is linked to the number of taps by the relation:

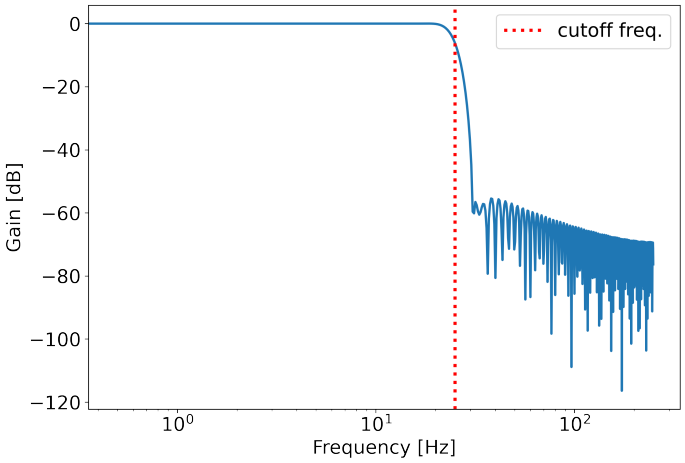


Figure 4: Frequency response of a filter with 150 taps

$$delay = \frac{N_{taps} - 1}{2} \frac{1}{f_{sampling}}$$

Using 5 taps, in order to compare it with the input one, the output signal will be shifted left by two samples.

3 Data format and precision

The width of the data that can be received and transmitted by the FPGA is 8 bits. However, the arithmetical operations performed by the FIR filter lead to an increase in the data size.
In fact, when considering two binary numbers a1, a2 represented by N1, N2 bits:

- a1+a2 is represented by $max\{N1, N2\} + 1$ bits,
- a1*a2 is represented by $N1 + N2$ bits.

Since we are processing our 8-bits input data with a 5 taps FIR-filter, the output will be 19 bits long, that we need to recast into an 8-bits representation.

In order to simplify arithmetic operations in the FPGA, only integer numbers are used. Since the FIR-filter coefficients are clearly float, we need to scale them up in order to represent them as integers. Considering that the coefficients values are always less than 1, to retrieve 8-bits signed integers, we scale-up the coefficients of a factor Q=2^7. This will allow us to consider the 7 LSBits of the output number as its fractional part. For the coefficient, the values actually used by the VHDL code, are obtained through an approximation leading eventually to get a number that may differ of $0.5 \cdot 2^{-7}$ from the expected one.

After the filtering process, taking into account the 7 bit right shift explained before, the results are integer numbers represented by 19-7=12 bits from which we need to select the 8 most significant ones, in which is included the information on the sign (see fig. 5).

The used coefficients are normalized such that they sum up to 1. That means that the the integer part of the output, when giving in input 8-bits signed numbers, always belongs to [-127, 127], therefore it could be precisely represented by 8 bits. Then, we could recast our output into 8 bits just by taking the 8 LSBs of the 12 bits output binary number, without any precision loss.

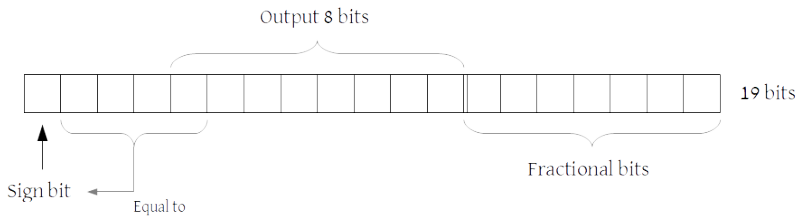


Figure 5: Binary data bits structure

Since also the python generated input data are smaller than one, we perform the same 2^7 multiplication before feeding them to the FPGA. So, after we get the output results, we shift them back, to get a comparison with the python simulation results.

4 VHDL Implementation

Here the VHDL code for the FIR-filter is shown. Two different implementations were tried, one using a State Machine and one using different processes. Since the most successful during the synthetization in the hardware is the State Machine, we will display the complete code of that implementation.

4.1 State Machine

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

entity fir_filter_SM is
    port (
      clk           : in std_logic;
```

```
      valid_input    : in std_logic;
      busy           : in std_logic;
      data_input     : in std_logic_vector(7 downto 0);
      valid_output   : out std_logic;
      data_output    : out std_logic_vector(7 downto 0)
    );
    end fir_filter_SM;
```

The input signals of the FIR-filter entity are the master clock of the system, the input data and input valid from receiver and the busy from the transmitter; the output signals are the output data and valid for the transmitter.

```
architecture rtl of fir_filter_SM is

    -- coefficients
    signal coeff_0      : signed(7 downto 0) := to_signed(4, 8);
    signal coeff_1      : signed(7 downto 0) := to_signed(31, 8);
    signal coeff_2      : signed(7 downto 0) := to_signed(58, 8);
```

Since the coefficient are symmetric we just need to retrieve 3 coefficients instead of 5. This allows us to optimize the computations limiting the number of multiplications. Coefficients are stored as signed signals with 8 bits.

```
    -- sum1
    signal sum0         : signed(8 downto 0):= (others => '0');
    signal sum1         : signed(8 downto 0):= (others => '0');

    -- mul
    signal mul0         : signed(16 downto 0):= (others => '0');
    signal mul1         : signed(16 downto 0):= (others => '0');
    signal mul2         : signed(15 downto 0):= (others => '0');

    --sum2
    signal sum_sf        : signed(17 downto 0):= (others => '0');

    --sum3
    signal sum_tot       : signed(18 downto 0):= (others => '0');
```

Here the needed signals are defined. They are used to store the results of sums and multiplications. Note that each signal has a different bit length depending on what it has to store.

```
    type data_type    is array (0 to 4) of signed(7  downto 0);
    signal proc_data  : data_type:= (others => (others => '0'));
```

An array data type is defined to store the input data. This allows to keep track up to five input numbers at a time which are used in the FIR-filter computation.

```
    type state_type is (IDLE, Input, Sum_1, Mul, Sum_2, Sum_3, Output);
    signal state : state_type := IDLE;
```

The State Machine will have an IDLE state and six other states corresponding to an equal number of simple operations.

```
begin
    main : process (clk) is
    begin
        if rising_edge(clk) then
            case state is
                when IDLE =>
                    valid_output <= '0';
                    if valid_input = '1' then
                        state <= Input;
                    end if;
                when Input =>
                 proc_data <= signed(data_input)&proc_data(0 to proc_data'length-2);

                    state <= Sum_1;
```

In the IDLE state we wait for the valid input signal to become '1' to start the data acquisition in the Input state. The **proc_data** is sequentially filled with the current input data value and the previous four ones to always store the last five data values.

```
                when Sum_1 =>
                    sum0 <= resize(proc_data(0), 9) + resize(proc_data(4), 9);
                    sum1 <= resize(proc_data(1), 9) + resize(proc_data(3), 9);

                    state <= Mul;

                when Mul =>
                    mul0 <= sum0 * coeff_0;
                    mul1 <= sum1 * coeff_1;
                    mul2 <= proc_data(2) * coeff_2;

                    state <= Sum_2;

                when Sum_2 =>
                    sum_sf <= resize(mul0, 18) + resize(mul1, 18);

                    state <= Sum_3;

                when Sum_3 =>
                    sum_tot <= resize(sum_sf, 19) + resize(mul2, 19);

                    state <= Output;
```

After the input state these four states are executed consecutively. As said before, the fact that the FIR-filter is symmetric allows to reduce the number of multiplications. In fact, we sum up the input values that share the same coefficient and then we multiply them for the taps value. After that, the obtained results are summed up in two different states.

```
                when Output =>
                                valid_output    <= '1';
                        data_output     <= std_logic_vector(sum_tot(14 downto 7));
                        if busy = '0' then
                            state           <= IDLE;
```

```
                        end if;

                when others => null;
            end case;
        end if;
    end process main;

end architecture rtl;
```

In the Output state the valid output is set to '1' and the output signal is filled following what we have discussed in Section 3, the process waits for the **busy** signal from the transmitter to be '0' to return to the IDLE state.

4.2 Processes implementation

With the purpose of optimize our code for further implementations we tried to pipeline the different arithmetic operations. To do that we tried to use a multiple processes approach. We have tried with 5 and 3 processes, for each of them the simulation results with vhdl are the same of the one using the State Machine. The 5 process version, however when implemented on FPGA, after a casual time starts to return wrong values.

5 Signal generation

In order to compare and validate our FPGA results, we simulate a noisy cosinusoidal signal through a python script. The main component amplitude is 0.5 and its frequency is 2 Hz. The noise follows a cosinusoidal behaviour too, in order to have a better understanding of the frequency spectrum before and after the filtering. We set its amplitude to 0.05 and its frequency to 200 Hz, pretty over the cut-off frequency.

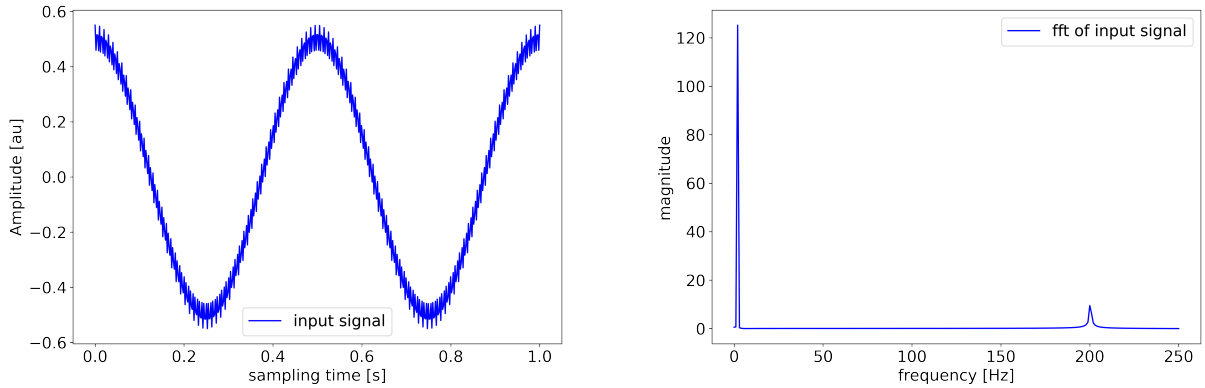


Figure 6: Python generated signal and its Fourier analysis

This sampled signal is converted to a stream of unsigned integers (as said before, through a 7 bits left-shifting operation), in order to feed it to the FPGA.

6 Python and GHDL behavioural simulation

6.1 Python simulation

Once defined the input signal and the coefficients, we proceed with the implementation of the filter in python. We performed both a simulation of the 'ideal', i.e. with python double-precision, behaviour of the filter and of the FPGA outcome (8-bits precision).

At first, we retrieved the ideal behaviour performing the mathematical operations with float numbers (signal and coefficients).

```
index = int((taps-1)/2)
filt_sig = np.zeros_like(sig)
for i in range(sig.size-taps+1):
    filt_sig[i+index] = np.dot(sig[i:taps+i], coeff[::-1])
```

Then to obtain the FPGA output, we filtered the input signal with the following code, in which we perform the mathematical operations as they are done by the FPGA itself (keeping track of the precision) and select the final 8-bit output as discussed previously.

```
for i in range(taps):
    coeff_int[i] = (round(coeff[i]*2**7, 0))

y = np.zeros_like(sig)
sim_filt_sig = np.zeros_like(sig_uns)
for i in range(sig_bin.size-taps+1):
    y[i+index] = np.dot(sig_bin[i:taps+i], coeff_int[::-1])
    bin_19 = np.binary_repr(int(y[i+index]), 19)
    bin_8 = bin_19[4:12]
    int_8 = int(bin_8, 2)
    sim_filt_sig[i+index] = convert_tosigned(int_8, 8)
```

6.2 GHDL simulation

Before testing our code on the FPGA, we build a testbench of the top structure that reads the input signal, generated in python, through the receiver and gives it back after the filtering thanks to the transmitter. The top entity was simulated in ghdl and the outcome, displayed with GTKWave software, is reported in Fig. 7:

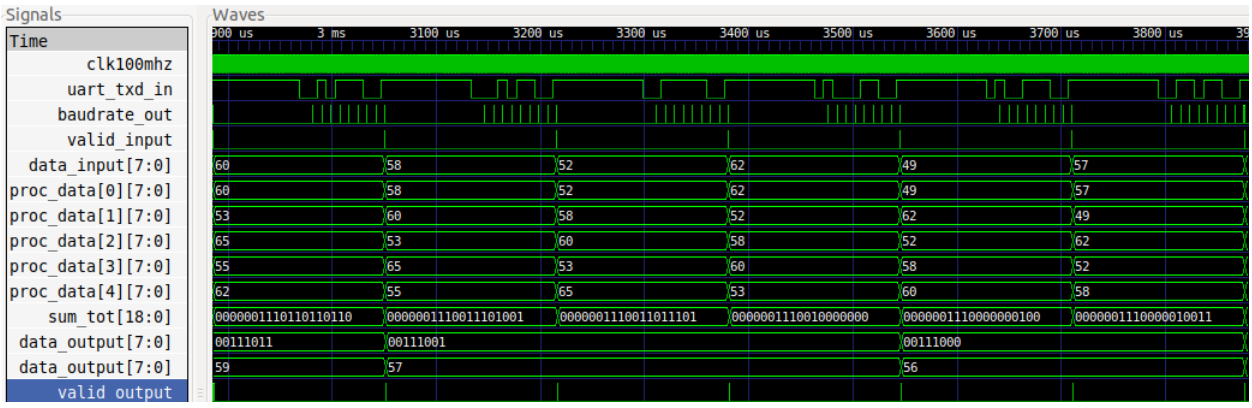


Figure 7: Waveform simulation of the whole configuration.

Focusing our attention to the State Machine, in Fig. 8 are shown the signals of FIR filter entity. One can see that the whole operation of weighted sum of the input data lasts for an interval of six clock cycles between one IDLE state and the next one.

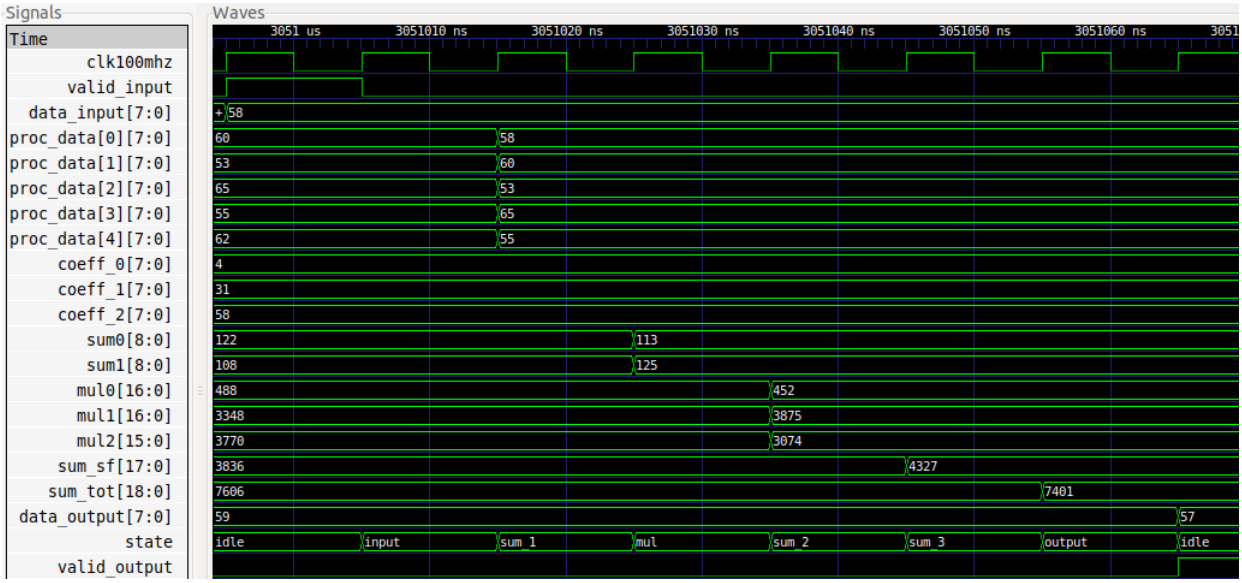


Figure 8: Waveform of FIR-filter State Machine

7 FPGA results

In this section we analyse the output data coming from the FPGA. The signal and its Fourier spectrum, obtained with FFT, are reported in Fig. 9. Remembering what was said about FIR-filter delay in Section 2, in order to compare the input signal with the FPGA output we plotted the filtered value, shifting it left by two sample. The filtering turns out to be executed successfully as we can infer from the one-peak trend of the spectrum analysis.

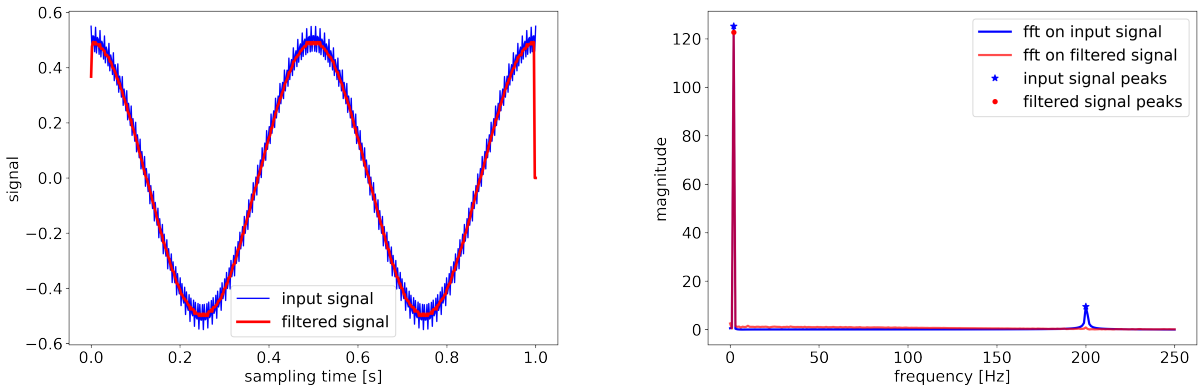
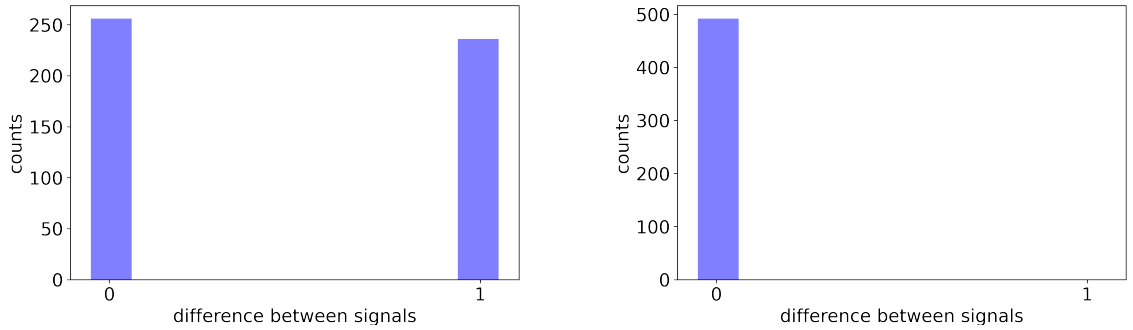


Figure 9: Original and filtered signal in time and frequency domain

In order to get further information on the FPGA implementation outcome, we compare its results both with the python double-precision ones (ideal behaviour) and with the python simulation, that approximates the output data with 8-bits precision. This comparison is resumed in the table 2, that reports some values as example, in which we considered the output values before the 7-bits right-shifting.



(a) Difference between the python double-precision ideal behaviour and the FPGA output

(b) Difference between the python 8-bits precision signal and the FPGA output

Figure 10: Distributions of the difference between simulation and FPGA output

Ideal sig. (python)	Sim. (python)	FPGA	Ideal - FPGA	Sim. - FPGA
53	53	53	0	0
52	51	51	1	0
52	51	51	1	0
50	50	50	0	0
49	49	49	0	0
48	48	48	0	0
47	46	46	1	0
47	46	46	1	0
44	44	44	0	0

Table 2: Comparison between python and FPGA signals

In the histogram in Fig. 10.a we report the distribution of the difference between the ideal behaviour and the FPGA output: about one half of the counts is in the 0-centered bin and the other half is in the 1-centered bin as a result of the truncation that the FPGA operates on the output. It should be taken into account that this difference has to be converted through right-shifting: it corresponds then to an error of order 2^{-7}. On the other hand, in Fig. 10.b we report the distribution of the difference of the FPGA output and of the python simulated 8-bits precision signal: all the counts are encountered in the 0-centered bin, validating our FPGA implementation.

As a further confirm that the FPGA output obtained is the correct one, we compared it with the ghdl simulation: they turn out to be equal.

8 Conclusions

We implemented a simple FIR-filter co-processor on FPGA using a State Machine and, before testing the code on the hardware, we simulated its behaviour with ghdl. We validated our results through the Fourier spectral analysis of the output signal and through the comparison of it with both python-simulated ideal behaviour and the ghdl simulation: both confirm that our implementation displays the expected behaviour.