Dario Barone, 2012265
Hilario Capettini, 2013031
Elena Leonelli, 2028635
Marika Sartore, 2017916

FPGA Implementation of FIR Filter

MANAGEMENT AND ANALYSIS OF PHYSICAL DATA MOD. A
April 30, 2021

Abstract

In this project we implemented a FIR filter in VHDL language, tested using GHDL and GTKwave and synthesized in a Xilinx Artix-7 FPGA via Vivado. We generated a noisy signal using a Python script, and sent it to the filter using the UART protocol. Eventually, we compared the filtered signal with the output of a FIR filter implemented in Scipy, in order to check the behaviour of the FPGA FIR filter.

1 Introduction

The FIR filters are one of the primary types of digital filters used in Digital Signal Processing applications. FIR stands for *Finite Impulse Response*, which means that the impulse response is of finite period and settles to zero in finite time. There is no feedback in the FIR and this guarantees that the impulse response will be finite.

As an alternative to FIR, there are *Infinite Impulse Response* filters which instead use feedback: when an impulse is given as input, the output theoretically continues to respond indefinitely.

The order of a filter represents the length of the filter's window: the impulse response of an N-th order discrete time FIR filter takes $N+1$ samples before it then settles to zero.

1.1 FIR filter operating principle

The realization of FIR filter is based on the implementation of this expression [1]:

$$\begin{aligned} y[n] &= h[n] * x[n] \\ &= h_0 x[n] + h_1 x[n-1] + \cdots + h_N x[n-N] \\ &= \sum_{i=0}^{N} h_i \cdot x[n-i] \end{aligned} \tag{1}$$

where:

- $x[n]$ is the input signal, which is a discrete sequence of numbers. Signal samples are spread equally in time because the sampling frequency is fixed;
- $y[n]$ is the output signal, which is a discrete sequence of numbers of the same length as the input;
- N is the order of the filter;
- h_i is a coefficient of the filter: it can also be seen as the value of the impulse response at the i-th instant of a FIR filter.

From a mathematical point of view, applying the filter means making a convolution of the filter impulse response (coefficients h_i) with the sampled data points $x[n]$ and the result is the output $y[n]$.

In Figure 1 is shown the basic FIR filter diagram with N length: the filter is designed with a series of delays, multipliers and adders.

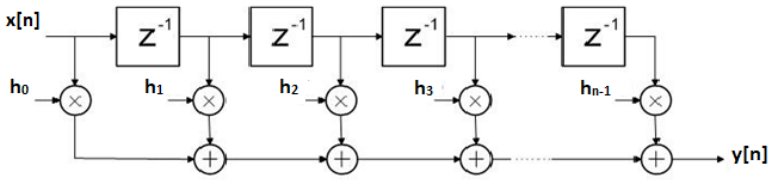


Figure 1: FIR filter diagram with N length.

The delay line allows the signal to be delayed by the number of samples processed by the filter at a time. It is composed by a set of memory elements that implement the Z^{-1} delay. Time series of samples come along the line, so the samples are available at the same. Then they can be used to implement a convolution by multiplying each sample by its coefficient (in number equal to the number of samples) and then a sum is performed (that corresponds to an integration in discrete time) to obtain the output.

This is done sequentially on the whole signal, repeating the same procedure shifting all the data by one, *i.e.* the window is moved along the data.

1.2 Effects of taps

A FIR tap is a coefficient/delay pair. The number of FIR taps is an indication of the number of calculations present, the amount of memory required to implement the filter and the amount of "filtering" the filter can do.

The values of the coefficients depend on: number of taps wanted, frequency above or below which to cut the frequencies (or the frequencies that define the band limits, in the case of a band-pass filter) and shape of the window.

Concerning the shape of the window, it determines a trade-off between selectivity (width of the transition band in which frequencies are not completely accepted or rejected) and suppression (total attenuation of the frequencies to be rejected). In our case, we are going to use the standard Hamming window, which is formed by using a raised cosine, optimized to minimize the nearest side lobe.

To briefly visualize the effects of FIR coefficients, we show some plots that compare different number of taps or different cut-off frequencies, because in our case we have decided to implement a low pass filter.

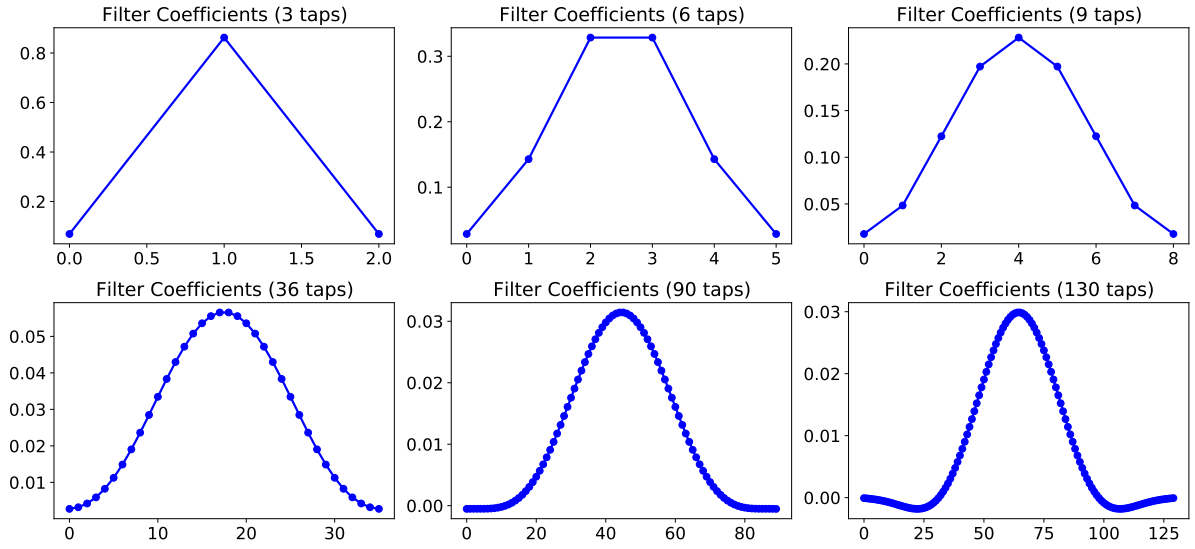


Figure 2: Comparison between different number of taps.

In Figure 2 the coefficients are computed at the same cut-off frequency ($f = 15\,\text{Hz}$), changing the number of taps requested.Increasing the number of filter taps yields more accurate information about the shape of the signal, because the filter takes into account more samples at a time. On the other side, this is more computationally demanding and the overall gain of the filter is lower.

The comparison between frequency responses is shown in Figure 3, where the frequency response of the filter is determined from the Fourier transform of an impulse response.

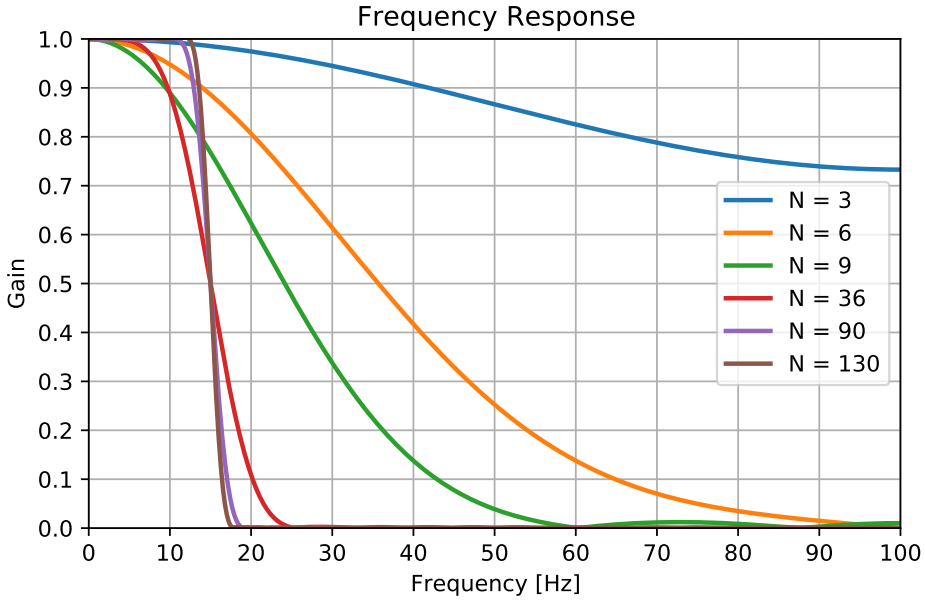


Figure 3: Frequency responses using different number of taps.

We can see that, increasing the number of taps, the roll-off of the filter sharpens but several sidelobes appear. With a sharper filter it is possible to obtain a narrower transition band.

Then, in Figure 4, the number of taps is fixed ($N = 90$) and the cut-off frequency is changed.
The higher the cut-off frequency, the more irregular the distribution of the coefficients and the lower the cut-off frequency, the smoother the distribution:

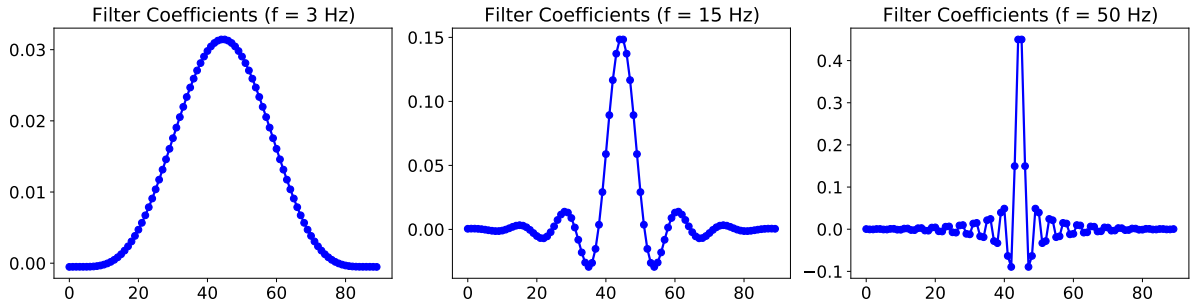


Figure 4: Comparison between different cut-off frequencies.

The frequency responses for the three cases is shown in Figure 5 where the behavior of the low pass filter is clear looking at where, in each case, the cut frequencies are.

From the plots above, we can see that the taps are always symmetric: symmetrical filter coefficients produce linear phase response, which is a characteristic of FIR filters. They delay the input signal but they do not distort its phase.

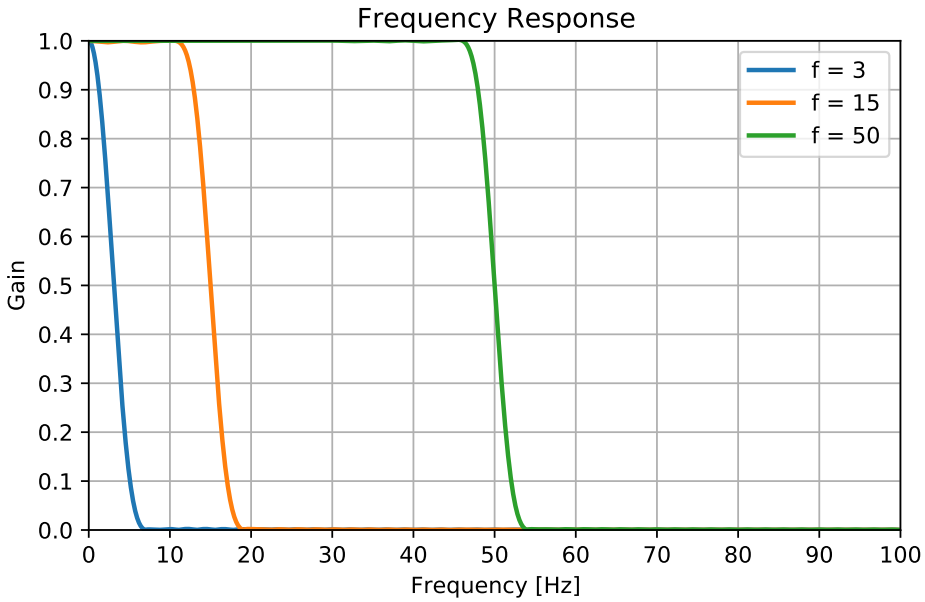


Figure 5: Frequency responses using different cut-off frequencies.

1.3 Structure of the project

In our project we are going to implement a FIR fiter on a FPGA, following the scheme in Figure 6.

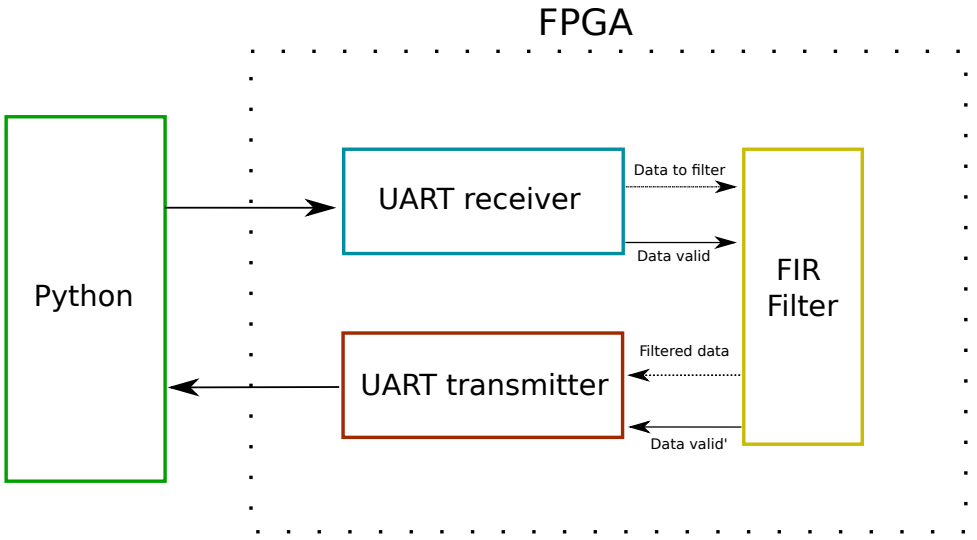


Figure 6: Diagram of the project.

Firstly, to communicate with the FPGA we have chosen to use the UART protocol, which is exposed in Section 2. Then, in Section 3 we present our FIR filter implementation in VHDL and in Section 4 and Section 5, respectively, the data conversion and taps conversion we have applied in order to pass data through the FPGA. In Section 6 there is our estimation of the errors introduced in our process. Once the setup was done, we generated the signal in Section 7 and we designed a FIR filter with properties fitted on our signal (Section 8). Then, in Section 9 we implemented a FIR filter with the same features also in Python, in order to make a comparison with the signal filtered by the FPGA (Section 10). Lastly we report our conclusions in Section 11.

2 UART protocol

The Universal Asynchronous Receiver-Transmitter (UART) is one of the simplest interfaces to transmit and receive data in an asynchronous manner using a single wire. There exist other alternatives for interfaces such as JTAG and Ethernet which are faster but considerably more complex.

The UART interface can be divided in two parts, the UART transmitter (UART-tx), which takes a byte of data and transmits the individual bits in a sequential fashion, and the UART receiver (UART-rx), which transforms the individual bits into a complete byte. To perform this task in an asynchronous manner (tx and rx do not share the clock), some parameters must be fixed and shared between both parts: the bit speed (baudrate), character length and the start/stop bit. For this project we choose a baudrate of 115 200 bit s^{-1}, a character length of 8 bits, a start bit equal to low (0) and a stop bit equal to high (1).

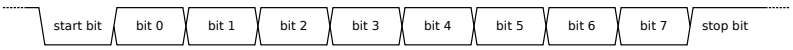


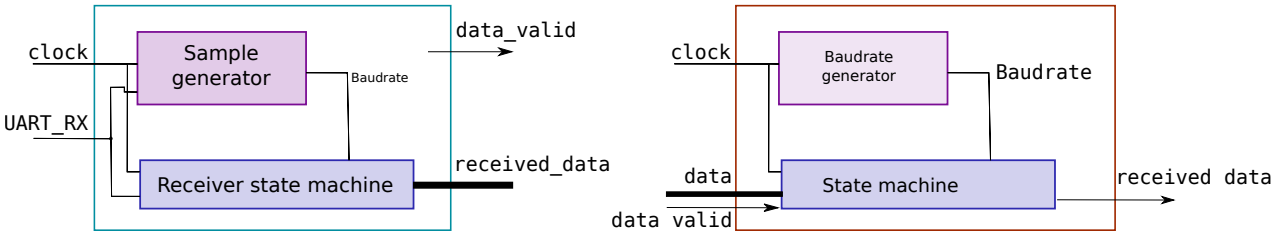
Figure 7: UART data stream

2.1 UART receiver implementation

The UART-rx is composed by two main entities as can be seen in Figure 8a, the sample generator and a state machine. The role of the sampler generator is to provide the UART-rx with the data sampling pulses, those pulses must be such to allow the sampling of the incoming bit at its half lifetime. To accomplish this task, when the initial bit of the message is detected, 8 pulses are emitted with a delay of half the length of the bit time. The UART-rx state machines task is to collect each of the 8 individual bits and convert them into a single byte. Once the message is complete it is sent to the filter in a parallel fashion, together with a data valid pulse which last a clock cycle.

2.2 UART transmitter implementation

The UART-tx is composed by two entities as depicted in Figure 8b, the baudrate generator and a state machine. The baudrate generator is a simple counter responsible for the generation of a pulse every 867 clock cycles, this pulse is an input of the transmitter state machine and trigger the changes from one state to the other, allowing a sequential sending of the message from the filter to the computer. The state machine is usually in the default state, called "idle", until a `data valid` takes the value one activating the reading and sending process, the first bit to be send is the start followed by the 8 bits message (from least significant to most significant bit) an finishing with the stop bit as showed in Figure 9.



(a) UART receiver

(b) UART transmitter

Figure 8

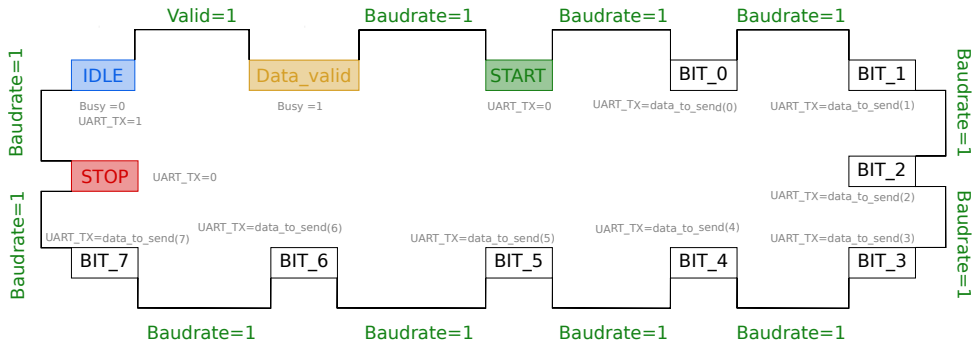


Figure 9: UART transmitter state machine

2.3 UART simulation

For the purpose of controlling the the UART behaviour, simulations of the VHDL code where conducted using the GHDL open-source simulator. In Figure 10 can be observed the simulation of the receiver module and in Figure 11 the simulation for the transmitter visualized with GTKwave.

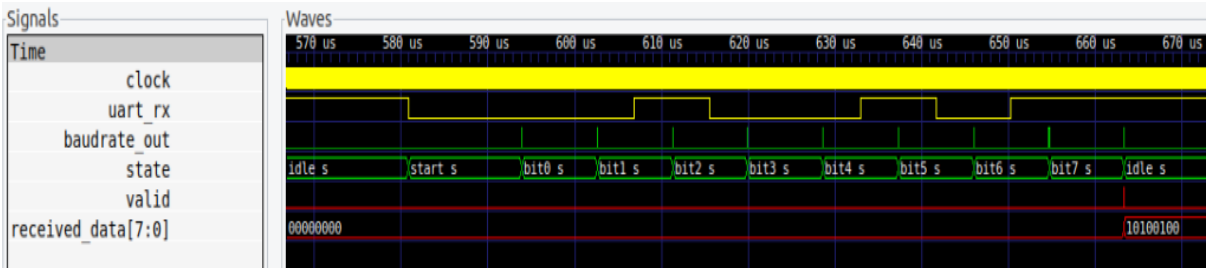


Figure 10: UART receiver simulation. In yellow can be seen the inputs, in green the internal signals and in red the output signals.

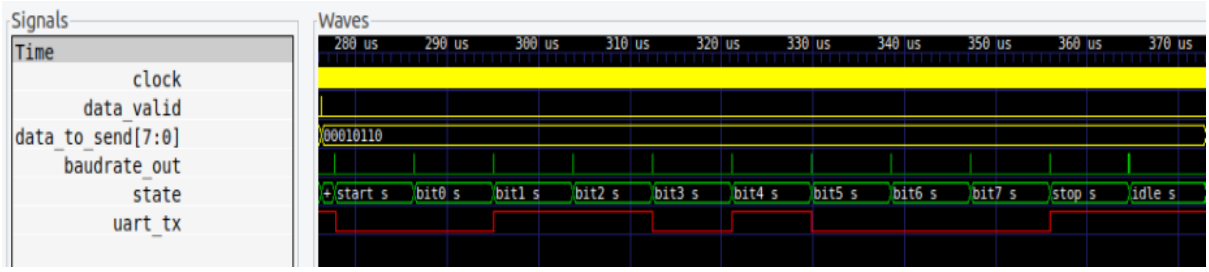


Figure 11: UART transmitter simulation. In yellow can be seen the inputs, in green the internal signals and in red the output signal.

3 Fir Filter implementation

3.1 VHDL code

We implemented the FIR filter in VHDL with a main process based on a Finite State Machine [2], driven by the clock signal. A FSM is convenient in this case because it is sequential and thus it is easy to follow the logic of the consequential operations that take place inside the FIR filter. Moreover, it is easy to track errors if they come out. Our FIR filter implementation has 9 taps, but it is easy to change this number if needed.

In our VHDL code we imported the `ieee` library. In particular, we used the `ieee.std_logic_1164.all` standard to deal with the variables connected to the UART, and the `ieee.numeric_std.all` standard to treat the data as `signed` types. This is necessary to perform calculations inside the FIR filter, since the values of the signal are both positive and negative. The `ieee.numeric_std.all` standard uses the *2-complement* representation, where `signed` value ranges from -2^{N-1} to $2^{N-1}-1$ (where N is the number of bits used to represent the `signed`).

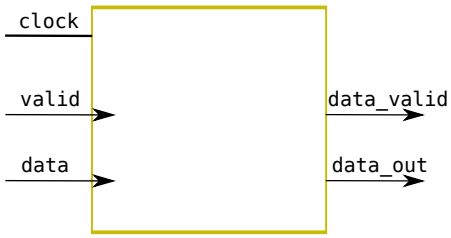


Figure 12: FIR diagram

The FIR block takes as input, from the UART receiver, the data valid (*i.e.* a signal that lasts clock period and indicates when the UART receiver sends the data) and an eight-bit signal , and returns as output another data valid and the eight-bit filtered signal to the UART transmitter. The inputs and the outputs of the FIR block are all `ieee.std_logic` variables: `clock` (in), `valid` (in) and `data_valid` (out) are all `std_logic` types; `data` (in) and `data_out` (out) are `std_logic_vector` of 8 bits. A scheme of the FIR input and output variables is presented in Figure 12.

Inside our FIR filter are defined four internal signals, used for store the results of each operation done. All these signals are of `signed` type:

- **`data_pipe`**, an array of size 9 made by 8-bit signed numbers used to store the data in input;
- **`coeff`**, an array of size 9 made by 8-bit signed numbers used to store the taps of the filter;
- **`mult`**, an array of size 9 made by 16-bit signed numbers used for the multiplications;
- **`adder`**, a signed number of 24 bits used for the final summation.

```
main : process (clock,valid) is
 begin  -- process main
   if rising_edge(clock) then -- rising clock edge

     case state is

       when s0 =>      --initialized to s0 state
         data_valid  <= '0';
         if valid = '1' then
           state <= s1;
         end if;

       when s1 =>
         data_valid  <= '0';
         data_pipe  <= signed(data)&data_pipe(0 to data_pipe'length-2);

         if valid = '0' then
           state <= s2;
         end if;

       when s2 =>
         mult(0) <= data_pipe(0)*coeff(0);
         mult(1) <= data_pipe(1)*coeff(1);
         mult(2) <= data_pipe(2)*coeff(2);
         mult(3) <= data_pipe(3)*coeff(3);
         mult(4) <= data_pipe(4)*coeff(4);
         mult(5) <= data_pipe(5)*coeff(5);
         mult(6) <= data_pipe(6)*coeff(6);
         mult(7) <= data_pipe(7)*coeff(7);
         mult(8) <= data_pipe(8)*coeff(8);

         if valid = '0' then
           state <= s3;
         end if;

       when s3 =>
         adder <= (resize(mult(0),24) + resize(mult(1),24) +
                   resize(mult(2),24) + resize(mult(3),24) +
                   resize(mult(4),24) + resize(mult(5),24) +
                   resize(mult(6),24) + resize(mult(7),24) +
                   resize(mult(8),24));

         if valid = '0' then
           state <= s4;
         end if;

       when s4 =>
         data_out <= std_logic_vector(adder(16 downto 9));
         data_valid  <= '1';
         if valid = '0' then
           state <= s0;
          end if;

       when others => null;
     end case;

   end if;

 end process main;
```

Source Code 1: FIR filter state machine

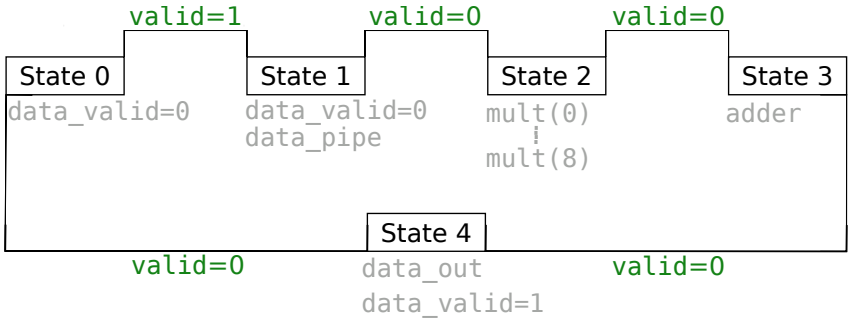


Figure 13: FIR state machine. In black the machine states, in green the keys to each state and in grey the definitions in each state.

State 0

This state is the start one. When the data valid in input becomes 1, then you move to state 1.

State 1

In this state the input data coming from the UART receiver are converted in a signed number, which is stored in `data_pipe` as its first element. All the other elements previously stored in `data_pipe` are shifted by one, so that the 9-th element previously stored is overwritten and the size of `data_pipe` is always 9.

When the data valid in input returns to 0, then you move to state 2.

State 2

This state is used to store in `mult` the result of the multiplications between each element of `coeff` and the corresponding element of `data_pipe`. The size of the vectors in `mult` is 16 bits.

If at the next clock's rising edge data valid in input is still 0, then you move to state 3.

State 3

In this state the sum of all the components of `mult` array is stored in `adder`. In order to avoid overflow, `adder` is a signed number represented with 24 bits, and we used `resize()` function to enlarge the size of `mult` components to reach 24 before the sum.

If at the next clock's rising edge data valid in input is still 0, then you move to state 4.

State 4

This is the last state, where the output must be send to the UART transmitter as a `std_logic_vector` of 8 bits. Because `adder` is a 24-bits vector, we need to resize it limiting as much as possible the loss of precision. Thus, we evaluated the largest number of bits that are needed to store in `adder` all the possible outcomes of the previous operations. The number M that has the largest length occurs when all the inputs in `data_pipe` are $(11111111)_2$:

$$M = (11111111)_2 \times \sum_{i=1}^{9} (h_i)_2 = (1111101110000111)_2 \tag{2}$$

where h_i are the taps coefficients. Defining

$$A = \lceil \log_2 M \rceil = 16 \tag{3}$$

as the index of the most significant bit of M, then the bits of `adder` that we must send as `data_out` are the ones with indexes in the range $[A, A-7]$. Thus, the value in output is `adder(16 downto 9)`.

It is important to notice that, due to this operation, we reduce the precision of our filtered data, because we truncate the $A - 7$ least significant bits of `adder`. This leads to an error, which will be estimated in Section Section 6.

In the same time of the output also the `data_valid` is sent. Then, the state machine returns to state 0, waiting for the next data valid in input.

3.2 FIR simulation

In order to check the behaviour of the FIR filter, we implemented a testbench. From the simulation results (Figure 14) it is possible to see that, in state 1, the first data received in input is correctly stored as the first element in `data_pipe`. Then, operations in states 2 and 3 are performed. In state 4 we can see that the output is sent, and at the same time the one-clock-period data valid is transmitted. We can also verify how the resizing of `adder` works, reducing the precision on `data_out`.

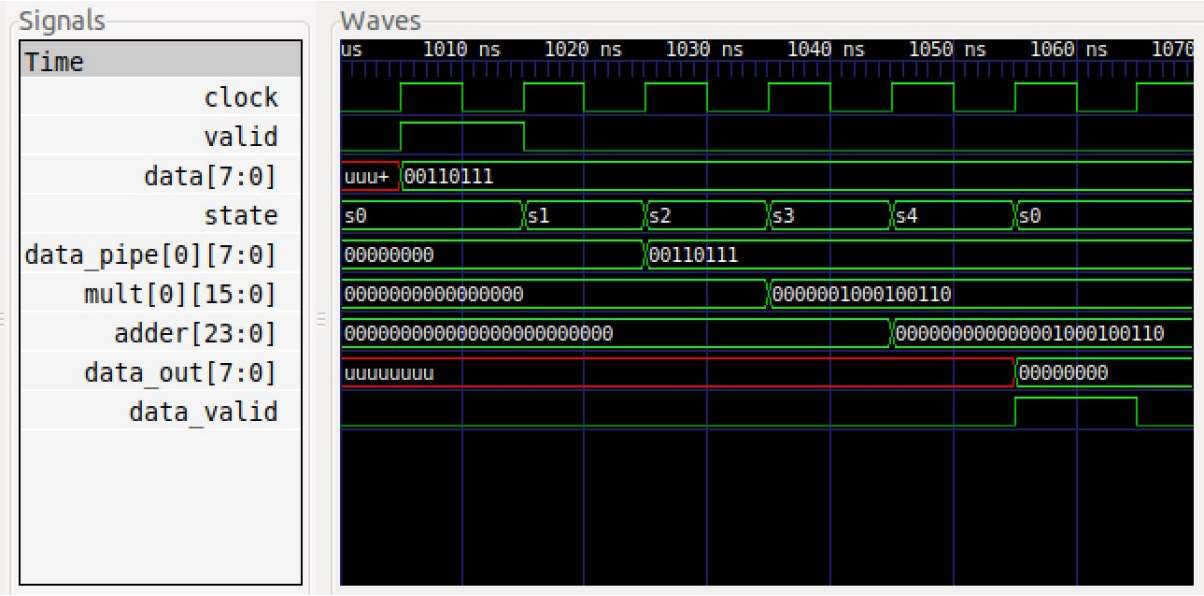


Figure 14: Simulation of the FIR filter's testbench shown with Gtkwave.

4 Data conversion

The data transfer from and to the FPGA must pass through the UART entities, which by construction deal with 8-bits data. This imposes a constraint on the format of both the raw and the processed data, that must fit within 8-bits. If the input signal is not already in this representation, a conversion must be performed. In our particular case (Python environment), the default is to use 64-bits variables to represent floating point numbers.

We have chosen to perform a custom cast from default variables to 8-bits ones, to be sure that the representation of the values is the same used inside the FPGA.

As explained above in Section 3, in order to manage both positive and negative numbers, the 2-complement representation has been adopted. This is a tricky notation, since, in this representation, the lowest negative number is greater than the highest positive one. We thus need to shift of all the negative numbers above the positive ones.

4.1 Rescaling and approximation

The magnitude of the variables must be rescaled keeping into account that there are only 7 bits available to represent the magnitude, implying that the representable values are the integers in the range $I_8 =$

$\left[-2^7, 2^7 - 1\right]$, and so every 64-bits value must be mapped into this range. We opt for a linear map from the domain of the signal to the range I_8, because it preserves the shape of the signal changing its amplitude (pure rescaling). The minor cardinality of the 8-bits set of values requires an approximation, that must be minimized in order to lose as small information as possible.

In order to accomplish the rescaling, we must compute the quantity Q that respects the following constraint:

$$\left\{x_i \cdot 2^Q\right\} \in \left[-2^7, 2^7 - 1\right] \qquad \forall\, x_i \in \text{signal} \tag{4}$$

The best value for Q is the one that maps the input signal into the whole range I_8: if the spanned interval is smaller, then the conversion is not optimal, because there is a loss on the precision bigger than required. For this purpose, we obtain the maximum Q inverting Equation 4:

$$Q = \log_2\left(2^7 - 1\right) - \log_2\left(\max_i |x_i|\right) \tag{5}$$

within the assumption that the input signal has a symmetric range around 0. If the input range is shifted, then the algorithm should still work, but not with optimal performance; in that case, a preprocessing to reset the offset of the signal would ensure better results.

We point out that, with this approach, the value of Q depends only on the amplitude of the input signal, so it can be computed in runtime before the filtering routine.

4.2 Conversions: back and forth

Once Q is computed, we can perform the conversion from 64-bits variables (labelled with x) to 8-bits ones (labelled with z). The conversion is ruled by the formula:

$$z_i = \begin{cases} \lfloor x_i \cdot 2^Q \rceil & \text{if} \quad x_i \geq 0 \\ \lfloor x_i \cdot 2^Q \rceil + 2^8 & \text{if} \quad x_i < 0 \end{cases} \tag{6}$$

The converted signal are ready to be sent to the FPGA and get processed by the filter. After the processing, the FPGA sends back data still in 8-bits format, so a reconversion to standard 64-bits floating point variables is needed. This operation is the inverse of Equation 6, and it is expressed by:

$$x_i = \begin{cases} z_i \cdot 2^{-Q} - 2^8 & \text{if} \quad z_i \geq 2^7 \\ z_i \cdot 2^{-Q} & \text{if} \quad z_i < 2^7 \end{cases} \tag{7}$$

5 Taps conversion

In order to set the values of the taps coefficients $\{h_k\}$, we use the utility function `firwin` provided by `scipy` package [3]. The values of the coefficients depend only on the number of taps N and on the cut-off frequency f_{cutoff}, which will be properly set and discussed in Section 7.

We need to convert the tap coefficients into signed 8-bits variables, in order to set their value inside the FPGA. Since the conversion strongly depends on the Q value, which in turn depends on the amplitude of the input signal, we would have that the converted values of the taps would depend on the amplitude of the processed signal. Though, this is something we want to avoid, because we want to have a FIR filter which filters, with same efficiency, all the signals having similar frequencies. Moreover, the setting of the taps in the FPGA is performed at compile time, while the Q value is computed at runtime.

Our solution is to convert the taps using a different Q_h value, computed on the taps only. In this way, we obtain that the taps will be always mapped to the range I_8, especially the maximum tap is always as large as $(01111111)_2$ (the maximum positive 8-bits value). All the other tap coefficients scale proportionally, so at the end all the tap coefficients will always have the same converted values, totally independently on the amplitude of the input signal.

This approach allows us to set N and f_{cutoff}, and consequently compute the converted 8-bits values of the taps at compile time, setting the FPGA implementation before having the signal. This, in turn, implies several benefits, which concatenate one with the other:

1. the maximum computable value M defined in Equation 2 has a fixed value, so it can be computed at compile time;
2. given M, we can compute the variable A defined in Equation 3 and so set the bits range which must correspond to the output of the FPGA filter (Line 48 of Source Code 1);
3. given A, it is possible to compute the relative error due to the truncation performed inside the FPGA (as in Equation 9).

5.1 Overall gain of the FPGA filter

The knowledge of M (and so of A) allows us to compute the overall gain of the filter inside the FPGA. Since, 2^A digits are needed to represent M, but none of the values in the range $\left(M, 2^A\right]$ could ever be computed, then the gain of the filter implemented in the FPGA is the fraction of computable values over all the representable values, so

$$G_{\text{FPGA}} = \frac{M}{2^A}. \tag{8}$$

6 Errors

During the processing of our signal, we lose precision in two stages

1. in the truncation of the `signed` variables inside the FPGA
2. in the conversion from 64-bits to 8-bits variables

and each of these losses introduces an error on the processed signal.

First of all, we analyse the error due to the truncation. The relative error can be computed as the ratio between the maximum value lost in the truncation and the maximum representable value. To compute the absolute error, the relative one must be multiplied by the amplitude of the input signal. This leads to the equation

$$\varepsilon_{\text{trunc}} = \frac{2^{A-7} - 1}{2^A - 1} \frac{\max_i(x_i) - \min_i(x_i)}{2} \tag{9}$$

where A is the index of the most significant bit (defined in Equation 3, in our case $A = 16$).

For what concerns the conversion error, it is introduced because of the rounding in Equation 6. The relative error can be easily estimated as one half (rounding to closest integer) divided by the extension of the spanned range $\left[-2^7 + 1, 2^7 - 1\right]$; multiplying it by the extension of the range of the input signal, we obtain the absolute error:

$$\varepsilon_{\text{conv}} = \frac{1}{2} \frac{\max_i(x_i) - \min_i(x_i)}{2^8 - 2} \tag{10}$$

In both cases, the errors computed with these equations are overestimation of the actual errors that affect each element of the signal. In fact, both the truncation and the conversion errors can be, for each sample, at most equal to the computed values. Despite these errors are quite small in magnitude, we must take them into account when making considerations about the signal filtered by the FPGA.

7 Generation of the signal

We need to produce a noisy signal to feed our FIR filter in order to test its performances.

The clean signal we produce is a monochromatic sinusoid of amplitude 0.7 and frequency 10 Hz, with null initial phase and sampling rate 200 Hz. We have chosen a monochromatic sinusoid for the input signal so that it is easy to check how good the filtering process is.

For the noise contribute, we decided to add a high frequency (60 Hz) monochromatic noise, with amplitude 1 (slightly bigger) to the clean signal wave. The choice for a monochromatic noise has two main reasons:

1. Even if the noise has the same amplitude of the signal, the two contributes are still well distinguishable. On the contrary, with a white noise with such an amplitude, the clean signal would have been totally hidden.

2. Having a specific noise frequency allows to set up a FIR filter which filters out the noise only. In our particular case, the noise has a higher frequency with respect to the signal, so a low-pass filter should make the job. A white noise would be much more difficult to be filtered out, because it would take all the frequencies and we could not remove it with a simple high/low-pass filter.

The input signal is then produced by summing the noise and the clean signals. In Figure 15, there is the graphical representation of the signal.

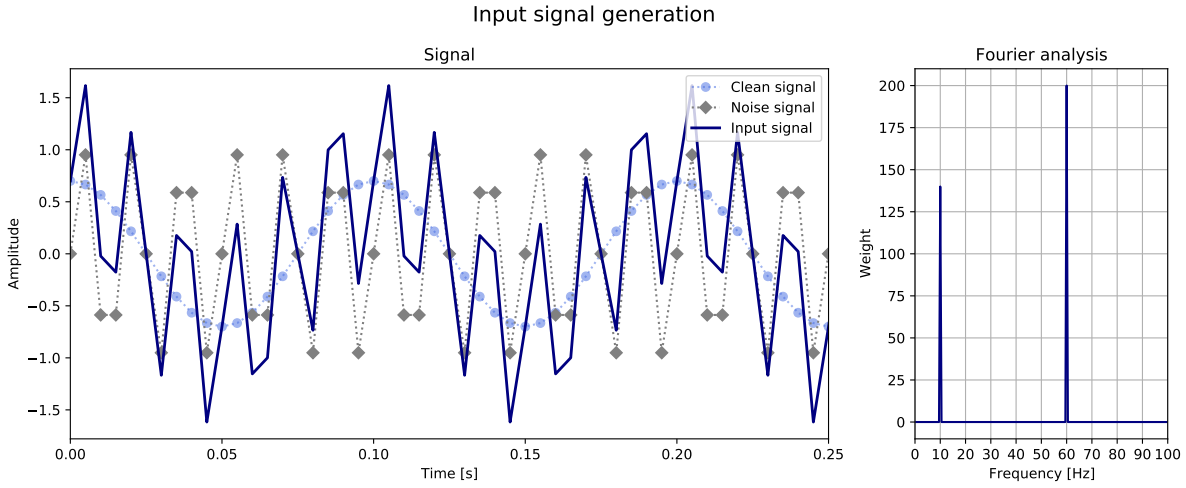


Figure 15: Input signal generation. On the left plot we can observe the input signal and its components. On the right plot we performed a Fourier analysis of the input signal.

The Fourier analysis on the input signal is essential to detect the frequencies of the real signal and the noise. In this specific case, since we built the signal, we already knew the component frequencies, but in general this could be not known in advance. Thus, we perform the Fourier decomposition and we verify that we have to component frequencies at 60 Hz and 10 Hz.

8 Set up of the FIR filter

We need to set up the structure of our FIR filter in order to perform a good filtering on this specific signal. Given that the noise has a high frequency, we choose to use a low-pass filter, with a simple Hamming window, as mentioned before.

When first approaching the problem, it seems that the most suitable setup for a filter working on our signal is a cut-off frequency $10\,\text{Hz} \leq f_{\text{cutoff}} \leq 60\,\text{Hz}$. Our main aim is having a filter with a very low gain on the noise frequency, so we opt for a cut-off frequency which is closer to the signal one, and we adjust the number of taps consequently. Indeed, recalling the results obtained in Figure 3 with a cut-off frequency $f_{\text{cutoff}} = 15\,\text{Hz}$, we observe that the gain curve with 9 taps is very suitable for our purpose. In Figure 16 we highlight our choice for the FIR filter setup.

Having a cut-off frequency so close to the signal one implies that the filter cuts also a part of the amplitude of the signal, so we must take this into account and renormalize the processed signal by the inverse of the gain on its frequency. In our specific case, the gain of the filter on the signal frequency is

$$G_f \sim 0.89 \tag{11}$$

.

To improve the gain on signal, we could have set up a filter with a higher number of taps, but this would have led to some other drawback which cannot be resolved by a mere renormalization factor:

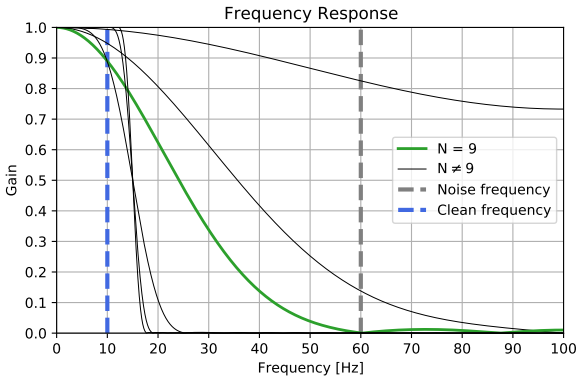


Figure 16: The solid curve represents the configuration we have chosen for our implementation. The dotted lines represent gain curves with same cut-off frequency and different number of taps. To be compared with Figure 3.

- The implementation in VHDL becomes longer.
- The processed signal is affected by a bigger delay.
- The sum and multiplication operations in the FIR filter need more approximations, which is particularly bad in the case the precision of the computation is already low.

8.1 Delay

As already stated in subsection 1.1, every FIR filter has a delay proportional to the length of the delay line (equal to the number of taps N). For this reason, the output signals processed by both FIR filters (Python and FPGA) are affected by the same delay, which must be subtracted in order to compare the results. Quantitatively, this delay time is computed with the formula

$$\text{delay} = \frac{1}{2}(N-1)\frac{1}{f_s} \tag{12}$$

where f_s is the sampling rate of the signal.

9 Filtering in Python

`scipy` provides `lfilter` function which performs the action of a FIR filter, so we rely on its output in order to check whether our implementation works properly. The filtering function `lfilter` is fed with the tap coefficients and the input signal, and returns the processed signal. Since the taps provided are normalized such that $\sum_k h_k = 1$, the gain on `lfilter` output will be $G_P = 1$.

After the signal is processed, it must be renormalized by the gain on the clean frequency (Equation 11) and shifted left by the time delay (Equation 12)

$$y_{\text{py}}(t) = \frac{1}{G_f}\, P(t - \text{delay}) \tag{13}$$

where $P(t)$ is the output of `lfilter` and $y_{\text{py}}(t)$ is the plotted curve with the corrections.

The result obtained after these corrections is shown in Figure 17.

The processed signal is very close to the the clean original signal, as expected from the considerations about the gain on the noise frequency in Section 8.

Once we know what to expect from the FIR filter, we can feed our signal into the programmed FPGA.

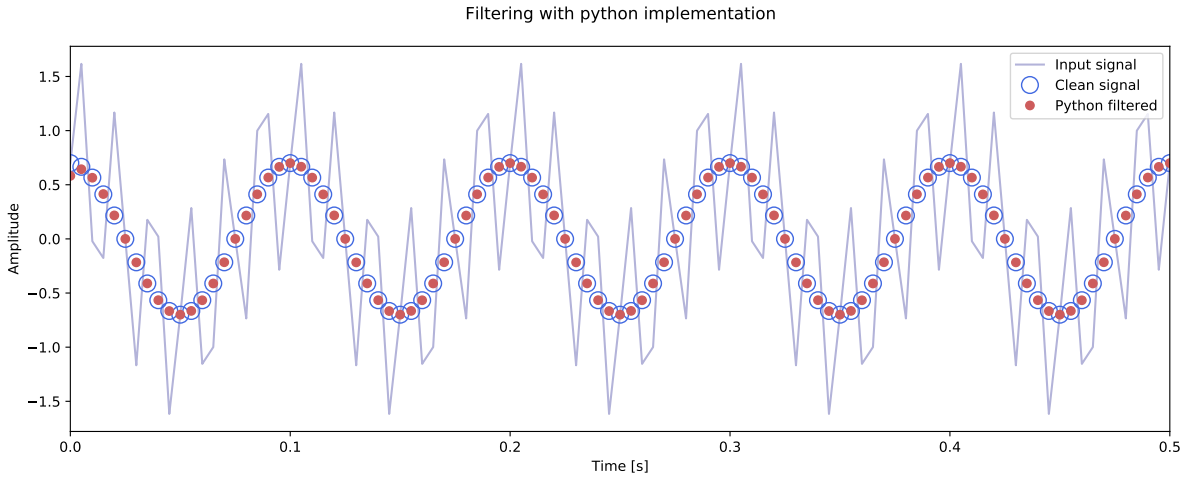


Figure 17: Input signal filtered with Scipy filter.

10 Comparison FPGA and python

The signal generated with Python, after the conversion in integers, is sent to the FPGA through the *Serial* library, again used when collecting the filtered signal. Then, we processed it in Python in order to convert again the data from integers to double (as explained in Section 4).

To compare the output from the FPGA with the signal filtered by Python, we need to shift the signal by the delay in Equation 12 and to renormalized it by both the gain on the frequency and the gain of the FPGA:

$$y_{\text{FPGA}}(t) = \frac{1}{G_f \, G_{\text{FPGA}}} \, F(t - \text{delay}) \tag{14}$$

where $F(t)$ is the output of the FPGA filter.

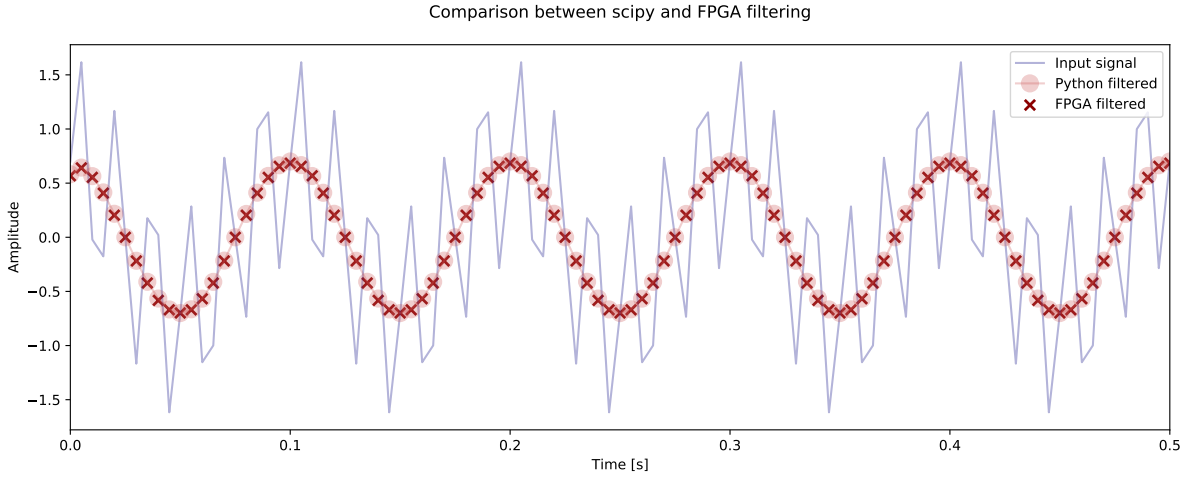


Figure 18: Comparison between signal filtered by Python and by FPGA .

In Figure 18 we present the comparison between the signal filtered by Python and the one filtered through the FPGA, where they seem to be very similar. In order to highlight the differences between the two, we plot the discrepancies $D(t) = P(t) - F(t)$ in Figure 19.

In the figure are also shown the two components of the errors ($\varepsilon_{\text{conv}}$ and $\varepsilon_{\text{trunc}}$), previously estimated in Section 6. As expected, the total error is asymmetric, because the truncation component causes value reduction only. On the contrary, the error due to conversion is symmetric. On average, the discrepancies assume a positive value, again because the signal filtered with the FPGA is affected by truncation.

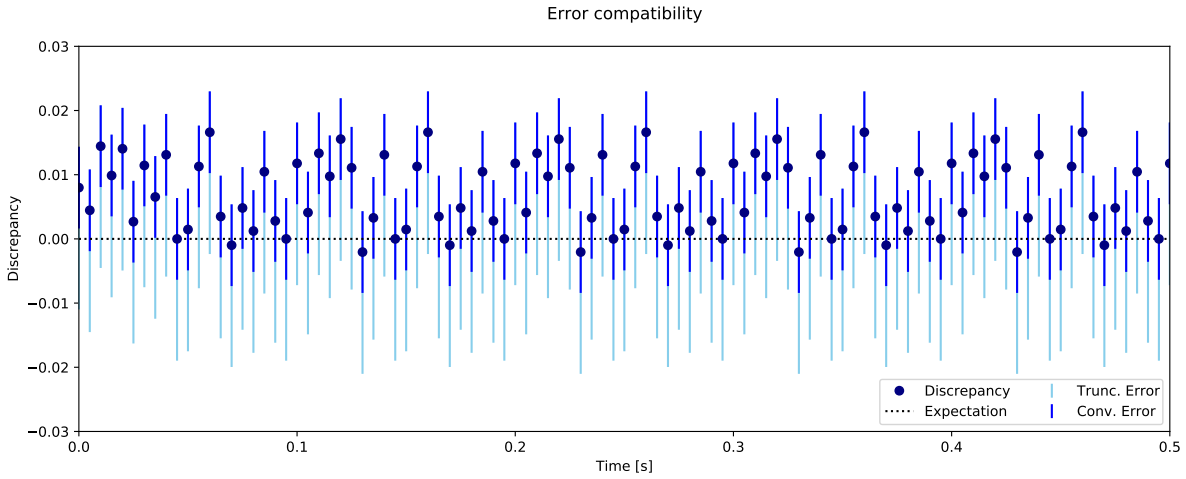


Figure 19: Discrepancies and errors due to conversion (drawn in blue) and due to truncation (drawn in cyan). The full error bar is obtained summing, for each point, the two errors.

The distance between the maximum value of discrepancy and zero stays within our error, therefore the estimation done in Section 6 is coherent.

We can also notice that the discrepancies are two order of magnitude smaller than the values of the signal. Thus the two filtered signals are in a good agreement one each other.

11 Conclusions

We synthesised a 9 taps low pass FIR filter inside an FPGA, designed such that it should perform an optimal denoising on the test signal. We focused on the logical decoupling between the specific signal characteristics (amplitude) and the set up of the filter, in order to have a generic architecture which can deal with a variety of signals. The results from the FPGA well match the ones simulated in Python using `lfilter`, with discrepancies within the range estimated a priori by a theoretical analysis on the processing routine. We then conclude that the FIR filter has been properly implemented and it has a reasonable accuracy considering the limitations imposed by the 8-bits variables.

References

[1] Walt Kester. *Mixed-Signal Design Seminar*. Analog Devices, 1991. ISBN: 0-916550-08-7.

[2] Bryan Mealy; Fabrizio Tappero. *Free Range VHDL*. 2018.

[3] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.