Management and Analysis of Physical Datasets

VHDL implementation of a moving average filter

E. Fella, C. Maccani, P. Miglioranza, C. Sgorlon Gaiatto

February 4, 2022

Abstract

A moving average filter is a common causal finite impulse response filter used in digital signal processing. It can be built through a simple and very fast algorithm, which makes it suitable for many different applications. The main objective of this work is the design of the architecture of a moving average filter using VHDL and its implementation on an FPGA. In particular, we have applied the filter in a cascade to approach the effect of a low-pass filter and analyzed the latency introduced by the cascade itself. Then we have tested it on a multi-frequency sinusoidal signal with noise background and we have compared the results with a python simulation. Since the FPGA uses a UART for serial communications that handles only 8 bit signed integers, we had to properly pre-process the input signal. The error introduced by this process along with the internal division approximation of the filter have been discussed. In the end, we have also briefly analyzed the latency and the resource utilization of the presented implementation.

1 Introduction

In digital signal processing, a finite impulse response (FIR) filter is a filter whose impulse response is of finite duration. Since we are interested in systems that process discrete time-domain signals in real time, we have considered only causal FIR filters for which it holds that the output depends only on the past and present inputs. This type of filter can be described mathematically as the convolution between a sequence of input samples $x[t]$ and the impulse response of the filter $h[t]$, usually referred to as the filter coefficients:

$$y[t] = \sum_{k=0}^{n-1} h[k] \cdot x[t-k] \tag{1}$$

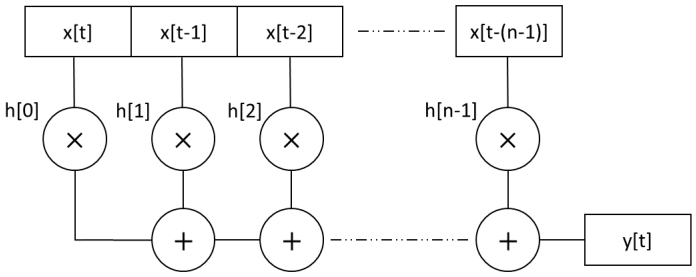

Figure 1: Block diagram of a causal FIR filter

One of the most common causal FIR filters is the moving average filter. It operates by averaging n points of the input signal, also known as taps, to produce each point of the output signal. In other terms, all the coefficients of the filter are equal to $1/n$:

$$y[t] = \frac{1}{n} \sum_{k=0}^{n-1} x[t-k] \tag{2}$$

The algorithm can be further simplified by observing that the difference between two adjacent outputs of the filter is equal to the difference between the incoming and outgoing inputs:

$$y[t] = \frac{1}{n}(x[t] + y[t-1] - x[t-n]) \tag{3}$$

As we can see, the moving average filter is very simple to implement and it is much faster than other digital filters. This is due to the fact that it requires only two operations regardless of the number of taps. Furthermore, these operations are an addition and a subtraction which take less time than the multiplications required by causal FIR filters with more complex coefficients. It is precisely this simplicity that makes the moving average filter attractive for many applications, such as random noise reduction. In particular, we are interested in showing that it can be used to approximate a low-pass filter. In order to do that we need to describe the effect of the filter in the frequency domain and this can be done by exploiting the convolution theorem. In fact, it states that the Fourier transform of the convolution of two functions in the time domain corresponds to the product of their transforms in the frequency domain. Since we are operating on discrete time-domain signals, the frequency response of the moving average filter is mathematically described by the discrete-time Fourier transform of the filter coefficients $H[\tilde{f}]$, which can be seen as the transform of a discrete rectangular pulse:

$$\left|H[\tilde{f}]\right| = \frac{\sin(n\pi\tilde{f}))}{n\sin(\pi\tilde{f})} \tag{4}$$

where $\tilde{f}$ represents a normalized frequency obtained dividing the frequency by the sampling rate f_s of the signal [1]. In Figure 2, we have plotted the frequency response of the moving average filter with different values for the number of taps.

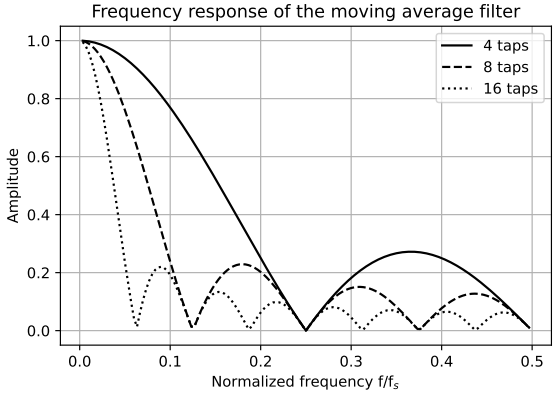

Figure 2: Frequency response of the moving average filter with n taps. The normalized frequency runs between 0 and the Nyquist frequency whose value is half the sampling rate.

We can see that the moving average filter has poor stop-band attenuation making it a bad low-pass filter. However, we would like to use to our advantage the fact that it is implementable with a very fast algorithm. Therefore, the idea is to apply a cascade of N filters in order to obtain a much deeper stop-band. As can be seen in Figure 3, a cascade of 4 filters with 4 taps each can approximately approach a low-pass filter although it is still far from a well designed one which would require more complex coefficients that would allow us to control its frequency selectivity.

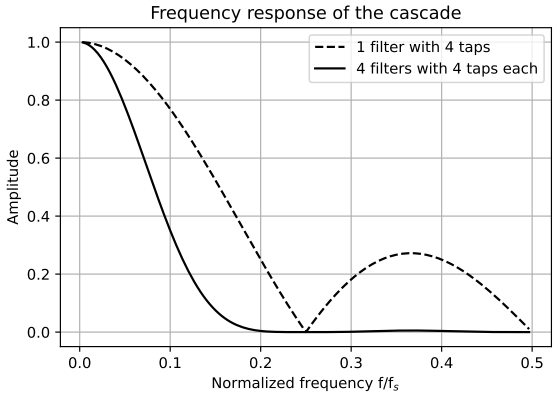

Figure 3: Frequency response of a cascade of 4 filters with 4 taps each. We can estimate the cutoff frequency of the cascade by considering the first root of the frequency response of the moving average filter, which is equal to $1/n$ in normalized units.

2 Objectives and dataset

The goal of this work is to design a working VHDL moving average filter and to implement a cascade of 4 filters with 4 taps each on a FPGA. As an application of our implementation we attempt to filter an artificial signal produced by us via python script. The input signal $x[t]$ is composed by a main sinusoidal signal x_0, three components to be filtered x_1, x_2, x_3 and a Gaussian noise background x_{noise}. In particular:

$$\begin{aligned} x_0[t] &= 10 \cdot sin(2\pi f_0 \cdot t) \\ x_1[t] &= 3 \cdot sin(2\pi f_1 \cdot t) \\ x_2[t] &= 2 \cdot sin(2\pi f_2 \cdot t) \\ x_3[t] &= 1 \cdot sin(2\pi f_3 \cdot t) \\ x_{noise}[t] &\sim \mathcal{N}(0,1) \end{aligned} \tag{5}$$

$$x[t] = x_0[t] + x_1[t] + x_2[t] + x_3[t] + x_{noise}[t] \tag{6}$$

where t is a discretized version of the time variable obtained with a sampling rate of $f_s = 192\, kHz$, while the frequencies that constitute the input signal are respectively $f_0 = 0.01 \cdot f_s$, $f_1 = 0.25 \cdot f_s$, $f_2 = 0.35 \cdot f_s$ and $f_3 = 0.45 \cdot f_s$. The resulting signal is shown in figure 4.

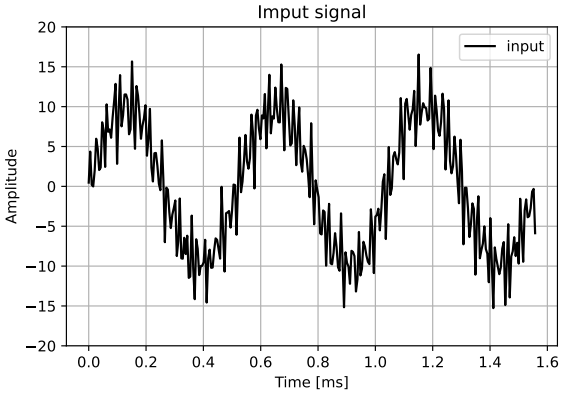

Figure 4: Discrete-time input signal consisting of a low main frequency, additional higher frequency components and a Gaussian noise background.

As can be seen in Figure 5, we have evaluated the discrete-time Fourier transform of the input signal with overlaid frequency response of the cascade. The expected cascade output $y[t]$, calculated through a python script able to deal with real numbers, is shown in Figure 6.

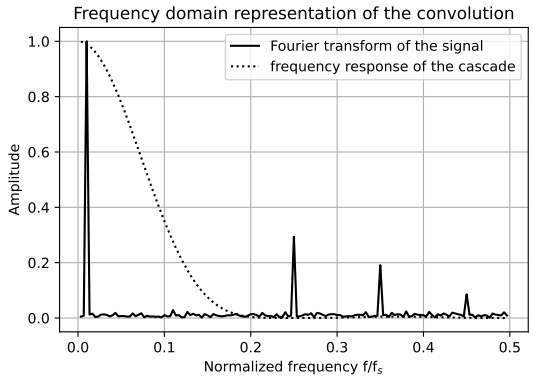

Figure 5: Both functions have been normalized to better visualize how the frequency components of the signal are shaped by the frequency response of the cascade.

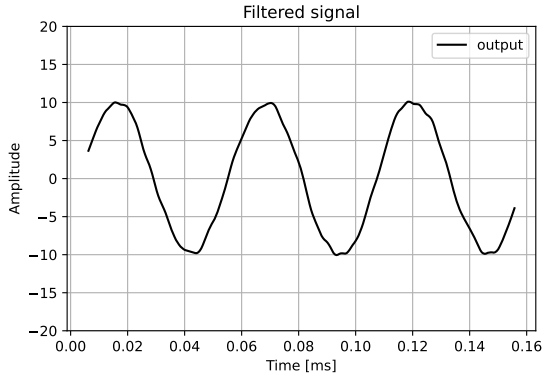

Figure 6: Filtered output signal obtained through a python simulation of the moving average filter cascade.

3 Data flow description

The schema in Figure 7 shows the entire flow of data. The sampled points of our dataset are considered as the input of the whole process. This vector is, first of all, handled by a python script, which is responsible for the pre-processing (see Section 5 for more details) and for the communication with the PC's serial port. This USB port is then physically connected through a cable to the one of the FPGA. From the FPGA point of view, the communication is managed by a UART.

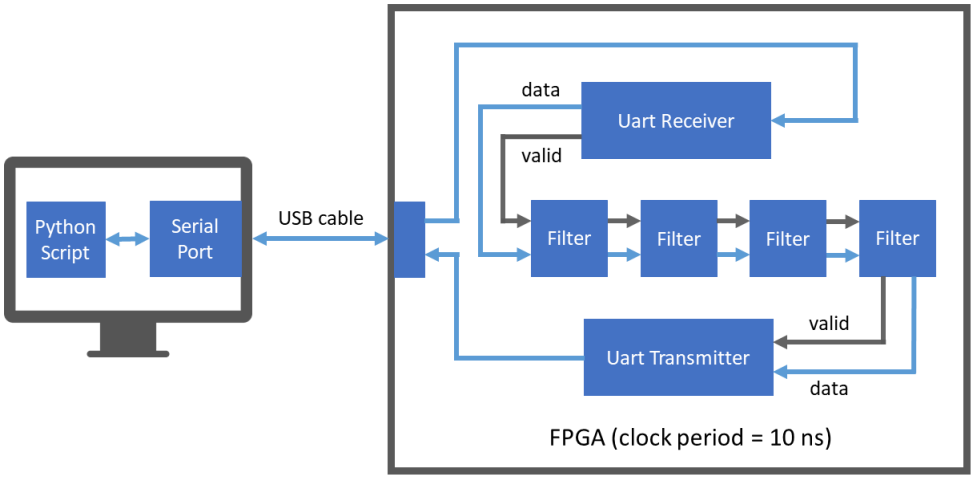

Figure 7: Data flow schema. All the code have been scripted in VHDL on Xilinx Vivado and the circuits are implemented on ARTIX-7100TCSG324 FPGA.

In Listing 1 are reported the lines of the python code used to send and read back the data. It is made used of the PySerial library. The baudrate of the communication is set to be $baudrate = 115200\ bit/s$, that means that the duration of each bit is set to be roughly $8,68\ \mu s$. This corresponds, from the FPGA's point of view, to 868 clock cycles, considering that the period is $10\ ns$. Since the UART we use is written to work with 8 bit signed numbers, before being transmitted from the PC, the data have to be converted into the `bytes` data type. Each input data is sent to the FPGA and then read back and reconverted. The next input has to wait the end of this procedure.

```python
import serial

# ------------ pre-processing of the input array x ------------

ser = serial.Serial('/dev/ttyUSB1', baudrate=115200)

for i in range(len(x)):
  ser.write(int(x[i]).to_bytes(1, "little", signed=True))
  d=ser.read()
  out[i]=int.from_bytes(d, "little", signed=True)

# ----- post-processing of the output array out and plots -----
```

Listing 1: Python script used to communicate

When the data arrive to the FPGA, they are managed by the receiver. It is implemented as a state machine: when the bit that encodes the beginning of byte arrives it exits from the idle state, then it waits for half baudrate time and finally starts reading, namely it fills a 8 bit size *std_logic_vector* with the value it has in input at that moment with a rate of a baudrate. The waiting procedure is performed in order to be sure that the reading is not done at the moment in which the value of the receiving bit is changing. At the end, when it reads the bit that encodes the end of the transmitted data, it generates a `valid` signal and sends the vector to the first filter. Each filter processes the data and, only when it finishes, it generates another `valid` signal and transmits the results to the next one.
At the end of the cascade the data are sent back to the PC by the transmitter. The transmitter is composed by a baudrate generator which constantly emits pulses at the distance of a baudrate time and a state machine that can change state only in correspondence of one of these pulses. When a `valid` signal arrives from the last filter, it waits until the next pulse and then starts to transmit the content of the result's vector with the rate of a baudrate. It sends back also a bit encoding the start and the stop of the transmitting procedure.

4 Moving average VHDL implementation

In this section we present the VHDL implementation for the single moving average filter [2, 3].

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity fir_filter is
  port(
    clk       : in  std_logic;      -- clock
    rst       : in  std_logic;      -- reset
    valid_in  : in  std_logic;      -- valid signal from UART receiver
    data_in   : in  std_logic_vector(7 downto 0);   -- input datum
    valid_out : out std_logic;      -- valid signal to UART transmitter
    data_out  : out std_logic_vector(7 downto 0)    -- filtered output
    );
end fir_filter;
```

Listing 2: Entity of the implemented code for the moving average FIR filter

- First of all there is the entity construct that provides how the filter interfaces with the UART system. The input ports are `clk`, `rst`, `valid_in` and `data_in`, they are all *std_logic* data types except the last one that is an 8-bit *std_logic_vector*. More precisely: `clk` is the same clock signal of the other components of the whole system, `rst` is a signal that controls if the filter is active (`rst`= 1) or not (`rst`= 0) and `valid_in` is the output of the receiver that communicates to the filter module that in the input port `data_in` a new datum is ready to be considered. The output ports instead are `valid_out`, a *std_logic* data type that sends to the transmitter the necessary control signal to start its task, and `data_out`, an 8-bit *std_logic_vector* that is the filtered output.

```
architecture behavioural of fir_filter is
-- declarations
    constant n_taps  : integer := 4;

    type type_byte_array is array (0 to n_taps-1) of signed(7  downto 0);
    signal data_sequence : type_byte_array;

    signal data_sum      : signed(9 downto 0);
    signal data_norm     : signed(9 downto 0);

    type state_t is (idle, sum, norm, output);
    signal state : state_t := idle;

begin

  state_machine : process(clk) is
  begin
        if rst = '0' then
            data_sequence <= (others => (others => '0'));
            data_sum <= (others => '0');
            data_norm <= (others => '0');
            valid_out <= '0';
            if valid_in = '1' then      -- bypass filter
                data_out <= data_in;
                valid_out <= '1';
            end if;
        elsif rising_edge(clk) then     -- filter activated
            case state is
                when idle =>
                  valid_out <= '0';
                    if valid_in = '1' then
                        state <= sum;
                    else
                        null;
                    end if;
                when sum =>
                    data_sum <= resize(signed(data_in), 10)
                              - resize(data_sequence(n_taps-1), 10)
                              + resize(data_sum, 10);
                    data_sequence <= signed(data_in) &
                                     data_sequence(0 to n_taps-2);
                    state <= norm;
                when norm =>
                    data_norm <= data_sum/to_signed(n_taps, 10);
                    state <= output;
                when output =>
                    data_out <= std_logic_vector(data_norm(7 downto 0));
                    valid_out <= '1';
                    state <= idle;
            end case;
        end if;
  end process state_machine;

end behavioural;
```

Listing 3: Architecture of the implemented code for the moving average filter

- What the circuit actually does is described in the behavioral architecture where, before the beginning, it is necessary to declare some signals and constant used in the process. The constant `n_taps` defines the number of taps considered for the filter; it determines some important indices in the development of the code as the dimensions of the `data_sequence` signal. To declare this signal in fact we need to define a new data type *type_byte_array* that represents a multidimensional array object that collects `n_taps` *std_logic_vector* of 8 bits in which we will sequentially store the sampled `n_taps` input data required to evaluate the output data of the filter. There are also the declaration of two other signals as *signed* data types of 10 bits which will be necessary inside the process. Finally, since we will model a case statement, we need also to define the *state_t* data type and declare the signal `state` initially assigned to `idle`.

- After the declaration, the architecture begins and inside it there is only the process whose sequential statements are re-evaluated every time the `clk` signal (the only one in the sensitivity list) changes its value. Inside the process there is an if statement whose first condition depends on the input port `rst`. When `rst` = 0 all the internal signals are assigned to zero except when the `valid_in` signal indicates that there is a new ready input datum from the receiver which will be directly assigned to the output signal `data_out` without being filtered; in turn this assignment will be communicated to the transmitter through the output signal `valid_out` = 1. When `rst` is not equal to 0 instead the filter operates and the case statement is evaluated every rising edge of the `clk` signal.

- Initially the signal `state` is assigned to `idle`; in this case the filter module just wait for a new input datum and so the `valid_out` signal is assigned to 0. When `valid_in` becomes 1 the input datum from the receiver is ready and the signal `state` is assigned to `sum` so the case statement can proceed.
  This case is exploited for two tasks: the discrete convolution of `n_taps` between the chosen coefficient (eq.1) and the current `data_sequence` and the update of the `data_sequence` itself with the arrived new input datum. Considering a moving average, it was shown that the convolution in eq.1 can be simplified to eq.3 and the same can be done also in the implemented filter where this computation can be reduced to just two simultaneous operations of subtraction and addition regardless of the value of `n_taps`. In this way we avoided not only the multiplication between each `data_sequence` elements and the coefficients (that are constant and equal to 1) but also the `n_taps` additions of the results of these multiplications. The sum of the new input datum `data_in` ($x[t]$ in eq.3) and the previous assigned `data_sum` ($y[t-1]$ in eq.3) minus the last stored datum in `data_sequence` ($x[t-n]$ in eq.3) is assigned to the signal `data_sum`. One thing to note is that for every addition (or subtraction) to represent the result of the operation it may be needed one bit more than the number of bits of the addends, so in the assignment of `data_sum` it is necessary to resize all the involved signals. When the signal `state` is assigned to `sum`, as said before, the signal `data_sequence` is updated, this is made through the concatenation operation between the new input datum `data_in` and the first three stored *std_logic_vector* of the previous assignment of `data_sequence`. Another important thing to note is that what makes possible to do both these assignments not only in just one *when* branch but also without recurring to other internal signals is that the assignments of the signals inside a process are only scheduled when the same process is completed.
  Once these assignments are done the signal `state` is assigned to `norm`, in this case it is introduced the division by the `n_taps` of eq.3 that was not considered in the previous state. This is done dividing the signal `data_sum` by the `n_taps` constant converted to *signed* data type and resized to 10 bits.
  After that, the signal `state` is assigned to `output`: in this case the `data_norm` signal converted to *std_logic_vector* data type and resized to 8 bits is assigned to the output port `data_out`. In the operation of resizing the last two bits (the most significant ones) are lost but this is acceptable considering that, if the input data are under the limit of representation for a *signed* data type of 8 bits (as in our case study), we expect that also the output data (that is just a moving average of the input data) will be under this limit. In this same *when* branch it is necessary to assign to `valid_out` the value 1 to report to the transmitter that `data_out` is correctly assigned. After that the `state` is again assigned to `idle`. One last thing to note is that this implementation produces a transient period in which the filter has not already received at least `n_taps` sequential input data and so it can not correctly assign the output data that so must be neglected in that phase.

- If we want to pass from just one filter to a cascade of them we can exploit the possibility to reuse the already shown module. This is possible thanks to the structural modeling of the top-level implementation that takes into account all the modules of the whole system. Particular attention must be paid in the declaration of the required internal signals used to connect the design units and in the mapping process of the various modules. The idea is that a filter receives the input signals `valid_in` and `data_in` from the correspondent output ports `valid_out` and `data_out` of the previous filter, except in the case it is the first filter (that instead receives them from the receiver) or the last filter (that communicates its output signals `data_out` and `valid_out` to the transmitter). The `clk` signal is of course the same for all the filters. The `rst` signals instead are linked to some FPGA pins through the constraint file: in particular for all the four filters considered the input `rst` is connected to four different switches in order to decide how many of them we want to activate.

5 Preprocessing

Since the receiver and transmitter can only read and send 8-bits signed data, before sending the signal to the FPGA we need to preprocces it. Each input signal sample is converted in the following way:

$$x_{FPGA}[t] = \lfloor x[t] \cdot s \rfloor \tag{7}$$

Then, before sending the input signal to the receiver, we perform two operations:

- **multiplication for a scale factor** s: input samples are multiplied for

$$s = \frac{\min\left(127, \frac{2^9 - 1}{n}\right)}{\max\left(|x[t]|\right)}$$

  Through this operation we resize the input signal amplitude in order to transfer as much information as possible to the samples integer part according to the sum over the taps superior limit, that is given by $2^9 - 1$, and according to the 8-bit representation superior limit.

- **Floor function application:** after the scaling operation we apply the floor function to retrieve the integer part of the rescaled input samples .

6 Error analysis

The fact that our implementation can read and send only 8-bits signed integers and can perform the sum and the division with 10-bits signed integers leads to an approximated output with respect to the case of a filter that is able to deal with real numbers. In particular the error is introduced by the input samples conversion to integers during the preprocessing phase and by the conversion to integers that occurs in the division step inside the filters. We can try to find the upper bound of the rounding error affecting the output signal through a qualitative analysis of the preprocessing phase and division phase maximum errors:

- **preprocessing error:** the rounding error due to the floor function is at most 1, then starting from Equation 7 we can see that the preprocessing of the data introduce a maximum error of $\mathcal{O}(1/s)$.

- **internal division error:** each filter does the following operation :

$$y[t] = \left\lfloor \frac{x_{FPGA}[t] + y[t-1] - x_{FPGA}[t-n]}{n} \right\rfloor$$

  then, as in the preprocessing phase, the rounding error affecting the output y[t] it is at most 1, and after the rescaling operation we get that the error introduced by a single filter is of $\mathcal{O}(1/s)$.

Therefore the maximum error after a cascade of N filters is:

$$\Delta y \sim \mathcal{O}\left(\frac{1}{s} + N \cdot \frac{1}{s}\right) = \mathcal{O}\left((N+1) \cdot \frac{1}{s}\right) = \begin{cases} \mathcal{O}(N+1)\ if\ n < 5 \\ \mathcal{O}\big((N+1) \cdot n\big)\ if\ n \geq 5 \end{cases} . \tag{8}$$

meaning that it is linear with the number of filters in the cascade and, for $n \geq 5$, it is also linear with the number of taps per filter. Moreover, as long as the number of taps per filter n is bigger or equal to 5, fixed a total number of taps, the error goes as $\mathcal{O}(1 + 1/N)$.
In order to verify if the error dependency on the number of taps for each filter (n) and on the number of filters in the cascade (N) corresponds to the one of Equation 8 we performed a python simulation of the filtering process and computed the mean absolute error Δy with respect to the case where the filters and the UART can deal with real numbers as a function of the number of taps per filter n and the number of filter in the cascade N:

$$\Delta y = \frac{1}{N_{samples}} \sum_{t=1}^{N_{samples}} \Big| y[t] - y_{FPGA}[t] \Big|$$

As it is possible to see from Figures 8 and 9, which show the simulation results, the error dependency on N and n is the same of Equation 8.

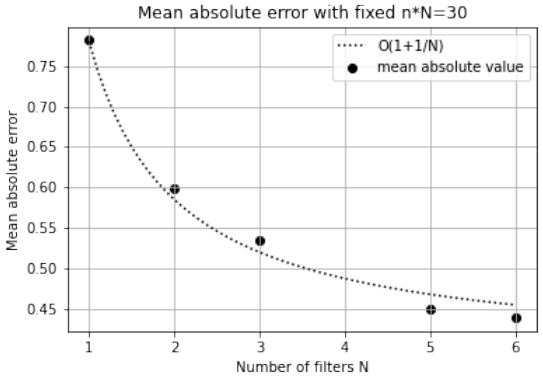

Figure 8: Mean absolute error as a function of the number of filters N keeping fixed the total number of taps. The dotted line correspond to a function $\Delta y = a \cdot (1 + \frac{1}{N})$ where $a = 0.39$

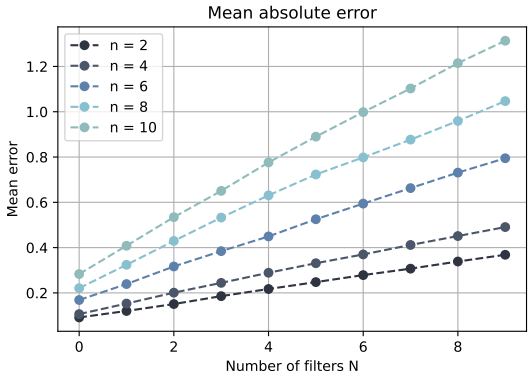

Figure 9: Mean absolute error as a function of the number of filters N with fixed number of taps per filter n. Every line corresponds to a cascade of filters with n taps each. As expected the error grows linearly with the number of filters in the cascade.

7 Latency

With latency is meant the time interval between the arrival of the first bit of a data in a certain unit and its departure. It needs to be discussed on several levels:

- Filter level: since the case statement inside the process used to implement the filter has 4 cases the delay introduced by it is of 4 clock cycles, namely 40 ns.
- Cascade level: depending on the number of active filters the delay introduced is of $N \cdot 40$ ns
- UART level: the receiver introduces a constant delay of 10 baudrate time and a half from the moment in which it sees the bit that encodes the start. On the other hand, to communicate the results back, the transmitter takes 10 baudrate time plus the time between the arrival of the `valid` signal from the last filter and the first pulse of its baudrate generator. The emission of this pulse is totally independent from what is happening before in the rest of the FPGA, so the delays introduced by activating 1, 2, 3 or 4 filters is, in every case, absorbed by the fact that the transmitter has to wait for that pulse. So,

in the limit in which not too much delay is introduced by the central blocks (roughly half of baudrate time), the latency measured at the UART level is independent from the number of active filters between receiver and transmitter.

- Python code level: in the real application the principal source of delay is introduced by the speed with which the serial ports of FPGA and PC communicate and by the python code. The lines reported in Section 3 show that the sending of a new data has to wait for the return of the previous one. This prevents the corruption of data, but slows down a lot the process, making the idle state of the receiver's state machine last for a non-optimized time. In Figure 10 is reported the histogram of the time interval between the command `ser.write()` and `ser.read()` measured for each input data in an example run on the FPGA. It can be observed that we are dealing with time intervals of the order of $\mathcal{O}(10)$ ms.

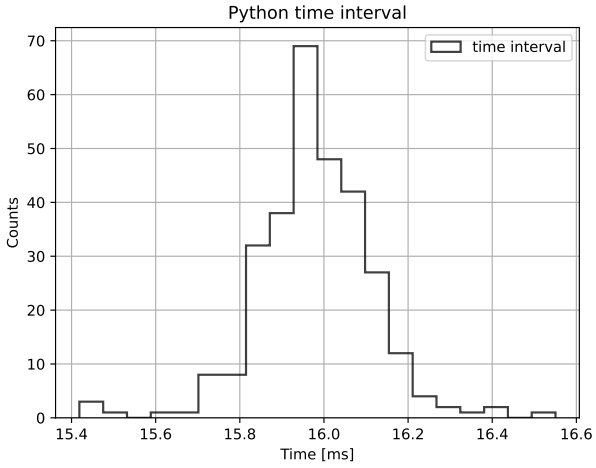

Figure 10: Histogram of the communication time of a data at the python code level

8 Resources

With resources is meant the number of elements inside the Configurable Logic Blocks (CLBs) of the FPGA that are inferred by the compiler in order to implement the VHDL code. In this section we focus only on the resources used by the moving average filter in three different optimizations. The first optimization (*opt1*) is the one presented in Section 4, in the second one (*opt2*) the division operation is substituted by the shift right logical operator (exploiting the fact that there are a number of taps equal to a power of two) while the third (*opt3*) directly implements Equation 2. In particular, in the third optimization the sum is implemented adding pairs of contiguous elements of the `data_sequence` array and since the filter has 4 taps this operation requires two steps. The resource utilization for all the three optimizations is shown in Table 1.

Optimization	Flip-Flops	LUTs as logic	LUTs as memory
opt1	68	57	8
opt2	68	43	8
opt3	100	63	0

Table 1: Resource utilization.

Comparing *opt1* and *opt2* it is possible to see that the substitution of the division operator by the shift right logical one (which is a low level operator) results in a reduction of the number of LUTs used for the logic implementation of the division function. Regarding *opt3* it can be observed that the number of LUTs increases because it requires more adding operations. Moreover, all the memory elements are implemented using Flip-Flops. Despite *opt2* is the optimization with the lowest resource utilization, it is important to note that *opt1* is the only one that allows generalization to any number of taps.

9 Results

The results obtained through the VHDL implementation of the cascade of 4 moving average filters composed by 4 taps each are identical to the ones got by running a python simulation. Regarding the filtering performances, the cascade is able to suppress the noise and cut higher frequencies: in Figure 11 it's possible to see the comparison between the main low frequency component of the input signal and the filtering operation result.

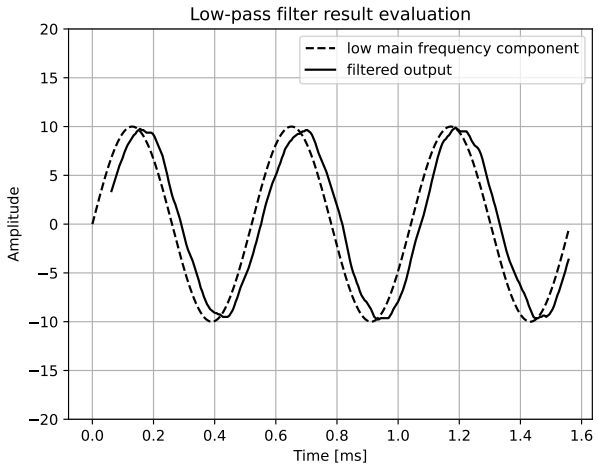

Figure 11: Comparison between the main low frequency component of the input signal and the filtering operation result.

References

[1] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997. URL: http://www.dspguide.com.

[2] B. Mealy, F. Tappero. *Free Range VHDL*. 2018. URL: http://www.freerangefactory.org.

[3] M. Zwolinski. *Digital System Design with VHDL*. Pearson Education, 2004.