Implementing a Finite Impulse Response filter on an FPGA

Lorenzo Borella Samuele Pio Lipani Marco Giunta

April 30, 2021

Abstract

A *Finite Impulse Response* low-pass filter has been implemented on a Xilinx Arty7 FPGA board, and its performance evaluated via a comparison with a `python` based simulation.

Contents

1 Introduction

1.1 Goal of the project

This project's tasks were centered around implementing a FIR filter in an FPGA board. To achieve this we needed to write, upload and compile VHDL code, write some `python` scripts to setup communication with the FPGA through a serial port, and finally analyze the results by comparing them to a `jupyter` notebook-based simulation. We had to confront multiple design choices in order to process an *ad hoc* built signal; the discussion of these is the main topic of the present report.

1.2 FIR Filter

1.2.1 The mathematics behind a FIR filter

To define what a FIR filter is we start from the following definition:[1]

> 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).

What really matters, though, is how to compute the output of the filter; in particular a causal discrete-time order N FIR filter computes each value of the output sequence as a weighted sum of the most recent input values:

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

In the above equation the following values appear:

- $x[n]$ and $y[n]$ are the n^{th} components of the input and output arrays, respectively;
- $b_0, b_1, \ldots, b_N$ are the $N+1$ coefficients which *define* the action of the filter on any signal x;
- N is the filter order, which represents the number of *taps* (i.e. b coefficients) minus 1.

Mathematically speaking the operation defined in equation (1) is nothing more than a *discrete convolution*, and as such is *linear* (as one can trivially prove) - a property we're about to make use of.

There are many other properties which we don't need to discuss; for our purposes it suffices to describe some properties of the two defining characteristics of our filter: the *taps coefficients* b_i and the *order parameter* N. In particular: one can show that *the b_i coefficients' values define which operation the filter implements, whereas the N parameter quantifies how close the output will be to that of an ideal filter.*

Say for example we want to implement an 8 taps low-pass filter with a cutoff frequency of 1 kHz to act on a signal sampled at 11025 Hz; we can then compute which values of the 8 b_i coefficients will implement the desired operation using the `scipy.signal.firwin` python function as follows.[2]

[1] Taken from this Wikipedia page.
[2] There's no need to cover exactly *how* this computation is carried out by `firwin`.

```
from scipy.signal import firwin
N = 8
fc = 1000 # cutoff frequency
fs = 11025 # signal sampling frequency
coeffs = firwin(numtaps = N, cutoff = fc, fs = fs,
    pass_zero = "lowpass") # filter type
```

These coefficients won't let us obtain an ideal low-pass - in the sense that our FIR filter won't be able to completely remove any frequency above the cutoff frequency f_c, but it can be shown that as N increases our filter becomes arbitrarily close to an ideal filter (whose log response is just a step function). Even though we can't practically reach the $N \to +\infty$ limit (of course) we'll explore this matter in the next section.

1.2.2 Filter design, implementation and `python` simulation

To gain some insight about how the order parameter N affects the performance of our filter we now plot the filter response - computed using the `scipy.signal.freqz` python function, which returns an interval of frequencies (to display on the $\vec{x}$ axis) and an interval of filter responses ($\vec{y}$ axis). By convention the frequency array computed by `freqz` is normalized to the $[0, \pi]$ interval and hence we rescale it to the $[0, f_{\text{Nyq}}]$ interval; we also need to take the base-10 logarithm of the modulus of the response array if we are to plot a complex signal in dB (as is customary to do in this kind of plots).

To actually compute the filter response one can use the following code:

```
import numpy as np
from scipy.signal import freqz

[w,h] = freqz(coeffs, worN = len(x))
w *= (fs/2)/np.pi # by default w is normalized to the [0,
    pi] interval; we map this to [0,f_Nyq]
h = np.log10(np.abs(h)) # h is complex -> abs, we want to
     show the the y axis in dB -> log10
```

During the course of this project we chose a FIR filter designed to act on signals sampled with $f_s = 11025$ Hz as a low-pass filter in order to remove some noise at 4 kHz; the next plot shows how the filter response changes if we modify the values of N and f_c, the two most relevant filter characteristics.

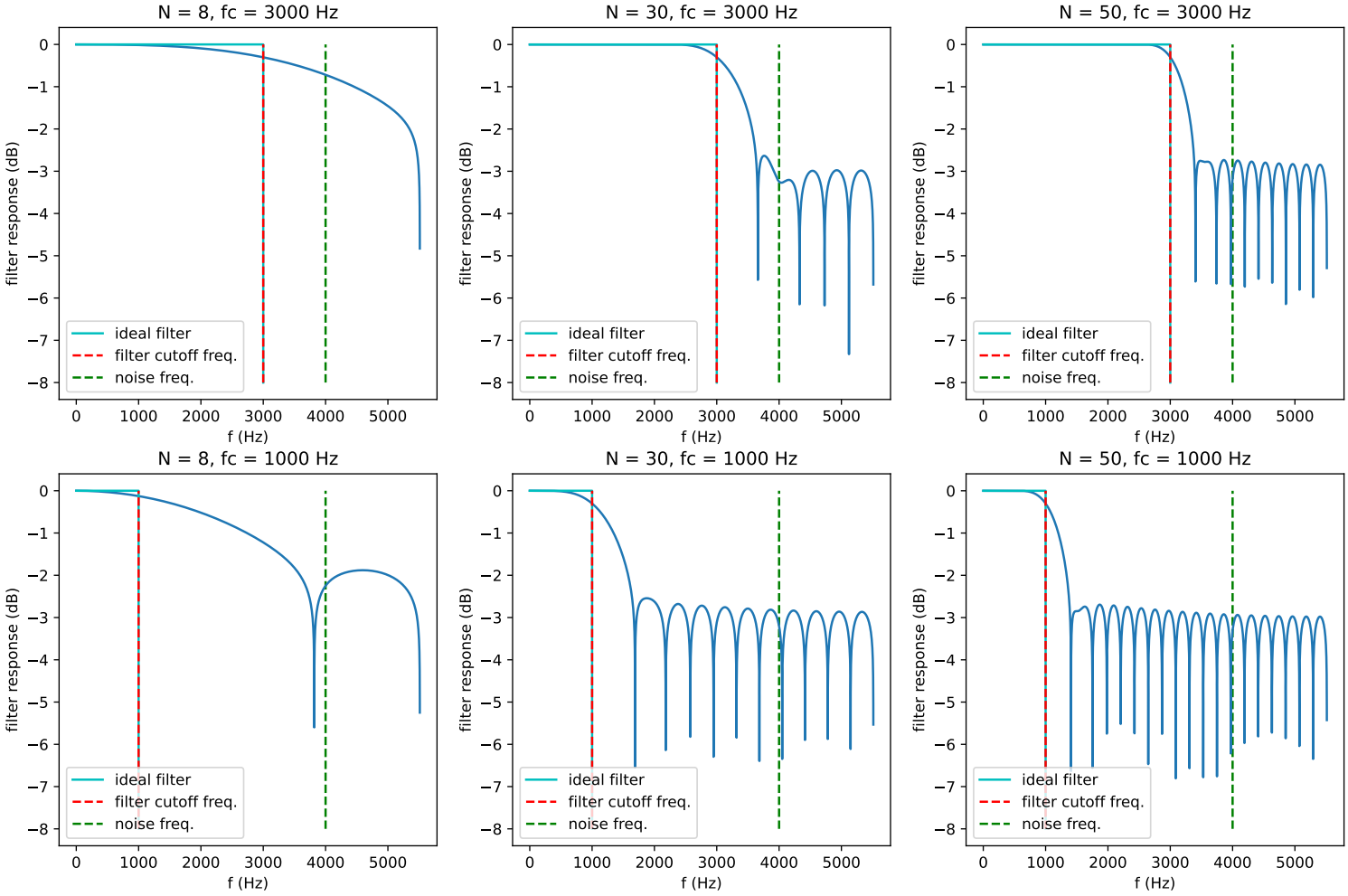

Figure 1: Ideal vs real FIR filter response for $N = 8,\ 30,\ 50$ and $f_c = 1$ kHz, 3 kHz.

Let's start from the top left. We notice that if we use a number of taps as low as $N = 8$ our filter is "lazy", and only starts significantly removing frequencies at much higher frequencies than f_c; for example with $f_c = 3$ kHz the response starts dropping only well above 5 kHz, which is a worse performing behaviour than the ideal filter (whose performance drops to $-\infty$ exactly at $f = f_c$).[3] This behaviour is clearly inadequate to

[3]The response drops to $-\infty$ because in this representation we're plotting the logarithm of $|h|$, and $h_{\text{ideal}}(f_c) = 0$.

significantly filter 4 kHz noise; to improve the situation we can take one of the two following measures.

1. We could in principle take larger and larger values of N while keeping f_c fixed; this clearly makes the response arbitrarily close to that of the ideal filter, but is an obviously impractical solution to implement.
2. Alternatively we can lower the cutoff frequency f_c, in order to move left the beginning of the "lazy response" interval; this makes for a much lighter implementation, but has the disadvantage of potentially unwanted dampening (frequencies with $f < f_{\text{noise}}$ will be reduced/eliminated, too).

For our particular implementation and data it makes more sense to keep a low value of N and set f_c equal to a much lower value than f_{noise}; why this is true will be explained in the following sections.

2 Programming the FPGA

2.1 Using signed 8 bit integers

We processed data in the form of 8 bits signed integers; in this way the most significant bit represents the sign of the number, which means we had to analyze integer data in the interval [-128,127]. This is actually an issue we need to solve since our `firwin` coefficients were all smaller than 1 in absolute value; because of this we needed to adopt the following integer representation.
A quick look at equation (1) immediately tells us that the fir filter is linear in the sense that if we scale the coefficients using a constant k_{coeff} (k_c for short) the output is scaled by k_c, too.
Proof: if $b'_i \equiv k_c b_i$ then the new output vector y' has components:

$$y'[n] = \sum_{i=0}^{N} b'_i x[n-i] = \sum_{i=0}^{N} k_c b_i x[n-i] = k_c \left(\sum_{i=0}^{N} b_i x[n-i] \right) = k_c y[n]$$

$$\implies y[n] = \frac{y'[n]}{k_c}$$

and hence if we scale the coefficients by k_c at the beginning and the output by $1/k_c$ at the end then nothing changes.
If we only use real numbers i.e. floats then this procedure rests upon no approximations, but if in between rescaling we cast to integers this means we're discarding almost all of the decimal part of the number; to see this consider the following example. Imagine $k_c = 100$ and $y = 0.12345$; if we multiply y by k_c, cast to int and then divide by k_c the result will be $\texttt{int}(12.345)/100 = 0.12$ instead of the original number 0.12345 - this is a small difference, of course, but that it is actually negligible is a matter that will be analyzed in later sections.
For now let's ask ourselves: *what if we use a larger value of k_c?* If we once again consider $y = 0.12345$ but switch to $k_c = 1000$ then the result of the double rescaling will be 0.123, which clearly is closer to the original input; this means that in order to obtain as accurate a result as possible we want to use the largest k_c that's still small enough to make our coefficients fit inside the 7 effective bits at our disposal.[4]
Notice that for reasons explained below one may want to choose the k_c

[4]The eighth bit must be sacrificed if we are to use *signed* integers.

that satisfies the above constraint and is a power of 2, too; since this may lead to a loss of precision without making the overall data analysis procedure significantly simpler we won't adopt this constraint, and instead pick a k_c which is only *approximately* equal to 2^9 (which is approximately equivalent to shifting 9 bits to the left).
Once we obtained our rescaled coefficients $\texttt{int}(k_c b_i)$ we inserted these values in the VHDL program with the conversion function `to_signed` and eventually we re-shifted the final result (times a correcting factor), in order to derive the proper integer value and compare it with the Python output behaviour; exactly how and why this takes place is explained below.

2.2 Pipeline design and VHDL implementation

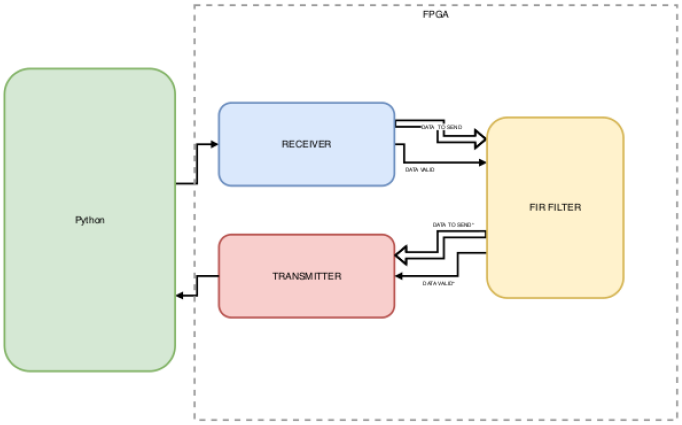

Figure 2: Block Diagram

The design we built in VHDL was made of three different components:

- **Uart Receiver**
- **Fir Filter**
- **Uart Transmitter**

All these entities shared the same 100 MHz clock as an input signal. Their behaviour was previously checked via GTKWave simulation and they were eventually connected by a top entity in order to properly program the FPGA.

```
entity uart_receiver is

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

end entity uart_receiver;
```

Figure 3: Uart Receiver Entity

The Uart Receiver takes in input the sequential series of bits we send from Python (`uart_rx` of type `std_logic`), it samples them and eventually

returns them in output (`data` of type `std_logic_vector`). The proper sampling time is given by the Sampler Generator in the form of a delayed Baudrate signal, in order to read the bits after half of their total duration. Everytime the Uart Receiver reads a series of 8 bits, it sends them in a parallelized fashion to the Fir Filter, together with a pulse signal (`valid` of type `std_logic`) as a validation command.

```
entity fir_filter is
port (
        clock  : in  std_logic;
        valid_in : in  std_logic;
        valid_out : out std_logic;

        coeff0 : in  std_logic_vector(7 downto 0);
        coeff1 : in  std_logic_vector(7 downto 0);
        coeff2 : in  std_logic_vector(7 downto 0);
        coeff3 : in  std_logic_vector(7 downto 0);
        coeff4 : in  std_logic_vector(7 downto 0);
        coeff5 : in  std_logic_vector(7 downto 0);
        coeff6 : in  std_logic_vector(7 downto 0);
        coeff7 : in  std_logic_vector(7 downto 0);

        input  : in  std_logic_vector(7 downto 0);
        output : out std_logic_vector(7 downto 0));
end fir_filter;
```

Figure 4: Fir Filter Entity

Once the Fir Filter has received the validation command (`valid_in`) from the Uart Receiver, it starts to read and process the incoming bytes (`input` of type `std_logic_vector`). The result of the filtering procedure is a signal of 19 bits: in order retain only 8 bits we kept the most significant one (the sign of each number) and the ones from position 15 to 9, performing in this way an 8 bits shift to the right. Notice that it's obvious that we need to keep only those 8 bits (we need to comply with the uart protocol), but it's not so trivial why choosing exactly the `15 downto 9` bits (i.e. discarding the 9 `8 downto 0` bits) is the correct thing to do.
For now let's pretend we used a value of k_c exactly equal to 2^9 to convert our FIR coefficients to `int8`; this means that in order to recover the correct result we need to divide by $k_c = 2^9$ - *which is exactly equivalent to shifting our binary number 9 times to the right!* Said in other words: the output of the filter is an unnecessarily high number, which needs 19 bits because we artificially increased the FIR coefficients by a 2^9 factor; this means that the "extra" `8 downto 0` bits don't contain meaningful information, since they're just the result of this "unphysical" rescaling.[5]
Once again we remark that the value of k_c we used was actually slightly smaller than 2^9 (because our coefficients wouldn't have fitted inside 8 bits with $k_c = 2^9$), but still larger than 2^8 to minimize the approximation error; it turns out that we can utilize an arbitrary value of k_c as long as we multiply the result by a correcting factor, which will be derived and used in the last section of this report.

After having filtered the data a validation pulse (`validation_out`) is sent to the Uart Transmitter, together with the newly filtered data vector (`output` of type `std_logic_vector`).

Eventually the Uart Transmitter transforms the filtered data (`data` of type `std_logic_vector`) from a parallel to a sequential fashion and returns them to the Python interface (`uart_tx` of type `std_logic`).

[5]By this we mean that this rescaling doesn't really add anything new about the information content of our signal even though it makes it bigger (hence the extra needed bits)

```
entity uart_transmitter is

  port (
    clock        : in  std_logic;
    data         : 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;
```

Figure 5: Transmitter Entity

3 Results

3.1 Data preprocessing

3.1.1 Signal corruption

Now that everything is in place we are ready to obtain some data, pass it to the FPGA and analyze the result.
Inspired by the 8 bit length of the values exchanged through the uart protocol we extracted the waveform of the first 2 seconds of the main *Super Mario Bros* 1985 theme using a MATLAB script; since the song at our disposal was sampled at 44100 Hz the obtained array was too long to be manageable, which is why we downsampled to $f_s = 11025$ Hz (a quarter of the original signal's frequency). The result of this operation is the waveform shown below.

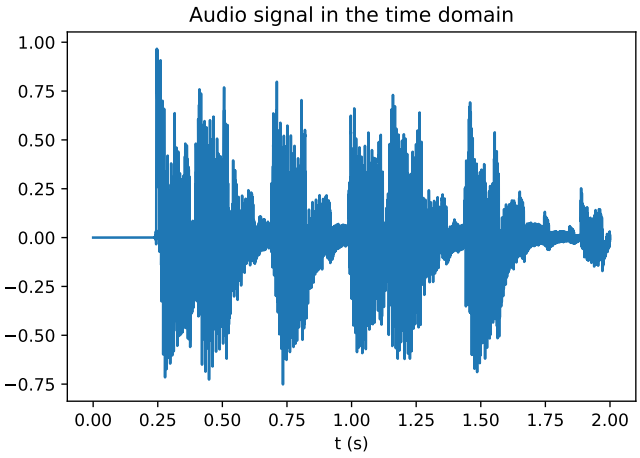

Figure 6: The first 7 famous notes of the *Super Mario Bros* theme.

We originally meant to corrupt this signal, filter it and then listen to the corrupted song before and after clean-up; due to issues in establishing a reliable serial communication with the FPGA we settled for the first 500 nonzero values of the above signal (which unfortunately corresponds to too small a portion of the song to be able to understand anything). The result of this cut is shown below.

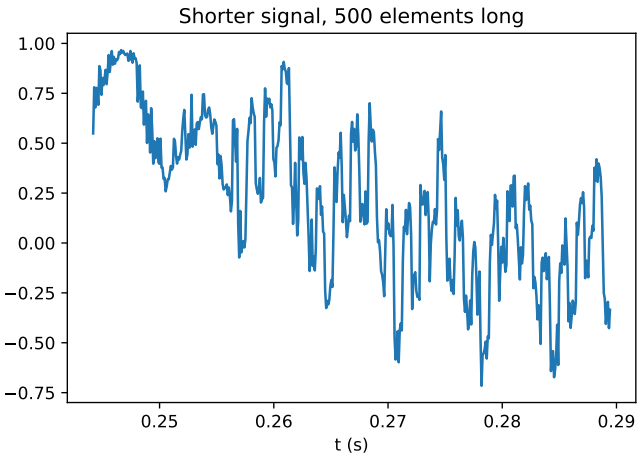

Figure 7: The fraction of the original data we actually used as the input of our algorithm.

It's easy to show this signal is the superposition of relatively few, relatively low-frequency components; in order to achieve this we now plot the frequency spectrum of this signal using an `fft` based custom function.

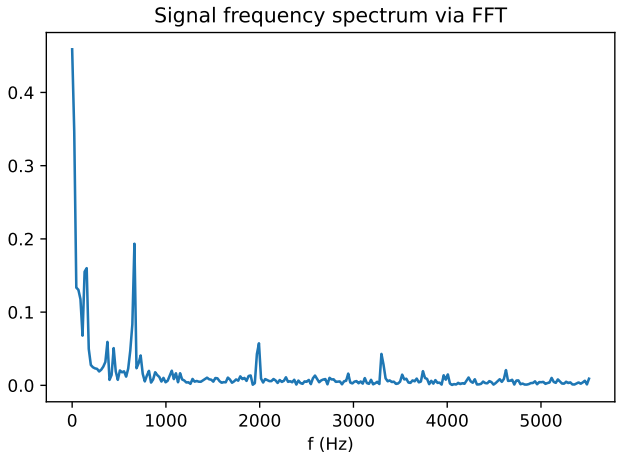

Figure 8: Frequency spectrum of the input data before corruption.

We notice that the most relevant contributions to the song are the peaks below 3 kHz; because of this we corrupt the signal by adding a pure sine wave at 4 kHz, so that our modified signal contains an outlier peak centered around $f_{\text{noise}} = 4$ kHz.
The resulting corrupted signal and its frequency spectrum are shown below.

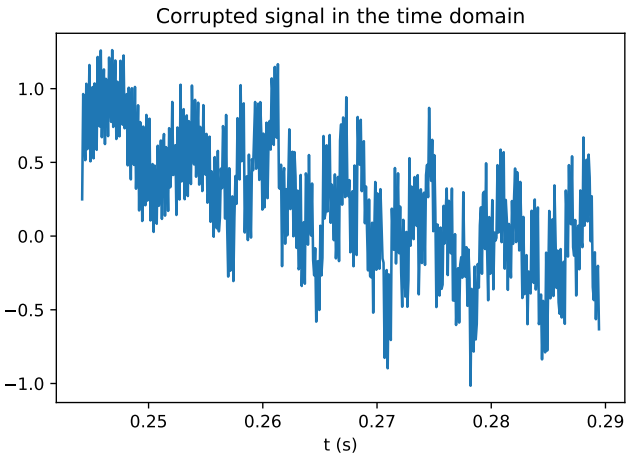

Figure 9: Input data after corruption with a pure sine wave.

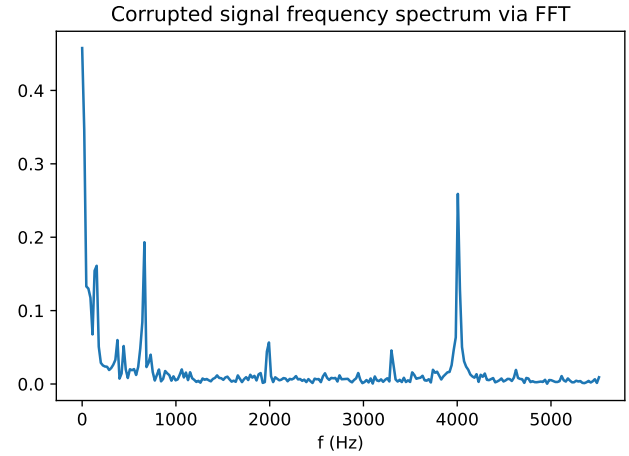

Figure 10: Frequency spectrum of the input data after corruption.

We notice that a new noise peak at 4 kHz has appeared in the frequency spectrum, and that because of this new high frequency component the signal oscillates much more; indeed we have a "beats[6]-like" behaviour, i.e. summing this pure sine wave is equivalent to adding a high-frequency sinusoidal modulation to the signal amplitude.

3.1.2 Choosing f_c to compute FIR coefficients

As we have seen from figure 1 with a fixed value of N the only way to decrease the filter response at the desired frequency is to lower f_c. If we simulate the output of our filter for $f_c = 3$ kHz and for $f_c = 1$ kHz (using code shown later) we can compare the results:

[6]"Battimenti" in italian.

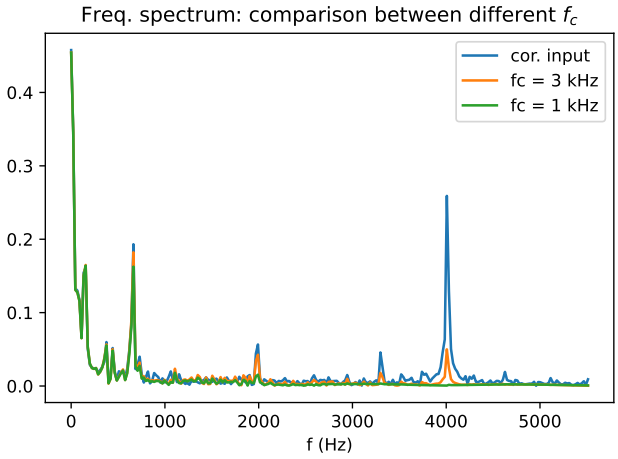

Figure 11: Comparison between frequency spectra of the filter output for different values of the cutoff frequency f_c.

As predicted from the analysis of the "lazy" behaviour shown in figure 1 the lower the cutoff frequency the better our filter can remove the 4 kHz noise - which is why only the 1 kHz cutoff is able to completely remove the noise peak. This comes at the price of being too "aggressive": non-noise peaks around 4 kHz are removed, too; this is a trade-off which must be considered while designing the FIR coefficients. Since in our case most of the interesting harmonics lie in low frequency intervals we deem this behaviour worthy of using, and hence prefer eliminating the noise completely.

3.1.3 Integer rescaling of data and FIR coefficients

Now that we have our input data we notice from figure 9 that this array has values between 0 and 1 in absolute value; we therefore rescale them using an appropriate constant k_{data} (k_d for short) in order to switch to an integer representation which fits inside the 8 available bits. We use the same procedure we described for the FIR coefficients: we choose the largest number k_d that guarantees $|\texttt{int}(xk_d)| \leq 2^8 - 1 = 127$, actually multiply our input array by k_d, cast to `int8`, send everything to the FPGA and then exploit linearity to be sure that dividing by k_d will give the correct output.
We can actually do something even simpler: since the absolute amplitude of our signal holds no physical information (only the relative one does as it's a song) we can simply multiply our data by k_d once and for all and then forget this isn't the original data. This can always be done; indeed even if values in our signal had an absolute physical significance this rescaling would simply be a change of units, which can of course always be implemented.
Having said all of this we notice that the original signal is made of values approximately between -1 and 1 (purely conventional interval, because as we said the amplitude has no absolute meaning), and since we added sinusoidal noise of approximately 0.3 amplitude a value of $k_d = 100$ guarantees the signal casted to integer can fit inside 7 effective bits.

If we now compute the FIR coefficients using `scipy.signal.firwin` for $N = 8$ and $f_c = 1$ kHz we obtain the following values:

$$[0.01062177, 0.05108371, 0.16472057, 0.27357395, 0.27357395, 0.16472057, 0.05108371, 0.01062177]$$

If we set $k_c = 467$, multiply by this value coefficients and then cast to integer we obtain:

$$b_0 = b_7 = 4,\ b_1 = b_6 = 42,\ b_2 = b_5 = 76,\ b_3 = b_4 = 127$$

These coefficients fit inside the `int8` type, but since this k_c isn't a power of 2 bit shifting to the right 9 times inside the actual filter means we're dividing by 2^9 instead of 467; to correct this it suffices to multiply the FPGA output by $2^9/k_c \approx 1.1$ (since this value is very close to 1 we could also do nothing and accept a slightly less accurate result). Now that we have the data and the coefficients to send to the FPGA the only other thing we need to do is to append some zeros at the beginning and end of the input data; the former are used to make equation (1) work for the first values too, while the latter are useful to clean up the pipeline and thus make debugging easier.

3.2 Setting up a serial port-based communication system

Python module `pyserial` was used to write to/read data from the serial port; in order to actually use the serial module we first had to convert our values to/from the `bytes` type using the `chr`/`ord` functions. It's important to note that these functions only accept/return *unsigned* integers - which implies we had to use the *2's complement* convention. This simply means that the $[-128, 127]$ interval was mapped to the $[0, 255]$ interval; one half of the $[0, 255]$ was reserved for the positive numbers interval $[0, 127]$, whereas the other for negative values $[-128, -1]$. To communicate with the FPGA, then, we simply check in which half-interval any value resides and shift it accordingly.
A simple python implementation of this conversion is shown below.

```
# n = input value, a = output value, ser = Serial object
if n < 0 :
    x = chr(255+n)
else :
    x = chr(n)
ser.write(x)
y = ord(ser.read())
if y > 127 :
    a = y - 256
else :
    a = y
```

3.3 FIR output `python` simulation

To simulate the predicted filter output we can easily turn equation (1) into a python function as follows.

```
def fir_int(input, coeffs_int, N_taps) :
    output_int = np.zeros(input.shape[0])
    for n in range(7, len(output_int)) :
        for i in range(N_taps) :
```

```
5                output_int[n] += int(coeffs_int[i]*input[n-i
    ])
6        return output_int
```

This not only lets us have the signal to compare to the actual output, but it also lets us see that working with integers instead of floats brings no significant loss of precision:[7]

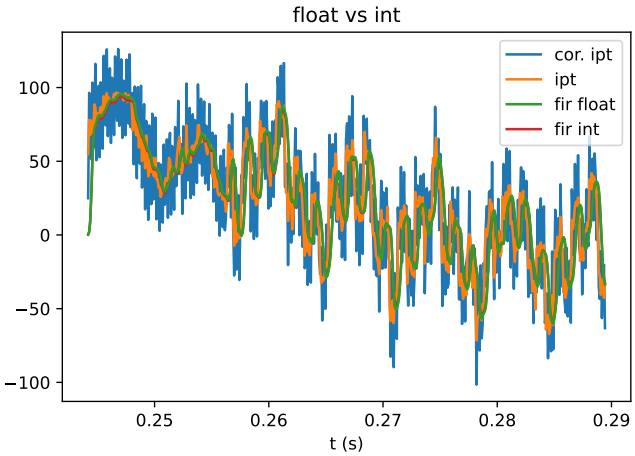

Figure 12: Comparison between integer and float versions of the filter.

We notice that the two simulations are basically indistinguishable.

3.4 Time domain analysis

We now plot what was read from the serial port.

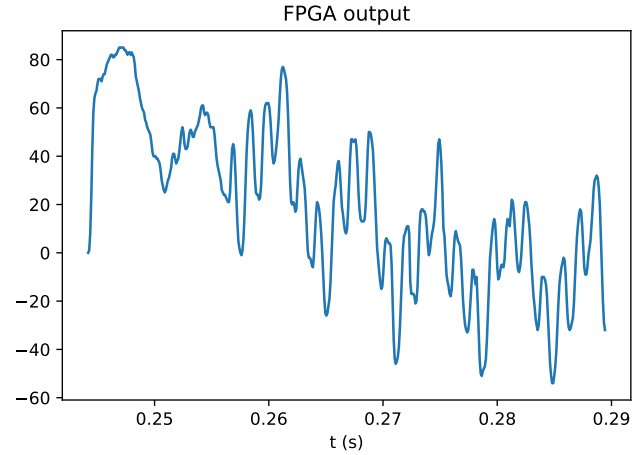

Figure 13: FPGA output.

[7] The float output is computed by omitting the `int` function and using the float version of the coefficients.

For curiosity's sake we now ask: *what happens if we discard the wrong bits before sending FIR output to the uart transmitter?* If for example we naively keep only the 8 most significant bits the result is what follows:

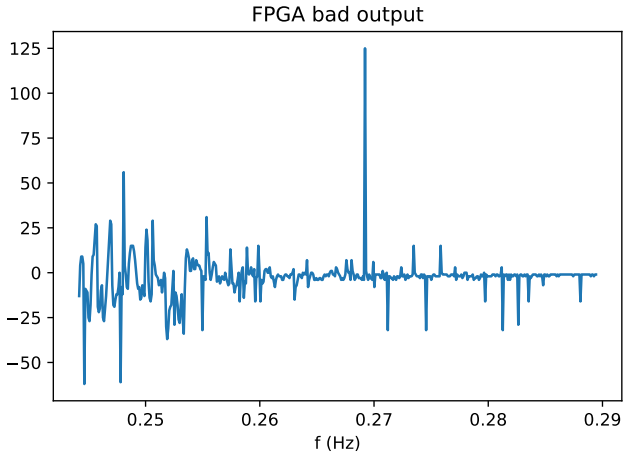

Figure 14: FPGA output if we keep the 8 leftmost bits.

The result is clearly wrong and looks nothing like the simulations; this is one way one can learn which bits to keep, though!

We finally compare FPGA output with either of the two FIR output simulations.

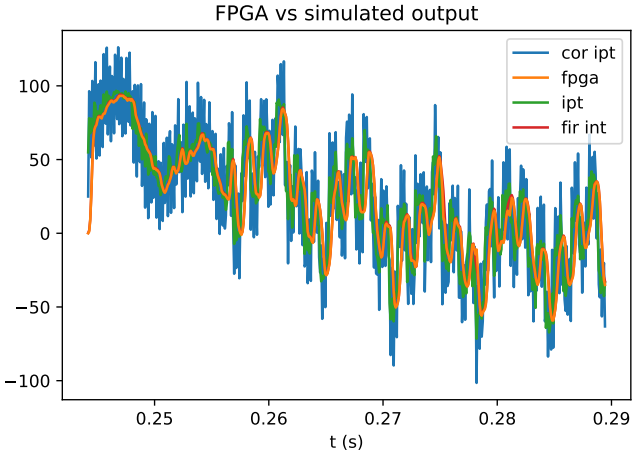

Figure 15: Comparison between simulated and actual output.

The fact that the orange line makes the red one almost invisible confirms that everything seems to be working correctly; we also notice that these signals are a good approximations of the original (green) signal, and that their harmonic components definitely do not reach frequencies as high as the ones in the corrupted (blue) signal.

In fact we can't help but notice that the filter output oscillates somewhat less wildly than the original signal; this is a consequence of the fact that we were quite aggressive in our choice of f_c and hence in the removal of the high frequency components (even of those we should in principle have left there).
To make this point more quantitative we plot the frequency spectrum of these 4 signals.

3.5 Frequency domain analysis

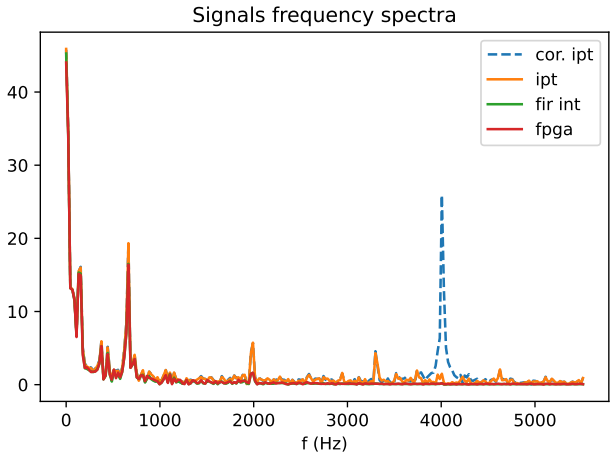

Figure 16: Comparison between the frequency spectra of the original, corrupted and filtered signals.

As expected the FIR filter completely removes the 4 kHz noise - but at the cost of smoothing the right part of the spectrum a bit too much. This compromise is made unavoidable by the low value of N we used, but since it's not too bad we consider it acceptable.
Notice that due to the lack of time we weren't able to produce the same plot for $f_c = 3$ kHz, but it's easy to guess that the new frequency spectrum would be more similar to the original except for a not completely eliminated peak at 4 kHz.

4 Conclusion

A low-pass FIR filter with $N = 8$ and $f_c = 1$ kHz has been successfully implemented on a FPGA board; in order to do so we designed and programmed the pipeline in VHDL as well as the whole data preprocessing procedure (to make some nontrivial conversions between signed float and unsigned integers possible).
Many challenges had to be overcome to achieve this result:

- It was very hard to actually receive data from the FPGA since the `ser.write` function was very unreliable; each time we executed the python script the number of values we could receive before an error occurred quickly decreased, and constrained us to only work with

500 values - forcing us to discard our original plan to filter and then listen an entire song.

- Connecting to the server and configuring the FPGA was not trivial; learning to use `ssh`, `scp` and to properly setup the `make` procedure was somewhat time consuming.
- Every time the `ser.write` function failed the only way to re-establish functioning communication with the FPGA was reprogram it again; this solution required some time to find, and once found it significantly slowed down the debugging process.

We end the present work with two final remarks.

1. To improve the quality of the approximation (and exploit the fact that the FPGA itself isn't limited to processing 8 bit numbers) one could in principle modify the uart protocol in order to be able to exchange longer bitstreams, but even if we kept the same receiver/-transmitter we could theoretically improve the result by freeing up the sign bit. Using an affine transformation it's always possible to map any value of the original signal to the $[0, 255]$ interval; not only is this easy to implement and in principle capable of improving the result, but it also frees us from the necessity to use the 2's complement convention in the serial script.
2. This whole project shouldn't be used as a true noise-cancelling tool to clean up corrupted audio files, since we somehow "cheated" by a) injecting unphysical noise, and b) exploiting the knowledge about how this signal was created to choose f_c in advance. In real world applications *adaptive filters* are actually used.

5 References

1. *Finite impulse response* - Wikipedia
2. GitHub repository containing code and data
3. How to implement a FIR filter in VHDL - surf-vhdl.com