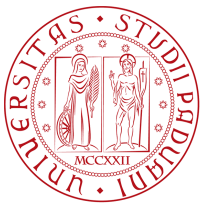

Managment and Analysis of Physics
Datasets mod.A , A.Y. 2021/22
Group: 14

Federico Chiarello 2027166 federico.chiarello@studenti.unipd.it
Giacomo Franceschetto 2053348 giacomo.franceschetto@studenti.unipd.it
Giulia Bellentani 2027590 giulia.bellentani@studenti.unipd.it
Tommaso Amico 2054478 tommaso.amico@studenti.unipd.it

FPGA-based Channel Selection Implementation for Audio Signal Processing

22-02-2022 

ABSTRACT

In the following paper we report the results obtained by the use of a FIR filter, implemented through an FPGA, to perform a channel selection operation. We worked with stereo audio signals using the Digilent Pmod I2S2 interface board. The correct implementation of the filter on the FPGA has been verified through comparison with a Python simulation, in which we also showed a simple use of this channel selection filter: the selection of the desired channel in an AM radio receiver.

CONTENTS

1 INTRODUCTION

Channel selection is a fundamental operation for several applications regarding data transmission. In this project, we tried to use a FIR filter implemented on an FPGA to perform a channel selection operation for an audio signal. The interface between the FPGA and the audio signal is performed through the Pmod I2S2 module, which allows the transmission and reception of stereo audio signals via the I2S protocol. This board is equipped with two audio jacks which allowed us to use a PC as a source and recorder of the stereo audio signal. The interaction between the implemented FIR filter and the I2S2 Pmod is achieved through the AXI4-Stream protocol within the *FIR controller* component. On the other hand, the translation from I2S and AXI4-Stream protocol is done via the *Axis i2s2* component provided by the manufacturers of the Pmod I2S2 in a demo project.

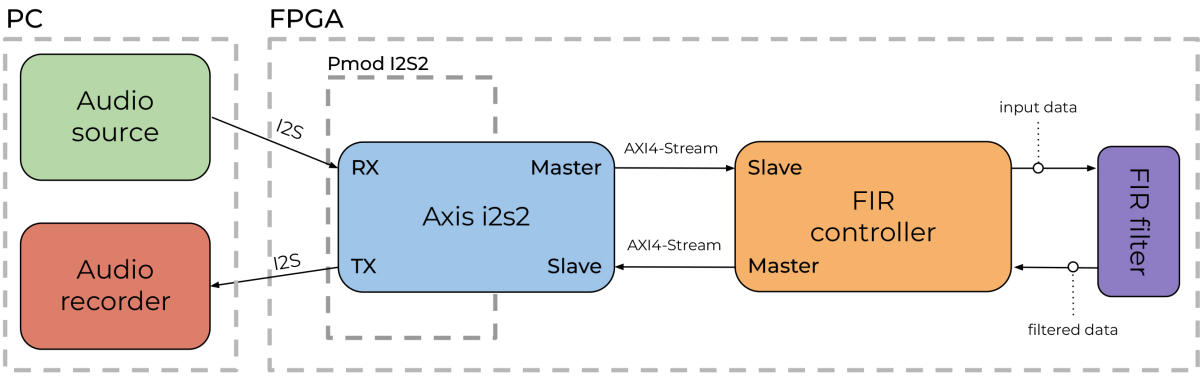

Figure 1: Project schematics

This paper is structured as follows: in Sections 2 and 3, we discuss how the AXI4-Stream protocol and the FIR filter work, respectively, and how they have been implemented in VHDL; Section 4 mainly presents the comparison between the results obtained with the VHDL hardware implementation and a Python simulation, but also shows a classical simple application example of the channel selection filter (AM radio receiver); finally Section 5 explores an extension of this project towards a model of an audio equalizer.

2 AXI4-STREAM PROTOCOL

AXI4-Stream is a protocol that permits to transfer data from the master to the slave based on the handshake signal. It is composed of the following signals: the transmission data (tdata), the boundary of the data packet (tlast), the ready for receiving of slave module (tready) and the one that indicates that the master is driving a valid transfer (tvalid). Only when the tvalid of the master module

and the tready of the slave module are both high, the handshake is successful and then the data transmission will be started.

2.1 FIR controller implementation

With the aim of accessing the data passing through the Pmod I2S2 we went to build an interface of the AXI4-Stream protocol for the FIR filter. The `fir_controller` entity 2.1 therefore presents all signals necessary for the transfer of incoming data, `slave` signals, and outgoing data, `master` signals.

```
entity fir_controller is
   generic(
      DATA_WIDTH : integer:=24 -- bit width of data
   );
   port (
     i_clk  : in std_logic;

     s_axis_tdata : in std_logic_vector(DATA_WIDTH-1 downto 0);
     s_axis_tlast : in std_logic;
     s_axis_tvalid : in std_logic;
     s_axis_tready : out std_logic;

     m_axis_tdata : out std_logic_vector(DATA_WIDTH-1 downto 0);
     m_axis_tlast : out std_logic;
     m_axis_tvalid : out std_logic;
     m_axis_tready : in std_logic
   );
end fir_controller;
```

Listing 2.1: Fir controller entity

As explained above, data are only transferred when the handshake between `tvalid` and `tready` takes place. In addition, the internal signals `_int` take over the situation where the `tvalid` occurs before the `tready` and therefore you have to wait for the slave interface to be ready to receive data.

```
  s_axis_tready_int <= m_axis_tready or not m_axis_tvalid_int;
  m_axis_tvalid <= m_axis_tvalid_int;
  s_axis_tready <= s_axis_tready_int;

  fir_controller: process(i_clk)
  begin
   if rising_edge(i_clk) then
      if s_axis_tvalid = '1' then
        m_axis_tvalid_int <= '1';
      elsif m_axis_tready = '1' then
        m_axis_tvalid_int <= '0';
      end if;
```

```
    if s_axis_tvalid = '1' and s_axis_tready_int = '1' then
      m_axis_tlast <= s_axis_tlast;
      m_axis_tdata <= filtered_data;
    end if;
  end if;
end process;
```

Listing 2.2: Fir controller implementation

3 FIR FILTER

In signal processing, a digital filter is a system that performs mathematical operations on a sampled, discrete-time signal to reduce or enhance certain aspects of that signal. 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. For a FIR filter of order $N-1$, each value of the output sequence is a weighted sum of the most recent (N) input values. This computation is also known as discrete convolution, and, for time n, it is written as follows:

$$y[n] = \sum_{i=0}^{N-1} b_i x[n-i] \tag{1}$$

where $x[n]$ is the input signal at time n, $y[n]$ is the output signal at time n, and b_i are the filter coefficients.

It should be noted that, working with finite signals, our filter will have a transient phase equal to the time it takes to load N data packets into memory before it will work properly.

3.1 FIR filter implementation

Before going into the details of the implementation of the FIR filter, it is worth pointing out some technical decisions that we have taken in order to meet the requirements of obtaining a module that is as flexible and simple as possible when varying the filter parameters.

First of all, we preferred to use only integers to simplify arithmetic operations in the FPGA. However, the coefficients of a filter are usually of the type $c_i \in (-1, 1)$. Therefore, we had to introduce a first scaling step to make the coefficients integer, and a second step to rescale the output and compensate for the effect of the initial scaling. A good compromise has been found in representing the coefficients with 24-bit, the same bit width with which data packets are represented. Consequently, we introduced a scaling factor of 2^{22},

which allows us to maintain an accuracy[1] close to the 0.001% for the coefficients.

Next, we have to figure out how to represent the filter parameters in the most convenient way. A filter of `N_TAPS` will need to store `N_TAPS` data and `N_TAPS` coefficients, so we decided to define matrices `N_TAPS*DATA_WIDTH` for the data and `N_TAPS*COEFF_WIDTH` for the coefficients. To be able to deal with a variable number of taps, the structure of the coefficients was defined as an unconstrained array and integrated into the design.

```
package fir_pkg is
      type coeff_array is array(natural range <>) of signed;
end package;
```

Listing 3.1: Type for coefficients

3.1.1 Entity

FIR filter entity will depend on specific information about the filter architecture[2] defined in the `generic` clause. In the port clause it's worth mentioning, among all the usual elements like clock and input data/output data, the presence of `i_valid` , a label for validating or not validating the data.

```
entity fir_filter is
   generic(
      N_TAPS  : integer:=11; -- number of data to store in memory to
          perform filter operation
      DATA_WIDTH : integer:=24; -- bit width of data
      COEFF_WIDTH: integer:=24; -- bit width of coeff
      SCALING : integer:=22 -- scaling factor for coefficients -> should
          be coeff_width -2(signed) if coeff in (-1,1)
   );
   port (
      i_clk : in std_logic;
      i_valid : in std_logic;
      i_coeff : in coeff_array;
      i_data : in std_logic_vector(DATA_WIDTH -1 downto 0);
      o_data : out std_logic_vector(DATA_WIDTH -1 downto 0)
   );
end fir_filter;
```

Listing 3.2: Fir filter implementation

[1] Accuracy computed as $\max_i$ of $(c_i - \text{int}(c_i \cdot 2^{22})/2^{22})/c_i \; \forall i$

[2] Default values are the ones used in the application that will be presented later in this report.

3.1.2 Processes

We start by defining all the data structures that are needed to calculate the discrete convolution (1) that is the core of the FIR filter.
The signal `s_data` is of the type `t_data_pipe`, a multidimensional array designed to contain the specific portion of the input data that is used to obtain $y[n]$ at time n, i.e. the `N_TAPS` most recent `i_data` arrays. We already introduced this choice for the data matrix format in the beginning of this very subsection 3.1, where we also explained the choice for the coefficient matrix that was initialized before in a separate package.

The signal `s_mult` is of the type `t_mult`, another multidimensional array: it has the `N_TAPS*(COEFF_WIDTH+DATA_WIDTH)` size necessary to store the result of the first step of the discrete convolution, i.e. the multiplication between the aforementioned `s_data` and the coefficients.

Then we have the signals `s_add_0` and `s_add_1`, both dedicated to the summation step, that is separated in two substeps for optimization and parallelization purposes. The signal `s_add_0` is of type "`t_add_0`", the last multidimensional array, which has the size $\frac{\texttt{N\_TAPS}}{2}$`*(COEFF_WIDTH+DATA_WIDTH+1)`, necessary to store the results of the sums between the adjacent even and odd members of the whole summation. The signal `s_add_1` instead is a simple one-dimensional array of signed that has the size `COEFF_WIDTH+DATA_WIDTH+2` and contains the result of the last sum, the one between all the arrays in `s_add_0`.

```
type t_data_pipe is array (0 to N_TAPS-1) of signed(DATA_WIDTH -1
    downto 0);
type t_mult is array (0 to N_TAPS-1) of
    signed(COEFF_WIDTH+DATA_WIDTH-1 downto 0); -- size[n*m] =
    size[m]*size[n]
type t_add_0 is array (0 to N_TAPS/2 -1) of
    signed(COEFF_WIDTH+DATA_WIDTH downto 0); -- size[n+m] =
    size[m]+size[n]+1

signal s_data : t_data_pipe:= (others => (others => '0'));   --
    container for data
signal s_mult : t_mult:= (others => (others => '0'));        --
    container for coeff*data
signal s_add_0 : t_add_0:= (others => (others => '0'));      --
    container for first sum ( sum all pairs )
signal s_add_1 : signed(COEFF_WIDTH+DATA_WIDTH+1 downto 0):= (others
    => '0'); -- container for second sum
```

Listing 3.3: Data structures

Then, we fill up the first data structure (`s_data`): we stack a new 24 bit input array at the top of the data matrix and we shift down by one position the previous ones, to remain always with exactly `N_TAPS` arrays.
We do so only if the clock is in the rising edge and the input valid is high. Is worth mentioning for good that all the processes explained in the rest of this subsection are triggered by a rising clock edge.

```
p_input : process (i_clk)
begin
   if(rising_edge(i_clk) and i_valid = '1') then -- stack new sample
       at the top, shift the others
     s_data <= signed(i_data)&s_data(0 to s_data'length-2);
   end if;
end process p_input;
```

Listing 3.4: Input data process

In the multiplication process we fill up the `s_mult` matrix: we multiply the `s_data` elements by the `i_coeff` elements inside a `N_taps` long loop.

```
p_mult : process (i_clk) -- multiply data with the coefficients
begin
    if(rising_edge(i_clk)) then
       for k in 0 to N_TAPS-1 loop
          s_mult(k) <= s_data(k) * i_coeff(k); -- size bin casting is
              automatic
       end loop;
    end if;
end process p_mult;
```

Listing 3.5: Multiplication process

As anticipated above, in the first summation process we sum the adjacent even and odd members of the whole summation inside a $\frac{N\_taps}{2}$ step loop. In contrast to what happened in the multiplication process, here the resizing is not automatic, so we have to call the resize function to add one bit to the width of each of the two addends.

```
p_add_0 : process (i_clk) -- sum - step 1
begin
    if(rising_edge(i_clk)) then
       for k in 0 to N_TAPS/2-1 loop
          s_add_0(k) <= resize(s_mult(2*k), COEFF_WIDTH+DATA_WIDTH+1) +
              resize(s_mult(2*k+1), COEFF_WIDTH+DATA_WIDTH+1);
       end loop;
    end if;
end process p_add_0;
```

Listing 3.6: First summation process

The second summation process consists in summing all the rows in the `s_add_0` matrix and eventually also the last row of `s_mult` matrix: this additional step is needed only when `N_TAPS` is chosen as odd. In fact, in this case the first summation process doesn't consider the aforementioned last row of `s_mult` matrix due to the summation of only "odd/even indexed pairs". For this purpose, we exploit again both a $\frac{N\_taps}{2}$ long loop and the resize function (in the very same way described before).

```
p_add_1 : process (i_clk) -- sum - step 2
variable tmp: signed(COEFF_WIDTH+DATA_WIDTH+1 downto 0):= (others =>
    '0');
begin
    tmp := (others => '0');
    if(rising_edge(i_clk)) then
       for k in 0 to N_TAPS/2-1 loop
          tmp := tmp + resize(s_add_0(k), COEFF_WIDTH+DATA_WIDTH+2);
       end loop;
       if(N_TAPS mod 2 /= 0) then -- odd
          tmp := tmp + resize(s_mult(N_TAPS-1),
             COEFF_WIDTH+DATA_WIDTH+2);
       end if;
       s_add_1 <= tmp;
    end if;
end process p_add_1;
```

Listing 3.7: Second summation process

In the end, the result `s_add_1` is a one-dimensional array, but of a different width from the one of the input data: here we have a `COEFF_WIDTH+DATA_WIDTH+2` array width, while the input data `i_data` array was just `DATA_WIDTH` long.

Previously, we explained that, even if we decided to work with integer coefficients, the original ones for this kind of FIR filter were in the range $(-1, 1)$. Therefore, this last process is a rescaling process, with a scaling factor that depends on the chosen `COEFF_WIDTH`. Consequently, we must first divide `s_add1` by the scaling factor applied to the coefficients. Since the factor is a power of two, this simply translates to excluding the rightmost `SCALING` bits. Then, we have to extract `DATA_WIDTH` bits from `DATA_WIDTH+COEFF_WIDTH-20`[3] . To do this we decided to select the rightmost `DATA_WIDTH` bits since there should be no results greater than $2^{\texttt{DATA\_WIDTH-1}}$ and signed vectors feature the property of sign extension.

In the end, the output `o_data` has the same width of the input `i_data`.

[3] Recall that *size*(`s_add1`) = `DATA_WIDTH+COEFF_WIDTH+2`, and, after the scaling it becomes `DATA_WIDTH+COEFF_WIDTH+2-22`

```
p_output : process (i_clk) -- compute output
begin
    if(rising_edge(i_clk)) then
       o_data <= std_logic_vector(s_add_1(DATA_WIDTH-1+SCALING downto
           SCALING)); -- recall sign extension in signed type
    end if;
end process p_output;
```

Listing 3.8: Output data process

4 PYTHON COMPARISON

In this section, we want to show the simulation performed in Python of our FIR filter and then compare it with the actual implementation.

4.1 Filter design

We began by initializing the filter using the *scipy* library through the *scipy.firwin* construct.

```
numtaps = 11
f = [9950,10050]
fs = 44100
coeff = signal.firwin(numtaps, f, pass_zero ='bandpass', fs=fs)
```

Listing 4.1: Filter configuration

Where we defined the `numptaps` variable, i.e. the length of the filter. The variable `f` defines the band of the filter in Hz, while `fs` is our sample frequency. Let's now see the plot of the filter:

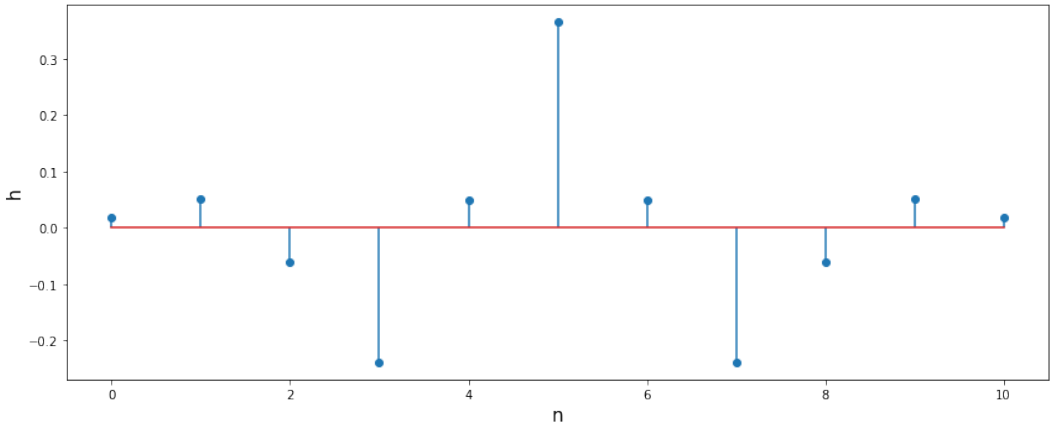

Figure 2: Plot of the filter coefficients

```
N=len(coeff)+len(sig_audio_1)-1 #Length of the convolution
fft_coeff = fftpack.fft(coeff,N)
```

```
pow = np.abs(fft_coeff)#amplitude response
freqs_H = fftpack.fftfreq(N)
```

Listing 4.2: Filter DFT

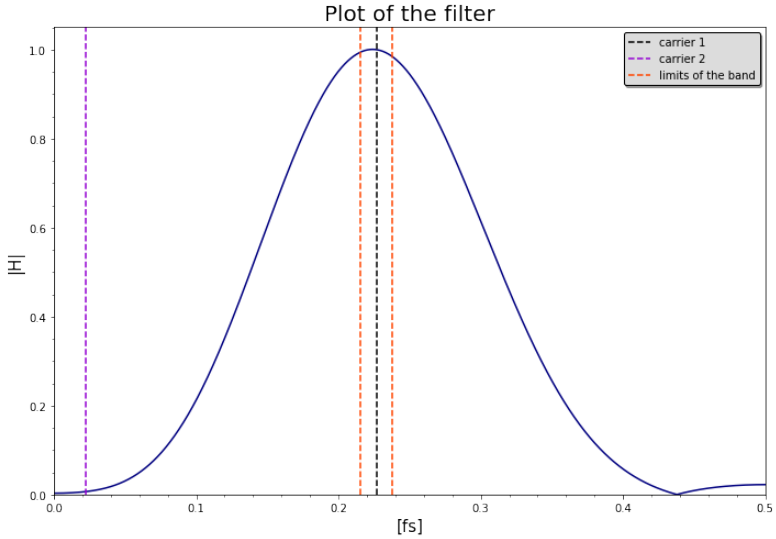

Figure 3: Plot of the filter magnitude in the frequency domain

In Figure 3, we marked in red the upper bound and the lower bound of the band, in black and in purple are plotted the frequency of our two carriers (we will define this concept better in the following sections) that are respectively a 10kHz signal and a 1kHz signal. We can see that the 10kHz signal is at the center of our band, as it should be, where the filter has a gain of 1. The x axis is expressed in units of the sample frequency as the label says.

By the plot of the filter coefficients in Figure 2 it is easy to see that the filter we are using is a FIR TYPE 1 (odd length-even symmetry). This type of filters introduce a linear phase shift equals to $\theta(\omega) = -\frac{Ntaps-1}{2} \times \omega$.
The linear phase delay is appreciable in Figure 4. We have removed the delay by introducing a correcting factor.

```
phase = np.angle(fft_coeff) #computing the phase
exp=(1j*(numtaps-1)*0.5)*freqs_H*2*math.pi
correction = np.exp(exp)
phase_correction = np.angle(correction)
fft_coeff_corr = fft_coeff*correction #correcting the phase
phase_corr = np.angle(fft_coeff_corr)
```

Listing 4.3: Phase analysis

4.2 Filter testing

We want now to see the filter in action, in order to do that we take two signals: one at 10kHz and one at 1kHz. Being our band between 9950Hz and 10050Hz,

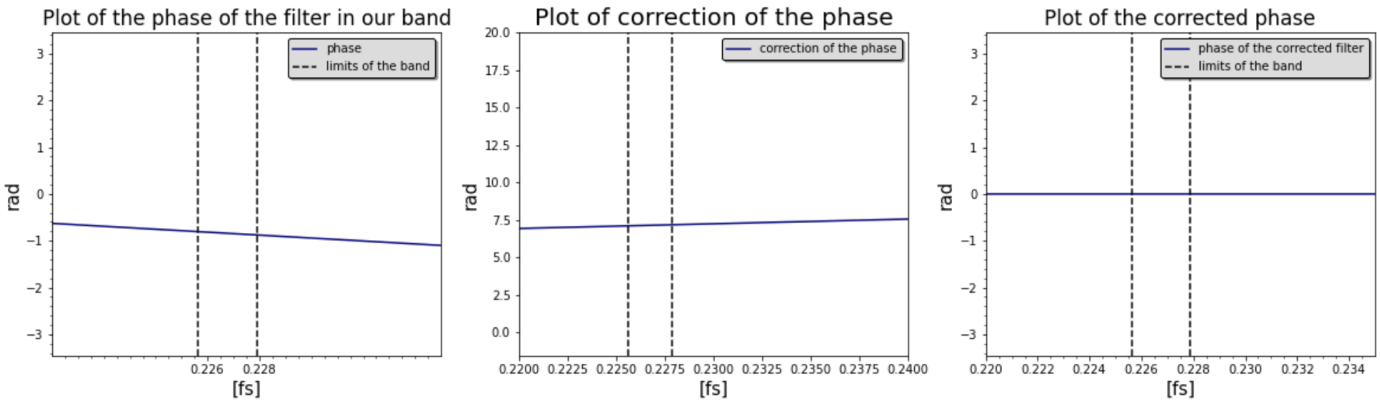

Figure 4: Plot of the phase, the correction and the corrected phase

the correct behaviour of the filter should be that of cancelling out the signal at lower frequency and leaving the higher frequency one untouched.
In order to see that we provide as an input for the filter the sum of the two signals, we expect that the filtered one contains only the 10KHz signal.
Below we show the code through which we imported the signals and then normalized them.
In Figure 5, we instead see the sum of the two signal on the left and in the image on the right it is possible to see how the filtered signal coincides with the 10kHz signal.

```
freq_sample_1, sig_audio_1 =
    wavfile.read("10kHz_44100Hz_16bit_05sec.wav")#10 kHz signal
freq_sample_2, sig_audio_2 =
    wavfile.read("1kHz_44100Hz_16bit_05sec.wav")#1 kHz signal
freq_sample_5, sig_audio_5 = wavfile.read("10KHz_sum_1KHz.wav") #sum
    signal
#Normalization
pow_audio_signal_1 = sig_audio_1 / np.power(2, 16)
pow_audio_signal_2 = sig_audio_2 / np.power(2, 16)
pow_audio_signal_5 = sig_audio_5 / np.power(2,15)
time_axis = np.arange(0, len(pow_audio_signal_1), 1) /
    float(freq_sample_1)
fft_somma = fftpack.fft(pow_audio_signal_5,N)
out = fft_somma*fft_coeff
out_corr = fft_somma*fft_coeff*correction
time_output = fftpack.ifft(out,N)
time_output_corr = fftpack.ifft(out_corr,N)
time_output_1=time_output[numtaps-1:]
time_output_corr_1=time_output_corr[numtaps-1:]
```

Listing 4.4: Importing .wav file

In addiction to this we had to shift the time axis of the filtered signal subtracting $-\frac{10}{\text{freq\_sample}}$ to take into account the fact that we removed the firsts 10 units of the time output array (the inverse fast fourier transform of the

output of the filter), representing the transient. We can see that all we have said

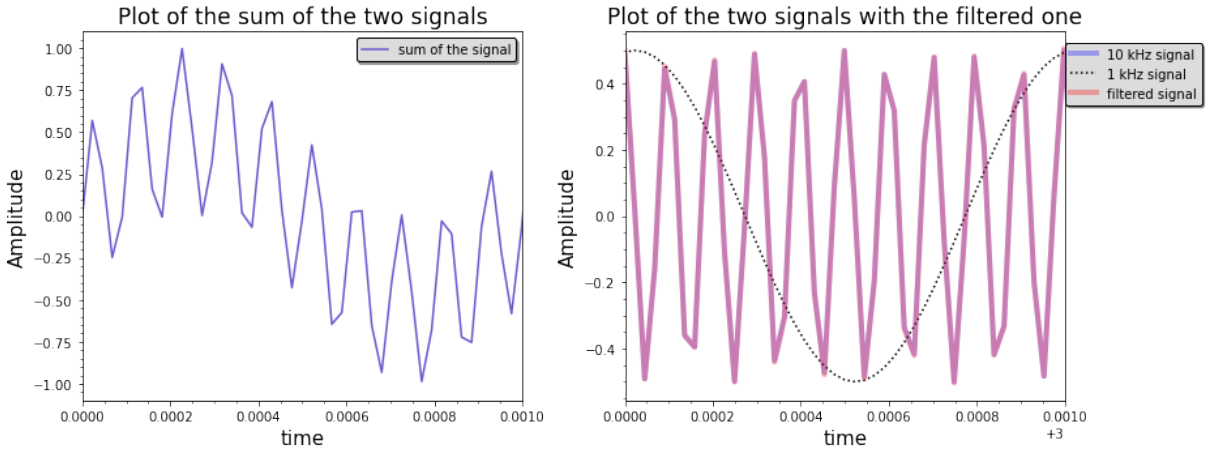

Figure 5: Plot of the before sum and after the filter

in the paragraph above is confirmed in the plot of the power spectrum, showed in Figure 6.

```
fft_audio_1 = fft_somma / N #normalization
fft_audio_2 = out / N #normalization
```

Listing 4.5: Power spectrum

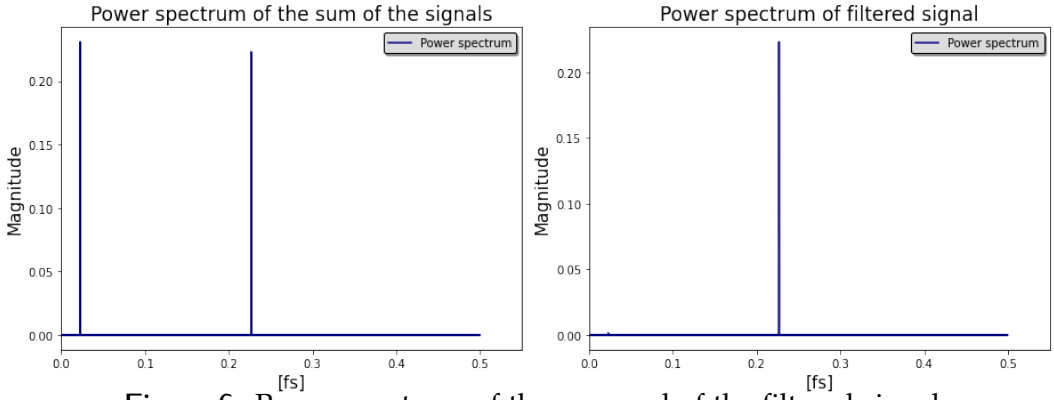

Figure 6: Power spectrum of the sum and of the filtered signal

4.3 A real world application

To emulate a possible real world application of our filter we consider the amplitude modulation, one of the simplest way to transmit radio signals. In this modulation technique the signal to transmit is composed by two parts, the modulator and the carrier.
The modulator is usually the signal we try to convey, the one containing the information, through it we modulate the carrier that has the task to define the frequency of the final signal. Using a passband filter we can decide which carrier, so which "channel", to listen to and remove all the others.

The filtered signal should have the frequency of the carrier and amplitude of the modulator. Through it we can then piece together the original signal.

We take into consideration two different carriers: one inside our band and one outside of it that should be nullified by the filter, we choose the same 10kHz and 1kHz signals used above.The modulator has usually a frequency that is much lower then of the one the carriers, we then choose to import a 100Hz signal to use as modulator.
Here it's shown how we imported the signal, normalized it, and computed the time output of the multiplication.

```
freq_sample_4, sig_audio_4 =
    wavfile.read("audiocheck.net_sin_100Hz_-3dBFS_5s.wav")#modulator
pow_audio_signal_4 = sig_audio_4 / np.power(2, 15) # Normalization
#modulator*carrier_1
multiplication_1 = pow_audio_signal_4*pow_audio_signal_1

out_p1 = fft_multiplication_1*fft_coeff
out_corr = fft_multiplication_1*fft_coeff_corr

time_output = fftpack.ifft(out_p1,N)
time_output_corr_2 = fftpack.ifft(out_corr,N)

time_output=time_output[numtaps-1:]
time_output_corr_2=time_output_corr_2[numtaps-1:]
```

Listing 4.6: Carrier 1

In Figure 7 we show the plot of the 10kHz signal and the filtered one, we can notice that the frequency of the filtered signal has in fact the same frequency of the carrier. We don't show the modulator for better clarity.

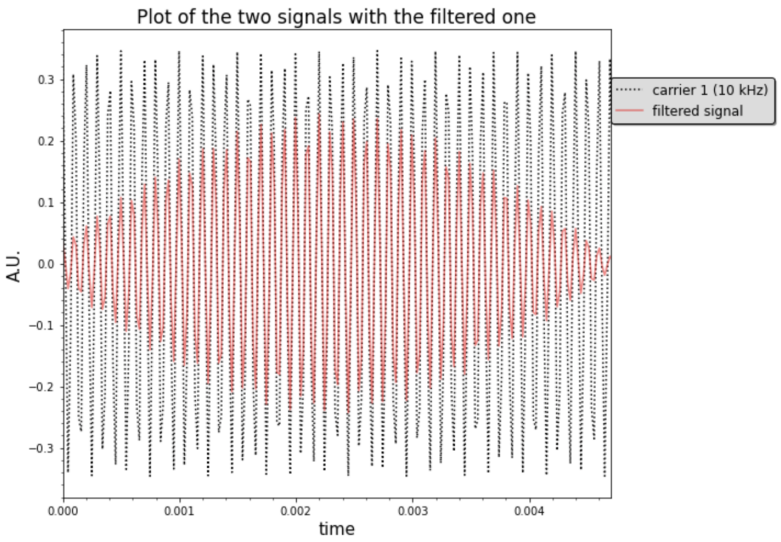

Figure 7: First carrier and relative filtered signal

We now repeat the same operations choosing this time as the carrier the 1kHz signal, falling outside our band. We see how the plot in Figure 8 gets modified accordingly.

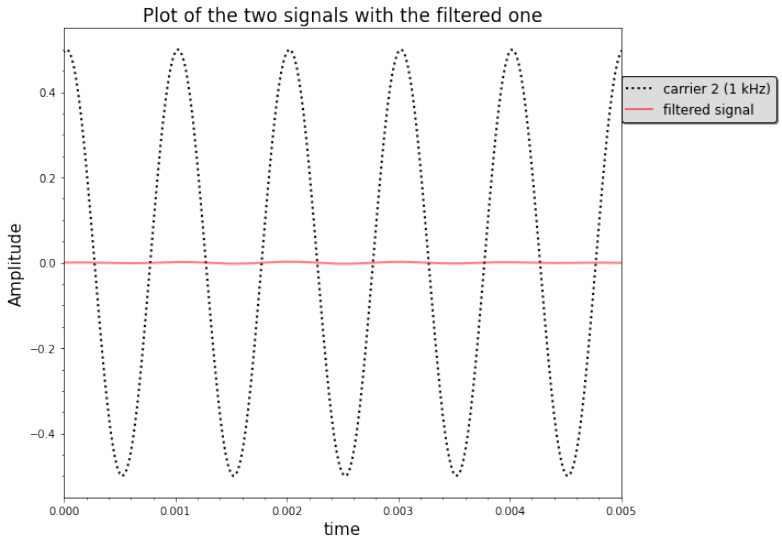

Figure 8: Second carrier and relative filtered signal

These results are confirmed by the power spectrum, showed in Figure 9.

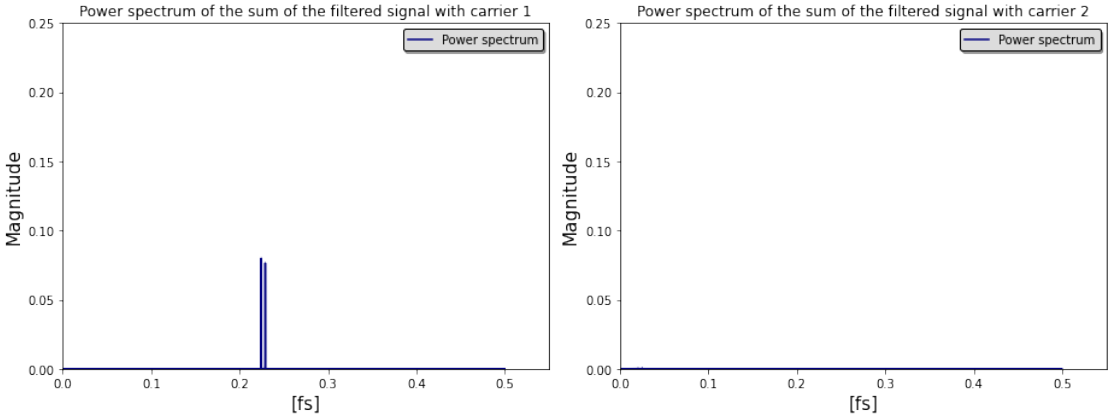

Figure 9: Second carrier and relative filtered signal

In both plots, performed with the *matplotlib* package, we set the keyword argument `scaley=False` to better highlight the `y axis` difference between the two plots.

4.4 Comparison with the FPGA implementation

In this subsection, we want to compare what we have done in Python with the actual FPGA implementation of the filter.
We put care in the length of the array of amplitudes returned by the *wavfile.read* function that is not exactly the same as the one returned by the file used for the Python simulation.

Therefore, the N variable defined at the beginning of this section has been replaced by the equivalent one N_{FPGA}. The comparison of the filtered signals of the sum of the 10kHz wave with the 1kHz wave are shown in Figure 10.

```
fft_fpga = fftpack.fft(pow_audio_signal_3,N_fpga)/N_fpga
freq_fpga = fftpack.fftfreq(N_fpga)
```

Listing 4.7: DFT FPGA filtered signal

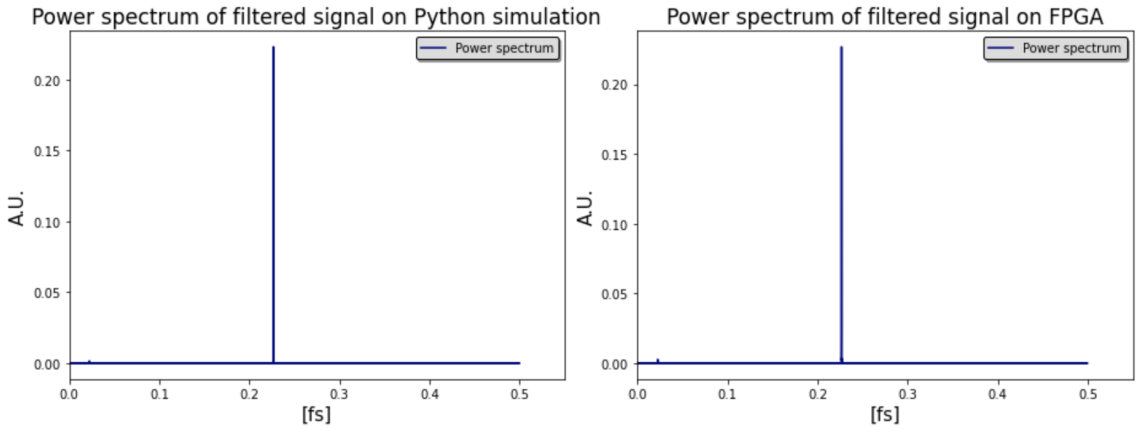

Figure 10: Power spectrums of the sum signal filtered

We can notice how the two power spectrum are similar, yelling a good performance of the implemented filter.

In Figure 11 we instead plotted the relative time output. In the image we show the signal in the proximity of 3 seconds where the signals do overlap.

```
time_axis_2 = np.arange(0,N_fpga,1)/float(freq_sample_3)
```

Listing 4.8: Time axis shift

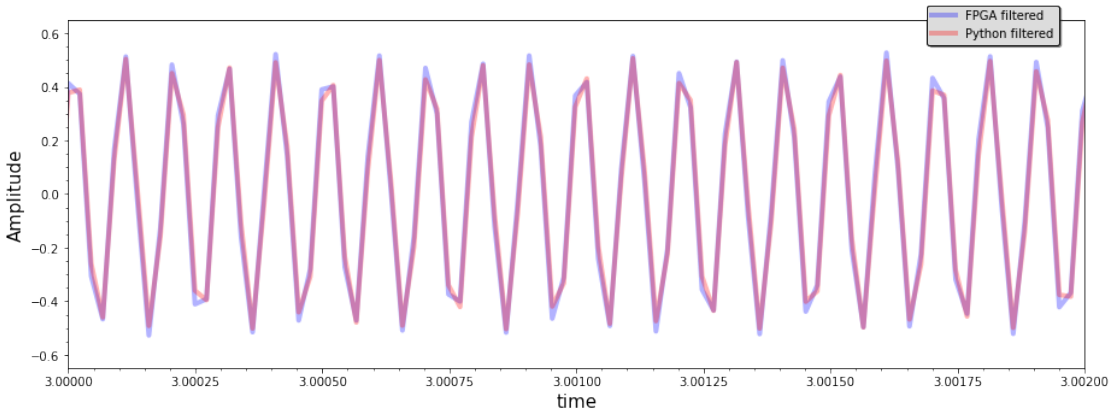

Figure 11: time outputs of the sum signal filtered

However, we noticed that focusing on another region of the time axis (e.g., around 1s) the two sinus like functions are not overlapping anymore, suggesting that the signal returned by the FPGA does not have a linear phase.

We think that this is due to the recording procedure we used to capture the FPGA-filtered signal.

5 EXTENSION – A MODEL AUDIO EQUALIZER

After the implementation of the architecture for a channel selection operation we tried to extend the project to a simple model of an audio equalizer.
An audio equalizer must have the ability to apply different gains according the frequency of the signal. We have therefore created an audio equalizer that filters two bands from one signal, and applies a gain of factor 2 to the high frequency band and decreases the intensity of the low frequency band by the same factor. This specific configuration was chosen because usually if we need that out audio signal travels a long distance, we will have no problem with low frequency signals since they can travel long distances easily, but high frequency signals suffer a high attenuation with the distance.

Starting from the top structure of the project for the channel selection operation we then used two `FIR filter` modules to select the two respective bands and then the gain and attenuation were managed within the `FIR controller`. In addition, the possibility of overflow was handled for high frequencies. Therefore, the two filtered signal after gain/attenuation are summed, part of the code is shown below 5.1.

```
abs_signal := abs(filtered_data_HP(DATA_WIDTH-1 downto 0)&'0');
 if(abs_signal(DATA_WIDTH-1)='1' and filtered_data_HP(DATA_WIDTH-1)='1')
     then
   amp_signal := (DATA_WIDTH-1 downto 0 => '1');
elsif(abs_signal(DATA_WIDTH-1)='1') then
   amp_signal := '0'&(DATA_WIDTH-2 downto 0 => '1');
else
  amp_signal := filtered_data_HP(DATA_WIDTH-2 downto 0)&'0';
end if;
if( hp='0' and lp='0') then
  sum_signal := (others => '0');
elsif (hp='0') then
   sum_signal := resize((DATA_WIDTH downto DATA_WIDTH-1 =>
                      filtered_data_LP(DATA_WIDTH-1))&
                      filtered_data_LP(DATA_WIDTH-1 downto 1),
                      DATA_WIDTH+1);
elsif(lp='0') then
   sum_signal := resize(amp_signal,DATA_WIDTH+1);
else
   sum_signal := resize(amp_signal,DATA_WIDTH+1)+
                 resize((DATA_WIDTH downto DATA_WIDTH-1 =>
                      filtered_data_LP(DATA_WIDTH-1))&
                      filtered_data_LP(DATA_WIDTH-1 downto 1),
```

```
                        DATA_WIDTH+1);
end if;
m_axis_tdata <= std_logic_vector(sum_signal(DATA_WIDTH-1 downto 0));
```

Listing 5.1: Audio equalizer model

It should be noted that `o_data`, the output of the filter mapped in `filtered_data_*` signals, has been redefined as a signed vector and that `hp` and `lp` are two `std_logic` controlled by two FPGA switches. In addition, `amp_signal`, `abs_signal`, `sum_signal` are properly defined variables.

6 CONCLUSION

In summary, we successfully implemented a general architecture for a FIR filter in VHDL and we managed to use it for a channel selection operation of an audio signal thanks to the AXI4-Stream protocol and the Pmod I2S2 board. Then, we compared the FPGA-based resulting signal with the expectation signal computed in Python and showed how the filter works correctly, but we have not been able to characterise the phase introduced by the hardware implementation of the filter. Lastly, we proposed an extension of this project with a model of an audio equalizer.