UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI FISICA E ASTRONOMIA “GALILEO GALILEI”
MASTER DEGREE IN PHYSICS OF DATA
COURSE: MANAGEMENT AND ANALYSIS OF PHYSICS DATASETS, MODULE A

Implementation of a FIR filter to process signals from a PMOD audio in VHDL

GITHUB REPO: HTTPS://GITHUB.COM/AIDINATTAR/PMOD-FIR-FILTER-VHDL

Agosti Luca[1], Attar Aidin[1], Baci Ema[1], and Coppi Alberto[1]

[1]Dipartimento di Fisica e Astronomia "Galileo Galilei", Università degli Studi di Padova

February 20, 2022

Abstract

The goal of this lab project is the implementation of a simple FIR-filter on a FPGA and its employment in an audio system obtained using a Pmod I2S2. Our task is to write down the VHDL source code that will be synthetized and implemented in the hardware.
The top entity implemented is shown in Fig. 1 and in Fig. 2. The filter block communicates with both the PC and the Pmod. Input data, digitized by the Pmod, are retrieved through a I2S receiver, filtered, and then sent back through a I2S transmitter; filtered data are also sent through a UART transmitter to the PC.

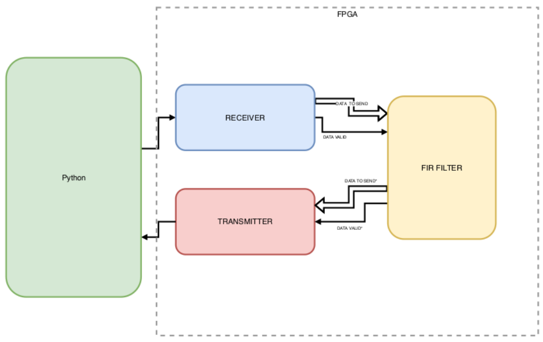


Figure 1: Scheme of communication between the Python script and the FIR filter block through the UART protocol.

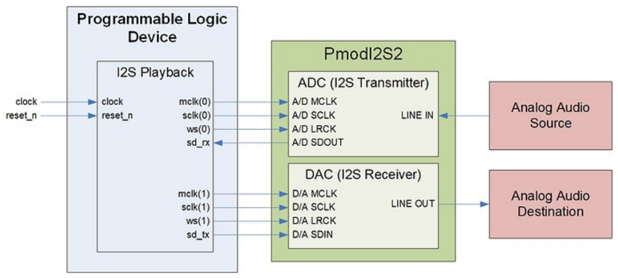


Figure 2: Scheme of communication between the the FIR filter block and the Pmod through the I2S protocol.

Keywords
VHDL, FPGA, FIR filter, PMOD

1 Introduction

1.1 PMOD I2S2

Inter-IC Sound (I^2S), is an electrical serial bus interface standard used for connecting digital audio devices together. The I2S bus separates clock and serial data signals, resulting in simpler receivers than those required for asynchronous communications systems that need to recover the clock from the data stream. A picture of I2S Pmod is shown in Figure 3. This module is equipped with an ADC, which digitizes the analog signal and sends it to the I2S transmitter, and a DAC, which takes the digital signals coming from the I2S receiver and converts them to analog signals.

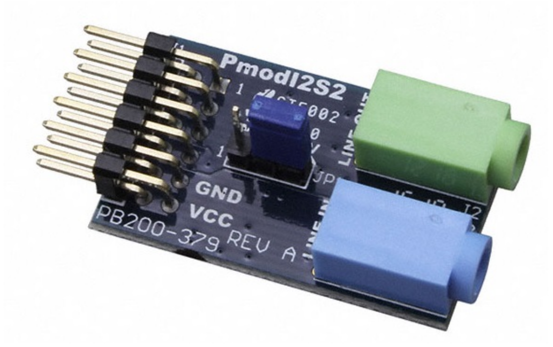


Figure 3: Digilent I2S Pmod

In our project input data, *i.e.* an audio stream, coming from the Pmod ADC are split into left and right channels and sent to two FIR filters, one for each channel. Then, each filter performs a different type of filtering, *i.e.* low-pass or high-pass, and finally filtered data are sent back to the Pmod DAC, thus obtaining a filtered analog audio stream with audible differences between the two channels.

1.2 FIR filter

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

$$y[n] = b_0 x[n] + b_1 x[n-1] + \dots + b_{k-1} x[n-k+1] = \sum_{i=0}^{k-1} b_i \cdot x[n-i] \tag{1}$$

The coefficients b_i represent the impulse response at the i^{th} instant and characterize the behaviour of the filter. The typical structure of the filter is schematized in Fig. 4.

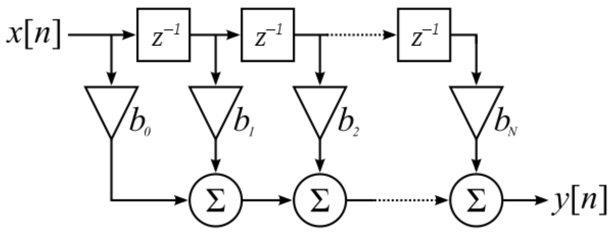


Figure 4: N^{th} order FIR filter.

We chose to implement two 15^{th} order FIR filter with cutoff frequency of 1000 Hz. In order to compute the corresponding filter coefficients, we use the `firwin` method of the `scipy.signal` library. It takes in input the number of coefficients, the cutoff frequency, the sampling frequency of the signal and the filter type.

```python
b = signal.firwin(numtaps=15, cutoff=1000, fs=44102,
    pass_zero='lowpass')
b = signal.firwin(numtaps=15, cutoff=1000, fs=44102,
    pass_zero='highpass')
```

The obtained coefficients for the low-pass filter are:

b_0	b_1	b_2	b_3	b_4
0.009	0.015	0.031	0.056	0.084
b_5	b_6	b_7	b_8	b_9
0.110	0.128	0.135	0.128	0.110
b_{10}	b_{11}	b_{12}	b_{13}	b_{14}
0.084	0.056	0.031	0.015	0.009

Table 1: Values of low-pass coefficients

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

Instead, for the high-pass filter:

b_0	b_1	b_2	b_3	b_4
-0.003	-0.005	-0.011	-0.018	-0.028
b_5	b_6	b_7	b_8	b_9
-0.037	-0.043	0.95	-0.043	-0.037
b_{10}	b_{11}	b_{12}	b_{13}	b_{14}
-0.028	-0.018	-0.011	-0.005	-0.003

Table 2: Values of high-pass coefficients

We note that in this case the coefficients are not normalized to 1.

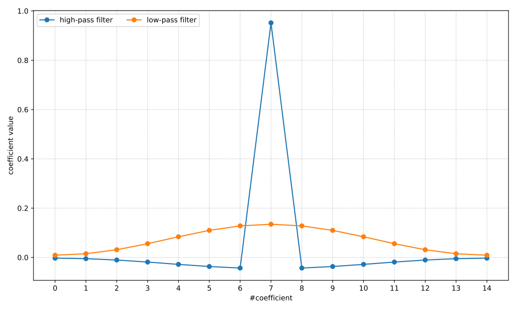


Figure 5: Coefficients of the two filters.

Then, using the `freqz` function of the `scipy.signal` package, we calculated the response in the frequency domain. Results with amplitude and phase as functions of the frequency are reported in Fig. 6 and Fig. 8.

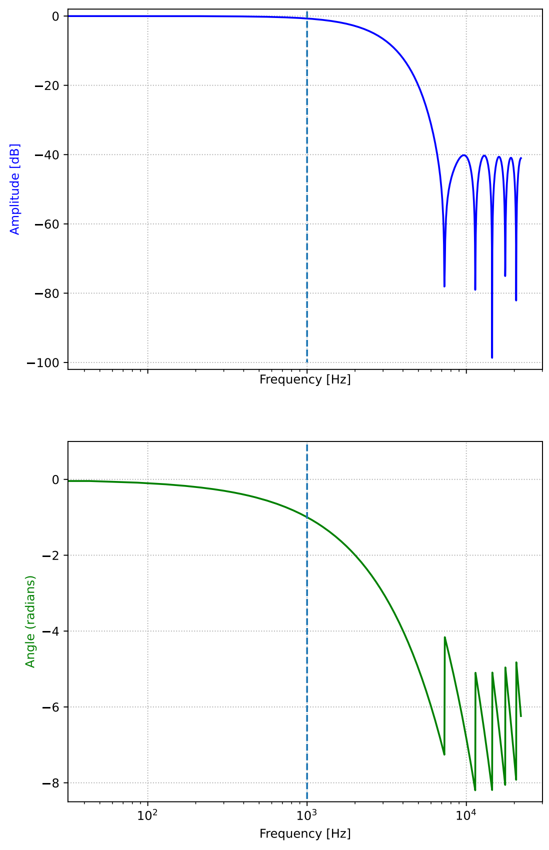


Figure 6: Frequency response of a low pass FIR filter.

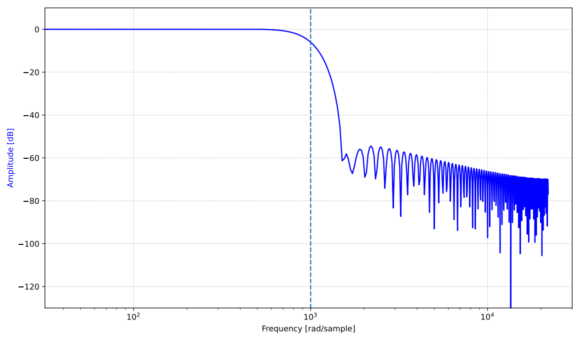


Figure 7: Frequency response of a low pass FIR filter with 150 taps.

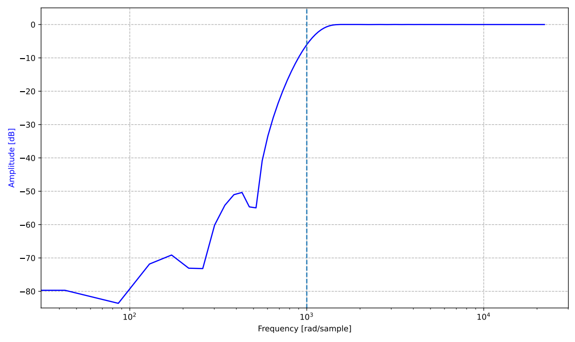


Figure 9: Frequency response of a high pass FIR filter with 151 taps.

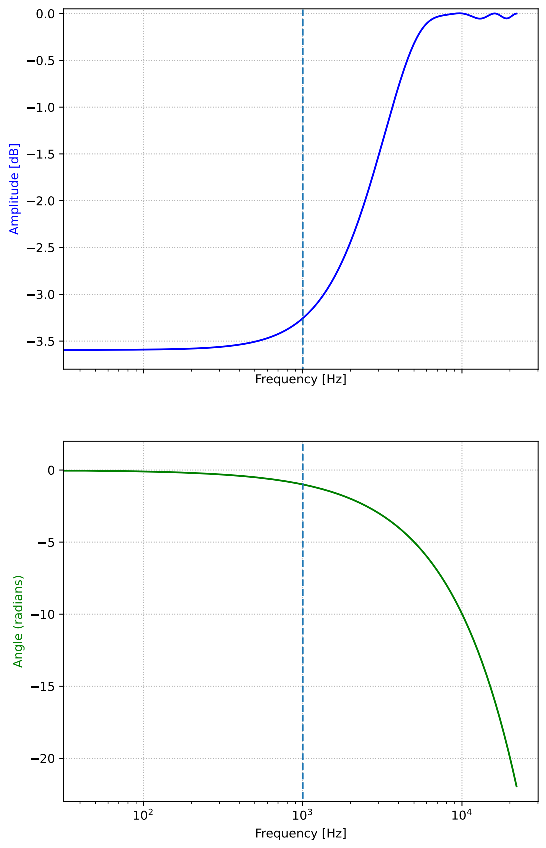


Figure 8: Frequency response of a high pass FIR filter.

Looking at the gain trend, one can recognize the typical low-pass (6) and high-pass (8) behaviours. Note that using only 15 taps the bandwidth seems to be wider than the one expected knowing the cutoff frequency. Adding more and more taps the filter behaviour starts to resemble the ideal behaviour, as we can see in Figure 7 and in Figure 9 obtained setting the number of taps respectively 150 and 151 (for the high-pass) we cannot have even number of taps). As expected the phase response is linear given the nature of the FIR filter (note the logarithmic scale on the frequency axis). This means that each frequency is shifted in time such that the result is an overall delay of the signal. From the theory we know that the delay is linked to the number of taps by the relation:

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

1.3 UART

To make the filter able to communicate with a serial port we connected it to a Full Universal Asynchronous Receiver-Transmitter (UART) machine: this is composed by a receiver module and a transmitter module whose composition is depicted in Fig. 10.

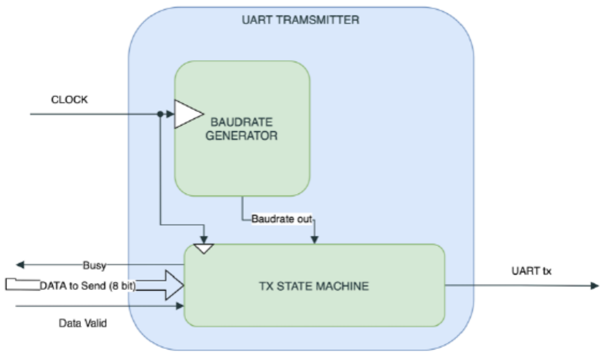


Figure 10: Scheme of UART transmitter

Originally data are transmitted with a baudrate of 115200 symbols per second, one bit per time, which is the 8-bit representation of an integer number in the range [0, 255]. In our project we use only the transmitter in order to send filtered data to the PC for further analysis. Moreover we modified the frequency of the baudrate in order to avoid data loss. The I2S transceiver provides data for left and right channels with a frequency of $f_d = 44.1\,\text{kHz}$. We chose to send 8 bits for each channel (we take the 8 MSB for each 24-bit data). Thus, as we spend 3 more baudrate clock cycles in order to change state of the UART, we need the frequency of the baudratee to be $f_b > 2 \cdot 11 \cdot f_d = 970.2\,\text{kHz}$. So, given in input a clock of $f_c = 11.29\,\text{MHz}$ the baudrate divisor must be $d = \frac{f_c}{f_b} \leq 11$.

2 Data format and precision

The width of the data that can be received and transmitted by the Pmod is 24 bits. However the arithmetical operations performed by the FIR filter lead to an increase in the data size.

Indeed, when considering two binary numbers A1, A2 represented by N1, N2 bits:

- $A1 + A2$ is represented by $\max\{N1, N2\} + 1$ bits,
- $A1 \cdot A2$ is represented by $N1 + N2$ bits.

Since we are processing our 24-bits input data with a 15 taps FIR-filter, the output will be 37 bits long, considering sums of couples for each step. As we need to send it back to the Pmod, we recast it into a 24-bits representation.

In order to simplify arithmetic operations in the FPGA, only integer numbers are used. Since the FIR-filter coefficients are clearly float, we need to scale them up in order to represent them as integers.

Considering that the coefficients values are always less than 1, to retrieve 8-bits signed integers, we scale-up the coefficients of a factor $Q = 2^7$. This will allow us to consider the 7 Least Significant Bits of the output number as its fractional part. For the coefficient, the values actually used by the VHDL code, are obtained through an approximation leading eventually to get a number that may differ of $0.5 \cdot 2^{-7}$ from the expected one.

On the other hand, since the UART protocol can deal with only 8-bits values in order to send data to the PC we need to do a further recast, taking only the first 8 Most Significant Bits (MSB) and eventually multiply the results by the constant value of 2^{16}.

3 VHDL Implementation

Here the VHDL code used for the project is shown.

3.1 FIR filter

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

entity fir_filter is
generic(
    d_width : integer;
    shift   : integer;
    coeff1  : integer;
    coeff2  : integer;
    coeff3  : integer;
    coeff4  : integer;
    coeff5  : integer;
    coeff6  : integer;
    coeff7  : integer;
    coeff8  : integer;
    coeff9  : integer;
    coeff10 : integer;
    coeff11 : integer;
    coeff12 : integer;
    coeff13 : integer;
    coeff14 : integer;
    coeff15 : integer);
port(
    clk        : in  std_logic;
    rst        : in  std_logic;
    i_data     : in  std_logic_vector( d_width-1 downto 0);
    o_data     : out std_logic_vector( d_width-1 downto 0));
end fir_filter;
```

The input signals of the FIR-filter entity are the clock, the reset, the input data, the width of the data in bits, the shift (to make the coefficients integers) and the coefficients. We decided to define the coefficients in the generic map in order to allow different filters for the two channels.

```vhdl
architecture rtl of fir_filter is

type t_data_pipe  is array (0 to 14) of signed(d_width-1 downto 0);
type t_coeff      is array (0 to 14) of signed(7 downto 0);
type t_mult       is array (0 to 14) of signed(8+d_width-1 downto 0);

signal r_coeff: t_coeff := (to_signed(coeff1 , 8), to_signed(coeff2 , 8), to_signed(coeff3 , 8), to_signed(coeff4 , 8), to_signed(coeff5 , 8), to_signed(coeff6 , 8), to_signed(coeff7 , 8), to_signed(coeff8 , 8), to_signed(coeff9 , 8), to_signed(coeff10, 8), to_signed(coeff11, 8), to_signed(coeff12, 8), to_signed(coeff13, 8), to_signed(coeff14, 8), to_signed(coeff15, 8));
signal p_data    : t_data_pipe;
signal r_mult    : t_mult;
signal r_add_st1 : signed(13+d_width downto 0);
```

Here the needed signals and custom types are defined, r_{coeff} is an array containing the coefficients for the FIR filter, converted to signed, since for the high-pass FIR filter we have both positive and negative coefficients. The other signals are used to store the results of sums and multiplications.

The code is then implemented using several processes in which sums and multiplications are performed:

```vhdl
p_input : process (rst,clk)
begin
  if(rst='1') then
    p_data       <= (others=>(others=>'0'));
    --r_coeff      <= (others=>(others=>'0'));
  elsif(rising_edge(clk)) then
    p_data       <= signed(i_data)&p_data(0 to p_data'length-2);
  end if;
end process p_input;

p_mult : process (rst,clk)
begin
  if(rst='1') then
    r_mult       <= (others=>(others=>'0'));
  elsif(rising_edge(clk)) then
    for k in 0 to p_data'length-1 loop
      r_mult(k)        <= p_data(k) * r_coeff(k);
    end loop;
  end if;
end process p_mult;

p_add_st1 : process (rst,clk)
begin
  if(rst='1') then
    r_add_st1      <= (others=>'0');
  elsif(rising_edge(clk)) then
    r_add_st1 <= resize(r_mult(0),r_add_st1'length) + resize(r_mult(1),r_add_st1'length) + resize(r_mult(2),r_add_st1'length) + resize(r_mult(3),r_add_st1'length) + resize(r_mult(4),r_add_st1'length) + resize(r_mult(5),r_add_st1'length) + resize(r_mult(6),r_add_st1'length) + resize(r_mult(7),r_add_st1'length) + resize(r_mult(8),r_add_st1'length) + resize(r_mult(9),r_add_st1'length) + resize(r_mult(10),r_add_st1'length) + resize(r_mult(11),r_add_st1'length) + resize(r_mult(12),
    r_add_st1'length) + resize(r_mult(13),r_add_st1'length)
     + resize(r_mult(14),r_add_st1'length);
  end if;
end process p_add_st1;

p_output : process (rst,clk)
begin
  if(rst='1') then
    o_data     <= (others=>'0');
  elsif(rising_edge(clk)) then
    o_data     <= std_logic_vector(r_add_st1(shift+23
     downto shift));
  end if;
end process p_output;
end rtl;
```

3.2 I2S tranceiver

The code used was taken from [1] and will not be reported in details, but just described regarding the key points of it.

The code uses three clocks: the master clock, the serial clock and the word select. The first is used to derive the others and to give rhythm to the operations performed by the I2S. The second, which is 4 times slower then the first one, is used to transmit and receive data on the I2S serial bus. Finally the word select is 64 times slower than the serial clock and is used to select the channel whose data we want to send or receive and to give rhythm to data, such that we are capable to transmit and receive data from each channel at this frequency. The factor 64 is due to the need of transmitting 48 bits of data on the serial bus (24 for each channel).

f_m	f_s	f_{ws}
11.29 MHz	2822.5 KHz	44.102 KHz

Table 3: Frequency of each clock

Beside this, what the module does is taking data digitalized through the Pmod send it to the FPGA and, after having filtered them through the FIR filter module, send them again in output to the PMOD which will convert again in an analogic signal.

3.3 Top level module

```
ENTITY top IS
    GENERIC(
        shift    : INTEGER := 7;
        d_width  : INTEGER := 24);
    PORT(
        CLK100MHZ     :  IN  STD_LOGIC;
        reset_n       :  IN  STD_LOGIC;
        rst_l         :  IN  STD_LOGIC;
        rst_r         :  IN  STD_LOGIC;
        rst_uart      :  IN  STD_LOGIC;
        mclk          :  OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        sclk          :  OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        ws            :  OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        sd_rx         :  IN  STD_LOGIC;
        sd_tx         :  OUT STD_LOGIC;
        uart_rxd_out  :  OUT STD_LOGIC);
END top;
```

The input signals of the top module are the system clock, the reset for the I2S module, UART transmitter and top level state machine, two resets for each FIR filter, one for the uart, and the serial data transmitted by the I2S Pmod. The outputs are the three clocks (useful for integration of this project into another one) and the serial data (to be transmitted to the Pmod) of the I2S module, and the output serial data to be sent to the UART transmitter.

```
ARCHITECTURE rtl OF top IS
    SIGNAL master_clk       :  STD_LOGIC;
    SIGNAL serial_clk       :  STD_LOGIC := '0';
    SIGNAL word_select      :  STD_LOGIC := '0';
    SIGNAL n_word_select    :  STD_LOGIC := '0';
    SIGNAL l_data_rx        :  STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0);
    SIGNAL r_data_rx        :  STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0);
    SIGNAL l_data_tx        :  STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0);
    SIGNAL r_data_tx        :  STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0);

    SIGNAL data_tx_uart     :  STD_LOGIC_VECTOR(7 DOWNTO 0);

    SIGNAL uart_tx          :  STD_LOGIC;
    SIGNAL data_valid       :  STD_LOGIC := '1';
    SIGNAL busy             :  STD_LOGIC;

    TYPE state_t IS (idle_l, left,  prebusy_l, checkbusy_l,
                     idle_r, right, prebusy_r, checkbusy_r)
     ;
    SIGNAL state : state_t := idle_l;
```

Here the needed signals and custom types are defined. We defined the state signal to use it in the state machine we will explain later in the report.

```
COMPONENT clk_wiz_0 IS
    PORT(
        clk_in1     :  IN STD_LOGIC  := '0';
        clk_out1    :  OUT STD_LOGIC);
END COMPONENT;

COMPONENT i2s_transceiver IS
    GENERIC(
        mclk_sclk_ratio :  INTEGER := 4;
        sclk_ws_ratio   :  INTEGER := 64;
        d_width         :  INTEGER := 24);
    PORT(
        reset_n     :  IN   STD_LOGIC;
        mclk        :  IN   STD_LOGIC;
        sclk        :  OUT  STD_LOGIC;
        ws          :  OUT  STD_LOGIC;
        sd_tx       :  OUT  STD_LOGIC;
        sd_rx       :  IN   STD_LOGIC;
        l_data_tx   :  IN   STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0);
        r_data_tx   :  IN   STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0);
        l_data_rx   :  OUT  STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0);
        r_data_rx   :  OUT  STD_LOGIC_VECTOR(d_width-1
     DOWNTO 0));
END COMPONENT;

COMPONENT fir_filter IS
    GENERIC(
        d_width   : integer := 24;
        shift     : integer :=  7;
        coeff1    : integer;
        coeff2    : integer;
        coeff3    : integer;
        coeff4    : integer;
        coeff5    : integer;
        coeff6    : integer;
        coeff7    : integer;
        coeff8    : integer;
        coeff9    : integer;
        coeff10   : integer;
        coeff11   : integer;
        coeff12   : integer;
        coeff13   : integer;
        coeff14   : integer;
        coeff15   : integer);
    PORT(
        clk        : in  std_logic;
        rst        : in  std_logic;
        i_data     : in  std_logic_vector( d_width-1 downto
      0);
        o_data     : out std_logic_vector( d_width-1 downto
      0));
END COMPONENT fir_filter;

COMPONENT uart_transmitter IS
   PORT(
      clock        : in  std_logic;
      data_to_send : in  std_logic_vector(7 downto 0);
      data_valid   : in  std_logic;
      busy         : out std_logic;
      uart_tx      : out std_logic);
END COMPONENT uart_transmitter;
```

Here the components are defined: we used `clk_wiz_0` which is a buffer for the system clock, `i2s_tranceiver` which inputs and outputs the serial data from each of the two channels, `fir_filter` which filters the data according to the coefficients passed and `uart_transmitter` which sends the data to the PC.

3.3.1 State Machine

Since we get data from two channels but the UART can be only one, due to the limitations in physical ports, we need to send all the data through a single serial port. Then, what we do is using a state machine as following to alternate data from the two channels.

```
p_uart_transmitter : process (master_clk, busy)
begin
    if rst_uart = '1' then
        data_valid <= '0';
        state <= idle_l;
    elsif rising_edge(master_clk) then
        case state is
            when idle_l =>
                if ws(0) = '1' then
                    state <= left;
                    data_valid <= '1';
                end if;
            when left =>
                data_tx_uart <= std_logic_vector(l_data_tx(23 downto 16));
                state <= prebusy_l;
            when prebusy_l =>
                state <= checkbusy_l;
            when checkbusy_l =>
                if busy = '0' then
                    state <= idle_r;
                end if;
                data_valid <= '0';
            when idle_r =>
                if ws(0) = '0' then
                    state <= right;
                    data_valid <= '1';
                end if;
            when right =>
                data_tx_uart <= std_logic_vector(r_data_tx(23 downto 16));
                state <= prebusy_r;
            when prebusy_r =>
                state <= checkbusy_r;
            when checkbusy_r =>
                if busy = '0' then
                    state <= idle_l;
                end if;
                data_valid <= '0';
        end case;
    end if;
end process p_uart_transmitter;
```

Basically what the state machine does is the following:

- if the reset button is pressed `data_valid` is set to zero in order to make UART not sending anything, to allow starting always with the left channel in data sending and in this way removing the ambiguity of the channel sent first;
- at each rising of the `master_clk` it begins a succession of states: since the value of `ws(0)` corresponds to the channel we are retrieving, we check we are in the left channel and so we give to the variable `data_tx_uart` such that at the next clock the data sent will be the one of the left channel;
- the next states are `prebusy` and `checkbusy` such that we wait one clock and then until the data is completely sent, making UART not busy anymore;
- the same operations are repeated for the right channel.

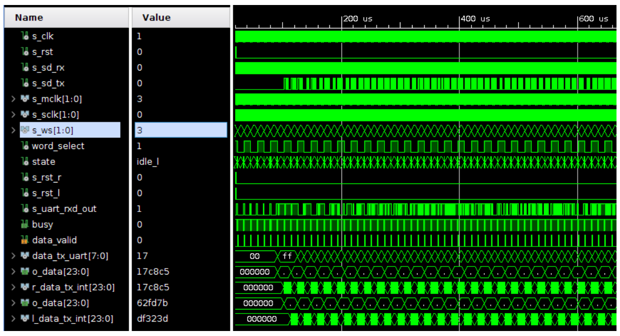
4 Vivado simulation

Before testing our code on the FPGA, we build a testbench of the top structure that reads an input signal from file and gives it back after the filtering thanks to the transmitter.

Here we present the principal processes involved:

```
uut : top port map( CLK100MHZ => s_clk, reset_n => s_rst,
     rst_r => s_rst_r, rst_l => s_rst_l, sd_rx => s_sd_rx,
     sd_tx => s_sd_tx, mclk => s_mclk, sclk => s_sclk, ws =>
      s_ws, uart_rxd_out => s_uart_rxd_out );

p_clk : process is
begin
    s_clk <= '1'; wait for 5 ns;
    s_clk <= '0'; wait for 5 ns;
end process p_clk;

p_rst : process is
begin
    s_rst <= '1'; wait for 10 ns;
    s_rst <= '0'; wait;
end process p_rst;

p_rst_l : process is
begin
    s_rst_l <= '1'; wait for 10 ns;
    s_rst_l <= '0'; wait;
end process p_rst_l;

p_rst_r : process is
begin
    s_rst_r <= '1'; wait for 10 ns;
    s_rst_r <= '0'; wait;
end process p_rst_r;

p_sd_rx : process is
variable v_LINE : line;
variable i_data : integer;
begin
    file_open(file_VECTORS, "/home/aidin/CrucialSSD/
     University/Management_and_Analysis_of_Physics_Datasets/
     Laboratory/input.txt", read_mode);
    wait until rising_edge(s_clk);
    data : while not endfile(file_VECTORS) loop
        readline(FILE_VECTORS, v_LINE);
        read(v_LINE, i_data);
        if i_data = 1 then
            s_sd_rx <= '1'; wait for 5 ns;
        else
            s_sd_rx <= '0'; wait for 5 ns;
        end if;
    end loop data;
end process p_sd_rx;
```

The outcome of the simulation is reported in Figure 11.



Name	Value
s_clk	1
s_rst	0
s_sd_rx	0
s_sd_tx	0
s_mclk[1:0]	3
s_sclk[1:0]	0
s_ws[1:0]	3
word_select	1
state	idle_l
s_rst_r	0
s_rst_l	0
s_uart_rxd_out	1
busy	0
data_valid	0
data_tx_uart[7:0]	17
o_data[23:0]	17c8c5
r_data_tx_int[23:0]	17c8c5
o_data[23:0]	62fd7b
l_data_tx_int[23:0]	df323d

Figure 11: Simulation of the whole configuration.

5 Results

In order to quantify the results obtained we tested the configuration using severe audio tracks and downloaded the data sent by the UART module trough a python script:

```
with serial.Serial('/dev/ttyUSB1', baudrate=1026364) as ser
    :
    d = ser.read(12000000) # 3 minutes

out_l = np.array(list(d[0:len(d):2]))
out_r = np.array(list(d[1:len(d):2]))
for i in range(len(out_l)):
    if out_l[i] > 127:
        out_l[i] = out_l[i] - 256
for i in range(len(out_r)):
    if out_r[i] > 127:
        out_r[i] = out_r[i] - 256
```

Since the UART module send the data alternating right and left channel we need to separate them, and then adjust them considering that we are dealing with signed data.

5.1 Range 20-15000 Hz

We first tried to input an audio track of frequencies going from 20Hz to 15KHz in a time interval of $\sim$ 2 minutes, using this video.

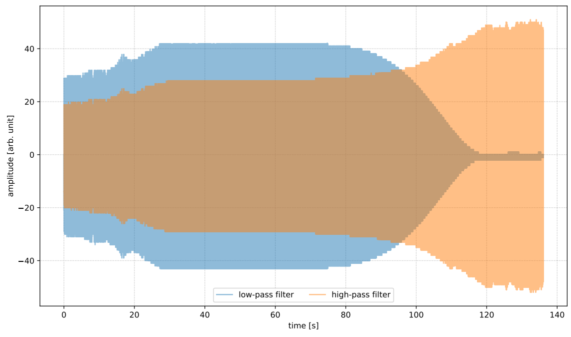


Figure 12: Wave form received from the FIR filter through UART protocol.

We immediately see the difference between the low-pass filter and the high-pass filter. Indeed while the first starts from a greater amplitude falling at high frequencies, the latter starts from a smaller amplitude increasing at high frequencies.

We can see better this behaviour in the chart in Figure 12: we plotted the response in frequency registered by the two filters, obtained with a Fast Fourier Transform (FFT) using the function `magnitude_spectrum` of the library `matplolib.pyplot`.

In the FFT plot one can see that the low-pass has a greater magnitude than the high-pass filter for low frequencies, while this behaviour is inverted for $f \gtrsim 2000$ Hz as expected.

5.2 Single Frequency

We then tried to input two single frequency audio track, one of 200Hz and one of 10KHz.

We can see that, as expected, the wave is reconstructed better for the low-pass filter, but both the low-pass and the high-pass

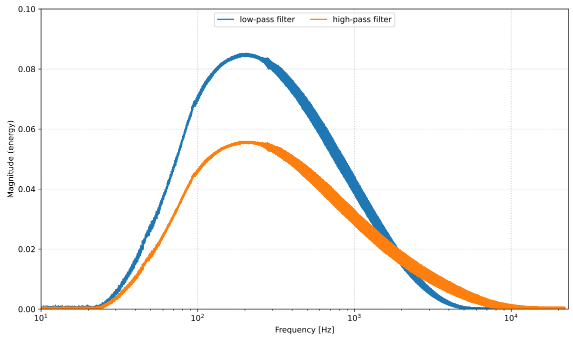


Figure 13: Response in frequency registered by the two filters.

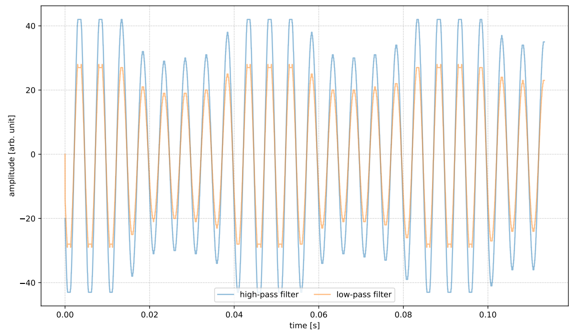


Figure 14: Wave form received from the FIR filter through UART protocol using a frequency of 200 Hz.

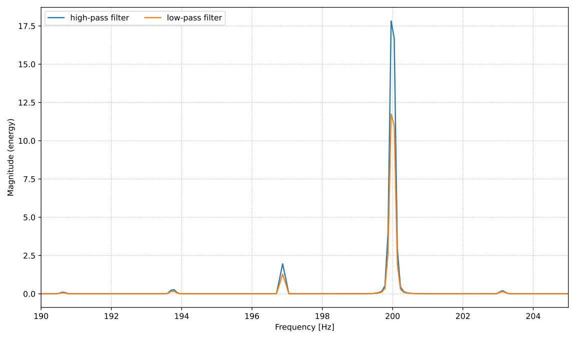


Figure 15: Response in frequency registered by the two filters using a frequency of 200 Hz.

filters reconstruct the wave at 200 Hz. There are some different frequencies for both the filters, such as $f = 197$ Hz which can be due to interference or not absolute cleanness of the audio track used.

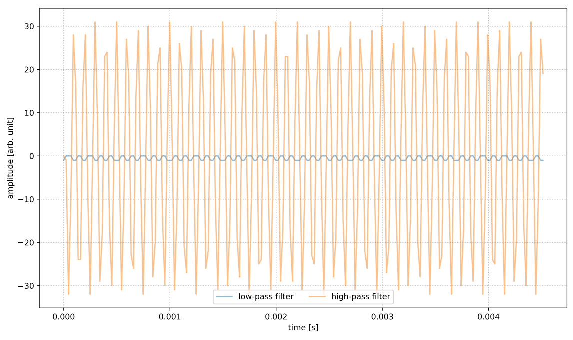


Figure 16: Wave form received from the FIR filter through UART protocol using a frequency of 10 KHz.

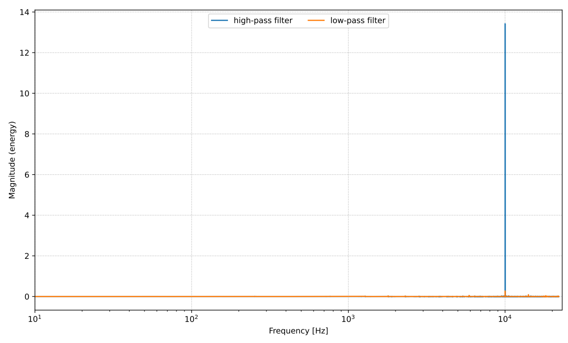


Figure 17: Response in frequency registered by the two filters using a frequency of 10 KHz.

For the 10KHz the difference is much more evident both in time domain and in frequency domain. This is due to the fact that both filters have a cutoff frequency of 1000Hz which is quite a middle frequency and so using a low frequency of 200Hz is not enough to see a evident difference. What it should be done is using a more 'extreme' cutoff frequency for both of the filters.

5.3 Severe frequencies together

The last thing we tried is an analysis of severe single frequencies all together.

We can see in Figures 18 and 19 that for low frequencies the high-pass filter tries to knock down the signals while for high frequencies the low-pass filter does that.

Conclusions

We implemented a FIR filter to process signals coming from a PMOD audio on FPGA and, before testing the code on the hardware, we simulated its behaviour with the Vivado tool. We validated our results through the Fourier spectral analysis of

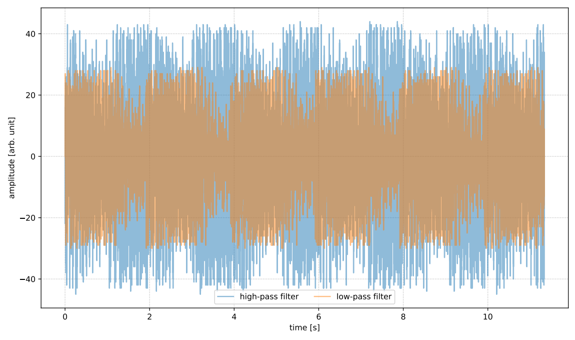


Figure 18: Wave form received from the FIR filter through UART protocol.

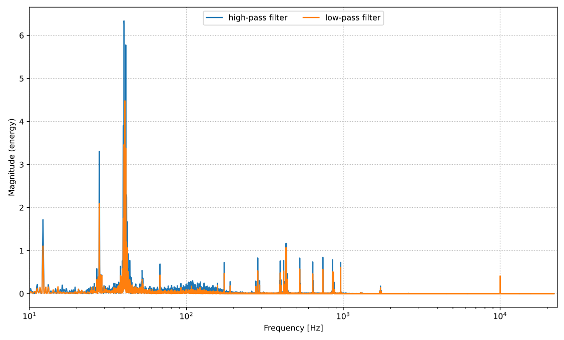


Figure 19: Response in frequency registered by the two filters.

the output signals. The results are comparable with our expectations.

Eventual future developments of the project will be the implementation of the UART receiver to take the coefficients of the filters as input through a python script.

References

[1] DigiKey-Electronics. I2s pmod quick start (vhdl). https://forum.digikey.com/t/i2s-pmod-quick-start-vhdl/13065. Accessed: 2022-02-17.