

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 Luca¹, Attar Aidin¹, Baci Ema¹, and Coppi Alberto¹

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

February 21, 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 synthesized and implemented in the hardware.

The top entity implemented is shown in Fig. 1. 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.

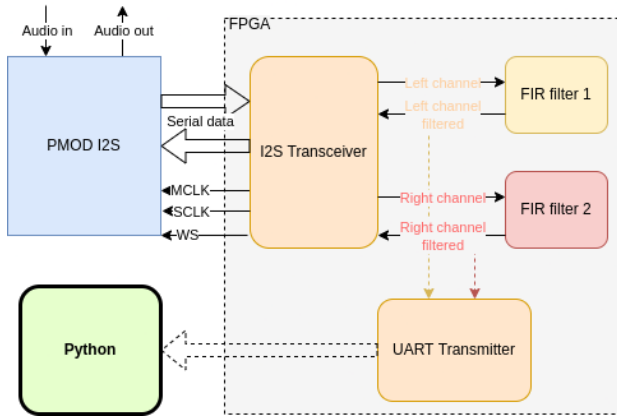


Figure 1: Scheme of communication between all the components of this project.

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 2. 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.

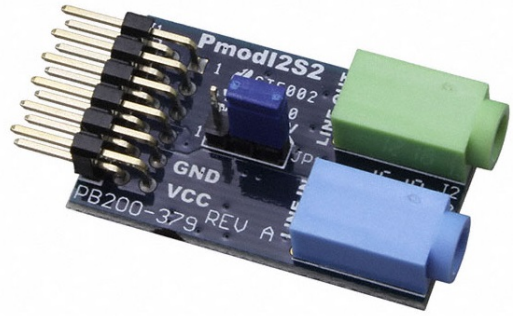


Figure 2: Digilent I2S PMOD

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 characterized by an impulse response 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:

$$y[n] = b_0x[n] + b_1x[n-1] + \dots + b_{k-1}x[n-k+1] = \sum_{i=0}^{k-1} b_i \cdot x[n-i] \quad (1)$$

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. 3.

We chose to implement two 15th order FIR filter, one high-pass and the other low-pass, both with cutoff frequency of 1 kHz. 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.

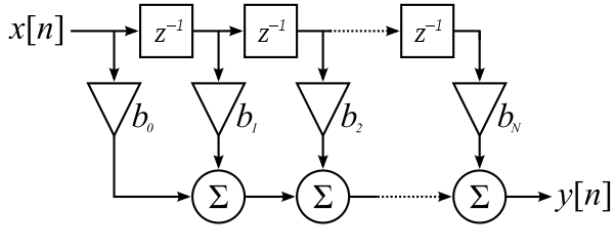
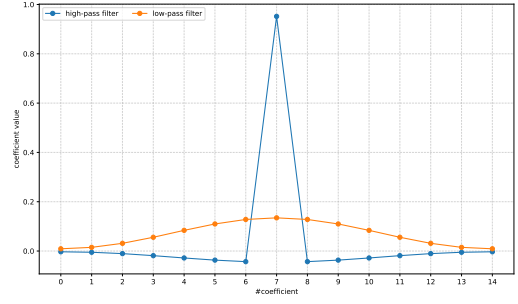
Figure 3: N^{th} order FIR filter.

Figure 4: Coefficients of the two filters.

```
1 b = signal.firwin(numtaps=15, cutoff=1000, fs=44102,
2   pass_zero='lowpass')
3 b = signal.firwin(numtaps=15, cutoff=1000, fs=44102,
4   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, but still symmetric.

The values of the coefficients are also represented in Figure 4, one can see how, while for the low-pass FIR filter we do a sort of weighted rolling average of the 15 points window which kills high frequencies, the high-pass focuses mostly on the central value allowing the selection of high frequencies only.

Then, using the `freqz` function of the `scipy.signal` package, we calculated the response in frequency domain. Results with amplitude and phase as functions of the frequency are reported in Fig. 5 and Fig. 7.

Looking at the gain trend, one can recognize the typical low-pass (5) and high-pass (7) 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 6 and in Figure 8 obtained setting the number of taps respectively 150 (for the low-pass) and 151 (for the high-pass, in this case we cannot have even number of taps).

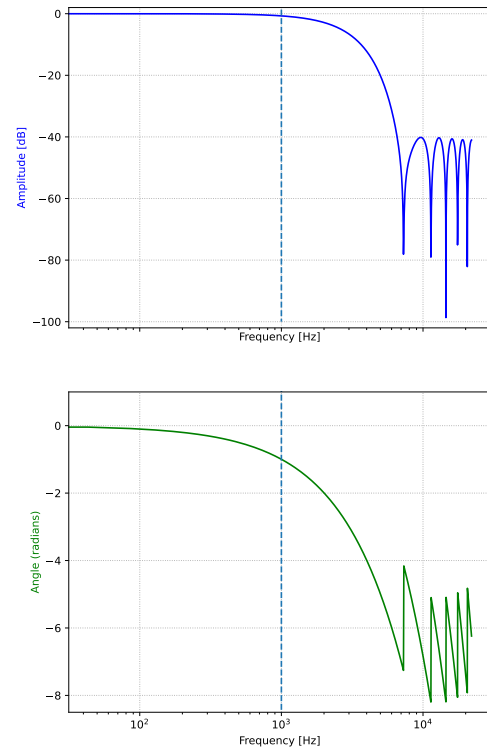


Figure 5: Frequency response of a low pass FIR filter.

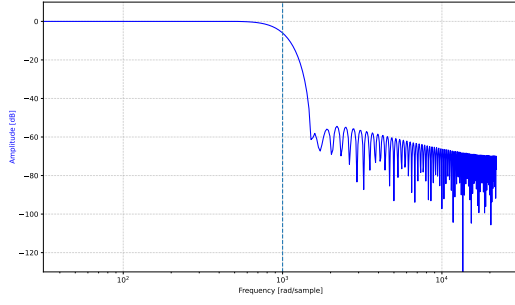


Figure 6: Frequency response of a low pass FIR filter with 150 taps.

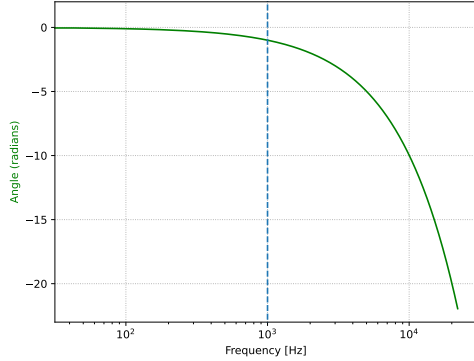
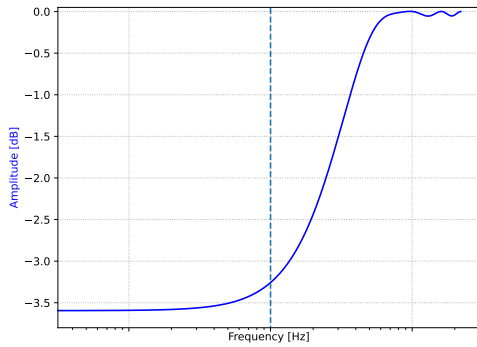


Figure 7: Frequency response of a high pass FIR filter.

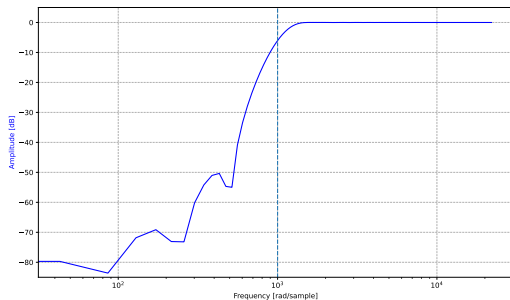


Figure 8: Frequency response of a high pass FIR filter with 151 taps.

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 overall delay is linked to the number of taps by the relation:

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

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. 9.

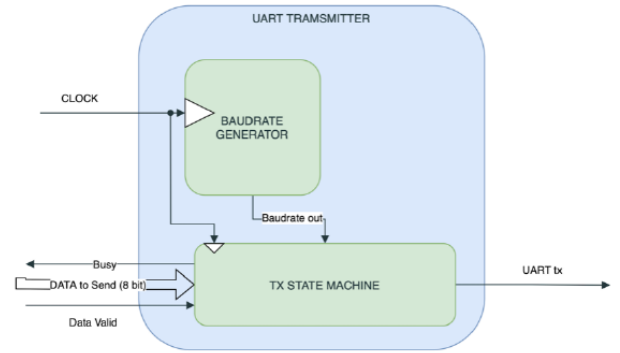


Figure 9: Scheme of UART transmitter

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 $f_d = 44.1$ 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 baudrate to be $f_b > 2 \cdot 11 \cdot f_d = 970.2$ kHz. So, given in input a clock of $f_c = 11.29$ 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 perform calculations with integer only numbers taking care of truncating the 7 Least Significant Bits of the output number, being its fractional part. 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

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity fir_filter is
6 generic(
7     d_width : integer;
8     shift   : integer;
9     coeff1  : integer;
10    coeff2  : integer;
11    coeff3  : integer;
12    coeff4  : integer;
13    coeff5  : integer;
14    coeff6  : integer;
15    coeff7  : integer;
16    coeff8  : integer;
17    coeff9  : integer;
18    coeff10 : integer;
19    coeff11 : integer;
20    coeff12 : integer;
21    coeff13 : integer;
22    coeff14 : integer;
23    coeff15 : integer);
24 port (
25     clk      : in  std_logic;
26     rst      : in  std_logic;
27     i_data   : in  std_logic_vector( d_width-1 downto 0);
28     o_data   : out std_logic_vector( d_width-1 downto 0);
29 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.

```
1 architecture rtl of fir_filter is
2
3 type t_data_pipe is array (0 to 14) of signed(d_width-1
4     downto 0);
5 type t_coeff is array (0 to 14) of signed(7
6     downto 0);
7 type t_mult is array (0 to 14) of signed(8+d_width-1
8     downto 0);
```

```
7 signal r_coeff: t_coeff := (to_signed(coeff1 , 8),
8     to_signed(coeff2 , 8), to_signed(coeff3 , 8), to_signed
9     (coeff4 , 8), to_signed(coeff5 , 8), to_signed(coeff6 ,
10    8), to_signed(coeff7 , 8), to_signed(coeff8 , 8),
11    to_signed(coeff9 , 8), to_signed(coeff10 , 8), to_signed
12    (coeff11 , 8), to_signed(coeff12 , 8), to_signed(coeff13 ,
13    8), to_signed(coeff14 , 8), to_signed(coeff15 , 8));
14 signal p_data : t_data_pipe;
15 signal r_mult : t_mult;
16 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 and data coming from the PMOD are represented as signed. The other signals are used to store the results of sums and multiplications.

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

```
1 p_input : process (rst,clk)
2 begin
3     if(rst='1') then
4         p_data <= (others=>(others=>'0'));
5     elsif(rising_edge(clk)) then
6         p_data <= signed(i_data)&p_data(0 to p_data'length-2);
7     end if;
8 end process p_input;
9
10 p_mult : process (rst,clk)
11 begin
12     if(rst='1') then
13         r_mult <= (others=>(others=>'0'));
14     elsif(rising_edge(clk)) then
15         for k in 0 to p_data'length-1 loop
16             r_mult(k) <= p_data(k) * r_coeff(k);
17         end loop;
18     end if;
19 end process p_mult;
20
21 p_add_st1 : process (rst,clk)
22 begin
23     if(rst='1') then
24         r_add_st1 <= (others=>'0');
25     elsif(rising_edge(clk)) then
26         r_add_st1 <= resize(r_mult(0),r_add_st1'length) +
27             resize(r_mult(1),r_add_st1'length) + resize(r_mult(2),
28                 r_add_st1'length) + resize(r_mult(3),r_add_st1'length)
29             + resize(r_mult(4),r_add_st1'length) + resize(r_mult(5),
30                 r_add_st1'length) + resize(r_mult(6),r_add_st1'length)
31             + resize(r_mult(7),r_add_st1'length) + resize(r_mult
32                 (8),r_add_st1'length) + resize(r_mult(9),r_add_st1'
33                 length) + resize(r_mult(10),r_add_st1'length) + resize(
34                 r_mult(11),r_add_st1'length) + resize(r_mult(12),
35                 r_add_st1'length) + resize(r_mult(13),r_add_st1'length)
36             + resize(r_mult(14),r_add_st1'length);
37     end if;
38 end process p_add_st1;
39
40 p_output : process (rst,clk)
41 begin
42     if(rst='1') then
43         o_data <= (others=>'0');
44     elsif(rising_edge(clk)) then
45         o_data <= std_logic_vector(r_add_st1(shift+23
46             downto shift));
47     end if;
48 end process p_output;
49 end rtl;
```

We used several processes in order to perform multiple operations at each clock, such as products between the coefficients and the input data, sums between the weighted signals and the final output.

3.2 I2S trceiver

Regarding the I2S module, the code used was taken from [1] and will not be reported in details, but just the key points of it will be described.

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 than 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, the module takes data digitized through the PMOD, sends them to the FPGA and, after the FIR filter module has filtered them, sends them back to the PMOD, which will convert them again to analogical signal.

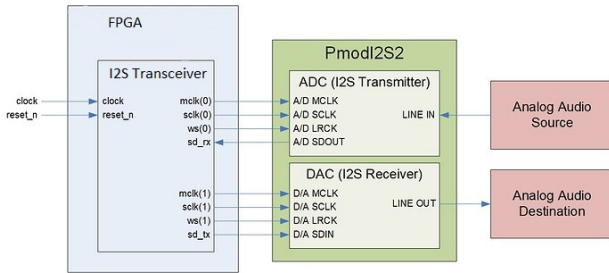


Figure 10: Scheme of communication between the FPGA and the PMOD through the I2S protocol.

3.3 Top level module

```
1 ENTITY top IS
2   GENERIC (
3     shift : INTEGER := 7;
4     d_width : INTEGER := 24);
5   PORT (
6     CLK100MHZ : IN STD_LOGIC;
7     reset_n : IN STD_LOGIC;
8     rst_l : IN STD_LOGIC;
9     rst_r : IN STD_LOGIC;
10    rst_uart : IN STD_LOGIC;
11    mclk : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
12    sclk : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
13    ws : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
14    sd_rx : IN STD_LOGIC;
15    sd_tx : OUT STD_LOGIC;
16    uart_rxd_out : OUT STD_LOGIC);
17 END top;
```

The input signals of the top module are the system clock, the reset for I2S module, 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 UART transmitter output serial data to be sent to the PC.

```
1 ARCHITECTURE rtl OF top IS
2   SIGNAL master_clk : STD_LOGIC;
3   SIGNAL serial_clk : STD_LOGIC := '0';
4   SIGNAL word_select : STD_LOGIC := '0';
5   SIGNAL n_word_select : STD_LOGIC := '0';
```

```
6   SIGNAL l_data_rx : STD_LOGIC_VECTOR(d_width-1
7     DOWNTO 0);
8   SIGNAL r_data_rx : STD_LOGIC_VECTOR(d_width-1
9     DOWNTO 0);
10  SIGNAL l_data_tx : STD_LOGIC_VECTOR(d_width-1
11    DOWNTO 0);
12  SIGNAL r_data_tx : STD_LOGIC_VECTOR(d_width-1
13    DOWNTO 0);
14  SIGNAL data_tx_uart : STD_LOGIC_VECTOR(7 DOWNTO 0);
15  SIGNAL uart_tx : STD_LOGIC;
16  SIGNAL data_valid : STD_LOGIC := '1';
17  SIGNAL busy : STD_LOGIC;
18  TYPE state_t IS (idle_l, left, prebusy_l, checkbusy_l,
19    idle_r, right, prebusy_r, checkbusy_r);
20  SIGNAL state : state_t := idle_l;
```

Here the needed signals and custom types are defined. We defined the state signal which will be used in the state machine explained in Section 3.3.1.

```
1 COMPONENT clk_wiz_0 IS
2   PORT (
3     clk_in1 : IN STD_LOGIC := '0';
4     clk_out1 : OUT STD_LOGIC);
5 END COMPONENT;
6
7 COMPONENT i2s_transceiver IS
8   GENERIC (
9     mclk_sclk_ratio : INTEGER := 4;
10    sclk_ws_ratio : INTEGER := 64;
11    d_width : INTEGER := 24);
12   PORT (
13     reset_n : IN STD_LOGIC;
14     mclk : IN STD_LOGIC;
15     sclk : OUT STD_LOGIC;
16     ws : OUT STD_LOGIC;
17     sd_tx : OUT STD_LOGIC;
18     sd_rx : IN STD_LOGIC;
19     l_data_tx : IN STD_LOGIC_VECTOR(d_width-1
20       DOWNTO 0);
21     r_data_tx : IN STD_LOGIC_VECTOR(d_width-1
22       DOWNTO 0);
23     l_data_rx : OUT STD_LOGIC_VECTOR(d_width-1
24       DOWNTO 0);
25     r_data_rx : OUT STD_LOGIC_VECTOR(d_width-1
26       DOWNTO 0);
27 END COMPONENT;
28
29 COMPONENT fir_filter IS
30   GENERIC (
31     d_width : integer := 24;
32     shift : integer := 7;
33     coeff1 : integer;
34     coeff2 : integer;
35     coeff3 : integer;
36     coeff4 : integer;
37     coeff5 : integer;
38     coeff6 : integer;
39     coeff7 : integer;
40     coeff8 : integer;
41     coeff9 : integer;
42     coeff10 : integer;
43     coeff11 : integer;
44     coeff12 : integer;
45     coeff13 : integer;
46     coeff14 : integer;
47     coeff15 : integer);
48   PORT (
49     clk : in std_logic;
50     rst : in std_logic;
51     i_data : in std_logic_vector( d_width-1 downto
52       0);
53     o_data : out std_logic_vector( d_width-1 downto
54       0));
55 END COMPONENT fir_filter;
56
57 COMPONENT uart_transmitter IS
58   PORT (
59     clock : in std_logic;
60     data_to_send : in std_logic_vector(7 downto 0);
61     data_valid : in std_logic;
62     busy : out std_logic;
63     uart_tx : out std_logic);
64 END COMPONENT uart_transmitter;
```


Here the components are defined: we used `clk_wiz_0` which is a buffer for the system clock, giving a clock with $f = 11.29\text{MHz}$, `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 only be one, due to the limitations in physical ports, we need to send all the data through a single serial port. Thus, we use a state machine, described in the following, to alternate data from the two channels.

```
1 p_uart_transmitter : process (master_clk, busy)
2 begin
3     if rst_uart = '1' then
4         data_valid <= '0';
5         state <= idle_l;
6     elsif rising_edge(master_clk) then
7         case state is
8             when idle_l =>
9                 if ws(0) = '1' then
10                     state <= left;
11                     data_valid <= '1';
12                 end if;
13             when left =>
14                 data_tx_uart <= std_logic_vector(l_data_tx
15                     (23 downto 16));
16                 state <= prebusy_l;
17             when prebusy_l =>
18                 state <= checkbusy_l;
19             when checkbusy_l =>
20                 if busy = '0' then
21                     state <= idle_r;
22                 end if;
23                 data_valid <= '0';
24             when idle_r =>
25                 if ws(0) = '0' then
26                     state <= right;
27                     data_valid <= '1';
28                 end if;
29             when right =>
30                 data_tx_uart <= std_logic_vector(r_data_tx
31                     (23 downto 16));
32                 state <= prebusy_r;
33             when prebusy_r =>
34                 state <= checkbusy_r;
35             when checkbusy_r =>
36                 if busy = '0' then
37                     state <= idle_l;
38                 end if;
39                 data_valid <= '0';
40             end case;
41         end if;
42     end process p_uart_transmitter;
```

Basically, the state machine works as follows:

- if the reset button is pressed `data_valid` is set to zero in order to make UART not sending anything. This ensures that the UART data sending starts always with the left channel. In this way we removed the ambiguity of the channel whose data are sent first when we start listening on the UART bus;
- at each rising edge of the `master_clk` it begins a succession of states: since the value of `ws(0)` represents the channel whose data are being retrieved, we check if we are in the left channel and so we give to the variable `data_tx_uart` the value coming from the left channel, such that at the next clock the data sent will be the one of the left channel;
- the next states are `prebusy` and `checkbusy`. In `prebusy` we wait one clock for timing reasons and then

in `checkbusy` we wait until the data is completely sent, so that we continue to the following states only if the UART is free to transmit another data;

- the same operations are repeated for the right channel.

4 Vivado simulation

Before testing our code on the FPGA, we built 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:

```
1 uut : top port map (CLK100MHZ => s_clk, reset_n => s_rst,
2   rst_r => s_rst_r, rst_l => s_rst_l, sd_rx => s_sd_rx,
3   sd_tx => s_sd_tx, mclk => s_mclk, sclk => s_sclk, ws =>
4   s_ws, uart_rxd_out => s_uart_rxd_out );
5
6 p_clk : process is
7 begin
8     s_clk <= '1'; wait for 5 ns;
9     s_clk <= '0'; wait for 5 ns;
10 end process p_clk;
11
12 p_rst : process is
13 begin
14     s_rst <= '1'; wait for 10 ns;
15     s_rst <= '0'; wait;
16 end process p_rst;
17
18 p_rst_l : process is
19 begin
20     s_rst_l <= '1'; wait for 10 ns;
21     s_rst_l <= '0'; wait;
22 end process p_rst_l;
23
24 p_rst_r : process is
25 begin
26     s_rst_r <= '1'; wait for 10 ns;
27     s_rst_r <= '0'; wait;
28 end process p_rst_r;
29
30 p_sd_rx : process is
31 variable v_LINE : line;
32 variable i_data : integer;
33 begin
34     file_open(file_VECTORS, "/home/aidin/CrucialSSD/
35         University/Management_and_Analysis_of_Physics_Datasets/
36         Laboratory/input.txt", read_mode);
37     wait until rising_edge(s_clk);
38     data : while not endfile(file_VECTORS) loop
39         readline(FILE_VECTORS, v_LINE);
40         read(v_LINE, i_data);
41         if i_data = 1 then
42             s_sd_rx <= '1'; wait for 5 ns;
43         else
44             s_sd_rx <= '0'; wait for 5 ns;
45         end if;
46     end loop data;
47 end process p_sd_rx;
```

We defined different processes, representing each input signal.

The outcome of the simulation is reported in Figure 11.

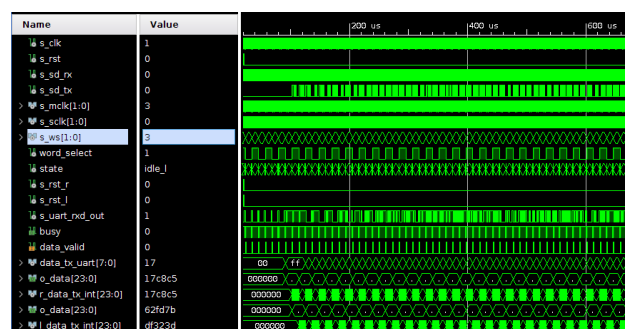


Figure 11: Simulation of the whole configuration.

Thanks to the simulation we were able to synchronize all the different components of the project, avoiding potential bugs.

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 through a python script:

```
1 with serial.Serial('/dev/ttyUSB1',baudrate=1026364) as ser:
2     d = ser.read(12000000)
3
4 out_l = np.array(list(d[0:len(d):2]))
5 out_r = np.array(list(d[1:len(d):2]))
6 for i in range(len(out_l)):
7     if out_l[i] > 127:
8         out_l[i] = out_l[i] - 256
9 for i in range(len(out_r)):
10    if out_r[i] > 127:
11        out_r[i] = out_r[i] - 256
```

Since the UART module sends the data alternating right and left channel we need to separate them, and then make an adjustment 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 ~ 2 minutes, using this [video](#).

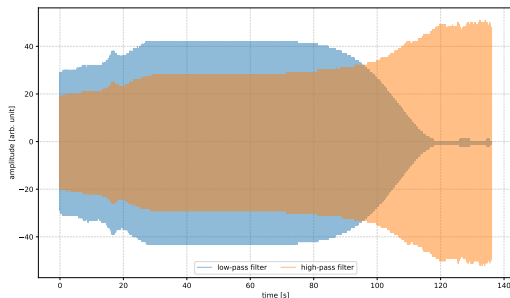


Figure 12: Wave form received from the FIR filter through UART protocol.

We immediately see in Figure 12 the difference between the low-pass 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 13: 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 `matplotlib.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 $f \gtrsim 2$ kHz as expected.

5.2 Single Frequency

Then we tried to input two single frequencies audio track, one of 200 Hz and one of 10 kHz.

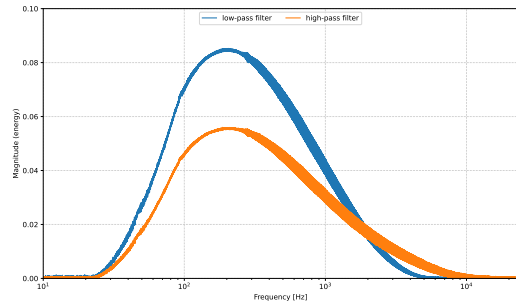


Figure 13: Response in frequency registered by the two filters.

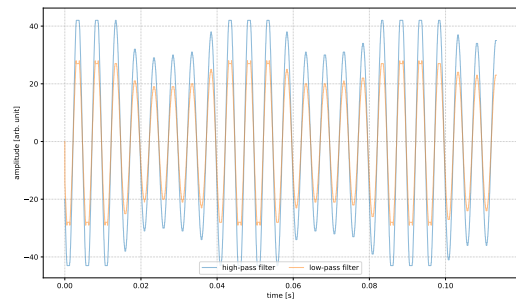


Figure 14: Wave form received from the FIR filter through UART protocol using a frequency of 200 Hz.

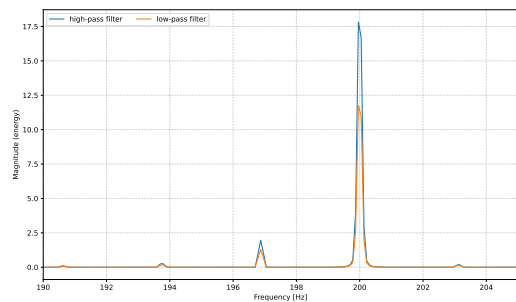


Figure 15: Response in frequency registered by the two filters using a frequency of 200 Hz.

We can see that, as expected, both the low-pass and the high-pass filters reconstruct the wave at 200 Hz, but it has higher magnitude for the low-pass filter. Note that other frequencies appear in both filters, such as $f = 197$ Hz. This phenomenon can be caused by interferences or not absolute cleanness of the audio track used.

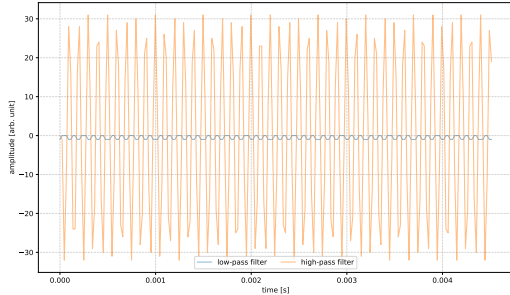


Figure 16: Wave form received from the FIR filter through UART protocol using a frequency of 10 KHz.

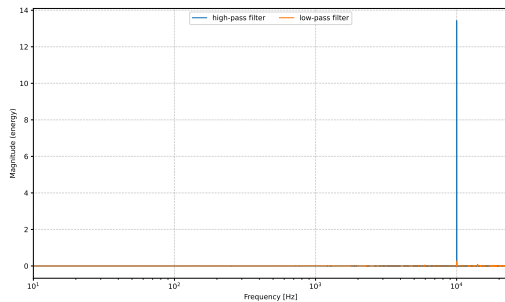


Figure 17: Response in frequency registered by the two filters using a frequency of 10 KHz.

For the 10 kHz the difference is much more evident both in time and frequency domain. This is due to the fact that both filters have a cutoff frequency of 1000 Hz which is quite a middle frequency and so using a low frequency of 200 Hz is not enough to see a evident difference. If we want a different behaviour we should use a more 'extreme' cutoff frequency for both 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 the output signals. The results are comparable with our expectations.

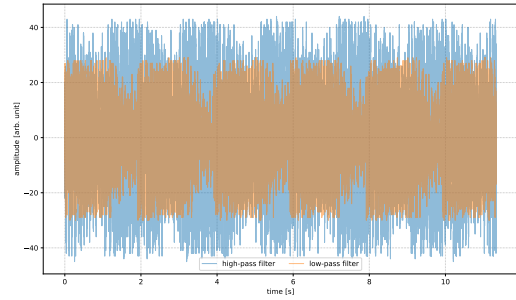


Figure 18: Wave form received from the FIR filter through UART protocol.

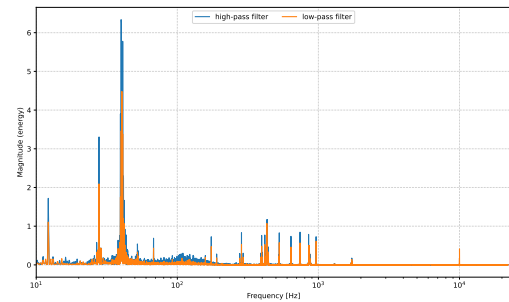


Figure 19: Response in frequency registered by the two filters.

Eventual future developments of the project would be the implementation of a UART receiver to take the coefficients of the filters as input through a python script. Furthermore we can implement a switch to decide if the UART should pass filtered or unfiltered data, or change the transmission frequency to allow passing not only the 8 MSB but the whole number of 24 bits, passing 3 series of 8 bits through the UART and rebuild the original number 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.