Management and Analysis of Physics Dataset (mod. A): Hardware accelerated FIR filter and application to an audio stream

Sebastiano Monti 2052399 | Gabriele Brotolato 2019062 | Mario Rossi 123456 | Mario Rossi 123456

Thursday 20th January, 2022

1 Aim

In this project we show an implementation of a FIR filter on an ARTY A7 FPGA, using two different architectures. For this purpose, a Digilent stereo audio Pmod I2S2 module has been used, together with I^{2}S protocol for communication with the FPGA board and an ADC/DAC ICs.

Produced modules have been tested using Python simulations, where we generated a wave form input, together with the expected output. In addition, hardware validation has been provided using real world audio samples, analyzed through an oscilloscope.

2 Implementation

The used modules are listed below and the block diagram shows the various connections.

- I^{2}S/AXIS interface;
- AXIS FIFO;
- FIR filter;
- AXIS volume controller;

Fig.1.

In the next sections, we describe in details the structure of the main components.

2.1 I^{2}S to AXIS interface

The Digilent Pmod I2S2 features an audio A/D converter and a stereo D/A converter, each connected to one of two audio jacks. These circuits allow the FPGA to transmit and receive stereo audio signals via the I^{2}S protocol. In particular, input signals are translated to an AXI-Stream (encoded into the I^{2}S protocol) with the last flag used as a selector between left and right channel. The Pmod I2S2 supports 24 bit resolution per channel at input sample rates up to 108 kHz and output sample rates up to 200 kHz.

More on this module can be found on the Digilent website [1].

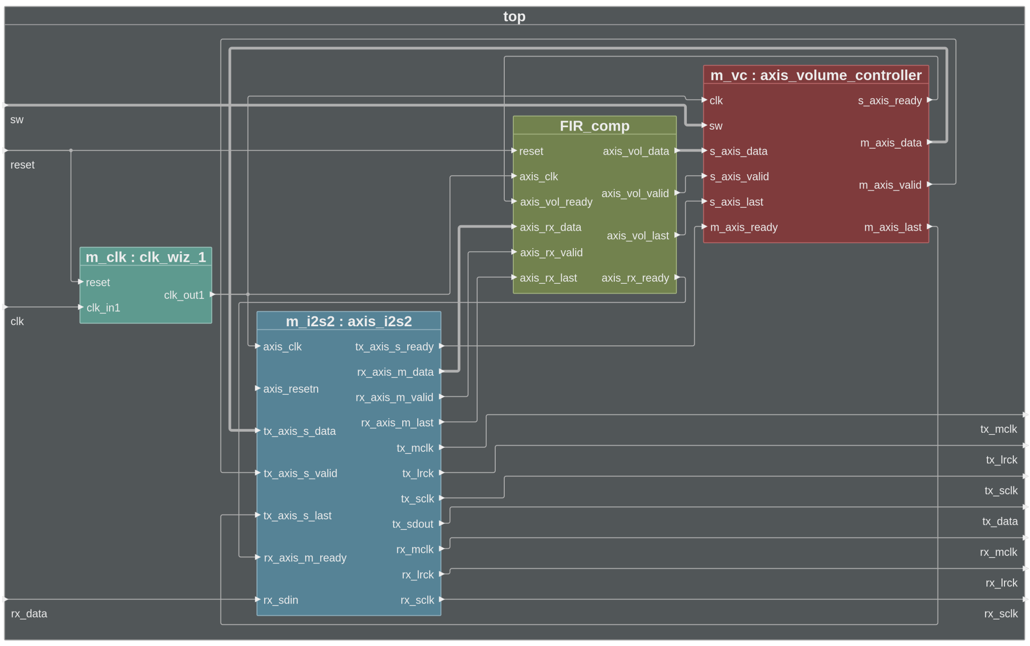

Figure 1: Diagram of top VHDL file.

2.2 FIR filter

We implement a finite impulse response (FIR) filter, which is a filter whose impulse response is of finite duration. This module is AXI-Stream compliant and input and output FIFO are added. We firstly provide a brief mathematical introduction. Given a sequence $\{x_i\}_{i=1,\ldots,N}$ of N input data samples, the output sequence of the filter is obtained by applying the following operation:

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

which is a convolution operation, or more simply, a weighted moving average. The b_i in Eq. 1 are the coefficients that characterize the filter and its order. So, a k-th order filter is a filter that works with k coefficients.

In our work, we consider a 7-th order FIR filter. The values of the coefficients are computed through an online calculator , by setting a cutoff frequency of 4.8 kHz and a sample rate of 48 kHz. The frequency analysis for this filter setup is showed in Figure 3.

The values of the coefficients are (16 bits signed integer format):

$$\begin{aligned} b_0 &= 1915 \\ b_1 &= 5389 \\ b_2 &= 8266 \\ b_3 &= 9979 \\ b_4 &= 8266 \\ b_5 &= 5389 \\ b_6 &= 1915 \end{aligned}$$

2.2.1 Latency architecture

This implementation is mainly focused to reduce the latency of the filter. To do so, a pipelined data flow has been employed. First of all, we implemented a shift register that stores the last seven inputs of the data stream, using a cascade of D flip-flops.

```vhdl
shift_reg_p : process (clk) is
    begin
        if rising_edge(clk) then
            if (s_new_word = '1') then
                if (s_select = 1) then  -- right audio data
                    audio_data_shift_r(0) <= signed(s_axis_tdata(
   AUDIO_DATA_WIDTH-1 downto 0));
                    audio_data_shift_r(1) <= audio_data_shift_r(0);
                    audio_data_shift_r(2) <= audio_data_shift_r(1);
                    audio_data_shift_r(3) <= audio_data_shift_r(2);
                    audio_data_shift_r(4) <= audio_data_shift_r(3);
                    audio_data_shift_r(5) <= audio_data_shift_r(4);
                    audio_data_shift_r(6) <= audio_data_shift_r(5);
                else    -- left audio data
                    audio_data_shift_l(0) <= signed(s_axis_tdata(
   AUDIO_DATA_WIDTH-1 downto 0));
                    audio_data_shift_l(1) <= audio_data_shift_l(0);
                    audio_data_shift_l(2) <= audio_data_shift_l(1);
                    audio_data_shift_l(3) <= audio_data_shift_l(2);
                    audio_data_shift_l(4) <= audio_data_shift_l(3);
                    audio_data_shift_l(5) <= audio_data_shift_l(4);
                    audio_data_shift_l(6) <= audio_data_shift_l(5);
                end if;
            end if;
        end if;

    end process shift_reg_p;
```

When the data are stored, the multiplication can take place. Here the whole 7 samples are processed, and finally, on the following clock cycle, the addition is performed.

```vhdl
process (clk) is
begin
   if rising_edge(clk) then
       if (s_new_packet_r(0) = '1') then  -- multiplication

           mult_reg_l(0) <= audio_data_shift_l(0) * to_signed(coeff(0), 16);
           mult_reg_l(1) <= audio_data_shift_l(1) * to_signed(coeff(1), 16);
           mult_reg_l(2) <= audio_data_shift_l(2) * to_signed(coeff(2), 16);
           mult_reg_l(3) <= audio_data_shift_l(3) * to_signed(coeff(3), 16);
           mult_reg_l(4) <= audio_data_shift_l(4) * to_signed(coeff(4), 16);
           mult_reg_l(5) <= audio_data_shift_l(5) * to_signed(coeff(5), 16);
           mult_reg_l(6) <= audio_data_shift_l(6) * to_signed(coeff(6), 16);

           mult_reg_r(0) <= audio_data_shift_r(0) * to_signed(coeff(0), 16);
           mult_reg_r(1) <= audio_data_shift_r(1) * to_signed(coeff(1), 16);
           mult_reg_r(2) <= audio_data_shift_r(2) * to_signed(coeff(2), 16);
           mult_reg_r(3) <= audio_data_shift_r(3) * to_signed(coeff(3), 16);
           mult_reg_r(4) <= audio_data_shift_r(4) * to_signed(coeff(4), 16);
           mult_reg_r(5) <= audio_data_shift_r(5) * to_signed(coeff(5), 16);
           mult_reg_r(6) <= audio_data_shift_r(6) * to_signed(coeff(6), 16);

        elsif (s_new_packet_r(1) = '1') then  -- addition
            data(0) <= mult_reg_l(0) + mult_reg_l(1) + mult_reg_l(2)
 + mult_reg_l(3) + mult_reg_l(4) + mult_reg_l(5) + mult_reg_l(6);
            data(1) <= mult_reg_r(0) + mult_reg_r(1) + mult_reg_r(2)
 + mult_reg_r(3) + mult_reg_r(4) + mult_reg_r(5) + mult_reg_r(6);
        end if;
    end if;
end process;
```

From the code above it is clear that in two clock cycles the data are processed, but the price in resources is very high. In this particular case, 14 DSP blocks are used (the ARTY A/ 35T has got 90 DSPs).

2.2.2 Multiplication and Accumulation (MAC) architecture

In this second architecture, the principle remains the same, but the accumulation is split in seven different steps. In our case, we used a FSM that multiplies and accumulates all the seven steps. In this way, only 2 DSPs are needed, at the cost of increased latency, in particular 7 clock cycles per data.

```
MAC_p : process (clk, rst) is
begin
    if (rst = '1') then
        state <= idle;
        s_axis_tready_r <= '0';
        m_axis_tvalid_r <= '0';
        res_l   <=  (others => '0') ;
        res_r   <=  (others => '0') ;
        sel     <= '0';
    elsif rising_edge(clk) then
        case state is
            when idle =>
                s_axis_tready_r <= '1';
                m_axis_tvalid_r <= '0';
                res_l   <=  (others => '0') ;
                res_r   <=  (others => '0') ;
                if (s_axis_tvalid = '1') then
                    sel <= s_axis_tlast;
                    state <= mult_0;
                end if;
            when mult_0 =>
                s_axis_tready_r <= '0';
                m_axis_tvalid_r <= '0';
                if (sel = '1') then
                    res_r <= res_r + (audio_data_shift_r(0) * to_signed(coeff(0)
    , 16));
                else
                    res_l <= res_l + (audio_data_shift_l(0) * to_signed(coeff(0)
    , 16));
                end if;
                state <= mult_1;
                .
                .
                .
            when mult_6 =>
                s_axis_tready_r <= '0';
                if (sel = '1') then
                    res_r <= res_r + (audio_data_shift_r(6) * to_signed(coeff(6)
    , 16));
                else
                    res_l <= res_l + (audio_data_shift_l(6) * to_signed(coeff(6)
    , 16));
                end if;
                if (m_axis_tready = '1') then
                    m_axis_tvalid_r <= '1';
                    state <= idle;
                else
                    m_axis_tvalid_r <= '0';
                    state <= send_data;
                end if;
                state <= idle;
            when send_data =>
```

```
                s_axis_tready_r <= '0';
                if (m_axis_tready = '1') then
                    m_axis_tvalid_r <= '1';
                    state <= idle;
                else
                    m_axis_tvalid_r <= '0';
                    state <= send_data;
                end if;
        end case;
    end if;
end process MAC_p;
```

3 Module validation

3.1 Testbench validation

At this point, the produced modules have been validated via simulation. In the testbench, the input array is read from a file and then sent to the FIR filter. Subsequently, the obtained results are firstly compared with a file containing the values calculated using Python and then written back to another file.

A snippet of the code is given below.

```
check_data_p : process (clk) is
    --------------------------------------------

    file test_vector                : text open write_mode is "output_file_fir.txt";
    variable row                    : line;

    --------------------------------------------
begin

    if(rising_edge(clk)) then
        if (m_axis_tvalid = '1' and m_axis_tlast = '0') then
            value1_fir_24_bit_out <= m_axis_tdata(23 downto 0);
            if (signed(m_axis_tdata(23 downto 0)) < signed(value1_down_out))then
                report "Left output does not match, expected " & integer'image(to_integer(signed(value1_std_logic_24_bit_out)))
                & " got " & integer'image(to_integer(signed(m_axis_tdata(23 downto 0)))) severity warning;
                err_cnt <= err_cnt + X"0001";
            elsif (signed(m_axis_tdata(23 downto 0)) > signed(value1_up_out)) then
                report "Left output does not match, expected " & integer'image(to_integer(signed(value1_std_logic_24_bit_out)))
                & " got " & integer'image(to_integer(signed(m_axis_tdata(23 downto 0)))) severity warning;
                err_cnt <= err_cnt + X"0001";
            end if;
        elsif (m_axis_tvalid = '1' and m_axis_tlast = '1') then
            write(row, to_integer(signed(value1_fir_24_bit_out))    , right, 15);
            write(row, to_integer(signed(m_axis_tdata(23 downto 0))), right, 15);
            writeline(test_vector,row);
            value2_fir_24_bit_out <= m_axis_tdata(23 downto 0);
            if (signed(m_axis_tdata(23 downto 0)) < signed(value2_down_out))then
                report "Right output does not match, expected " & integer'image(to_integer(signed(value2_std_logic_24_bit_out)))
                & " got " & integer'image(to_integer(signed(m_axis_tdata(23 downto 0)))) severity warning;
                err_cnt <= err_cnt + X"0001";
            elsif (signed(m_axis_tdata(23 downto 0)) > signed(value2_up_out)) then
                report "Right output does not match, expected " & integer'image(to_integer(signed(value2_std_logic_24_bit_out)))
                & " got " & integer'image(to_integer(signed(m_axis_tdata(23 downto 0)))) severity warning;
                err_cnt <= err_cnt + X"0001";
                end if;
        end if;
    end if;

end process;
```

Due to rounding methods, some output values of the VHDL simulation weren't coherent with the ones calculated using Python. For this reason a tolerance has been implemented (in this case a tolerance of 2 was selected).

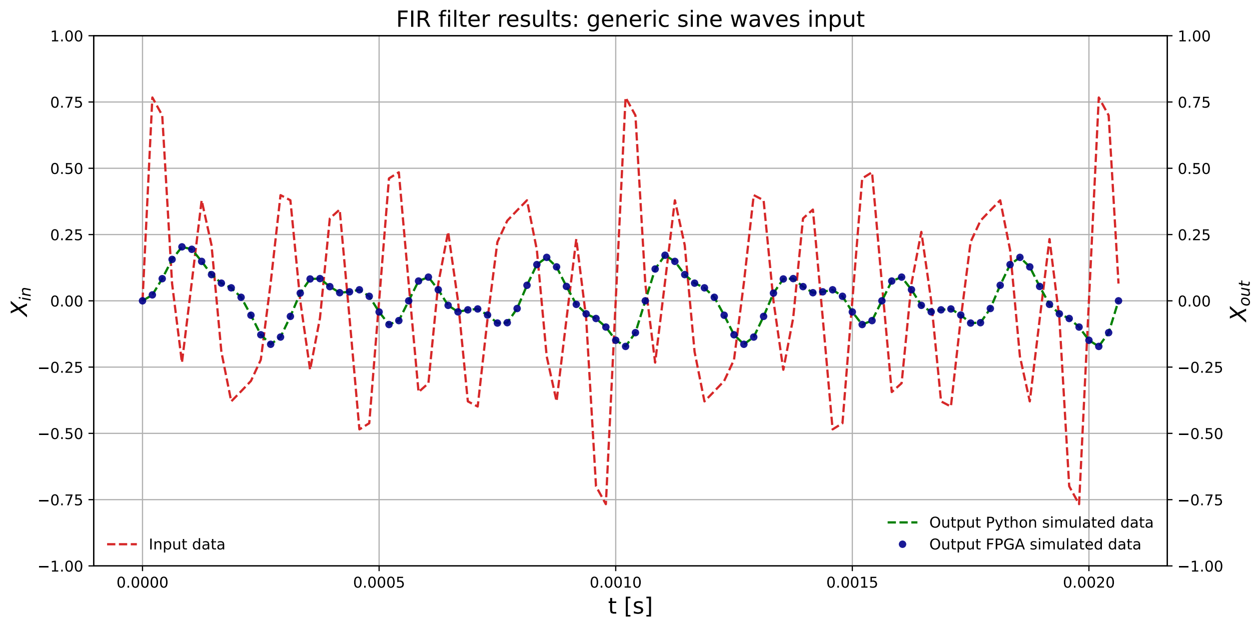

Figure 2: FIR filter response with a generic sine waves.

3.2 Real-world validation

The filter has been finally tested with real audio data samples. In particular, 31 different frequencies were sent to the FPGA and the output RMS values have been measured with an oscilloscope.

The output has been then rescaled computing the logarithm, in order to match the simulation data. The values in mV and dB are given in the table below.

From plot 4, it is clear that the filter responds as expected, but at higher frequencies the oscilloscope's resolution hides the real filter's behaviour.

4 Conclusion

In this assignment we presented two different architectures of a FIR filter, implemented in FPGA hardware. We also exploited I^{2}S protocol and modules provided by Digilent to exchange and sample the audio stream.

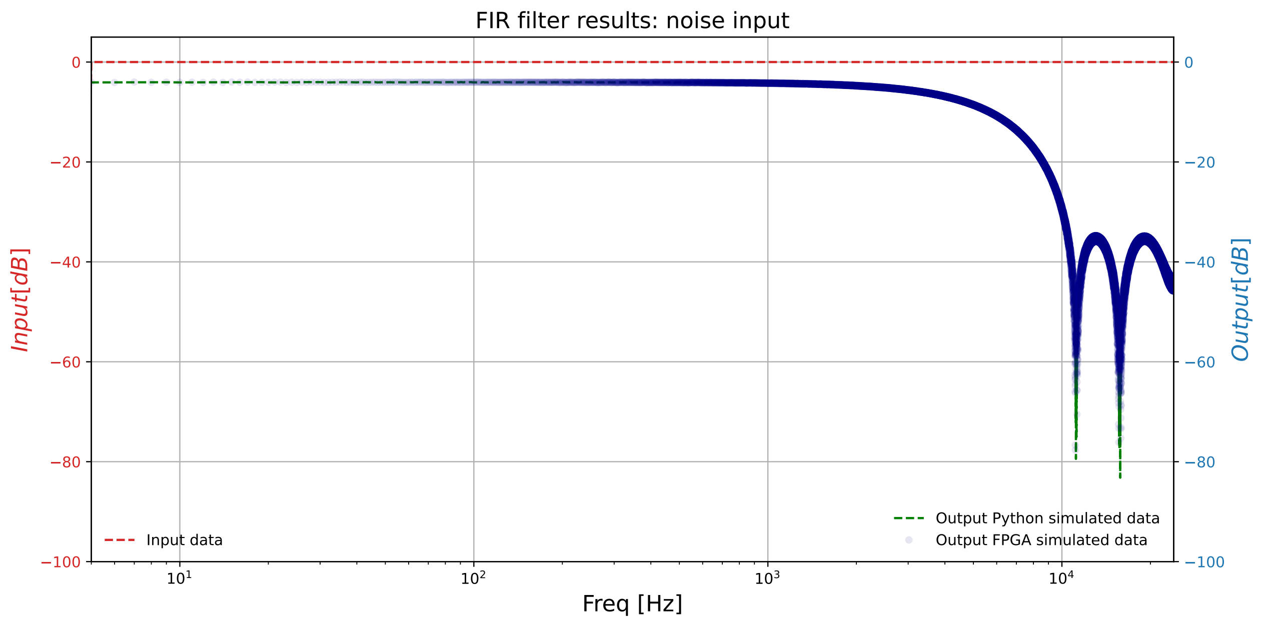

Figure 3: Frequency analysis of the FIR filter with the given configuration.

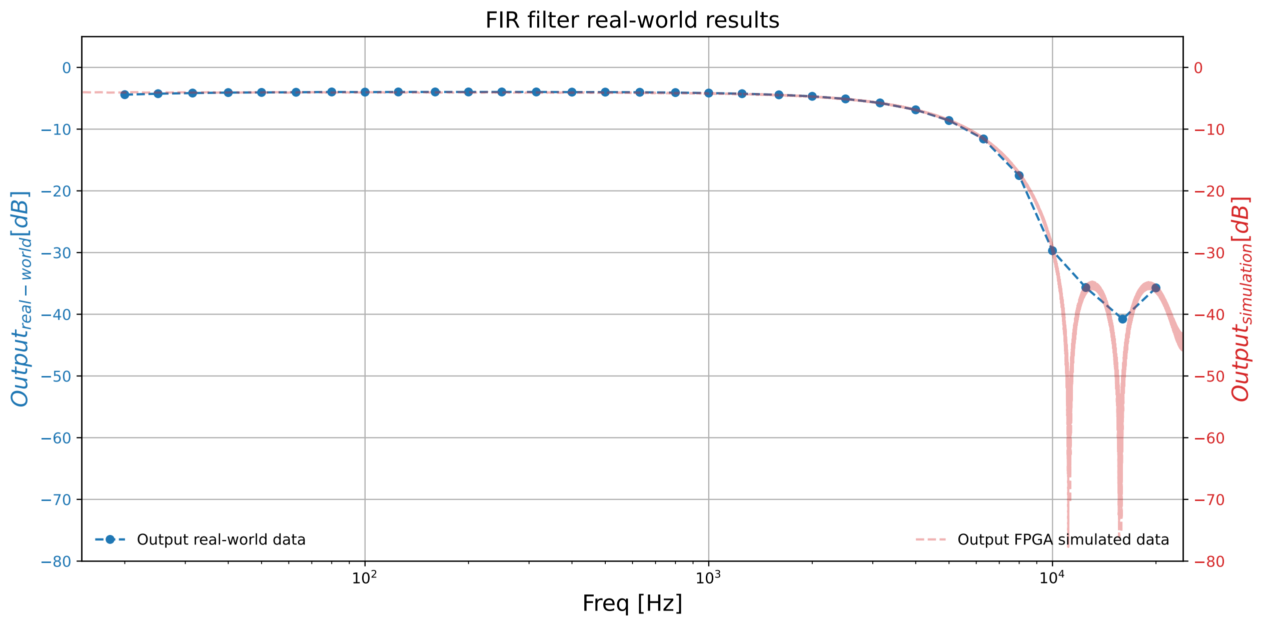

Figure 4: FIR filter response with different input sine wave frequencies

References

[1] Digilent website, `https://digilent.com/reference/pmod/pmodi2s2/start?redirect=1`

[2] Github repository, `https://github.com/Gabriele-bot/MAPD_LAB/tree/main/FIR_project`