Implement a FIR filter co-processor in FPGA

Andrea Nicolai, Michele Guadagnini, Sandeep Kumar Shekhar and Walter Zigliotto

Department of Physics and Astronomy
MANAGEMENT AND ANALYSIS OF PHYSICS DATASET (Mod. A)
Università degli Studi di Padova
via 8 Febbraio 1848, 2, 35122 Padova PD

1. Aim

The aim of this study is to implement a FIR filter co-processor in FPGA. In particular, we have to develop such filter both in Python and VHDL, in order to compare the evaluated results. For this purpose, the DPRAM component is used as a memory and the IPBUS protocol is employed to connect the *Arty A7* board with the PC.

2. Introduction

A Finite Impulse Response (FIR) filter is a particular type of digital filter with a impulse response of finite duration. Mathematically, it can be described as a weighted mean that follows this equation:

$$y[n] = b_0x[n] + b_1x[n-1] + ... + b_Nx[n-N] = \sum_{i=0}^{N} b_i * x[n-i] \qquad (1)$$

where, $y[n]$ is the filtered signal, x_i with $i = 1, .., N$ is the input sequence and N is the order of the filter and b_i are the coefficients of the designed filter.
In this study a fifth-order FIR filter with a cutoff frequency equal to 0.1 is implemented. The coefficients are evaluated through the **scipy.signal.firwin** function of the scipy libraries.

Moreover, to convert the coefficients found using the aforementioned function in the FPGA, we need firstly to multiply these values by 10^3 (since the

Coef. #	**scipy** (*dec*)	**FPGA** (*hex*)
1	0.19335315	000000C1
2	0.20330353	000000CB
3	0.20668665	000000CE
4	0.20330353	000000CB
5	0.19335315	000000C1

Table 1: FIR filter coefficients.

FPGA operations are executed with integer arithmetics) and secondly to convert them into *hex*. The obtained *hex* coefficients are reported in Table 1 and the frequency response of the filter is resumed in Figure 1.

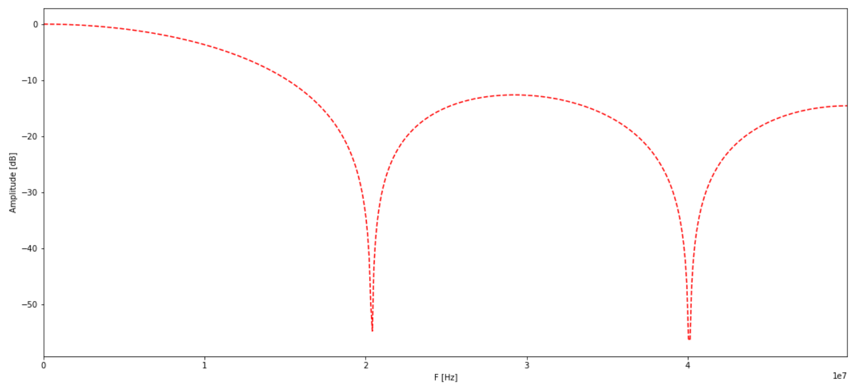

Figure 1: FIR filter frequency response.

3. Development

3.1. FIR filter implementation

The FIR implementation started with the definition of its entity that contains, as reported in Listing 1, the clock and reset signals, the filter coefficients, two logical arrays for input and output data, the write flag and the memory address.

The FIR architecture has been created as a *Finite State Machine* (see Listing 2) with three possible states:

```vhdl
entity fir_filter_5 is
  port (
    i_clk     : in  std_logic;
    i_rstb    : in  std_logic;
    -- coefficients
    i_coeff_0 : in  std_logic_vector(31 downto 0);
    i_coeff_1 : in  std_logic_vector(31 downto 0);
    i_coeff_2 : in  std_logic_vector(31 downto 0);
    i_coeff_3 : in  std_logic_vector(31 downto 0);
    i_coeff_4 : in  std_logic_vector(31 downto 0);
    -- data input
    i_data    : in  std_logic_vector(31 downto 0);
    -- filtered data
    o_data    : out std_logic_vector(31 downto 0);
    --write flag and address for dpram
    we_flag   : out std_logic;
    address   : out std_logic_vector(9 downto 0)
    );
end fir_filter_5;
```

Listing 1: Definition of the entity *fir_filter_5*.

- *s_idle*: this is the initial state; it simply sets to 0 the write flag and then changes to the *s_read* state.
- *s_read*: in this state the program sets the address of the current sample, then it reads the data from this address and stores them in a signed variable. Finally it changes to *s_write*.
- *s_write*: it starts by updating the data pipe with the new input just read, then it enables writing, it sets the address to the second half of the memory and applies the filter to the data pipe. Finally, if the end of the available memory is reached, namely 512 samples, it resets the sample index and data pipe and switches to the initial state *s_idle*, otherwise it increases the sample index by one and changes again the state to *s_read* to continue the signal processing.

3.2. Data Input/Output

In order to exchange input and output data between the PC and the Arty A7 board the IPBUS protocol is used in combination with a dual-port RAM architecture (DPRAM). A DPRAM allows to do simultaneous read and write

```
when s_idle =>
    we_signal  <= '0';
    state_curr <= s_read;

when s_read =>
    --enable reading from RAM
    we_signal   <= '0';
    --select proper RAM address from which to read
    address_output <= std_logic_vector(to_unsigned(sample_index, address'length));
    --read data and store the result as a signed number
    data_temp <= std_logic_vector(signed(i_data));
    --go to FSM next state
    state_curr <= s_write;

when s_write =>
    --shift data towards right by one "block" in data pipe
    p_data  <= signed(data_temp)&p_data(0 to p_data'length-2);
    --enable writing onto RAM
    we_signal   <= '1';
    --select proper RAM address onto which write
    address_output <= std_logic_vector(to_unsigned(sample_index + 512, address'length));
    --compute the filtered output
    output_data <= std_logic_vector( resize(
                    p_data(0)*r_coeff(0)+
                    p_data(1)*r_coeff(1)+
                    p_data(2)*r_coeff(2)+
                    p_data(3)*r_coeff(3)+
                    p_data(4)*r_coeff(4), 32) );
    --check if we already filled the ram available (i.e. 511 slots)
    if sample_index = 511 then
    --if so, then reset the index for sampling, the cached data and go idle
        sample_index <= 0;
        p_data   <= (others => (others => '0'));
        we_signal  <= '0';
        state_curr <= s_idle;
    else
    --else increase it
        sample_index <= sample_index + 1;
    --and read a new value
        state_curr <= s_read;
    end if;
```

Listing 2: Implementation of the FIR finite state machine.

```
entity ipbus_dpram is
        generic( ADDR_WIDTH: natural );
        port(
                clk: in std_logic;
                rst: in std_logic;
                ipb_in: in ipb_wbus;
                ipb_out: out ipb_rbus;
                rclk: in std_logic;
                we: in std_logic := '0';
                --d is data to be written
                d: in std_logic_vector(31 downto 0) := (others => '0');
                --q is data to be read
                q: out std_logic_vector(31 downto 0);
                addr: in std_logic_vector(ADDR_WIDTH - 1 downto 0)
        );
end ipbus_dpram;
```

Listing 3: Definition of the entity *ipbus_dpram*.

operations on the memory; in this way it is possible to send the input signal from the PC to the FPGA and retrieve the filtered signal from the FPGA to the PC. The entity that defines the dual-port RAM to be used with the IPBUS protocol is reported in Listing 3.

All the entities defined above have been included in the *top_level* architecture and their signals have been properly mapped.

Finally, data transfer between PC and FPGA is done using a Python script that makes use of the library *uHAL*, an API for the IPBUS protocol. This script creates the input signal, sends it to the Arty A7 board and reads the filtered signal, allowing us to store or plot the results.

4. Results

In order to compare the results between the hardware implementation and the Python simulation of the FIR, four different input waveforms are generated:

- Sine wave.
- Triangular wave.
- Square wave.

- Train of square waves.

As it is possible to see for each plot (Figures 2,3,4,5), the simulated data superimposed the FIR filter implemented with the FPGA.

In particular, the input is of 512 samples as the output. Further, we can see the expected delay on the output due the coefficients order of the filter.

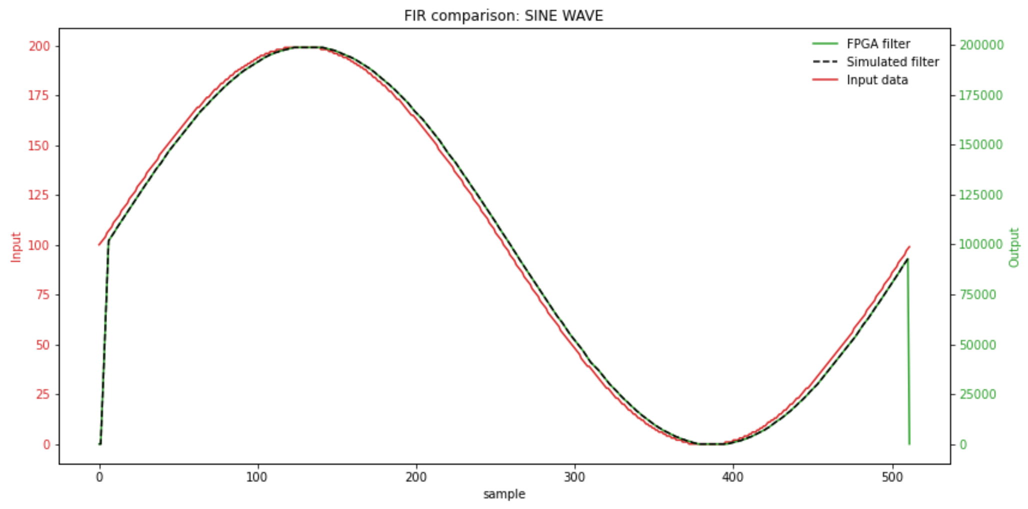

Figure 2: Sine wave.

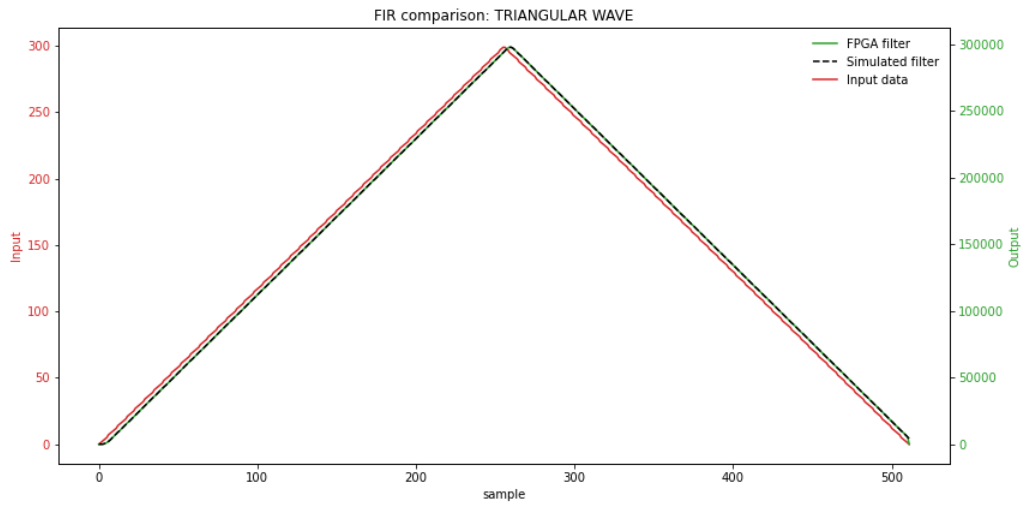

Figure 3: Triangular wave.

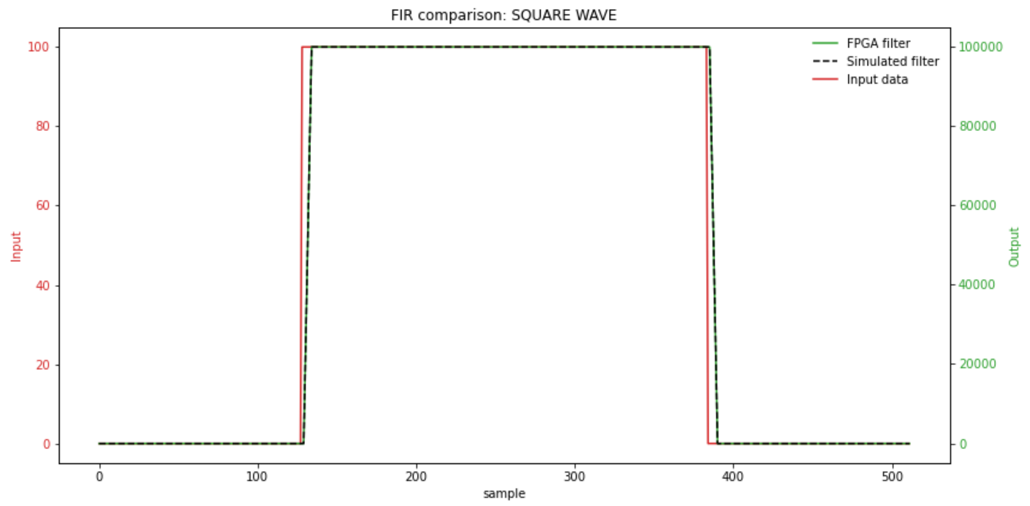

Figure 4: Square wave.

5. Conclusions

In this work, the implementation of a FIR filter co-processor in FPGA is studied. In particular, the IPBUS protocol is developed and the DPRAM memory is managed. Furthermore, the FIR FPGA output is successfully compared with the Python simulation ones, for four different input signals.

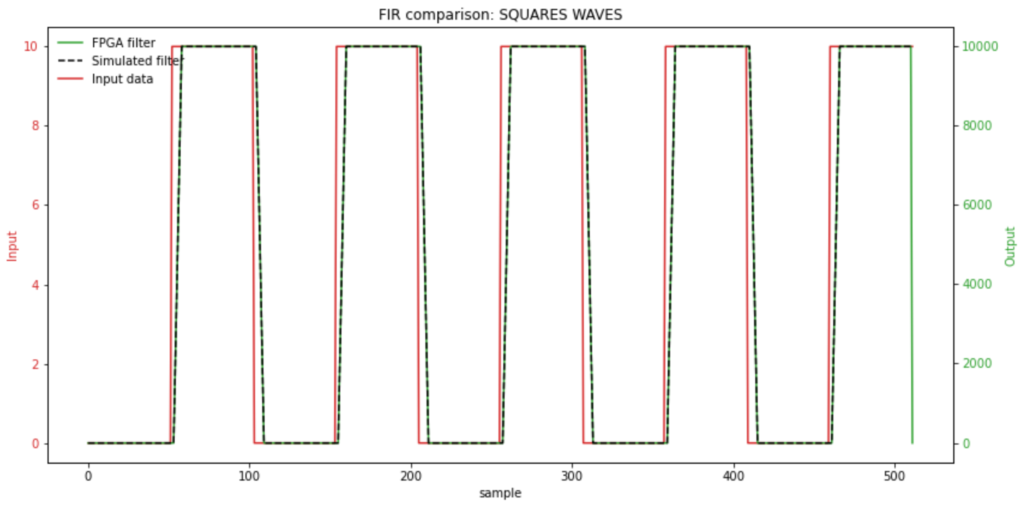

Figure 5: Squares waves.