Management and Analysis of Physics Dataset (mod. A): FIR filter co-processor in FPGA with IPbus protocol

Rocco Ardino 1231629 | Alessandro Lambertini 1242885 | Alice Pagano 1236916 | Michele Puppin 1227474

Thursday 21st May, 2020

1 Aim

In this project we implement a FIR filter co-processor in FPGA, along with input/output data storage and transfer protocols. In particular, we use the IPbus protocol for communication with the FPGA board and a DPRAM component as memory source. We test the hardware implementation of the filter on several input waveforms and we compare the results with the ones obtained through a Python simulation.

2 Implementation

We send data to be filtered from the PC to the FPGA, connected with an Ethernet cable, through the IPbus protocol. Then, data are stored in FPGA memory registers and we use DPRAM to read data from memory. We apply the FIR filter and we store back filtered data using again DPRAM. A cartoon of the entire process is schematized in Fig.1. Eventually, we make use of a Python script in order to interface with the FPGA via IPbus.

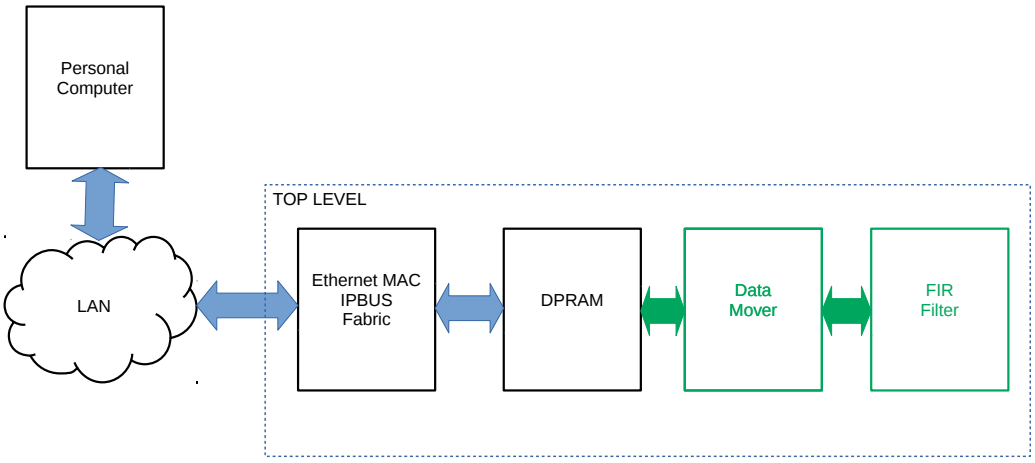

Figure 1: Diagram of implemented FIR filter co-processor in FPGA with IPbus protocol.

Now, we describe in details the structure of the main components and the finite state machine we implement to interface the DPRAM with the FIR filter.

2.1 DPRAM

By definition, a DPRAM is a type of random-access memory that allows multiple reads or writes to occur at the same time. Its VHDL IPbus based implementation is showed in Listing 1 and the main ports are explained below:

- `we`: write/read flag. If set to '0', the DPRAM is in read mode; otherwise it is in write mode.
- `d`: data to be written to a specified address of the DPRAM.
- `q`: data to be read from a specified address of the DPRAM.
- `addr`: address of the DPRAM in which read/write operations are executed.

```
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       : in  std_logic_vector(31 downto 0) := (others => '0');
        q       : out std_logic_vector(31 downto 0);
        addr    : in  std_logic_vector(ADDR_WIDTH - 1 downto 0) );
end ipbus_dpram;
```

Listing 1: `ipbus_dpram` entity.

In particular, the "reading from" and "writing to" process is displayed in Listing 2. At every cycle of the DPRAM clock signal `rclk`, we access to the specified address of the DPRAM by index. Then, if `we` is '0', we read from it, otherwise we write to it.

```
rsel <= to_integer(unsigned(addr));

process(rclk)
begin
    if rising_edge(rclk) then
        q <= ram(rsel);
        if we = '1' then
            ram(rsel) := d;
        end if;
    end if;
end process;
```

Listing 2: Read from/Write to DPRAM process.

In our `top_level` entity we instantiate the IPbus based DPRAM as represented in Listing 3.

```
-- Flash registers
dpram : ipbus_dpram
    generic map(ADDR_WIDTH => ADDR_WIDTH)
    port map(
        clk     => ipb_clk,
        rst     => rst_ipb,
        ipb_in  => ipbw(0),
        ipb_out => ipbr(0),
        rclk    => ipb_clk,
        we      => we_s,
        d       => data_out,    --data to write
        q       => data_in,     --data to read
        addr    => addr_s);
```

Listing 3: `ipbus_dpram` instantiation in the `top_level`.

2.2 FIR Filter

We implement a finite impulse response (FIR) filter, which is a filter whose impulse response is of finite duration. Firstly, we provide a brief mathematical introduction. Given a sequence $\{x_i\}_{i=1,\dots,N}$ 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 5-th order FIR filter. The values of the coefficients are computed through the library *signal* of the Python module *scipy*, by setting a cutoff frequency of 0.1. The frequency analysis for this filter setup is showed in Figure 2.

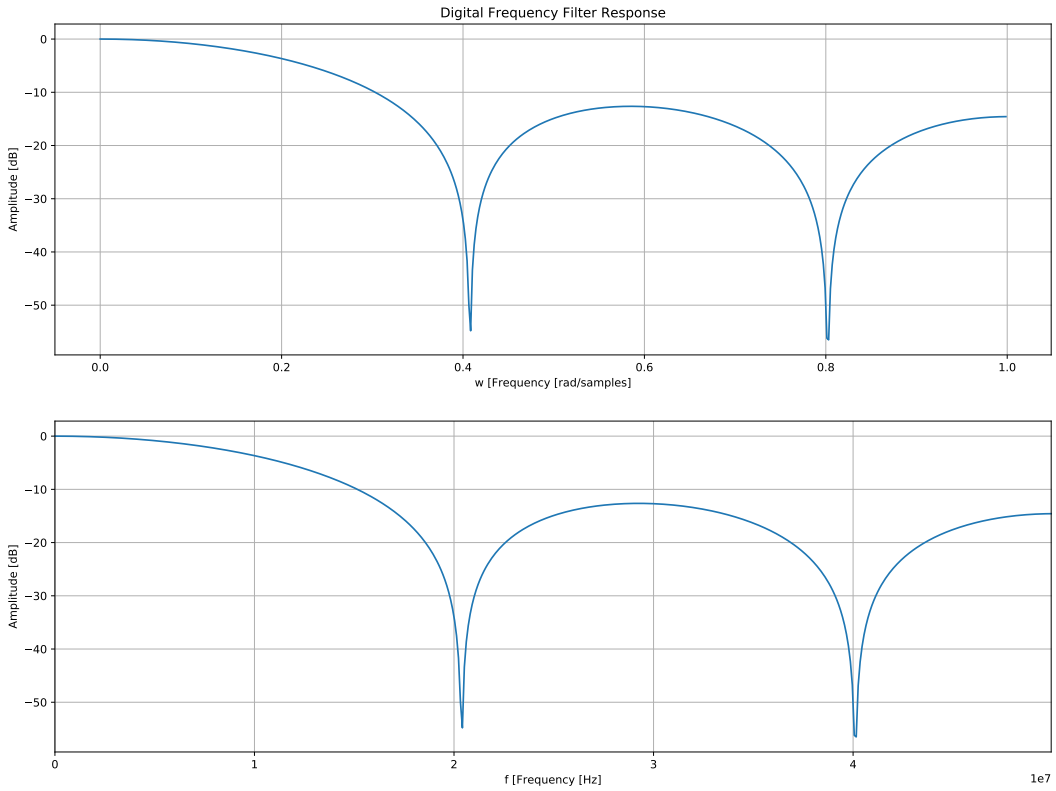

Figure 2: Frequency analysis of the FIR filter with the given configuration.

The values of the coefficients are:

$$\begin{aligned} b_0 &= 0.1933 \\ b_1 &= 0.2033 \\ b_2 &= 0.2066 \\ b_3 &= 0.2033 \\ b_4 &= 0.1933 \end{aligned}$$

Since every operation on the FPGA is done with integer arithmetics, it is fundamental to overcome the limit of the finite precision of those coefficients. An idea is to multipilicate them by a large number such 10^3 and then truncate the floating part. Hence, the values of the coefficients in the VHDL code are set to:

$$\begin{aligned} b_0 = 193_{\text{dec}} &\implies 000000\text{C}1_{\text{hex}} \\ b_1 = 203_{\text{dec}} &\implies 000000\text{CB}_{\text{hex}} \\ b_2 = 206_{\text{dec}} &\implies 000000\text{CE}_{\text{hex}} \\ b_3 = 203_{\text{dec}} &\implies 000000\text{CB}_{\text{hex}} \\ b_4 = 193_{\text{dec}} &\implies 000000\text{C}1_{\text{hex}} \end{aligned}$$

2.3 Finite State Machine

In order to interface the FIR filter with the DPRAM, we implement a finite state machine able to perform read, filter and write operations. In particular, we define the entity `fir_filter` in Listing 4 and the main ports are explained below:

- `i_coeff_*`: coefficients of the FIR filter.
- `we_out`: write/read flag. If set to '0', the state machine enables data read operation; otherwise it enables data write operations.
- `x_in`: input data in 32-bit format.
- `y_out`: output data in 32-bit format.
- `address`: address where input and output data are read or written.

```
entity fir_filter is
    port (
        clk       : in  std_logic;
        rst       : in  std_logic;
        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);
        x_in      : in  std_logic_vector(31 downto 0);
        y_out     : out std_logic_vector(31 downto 0);
        we_out    : out std_logic; -- write enable for the dpram
        address   : out std_logic_vector(9 downto 0) );
end fir_filter;
```

Listing 4: `fir_filter` entity.

In the `top_level` entity we instantiate the component `fir_filter` as in Listing 5. In particular, the ports `x_in`, `y_out`, `we_out` and `address` are mapped to the same signals to which `q`, `p`, `we_s` and `addr_s` are respectively mapped. Note that clock and reset (`clk` and `rst`) are mapped to the ones of IPbus.

```
fir: fir_filter
    port map (
        clk       => ipb_clk,
        rst       => rst_ipb,
        i_coeff_0 => i_coeff_0,
        i_coeff_1 => i_coeff_1,
        i_coeff_2 => i_coeff_2,
        i_coeff_3 => i_coeff_3,
        i_coeff_4 => i_coeff_4,
        x_in      => data_in,
        y_out     => data_out,
        we_out    => we_s, -- write enable for the dpram
        address   => addr_s );
```

Listing 5: `fir_filter` instantiation in the `top_level`.

In the entity `fir_filter` we build a process `fsm_fir` in which the finite state machine is implemented. We perform read and write operations so that we read from the first half of the memory and we write in the second half, until the memory is full and the process starts again overwriting data. In particular, we define a signal `state_fsm` which can be one of the following three possible states: `s_idle`, `s_read` and `s_write`. Moreover, we define an integer signal `samples` which is used as a counter of read/write operations.

Initially, the state is set to `s_idle` where `we_out` and `samples` are set to '0'. Then, at each rising edge of the clock, the machine switches its state from `s_read` to `s_write` or vice versa.

In case `s_read` is selected, as showed in Listing 6, read flag is enabled. The integer signal `samples` is converted to the reading address in order to read from the first half of the memory. The data at that address is then read. Eventually, the state is switched to `s_write`.

```
when s_read =>
    we_out    <= '0';
    address   <= std_logic_vector(to_unsigned(samples, address'length));
    x         <= std_logic_vector(signed(x_in));
    state_fsm <= s_write;
```

Listing 6: Read state of the finite state machine.

In case `s_write` is selected, as shown in Listing 7, write flag is enabled. The writing address is obtained by taking the binary value of the signal `samples` incremented by half of the length of the memory. The data previously read is now filtered and sent to the output variable. Then, the signal `samples` is incremented by one unit. When `samples` reaches 512 (i.e. half of the memory length), we reset the counter in order to start the process all over again.

```
when s_write =>
    we_out <= '1';
    address   <= std_logic_vector(to_unsigned(samples+512, address'length));
    p_data    <= signed(x) & p_data(0 to p_data'length-2);
    y_out     <= std_logic_vector(resize(
                                    p_data(0)*signed(i_coeff_0) +
                                    p_data(1)*signed(i_coeff_1) +
                                    p_data(2)*signed(i_coeff_2) +
                                    p_data(3)*signed(i_coeff_3) +
                                    p_data(4)*signed(i_coeff_4), 32) );
    samples   <= samples + 1;
    if samples = 512 then
        p_data    <= (others => (others => '0'));
        samples   <= 0;
        we_out    <= '0';
        state_fsm <= s_idle;
    else
        state_fsm <= s_read;
    end if;
```

Listing 7: Write state of the finite state machine.

Lastly, a consideration must be done on the intrinsic transient states of the filter. Indeed, since the filter needs the four previous data to produce an output value, we need to set up a data pipe to store them at every cycle of the finite state machine. This is done by exploiting the VHDL operator "&", which concatenates the new input data to be processed with the first four data of the data pipe. However, this procedure is not well defined for the first four input data in the memory since the data pipe is not completely filled when they are processed. Therefore, the filter will behave as expected after four samples are processed.

Every procedure described before is summarized in the schematics in Figure 3.

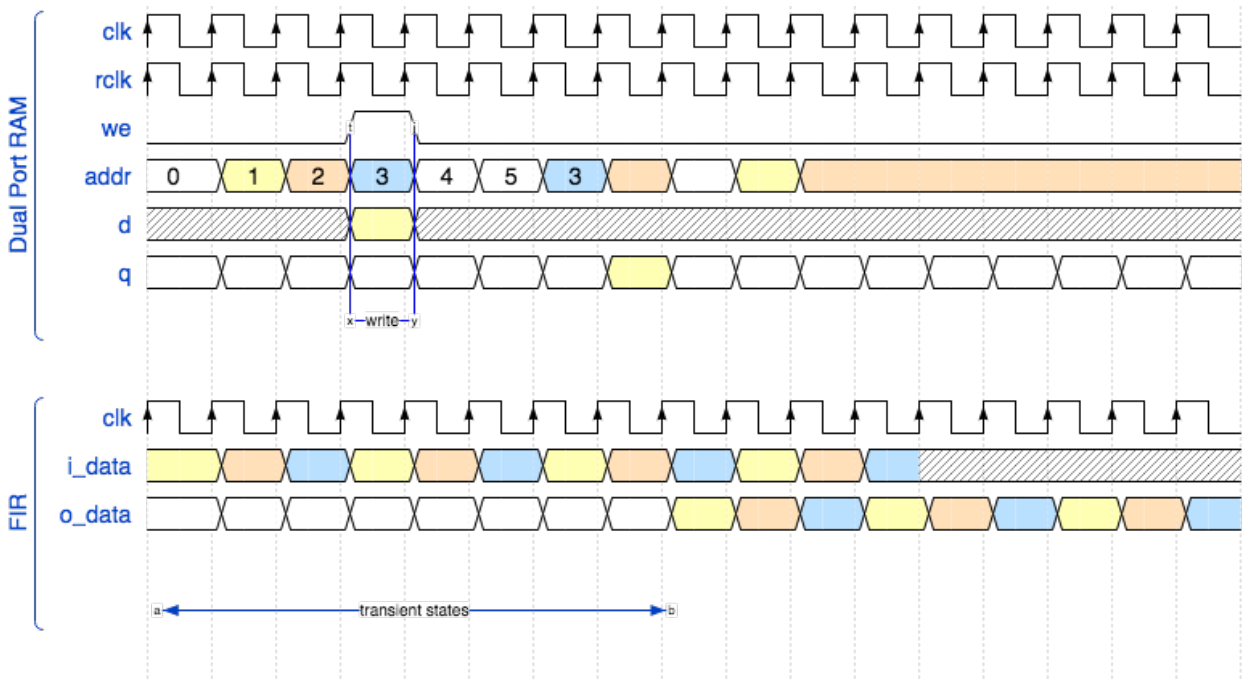

Figure 3: Summary of the working principles of the whole implemented system.

3 Behavioral validation

First of all, we generate the bitstream and program the device. Then, the Arty7 board is connected to a PC through an Ethernet cable and the IP address is configured. In order to upload data and download filtered data respectively to and from the board, we use the Python API uHAL.

We send in input several waveforms to test the FIR filter. Moreover, we compare the results using a Python script which uses the same type of filter with the same parameters.

3.1 Sine wave input

The first waveform under study is a sine wave with the following parameters:

- amplitude: $A = 1024$;
- period: $T = 500$ samples.

The results for this waveform are showed in Figure 4. As we can see from the plot, the filter behaves as expected, except for the samples corresponding to the initial transient states.

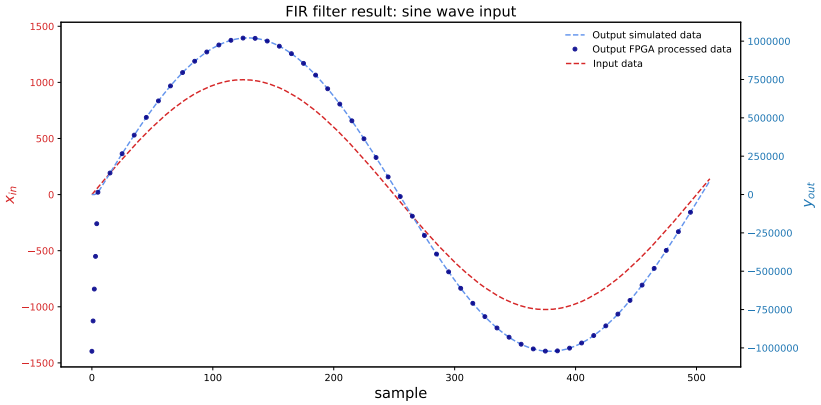

Figure 4: Results for a sine wave input.

3.2 Square wave input

The second waveform under study is a square wave with the following parameters:

- amplitude: $A = 10$;
- period: $T = 100$ samples.

The results for this waveform are showed in Figure 5. As before, the plot proves that the filter behaves as expected, except for the initial transient states.

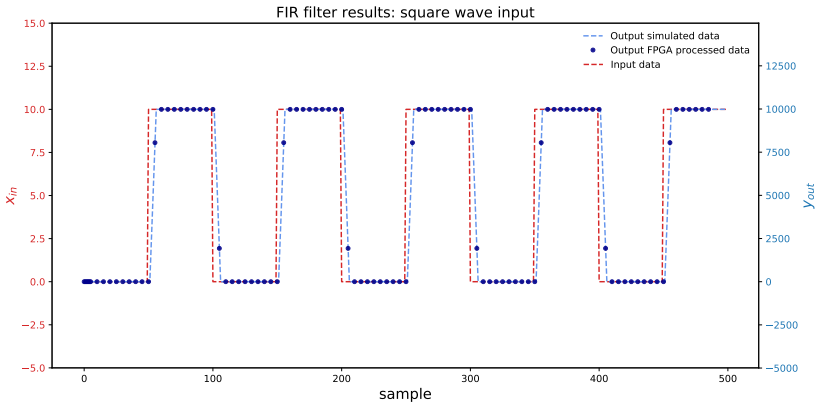

Figure 5: Results for a square wave input.

4 Conclusion

In this assignment we present a FIR filter implemented in FPGA hardware. We exploit IPbus protocol and DPRAM for data transferring and storing. The system has been experimentally tested on Arty7 board for several input waveforms and results have been compared to the ones obtained from a Python script. In both cases results are coherent, except for the transient states.