Management and Analysis of Physics Dataset (Mod. A)
FIR filter with DPRAM in VHDL

Alberto Chimenti[†], Clara Eminente[†]

[†] *Physics of Data M.D. - Dipartimento di Fisica e Astronomia G. Galilei, Università degli Studi di Padova*

December 17, 2020

Abstract

In this project we describe the implementation of a Finite Impulse Response (FIR) filter together with a Dual Port RAM (DPRAM) for data access and memorization in VHDL. In the following we treat an overview of the components to implement and explain the limitations and achievements of our version, showing some application results.

1 Implementation

The project structure follows the diagram in **Figure (1)**.

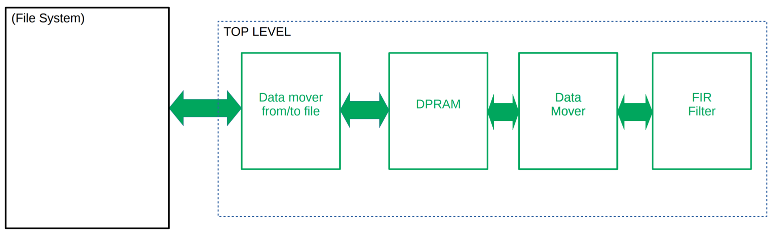


Figure 1: Flowchart of the project

As we can see from the figure the *FIR filter* passes data to the *DPRAM* through the *Data Mover* which handles the address and the write enable signal.

The rightmost block is represented by the testbench simulation script which connects to the file system for reading the input data file containing the signal and writes the output one.
We stress that the whole implementation was not tested on an actual FPGA but that synthesizing the design on *Vivado* resulted in no errors: we can thus conclude that the design should behave as expected and as shown in the final testbenches when uploaded to the board.

1.1 FIR Filter

In digital circuits, a FIR (Finite Impulse Response) filter can be viewed as a functional block that implements equation (1):

$$y[n+1] = \sum_{i=0}^{N} x[n-i] * C_i \tag{1}$$

where:

- N is the number of taps in the filter
- $y[n+1]$ is the output of the filter at the $n+1-th$ timestep
- $x[n-i]$ is the input of the $i-th$ tap at time n
- C_i are the coefficients of the taps

as shown in **Figure (2)** for a 5-tap filter.
We defined the simple script *FF.vhd* to implement the single Flip-Flop components:

```
flipflop: process(clk, rst)
begin
    if rst = '1' then
        Q <= (others => '0');
    elsif(rising_edge(clk) and we='1') then
        Q <= D;
    end if;
end process;
```

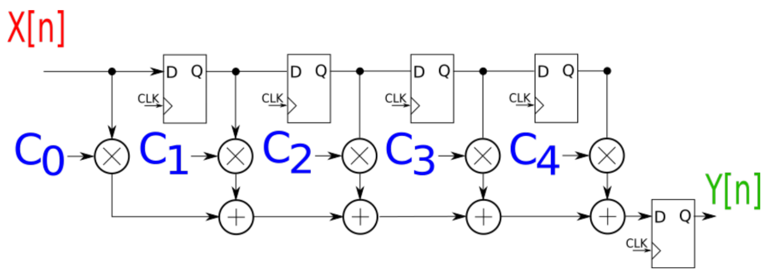


Figure 2: 5-tap FIR schematics

```
ff1 : FF port map(Q => Q1,clk => clk,
    rst => rst,D => x, we=>we_s);
ff2 : FF port map(Q => Q2,clk => clk,
    rst => rst,D => Q1, we=>we_s);
ff3 : FF port map(Q => Q3,clk => clk,
    rst => rst,D => Q2, we=>we_s);
ff4 : FF port map(Q => Q4,clk => clk,
    rst => rst,D => Q3, we=>we_s);
```

We implemented a 5-tap filter with the following coefficients:

$$
\begin{aligned}
C_0 &= 0.193\\
C_1 &= 0.203\\
C_2 &= 0.206\\
C_3 &= 0.203\\
C_4 &= 0.193
\end{aligned}
$$

This corresponds to a filter with cutoff frequency of 0.1 times the sampling frequency. The coefficients were computed using the *scipy.signal* Python library.
It is important to note that it is not possible to represent non-integer values in VHDL. In order to represent and store the coefficients C_i their decimal representation was multiplied by $2^{10} = 1024$. In bynary representation this is equivalent to left-shifting the values of 10 positions. In order to get the correct result of the FIR, its output was right-shifted of the same quantity.

```
y_shifted <= std_logic_vector(RESIZE(SHIFT_RIGHT
    (C0*signed(x)+C1*signed(Q1)+
    C2*signed(Q2)+C3*signed(Q3)+
    C4*signed(Q4)+to_signed(512,2*N),shift),N));
```

Listing 1: FIR output computation

Note that, before shifting back we add $0.5 * 1024 = 512$. This is done to avoid brute floor approximation of the output. Finally we mention that given the bit allocation size N of the input data and coefficients, their multiplication is stored in an allocator of size $2N$ which therefore has to be reshaped with the `RESIZE` function.

1.2 DPRAM

The *dpram.vhd* module implements a very simple process structure:

```
entity dpram is
        generic(
            N       : integer := 16;
                ADDR_WIDTH: natural);
        Port(
                clk: in std_logic;
                rst: in std_logic;
                we: in std_logic := '0';
                d: in std_logic_vector(N-1 downto 0) := (others => '0');
                q: out std_logic_vector(N-1 downto 0);
                addr: in std_logic_vector(ADDR_WIDTH - 1 downto 0));
end dpram;

architecture rtl of dpram is
        type ram_array is array(2 ** ADDR_WIDTH - 1 downto 0) of
            std_logic_vector(15 downto 0);
        shared variable ram: ram_array;
begin
        process(clk)
        begin
                if rising_edge(clk) then
                        if we = '1' then
                            ram(to_integer(unsigned(addr))) := d;
                        end if;
                        q <= ram(to_integer(unsigned(addr)));
                end if;
        end process;
end rtl;
```

Therefore, depending on the write enable signal and the address value the process writes and/or reads the correct data.
The DPRAM itself was virtualized having the FPGA flash memory in mind: it is composed of 2^{ADDR_WIDTH} addresses, each one containing a binary 16-bits value. In our case 1024 addresses were considered and, as it will be more clear in the next section, the first 512 slots were reserved for reading and the half for writing the outputs.

1.3 Finite State Machine

The *fir.vhd* script implements itself the finite state machine (FSM) used for the data mover handler.
Depending on the combination of the enabling signals, the FSM can be in three states: s_{idle}, i.e. waiting, s_{read}, i.e. reading from the given address in the DPRAM and s_{write}, i.e writing in the DPRAM.

```
fsm_fir: process(clk, rst) is
    -- clk, rst: sensitivity list
        begin
        if rst = '1' then
                y_shifted <= (others => '0');
                x <= (others => '0');
                samples <= 0;
                we_s <= '0';
                state_fsm <= s_idle;
        elsif rising_edge(clk) then
                case state_fsm is
                    when s_idle =>
                                when s_idle =>
                                we_s           <= '0';
                                state_fsm      <= s_read;
                    when s_read =>
                                x          <= std_logic_vector(signed(x_in));
                                -- fake signal to change from write value
                                dpram_s    <= (others => '0');
                                ---------------------------------------
                                state_fsm <= s_write;
                                addr_s    <= std_logic_vector(to_unsigned(
                                    samples, addr_s'length));
```

```
                        we_s        <= '0';
                when s_write =>
                        addr_s    <= std_logic_vector(to_unsigned(
                            samples+512, addr_s'length));
                        y_shifted <= std_logic_vector(resize(SHIFT_RIGHT(
                            C0*signed(x)+C1*signed(Q1)+
                            C2*signed(Q2)+C3*signed(Q3)+
                            C4*signed(Q4)+to_signed(512,2*N),shift),N));
                        samples    <= samples + 1;
                        if samples = 512 then
                                y_shifted <= (others => '0');
                                samples   <= 0;
                                we_s        <= '0';
                                state_fsm <= s_idle;
                        else
                                state_fsm <= s_read;
                        end if;
                        we_s        <= '1';
                        dpram_s    <= y_shifted;
                when others =>
                        state_fsm <= s_idle;
            end case;
```

Where the signals and the *dpram* component are defined as:

```
signal x : std_logic_vector(N-1 downto 0);
signal y_shifted : std_logic_vector(N-1 downto 0);
-- write enable and memory address signals
signal we_s : std_logic;
signal addr_s: std_logic_vector(9 downto 0); --address counter
signal samples : integer;

-- finite state machine
type state is (s_idle, s_read, s_write);
signal state_fsm : state;

-- dpram output
signal dpram_s : std_logic_vector(N-1 downto 0);

ram : dpram port map(clk => clk, rst => rst, d => dpram_s, we => we_s, addr => addr_s);
```

The *samples* signal serves as a placeholder for the memory address value. It was crucial in this implementation to correctly synchronize the signal updates with the *dpram* and *ff* processes running in parallel.

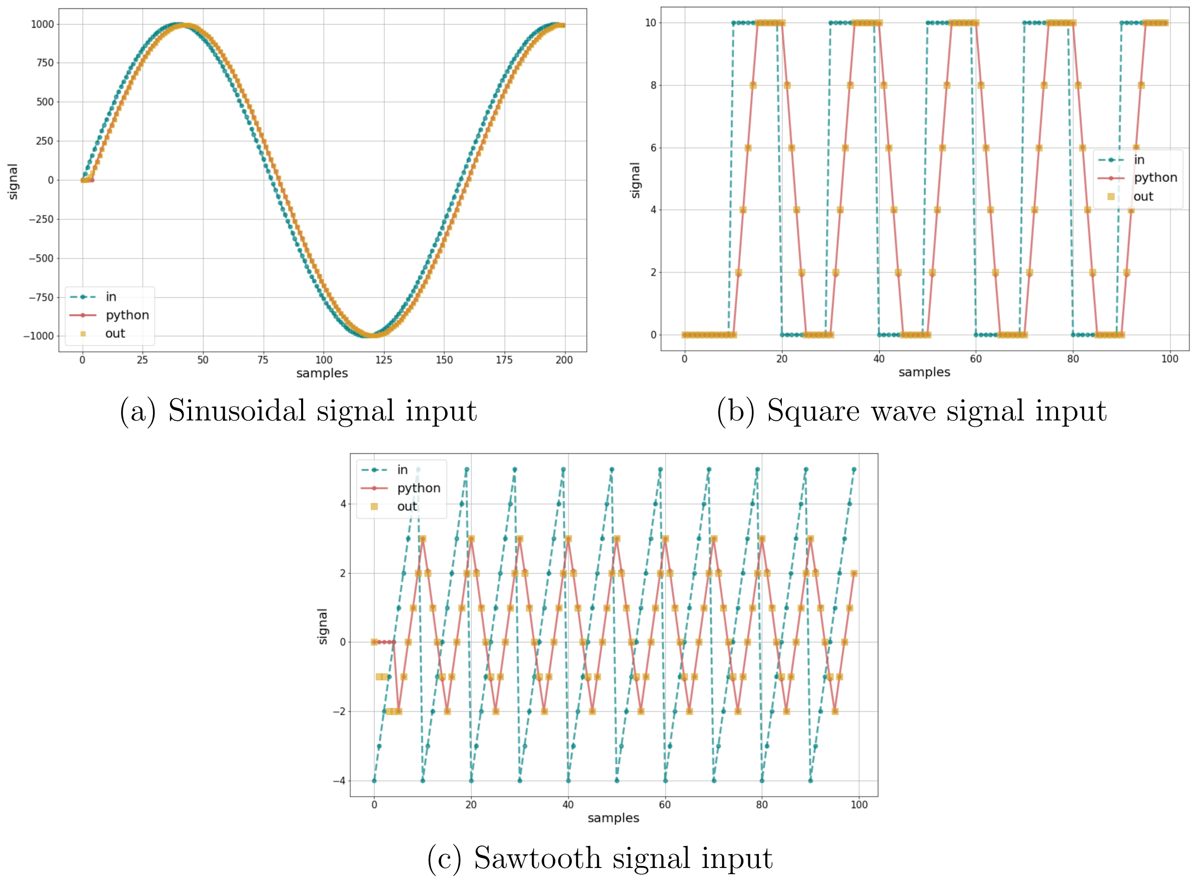
Note that, after the first rising edge of the clock signal (when the reset signal is low) the process needs three clock cycles to compute the first output. Moreover, at this point the FIR starts processing the data and therefore needs #taps − 1 to propagate the first datapoint to the last tap; only at this point the outputs start being actually correct. Therefore, in our case, we need $3 + 2 * (5 - 1) = 11$ clock cycles to output the first meaningful result. Note that the second element was multiplied by two given that a single data point read-write operation takes two clock cycles.

Finally we remark that, during the read state, the process reads the input data directly from the filesystem instead of reading from the dpram. This choice is accommodating the lack of an actual hardware and memory access protocol for the read/write task. Therefore on a practical level, the simulation testbench we implemented simply: reads from the filesystem, processes the data in the FIR module, writes output data in the correct DPRAM address and finally write a text file with the before mentioned outputs.

2 Behavioral Simulation Results

In the following section we report the results obtained confronting the VHDL simulation performed with Vivado and a simple Python script.

We can see in all of the plots that the VHDL simulations results match quite well with the



(a) Sinusoidal signal input

(b) Square wave signal input

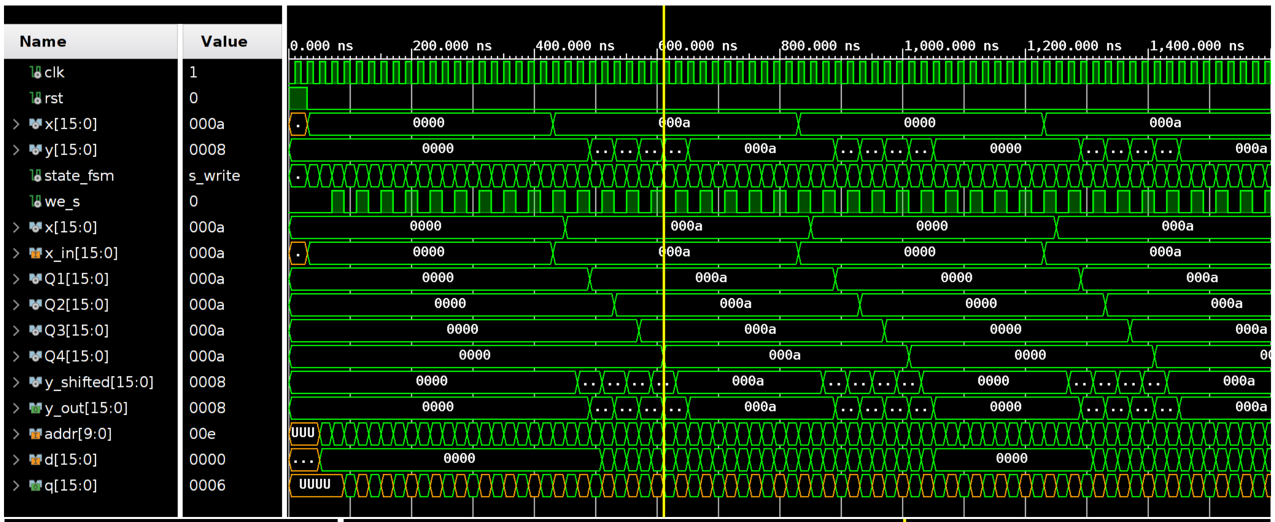
(c) Sawtooth signal input

Figure 3: FIR input/output signal comparison

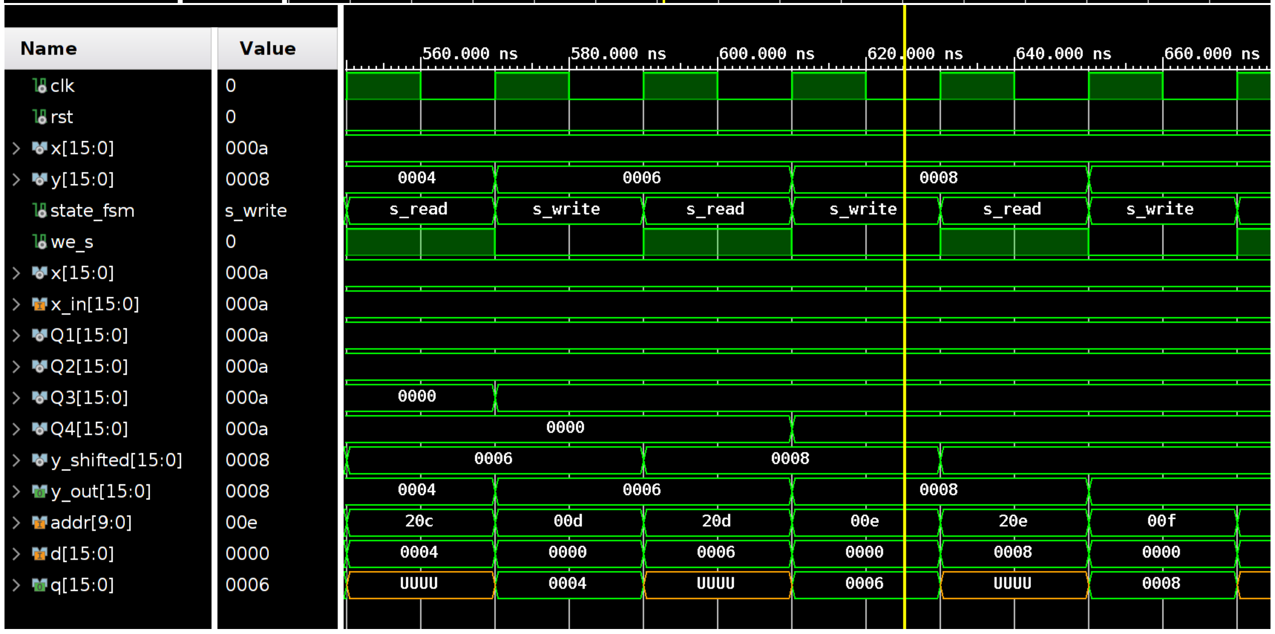
ones obtained with the Python script. However, notice that the FIR outputs recorded during the transient state (i.e. the first four outputs) have been emulated in the Python script by simply padding the output with a number of 0 values equal to the needed clock cycles. This way we match the corresponding output delay.

Testbench Vivado Simulation

Finally, for clarity reasons we report the screenshots of our testbench simulation from Vivado.



Name	Value
clk	1
rst	0
x[15:0]	000a
y[15:0]	0008
state_fsm	s_write
we_s	0
x[15:0]	000a
x_in[15:0]	000a
Q1[15:0]	000a
Q2[15:0]	000a
Q3[15:0]	000a
Q4[15:0]	000a
y_shifted[15:0]	0008
y_out[15:0]	0008
addr[9:0]	00e
d[15:0]	0000
q[15:0]	0006



Name	Value
clk	0
rst	0
x[15:0]	000a
y[15:0]	0008
state_fsm	s_write
we_s	0
x[15:0]	000a
x_in[15:0]	000a
Q1[15:0]	000a
Q2[15:0]	000a
Q3[15:0]	000a
Q4[15:0]	000a
y_shifted[15:0]	0008
y_out[15:0]	0008
addr[9:0]	00e
d[15:0]	0000
q[15:0]	0006