Design and implementation of a FIR filter co-processor in a simulation environment

Segalini Beatrice (1234430) and Iriarte Delfina (1231682)

October 23, 2020

1 Introduction

In this project, a FIR filter is implemented and simulated using **VHDL**. FIR filter coefficients have been computed using the `scipy` module provided by Python. To test the correctness of the design, the observed output has been compared with the calculated output from the Python simulation.

2 Low pass FIR Filter

A digital filter is a system that performs mathematical operations on a signal to reduce or enhance certain features of the processed signal. Digital filters are widely used in digital signal processing applications, such as noise filtering, signal frequency analysis, speech and audio compression and more. A Finite Impulse Response filter, or **FIR filter**, is a common general digital filter which performs time-domain convolution by summing the products of the shifted input samples and a sequence of filter coefficients.

Mathematically, for a causal discrete-time FIR filter of order N, the output signal is given by:

$$\begin{aligned} y[n] &= b_0 \times x[n] + b_1 \times x[n-1] + \cdots + b_n \times x[n-N] \\ &= \sum_{i=0}^{N} b_i x[n-i] \end{aligned}$$

where $x[n]$ is the input signal, $y[n]$ is the output signal, N is the filter order (for a total of $N+1$ taps) and b_i are the values of the impulse response (i.e. the coefficients of the filter).

To practically implement a $N+1-$tap FIR filter, N flip-flops that store the previous values of the input x and another flip-flop that store the output y are required, as shown in Figure 1.

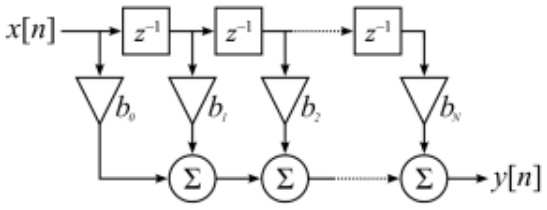

Figure 1: Basic implementation of a FIR Filter.

3 Implementation in Python

In this work, a FIR Filter is implemented by setting a natural frequency of $fs = 400.0\,\text{Hz}$ and a cutoff frequency of 0.1 Hz. The coefficients are computed by using the `scipy.signal` library provided by Python. The frequency analysis for a 16 taps-filter setup is shown in Figure 2.

At the beginning of this analysis, a 5 Hz square waveform sampled at 400 Hz for 1 s is considered. The FIR filter is implemented by using the `lfilter` command provided by `scipy.signal`. The implementation is shown in Listing 1 and corresponds to the convolution between the values. As a further proof of proper design and coding, a composition of sinusoidal waves with different amplitudes and frequencies is modelled as before.

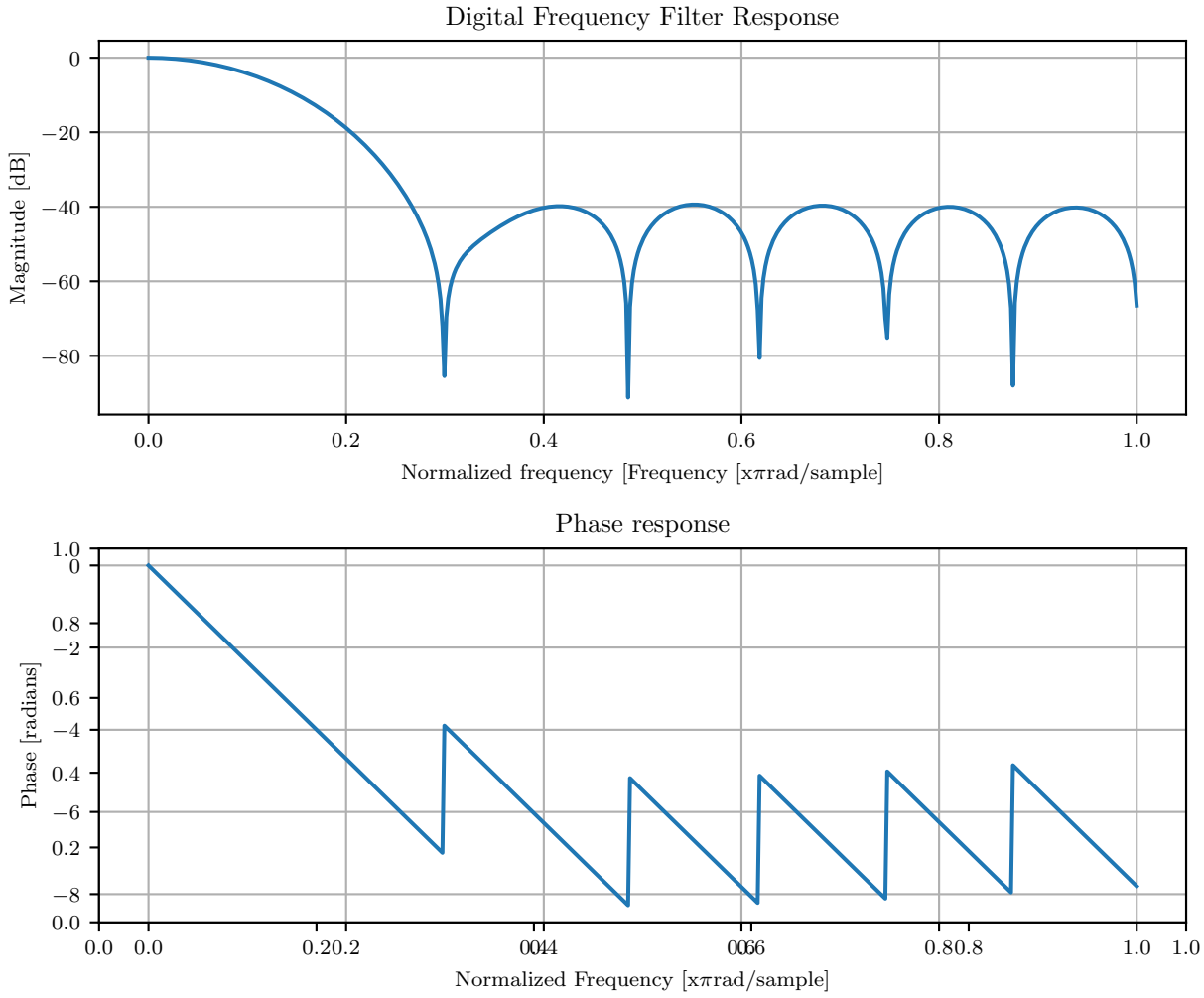

Figure 2: Frequency response of a 16-taps FIR filter.

Listing 1: FIR filter definition with Python

```
 # Use lfilter to filter x with the FIR filter.
filtered_x = lfilter(coefficients, 1.0, input_signal)
```

Generally, in VHDL, binary signed fixed-point number representation is adopted. For this reason, impulse response of a FIR filter needs to be scaled. This is done by multiplying each of the coefficients by 10^4 and then truncate the floating part, in order to avoid loss of precision without altering the shape of the results.

4 Implementation in VHDL

4.1 General structure

The design of the FIR filter is achieved by using a **dual-port RAM (DPRAM)** that allows the process of data reading, and in particular this is done thanks to a *lower half memory* BLOCK-RAM. The input data is then sent to the FIR filter. After applying the FIR filter, the filtered data is written, operating from the *upper half memory* using again DPRAM.

A dedicated state machine able to perform reading and writing operations is implemented as an interface between the FIR filter and the DPRAM, to regulate the read-write process in a controllable way. A schematic representation of the implementation of the project is given in figure 3.

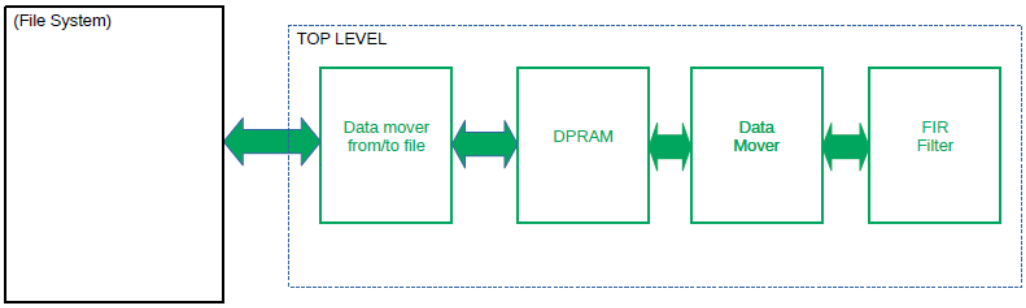

Figure 3: Simple diagram of RAM-FIR filter connections.

In the next sections, a detailed description of the main components of the project can be found.

4.2 FIR filter design in VHDL

The FIR filter is configured by setting two main constant parameters, namely:

- N_TAPS: number of coefficients the FIR filter will have.
- DATA_WIDTH: width in bits of the data input, including the sign bit.

As one can see, they are fundamental to define the data types and comprise the ports, internal pipelines, and internal arrays.

Initially, by setting the configuration of the reset button, all pipeline values and coefficients are set to 0. The system is driven by a clock signal that, going from 0 to 1 (the rising edge), triggers the pipeline data, where a new sample is placed at the beginning and the others are shifted. The convolution of the FIR filter is given simply by the following process (Listing 2):

Listing 2: Convolution of FIR filter implemented in VHDL.

```
p_mult : process (i_rstb, i_clk)
begin
    if(i_rstb = '1') then
        r_mult <= (others => (others => '0'));
    elsif(rising_edge(i_clk)) then
        for k in 0 to N_TAPS-1 loop
            r_mult(k) <= p_data(k) * r_coeff(k); --Perform convolution
        end loop;
    end if;
end process p_mult;
```

4.3 Dual-Port RAM

Random-Access Memory (RAM) is a form of digital data storage. A random-access control circuit allows stored data to be accessed directly in any random order. Dual-Port RAM (**DPRAM**) is a type of random-access memory that allows multiple readings or writings to occur at the same time.

The main ports of a DPRAM are shown in figure 4.

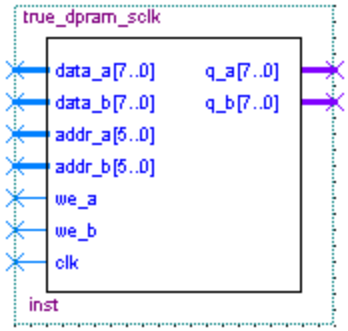

Figure 4: Component ports of the DPRAM.

This device dynamically switches between read and write operations with the write-enable input of the respective port. In Block-RAM, the WRITE operation and the READ are synchronous, meaning that data is read/written simultaneously at rising edge of clock. The code implementation in VHDL for one of the two ports of the DPRAM used in the project is shown in Listing 3.

Listing 3: Main code of port A of DPRAM. The same holds for Port B.

```
if(rising_edge(clk)) then -- Port A
   if(we_a = '1') then
      ram(loc_a) := data_a; -- writes the 'new' data to the RAM
      q_a <= data_a;
   else
      q_a <= ram(loc_a); -- reads the 'old' data to the output
   end if;
end if;
```

4.4 TOP entity: connection with FIR and DPRAM

In the TOP entity, the dedicated state machine is implemented: as stated before, this machine defines the interface between the FIR filter and the DPRAM. The top entity is composed by the ports represented in Figure 5.

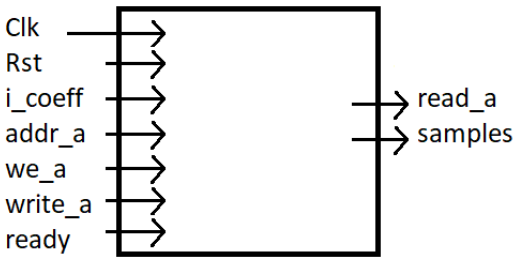

Figure 5: Component ports of the top entity.

Apart from the port declaration, there are many signals that are important in order to communicate among the modules. The signal declarations are the following:

- `we_dpram`: write/enable of the FIR filter.
- `we_b`: write/enable from port B of the DPRAM.
- `write_b`: data input from port B.
- `read_b`: data output from port B.
- `input_data`: data input to the FIR filter.
- `output_data`: data output from the FIR filter.
- `counter`: in order to read/write until the address reaches the HALF RAM value.

For the purpose of this project, a **finite state machine** (FSM) is chosen as interface between the FIR filter and the DPRAM.

A FSM is a hardware component that advances from the current state to the next state at each clock edge. Specifically, the FSM that controls the dual-port memory array is composed of four "dependent" state machines: `s_idle`, `s_ready`, `s_read` and `s_write`.

First of all, to regulate the transition between one state to the other, a clocked process is required. This is done through the code provided in Listing 4. Notice that the next state and the outputs only depend on the current state.

Listing 4: Clocked process.

```
p_clk : process(clk, rstb) is
begin
    if rstb = '1' then
        state <= s_idle;
    elsif rising_edge(clk) then
        state <= state_fsm;
   end if;
end process;
```

Following this, a process `fsm_fir` is defined, in which all the transitions between states are outlined. A schematic representation of the functioning of the FSM is shown in figure 6.

Initially, the state is set to `s_idle` where the signals `we_dpram`, `we_a` are set to 0. This state correspond to the *empty* state, that is triggered by the RESET button. Once the write flag (`we_a` = 1) from port A is enabled, the new data is allowed to be stored in the corresponding address, and hence the FSM moves from `s_idle` to `s_ready`.

The input data is said to be *ready* once all of it has been written into address A, enabling the read flag from port A (`we_a` = 0) and transitioning to state `s_read`.

When `s_read` is selected, data is loaded to the FIR filter and read flag from port B is enabled (`we_b` = 0). Eventually, the state is switched to `s_write`, where the data previously read is now filtered and sent to the output variable, where the output file will be written with a load vector. Consequently, the write flag is enabled from port B. In the case that `s_write` is selected, any FIR input is prevented to load by setting the signal `we_dpram` equal to 0.

The reading process occurs only from the lower half of the memory available, whereas the writing one from the upper half memory of the correspondent address. At each rising edge of the clock, the counter signal is updated and the machine switches its state from `s_read` to `s_write` (or vice versa) until we reach half of the memory capacity, where all the process starts again. The VHDL code implementation is shown in Listing 5.

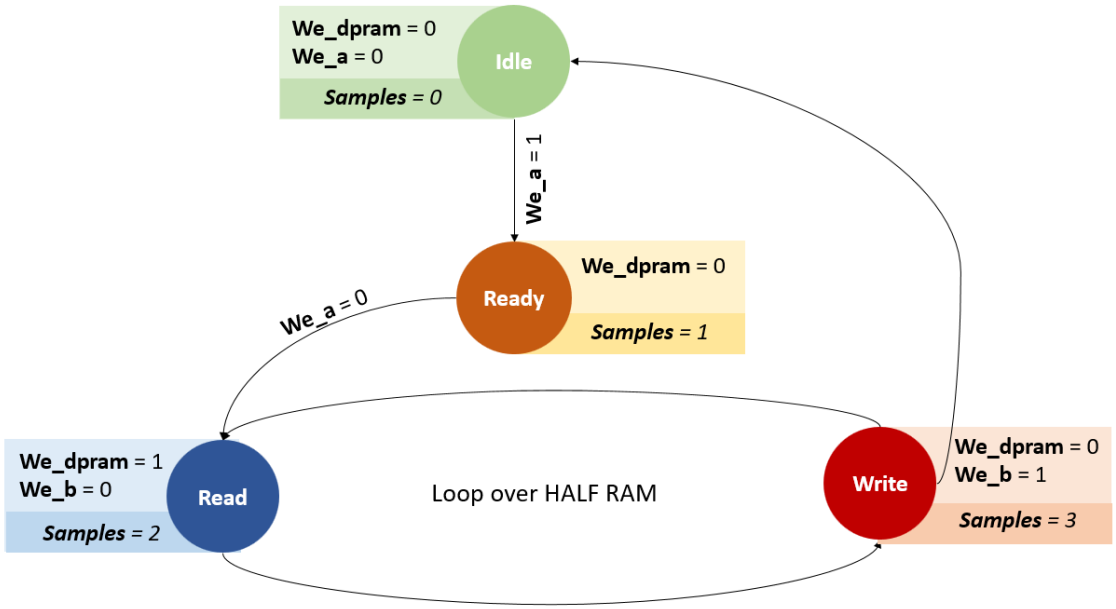

Figure 6: Finite State Machine: depiction of all the states and the processes between them, with associated signal values.

Listing 5: Implementation of the states `s_read` and `s_write` of the finite state machine in VHDL.

```
when s_read =>
    samples <= 2;
    we_b <= '0';
    we_dpram <= '1';
    addr_b <= std_logic_vector(to_unsigned(counter, addr_b'length));
    state_fsm <= s_write;

when s_write =>
    samples <= 3;
    we_dpram <= '0';
    we_b <= '1';
    input_data <= read_b;
    addr_b <= std_logic_vector(to_unsigned(counter + HALF_RAM,
        addr_b'length));
    write_b <= output_data;

    if counter /= HALF_RAM then
        state_fsm <= s_read;
    else
        state_fsm <= s_idle;
    end if;

    if rising_edge(clk) then
        if counter /= HALF_RAM then
            counter <= counter + 1;
        else
            counter <= 0;
        end if;
    end if;
```

4.5 Testbench

The next step is to test the FIR filter. To do that, one needs to provide the input stimuli and store the output. Apart from a clock simulation, the main process of the testbench is given by `WaveGen_Proc` process.

Firstly, the signal is boosted in order to move from the idle state (the "empty" one) into the ready state, where the input vectors are loaded, looping in the lower half ram of the block RAM. Code is displayed in Listing 6.

Listing 6: Simulation of reading.

```
--Next state
wait until rising_edge(clk);

--load all the input vectors
for i in 0 to HALF_RAM - 1 loop
    if not endfile(file_VECTORS) then
        readline(file_VECTORS, v_ILINE);
        read(v_ILINE, i_data_integer);
        write_a <= std_logic_vector(to_signed(i_data_integer,
            write_a'length));
    else
        write_a <= (others => '0');
    end if;

    wait until rising_edge(clk);
    addr_a <= std_logic_vector(unsigned(addr_a) + 1);
end loop;
```

The taps of the FIR filter are loaded when the ready-signal is set to 1. This is done by looping over the TAPS, where coefficients, computed as described in Section 3, are read and stored into the `i_coeff` port.

Recall that the pipeline data of the FIR filter were obtained by shifting the data, since the filter needs the previous data to produce an output value (Section 2). This was done by using the operator "&", which concatenates the new input data to be processed with the first four data of the pipeline data, namely (Listing 7):

Listing 7: Data concatenation.

```
p_data <= signed(i_data)&p_data(0 to p_data'length-2);
if ready = '1' then
   r_coeff <= signed(i_coeff)&r_coeff(0 to r_coeff'length-2);
 end if;
```

This is indeed important since it produces some not clearly defined input. For this reason, when the results obtained are written, we loop over N_TAPS +2 as it can be seen from Listing 8. Notice that the loop is over the upper half RAM.

Listing 8: Writing process with associated shift.

```
 --Next state
wait until rising_edge(clk);
addr_a <= std_logic_vector(to_unsigned(N_TAPS + 2 + HALF_RAM,
    addr_a'length));
wait until rising_edge(clk);

--Loop over the upper half of RAM
for i in N_TAPS + 3 to 2 ** (ADDR_WIDTH - 1) - 1 loop
    addr_a <= std_logic_vector(to_unsigned(i + HALF_RAM, addr_a'length));
    wait until rising_edge(clk);
    --write file
    o_data_integer := to_integer(signed(read_a));
    write(v_OLINE, o_data_integer, left, DATA_WIDTH);
    writeline(file_RESULTS, v_OLINE);

end loop;
```

4.5.1 Simulation results

The simulation obtained by VHDL is reported in Figure 7. It is noticeable that at the beginning all the input data is loaded, using port A, until it reaches the lower HALF RAM of it, corresponding to the value 512. After that, the data is sent to the FIR filter where all the computation is performed.

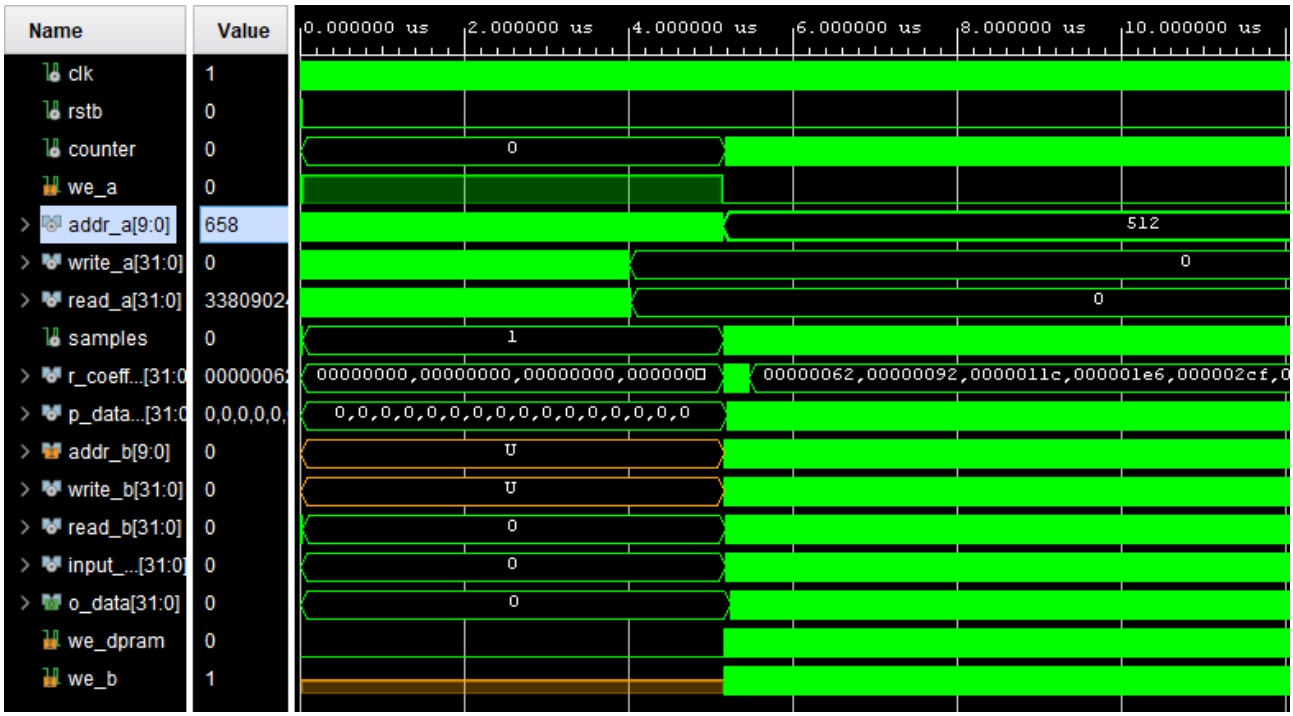

Figure 7: Simulation overview.

A more detailed explanation of the first part of the simulation is given by Figure 8. As it can be seen, the `we_a` is enabled, letting the input data flow trough Port A. In particular, the data enter in the DPRAM using the `write_a` signal and then the DPRAM gives as output the same data, as it can be seen observing the `read_a` signal. This correspond to the *load phase*, which is represented by the signal `samples` equal to 1. As expected, all the pipelines are set to

0, since the data have not been passed to the FIR filter yet. Once the address A reaches HALF RAM, the state machine moves into the next state: `s_read`.

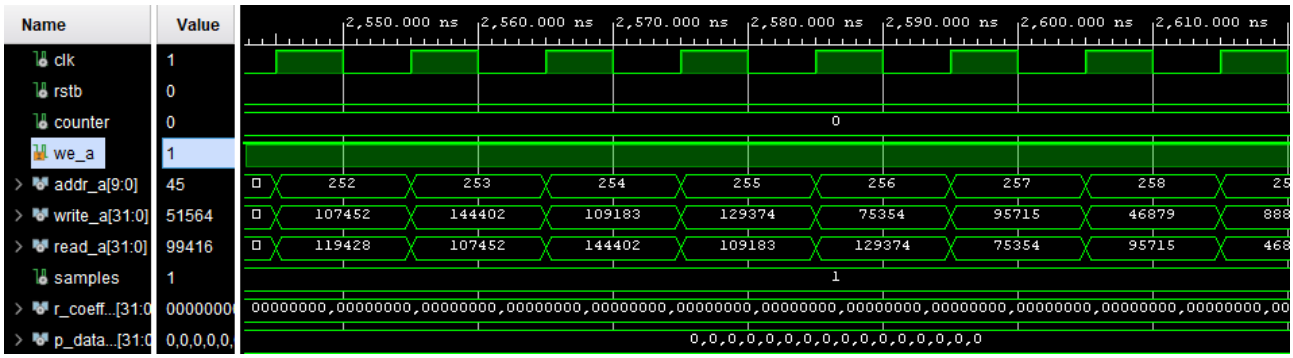

Figure 8: Simulation detail of the loading part.

Once the input vectors are loaded, and lower HALF RAM is filled, the FSM moves to the transition states `s_read` and `s_write`, which correspond to the values of the signal `samples` that are equal to 2 and 3 respectively. The simulation details of this part are displayed in Figure 9.

As expected, one can notice that lower HALF RAM (corresponding of port B of the DPRAM) is dedicated for reading and the upper half for writing. At each rising edge of the clock, the data move into the `s_read` and `s_write` that correspond to shifting the signals `we_dpram` and `we_b` ON and OFF. This behaviour is of course consistent, as it corresponds to enabling the FIR filter and enabling the port B respectively. At each stage it can be observed that the pipeline data is always shifted to receive new data (as desired) and the output data is being written into the external file.

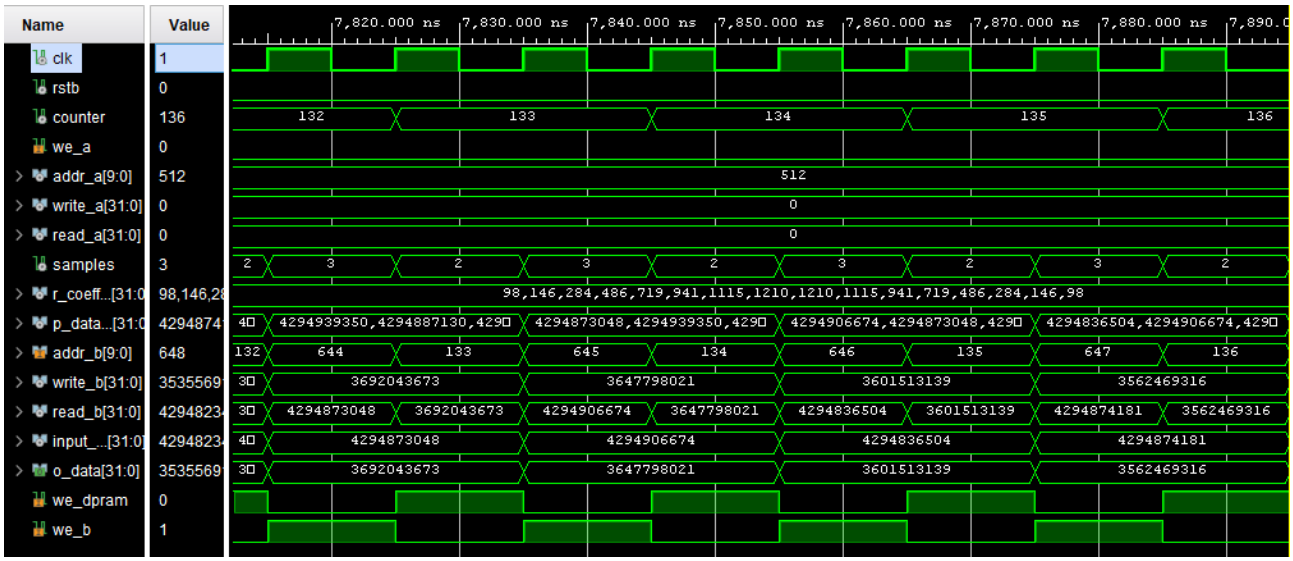

Figure 9: Simulation details of the computational part.

5 Comparison between different implementations

A comparison of a 16-taps FIR filter implemented in both Python and VHDL is analysed in this section and shown in Figure 10. As it can be seen by observing the simulated data and the

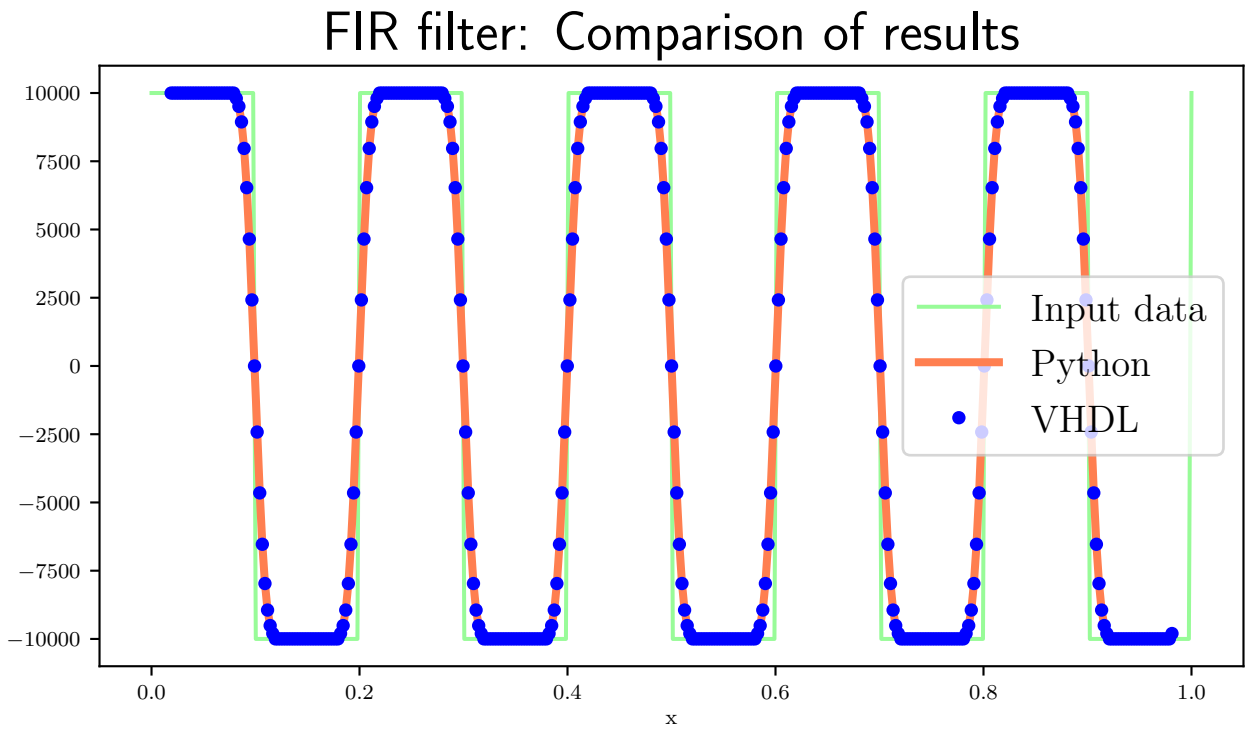

Figure 10: Square wave input signal filtered by a 16-taps FIR filter.

output of the VHDL, the filter behaves as expected: the signal shape of the Python simulated data is the same of the VHDL ones.

Changing the shape of the signal from a square wave to a sinusoidal one still leads to good results. The filter behaviour obtained by Python and VHDL is shown in Figure 11 which exhibits again the correct outcome.

In order to have more insight about the FIR filter, a 5-tap FIR filter is also tested with the sinusoidal wave input. The results achieved by the Python simulation and VHDL are shown in Figure 12. As expected, the filtered signal is more "noisy" than the one obtained by the 16-taps FIR filter.

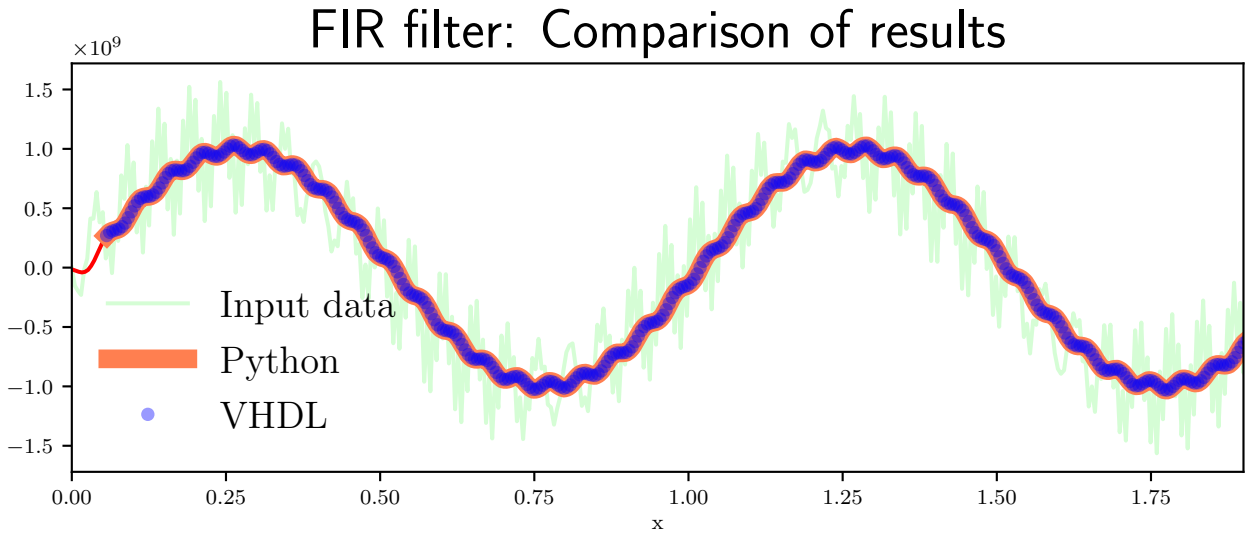

Figure 11: Sinusoidal wave input signal filtered by a 16-taps FIR filter.

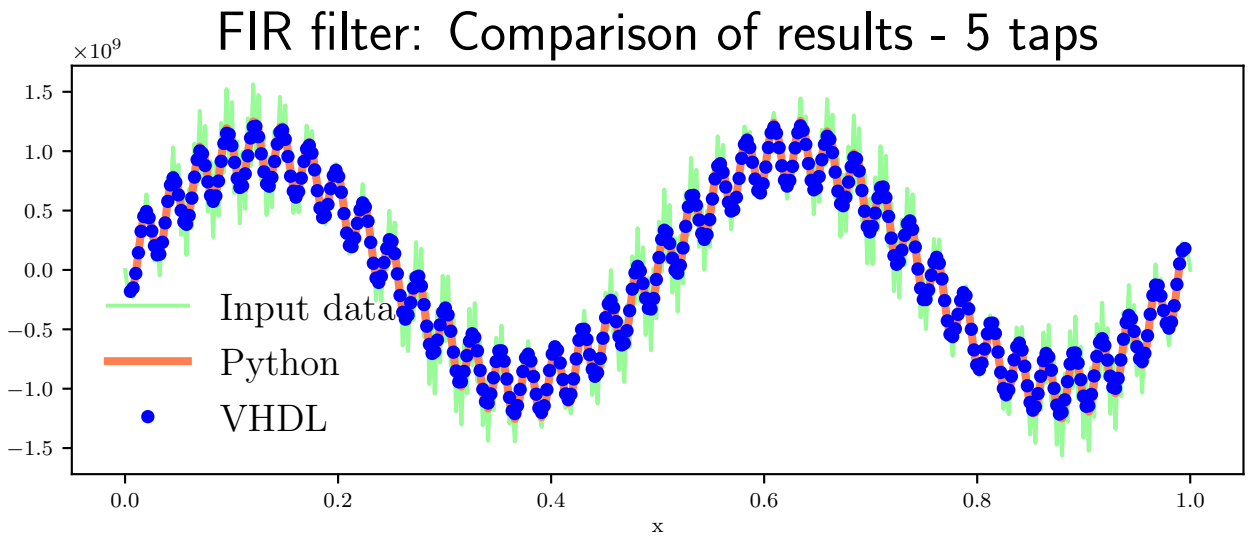

Figure 12: Sinusoidal wave input signal filtered by a 5-taps FIR filter.

6 Conclusions

In this project, a low-pass FIR filter has been successfully implemented in a simulation environment using VHDL and Pyhton. A dual port RAM has been used in order to read/write data and it has been successfully interfaced with a FIR filter by using a dedicated Finite State Machine. Several input signals and tap configurations has been exploited in order to test the programmed device. For all of the cases, the observed output implemented in VHDL proved to be coherent with the simulated output from Python.