Management and Analysis of Physics Datasets - MOD. A

Finite Impulse Response Filter

Samuele Piccinelli 2027650
Cristina Venturini 2022461

1 Introduction

In this project we implemented and simulated a FIR filter using VHDL.
The FIR Filter unit was connected to a UART (receiver and transmitter) and implemented in an FPGA, remotely controlled through *ssh protocol*.
To test the correctness of the design, we compared the observed output with the calculated output from a Python simulation.

1.1 FIR Filter

A digital filter performs mathematical operations on a signal to reduce or enhance specific features of the processed signal.
A Finite Impulse Response Filter, or FIR Filter, is a common digital filter which performs time-domain convolution by summing the products of the shifted input samples and a number of coefficients.
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 coefficients of the filter. To practically implement a Fir Filter with $N+1$ taps,

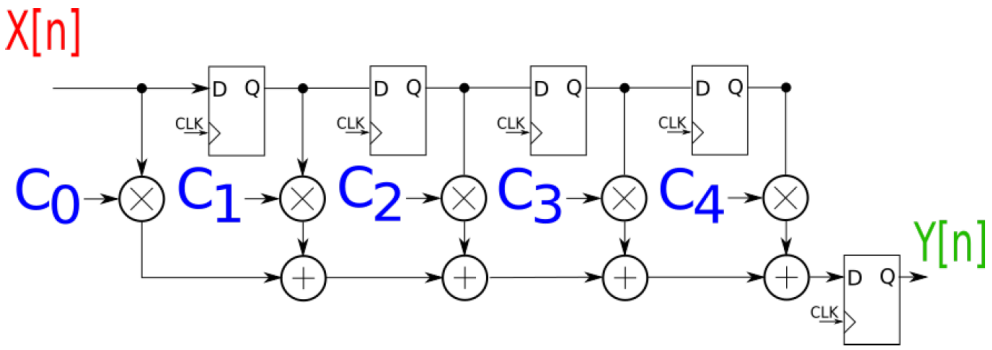

Figure 1: Schematic behaviour of FIR Filter

one needs N flip-flops to store the previous values of the input and another flip-flop to store the output (Figure 1).
In this work we used a 4-taps FIR Filter which acts as low-pass. Coefficients have been computed using the `scipy` module provided by Python; since in order to be implemented in the FPGA they needed to be integer numbers, we rounded them up to the first significant figure and multiplied them by 100.

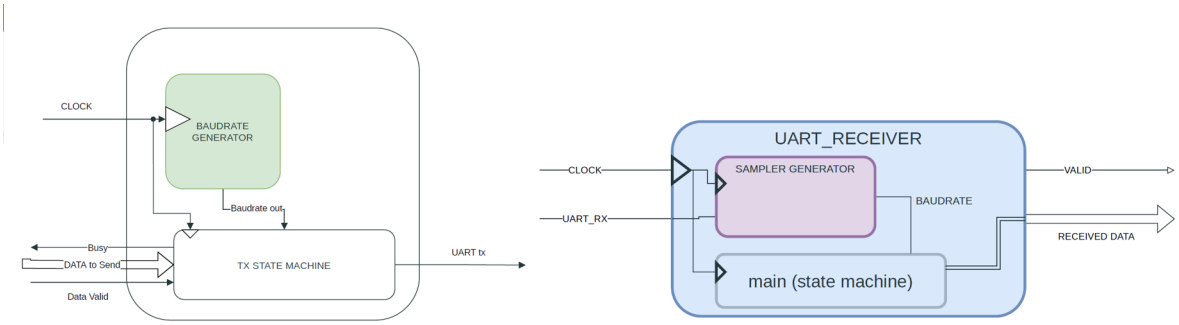

Figure 2: Schematics of the transmitter (left) and receiver (right

1.2 UART

A universal asynchronous receiver-transmitter (UART) is a computer hardware device for asynchronous serial communication.
The main characteristics of an UART receiver-transmitter is that it receives (transmits) bits one by one, framing them between start and stop bits (with respective value of 0 and 1) and does so at a specific rate, called *baudrate*, which is different from the one of the incoming clock. In our example the clock had a 10 ns period (10 MHz frequency) and the baudrate was 115200 (bit/s), meaning that one bit was sent every $100.000.000/115200 = 868.055$ clock cycles.

2 VHDL implementation

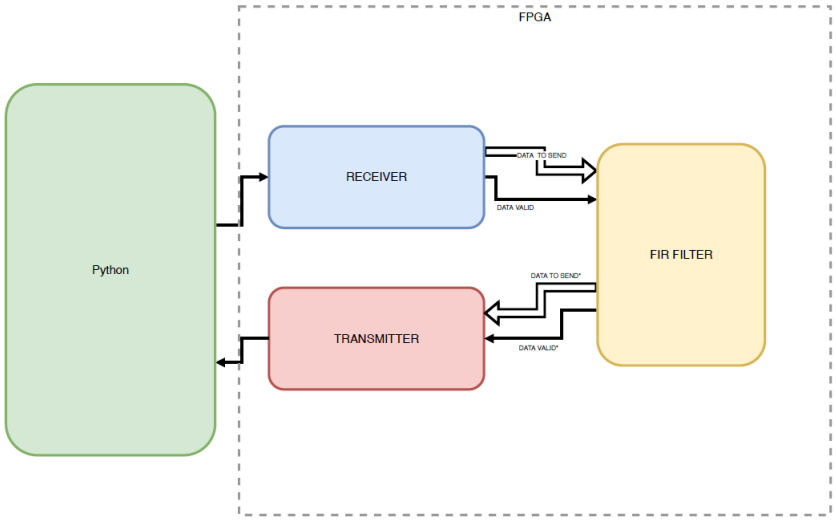

Figure 3: Schematic of the FIR Filter design

The design of the FIR Filter is achieved by using the UART to allow the process of data reading. The input data is written on a serial port, sent to the receiver and then to the FIR Filter. Afterwards, filtered data is sent to the transmitter and then written again on the same serial port.
A schematic representation of the implementation of the project is given in Figure 3.
The receiver unit has one port in input, where data coming from the serial port enter the receiver with a baudrate of 115200 bit per second, one bit at a time. The input is the 8-bit representation of a signed integer natural number.
Once passed through the receiver, the full 8-bit vector is then sent to the FIR Filter through one of

```
data_input : process (i_rstb, i_clk)
begin

    if(i_rstb = '0') then -- reset all signals
        data  <= (others => (others => '0')); -- clear data pipeline values
        coeff <= (others => (others => '0')); -- clear coefficient registers
    elsif(rising_edge(i_clk)) then -- insert new sample at the beginning, shift the others
        if i_valid = '1' then
            data     <= signed(i_data)&data(0 to data'length-2); -- shift new data into data pipeline
            coeff(0) <= signed(i_coeff_0); --input coefficients
            coeff(1) <= signed(i_coeff_1);
            coeff(2) <= signed(i_coeff_2);
            coeff(3) <= signed(i_coeff_3);
        end if;
    end if;
end process data_input;

convolution : process (i_rstb, i_clk)
begin

    if(i_rstb = '0') then
        conv <= (others => (others => '0'));
    elsif(rising_edge(i_clk)) then
        if i_valid = '1' then
            for k in 0 to 3 loop
                conv(k) <= data(k) * coeff(k); -- perform convolution
            end loop;
        end if;
    end if;
```

Figure 4: Processes of FIR entity

the output ports; the other output port carries the validation signal, which activates the processes of the FIR Filter when it has value 1.
Inside the FIR Filter the processes involve the value which is being sent by the receiver and the three previous ones (since we have 4 taps), effectively working as a pipeline and shifting the set of data as needed.
In Figure 4 and 5 we report the main processes of the FIR Filter entity in VHDL: the input process, taking care of the input data and the shifting pipeline, and the computing processes (convolution and sums), which actively perform the filtering.
The filter has two output ports, which connect it to the transmitter. One port is for the actual output, which is sampled by the transmitter one bit at a time at baudrate frequency, the other for a validation signal which is activated in the output process only if the valid signal in input is high. The sampled bits are then sent back to the serial port and information is retrieved.
To perform reading and writing operations on the serial port a Python program was used, executed on the remote machine connected to the board. The program read the input and wrote the output on `.txt` files.
Each entity described above has been implemented and then tested on an appropriate *test bench*; the same has been done for the complete configuration (TOP entity).

3 Python simulation

To compare the results obtained with the FPGA we built a simulation of a FIR Filter with a Python program. Using the `scipy` library the coefficient for the wanted filter behaviour can be found through the method

```
signal.firwin(ntaps, cutoff, passzero = 'lowpass', fs = fs)
```

with `ntaps` being the number of taps (4 in our case) and `cutoffhz` the cutoff frequency of the filter normalised to the sampling frequency `fs`, which yields the values $C_i = [0.047, 0.45, 0.45, 0.047]$. Note that the coefficient are symmetric as expected for a linear phase FIR filter.

```
add0 : process (i_rstb, i_clk)
begin

    if(i_rstb = '0') then
        sum0 <= (others => (others => '0'));
    elsif(rising_edge(i_clk)) then
        if i_valid = '1' then
            for k in 0 to 1 loop
                sum0(k) <= resize(conv(2*k), 2*8+1) + resize(conv(2*k+1), 2*8+1);
            end loop;
        end if;
    end if;
end process add0;

add1 : process (i_rstb, i_clk)
begin

    if(i_rstb = '0') then
        sum1 <= (others => '0');
    elsif(rising_edge(i_clk)) then
        if i_valid = '1' then
            sum1 <= resize(sum0(0), 2*8+2) + resize(sum0(1), 2*8+2);
        end if;
    end if;
end process add1;
```

Figure 5: Processes of FIR entity

We then computed the *low-pass filter* frequency response: the graph of the corresponding gain and phase is shown in Figure 6. The signal used for both the Python simulation and for the VHDL

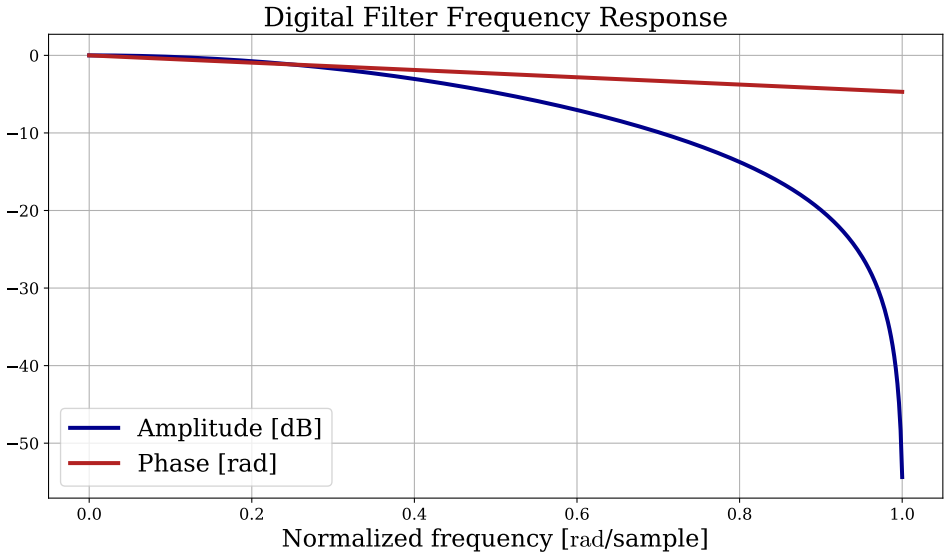

Figure 6: FIR Filter frequency response

program is a sinusoidal waveform sampled at $fs = 200\,\text{Hz}$ composed as the sum of 2 waves weighted with different amplitudes in the ratio $A_2 : A_1 = 1 : 30$: the main component has a frequency $f_1 = 10\,\text{Hz}$, while the second represents the high-frequency noise with $f_2 = 500\,\text{Hz}$.

4 FPGA Response

As input data we used the signal generated for the Python simulation scaled to the amplitude range of $[-127; 126]$.
The input to the serial port needs to be integers in the range $[0; 255]$ (in order to send the information the built-in Python function `chr()` is used) so we scaled the signal accordingly (representing a

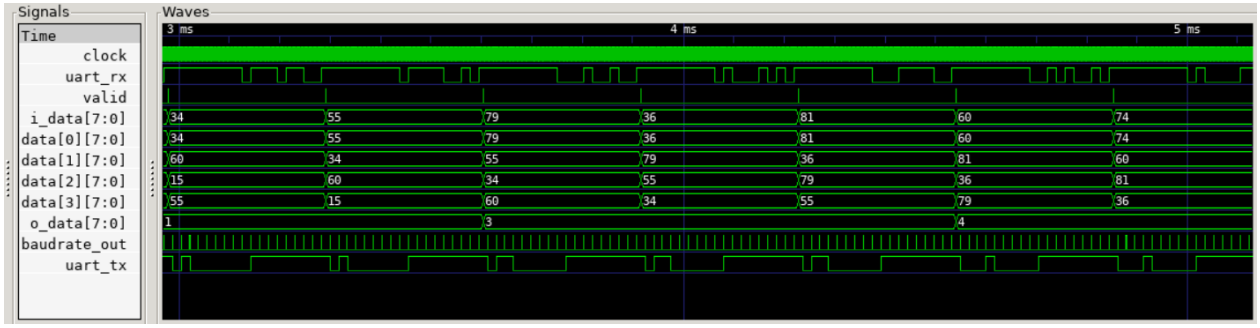

Figure 7: TOP testbench

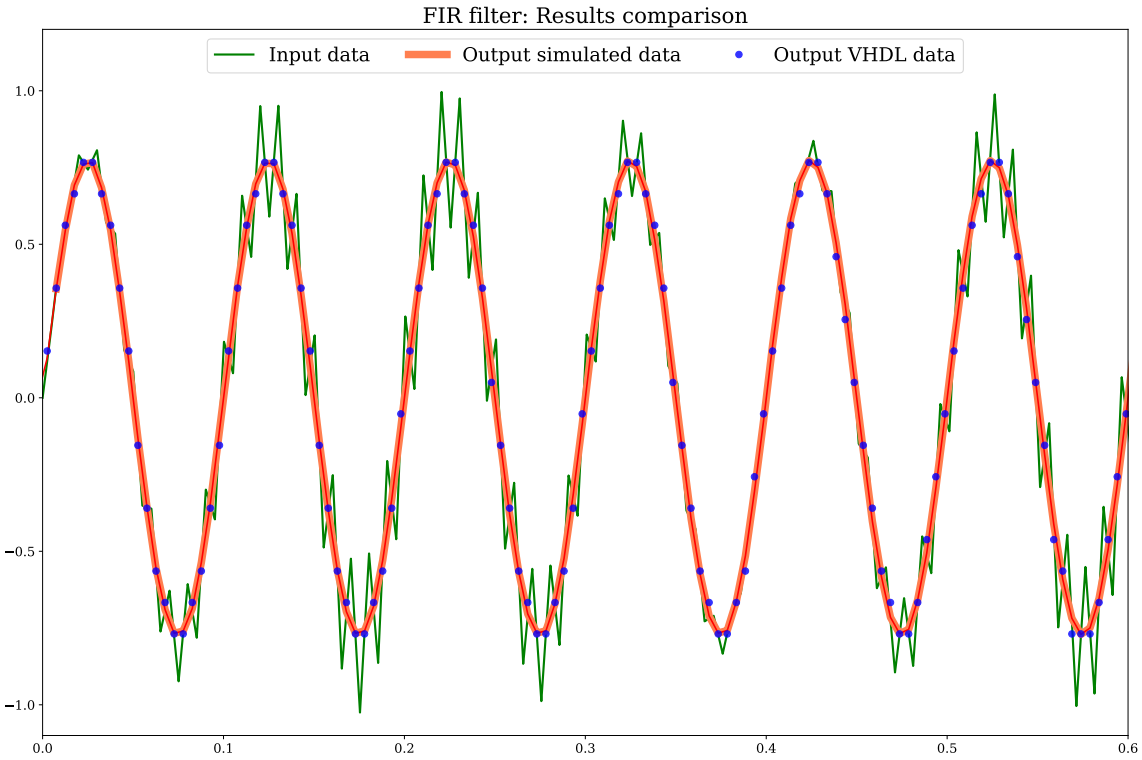

Figure 8: FIR Filter: Results comparison

certain negative value a as $256 + a$). The 2-complement representation was adopted.
The complete VHDL program (TOP entity: UART and FIR Filter combined) was simulated through a test bench, which confirmed that the program was working correctly, as it can be seen from Figure 7. Since the filter works combining linearly the incoming signals, the output is not directly a 8-bit vector, but an 18-bit one. This means that the filtered data needs to be truncated. Data will then need to be re-normalised in order to have the correct output signal; the normalisation factor is $2^{10}/10^4$, since truncating an N-bit number to an M-bit number corresponds to dividing by 2^{N-M} and the multiplication of both input signal and coefficients by 10^2 needs to be accounted of.
The rounding error made when the coefficients and the input data are transformed from float to integers in order to be implemented in VHDL is probably the main reason data coming from the FPGA differs from the output simulated in Python.

In Figure 8 we show the plots of the simulated output and of the VHDL output. The VHDL output points lie on the simulated output signal, further confirming the FIR Filter has been correctly implemented.
Furthermore, even though the number of taps is really low (only 4), it can clearly be seen that the

output signal is significantly smoother than the input one.

5 Conclusions

Since results from the VHDL implementation ran on the FPGA optimally match the ones simulated in Python, we conclude that the FIR Filter has been correctly implemented and that it efficiently simulates a low-pass filter.

6 Appendix and Backup

Nyquist–Shannon sampling theorem: If a function $x(t)$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced $1/(2B)$ seconds apart.

We generated a new wave as sum of 3 monochromatic waves with different amplitudes and set $f_s = 800\,\text{Hz}$; the Python code reads as follows:

```
import numpy as np

def sine_wave(A, time, f): # creates a sine wave
    return A * np.sin(2 * np.pi * f * time)

f1, f2, f3 = 40, 280, 320
A1, A2, A3 = 0.8, 0.4, 0.3
t = np.linspace(0, 1., 800) # f* waveform sampled at 800 Hz for 1s
in_signal_sin = sine_wave(A1, t, f1) + sine_wave(A2, t, f2) + sine_wave(A3, t, f3)
```

In order to plot the filtered signal, the resulting output wave needs to be shifted: the first $N-1$ samples are "corrupted" by the initial conditions since it takes $N-1$ iterations to fill the pipeline with the first values initially set to 0.

```
# The phase delay of the filtered signal.
delay = 0.5 * (n_taps-1) / 800
```

We choose $f_{cut} = 160\,\text{Hz}$, leading to `cutoff`$= 0.4$.

In Figure 9 the obtained results for this analysis are shown.

Figure 9: Results comparison

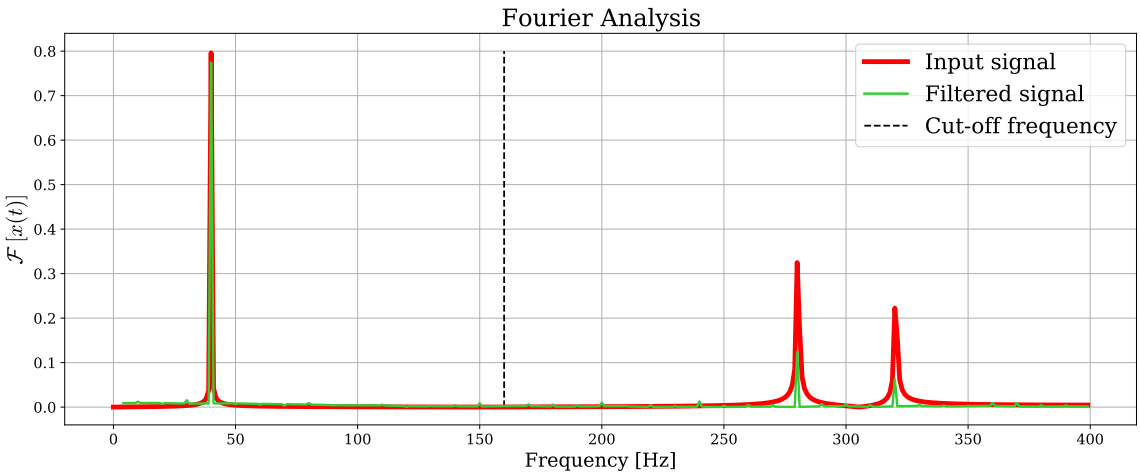

Figure 10: Fourier analysis

The Fourier analysis of incoming and outgoing signals in Figure 10 for the filter shows the expected behaviour for a low-pass filter, i.e. the 320 Hz and 280 Hz wave component is greatly reduced while the ground frequency of 40 Hz shows little affection. This further validates the correct implementation of the filter.

We additionally tested the filter behaviour on a square input signal generated as follows:

```
t_sq = np.linspace(0, 1, 400)
in_signal_sq = 126*signal.square(2 * np.pi * 5 * t_sq)
# 5Hz waveform sampled at 200 Hz for 1s
```

The results obtained are show in Figure 11.

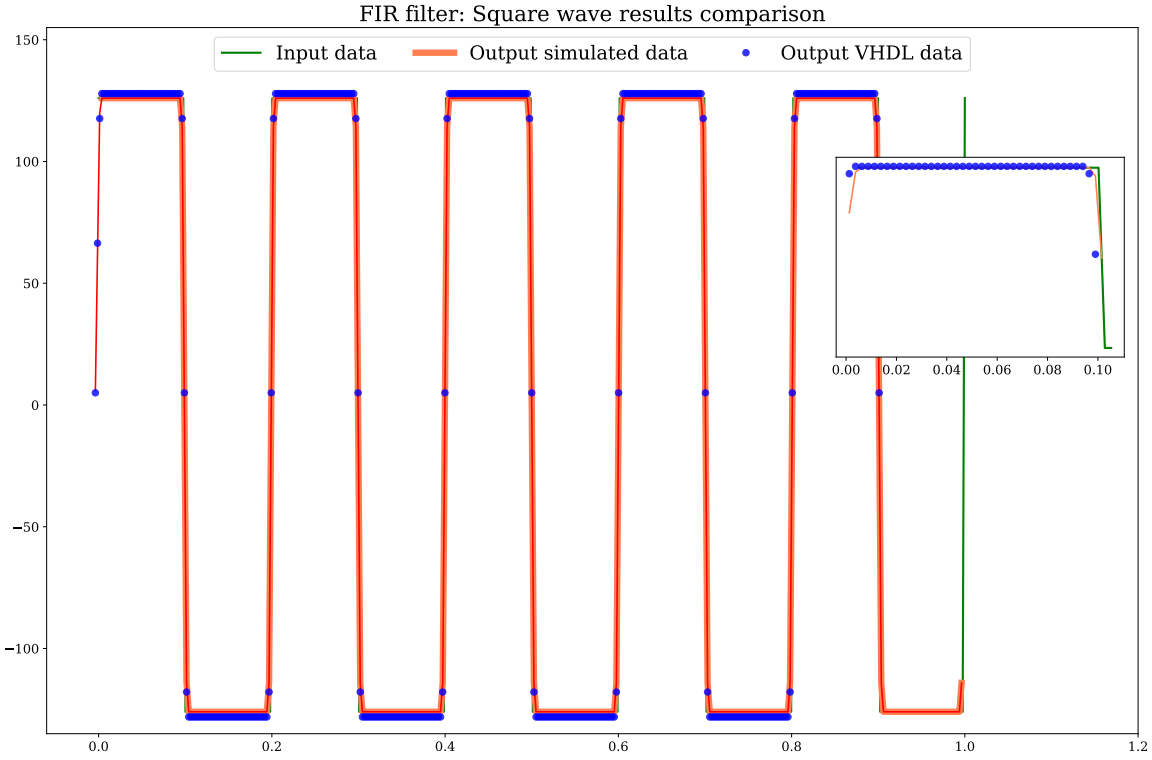

Figure 11: Results comparison for a square wave