MANAGEMENT AND ANALYSIS OF PHYSICS DATASET - MOD. A - Finite Impulse Response Filter

Marcomini Alessandro - 2024286
Scanu Andrea - 2022460

26/02/2021

1 Project goals

Our work aims to analyze the response of a Finite Impulse Response (FIR) filter to a given input signal, both in simulation and under real implementation in a FPGA board. Moreover, we provide a direct comparison between the filter designed both in VHDL language and in Python.
Our full code can be found on GitHub at: https://github.com/AleMarcomini/MAPD_A_y2021.

2 FIR filter and UART

The behaviour of a 4-taps FIR filter is schematized in the picture below.

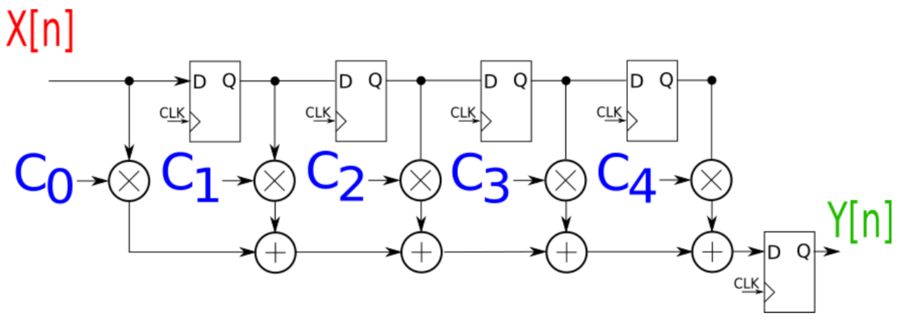
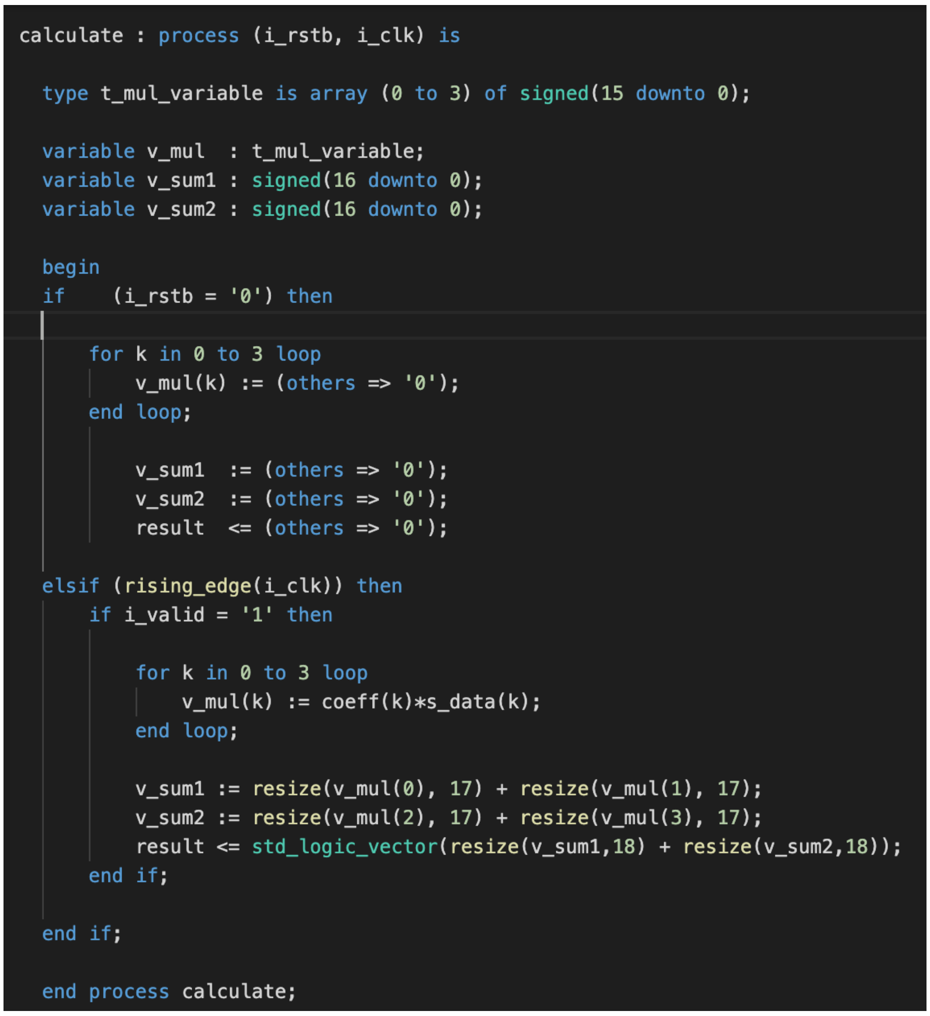


Figura 1: Schematic behaviour of FIR filter.

The filter is composed of four registers that save four data vectors and a set of four coefficients. The filter performs the multiplication of each input by its correspondent coefficient and returns the sum of these values (we discuss our choices for the filter as well as weights values in the next section). In order to implement such coefficients in the FPGA board it has been necessary to turn them into integer numbers. Therefore, we set the values by approximating at the first significant figure and multiplying by 100. Such truncation leads to a little error which is analyzed in the following sections. Data are eventually adapted by use of normalization factors after the output.

We report the main process (calculation) of the FIR filter module. Other processes composing the filter are the input and output ones:



```vhdl
calculate : process (i_rstb, i_clk) is

  type t_mul_variable is array (0 to 3) of signed(15 downto 0);

  variable v_mul  : t_mul_variable;
  variable v_sum1 : signed(16 downto 0);
  variable v_sum2 : signed(16 downto 0);

  begin
  if    (i_rstb = '0') then

      for k in 0 to 3 loop
          v_mul(k) := (others => '0');
      end loop;

          v_sum1  := (others => '0');
          v_sum2  := (others => '0');
          result  <= (others => '0');

  elsif (rising_edge(i_clk)) then
      if i_valid = '1' then

          for k in 0 to 3 loop
              v_mul(k) := coeff(k)*s_data(k);
          end loop;

          v_sum1 := resize(v_mul(0), 17) + resize(v_mul(1), 17);
          v_sum2 := resize(v_mul(2), 17) + resize(v_mul(3), 17);
          result <= std_logic_vector(resize(v_sum1,18) + resize(v_sum2,18));
      end if;

  end if;

  end process calculate;
```

Figura 2: Main process in FIR_filter.vhd

To make the filter able to communicate with a serial port we connected it to a full universal asynchronous receiver-transmitter (UART) machine: this is composed by a receiver module and a transmitter module whose composition is depicted in the figure 3.

The receiver takes the input with a baudrate of 115200 symbols per second from the serial port one bit per time, which is the 8-bit representation of an integer number in the range $[0, 255]$, and stores it until the full 8-bit vector is reconstructed. Once the reconstruction is complete, it sends the byte in output together with a validation signal. This output was connected to the filter that, accordingly to the value of the validation bit, activates its internal processes storing the new signal and shifting the previous ones. This means that for each upcoming value the filtering operations involve the value itself and the previous three ones, making our machine to be able to shift over the whole input set of data. The output of the filter is then connected to the transmitter value

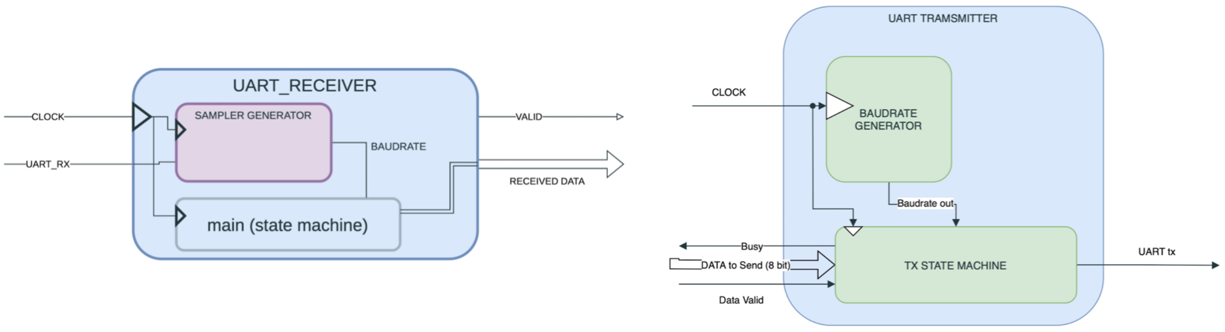


Figura 3: Schematic behaviour of UART receiver and transmitter.

together with another validation signal. The transmitter receives filtered data once ready and starts to sample each bit of the 8-bit vector at a baudrate frequency, so to send them back to the serial port in an optimal way.

Input and output values have been managed by a Python program placed on the remote machine physically connected to the board. The program reads input data from a .txt file and writes the outputs on another one, being this a comfortable way for us to collect them and directly compare them to the expected results.

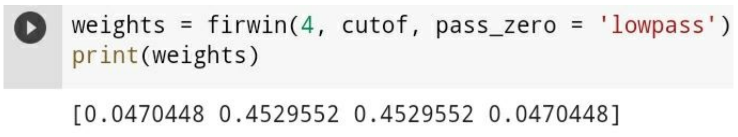
Before moving to program the FPGA, we tested the correct behaviour of our code on different test benches for the individual components and on another for the whole machine. After the FPGA implementation we compared expected and real behaviours to observe that outcomes were optimally similar.

3 Simulation with Python

In order to compare the FPGA board output with the behaviour of a FIR filter we implemented this kind of filter in a Python program. The scipy library contains different methods to find the coefficients for every kind of possible filters (highpass, lowpass,...), in particular with the method

scipy.signal.firwin(numtaps, cutoff, fs)

it is possible to choose the number of taps (4 in our case), the sample frequency and the window where we will filter the input signal, using the cutoff variable. The filter we have choosen is a *lowpass* filter with a cutoff frequency of $0.01\frac{rad}{sample}$ (normalized with the sample frequency), so using the following code

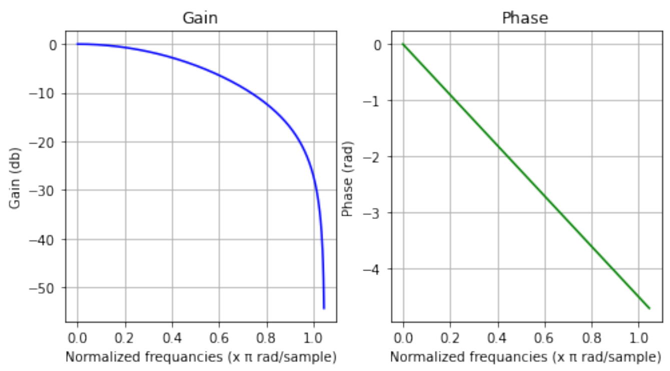


```
weights = firwin(4, cutof, pass_zero = 'lowpass')
print(weights)
```

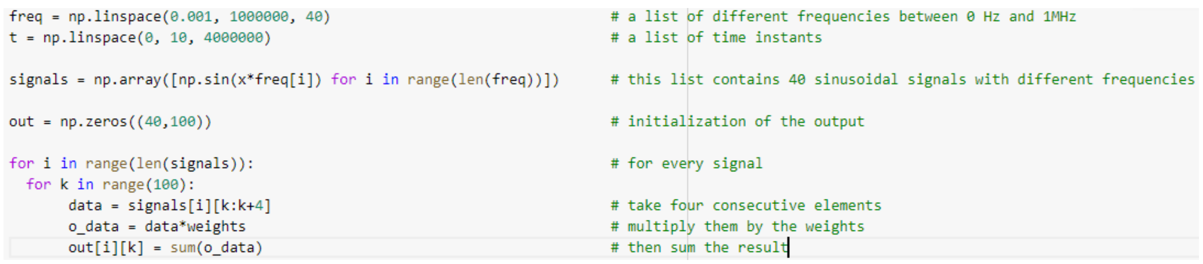
```
[0.0470448 0.4529552 0.4529552 0.0470448]
```

we get the weights $[0.047, 0.453, 0.453, 0.047]$ to be used.

Using a FIR filter with those weights the response in the frequency domain is



Since we have the weights for all the four taps we can perform the simulation of a FIR filter: the code we implemented is



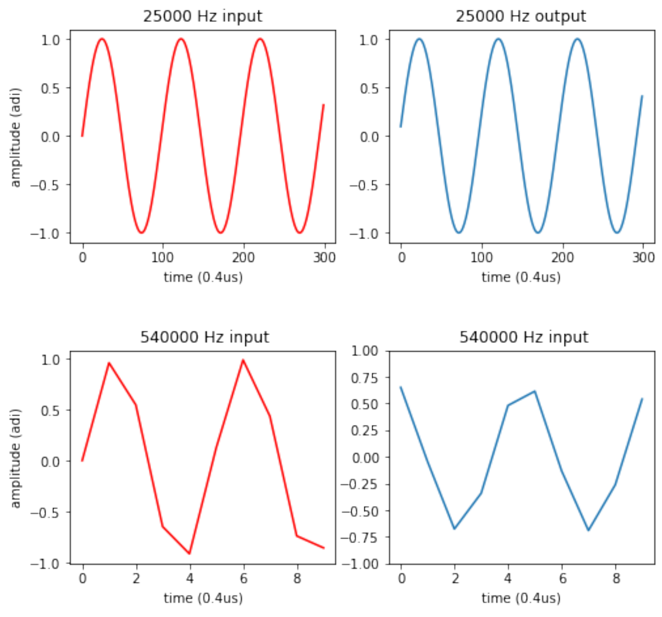
```
freq = np.linspace(0.001, 1000000, 40)                                  # a list of different frequencies between 0 Hz and 1MHz
t = np.linspace(0, 10, 4000000)                                         # a list of time instants

signals = np.array([np.sin(x*freq[i]) for i in range(len(freq))])      # this list contains 40 sinusoidal signals with different frequencies

out = np.zeros((40,100))                                                # initialization of the output

for i in range(len(signals)):                                           # for every signal
  for k in range(100):
       data = signals[i][k:k+4]                                         # take four consecutive elements
       o_data = data*weights                                            # multiply them by the weights
       out[i][k] = sum(o_data)                                          # then sum the result
```

We can now see how the filter acts on monochromatic waves and check if it is a *lowpass* kind:



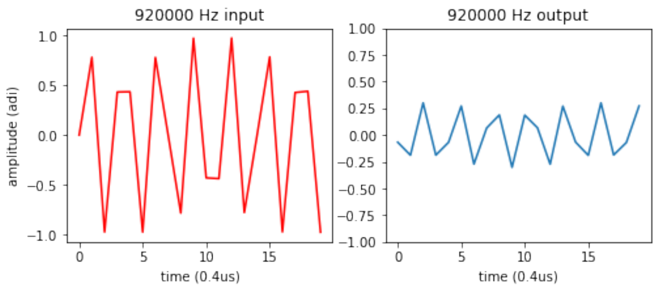


Figura 4: Behaviour of the filter on monochromatic waves.

As we can see the filter acts reducing the amplitude of the high frequencies waves and leaves unaltered the low frequencies waves, as we expected.

The final step of this simulation is to use a weighted superposition of monochromatic waves with different weight and check if the filter neglects the high frequency components. The wave we have chosen is the following one

```
input1 = (1.5*signals[1]+0.1*signals[21]+0.4*signals[39])
```

where the three weights are (1.5, 0.1, 0.4) and the three frequencies are (25 kHz, 540 kHz, 1 MHz). The response of the Python filter is reported in figure 5.

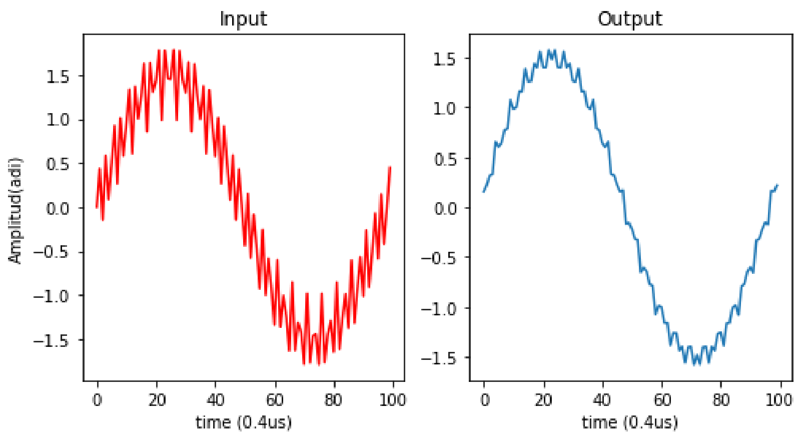


Figura 5: Behaviour of Python-implemented 4-taps FIR filter.

where we can notice that the right figure is smoother than the left one.

4 FPGA behaviour

After checking the expected behaviour of the filter in Python, we use the filter test bench to observe the correct functioning of the VHDL FIR filter. We use as input the previous signal scaled so to have an amplitude range of $[-128, 127]$. Since outcomes are sufficiently close, we connect the FIR filter module with the UART ones so to have a full working machine that receives the input vectors one bit per time, constructs the input value as in integer $n \in [0, 255]$ (where the 2-complement technique was adopted so to solve issues coming from negative signal values), performs filtering operations and returns the output one bit per time. Such VHDL program was tested both directly on the FPGA provided and writing a test bench for the top entity to simulate (full code can be found in given directory). The results of the test bench are aligned with the ones provided by the FPGA: the picture below shows how the input bits are read by the receiver according to its baudrate, sent to the filter and imported into the four-vectors register (named "s_data" below). Moreover one can see how data in the register properly shift. Outputs are synchronized and the final module (transmitter) sends well the output bits to the serial port.

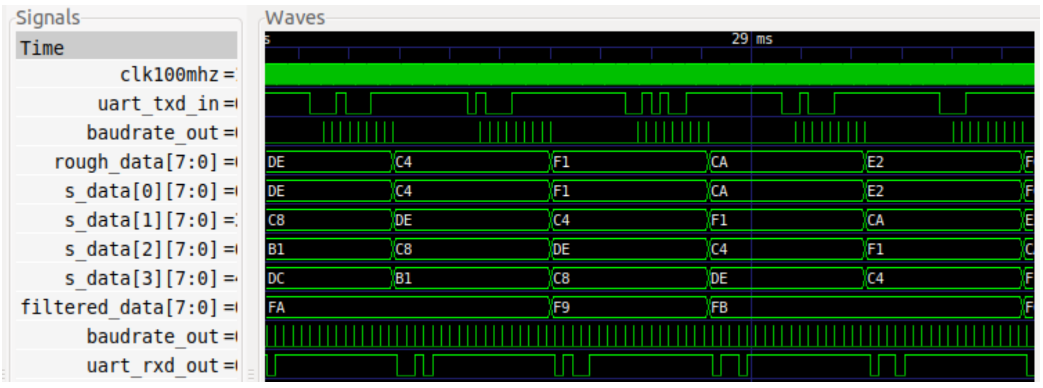


Figura 6: Waveform of top entity test bench.

Given the fact that the filter works linearly combining incoming signals which are 8-bit long, the output of the module cannot be directly synthesized by a 8-bit vector (transmitter' signal size): this leads to an unavoidable truncation of filtered data. This means that data must be later re-normalized so to have the proper output wave to be compared to the expected one. This truncation error makes outcomes slightly different from the results of a Python simulation, which instead works with float numbers and therefore shows better sensitivity. Moreover, the coefficients calculated with Python's *firwin* functions for the low-pass filter cannot directly be converted into integer numbers, as the FPGA requires: this leads to another truncation error in the estimation of coefficients.

Figure 7 shows and compares input and output for the Python/VHDL filters: we can notice that the previsions of Python filter are optimally matched by VHDL filter outcomes, up to the small fluctuations due to truncation effects. The result can be consider satisfactory since even though the amount of taps taken into account offers limited filtering properties the output signals can be seen to be substantially better and neater than the input one. To obtain a better-shaped signal one could eventually repeat the whole procedure.

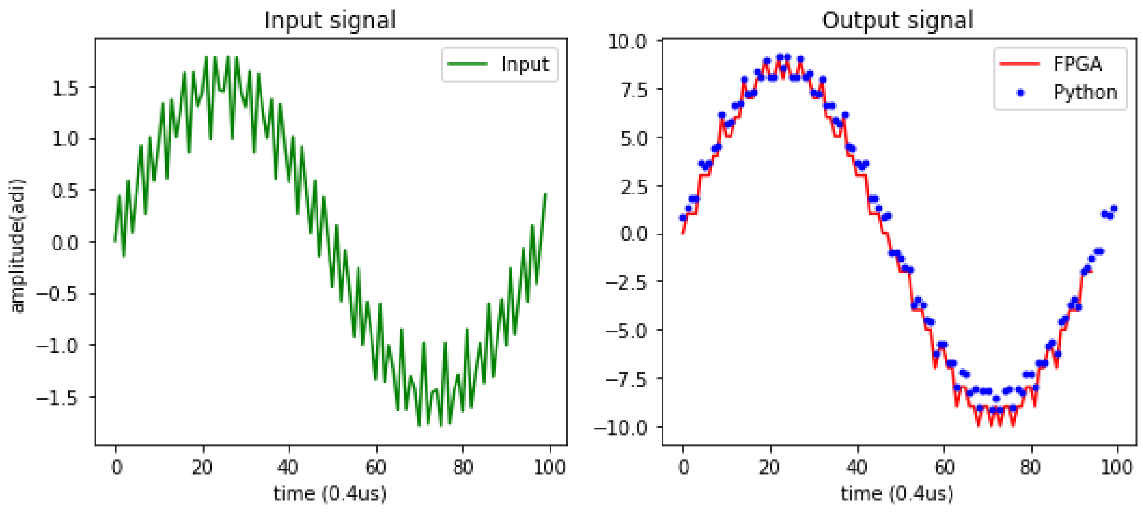


Figura 7: Final output of full machine comparison.

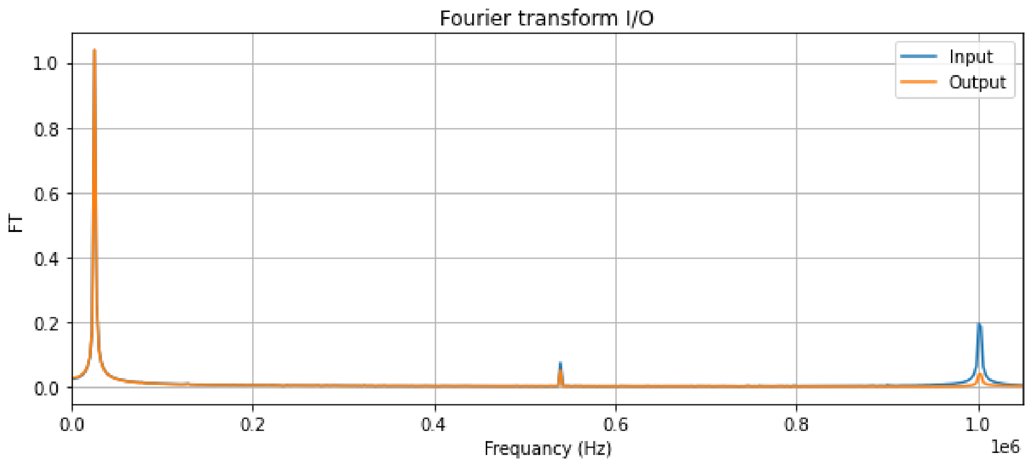


Figura 8: Fourier analysis of input and output.

Finally, the Fourier analysis of incoming and outgoing signals for the filter depicts the standard effects of a low-pass filter: the peak of frequency at 1 MHz is greatly reduced, while the medium-range frequency shows little affection and the low 25 kHz frequency remains steadily dominant. This provides another visible result of correct filtering.

5 Conclusions

Comparison between different VHDL and Python, as well as graphical results, suggest the correct implementation of the FIR filter and UART modules in the FPGA, as well as it efficiency as a low-pass filter.