Implementation of a FIR filter on an Arty7 FPGA board

Federica Benassi, Nguyen Xuan Tung, Alessandro Fella

March 2021

1 Introduction

The required task is about implementing a fully-working FIR filter on an Arty7 FPGA board, along with the input/output data management. Our work focused on:

- the writing, running and testing of the code;
- the comparison of the obtained results with the ones from an analogous FIR filter implemented in a Python script.

2 FIR filter

In the field of signal processing, for a **F**inite **I**mpulse **R**sponse filter we refer to a filter whose impulse response is of *finite* duration, because it settles to zero in finite time. The main characteristic of this type of device is that there is no internal feedback; the output is based on input only, unlike other filter classes.

It is possible, by using the right values for coefficients b_i (*taps*), to implement any kind of filter like high-pass, low-pass and so on. How the filter works can be synthesized by the analytical formula:

$$y[n] = \sum_{i=0}^{N-1} b_i \cdot x[n-i] \tag{1}$$

where

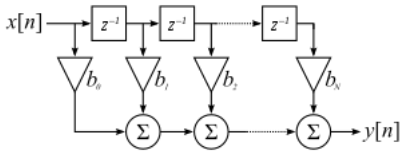

Figure 1: scheme of FIR filter.

- x is the input signal array, and $x[n]$ is the nth element;
- $y[n]$ is the n-th output element;
- N is the filter order (number of taps).

The value of the taps depends on the choice of the cutoff frequencies and on the type of filter we want to implement, and the performance of the filter improves by increasing the number of taps: an ideal filter corresponds to a $(N = \infty)$-order FIR filter (Figure 2).

3 Implementation

We implemented a 8-taps, low-pass FIR filter, assuming:

- a sampling frequency $f_s = 11025$ Hz
- a cutoff frequency $f_c = 551.25$ Hz.

The coefficients have been calculated thanks to the function `firwin` from the `scipy` library of Python: it takes as input the number of taps, the cutoff frequency and the sampling frequency, and it outputs an array with the coefficients.

```
1    N = 8 #taps
2    fs = 11025 #sampling frequency (Hz)
3    fc = 0.1*fs/2 #cutoff frequency (Hz)
4    coefs = sig.firwin(N, fc, fs=fs)
```

The obtained values of the coefficients are:

$$b_0 = b_7 = 0.01740632$$
$$b1 = b_6 = 0.06120739$$
$$b_2 = b_5 = 0.16616441$$
$$b_3 = b_4 = 0.25522188.$$

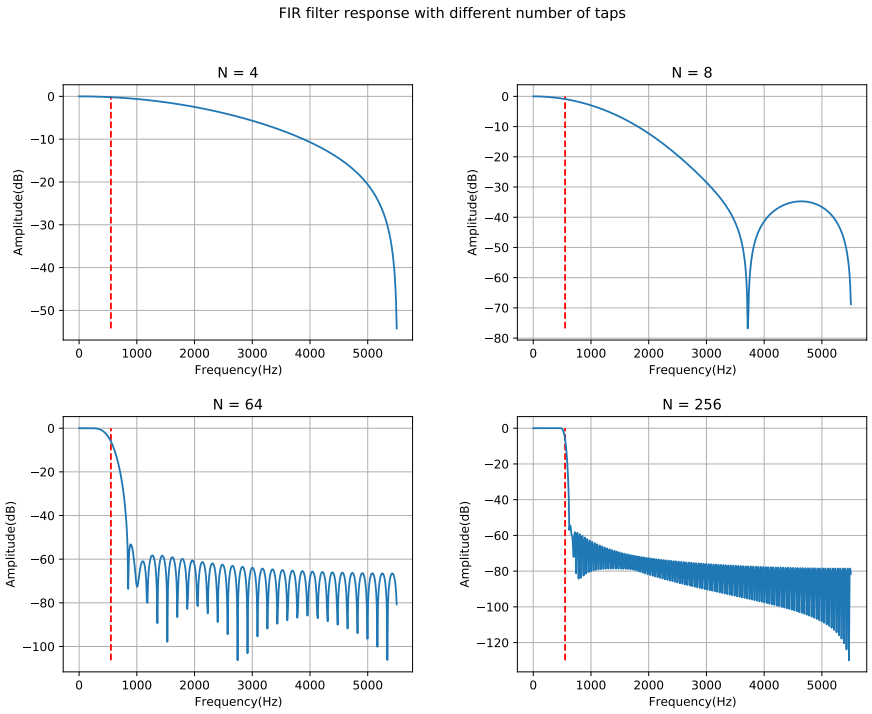

Figure 2: response of FIR filter w.r.t. the number N of taps. The red dashed line represents the cutoff frequency (551.25 Hz).

3.1 VHDL code and implementation on FPGA

3.1.1 Data conversion

The written VHDL code works with 8-bit signed integers, therefore the input numbers have to be converted in binary before being sent to the FPGA. Since the last bit determines the sign of the number, the input data need to be in a range between -128 and 127. The coefficients are smaller than one, so we decided to scale up the numbers by multiplying the values by 2^8. Only the integer part is considered, therefore the coefficients become:

$$b_0 = b_7 = 4$$
$$b_1 = b_6 = 15$$
$$b_2 = b_5 = 42$$
$$b_3 = b_4 = 65.$$

Finally, the coefficients are converted with the VHDL function `to_signed` that takes as input the integer number we want to convert and the length of the string:

```
signal i_coeff_0 : std_logic_vector(7 downto 0) :=
   std_logic_vector(to_signed(4,8));
```

3.1.2 FIR filter code

As the Equation 1 states, the filtered output element $y[n]$ of the input data $x[n]$ also depends on the previous 7 elements $x[n-1]$ ... $x[n-7]$, which need to be memorized. The code performs the following processes:

1. the input elements are stored in an array of 8 standard logic vectors of length 8 called `s_buffer`. The new input data is stored in the 0-th position of `s_buffer`, and all the other elements scale on the following position;

2. the multiplication with the respective coefficients is performed and stored in the array `s_mult`;

3. the elements of `s_mult` are pair-wise summed, and the results are stored in the array `s_add_0`. These new elements are pair-wise summed again and stored in `s_add_1`. This operation is repeated again, and the final sum of all the terms is stored in a 19-bit string associated to the variable `s_add_2`;

4. the final output data is rescaled to 8 bits, taking care of choosing the right bits to preserve the most information.

3.1.3 Run the code inside the FPGA

In order to send data to the FPGA and receive the filtered outputs, the FIR filter needs to be connected with a UART receiver and a UART transmitter. The baudrate at which the data are sent is 115200 bit/s, while the FPGA clocks ticks at a 100 MHz frequency.

The input data are read from the `input.txt` file, converted into Unicode characters and sent to the FPGA with a Python script, that also applies the 2-complement notation. The same script reads the filtered output data and writes it to the `output.txt` file.

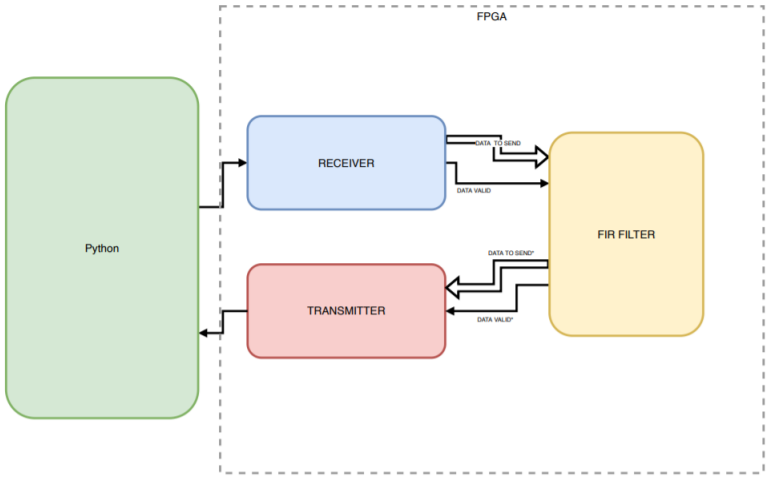

Figure 3: workflow of the data.

The UART receiver sends the data to the FIR filter with an additional signal called `raw_data_valid` that allows the FIR filter to start to compute. The filter sends a `fil_data_valid` to the transmitter when the output is correctly processed.

3.2 Implementation on Python

Referring to Equation 1, we wrote a Python function which computes the filtered data from the coefficients and the input data.

```
def python_fir_filter(input_data, coefs):
  filter_output = np.zeros(len(input_data))
  for n in range(len(input_data)):
      for i in range(N):
        filter_output[n] += int(coefs[i]*input_data[n-i])
  return filter_output
```

4 Results and analysis

We tested the performance of the FIR filter on random noise and on a soundtrack, comparing the results with the Python script. Also a frequency analysis has been performed.

4.1 Waveforms

4.1.1 Random noise

We tested the FIR filter with an input random noise wave (sequence of random numbers between -128 and 127).

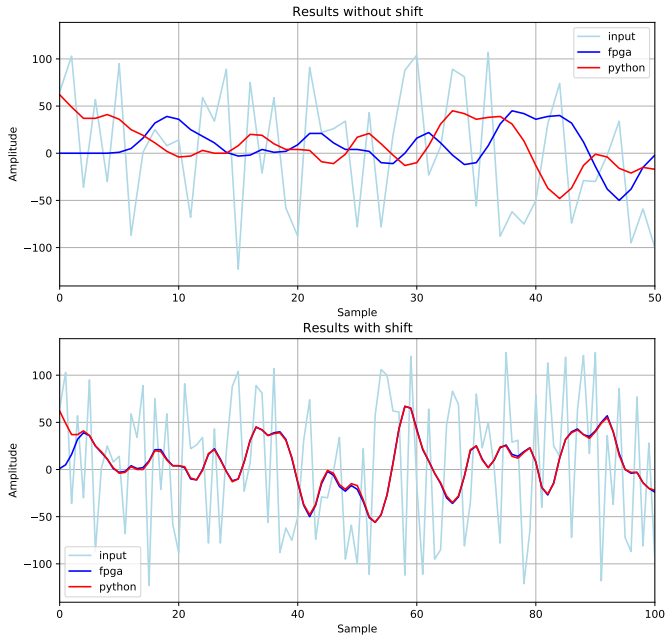

Figure 4: comparison of the filtered results with random noise input.

The output of both FPGA and Python is shown in figure 4. As it can be seen from the first panel, the output of the FPGA follows the same pattern of the one of Python, but it is shifted. This comes to the fact that the VHDL code is such that the sum is not performed at a single time, but it takes several steps: therefore, the first output elements will be zeros. This also explains the different behavior of the scripts in the first samples: moreover, Python automatically takes the last elements of the array when the index is negative.

4.1.2 *Star Wars* soundtrack

In order to see a more realistic application of the filter we also tested it on the *Star Wars* soundtrack. We used the `wave` Python library to import the `.wav` file of some seconds of the soundtrack as a number array. Then we reduced the amplitude of the signal of an empirical factor of 1.8×10^7 and converted the elements into integer numbers. With this procedure it has been possible to send the input file to the FPGA, even though the audio quality has been compromised. The final result can be seen in the plot 5.

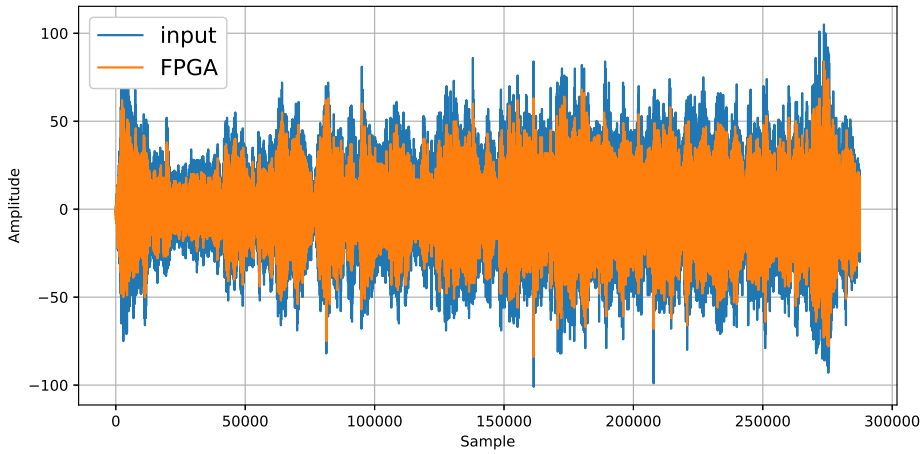

Figure 5: response of implemented FIR filter on FPGA using some seconds of soundtrack

4.2 Frequency analysis

The figure below shows the spectrum analysis of the outputs obtained. As it can be seen, the amplitude of the signal actually decreases for frequencies greater than the cutoff.

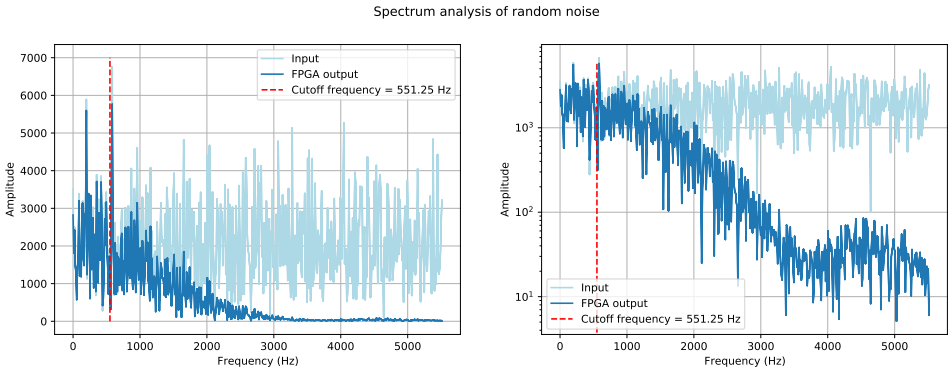

Figure 6: spectrum analysis of random noise.

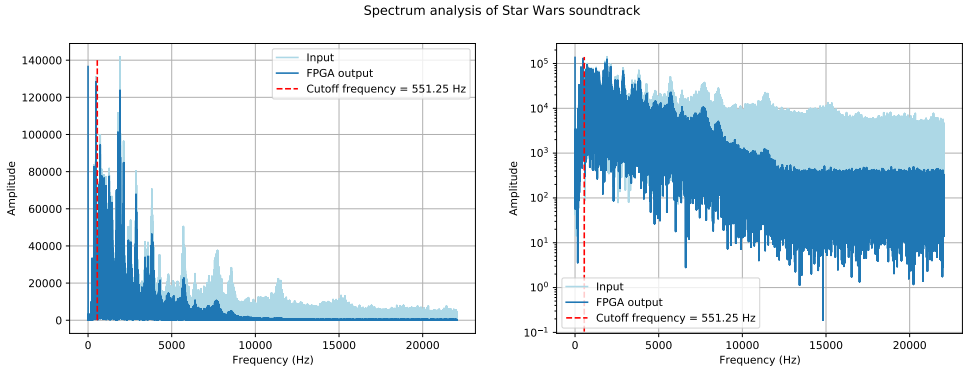

Figure 7: spectrum analysis of *Star Wars* soundtrack.

5 Conclusion

We implemented and tested a FIR filter on an Arty7 FPGA board, and compared the results with an analogous Python implementation. The behavior of the filter reflects what expected: the effectiveness and the similarity with an ideal low-pass filter can be improved by increasing the number of coefficients.