Implementation of a Fir filter on an FPGA board to be used remotely.

Luca Rinaldi, Luca Negri, Rebecca Ghidoni

February 2021

1 Introduction

The aim of this project was to implement a FIR filter programmed in VHDL on an Arty7 FPGA board (fig. 1) operated remotly. To confirm the effectiveness of the filter we compared the results obtained with the FPGA with the ones computed through a Python script.

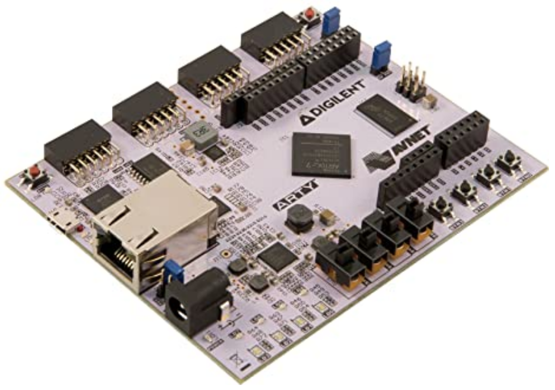

Figure 1: *An FPGA similar to the one used in our project.*

2 FIR filter

The feature that gives **F**inite **I**mpulse **R**esponse (**FIR**) filters their name is that their output settles to zero in a finite amount of time (in contrast to **I**nfinite **I**mpulse **R**esponse filters). These are really useful in digital filtering of signals, since a FIR filter can be implemented to have the behaviuor of any kind of filter (high-pass, low-pass, band-pass ...) and takes a finite amount of operations to complete its job.

As shown in fig. 2, given an input signal x and a vector of coefficients b of lenght N, the output y of the FIR filter is calculated by

$$y_i = \sum_{k=0}^{N-1} b_k x_{i-k} \tag{1}$$

The number of taps (N) and the coefficients b are what determines the filter's behaviour.

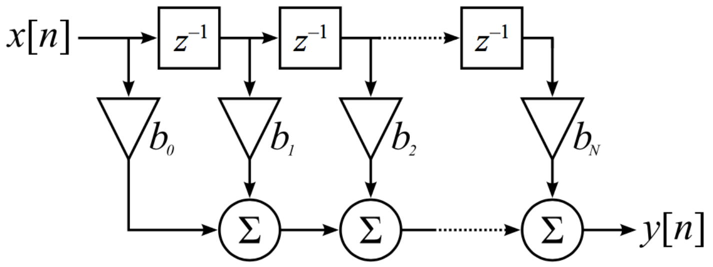

Figure 2: *FIR Filter design.*

3 Implementation in Python

To check if our FIR filter is working correctly, we have also written one in Python. To get the coefficients we need to operate the filter we used the `scipy.signal` function `firwin`, wich is also used to obtain the coefficents needed to its VHDL counterpart.

```python
import scipy.signal as sg
import numpy as np

N=1000
X,Y=np.genfromtxt('input.txt'),np.zeros(N)

cutoff=550
sampl=11025
taps=8
C=sg.firwin(taps,cutoff,fs=sampl)

for i in range(N):
    for j in range(taps):
        Y[i]+=C[j]*X[i-j]
```

Listing 1: *FIR in Python*

This filter is operated with 8 taps and simulates a low-pass filter operating on a signal sampled at 11025 Hz with the cutoff frequency at ≈ 550 Hz .

4 Implementation in VHDL

4.1 Input data

The FIR filter coefficients are real numbers, but our VHDL code is implemented to work with 8-bit signed integers. To translate the coefficients into VHDL we opted to look at the binary representation of the coefficients and then take the first 7 bits after the first `1` of the biggest coefficient, the first bit needs to be left blank otherwise the code will think it's a negative number. This operation is equal to multipling all the coefficinets to a certain power of two, 2^8 in our case, and then translating the integer part in binary.

These are the coefficients we used and their binary rappresentation:

b_0	0.017	`00000100`
b_1	0.061	`00001111`
b_2	0.166	`00101010`
b_3	0.255	`01000001`
b_4	0.255	`01000001`
b_5	0.166	`00101010`
b_6	0.061	`00001111`
b_7	0.017	`00000100`

The input signal of the FIR filter needs to be a string of 8-bit signed integers also. To represent the signal with the most precision we scale every input to occupy all of the -128 to 127 range before feeding it to the FPGA. The signal we fed to the FPGA were random noise and a square wave, both signals that span the entire frequency domain.

4.2 UART receiver and transmitter

The bits of the data are sent throught the pins to the FPGA in series, at a specific rate, while the FIR filter expects 8-bit data in parallel. We thus need a UART receiver and a UART transmitter to convert the data between these two modes. In figure 3 we see the components and their connections.

The baudrate at wich the data is sent is 115200 bit/s while the FPGA clock is ticking at 100 MHz. This means the FIR filter needs to be slowed down to mactch the UART receiver output rate. To do this we linked the `data valid` line of the UART receiver to the FIR clock. Now the filter processes the data at the same rate at wich it receives it, and it gives in output a new processed sample for each one it receives in input.

The transmitter is also synced to the receiver, but it reads the data from the FIR filter one clock cycle (10 ns) later. This is done to avoid reading the FIR output while the filter is still computing the results.

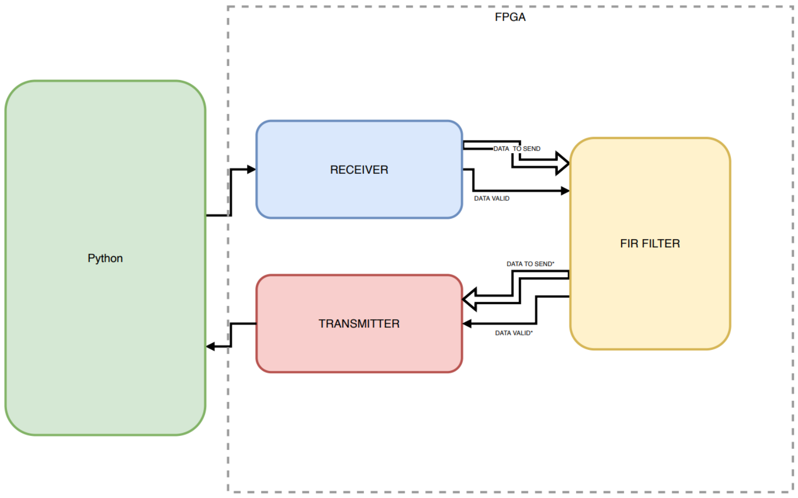

Figure 3: *Project entity's connection.*

4.3 Data handling inside the FIR

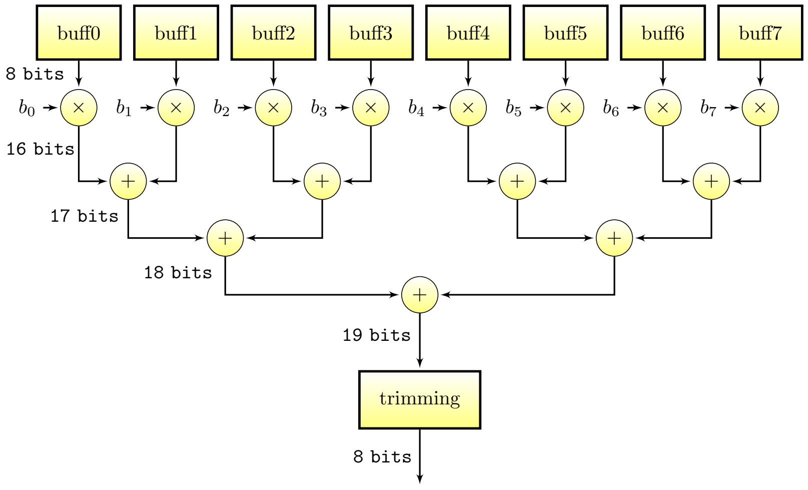

Figure 4: *8-taps FIR filter architecture.*

The figure 4 reports the 8-taps FIR filter architecture where the input buff_i and the coefficient b_i are 8-bit signed integers. After the filter coefficients multiplication, the product will be a 16-bit signed signal to avoid overflows, following the multiplication rule:

$$(M\text{-}bit) \times (N\text{-}bit) = (M + N)\text{-}bit \tag{2}$$

Instead, when we perform an addition, the number of bits of the result will be incremented by 1. After the last addition we have 19-bit signed signal that will be rescaled to fit the 8-bit output of the FIR filter with this line of code:

```
o_data <= std_logic_vector(s_add_2(18)&s_add_2(14 downto 8));
```

As it can be seen we keep the most significant bit to preserve the right sign, then we take the first 7 bits starting from the 14th. We can do this because we saw that with our coefficients, the 17th,16th and 15th bit are always `000` and `111` (if the number is respectively positive or negative).

4.4 Python script to manage the dataflows

The Arty7 FPGA board is connected to the server *lxilinx1* and the board itself exposes two USB devices to the server, in our case `ttyUSB8` and `ttyUSB9`. The former is devoted to download the bitstream while the latter is used to connect to the board serial port.

The port names of the top level entity are assigned to the correspondent pins in the mapping file `pins.tcl` trough the statements:

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [
    get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=gclk[100]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [
    get_ports { CLK100MHZ }];

## USB-UART Interface
set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33 } [
    get_ports { uart_rxd_out }];
set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33 } [
    get_ports { uart_txd_in }];
```

In order to send and receive data trough the serial port we exploited a Python script(lis. 2) utilizing a specific routine, `Serial`, from the `serial` package.

```
import serial

ser = serial.Serial('/dev/ttyUSB9', baudrate=115200)
f_in  = open('input.txt','r')
f_out = open('output.txt','w')

for line in f_in:
    if (int(line)<0):
        ser.write(chr(256+int(line)))
    else:
        ser.write(chr(int(line)))
    d = ser.read()
    if(ord(d)>127):
        g=ord(d)-256
    else:
        g=ord(d)
    f_out.write(str(g)+'\n')
```

Listing 2: *I/O script.*

As shown in line 3 we first enstablish a connection to the serial port specifying the baud rate, then we open the input and output file. For every line in `input.txt` the script reads the data and send it to the UART receiver through the serial port, taking care to convert the input to the corresponding integer considering the 2-complement notation (`if` statement of line 8).

The script receives the output data from the UART transmitter and writes it to the `output.txt` file rescaling it to the usual range $[-128, 127]$ (`if` statement of line 13).

5 Analysis of results

5.1 Python-FPGA comparison

As we can see in fig. 5 There is almost no difference between the results of the Python filter and the one implemented on the FPGA. the approximation made by cutting the output bits is the most significant source of error, but it's still rather small.

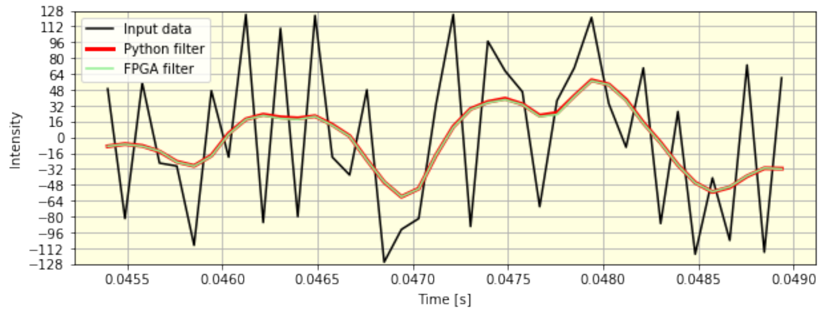

Figure 5: *Confrontation between Python and FPGA filter results*

Another difference comes from how the samples are shifted. FIR filters normally tend to shift the data to the right by a few samples, but we also need to consider that our FPGA implementation has a bit of extra shifts due to the samples taking 3 FIR clock cycles to make their way up the adder. To make the graphs align we need to shift the data to the right position ourselfs. This shift also means that the first few samples are polluted by the data that is present in the buffer and the adder at inizialization (almost always 0s). We do not consider the first samples of each run in this analysis.

5.2 Waveforms

We see in fig. 6 the whole wave and the filtered conterpart in the case of the noisy wave and the square wave. In fig. 7 we have a zoomed in part of the square wave to highlight the effects that a low-pass filter has on this type of signal.

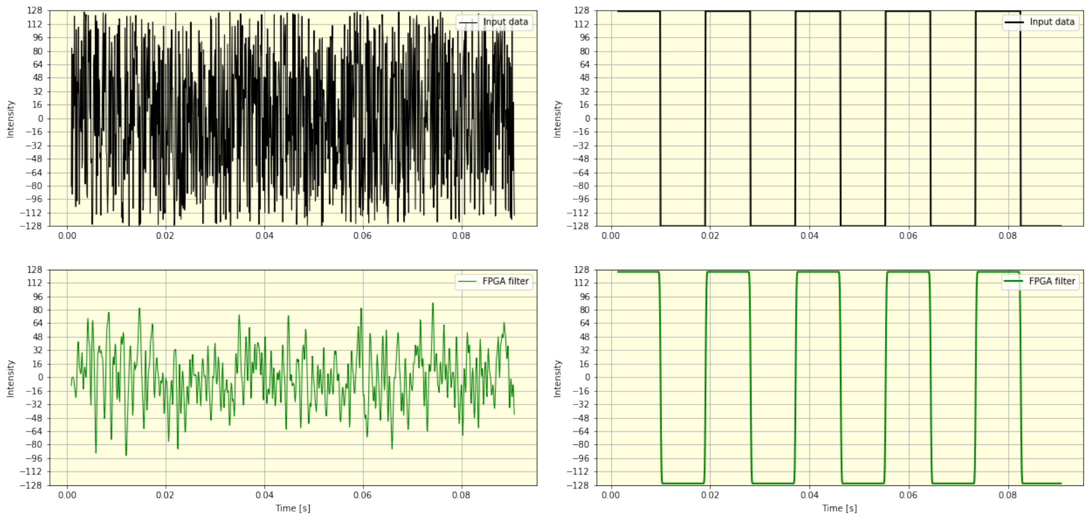

Figure 6: *Input waveforms and the filtered conterparts, on the left we have the noise waveform while on the right the square wave.*

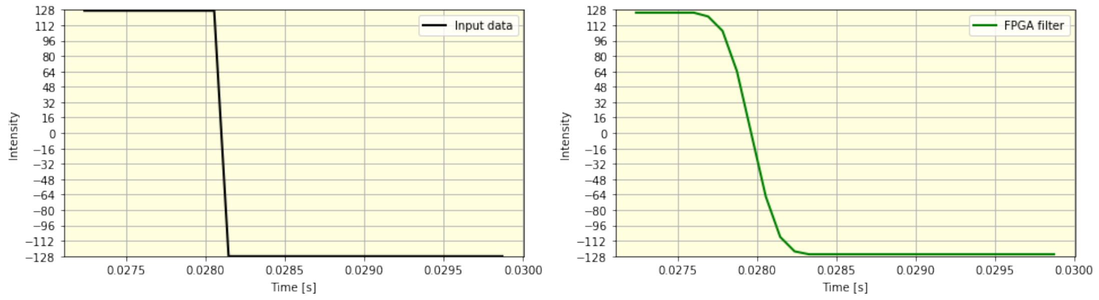

Figure 7: *Comparison between input square waveform and FPGA filter results, zoommed at the edge to better appreciate the differences.*

5.3 Spectrum analysis.

We have plotted in figure 8 , for the noise wave, and figure 9, for the square wave, the spectrum in logarithmic scale of the input and filtered signal. We can see that after the cutoff frequency there is an attenuation of the intensity, as we expected.

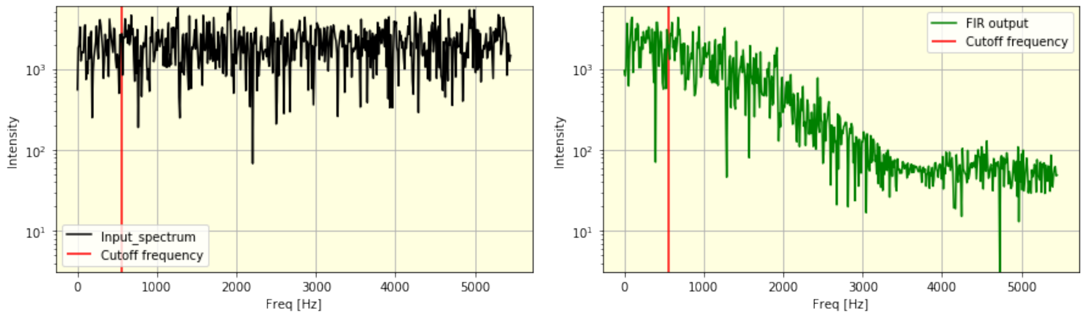

Figure 8: *Spectral analysis of the noise wave input file and the FIR output*

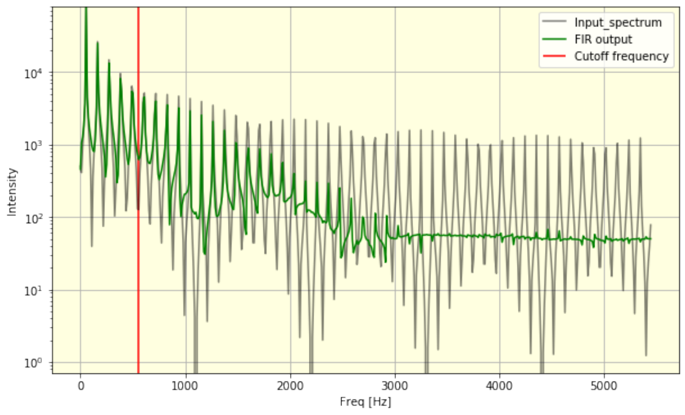

Figure 9: *Spectral analysis of the square wave input file and the FIR output*

6 Conclusion

In this project we succesfully implemented a FIR filter, working as a low-pass filter, on an Arty7 FPGA board and operated it succesfully. The comparison with the same filter implemented in Python showed that the code on the FPGA is working correctly. To improve the effectiveness of the filter in the future we could insert more taps to improve precision.