Low Pass FIR Filter in FPGA
Management and Analysis of Physics Datasets - MOD. A

Saverio Monaco, Gerardo Carmona
2012264 ???????

***Abstract*—FIR Filters are a class of filters that are relatively easy to implement in a FPGA. They can be pretty versatile since they can comprehend Low Pass, High Pass, Band Stop and Band Pass filters. In this project we present an example of a Low Pass filter.**

I. FIR Filter

A Filter takes as an input a signal $X[n]$ and outputs a refined one $Y[n]$, for the case of a FIR Filter, the law $X[n] \to Y[n]$ is described as follows:

$$Y[n] = \sum_{i=0}^{N} C_i x[n-i]$$

where N is the filter order. This order is totally arbitrary, the higher the order is, the better the filter performs. In this project, we choosed to implement a 3 order filter (4-Taps FIR Filter). Our transformation law is then

$$Y[n] = \sum_{i=0}^{3} C_i x[n-i] = \\ = C_0 x[n] + C_1 x[n-1] + C_2 x[n-2] + C_3 x[n-3]$$

The circuit able to implement this is the following:

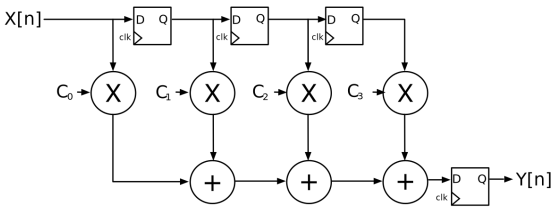

Fig. 1: Circuit for a FIR Filter

The component represented by a square is a *Flip-flop* (necessary to store the past 3 input for the weighted sum), while ⓧ and ⊕ represent respectively the operations of multiplication and addition.

The coefficients

The value of the coefficients depends on what type of filter do we want to implement and which value of frequencies do we want to filter out. The values of the coefficients can be computed with *Python* using the function `firwin` in Scipy:

```
from scipy import signal

numtaps = 4
f = 3

signal.firwin(numtaps, f, fs=125)

[0.04673608 0.45326392 0.45326392
   0.04673608]
```

The coefficients in output will make us build a low pass filter with a cut-off frequency of 3 Hz for a sampling frequency of 125 Hz

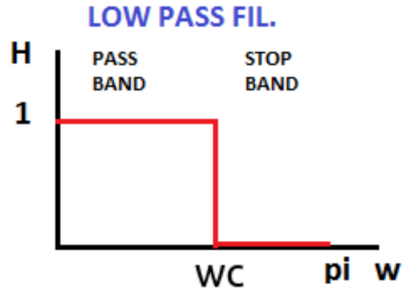

Fig. 2: Low Pass filter

II. Implementation of a FIR Filter in VHDL

We cannot directly implement the coefficient just found since they are not integers, one way to solve it is to multiplying them before the filtering process and rescaling consequently the output signal. For example:

$$20 \times 0.75 = 15$$

In binary terms it means

$$10100_2 \times 0.11_2 = 1111_2$$

To realize this, we must scale the quantity 0.11_2 by multiplying it by 2^Q (resulting in a shift of bits) until we obtain an integer, then after the multiplication we can rescale back (shifting to the other direction).

$$0.11_2 * \times 2^3 = 0.11_2 << 3 = 110$$

Now we can do the multiplication

$$10100_2 \times 110_2 = 1111000_2$$

Then we rescale back

$$1111000_2 >> 3 = 1111_2 = 15$$

In practice, we needed to scale the coefficie and write them in hexadecimal:

```
c = signal.firwin(4, 3, fs=125)
rc = c * 2**8
hex_rc = []

for i in range(numtaps):
hex_rc.append(hex(trunc_rc[i]))

print(rc)
print(hex_rc)

[ 11.964 116.035 116.035  11.964]
['0xc', '0x74', '0x74', '0xc']
```

The algorithm

When you multiply two numbers of N-bit and M-bit the output dynamic of the multiplication result is (N+M)-bits.
When you perform addition, the number of bit of the result will be incremented by 1.
This means that, considering the coefficients being 8 bits wide, if the input signal is 8 bits, the output signal should have 18 bits.
In order to write a code able to compute this operation:

$$Y[n] = \sum_{i=0}^{N} C_i x[n-i]$$

We implemented 5 processes in it:

- **p_input** (initialization): The new value is stored in X[N] and the old ones are shifted one place to the right
- **p_mult** (multiplication): All the for values X[i] are multiplied by their coefficient C_i
- **p_add_0** (first additions): The additions $X[N] \times C_0 + X[N-1] \times C_1$ and $X[N-2] \times C_2 + X[N-3] \times C_3$ are performed and stored paying attention in avoiding overflows
- **p_add_1** (final addition): The values of the last process are added together as always paying attention to overflows
- **p_output** (output): The 18 bits output is resized to 8 (same as the input)

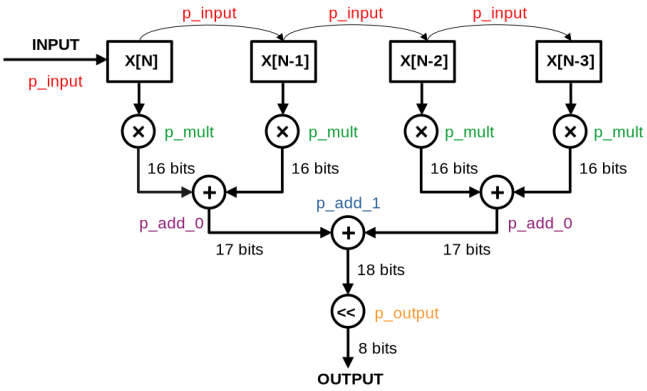

Fig. 3: Algorithm of FIR Filter

III. UART IMPLEMENTATION

A UART is one of the simplest hardware devices capable of transfer data from and to the FPGA. To set up a UART correctly, the device and the FPGA must agree on how quickly the data is transfered, this speed is called *Baud rate*. The data stream is the following:

Fig. 4: UART communication

By default the *Idle state* (the state where the transmitter is waiting for data to send) is `HIGH`, the *start bit* is `LOW`, no information is transfered with the start bit, it just tells that some data will be transfered after. Then the bits will be transfered, obviously they can be either 0 or 1, then the end of the stream is announced by the *stop bit* that is `HIGH`

The complete project can be schematized as it follows:
Inside the FPGA we will have a *receiver* that will receive data from Python through a USB, the data will be sent to the FIR Filter, it will be filtered and then sent to the transmitter, from there Python can read the output.
For this project we need then 3 main entities: the Fir Filter (already explained), the Receiver and the

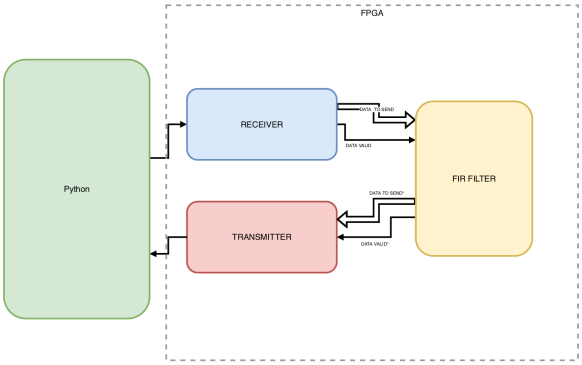

Fig. 5: FPGA structure

Transmitter (already provdided). These 3 entities are managed by another entity *top*:

```
begin  -- architecture str
-- This is the core process that manages the interaction
-- between UART and FIR:
-- First we want to use the receiver to receive data:
  uart_receiver_1 : uart_receiver
    port map (
      clock            => CLK100MHZ,
      uart_rx          => uart_txd_in,
      valid            => data_valid,
      data_from_python => unfiltered_data);

-- Then we want to use the Filter:
  fir_filter_1 : fir_filter_4
    port map (
      clk       => CLK100MHZ,
      nxt       => data_valid,
      rst       => i_rstb,
      valid_out => data_valid_fil,

      data_in   => unfiltered_data,
      -- filtering here
      data_out  => filtered_data);

-- Finally we want to transmit back out filtered data:
  uart_transmitter_1 : uart_transmitter
    port map (
      clock          => CLK100MHZ,
      data_to_python => filtered_data,
      data_valid     => data_valid_fil,
      busy           => busy,
      uart_tx        => uart_rxd_out);
```

IV. Results

The results shown are for a signal consisting of two frequencies: 10 Hz and 60 Hz:

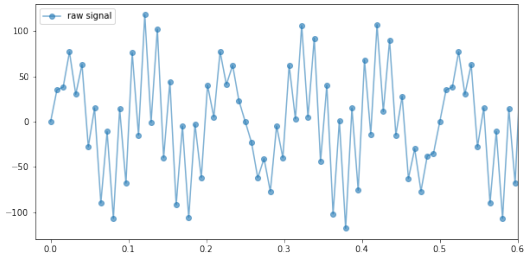

Fig. 6: Raw signal

Implementing a Filter with a cutoff frequency of 30 Hz we get rid of the higher frequency (the noise), obtaining the following result:

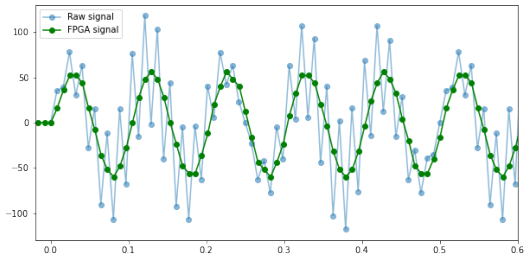

Fig. 7: FPGA performance

As we can see the output from the FPGA is a signal of a single frequency of 10 Hz as expected. We can also make a comparison between the output from the FPGA and a simulated FPGA in Python using the function

`python_sig = lfilter(c, 1, wave)`

Where c is the matrix of the coefficients, wave is the raw signal input:

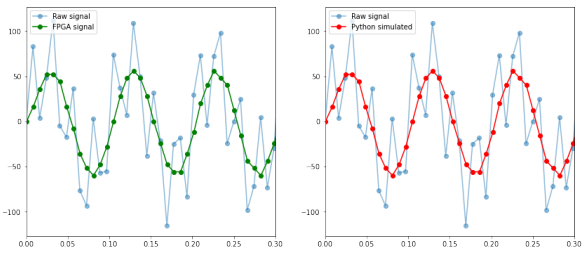

Fig. 8: Comparison with Python

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.