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

Saverio Monaco, Gerardo Carmona
2012264 2005005

***Abstract*—FIR Filters are simple, yet effective filters which can be implemented in an FPGA. They may function as Low Pass, High Pass, Band Stop or Band Pass filters. We present a Low Pass FIR filter implementation which is connected to a UART source, fed by a simulated wave signal.**

I. FIR Filter

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

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

where N is the *filter order*. The filter performance improves as the order is increased. For this project, we have chosen 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 filter described above has the following circuit representation:

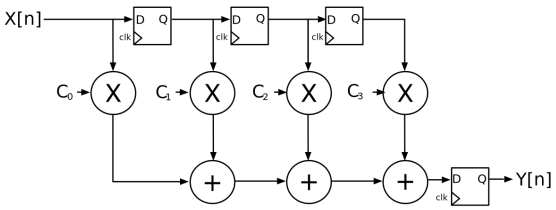

Fig. 1: Circuit for a FIR Filter

The square holds the place of a *flip-flop* (necessary to store the past 3 inputs for the weighted sum), whereas the symbols ⓧ and ⊕ correspond to the multiplication and addition operations, respectively.

The coefficients

The value of the coefficients depends on the type of filter we wish to implement and the range of frequencies we aim to filter out.
These values can be computed with the function `firwin` from the *python* package `scipy`:

```
from scipy import signal

numtaps = 4
f = 30

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

[0.0187 0.4812 0.4812 0.0187]
```

The coefficients so obtained will be used to build a low pass filter with a cut-off frequency of 30 Hz for a sampling frequency of 125 Hz

Fig. 2: Low Pass filter

II. Implementation of a FIR Filter in VHDL

Although these coefficients would ideally be implemented directly, the FPGA is unable to process floating point values. In order to overcome this, we proceed in the typical fashion of scaling the values and rounding them up once sufficient significant digits are guaranteed.

For example:

$$20 \times 0.75 = 15$$

In binary this is translated to:

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

To realize this, we must scale the quantity 0.11_2 through multiplication by 2^Q (resulting in a shift of bits) until we obtain an integer (or until we arbitrarly decide to truncate), 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 carry the multiplication

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

then we rescale back

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

In practice, we scaled the coefficients and wrote them in hexadecimal:

```
c = signal.firwin(4, 30, 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)

[0.0187 0.4812 0.4812 0.0187]
['0x5', '0x7b', '0x7b', '0x5']
```

The algorithm

The product of an N-bits number and an M-bits one results in an (N+M)-bits number.
When addition is performed, the number of bits in the result will be incremented by 1.
Since the coefficients are 8 bits long, and the input signal is 8 bits long, the output signal should have 18 bits (see Fig. 3).
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): Every value X[N-i] is multiplied by the 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, with due care to avoid overflows
- **p_add_1** (final addition): The values of the last process are added together.
- **p_output** (output): The 18 bits output is resized to 8 (same as the input)

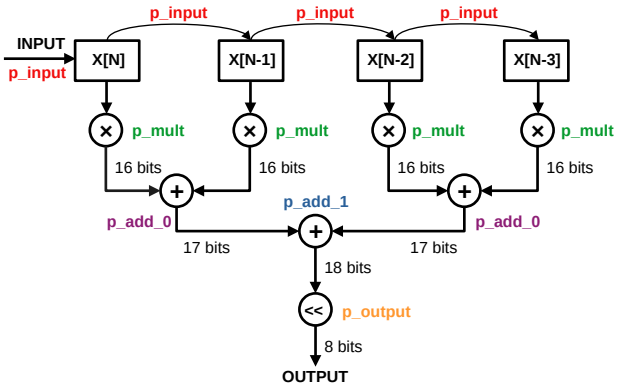

Fig. 3: Algorithm of FIR Filter

III. UART IMPLEMENTATION

The UART is among the simplest devices capable of data exchange with 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* (in which the transmitter is waiting for data) is `HIGH`, the *start bit* is `LOW`. No information is transfered with the start bit, it just states that data will be transfered. Then the bits will be transfered, until the end of the stream is announced by the *stop bit*, that is, a `HIGH` signal is received.

The complete project can be schematized as it follows:

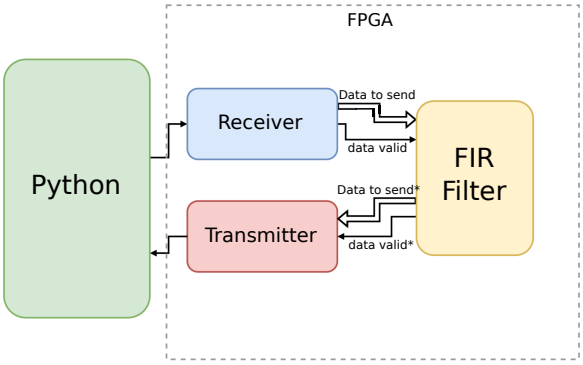

Fig. 5: FPGA structure

Inside the FPGA there is a *receiver* that obtains data from a python script through a USB. The data is sent to the FIR Filter, where it is processed and then sent to the transmitter. Once there, the output can be read by python.
For this project we need therefore three main entities: the FIR Filter, the Receiver and the Transmitter. All these entities are managed by another entity, named *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 correspond to an input signal of two frequencies, 10 Hz and 60 Hz:

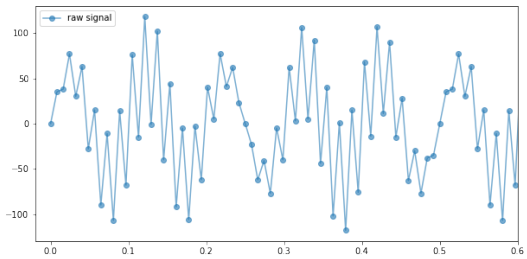

Fig. 6: Raw signal

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

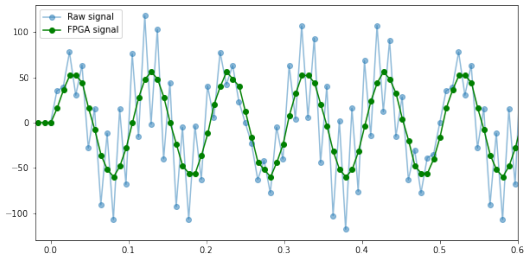

Fig. 7: FPGA performance

The output of the FPGA is a single frequency signal of 10 Hz, as expected. We can also compare the output from the FPGA and an FPGA simulation in Python using the function

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

Where `c` is the array of the coefficients and `wave` is the noisy signal input:

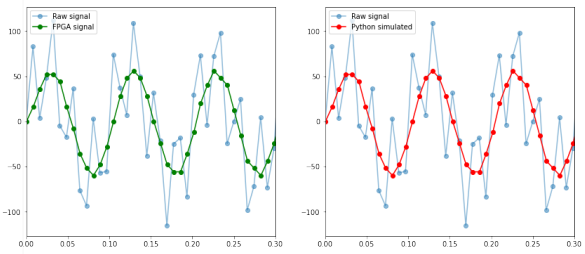

Fig. 8: Comparison with Python

V. Conclusion

We implemented a four taps FIR filter, which successfully filtered the higher frequency signals

from the simulated wave signal. We used a scaled, rounded-up value of the coefficients. Since results from the server FPGA optimally match the ones simulated in Python, we conclude that the FIR Filter has been properly implemented and that it efficiently simulates a low-pass filter.