MAPD - Lab report

Study of the implementation of a FIR filter on FPGA

Conforto Filippo - 2021856
Domenichetti Lorenzo - 20116534
Faorlin Tommaso - 2021857

Table of Contents

List of Figures

1. Goal

Design of a Finite Impulse Response (FIR) filter both in VHDL and in a programming environment (*python*). Compare the results from the two implementations. Build a FIR filter having an arbitrary number of "taps" and an arbitrary behaviour in the frequency domain (i.e. low-pass, high-pass, band-pass, notch). Samples data width will be 8, equal to the processed input size.

2. Overview

The diagram below shows the scheme of the designed project.

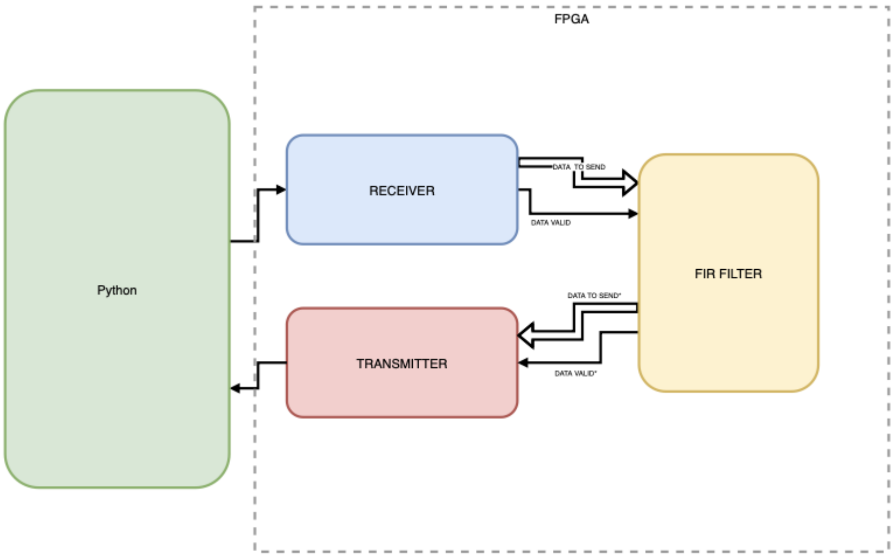

Fig. 1. Block diagram of the circuit.

A *Python* interface sends the input data to the FPGA board, where it is read through an USB UART receiver. Then, the receiver sends the input to be processed to the filter. Finally the data is collected by the USB UART transmitter that sends back the collected data to the *Python* code, where it is then printed to an output file.

2.1 FIR filter in a nutshell

The FIR filter behaviour is displayed in the following figure. Basically, the output value of the filter is the weighted sum of the most recent input values. As the input comes in, also the previous input values get shifted accordingly. A FIR filter is characterised by the number of **taps**, namely the number of multiplications, and so the number of coefficients within the filter. This property leads the filter to have a finite response to a impulse (a Kronecker delta input).

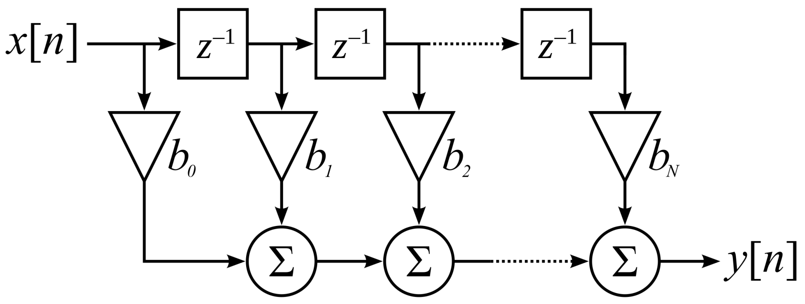

Fig. 2. Fir Filter behaviour.

3. FIR filter on VHDL

We implemented a low-pass FIR filter that applies the coefficients to the input arrays - 4 or 5 in our case - returning a final output value. The filter was built exploiting a five stages state machine. The process has the following states:

idle When a *valid* signal is received, the first three data are shifted rightward and the last one is discarded. Finally, the vacant *signed* is occupied by the incoming data;

mult Then the machine multiplies each value in the array of *signed* by the correspondent coefficient, and stores the result in the *v_mult* array;

sum1 The multiplication results are summed two at a time;

sum2 The second sum adds finally the two previously obtained intermediate sums;

outp Stage used only to send the final value to the transmitter module.

Here we show the actual code for a 4-taps filter implementation, excluding the port map.

```vhdl
ARCHITECTURE rtl OF fir_filter IS
    TYPE t_coeff IS ARRAY (0 TO 3) OF signed(7 DOWNTO 0);
    TYPE t_mult IS ARRAY (0 TO 3) OF signed(15 DOWNTO 0);
    TYPE data_pipe IS ARRAY (0 TO 3) OF signed(7 DOWNTO 0);

    TYPE state_type IS (idle, mult, sum1, sum2, outp);

    SIGNAL state : state_type := idle;
    SIGNAL data : data_pipe := (OTHERS => (OTHERS => '0'));
    SIGNAL v_mult : t_mult := (OTHERS => (OTHERS => '0'));
    SIGNAL v_add_0 : signed(16 DOWNTO 0) := (OTHERS => '0');
    SIGNAL v_add_1 : signed(16 DOWNTO 0) := (OTHERS => '0');
    SIGNAL result : signed(15 + 2 DOWNTO 0) := (OTHERS => '0');
```

We have defined useful data types and signals for the internal calculations.

```
BEGIN

    processing : PROCESS (i_rstb, i_clk) IS
        -- coefficient
        VARIABLE coeff_0 : STD_LOGIC_VECTOR(7 DOWNTO 0) := X"0C";
        VARIABLE coeff_1 : STD_LOGIC_VECTOR(7 DOWNTO 0) := X"73";
        VARIABLE coeff_2 : STD_LOGIC_VECTOR(7 DOWNTO 0) := X"73";
        VARIABLE coeff_3 : STD_LOGIC_VECTOR(7 DOWNTO 0) := X"0C";
        VARIABLE v_coeff : t_coeff := (signed(coeff_0), signed(
    coeff_1), signed(coeff_2), signed(coeff_3));
```

The FIR filter coefficients are integer values hard-coded inside the main process as variables. These are obtained with a 8-bit left shift of the original coefficients, since it is not possible to code them as fractions.

```
    BEGIN

        IF (i_rstb = '0') THEN
            state <= idle;
            v_mult <= (OTHERS => (OTHERS => '0'));
            result <= (OTHERS => '0');
            data <= (OTHERS => (OTHERS => '0'));
            v_add_1 <= (OTHERS => '0');
            v_add_0 <= (OTHERS => '0');
            o_data <= (OTHERS => '0');

        ELSIF (rising_edge(i_clk)) THEN
            CASE state IS
                WHEN idle =>
                    o_valid <= '0';
                    IF i_valid = '1' THEN
                        data <= signed(i_data) & data(0 TO data'
    length - 2);
                        state <= mult;
                    END IF;
                WHEN mult =>
                    FOR k IN 0 TO 3 LOOP
                        v_mult(k) <= v_coeff(k) * data(k);
                    END LOOP;

                    state <= sum1;
                WHEN sum1 =>
                    v_add_0 <= resize(v_mult(0), 17) + resize(v_mult
    (1), 17);
                    v_add_1 <= resize(v_mult(2), 17) + resize(v_mult
    (3), 17);

                    state <= sum2;
                WHEN sum2 =>
                    result <= resize(v_add_0, 18) + resize(v_add_1,
    18);

                    state <= outp;
                WHEN outp =>
                    o_valid <= '1';
                    o_data <= STD_LOGIC_VECTOR(result(17 DOWNTO 10))
    ;
                    state <= idle;
            END CASE;
        END IF;
    END PROCESS processing;
END rtl;
```

The final output is the vector containing the 8 MSB from a 18 bits vector. 8 bits are truncated to get the correct result, given the coefficient representation, and 2 bits are truncated in order to transmit the signal through the UART.

3.1 UART transmission

UART transmitter and receiver were built as the ones used during lessons.

4. Python interface

In order to send and receive values from the fir filter we used a *python* script exploiting the `pyserial` library. The script establishes a connection to the serial port and sends the input values in binary representation.

Since the input values may be negative, we have to transform them using the correspondent signed representation (two's complement). In doing so, a positive input number is sent directly to the receiver, while for negative ones the two's complement is applied to their absolute value.

```python
for line in input_file:
    if int(line) < 0:
        inp = - twos_comp(- int(line),8, True)
    else:
        inp = int(line)
```

Listing 1. Input check with two's complement

Output values are sometimes truncated, so one needs to manage them carefully: to obtain the corresponding integer we shift them left by 2 (or 3) positions. If the considered value represents a negative number the two's complement is applied and the correct result is finally obtained.

```python
for signal in enumerate(signal_input):
    signal_temp = ord(ser.read())
    # Bitwise left shift (+2 zeros to the right) for the output
   value
    signal_temp = signal_temp << 2
    # If signal_temp > 511, it represents a negative number
    if signal_temp > 511:
        # 2 Complement to get the corresponding negative number
        signal_temp = twos_comp(signal_temp, 10)
    signal_output.append(signal_temp)
```

Listing 2. Method 1: output check

5. Results on VHDL

For the sake of comparison we decided to build four different simulations with different numbers of taps and truncation methods.

Method 1 Filter with **four taps** truncating the two LSB;

Method 2 Filter with **four taps** with **resize**, so truncating the two MSB;

Method 3 Filter with **five taps**, truncating three LSB;

Method 4 Filter with **five taps** with **resize**, so truncating the three MSB.

The last method should return the best results, due to an higher number of taps and the soften of approximations introduced by truncation.

6. Simulation on Python and comparison with VHDL

We implemented the same FIR filters as we did in *VHDL* also in *Python*, in order to make a comparison between the two. Here are the two functions that we used to implement the FIR filter. The first one, given the n taps coefficients, returns their integer value after being left-shifted by 8 bits. The second function instead is used to effectively implement the filtering of the signal with different taps.

```
def filter(wave, taps):
    '''

    fir filter implementation

    '''
    result = []
    data = np.zeros(len(taps))
    for i in range(len(wave)):
        data = np.insert(data[:len(taps)-1],0,wave[i])
        result.append((taps*data).sum())
    return np.asarray(result)
```

Listing 3. Filtering function in Python

Now we show how we created the wave signal. We used a carrier frequency of 20 Hz and a noise component at 4 kHz. In the end, we took the linear combination of the two waves.

```
import numpy as np

fs = 10000          # sampling rate [Hz]
duration = 0.1      # [s]
f = 20              # [Hz]
f_noise = 4000      # [Hz]

#time mesh
t = np.arange(fs*duration)/fs

#carrier wave
samples = (np.sin(2*np.pi*np.arange(fs*duration)*f/fs)).astype(np.
    float32)

#noise wave
noise_samples = 0.1*(np.sin(2*np.pi*np.arange(fs*duration)*f_noise/
    fs)).astype(np.float32)

wave = samples + noise_samples

plt.plot(t,wave)
plt.show()
```

The output is the following:

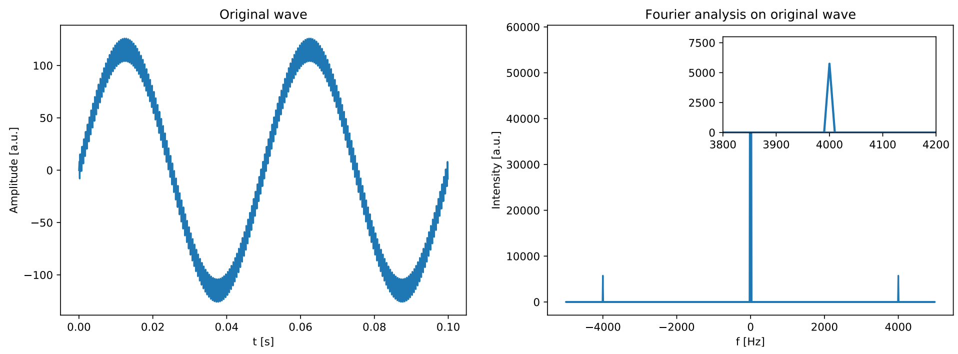

Fig. 3. Shape and Fourier analysis on the original wave.

6.0.1 Spectrum analysis

Firstly, we report the output wave and the frequency spectrum in Fourier's space of the signal filtered with the function `filter` defined at Listing 3 right above.

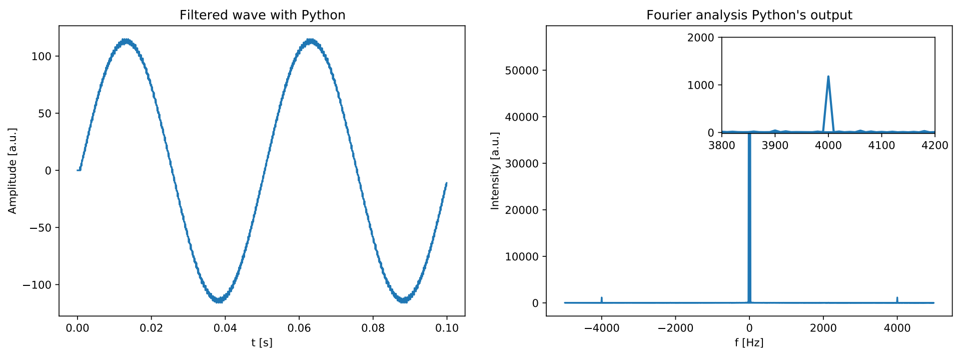

Fig. 4. Shape and Fourier analysis on the Python's output.

In order to compare *Python*'s and FPGA implementation's results we sampled the simulated wave (Fig. 3) and saved the points in a file - the input of our FPGA. Then we ran the FIR filter on the board and printed the filtered wave on another file. From these files we finally obtained the results presented in the following graphs. As one can notice, the truncation on the last two bits leads to strong approximation. In fact, we get as output a *discretized* version of the previous wave - anyhow a good results considered the bias introduced.

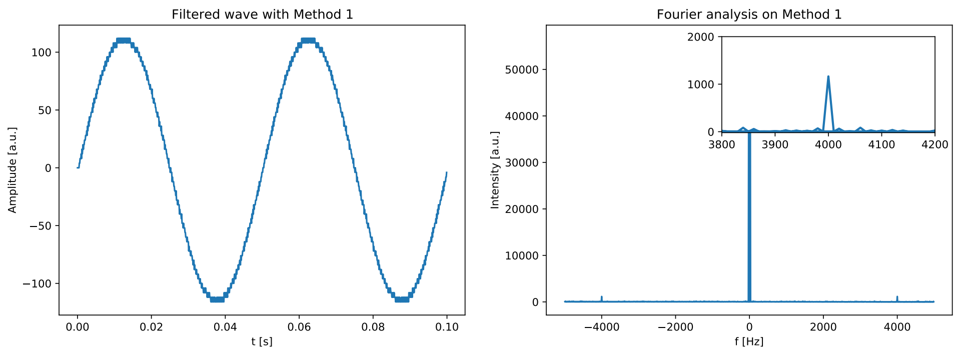

Fig. 5. Shape and Fourier analysis obtained with method 1.

Since the output values for the filter can be represented in 8-bit, it is also possible to truncate the two MSBs of the 10-bit-output of the sum vector and obtain a correct result.

Due to a technical problem, the following results are only simulated with `ghdl`. Anyway they give practically identical results.

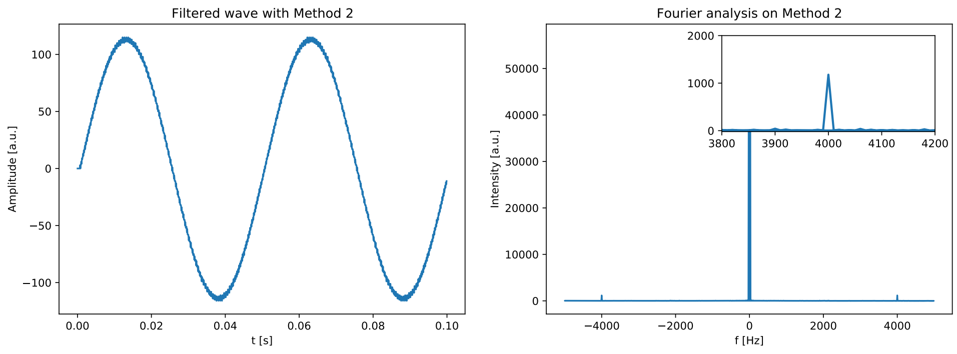

Fig. 6. Shape and Fourier analysis obtained with method 2.

The wave presented in Fig.6 is smoother than the previous one. Between the two 4-taps filters, the last one surely yields the best result.
For a more complete analysis we studied the behaviour of a 5 taps filter truncating 3 bits. The results are displayed below.

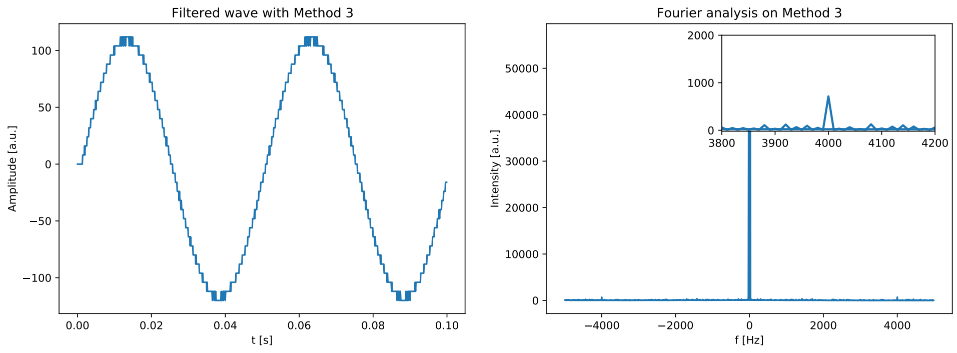

Fig. 7. Shape and Fourier analysis obtained with method 3.

The first method for the 5-taps filter gives an approximated wave, due to the LSBs cut. If we cut the MSBs as before the result is cleaner. It can be also noticed from the noise reduction how the filter has improved from the 4-taps implementation (Fig. 9).

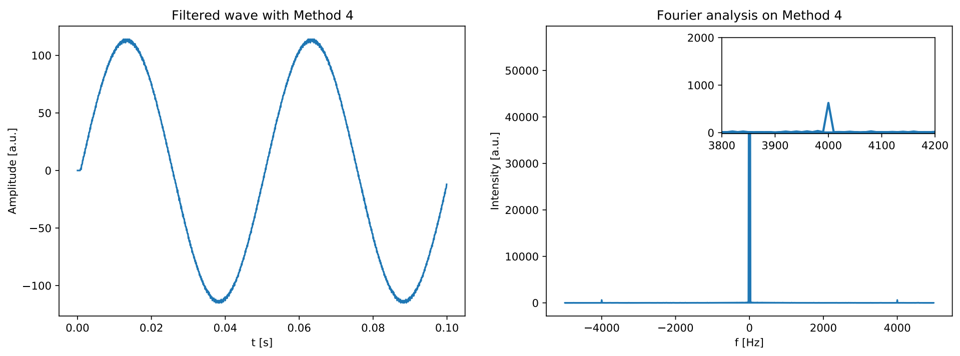

Fig. 8. Shape and Fourier analysis obtained with method 4.

As a last thing for this paragraph, we report the comparison between the two best methods (2 and 4) on how they behave in the suppression of the peak at 4 kHz. It is clear that slightly increasing the number of taps a better noise suppression can be obtained.

Further considerations on this comparison are treated in the last paragraph.

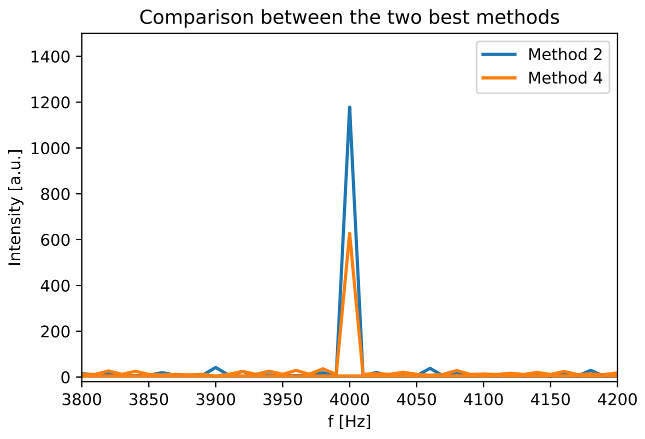

Fig. 9. Comparison between the two best methods.

6.0.2 Frequency response (Bode)

The `signal` library made also possible the creation of some Bode plots, which are useful to see the filter's frequency response. As we could have imagined, the number of taps we are working with is too low to have efficient FIR filters. In the following figures we can clearly see how the frequency we have chosen is more or less (better with 5, worse with 4 taps) cut off. If our signal had contained lower frequencies (e.g. 400 Hz) we would not have been able suppress them. Also, we can see how the cutoff frequency is not displayed around the 'shoulder' of the Bode plots in both cases, meaning that the two filters have a somehow imprecise frequency response.

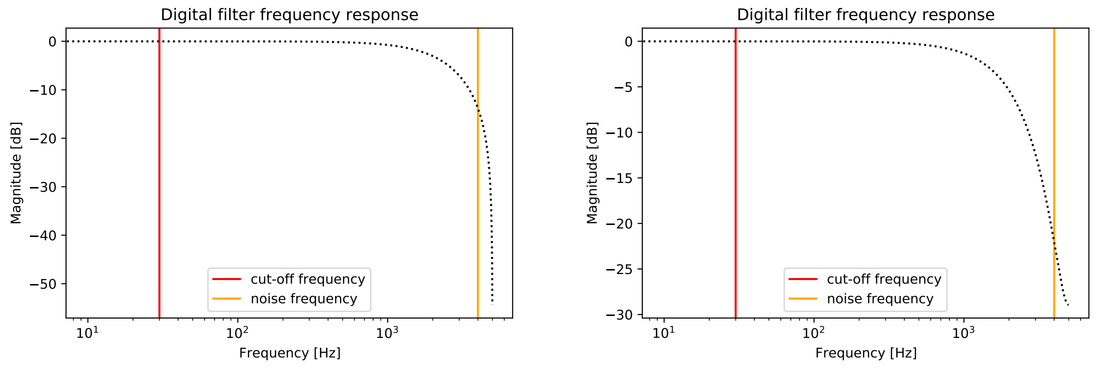

Fig. 10. Frequency response of FIR filter with 4 (on the left) and 5 (on the right) taps.

The case is different for an ideal filter with 300 taps. Here we see the characteristic 'bouncing' behaviour of the FIR filter and the cutoff frequency more or less in its correct place.

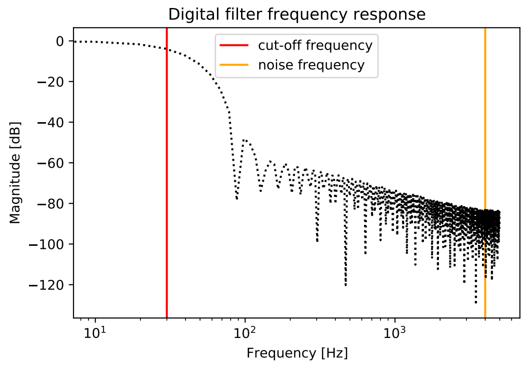

Fig. 11. Frequency response of FIR filter with 300 taps.

7. Conclusion

In conclusion, we have been able to develop a lowpass FIR filter in order to perform filtering on a noisy wave. We compared the behaviours of two different FIR filters, with different number of taps and rounding methods from the 10-bits vector down to the 8-bits one. We performed a spectrum analysis on the output wave and we showed the frequency response of the filters.