Implementation of a FIR filter co-processor in FPGA

Avella Michele, Giorgetti Sabrina, Ziliotto Filippo
2024548 - 2013375 - 2017425

April 2021

Abstract

In this project the goal was to implement a FIR filter co-processor in a FPGA: we first designed a 4-tap low-pass FIR filter in VHDL and, with Python's environment libraries, we calculated the corresponding FIR filter coefficients, performing also a frequency analysis. Then we compared the two results. The report is organized as follows. In the first section we describe the system circuit and its hardware components, we show also how the FIR filter has been design in VHDL. Then we focus on how Python simulation and the FPGA test have been performed. Finally, in the last parts we plot the results comparing the two implementations.

1 System implementation

To implement the FIR filter in the FPGA the system is composed as shown in the figure 1:

- UART receiver
- FIR filter
- UART transmitter
- Python interface

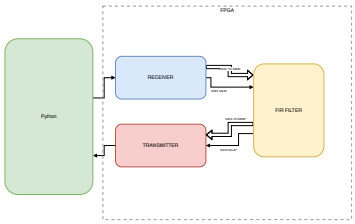

Figure 1: Block diagram

The goal, in order to control the correct functioning of the circuit, is to have as output a filtered signal: through a python script we'll sent to the UART receiver the input signal, that will be made readable for the FIR filter; the output will be then returned by the UART transmitter and read again with a python script. As shown in the block diagram 1, it's introduced a "data-valid" signal in order to ensure the correct reception and transmission of the signals at the correct time.

To program the FPGA it will be used VHDL: VHDL is an hardware description language that can be used to configure a programmable device, as an FPGA. In particular a VHDL code that describes a circuit, it's made up by two main parts: the entity and the architecture. The entity describes the inputs and outputs of the circuit while the architecture describes what the circuit actually does. The main feature of VHDL is that it executes unlimited number of statements at the same time, so the programming paradigm is built around parallelism and this is because digital circuits operate in parallel.

In the sections below we'll describe the functioning of the different components and show the written VHDL code for each one of them. The entire code can be found at the following link: `https://github.com/micheleavella/MAPD_A`.

1.1 UART receiver

The UART receiver, where UART stands for universal asynchronous receiver-transmitter, it's an hardware device for asynchronous communication. In the system, the UART receiver takes as input a single line of sequential bits and a clock signal, and as outputs it returns bytes of data and a data valid bit, that indicates when a sample is available to the next device connected, in our case the FIR filter.

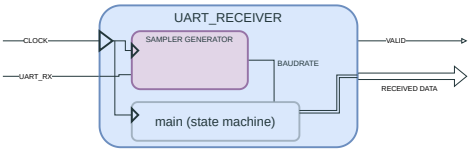

Figure 2: UART receiver block diagram

The UART receiver is composed by two main module: the sampler generator and a state machine, as in figure 2. The state machine is initially in the *idle* state and it remains in this state until the input signal turns 0; when this happens, the state changes to *start* and the sampler generator is enabled. The state machine then changes state each time the shifted baudrate (from the sampler generator) turns 1, until it returns to the *idle* state. During the *bitn_s* state the UR reads the input signal and store its value into the nth bit of the received data. The *valid* signal is 0 in the *idle* state and is 1 in the other states. The sampler generator, needed to provide the main UART receiver module with data sampling pulses, it's instead made of three sub-modules: a pulse generator, a state machine and a delay line. When the pulse generator is enabled it emits a pulse, driven by the baud-rate; the sampler state machine then counts a number of pulses that is equal to the numbers of bits plus the start and the stop ones, and then disable the pulse generator. The delay line, meanwhile, delays every pulse by half of the baud-rate pulses period in order to read more precisely the input signal. In figure 3 is shown the first part of VHDL code where the mentioned signals are defined.

```vhdl
entity uart_receiver is

  port (
    clock         : in  std_logic;
    uart_rx       : in  std_logic;
    valid         : out std_logic;
    received_data : out std_logic_vector(7 downto 0));

end entity uart_receiver;


architecture str of uart_receiver is

  type state_t is (idle_s, start_s, bit0_s, bit1_s, bit2_s, bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s, stop_s);
  signal state : state_t := idle_s;

  signal baudrate_out : std_logic;
  signal received_data_s : std_logic_vector(7 downto 0);

  component sampler_generator is
    port (
      clock        : in  std_logic;
      uart_rx      : in  std_logic;
      baudrate_out : out std_logic);
  end component sampler_generator;
```

Figure 3: UART receiver code in VHDL

1.2 UART transmitter

The UART transmitter is instead a hardware device that takes in input bytes of data in parallel, converts them in a series of sequential bits, shifted bit by bit at a specific rate, and transmit them as output. As shown in figure 4, it's main components are a baudrate generator and a state machine.

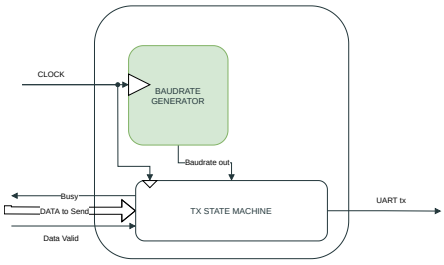

Figure 4: UART transmitter block diagram

The baudrate generator acts as a counter for the state machine, in fact in our case, its output returns a 1 bit signal only after every 868 clock cycles, with a baud-rate equal to 115200 bit/s. The state machine, starting from the *idle* state and relying on the output of the baudrate generator, changes state every cycle. In the end, the state machine returns in *idle* state, after the transmission bit by bit from the LSB to the MSB. As for the receiver, there are three other intermediate states: the start, data-valid and stop state, which guarantee the correct transmission of data. In our circuit, the transmitter takes as input the clock signal and the data already processed by the FIR filter and then returns as an output a bit by bit sequence of the input data. Below, in figure 5 we provide the VHDL code.

```
entity uart_transmitter is

  port (
    clock        : in  std_logic;
    data_to_send : in  std_logic_vector(7 downto 0);
    data_valid   : in  std_logic;
    busy         : out std_logic;
    uart_tx      : out std_logic);

end entity uart_transmitter;


architecture rtl of uart_transmitter is

  component baudrate_generator is
    port (
      clock        : in  std_logic;
      baudrate_out : out std_logic);
  end component baudrate_generator;

  signal baudrate_out : std_logic;
  -- state machine signals
  type state_t is (idle_s, data_valid_s, start_s, bit0_s, bit1_s, bit2_s,
                   bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s, stop_s);

  signal state : state_t := idle_s;
```

Figure 5: UART transmitter code VHDL

1.3 FIR filter

A finite impulse response (FIR) filter is characterized by an impulse response of finite duration, as it settle to zero after a finite time. The FIR filter behaviour can be described with the equation:

$$y[n+1] = \sum_{i=0}^{N} x[n-i] * C_i = x[n] * C_0 + x[n-1] * C_1 + \cdots + x[n-N] * C_N \tag{1}$$

where $y[n+1]$ is the output signal, $x[n-i]$ is the input signal and C_i the coefficients of the filter. Usually we refer to a filter specifying the number N of taps, where N coincides with the number of coefficients. Let's notice that the equation (1) is a convolution operation, or more simply, a weighted moving average.

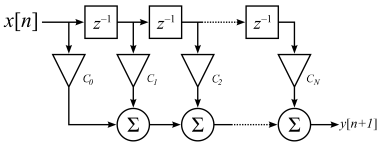

Figure 6: Fir filter

In our case we'll consider a four taps filter, such that 1 can be written as :

$$y[4] = x[3] * C_0 + x[2] * C_1 + x[1] * C_2 + x[0] * C_3 \tag{2}$$

so the FIR filter takes in input both the $x[0], \cdots, x[3]$ signals and the four coefficients C_i, with $i = 0, \cdots 3$. Firstly it performs all the multiplications operation and then it will calculate the sum of the corresponding product outputs.

In figure 7 is shown the entity code where are defined the I/O terminals. Specifically the FIR filter has seven inputs: the clock,the in data in bits, the data valid and the values of the coefficients; as output only the data out in bits.

```
entity fir_filter_4 is

port (
      clk          : in std_logic;
      data_in      : in  std_logic_vector(7 downto 0);
      data_valid   : in std_logic;
      data_out     : out std_logic_vector(7 downto 0);


     -- coefficients
     coeff_0 : in  signed(8 downto 0);
     coeff_1 : in  signed(8 downto 0);
     coeff_2 : in  signed(8 downto 0);
     coeff_3 : in  signed(8 downto 0)
    );

end fir_filter_4;
```

Figure 7: FIR filter entity

```
begin

reading : process(clk)
        begin
                if rising_edge(clk) then
                        if (data_valid = '0' and read ='0') then
                                x_3<=x_2;
                                x_2<=x_1;
                                x_1<=x_0;
                                x_0<=signed(data_in);
                                read <= '1';
                                data_out<= tot_out;
                        end if;
                        if data_valid = '1' then read <= '0'; end if;
                end if;
        end process reading;


prod : process(clk)
        begin
                if rising_edge(clk) then
                        p_0<=x_0*(coeff_0);
                        p_1<=x_1*(coeff_1);
                        p_2<=x_2*(coeff_2);
                        p_3<=x_3*(coeff_3);
                end if;
        end process prod;
```

```
sum1 : process(clk)
        begin
                if rising_edge(clk) then
                        s_01<=resize(p_0, 18)+resize(p_1, 18);
                        s_23<=resize(p_2, 18)+resize(p_3, 18);
                end if;
        end process sum1;

sum2 : process(clk)
        begin
                if rising_edge(clk) then
                        y<=resize(s_01, 19)+resize(s_23, 19);
                end if;
        end process sum2;


output : process(clk)
        begin
                if rising_edge(clk) then
                        tot_out<=std_logic_vector(resize(shift_right(y,11),8));
                end if;
        end process output;

end rtl;
```

Figure 8: FIR filter assertions part of the VHDL code

The filter reads the input data each time data valid turns 0, because that means that the receiver has stopped to read the signal; this is done using also an auxiliary internal signal in order to avoid reading the same data twice. The new data is stored in x_0, the old content of x_0 is stored in x_1, the old content of x_1 is stored in x_2 and so on. As described before and shown in figure 8, the filter then performs firstly all the multiplications, returning a 17-bits signal, then the sums having as output a 19-bits number. At this point, in order to have an 8-bit number to send to the transmitter, we have to reduce the number of bits, so we removed the 11 least significant bits, using the $shift\_right()$ function.

2 Simulation in GTKWave

In order to test our VHDL code and before putting it in the FPGA we wrote a testbench and we simulated it with GTKWave. A testbench is a code written in VHDL that is linked to the top file and gives it the information about the input signals. The top file is a code written in VHDL that connects different components and describes how they interact with each other. GTKWave is a software that simulates what is described in the testbench and displays all the signals involved during the process: input, output, internal; in this way we could monitor everything and check if the code was working correctly.

2.1 Simulation of the single components

The first step was to check whether each component was working as expected. Using GTKWave we checked the behaviour of all the three components, writing the corresponding testbench for each one of them, the results are shown in the figures 9, 10, 11.

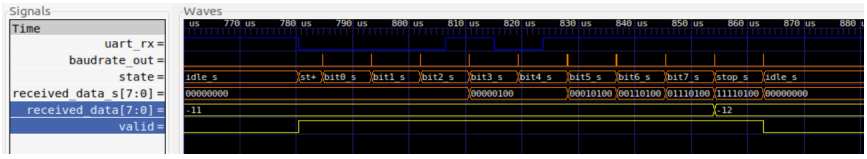

Figure 9: UART receiver simulation in GTKWave

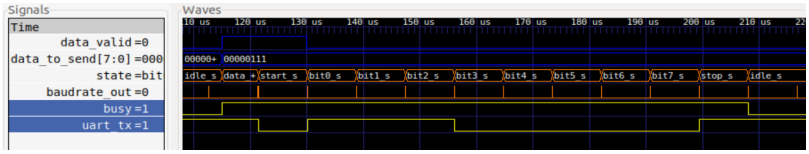

Figure 10: UART transmitter simulation in GTKWave

Figure 11: FIR filter simulation in GTKWave

2.2 Simulation of the implemented circuit

In order to simulate all the process we connected a UART transmitter and a UART receiver to our FIR filter implementation as shown below.
In this way we could "write" the input and "read" the output in GTKWave directly in binary (or decimal) form, as we will do with the python script and the FPGA in the last part of the project. After connecting everything in a new top file, we wrote a testbench giving some numbers as input. Using GTKWave we checked the output of the simulation and we compared it to the simulation done with python.

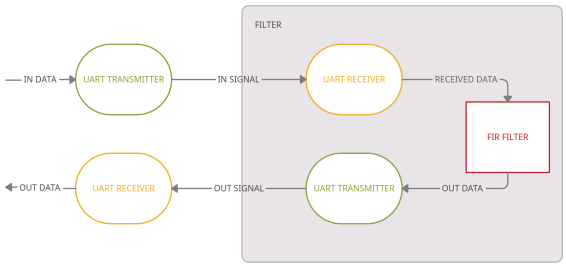

Figure 12: Block Diagram

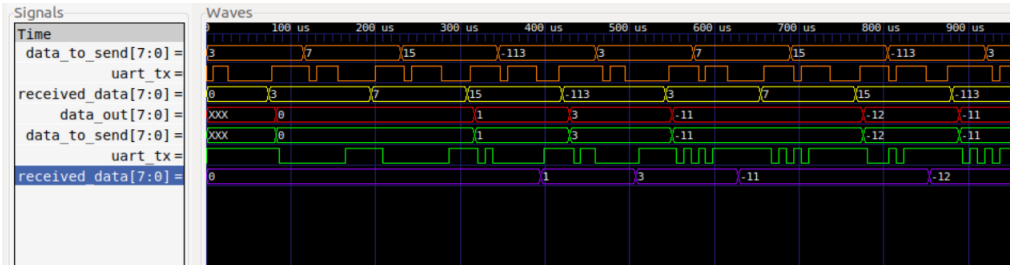

Figure 13: GTKWave simulation.

In the figure 13 we can observe some of the signal and data of the simulation, everything is working correctly as expected. Starting from the top:

- orange: UART transmitter;
- yellow: UART receiver of the FIR filter block;
- red: FIR filter;
- green: UART transmitter of the FIR filter block;
- purple: UART receiver;

3 FIR filter testing

In the project, we considered a 4-th order low-pass FIR filter. In Python it has been implemented as in figure 14. The values of the coefficients are computed through the Python library *scipy.signal* `https://docs.scipy.org/doc/scipy/reference/signal.html`, by setting a cutoff frequency of 0.1.

```
output_sim = []
coeff=b
for i in range(100):  #Using 100 points
    a=0
    for j in range(len(coeff)):
        a += input_fir[i-j]*coeff[j] #fir filter formula
    output_sim.append(round(float(a),2))

print('Output Simulation',output_sim)
```

Figure 14: FIR Filter's computational formula

The frequency analysis for this filter setup is showed in figure 15.

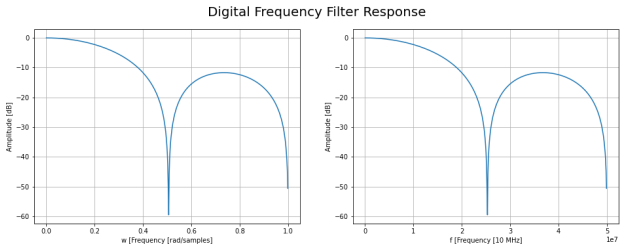

Figure 15: Frequency analysis for 4-tap FIR Filter

The values of the 4 coefficients are:

- $b_0 = 0.2459$
- $b_1 = 0.2541$
- $b_2 = 0.2541$
- $b_3 = 0.2459$

As we can see, they are two by two equal. Since every operation on the FPGA is done with integer arithmetics, it is fundamental to overcome the limit of the finite precision of those coefficients. An idea is to multiply them by a large number such 10^3 and then truncating the floating part. Hence, the values of the coefficients in the VHDL code are set to:

- $245_{dec} \rightarrow F5_{hex}$
- $254_{dec} \rightarrow FE_{hex}$
- $254_{dec} \rightarrow FE_{hex}$
- $245_{dec} \rightarrow F5_{hex}$

```
1 fs = 100000000.0 #Arty7 board frequency oscillator
2 b = signal.firwin(4, 0.1, window='boxcar')  # 0.1 is the cutoff frequency
3 w, h = signal.freqz(b)
4 print([hex(int(b[i]*1000)) for i in range(len(b))])
```

```
['0xf5', '0xfe', '0xfe', '0xf5']
```

Figure 16: Generating FIR filter coefficients

Obtained the coefficients, we could program the FPGA. We remotely connected to the FPGA via the Xilinx server and the test was performed on the Arty7 board (USB port 15). First of all, we generated the bitstream and programmed the device with the $make - program$ command. In order to communicate with the FPGA, to receive and sent data, we used the library `serial.Serial`, an example of the code is presented in figure 17. We had to be careful to send data to the FPGA because it reads the input bits as *signed* type using the two's component notation. In order to send the data correctly we had to change the values between $[-127, -1]$ to $[129, 255]$. *(ex: the number -2 in the two's component notation is 11111110 that corresponds to the number*

```python
from tqdm import tqdm
from time import sleep
import serial

ser = serial.Serial('/dev/ttyUSB15', baudrate=115200)

for i in tqdm(data):

    ser.write(chr(i))
    sleep(0.01)
    d = ser.read()
    out_data.append(ord(d))

ser.close()
```

Figure 17: Example script for FPGA

254 in the binary notation; so if we want to send -2 we have to send the number 254). For this reason, after generating the data, we shifted the negative number by +256.
For the entire code of the simulation and the actual implementation on the hardware, both in VHDL and Python see the github page `https://github.com/micheleavella/MAPD_A`.

4 Results

This section displays all the results we obtained for the project, combining the simulated results with the FPGA ones. The Python simulation results have been shifted by adding the needed 0's to the initial values, in order to temporarily match the FPGA output.

4.1 Sinusoidal Waveform Input

The first waveform under study is a simple sine wave where we added some random noise in order to show how the FIR-Filter behaves. The considered parameters are the following:

- Amplitude: $A = 70$
- Period: $T = 31$ samples
- Random noise

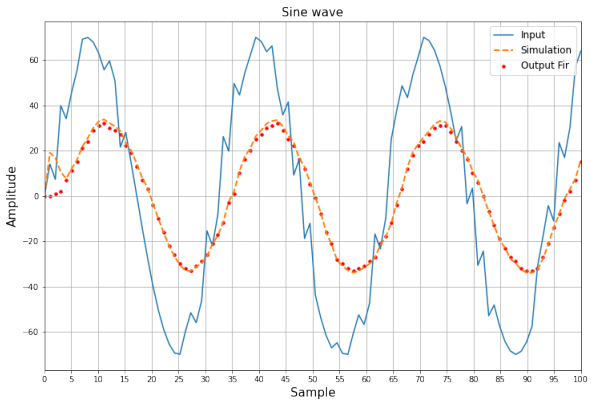

Figure 18: Output for sinusoidal waveform

As we can see from the results of the plot, the filter behaves as expected, in other words it acts like a low-pass FIR filter smoothing the input data. There are some few exception for the samples corresponding to the initial transient states, either for the output of the FIR Filter and the simulation.

4.2 Square Waveform Input

The second waveform under study is a square wave with the following parameters:

- Amplitude: $A = 50$
- Period: $T = 22$ samples

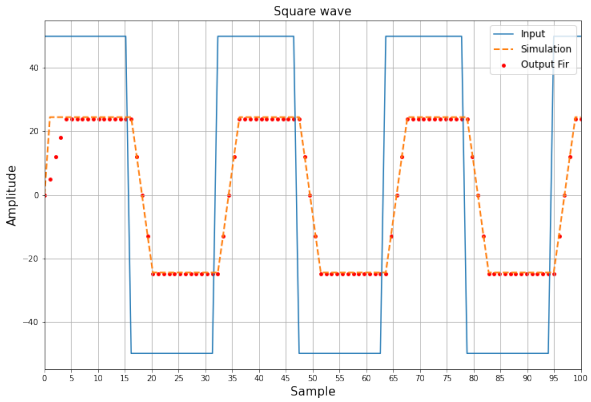

Figure 19: Output for square Waveform

The results for this waveform are showed in the Figure 19. As for the previous waveform, the filter behaves as expected, except for the samples corresponding to the initial transient states.

5 Conclusions

In this report we presented a low-pass FIR filter implementation in the FPGA hardware. Firstly we described how the single components of the circuit works and how they can be designed in hardware programming language as VHDL. Exploiting the GTKWave program we performed all the simulation in order to check the correct functioning of our circuit. With the Python interface, thanks to the several packages available, we have calculated the FIR filter coefficient and simulated the FIR filter functioning in this particular environment. The system has been then experimentally tested on the FPGA, a Arty7 board, via the Xilinx server remote connection. The input waveform results displayed in the previous section have then been compared to the ones obtained in Python: in both cases the results are coherent, except for the initial transient states.