UART with FIR Filter

MAPD A - Group 3

Academic Year 2020-2021

Members:
Manara Noemi (2022909)
Simionato Giuseppe (2029013)
Zambelli Francesco (2029014)

Abstract: *This report presents the implementation of an UART (Universal Asynchronous Receiver-Transmitter) interface combined with a FIR (Finite Impulse Response) Filter on an FPGA (Filed-Programmable Gate Array). The FPGA is an Integrated Circuit designed to be configured after manufacturing, for this reason the term 'field-programmable'. It is composed by an array of logic blocks that can be reprogrammed: each block is individually programmed to perform a logic function. It is composed also by reconfigurable interconnections that allow the different blocks to be wired together to create different configurations. The device can be designed with the hardware description language VHDL (Very High Speed Integrated Circuit Hardware Description Language). The use of FPGA components has some advantages over ASICs (Application Specific Integrated Circuits): they are in fact standard devices whose functionality to be implemented is not set by the manufacturer, who can therefore produce on a large scale at a low price. Their generic nature makes them suitable for a large number of applications in the consumer, communications, automotive and other sectors. They are programmed directly by the end user, displaying design times, verification by simulations and field testing of the application. So, the big advantage over ASICs is that they allow the end user to make any change or correct errors simply by reprogramming the device at any time.*

Keywords: finite impulse response filter; universal asynchronous receiver transmitter; digital circuits; field programmable gate arrays; high-pass

Introduction: The transmission is the process of sending and propagating an analogue or digital signal using a transmission medium. The asynchronous communication is a form of communication in which the two endpoints' interfaces are not continuously synchronized by a common clock signal. Indeed, the data stream contains synchronization information in form of start and stop signals (start and stop bits) before and after each unit of transmission. The start signal prepares the receiver for the arrival of data and the stop signal resets its state to enable triggering of a new sequence. The UART hardware device is used for asynchronous communication where all operations are controlled by an internal clock. We need a periodic signal at the rate of the transmission to synchronize the outcoming data and the incoming data of the two devices, in order to communicate without any loss of information. This signal is known as "baudrate" and is provided by the baudrate generator component.

This work presents the designed UART-with-FIR circuit and analyzes the proper operation of that device. More in detail, it is shown which is the best configuration found for the implementation of the filter and the various components. The consequent circuit was tested firstly with a proper testbench and then it was synthesized on the FPGA. The results are compared with a Python simulation of the same circuit to see if the behavior of the hardware is the expected one and if it is all properly implemented. At the end, a practical application of this circuit is proposed.

1 Structure of the circuit

The designed circuit combines a UART interface with a FIR Filter and it contains the following components:

1. the Baudrate generator
2. the Transmitter
3. the Sampler generator
4. the Receiver
5. the FIR Filter

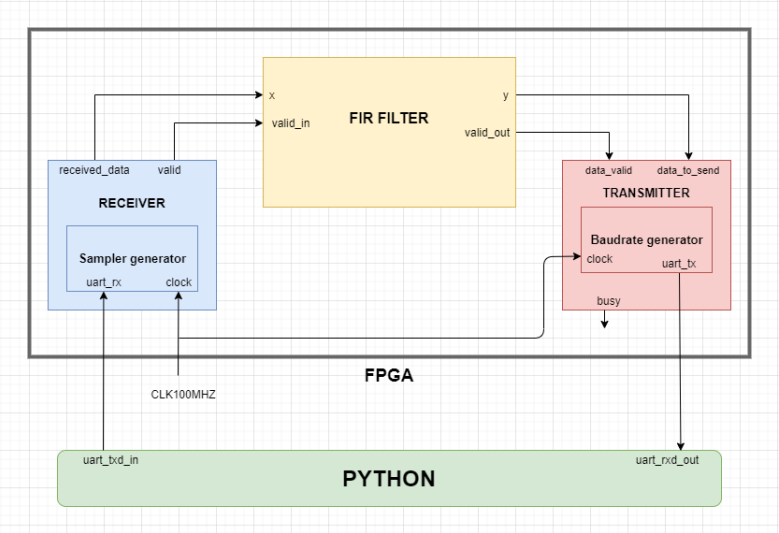

Table 1: Block Diagram

The Baudrate generator is a component of the transmitter and also the Sampler generator is a component of the receiver. Their utility will be explained in the following sections, but it is important to note that the two generators are components of the other two devices. The block diagram of the circuit is shown in Figure 1. We can see that the interface with the UART is implemented with Python, where the code provides the input data and processes the output from the FPGA through a serial port.

2 UART Transmitter

The UART Transmitter is an hardware device that, given a byte of data, allows to transmit it in a sequential format of individual bits. In our implementation its role is of retrieving the signal in output from the fir filter, and transmit it as a sequence of bits (impulses) back to the interface. Its design provides five ports, three in input (clock, data valid, data as bytes) and two in output (busy state, data bits), and the baudrate generator component.

This device implements a state machine: the initial state is `idle`, and it changes in the clock cycle in which the signal of data valid becomes one, as the transmission process is triggered. Synchronized with the baudrate generator, the machine changes state and transmits accordingly a data bit, most significant to least significant, providing the appropriate padding of a `0` as the start bit before the signal bits and `1` as the stop bit after the last signal bit. After the transmission of the 2 padding bits and the 8 data bits, the state returns to `idle`, ready for the next data byte to be transferred.

2.1 Baudrate generator

The internal component of the UART transmitter is the baudrate generator: it provides a signal that changes from 0 to 1 for exactly one clock cycle every bit time. It is basically a new clock with period equal to the rate of transmission. Given our constraints:

- a board clock frequency of 100 Mhz, with 10 ns period
- a baudrate of 115200 bit/s

the number of clock cycles for the transmission of one bit is of 100 000 000 / 115200 $\approx$ 868 cycles. We designed and implemented a baudrate generator with these parameters, as an hardware component with an input port with the clock signal, and an output port with the baudrate signal. An internal counter triggers the pulse of the baudrate signal when 868 clock cycles are reached, and then restarts from 0.

3 UART Receiver

The UART Receiver is an hardware device that, given a sequence of bits, collects them in the parallel format of bytes of data. Its role in our circuit is of collecting the data bits from the I/O interface and provide them to the fir filter in the parallel format, as well as generating the data valid pulse every time a new data sample is given in input.

This device has four ports, two in input (clock and data as bits) and two in output (the data valid signal and data as bytes). It also implements a sampler generator. The receiver follows a state machine process, that, starting from `idle`, changes in sync with the pulses of the sampler generator, storing the bits in a byte in the appropriate importance order: from most significant bit to least significant bit. With the ninth pulse the state goes back to `idle`.

3.1 Sampler Generator

The Sampler Generator implemented in the UART receiver gets as input the clock signal and the data bits and generates a sequence of nine pulses with the baudrate frequency, starting at the identification of the start bit. It is of paramount importance since the pulses it gives in output trigger the change of state of the receiver, allowing it to correctly read every bit of information.

Three processes join in this component: an internal baudrate generator, as discussed for the transmitter, a state machine and a delay line. The state machine triggers the baudrate generator and keeps it going for the nine baudrate cycles the receiver needs to read the data. The last process delays the pulses in output of half baudrate time, so that the bits' signal waves will be read far from their edges.

4 Testbench of the UART

We display the operation of the whole UART module as a wave file, generated from the run of a testbench. The signals in orange are related to the UART receiver, the ones in yellow to the UART transmitter, and the ones in green are the input given to the circuit.

In Figure 1 we see the transmission of three signed data: 5, -2 and 9. It is possible to appreciate the change of state triggered by the data valid signal and in sync with the baudrate generator for both receiver and transmitter, with a delay of one baudrate in the second case due to the byte composition process. The last yellow wave represents the bit flow in output from the module. As one can see, the input is faithfully transferred through the UART, which corroborates the design and implementation of these components.

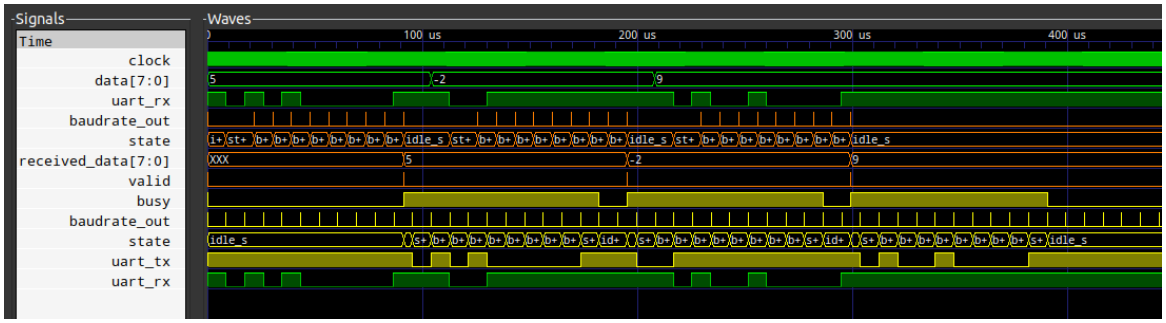

Figure 1: Gtkwave simulation UART

5 FIR Filter

A Finite Impulse Response Filter is a digital filter whose response settles to zero after a finite time. It obeys the relation:

$$y[n+1] = \sum_{i=0}^{N} x[n-i] \cdot C_i$$

Where y is the data sample in output, N is the number of taps (coefficients), x are the N data samples in the pipeline and C_i are the coefficients of the filter. The algorithm of the FIR is schematized in Figure 2, where we can see the different phases of the process. At the beginning the filter receives the data and updates the pipeline with the inputs received, then it calculates the different products for the coefficients and finally it calculates the sum over all the products. The output is given after the sum; in the case that another input is received the algorithm is repeated in the same way, updating the pipeline.

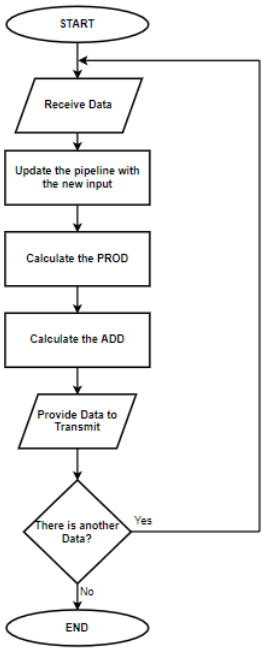

Table 2: Flowchart of FIR filter algorithm

We designed a high pass filter for high frequencies with 5 taps and the cutoff frequency equal to 0.85 of the highest frequency that can be sampled given the *Nyquist frequency*.

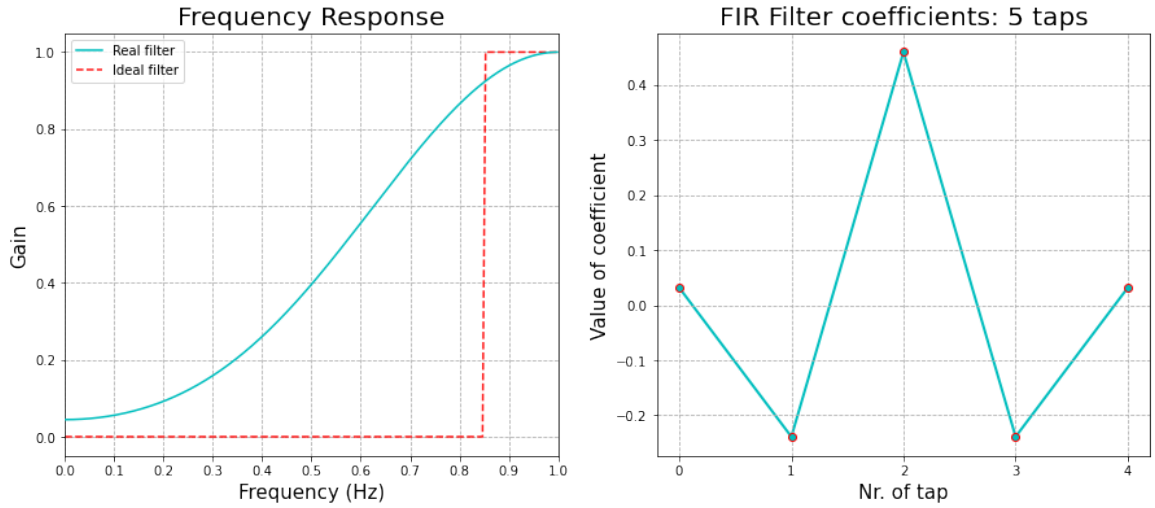

Figure 2: Graphical representation of the frequency response of the filter (left) and the symmetric coefficients (right)

The coefficients, found with the scipy module firwin, are the following: $C_0 = C_4 = 0.03524069, C_1 = C_3 = -0.2408398, C_2 = 0.44783902$. We find that multiplying these numbers with 32 the results can be approximated to non-zero integers, which are the coefficients inserted in the fir filter:

$$C_0 = C_4 = \frac{1}{32}, C_1 = C_3 = -\frac{4}{32}, C_2 = \frac{7}{32}$$

In this way, it is sufficient to allocate 4 bits for the binary two's complement representation of the coefficients. We know that, when multiplying two `std_logic_vectors` in VHDL, it is mandatory to allocate for the result a number of bits equal to the sum of the bit length of the two factors, in our case

8+4=12. After the product, the data undergoes four additive processes, and in this case, since each addend has length 12, a number of bit equal to 12+`floor`($\log_2 5$), where 5 is the number of addends, must be allocated for the result, so in our case a total of 14 bits.

5.1 Precision and assumptions on the applicability of the designed filter

As a final step of the FIR Filter design, it was incumbent to choose which bits of the result of the final additions were to be passed to the UART Transmitter: the result signal of the FIR is encoded in 14 bits, whilst the UART Transmitter transmits 8 bits for each sample.

We state that this filter was designed to work properly for data in the range [-32,31], which can be represented with 6 bits in binary two's complement. So the final result won't ever exceed a number that can be represented with 12 bits (6 of data samples, 4 due to the products, 2 due to the sums). For this reason, when selecting the bits for the output we combined the sign bit (14th) with the sequence of bits from the 11th to the 5th included. In this way we avoided the issue of overflow while preserving an acceptable precision. The final output of our designed FIR Filter can be interpreted as a signed integer that can be converted to the decimal representation dividing it by two: by shifting of 4 positions in the FIR module, we already divided the result by 16, half of the taps' coefficient, 32. So in the end we get an output with a sensitivity of 0.5 data units.

To tackle the issue of resetting the filter's pipeline before the next usage, the easiest solution found is to send through the serial port a final sequence of five zeros, that replaces the data in the pipeline and also resets consequently the product signals.

6 Testbench of UART with FIR filter

In order to validate our design of the FIR Filter and its connections to the other components, we simulated the circuit with the aid of a testbench.

We transmitted to the receiver eight signed data samples: 5, -2, 9, 8, 25, -12, 1 and 1. In Figure 3 it is possible to see the internal responses of the circuit. In orange are displayed the signals of the UART Receiver, in blue the ones of the FIR Filter and in yellow the ones of the UART Transmitter. In green are displayed the input and the clock signal. The behaviour of the UART module has already been discussed and presents no variations from before, except from the data in output, which are now provided by the FIR. The behaviour of the fir is regular: triggered by the data valid signal, bytes are inserted in the pipeline (consisting of the signed variables x0, x1, x2, x3, x4). At each new insertion the multiplication process is triggered, each sample in the data pipeline is multiplied by the appropriate coefficient of its position and the results are stored in the `prod` variables. Then the sum of the products is computed and is transmitted with smaller precision (it is needed to resize from 14 bits to 8) to the UART Transmitter, that generates the bitstream in output from the circuit.

7 Results: Simulation & Synthesis

7.1 Dataset

The dataset used to test the functioning of the circuit consists of 200 uniform samples of function 1, a high frequency beat function with a low frequency sinusoidal noise, in the t range [0,20].

$$y(\mathrm{t}) = 15 \cdot sin(30 \cdot \mathrm{t}) + 10 \cdot sin(0.5 \cdot \mathrm{t}) + 5 \cdot sin(15 \cdot \mathrm{t}) \tag{1}$$

7.2 Simulation with Python

In order to check the behaviour of the whole circuit and get the expected results for some hundreds of samples we couldn't use a simple VHDL testbench. Having already checked that the circuit architecture has a regular behaviour with a small number of samples in the previously mentioned testbench, we proceeded in writing a Python code to simulate the whole circuit and get the numeric results expected in output from the FPGA.

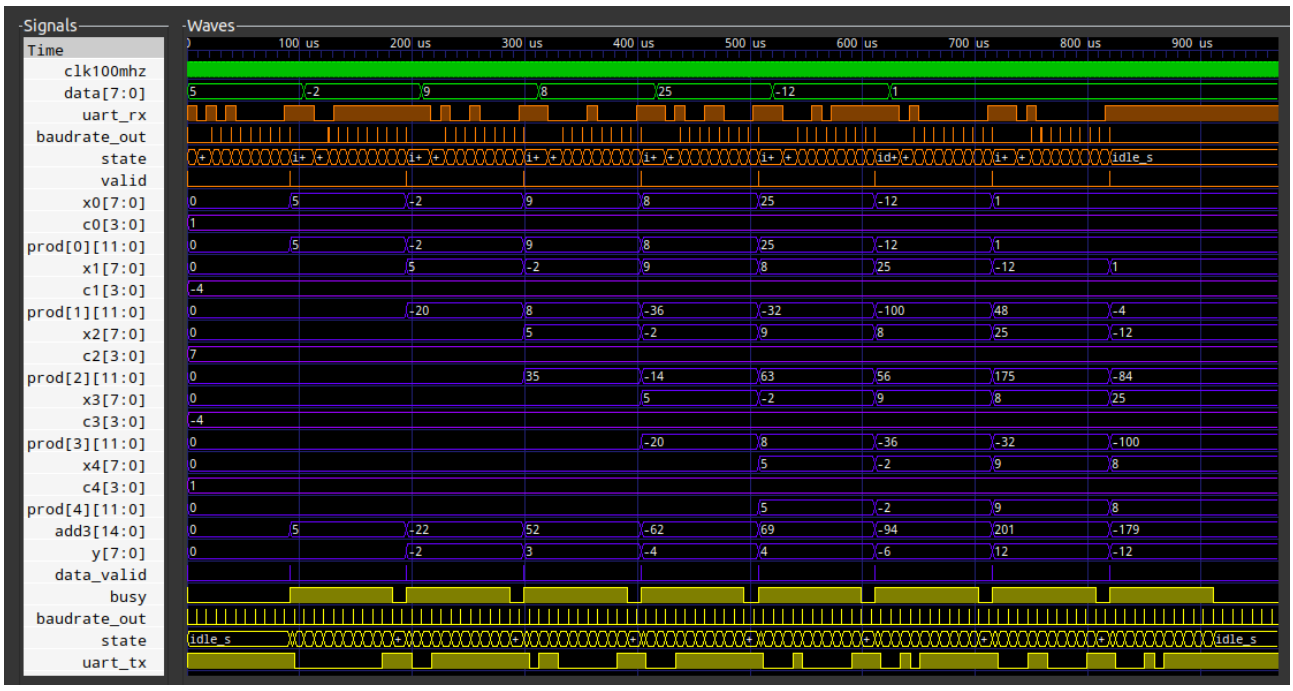

Figure 3: Gtkwave simulation of the complete circuit

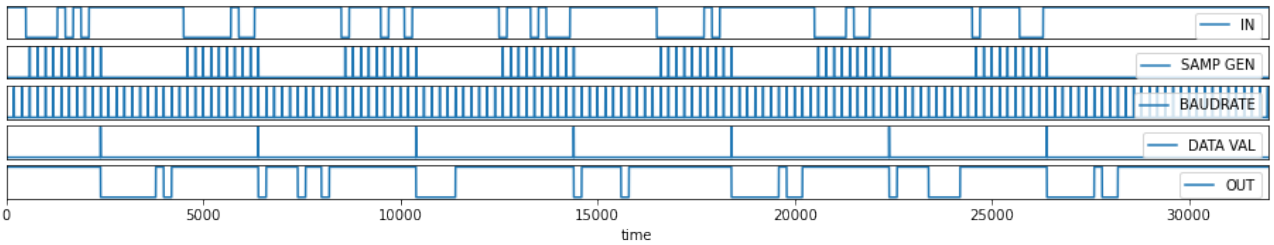

Figure 4: Some results of the simulation with Python

Every component was translated into a Python function, and the code was run on the same set of data that was sent to the programmed FPGA. The results of the simulation will be presented and discussed along the real outcome in the next section.

The behaviour of main variables of the circuit is shown in Figure 4 as from the Python script, with input: 21, 4, -10, -24, 5, 19, -8. The baudrate is taken equal to 20 to reduce the computational weight of the program, since it doesn't modify the function of the circuit. The similarity with the VHDL circuit is evident.

7.3 Synthesis on the FPGA

The FPGA was programmed via Vivado with the VHDL code of the circuit tested up to now. The ports connected to the top entity are: the pin of the 100MHz clock signal with a 10 ns period and the pins of USB-UART interface.

To connect to the USB-UART ports from a remote ssh and provide the input data, the pySerial Python module was exploited. We designed a Python interface that accessed the serial port, read data from a csv file, sent it in the ASCII format (so for negative data we wrote the binary interpretation of their two's complement representation), and finally received, translated from ASCII to signed and wrote to a csv file the output of the FPGA board. This way the results were easily accessed and interpreted with Python. Finally we gave in input a list of zeros to initialize the circuit for another dataset analysis.

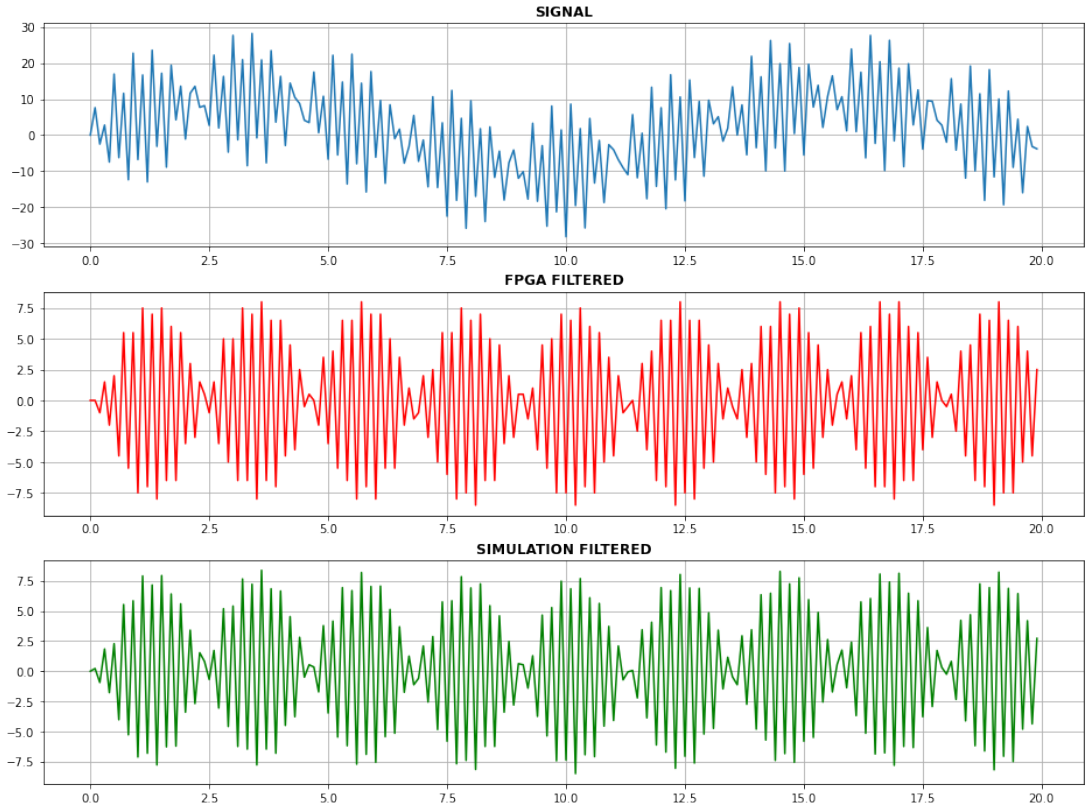

Figure 5: Plot of data waves. In blue the original samples of the dataset, in red the results of the FPGA implementation, in green the ones of the Python script.

7.4 Comparison of results

By comparing the results obtained from the FPGA board and the ones from the Python script and remarking their complete accordance, we deduce that the functioning of the real programmed circuit was correct.

The results are displayed in graphical form in Figure 5. It is possible to see that the simulated and real results of filtering match. Also, the circuit does manage to drop the low frequency component of the signal, preparing in this way the dataset for further analysis.

8 Conclusions and Applications

The circuit has been designed in VHDL, tested in ghdl with a testbench, then synthesized on the Arty7 FPGA. The results were compared to the ones obtained from a Python simulation. All of the tests and the comparison of results corroborates that the circuit designed works properly.

A possible application of this filter could be in the digital audio signal elaboration area, for isolating high frequencies when studying animals, as: birds chirping, mice and dolphins communicating. For instance, given the audio signal of a recorder aboard a ship (or more likely placed in the sea), applying a high pass filter as ours would allow to clean the record from the background noise of the ship's motion and the rolling of the sea, enhancing signals such as dolphins communicating.

Our specific design could be applied to the study of harbor porpoises, which emit narrowband, high-frequency clicks within 110-150 kHz for echolocation [1]. The peak is a centered at 130kHz [2] so, we could apply the filter in order to preserve the signal in the range 120-140 kHz, given a sampling rate of 280 000 samples per second. This could be a good idea for at least carrying out a simple real time analysis able to determine whether there are harbor porpoises signals in the area.

9 Appendix

A Reset

A finer approach to the reset of the FIR Filter pipeline was attempted by adapting the uart receiver to send a `reset` signal to the FIR whenever a sample in input exceeded the range of functioning of the filter, so any input signal in the ranges [-128,-33] and [32,127] triggered a process that led to the emission of a `reset` pulse. Also the FIR Filter was modified accordingly, in order to set to zero each entry of the pipeline whenever such signal was received. In this way two issues were tackled: a neater reset protocol and a way to prevent overflow issues in the case of out of range input.

The modified components led to the results displayed in figure 6, obtained via a testbench run. As it is possible to notice, the behaviour of the circuit is regular and each signal is shaped as expected. Even though, when the FPGA was programmed with such design, the transmission process always failed after a random number of transmitted and received data samples. Given the good results of the testbench in each aspect we concluded that the problem should be in some intercommunicating protocol of the pySerial module.

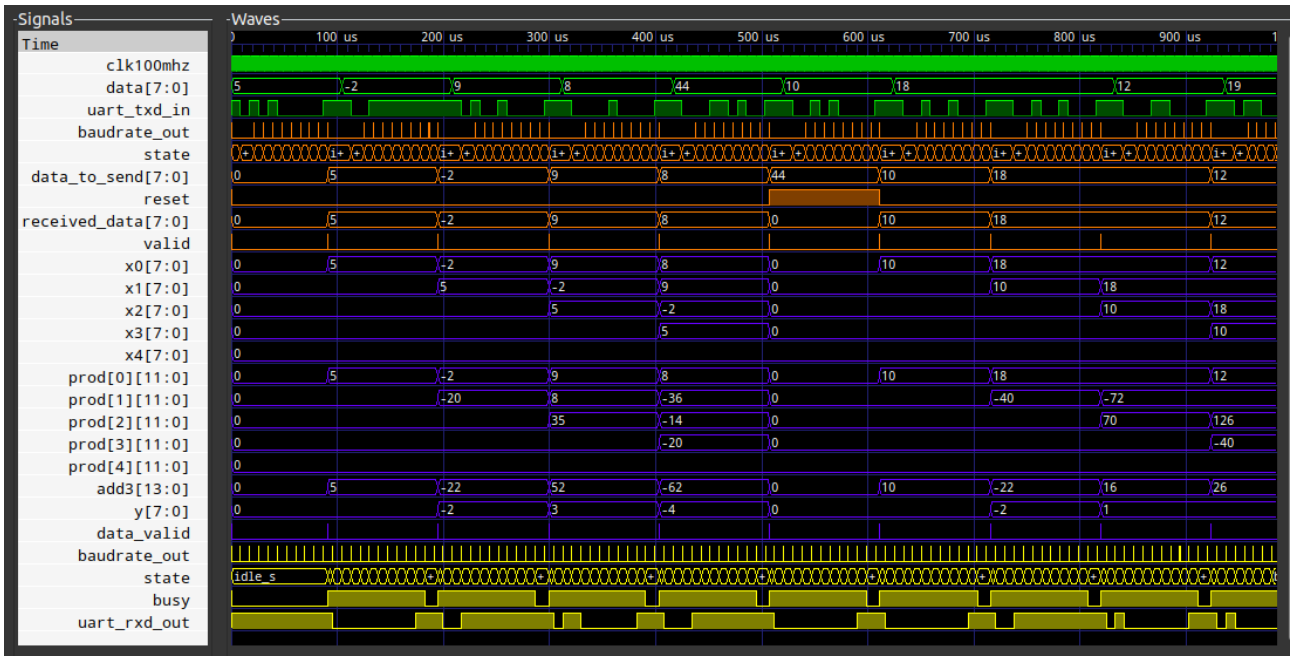

Figure 6: Gtkwave simulation of the complete circuit with the reset

References

[1] `https://dosits.org/galleries/audio-gallery/marine-mammals/toothed-whales/harbor-porpoise/`

[2] `https://www.americanscientist.org/article/the-acoustic-world-of-harbor-porpoises`