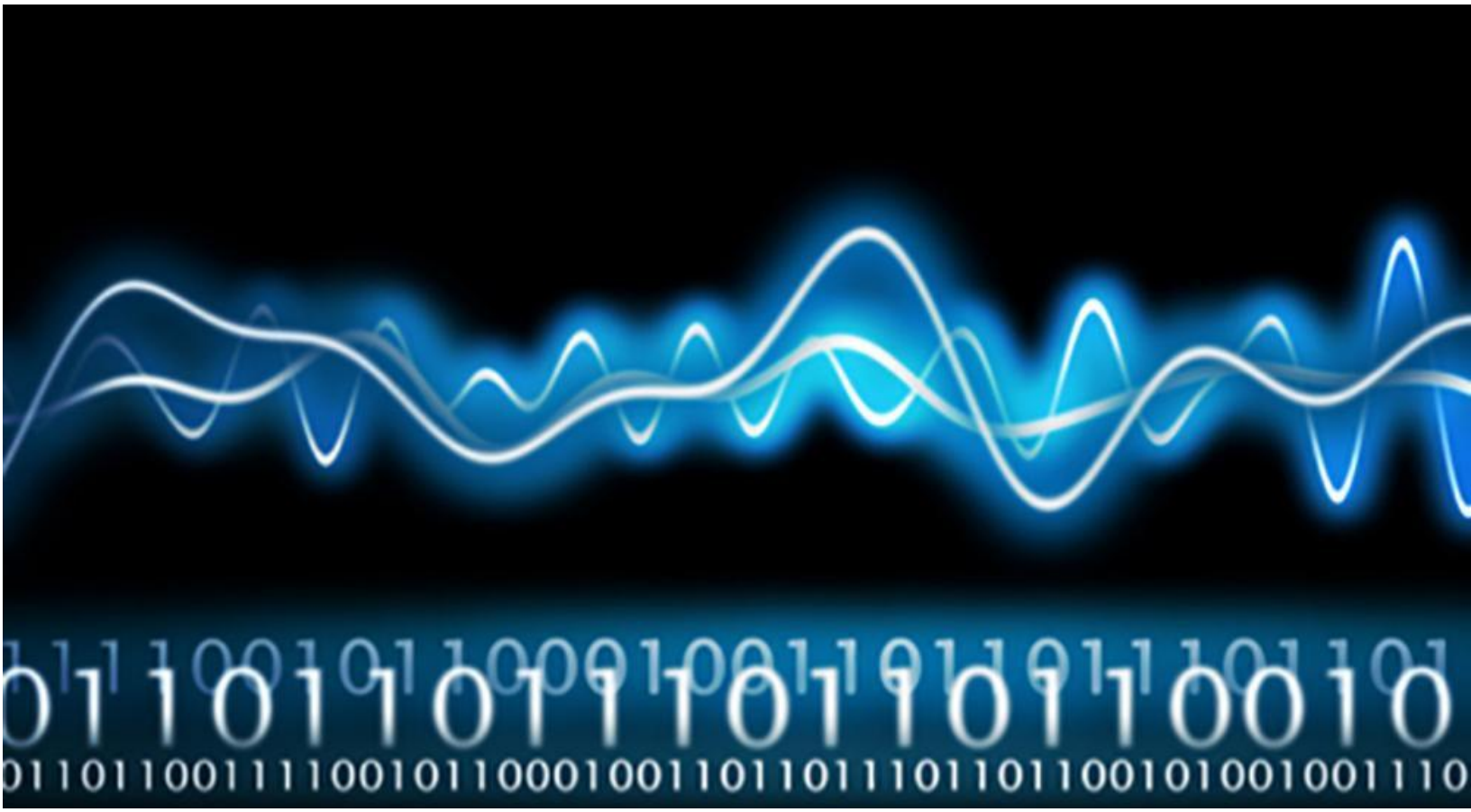
# Digital Signal Filtering Circuit



Name: Ciufudean Teodora – Gabriela

Group: 30433

# Contents

**Digital Signal Filtering Circuit** ..... 1
**1. Introduction** ..... 3
**1.1 Context** ..... 3
**1.2 Specifications** ..... 3
**1.3 Objectives** ..... 3
**2. Bibliographic study** ..... 4
**3. Analysis** ..... 5
**3.1 Use Case** ..... 5
**3.2 Data representation** ..... 6
**3.3 Components** ..... 6
**4. Design** ..... 7
**5. Implementation** ..... 12
**6. Bibliography** ..... 14

# 1. Introduction

## 1.1 Context

Digital Signal Filtering Circuit – the aim of this project is to design a circuit that can effectively process a sequence of input values X(i) based on a given formula. In our case the input values considered data from a "temperature sensor". We can apply three different formulas on the input sequence, depending on the user choice. Our project generates an output signal which represents the computed result. This is displayed on the board, to be easier for the user to see it. The Digital Signal Filtering Circuit, with some additional changes and improvements, may be used in such applications as noise reduction, video signal enhancement, graphic equaliser in hi-fi systems, and many other areas.

## 1.2 Specifications

This project will be implemented on a FPGA Basys3 development board, which provides a versatile platform for accommodating complex digital circuitry. The entire design, including data processing, filtering, memory storage, display will be constructed in VHDL, using the Vivado Design Suite. It is useful for code writing, simulation and testing. For implementation we will use some of the features of the board like switches, the digits of the seven-segment display, the internal clock, the components used for connecting with the computer.

## 1.3 Objectives

The primary objective is to create a functional Digital Signal Filter Circuit on the FPGA using VHDL, allowing users to select from three different filtering formulas (specified below in the next chapters) using the switches of the FPGA board and process an incoming stream of data and display the filtered output on a seven-segment display. Also, it's very important that code which effectively implements the digital signal filtering process to be correct and to obtain the desired result.

# 2. Bibliographic study

A digital filter is a computational algorithm or a hardware system that processes a discrete-time sequence of digital samples to perform operations like smoothing, noise reduction, signal separation, frequency response modifications, or other types of signal manipulation. It's designed to achieve desired changes in the frequency content or time-domain characteristics of a digital signal. A digital filter is characterized by its transfer function, or equivalently, its difference equation. Mathematical analysis of the transfer function can describe how it will respond to any input.

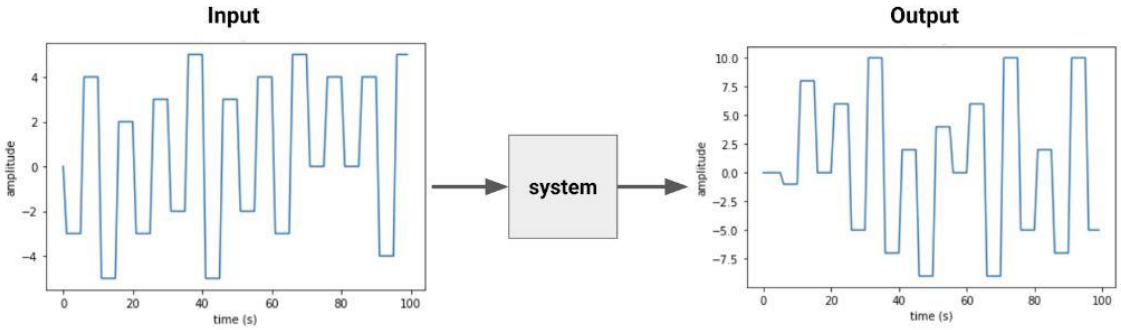



*Figure 1. Digital filters* [1]

Digital filters can be broadly classified into Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. FIR filters have a finite duration of response, while IIR filters have an infinite duration. There are various ways to characterize filters, like:

- linear filter - is a linear transformation of input samples. There are nonlinear filters, too. Linear filters satisfy the superposition principle.
- causal filter - uses only previous samples of the input or output signals, while a non-causal filter uses future input samples.
- time-invariant filter - has constant properties over time.
- stable filter - produces an output that converges to a constant value with time, or remains bounded within a finite interval. An unstable filter can produce an output that grows without bounds, with bounded or even zero input. [2]

Digital filters can be implemented in software (using programming languages), in specialized digital signal processors (DSPs), or within FPGAs for hardware-based filtering applications.

On the other hand, analog filters process continuous signals using electronic components, such as resistors and capacitors, and are often used in systems where continuous signal manipulation is crucial, such as audio systems. However, they may be more susceptible to noise, variations due to component tolerances, and environmental influences. Each filter type has distinct advantages and limitations, making their choice dependent on the specific needs and constraints of the application at hand.

# 3. Analysis

## 3.1 Use Case

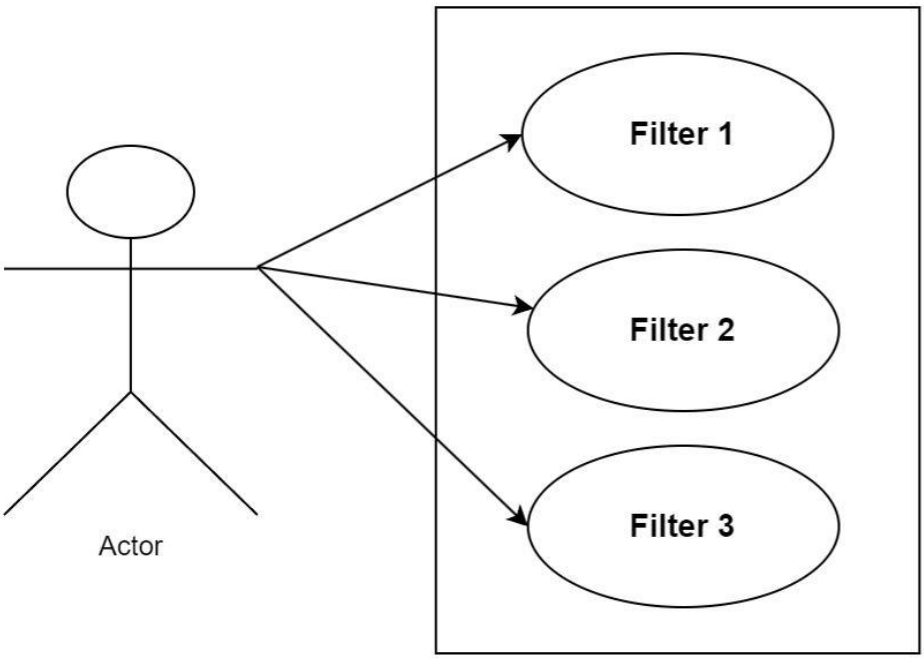



*Figure 2. Use Case diagram*

Based on the switches from the FPGA board, the user (actor) can choose the filter he wants to be applied on the input sequence. He has three options:

- Filter 1: acts like a combination between a high and a low pass filter; from the input values it displays only the values that are in the interval [10, 20], the ones that fall outside the predefined range are removed.
- Filter 2: computes the arithmetic average of the input sequence and display the result.
- Filter 3: is a recurrent formula $y_n = x_n * x_n + a * x_{n-1}$, where $y_n$ is the output signal, $x_n$ the input signal and *a* is a constant.

In all three cases, the input values are written in an input file, from where the input buffers should take them. The filter modifies the data and then other buffers (each filter has two FIFO: one for the input, respectively one for the output) picks up the final result and transmits it to an output file. It is also displayed on the board.

## 3.2 Data representation

I choose to work with 8 bits unsigned integer to represent the input data, because the range is [0, 255], so there are pretty much distinct numbers. My first choice was on 4 bits, but I could represent only few numbers in that case. Also, I decided to use unsigned values, because I am not interested in operations on negative numbers.

## 3.3 Components

Some important components used for implementation of the filters and transmission of data are briefly presented below.

- Carry Lookahead Adder

It is a type of adder used in digital circuit design to efficiently perform binary addition. It overcomes the delay inherent in the ripple carry adder by computing the carry signals in advance, reducing the propagation time of the carry to all the bits. It precomputes the carry signals using intermediate carry-generate (G) and carry-propagate (P) signals to determine the carry for each bit without waiting for the carry to ripple through the lower-order bits.

- Wallace Tree Multiplier

This technique is based on combining pairs of partial products with the help of multiple levels of Carry Save Adder. At each level of the tree, the numbers are grouped into three and are added together. Levels continue until two numbers are left to be added. A Carry Propagate Adder is used to add the last two numbers and deliver the final result. The total time is reduced significantly.

- AXI4 Stream

The AXI4-Stream protocol is used as a standard interface to connect components that wish to exchange data. The interface can be used to connect a sender, that generates data, to a receiver, that receives data. It works based on a ready / valid handshake. In case of AXI4-Stream modules which perform simple operations in just one clock cycle, the internal behavior can be synthetically represented as a two-state Finite State Machine (FSM). In one state, the module accepts data and performs the actual operation, while in the other state it provides the result at the output. [3]

# 4. Design

## 4.1 Block diagram:

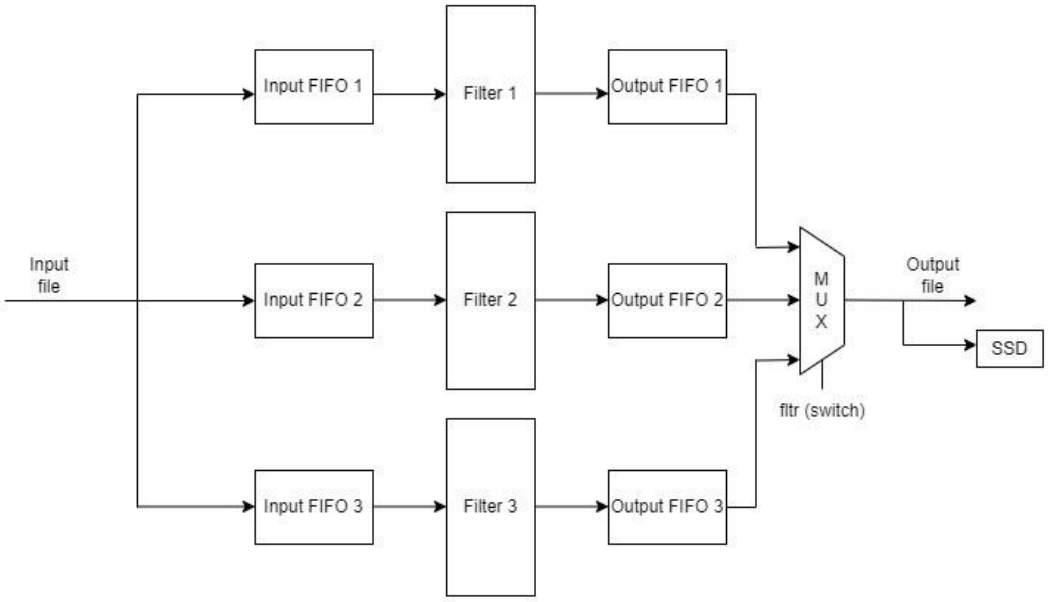



*Figure 3. Block diagram of the circuit*

- The diagram is schematic, to have an overview of the project.
- The files from the diagram are Stimulus Files used for Vivado Simulation. Reading signal values from file is an alternative way of generating stimuli for the design under test (DUT). This is typically useful when the design has to be tested using real data

such as values measured by sensors. The testbench sequence and timing is hard-coded in a stimulus file that is read by the VHDL testbench, line by line.

- FIFO-based data processing is performed using AXI4-Stream Protocol. It provides control of the data flow within the designed system and eliminate latencies caused by differences in frequencies that appear. There are some main signals of the interface like: ACLK – global clock signal, which "activates" the actions on the rising edge, ARESETn – global reset source, which is active low, TDATA – actual data, TVALID – indicator for valid data, TREADY – indicator of the receiver status). We use six buffers, three for temporary storage and transmission of the input, respective three for the output.
- There are three types of filters, each needing different number of inputs (at least eight inputs for the second formula and at least two inputs for the last one, the first one can work on any number, because it doesn't depend only on the current number).
- To implement the last two filters I need an adder and a multiplier. I use the CarryLookahead Adder and the Wallace Tree method for multiplication, because I studied and implemented them at the SCS laboratory. The CarryLookahead Adder is on 4 bits, so I have to make some changes. The multiplexer, whose selection is represented by the switches, helps in choosing the aimed result. It is placed on the diagram more to have an idea only, the multiplexer can be replaced with a decoder that chooses what formula to be applied. For division, in the case of arithmetic average, I make a shift right on three bits, the number of input values being a power of two (eight).
- The output is displayed on the seven-segment display and also in the output file.

## 4.2 Filter 1 block:

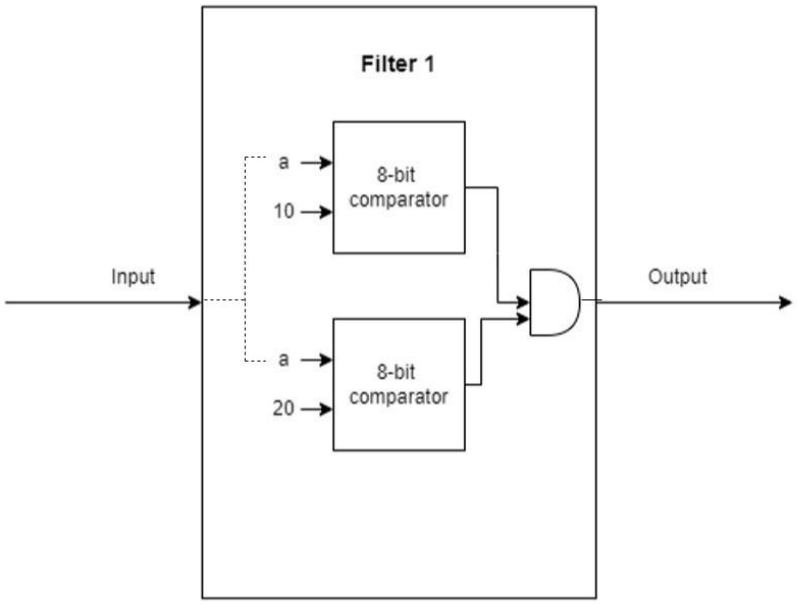



*Figure 4. First filter block diagram*

- This filter compares every input value with 10 and 20 to check if they are in the interval [10, 20]. Only those values that satisfy this condition are displayed.
- To implement it, two 8-bit comparators are used.

## 4.3 Filter 2 block:

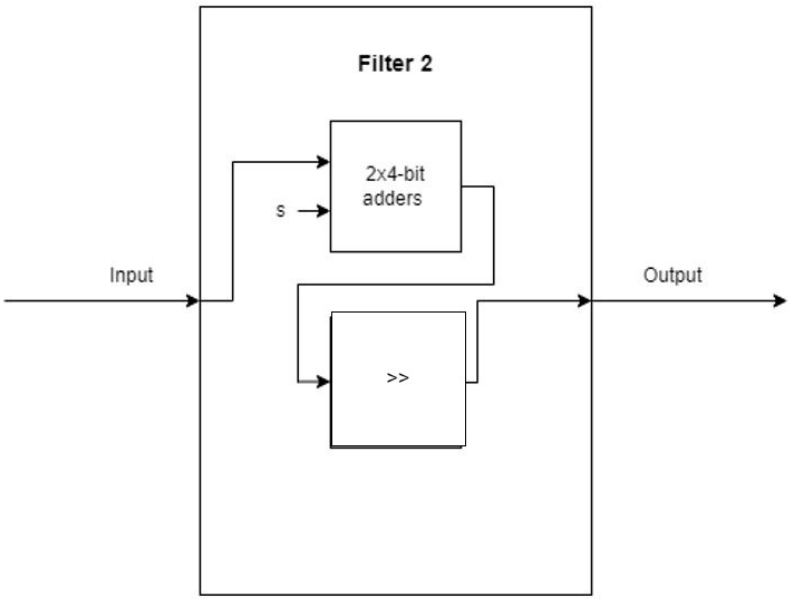



*Figure 5. Second filter block diagram*

- The second filter computes the arithmetical average of the input sequence (8 values). It makes the sum of all values and then shift it three bits right (because we have 8 numbers and 2 to the power 3 equals 8).
- For this, 2 Carry Lookahead Adders on 4 bits are cascaded and it's made a shifting.

## 4.4 Filter 3 block:

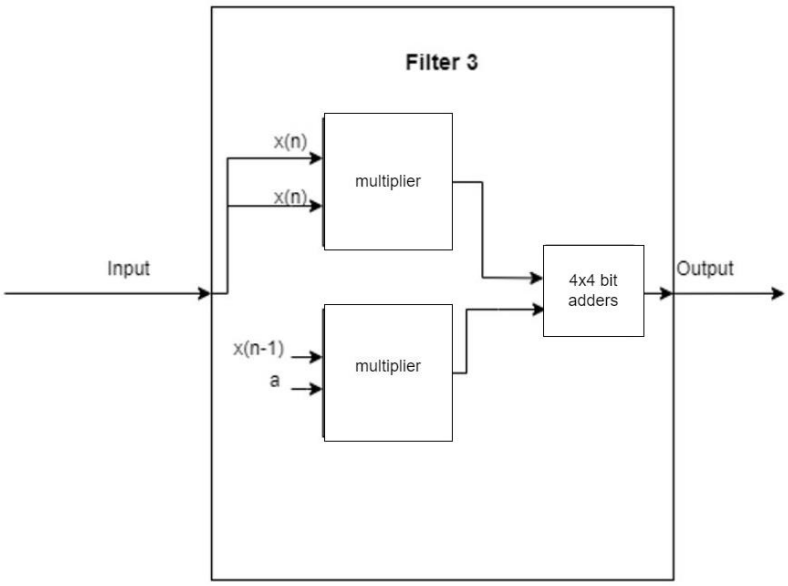



*Figure 6. Third filter block diagram*

- The last one works according to the recurrence formula: $y_n = x_n * x_n + a * x_{n-1}$. It computes the sum between the squared current input ($x_n$) and the product of a constant ($a$) with the previous input value ($x_{n-1}$).
- To implement this formula, I made the Wallace Tree multiplier on 8- bit input and 16-bit output and for the sum, we proceed as in the second filter, needing to divide the numbers in four parts.

## 4.5 Addition on 8 bits block:

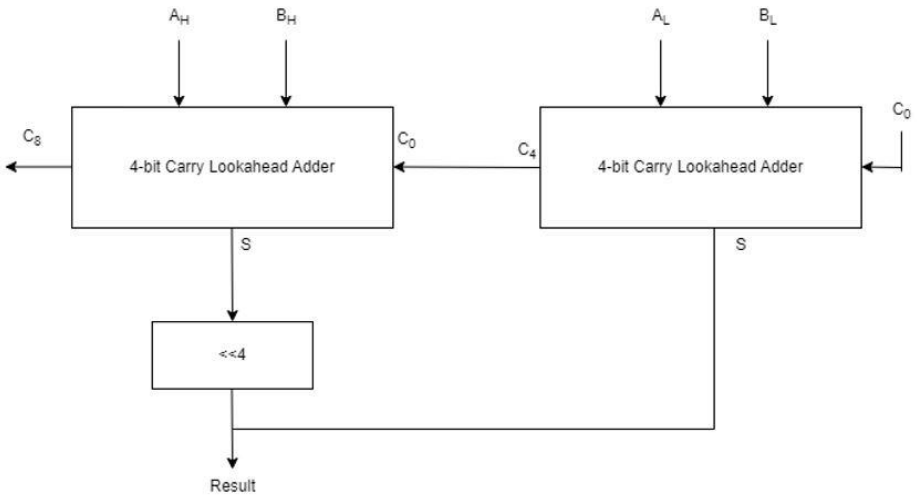



*Figure 7. Addition block diagram*

# 5. Implementation

- **Filter1**

This module acts as a filter based on two comparisons between the input data (s_axis_a_tdata) and predefined values (a and b). In order to do this I use an 8 bit comparator and if the input value belongs to the interval [a, b] (se1 = '1' and se2 = '1'), the result keeps it, otherwise the result is 'Z'.

Using Axi4 Streams, I have two-state machine (READ_OPERANDS, WRITE_RESULT) to manage the process flow. In the first state, it checks for input readiness and valid input data and make the transition to the second state if the conditions are met. In WRITE_RESULT it waits for the output to be ready before transitioning back to READ_OPERANDS to process new inputs.

- **Filter2**

This filter implements an arithmetic average using an 8 bit data window. To compute the sum I use a CarryLookaheadAdder on 4 bits, so I have to divide the the numbers in 4 bit segments (al, ah – low and high). I keep the partial sum in the result signal. Then I sum up the segments of the incoming data with the corresponding segments of the previous elements.

Like in the previous filter, the structure is the same, because I use Axi4 Streams protocol, too (so I have the two states, the signals for readiness, valid, etc.).

- **Filter3**

This module calculates a new value based on a mathematical equation using the incoming data and the previously stored value. The formula is $y_n = x_n * x_n + a * x_{n-1}$. For the multiplications I use WallaceTreeMultiplier which is on 8 bits. To perform the multiplication I use Full Adders like in the laboratory to sum the partial product and obtain the final result. I store the previous number in the result signal to be able to make the computation. I multiply it with a predefined value and then I sum the result with the squared value of the current input data. This final sum is on 16 bits so I divided the number in 4 segments in order to compute it.

Also, it is similar, talking about the streams process, with the modules described above.

- **Circuit**

This module is the big one, that includes and links all the other components. Here, each filter receives the input from its FIFO and transmits the output to the other FIFO. All the six FIFO were already implemented, I took them from the IP Catalog.

# 6. Testing

The following examples were used to test the functionality of the three filters:

- Filter1:


- at the beginning the result is always 0.
- $(01)_{16} = (1)_2$ which is not in the interval [10, 20], so the result is 'Z' (the same thing happens for 02, 09, 04, 05);
- $(0c)_{16} = (12)_2$ and it lies in the interval [10, 20], so the result takes this value (same case for 0b and 0a).

- Filter2:

$(01)_{16} + (0c)_{16} + (0b)_{16} + (02)_{16} + (09)_{16} + (0a)_{16} + (04)_{16} + (05)_{16} = (36)_{16} = (54)_2$. The average is 6 (in both representations).

The last value is the final result (0006).

- Filter3:


The result we are interested in is always the second one from the pair (when we take two numbers at the beginning the result is the first number, and only the second result is the right one).

- For $(01)_{16}$, $(0c)_{16} : (12)_2 * (12)_2 + (3)_2 * (1)_2 = (147)_2 = (0093)_{16}$ ;
- For $(0b)_{16}$, $(02)_{16} : (2)_2 * (2)_2 + (3)_2 * (11)_2 = (37)_2 = (0025)_{16}$ ;
- For $(09)_{16}$, $(0a)_{16} : (10)_2 * (10)_2 + (3)_2 * (9)_2 = (127)_2 = (007f)_{16}$ ;
- For $(04)_{16}$, $(05)_{16} : (5)_2 * (5)_2 + (3)_2 * (4)_2 = (37)_2 = (0025)_{16}$ .

# 7. Conclusions

As I said in the introduction, the aim of this project is to design a circuit that modifies some input data based on three formulas in order to obtain different results. I can say it was an interesting task and I liked to work at it.

The most challenging part I think it was to work with Axi4-Streams, because it was something new for me. With VHDL I was pretty used from the previous years, but I did not really like this language, so I was a little bit scared of this project at first.

Unfortunately, I did not manage to put it on the FPGA board, so it can be run only in the simulation.

# 8. Bibliography

[1] Isabel Kaspriskie, “Demystifying Digital Filters”, [Online], https://cycling74.com/tutorials/demystifying-digital-filters-part-1, 2021.

[2] “Digital filter”, [Online], https://en.wikipedia.org/wiki/Digital_filter.

[3] “AXI4-Stream Communication Protocol. Introduction” (Laboratory work 6), [Online], https://moodle.cs.utcluj.ro/mod/resource/view.php?id=47825 .

[4] “Vitis High-Level Synthesis User Guide (UG1399)”, [Online], https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Interfaces-for-Vivado-IP-Flow.

[5] Dimitar Marinov, “FIR Filters”, [Online], https://vhdlwhiz.com/category/fir-filters/, 2022.