UNIVERSITÀ DI PISA

MSc Computer Engineering
Electronics & Communication Systems

Square Root Raised Cosine FIR Filter

(Digital Circuit)

Author: Gaetano Sferrazza

Academic Year: 2024/2025

Table Of Contents

Introduction

In signal processing, a Finite Impulse Response (FIR) filter is a type of digital filter whose impulse response settles to zero within a finite duration. This behavior contrasts with Infinite Impulse Response (IIR) filters, which may include internal feedback and potentially produce non-zero output indefinitely.

FIR filters are inherently stable and exhibit a linear phase response when designed symmetrically, making them highly suitable for many digital communication applications.

The output of a causal, discrete-time FIR filter of order N is computed as a weighted sum of the current and the N most recent input samples:

$$y[n] = c_0 x[n] + c_1 x[n-1] + \ldots + c_N x[n-N] = \sum_{i=0}^{N} c_i x[n-i]$$

where:

- $x[n]$ is the input signal,
- $y[n]$ is the output signal,
- c_i are the filter coefficients (impulse response values),
- N is the filter order.

This process is known as **discrete convolution**, and it can be implemented through various architectures depending on design constraints.

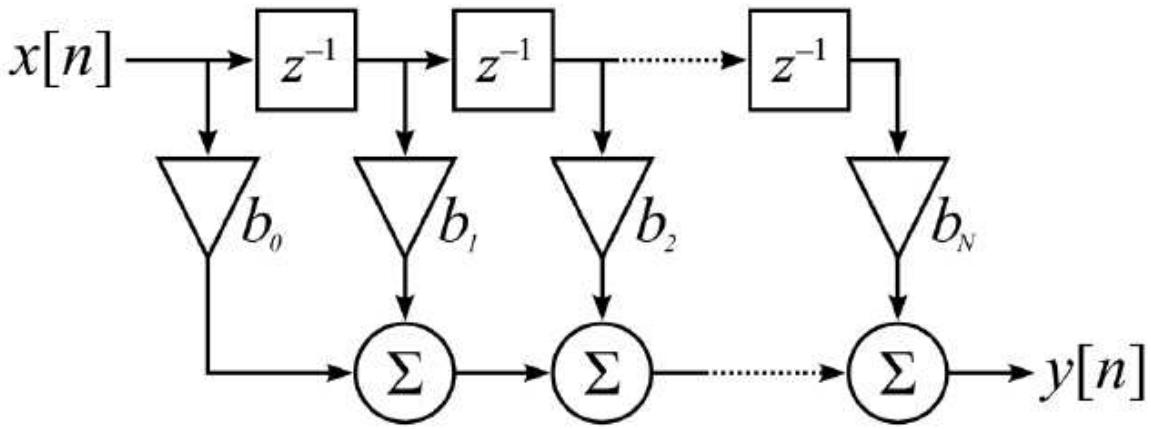

Figure 1: FIR Filter Functional Convolution Schematic – Source: [3]

Application Domains

The FIR Filter, specifically for the filter under study *Square Root Raised Cosine* (SRRC) filter, plays a fundamental role in modern digital communication systems. It is primarily used for pulse shaping, a process aimed at minimizing intersymbol interference (ISI) and limiting the signal bandwidth while preserving symbol integrity.

Below are the most common application domains in which the SRRC filter is employed:

- **Digital Modulation Schemes (e.g., QPSK, QAM):** The SRRC filter is widely used in digital modulators to shape the transmitted symbols and control spectral occupation. It ensures that the transmitted pulses do not interfere with each other, which is crucial for maintaining a low bit error rate.

- **Matched Filtering to Maximize SNR:** In digital receivers, the SRRC filter is used as a matched filter to maximize the signal-to-noise ratio (SNR) in the presence of additive white Gaussian noise (AWGN), thus improving detection performance.

- **Intersymbol Interference (ISI) Mitigation:** One of the primary motivations for using the SRRC filter is to mitigate ISI. In telecommunication is a form of distortion of a signal in which one symbol interferes with subsequent symbols. This is an unwanted phenomenon as the previous symbols have similar effect as noise, thus making the communication less reliable. Therefore, in the design of the transmitting and receiving filters, the objective is to minimize the effects of ISI, and thereby deliver the digital data to its destination with the smallest error rate possible. By shaping pulses such that they are zero (or nearly zero) at symbol decision points other than the intended one, the filter ensures accurate symbol recovery.

- **Pulse Shaping in Bandwidth-Constrained Channels:** In electronics and telecommunications, pulse shaping is the process of changing the waveform of transmitted pulses. SRRC filters help meet strict spectral requirements by controlling the roll-off of the transmitted signal, making them suitable for applications that operate under bandwidth limitations.

- **Wireless Communication Systems (e.g., LTE, 5G, DVB, Wi-Fi):** Many modern wireless standards rely on SRRC filtering at the transmitter and receiver to maintain spectral efficiency and reduce adjacent channel interference.

- **Optical Fiber Transmission:** In high-speed optical communication systems, SRRC filters are used to limit bandwidth and prevent signal spreading that could lead to ISI, ensuring higher data integrity over long distances.

- **Software Defined Radio (SDR):** In SDR platforms, where baseband processing is performed via programmable logic or software, the SRRC filter is essential for generating clean baseband waveforms before upconversion.

- **Channel Selection and Separation:** In multichannel systems, SRRC filters can aid in isolating the desired channel or signal component, especially when combined with other filtering stages in the receiver chain. For instance, a digital radio receives and converts the analog signal to the intermediate frequency and then converts it to digital. Then uses the finite impulse response to choose the preferred frequency.

FIR-Filter Architectures

As mentioned before, the convolution performed by the FIR-Filter can be realized through different architectures, the optimal one depending on the specific requirements of the application, including desired *performance*, *hardware* constraints, *speed*, and *power consumption* considerations.

FIR filter architectures can be broadly categorized into **Direct Form**, **Systolic**, and **Distributed Arithmetic (DA)** based architectures and another that can be useful to take into accounte is the **Associativity** one. These architectures differ in their hardware implementation, computational efficiency, and suitability for various applications, particularly in FPGAs and ASICs. Based on these observations, analysis were then made for the best choice of architecture for the purposes of this project for the circuit of the ***SRRC-FIR-Filter***:

1. **Classic (Direct Form) FIR Architecture:**
   A straightforward implementation using a chain of *multipliers*, *delay elements*, and *adders*. For an N-tap filter, this architecture requires N multipliers, (N-1) adders, and (N-1) delay elements. It is simple and intuitive but may suffer from a longer critical path and lower performance at high frequencies.

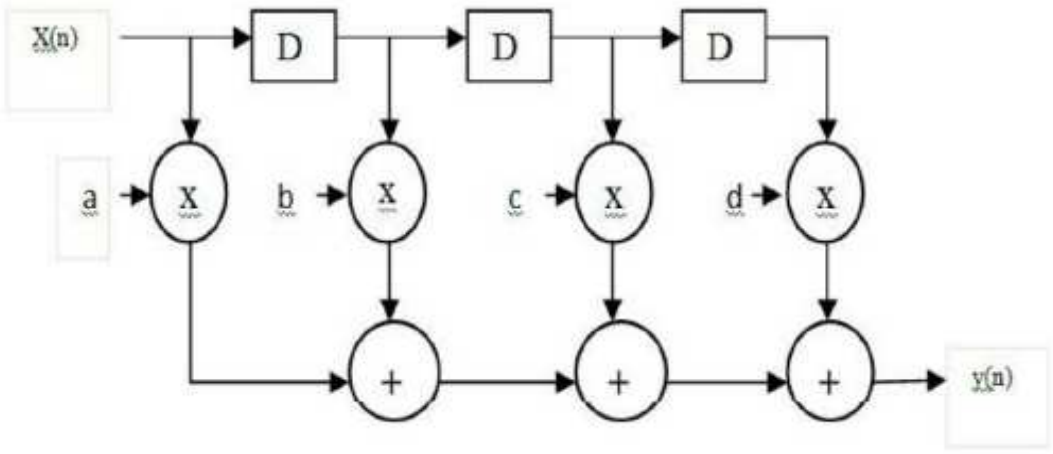

Figure 2: Classic FIR-Filter architecture functional schematic – (4 tap filter) – Source: [1]

2. **Systolic Architecture:**
   Involves a regular array of Processing Elements (PEs) which process and transfer data in a pipelined fashion. This structure greatly reduces the critical path and enhances throughput, making it suitable for high-speed applications. The systolic design can also be mapped efficiently onto FPGAs due to its regularity.

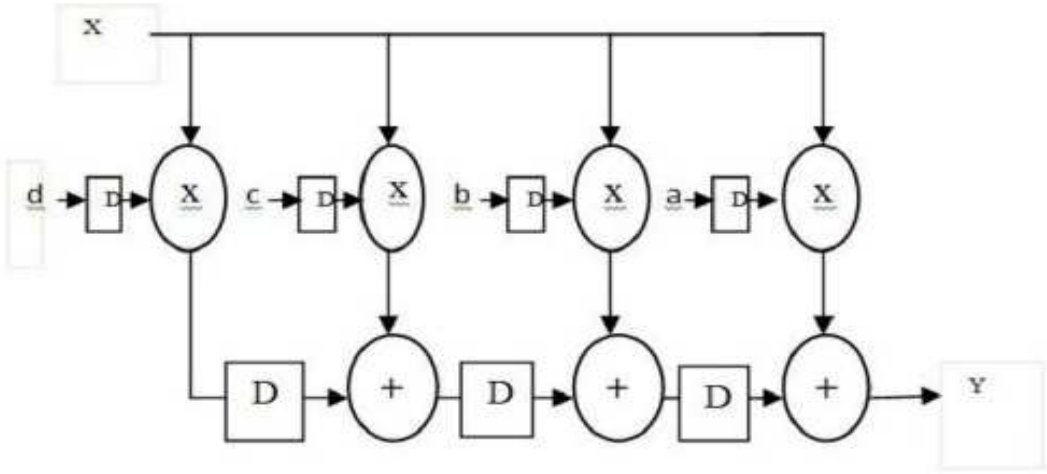

Figure 3: Systolic FIR-Filter architecture functional schematic – (4 tap filter) – Source: [1]

3. **Distributed Arithmetic (DA) based Architectures:**
   DA-based FIR filters replace multiplication with look-up tables (LUTs) and additions, leading to efficient hardware implementation, especially for high-order filters. Specifically LUT-based DA uses pre-computed values stored in LUTs to perform multiplications.

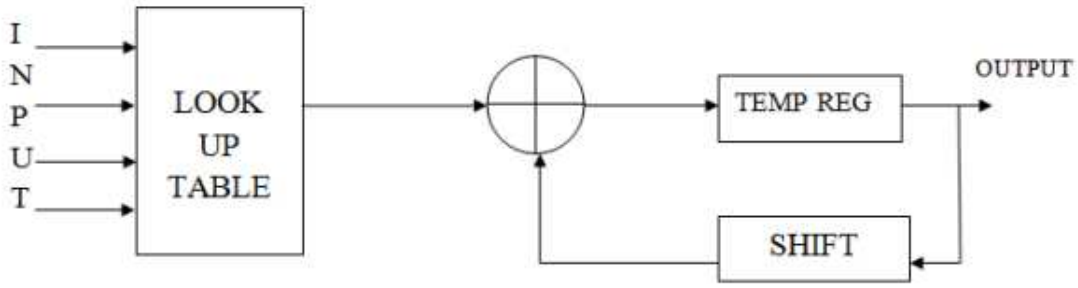

Figure 4: Distributed Arithmetic (DA)-based FIR-Filter architecture functional schematic – Source: [2]

4. **Associativity Architecture:**
   Leverages the associative property of addition to restructure the summation process in a way that can reduce the depth of the adder tree. This can further reduce latency and enable parallelism, though care must be taken in implementation to avoid timing imbalances.

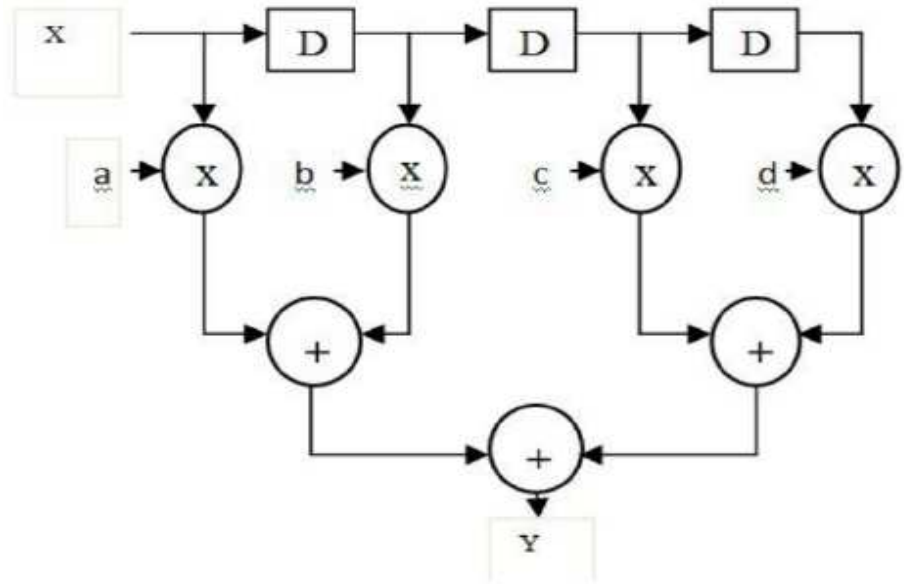

Figure 5: Associativity FIR-Filter architecture functional schematic – (4 tap filter) – Source: [1]

SRRC Filter Architecture - Description

In this project has been considered a digital circuit which realizes FIR (Finite Impulse Response) filter with a SRRC (Square Root Raised Cosine) impulse response and with the following characteristics:

- Order of the filter N = 22
- Samples per symbol = 4
- A roll-off factor of 0.5

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

Input, **output** and **coefficients** are represented in **16-bit**. The coefficients have this values: *ck : c0 = c22 = -0.0165 ; c1 = c21 = -0.0150 ; c2 = c20 = 0.0155 ; c3 = c19 = 0.0424 ; c4 = c18 = 0.0155 ; c5 = c17 = -0.0750 ; c6 = c16 = -0.1568 ; c7 = c15 = -0.1061 ; c8 = c14 = 0.1568 ; c9 = c13 = 0.5786 ; c10 = c12 = 0.9745 ; c11 = 1.1366.*

In the **Figure 6** it is possible to see the schema of the interface of the circuit to be designed, where inputs (c_k, x, reset, clock) and outputs (y) are specified.

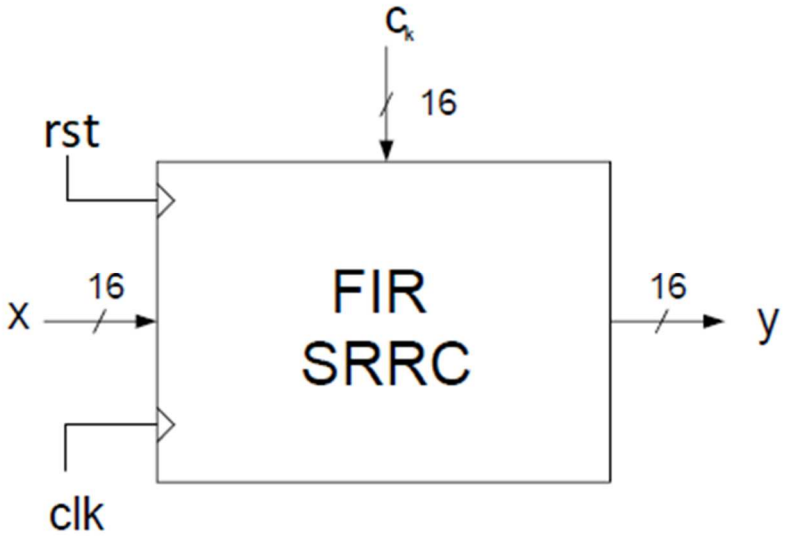

Figure 6: SRRC-FIR Filter Circuit Interface

On the basis of what was observed in the analysis of the various possible architectures that could be used for the realization of this circuit, it was kept in mind that this choice is strongly influenced by the filter order and a set of hardware design constraints, including logic resource usage, maximum clock frequency, latency, power consumption, and overall implementation complexity.

For filters of very high order, advanced architectures such as systolic arrays or Distributed Arithmetic (DA) are often preferred due to their ability to efficiently parallelize computation and minimize critical paths. These approaches offer excellent throughput and are well suited for real-time high-speed applications or highly resource-constrained environments. However, in the context of this project, the the primary focus was on functional correctness, timing compliance, and design clarity, rather than on extreme resource optimization or ultra-high performance.

Therefore, a **pipeline-optimized direct-form architecture** was adopted. This choice balances hardware simplicity, timing optimization, and low latency, while avoiding the additional complexity and design overhead that would come with Systolic or DA implementations.

This kind of architecture is based on a refined version of the classic Direct Form FIR structure, enhanced with several key optimization strategies derived from the analysis of different architectural paradigms. In particular, the design introduces:

- A **fully-pipelined structure**, with regesters from one step to another, aimed at minimizing the ***critical path*** and increasing throughput, in line with techniques typically employed in systolic designs.
- The use of an **adder tree** to perform additions efficiently, reducing the depth of combinational logic and enabling better timing performance.
- An initial **pre-multiplication** stage that takes advantage of the **coefficient symmetry** of the SRRC filter, which is clearly noticeable. This allows **summing symmetric** input samples before multiplication, effectively reducing the number of required multipliers, due to the filter order length, and thus optimizing area usage.

This hybrid architecture represents a balanced **trade-off** between speed, area, and latency, addressing the main project constraints. It reflects a design choice informed by the strengths of different FIR implementations, combining the structural clarity of the classic form, the timing efficiency of pipelining, and the computational savings enabled by coefficient symmetry.

Mainly stages in the architecture:

- **Symmetrical Sums Stage:**
  In this stage, the input samples are paired symmetrically based on the filter's impulse response symmetry. If the coefficients are denoted as $c[0], c[1], \dots c[N-1]$ the SRRC filter exhibits even symmetry, i.e. :

  $$c[i] = c[N-1-i] \rightarrow (for\ all\ i)$$

  This allows us to **pre-add** the corresponding symmetric input samples:

  $$x[i] + x[N-1-i]$$

  These sums are then used in place of the individual samples in the multiplication stage. This technique reduces the number of multiplications by almost **half**, minimizing both hardware complexity and area usage, *without affecting* the result.

  Since the **central** sample is not coupled with anyone it will go directly to another register maintaining alignment with the whole pipeline being multiplied and summed correctly in the next stages.

- **Multiplication Stage:**
  Each of the **symmetrical sums** computed previously is now **multiplied** by the corresponding **coefficient** from the first **half** of the coefficient set:

$$y_{partial[i]} = (x[i] + x[N-1-i]) * c[i]$$

  If the filter had N=22+1 coefficients, the number of multipliers would be reduced from 23 to 12. This optimization is particularly effective for this SRRC Filter, due to its linear-phase characteristic.

- **First Sum Stage:**
  This stage begins the process of summing the partial products from the multiplication stage. The results are summed two by two leveraging the **adder tree** structure to keep the logic depth minimal. This step produces intermediate values to be further reduced in the next summation levels

- ***Second* and *Third* Sum Stages:**
  These are **intermediate** steps in the **adder tree**, where the outputs from the previous level are combined again, with each stage halving the number of active sum lines. This structure ensures balanced timing and pipeline regularity, contributing to overall speed.

- **Fourth Sum Stage:**
  This is the **final summation stage** of the adder tree. Here, the last two intermediate results are added to produce the final **filter output** sample. The result is then propagated through the pipeline to the output register, completing the fully pipelined filter cycle.

The following **Figure 7** below shows the block diagram representing the structure of the fully-pipelined architecture by highlighting the details of the circuit components.

Circuit Block Diagram

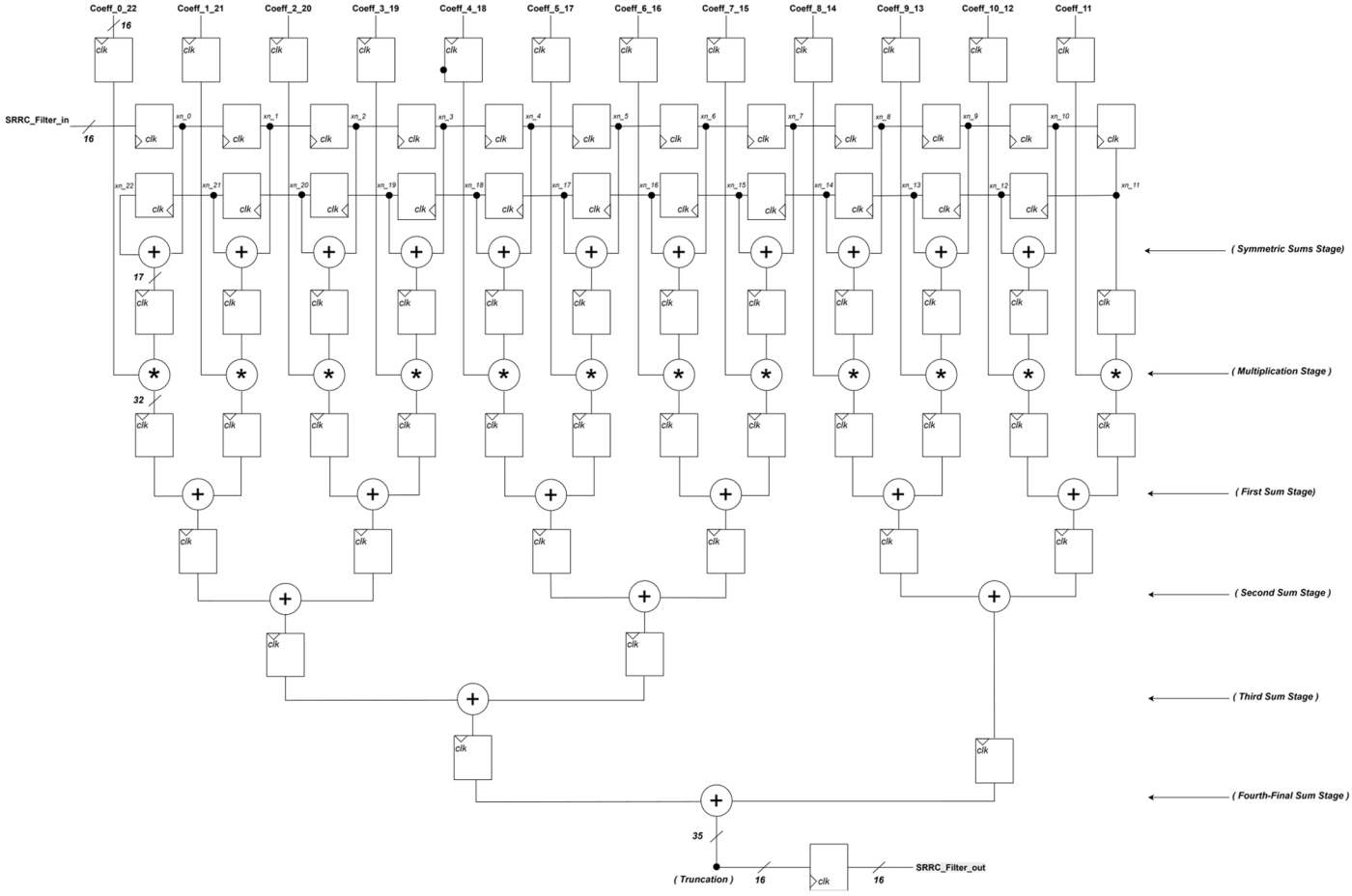

Figure 7: Block Diagram of the fully-pipelined architecture for SRRC_Filter circuit

VHDL Description

The code used to describe the circuit is VHDL by using an asynchronous reset positive edge triggered D flip flop and in this section we will see the different choice for code organization and Bit-Width selection. Also the testbench approach and the various tests conducted.

Arrays of std logic vectors have been considered in order to use the "**GENERATE**" and "**FOR**" statements, accessing signals with indexes instead of defining all the operations by hand.

The Inputs/Output ports defined are:

- **clk**: std logic representing the clock;
- **reset_n**: std logic representing the reset; When it's equal to '0' all the registers are "cleared".
- **x_in**: 16 bits std logic vector representing the filter's input;
- **y_out**: 16 bits std logic vector representing the filter's output;
- **coef_i_N-1**: 16 bits std logic vector representing the coefficient with labal used to emphasize the symmetry of the coefficients. There are 12 like this where the last one has no pair so it is simply c_11_11.

Bit-Width & Quantization Choice

This section will show all the observations made for the correct choice of bit-size that was decided to use to try to best handle situations related to possible ***overflow***, ***saturation***, and ***signed values*** for the various stages of the pipeline considering the limit cases that can be calculated through the known values in realtion to the operations between the inputs and how the ***truncation*** was decided to be done.

It is required to design a symmetric 23-tap FIR filter (12 useful products due to symmetry) and this was decided to be done by using **fixed-point** logic, with objective to minimizes bit-width as much as possible while ensuring no overflow and try to reach high accuracy as already said before.

Specifically the choice involves:

- Input in Q8.7 (16-bit):
- Coefficients in Q1.14 (16-bit):
- Output in Q11.4(16-bit):

For the input that choice it was made to allow a range representation [−256, +255.9922] and for the coefficients to allow a representation in range [−2, +1.9999], since as can be seen the **central coefficient** that, even if slightly, exceeds the value " 1 " so it was necessary to extend the range for the integer part while still leaving high precision for the fractional part, otherwise they would have been too approximate.

As it is, with this information, it would be possible to make calculations to figure out what would be the **maximum value** of the products in the convolution. But given the choice to perform **symmetric sums** before the multiplication stage it becomes necessary first to

understand what would be the maximum value in the symmetric sums and to be sure what the actual maximum attainable value in the product would be.

Accordingly we will have (**worst-case evaluation**) considering that each symmetric sum involves two **Q8.7** input values, wich in turn they vary between ±256 range as we can see in (1) equation:

$$(1)\ Sym\_Sum_{max} = 255.9922 + 255.9922 = 511.9844$$

So to guarantees safe representation of all possible symmetric sums could be a good approach to adopt **17-bi**t to contain the result in **Q9.7**, specifically 1 for sign bit, 9 for integer part (up to ±512) and 7 for fractional part (like input Q8.7).

At this point we can proceed to calculate the maximum possible value we will be dealing in multiplication stage with more accuracy, which is in follow equation (2):

$$(2)\ Product_{max} = 1.1366 \times 511.9844 \approx 582.21;$$

$$Prodact_{max} < 2^{10} = 1024$$

Considering that the multiplication is performed between values represented in **Q1.14** – *Coefficeitns* – and in **Q9.7** – *Summetric Sums* – the choice of the best bit-with ends up leading us to the **Q10.21**, that means 10 for integer part, 21 for fractional part and 1 for sign bit for **32-bit** total, that results sufficient for the products based on what we address so far.

Now it's the turn of the accumulation stage done in **adder-tree** mode that will have to deal with adding up all the products obtained from the previous stage, the multiplication one which are 12 in total considering the symmetry, and then we're going to check how potentially the results could grow so that we can figure out again what may be the best choice to make.

So the accumulator must handle the following situation, preserving 21 fractional bits, as you can see in (3):

$$(3)\ \sum_{i=0}^{11} |\ ci \cdot xi\ |\ \approx 582.21 + smaller\ terms \approx 2495$$

This means that we can consider something that reguard a total sum roughly up to ±2500. ***Small terms*** refers to the contribution made by the 11 remaining sums with an intentional overestimation of the mean value of the coefficients to remain more conservative Therefore it is reasonable in this case take into account the following approach:

- Integer part: 12-bit ($2^{12} = 4096 > 2500$)
- Fractional: 21-bit (To maintain the fractional part precision)
- Sign bit 1

So that the final choice ends up leading us to state that **34 bits** ar sufficient to handle the situation, just for safety we can add 1-bit more and we obtain finally the total size, that we can use for semplicity into **all stages** after the multiplication stage until the end of the pipeline, where the value will be truncated to 16-bit, which is **35 bits** in **Q13.21.**

The final output will be **truncated**, without rounding, in 16-bit as already said and specifically we should achieve this following condition to avoid information loss:

- **Final output:** 1 sign + 11 integer + 4 fraction = 16 bits

We need to truncate the fractional part from 21 to 4 bits, so was performed a right-shift of **17 bits**.

It is worth saying that initially a different decision had been made, to use a different format from the already introduced **Q11.4** trying to remain consistent with the input in **Q8.7**, but this gave problems in testing by generating loss of information and premature saturation with the tests for large numbers. for this reason it was decided to think of a **trade-off** by slightly reducing the precision on the fractional part in favor of a higher gain for the integer part, which is essential to represent correctly without loss of information even for large numbers while still leaving a good precision also for the fractional part.

So the final consideration that it is possible to do is that this fixed-point configuration, which employs 35-bit **accumulators** and a clean truncation strategy toward a Q11.4 **output**, ensures both hardware efficiency and numerical reliability. By using Q9.7 for **symmetric sums** and Q10.21 for **multiplications**, the design guarantees full precision throughout the signal path while avoiding any risk of overflow, even under worst-case conditions.

Test-Plan

In order to verify the SRRC filter's correct functioning has been conducted the following tests:

1. The asynchronous reset.

2. All the ***limit cases*** of the input to see if there are some strange behaviours.

3. The ***behaviour*** after a sudden change of input, for example with a different sequences of values for the ***dynamic*** filter evaluation. When a new input is applied each cycle, past responses overlap—just like in real signals—so the filter then accumulates the effect of past values consequently and the output becomes a time-distributed sum of all convolved inputs.

4. If the ***frequency*** components are correctly handled in ***typical scenarios*** related to:

    - **Zero ISI** (Inter-Symbol Interference)
    - **Out-band Frequency**
    - **In-band Frequency**

Testbench - Modelsim/Python Simulation

Two types of Simulation Tool have been used:

- **ModelSim:** ModelSim is a digital circuit simulator used for functional verification of hardware designs described in VHDL, and the results obtained through this simulator are the ones that tell us whether the circuit design may have problems and allows us to test its behaviors as if it were in Hardware. To perform the tests by this tool it has been considered a "**wrapper**" where the coefficients are specified used into the testbench files written in vhdl, that specifically are "**SRRC_Filter_boundery_tb.vhd**" and "**SRRC_Filter_waveform_tb.vhd**".

- **Python:** In order to check the correctness of the test, a couple of python scripts has been realized and used as co-simulator supporter. With this scripts it is possible to build the coefficients and the values used in the test in **fixed point** resambling the design of the SRRC-Filter, as the vhdl description realized, and it computes the filter outputs "**analytically**" using "**Fxp**" library for a group of the different tests conducted and library as "**Scipy**" for the others. According with the two files previously mentioned for modelsim sumulation the scripts are "**srrc_boundery_tests.py**" and "**srrc_waveform_analysis.py**".

To test VHDL description, the testbench conducted follow the so called "**Self-Checking Testbench**" where the actual response is compared to the expected response with the below workflow:

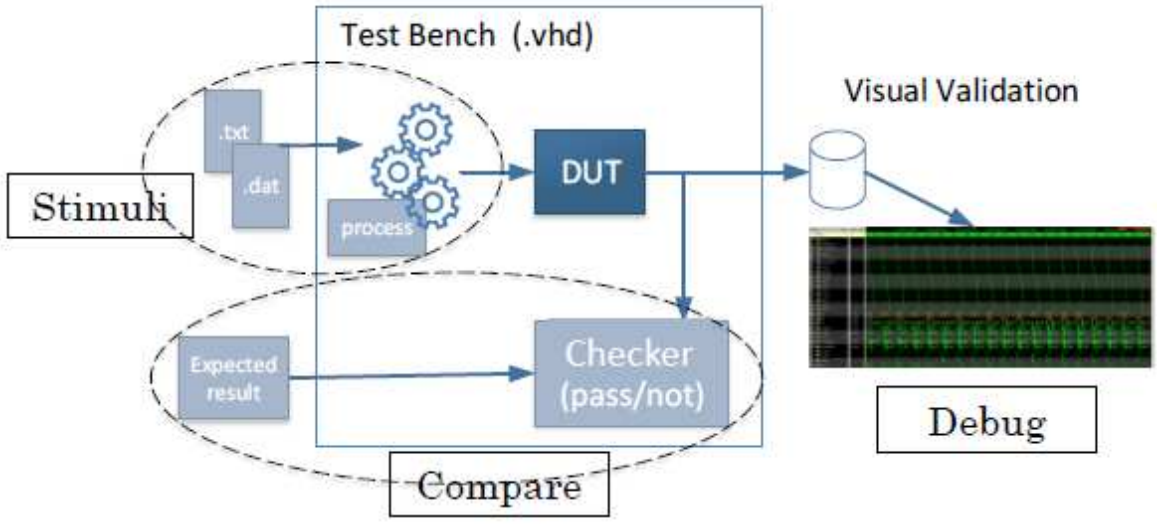

Figure 8: Self-Checking Testbench

Consist of generation of **stimuli** for simulation (waveforms, test vector, stimuli file) that than is applied to the **Design Under Test** (DUT) and monitoring the **output responses** to compare it with expected known values.

For the test, a period of 8 ns has been considered for the clock and then the tests was checked.

The **Figure 9** below shows that the ***reset*** test behaves as expected, meaning that no output is produced while the reset signal is active (logic '0').

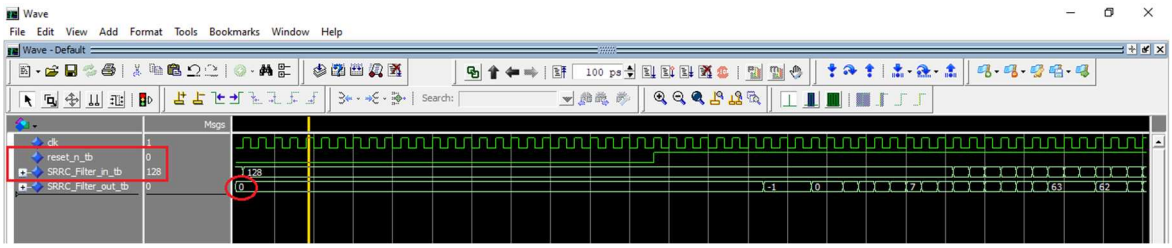

Figure 9: ModelSim Simulation response for reset check

The two figures below, namely F**igure 10** and **Figure 10.1**, show the results of the simulations performed for the test related to the SRRC filter output when the input is ***" 1 "*** for the convolution. The Python simulation shows the expected result we should obtain from the simulation run on ModelSim, to assess if the circuit design is correct. By observing the results, we can say that the test is successful passed because the outputs are the same. Specifically, the test below checks whether there is a match for the "***peak value***" of the impulse response.

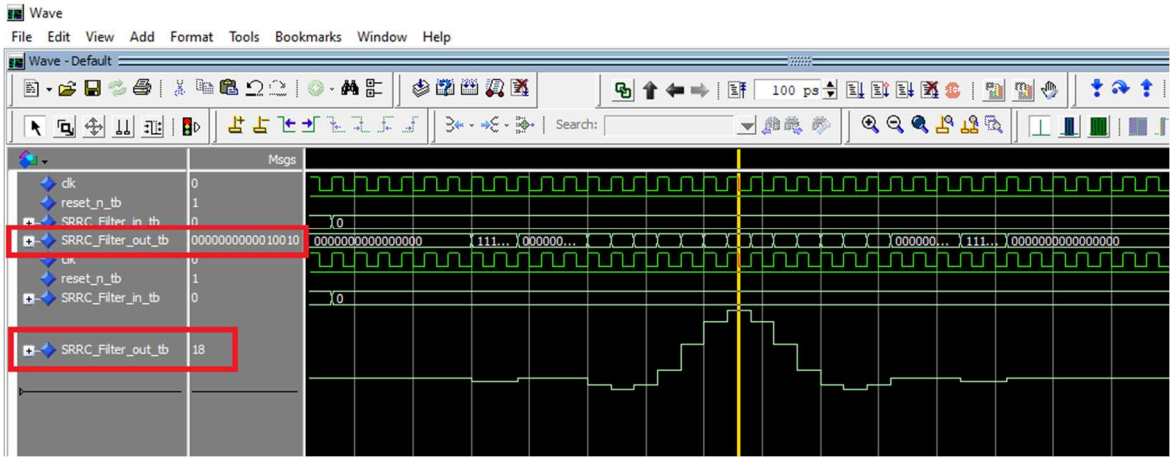

Figure 10: ModelSim Simulation response for " 1 " as input Test – ***(Peak Value Check)***

```
#-------------------------------------------------------------------------------
#       1) Analysis of the Central Coefficient Value (Impulse Response Peak)
#-------------------------------------------------------------------------------

        - @ Input Value Details -> 1) Real: 1.0
                                   2) Q8.7 Format: 128

        - @ Central Coefficient -> 1) Real: 1.1366
                                   2) Q1.14 Format: 18622

-- @ Expected Values @ --

1) Real Value of the Convolution (Peak) is [ 1.0 * 1.1366] : 1.13660
2) Scaled Value of the (Peak) is [ (1.0 * 2^7) * (1.1366 * 2^14) = 128 * 18622 ] = 2383616

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the Convolution (Peak) (35-bit): 00000000000001001000101111100000000
4) Full-Raw integer value of the Convolution (Peak) (Fixed Point): 2383616

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out (Peak Value)            : 0000000000010010
6) Raw Integer SRRC_filter_out (Peak Value)       : 18
7) Real SRRC_filter_out Value (Peak Value - Q11.4) : 1.125
-------------------------------------------------------------------------------
```

*Figure 10.1: Python Simulation for "1" as input Test – **(Peak Value Check)***

The two figures below, namely **Figure 10.2** and **Figure 10.3**, for the same input configuration as the previously test explained, instead show the results of the simulations to checks whether there is a match for the "***total sum***" of the convolution value.

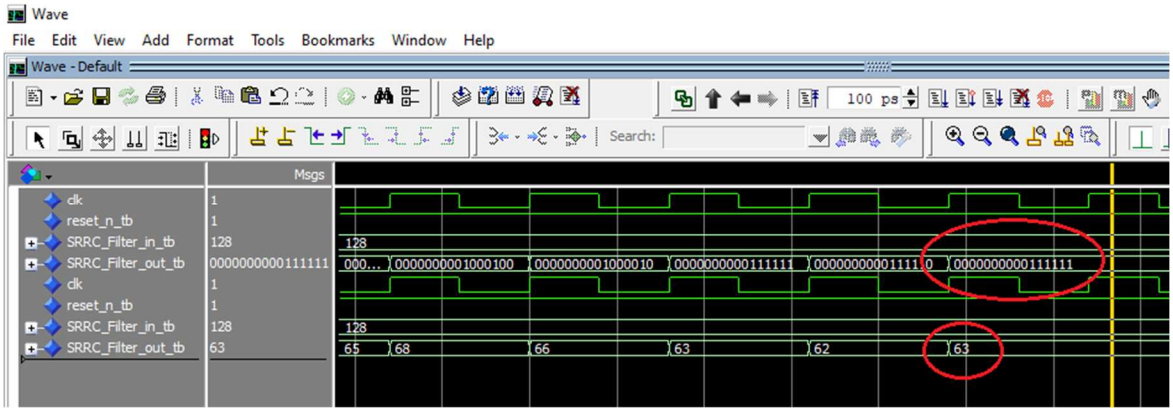

*Figure 10.2: ModelSim Simulation response for " 1 " as input Test – **(Total Sum of Convolution Check)***

```
#-------------------------------------------------------------------------------
#       2) Analysis of Total Sum Result (Full Convolution Value)
#-------------------------------------------------------------------------------

-- @ Expected Values @ --

1) Real Value of total sum with 1.0 as input is: 3.96423
2) Scaled Value (Q13.21) of total sum with 128 as input is = 8313600

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the total sum (35-bit): 00000000000011111101101101100000000
4) Full-Raw integer value of the total sum (Fixed Point): 8313600

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out Value (Total Sum)       : 0000000000111111
6) Raw Integer SRRC_filter_out Value (Total Sum)  : 63
7) Real SRRC_filter_out Value (Total Sum - Q11.4) : 3.9375
-------------------------------------------------------------------------------
```

*Figure 10.3: Python Simulation for "1" as input test – **(Total Sum of Convolution Check)***

The two figures below, namely **Figure 11** and **Figure 11.1**, show the results of the simulations performed for the test related to the SRRC filter output when the input is ***" -1 "*** for the convolution. The Python simulation shows the expected result we should obtain from the simulation run on ModelSim, to assess if the circuit design is correct. By observing the results, we can say that the test is successful passed because the outputs are the same. Specifically, the test below checks whether there is a match for the "***peak value***" of the impulse response.

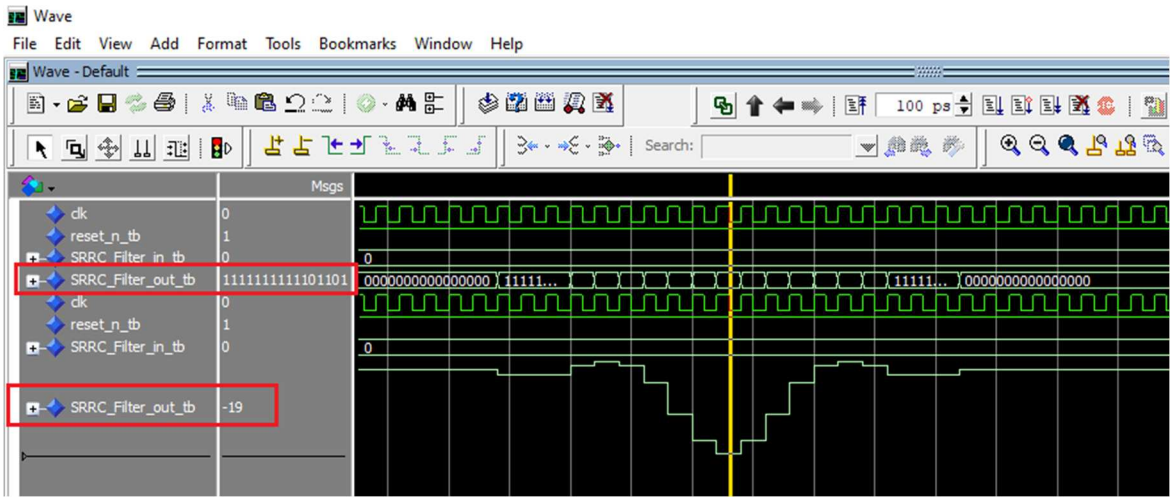

Figure 11: ModelSim Simulation response for " -1 " as input Test – ***(Peak Value Check)***

```
#--------------------------------------------------------------------------------
#       1) Analysis of the Central Coefficient Value (Impulse Response Peak)
#--------------------------------------------------------------------------------

        - @ Input Value Details -> 1) Real: -1.0
                                   2) Q8.7 Format: -128

        - @ Central Coefficient -> 1) Real: 1.1366
                                   2) Q1.14 Format: 18622

-- @ Expected Values @ --

1) Real Value of the Convolution (Peak) is [ -1.0 * 1.1366] : -1.13660
2) Scaled Value of the (Peak) is [ (-1.0 * 2^7) * (1.1366 * 2^14) = -128 * 18622 ] = -2383616

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the Convolution (Peak) (35-bit): 11111111111111011011010000100000000
4) Full-Raw integer value of the Convolution (Peak) (Fixed Point): -2383616

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out (Peak Value)            : 1111111111101101
6) Raw Integer SRRC_filter_out (Peak Value)       : -19
7) Real SRRC_filter_out Value (Peak Value - Q11.4) : -1.1875
--------------------------------------------------------------------------------
```

Figure 11.1: Python Simulation for " -1 " as input Test – ***(Peak Value Check)***

The two figures below, namely **Figure 11.2** and **Figure 11.3**, for the same input configuration as the previously test explained, instead show the results of the simulations to checks whether there is a match for the "***total sum***" of the convolution value. By observing the results, we can say that the test is successful passed because the outputs are the same.

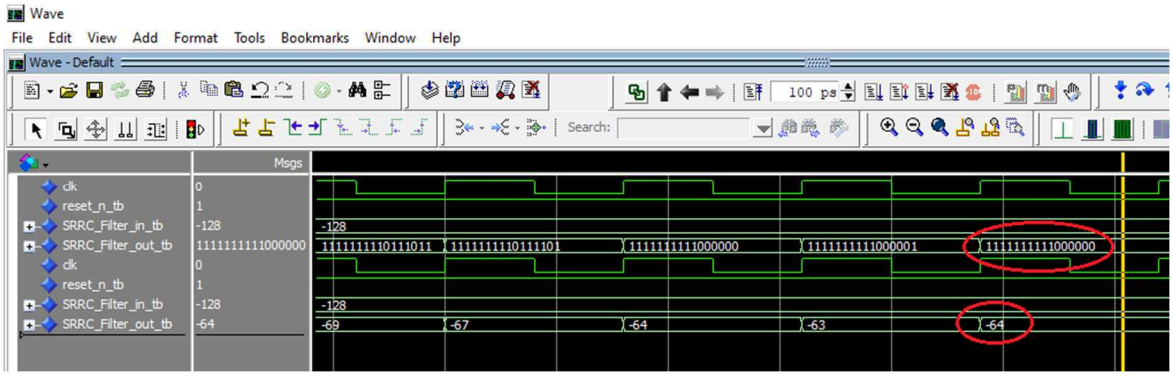

*Figure 11.2: ModelSim Simulation response for " -1 " as input Test – **(Total Sum of Convolution Check)***

```
#------------------------------------------------------------------------------
#       2) Analysis of Total Sum Result (Full Convolution Value)
#------------------------------------------------------------------------------

-- @ Expected Values @ --

1) Real Value of total sum with -1.0 as input is: -3.96423
2) Scaled Value (Q13.21) of total sum with -128 as input is = -8313600

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the total sum (35-bit): 11111111111100000010010010100000000
4) Full-Raw integer value of the total sum (Fixed Point): -8313600

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out Value (Total Sum)      : 1111111111000000
6) Raw Integer SRRC_filter_out Value (Total Sum) : -64
7) Real SRRC_filter_out Value (Total Sum - Q11.4) : -4.0
------------------------------------------------------------------------------
```

*Figure 11.3: Python Simulation for " -1 " as input Test – **(Total Sum of Convolution Check)***

The two figures below, namely **Figure 12** and **Figure 12.1**, show the results of the simulations performed for the test related to the SRRC filter output when the input is ***" 2 "*** for the convolution. The Python simulation shows the expected result we should obtain from the simulation run on ModelSim, to assess if the circuit design is correct. By observing the results, we can say that the test is successful passed because the outputs are the same. Specifically, the test below checks whether there is a match for the "***peak value***" of the impulse response.

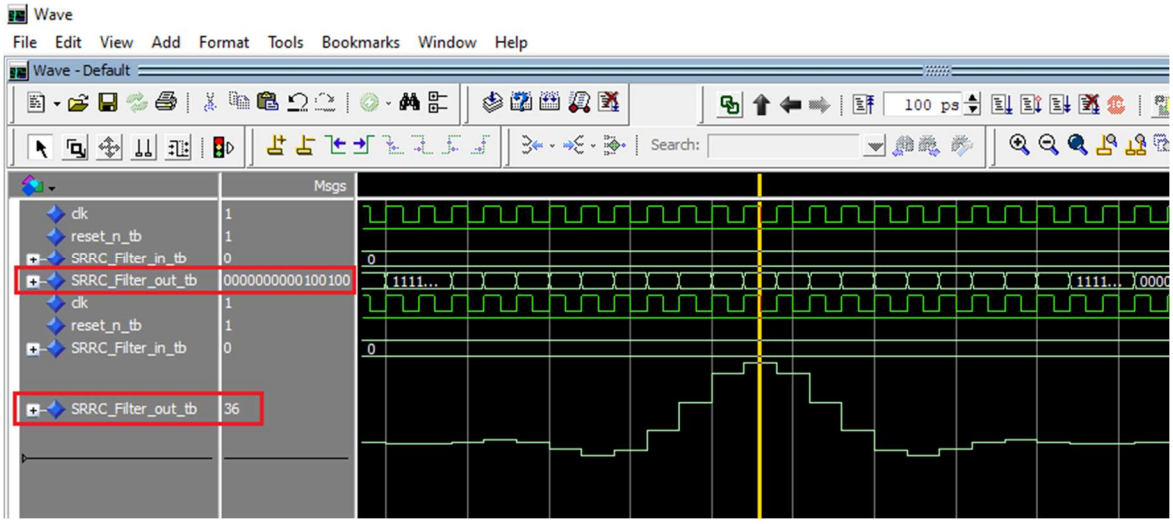

*Figure 12: ModelSim Simulation response for " 2 " as input Test – **(Peak Value Check)***

```
#--------------------------------------------------------------------------------
#       1) Analysis of the Central Coefficient Value (Impulse Response Peak)
#--------------------------------------------------------------------------------

        - @ Input Value Details -> 1) Real: 2.0
                                   2) Q8.7 Format: 256

        - @ Central Coefficient -> 1) Real: 1.1366
                                   2) Q1.14 Format: 18622

-- @ Expected Values @ --

1) Real Value of the Convolution (Peak) is [ 2.0 * 1.1366] : 2.27320
2) Scaled Value of the (Peak) is [ (2.0 * 2^7) * (1.1366 * 2^14) = 256 * 18622 ] = 4767232

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the Convolution (Peak) (35-bit): 00000000000010010001011111000000000
4) Full-Raw integer value of the Convolution (Peak) (Fixed Point): 4767232

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out (Peak Value)          : 0000000000100100
6) Raw Integer SRRC_filter_out (Peak Value)     : 36
7) Real SRRC_filter_out Value (Peak Value - Q11.4) : 2.25
--------------------------------------------------------------------------------
```

*Figure 12.1: Python Simulation for " 2 " as input Test – **(Peak Value Check)***

The two figures below, namely **Figure 12.2** and **Figure 12.3**, for the same input configuration as the previously test explained, instead show the results of the simulations to checks whether there is a match for the "***total sum***" of the convolution value. By observing the results, we can say that the test is successful passed because the outputs are the same.

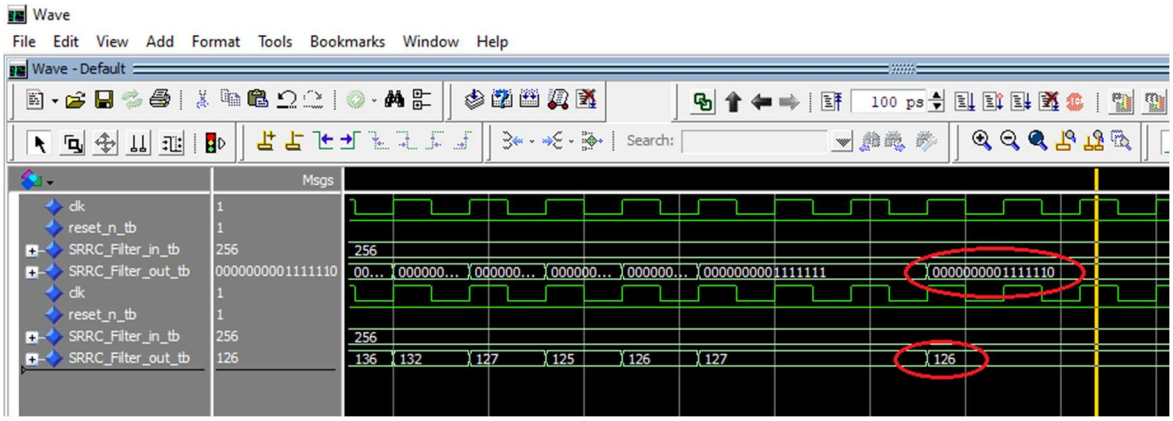

*Figure 12.2: ModelSim Simulation response for " 2 " as input Test - **(Total Sum of Convolution Check)***

```
#--------------------------------------------------------------------------------
#       2) Analysis of Total Sum Result (Full Convolution Value)
#--------------------------------------------------------------------------------

-- @ Expected Values @ --

1) Real Value of total sum with 2.0 as input is: 7.92847
2) Scaled Value (Q13.21) of total sum with 256 as input is = 16627200

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the total sum (35-bit): 00000000000111111011011011000000000
4) Full-Raw integer value of the total sum (Fixed Point): 16627200

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out Value (Total Sum)       : 0000000001111110
6) Raw Integer SRRC_filter_out Value (Total Sum)  : 126
7) Real SRRC_filter_out Value (Total Sum - Q11.4) : 7.875
--------------------------------------------------------------------------------
```

*Figure 12.3: Python Simulation for " 2 " as input Test – **(Total Sum of Convolution Check)***

The two figures below, namely **Figure 13** and **Figure 13.1**, show the results of the simulations performed for the test related to the SRRC filter output when the input is ***" -2 "*** for the convolution. The Python simulation shows the expected result we should obtain from the simulation run on ModelSim, to assess if the circuit design is correct. By observing the results, we can say that the test is successful passed because the outputs are the same. Specifically, the test below checks whether there is a match for the "***peak value***" of the impulse response.

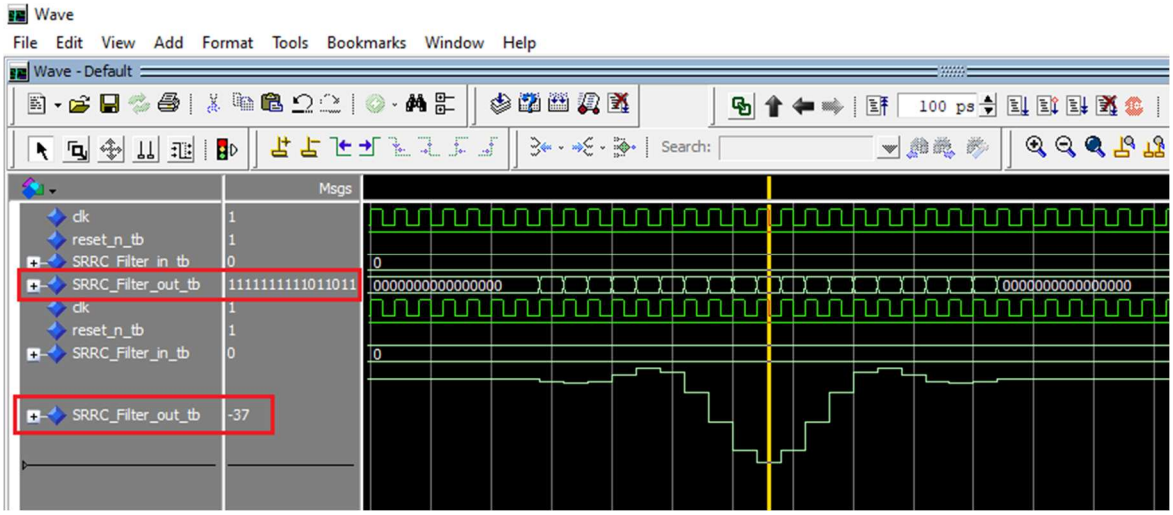

Figure 13: ModelSim Simulation response for " -2 " as input Test – ***(Peak Value Check)***

```
#--------------------------------------------------------------------------------
#       1) Analysis of the Central Coefficient Value (Impulse Response Peak)
#--------------------------------------------------------------------------------

        - @ Input Value Details -> 1) Real: -2.0
                                   2) Q8.7 Format: -256

        - @ Central Coefficient -> 1) Real: 1.1366
                                   2) Q1.14 Format: 18622

-- @ Expected Values @ --

1) Real Value of the Convolution (Peak) is [ -2.0 * 1.1366] : -2.27320
2) Scaled Value of the (Peak) is [ (-2.0 * 2^7) * (1.1366 * 2^14) = -256 * 18622 ] = -4767232

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the Convolution (Peak) (35-bit): 11111111111101101110100001000000000
4) Full-Raw integer value of the Convolution (Peak) (Fixed Point): -4767232

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out (Peak Value)            : 1111111111011011
6) Raw Integer SRRC_filter_out (Peak Value)       : -37
7) Real SRRC_filter_out Value (Peak Value - Q11.4) : -2.3125
--------------------------------------------------------------------------------
```

Figure 13.1: Python Simulation for " -2 " as input Test – ***(Peak Value Check)***

The two figures below, namely **Figure 13.2** and **Figure 13.3**, for the same input configuration as the previously test explained, instead show the results of the simulations to checks whether there is a match for the "***total sum***" of the convolution value. By observing the results, we can say that the test is successful passed because the outputs are the same.

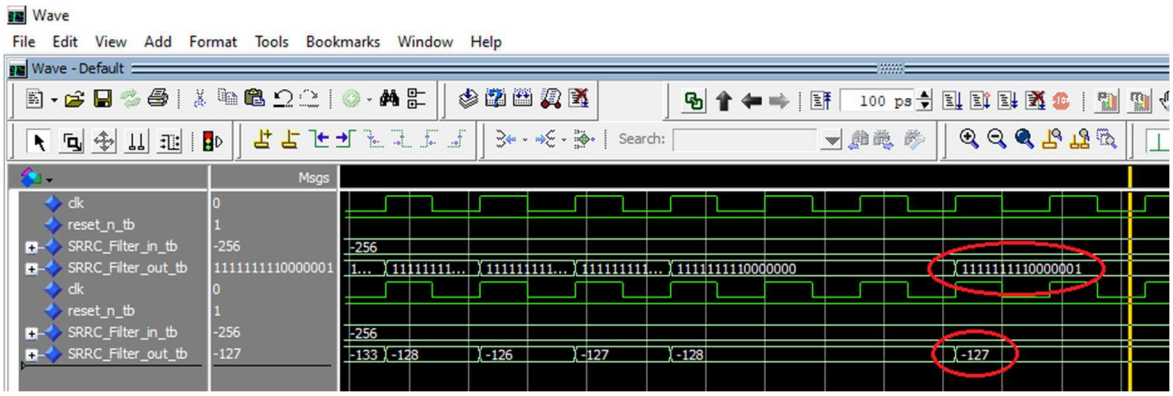

Figure 13.2: ModelSim Simulation response for " -2 " as input Test – ***(Total Sum of Convolution Check)***

```
#--------------------------------------------------------------------------------
#       2) Analysis of Total Sum Result (Full Convolution Value)
#--------------------------------------------------------------------------------

-- @ Expected Values @ --

1) Real Value of total sum with -2.0 as input is: -7.92847
2) Scaled Value (Q13.21) of total sum with -256 as input is = -16627200

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the total sum (35-bit): 11111111111000000100100101000000000
4) Full-Raw integer value of the total sum (Fixed Point): -16627200

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out Value (Total Sum)       : 1111111110000001
6) Raw Integer SRRC_filter_out Value (Total Sum)  : -127
7) Real SRRC_filter_out Value (Total Sum - Q11.4) : -7.9375
--------------------------------------------------------------------------------
```

Figure 13.3: Python Simulation for " -2 " as input Test – ***(Total Sum of Convolution Check)***

The two figures below, namely **Figure 14** and **Figure 14.*1***, show the results of the simulations performed for the test related to the SRRC filter output when the input is ***" 255.9921875 "*** for the convolution that is the ***Maximum Value Positive*** rapresentable. The Python simulation shows the expected result we should obtain from the simulation run on ModelSim, to assess if the circuit design is correct. By observing the results, we can say that the test is successful passed because the outputs are the same. Specifically, the test below checks whether there is a match for the "***peak value***" of the impulse response.

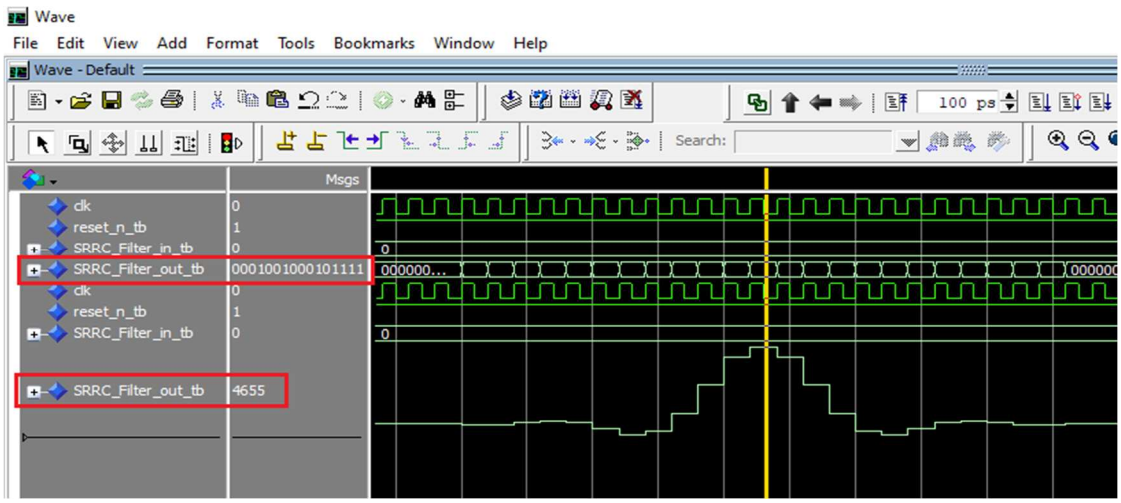

Figure 14: ModelSim Simulation response for " 255.9921875 " as input Test (Maximum Positive Value) – ***(Peak Value Check)***

```
#--------------------------------------------------------------------------------
#       1) Analysis of the Central Coefficient Value (Impulse Response Peak)
#--------------------------------------------------------------------------------

        - @ Input Value Details -> 1) Real: 255.9921875
                                   2) Q8.7 Format: 32767

        - @ Central Coefficient -> 1) Real: 1.1366
                                   2) Q1.14 Format: 18622

-- @ Expected Values @ --

1) Real Value of the Convolution (Peak) is [ 255.9921875 * 1.1366] : 290.96072
2) Scaled Value of the (Peak) is [ (255.9921875 * 2^7) * (1.1366 * 2^14) = 32767 * 18622 ] = 610187074

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the Convolution (Peak) (35-bit): 00000100100010111101011011101000010
4) Full-Raw integer value of the Convolution (Peak) (Fixed Point): 610187074

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out (Peak Value)            : 0001001000101111
6) Raw Integer SRRC_filter_out (Peak Value)       : 4655
7) Real SRRC_filter_out Value (Peak Value - Q11.4) : 290.9375
--------------------------------------------------------------------------------
```

*Figure 14.1: Python Simulation for " 255.9921875 " as input Test (Maximum Positive Value) – **(Peak Value Check)***

The two figures below, namely **Figure 14.2** and **Figure 14.3**, for the same input configuration as the previously test explained, instead show the results of the simulations to checks whether there is a match for the "***total sum***" of the convolution value. By observing the results, we can say that the test is successful passed because the outputs are the same.

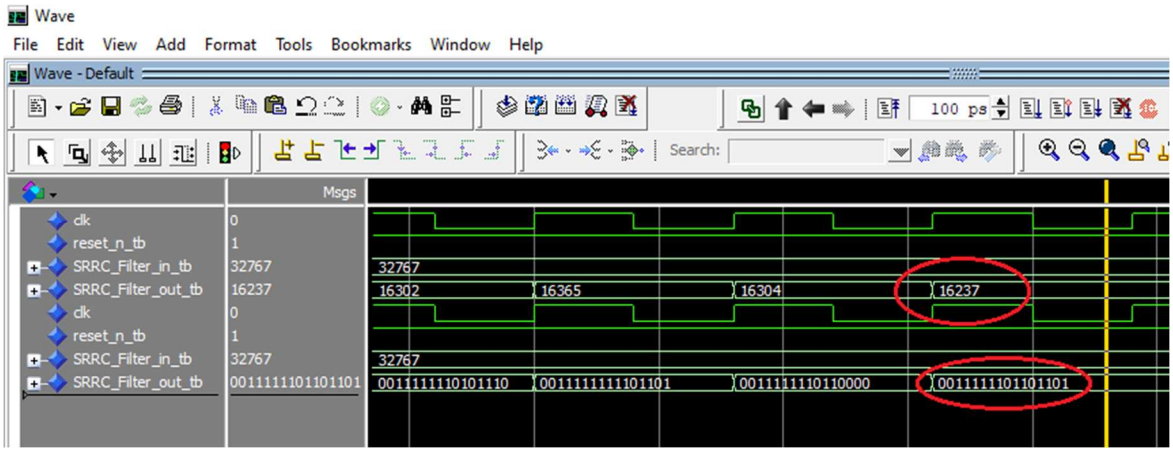

*Figure 14.2: ModelSim Simulation response for " 255.9921875 " as input Test (Maximum Positive Value) – **(Total Sum of Convolution Check)***

```
#--------------------------------------------------------------------------------
#       2) Analysis of Total Sum Result (Full Convolution Value)
#--------------------------------------------------------------------------------

-- @ Expected Values @ --

1) Real Value of total sum with 255.9921875 as input is: 1014.81278
2) Scaled Value (Q13.21) of total sum with 32767 as input is = 2128216650

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the total sum (35-bit): 00001111110110110100000001001001010
4) Full-Raw integer value of the total sum (Fixed Point): 2128216650

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out Value (Total Sum)       : 0011111101101101
6) Raw Integer SRRC_filter_out Value (Total Sum)  : 16237
7) Real SRRC_filter_out Value (Total Sum - Q11.4) : 1014.8125
--------------------------------------------------------------------------------
```

*Figure 14.3: Python Simulation for " 255.9921875 " as input Test (Maximum Positive Value) – **(Total Sum of Convolution Check)***

The two figures below, namely **Figure 15** and **Figure 15.1**, show the results of the simulations performed for the test related to the SRRC filter output when the input is ***" -256 "*** for the convolution that is the ***Minimum Value Negative*** rapresentable. The Python simulation shows the expected result we should obtain from the simulation run on ModelSim, to assess if the circuit design is correct. By observing the results, we can say that the test is successful passed because the outputs are the same. Specifically, the test below checks whether there is a match for the "***peak value***" of the impulse response.

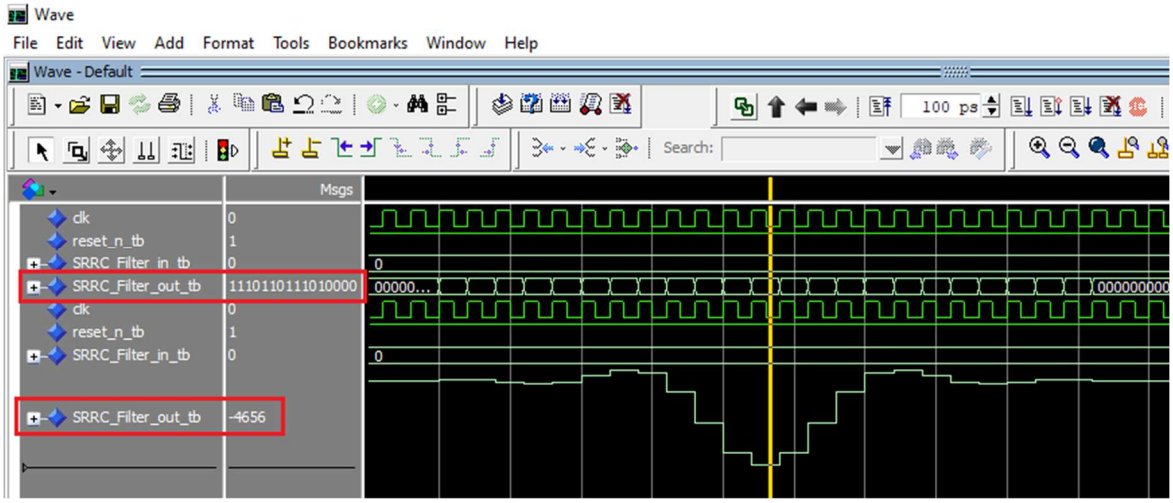

*Figure 15: ModelSim Simulation response for " -256 " as input Test (Maximum Negative Value) – **(Peak Value Check)***

```
#--------------------------------------------------------------------------------
#       1) Analysis of the Central Coefficient Value (Impulse Response Peak)
#--------------------------------------------------------------------------------

        - @ Input Value Details -> 1) Real: -256.0
                                   2) Q8.7 Format: -32768

        - @ Central Coefficient -> 1) Real: 1.1366
                                   2) Q1.14 Format: 18622

-- @ Expected Values @ --

1) Real Value of the Convolution (Peak) is [ -256.0 * 1.1366] : -290.96960
2) Scaled Value of the (Peak) is [ (-256.0 * 2^7) * (1.1366 * 2^14) = -32768 * 18622 ] = -610205696

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the Convolution (Peak) (35-bit): 11111011011101000010000000000000000
4) Full-Raw integer value of the Convolution (Peak) (Fixed Point): -610205696

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out (Peak Value)           : 1110110111010000
6) Raw Integer SRRC_filter_out (Peak Value)      : -4656
7) Real SRRC_filter_out Value (Peak Value - Q11.4) : -291.0
--------------------------------------------------------------------------------
```

*Figure 15.1: Python Simulation for " -256 " as input Test (Maximum Negative Value) – **(Peak Value Check)***

The two figures below, namely **Figure 15.2** and **Figure 15.3**, for the same input configuration as the previously test explained, instead show the results of the simulations to checks whether there is a match for the "***total sum***" of the convolution value. By observing the results, we can say that the test is successful passed because the outputs are the same.

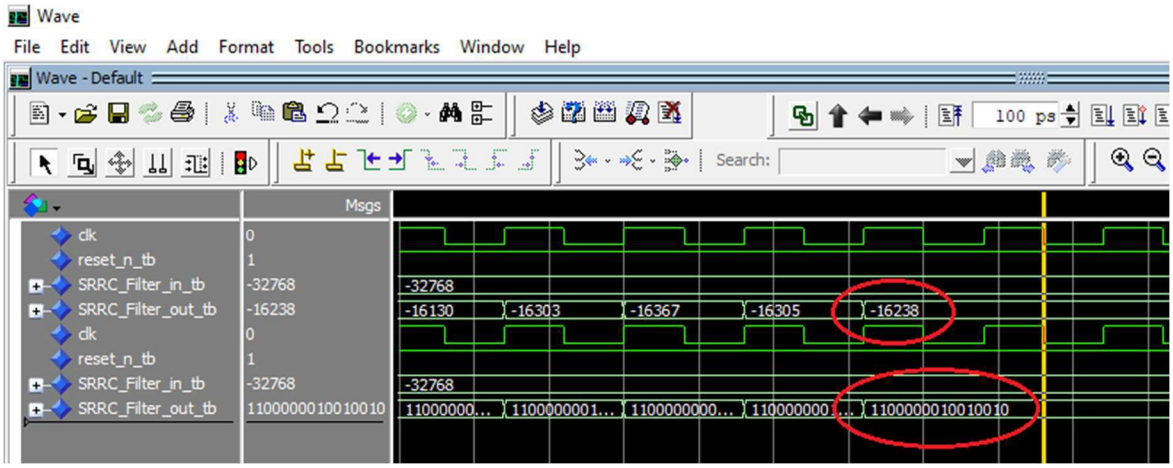

Figure 15.2: ModelSim Simulation response for " -256 " as input Test (Minimum Negative Value) - ***(Total Sum of Convolution Check)***

```
#--------------------------------------------------------------------------------
#       2) Analysis of Total Sum Result (Full Convolution Value)
#--------------------------------------------------------------------------------

-- @ Expected Values @ --

1) Real Value of total sum with -256.0 as input is: -1014.84375
2) Scaled Value (Q13.21) of total sum with -32768 as input is = -2128281600

-- @ Filter Output Results Pre-Truncation @ --

3) Full-Binary Value of the total sum (35-bit): 11110000001001001010000000000000000
4) Full-Raw integer value of the total sum (Fixed Point): -2128281600

-- @ Filter Output Results Post-Truncation @ --
-- @ Use for comparison with ModelSim results @ --

5) Binary SRRC_filter_out Value (Total Sum)       : 1100000010010010
6) Raw Integer SRRC_filter_out Value (Total Sum)  : -16238
7) Real SRRC_filter_out Value (Total Sum - Q11.4) : -1014.875
--------------------------------------------------------------------------------
```

Figure 15.3: Python Simulation for " -256 " as input Test (Minimum Negative Value) – ***(Total Sum of Convolution Check)***

The two figures below, namely **Figure 16** and **Figure 16.1**, show the results of the simulations performed for the test concerning the output of the SRRC filter when the input is the sequence ***" 1, 2, 3…23 "*** for the convolution. Specifically, the following test allows us to evaluate the dynamic behavior of the filter in according with what was described at the point 3) in **Test-Plan** description. The **response** of this type of filter is *considerably long*, so for space and visualization reasons, only some sequences and the observed correspondence will be shown. This correspondence, which allows us to state that the test has been passed, is also valid for all the other sequences. This can be verified by running the tests using the corresponding tools and the related code provided along with this report..

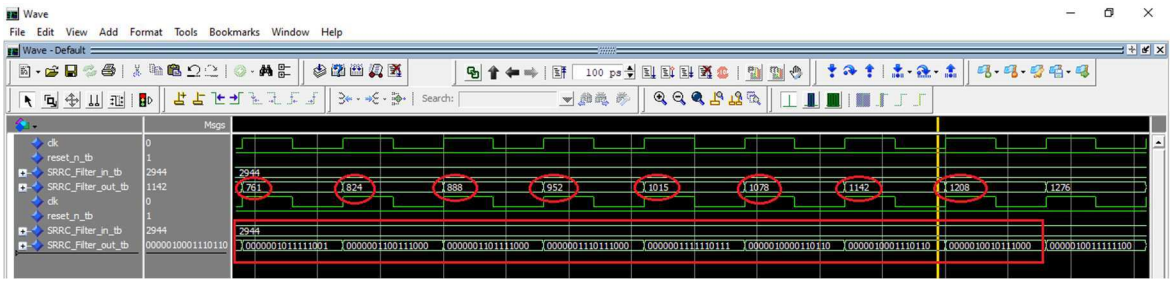

Figure 16: ModelSim Simulation response for " 1, 2, 3, 4…23 " sequence as input Test – ***(Total Sums of Convolution Check – Left_to_Right)***

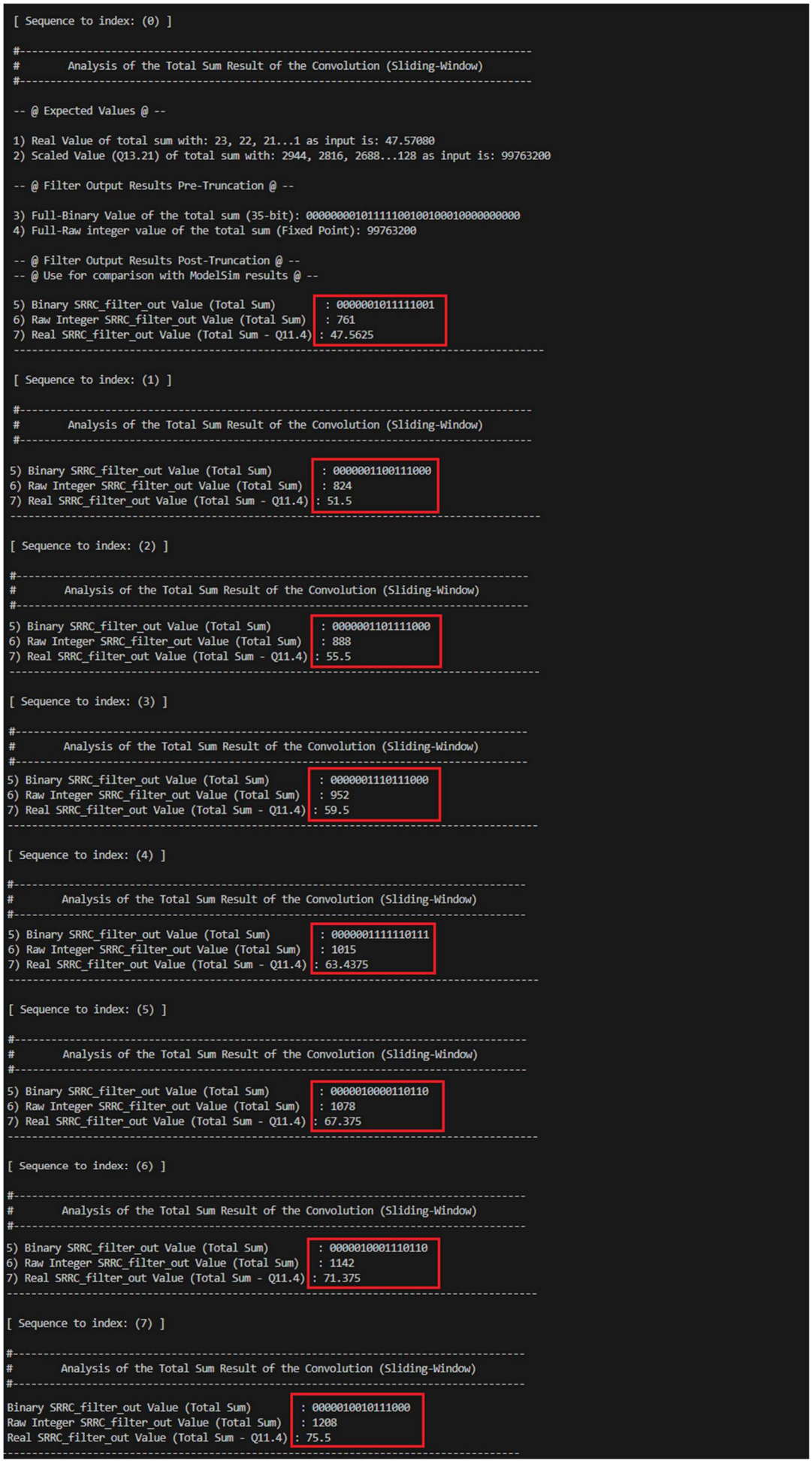

Figure 16.1: Python Simulation response for " 1, 2, 3, 4…23 " sequence as input Test –
(Total Sums of Convolution Check – Sliding Window)

The two figures below, namely **Figure 17** and **Figure 17.1**, show the results of the simulations carried out for the test regarding the output of the SRRC filter when the input is the sequence ***" -2, 4, -2, 4…-2"*** for the convolution. *Once again*, for space and visualization reasons, only some sequences will be shown. However, the observed correspondence—allowing us to state that the test has been passed—is also valid for all the other sequences. This can be confirmed by running the tests using the appropriate tools and the corresponding code provided with this report. By observing the results, we can conclude that the test has been successfully passed.

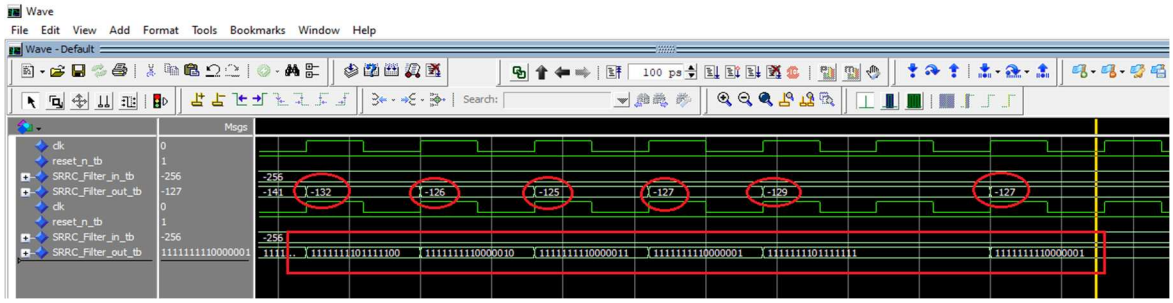

Figure 17: ModelSim Simulation response for " -2, 4, -2, 4,…-2 " sequence as input Test – ***(Total Sums of Convolution Check – Right_to_Left)***

Figure 17.1: Python Simulation response for " -2, 4, -2, 4,…-2 " sequence as input Test – ***(Total Sums of Convolution Check – Sliding Window)***

The two figures below, namely **Figure 18** and **Figure 18.1**, show the results of the simulations carried out for the test regarding the output of the SRRC filter when the input is the sequence ***"-1,-2,-1,-2…-1"*** for the convolution. *Once again*, for space and visualization reasons, only some sequences will be shown. However, the observed correspondence—allowing us to state that the test has been passed—is also valid for all the other sequences. This can be confirmed by running the tests using the appropriate tools and the corresponding code provided with this report. By observing the results, we can conclude that the test has been successfully passed

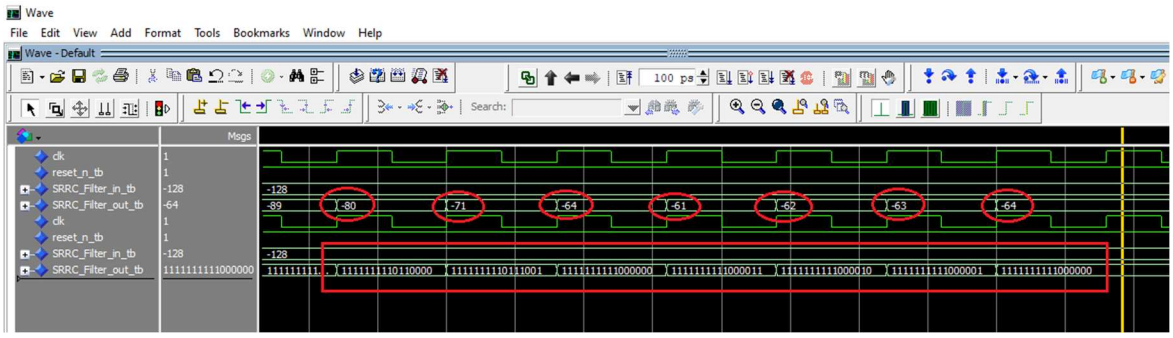

Figure 18: ModelSim Simulation response for " -1, -2, -1, -2,…-1 " sequence as input Test – ***(Total Sums of Convolution Check – Right_to_Left)***

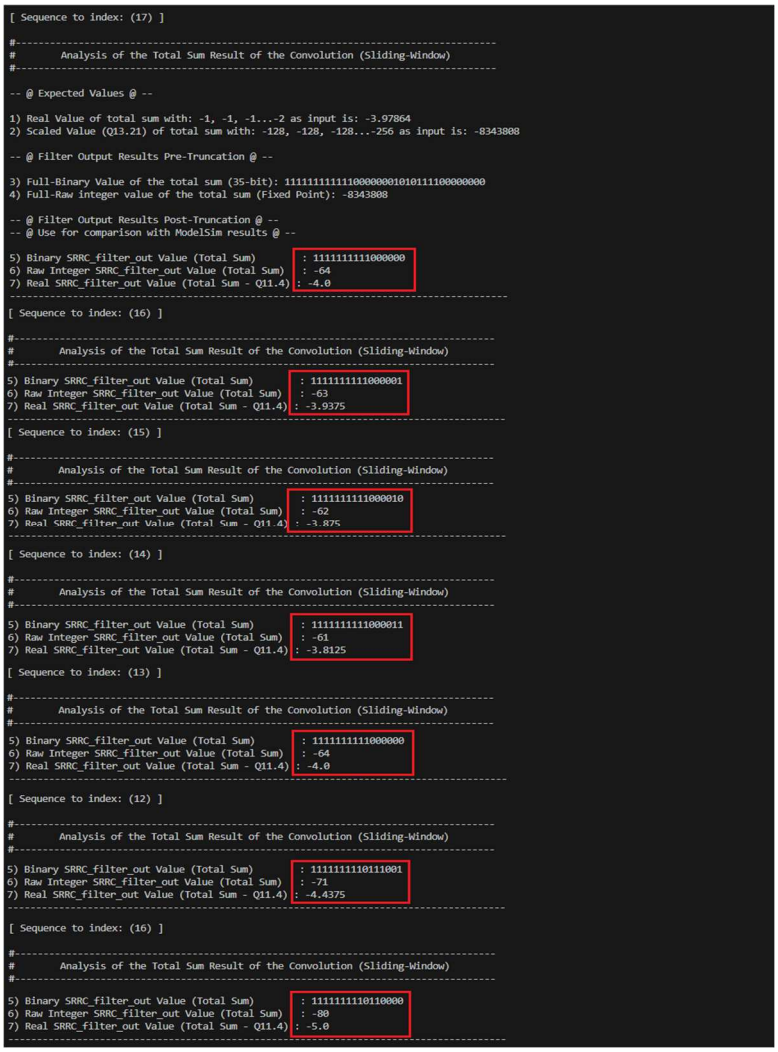

Figure 18.1: Python Simulation response for " -1, -2, -1, -2,…-1 " sequence as input Test – ***(Total Sums of Convolution Check – Sliding Window)***

After completing all the previously described tests — which confirmed the filter's ability to handle *overflow* conditions, potential *saturation*, *dynamic behavior*, and edge cases such as the representation of *large numbers* — I found it useful to conduct an additional analysis focused on scenarios related to the *filter's typical application domains*. This further assessment was intended to support the verification of the filter's correct functionality and the overall circuit design.

Specifically, the scenarios considered include the response to a *sinusoidal* input with a **frequency higher than the allowed bandwidth** — expected result: **strong attenuation**; a **frequency within the allowed bandwidth** — expected result: **low attenuation**; and the case of **Inter-Symbol Interference** (ISI), where the filter should prevent symbol overlap by producing non-zero values only at symbol instants and values close to zero elsewhere. For completeness, the test also includes the **filter's impulse response**, which should display the expected symmetric waveform.

Figures 19 and **19.1** below graphically show both the filter's output and the input waveform, enabling a *qualitative* assessment of the results obtained for each scenario. These results are compared between the simulations run in ModelSim and those in Python. In ModelSim, waveform visualization can be achieved by adjusting appropriate settings related to the display format. By examining the python graphs and the waveforms by ModelSim, one can clearly observe that the outcomes match the expected behavior for each case, confirming that these tests were also successfully passed.

Figure 19: ModelSim Simulation response for " single impulse, +1/-1 alternated, out-band and in-band sinusoid waveform " as input Test – ***(Waveform Analysis)***

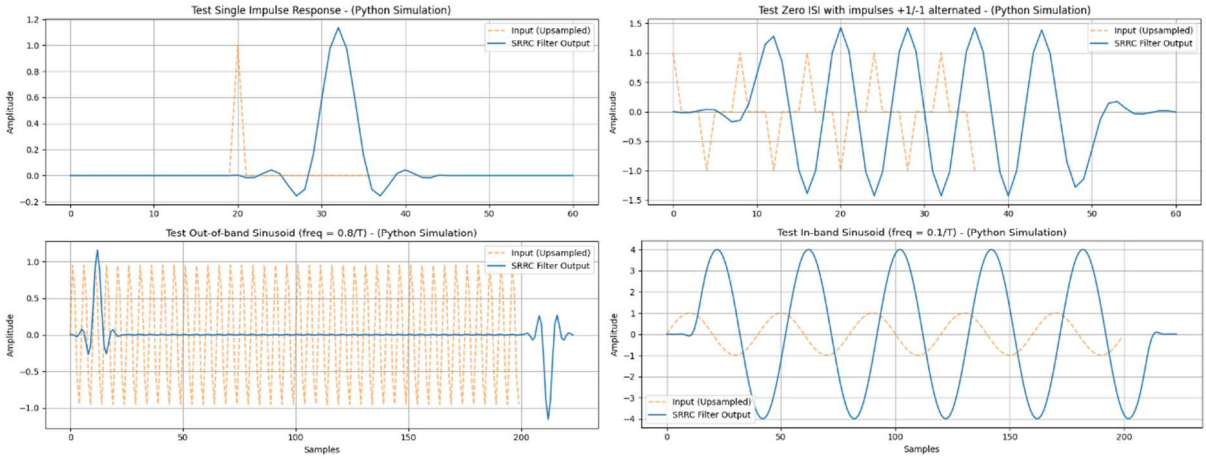

Figure 19.1: Python Simulation response for " single impulse, +1/-1 alternated, out-band and in-band sinusoid waveform " as input Test – ***(Waveform Analysis)***

FPGA Synthesis/Implementation – Vivado

In this section we will look in detail at the stages of *Design Flow* related to **Vivado** Software that is an integrated development environment (IDE) provided by Xilinx for the design, synthesis, simulation, and implementation of digital circuits on FPGAs, from RTL Elaboration up to Implementation phase.

RTL Elaboration

As first step, the design elaboration phase of the circuit was initiated. This process allows the extraction of the schematic containing the structure of the various components and the complete netlist of the design. The following **Figures 20** and **21** provide an overview of the wrapper and a detailed view of the full schematic design of the SRRC filter.

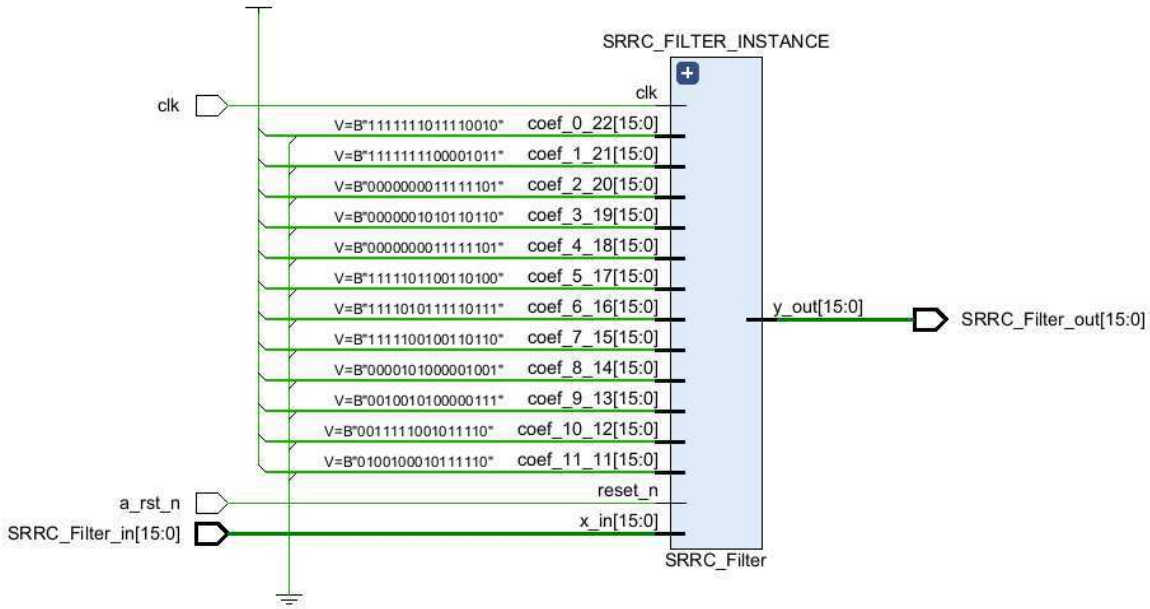

Figure 20: SRRC FIR Fiter Wrapper Design Schematic

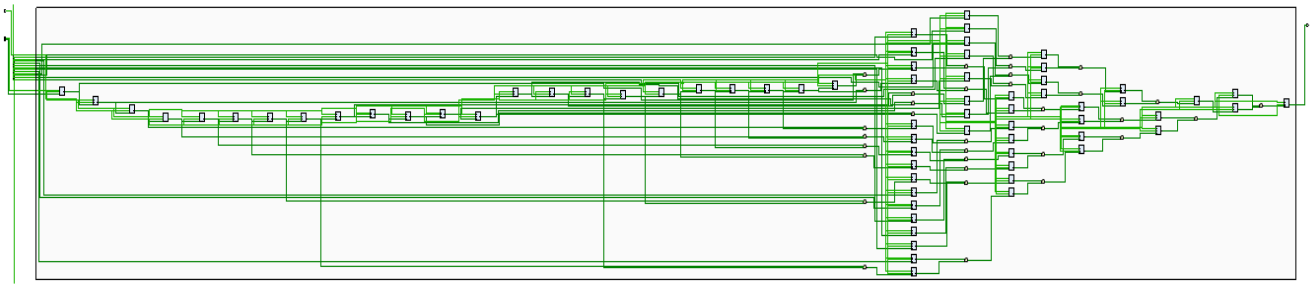

Figure 21: SRRC FIR Fiter Design full Schematic

Synthesis

The next step was to perform the actual synthesis of the circuit in order to generate the schematic of the synthesized design, showing which resources will be used on the FPGA and whether there are any issues with the design, ensuring that everything works correctly and that the design is free of bugs.

Before launching the synthesis, two actions were taken to meet the project requirements: setting the *constraints* and configuring the synthesis settings to run it in "*out_of_context*" mode.

- **Clock Constraint**: The only constraint required to be respected concerns the clock and has been declared according to the specifications of the board used to start the project on Vivado – **Zybo Board: ZYNQ XC7Z010-1CLG400C** – specifying a clock period of **8 nanoseconds** with a duty cycle of *4 ns*, thus achieving a maximum clock frequency of **125 MHz**.

- **Out of Context Mode**: Configuring this synthesis settings was necessary because, not having the actual FPGA on which the post-implementation circuit would be loaded, and therefore being unable to perform any I/O planning, it was essential to ensure that the timing analyses performed by the software were as accurate as possible and not affected by delays related to external pins. Thanks to this mode, the software Vivado does not attempt to connect the I/O ports of our Top-Level circuit to the FPGA's external pins, aiming instead to achieve the best possible **characterization** of the circuit, consistent with an approach focused on completing the development of an **Intellectual Property** (IP) Block.

As shown in the **Figure 22** below, the circuit synthesis was successfully completed and the design was correctly synthesized. No design-related errors appeared, except for a **warning** that can be safely ignored, as it is not related to the design itself. In fact, it refers to a synthesis **optimization** procedure that the software Vivado attempts to perform, called **parallel synthesis**, which is not a functional error and is in no way tied to the project.

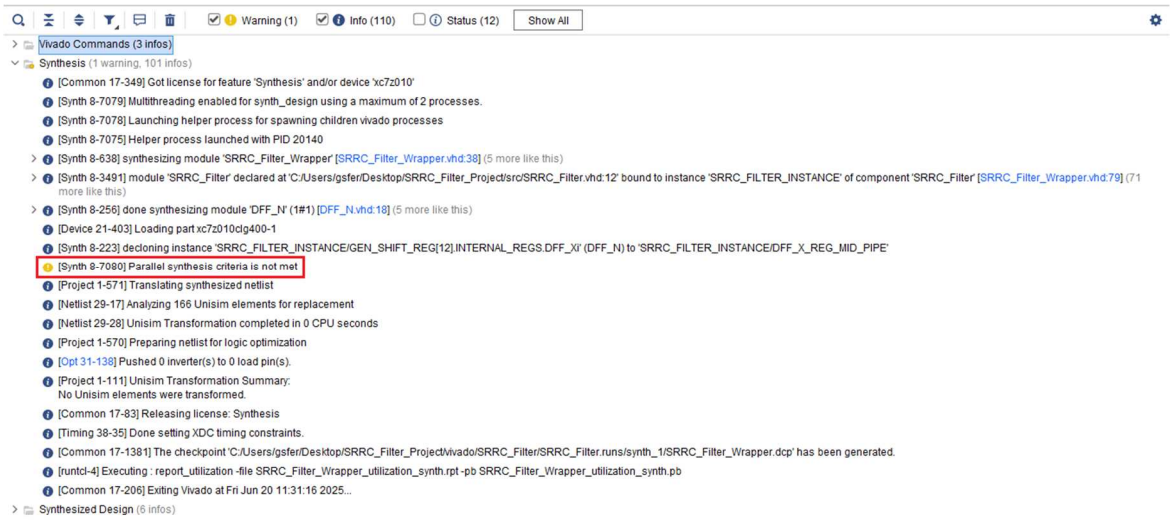

Figure 22: Messages listed after Synthesis procedure by Vivado Software

Figures 23 and **24** below show the schematic of the synthesized circuit design.

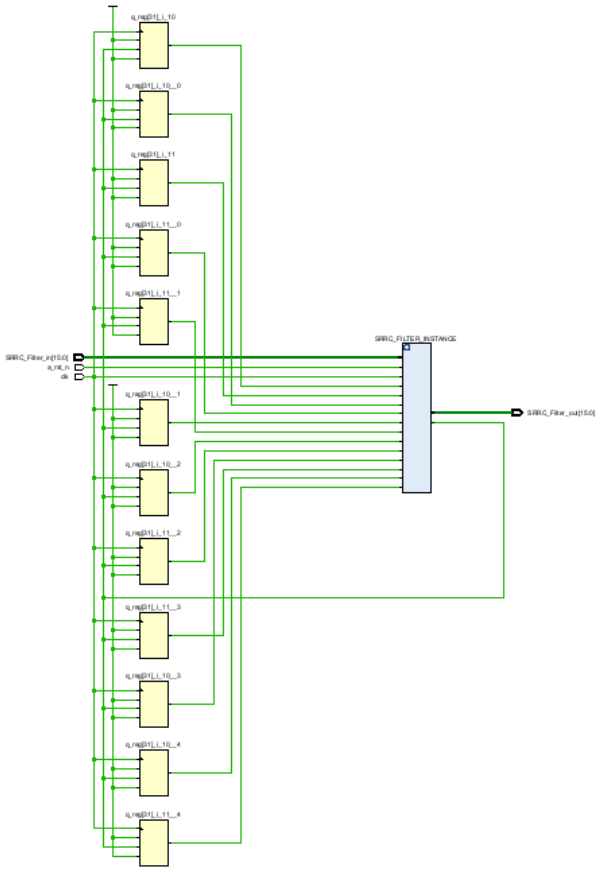

Figure 23: SRRC FIR Fiter Synthesized Wrapper Design Schematic

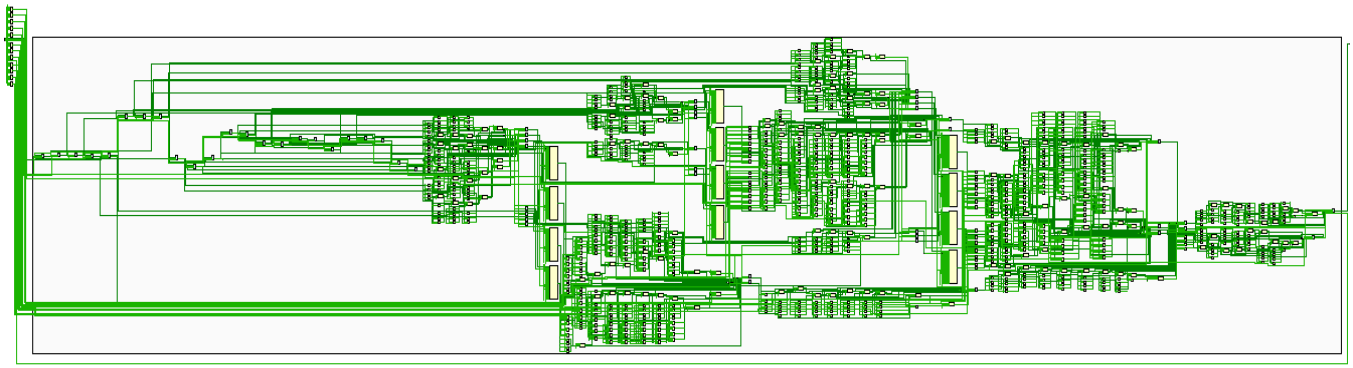

Figure 24: SRRC FIR Fiter Synthesized Design Full Schematic

Implementation

The final step after synthesis is the implementation, which essentially aims to transform the synthesized design into a concrete mapping of physical resources on the FPGA. This process allows access to useful information such as how much and which part of the FPGA area will be occupied by the design, the actual resources that the circuit will use, and how the various components are interconnected (connecting LUTs, flip-flops, DSP blocks, etc.) through the available routing resources. It also generates various analysis reports regarding timing, power consumption, and any potential critical paths.

As shown in the **Figure 25** below, the circuit implementation was also successfully completed. No design-related errors appeared during this phase, except for a series of **warnings** that can safely be ignored. These warnings are not related to the design itself but are instead due to the fact that *Vivado* is operating in **out-of-context** mode. Specifically, as visible in the image, the warnings refer to the **optimization** and **routing** phases. Most of them are related to the latter because, in *out-of-context* mode, Vivado does not have access to the complete project context and therefore cannot determine the exact clock source — such as the physical pin — or evaluate potential delays. This also explains the optimization-related warning, where the software attempts to optimize the design paths in relation to external pins, trying to minimize the distance between logic blocks.

Of course, in a different context, these warnings should be reviewed carefully. However, for the purposes of this project, they are not considered problematic, given the awareness of the specific working mode intentionally selected.

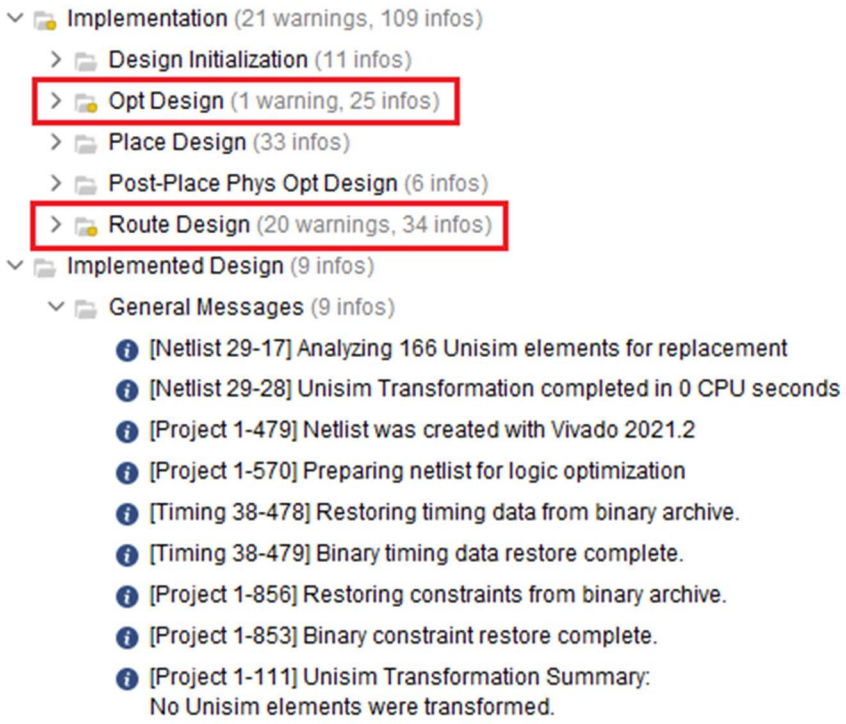

Figure 25: Messages listed after Implementation procedure by Vivado Software

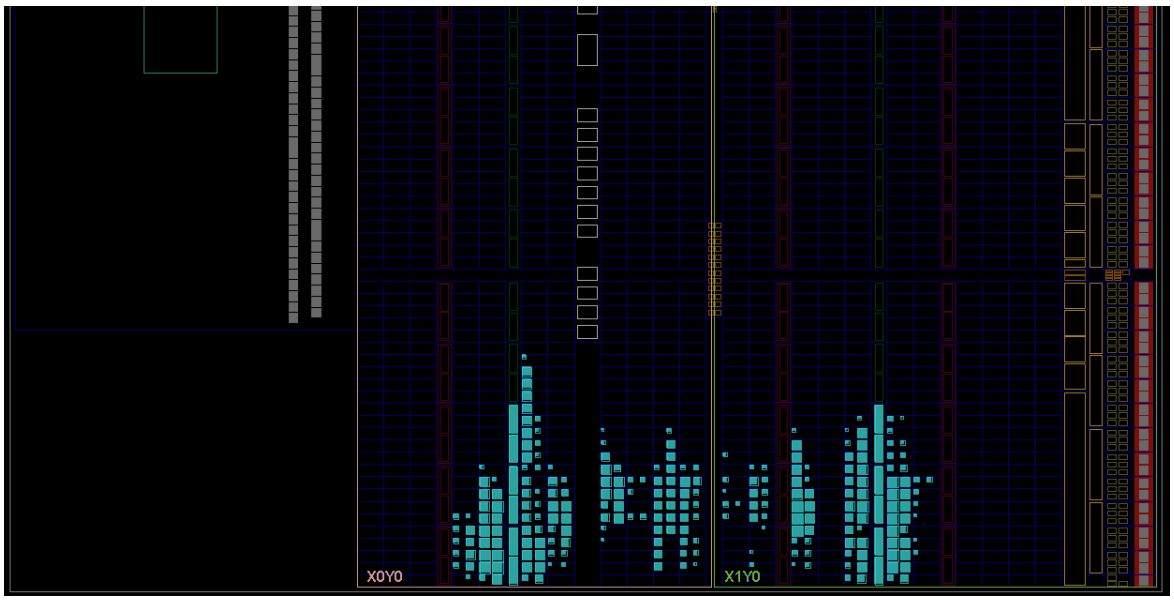

Figure 26: Picture just to provide an overview for the area occupied by the circuit on the FPGA

Timing Report

Figure 27 below shows the results related to **Timing Analysis**, providing the value of what is the **Worst Negative Slack** (WNS), which represents the negative slack value of a *critical path* in the worst case, through which it is possible to understand whether there are paths that do not meet the timing in relation to the frequency constraint.

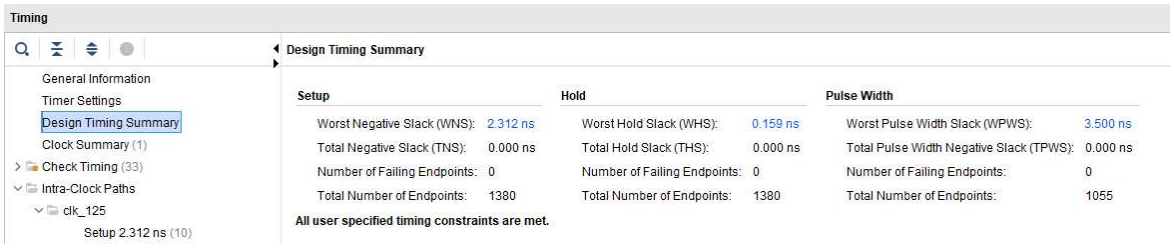

Figure 27: Timing Report for Worst Negative Slack Analysis

Thanks to this information it is then possible to calculate how much flexibility related to the frequency to which this circuit wanting can still go, due to its efficeinte design, without incurring violations.

Since the *WNS* is **2.312** ns considering the clock period of **8** ns we get a minimum clock period of **5.688** ns, which means a **maximum clock frequency** of **175.8Mhz**.

Critical Paths

More information regarding the path that led to this slack value can be viewed in the following **Figure 28**.

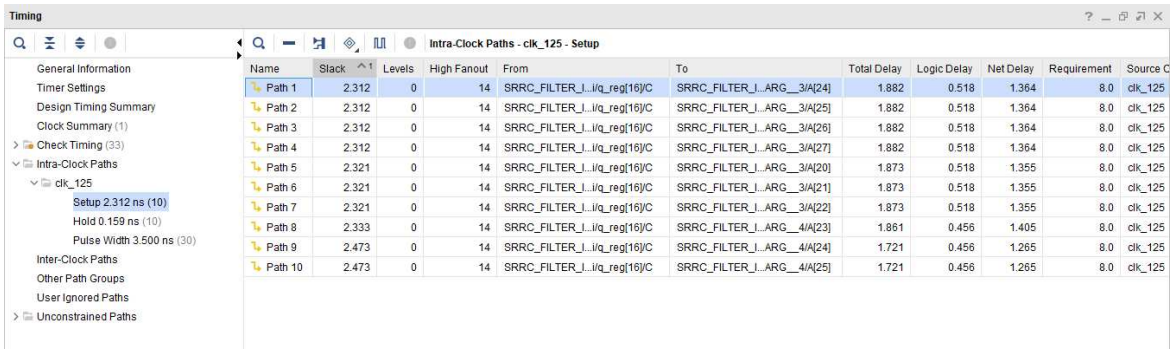

Figure 28: List of the Paths

More precisely, the image above shows us what are the umber of relevant critical paths. As an initial matter, we can state that on thing that we won't see usually in this case are **negative** or **near-zero** slack values, latters are cases in which a clock violation would have occurred or almost there (when about 0 zero). Thus, the feedback is positive because we can see how the design exhibits **positive slack** in all major paths, with values hovering around **2.3** ns.

This means that the circuit is temporally well **balanced**, moreover the **total delays** are much less than the reference clock period (8 ns), and this indicates that the design could potentially be pushed even **to higher frequencies** without violations as seen previously in the analysis of the maximum clock frequency. If desired, one could think of further improving performance by acting on one or more of the paths highlighted in the analysis, although in this case we can say that in overall, given the worst case, the design provides excellent performance, stability, and robustness.

Figure 29 highlights the structure of the analyzed paths showing that they go from a register to the DSP.

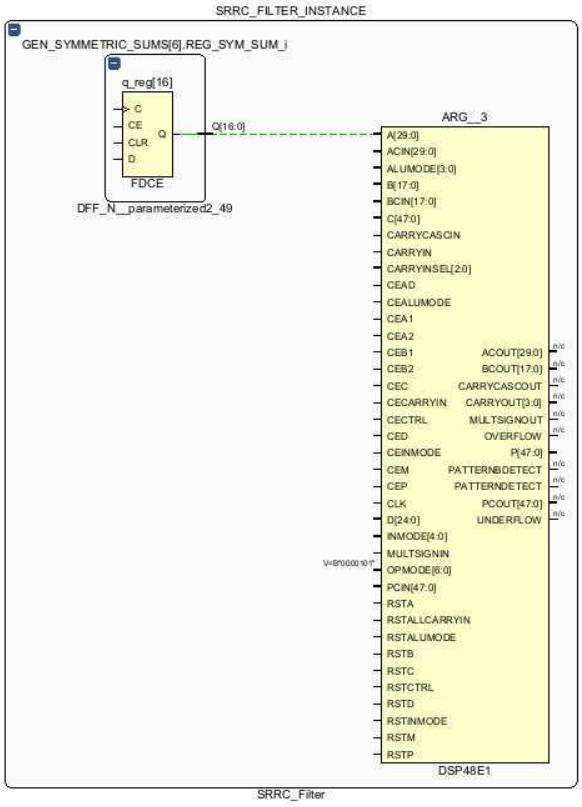

Figure 29: Path 1 in detail

Resources Utilization

Figure 30 below shows the report for the device utilization, and **Figure 30.1** shows a view of resource utilization in detail.

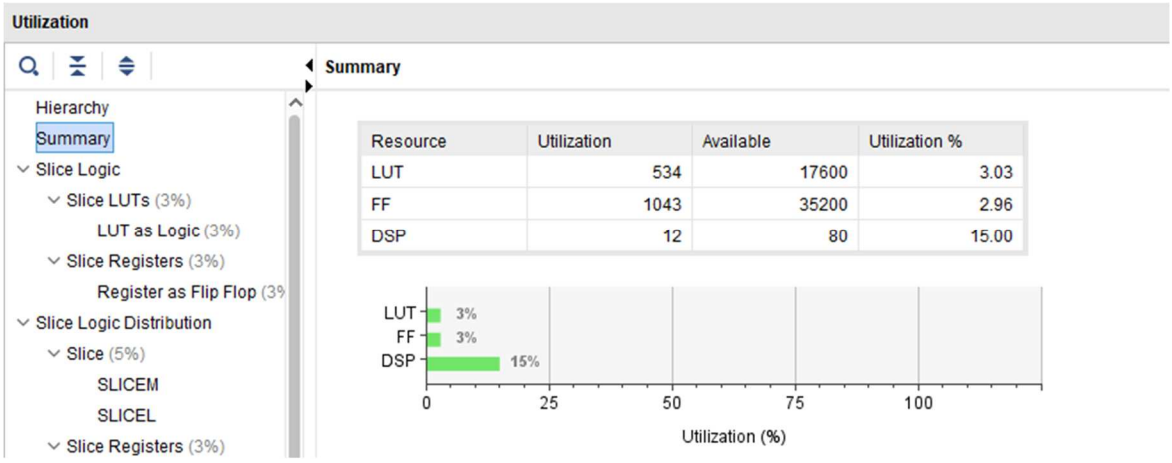

Figure 30: Utilization Report

One can see from the results obtained a decidedly low use of the device's logic resources and a very good balance between them and the DSP blocks. The very low use of **LUTs** and **FFs** suggests a compact and **well-optimized design**, thus allowing flexibility for any extensions or modifications that one might wish to make. Regarding the obvious more pronounced use of **DSP** blocks (15%) is something expected and justified by the type of design that was intended to implement the SRRC_Filter, where multiplications are frequent and DSPs are the dedicated resources to handle them efficiently thus improving both performance and energy efficiency.

Again, the results tell us that the design is efficient, logic-light and properly intensive in DSP blocks, consistent with the functional requirements of the implemented circuit.

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	DSPs (80)
SRRC_Filter_Wrapper	534	1043	235	534	12
SRRC_FILTER_INSTANCE (SRRC_Filter)	534	1031	233	534	12
GEN_ADDER_STAGE1_REGS[0].REG_A	0	33	9	0	0
GEN_ADDER_STAGE1_REGS[1].REG_A	0	33	9	0	0
GEN_ADDER_STAGE1_REGS[2].REG_A	0	33	9	0	0
GEN_ADDER_STAGE1_REGS[3].REG_A	0	33	9	0	0
GEN_ADDER_STAGE1_REGS[4].REG_A	0	33	9	0	0
GEN_ADDER_STAGE1_REGS[5].REG_A	0	33	9	0	0
GEN_ADDER_STAGE2_REGS[0].REG_A	1	33	10	1	0
GEN_ADDER_STAGE2_REGS[1].REG_A	0	33	9	0	0
GEN_ADDER_STAGE2_REGS[2].REG_A	0	33	9	0	0
GEN_ADDER_STAGE3_REGS[0].REG_A	0	33	9	0	0
GEN_ADDER_STAGE3_REGS[1].REG_A	0	33	14	0	0
GEN_COEFF_REGS[0].REG_COEF_i (DF	0	12	7	0	0
GEN_COEFF_REGS[1].REG_COEF_i (DF	0	11	5	0	0
GEN_COEFF_REGS[2].REG_COEF_i (DF	0	7	3	0	0
GEN_COEFF_REGS[3].REG_COEF_i (DF	0	6	3	0	0
GEN_COEFF_REGS[5].REG_COEF_i (DF	0	10	4	0	0
GEN_COEFF_REGS[6].REG_COEF_i (DF	0	13	4	0	0
GEN_COEFF_REGS[7].REG_COEF_i (DF	0	10	8	0	0
GEN_COEFF_REGS[8].REG_COEF_i (DF	0	4	3	0	0
GEN_COEFF_REGS[9].REG_COEF_i (DF	0	6	4	0	0
GEN_COEFF_REGS[10].REG_COEF_i (D	0	10	4	0	0
GEN_COEFF_REGS[11].REG_COEF_i (D	0	8	5	0	0
GEN_SHIFT_REG[0].FIRST_REG.DFF_X(	1	16	9	1	0

Figure 30.1: (1) Resources Utilization in Detail

GEN_SHIFT_REG[1].INTERNAL_REGS.D	1	16	10	1	0
GEN_SHIFT_REG[2].INTERNAL_REGS.D	1	16	10	1	0
GEN_SHIFT_REG[3].INTERNAL_REGS.D	1	16	11	1	0
GEN_SHIFT_REG[4].INTERNAL_REGS.D	1	16	11	1	0
GEN_SHIFT_REG[5].INTERNAL_REGS.D	1	16	7	1	0
GEN_SHIFT_REG[6].INTERNAL_REGS.D	1	16	8	1	0
GEN_SHIFT_REG[7].INTERNAL_REGS.D	1	16	9	1	0
GEN_SHIFT_REG[8].INTERNAL_REGS.D	1	16	10	1	0
GEN_SHIFT_REG[9].INTERNAL_REGS.D	1	16	10	1	0
GEN_SHIFT_REG[10].INTERNAL_REGS.	1	16	11	1	0
GEN_SHIFT_REG[11].INTERNAL_REGS.	0	16	13	0	0
GEN_SHIFT_REG[12].INTERNAL_REGS.	0	16	12	0	0
GEN_SHIFT_REG[13].INTERNAL_REGS.	0	16	10	0	0
GEN_SHIFT_REG[14].INTERNAL_REGS.	0	16	6	0	0
GEN_SHIFT_REG[15].INTERNAL_REGS.	0	16	8	0	0
GEN_SHIFT_REG[16].INTERNAL_REGS.	0	16	10	0	0
GEN_SHIFT_REG[17].INTERNAL_REGS.	0	16	6	0	0
GEN_SHIFT_REG[18].INTERNAL_REGS.	0	16	8	0	0
GEN_SHIFT_REG[19].INTERNAL_REGS.	0	16	9	0	0
GEN_SHIFT_REG[20].INTERNAL_REGS.	0	16	9	0	0
GEN_SHIFT_REG[21].INTERNAL_REGS.	0	16	8	0	0
GEN_SHIFT_REG[22].LAST_REG.DFF_X	0	16	8	0	0
GEN_SYMMETRIC_SUMS[0].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[1].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[2].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[3].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[4].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[5].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[6].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[7].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[8].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[9].REG_SYM_S	0	17	5	0	0
GEN_SYMMETRIC_SUMS[10].REG_SYM_	0	17	5	0	0
REG_OUTPUT (DFF_N_53)	0	16	5	0	0

Figure 30.1: (2) Resources Utilization in Detail

Power Consumption

As a final analysis below in **Figure 31** it is possible to see the report on power consumption.

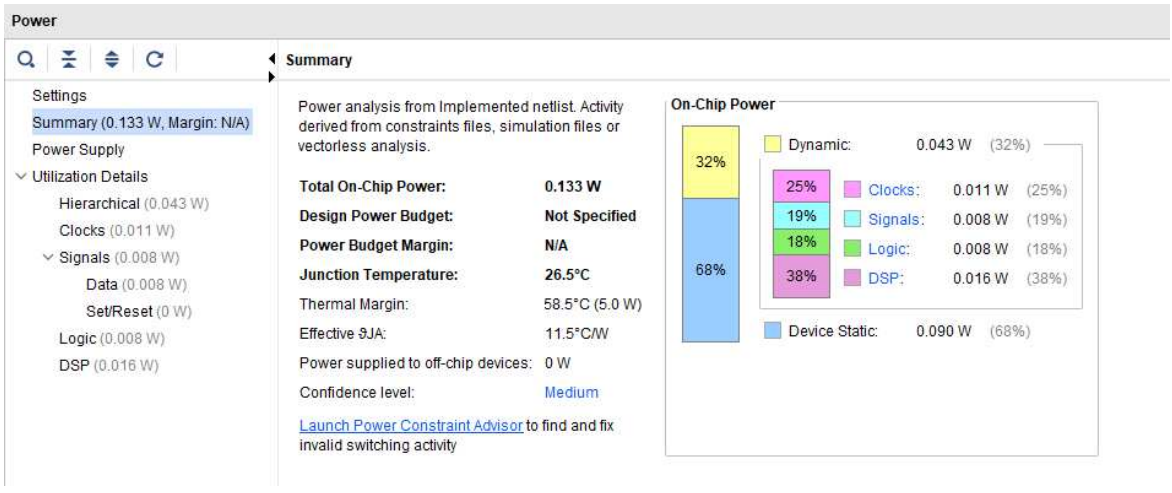

Figure 31: Power Report

The power analysis shows us that the design has a total power of about **133 mW**, which is very low and it is a good thing, of which **68%** is due to **static** power and only **32%** to **dynamic** power. The prevalence of static power is typical of a simulation without real switching *(out_of_context),* and is an indication that most of the dissipation is to be attributed to possible transistor leakage rather than to the actual logic activity of the circuit.

However, the dynamic power distribution indicates a **balanced** use of **resources**, with particular emphasis on the DSP blocks and clocks as seen earlier in the device utilization analysis. Thus, this reflects that a good design has been made, geared toward performance efficiency and throughput, optimizing the use of the FPGA's dedicated hardware without waste.

Obviously in a real application, the circuit will be more active, so it is expected that ***dynamic power*** will grow and ***static power*** will weigh less in percentage.

Conclusion

The design, simulation, and implementation of the SRRC FIR filter carried out in this project successfully met all the functional and structural requirements defined at the beginning. Through extensive simulation campaigns in both high-level (Python) and RTL (VHDL via ModelSim) environments, the filter demonstrated correct behavior in a variety of test scenarios, including impulse response, ISI mitigation, out-of-band attenuation and limt cases such as large numbers.

From an architectural point of view, the chosen implementation leveraged a pipelined structure with coefficient symmetry optimization and adder-tree summation, providing a good trade-off between area usage, latency, and timing performance. Despite not opting for more complex alternatives such as systolic arrays or distributed arithmetic—which are more suited for very high-order filters—the design proved to be efficient and well-aligned with the constraints and educational objectives of the project.

The verification process included all the standard implementation steps on Vivado, up to the synthesis and implementation phases, confirming both the correctness of the design and the absence of timing violations. Minor warnings due to the out-of-context configuration were deemed acceptable within the given development context.

Future Works

Although the current design meets the required objectives, several improvements and extensions could be considered in future developments:

- **Parametric Architecture:** Generalizing the VHDL code to support configurable filter order and roll-off factor would improve reusability and flexibility.
- **Hardware Validation:** Integrating the design on actual FPGA hardware (e.g., using the ZYBO board) would allow for real-time validation, power measurement, and I/O testing.
- **Alternative Architectures:** Exploring systolic or distributed arithmetic implementations could be beneficial for higher-order versions of the filter where pipelining alone may not be optimal.
- **Fixed-point Optimization:** Further tuning the fixed-point representation to balance dynamic range and resource usage could lead to more efficient implementations.

References

[1] Design and Implementation of FIR Filter Architecture using High Level Transformation Techniques; Electronics and Communication Engineering, M. Kumarasamy College of Engineering, Karur, Thalavapalayam – V. Jamuna*, P. Gomathi and A. Arun

[2] Realization of FIR filter using modified Distributed Arithmetic Architecture; (1) Department of Electronics and Communication Engineering, Saveetha Engineering College, Chennai, Tamil Nadu, India. – (2) Department of Electronics and Communication Engineering, SRM University, Kattankulathur, Chennai, Tamil Nadu, India - Ramesh .R, Nathiya .R

[3] https://www.typhoon-hil.com/documentation/typhoon-hil-software-manual/References/fir_filter.html

[4] https://github.com/g-sferr