UNIVERSITÀ DI PISA

Computer Engineering
Electronics and Communications Systems

Finite Impulse Response filter with SRRC impulsive response

Project Documentation

Matteo Guidotti

Academic Year 2020/2021

Contents

1 Introduction

1.1 Specifications

This project is about the implementation of a **Finite Impulse Response filter (FIR)** characterized by a **Square Root Raised Cosined (SRRC)** impulsive response. The filter is characterized by:

- Filter order equal to N = 22
- Number of samples per symbol = 4
- Roll-Off factor = 0.5
- The coefficients are:

$$c_0 = c_{22} = -0.0165; c_1 = c_{21} = -0.0150; c_2 = c_{20} = 0.0155; c_3 = c_{19} = 0.0424;$$

$$c_4 = c_{18} = 0.0155; c_5 = c_{17} = -0.0750; c_6 = c_{16} = -0.1568; c_7 = c_{15} = -0.1061;$$

$$c_8 = c_{14} = 0.1568; c_9 = c_{13} = 0.5786; c_{10} = c_{12} = 0.9745; c_{11} = 1.1366.$$

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

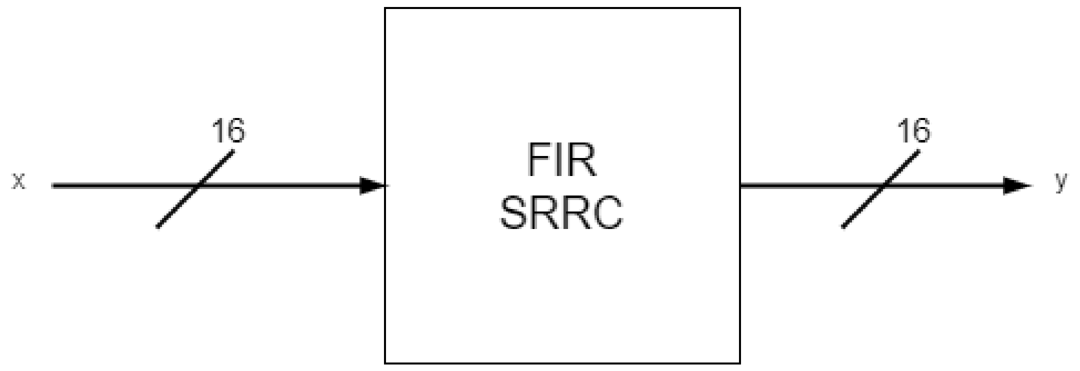


Figure 1: Filter scheme

1.2 The intersymbol interference

In telecommunication, intersymbol interference (ISI) 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 a similar effect as noise, thus making the communication less reliable. The spreading of the pulse beyond its allotted time interval causes it to interfere with neighboring pulses. ISI is usually caused by multipath propagation or the inherent linear or non-linear frequency response of a communication channel causing successive symbols to blur together. The presence of ISI in the system introduces errors in the decision device at the receiver output. 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.

1.3 The SRRC filter

The **Square Root Raised Cosine filter (SRRC)**, sometimes known as **Root Raised Cosine** filter, is frequently used as the transmit and receive filter in a digital communication system to perform matched filtering. This helps in minimizing intersymbol interference (ISI). The combined response of two such filters is that of the raised-cosine filter. It obtains its name from the fact that its frequency response is the square root of the frequency response of the raised-cosine filter. To have minimum ISI, the overall response of transmit filter, channel response and receive filter has to satisfy Nyquist ISI criterion. The raised-cosine filter is the most popular filter response satisfying this criterion. Summing up, the inter-symbol interference (ISI) deteriorates the received symbol quality in a band-limited digital communication system, but as I mentioned before it is well known that when a pair of matched SRRC filters are employed in the transmitter and the receiver, the overall frequency response of the system has a raised-cosine spectrum such that the ISI is gone at the time when the symbol is sampled. Unfortunately, ideal SRRC filters are not realizable. In reality, they can only be approximately implemented. In particular, such filters can be approximated with **Finite Impulse Response (FIR)** filter

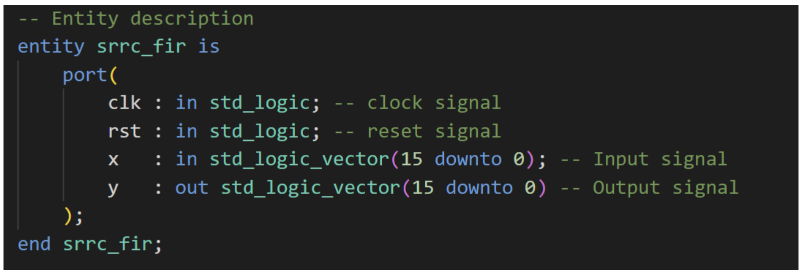
1.4 Finite Impulse Response filters

A finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time. This is in contrast to infinite impulse response (IIR) filters, which may have internal feedback and may continue to respond indefinitely (usually decaying). FIR filters can be discrete or continous time, in our case we'll use the first one. For such a filter of order N, each value of the output is a weighted sum of the N+1 most recent input values, where the weights are the N+1 coefficients of the filter. FIR filters have also other important characteristics:

- They require no feedback. This means that any rounding errors are not compounded by summed iterations. The same relative error occurs in each calculation.
- They are inherently stable, since the output is a sum of a finite number of finite multiples of the input values
- They can easily be designed to be linear phase by making the coefficient sequence symmetric. This property is sometimes desired for phase-sensitive applications, for example data communications, seismology, crossover filters, and mastering.
- The main disadvantage of FIR filters is that considerably more computation power in a general purpose processor is required compared to an IIR filter with similar sharpness or selectivity

2 Architecture Description

The FIR filter has been implemented by means of the VHDL code language and it is available in the file *VHDL/src/srrc_fir.vhd*.
The filter is characterized by three input and an output signals, that are the clock and the reset signal (1 bit) and x, that is the signal to be filtered (16 bits), in addition to y, that is the result of the filtering (16 bits).



```
-- Entity description
entity srrc_fir is
    port(
        clk : in std_logic; -- clock signal
        rst : in std_logic; -- reset signal
        x   : in std_logic_vector(15 downto 0); -- Input signal
        y   : out std_logic_vector(15 downto 0) -- Output signal
    );
end srrc_fir;
```

Figure 2: Entity structure

The computation of the result requires to use the last 23 signal samples, so it is necessary to store 22 samples, because the most recent one is simply taken by the input signal. In order to store them, 22 D-flip-flops are instantiated and at every clock rising edge they will be uploaded with the correct value, with the respect to their positioning. The DFFs are implemented using a synchronous active low reset. Given that the multiplier circuits are very expensive from the power and time efficiency point of views, I decided to exploit the symmetry of the coefficients to instantiate only 12 multipliers, instead of 23. So, before the multiplier circuits, every couple of samples that must be multiplicated with the same coefficients are summed up and each output will be then multiplied by the related coefficient. This is true for all the stored samples, except for the one that must be multiplied with the c_{11} coefficient, that must not be summed to any other sample. Then, all the outputs of the multiplications must be summed up together in order to obtain the final result. After the simulation of the testbench I decided to represent the result of the first sums with 17 bits in order to be sure to not loose any information, and I applied the same reasoning to the result of the multiplications, so that the related output are represented using 33 bits. I decided, instead, to not consider the carry out of the last sums because I noticed that I would not loose any information. At the end, the final output must be represented with 16 bits, because of the specification; by observing the result of the testbench I decided that the best configuration is the one in which the 15 LSBs of the output of the final sum are truncated and the MSB is not taken into account, so that we have a saturation of 1 bit.

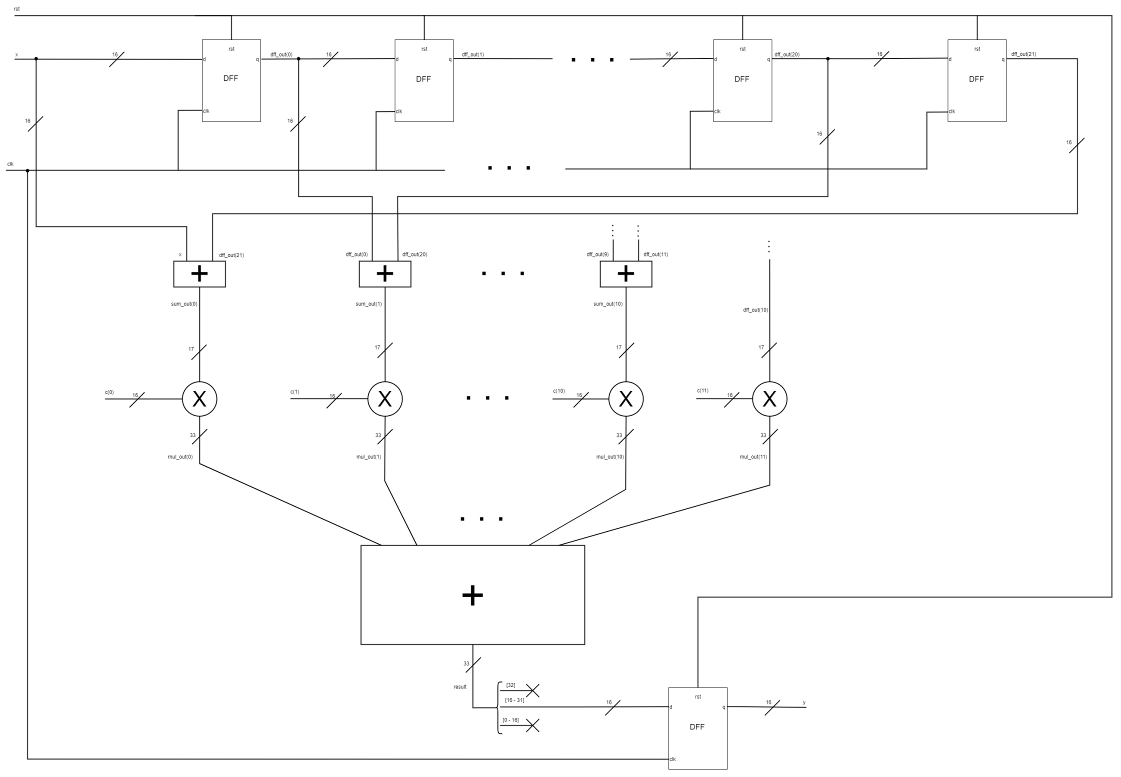


Figure 3: Architecture schema

3 Test Plan

In order to test the correct functioning of the implemented filter, I decided to simulate a sin waveform as the input signal. Now I will present all the utilities that I implemented so that I could be able to perform such a test.

3.1 Coefficients Quantization

The coefficients must be represented with 16 bits and they are all real numbers, so it is necessary to quantize them with a LSB in order to be represented in a fixed-point manner. So the values that will be represented on 16 bits are obtained by dividing each coeffcient by the LSB and applying the floor function to the result. As a result, the quantization function is:

$$Q[x] = floor\left(\frac{x}{LSB}\right) \cdot LSB$$

For the sake of simplicity, I decided to use LSB that is a power of 2 and given that all the coefficients are contained in the set $[-0.1568; 1.1366]$ I compute the mean quantization errors by using LSBs that were as small as possible and, at the same time, that return a result representable with 16 bits. The result of this study is that the best LSB for the quantization of the coefficients is 2^{-14}, which consent me to represent the coefficients with a mean quantization error equal to $3.38 * 10^{-5}$. The functions contained in the file *coefficient_quantization.py* compute the quantized values and print them in two different files. The file *coefficients.txt* will contain the values as they are, one for each row; while *coefToCode.txt* will contain the result of the computations in a way so that it consents me to copy and paste it directly to the vhd code. Then, the python utility compute also the mean quantization error for the just computed values.

3.2 Binary Conversion

The *binary_conversion.py* file contains a function that, given as an input an integer number, returns as a result its two's complement representation on 16 bits

3.3 Signal Generator

The *signal_generator.py* file is responsible for generating an input for the filter. It generates a sin waveform signal computed on a customizable number of frequencies contained between 0 and the period of the waveform. The utility contains also a function that, given a LSB, compute the quantization for the signal values and the related quantization error. I decided to use a power of 2 LSB in this case too and, using the formula below, I found that the best LSB to represent values in the set $[-1; 1]$ is 2^{-15}, obtaining a mean quantization error equal to $1.53 * 10^{-5}$

$$LSB = \frac{|\min x|}{2^{N-1}}$$

The quantized values are printed in two different files in two different formats.

The file *quantized_signal.txt* will contain the quantized values as they are, one for each row; while *signal_to_code.txt* will contain the quantized values that are at first converted to a 16 bits representation and then encoded in a way so that I'm able to copy and paste it directly to the testbench file in the case statement that consents to the input signal to vary.

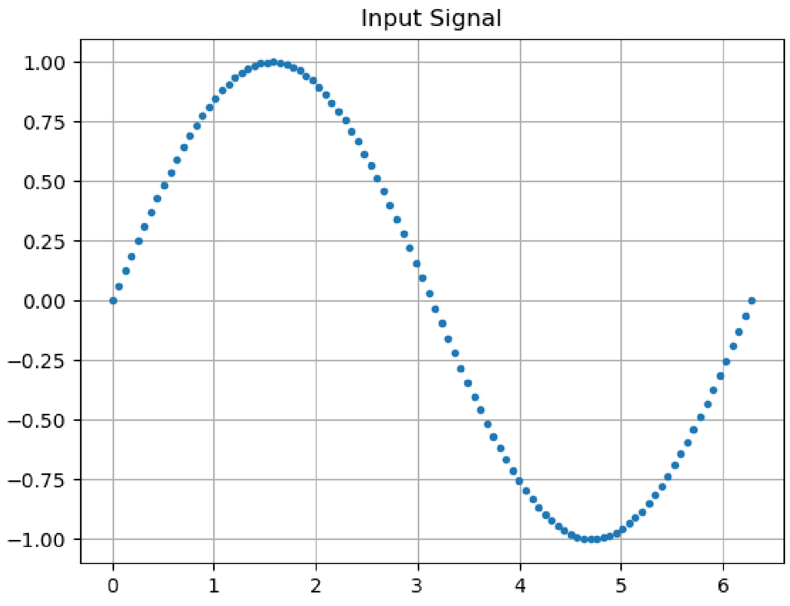


Figure 4: Example of 100 values between 0 and $2 * \pi$

3.4 FIR Software Simulation

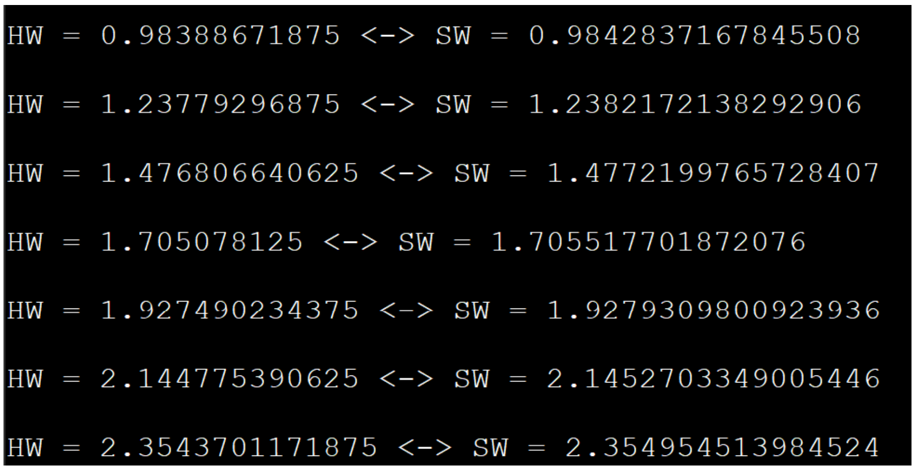
In order to check the correctness of the output values returned by the implemented filter in vhdl, I decided to implement the *fir_simulation.py* utility, that simulates in software the computations that the filter must perform. This script will request the input signal to the same utility that generates the input signal for the testbench and then it will calculate the sequence of output values. The results are printed to the file *fir_sw_simulation_output.txt*: in each row of this file, a concatenation of the last input and the result of the previous output result is printed. This format is due to the fact that, in the hardware filter, the output is delayed with the respect of the input and so I simulated this behaviour also in the software implementation, such that the comparison between the results would be simpler.

3.5 VHDL Testbench

The file *srrc_fir_tb.vhd* is the testbench file used in ModelSim to execute my filter implementation and check its behaviour. At first all the flip-flop are reset, then the reset signal is driven high in order to consent the regular functioning of the DFFs. Starting from the third clock cycle, the input changes its values every clock cycle until the 103^{th} cycle, in which the simulation is stopped. The case statement that regulates this behaviour is returned by the signal generator utility, as I already mentioned. At the end of the simulation, it is necessary to export an event list that reports all the output values returned by the filter in order to check them respect to the software simulation results.

3.6 Result Checker

Once we obtained the results from both the software and the hardware simulation of the filter it is possible to check the correctness of the results obtained through the testbench simulation. The *result_checker.py* utility is responsible for taking the two files containing the outputs of the respective simulations and compare them. In order to perform a fair comparison, the file requires the LSB that must be used to interpret the values outputted by the testbench execution. The execution of the result checker will write one file and print to the standard output the mean error value obtained by the comparison of each value returned by the two simulations. Instead, the file *results_comparison.txt* will contain, for each row, the concatenation of the software output and the relative hardware one, in order to have an immediate comparison between the two executions.



```
HW = 0.98388671875 <-> SW = 0.9842837167845508

HW = 1.23779296875 <-> SW = 1.2382172138292906

HW = 1.476806640625 <-> SW = 1.4772199765728407

HW = 1.705078125 <-> SW = 1.705517701872076

HW = 1.927490234375 <-> SW = 1.9279309800923936

HW = 2.144775390625 <-> SW = 2.1452703349005446

HW = 2.3543701171875 <-> SW = 2.354954513984524
```

Figure 5: Snippet of *results_comparison.txt*

3.7 Results Comparison

Thanks to these python scripts I has been able to check that the behaviour of my implementation of the fir filter is able to return the correct values. As I already mentioned, the final results is representable with 33 bits, but the specifications consent me to use only 16 bits for the output signal. Given that the coefficients were represented using a LSB of 2^{-14} and the input signal exploited a LSB equal to 2^{-15}, the resulting LSB for the final result is 2^{-29}, due to the multiplications between input signal samples and coefficients. This fact means that I have a maximum of 4 bits to represent the integer part of the results, but by watching at the results I observed that they were always contained in $(-4; 4)$, so it made me think that I could try to saturate the result by discarding the most significant bit.
I performed the testbench simulation at first by obtaining the outputs discarding the 17 least significant bits of the final results and I measured a mean error value of $4.67996 * 10^{-4}$. On the other hand, the simulation has been also performed by truncating 16 least significant bits and discarding the most significant one, obtaining a mean error value of $4.493246 * 10^{-4}$. While checking the results is important to tune in a correct way the LSB used to interpret the output values of the hardware simulation, in fact it depends on how many bits are truncated. When 17 bits are truncated, it will be equal to 2^{-12}, while in the other case it will be 2^{-13}. Said that, it is clear that both the two results are very good from the point of view of the precision of the results, it would occur some additional information on the input signals to define which bits is better to discard. Obviously enough, if the signal is often characterized by values near the 0, it would be better the saturation scenario, while if the signal is often near, in absolute value, to 1, it would be better to choose the only truncation method. However, given the very small but observable difference between the two mean error values, I decided to use the configuration in which the most significant and 16 least significant bits are discarded.

4 Synthesis and Implementation

Before performing the synthesis, the implementation of a vhdl wrapper is necessary. Vivado tool presents us the filter in this way:

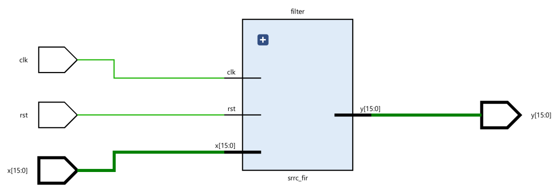
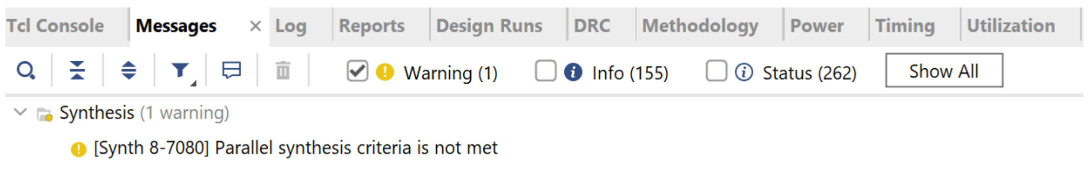


Figure 6: Schematic of the filter

4.1 Warning Analysis

The only warning that popped up arise during the synthesis phase, that is:



Synthesis (1 warning)
[Synth 8-7080] Parallel synthesis criteria is not met

Figure 7: Synthesis warning

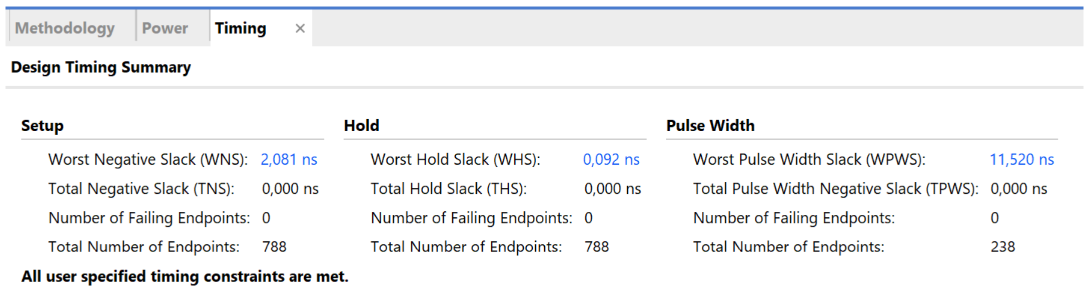
Let's start by understanding what is the *Parallel synthesis*: from the https://support.xilinx.com/ webpage we can read:

"In order to improve overall runtime of the design, Vivado synthesis launches parallel processes if the design is large enough and can benefit from the parallel flow. The parallel flow partitions the design into smaller 'RTL Partitions' which are processed independently by parallel processes. Vivado Synthesis will decide to use parallel flow only if the design size is large enough."

So the warning simply indicates that the design is not large enough to benefit from parallel synthesis. As a result, the parallel flow will not be run. This fact will not be a problem for the characteristics and the performance of my implementation.

4.2 Timing Summary

By imposing the constraint in which the clock has a period of $25ns$, the timing summary at the implementation stage is the following:



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2,081 ns	Worst Hold Slack (WHS):	0,092 ns	Worst Pulse Width Slack (WPWS):	11,520 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	788	Total Number of Endpoints:	788	Total Number of Endpoints:	238

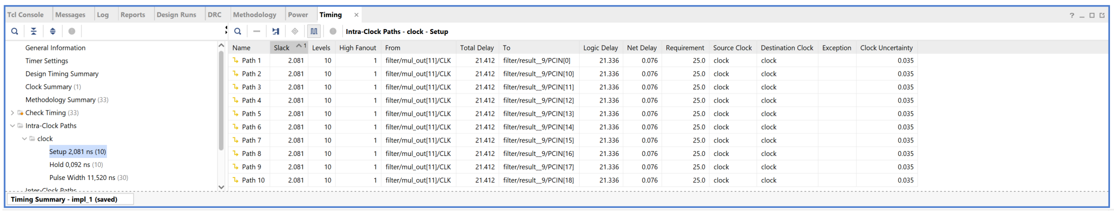
All user specified timing constraints are met.

Figure 8: Timing summary

As a consequence, the theoretical maximum usable frequency is:

$$f_{max} = \frac{1}{(25 \; - \; 2.081) \cdot 10^{-9}} \approx 43.6 \; MHz$$

The resulting critical path is the following



Intra-Clock Paths - clock - Setup

- General Information
- Timer Settings
- Design Timing Summary
- Clock Summary (1)
- Methodology Summary (33)
- Check Timing (33)
- Intra-Clock Paths
  - clock
    - Setup 2,081 ns (10)
    - Hold 0,092 ns (10)
    - Pulse Width 11,520 ns (30)
- Inter-Clock Paths

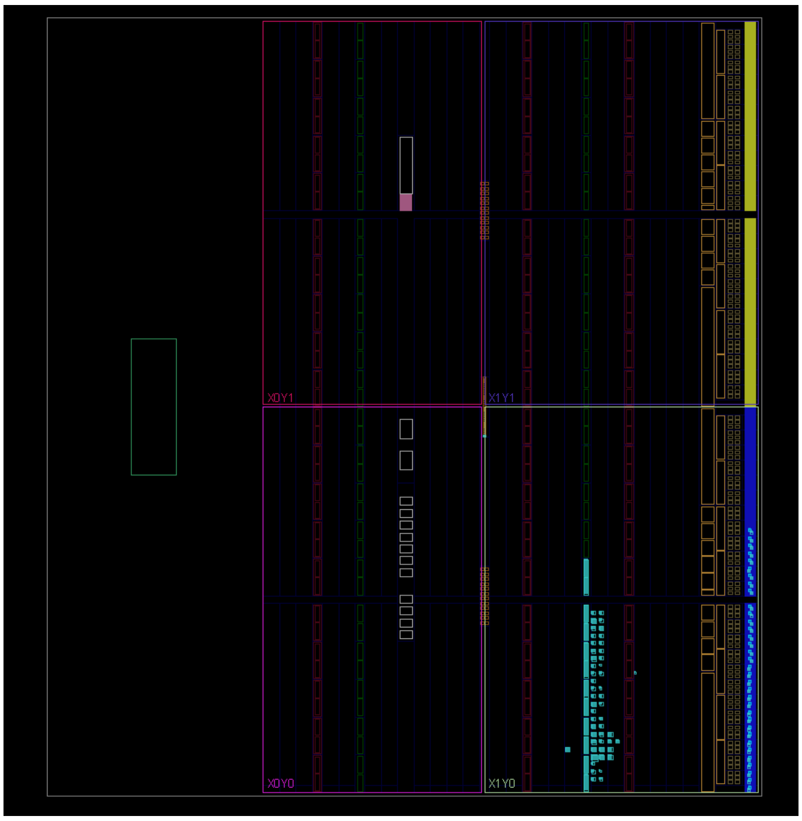
Timing Summary - impl_1 (saved)

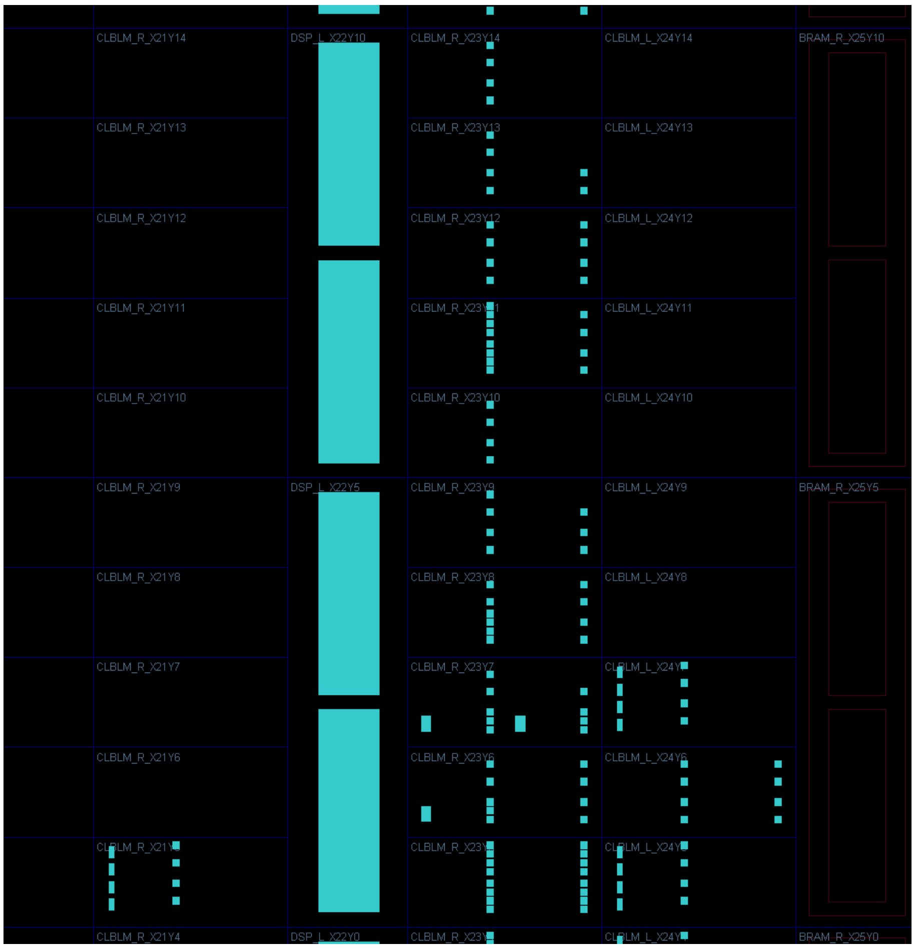
Name	Slack	Levels	High Fanout	From	Total Delay	To	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[0]	21.336	0.076	25.0	clock	clock		0.035
Path 2	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[10]	21.336	0.076	25.0	clock	clock		0.035
Path 3	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[11]	21.336	0.076	25.0	clock	clock		0.035
Path 4	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[12]	21.336	0.076	25.0	clock	clock		0.035
Path 5	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[13]	21.336	0.076	25.0	clock	clock		0.035
Path 6	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[14]	21.336	0.076	25.0	clock	clock		0.035
Path 7	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[15]	21.336	0.076	25.0	clock	clock		0.035
Path 8	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[16]	21.336	0.076	25.0	clock	clock		0.035
Path 9	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[17]	21.336	0.076	25.0	clock	clock		0.035
Path 10	2.081	10	1	filter/mul_out[11]/CLK	21.412	filter/result__9/PCIN[18]	21.336	0.076	25.0	clock	clock		0.035

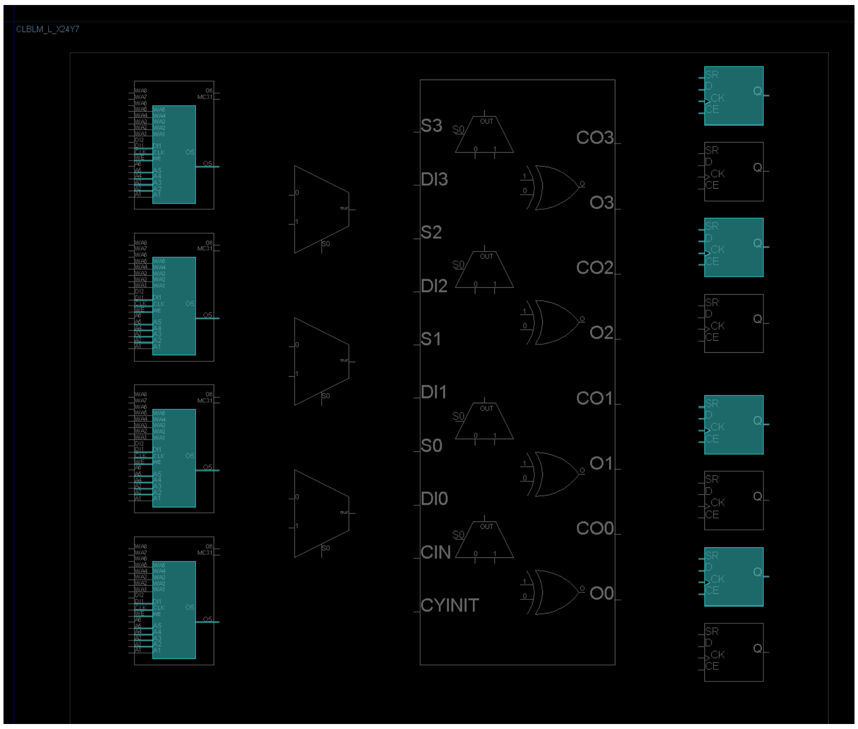
Figure 9: Critical path

4.3 Area Occupation

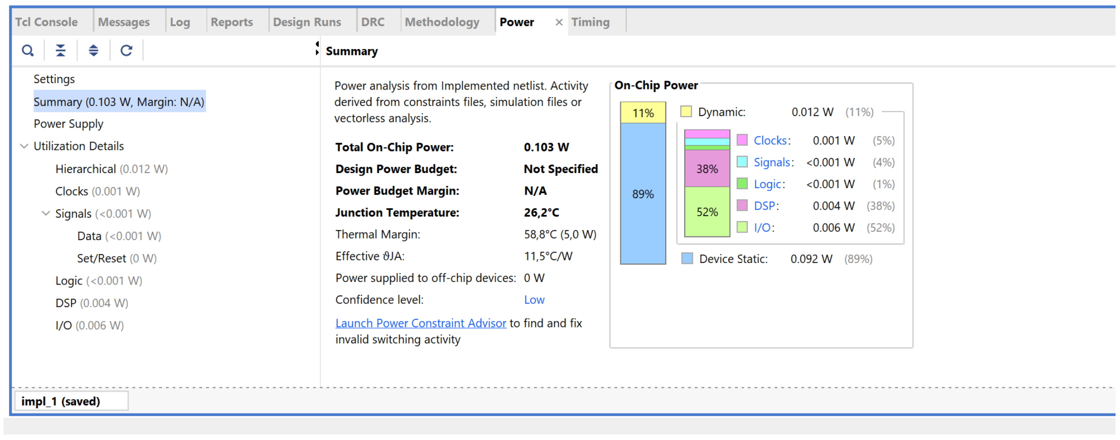
In the following, there are the screenshots that show the area occupied by the filter




CLBLM_R_X21Y14
DSP_L_X22Y10
CLBLM_R_X23Y14
CLBLM_L_X24Y14
BRAM_R_X25Y10
CLBLM_R_X21Y13
CLBLM_R_X23Y13
CLBLM_L_X24Y13
CLBLM_R_X21Y12
CLBLM_R_X23Y12
CLBLM_L_X24Y12
CLBLM_R_X21Y11
CLBLM_L_X24Y11
CLBLM_R_X21Y10
CLBLM_R_X23Y10
CLBLM_L_X24Y10
CLBLM_R_X21Y9
DSP_L_X22Y5
CLBLM_R_X23Y9
CLBLM_L_X24Y9
BRAM_R_X25Y5
CLBLM_R_X21Y8
CLBLM_R_X23Y8
CLBLM_L_X24Y8
CLBLM_R_X21Y7
CLBLM_R_X23Y7
CLBLM_R_X21Y6
CLBLM_R_X23Y6
CLBLM_L_X24Y6
CLBLM_R_X21Y4
DSP_L_X22Y0
BRAM_R_X25Y0


CLBLM_L_X24Y7
S3
DI3
S2
DI2
S1
DI1
S0
DI0
CIN
CYINIT
CO3
O3
CO2
O2
CO1
O1
CO0
O0
SR
D
CK
CE
Q

4.4 Power Summary



Settings
Summary (0.103 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (0.012 W)
Clocks (0.001 W)
Signals (<0.001 W)
Data (<0.001 W)
Set/Reset (0 W)
Logic (<0.001 W)
DSP (0.004 W)
I/O (0.006 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.103 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,2°C**
Thermal Margin:	58,8°C (5,0 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	0.012 W	(11%)
🟪	Clocks:	0.001 W	(5%)
🟦	Signals:	<0.001 W	(4%)
🟩	Logic:	<0.001 W	(1%)
🟪	DSP:	0.004 W	(38%)
🟩	I/O:	0.006 W	(52%)
🟦	Device Static:	0.092 W	(89%)

Figure 10: Power Summary

5 Conclusions

In conclusion, I realized a FIR filter with a SRRC impulsive response that takes as an input a signal represented on 16 bits and returns as an output a 16 bits signal. The performed checks demonstrated that the filter performs well, with a very good precision. The maximum usable frequency could be incremented by dividing the combinatorial network in different sections, interleaved by flip-flops, in order to obtain a pipeline with differnt stages, but this lies outside from the objective of this project and I preferred a lower frequency, in order to have an output that is not too much delayed with the respect of the related input