Universidad del Bío-Bío
Engineering faculty
Department of Electrical and Electronic Engineering

"Estudy, implementation and verification of a PDM decoder circuit for a Sonar acoustics Application"

Author: Maximiliano Roberto Cerda Cid
Professor: Krzysztof Herman

Thesis project to obtain the degree of Automation and control engineering BSc.

September 2023
Concepción, Chile

In thanks to Dr. Ing. Krzysztof Herman and colleagues involved in the Sonar On Chip *project*

This page was intentionally left blank

Abstract

The core of this project belongs to a bigger one called "Sonar On Chip" which is a digital front-end designed to capture and process acoustic signals coming from an *MEMS* microphone array [1].

one of the facts that makes this project interesting is because lots of solutions designed in discrete components forming a complete embedded system are migrating to a single integrated circuit. This phenomenon is possible due to the continuous advances of the microelectronic industry. Such that, by 2010 due to the transistors reduction, a single grain of rice was equivalent in cost of a 100 thousand transistors [2]. Such advances are enabling the possibility to migrate a complete system to a single Chip, this tendency is called *System On Chip*.

In order to not deviate from the main topic, the next steps are a general description of the main sections of this project. First of all, the state of the art solutions related to sonar applications are reviewed, specifically air-coupled sonars. Then, it comes a general explanation of the building blocks that compound the entire system, beginning from the signal acquisition, and the processing of the acquired signal.

With this idea of the problem and the possible solutions, we get into details of the core of this project, in this case we only focus on a single stage of "Sonar On Chip" design, the verification of decoder circuit of the sonar system. to address that task we will explain in a general manner the *ASIC* design flow and *FPGA* deisgn flow, and why is usefull verify a design on the FPGA. Finally, we give a description of the architecture solution of this project for the *PDM* decoder circuit (digital front-end). generate behavioral simulation based on PDM stimulus generated in *Python*, also a post-implementation test of the decoder circuit in FPGA with the corresponding results.

Keywords– *Sonar On Chip, MEMS, System On Chip (SOC), ASIC, FPGA, PDM, Python.*

Abbreviations

- ***Sonar On Chip***: A sonar on a chip.
- ***MEMS***: Micro electro-mechanical system.
- ***SOC***: System On Chip.
- ***CIC***: Cascaded Combinational-integrator filter.
- ***SRP***: Stereed Response Power,
- ***PDM***: Pulse Desnity Modulation.
- ***CRRS***: Cascaded Recursive Runing Sum filter.
- ***FPGA***: Field Programmable Gate Array.
- ***ECM***: Electro-condencer Microphone.
- ***ADC***: Analog to Digital Converter.
- ***GPU***: Graphic Processing Unit.
- ***TF***: Transfer function.
- ***TFD***: Discrete fourier transform.
- ***OSR***: Oversampling Ratio.
- ***STF***: Signal Transfer Function.
- ***NTF***: Noise Transfer Function.
- ***HPF***: High Pass Filter.
- ***MAF***: Moving Average Filter.
- ***RRS***: Recursive Runing Sum Filter.
- ***PCM***: Pulse Code Modulation.
- ***LTI***: Lineal Time Invariant System.
- ***CAD***: Computer Aided Design.

- ***HDL***: Hardware Description Language.
- ***PLA***: Programmable Logic Array.
- ***SOP***: Sum of Products.
- ***CLB***: Configurable Logic Block.
- ***AXI***: Advanced Extensible Interface.
- ***IDE***: Integrated Development Environment.
- ***Verilog***:The Verification and logic HDL.
- ***VHDL***: Very High Speed Integrated Circuit HDL.
- ***IC***: Integrated Circuit.
- ***I/O***: Input-Output.
- ***LE***: Logic Element.

Contents

List of Figures

1 Introduction

Echolocation techniques have been widely applied, from underwater sonar, acoustic cameras and to ultrasonic scanning techniques[11]. However, its study has become attractive again due to improvements in computing power and sensor technology.

Due to the rapid development of microelectronics, it is possible to acquire microphones which contain transducer, conditioning and signal conversion in the form of a single Integrated Circuit at a very low cost as is the case of MEMS microphones[10].This makes it possible, in conjunction with current HW, to use arrays of sensors whose signals are processed by devices such as an *FPGA* or *ASIC* as a superior alternative to general purpose processors. They use spatial filtering algorithms such as beamforming and other computational operations such as *SRP*, *TDOA*, *HRES*, which support sound,image generation and Object Detection applications, in this case using air-coupled microphones. All this in a single embedded device[10].

In the previous scenarios of applications, we can be sure that 3 major task will be involved, the first one is signal conditioning and acquisition, in second place the application of data conversion circuits and filtering technique's and finally the execution of some of the previous mentioned algorithm's. The core of the project is based in the stage of data conversion, in this case the conversion from *PDM* format coming from a *MEMS* microphone to *PCM* format. as we can see further some general architectures are proposed by several authors to tackle this kind of applications. We will study some of the state of the art solutions based on air-coupled microphone array processing techniques. as we gain a general frame of the project we introduce the main task of this project, the verification of a *PDM* to *PCM* conversion circuit based on a specific class of filter called *CIC* filter proposed by [8].In this case, the specific implementation is based on a *CRRS* architecture proposed by the authors [12].

This specific PDM decoder circuit is one of the several blocks involved in the *Sonar On Chip* project. this project implements a digital system for signal processing to capture and process acoustics signals from 36 MEMS microphones with an extended frequency range up to 85 kHz (low ultrasonic band). The system itself is a part of the Caravel harness and can be configured and managed from the Caravel using Wishbone bus [1]. A version of 8 channels of the project was successfully designed and send to foundry

sponsored by *SSCS*. As we discuss further the verification phase involving *FPGA* is crucial to a successfully design, in this case this project involves the verification of this *PDM* decoder circuit in a implemented design on a *FPGA*. Hence this way the designers can ensure that a correct design is sent to fabrication. Several iterations using *CAD* tools and *FPGA* or *HW* emulators are done previously to avoid post-fabrication errors.

Keywords *– HW,SRP, TDOA, HRES, PCM, CIC, CRRS, SSCS, CAD.*

1.1 State of the art

To give a general perspective and a main reference is the work done in [10] where the authors present a two general ways to implement an acoustic beamforming based on microphone arrays in fpga where both figures are a simplified version of the original source.

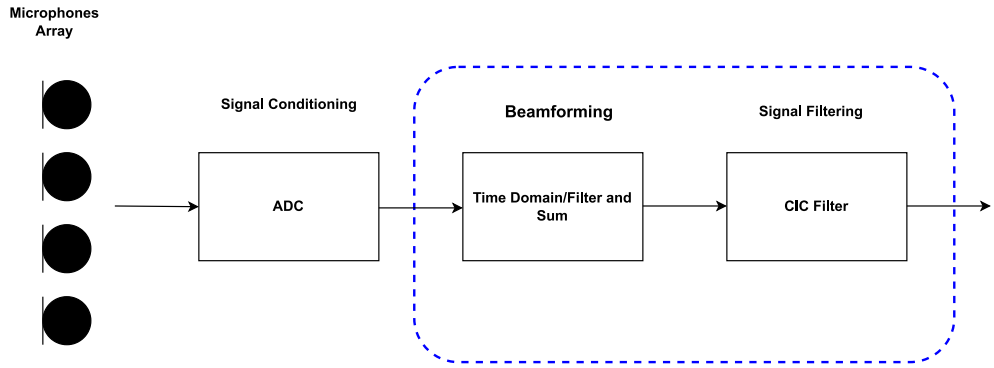

Figure 1.1: Acoustic beamforming for analog microphones

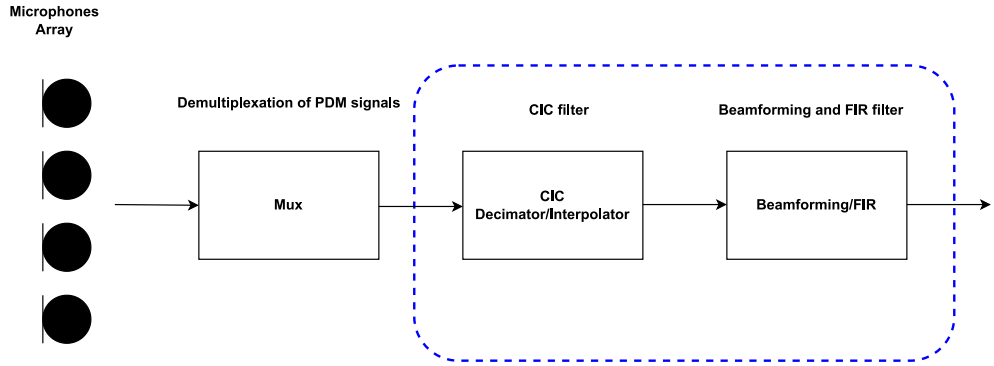

Figure 1.2: Acoustic beamforming for digital microphones

The main difference between this two figures are based in the microphone type. In the market we can find two types, the *ECM* and *MEMS* microphone, both types can be in digital or analog output version. The main difference is in the **figure 1.1** where extra electronics is needed for signal conditioning and conversion, this because the analog input coming from the microphone. Also the case of **figure 1.2** only a demultiplexation channel is used due to the

digital output of the microphone, which means that the sensor already has an *ADC* and signal conditioning.
Once signal is acquire, some *DSP* are needed, in this case we can implement in a interchangeable form the time domain beamforming prior or after the cic filter (blue segmented block). as we can see further is better implement a cic filter for signal decimation or interpolation and then some FIR filter implementation for signal "smoothing" and filtering.

In the case of mems type microphones, which have gained market share in comparison with respect to ecm's [10] they have the signal conversion from analog to digital and also signal conditioning in a single die, given a digital output in pdm format, usually the frequency of the clock in the microphone is slower respect to the fpga/soc, therefore a demux. block is used (**figure 1.2**).Next block to come is the pdm to pcm conversion (the core of this project) in this case for acquire more dynamic range of the signal and also reduce the sampling frequency, for this task the cic filter structure is the best suit due to the economic use of hardware resources as well as they can do some filtering process at the same time as they convert the signal format [8]. finally the last block is the beamforming techniques for spatial filtering and also *FIR* filters are used.

Now that the general structure of the system has been defined, we can describe interesting applications that have arisen in the last decade.the authors of [3], propose a complete embedded system (sensors, processing, image generation) intended for *ADAS*, *UAV* applications to complement the monitoring that is presented mostly based on lidar sensors. Includes an array of circular geometry, decoding and filtering of mems microphones with pdm output using an fpga and the use of an*Nvidia jetson Nano GPU* for the implementation of beamforming and image generation. all this processes are controlled by a microcontroller. As we can see in **figure 1.3** the system has been thinked to utilize small physical space and be modular (it also have the possibility to eliminated the ultrasonic transducer head so they can operate as a passive sonar).

The authors of [4] proposes design considerations for fpga-based sound source localization systems based on microphone arrays. They segment the system in three stages (**figure 1.4**). The first one is the pdm demodulation and filtering stage, the second one is the stage of spatial filtering (for example, beamforming) and the last one is the power stage where algorithms like *P-SRP* is applied so they can infer the position of the source in a polar plane

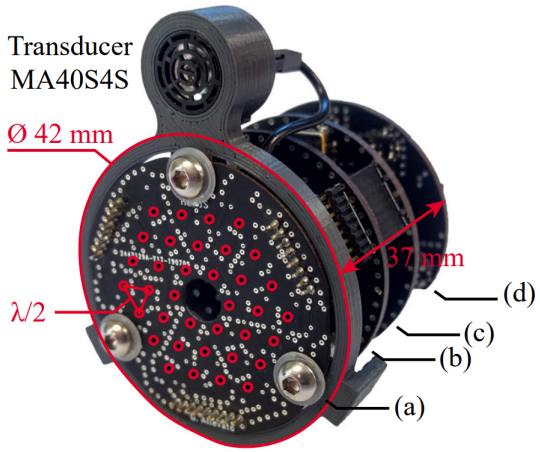

Figure 1.3: Ultrasonic Sonar [3]

where the angle points the angular positions where the power is maximal. Finally we have a similar array of the authors of [3], but in this case the authors of [4] offer a flexible and scaleable solution in such a way that the user can choose the amount of microphones to use selecting sub-arrays in the same array. in that way reducing power consumption sacrificing the least amount of resolution (due to the parallelization capabilities of fpga's.)

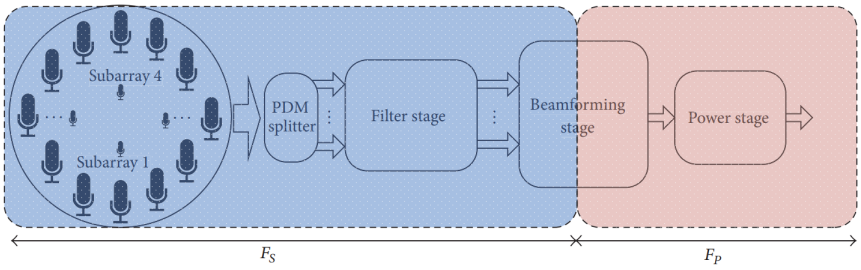

Figure 1.4: Block diagram of the sound source localization system [4]

The authors of [5] propose an microphone array acoustic camera using low-cost fpga's reaching a real time acoustic camera of 10 fps with a resolution of 320x240 pixels with an array of 32 microphones. In the **figure 1.5** we can see the microphones array on the right board and on the left the fpga with his custom pcb board. the array is connected using 4 adc's with 8 channels

each one of them.

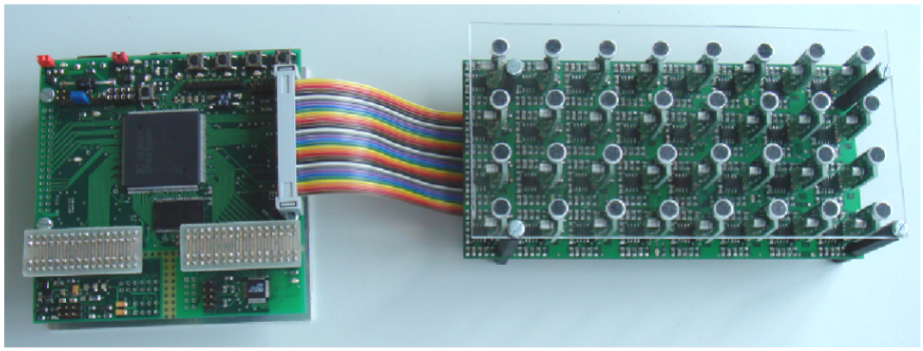

Figure 1.5: Acoustic Camera Prototype [5]

Until this moment only fpga/soc based sonar systems are presented. The authors of [6] propose a 3D ultrasonic rangefinder on a chip. this is the unique case of the works presented that fits into the category of asic (like the Sonar on Chip project) this project integrated a linear array of 10 microphones, including a sigma-delta converter (usually used in traditional commercial MEMS microphones). the main difference is the completed *ToF* sensor fits in a 5 [mm] single package including array of transducers, signal conditioning and data conversion (**figure 1.6**).

Finally the prototype of the system presented in **figure 1.7** shows in the blue board the ultrasonic range finder and also in the green board the fpga that implements the beamforming algorithm. The ToF sensor is capable to detect 3D objects up to 1 [m] of distance, also due level of integration in a single chip the power consumption is reduced (to the order of μW) in comparison to the other works presented (to the order of mW).

Keywords *– ECM, DSP, ADC, FIR, ADAS, GPU, P-SRP, ToF.*

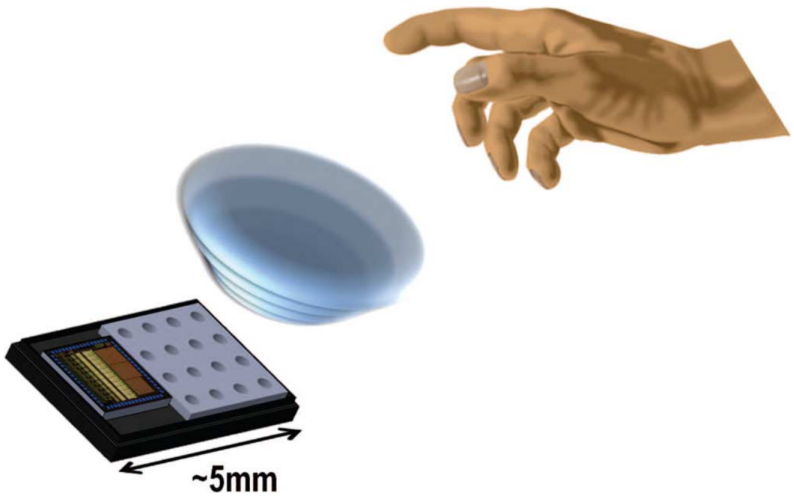

Figure 1.6: A concept design of the ultrasonic rangefinder[6]

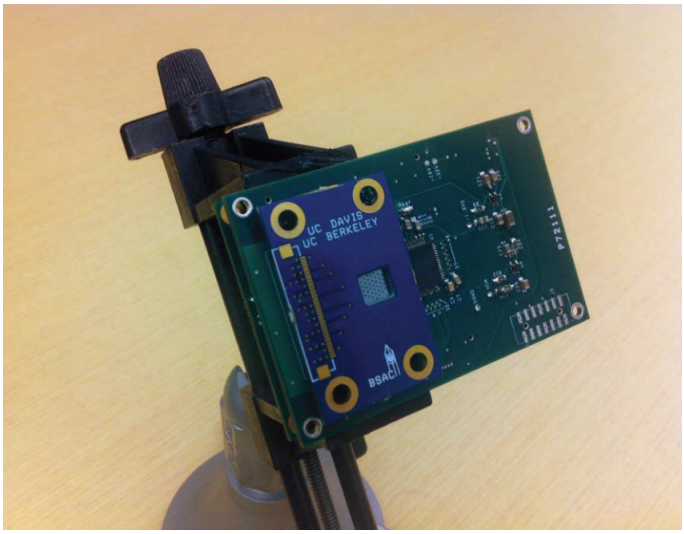

Figure 1.7: Ultrasonic rangefinder prototype [6]

1.2 General Objectives

Implement and verify the PDM decoder circuit from Sonar on Chip project on a FPGA.

1.3 Specific Objectives

Describe in a general way the theory of CIC filters [8] and the implementation of CRRS filter from [12].

Do a behavioral simulation of the CRRS filter version from Sonar on Chip project [1] usign vivado 2018.3 simulator usign a PDM stimulus generated with Python, save the data in a custom ROM memory simulating a PDM microphone on the FPGA.

Do a implementation of the CRRS filter in a *Zybo* Board usign a AXI lite custom ip core, test the filter implemented doing a transfer of data from CRRS filter ip core to 8kb Axi BRAM Controller and finally a transfer to DRR memory from usign Zynq7000 SoC drivers.

interpret and post-process the data using python (compensation and bandpass filters).

1.4 Project Scope

The project involves the generation of the designs of python scripts for data generation, post-processing and data interpretation of the results. the creation and integration of HDL designs of the project, the behavioral and post implementation testing of the several blocks involved in the project.

2 About microphones and Sigma-Delta modulation

2.1 Microphones

As we discuss earlier, ecm and mems microphones are the biggest two in the market, in the first case, the functioning principle of both types of transducers are similar (but the fabrication process differs). in the **figure 2.1** we can see a equivalent circuit which describe how a electro-conderser microphone works (**figure 2.2**), we see a of the mechanical waves to a variation of voltage in the resistor R, this is because the transducer conformed by a conductor, a dielectric material and a diaphragm which, when disturbed cause a variance in the capacitance of the transducer (the equivalence circuit of the transducer is a capacitor) this will cause a variation in the resistor R proportional of the capacitance variation C of the transducer (because Vdc and R are constant).

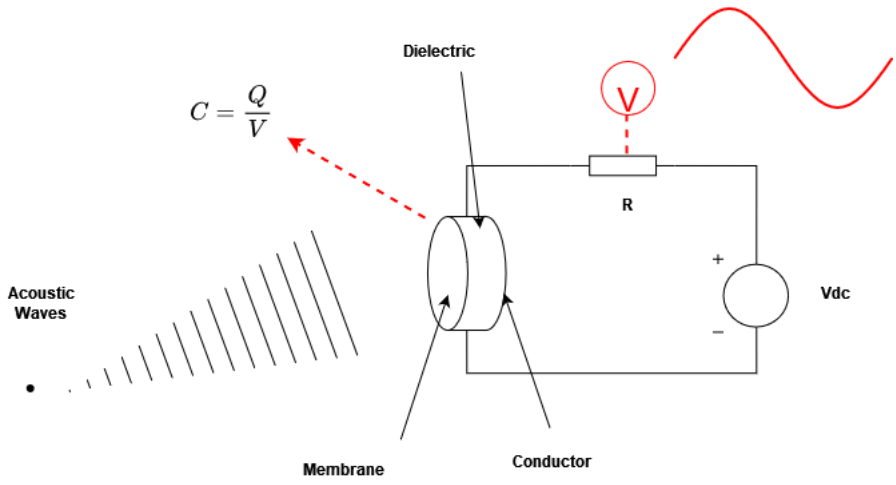

Figure 2.1: Principle of operation of the Microphone

This is the simpler equivalent circuit model for a microphone, its only involves the transducer part. It should be noted that the microphones also incorporate electronic for signal conditioning like a pre-amplification circuits stages.

Figure 2.2: Commercial *ECM* microphone

The microphones of interest in this project is the pdm microphone (**figure 2.3**) this type of microphone also incorporate additional electronics for signal conditional as well in this case, data converter circuit to give a digital output. as we discuss in the next section this type of microphones utilizes sigma-delta modulation that over samples the signal so they can have resolutions of 16 bits converters only using 1 bit (2 levels) of quantization.

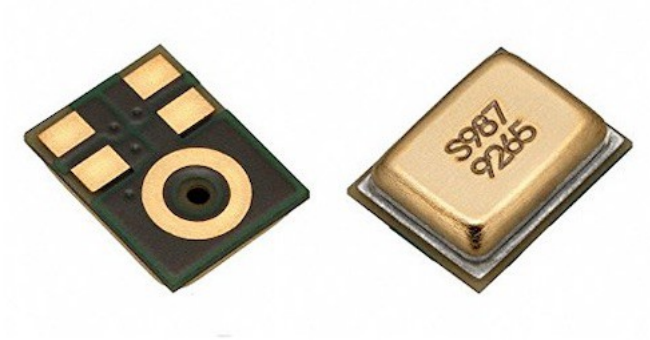

Figure 2.3: commercial *MEMS* microphones

2.2 Sigma-delta modulation

The explanation that we are going to do until this moment refer to the DAC design, which means the data conversion circuitry inside the mems microphones, what we can see in the figure **figure 2.4** is a block diagram of a first order sigma-delta converter from [7] this is the simpler version of this type of converter. the core idea of this converter is the fact that the signal is sampled several times more than the nyquist rate.

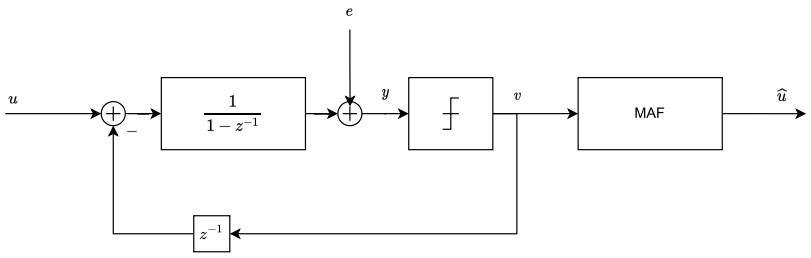

Figure 2.4: Block diagram of a 1st order ΣΔ converter [7]

The oversampling ratio or *OSR* refer to a factor that is a function of the nyquist rate, for example if we are using an OSR of 10 means that the signal is being sampled ten times faster than the nyquist sampling rate.

Lets check the math involves in the block diagram of the ΣΔ converter, first the signal u enters into the discrete integrator transfer function (which means that the signal is sampled in time but no quantified), y is the same signal being added the quantization noise, then after the signal is quantize v is negatively feedback (including a natural delay), the difference equation that describe this behavior is:

$$y[n] = y[n-1] + u[n-1] - v[n-1] \tag{2.1}$$

Where $v[n] = Q[y[n]]$ means that v is a product of the quantization operator Q in y.

Using $e = v - y$ and considering that $e[n]$ white noise sequence we have:

$$V(z) = z^{-1}U(z) + (1 - z^{-1})E(z) \tag{2.2}$$

in eq. 2.2 the TF that multiplies $U(z)$ is called signal transfer function or *STF* and and in the case of $E(z)$ is called noise transfer function or *NTF*. we will analyze the noise transfer function:

$$NTF(z) = 1 - z^{-1} \tag{2.3}$$

replacing $/z = e^{j\omega T}$

$$NTF(z) = 1 - e^{-j\omega T} = 2e^{-j\omega T}\left(\frac{e^{j\omega T} - e^{-j\omega T}}{2}\right) \tag{2.4}$$

$\Rightarrow$

$$NTF(z) = 2e^{\frac{-j\omega t}{2}} \cdot j \cdot sin(\omega T/2) = 2sin(\frac{\omega T}{2})e^{\frac{-j\omega-\pi}{2}} \tag{2.5}$$

$$NTF(z) = 2e^{\frac{-j\omega t}{2}} \cdot j \cdot sin(\omega T/2) = 2sin(\frac{\omega T}{2})e^{\frac{-j\omega-\pi}{2}} \tag{2.6}$$

$\Rightarrow$

$$|NTF(z)| = 2\,|sin(\pi fT)| = 2\left|sin(\pi\frac{f}{f_s})\right| \tag{2.7}$$

If we graphically analyze eq.2.7 we see in **figure 2.5** we can see that the noise is distributed in the entire band, meaning that great portion of the noise is outside of the band B of interest. Hence, with the NTF we apply a high pass filter or HPF to the noise outside the band B, this concept is called *Noise Shaping*. We will show the improvements calculating the P_{Qnoise} and $SQNR$.

We calculate the quantization noise power in the band of interest using the eq. 2.8 (where Δ is the quantization step). or in other words, we calculate the SQNR in function of the OSR like it has been passed through an ideal low pass filter (see **figure 2.5** b))

$$P_{Qnoise} = \int_0^{f_b} \frac{\Delta}{12} \cdot \frac{2}{f_s}\left[2sin\left(\pi\frac{f}{f_s}\right)\right]^2 df \tag{2.8}$$

Due to f_b is much less than f_s we can approximate $sin(\epsilon) \to \epsilon$, hence.

$$P_{Qnoise} \approx \int_0^{f_b} \frac{\Delta}{12} \cdot \frac{2}{f_s}\left[2\left(\pi\frac{f}{f_s}\right)\right]^2 df \tag{2.9}$$

$\to$

$$P_{Qnoise} \approx \frac{\Delta}{12} \cdot \frac{\pi^2}{3} \cdot \left[2\frac{f_b}{f_s}\right]^2 = \frac{\Delta}{12} \cdot \frac{\pi^2}{3} \cdot \frac{\pi^2}{OSR^3} \tag{2.10}$$

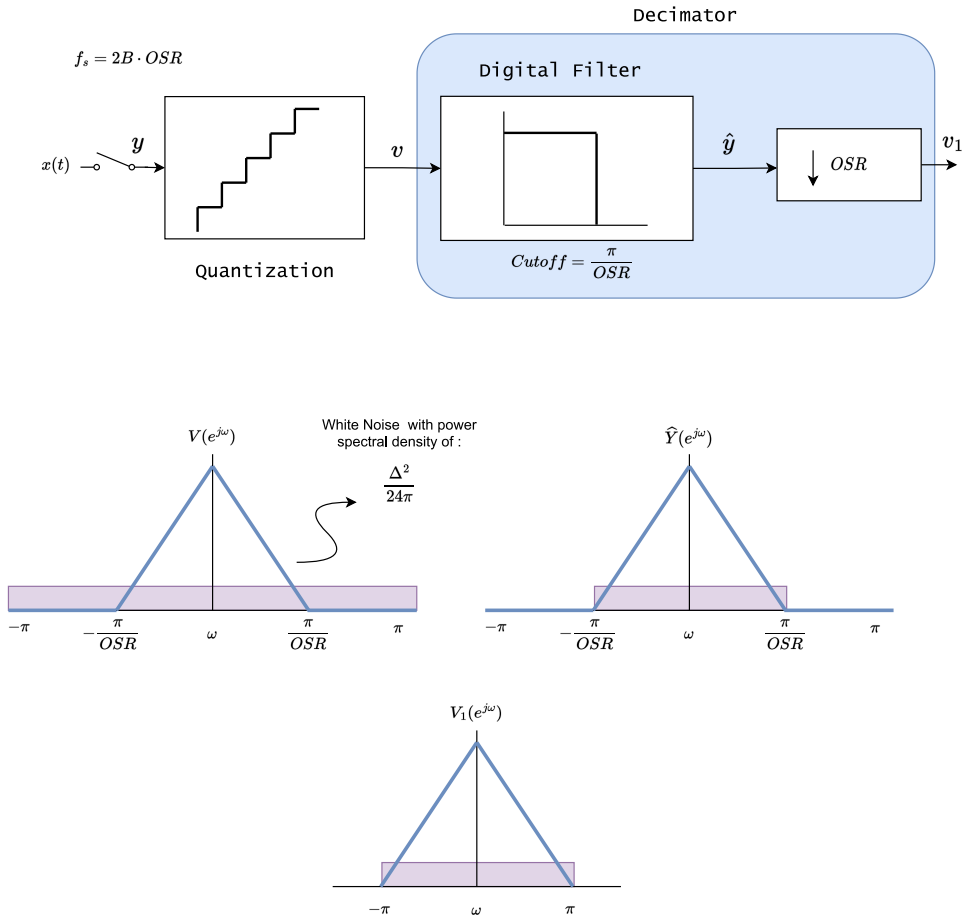

Figure 2.5: a) Block diagram of decimation stage b) power spectral density before and after noise shaping[7]

Now having P_{Qnoise} we obtain $SQNR$:

$$SNQR = \frac{P_{sig}}{P_{Qnoise}} = \frac{\frac{1}{2}\left(\frac{2^B-1}{2}\Delta\right)^2}{\frac{\Delta^2}{12} \cdot \frac{\pi}{3} \cdot \frac{1}{OSR^3}} \tag{2.11}$$

$$SNQR = 1.5(2^B - 1)\frac{3}{\pi^2}OSR^3 \tag{2.12}$$

Where B is the number of bits, we see that the OSR scaled cubed respect to the $SQNR$, if we see la eq. 2.12 in a logarithmic scale:

We conclude that more the oversampling ratio improves the SQNR and hence the bits of resolution. for example if we $2x$ the OSR in the same 1st order data converter we $8x$ the $SQNR$ or we improve 9 dB in an equivalent way. this produce an improvement of 1.5 Bits of resolution.

In the **figure 2.6** we can see a more accurate implementation of the same 1st order $\Delta\Sigma$ converter. we can say that we can still improve the OSR to get a better resolution of bits or implements a higher order of data converter as well as other architectures like cascaded single bits delta-sigma converters. For example the LC706 microphone from [13] implements a 4th order 1 bit $\Delta\Sigma$ converter with a PDM digital output.

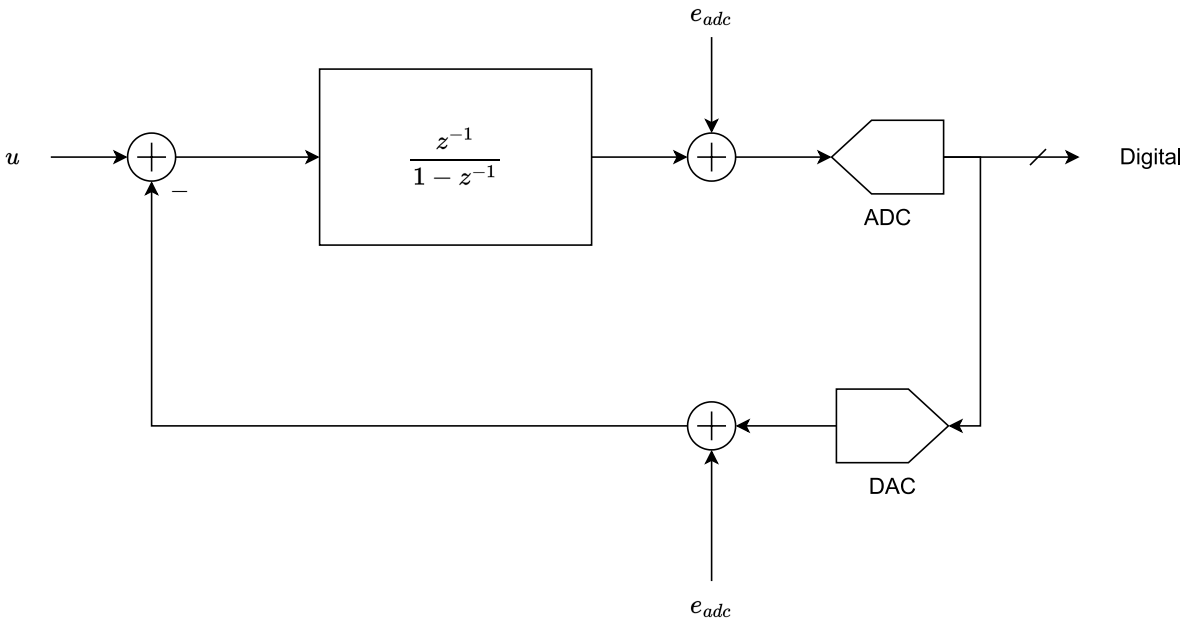

Figure 2.6: Block diagram 1st order $\Sigma\Delta$ converter with a cascaded ADC and DAC [7]

.

Keywords *– STF, NTF, HPF, LPF.*

3 A brief description of CIC filters

We already have shown the oversampling data converters must have a average window in digital signal processing part. this can be implemented in the form of a classical FIR type filter, but when also data conversions is needed we can implement other types of FIR filter appear. The cascaded combinational integrator filter or CIC is a very suitable filter for decimation or interpolation type of filters. that's is exactly what we are going to do, we have a heavily over-sampled signal (up to Megahertz) and our frequency of interest is in the low ultrasonic band (below 85 [Khz]) we can decimated de frequency maintaining the band of interest as well as do data conversion to pulse coded modulation or PCM format, gaining dynamic range.

But what are the benefits of using a CIC filter, this type of filters have almost the same response of MAF or RRS filters but does not use the same structure [14]. The case of this type of filter is that implement a integrator stage followed by a combinational stage for decimation case and also does not use any multiplication being a hardware efficient alternative [8] (**figure 3.1**)

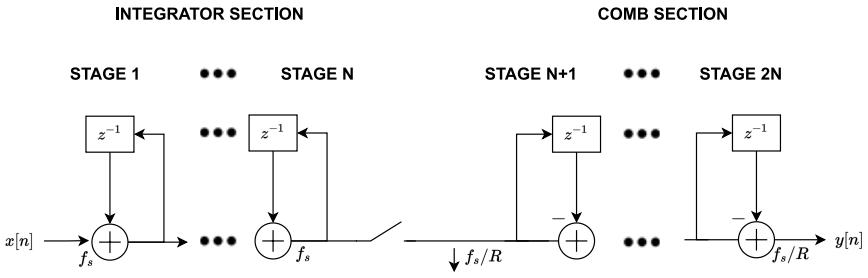

Figure 3.1: Block diagram of the *CIC* decimator filter [8]

As we can see, if we want to design an N order decimator filter we need $2N$ blocks, from 0 to N correspond to the integrator stages and from $N+1$ to $2N$ the decimation stages, we can deduce the TF from integrator H_I and combinational transfer function H_C as:

$$H_I = \frac{1}{1 - z^1} \tag{3.1}$$

$$H_C = 1 - z^{-RM} \tag{3.2}$$

The CIC transfer function ends being a convolution between those two:

$$H(z)_{CIC} = H_I^N H_C^N = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \left[\sum_{k=0}^{RM-1} z^{-k} \right]^N \tag{3.3}$$

Where R correspond to the decimation factor and M to the differential delay (usually $M = 1$ or $M = 2$).

In refer to the frequency response , it has a very similar response to MAF and RRS except for the modifications of M and R[1]. On the other hand, due to we assume that all this filters are LTI systems, we can say that the integrator-comb or comb-integrator block diagram has the same frequency response (eq. 3.3).

$$P(f) = \left[\frac{sin(\pi M f)}{sin(\pi M f / R)} \right]^{2N} \tag{3.4}$$

We will denote f_c as the post-decimation nyquist frequency, where we can see nulities in the multiples of $f = \frac{1}{M}$. Hence aliasing where appear in regions around those frequencies, specifically:

$$(i - f_c) \leq f \leq (i + f_c) \tag{3.5}$$

In the **figure 3.2** we see a CIC filter frequency response for different orders. In this case letting the differential delay $M = 1$ and $R = 10$ as the fixed decimation ratio because is the amount of decimation used in the implemented design of the project. As we can see there is a non-zero gain in dB, that is because is a pure integrator that have a increasing response, that also makes the filter a marginally stable system. We can also define the $Gain$ of the filter with the following equation [8]:

$$Gain = RM^N \tag{3.6}$$

Another important issue is define the amount of bits in the output of every stage, due integrator nature. The author [8] define a expression to obtain the bit width on the registers to avoid overflow:

$$B_{max} = [Nlog_2(RM) + B_{in} - 1] \tag{3.7}$$

[1]the terms $R = D$ corresponds to the decimation factor so when we use R or D we refer to the same term.

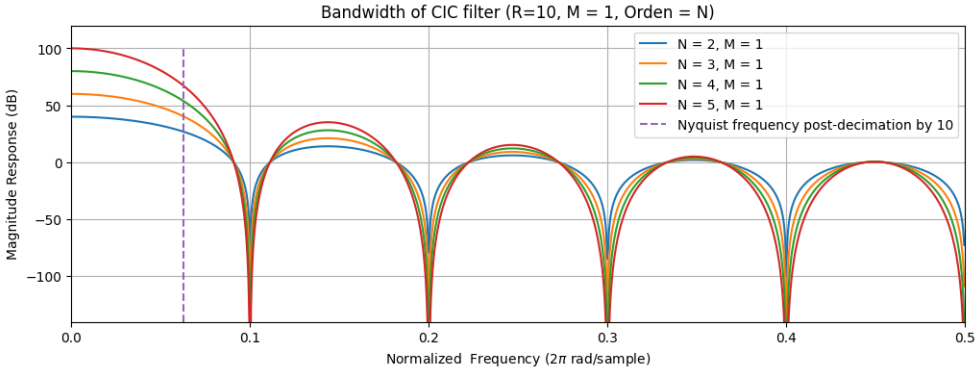

Figure 3.2: CIC filter with $R = 10$,$M = 1$ for different values of N

where B_{in} is the bit width for the filter input and and B_{max} is the necessary bit width for the registers. To close this chapter, due to the marginally stability of the filter, may cause overflow in every integrator of the filter, so two rules need to be follow to avoid this issue.

1. The filter should be implemented in a two complement arithmetic (or other numeric system that wraps around).

2. The dynamic range that the filter support must be equal or greater than the maximun magnitude value expected in the filter output.

3.1 Compensation filter

To summarize the previous chapter the CIC filter is an excellent choice for applications where we need a interpolator-decimator stage and also want to save hardware resources, no multipliers are required and also storing of filter coefficients, but we have aliasing or imaging in an specific intervals related to multiples of f. One solution to overcome this behavior is just apply a windows that his frequency response is inverse of the CIC filter response:

$$|H(j\omega)_{comp}| = RM_N \left| \frac{sin(\frac{\omega}{2R})}{sin(\frac{\omega M}{2})} \right|^N \tag{3.8}$$

We denote $H(z)_{comp}$ as the compensation filter (we can choose a multiple of two as the gain RM hence we can apply a simple bit shifting at the filter output) this windows have the quality of counteract the nullity's and also does not require as much hardware resources, for example using a low cut frequency bellow the post-decimation sampling frequency can reduce the number of filter coefficients to use (or filter order), using a classic FIR filter type. In the figure 3.3 we present a compensation filter design with $M = 2$, $N = 5$. we see that the frequency response (the normalized frequency is calculated respect to the post-decimation sampling frequency). we can see how the combined response generate a plane response in the desired band.

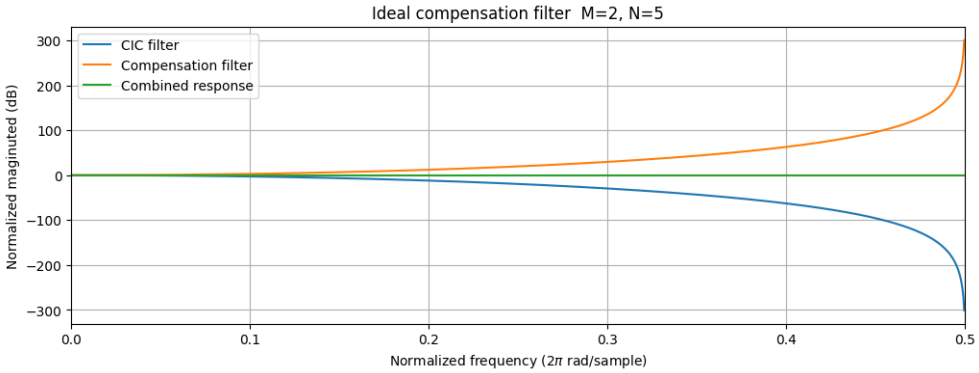

Figure 3.3: Compensation filter frequency response

Now, due to the desire band will be bellow the nyquist rate, we can make compensation filter with a cuttoff frequency of $0.1f_s$. (figure 3.4).

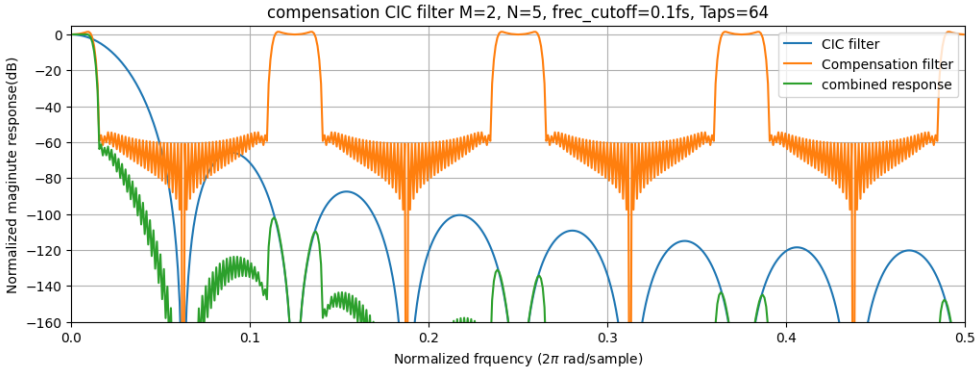

Figure 3.4: Compensation filter response,with a cutoff frequency of $0.1f_s$

It is necessary to mention that when we choose the fraction of f_s as a cutoff frequency we need to pay attention of the behavior of the filter. Seeing figure 3.5 we can say that when the cutoff frequency get more closer to f_s , specifically in the case of $0.4f_s$, the behavior of the response get worse in the nullity's. So a rule must be obey (or be aware of), the cutoff frequency of the compensation filter must be less than the quarter of the nyquist rate $f_c < \frac{1}{4}f_s$, from the perspective of the compensation filter design.

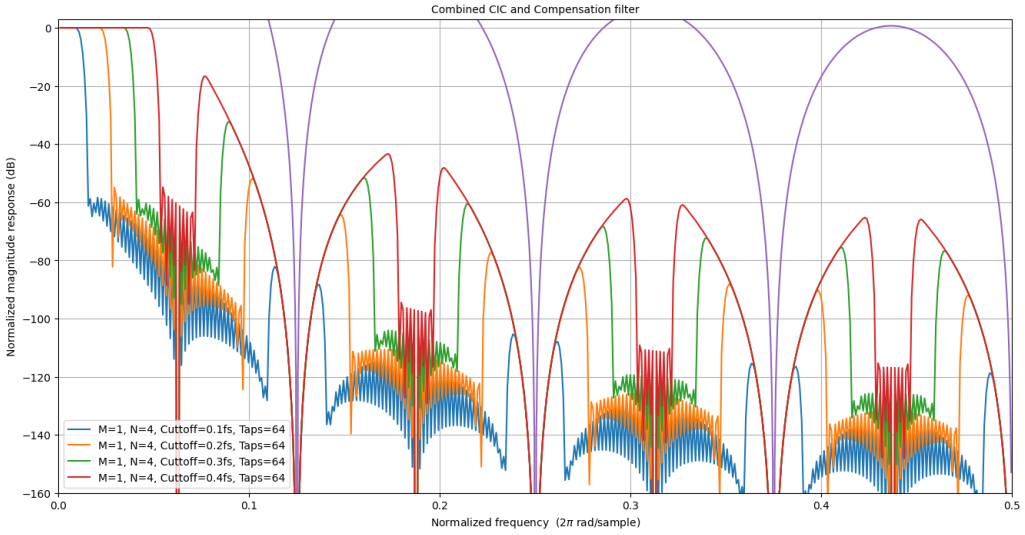

Figure 3.5: Compensation filter response,with a cutoff frequency of $0.1f_s$

Finally the filter design section is conclude, we show all the design criteria to understand the implementation stage. In this case further we will implement *CRSS* architecture based on [12] but has the same frequency response and behavior of CIC filter.

4 About FPGA based design

At the beginnings of the decade of 1980, designers used to draw digital circuits schematics on graphical computers. almost every digital logic design was build based on discrete components,one of common digital integrated circuits was the *74xx* family (figure 4.1).

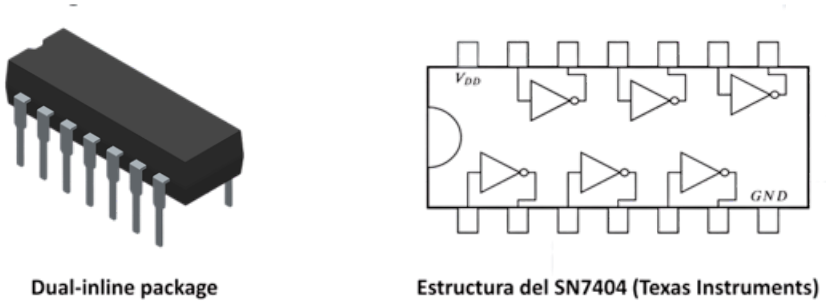

Figure 4.1: Physical package of ***IC*** SN7404 on the left, on the right the respective schematic

Suppose that we want to implement the Boolean function $Y = AB + \overline{B}C$ using discrete components like $SN7404$, $SN7408$ and $SN7432$ we can build that function (figure 4.2). Why do this exercise? even if we meet the requirements, there is not a regular structure of the design. If we want to elaborate more complex functions it must be of the work must be doing again (it also does not exist a reference design). This scenario is one of the reason that design migrate to the use of HDL or hardware description languages, in the late 80's this type of design paradigm gets even better do the incorporation of CAD tools and re-configurable hardware devices.

The re-configurable hardware devices appear at the ends of the 80's and are used up to the present are the standard way of design (as alternative to ASIC's). The FPGA means field programmable logic array, in this device a designer can implement a design using cad tools based on HDL design or schematic. the basic logic structure of this device is the *CLB* which stands for configurable logic block. As we can see in the (figure 4.3) is build a matrix structure, it also have several I/O around the logic elements (LE or CLB) so in that manner connects to the external world.

In the figure 4.4 we see an actual model of FPGA from one of the world leaders fabricators of this devices, the IC selected in red is the real device, the board is designed by a 3th party partner that implement fast prototyping

training boards. As we can see the in figure 4.5 the block diagram describe a modern SoC, in this case a Xilinx zynq7010 it includes not also the programmable logic elements enriched by other memory and dsp blocks but also a hard core processor. Due to the complexity of modern SoC we can build custom HW blocks that works with Software building complex embedded systems (in this case the communication protocol of this device used is Axi).

To implement a design in this device we use specialized cad tools like *Xilinx Vivado IDE*. viewing figure 4.6 we can see the fpga based design flow, it separated in different stages that are iterable. It begins by given a HDL or schematic design (not suitable for big designs). the logic synthesis stage translate the HDL code into a physical implementable logic gates (Mux, flip-flops, gates, etc.), then in the implementation stage that its divided in three stages, the first one is the translate which translates the synthesized design in a implementable technology for the target fpga, next the map and place and route stages proceeds to do a mapping of the blocks in specific parts of the target chip and route the connection between every block (several optimization are done at this stage). Finally bitstream generation maps the 0's and 1's that are involved in the switches that connects every logic element of the fpga. Several iteration are usually done in the flow due to different factors like, the hdl description are wrong, the design needs timing or area optimization that usually depends of the design specification requirements.

Until this moment we only describe the fpga and his design flow, but the main focus of this project is the verification of a HDL design that belongs to a bigger ASIC. So our finall goal is not generate the best performance or suitable fpga design. is only verify if the logic described in that filter is correctly done and in fact, works.

Keywords– *CAD, IC, HDL, PLA, CLBs, I/O, LE, AXI, IDE, Verilog, VHDL.*

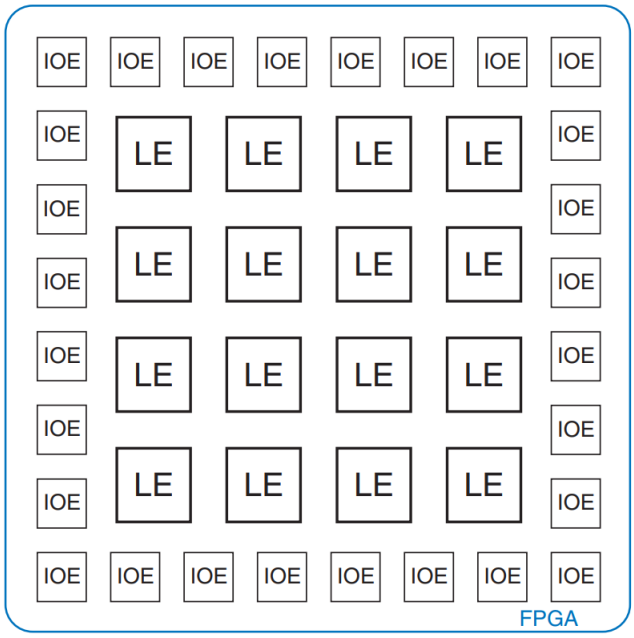

Figure 4.2: Generic *FPGA* structure [9]

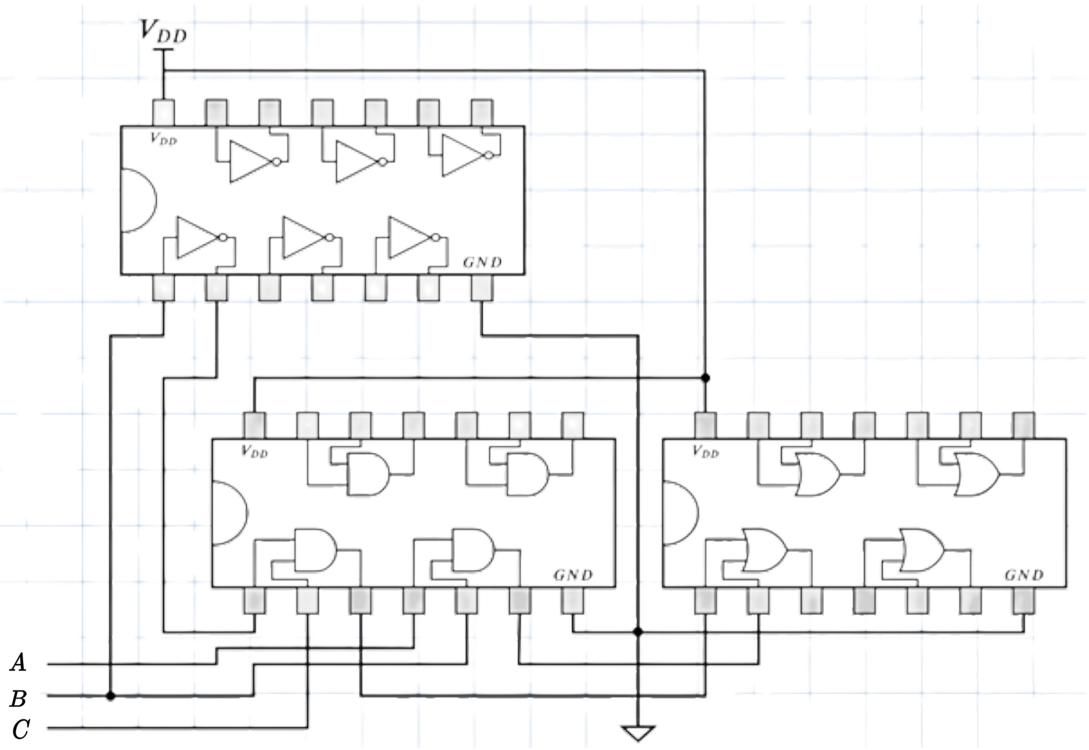

Figure 4.3: Implementation of $Y = AB + \overline{B}C$ using discrete components

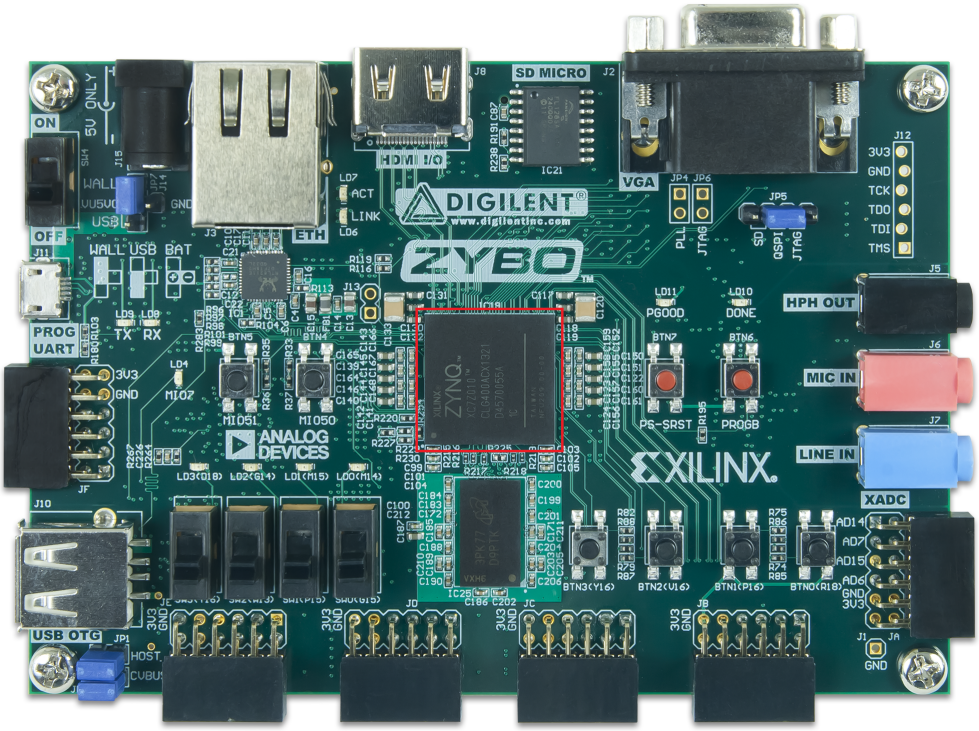

Figure 4.4: *Zybo7010* evaluation board by *Digilent Inc.*

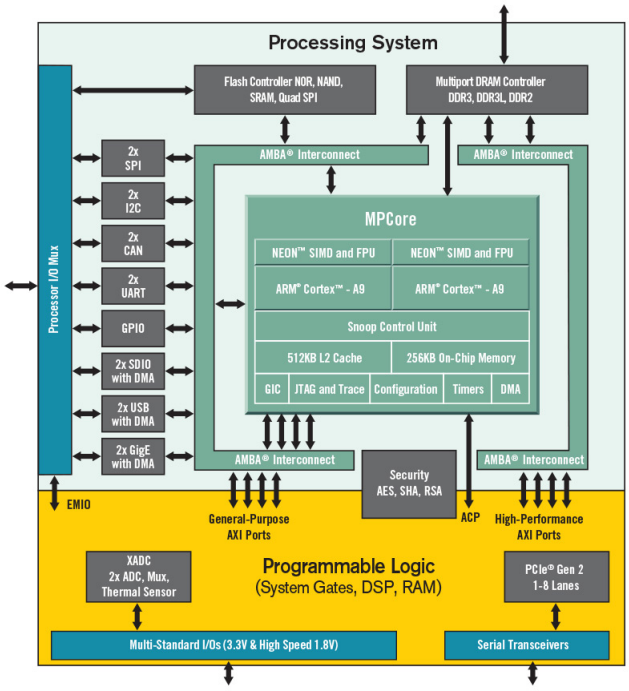

Figure 4.5: Architecture of *Zynq7000 SoC* by *Xilinx*

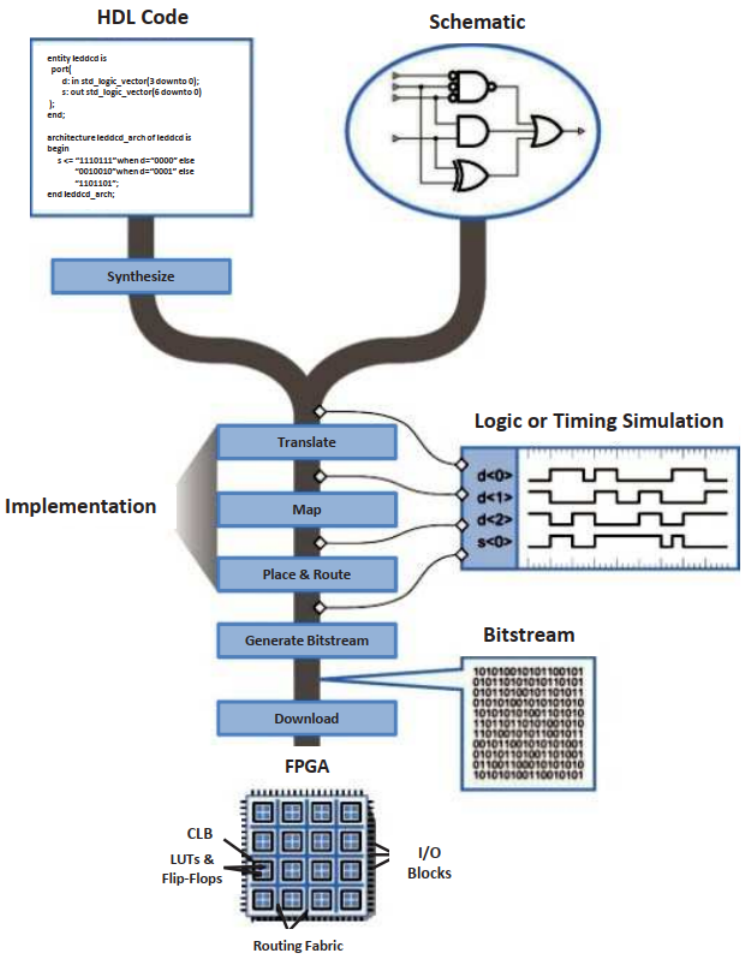

Figure 4.6: *FPGA's* based design flow [10]

4.1 Why use FPGA's for verification?

They are several reasons why an fpga implementation during the ASIC design flow is recommended, here are some of the reasons:

- Early Validation: FPGA-based prototypes allow designers to verify and validate the functionality of the design much earlier in the development cycle compared to waiting for the ASIC to be fabricated. This helps catch design issues and bugs early on, reducing overall development time and costs.
- Rapid Iterations: FPGA prototypes can be modified and reprogrammed relatively quickly compared to respinning an ASIC. This allows designers to make iterative changes and test different configurations to optimize performance and functionality.
- Reduced Risk: FPGA-based verification provides an opportunity to identify design flaws, missing features, or unexpected behaviors that might have been missed during the design phase. This reduces the risk of costly re-spins of the ASIC.
- Real-world Testing: FPGA prototypes allow testing of the design in a more real-world environment, including integration with other components or systems, handling real data or signals, and interfacing with external devices.
- Debugging: FPGA prototypes offer a platform for debugging and tracing issues that might arise during the verification process. Debugging is often easier in FPGA prototypes compared to ASICs, as FPGA tools offer powerful debugging features.
- Software Development: FPGA prototypes can be used for early software development and testing. This is especially important if the ASIC is part of a larger system that includes embedded software.
- Validation of Interfaces: FPGA-based prototypes can help validate the functionality of interfaces with other components or systems, ensuring proper communication and compatibility.
- Parallel Development: FPGA and ASIC designs often share similarities in terms of architecture and functionality. Developing and testing on an

FPGA can be done in parallel with the ASIC development, optimizing the overall development timeline.

- User Feedback: If the ASIC is intended for end-users, an FPGA prototype can provide a platform for user testing and feedback before the final ASIC is ready.
- Performance Tuning: FPGA prototypes allow designers to evaluate and optimize the performance of the design. This includes evaluating timing constraints, clock frequencies, and power consumption.
- Trade-off Analysis: FPGA prototypes can help with evaluating different design trade-offs and making informed decisions about design choices.

Overall, FPGA-based verification offers a cost-effective and efficient way to validate and test designs before committing to the expensive and time-consuming ASIC fabrication process. It's an essential step in reducing risks, improving functionality, and ensuring the success of the final ASIC.

5 Development of the project

Now that the theory has been presented, we show the digital front-end of the "Sonar On Chip", were going to describe the parameters of the decoder circuit as well as the input stimulus. Then do the segmentation between the hardware and software implementation involved in the verification process.

5.1 Design Implementation

As we already know this project "The project implements a digital system for signal processing to capture and process acoustics signals from 36 MEMS microphones with an extended frequency range up to 85 kHz (low ultrasonic band). The system itself is a part of the Caravel harness and can be configured and managed from the Caravel using Wishbone bus. The principle of operation of the system is the following: Each, pulse density modulated (PDM), the microphone signal is processed individually using a separate channel, which demodulates PDM data recovering PCM (Pulse Code Modulation) samples, filters out audible frequencies, detects the envelope, and compare its value to a configurable threshold."[1]. As we can see in figure 5.1 focusing only in the pdm demodulator stage, we segment this block in 3 main sub-blocks, first the pdm-pcm decoder circuit implementing the CRRS architecture, a decimation stage and finally a post processing filter that works as the compensation filter.

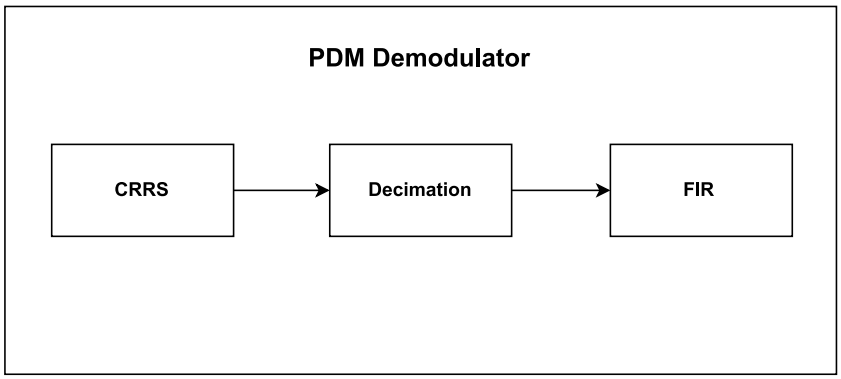

Figure 5.1: PDM decoder block diagram [1]

.

The design is made implementing two RRS stages with an 12 bits based

on (eq. 5.1) final output, giving a total dynamic range of 72 dB, the amount of dynamic range is greater than the required (the target mems microphone have a dynamic range of 62 dB). The expression used are based on [12] where the authors shown that this kind of structure reduce the number of bits at the output and the memory requirements.

$$B_{out} = [J \log_2 L - (J-1) \log_2 D] \tag{5.1}$$

Viewing the eq. 5.1 J stands for the number of cascaded RRS stages, L the amount of delay or filter length and D the decimation factor (ideally choosing $L = 0SR$). In figure 5.2 we see the block diagram of the Target CRRS structure.

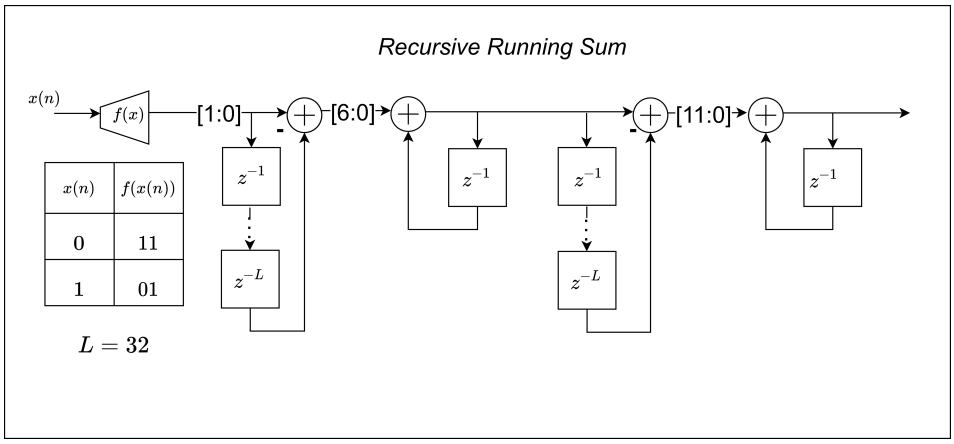

Figure 5.2: Block diagram of the implemented $CRRS$ filter [1]

In figure 5.3 we can see the block diagram of the system to verify, in this case only the CRRS block is the verification target, but also the clock generations are part of the ASIC design, so as we see further in a behavioral level the CRRS,MIC/CRRS Clock's and PDM clock blocks are implemented in a top level test-bench implemented in verilog usign *Vivado 2018.3 Xsim* with also use as an input a read only memory that sends pdm samples generated at script level, these samples are stored in a .mem file an use to initialized the read only memory that use *mclk* signal as a guide to send these samples every clock edge. The figure 5.4 shows the expected behavior of the clock signals where *mclk* is the signal used from de microphones clock, *ce_pdm* the clock signal for the CRRS filter an finally a *ce_pcm* signal that acts as the decimated output.

A summary of the clocks behaviors for the require design are:

- a 4.8 MHz clock signal mclk for the external microphones (50 % duty cycle),
- a 4.8 MHz clock enable signal ce_pdm for PDM demodulator
- a 480 kHz (configurable) clock enable signal ce_pcm for the PCM datapath.

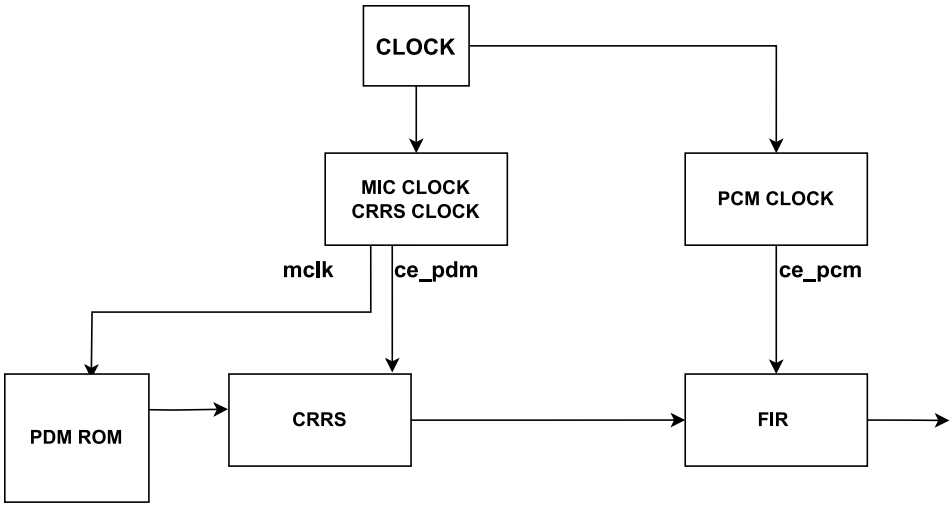

Figure 5.3: Block diagram of the system

.

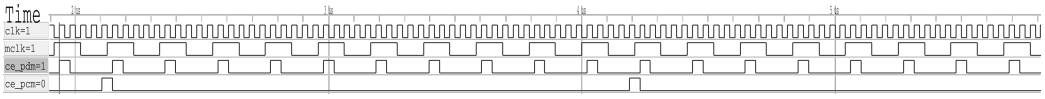

Figure 5.4: Clocks behavior [1]

5.2 Behavioral simulation test

In the behavioral simulation results a top module is generated in verilog including all the components of the system previously exposes, in this case a decimation stage is implemented at hardware level also. In figure 5.5 we can

see a block diagram describing how is instantiated each block of the system and also the internal connections between them.

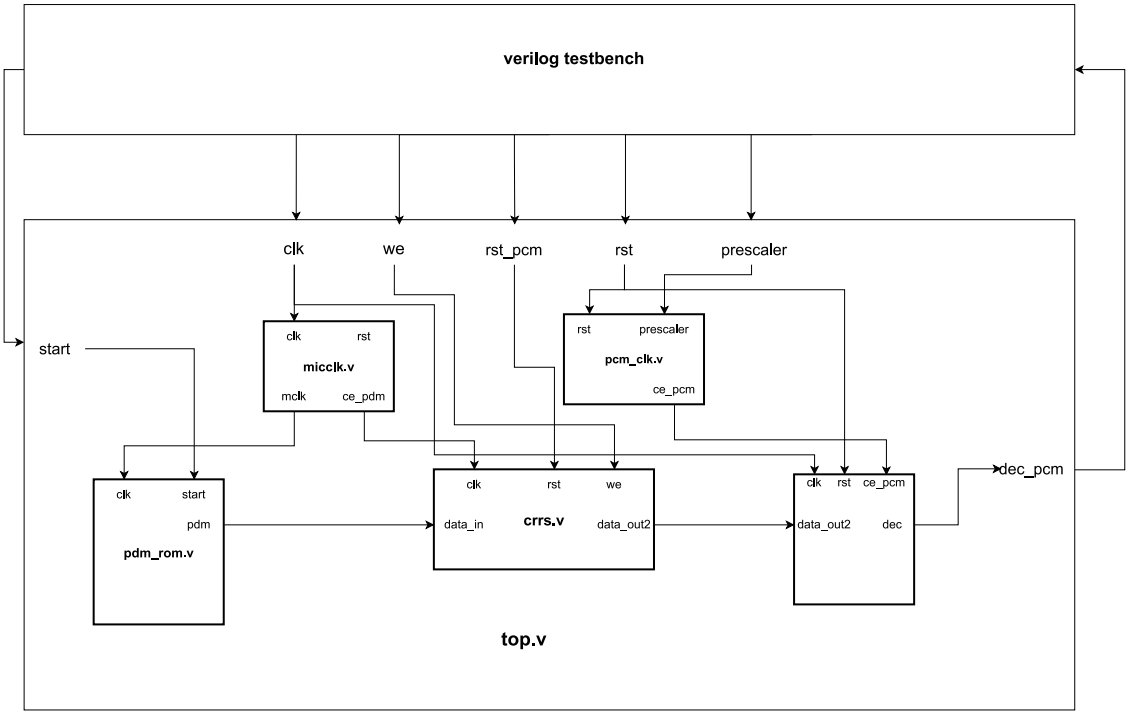

Figure 5.5: Behavioral simulation block diagram

.

In the table 1 we can see the inputs/outputs of the top block including the bit width, in this case the prescaler input desire decimation ratio, in this case we use $prescaler = 20$ due to the clocks of pdm rom and CRRS block are 5 times slower than the main clock, in that way with 20 clock cycles we have an exact decimation of 10.

The generated clock period is 42 nanoseconds, this period is not exactly the amount of 24 [Mhz] of clock frequency that we are looking to achieve, the exact amount is 23.80 [Mhz], as we can see in the following results a slight frequency shifting will appear. Lets check the figure 5.6 that correspond of the crude pdm modulation with of a 40 [khz] sine wave with a dc value of 1 and a amplitude off 0.5 is generated with an oversample frequency of 4.8 [Mhz] (this samples are saved in the .mem file containing in the pdm rom memory). In figure 5.7 we can see the plots of pdm rom signals ans well the

input/output	bit width
clk	[0:0]
rst	[0:0]
rst_pcm	[0:0]
start	[0:0]
we	[0:0]
prescaler	[9:0]
dec_pcm	[31:0]

Table 1: top.v input/output description

results of the behavioral simulation (the signals are normalized for plotting purpose).

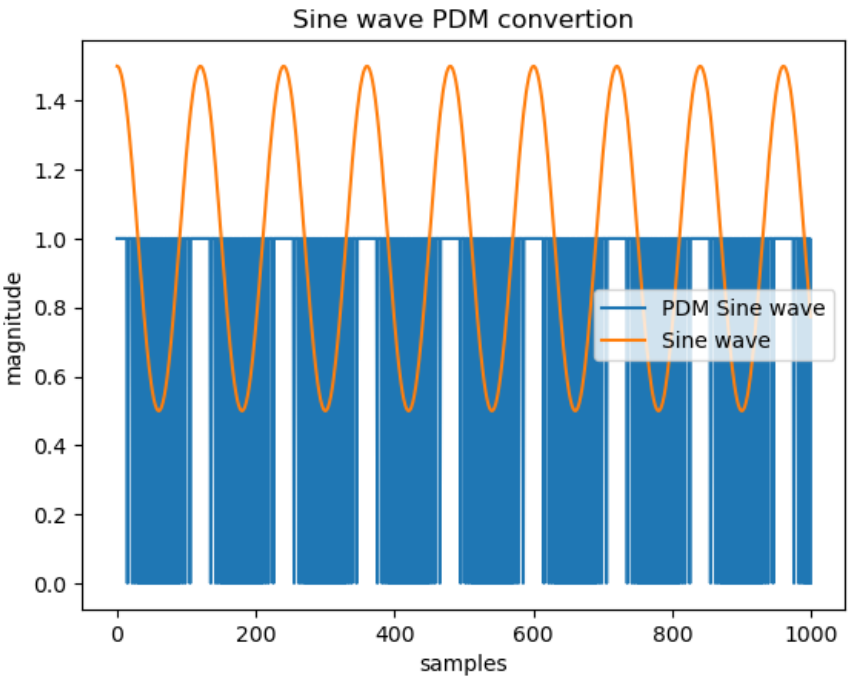

Figure 5.6: PDM sine wave generation

Based on the explanation of CIC filters and compensation filters showed in chapter 4 a compensation filtering is applied, in figure 5.8 we see several plots of the original signal, fft of the post simulation signal as well as the

Original signal freq.	Post-simulation signal freq.	$\Delta freq.$ %	$Gain$
40 [khz]	40.336 [khz]	0.83%	$6.14Gain$

Table 2: post-simulation results

graph of the filtered signal. table 2 we can see the variation of frequency and real gain,the frequency variation is obtaining by taking the bigger harmonic component of the fft once the post filtering is done. We can also see that is 6.14 times the theoretical gain of $Gain = (MR)^N$.

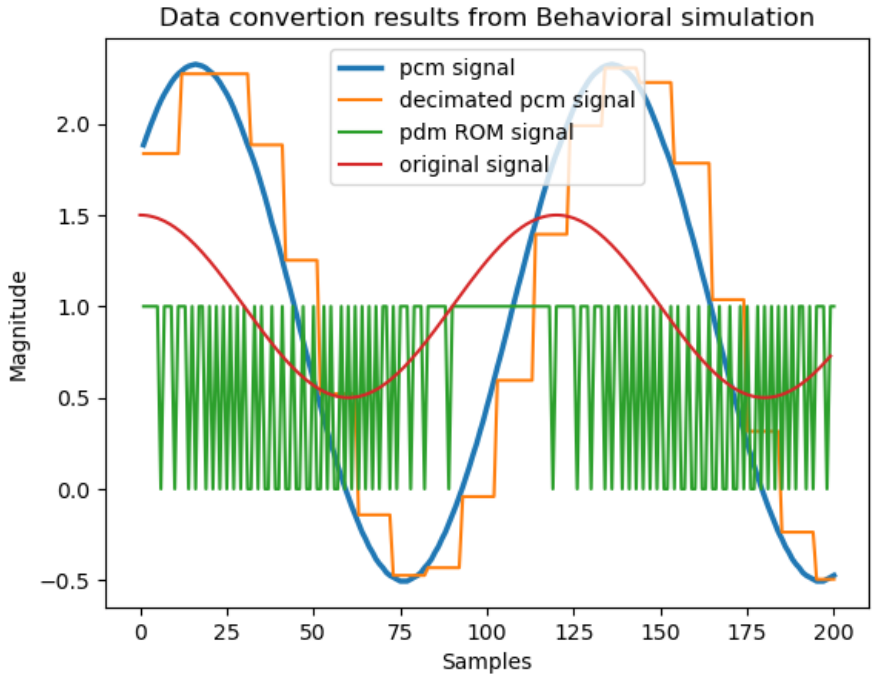

Figure 5.7: Post-simulation signal plotting

All these signals are obtaining via $monitor verilog system task, then save the data in a .txt formats for posterior processing in python, in this case the results are pretty accurate because we are talking of behavioral simulation with ideal clock signals and accurate data acquisition, as we can see further in the post-implementation stage, contamination in the signal due to several factors appear and some extra processing is needed.

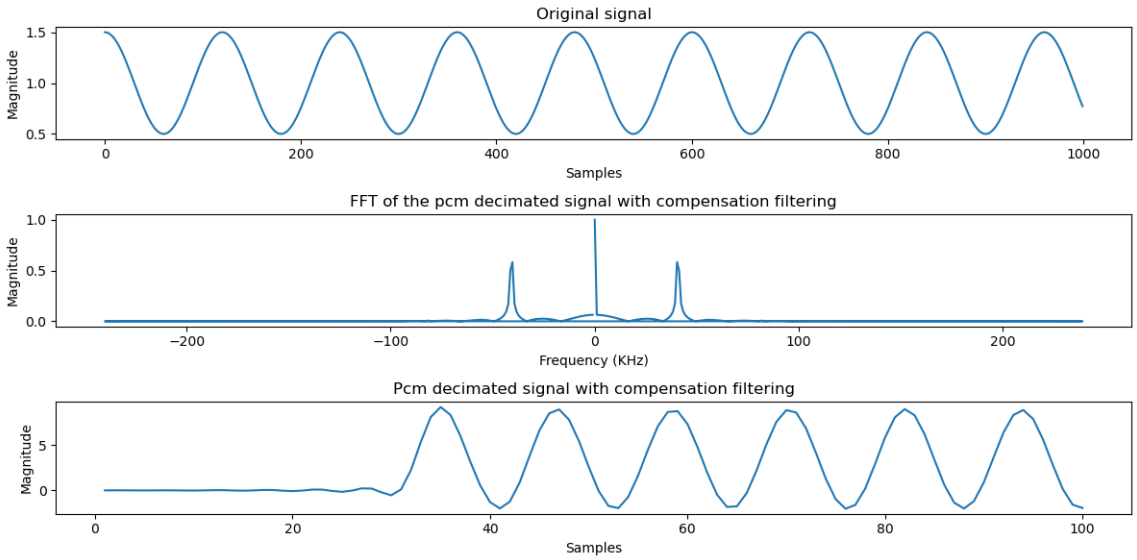

Figure 5.8: post-simulation and signal processing results

5.3 Post-implementation test

The post implementation process involves in this case the generation of a custom ip core in the Vivado 2018.3 environment, in this case a custom AXI lite ip core is generated with the same top.v file. AXI stands for advanced Advanced eXtensible Interface and is a bus communication protocol widely used and adopted by xilinx Zynq7000 series device. The advantage to use this protocol is the rapid prototyping a testing of the custom modules, the user can take a build template of the axi-lite and instantiate internally the custom hdl files, then use the Block Design environment of Vivado to instantiate the zynq700 interface and custom ip. Vivado itself incorporate all the intermediate blocks to communicate the custom ip core with the hard core processor, so in that way the user can utilize the build-in drivers to easily communicate with the custom ip. In this case, the figure 5.9 shows the complete block diagram of the system, in red our custom ip block (the top.v file with all the blocks except the decimation stage), in green the zynq7000 processing system and in orange all the blocks or infrastructure automaticly generated by vivado to established communication between HW and SW.

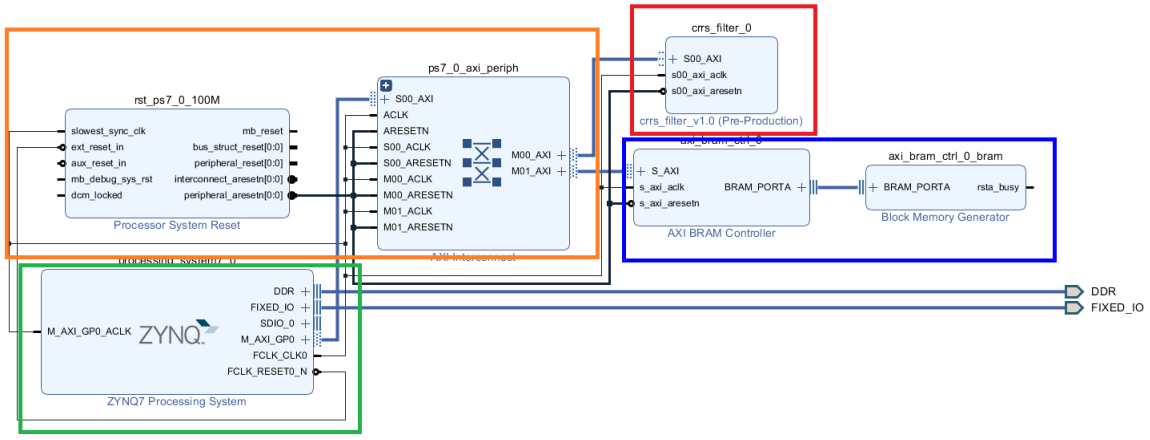

Figure 5.9: Block diagram of the implemented system

Where are the inputs and outputs of the ip? in the axi-lite custom ip there exist an memory mapped register that are configurable. this memory

Ip	Base address	High address
CRRS_FILTER	0x43C00000	0x43C0FFFF
AXI_BRAM_CTRL_0	0x40000000U	0x40001FFFU
PS7_DDR_0	0x00100000	0x1FFFFFFF

Table 3: Memory mapped registers

CRRS_FILTER address	data content [31:0]
0x43C00000	0000_0000_0000_we,start,rst_pcm,rst
0x43C00004	pcm_data

Table 4: Control register for the custom ip

mapped register can be accessed (or in other words, read and write into them) via c code, the main idea in the generation of all this interconnection blocks between hard cpu and re configurable hardware is that we can use pointers with the exact register memory address. In table 1 we can see the exact mapping of the top.v inputs and outputs with his exact memory address. As we also can see in the same block diagram is a blue colored blocks, in this case is an AXI block ram memory of 8kb, the idea stands that while reading the output samples of the custom ip, we copy a stream of 8000 thousand samples (almost all the block ram memory) into the block ram and then to a hard memory like the ddr memory that the hardcore processor offer.Again, we can easily access into the block ram via the memory mapped registers.

The Table 3 shows the address range of the custom ip as well as the memory resources, we can write value into the first register 0x43C00000 that has a 32 bit width data. In that way if we want to write or read the next address we jump at 32 bits values (byte addressable) in Table 4 we can see the two memory registers that are used to manage the custom filter ip, these are the control register. For example, in the first memory address we have the first 4 bits or 1 hex digit the inputs of the custom ip, then in the next address the output which is in a 32 bits data width.In this implementation, the decimation stage is made at scripting level, so a prescaler signal is not present. Finally in a few lines of c code using the hard processor driver libraries, we can exchange data between the custom ip and the processing system.

Now that we are going to see the results of the post implementation stage, is important to mention that to achieve the transfer of successive samples. Due to the incorporate hard core processor of the board (zybo with the

zynq7000) has a 650 Mhz clock and our custom ip works at a lower clock of 24 Mhz for the principal clock and 4.8 Mhz clocks generated via the custom hardware modules already present, we gets use of the AXI-lite transaction principle [15]. Is already knew that using this template we have all the signals that allows a successfully transfer of data. In order to not describe this entire process were going to mention only one signal of the protocol the *ARREADY* signal, this signal basically stands that a in a transfer between the master (in this case the processing system) and the server (the custom ip) if the *ARREADY* signal is asserted means that the from the server perspective, he is ready to accept the reading request of his data. In other words, the custom ip filter is ready to send a new data to the processing system. We use activation of this signal in the firmware development for the filter test, due to the significantly faster clock speed of the cpu vs the custom ip, while the cpu has always work faster at executing the reading request, it has to wait until a new data value appear from the custom filter, in that way we can assure the correct arrival of new data and not be excessively over reading the same data over and over. Using this approach we can generate a subset of data samples coming from the post-implemented design using the Vivado sdk 2018.3 platform reading the serial terminal via the processing system UART. the c code of the firmware development for the test is in the appendix B and it is self explanatory. So, in that manner we can save a post-implementation samples via .txt file and do the processing of the data via python scripts.

The figure 5.10 show a comparison between the original signal and the signal with post-decimation (and also with compensation filter), we see are some high frequency components that are not indicating a correct post-processing. To get a better information we apply a band-pass fir filter with python using the scipy library, the filter has a low cutoff frequency of 30 khz and a high cutoff freq. of 80 khz. In that way we erase the dc component and also isolate in a more accurate way the 40 khz sine wave. The figure 5.11 shows now the exact comparison as before, we see clearly the main frequency components slightly shifted again, and also with more gain that the theoretical expected value. Table 5 shows the different changes between the ideal and the implemented results.

As we can see in the Table 5 we see the different between the ideal signal and the post-implemented signal. we can see that there is a greater delta in the frequency of the signal. Also in the previous figure the fft shows a increasing amount of harmonics around the fundamental harmonic of the sine wave. in the case of the *gain* of the systems holds the same amount

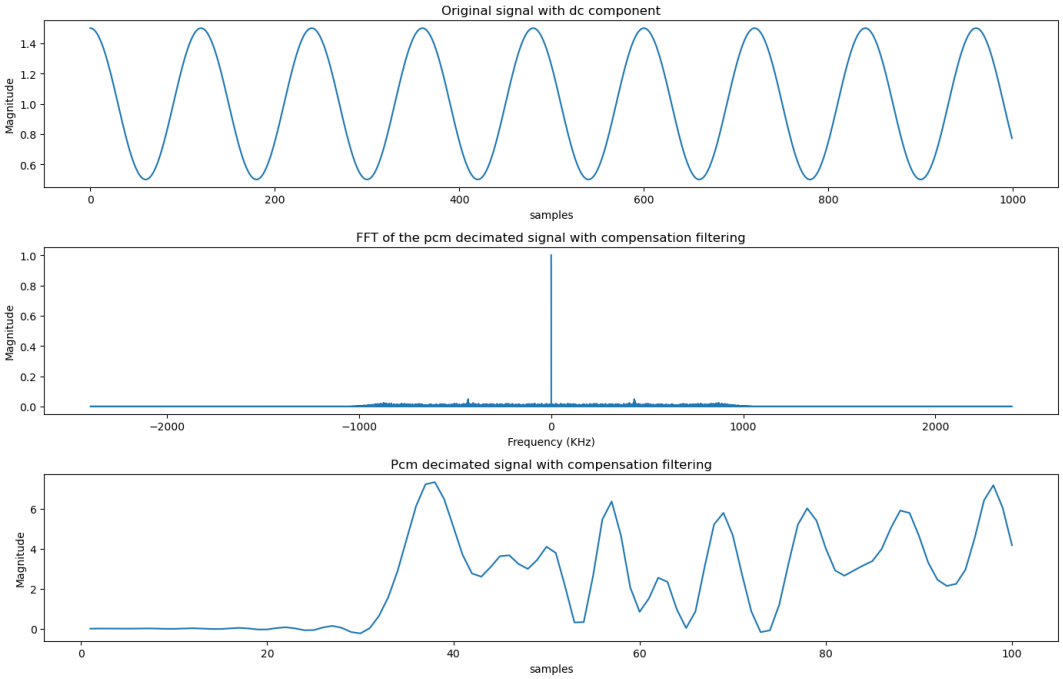

Figure 5.10: Frequency analysis after compensation filtering

Original signal freq.	Post-simulation signal freq.	$\Delta freq.\ \%$	$Gain$
40 [khz]	43.247 [khz]	7.51%	$6.14Gain$

Table 5: post-implementation results

of increase of $6.14Gain$. A good think to mention and that we will explain further in the conclusions of this project is that again the clock generation, in this case from the PLL of the processing system is 23.80 Mhz (not 24 Mhz as the target clock frequency) due to the post processing (at scripting level) is made with closed number like 4.8 Mhz or 480 khz for the filtering and fft calculation it also affects the results.

The project that involves the behavioral testing is named model_test2 and the post-implementation testing project is called model_test5. both of the files are Vivado project that are available in the link presented in the Appendix B as well as the jupyter notebook with the post-processing.

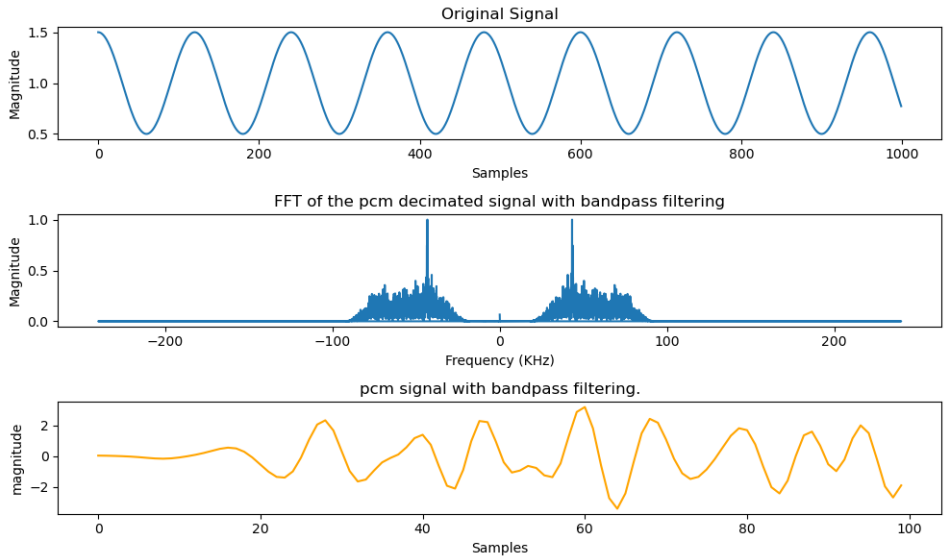

Figure 5.11: Frequency analysis after band pass filtering

6 Conclusions and future work

Based on the [1] project, it is possible to test the CRRS filter in a *FPGA*, verify his functionality by generate a transfer from rom memory to custom filter to a hard memory in the processing system. We also see the frequency shifting in behavioral and post-implemented design. one of the reason that this may occur is the no exact clock definitions, as we see during the elaboration of the testing the clock variations could affect the calculus of the compensation filtering and also the fft cause both of the processing techniques was build with an ideal sampling frequency (what in fact, does not happen). This frequency shifting behavior are intentional in some DSP application but in this specific kind of uses this behavior has to be mitigate. Even if we had a working system we also see the apparition of several harmonics around the fundamental harmonic of the sine wave used to the test, one of the other effects that happens when we do some frequency analysis is the Gibbs effect. Due to the fact that we have at most 4 point between every clock period of the signal (one of the causes of decimation) this "step like" behavior can propitiate the Gibbs effect, get more samples in every period and design a more accurate filtering process mitigate this effect.

Based on the [1] project, it is possible to test the CRRS filter in a *FPGA*, verify his functionality by generate a transfer from rom memory to custom filter to a hard memory in the processing system. We also see the frequency shifting in behavioral and post-implemented design. one of the reason that this may occur is the no exact clock definitions, as we see during the elaboration of the testing the clock variations could affect the calculus of the compensation filtering and also the fft cause both of the processing techniques was build with an ideal sampling frequency (what in fact, does not happen). This frequency shifting behavior are intentional in some DSP application but in this specific kind of uses this behavior has to be mitigate. Even if we had a working system we also see the apparition of several harmonics around the fundamental harmonic of the sine wave used to the test, one of the other effects that happens when we do some frequency analysis is the Gibbs effect. Due to the fact that we have at most 4 point between every clock period of the signal (one of the causes of decimation) this "step like" behavior can propitiate the Gibbs effect, get more samples in every period and design a more accurate filtering process mitigate this effect.

6.1 future work

Abut the future works that this project opens, we can separate in mainly two main works, all of them involves the use of the already tested and implemented blocks from [1]:

- In the first main work, testing the same project with a custom pcb board with a low-ultrasonic MEMS microphones, in that way we have real signals to the test this filter. In other hand, keep implementing the subsequent stages and eventually the complete data-path of [1].
- In the second main work, with already probed pdm decoder circuit, use this project as the front-end of a sounds localization system, implementing the filtering stages and a spatial filtering algorithm like beamforming. In this case is suitable to have the array of microphones from the first main project.

There's also a lot of work not only in the hardware side, some embedded software development can be done to improve the functionality of the system as well. In other hand, post-silicon validation of the incoming Sonar On Chip needs to be done.

Appendix A: Verilog HDL design

```
1  // Code your design here
2  // ------------------------------------------------- //
3  //                     DEMODULATOR FILTER RSS
4  //         Mauricio Montanares, Luis Osses, Max Cerda 2021
5  // ------------------------------------------------- //
6
7  `define OverSample 32
8
9  // =========== Top module START =========== //
10 module cic #(parameter N =- 2)(
11   input clk, rst, we,
12   input                        data_in,
13   output wire signed [11:0] data_out2
14 );
15   integer i;
16   wire signed [6:0] sum1;
17   wire signed [1:0] data_1_in;
18   reg         [1:0] ff1[`OverSample];
19   wire        [1:0] ff1_last;
20   reg         [6:0] ff1out;
21   wire        [6:0] dinext1;
22   wire        [6:0] ffext1;
23
24
25   wire signed [11:0] sum2;
26   reg         [6:0] ff2[`OverSample];
27   wire        [6:0] ff2_last;
28   reg         [11:0] ff2out;
29   wire        [11:0] dinext2;
30   wire        [11:0] ffext2;
31
32
33    /* input 2'complement extension */
34    assign data_1_in = (data_in == 1'b0) ? 2'b11 : 2'b01;
35    assign dinext1 =  {{6{data_1_in[1]}},data_1_in};
```

```
36      assign ff1_last = ff1[`OverSample-1];
37      assign ffext1 =  {{5{ff1_last[1]}}, ff1_last };
38      assign sum1 = dinext1-ffext1;
39
40
41      assign dinext2 =  {{5{ff1out[6]}},ff1out};
42      assign ff2_last =ff2[`OverSample-1];
43      assign ffext2 =  {{5{ff2_last[6]}}, ff2_last };
44      assign sum2 = dinext2 - ffext2;
45      assign data_out2 = ff2out;
46
47      always @(posedge clk) begin
48         if (rst) begin
49            ff1out <=0;
50            ff2out <=0;
51                    for(i = 0; i<`OverSample; i=i+1 ) begin
52                        ff1[i] <= 0;
53                        ff2[i] <= 0;
54                    end
55         end
56         else if (we) begin
57               ff1[0] <= data_1_in;
58                   ff2[0] <= ff1out;
59                   for(i = 1; i<`OverSample; i=i+1 ) begin
60                           ff1[i] <= ff1[i-1];
61               ff2[i] <= ff2[i-1];
62           end
63           /*  integrator */
64           ff1out <= ff1out + sum1;
65           ff2out <= ff2out + sum2;
66         end
67
68      end
69   endmodule
70
71
72
```

```
73
74
75  /*  ---------------------- CLOCK DIVIDERS ------------------------- */
76  /*
77  with 8 MHz cristal
78  PLL feedback divider = 18
79  18 *8 = 144
80  main clock
81  PLL 1 divider  = 6
82  144/6 = 24
83  second clock
84  144/3 = 48
85  */
86
87  module micclk(clk, rst, mclk, ce_pdm);
88    input clk, rst;
89    output mclk;
90    output ce_pdm;
91
92    reg  tmp1, tmp2;
93
94    assign mclk = ~(tmp1 | tmp2);
95    reg [3:0] cnt1, cnt2;
96
97  /* delay one cycle vs. MIC Clk*/
98    assign ce_pdm = cnt1==4'b001 ? 1 : 0;
99
100   always @(posedge clk)
101     begin
102       if (rst) begin
103             cnt1 <= 0;
104             tmp1 <= 0;
105       end
106       else
107         cnt1 <= cnt1 + 1;
108       if (cnt1 >=  2)
109             tmp1 <=  1'b1;
```

```
110       if (cnt1 == 4) begin
111           cnt1 <= 0;
112           tmp1 <= 0;
113       end
114     end
115 
116   always @(negedge clk)
117     begin
118       if (rst) begin
119             cnt2 <= 0;
120             tmp2 <= 0;
121       end
122       else
123         cnt2 <= cnt2 + 1;
124       if (cnt2 >=  2)
125             tmp2 <=  1'b1;
126       if (cnt2 == 4) begin
127           cnt2 <= 0;
128           tmp2 <= 0;
129       end
130     end
131 endmodule
132 
133 
134 module pcm_clk(clk, rst, prescaler, ce_pcm);
135 
136   input clk, rst;
137   input [9:0] prescaler;
138   output reg ce_pcm;
139 
140   reg [9:0] count;
141 
142   always@(posedge clk) begin
143     if(rst)begin
144       ce_pcm <=0;
145       count <= 0;
146     end
```

```
147     else if(count == prescaler) begin
148         ce_pcm <= 1;
149         count <=0;
150     end
151     else begin
152             count <= count + 1;
153         ce_pcm <= 0;
154     end
155 
156   end
157 
158 endmodule
159 
160 
```

The RTL description of the entire project in *Sonar On Chip* :`https://github.com/HALxmont/SonarOnChip8/tree/main/verilog/rtl`

Appendix B: C code for firmware development

```c
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xil_io.h"
#include "stdio.h"
#include "xtime_l.h"

//define adresses
#define BASE_ADDR XPAR_CRRS_FILTER_0_S00_AXI_BASEADDR
#define BRAM_BASE_ADDR XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR
#define DDR_BASE_ADDR XPAR_PS7_DDR_0_S_AXI_BASEADDR


//function to generate nanosleep wih cpu timers.
void nanosleep_delay(unsigned long nanoseconds) {
    XTime start, end;
    unsigned long cycles;

    // Convert nanoseconds to cycles
    cycles = (unsigned long)((double)XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ * nanoseconds / 1e9);

    // Get the start time
    XTime_GetTime(&start);

    // Wait until the specified number of cycles has passed
    do {
        XTime_GetTime(&end);
    } while ((end - start) < cycles);
}



int main()
{
    int num_samples = 8000;  //number of samples to transfer.
    int i;

    volatile int *bram_ptr = (volatile int *)BRAM_BASE_ADDR;
    volatile int *ddr_ptr = (volatile int *)DDR_BASE_ADDR;

    print("we = 0 y rst = 1,rst_pcm = 1, start = 0 \n\r");
    Xil_Out32(XPAR_CRRS_FILTER_0_S00_AXI_BASEADDR,0x00000003);
    nanosleep_delay(210);
```

```
    print("we = 0 y rst = 0,rst_pcm = 1, start = 1 \n\r");
    Xil_Out32(XPAR_CRRS_FILTER_0_S00_AXI_BASEADDR,0x0000000A);
    print("we = 1 y rst = 0,rst_pcm = 0, start = 1 \n\r");
    nanosleep_delay(420);
    Xil_Out32(XPAR_CRRS_FILTER_0_S00_AXI_BASEADDR,0x0000000C);

    //while(1){
    // reading data from the second personalized register of the crrs filter core.
    for (i = 0; i < num_samples; i++) {
        while (!(Xil_In32(BASE_ADDR + 4) & 0x2));  // wait until ARREADY is true
        int data = Xil_In32(BASE_ADDR + 4);  // read the data register.
        printf("pcm=%d\n",data);
        *(bram_ptr + i) = data;
        int bram_value = *(bram_ptr + i);
        printf("bram pcm=%d\n",bram_value);
        *(ddr_ptr + i) = bram_value;
    }
    printf("transfer done \n\r");
    //}
}
```

The implemented version of the code is in the test_model5 folder available in: `https://github.com/MaxRCC/CRRS_filter_test.git` inside the directory we can find the file system of a standard Vivado project, all the sources including the AXI lite custom ip and the driver design in sdk 2018.3 are available.

If we want to see the printed data, we have to connect and program the fpga board (zybo board) and connect the micro usb in the port shown in the **figure 6.1**, proceed to program the fpga in Vivado sdk 2018.3 and initialize the serial communication port. If the user does not want to check the printed data via serial terminal, use XSCT console from sdk and extract the data utilizing the memory content viewer available in sdk 2018.3 (the used version in this project).

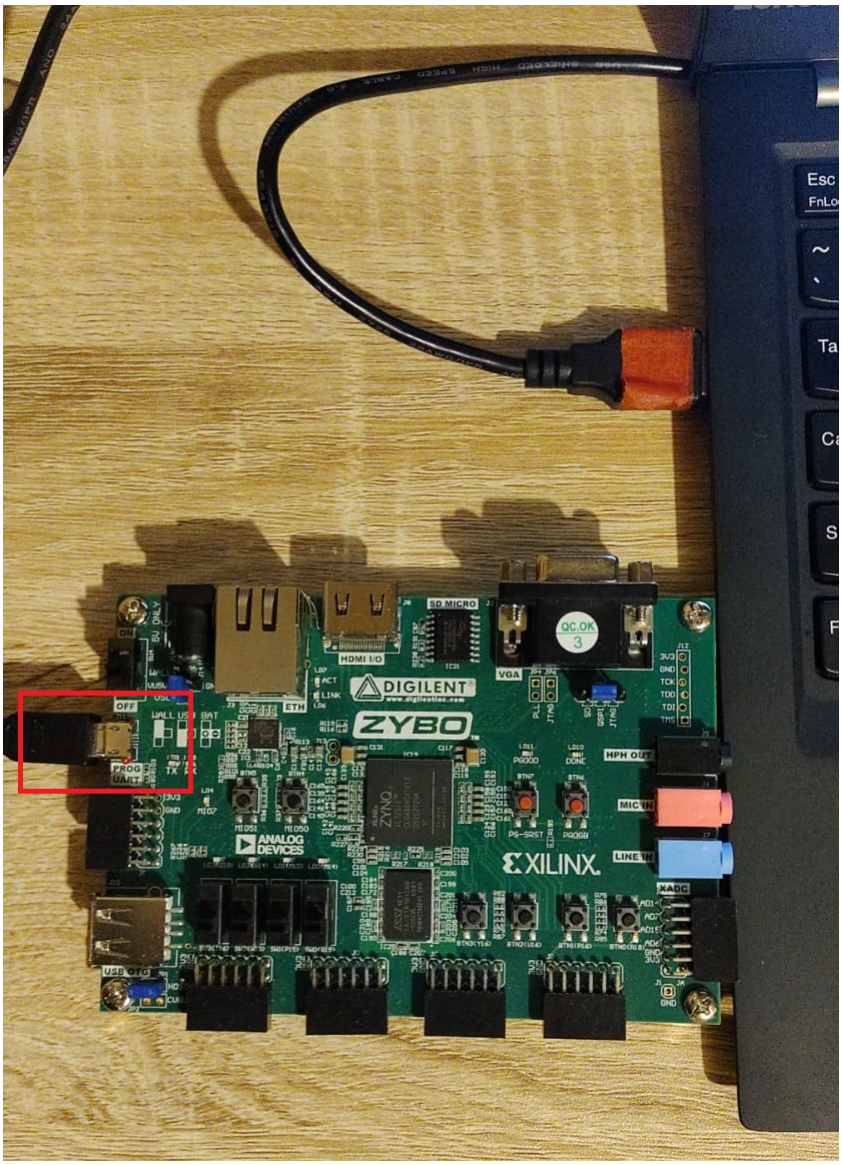

Figure 6.1: *Connection to uart*

References

[1] K. Herman, M. Montanares, M. Cerda, and L. Osses, "Sonar on chip," https://www.hackster.io/ef/sonar-on-chip-74a4c3, 2022, [Online; accessed 04-Jan-2023].

[2] R. Reis, "Trends on micro and nanoelectronics," 2021.

[3] G. Allevato, M. Rutsch, J. Hinrichs, M. Pesavento, and M. Kupnik, "Embedded air-coupled ultrasonic 3d sonar system with gpu acceleration," in *2020 IEEE SENSORS*. IEEE, 2020, pp. 1–4.

[4] B. da Silva, A. Braeken, K. Steenhaut, and A. Touhafi, "Design considerations when accelerating an fpga-based digital microphone array for sound-source localization," *Journal of Sensors*, vol. 2017, 2017.

[5] B. Zimmermann and C. Studer, "Fpga-based real-time acoustic camera prototype," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 1419–1419.

[6] R. J. Przybyla, H.-Y. Tang, A. Guedes, S. E. Shelton, D. A. Horsley, and B. E. Boser, "3d ultrasonic rangefinder on a chip," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 320–334, 2014.

[7] S. Pavan, R. Schreier, and G. C. Temes, *Understanding delta-sigma data converters*. John Wiley & Sons, 2017.

[8] E. Hogenauer, "An economical class of digital filters for decimation and interpolation," *IEEE transactions on acoustics, speech, and signal processing*, vol. 29, no. 2, pp. 155–162, 1981.

[9] D. Harris and S. L. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[10] B. Da Silva, A. Braeken, and A. Touhafi, "Fpga-based architectures for acoustic beamforming with microphone arrays: trends, challenges and research opportunities," *Computers*, vol. 7, no. 3, p. 41, 2018.

[11] K. Herman, E. Boemo, W. Fernandez, C. Duran-Faundez, and E. Rubio, "Preliminary studies on fpga implementation of a real-time ultrasonic

air-coupled sonar," in *2018 IEEE International Conference on Automation/XXIII Congress of the Chilean Association of Automatic Control (ICA-ACCA)*. IEEE, 2018, pp. 1–4.

[12] H. A. Sánchez-Hevia, R. Gil-Pita, and M. Rosa-Zurera, "Fpga-based real-time acoustic camera using pdm mems microphones with a custom demodulation filter," in *2014 IEEE 8th Sensor Array and Multichannel Signal Processing Workshop (SAM)*. IEEE, 2014, pp. 181–184.

[13] O. Semiconductor, "Lc706206ca - digital mems microphone controller including pre-amplifier," https://datasheets.su/DS/OnSemi/LC706206CA-D.pdf, 2017, [Online; accessed 04-Jan-2023].

[14] A. V. Oppenheim, "Applications of digital signal processing," *Englewood Cliffs*, 1978.

[15] "Learn the architecture - an introduction to amba axi," https://developer.arm.com/documentation/102202/0300/Channel-transfers-and-transactions, [Online; accessed 01-Sep-2023].