Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Electrónica.
Maestría en Ingeniería Electrónica, opción
Instrumentación Electrónica

Tesis presentada para obtener el grado de:
Maestro en Ingeniería Electrónica

"Portable Passive Digital SONAR Signal Processing Architecture Design and Implementation"

Presenta:
Ing. Luis Miguel Huesca Álvarez

Asesores:

M.C. José Franciso Portillo Robledo

M.C. Selene Edith Maya Rueda

Coasesor Externo:

Dr. Guillermo Quintero Pérez (SEMAR-UNINDETEC)

Puebla, Pue. Agosto de 2021

Agradecimientos

A mis profesores:
M.C. José Francisco Portillo Robledo, por su guía y atenciones
M.C. Selene Edith Maya Rueda, por sus atenciones y su tutela
M.C. Ana María Rodríguez Domínguez, por todo el apoyo durante la maestría.
A mis sinodales
Dra. Josefina Castañeda Camacho
Dra. María Monserrat Morín Castillo
M.C. Nicolás Quiroz Hernández
por darme la oportunidad de presentar mi trabajo.
A mi familia:
Ing. Clara Luz Huesca Álvarez, por todo siempre.
Miriam Hernández Huesca, por todo siempre.
Ing. Victor Ortíz Estévez, por su apoyo en esta etapa.
MVZ. Araceli Loyo Álvarez, hasta el cielo gracias por todo.
A mis comandantes:
Contralm. CG. Jesús Ortíz Estévez, por la oportunidad brindada.
Cap. Nav. CG. DEM. MSc. Miguel Alvarado Juárez, por ser un ejemplo a seguir.
Cap. Frag. CG. DEM. Ret. Javier Cornejo García, por sus enseñanzas.
A mis compañeros:
Dr. Guillermo Quintero Pérez, por sus consejos.
M.I. Josué Moisés Aguilar Garrido, por todo su apoyo y conocimientos
Tte. Nav. SIA. IMN. Alex Aguilar, por sus recomendaciones.
A mi mentora:
Dra. Andrea Guadalupe Martínez, por inspirarme a superarme.

0.1 Glossary

SONAR: Sound Navigation and Ranging
DSP: Digital Signal Processor or Processing, context-dependent
SAS: Synthetic Aperture Sonar
HD: High Definition
FPGA: Field-Programmable Gate Array
SUV: Submarine Unmanned Vehicle
FIFO: First-in, First-out
DDR: Double Data Rate
ST-FFT: Short-time Fast Fourier Transform
DEMON: Demodulation Of Envelope Modulation On Noise
IC: Integrated Circuit
I/O: Inputs Outputs
SoC: System On Chip
PS: Processing System
PL: Programmable Logic
BOM: Bill of Materials
LOFAR: Low Frecuency Analysis and Recording
JTAG: Joint Test Action Group
UART: Universal Asynchronous Receiver-Transmitter
HDMI: High Definition Multimedia Interface
OTG: On The Go
TPSW: Two-Pass Split Window
DC: Direct Current
BTR: Bearing-Time Reset
APU: Application Processor Unit
IOP: Inputs Outputs Peripherals
SNR: Signal-to Noise Ratio
IP: Intellectual Property
SPI: Serial Peripheral Interface
HDL: Hardware Description Language
CPU: Central Processing Unit
USB: Universal Serial Bus
ASCII: American Standard Code For Information Interchange
DFT: Discrete Fourier Transform
FIR: Finite Impulse Response
ADC: Analog to Digital Converter
GUI: Graphical User Interface

Contents

List of Figures

Listings

Abstract

The present work shows the development of a signal processing architecture for a passive sonar with portability capability, using the new hybrid technology of FPGA and CPU in the same IC package. This is made by introducing the sonar technology and some key concepts used such as spatial filtering, beamforming and special spectral analysis; then a design ins presented to fulfill the minimum requirements of a passive sonar system, then the main algorithm is developed using Python programming language, to finally implement personalized HDL modules, open source IP Cores and other open source tools in order to save complexity and time. Finally, air-based tests are made and the results and conclusions are presented.

Chapter 1

Introduction

1.1 Overview

SONAR stands for Sound Navigation and Ranging, it is based on the propagation of waves between a **target** and a **receiver**. [1] It is an equipment based on sound propagation in sea water that is used to navigate, communicate with or detect other vessels.
The two most common types of sonar systems are **passive** and **active**. In a **passive sonar system**, energy originates at a target and propagates to a receiver. In an **active sonar system**, waves propagate from a transmitter to a target and back to a receiver, analogous to pulse-echo radar.[2]
For getting in context, we will define the basic parts in a sonar system as follows:

1. Transducer receiver (in the vast majority of the systems, the transducer are in an array setting).
2. Signal conditioning stage
3. Signal processing stage
4. Control and display

The sonar equipment consist in two main parts, the **wet end** (front-end) and the **dry end** (back-end) which is usually considered as follows:

Wet end: Receiver Array, Cable/optical fiber, winch/connector.

Dry end: Signal processing and control console.

1.1.1 Passive Sonar

As mentioned before, passive sonar systems listens to the sound radiated by a target using a hydrophone (an underwater microphone) and detects signals against a background of the ambient noise of the sea and the self-noise of the sonar platform (all sounds made by the vessel in which the system is installed). Passive systems can be made directional, therefore the azimuth (horizontal bearing) of a signal is known. The received signal frequency spectrum and how it varies with time will help to classify the target. [1]

1.1.2 National background

In our country, it is possible to trace back the most important effort in this research area, to 2014. Such year the Marine Surveillance Sonar System (SIVISO) was approved in a cooperation between the Mexican Navy (SEMAR) and the Science and Technology National Council (CONACYT) in order to develop different sonar system prototypes such as passive (listen only), active (sound emitting) and bistatic (emission and listening) capable of being mounted in their Oceanic Patrol vessels. The mentioned project was fully made in the INIDETAM.
During this project development, it was stated the idea of making a portable sonar system in order to minimize the impact in the vessel architecture during the installation of this system, cost shrinking and many other advantages.

1.2 Goals

To design a complete signal processing scheme for a digital sonar system, fully implemented in a Xilinx Zynq Device, for using it in as the basis of a portable sonar probably soon-to-be-built sonar system in the Mexican Navy.

1.2.1 Specific Goals

- To design a digital sonar architecture first in a high level programming language, and then implement it in a programmable device
- To get the most of the device used for the implementation by exploiting its key features

1.3 Justification

As presented in 1.1.2, in Mexico, the design and development of this technology is nearly non-existent, and this is the key reason to aspire of extending it to as many institututions as possible, not only in the Mexican Navy[3] but in other institutions all across the country. Since the sonar technology is a combined effort in the research fields of underwater physics, acoustic engineering, electronic instrumentation, electronic system design and ocean engineering; several research projects could be made within this areas. It is sure that in the incoming years, this research are will be in expansion within the Mexican Navy. It is important to mention the author's experience in this subject, as a guarantee of certain degree of familiarity with this specific topic.

1.4 State-of-the-art

In the field of modern digital sonar systems, the author consider the following background:
1996. In France, the **DELPH** [4] side-scan sonar system was developed for seismic research. It has data display capabilities in a high resolution screen. Its major components were: for the analogic side, an analogic programmable amplifier with 20 KHz bandwidth and a 16 bit $\Delta - \Sigma$ Analogic-to-Digital Converter (ADC); for the digital side, a TMS320C31 Digital Signal Processor (DSP) running at 33 MHz and a TMS340-20 graphic card, both from Texas Instruments. Thus system realization demonstrates the great capabilities and reliability of digital sonars, all done with the minimum hardware.
1998. One of the first articles [5] introducing the FPGA devices in

the sonar signal processing is published. This is a very valuable article because it presents a qualitative comparison versus a DSP (as the one used in the DELPH system) specifically speaking in the Beamforming stage, which is crucial for this kind of systems. One of the conclusions is the advantage of FPGA devices among DSP in this application. The work was developed in the United States.

2008. Two Chinese universities publish an article which describes an *Beamforming* implementation for a 24x24 3D sonar working with 576 sensors, sampled in a rate of 2 KHz and 12 bits resolution, all done in a 37 FPGA architecture, thus demonstrating this device capabilities for such high performance systems.

2009. A Synthetic Aperture Sonar (SAS) [6] with coherent sound data processing for realizing a virtual sensor array with greater virtual dimensions than the real one, which is one of the most common techniques in today's sonar development. This system is mounted in an Submarine Unmanned Vehicle (SUV), and mainly consists: a) for the analogic side, 4 channels with low-noise amplifier, programmable gain amplifier, anti-alias filtering and a 12 bit resolution ADC; and b) for the digital side, a FPGA with embedded processors and FIFO memories. Later in 2011, a commercial system developed in Norway show us a state-of-the-art SAS, by accomplishing HD quality pictures of the sea bed.

2016. Mexico's first work in the digital sonar system field is presented, implementing a full FPGA architecture with embedded processors [3]

2018. In China is presented a recent work [7] showing us the new trend in high frequency sonars. The system presented consist in a multiple FPGA and Doube Data Rate (DDR) memories as key part of the digital architecture.

1.5 Proposed System Description

1.5.1 System Requirements

The very first step is define the design requirements. These requirements were based not only in the specialized literature but in the previous implementations made by the author.

- Get the best partitioning possible of the digital sonar
- Avoid as much as possible physical connections between the processing elements
- Different Frequency Analysis on the screen display.

1.5.2 Overview

The originally proposed system were a full implementation of a digital sonar divided in a analogic and a digital stage, but due to several factors beyond of the authors control, such as time restriction and an closely-related upcoming project, the decision of changing the structure of this work, without altering the original goal: implementing portable digital sonar signal processing architecture within a programmable device and also display the results. Refer to **Fig. 1.1**.

Figure 1.1: Architecture overview

1.5.3 Signal Processing Architecture Description

In any sonar system, this stage is immediately after the analogic system, but in this case it will be replaced with synthetic lab-made signals. In **Fig. 1.2**, a block diagram of the signal processing architecture is presented.

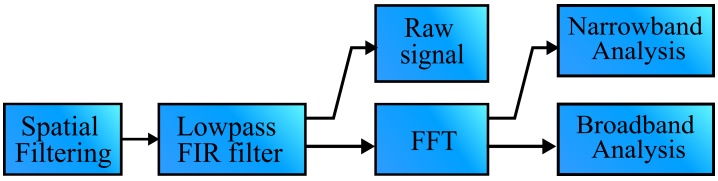

Figure 1.2: Preliminary Design of the signal processing architecture

The **spatial filtering** module is the so called *beamformer* is the space-time processor responsible to electronically "steer" the receiver array in order to improve the reception gain. In this case, it is proposed to be based in the **Delay-Sum Algorithm** which take advantage of the parallelism of the FPGA. The input of this module is the total number of receiving channels and its output is one signal made of the sum of all channels.

The **Lowpass filtering** adds an extra filtering layer that removes errors that could have origin during the digitization of the signal. It outputs a digitally filtered signal.

At this point, the signal goes through two different paths, one is to a memory to store the **raw received signal** for use it later and to stream directly to the console. The second path represents the spectral analysis which starts with a Fourier Transform implemented by means of the Fast Fourier Transform **(FFT)** and it is divided into two groups: the narrowband and the broadband analysis.[8]

The **narrowband analysis** is a very low frequency analysis (sometimes less than 100 Hz), that aims to know the key characteristics of the propulsion system of a vessel and make use of certain signal processing algorithms. The most known is the DEMON (Demodulation of Envelope Modulation On Noise) algorithm.

The **broadband analysis** is a low frequency analysis (from 0 to approx. 20 KHz) that attempts to get as most information as possible about the target. It is based on a technique called LOFAR (Low Frequency Analysis and Recording) widely used in the passive sonar system, useful for the target classification.

Either the broadband and narrowband analysis will be send to the

console for graphical visualization of the target.

1.5.4 Display Description

The systems needs the means to display the processed signal information in a coherent and meaningful way; the almost obvious way is in the form of sound, but also in the means of spectral graphics that show the frequency spectrum of the signal and valuable information product of the special analysis (such as DEMON and LOFAR). The proposed system will make of use of a Personal Computer and a custom Graphical User Interface (GUI) built upon a high level programming language framework (the proposed language is the Python 3.0 with PyQt 5 bindings).

1.5.5 Proposed Hardware

Due to lack of time and equipment to design and make a complete Printed Circuit Board, an FPGA Evaluation board was selected as the house of the logic stage of this project. The chosen board to build the signal processing architecture upon, is the **MiniZed** (shown in **Fig. 1.3**), manufactured by AVNET with a cost of 89 USD. [9] This board meets with the following criteria:

- Lowest cost in market for a Zynq 7000 Series IC
- Lightweight and small form factor
- Extensive documentation and wide community support

The main features of this boards are listed as follows:

- Main IC: Xilinx Zynq XC7Z007S SoC
- Memory:
  - Micron 512 MB DDR3L
  - Micron 128 MB QSPI flash
  - Micron 8GB eMMC mass storage

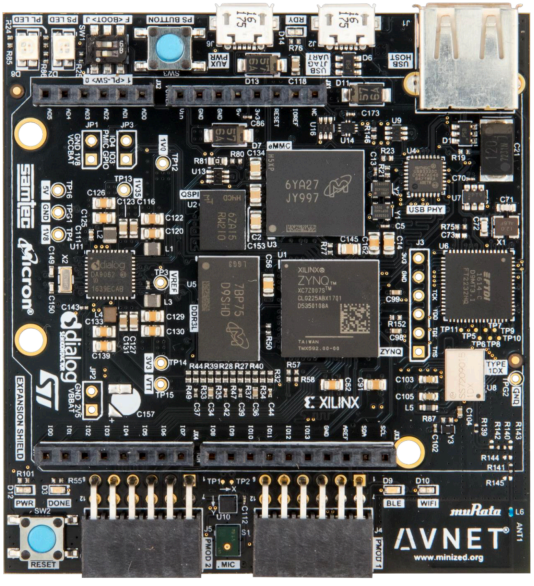

Figure 1.3: AVNET MiniZed Development Board

- On-board USB to JTAG and debug UART circuit

Due to the size of the final FPGA implementation, it was mandatory to change the development board. The immediate option was **Arty Z7-20** (shown in **Fig. 1.4**) because these board not only meets and exceeds the same criteria of the previous board but it is also in the price range (199 USD) for educational boards, and it has high availability and it does not have exportation restriction.

The main features of this boards are listed as follows:

- Main IC: Xilinx Zynq XC7Z020 SoC
- Memory:
  - Micron 512 MB DDR3
  - Micron 16 MB QSPI flash
  - MicroSD slot

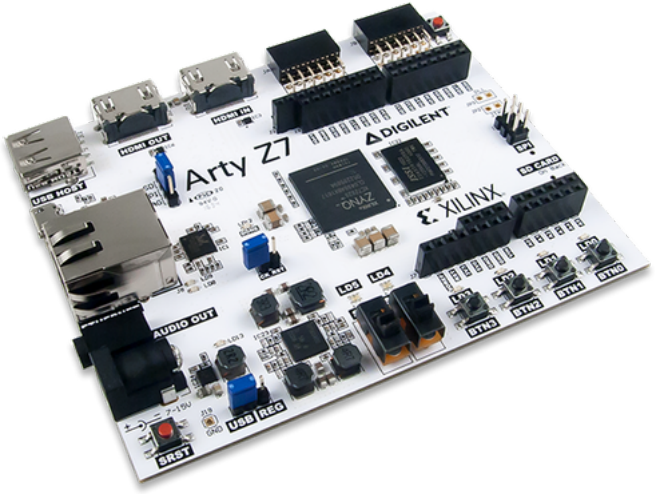

Figure 1.4: Digilent Arty Z7

- Communications:
  - Gigabit Ethernet PHY
  - USB 2.0 host interface: JTAG, UART and OTG
  - HDMI Sink
  - HDMI Source
- Expansion connectors
  - Arduino-compatible shield interface
  - 2x Digilent Pmod-compatible interfaces
- General Purpose I/O
  - 4x User buttons
  - 4x User switches
  - 2x User LEDs
  - 2x RGB LEDs

1.6 Document Organization

The **Chapter 1** gives an introduction of sonar technology and the project itself, next, **Chapter 2** contains the basic information about passive sonar system signal processing design, then **Chapter 3** fully describes the procedure of design and implementation of the system, and finally in **Chapter 4** results, conclusions and future work are presented.

Chapter 2

Chapter 2: Background

2.1 Brief History of the Passive Sonar Systems

The concept of passive sonar is very intuitive and ancient, this is the reason why we could trace back to the famous Leonardo da Vinci with his first attempts to hear underwater noises aiming to detect distant boats.[10] In 1889, the U.S. Lighthouse Board described L.I. Blake comprising and underwater bell and a microphone receiver, and a similar system. Portable omnidirectional hydrophones were available in 1915, directional hydrophones came in 1917. In 1918, a prototype passive range-ranging system was fitted to an American destroyer (USS Jouet). The first Sonar systems were born due to the urge to detect submarines and other vessels during the World War I. Thus we could see that this equipment was born for military applications and then spread to civilian applications.

2.2 Digital Sonar

In the 1960s, designers began to use digital techniques to implement their designs and thus began the trend to include more and more digital systems, replacing analog electronics when possible because it represents an increase in the processing capabilities and the general performance. There are several advantages of the digital sonar, specifically in the

signal processing subject, compared with the analog:

- Avoiding common errors such as phase-shifting, non-uniform filtering, etc; thus increasing the precision of beamforming.
- A digital system can store necessary data and characteristic parameters in the sonar signal processing procedure for later output. This will be helpful in the development of simulators and training modules.
- Increased ease in the realization of modules, series and standard designs, therefore, it is possible to design a single general purpose sonar signal processing platform.
- Improved system reliability and maintainability.
- Fault diagnosis can be carried out in a real time and the position of failures can be located precisely.
- It is easier to realize a new algorithm in a digital form. Only the high speed digital chips can support such techniques.

2.2.1 FPGA Use in Sonar

About a decade ago began the trend of implementing the signal processing blocks in configurable devices like the Field-Programmable Gate Array (*FPGA*), taking advantage of their key features such as speed, flexibility and paralelism [11], [12].
The main advantages of using these devices in the development of project prototypes is described in [13], being:

Performance: similar to a final device.

Low cost: thanks to the hardware reuse and the low cost of the development boards and platforms.

Custom Inputs/Outputs: gives the ability to reuse existent hardware along with the new system design, besides of being able to change major system functionalities even after the implementation.

Fault-tolerance: thanks innovations in these devices manufacture technology, and also the ease of implementation of redundant system in them.

As cited in [5], the FPGA is nearly a natural choice because of their great simultaneous data processing capabilities.

2.2.2 Commercial Systems

Nowadays, in the maritime industry, there many international companies that fully develop this kind of systems in all the stages: research, development, commercialization and client support. Some of the most important, as considered by the author, are:

Military systems

- Raytheon (United States of America)
- Thales (France)
- Furuno (Japan)
- Kongsberg (Polish)

Civilian systems

- Simrad (Norway)
- Lowrance (United States of America)
- Garmin (United States of America)

It is important to keep in mind that many different sonar systems have been developed for every application, in order to adapt the specific needs and requirements. The most important applications could be:

- Fishing
- Bathymetry and seismic imagenology.
- Navigation
- Target Detection
- Underwater Communication

2.3 Discrete Sensor Arrays

2.3.1 Overview

A sensor array can be considered to be a sampled version of a continuous aperture, where the aperture is only excited at a finite number of discrete points. As each element can itself be considered as a continuous aperture.[14]
Superposition of sensor responses results in an array response that approximates the equivalent continuous aperture.

$$A(f, x_a) = \sum_{\frac{N-1}{2}}^{-\frac{N-1}{2}} w_n(f) e_n(f, x_a - x_n) \tag{2.1}$$

where:
$w_n(f)$: the complex weight for element n
$e_n(f, x_a$: complex frequency response
x_n: spatial position on the x axis
For better understanding, refer to **Fig. 2.1**

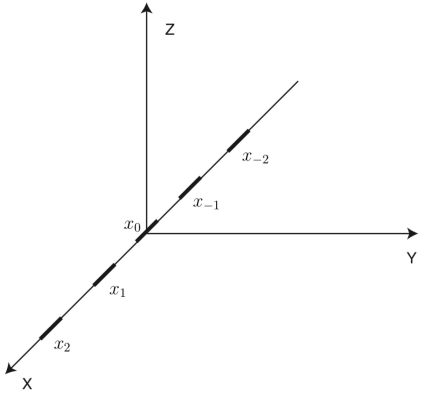

Figure 2.1: A linear discrete sensor array

2.3.2 Directivity Pattern

The response of a receiving aperture is directional (see **Fig. 2.2**, because the amount of signal seen by the aperture varies with the direction of arrival of the wave.
The aperture response as a function of frequency and direction of arrival is known as the aperture directivity pattern or beam pattern.
directivity pattern depends upon:

- the number of array elements N
- the space between array elements d
- the frequency f

The effective length of a sensor array is the length of the continuous aperture which it samples, and is given by

$$L = Nd \tag{2.2}$$

where:
L: the array length
N: the number of array elements
d: the inter-element spacing

The actual physical length of the array, as given by the distance between the first and last sensors.

There are some special phenomena that we have to keep in mind when we implement processing algorithms:

- the beam width decreases as the effective array length increases, that is also increasing the spacing between elements L.
- beam width is inversely proportional to the product fL.
- the sidelobe level decreases with increasing spatial sampling frequency - that is, the more sensors we use, the lower the sidelobe level.
- beam width and the sidelobe level, are directly determined by the inter-element spacing and the number of sensors respectively.

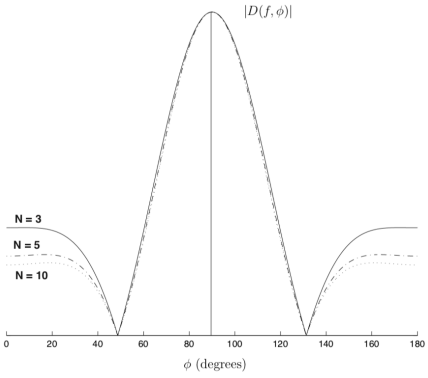

Figure 2.2: An example of a directivity pattern

2.3.3 Spatio-Temporal Sampling

Recall the Nyquist sampling theorem, which states that a signal must be sampled at a rate f_s (with a period T_s) such that

$$f_s = \frac{1}{T_s} 2 f_{max} \tag{2.3}$$

Analogous to this, there is a spatial sampling requirement, since the sensor arrays implement spatial sampling. This *temporal sampling theorem* states the next:

$$d = \frac{\lambda_{min}}{2} \tag{2.4}$$

where:
d: the inter-element spacing.
λ_{min}: minimum wavelength in the signal of interest.

2.3.4 Beamforming

From 2.1 we can recall the complex weights w_n and define them as

$$w_n(f) = a_n(f) e^{j\phi_n(f)} \tag{2.5}$$

where:
a_n and $\phi_n(f)$ are real, frequency dependent amplitude and phase weights respectively. Now it is very important to state that, if we modify this parameters, we can modify parameters of the directivity pattern, this way:

- If we modify the amplitude weights a_n we can modify the *shape* of the directivity pattern.
- If we modify the phase weights $\phi_n(f)$ we can modify the *angular location* of the main lobe.

Keeping this in mind, we define the **Beamforming techniques** as algorithms for determining the complex sensor weights $w_n(f)$ in order to implement a desired *shaping* and *steering* of the array directivity pattern. Refer to **Fig. 2.3**

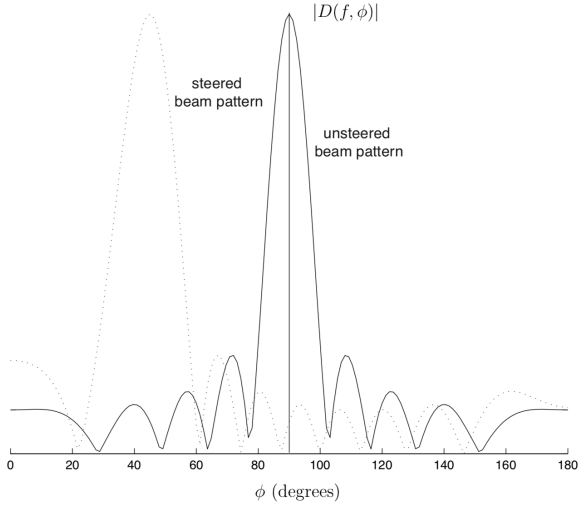

Figure 2.3: Illustrating the steering of a directivity pattern

From Fourier transform theory, we can recall that a phase shift in the frequency domain corresponds to a time delay in the time domain

and vice versa. Thus, beam steering can be implemented by applying time delays to the sensor array inputs. The delay is given by:

$$\tau_n(f) = \frac{nd\cos\phi'}{c} \tag{2.6}$$

where:
n: total number of sensors
d: the inter-element spacing
ϕ :: desired angle
c: speed of sound (approx. 330 m/s)

The simplest of all beamforming techniques, known as *Delay-sum beamforming*, where the time domain sensor inputs are first delayed by τ_n seconds, and then summed to give a single array output.

2.3.5 Delay-Sum Beamforming

This is the simplest of all microphone array beamforming techniques. It is called this way because the time domain sensor inputs are first delayed by τ_n seconds, and then summed to give a single array output. Each channel each channel is given an equal amplitude weighting in the summation, thus the directivity pattern demonstrates unity gain in the desired direction. The weights are now given by:

$$w_n(f) = \frac{1}{N}(f)e^{j\frac{-2\pi f}{c}(n-1)d\cos\phi'} \tag{2.7}$$

And in the time domain we have

$$y(t) = \frac{1}{N}\sum_{N}^{n=1} x_n(t-\tau_n) \tag{2.8}$$

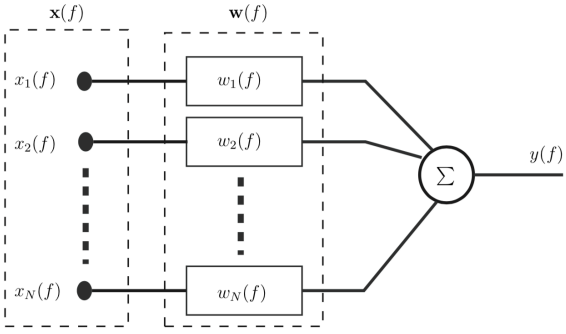

Figure 2.4: Basic Delay-Sum Beamformer scheme

2.4 Sonar Signal Processing

2.4.1 Sonar Signal Analysis

Major difficulty in passive sonar systems is to detect the target in huge background noise environments. Several techniques have been developed in order to do this. According to the autor, two of the most important, are the industry-standard methods: LOFAR and DEMON analysis. [15]

2.4.2 LOFAR Analysis

LOFAR stands for *Low Frequency Analysis and Recording* is a broad-band analysis mainly focused in the frequency of interest range of the target noise.
The algorithm consist in the next steps (See **Fig 2.5**):

1. The summed bearing signal from the sensor array is multiplied by a Hanning window

2. this product is passed to a Short-Time Fast Fourier Transform (ST-FFT)

3. the module of this spectral representation is obtained

4. finally, a Two-Pass Split Window (TPSW) is applied for normalization

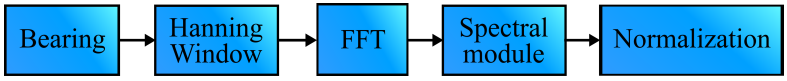

Figure 2.5: LOFAR analysis scheme

In simple terms, we can present the LOFAR Analysis as common spectral analysis focused

2.4.3 DEMON Analysis

DEMON stands for *Demodulation of Envelope Modulation On Noise*. It is a narrowband analysis focused in a very low range of frequency that is directly associated with the machinery and ship propulsion noise. The algorithm consist in the next steps (See **Fig 2.6**):

1. The summed bearing signal from the sensor array is bandpass filtered
2. this filtered signal is squared
3. this squared signal is also plugged into a TPSW to remove the DC component
4. this signal is now resampled
5. finally, a ST-FFT is used to get the spectral components

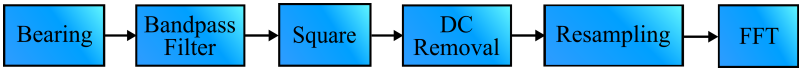

Figure 2.6: DEMON analysis scheme

2.5 Zynq Device Introduction

The Xilinx Zynq device is a SoC (System On Chip) that contains a either dual or single-core ARM® CortexTM-A9 MPCore with traditional Field Programmable Gate Array (FPGA) logic fabric, all in the same encapsulated die. In the Zynq device, the ARM Cortex-A9 is an application grade processor, capable of running full operating systems such as Linux, while the programmable logic is based on Xilinx 7-series FPGA architecture. The architecture is completed by industry standard AXI interfaces, which provide high bandwidth, low latency connections between the two parts of the device [16]. A diagram of this architecture is presented in **Fig. 2.7**

The parts of the Zynq device are:

Processing System (PS): dual or single-core ARM® CortexTM-A9 MPCore.

- Application processor unit (APU)
- Memory interfaces
- I/O peripherals (IOP) Interconnect

Programmable Logic (PL): Xilinx 7-series FPGA architecture.

This concepts are extremely important because, from now on, they will be used across this document.

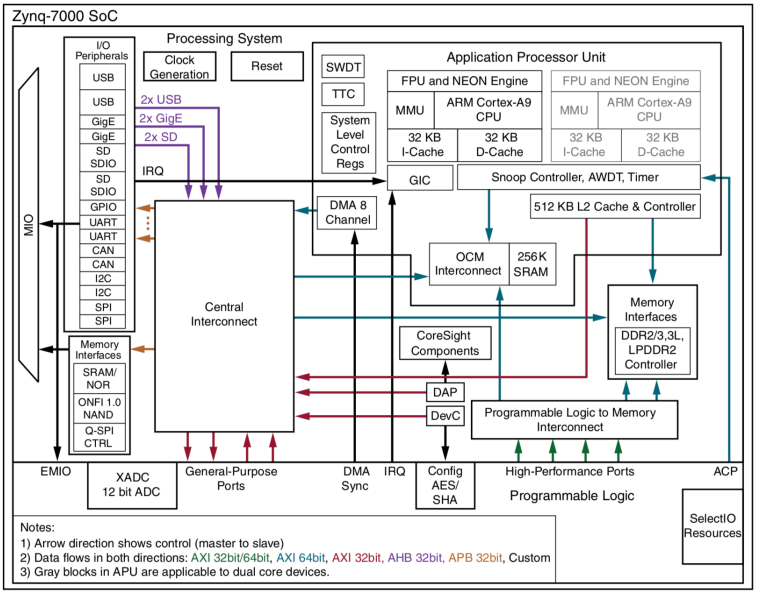

Figure 2.7: Zynq 7000 SoC Overview

2.6 NMEA 0183 Communication Protocol

It is a de facto standard in onboard electronics developed and maintained by the National Maritime Electronics Association, used for communicating sensors with displays and other processing electronics. It is based in the concept of talker and listener, as viewed in The technical specifications are:

- Serial based
- 4800 bps
- 8 bits width
- the last bit must be a 0
- No parity and one stop bit

The NMEA format sentences must follow the next structure:

1. `$`

   (HEX 24) or

   `!`

   (HEX 21) as the start of the sequence.

2. `<data field>`

   this contains the talker ID and the sentence formatter.

3. `[',' <data fields>]`

   the actual information of the sensor

4. `['*' <checksum field>]`

   for verifying the transmission integrity

5. `<CR>`

   (HEX 0D)

   `<CR>`

   (HEX 0A) *Carriage Return* and *Line Feed*, respectively: as the end of the sequence

We will use the Talker Identifier Mnemonics

`SS`

which stands for Sounder Scanning

Chapter 3

Design and Implementation

3.1 Design Criteria

As stated in the Chapter 2, the goal of this project is "design and develop a digital sonar signal processing architecture for using it in a portable system".
For the author, one very important goal is to use as much of Open Source software as was possible. The design of the algorithm was made in Python 3, specifically within the Anaconda Individual Distribution due to the different constraints such as time restriction and limited set of hardware. During the realization of this work, it a Jupyter Notebook (see **Fig. 3.1**) with basic signal processing examples as a companion for the author to develop these algoritms.

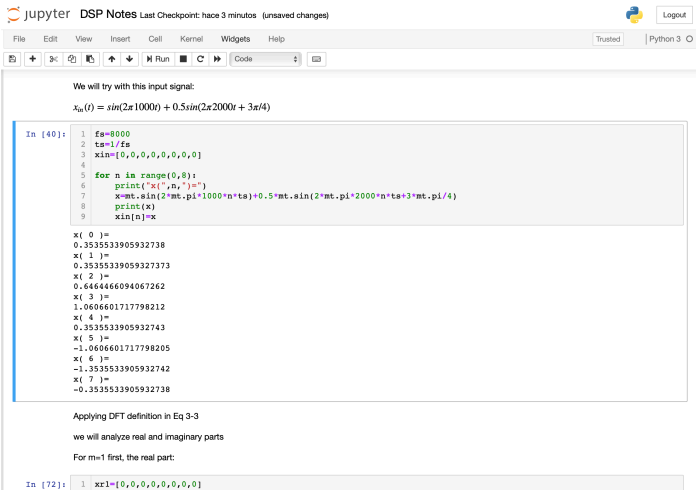

Figure 3.1: Jupyter environment being used

3.2 Architecture Design

The design was made sequentially following the basic passive sonar signal processing architecture presented in **Fig. 1.2**.

3.2.1 Beamformer Design

Angle Calculation

The first step for the Design of the beamformer is to define the next criteria:

Array shape: linear. This geometry was selected because it is simple yet useful when implementing the algorithms.

Number of elements: 3 elements. This number is the minimum required to do the beamforming, and it was selected to simplify the development; the number of elements in the array will be the number of channels.

Spacing between elements: 37.8 cm. This value was set due to the dimensions of the model constructed, otherwise, it would have been larger and more acoustic power were needed in order to catch up the signal on the transducer elements.

Frequency: 10 kHz. This value was selected based in the author's previous experience (from 0 to 15 KHz) and it is also used in sonar applications, and falls in the human acoustic range, thus, audio-compliant devices could be used so.

Resolution: 16 angles. This value was not only selected based in the author's previous experience but also in the fact that increase this value will exponentially grow the hardware processing requirements and increment the difficulty.

This parameters were defined keeping the system as simple as possible. Now we proceed to calculate the value of the angles, considering that a linear array has a theoretical field of 180^{o}.

Then, the time delays are calculated used the formula in [3], which states:

$$\tau_\theta = \frac{d\sin(90-\theta)}{c} \tag{3.1}$$

where:
τ_θ: time delay
θ: specific angle
c: sound velocity. In water $c = 1500m/s$, in air $c = 333m/s$

The delay and sum-beamformer is described by

$$r(t,\theta) = \sum_{m=1}^{M} S_m(t-\tau_\theta) \tag{3.2}$$

3.2.2 Signal Processing Scheme Design

In matter of the signal processing, for the spectral analysis, a test signal was used, defined as:

$$x_{in}(t) = sin(2\pi 1000t) + 0.5sin(2\pi 2000t + 3\pi/4) \tag{3.3}$$

The flow diagram (**Fig. 3.3**), the code (**List. 3.1**) and the output plots (**Fig. 3.2**) are shown next. All the Flow Diagrams of this section were made using the **PyFlowchart** package available in [**?**].

```
for n in range(0,15):
    tn.append(n)
    print("x(%i)= x"%n)
    x=mt.sin(2*mt.pi*1000*n*ts)+0.5*mt.sin(2*mt.pi*2000*n*ts
    +3*mt.pi/4)
    print(x)
    xin[n]=x
```

Listing 3.1: Generating test signal

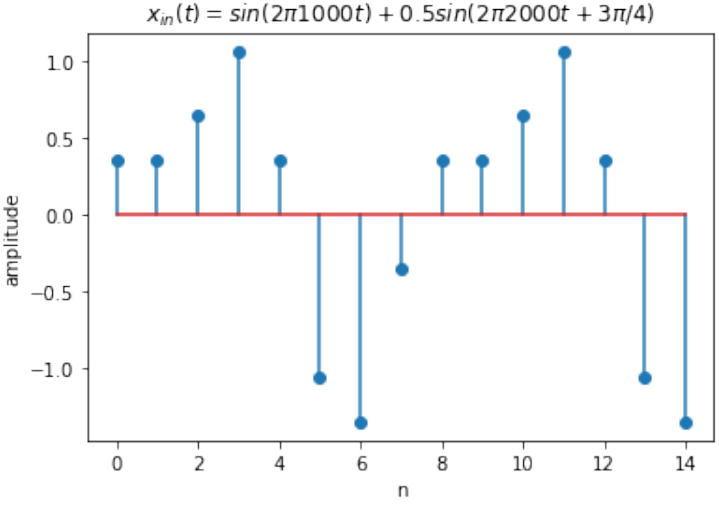

Figure 3.2: Test signal

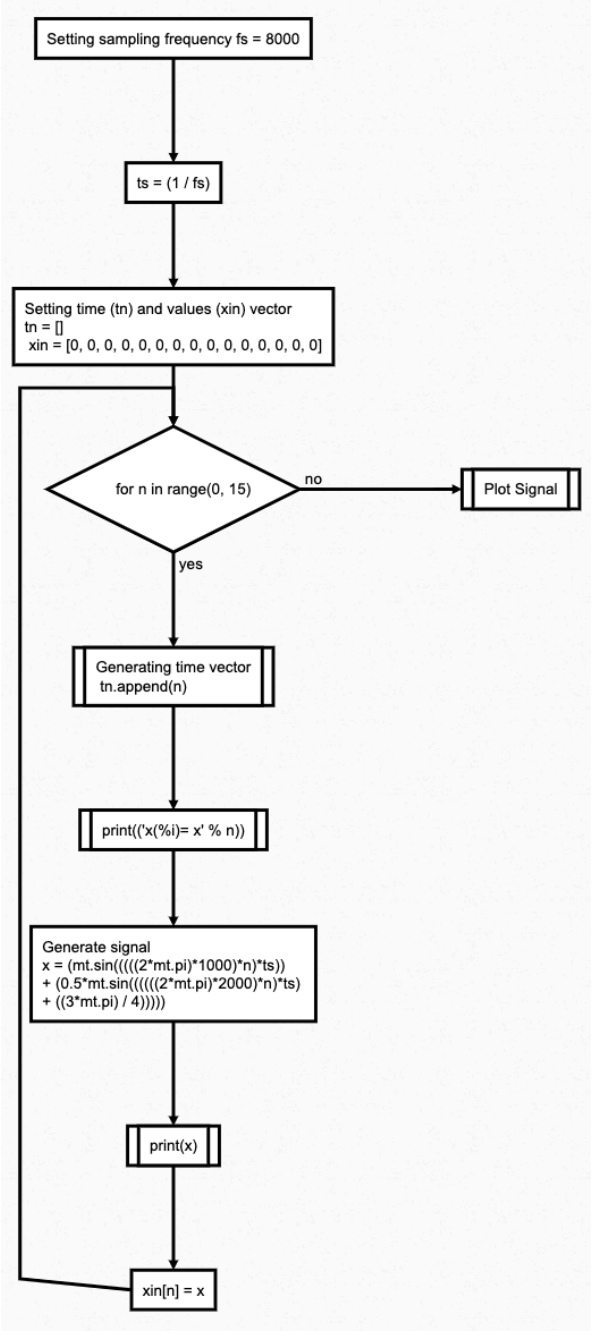

Figure 3.3: Flow diagram for the Signal Generator

Creating generalized functions to calculate real, imaginary parts, magnitude and degrees using only pure Python native functions and the Discrete Fourier Transform Definition (**Eq. 3.4**):

$$X(m) = \Sigma_{n=0}^{N-1} x(n)[cos(2\pi nm/N) - jsin(2\pi nm/N)] \qquad (3.4)$$

The code and the output plots are shown in **List. 3.2** and **Fig. 3.4** respectively.

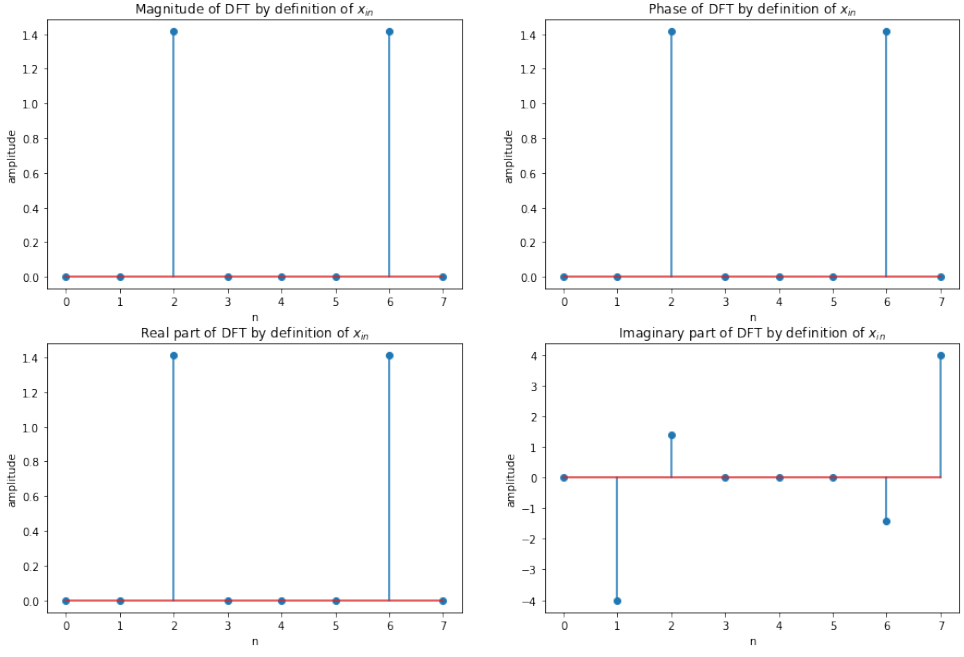

Figure 3.4: Fourier Transform calculation using definition

```
def calculateRealPartXn(xinput,n,N,m):
    #real part
    x_r_n = []
   # nn=n
   #  n=0
    for n in range(0,N):
        x_r_n.append(xinput[n]*mt.cos(m*2*mt.pi*n/N))
    sum_x_r_n = sum(x_r_n)
    #print("x(",nn,") real=",sum_x_r_n)
    return sum_x_r_n

def calculateImagPartXn(xinput,n,N,m):
    #imaginary part
    x_i_n = []
    #nn=n
    #n=0
    for n in range(0,N):
        x_i_n.append(xinput[n]*-mt.sin(m*2*mt.pi*n/N))
    sum_x_i_n = sum(x_i_n)
    #print("x(",nn,") imag:",sum_x_i_n)
    return sum_x_i_n

def getPolarForm(sum_x_r_n, sum_x_i_n, m):
    #X(n) magnitude
    x_n_mag=mt.sqrt((sum_x_r_n)**2+(sum_x_i_n)**2)
    #print("X_mag(",m,")=",x_n_mag)
    #X(n) phase
    x_n_pha=mt.atan(sum_x_i_n/sum_x_r_n)
    #print("X_pha(",m,")=",x_n_pha)
    x_n_pha_deg=(x_n_pha*180)/(mt.pi)
    #print("X_pha_Deg(",m,")=",x_n_pha_deg,"º\n")
    return x_n_mag, x_n_pha
```

Listing 3.2: Fourier Transform calculation using definition

In the signal processing part, a Hanning Window is used and its calculation is presented as flow diagram in **Fig. 36**, then the code and the output plots are presented in **List. 3.3** and **Fig. 3.5**, respectively:

```
N_=32
w=[]
tn = []
for n in range(0, N_):
   tn.append(n)
   w_ = 0.5-0.5*mt.cos((2*mt.pi*n)/N_)
   print("w(",n,"): ",w_)
   w.append(w_)

def HanningValues(N):
    w=[]
    for n in range(0, N_):
        w_ = 0.5-0.5*mt.cos((2*mt.pi*n)/N_)
        w.append(w_)
    return w
```

Listing 3.3: Hanning Window Calculation

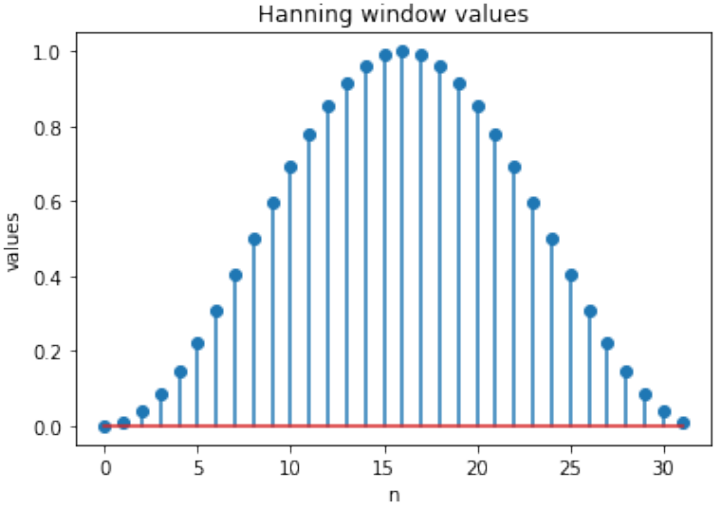

Figure 3.5: Hanning Window Calculation

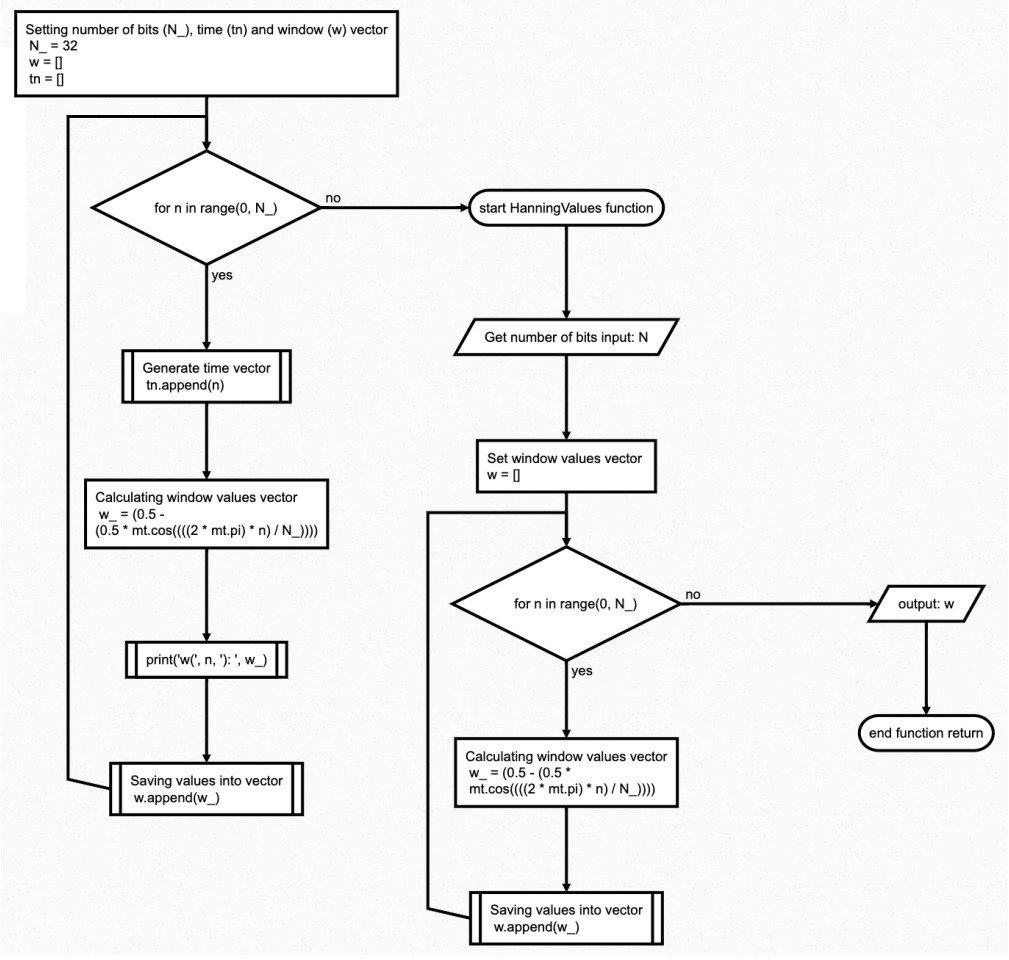

Figure 3.6: Flow diagram of the Hanning Window Calculation

Applying a window to the signal makes it more suitable for processing. The flow diagram (**Fig. 3.8**), the code (**List. 3.4**) and the output plots (**Fig. 3.7**) are shown next.

```
N_=64
w=[]
tn = []
s = []
ws = []

#generating signal and discrete time vector
for n in range(0, N_):
    tn.append(n)
    s_ = mt.sin((2*mt.pi*3.4)*n/N_)
    s.append(s_)

#generating window
for n in range(0, N_):
    w_ = 0.5-0.5*mt.cos((2*mt.pi*n)/N_)
    w.append(w_)

#generating windowed signal
for n in range(0, N_):
    ws_ = s[n] * w[n]
    ws.append(ws_)
```

Listing 3.4: Signal Windowed

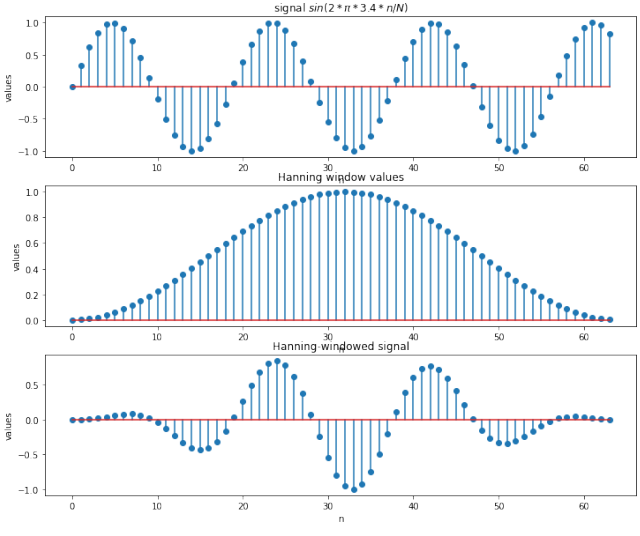

Figure 3.7: Steps of a windowing operation

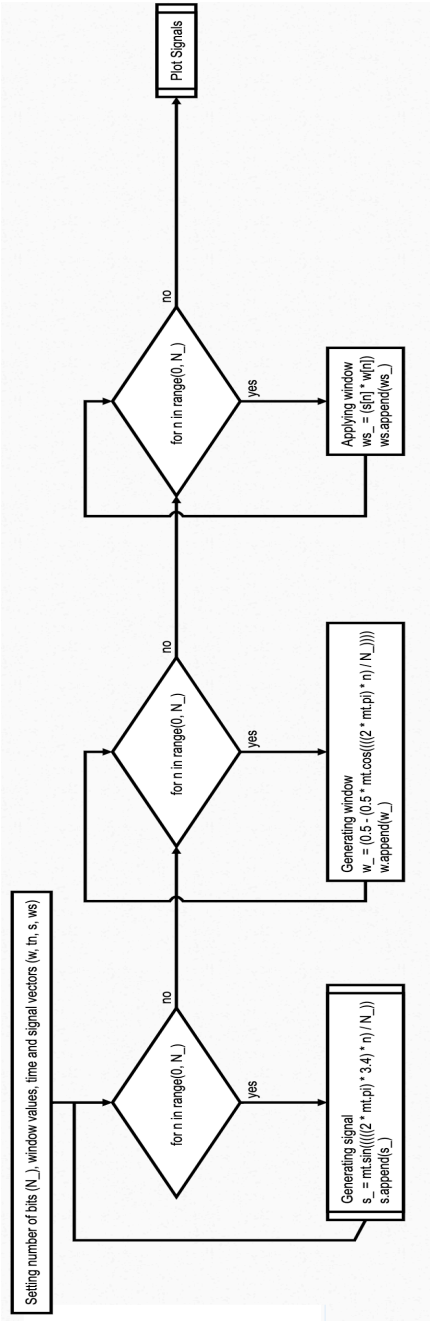

Figure 3.8: Flow diagram of the windowing operation

This operation is well-known for minimize the noise in the output Fourier Transform. The flow diagram (**Fig. 3.10**), the code (**List. 3.5**) and the output plots (**Fig. 3.9**) are shown next.:

```
N_=64
w=[]
tn = []
s = []
ws = []

#generating signal and discrete time vector
for n in range(0, N_):
    tn.append(n)
    s_ = (mt.sin((2*mt.pi*3.4)*n/N_)+0.1*mt.sin((2*mt.pi*7)*n/
    N_))
    s.append(s_)

#generating Hanning window
hanning_ = HanningValues(N)

#generating windowed signal
for n in range(0, N_):
    ws_ = s[n] * hanning_[n]
    ws.append(ws_)

#rect form
Xr_lsts=[] #real part list
Xi_lsts=[] #imaginary part list
#polar form
Xm_lsts=[] #magnitude
Xp_lsts=[] #phase
N=64

Xr_lsts, Xi_lsts, Xm_lsts, Xp_lsts = DFTbyDef(s,N)
Xr_lstws, Xi_lstws, Xm_lstws, Xp_lstws = DFTbyDef(ws,N)

fig = plt.figure(figsize=(16,5))
plt.subplot(121)
plt.stem(tn[0:31],Xm_lsts[0:31])
plt.axis([0, 32, 0, 30])
plt.ylabel('values')
plt.xlabel('n')
plt.title("DFT of signal")
plt.subplot(122)
```

```
plt.stem(tn[0:31],Xm_lstws[0:31])
plt.axis([0, 32, 0, 30])
plt.ylabel('values')
plt.xlabel('n')
plt.title("DFT of Windowed signal")
plt.show()
```

Listing 3.5: Windowing effect

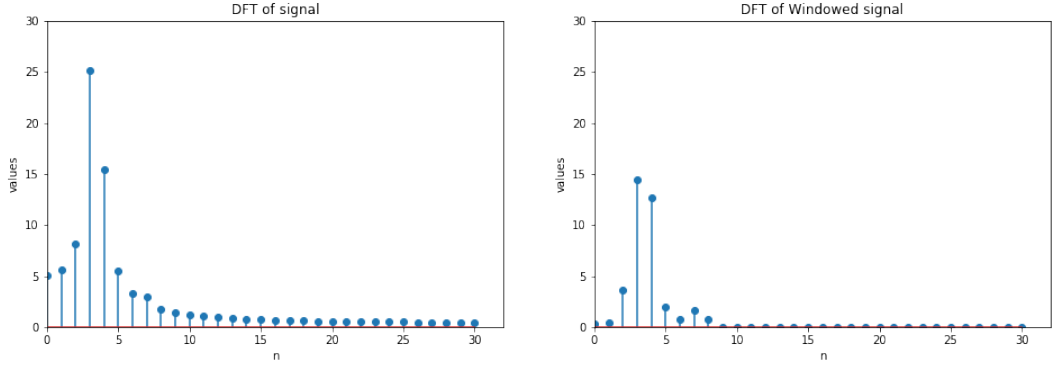

Figure 3.9: Effects of windowing in the Fourier Transform

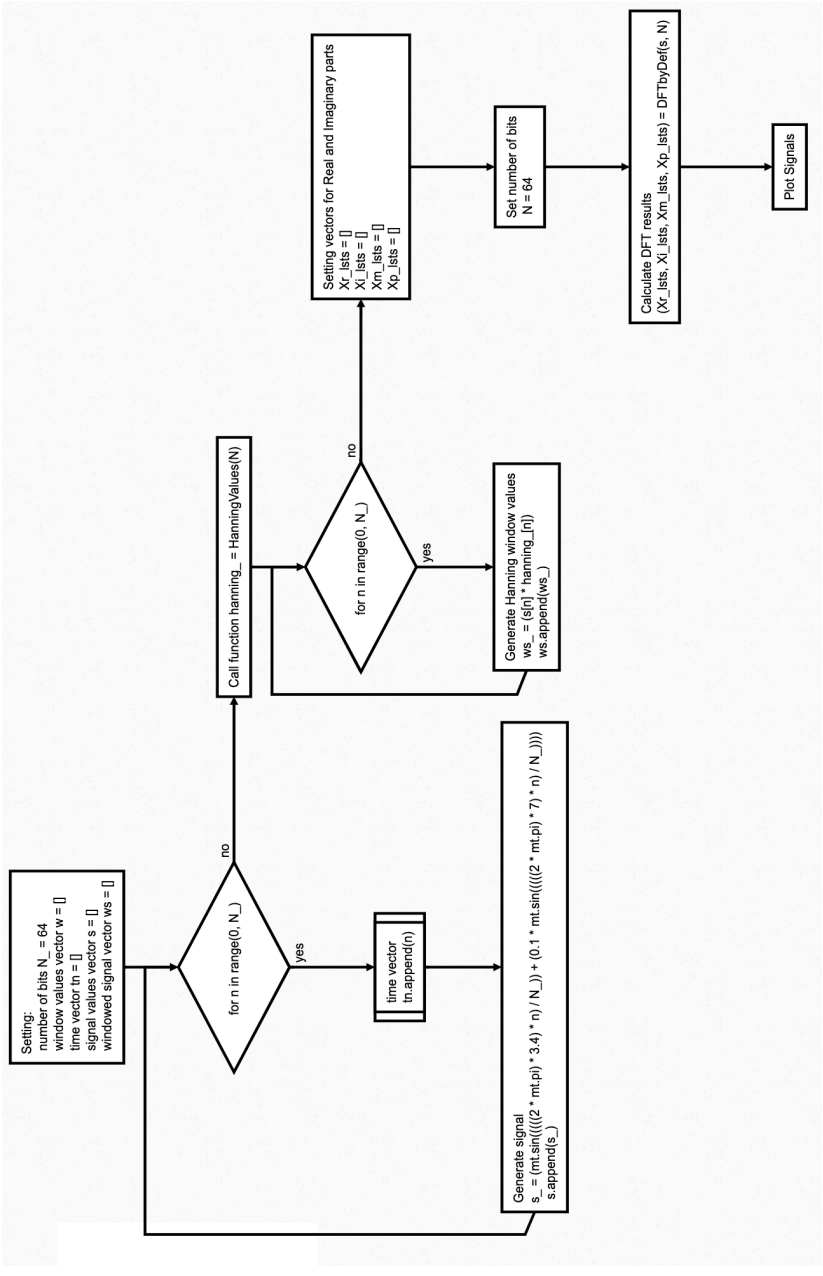

Figure 3.10: Flow diagram of the windowing effect

The Fast Fourier Transform implemented in this project uses **zero-padding**, this technique must be used with caution due to its effects in spectral distortion. The flow diagram (**Fig. 3.12**), the code (**List. 3.6**) and the output plots (**Fig. 3.11**) are shown next. :

```
N_=16
tn = []
s = []

#generating signal and discrete time vector
for n in range(0, N_):
    tn.append(n)
    s_ = mt.sin((2*mt.pi*3)*n/N_)
    s.append(s_)

#rect & polar form in a nicer way
Xr_lsts,Xi_lsts,Xm_lsts,Xp_lsts=[],[],[],[]
Xr_lsts, Xi_lsts, Xm_lsts, Xp_lsts = DFTbyDef(s,N_)

#now zero-padding N=32, 16 padded zeros
N=32
for n in range(N_, N):
    s.append(0)

tnz1=[]
for i in range(0,N):
    tnz1.append(i)

Xr_zp1, Xi_zp1, Xm_zp1, Xp_zp1 = DFTbyDef(s,N)
```

Listing 3.6: Zero-padding effect on the DFT

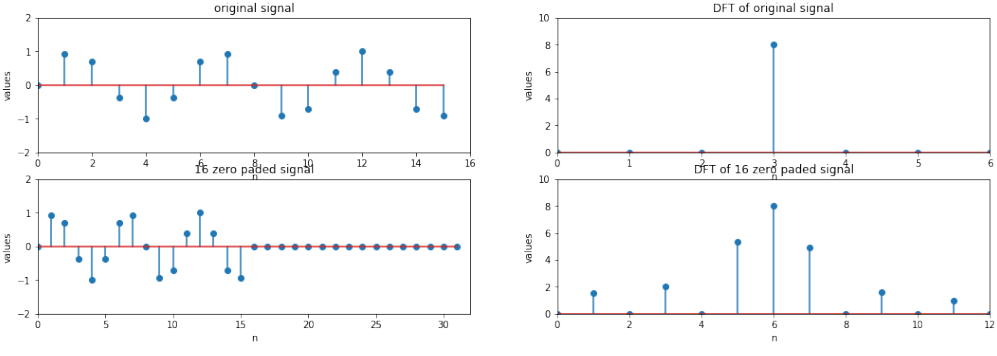

Figure 3.11: Zero-padding effect on the DFT

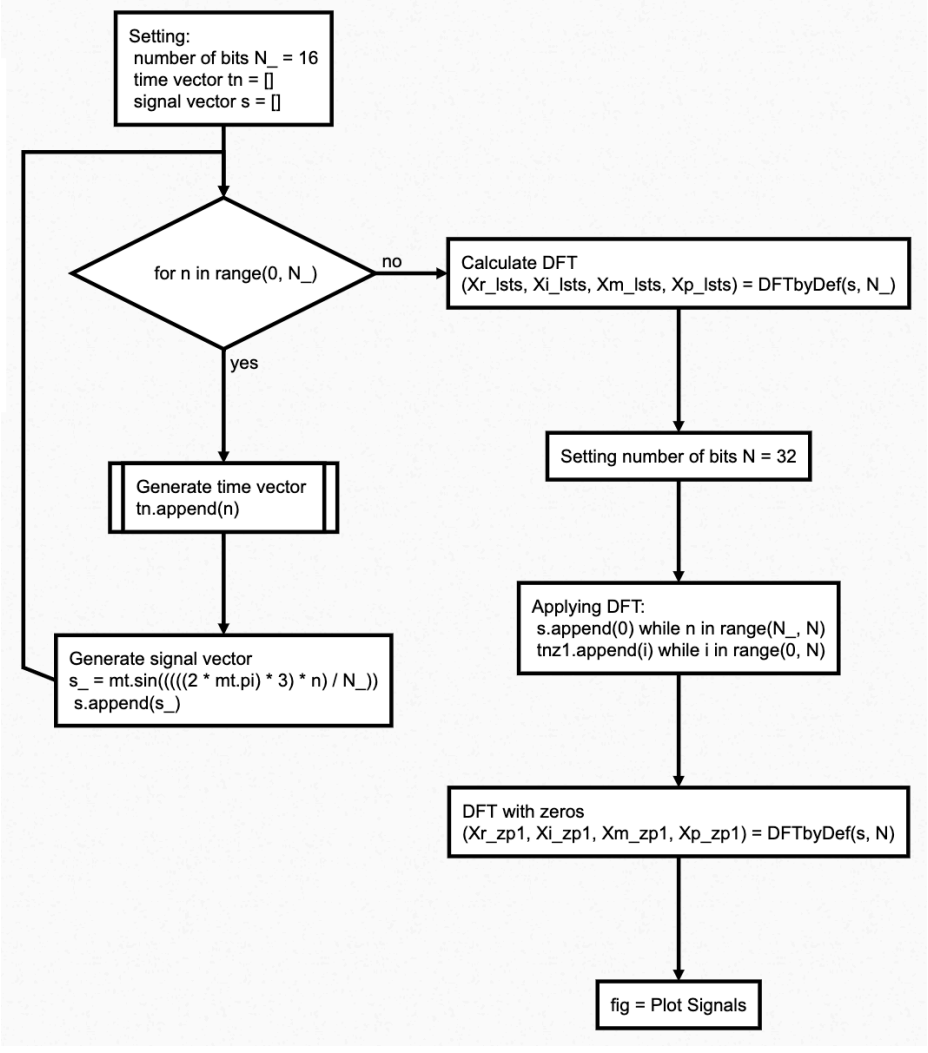

Figure 3.12: Flow diagram of Zero-padding

The sole operation of applying the Discrete Fourier Transformations yields us to have a Processing Gain associated with this process, which in the case of the spectral analysis used in passive sonar is particularly useful. The amplitude of the main frequency bin is enhanced as the N increases, even in a really noisy environment such as the sea. As we increase the number of N points of the DFT, the the input tone amplitude gets enhanced, thus we are incrementing the gain and improving the Signal-to-Noise Ratio (SNR). The flow diagram (**Fig. 3.14**), the code (**List. 3.7**) and the output plots (**Fig. 3.13**) are shown next.

```
"""new definition of the function"""
def generateRandomNoisewithLimits(N,ll,lu): #points, lower
    limit of the random number, upper limit of the random
    number
    ns = []
    for i in range(0,N):
        ns.append(rm.uniform(ll,lu))  #-1<number<1
    return ns

N=16 #----------------N=16 points
ns = []
ll=-3 #lower limit for the magnitude of the random noise
lu=5  #upper limit for the magnitude of the random noise

ns = generateRandomNoisewithLimits(N,ll,lu)

#generating signal and time vector
s = []
tn = [] #discrete time vector

for n in range(0, N):
    tn.append(n)
    s_ = mt.sin((2*mt.pi*2)*n/N)
    s.append(s_)

#mixing signal (+)
ms = []
for n in range(0, N):
    ms.append(ns[n]+s[n])

Xr_l, Xi_l, Xm_ms, Xp_l = DFTbyDef(ms,N)

#-----------------------------------------------------------
```

```
#----------Now we try N=64 points with the same noise
Nnew=64
#-----
nsmod = []
nsmod = generateRandomNoisewithLimits(Nnew, ll ,lu)

#generating signal and time vector
smod = []
tnmod = [] #discrete time vector

for n in range(0, Nnew):
    tnmod.append(n)
    s_ = mt.sin((2*mt.pi*2)*n/N)
    smod.append(s_)

#mixing signal (+)
msmod = []
for n in range(0, Nnew):
    msmod.append(nsmod[n]+smod[n])

xM_Msmod=[]
Xr_1, Xi_1, Xm_msNmod, Xp_1 = DFTbyDef(mss,Nnew)

halfN = int((Nnew/2))
```

Listing 3.7: DFT Processing gain

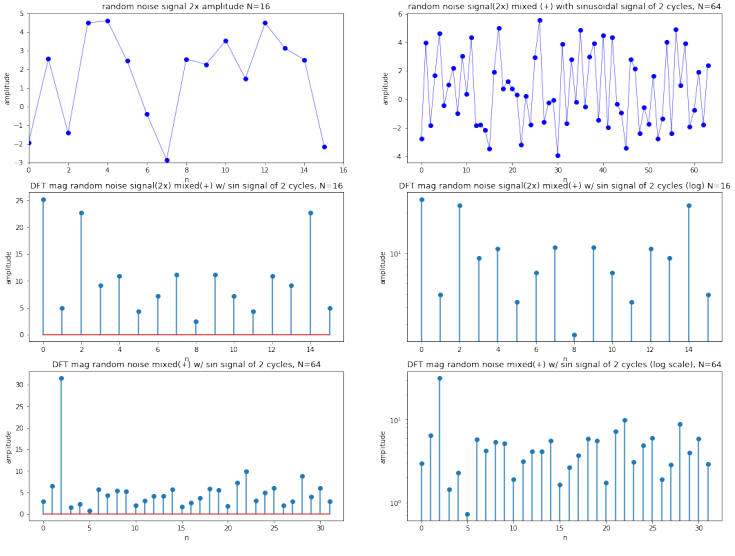

Figure 3.13: DFT Processing gain in logarithmic scale

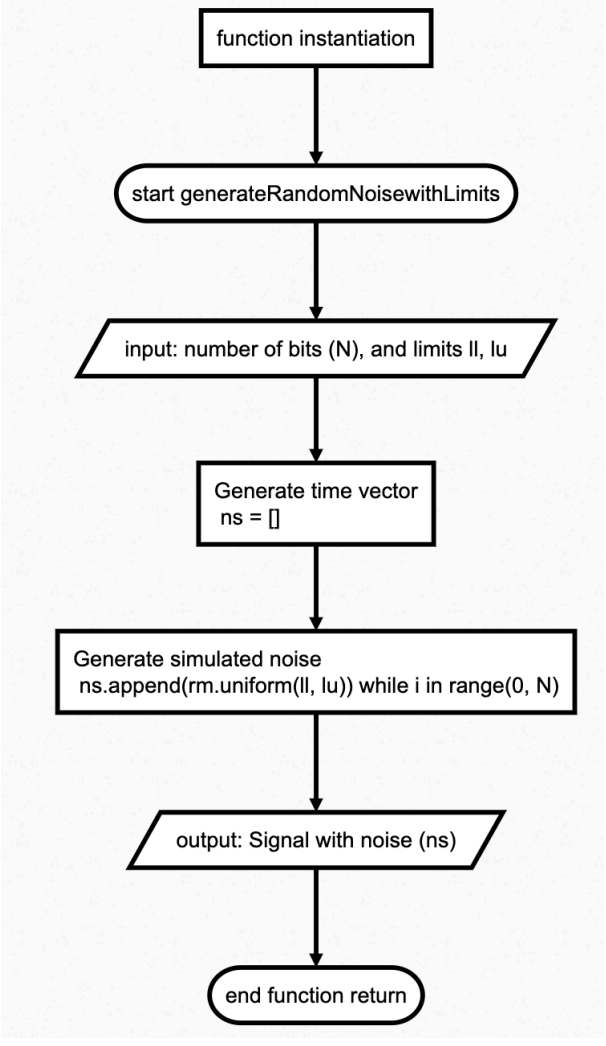

Figure 3.14: Flow diagram for processing gain

The Finite Impulse Response Filter was implemented to demonstrate its functions, but in the implementation, open source tools are used to generate the filter's coefficient. The flow diagram (**Fig. 3.16**), the code (**List. 3.8**) and the output plots (**Fig. 3.15**) are shown next.

```
ampl = [10, 22, 24, 42, 37, 77, 89, 22, 63, 9]
min_index = []
avg_ampl = []
c=0

for c in range(0,len(ampl)):
    min_index.append(c+1)
    if c <= 3: #bc the first valid result is until we get 5 samples, would be the index 4, but compensating the zero-index
        avg_ampl.append(0)
    else:
        avg_ampl.append((ampl[c-4]+ampl[c-3]+ampl[c-2]+ampl[c-1]+ampl[c])/5)
```

Listing 3.8: Basic FIR Implementation

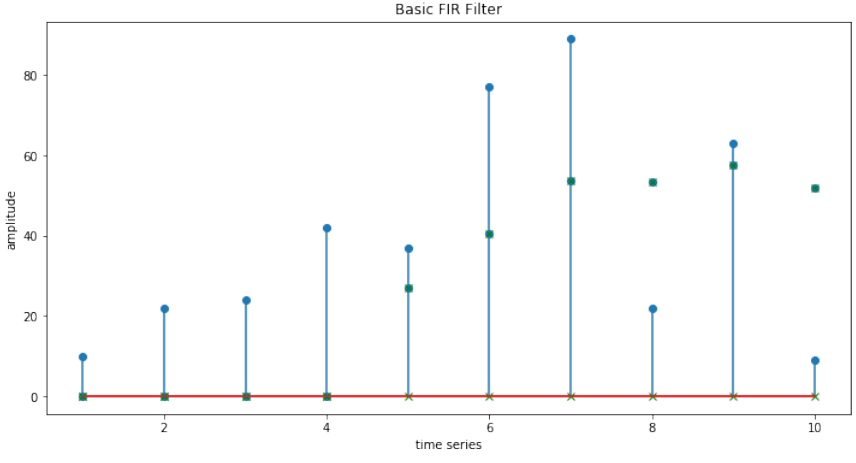

Figure 3.15: FIR Filter Implementation

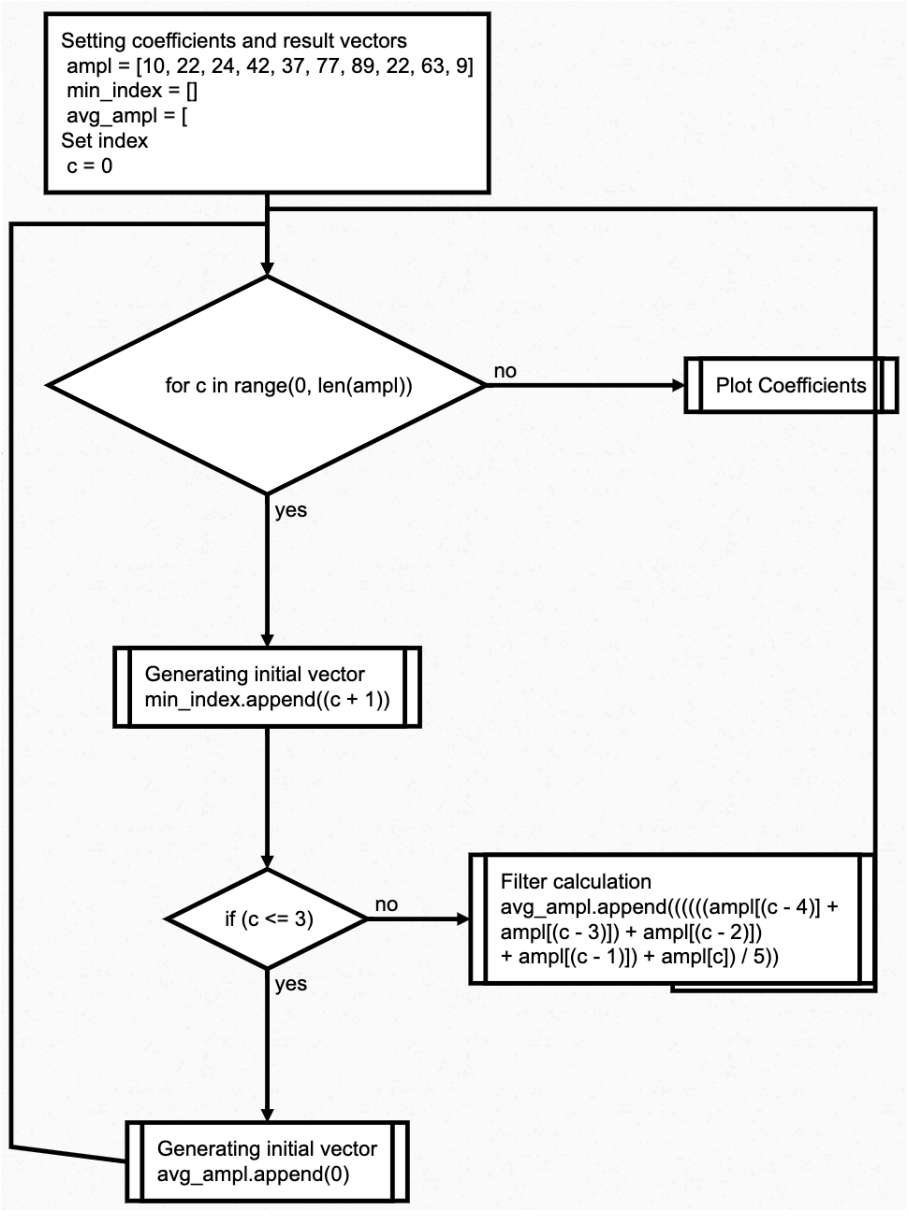

Figure 3.16: Flow diagram of the FIR Filter Implementation

3.2.3 Physical Implementation

Analog Stage

For the analog stage, we need a transducer element, a filter, an amplifier and some high voltage protection, so a commercial device was selected in order to save the time and cost associated to design the PCB for this stage. The device used is **Sparkfun Electret Microphone Breakout BOB-12758** (shown in **Fig. 3.17**). The advantages that were taken in account to use this device are:

1. Integration ease
2. reduced BOM
3. reduced form factor
4. reduced cost
5. Open source schematic and layout

There are many other alternatives in the so-called maker market like the **Electret Microphone Amplifier MAX4466** that is very similar but at the time of the acquisition of equipment wasn't available.

Figure 3.17: Electret Microphone Breakout

This board includes a mic preamplifier with a gain of 60, based in the OPA344 Op Amp and works in the range from 2.7 V to 5.5 V, making the integration with the FPGA board seamless.
The schematic of the amplifier circuit in this board is shown in **Fig. 3.18**:

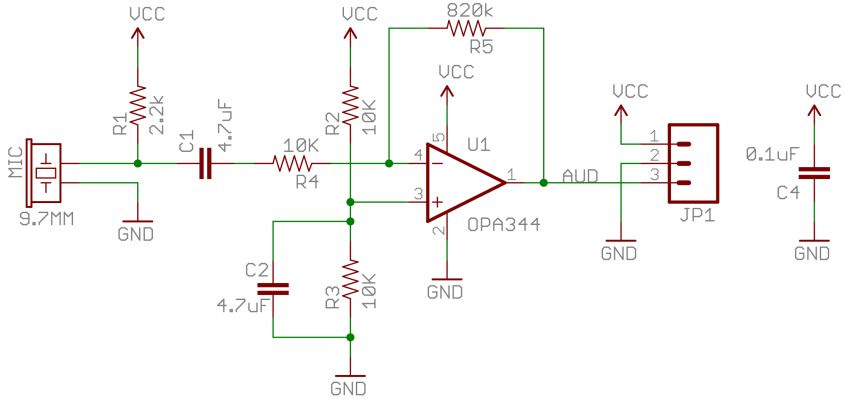

Figure 3.18: Electret Microphone Breakout Schematic

Linear Array

A simple mechanical structure was built to hold the 3 amplified microphones boards for air-based tests. See **Fig. 3.19**.

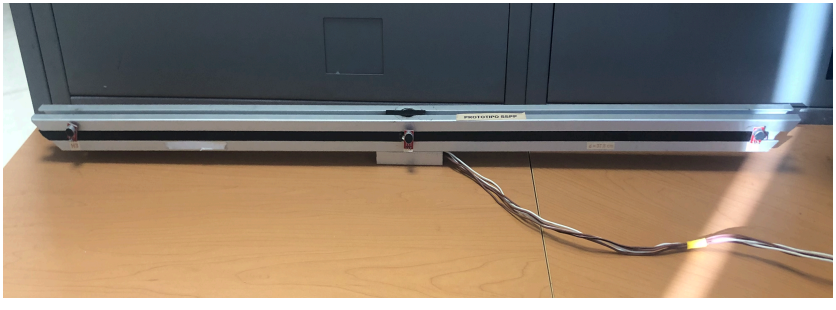

Figure 3.19: Air-based model of the Linear Array

Analog-to-Digital Converter

The last part of the analog stage is the ADC; the selected device is the **Pmod AD1** board (shown in **Fig. 3.29**) from **Digilent** manufacturer, which contains an **AD7476** converter from **Analog Devices** manufacturer and it was selected following these considerations:

1. Reduced form factor
2. Ease of mating with FPGA board
3. Reference design availability
4. author previous experience

The technical characteristics of this converter are:

- **Sampling Rate:** 1 MSPS.
- **Resolution:** 12 bits.
- **Range:** 0 to 3.3V.
- **Communications:** SPI.

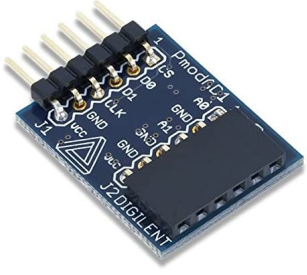

Figure 3.20: Pmod AD1

At the beginning, it was considered the use of Vivado HLS tools (C/C++ programming to describe hardware modules) to build the processing system, but it was due to two main factors that the author took the decision to use a different approach:

1. The high complexity of this tool
2. the lack of technical information available.
3. the expense of the license for commercial uses of the designs

3.3 Architecture Implementation

The full proposed architecture consists in a mixed implementation of HDL modules and embedded software, and it is described as follows: the beamformer which generates the control signals for the ADC and takes the digitalized conditioned signal that comes in first place from the transducers; and outputs the signals that represent the bearing of the source, then gets through a Lowpass filter, then it feeds a Fast Fourier Transform block and gets communicated to the firmware in the processor to output the results in UART protocol to a graphic interface within a console. This proposal is depicted in the block diagram in **Fig. 3.21**, in which the blue blocks represent HDL modules within the FPGA, also within the yellow block that represents the Processor there is the pink block that represent the embedded firmware, finally, the graphic interface block, also in pink represents the software running in the green color block of the console:

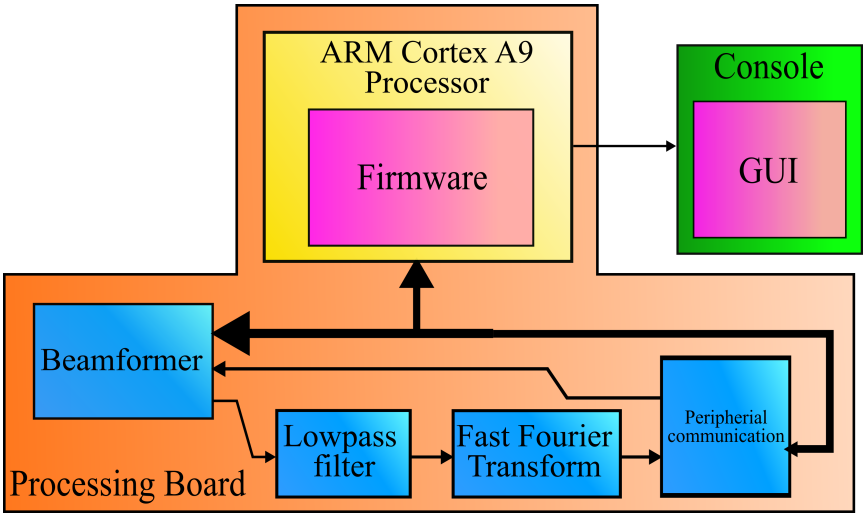

Figure 3.21: Proposed Implementation

The function of this blocks are fully described in the next section.

3.3.1 Beamformer

Due to its crucial role, the Delay-Sum beamformer is firstly described. It is implemented within the FPGA (see **Fig. 3.22**), the blue blocks are HDL modules, and the red background represents this whole block in the main system architecture:

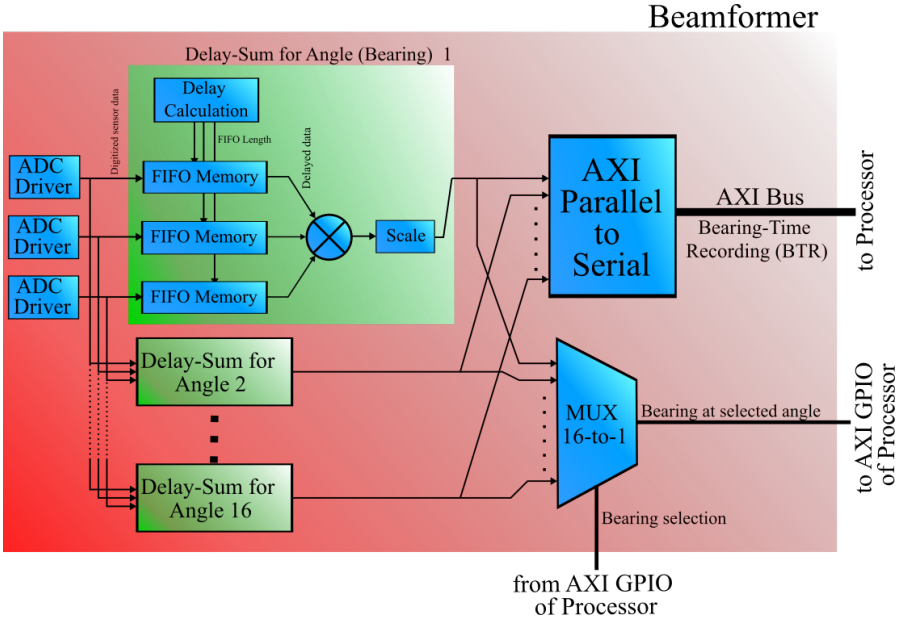

Figure 3.22: The beamformer design

ADC Controller

This block is the Pmod AD1 Reference Component, it generates the control signals for the AD7441 ADC and obtain the sampled 12-bit positive integer data in the form of a logic vector of a single data sample at the rate of 500 kS which represent the audio data. The sample rate was chosen due to previous working experience with this module and it's the lowest frequency that can be generated and the lowest that worked well with the converters.

FIFO Memory

Variable-Depth FIFO memory included as IP Core in the Vivado Design Suite, it takes as input the 12 bit sample from the ADC; the depth of this memory varies according the delay calculated and applies a discrete delay for a calculated amount of time, and outputs the time-delayed

sample. The output is the same 12 bit integer value but delayed in time.

Delay Calculation

It is a fixed ROM memory with the calculation the amount of delay needed for every angle. This module has 3 outputs of a 12 bit integer for all of the FIFO memories. The data with for the delays was selected to standardize the overall data length in all the design.

Sum and Norm

its task is to sum the three delayed channels and then normalize by 12 bits (the ADC sample width) in order to prevent possible overflow of the summing operation, resulting in a normalized coherent 12 bit single data output.
The Delay calculation, FIFO Memory and Sum and Norm blue blocks, forms the Delay Sum for Angle N green block, that is repeated 16 times across the implementation.

Multiplexer

This used for selecting the desired angle in which the LOFAR Analysis will be done, it takes as inputs each of the 16 channels (12 bit integer) and a control signal that is a 32 bits (only the 4 lower bits are used) from the ARM microprocessor and output the desired angle channel.

Serializer

All the 16 stages are connected to an independent 12 bits integer input chanel of the Parallel-to-Serial block whose output is the 16 channels serialized in the 32 bit wide AXI Stream interface (data bus and control signals described in **Section 3.4.2**), this is for the ARM processor to read the result, which is a space-sampled version of the wavefront, also known as the BTR vector.
The selected data width used across the design was intended to standardize the overall data length in all the design. The reasons to use the

positive integer is to avoid the hassle associated with floating point and signed data representation.

3.3.2 Communications

The main output of the processing is sent via the serial-over-USB connection (see **Fig. 3.23**) between the FPGA board and the display CPU (which hosts the graphical interface program), the format of the serial outputs minimally meets the NMEA 0183 communication standard as explained in the first chapter.
All data is transmitted in the form of sentences. Only printable ASCII characters are allowed, plus CR (carriage return) and LF (line feed). Each sentence starts with a "$" sign and ends with <CR><LF>. In the case of Talker Sentences. The format for the talker sentence is: **$ttsss,d1,d2,....<CR><LF>**. The first two letters following the $" are the talker identifier. The next three characters (sss) are the sentence identifier; in this case, **SS** was used, it stands for **Sounder Scanning** in order to meet the standard, then followed by a number of data fields separated by commas, that in this case are, the output of the processing, and terminated by carriage return/line feed.

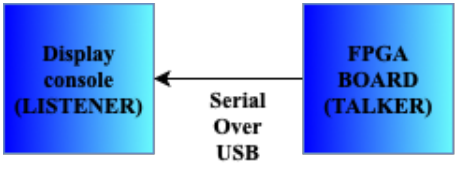

Figure 3.23: NMEA 0183 compliant communications

3.4 FPGA Implementation

The digital architecture have been designed as presented in **Fig. 3.24**:

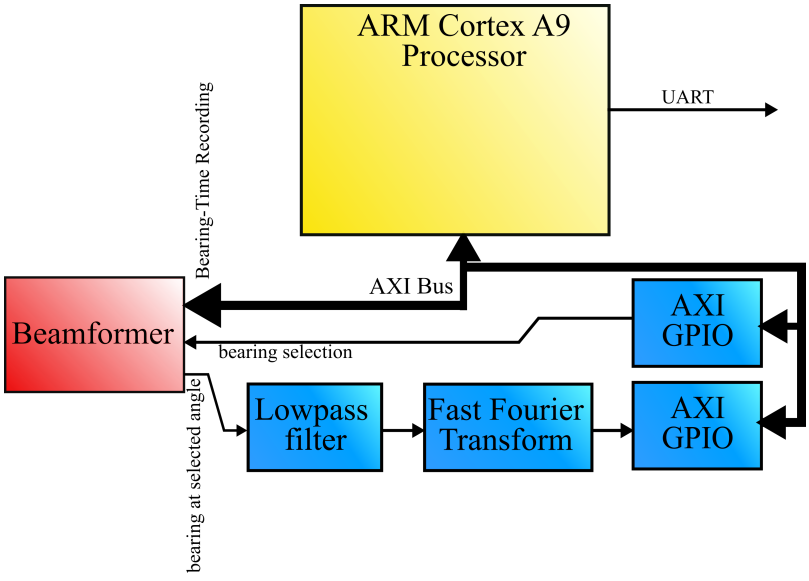

Figure 3.24: Full architecture of the system

This architecture will be implemented within the FPGA (Programmable Logic of the Zynq device), this mixed architecture uses user HDL modules and open source IP Cores.

3.4.1 LOFAR

Lowpass Filter

This a HDL module obtained using the pyFDA open-source tool for calculating the coefficients and implementing the filter in Verilog HDL language, the inputs and outputs are 12 bits wide logic vector. The selected frequency was 20 KHz in order to work with the audio spectrum.

Fourier Transform

A HDL module block that calculates the Fourier Transform, using the Fast Fourier Transform Algorithm in a Pipelined fashion with 128 bits, and an output of a 12 bits wide logic vector

AXI GPIO

This HDL block is provided as IP Core and its function is to communicate the Processor with the custom HDL blocks in the form of a Memory maped device. Its inputs are a 12 bits wide logic vector that gets assigned and hexadecimal memory direction and gets translated into a AXI standard 32 bits output, this way, it is possible to read the values from the FPGA logic and get them in a

3.4.2 Used HDL Hardware

ADC Controller

Provided by the board manufacurer Digilent, it is used to control the ADC on the Pmod, setting the sample frequency, and getting the result of the conversion

FFT Core

An arbitrary-sized open source Fast Fourier Transform Core generator[17], that works one sample in per clock. This tool comes in the form of a console program that gets configuration parameters and generates, a top Verilog file with other Verilog blocks that, as a whole, implements the pipelined FFT with the previous specified parameters,[18] Parameters used for generating the Pipelined FFT core:
fftgen -a info -c 2 -f 64 -m 12 -k 2 -n 12 -p 10
These parameters are explained below:

-a hdrname Create a header of information describing the built-in parameters, useful for module-level testing with Verilator

-c cbits Causes all internal complex coefficients to be longer than the corresponding data bits, to help avoid coefficient truncation errors. The default is 4 bits longer than the data bits. *The value used in the design is 64 bits*

-f size Sets the size of the FFT as the number of complex samples input to the transform. No default value, this is a required parameter.

-k Sets number of clocks per sample, used to minimize multiplies. *The value used in the design is 2 bits*

-m mxbits Sets the maximum bit width that the FFT should ever produce. Internal values greater than this value will be truncated to this value. The default value grows the input size by one bit for every two FFT stages. *The value used in the design is 12 bits*

-n nbits Sets the bitwidth for values coming into the (i)FFT. The default is 16 bits input for each component of the two complex values into the FFT. *The value used in the design is 12 bits*

-p nmpy Sets the number of hardware multiplies (DSPs) to use, versus shift-add emulation. The default is not to use any hardware multipliers. *The value used in the design is 10*

In order to export the core, we must export the files contained in core directory/sw/fft-core. In order to get the core working in Vivado, the *.hex files must be renamed to the extension *.dat. It is needed to change all the references to the *.hex file, to the *.dat in the fftmain.v and fftstage.v files. The line 76 of the fftmain.v and ifftmain.v must be chaged from *'default nettype none* to *default nettype wire* in order to run the testbench located in core directory/bench

FIR Filter Core

The HDL was generated used the **PyFDA** open source filter design in Python, with the previously proposed values for the Cutoff Frequency (20 KHz) and number of taps (16). The main window of the program and the generating filter window can be seen in **Fig. 3.25** and **Fig. 3.26**, respectively.

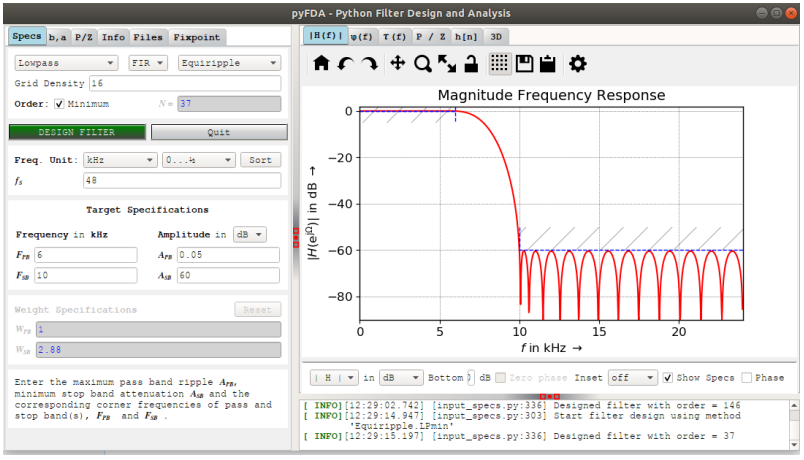

Figure 3.25: PyFDA Tool main window

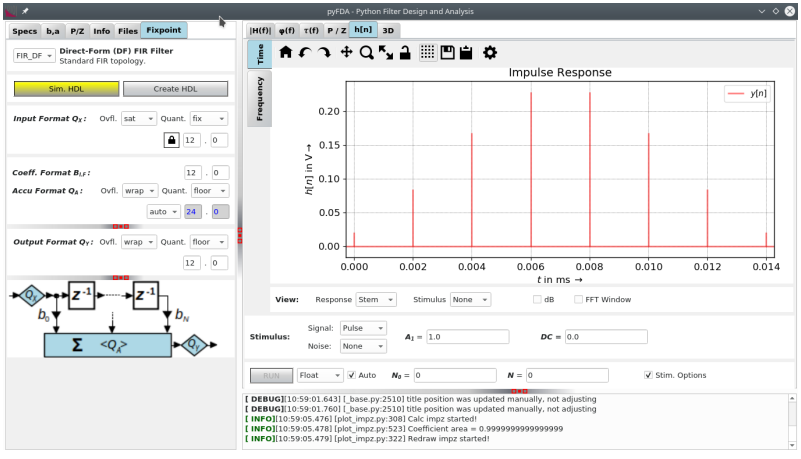

Figure 3.26: PyFDA Tool: filter generating window

3.4.3 Custom HDL Hardware Implemented

AXI Serializer

In order to get the location of the sound source in the simplest possible way, we must implement the serialization of the parallel outputs of the

ADC Controller, that was achieve using a AXI Master controller for communicate directly to the CPU in the Zynq device. The waveforms wanted from this module are depicted in the time diagram in **Fig. 3.27** and the full code of this module is presented in **List. 3.9**.

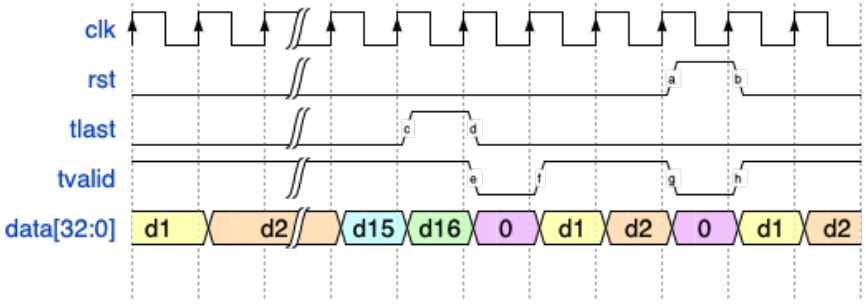

Figure 3.27: Input and Output waveforms for the AXI Serializer

```
1  --this serializer have an Master AXI Stream interface
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use IEEE.NUMERIC_STD.ALL;
5
6  entity SerializerAXI is
7      generic(
8          --from serializer
9          numbits     : integer := 12; --ADC result wide = 12
10         numchn      : integer := 16;
11         --from AXI stream interface
12         NUMBITSout : natural := 32; --added line 15/04/20
       salida AXI
13         numintime   : natural := 12  --to configure the
       integration time
14 );
15     port(
16         --Serializer----------------
17         --inputs
18         clk :    in std_logic;
19         rst:     in std_logic;
20         din0:    in std_logic_vector(numbits-1 downto 0);
21         din1:    in std_logic_vector(numbits-1 downto 0);
```

```
22          din2:    in std_logic_vector(numbits-1 downto 0);
23          din3:    in std_logic_vector(numbits-1 downto 0);
24          din4:    in std_logic_vector(numbits-1 downto 0);
25          din5:    in std_logic_vector(numbits-1 downto 0);
26          din6:    in std_logic_vector(numbits-1 downto 0);
27          din7:    in std_logic_vector(numbits-1 downto 0);
28          din8:    in std_logic_vector(numbits-1 downto 0);
29          din9:    in std_logic_vector(numbits-1 downto 0);
30          din10:   in std_logic_vector(numbits-1 downto 0);
31          din11:   in std_logic_vector(numbits-1 downto 0);
32          din12:   in std_logic_vector(numbits-1 downto 0);
33          din13:   in std_logic_vector(numbits-1 downto 0);
34          din14:   in std_logic_vector(numbits-1 downto 0);
35          din15:   in std_logic_vector(numbits-1 downto 0);
36          intime: in std_logic_vector(numintime-1 downto 0); --
        to modify the integration time
37          ------------------------------------------
38          --AXI Interface++++++++++++++++++++++++++++++
39          -- axi stream ports
40          m_axis_tvalid : out std_logic;
41          m_axis_tdata : out std_logic_vector(NUMBITSout-1
        downto 0);
42          m_axis_tstrb : out std_logic_vector(3 downto 0);
43          m_axis_tlast : out std_logic
44          --++++++++++++++++++++++++++++++++++++++++++++++
45          );
46  end SerializerAXI;
47
48  architecture bhv of SerializerAXI is
49    --Serializer signals
50    signal internal: std_logic_vector(numbits-1 downto 0);
51    -- AXI Stream internal signals
52    signal tvalid          : std_logic := '0';
53    signal tlast           : std_logic := '0';
54    signal padding         : std_logic_vector(NUMBITSout -
        numbits - 1 downto 0) := (others => '0'); --modified
55
56
57
58    begin
59    --connections of AXI Interface
60    m_axis_tstrb   <= (others => '1');      -- byte enables
        always high
61    m_axis_tvalid  <= tvalid;
```

```
62    m_axis_tlast   <= tlast;
63
64    --Serializer arch-----------------------------------------
65      process(clk)
66          variable count: integer range 0 to numchn+1; --+1 for
        test
67          variable intime_val : integer range 0 to 2**numintime
      +1 := 0; --for the integration time control
68          variable conitime   : integer range 0 to 2**numintime
      +1 := 0;
69
70      begin
71      intime_val := to_integer(unsigned(intime));
72          if (rst='1') then
73          --reset counters
74          count := 0;
75          intime_val := 0;
76          conitime := 0;
77          m_axis_tdata <= (others=>'0');
78
79          elsif (clk'event and clk='1') then
80              count := count + 1; --increment the counters
81              conitime := conitime + 1;
82              tvalid <= '1';
83              tlast  <= '0';
84              ----------------------------counter of AXI Signals
85              if (conitime = intime_val) then --former 15
86                  tlast <= '1';
87
88              end if;
89              if (conitime = intime_val+1) then --former 16
90                  tvalid <= '0';
91                  conitime := 0;
92                  count := 17; --let's see
93              end if;
94              ---------------------count of serializer
95              if count = 17 then
96                 count := count - 1;
97              end if;
98              if count = 16 then
99                 count := 0;
100             end if;
101
102             case count is
```

```
                when 0 =>  internal <= din0;
                when 1 =>  internal <= din1;
                when 2 =>  internal <= din2;
                when 3 =>  internal <= din3;
                when 4 =>  internal <= din4;
                when 5 =>  internal <= din5;
                when 6 =>  internal <= din6;
                when 7 =>  internal <= din7;
                when 8 =>  internal <= din8;
                when 9 =>  internal <= din9;
                when 10 => internal <= din10;
                when 11 => internal <= din11;
                when 12 => internal <= din12;
                when 13 => internal <= din13;
                when 14 => internal <= din14;
                when 15 => internal <= din15;
```

Listing 3.9: VHDL Master AXI Serializer

100 KHz Clock Divider

Since we are working with signals whithin the acoustic range, we does not need very fast sampling frequencies, and the Clock Divider IP by Xilinx gets as low as 1 MHz, so its necessary to scale down this frequency, in this case to 100 KHz, the code for achieve it, is presented in **List. 3.10**.

```
----------------------------------
--100k FREQ DIVIDER
----------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;


entity div_freq_100k is
    Port ( clk_5M : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           --address: out STD_LOGIC_VECTOR(8 DOWNTO 0);
              fmuestreo_500k : out  STD_LOGIC);
end div_freq_100k;

architecture Behavioral of div_freq_100k is
```

```
17
18 --SIGNALS
19 SIGNAL aux_clk: STD_LOGIC;
20
21 BEGIN
22 --WIRE SIGNAL
23 fmuestreo_500k <= aux_clk;
24
25   --FREQ DIVIDER
26   PROCESS(clk_5M)
27     VARIABLE count: NATURAL RANGE 0 to 60;
28   BEGIN
29     IF (rst = '1') THEN
30     count:= 0;
31     aux_clk <= '0';
32     ELSE
33       IF (clk_5M'EVENT AND clk_5M='1') THEN
34         count:= count + 1;
35         IF (count = 50) THEN
36           aux_clk <= NOT aux_clk;
37           count:= 0;
38         END IF;
39       END IF;
40     END IF;
41   END PROCESS;
42
43
44 end Behavioral;
```

Listing 3.10: VHDL code for the clock divider

3.4.4 Processor Firmware

The firmware is located in the ARM Processor within the Zynq device, and it is written and tested using the **Vivado SDK** as shows in **Fig. 3.28**

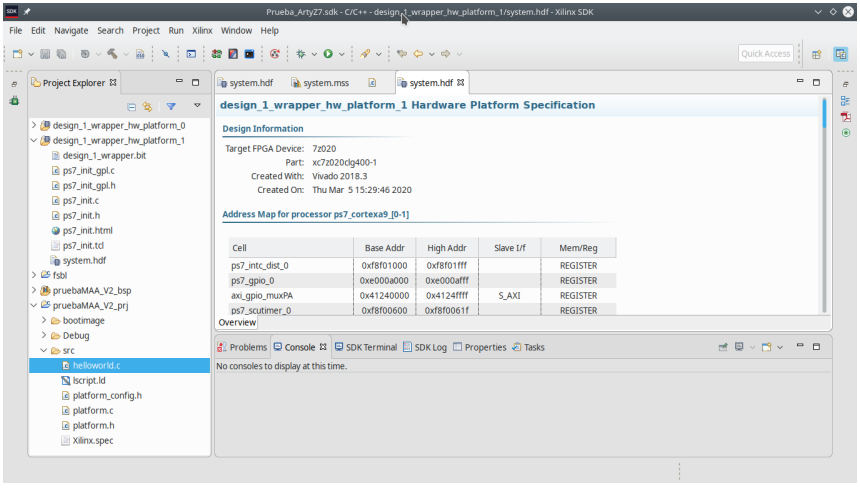

Figure 3.28: SDK open with the main firmware file

3.4.5 Graphical Interface Development

The Graphical Interface software was designed in the **Qt Designer** under the pyQt 5 environment, and gives the ability to graphically see the results of the processing. This could be seen in **Fig. 3.29**

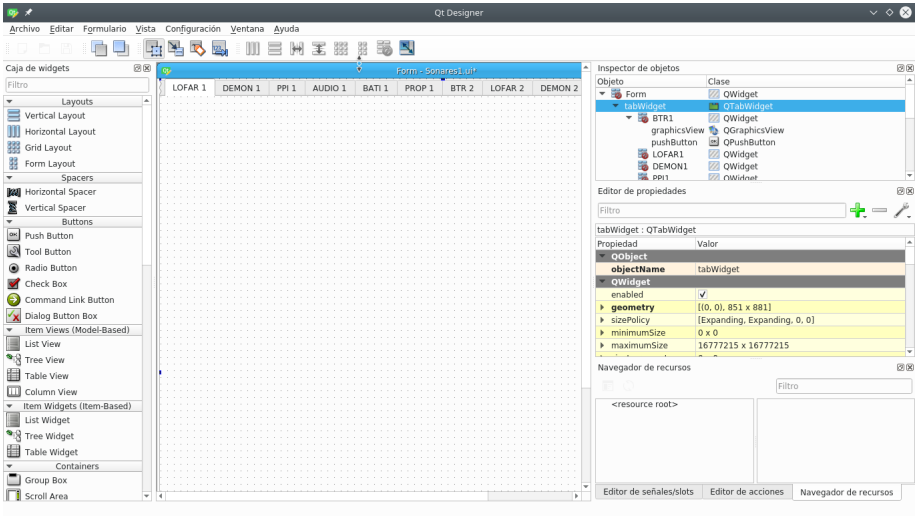

Figure 3.29: Qt Designer showing the main form of the GUI

Chapter 4

Results

4.1 Custom HDL Hardware Implemented

These section presents the result of the simulations of the HDL developed.

ADC Controller

In **Fig. 4.1** the ADC controller test is presented, in this case this module was instantiated 16 times in order to check for coherency. The ADC Controller HDL receives a 500 KHz signal from a the HDL clock divider which sets the sampling rate (second signal from top to bottom in **Fig. 4.1**), and gives the digitized instant value as a 12-bit integer (from third to the last signal, from top to bottom in **Fig. 4.1**), and a logic done flag that indicates the conversion is over (first signal from top to bottom in **Fig. 4.1**), this is used for synchronize the next modules. During testing, control signals are generated to drive the ADC, and 16 input channels (signals *AD1* to *AD16*) of the ADC (8 Pmod boards) were wired to 1.65 VDC in order to get approximately 2048 in the conversion result, since the ADC resolution is 12 bits, the maximum result is 4096. Thus we can validate the correct operation of this module.

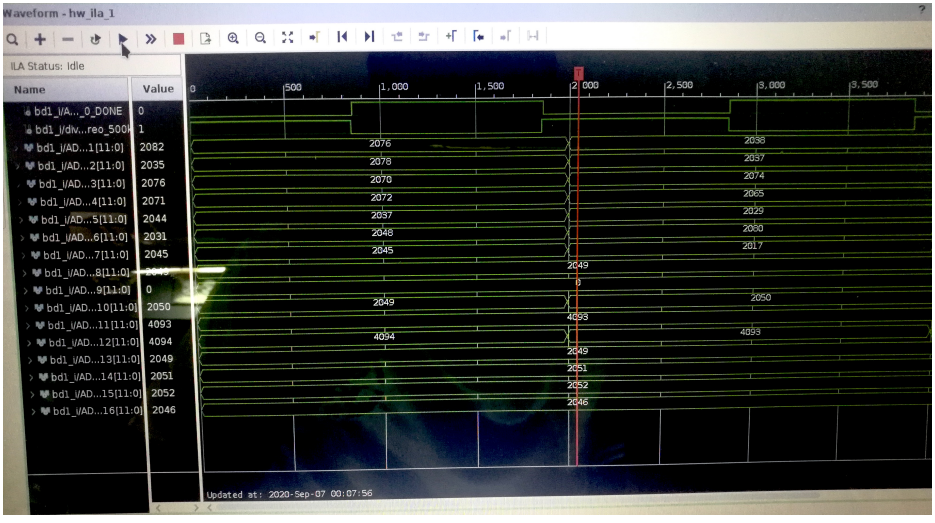

Figure 4.1: ADC conversion result

4.1.1 FIFO Delay of the Beamformer

In **Fig. 4.2** the FIFO Delay test is presented.This FIFO memory receive an input data coming from the is an IP Core integrated in Vivado and it has the role of making the discrete time delay of the signal in the Beamformer. In the simulation it can be seen that the input *DATAINPUT* test value is at the *output* signal of the module.

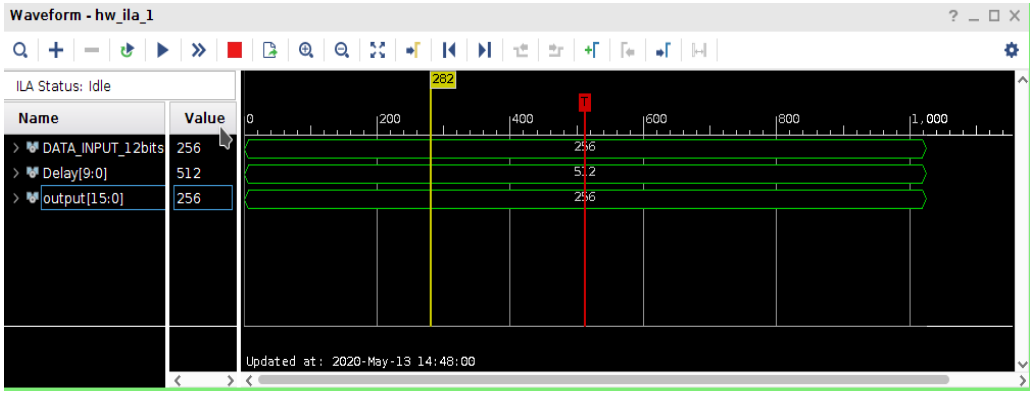

Figure 4.2: Validation of the FIFO Delay

4.1.2 AXI Serializer

The serializer was tested with constant values and simulated using the GHDL Simulator and GTKWave Wave viewer, results can be seen in **Fig. 4.3** and in **Fig. 4.4** is shown the serializer implemented in a Vivado Block Diagram. We could compare this results with the time diagram depicted in **Fig. 3.27**. The simulation consists of 16 12-bit constant numbers *dinN* signals that are present in a serial fashion at the output of the module *maxistdata*, also the control signals of the AXI protocol are generated *axistlast* and *axistvalid* by the module.

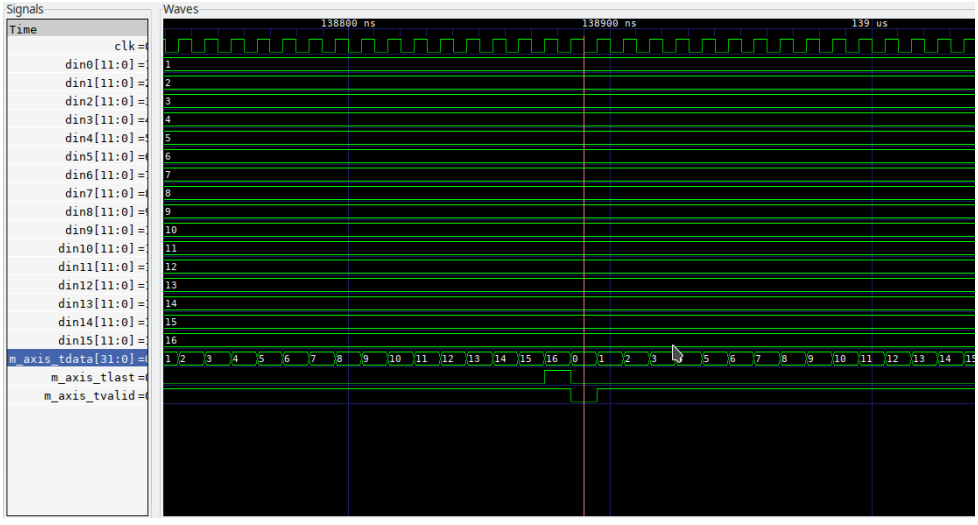

Figure 4.3: Validation of the AXI Serializer

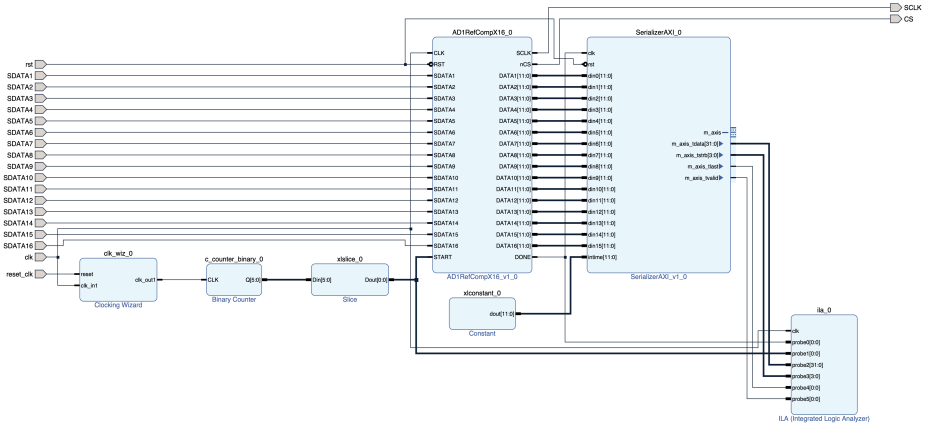

Figure 4.4: 16 channel Serializer

4.1.3 Full Beamformer

The Vivado Block Diagram Implementation of the beamformer is depicted in **Fig. 4.5**:

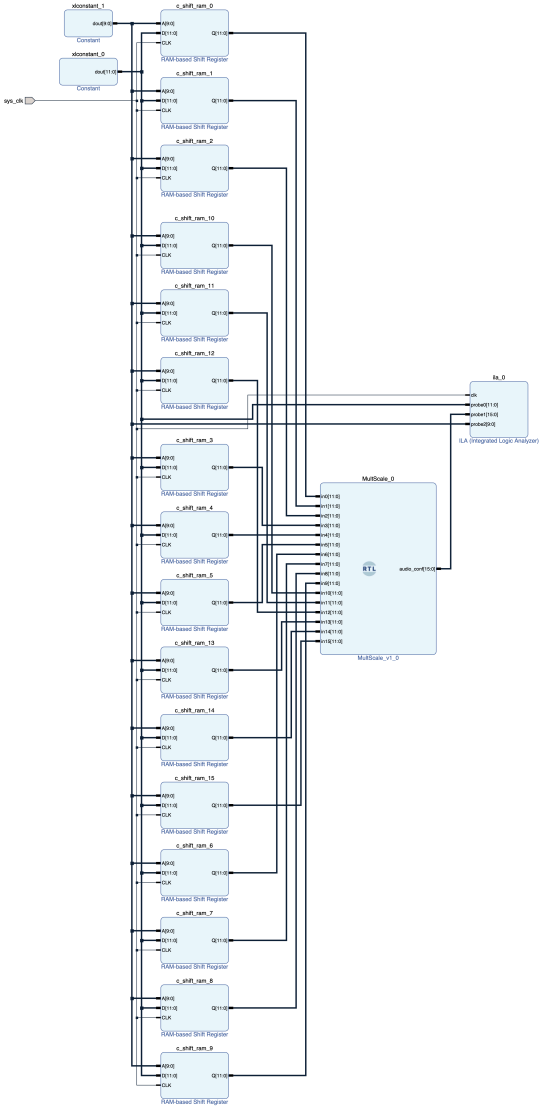

Figure 4.5: Full Beamformer Test

4.2 Third-party HDL Hardware Used

These section presents the result of the simulations of the third-party HDL used.

4.2.1 Lowpass filter

The testbench simulation of the Lowpass filter generated core in **pyFDA** was instantiated and then run in the **ModelSim** Student Edition Simulator, and the results are presented in **Fig. 4.6**, in this figure we could see a couple of 12-bit input data *datain*, the flag of data readiness *dataready* and the *dataout* that shows the result of the filtering operation:

Figure 4.6: Lowpass filter simulation results

4.2.2 FFT core

The testbench simulation of the Double clock FFT generated core, was instantiated and then run in the **ModelSim** Student Edition Simulator, and the results are presented in **Fig. 4.7**, in this figure we could see a couple of 12-bit input data *isample*, the flag of synchronization *osync* and the *oresult* that shows the result of the filtering operation:

Figure 4.7: Double-clock FFT simulation results

4.3 Physical Integration

The following diagram **Fig. 4.8** illustrates the final physical implementation of the system, which presents not much differences between the diagram in **Fig. 1.2**, and the architecture programmed and running in the board is depicted in **Fig. 4.9**. In the final test, a loudspeaker emits a sound at 1 KHz, (acting as the target) in front of the mechanical structure (whose angle system is shown in **Fig. 4.9**), then the signal is processed and the results are sent to the laptop and presented via the graphical interface.

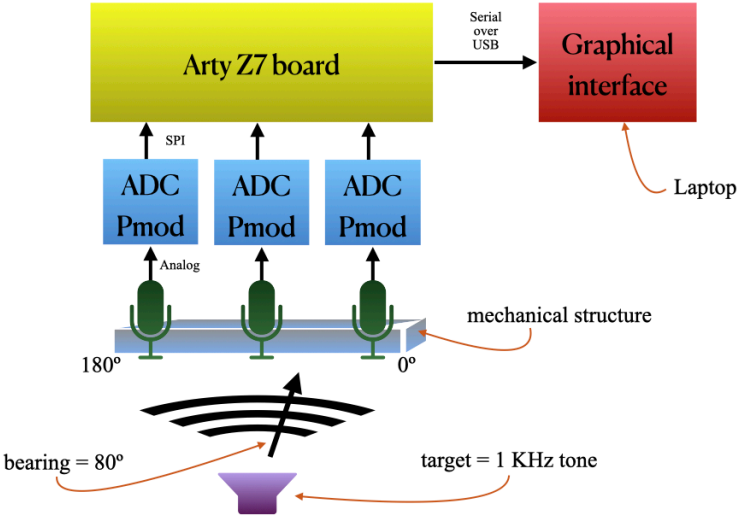

Figure 4.8: Final physical implementation diagram

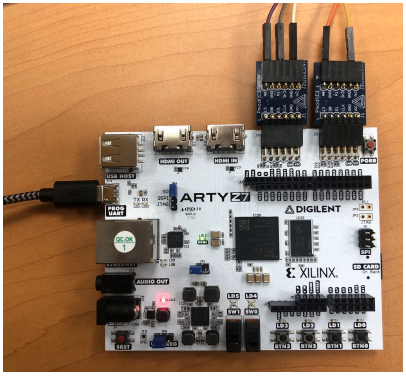

Figure 4.9: System running on board

4.3.1 Communications Tests

Connecting the FPGA Board to the Display console, via serial over USB and displaying the NMEA sentences is shown in **Fig. 4.10**. This test is made before launching the graphical interface, in order to probe the communication between this two devices.

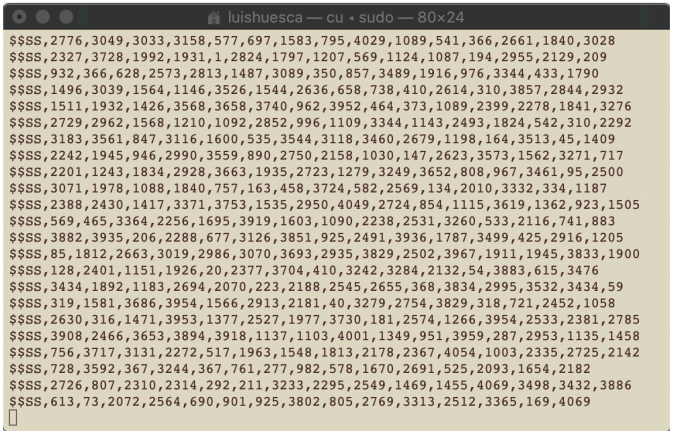

Figure 4.10: Successful Communication

4.3.2 Graphical interface integration and results displaying

Finally, the developed software for the graphical interface results are shown, given the results seen in **Fig. 4.11** for the LOFAR, and **Fig. 4.12** for the BTR. The results for each one of the analyses are presented in its corresponding figure. The **Fig. 4.11** depicts the results window of the LOFAR analysis, which is the spectral analysis showing the main component and its harmonics; the Y-axis is the 12-bits wide discrete amplitude of the frequency bins and the X-axis is the 11 discrete frequency bins. Similarly, the **Fig. 4.12** depicts the results window of the BTR analysis, which is the bearing relative to the linear array, showing the maximum amount of energy and its arriving angle; the Y-axis is the 12-bits wide discrete energy of the discrete angles, the X-axis is the 16 discrete angles vector.

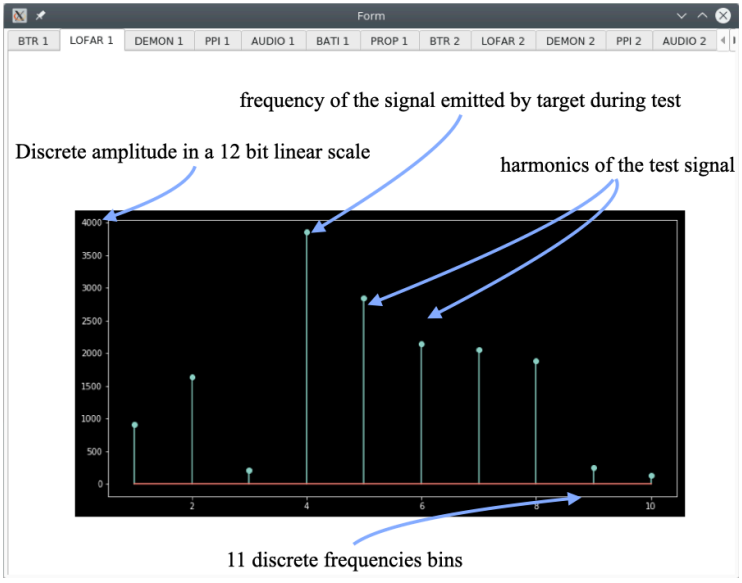

Figure 4.11: LOFAR results displaying

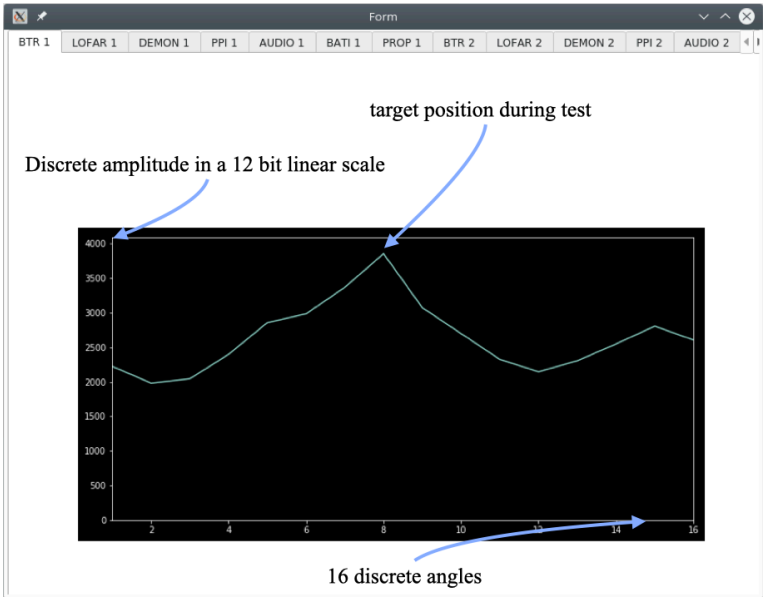

Figure 4.12: BTR results displaying

Conclusion

Beside the steep learning curve to follow with the new FPGA technology, even when there is a previous background, it is worth the sacrifice due to the relatively straight-forward implementation of the algorithms in the form of custom hardware. A thing for sure, is that the new technology provided by the FPGAs manufacturers gives the system designer a broader palette of tools for implementing signal processing algorithms. Another important fact to take in account, is that nowadays, there are several open source options (such as filter design assistants, spectral analysis tools in both HDL modules or software libraries) for reduce time in the development of digital signal processing schemes.

In terms of the scope of this work, a full but simple architecture could be implemented in a Xilinx Zynq device, but it is clear to see that in order to get the most of this technology and achieve the best fitted signal processing, it is mandatory to get a little group of developers due to the inner complexity and the vast amount of techniques related to FPGA programming, HDL coding, CPU programming and embedded software coding.

Even when this work cannot be compared directly with a commercial system because its specifications differs completely and the fact that making qualitative tests of a commercial sonar system it is nearly impossible in the laboratory because the dimensions and complexity of the equipment; however, this system appears to be very simple, but it is extremely powerful because it is open to any modification and further scale, this is possible thanks to two main factors: 1) the programmable device in which is implemented, and 2) all the algorithms were developed and are visible to the user (in contrast to the extremely closed commercial systems) and this gives the ability to fully comprehend the inner workings of a similar system, and even serve as a base for future developments.

It is an original work that emerges from the previous working experience and, because of its nature, there are only a couples of developments like this in the country.

This is an original work, and it reflects merely all the experience of the author in the field of signal processing.

Futurework

There is plenty of future work, such as a real underwater implementation, achieve a very efficient algorithm for passive sonar, or a fully integrated system development in a single board; maybe another related technology development, such as the single beam echosounder and even multibeam echosounder. This subject (underwater signal processing) is still a very unknown matter in our country, but things can change.

Bibliography

[1] A. D. Waite, *Sonar for Practising Engineers*, 3rd ed. John Wiley & Sons, 2001.

[2] R. P. Hodges, *Underwater Acoustics: Analysis, Design and Performance of Sonar*, 2nd ed. John Wiley & Sons, 2010.

[3] L. Castellanos, J. Aguilar, and M. Alvarado, "A lofar and beamforming implementation in a fpga for a digital passive sonar," in *2016 13th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, Sept 2016, pp. 1–5.

[4] D. Charlot, R. Girault, and B. Zerr, "Delph-sonar: a compact system for the acquisition and processing of side scan sonar images," in *Proceedings of OCEANS'94*, vol. 2, Sept 1994, pp. II/428–II/432 vol.2.

[5] P. Graham and B. Nelson, "Frequency-domain sonar processing in fpgas and dsps," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, April 1998, pp. 306–307.

[6] S. R. S. (Editor), *Advances in Sonar Technology*, 1st ed. In Tech, 2009.

[7] J. Wang, C. Tao, and J. Jiao, "Method for high frequency sonar multi-beam matched filtering based on single-chip fpga," in *2018 International Conference on Electronics Technology (ICET)*, May 2018, pp. 157–161.

[8] J. M. d. S. Natanael Nunes de Moura, Eduardo Simas Filho, "Independent component analysis for passive sonar signal processing," in *Advances in Sonar Technology*, S. R. Silva, Ed. Rijeka: IntechOpen, 2009, ch. 5. [Online]. Available: https://doi.org/10.5772/39410

[9] Avnet, *MiniZed Hardware User Guide*, 1st ed., Avnet Inc., 06 2017.

[10] M. A. Ainslie, *Principles of Sonar Performance Modeling*, 2nd ed. Springer-Verlag, 2010.

[11] P. Chen, X. Tian, Y. Chen, and X. Yang, "Delay-sum beamforming on fpga," in *2008 9th International Conference on Signal Processing*, Oct 2008, pp. 2542–2545.

[12] G. Bienvenu, "Signal sonar processing," in *2002 14th International Conference on Digital Signal Processing Proceedings. DSP 2002 (Cat. No.02TH8628)*, vol. 1, July 2002, pp. 441–446 vol.1.

[13] W. H. Mangione-smith and B. L. Hutchings, "Configurable computing: The road ahead," 05 2000.

[14] I. McCowan, "Robust speech recognition using microphone arrays," Ph.D. dissertation, Queensland Uni- versity of Technology, 2001.

[15] J. M. d. S. Natanael Nunes de Moura, Eduardo Simas Filho, *Advances in Sonar Technology*. IntechOpen, 2009, ch. Independent Component Analysis for Passive Sonar Signal Processing.

[16] L. H. C. R. A. E. M. A. E. R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex- A9 on the Xilinx Zynq-7000 All Programmable SoC*, 1st ed. Strathclyde Academic Media, 2014.

[17] D. Gisselquist, "A generic piplined fft core generator."

[18] ——, "Double clock fft," 2018. [Online]. Available: https://github.com/ZipCPU/dblclockfft

"You are merely proving [their] technology is more advanced than ours. What has that to do with intelligence?" The Gods Themselves, Isaac Asimov, 1972.