DEVELOPMENT OF AN ACOUSTIC MODEM USING SYNTHESIZABLE MICROCONTROLLER

by **Muhammad Fahad Hassan-Mobshar**

BACHELOR'S THESIS

Bachelor's Degree in Industrial Electronics and Automatic Control Engineering

Barcelona, 4 June 2019

Director: **Jordi Cosp Vilella**

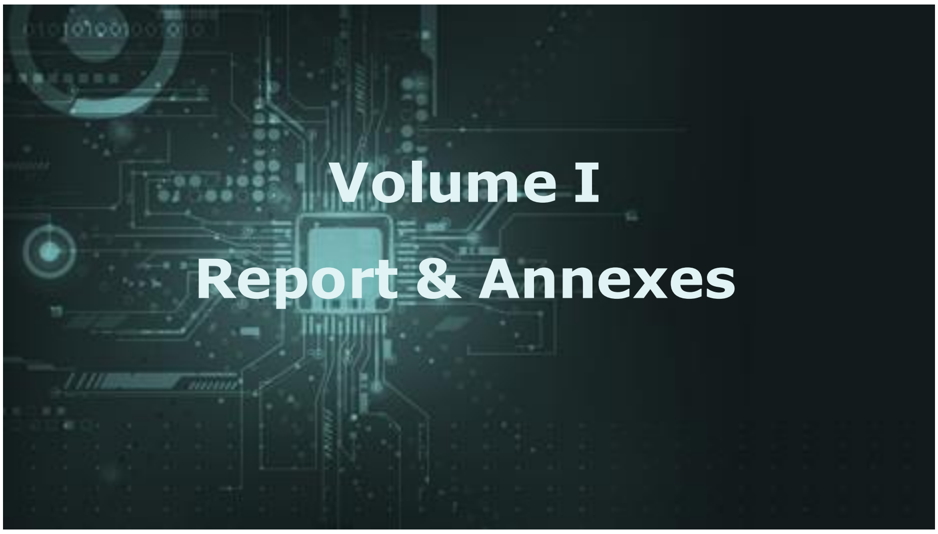
Electronic Engineering Department

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

DEVELOPMENT OF AN ACOUSTIC MODEM USING SYNTHESIZABLE MICROCONTROLLER

BACHELOR'S THESIS

Bachelor's degree in Industrial Electronics and Automatic Control Engineering



Author: M. Fahad Hassan-Mobshar

Director: Jordi Cosp Vilella

Call: June 2019

"*He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast.*" Leonardo Da Vinci

ABSTRACT

In this thesis, an acoustic modem is developed on a digital programmable device for underwater communications. The system consists of a synthesize microcontroller as well the different elements necessaries for the modulation and demodulation of the signals. That should be developed for the design.

The objective is to develop the system using a high-level hardware description language and to show the system's functioning on a flexible platform (a programable logical device) with the perspective that, in the future, it can be implemented in a personalized integrated circuit and thus obtain a compact and energy efficient system.

RESUM

En aquest treball es desenvolupa un mòdem acústic per a comunicacions submarines sobre un dispositiu digital programable. El sistema està compost per un microcontrolador sintetitzable més els diferents elements necessàries per a la modulació i desmodulació dels senyals que cal desenvolupar per al disseny.

L'objectiu és desenvolupar el sistema utilitzant un llenguatge d'alt nivell de descripció hardware i demostrar el funcionament del sistema sobre una plataforma flexible (un dispositiu lògic programable) amb la perspectiva que, en un futur, pugui ser implementat en un circuit integrat a mida i així obtenir un sistema compacte i de baix consum.

RESUMEN

En este trabajo se desarrolla un módem acústico para comunicaciones submarinas sobre un dispositivo digitar programable. El sistema consiste en un microcontrolador sintetizable y los diferentes elementos necesarios para la modulación y demodulación de las señales a desarrollar para el diseño.

El objetivo es desarrollar el sistema utilizando un lenguaje de descripción de hardware i demostrar el funcionamiento del sistema sobre una plataforma flexible (un dispositivo lógico programable) con el fin de que, en el futuro, pueda implementarse en un circuito integrado personalizado y así obtener un sistema compacto y de bajo consumo.

ACKNOLWLEDGEMENT

First of all, thanks to my director Jordi Cosp to give me the opportunity to develop an interesting project, and for their help and advice which made this thesis possible.

I would like to thanks my family; mom, dad and aunt Misbaha for their unconditional support. They had always encouraged me to continue working despite the complications. And without their help, this project would have never been accomplished.

I also like to thanks to Gerard, Toni and Marina for the good time we spent in the laboratory helping and encouraging each other.

Last but not least, I would like to thank my friend Andres for their encouragement and motivating speeches.

GLOSSARY

0xff: Number 255 in hexadecimal format and occupied 8 bits.

ADC: Analogue to Digital Converter.

AUV: Autonomous Underwater Vehicle.

FPGA: Field Programable Gate Array.

GPIO: General Purpose Input Output.

HDL: This acronym stands by Hardware Description Language.

I/O: Input Output port.

Logic-Low: In computer science logic-Low state of bit or signal is logic '0'.

Logic-High: In computer science logic-High state of bit or signal is logic '1'

Node: Communication point

Verilog: This is an example of Hardware description language. This language is one of the most used HDL.

Vivado: Program to generate design using HDL, developed by Xilinx.

INDEX

1. PREFACE

1.1. Background

In the last few decades, the progress of technology has increased in oceanographic researches. The use of Autonomous Underwater Vehicles (AUV) and sensors networks are very common in this field to study oceans activities or use for marine security. Apart from these activities, exist a strong scientific interest to develop systems to observe, monitor and track the aquatic species.

Figure 1-1: ***GUANAY II AUV developed by The Technological Development Centre for Remote Acquisition System and Information Treatment (SARTI). Extract from*** *[13]*

In the electronic engineering department of this campus, they are studying a positioned and monitoring system for aquatics species. Two different environments have been considered to locate the species; localization of the species in open areas using autonomous underwater vehicles, and localization of the species in specific areas using underwater observatories. The aim of the thesis is to locate the species, control and examine with autonomous vehicles.

Two basic methods can be used to monitor the spices: the active and the passive. In the passive method, a hydrophone is used to hear the sound produced by some species. Different species can be identified by applying filters algorithms and pattern recognition. The bioacoustics is the discipline which studies this area. In the UPC exist an investigation group specialized in this discipline; Laboratori d'Aplicacións Bioacústiques (LAB).

In the active method, the animal es identified by placing on them an acoustic transductor. Because of the versatility, size and the features, aquatic TAG is the common systems used to monetarize the species. This method offers certain advantages in front of passive systems, such as, identification of individual species or localization of species which not emits any characteristic sound. The active method is interesting since the same system can be used to locate and to monitor de species as well to position autonomous submarines, ships or different sensors.

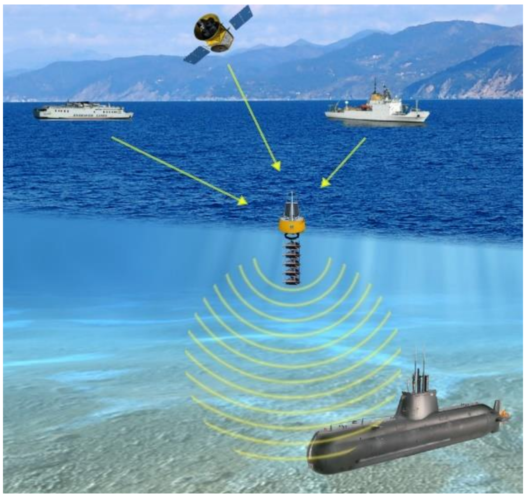


Figure 1-2: ***Underwater active method of communication. Submarines, ships and satellites communicating with node. Extract from*** *[21]*

1.1.1. Positioning algorithms

The positioning algorithms are used to calculate distances between two nodes[1]. There is two way to calculate the distance:

- *Time of flight (TOF):* TOF is the time lag between a device sends the signal and receives the answer from another device. And in case the propagation

[1] A node is either a redistribution point or a communication endpoint.

speed of the signal is known, the distance between two nodes can be calculated easily.

- *Time Difference of Flight (TDOF):* In this case, the distance is calculated using the differences of time between two received signals. The signal is generated in one point and is received in different nodes. This method is really useful when the initial broadcasting time is not known, thus the localization is determined by receiving two signals. However, all nodes have to be perfectly synchronized to know the time with precision.

1.2. Aim of the thesis

As it has mentioned, the aim of the university investigation work is to develop a system to locate and fallow underwater species using the acoustic positioned system. The active method is used to locate the species in which an acoustic TAG is incorporated in the aquatic species thus the nodes or autonomous underwater vehicles could detect them.

The aim of this thesis is to develop a modem which will be incorporated on the aquatic species. And it is going to study a device which could be compact and as small as possible. A compact device would not be annoying for the species and could allow monitoring small aquatic animals.

A microcontroller and ADC are used to create the modem which is going to implement in Field-Programmable Gate Array (FPGA) thus the modem is going to develop with Hardware Description Language (HDL). Developing the modem in FPGA will allow to emulate it, besides to know the dimension and the maximum clock speed of the device, among other features.

The main part of the thesis is to develop an application using a microcontroller in FPGA. Some characteristics are going to study to develop microcontroller in FGPA; such as how it works, how it could program, how it could connect with other parts, and how much space is required to develop it.

Furthermore, the advantages and disadvantages of using a microcontroller for this application will be discussed. Because this work is the first stage in which the functionality of the modem is observed. If the modem works properly, this could be implemented in an integrated device.

1.3. Objectives

This thesis is a part of a University Project hence the objective are chosen which are important for this part. These are principals' objectives on which the thesis is focused on.

1- The main objective of the thesis is to develop the microcontroller in FPGA and programme with a little application to know either is function correctly or not.

2- Study how to connect an Analogue-to-Digital Converter, and examine the communication between ADC and microcontroller. In case this could not work properly, the modem would not get the signal correctly.

3- Observe that the modem works properly; it obtains data from ADC, the microcontroller works correctly, and send the echo for the proper input signal and not by error.

4- The output signal is going to use to calculate the distance between the transmitter and receptor. Hence observe that the processing time is known so, to not create an error by the node or AUV to establish the exact position of the animal.

5- The animals could change their position; therefore, it is important to establish the minimum processing time so the AUV could calculate the distance accurately.

6- Observe the sources that have been used to develop the modem, and estimate the space. This is important especially for the future case when the modem is implemented in an integrated circuit.

1.4. Work planning

To achieve the objectives, previous commented, it is going to follow following planning. In the following sections, the main parts of the thesis are going to discuss and a resume of their uses.

1.4.1. Conversion of the analogue signal to digital

The ADC is used to convert the input analogic signal into a digital signal. This device is created in HDL language. Vivado is used to create all the Modem, and in this program exist an IP named XADC which is used to convert analogue data to digital.

Three weeks are planned to dedicate to this section and make the ADC working in the FPGA. In this time the XADC core is going to study; ports descriptions, XADC configuration, among other concepts.

1.4.2. Adapting the microcontroller and implementing in FPGA

The thesis consists of designing an acoustic modem using a microcontroller. The modem receives an electric signal which is converted in 16 bits digital signal.

This microcontroller has to be created with HDL though to implement in an FPGA and observe the functionality. The ARM developers have public available microcontroller Cortex-M3. This device has peripherals to connect with Vivado elements and could be synthesised in Vivado and later implement in FPGA thus to observe the functionality.

The microcontroller detects the signal signature using the Goertzel Algorithm and obtain the power of the signal. If the power is greater than the set point, the input signal corresponds to the detection signal. When this occurs, the microcontroller sends the eco of this signal. The microcontroller is programmed using C language with Keil μVision program.

This way marine animal which has the modem, will be detected. After that, the distance between the modem and the node will be calculated.

Five weeks are planned to dedicate to this section and make the microcontroller working in the FPGA.

1.4.3. Connecting ADC with microcontroller to detect the signal

In this period the modem is going to test; obtaining the analogue signal, processing with a microcontroller with Goertzel algorithm, and send the echo in the case is the input signal correspond to the desired signal.

Six weeks are planned to dedicate to this part.

1.4.4. Drafting the report

Some parts of the report are going to write during previous stages, thus not to have a lot of work in the final weeks. Moreover, the last three weeks are going to spend to draft the report.

The *Figure 1-3* shows the Gantt chart of the planning, which also includes previous works.

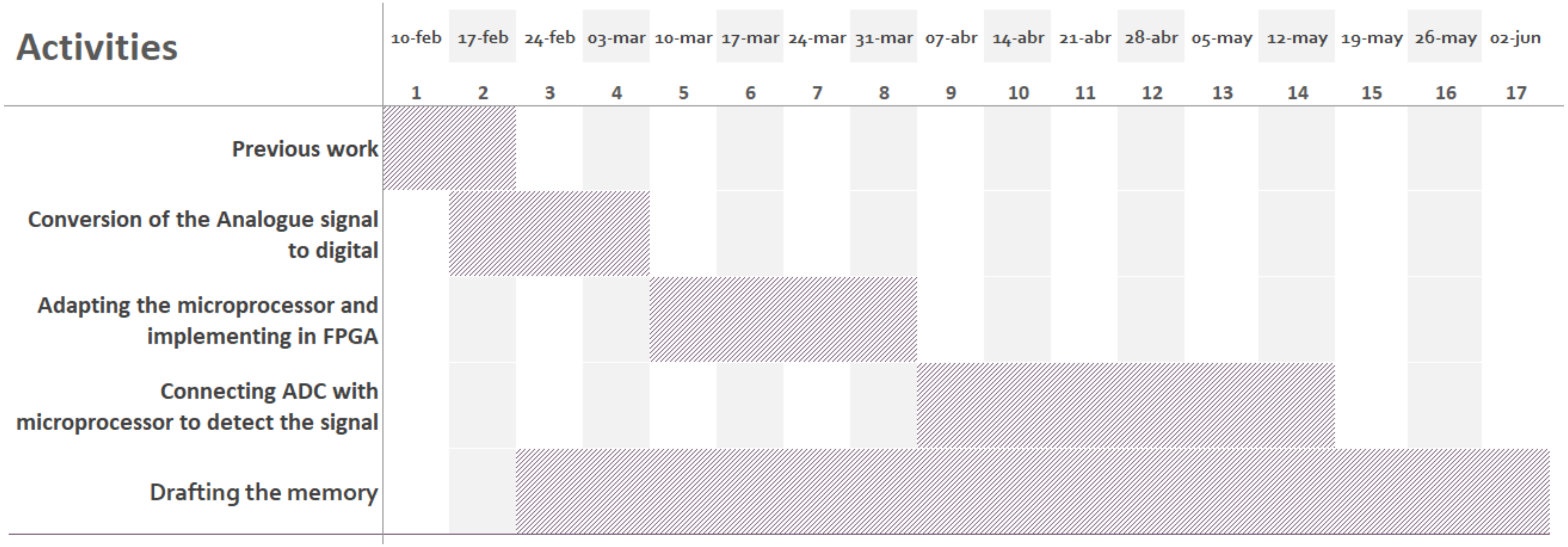


Figure 1-3: ***Thesis planning that lasted 17 weeks.***

This page intentionally left blank

2. BACKGROUND

In the introduction and through their development there is mentioned some concepts about different development languages and technologies used to develop the thesis. It is important to know some basic concepts about that. Readers who have this knowledge can skip this chapter.

This thesis has an important background based on these three concepts:

- FPGA
- C
- Verilog

The FPGA is the circuit where the modem is developed. And it will be explained the basic use of this for this application.

C is the programming language which is used to generate program file for the microcontroller. It will be explained the basic concepts that are used to create a program.

Verilog is a Hardware Description Language which is used to develop the modem. It will be explained basic concepts to understand and how to define a digital block in this language.

2.1. Field Programmable Gate Array (FPGA)

A Field Programmable Gate Array is defined as a matrix of configurable logic blocks (CLBs) connected to each other with interconnection networks which are entirely programmable. The blocks could be combinational and/or sequential.

FPGAs belongs to the family of programmable logic components. Exist diverse configurable technologies in which reprogrammable are of interest, such as: Flash, EPROM or SRAM. The FPGA is configured using Hardware description language (HDL). Two HDLs are widely used and these are VHDL and Verilog.

The generic architecture of an FPGA is shown in *Figure 2-1*. In red is the Configurable Input/Output Block, in blue the Configurable Logic Block and n black the Interconnection Network.

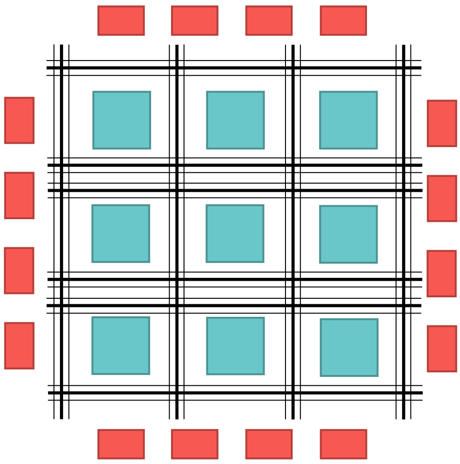


Figure 2-1: ***Generic architecture of a FPGA***

The two main FPGA manufacturers are Altera and Xilinx. For this thesis, the design is implemented in Nexys 4 DDR FPGA board using Verilog HDL developed by Xilinx. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys4 DDR to be used for a wide range of designs [1] .

The Nexys 4 DDR board is based on Artix-7 FPGA. The board offers more than fifteen thousand logic slices, each of them has four 6-inputs Look Up Tables (LUT) and 8 Flip-Flops; up to 4860 Kbits of fast block RAM, and the internal clock speed can exceed 450 MHz. The board also includes four digital Pmod ports and one analogic input port with three channels.

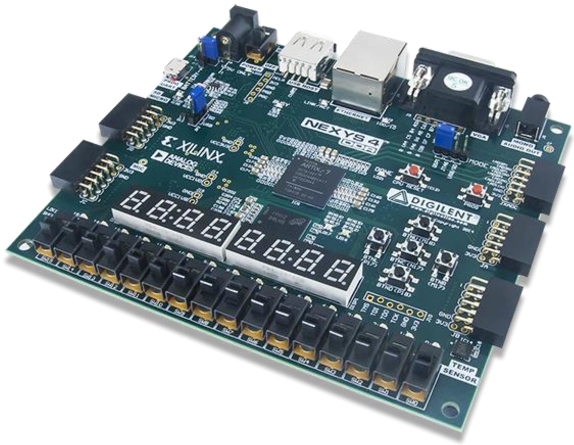


Figure 2-2: ***Nexys 4 DDR FPGA Board developed by Xilinx. Extract from*** *[1]*

2.2. C programming language

2.2.1. Introduction

C is a general-purpose programming language, which has been called a "system programming language". It is useful for writing compilers and operating systems. The language provides a variety of data type; the fundamental are characters ($char$), integers (int) and floating-point ($float$) numbers of several sizes. Moreover, there is a hierarchy of derived data types created with pointers, arrays, structures, and unions.

It also provides fundamental control-flow constructions: statements grouping, decision making ($if-else$), selecting one of a set of possible cases ($switch$), lopping with the termination test at the top ($while, for$) or at the bottom (do), and early loop exit ($break$).

C has no operations to deal directly with composite objects such as character strings, sets, lists, or arrays. And it itself provides no input/output facilities, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly-called functions.

C offers only straightforward, single-thread control flow, but no multiprogramming, parallel operations, synchronization, or coroutines.

The run-time library required to implement the self-contained program because the data types and control structures provided by C are supported directly by most computers. Although C matches the capabilities of many computers, it is independent of any particular machine architecture.

2.2.2. Creating a program with C

This section describes an example of the C program, which includes some of the constructions using a different type of data. The example given is, calculate the first ten primer numbers. With this example, some important aspect of the program will be described.

First of all, libraries have to be included in the program, which includes a group of functions of the C programming. The libraries have a format of .h (header file). These libraries can be C standard libraries –such as stdio.h or math.h– or created by the user. The following connotation is used to include a library, where the name of the library is indicated in the brackets. For this example, only one library is needed: the stdio.h.

```
#include <stdio.h>
```

Code 2-1: ***Declaration of library for example code***

After that, the symbolic constants could be defined. These are used when a constant "magic number" convey little information to someone who might have to read the program later. One way to deal with the "magic number" is to give it a meaningful name. A #define line defines a symbolic name or symbolic constant to be a particular string of characters. In this example case, it is going to define the number ten (number of prime numbers) as a symbolic constant, shown next:

```
#include <stdio.h>
#define    num   10          // number of primer numbers
```

Code 2-2: ***Defining a symbolic constant for example code***

As is could see in *Code 2-2* after double-slash (//) a single line comment can be written, which in this case explains briefly what the meaning of the code. It can also use "/* -comment- */" notation for multi-line comment.

Before the deceleration of libraries and symbolic variables, used variables are declared with their data type. For this example, four variables are needed: two variables to do the loops, one for compare if the number is prime and another is the vector variable which is used to save numbers. The vector variable has range of prime numbers required. The name of the variable has to begin with a string data. These variables can be initialized while declaring or after declaration.

```
#include <stdio.h>
#define    num   10          // number of prime numbers

/* Variables declaration */
int   i = 3;
int   j, c, prime[num];
```

Code 2-3: ***Declaration and initialization of the variables for the example***

After declaration the variables, the code is written. It could use all the architecture allowed in C language, for example, $if-else$ statements, loops for, $while$ and do, select one of the possible cases *switch*, and functions.

For this example, one loop for, one loop $while$ and two $if-else$ conditions are used to obtain prime numbers.

```c
#include <stdio.h>
#define    num   10         // number of prime numbers

/* Variables declaration */
int   c = 3;
int   i, j, prime[num];

/* Obtain prime numbers */
prime[0] = 2;          //The first prime number is 2
// next are going to calculate with next algorithm
j = 2;
while (j <= num)
{
      for (i = 2; i <= c -1; i++)
      {
            if (c%i == 0) break;
      }
      if (i == c)
      {
            prime[j-1]= c;
            j++;
      }
      c++;
}
```

Code 2-4: ***Complete example code using C language***

This simple example shows how a program can be generated in the C language. For more information about the C programming language visit [2].

2.3. Verilog

2.3.1. Hardware description languages

Hardware Description Language (HDL) is used in design electronics to describe models which describe logic circuits, for example, truth tables or integrated circuits. HDL has a lot of advantages. One of these is that design functionality could be verified before translating in a real hardware circuit.

They are different HDL to describe a logic model, the two most common are VHDL and Verilog because these are more widely used and well supported HDLs.

VHDL stands by Very High-Speed Integrated Circuit Hardware Description Language developed in 1980. VHDL is more verbose than Verilog and have less common with programming language C. On the other hand, the Verilog is more compact but has a lower level of programming construct.

In this chapter, only Verilog HDL is described because of the used in the thesis. To know about VHDL visit [3].

2.3.2. Introduction to Verilog

Verilog is a computer-based language to describe the hardware of the digital systems in a textual form. Functionality is described to implemented in hardware which can be defined in Boolean logic equations, truth tables and netlist of block interconnections.

The Verilog language has similarities with the C programming language. Verilog is case-sensitive and has a basic preprocessor to control flow keywords, such as $if/else$, for or $while$. These keywords have to be lowercase. The module begins with the keyword $module$ and ends with $endmodule$. The variables needed to declare with bit-widths. The Verilog differs from C for control-flow statements which are initiated with keyword $begin$ and close with end statement, however, other characteristics are same such as: ending line with a semicolon, comments begin with double-slash (//) or multi-comment with "/* … */".

2.3.3. Generating a digital block with Verilog

In this section, a simple digital block is created using Verilog to understand the functionality of the language. An adder is created as a demonstration example of a digital block. The block obtains two input signals which are going to save and will be allowed any time at the output port. It will make an add operation and save data in D Flip-Flop thus to access any time.

The Verilog design begins with the name *module,* after which the module name is given. Next, they are I/O ports declarations which could be input, output or bidirectional, beginning with *input*, *output* or *inout* keywords, respectively. In Verilog when any variable or port are declared, it is also needed the width size of the port or variable. For the example, the input ports have width two and the output port has width three –the third MSB is the carryout bit–, and there is not any port which is declared as bidirectional.

```
module add_2bits (CLK, RST, A, B, ADD);
input           CLK, RST;
input   [1:0]   A, B;
output  [2:0]   ADD;


...


endmoudle
```

Code 2-5: ***Ports declaration of the example module***

After the port's declaration, internal variables or *nets* are declared. *Net* data type represents physical interconnect between structures. And the variable data type represents elements to store data temporarily. For this example, variables are used to save data in D Flip-Flop.

```
module add_2bits (CLK, RST, A, B, ADD);
input           CLK, RST;
input   [1:0]   A, B;
output [2:0]    ADD;

reg   [2:0] Add_aux;

endmoudle
```

Code 2-6: ***Variables declaration for the example design***

After the variable's declaration, the hardware is described using different type of structures. The complete Hardware description is shown in *Code 2-7*.

An $always$ assignment is used to create a synchronous system and inside the statement, the process is executed sequentially. The process is executed when the clock signal has a rising edge ($posedge$).

$If - else$ statements are used to initiate variables with reset. Because the reset is Active-Low the not operation is made using "!" symbol.

"{ }" symbol is used to concatenate two variables. In the example this is used to create input signals of width three thus not to have width problems. After adding, the result is assigned to the output port.

```
module add_2bits (CLK, RST, A, B, ADD);
input             CLK, RST;
input [1:0]       A, B;
output[2:0]       ADD;

reg   [2:0]  Add_aux;

always @(posedge CLK) begin
        if (!RST) begin
            Add_aux <= 0;
        end
        else begin
            Add_aux <= {A,1'b0} + {B,1'b0};
        end
end
assign ADD = Add_aux;
endmoudle
```

Code 2-7: ***Complete hardware description of the adder***

3. MODEM

The modem is developed using a microcontroller and an Analogue to Digital Converter.

The FPGA has an analogic port thus to obtain the analogue signal. The sinusoidal signal of a specific frequency is generated using a wave generator to emulate the input signal coming from AUV or node. This signal is converted using the ADC to obtain digital signal.

This data is read by the microcontroller which is connected to ADC by AXI Interconnect and GPIO. The microcontroller performs the Goertzel Algorithm to detect the tone of the signal. When the input signal frequency corresponds to the selected frequency for the Goertzel Algorithm, a significant value of power from the Algorithm is received.

This power is compared with a setpoint value and in case this is higher than the setpoint, an Enable signal is set. This signal allowed to generate a square signal thus to emulate the Echo.

The modem consists of three parts:

- An ADC
- A Microcontroller
- Other parts for connection

In the following chapters, these parts are described with detail.

This page intentionally left blank

4. ANALOGUE TO DIGITAL CONVERTER (ADC)

This chapter described an Analogue-to-Digital Converter which is used in the modem. As is mentioned, this part is used to convert external electric, analogue, signal to digital signal.

The node device sends an acoustic wave to communicate with the animal, modem. A medium stage consists of an acoustic sensor which converts an acoustic wave into an electrical signal, known as the acoustic signal. The ADC receives this analogue signal to convert it to digital.

The ADC block could be modelled in Vivado. The program Vivado has an IP catalogue in which a lot of components are pre-designed as a block, knowing as IP. This could be imported in a Diagram when a Block Design is created. All connections of IP could be made external so to control or interact with the ADC after creating the wrapper in Vivado in Verilog or VHDL language. Furthermore, it could import more than one IP and interconnect in these in the same block diagram.

In the following section, the main features of XADC are described because the core could be modified from external, it is important to know the characteristics of the XADC.

Figure 4-1 illustrates the block design of this device.

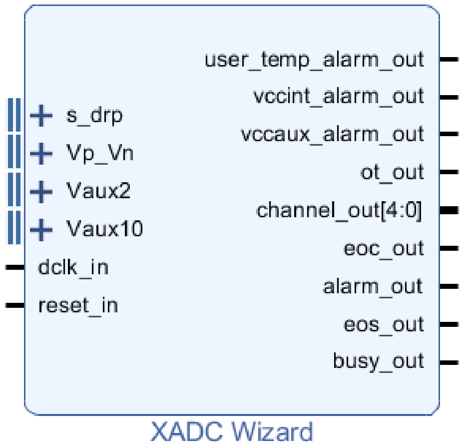


Figure 4-1: ***XADC Wizard IP***

4.1. ADC channels definition

The ADC has different pins for connection, which are used to configure ADC or connect with external ports to get the analogue signal or send the digital. The definition of these is described in this section.

4.1.1. Dynamic Reconfiguration Port bus (s_drp)

The DRP bus consists of: input address bus (*daddr_in[6:0]*), input enable signal (*den_in*), data input bus (*di_in[15:0]*), output data bus (*do_out[15:0]*), data ready signal (*drdy_out*) and write input enable (*dwe_in*).

The address bus is an input bus and indicates the address to access up to 128 registers (daddr_in[6:0] = 00h to 7Fh). Data input bus is used to save data in these registers. Before saving this data, the write enable signal has to be logic-High. Data ready signal goes logic-High when the data are written successfully in the register.

The input-enable signal has to be High to set an address or the write-enable signal. It should also be a logic-High to set data in the output bus. The bus has 16-bit width but only the 12 MSBs are the useful converted data.

4.1.2. Dedicated analogue input (Vp_Vn)

This port is used to get analogue signals. It is formed by two channels: V_P and V_N. V_P (vp_in) is a positive input terminal of the dedicated differential analogue channel and V_N (vn_in) is the negative input terminal of the dedicated differential analogue channel. These pins should be connected to GND when they are not using.

4.1.3. Auxiliary analogue inputs

These are multi-function pins that could support an analogue or a digital input-output (I/O). These pins support sixteen pares of analogue inputs, auxiliary positive analogue input (V_{AUXP}) and auxiliary negative analogue input (V_{AUXN}). The analogue input channels support multiple analogue input signal types. These pins could be used as digital I/O when not being used as analogue inputs.

Other core channels are described in *Table A-1.*

4.2. ADC registers

All registers in the register interface are accessible through the dynamic reconfiguration port (DRP). The DRP allows accessing up to 128 registers with 16 bits width.

The first 64 address locations ($daddr\_in[6:0] = 00h\ to\ 3Fh$) contain the read-only status registers, and they are known as status registers. These registers contain the result of analogue to digital conversion of the channels. The next 64 registers ($daddr_{in[6:0]} = 40h\ to\ 7Fh$) are control registers and are readable or writeable through the DRP. In *Figure 4-2* are shown these registers.

4.2.1. Status registers

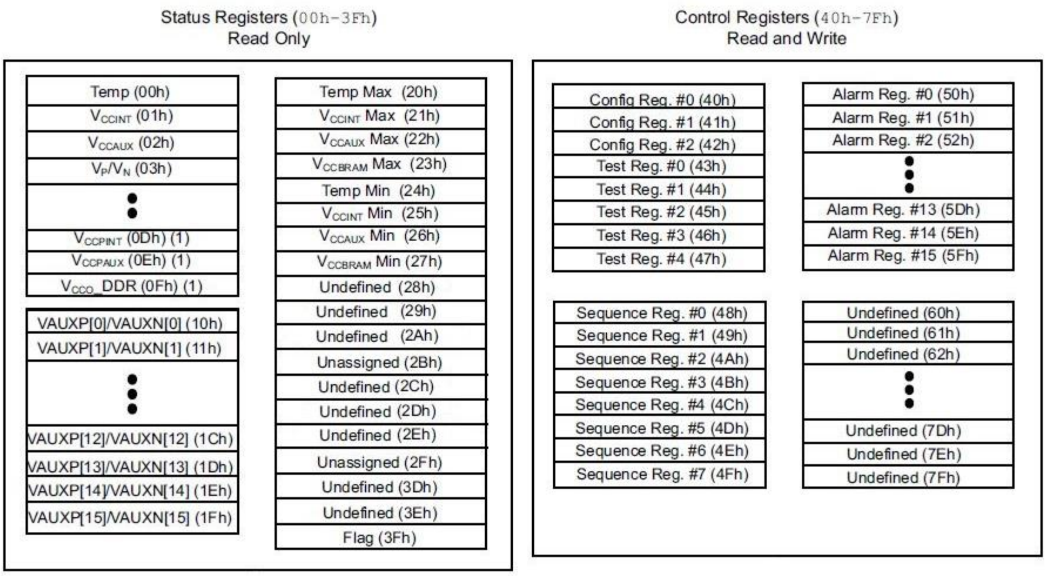


Figure 4-2: ***XADC registers (status and control registers). Extract from*** *[32]*.

The first 64 address locations contain the result of an Analogue-to-Digital conversion of on-chip sensors and external analogue channels. All sensors and external analogue input channels have a unique channel address. The measurement result from each channel is stored in a status register.

The status registers also store the maximum and minimum measurements recorded for the on-chip sensors since the device power-up or since the last user reset of the ADC. *Table A-2* defines the status registers.

4.2.1.1. Flag register

The 16-bits flag register is accessed by address $ADDR[6:0] = 3Fh$. It is consisted of:

15:12	11	10	9	8	7:4	3	2:0
X	JTGD	JTGR	REF	X	ALM[6:3]	OT	ALM[2:0]

Figure 4-3: ***Flag register (Address*** $= 3Fh$***)***

The bus $ALM[6:0]$ reflect the status of the alarm output $ALM[6:0]$.

The *OT* signal reflects the status of over temperature logic.

The REF signal indicates whether the ADC is using the internal voltage reference (logic-High) or the external reference (logic-Low).

The signals JTGR and JTGD are not important for this thesis so they are not described with details. However, when JTGR is logic 1, indicates that JTAG_XADC bitstream option has been used to restrict JTAG access to read-only. And when JTGD is logic 1 indicates that the JTAG_XADC bitstream option has been used to disable all JTAG access.

4.2.1.2. XADC calibration modes

The XADC can digitally calibrate out any offset and gain errors by connecting known voltages (V_{REFP} and V_{REFN}). These calibration coefficients are stored in status registers $08h$ to $0Ah$ for ADC A and $30h$ to $32h$ for ADC B (see *Table A-2*). This mode is used by enabling the calibration bits ($CAL0\ to\ CLA3$) in configuration register 1 ($41h$) (see *Table A-8*).

The XADC default operating mode automatically uses calibration by initialising a conversion on channel 8 ($08h$). During the calibration mode BUSY signal transitions High. This calibration sequence is four times longer than a regular conversion as offset and gain are measure for both ADCs and the power supply sensor.

4.2.1.3. XADC calibration coefficients definitions

As mentioned previously, the offset and gain calibration coefficients are stored in the status registers. These are read-only-registers, and it is not possible to modify the contents through the DRP. See *Figure 4-4*.

15:8	7	6	5	4	3:0	Bits
DATA[11:0]					Note	ADC A/B supply offset
DATA[11:0]					Note	ADC A/B bipolar offset
N/A		Sign	MAG[5:0]			ADC A/B Gain

Figure 4-4: ***Calibration bits in status registers***

Note: the unreferenced LSBs can be used to minimize quantization effects or improve resolution through averaging or filtering.

The offset calibration registers store the offset correction factor. The offset correction factor is a 12-bit, two's complements number and is expressed in LSBs.

The ADC gain calibration coefficient stores the correction factor for any gain error in the ADCs. The correction factor is stored in the seven LSBs (SIG and MAG[5:0]) of register and these stores both sign and magnitude. If the seventh bit is a logic-High, the correction factor is positive. If it is logic-Low, the correction factor is negative. The next six bits indicate the gain correction factor. Each bit is equivalent to 0.1% so, the calibration can correct errors in the range of $\pm 6.3\,\%$.

4.2.2. Control registers

The XADC has 32 control registers that are located addresses $40h$ to $5Fh$. These registers are used to configure the XADC operation. The XADC can be configured to start in a predefined mode after FPGA configuration.

Table 4-1: ***XADC Control registres***

Name	Address	Description
Configuration registers 0 to 2	$40h$ to $42h$	These are XADC configuration registers (see *Configuration registers*)
Test registers 0 to 4	$43h$ to $47h$	These are test registers. The default initialization is $0000h$. These registers are used for factory test and should be left at the default initialization.
Sequence registers	$48h$ to $4Fh$	These registers are used to program the channel sequencer function (see *Operating Modes*).
Alarm registers	$50h$ to $5Fh$	These are the alarm threshold registers for the XADC alarm function.

4.2.2.1. Configuration registers

The first three registers (address $40h$ to $42h$) in the control register block, are known as XADC configuration registers. The configuration could be modified through the DRP after the FPGA has been configured, for example using a state machine.

These configuration registers ($40h$, $41h$ and $42h$) are defined in *Table A-3*, *Table A-4* and *Table A-5*, respectively.

4.3. Transfer Functions

The ADCs have transfer functions which could be operating in unipolar or bipolar modes. All on-chip sensors use the unipolar mode of operating for the ADC.

The data received after the conversion is shown in figure… The ADC always produce a 16-bit conversion result. The 12 MSBs in the 16-bit status registers. The unreferenced LSBs can be used to minimize quantization effects or improve resolution through averaging of filtering.

15:12	11:8	7:4	3:0
DATA[11:0]			**Unreferenced**

*Figure 4-5: **Status registers DADDR[6:0] = 00h to 07h & 10h to 2Fh***

4.3.1. Unipolar mode

In this mode, the nominal analogue input range to the ADCs is 0 V to 1 V. The ADC produces a zero code (000h) when 0 V is present on the ADC input and a full-scale code ($FFFh$) when 1 V is present on the input.

The ADC output coding in unipolar mode is straight binary. The LSB size in volts is equal to $\frac{1V}{2^{12}} = 244\ \mu V$. The analogue input channels are differential in nature and require both the positive (V_p) and negative (V_N) inputs of the differential input to be driven.

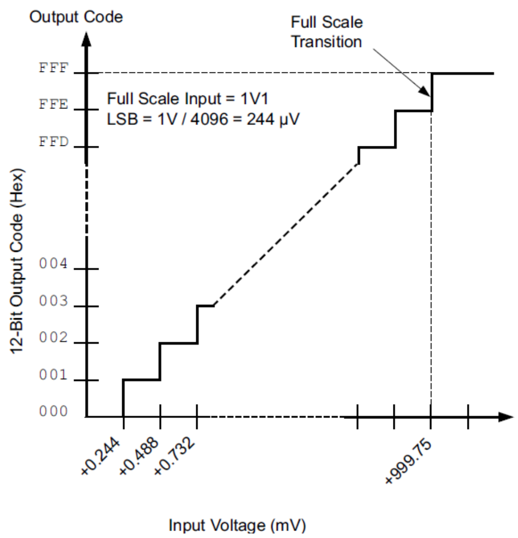


Figure 4-6: ***Unipolar transfer function***

4.3.2. Bipolar mode

It is useful to have both magnitude and sign information about the signal when dealing with differential signal types. The output coding of the ADC in bipolar mode is two's complement and the sign is indicated V_P relative to V_N. In the differential analogue input $(V_P - V_N)$ can have a maximum input range of $\pm 0.5\,V$. The LSB size in volts is equal to $1V/2^{12} = 244\ \mu V$.

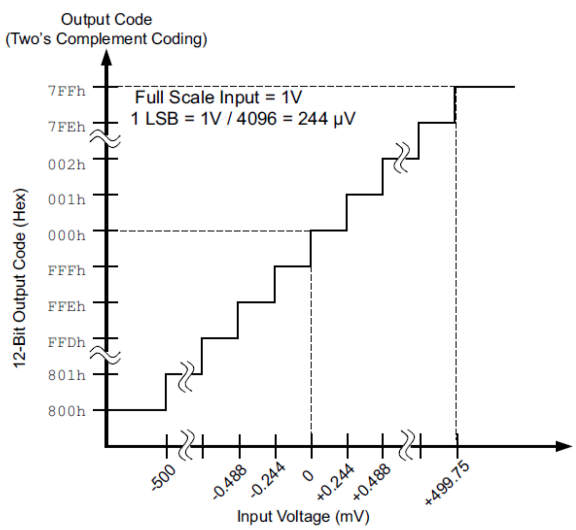


Figure 4-7: ***Bipolar transfer function***

4.3.3. Temperature sensor

The XADC contains a temperature sensor that produces a voltage output proportional to the die temperature. The output voltage of the temperature sensor is shown in the next Equation

$$V = 10 \cdot \ln(10) \cdot \frac{k \cdot T}{q}$$

Where:

V = voltage in volts

k = Boltzmann's constant = $1.38 \cdot 10^{-23}\,J/K$

T = Temperature in Kelvin (ºC + 273.15)

q = Charge on an electron = $1.6 \cdot 10^{-19}\,C$

the temperature sensor plus the ADC transfer function is rewritten as shown in next equation:

$$T\,(°C) = \frac{ADCcode \cdot 503.97}{4096} - 273.15$$

The LSB in degree centigrade is equal to 0.123 ºC.

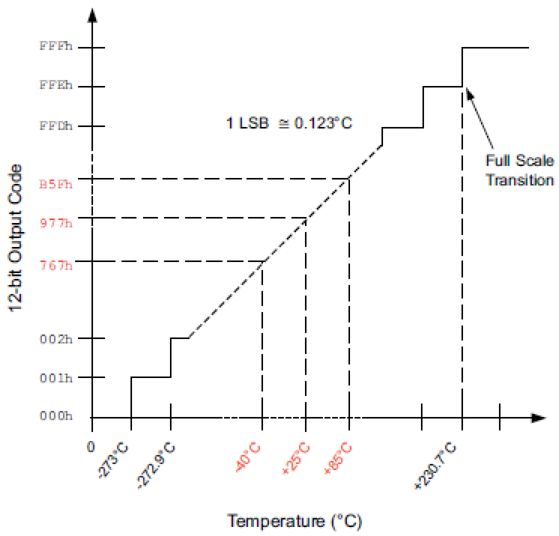


Figure 4-8: ***Temperature transfer function***

4.3.4. Power supply sensor

The XADC also includes on-chip sensors that allow a user to monitor the FPGA power-supply voltage using the ADC. The sensor sample and attenuate (by a factor of three) the power supply voltages V_{CCINT},V_{CCAUX} and V_{CCBRAM}.

The power supply sensor could be used to measure voltages in the range 0 V to V_{CCAUX} + 5% with a resolution of approximately 0.73 mV. The transfer function for the supply sensor is shown in equation… and is visualized in figure…

$$V\,(V) = \frac{ADCcode}{2^{12}} \cdot 3\,V$$

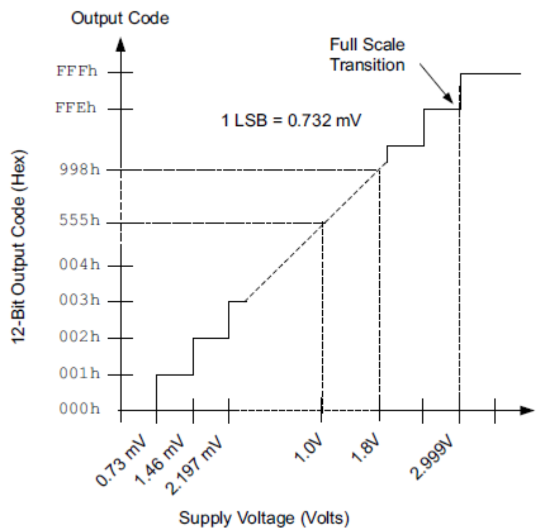


*Figure 4-9: **Ideal power supply transfer function***

4.4. Operating Modes

The XADC includes several operating modes. The most basic mode of operation is called default mode, where the XADC monitors all on-chip sensors and requires no configuration. In the simultaneous sampling mode, the sequencer is used to operate both ADCs in lock step to sample two external analogue inputs and store results in the status registers.

4.4.1. Single channel mode

In this mode, an analogue-to-digital conversion channel is selected by writing to bit locations $CH[4:0]$ in control register 40h. See *Table A-3* so how to select this mode.

The configuration of analogue input mode ($B\overline{U}$) and settling time (ACQ) also be set by writing to control register $40h$.

4.4.2. Automatic channel sequencer

The automatic channel sequencer sets up a range of predefined operating modes. The sequencer automatically selects the next channel for conversion, sets the averaging, configures the analogue input channels, sets the required setting time for acquisition, and stores the results in the status registers. To select sequencer mode, see *Table A-9*.

This mode needs several configuration registers to set.

4.4.2.1. ADC channel selection registers

The ADC channel selection registers enable and disable a channel in the automatic channel sequencer. The bits for these registers are defined in *Table A-12* and *Table A-13*. A logic 1 enables a channel in the sequence. It consists of one pair of 16-bit registers ($48h$ and $49h$).

4.4.2.2. ADC channel averaging

Averaging could be selected independently for each channel in the sequence. It consists of one pair of 16-bit registers ($4Ah$ and $4Bh$). These registers also have the same bit assignments as the channel sequence registers listed in *Table A-12* and *Table A-13*.

When averaging is enables for some of the channels of the sequence, the EOS is only pulsed after the sequence has completed the amount of averaging selected by using $AVG[1:0]$ bits (see *Table A-7*). If a channel in the sequence does not have averaging enabled, its status register is updated for every pass through the sequencer. When a channel has averaging enabled, its status register is only updated after the averaging is complete.

4.4.2.3. ADC channel analogue-input mode

These registers are used to configure an ADC channel as either unipolar or bipolar in the automatic sequence. It consists of one pair of 16-bit registers ($4Ch$ and $4Dh$). These registers also have the same bit assignments as the channel sequence registers listed in *Table A-12* and *Table A-13*. However, only external analogue input channels, such as the dedicated input channels (V_P and V_N) and the auxiliary analogue input (VAUXP[15:0] and VAUXN[15:0]) could be configured in this way.

4.4.2.4. ADC channel setting time

The default setting time for an external channel in continuous sampling mode is four ADCCLK cycles. The settling time is additional acquisition time after the end of a conversion. However, by setting the corresponding bits (for external channels) to logic 1 in registers $4Eh$ and $4Fh$, the associated channel could have its settling time extended to 10 ADCCLK cycles. The bit definition (which bits correspond to which external channels) for these registers are the same as the sequencer channel selection shown in *Table A-12* and *Table A-13*.

4.4.3. External multiplexer mode

The XADC allows an output bus called MUXADDR[4:0] (up to 16 external signals) to control an external multiplexor. The address on this bus reflects the channel currently being acquired.

The external multiplexor is useful where FPGA I/O resources are limited, and auxiliary analogue inputs are not available. The dedicated analogue inputs (V_P/V_N) or internal auxiliary inputs ($V_{AUX}[15:0]$) are used to connect the external multiplexer to the XADC block. Two external multiplexers are used to support simultaneous sampling.

The MUX bit is set to 1 to select this mode. The channel selection bits ($CH[4:0]$) in $Configuration\ Register\ 0$ are used to nominate the channel for connection to the external multiplexor.

4.5. Timing

All XADC timing is synchronized to the DRP clock (DCLK). The ADCCLK generated by dividing DCLK by the user selection in the configuration register (see *Table 4-1*).

The ADC operates either in continuous sampling mode or event sampling mode. The operating mode is selected by writing to configuration register 0 (see *Table A-3*).

4.5.1. Continuous sampling

In continuous sampling mode, the ADC automatically starts a new conversion at the end of the current conversion cycle. The analogue-to-digital conversion process is made up of two parts, the acquisition phase and the conversion phase.

4.5.1.1. Acquisition phase

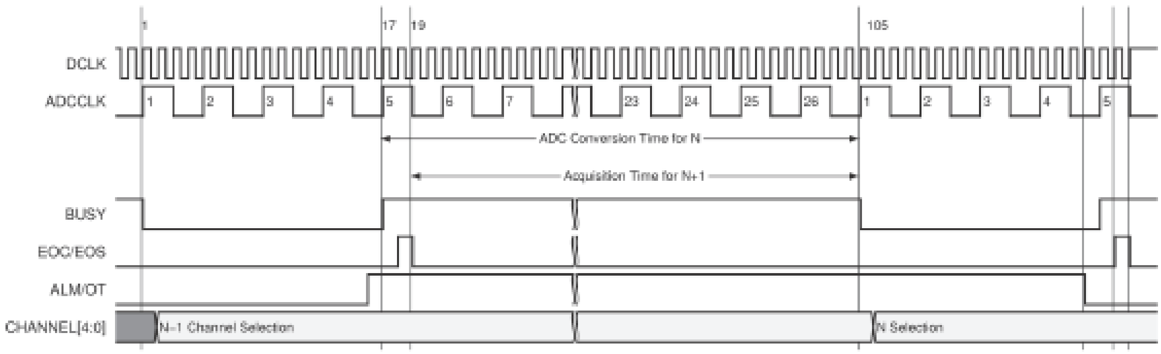


Figure 4-10: ***Continuous sampling mode. Extract from*** *[32].*

The acquisition phase involves charging a capacitor in the ADC to the voltage on the selected channel. The time required to charge this capacitor depends on the selected input-channel source impedance. A settling period of four ADCCLK cycles is left between the end of the current conversion and the start of the next conversion.

In single channel mode, the $configuration\,register\,0$ has to be written to select the next channel for conversion.

4.5.1.2. Conversion phase

The conversion phase starts at the end of the 4 or 10 ADCCLK cycles settling time and it takes 22 ADCCLK. The BUSY signal transitions to an active-High to indicate the ADC is carrying out a conversion. 16 DCLK cycles after BUSY goes Low, EOC pulses High for one DCLK cycles.

EOS indicates the end of the sequence, which depends on the automatic channel sequencer settings and averaging settings.

4.5.2. Event sampling mode

In event sampling mode next conversion is initiated using a trigger input signal called CONVST. A rising edge on CONVST defines the exact sampling instant for the selected analogue-input channel. The BUSY signal transition High just after the sampling instant on the next rising edge of DCLK.

As with the continuous sampling mode, enough time must be allowed between a channel change and the sampling edge (rising edge of CONVST). A settling period of four ADCCLK cycles is recommended between the end of the current conversion (BUSY going logic-Low) and the start of the next conversion. 16 DCLK cycles after Busy goes Low, EOC pulses High for one DCLK.

It is not possible to interrupt the conversion or start a new conversion until BUSY goes Low after a conversion has been initiated by CONVST.

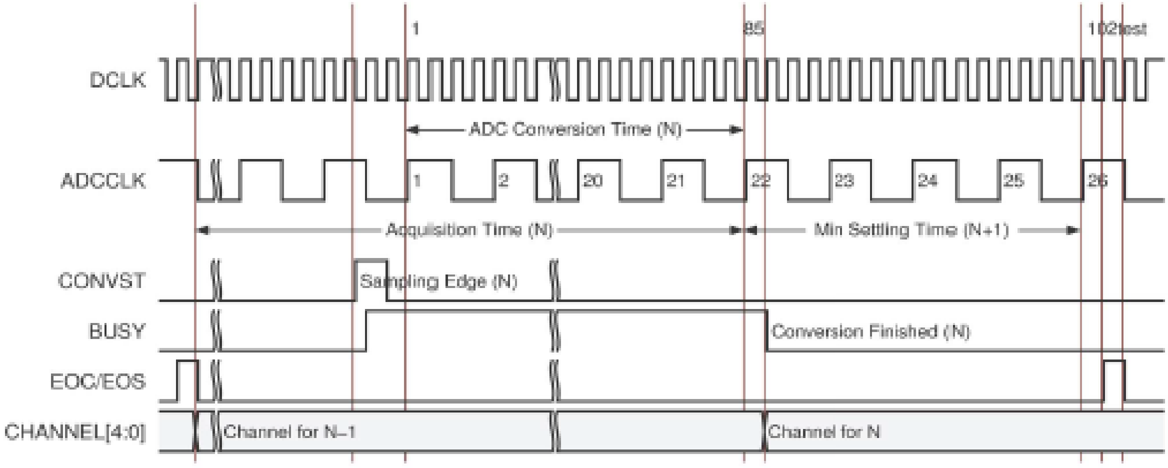


*Figure 4-11: **Event sampling mode. Extract from** [32].*

4.5.3. Dynamic reconfiguration Port Timing

The DRP address (DADDR) and write enable (DWE) inputs are captured when the DRP enables is logic High (DEN).

When DRDY goes High, the data for this read operation is valid on the DO bus. DWE signal has to be logic Low. For a write operation, the DWE signal is logic High and the DI bus and DRP address (DADDR) is captured. The DRDY signal goes logic High when the data has been successfully written to the DRP registers. A ness read or write operation cannot be initiated until the DRDY signal has gone logic-Low.

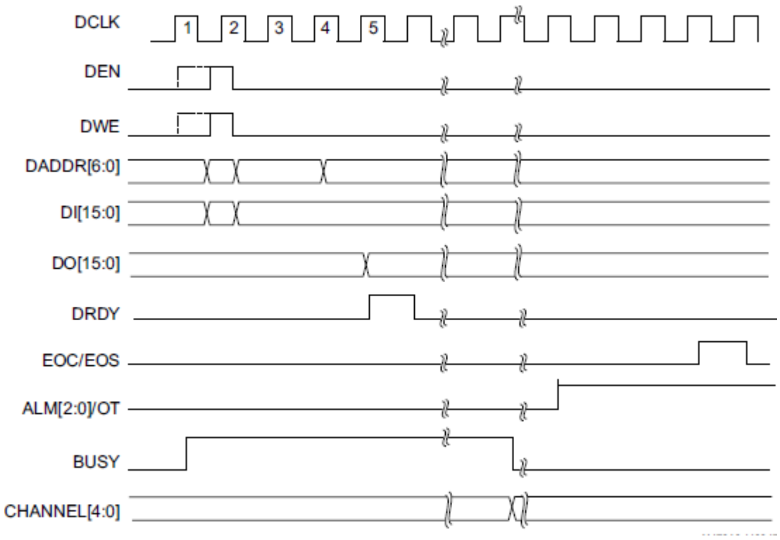


Figure 4-12: **DRP timing. Extract from** [32]**.**

5. ARM CORTEX-M3

ARM processors address many different market segments including enterprise application, automotive systems, home networking and wireless technologies. The ARM Cortex family of processors provided a standard architecture to address the board performance.

The ARM Cortex family includes processors based on the three distinct profiles; the A profile for sophisticated and complex operating systems; the R profile for real-time systems; and the M profile optimized for cost-sensitive and microcontroller applications.

The ARM Developer has different Cortex-M microcontrollers without a licence to develop in FPGA. The most common microcontroller is based on Cortex M0, known as Cortex M1. This microcontroller supports all FPGA functionalities and with the facility to edit using Hardware Description Languages.

Nevertheless, for this thesis, a Cortex-M3 microcontroller is used because of their easy connections with Vivado components through AXI-3 bus. Furthermore, the Cortex-M3 does not need extra third-party components to function properly because it is packed as a Vivado IP.

This core is also available by ARM Developer to implement in FPGA however it is developed for Arty A7 Xilinx board and has an example Hardware and Software program. Hence a lot of time is dedicated to this part; first to understand the example and adapt for Nexys 4DDR board, and then to develop the modem in this board.

5.1. Introduction

The ARM Cortex-M processors are high performance, low cost, low power, 32-bit RISC processors, designed for microcontroller applications [4]. The Cortex M processors are based on the ARMv7-M architecture. This architecture includes efficient 3-stage pipeline instructions and low-latency Interrupt Service Routine (ISR) entry and exit.

Cortex M3 also includes hardware divide and Multiply-Accumulate (MAC) operations, a Nested Vectored Interrupt Controller (NVIC), an optional memory protection Unit (MPU), Timer, Debug Access Port (DAP) and optional Embedded Trace Macrocell (ETM).

The Cortex M3 processor is used for different applications such as microcontrollers, industrial control, in mobile phones (as secondary core), in an FPGA (as softcore), and other applications.

5.2. FPGA edition

The Cortex M3 is adapted to implement in FPGA with similar internal features. The core includes a Nested Vectored Interrupt Controller which support up to 240 interrupts, the priority of each of them could be changed dynamically.

It supports configurable embedded debug support, ITCM alias support and Serial Wire (SW), JTAG, or combined SWJ-DP debug port. An AHB to AXI bridge is integrated to make possible the communication with GPIOs or other external Vivado components.

The core allows by their pins (see *Table A-14*) the following optional features:

- Embedded Trace Macrocell (ETM)
- Data Watchpoint and Trace (DWT)
- Instrumentation Trace Macrocell (ITM) coupled with Trace Port Interface Unit (TPIU)
- Memory Protection Unit (MPU)
- AHB access through Serial-Wire or JTAG Debug Port
- Flash Patch Breakpoint (FPB)

There are two Tightly Coupled Memory (TCM), for code and data, which are configurable in size. The Instruction Tightly Coupled Memory (ITCM) can be configured at run time to be aliased to either or both of 0x00000000 and

0x10000000. The Data Tightly Coupled Memory (DTCM) is at a fixed location of 0x20000000. The size of ITCM and DTCM could be chosen, each of them, from 8 kB to 1 GB with multiple of 8.

The instruction code and data code AHB interfaces from the processor are combined internally. Any access from either of these buses which does not match an active ITCM alias is presented on the external instruction AXI interface. The system AHB interface from the processor is used to access the DTCM. Any access which is not within the range of the configured DTCM size is presented on the system AXI interface.

The ITCM and DTCM connection is shown in *Figure 5-1*.

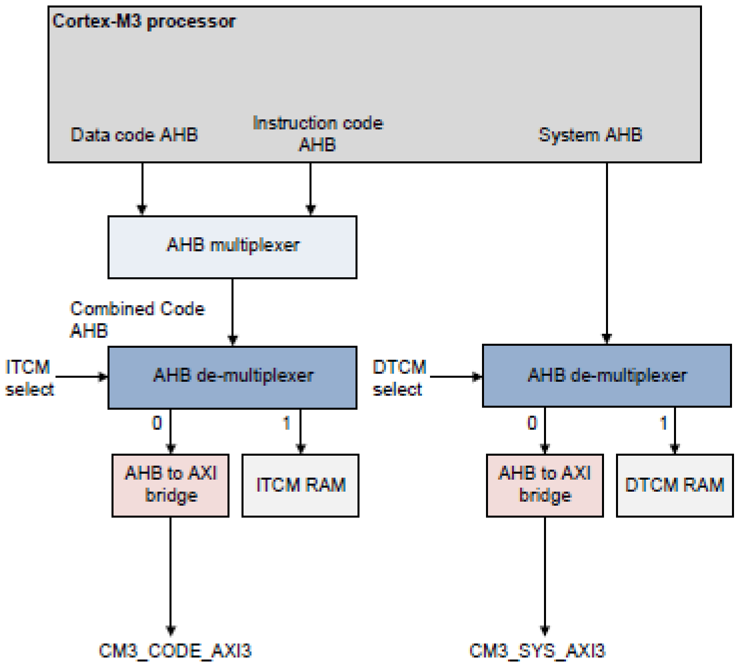


Figure 5-1: ***Internal memory processing multiplexor. Extract from*** *[14]*.

5.3. Core features

As is mentioned, the core has features which could be accessible by their ports. It is interesting to know the functionality of these features.

5.3.1. Nested Vectored Interrupt Controller (NVIC)

NVIC can support up to 240 external interrupts with 256 levels of priority which can be changed dynamically. The ARMv7-M architecture supports level-sensitive and pulse-sensitive interrupt behaviour.

NVIC can be enabled or disabled by writing to their corresponding Interrupt Set-Enable or Interrupt Clear-Enable register bit field. A function is used in this thesis to enable and disable the interrupts. When an interrupt is disabled, interrupt assertion causes the interrupt to become pending, but it cannot become active. In this case, it remains active until this is cleared by a reset or an exception return.

The software can set or remove the pending state of NVIC interrupts using registers, the Set Pending Register and Clear-Pending Register. Two functions are used to Set-Pending interrupt and Clear-Pending Register, which accesses to Set pending Register and Clear-Pending Register to write-one to enable and write-zero to disable.

5.3.2. The system timer (SysTick)

The core includes a system timer, SysTick which provide a 24-bit clear-on-write, decrementing counter with the flexible control mechanism. The timer could be used

- As an RTOS tick timer, that fire at a programmable rate and invokes a SysTick routine each time is fire.
- As a high-speed alarm timer.
- As a variable rate signal timer.
- As a simple counter. The software can use this to measure time to completion and time used.
- As an internal clock source control base on missing or meeting duration.

The timer consists of four registers:

- A control and status register, which is used to configures the SysTick clock, enable the counter, enable the SysTick interrupt, and indicates the counter status.
- A counter reloads value register.
- A counter current value register.
- A calibration value register, which indicates the preload value required for a 10 ms (100 Hz) system clock.

The STCLK input is a reference input for the SysTick counter which has to be less than half the frequency of HCLK. STCLK is synchronized internally by the processor to HCLK.

5.3.3. Non-maskable Interrupt (NMI)

The Non-Maskable Interrupt is a hardware interrupt which cannot be ignored by the system. It is used for signal attention of non-recoverable hardware errors, system debugging and handling of special cases such as system resets. Programmers debugging NMIs to diagnose and fix the faulty code.

When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. The exception vector contains the first instruction of the interrupt service routine to deal with the exception.

NMI cannot be disabled by NVIC and also the prioritization does not affect this signal because it has a higher priority than external interrupts.

5.3.4. Serial Wire JTAG Debug Port (SWJ-DP)

The processor contains an Advanced High-performance Bus Access Port (AHB-AP) interface for debus accesses. The Cortex M3 system supports two possible Debug Port implementations:

- The JTAG Debug Port (JTAG-DP) which is standard debug port. It is designed to permit pin sharing of JTAG-TDO and JTAG-TDI when they are not being used for JTAG debug access.

- The Serial Wire Debug Port (SW-DP) which provide a two-pin (clock and data) interface to the AHB-AP port.

These two DP implementations provide different mechanisms for debug access to the processor.

FPGA pins are required for all output, with the exception of the SWDO, SWDOEN, and SWDITMS, to use package pins efficiently, serial wire shares, or overlays, the JTAG pings. It also uses an autodetect mechanism which switches between JTAG-DP and SW-DP based on SWDIOTMS signal. The external connection of the SWJ-DP is shown in *Figure 5-2*.

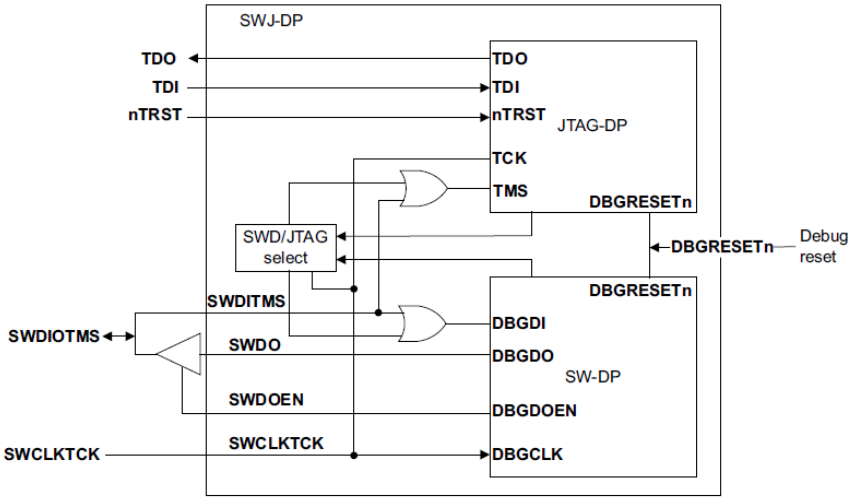


*Figure 5-2: **SWJ-DP external connection. Extract from. Extract from** [17]*

5.3.5. External debug request

When the processor is in the non-Debus state, an external agent can signal an external debug request, asserting EDBGRQ, which can cause a debug event (either entry to Debug stat or a debug Monitor Exception).

The processor ignores external debug requests when it is in Debug state.

5.3.6. External restart request

When the processor is in debus stat, an external agent can ask an External Restart request which causes the processor to exit Debug state. The processor ignores external restart requests when it is in Non-Debug state.

When DBGRESTART is asserted, it has to be held High until DBGRESTARTED is deserted. DBGRESTART is ignored unless HALTED and DBGRESTARTED are asserted.

In the process of leaving Debug state, the processor sets the HALTED signal to logic-Low.

5.3.7. The boundary scan chain

A boundary scan chain is made up of serially-connected device which implements a boundary scan technology using a standard JTAG TAP interface. The core contains at least one TAP controller containing shift registers which from chain connected between TDI and TDO.

Test Data Input (TDI) is an input line to allow the test instruction and test data to be loaded into the instruction register and the various test data registers, respectively. The Test Data Output (TDO) is an output line used to serially output the data from the JTAG registers to the equipment controlling the test.

5.3.8. Bit-banding

The processor memory map includes two bit-band regions. These occupy the lowest 1 MB of the SRAM and Peripheral memory regions respectively. These bit-band regions map each word in an alias region of memory to a bit in a bit-band region of memory.

A mapping formula shows how to reference each word in the alias region to a corresponding bit, or target bit, in the bit-band region. The mapping formula is:

$$bit\_word\_offset - (byte\_offset\ x\ 32) + (bit\_number\ x\ 4)$$

$$bit\_word\_addr - bit\_band\_base + bit\_word\_offset$$

where

- Bit_word_offset is the position of the target bit in the bit-band memory region.

- Bit_word_addr is the address the word in the alias memory region which maps to the targeted bit.
- Bit_band_base is the starting address of the alias region.
- Byte_offset is the number of the byte in the bit-band region which contains the targeted bit.
- Bit_number is the bit position (0-7) of the targeted bit.

5.4. Memory map

The Cortex-M3 has up to 4 GB memory access. The memory map is shown in *Figure 5-3*.

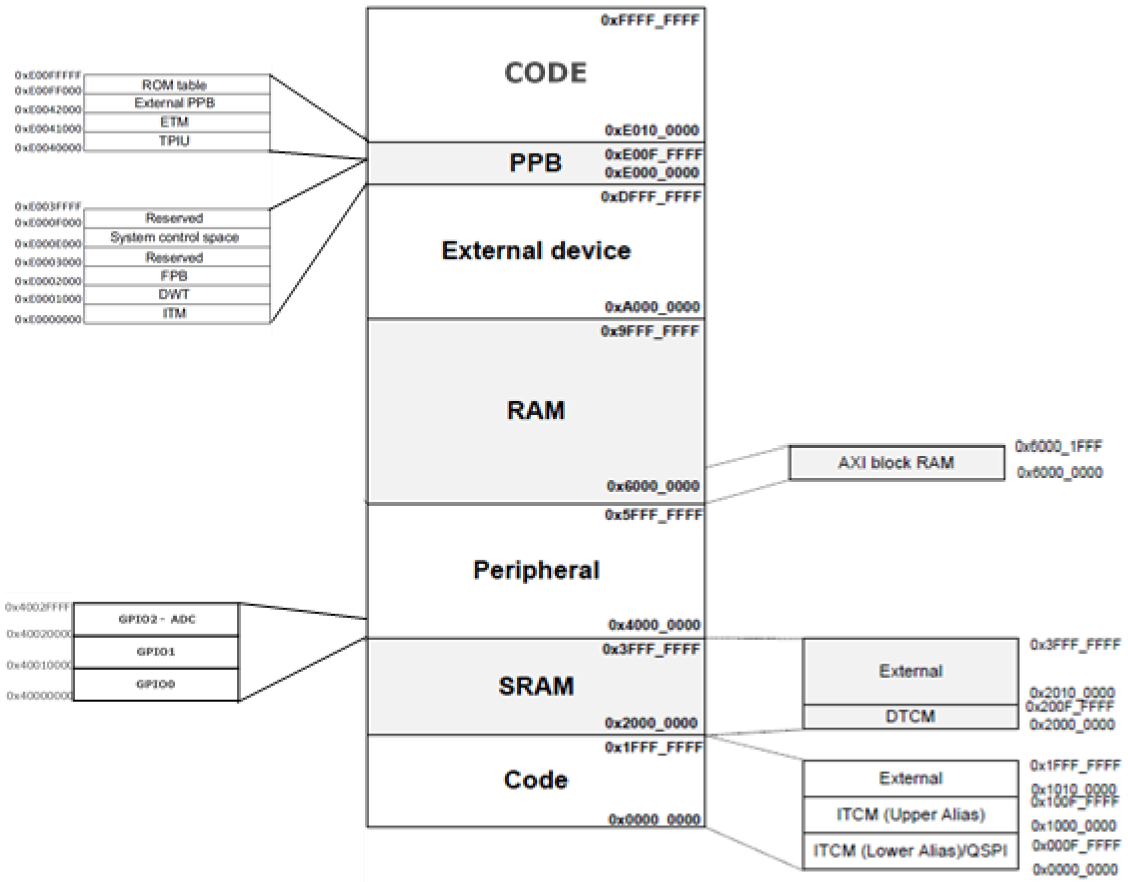


Figure 5-3: ***Cortex M3 processor memory map***

In the following table, the processor which are addressed by the different memory map regions is described.

*Table 5-1: **Memory Interface***

Memory Map	Interface
Code	Instruction fetched are performed over the ICode bus. Data access are performed over DCode bus.
SRAM	Instruction fetches and data accesses are performed over the system bus.
SRAM_bitband	Alias region. Data accesses are aliases. Instruction accesses are not aliases.
Peripheral	Instruction fetches and data accesses are performed over the system bus.
Periph_bitband	Alias region.
External RAM	Instruction fetches and data accesses are performed over the system bus.
External Device	Instruction fetches and data accesses are performed over the system bus.
Private Peripheral Bus	Accesses to: - System areas of the PPB memory map - Flashpatch and Breakpoint (FPB) - Data Watchpoint and Trace (DWT) - Instrumentation Trace Macrocell (ITM) - Nested Vectored Interrupt Controller (NVIC) - Memory Protection Unit (MPU) - External Private Peripheral Bus - Embedded Trace Macrocell (ETM) - TracePoint Interface Unit (TPIU)

System	System segment for vendor system peripherals. This memory region is Execute Never (XN), and thus instruction fetches are prohibited.

5.5. Cortex M3 configuration

The Cortex M3 core is a version of Cortex-M3 r2p1 processor with debugging and two BP136 AHB to AXI Bridges r0p1 pre-integrated. This section describes the core configuration windows.

5.5.1. Configuration window

This tab includes the basic configuration of the core.

It describes the vector of interrupts the length of this vector is automatically calculated depending on the input interrupt numbers, IRQ input bus. Hence the value of this vector cannot be set directly. The block diagram is validated thus to update the width of the IRQ bus.

The *IRQ Priority level width* determines the number of IRQ priority level which is equal to $2^{\textit{Priority level Width}}$. The range is between 3 to 8 bits, with default value 3 which means 8 levels of priority.

When the *MPU present* is set, the Memory Protection Unit in the core is enabled.

With *WIC present* the Wake-up Interrupt Controller is enabled in the Cortex M3. And *WIC Lines* determine the number of internal priority lines used for the WIC, which only be changed if the *WIC present* is enabled.

Bit-banding present enables bit-banding in the Cortex-M3 core, see *5.3.8. Bit-banding* for more information.

5.5.2. Debug window

Debug level enables to set the level of debug supported. Different debug levels can be selected from zero to three; where 0 is no debugging, 1 select two breakpoints and one watchpoint, 2 is full debug except DWT, and 3 is full debug including DWT.

Trace level enables to set the level of debug trace which is supported from no trace to full trace. If the Debug level is set to 0 (no debug), the Trace level is automatically set to 0. In other cases, the Trace level has to be reset to the required value.

JTAG present enables Cortex-M3 JTAG-debug pins, by default *Serial Wire* (SW) interface is used.

5.5.3. Instruction and data memory window

ITCM or DTCM size select the size of *Instruction Tightly Coupled Memory* or *Data Tightly Coupled Memory,* respectively. These memories can have range 8 KB to 1 MB.

To select instruction memory, *Initialize ITCM* is enabled and then the filename is specified. The same procedure is done to select data memory –enable the *Initialize DTCM*, and specified the filename– but in data memory window.

The filename cannot have quote marks around it and the filename has to be added into the design and marked as a memory initialization file.

Caution. Vivado reads the memory file during synthesis. That is to say, any modification in memory will not be updated after the synthesis has been completed. Hence it is needed to rerun the synthesis, and generate the bitstream file again.

5.6. Program

This section describes the microcontroller program file using Keil uVision. Keil uVision is a program dedicated to generate program files for microcontrollers. This program is used to generate file .hex to run the Cortex M3.

The example program has included also a software example in Keil uVision with an example application. These files have been studied because some files are common files which do not depend on the application, such as *core_cm3*.

To generate the software file, the next steps are followed:

- After the hardware design is completed, the standalone BSP files are generated using SDK program, which is included in Vivado. This program generates from Hardware Description File the BSP files to include in Keil project. This way the hardware design is included software files to interact with Hardware design by the program.
- The program is developed in Keil using C language. The main part of this program is the Goertzel Algorithm to detect the input signal (see 5.6.1. Goertzel Algorithm).
- When the program is finished, it is compiled to generate .hex files.
- Finally, the synthesis is rerun to include in bitstream the new hex file.

5.6.1. Goertzel Algorithm

The Goertzel Algorithm is used to detect a specific predeterminate frequency. The algorithm can be used for tone detection and use less CPU than the Fast Fourier Transform. It gives real and imaginary frequency components as a Discrete Fourier Transform (DFT) or Fast Fourier Transform (FFT). This algorithm only can detect the tone when the input signal corresponds to a sinusoidal. For another sort of signal, this algorithm is not useful.

The number of samples and sampling rate has to know previously to calculate the algorithm parameters. Furthermore, it needs n number of samples to obtain the frequency components. This could be a problem because the number of samples has to be stored in RAM.

However, the algorithm is modified to work real-time thus not to save data in memory which could take time and power of microcontroller. Hence, when the sample data is obtained, an output signal goes logic-High thus not to obtain more data until the algorithm has finished the calculation. After the calculation has been done the signal return logic-Low again.

The algorithm is shown in *Code D-11* and *Code D-12*.

This page intentionally left blank

6. IP USED IN BLOCK DESIGN

To develop the design there have been used some other Vivado IPs. These IPs are necessary thus to interconnect the main blocks.

6.1.1. AXI Interconnect

This IP is used to interconnect one or more AXI memory-mapped master device to one or more AXI memory-mapped slave device. AXI Interconnect core can be configured to perform one of the following general connectivity patterns:

- *N-to-1 Interconnect:* in this configuration mode multiple master devices access to the single slave device.
- *1-to-N Interconnect:* in this configuration mode only one master device access to multiple slave devices.
- *N-to-M interconnect:* in this configuration mode multiple master devices access to multiple slave devices.

A slave device is a target of an AXI transfer which receives in-bound AXI transactions and a master device is a source of an AXI transfer which generates out-bound AXI transactions.

The block diagram of AXI Interconnect is shown in *Figure 6-1*.

The definition of AXI Interconnect are divided into three tables: AXI Interconnect Slave I/O pins, AXI Interconnect Master I/O pins, and AXI Global signals; which are shown in *Table A-15*, *Table A-16* and *Table A-17*, respectively.

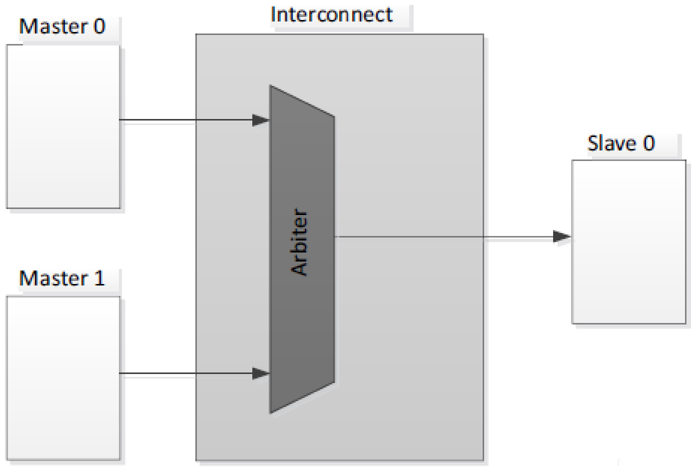


Figure 6-1: ***AXI Interconnect Block diagram. Extract from*** [24]

6.1.2. Utility

The AXI Interconnect IP is used to connect the Microcontroller to external GPIOs. The Microcontroller is the only one which performs as master. But there are four Slave devices which are: GPIO 0, GPIO1, GPIO 2 (see *6.2. AXI GPIO*) and Block RAM Controller (*6.3. AXI BRAM Controller*).

The AXI Interconnect is shown in *Figure 6-2*.

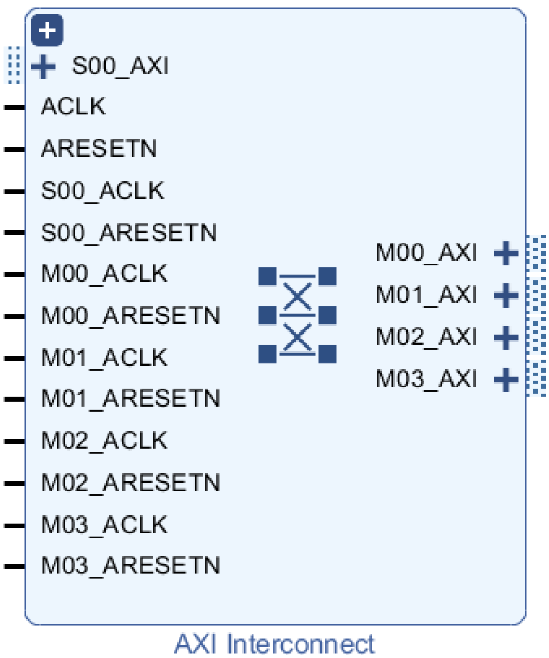


Figure 6-2: ***AXI Interconnect IP***

6.2. AXI GPIO

The General-Purpose Input Output (GPIO) can be configured as either a single or dual-channel device, each of this could be configured input or output. Channel one is always present. Channel two is enabled only if the core is configured for dual channel.

The maximum GPIO width is thirty-two bits. When the channels are configured as outputs, the default value could be selected.

The AXI GPIO also be configured to generate an interrupt when the Enable Interrupt option is set. The interrupt is generated when a transition occurs in any of their input.

The Block diagram of the GPIO is shown in *Figure 6-3*.

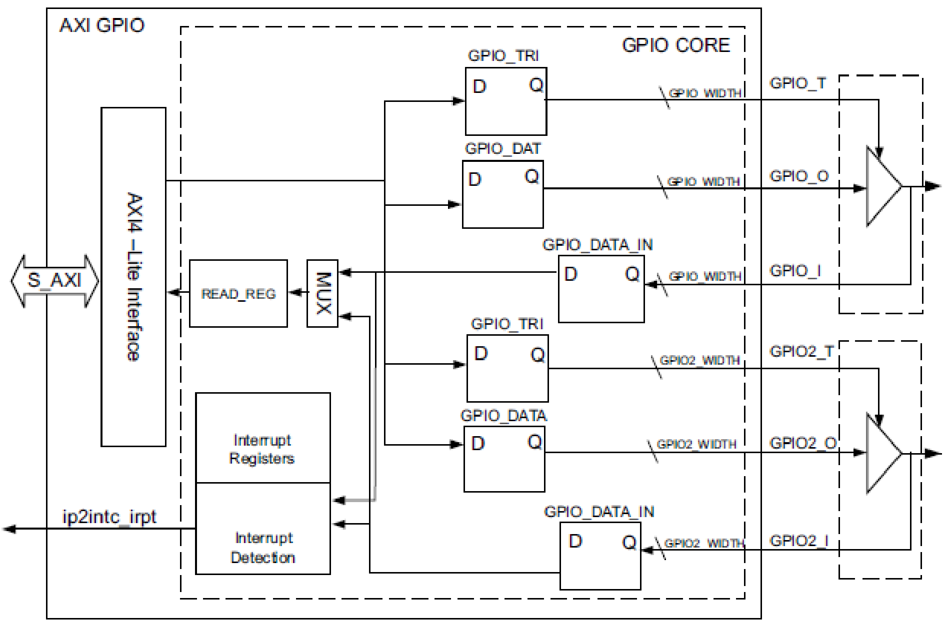


*Figure 6-3: **AXI GPIO block diagram. Extract from** [23]*

The port description is could observe in *Table A-18*.

6.2.1. Utility

The GPIOs are used to get data from output or send data to output. The AXI GPIO IP is shown in Figure 6-4: **AXI GPIO IP***Figure 6-4*.

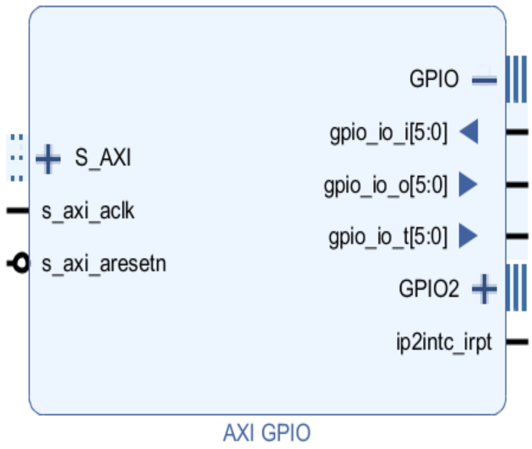


Figure 6-4: ***AXI GPIO IP***

Three GPIOs are used for different purposes which are explained below.

The AXI GPIO 2 is explained in *6.11. ADC diagram*.

6.2.1.1. AXI GPIO 0

The GPIO 0 is used to interact with outside sending data coming from the Microcontroller. This port sends the power of the signal obtained from the Goertzel Algorithm.

The uses of this GPIO is 16-bits width output port. Therefore, there is no need for an interrupt signal thus the *Enable Interrupt* is not selected. Only one channel of this GPIO is used therefore the *Enable Dual Channel* also not selected. The default value of the port is set to zero.

6.2.1.2. AXI GPIO 1

The GPIO 1 is used to get data from the button and send data to LED. These two components are used to know if the Microcontroller is functioning. The application consists of; when a button is pulsed, a LED is illuminated. This application is developed in the Microcontroller thus to know the correct functioning of Hardware and Software of Microcontroller.

In this version of thesis two push buttons and two RGB LEDs are used to do this application. However, when this application is will implement in Silicon, the application is worked using one push button and one LED, or will be removed.

The GPIO 1 uses two channels so *Enable Dual Channel* is set. The push button signal is obtained from channel 2 and the signal is sent by channel 1 to illuminate the LED.

The microcontroller does not pay attention to this GPIO unless it receives the interrupt, which is generated by GPIO when the button is pushed.

6.3. AXI BRAM Controller

AXI BRAM Controller is designed to communicate the system master device with local Block RAM. The core could be configured as a single port or both ports to connect with the BRAM Block.

The AXI BRAM Controller IP could be configured with ECC functionality with an available external ECC register set via second AXI port. The Error Correction Codes (ECC) is used to mitigate the effect of BRAM single Event Upsets. ECC bits are generated when writing to the block RAM and stored together with the written data. The ECC bits are used to correct any single bit errors and detect any double bit errors in the data when the block RAM is being read. You could read [6] for more detail about this function.

Five-channel AXI interface is used to perform all communication with the AXI master device. The write operation is initiated on the Write Address Channel (AW) which specifies the type of write transaction and the information address. The Write Data Channel (W) is used to communicate all write data. The Write Response Channel (B) is used to respond to the operation.

The Read Address Channel (AR) communicates all address and control signals on the master device request. The Read Data Channel show the data and status of the operation when the read data is available to send.

Figure 6-5 shows the Block Diagram of used Bram Controller.

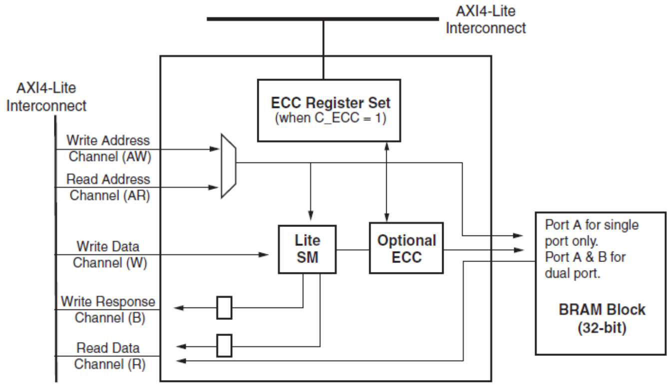


*Figure 6-5: **AXI BRAM Controller Block Diagram. Extract from** [6].*

The channels definition of AXI BRAM Controller IP is explained in *Table A-19*.

6.3.1. Utility

AXI BRAM Controller IP is used to communicate Memory Generator Block with Microcontroller Cortex M3. This block is connected through AXI Interconnect and communicate with this as Master device.

32-bits memory data is used which need 13-bits address. The AXI BRAM Controller is configured as a single port with ECC disabled.

The AXI BRAM Controller is shown in *Figure 6-6*.

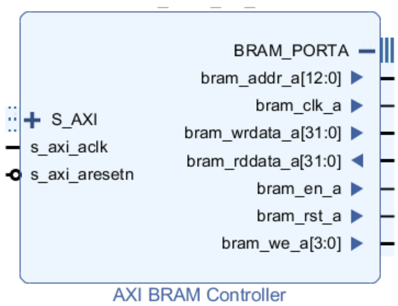


*Figure 6-6: **AXI BRAM Controller IP***

6.4. Block Memory Generator

The Block Memory Generator core is a memory constructor which generates optimized memories using block RAM resources in Xilinx FPGA.

The core has two fully independent ports each of them with write and read interface. These four ports can be configured with individual width.

The Block Memory Generator supports both Native interface and AXI4 interfaces, which has industry-standards bus protocols.

6.4.1. Memory Core configurations

6.4.1.1. Memory types

The core can be configured to generate five types of memories:

- *Single-port RAM:* allows Read and Write access through a single port.
- *Simple Dual-port RAM:* Provides two ports, A and B. The port A is used for Write access and port B used for Read access.
- *True Dual-port RAM:* provides two ports, A and B, each of them allows Read and write access to the memory space.
- *Single-port ROM:* allows Read access to the memory space through a single port.
- *Dual-port ROM:* allows Read access to the memory space through two ports.

6.4.1.2. Connecting algorithms

Three different algorithms are available to connect block RAM primitives to generate the core:

- *Minimum Area Algorithm:* The memory is generated using minimum numbers of block RAM primitives. Figure 6-7 shows an example of this.

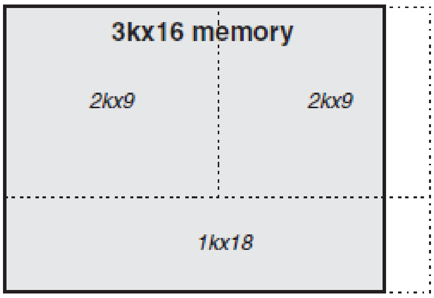


Figure 6-7: ***Examples of the Minimum Area Algorithm. Extract from*** [25]

- *Low Power Algorithm:* The memory is generated such that the minimum number of block RAM primitives are available during a Read or Write access. *Figure* 6-8 shows an example of this.

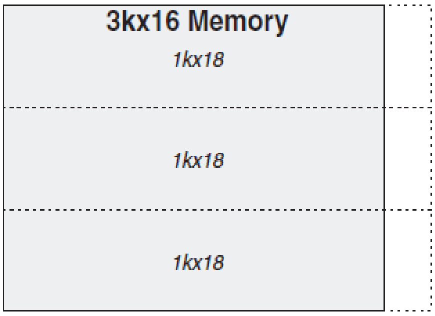


Figure 6-8: ***Examples of the Low Power Algorithm. Extract from*** [25]

- *Fixed Primitive Algorithm:* The memory is generated using only one type of Block RAM primitives. *Figure 6-9* shows an example of this.

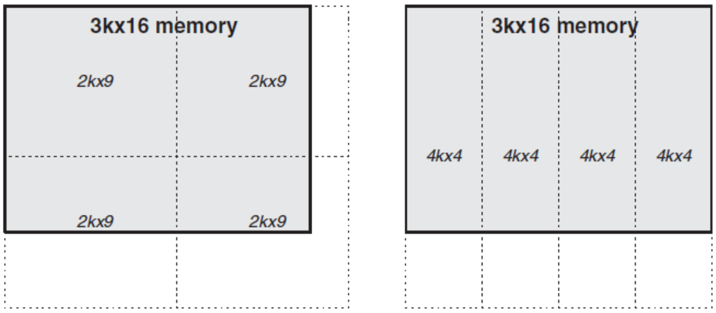


Figure 6-9: ***Examples of the Fixed Primitive Algorithm. Extract from*** [25].

6.4.1.3. Operating mode

Each port can be assigned one of the following operating modes:

- *Write first:* the input data is simultaneously written into memory and driven on the data output.
- *Read first:* the previous write address data is stored on the data output, while input data is being saved.
- *No change mode:* the output data is not changed during a Write operation.

6.4.1.4. Others configuration enables

- *Selectable Port Aspect Ratio:* The A port width could differ from port B by the factor of 2^n where n is between 0 to 5. The Read width could differ from Write by the same factor. The maximum ratio between any two of the data widths is 32:1.
- *Optional Byte-Write Enable:* The core provides byte-Write enable for memory width which is multiple of eight bits or nine with parity. This enable is used to select the specific byte or bytes from input data to save in memory. The byte-Write enable bus is N bits width, where N is the number of bytes in data input. The MSB bit of byte-Write Enable allows the MSB byte of the input data.
- *Core Output Register:* the core provides two optional output registering to increase memory performance, one for each port.
- *Optional Enable Pin:* the core provides an optional port enable pin (one for each port) to control the memory operation.
- *Optional Set/Reset Pin:* the core provides optional set-reset pins (one for each port).
- *Hamming Error Correction Capability (ECC):* The memory could automatically detect single- and double-bit errors when the core type is simple dual port RAM and the data width is greater than 64 bits.

The ports description is described in *Table A-20*.

6.4.2. Utility

There is only one Block Memory Generator is used to extend Code part of internal Microcontroller memory (see *5.4. Memory map*). This Block Memory Generator is used as RAM.

The Single Port RAM type of memory is used to create this core. This used the Minimum Area algorithm to connect Block memory primitives. The core work in Write First operating mode.

As is mentioned, Read and Write port have same width of 32-bits. The byte-Write enable is set and it has 4-bit width because the input data is formed by 4 bytes (8-bit byte size). The port Enable pin also used to control the memory operation. The Hamming Error Correction Capability single is disabled.

The Memory Core Generator is shown in *Figure 6-10*.

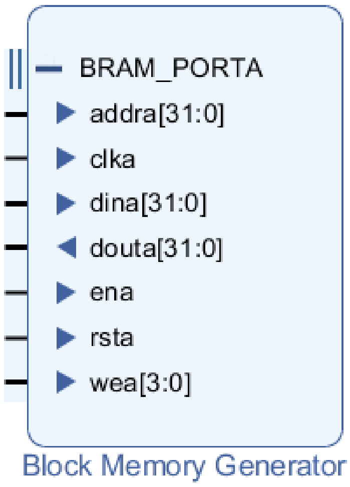


Figure 6-10: ***Block Memory Generator IP***

6.5. Processor System Reset

The Processor System Reset is a core used to handle the reset conditions of the system. The core handle numerous reset conditions at the input and generates appropriate reset at the output, which are based on input or internal conditions.

The core receives some parameters to establish the output signals. The External Reset Active Width and Aux Reset Active Width signals are used to determine the minimum width of the external and auxiliary reset signals with respect to internal clock. The input reset signal has to be stay active for at least the number of External Reset Active Width or Aux Reset Active Width before a reset is initiated. External Reset Logic Level and Auxiliary Reset Active Level signals are used to set active level of input resets.

External Reset Active Polarity and Auxiliary Reset Active Polarity is used to set reset when external reset input or auxiliary reset input is active, respectively.

Bus Structure (Active-High) and Interconnect (Active-Low) are used to select additional numbers of Bus Structure Reset and Interconnects. Peripherals (Active-High and Active-Low) is used to select additional numbers of Peripherals resets. The width of this signal complies to the same width requirement as for External Reset Active Width.

The Ports description is shown in *Table A-23*.

6.5.1. Utility

There are two Processor System Reset used to create resets for Microcontroller, Peripherals (GPIOs), AXI Interconnect and AXI BRAM Controller. See *Figure 6-15* Clocks and Resets diagram for more detail.

External Reset Logic Level and Aux Reset Logic Level are established active-Low (binary 0). Only one width is selected for all external resets (Active Width of signals).

The Processor System Reset IP is shown in *Figure 6-11*.

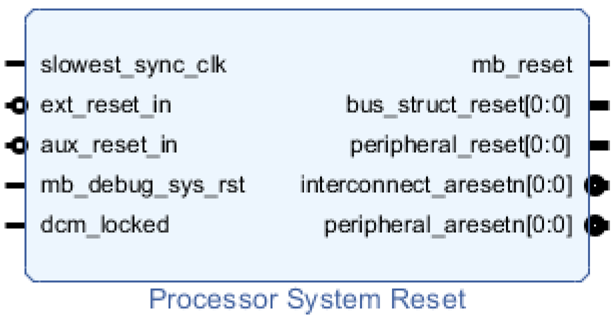


Figure 6-11: ***Processor System Reset IP***

6.6. Clocking Wizard

The Clocking Wizard core generates a clocking network matched to specific requirements. The core helps to create the clocking circuit for the required output frequency, phase and duty cycle. The core uses Mixed-Mode Clock Manager (MMCM)(E2/E3) primitives, which is used to generate several clocks with different frequencies but with dependency on input clock, or Phase-locked loop (PLL)(E2/E3) primitives, which generates an output signal which phase is related to the input signal phase.

The core can be configured to have two input clocks. The core uses input clocks parameters to configure the output clock. The wizard allows specifying the input clock jitter[2] either in PS (picosecond peak-to-peak) or UI (one-bit time) units.

The core has maximum output clocks numbers which depend upon the selected device or primitives. The maximum number of clocks is seven for MMCM (E2/E3), are six for PLLE2 and two for PLLE3. Seven maximum output clocks can be configured if the primitives selected is Auto. The clocking wizard set and configure the clocking primitives and network automatically receiving desired input timing parameters (frequency, phase and duty cycle).

6.6.1. Clocking Options

In this section, they are described some features and configuration parameters to create the Clocking Wizard. Some features may consume additional resources, and some can result in increased power consumption. Furthermore, certain combinations of features are not allowed.

[2] Jitter is the unwanted variations of bits in a digital signal. Most of them are caused by noise picked up from a phase-locked loop (PLL).

- *Enable Clock Monitoring:* This feature allowed to monitor the clocks in a system. The Clock Monitor can detect the change in clock frequency, a glitch[3] in the clock or a stop clock.
- *MMCM or PLL:* this option is used to select either MMCM primitives or PLL primitives.
- *Frequency synthesis:* this feature is enabled when output clocks frequencies needed to be different active input-output clocks.
- *Phase alignment*: this feature allows the output clock to be phase locked to the reference.
- *Dynamic reconfiguration:* this feature allows to programme the primitive after the device is configured, through AXI4-Lite interface.
- *Safe Clock Startup and Sequencing:* this feature allows to generate stable clock at the output using BUFGCF after *Locked* is sampled High for 8 input clocks. The delay between two enabled output clocks in a sequence is 8 cycle of the second clock in the sequence clock.
- *Minimize power:* This feature minimizes the amount of power needed for the primitives. This feature is not available when the *Spread Spectrum (SS)* option is selected.
- *Spread Spectrum (SS):* This feature provides modulated output clocks which reduces the spectral density of the electromagnetic interference (EMI) generated by electronic devices. this feature is only available for MMCM (E2/E3) primitives.
- *Dynamic phase shift:* This feature allows changing the phase relationship on the output clocks. This feature is not available when the *Spread Spectrum (SS)* option is selected.
- *Balanced:* this feature selects balanced results in the software choosing the correct Bandwidth for jitter optimization.

[3] Glitch is a transition that occurs on signal before the settlement of this to intended value. In clock signal this can cause an asynchronous behaviour or unstable data.

- *Minimize output Jitter:* this feature minimizes the jitter on the output clocks but consumes more power and may cause possible errors on the phase. This feature is not available when *Maximize input jitter filtering* option is chosen.
- *Maximize input jitter filtering:* This feature allows to filter larger input jitter on the input. This feature is not allowed when *Minimize output jitter* option is chosen.

The ports description is showed in *Table A-24*.

6.6.2. Utility

The clocking wizard is used to create the system clock. The input clock of this Wizard is 100 MHz (Xilinx Nexys 4DDR board used a quartz crystal of 100 MHz) and the system clock has 50 MHz frequency.

In this case, controlling wizard is not relevant therefore *Enable Clock Monitoring* is disabled. To create output clock MMCM primitives are used. Since the output clock frequency is different from input clock frequency the *Frequency Synthesis* option is enabled and *Phase Assignment* also enabled so to lock the phase. Furthermore, *Safe Clock Start-up* is enabled.

The *Balanced* options are used to choose the correct Bandwidth of Jitter Optimization.

In the Output Clock option, two output clocks are enabled, one as a system clock and another auxiliary clock which could be used for other purposes. 50 MHz is chosen as the frequency of both output clocks. *Automatic Control On-Chip* is enabled to create source.

The MMCM/PLL option is locked so Wizard would choose automatically these options; such as Bandwidth, RIVCLK1_PERIOD, REF_JITTER1, etcetera.

The Clocking Wizard is shown in *Figure 6-12*.

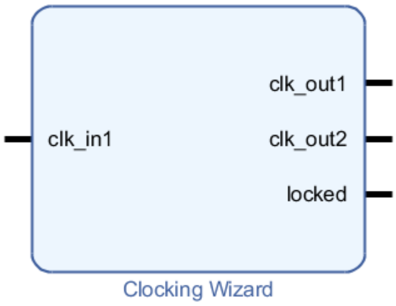


*Figure 6-12:**Clocking Wizard IP***

6.7. Utility Vector Logic

The Utility Vector Logic core is used to create logic functions. The core support bitwise AND, OR, XOR and NOT functions. The core gets two input ports, except for NOT operation in which case only one input is needed, and an output port. The width of these ports has to be the same, and the minimum width has to be one.

The Utility Vector Logic core is used to make AND, OR and NOT logic operations. For more detail about the function of this core, see *6.11. ADC diagram* and *6.12. Clock and reset diagram*.

6.8. Constant

The constant core used to drive a constant value in the output. The constant value is could express in Decimal, binary (b at the beginning), octal (O at the beginning) and hexadecimal (0x or 0X at the beginning). The core only has one output port, which could have 4096 as maximum port width.

These constant cores are used to set different signals.

6.9. Concat

The Concat core is used a mechanism to combine bus signals of varying width into a signal bus. The core allowed to select maximum 32 input ports, each of them could have 4096-bits width. The output width is selected automatically with inputs width.

The functionality of the Concat block diagram is shown in *Figure 6-13*.

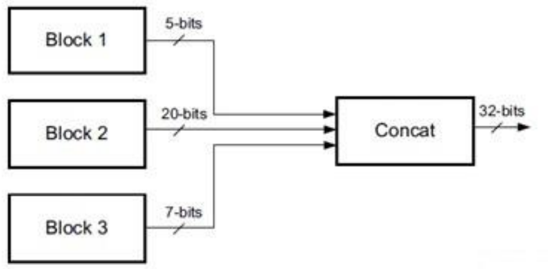


*Figure 6-13: **Concat block diagram. Extract from** [27].*

6.10. Slice

The slice core is used to select some specific bits from the input bus. The core allowed to select input width and from which width to which width wants to select from input data.

6.11. ADC diagram

The function of this diagram is to connect the XADC Wizard with AXI GPIO and generate the Echo signal when an enable is received. This diagram consists of XADC Wizard, AXI GPIO, FF_D_T, Square Generator, Concat, Slices and Utility Vector Logic.

The ADC diagram is shown in *Figure 6-14*.

6.11.1. AXI GPIO

The GPIO 2 is used to configure the ADC or obtain converted analogic data from it.

Each channel of the GPIO is could be configured, by Microcontroller, as input or output, because both ports, input and output, of each channel are connected to even send data or receive data.

The channel one of GPIO is used to obtain or send control signals. When this port sends data, the bus consists of; Square Generate signal *Enable* ($SIGNALS[9]$)

(see *6.11.4. Square Generator*), the signal *GOERTZ_STARTED* signal (SIGNALS[8]) (see *6.11.3. FF_D_T*), write enable signal ($SIGNALS[7]$) and desired ADC registers address ($SIGNALS[6:0]$). Four slices are used to separate these signals and, send to the corresponding block.

When the channel one of GPIO port is configured as input, it obtains different signals from ADC. In this case, bus $\mathrm{SIGNALS}[9:0]$ is formed by *Data Enable* signal ($SIGNALS[7]$), BUSY signal ($SIGNALS[6]$) and Channel Out bus ($SIGNALS[4:0]$) (see *4.1. ADC channels definition*). Other bits are not used but they are necessary to have the same width as input bus. These bits of SIGNALS are set to zero. One concat element is used to combine this signal in one port. The connection of this concat is shown in *Figure 6-14*.

The channel two of GPIO is used to receive and send data. When the channel is configured as input, the GPIO receives data from FF_D_T block. In FF_D_T block the converted data is saved (see *6.11.3. FF_D_T*). Otherwise, this port sends data, which have to be saved in ADC registers (see *4.2. ADC registers*). The width of this channel is 16 bits.

The interruption of the GPIO is enabled to obtain data when an acoustic signal is received.

6.11.2. XADC Wizard

The XADC Wizard is the element which gets analogic input and is configured to obtain digital data. (see *4.Analogue to digital converter (ADC)*).

The core has enabled two input analogic channels, one of them is configured to bipolar input. The ADC uses DRP interface and it works in continuous sampling mode.

Input clock of ADC is same as the system clock (50 MHz) and configured to obtain 80 kilo-Samples (KSPS). The reset of this core is active-High, but the system reset is active-Low. Therefore, the Utility Vector Logic core is used to covert active-Low to active-High. This core is configured to make a NOT logic operation thus to reset the core properly.

Other ports connection are described in *6.11.1 AXI GPIO*.

The connection of this core is shown in *Figure 6-14*. Alarm outputs ports ($user\_tempe\_alarm\_out$, $vccint\_alarm\_out$, $vccaux\_alarm\_out$, ot_out and $alarm\_out$) are not connected because this are not useful for the modem. The eos_out pin is also not used because only one input port is used at the time to convert data.

6.11.3. FF_D_T

In Vivado there are no Flip Flop component IP however it can be described using Hardware Description Language and packed as a block diagram.

The FF_D_T block is generated using Verilog HDL which includes one D Flip Flop and one T Flip Flop.

The D Flip Flop obtain data from ADC wizard and when the Data Ready (DRDY) signal is active, save the data in the register. Then the saved data is sent to the microcontroller through GPIO. The input and output bus have the same width which could be selected from external however the predefined width is 16 bits.

The T Flip Flop obtain the Data Ready (*DRDY*) signal and Goertzel Started (*GOERT_STARTED*) signal to determine the Data Enable (*DATA_ENABLE*) signal. When there is a rising edge of Goertzel Started signal that means the Goertzel Algorithm has started the calculation and no data is needed. Hence the Data Enable signal is set to logic-Low. When the Goertzel Started is logic-Low and there is a Data Ready signal, the Data Enable signal is set to logic-High.

The Data Enable is created because the DRDY only is active one clock and the microcontroller could not have time to catch the signal and thus the signal is disabled during the Goertzel calculation so not to make an error with input data.

The core ports description is shown in *Table A-21*.

6.11.4. Square Generator

The Square Generator block, as is named say, is a block that generate a square digital signal. The block consists of two counters; one counter is used to obtain desired output signal frequency, and the second one is used to determine output signal timing.

When the input Enable signal has a rising edge the two counters start counting. The frequency counter gives signal every half period of output Signal thus to set logic-High or logic-Low the output signal (the signal begins with logic-Low). And when the timing counter has achieved the desired value the output signal goes permanently logic-Low unless there is another input Enable. If during the timing counting, Enable signal goes logic-High, this is ignored until the counter achieved the objective value.

The core pins are described in *Table A-22*.

6.12. Clock and reset diagram

The function of this diagram is to create appropriate clocks and resets for different elements of the modem, such as AXI Interconnect, Cortex M3, etcetera.

The design receives the input clock (sys_clock) and the reset (sys_reset_n) signal from Nexys 4 DDR, and auxiliary reset signal ($sysrestreq$) from Cortex M3. The clock signal is received by clocking wizard to generate system clock.

Individual Processor System Reset gets the each of reset signals to generate output resets. Three resets are generated with these cores: one reset signal is for peripheral; one reset signal is for interconnect, and the last reset signal is for system general reset. Furthermore, the core generates a reset for debug port.

The diagram is shown in *Figure 6-15*.

This page intentionally left blank

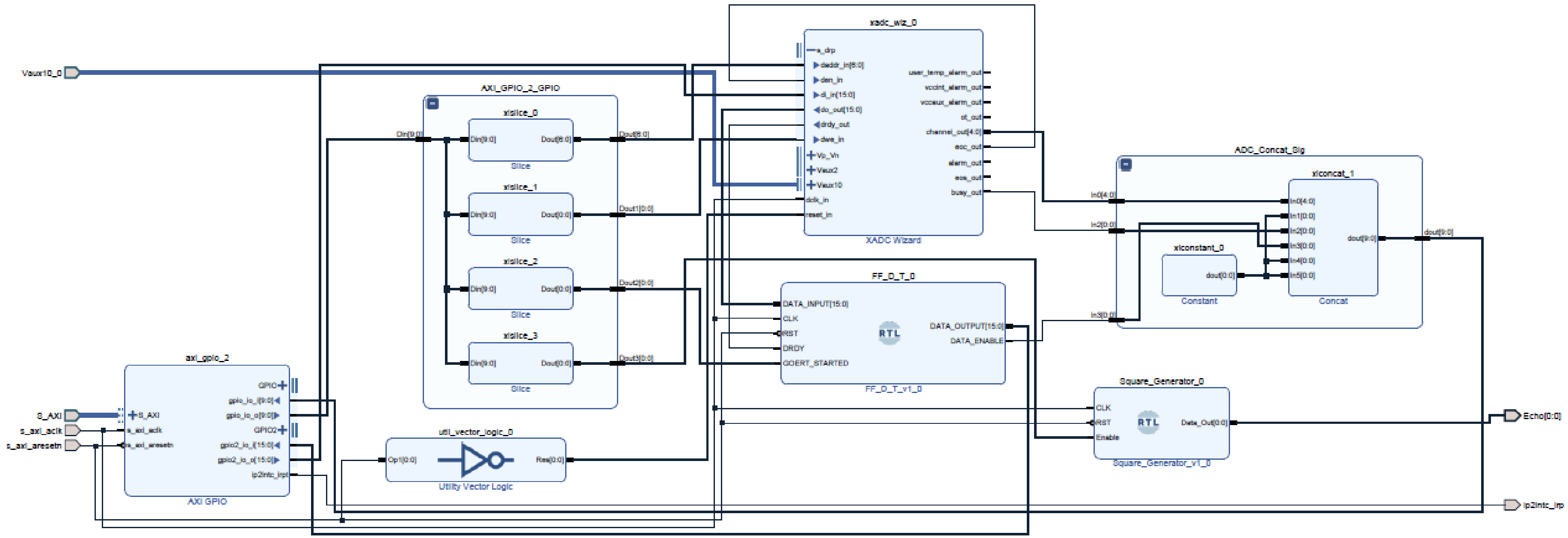


Figure 6-14: ***ADC Diagram with all connection. Note: for better view visit Annex Schematic.***

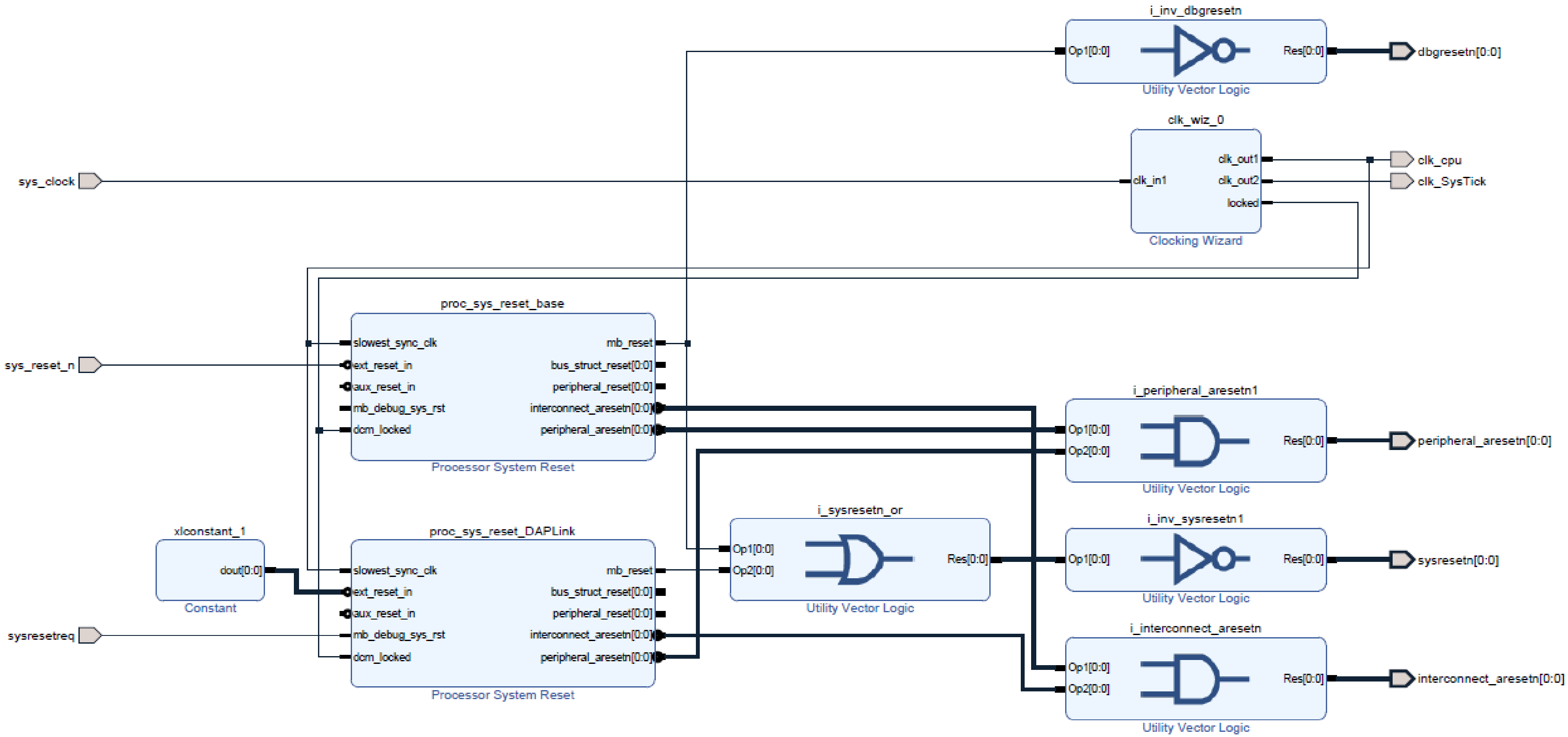


Figure 6-15: ***Clock and reset diagram with all connection. Note: for better view visit Annex Schematic***

7. RESULTS

The modem is divided into two main stages to calculate whether the input signal frequency corresponds to the defined frequency.

In the first stage, the input analogue signal is obtained by the ADC, to convert the signal in 16-bit digital data. This data is saved in a register when the DRDY signal is enabled. The Data Enable signal is generated based on DRDY and Goertzel Started signal. The hardware block diagram is shown in Annexe Schematic.

In the second stage, the microcontroller obtains data from the register and the Data Enable through GPIO. When the Data Enable is logic-High, the input data is compared to Error Setpoint (see *7.2. Error Setpoint*), and if the data value is upper then Error Setpoint, the Goertzel Algorithm is begun. Otherwise, the microcontroller obtains data and compare with Error Setpoint.

At the starting of the Goertzel Algorithm, the Goertzel Started signal is set to logic-High. This is to avoid obtaining more data during the calculation of the Algorithm. After the calculation is done, the Goertzel Started signal is set to logic-Low. The algorithm needs n number of samples to calculate the power of the input signal. This means to save data in the memory before the algorithm starts, which could take more time to the microcontroller to calculate and send the Echo. Hence, the data is obtained in real time to do the algorithm using a loop. And in the end, the power of the signal is calculated.

After all, the calculation is finished, the power of the signal is compared with Power Setpoint (see *7.3. Power Setpoint*). In case the value of the power is more than the Power Setpoint value, the Echo Enable signal is set logic-High if the Signal Enable was logic-High. If not, the Echo Enable is set logic-Low. The Square Generator core receives the Echo Enable signal to generate the Echo signal of input frequency for 5 ms. Although the Echo generated is a square signal, it could be converted in sinusoidal using filter.

Ten kilohertz processed signal frequency is established because of the limitation of the microcontroller (see *7.1. Maximum frequency detection*).

It has studied set the timer before the Goertzel Algorithm starts. This could be used to generate the echo in establish time. Thus, the broadcaster device could know the exact modem calculation time and will calculate the position of the device with precision. When the timer has achieved the value established time and the signal corresponds to the 10 kHz frequency, the Echo Enable is could set logic-Low. It did not archive this functionality because of lack of time.

7.1. Maximum frequency detection

The algorithm is designed to detect 10 kHz input signal frequency. This is because the microcontroller cannot process in real time a lot of samples.

With experimentation, it was observed that the maximum sampling acquisition frequency that microcontroller can support with proper function, is 80 kilo-samples. And an additional Enable is generated to get data without mistake.

It was observed that with this sampling rate the maximum accurate frequency that could obtain is 18 kilohertz. In the *Figure 7-2*, the power is shown when the input signal has 18 kHz frequency and this is the frequency which is required to detect. *Figure 7-1* shows the power when the input signal has 20 kHz frequency and this is the frequency which is required to detect.

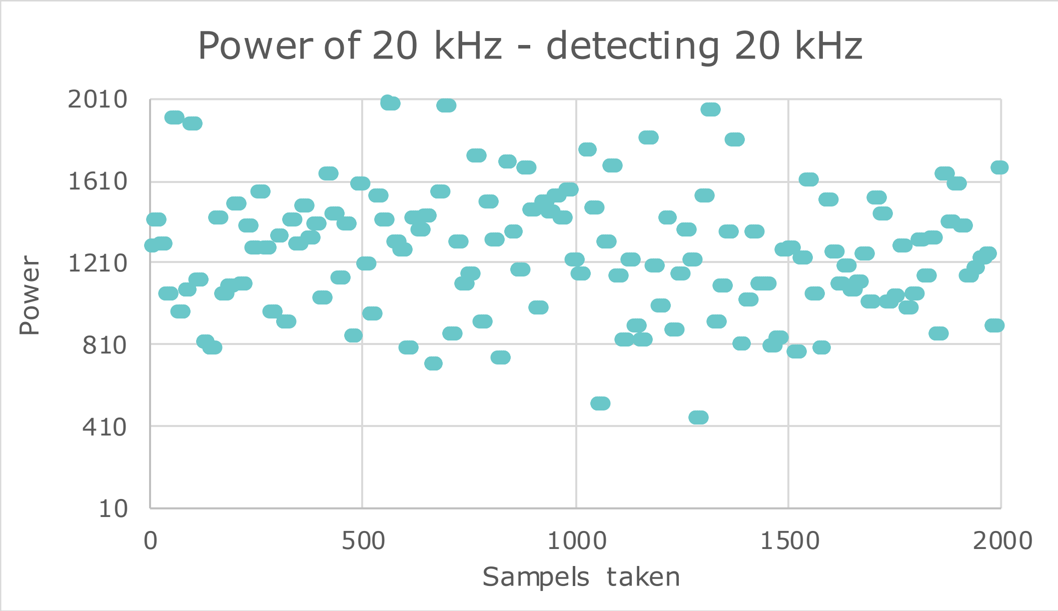


Figure 7-1: ***Observing the power of input signal of 20 kHz frequency when 20 kHz frequency is attempted to detect.***

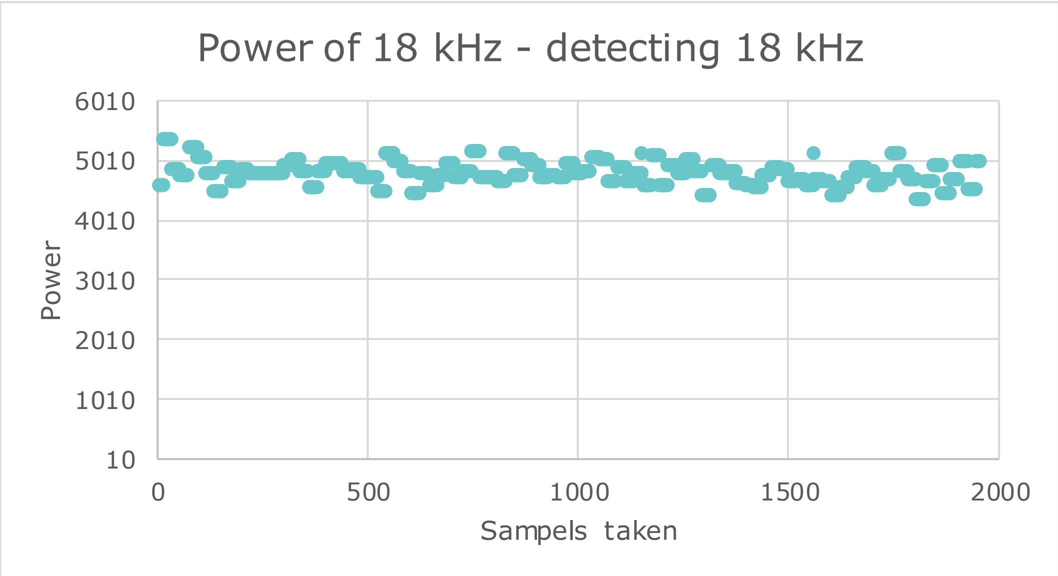


Figure 7-2: ***Observing the power of input signal of 18 kHz frequency when 18 kHz frequency is attempted to detect.***

It can observe in *Figure 7-1* that with 20 kHz the power is not significant compared to 18 kHz shown in *Figure 7-2*. Moreover, the power of 20 kHz frequency has a big range of values on the other hand the power of 18 kHz has a small range, from 4350 to 5350.

Finally, to detect 69 kHz, the sampling frequency has to be at least two times the input frequency, that is to say, the sampling frequency at least has to be 138 kHz -Nyquist rate[4]-. Hence, it is not possible with this system to detect 69 kHz. However, a microcontroller with better performance could detect higher frequencies than 18 kHz.

Thus, to observe the functionality of this modem, 10 kHz frequency has been chosen.

7.2. Error Setpoint

The Error Setpoint is the previous condition which has to be satisfied to start the Goertzel Algorithm. This control is used to avoid noise and signal which could not have enough power to be the detected-signal. And if the signal does not satisfy the condition, the algorithm is not calculated.

To establish the value of the Error Setpoint, it has been observed noise data. The noise signal is shown in *Figure 7-3*, the analogue form and in *Figure 7-4* the digital form.

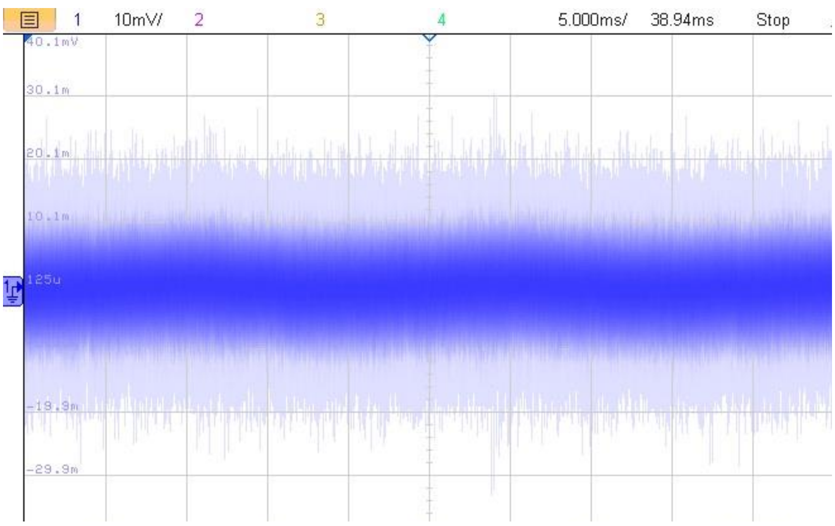


Figure 7-3: ***Noise in analogue form. Time 5ms/div. Voltage 10 mV/div.***

[4] A signal of finite bandwidth W Hz may be completely recovered from a knowledge of its samples taken at the rate of 2W per second [8]. This is known as Nyquist Rate.

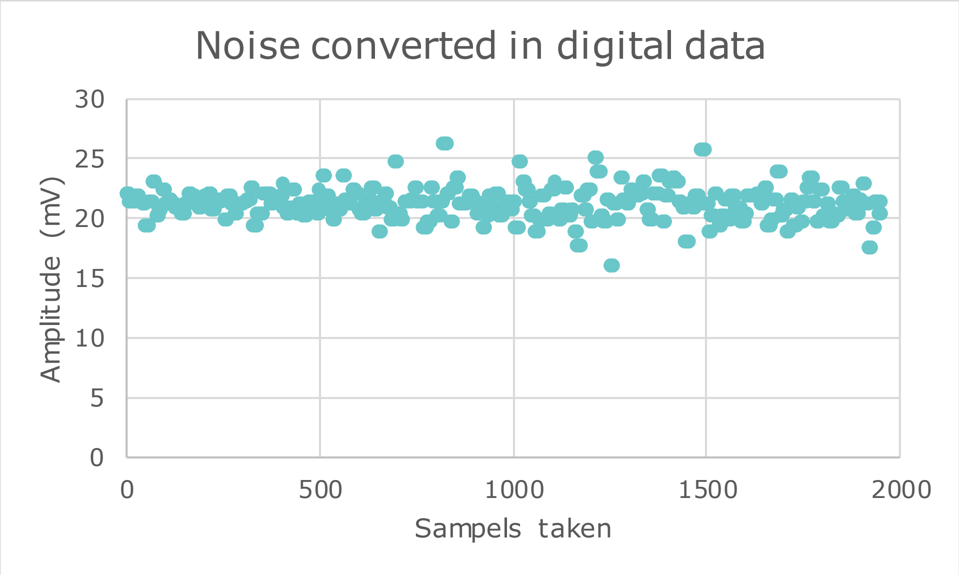


Figure 7-4: ***Noise converted in digital data.***

It could observe that the maximum value of the noise is 26 mV. Therefore, double of this value is chosen as Error Setpoint.

7.3. Power Setpoint

This parameter is used to establish whether the input signal's power corresponds to the 10 kHz sinusoidal frequency or not. To establish the value of this parameter, power of different frequencies -especially 10 kHz- has been observed to determine the limits that separate the correct signal and others. The magnitude of the Algorithm output depends on the input signal amplitude. Hence, the input amplitude is set to 200 mV.

The *Figure 7-5* shows the power of the 10 kHz frequency.

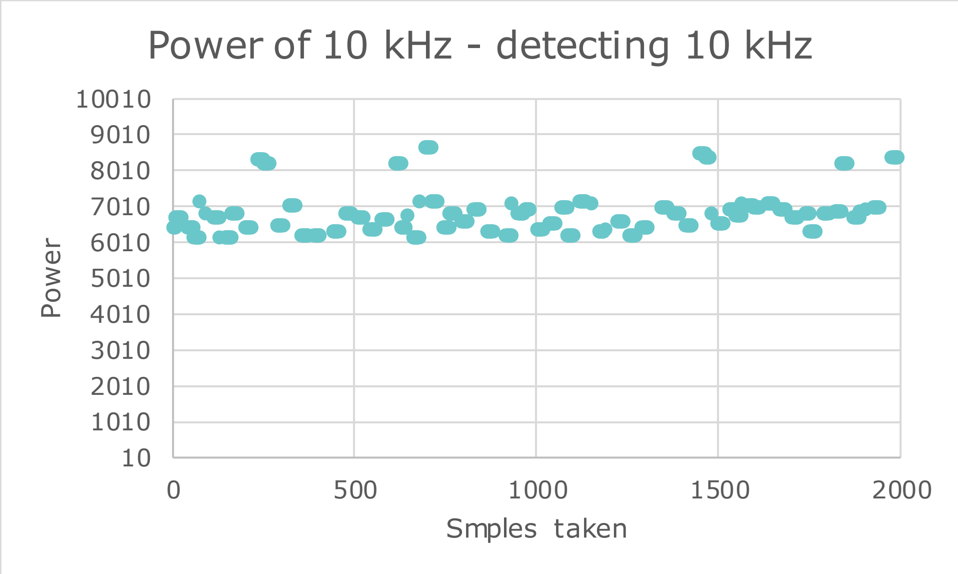


Figure 7-5: ***Observing the power of input signal of 10 kHz frequency when 10 kHz frequency is attempted to detect.***

As it could observe the power for this signal is more than 6000. And establishing this as Power Setpoint the signal of 10 kHz frequency will be detected. The Echo only will be generated when the power of the signal is higher than Power Setpoint.

The *Figure 7-6* shows the power of input signals with different frequencies. It can observe that power is not higher than Power Setpoint.

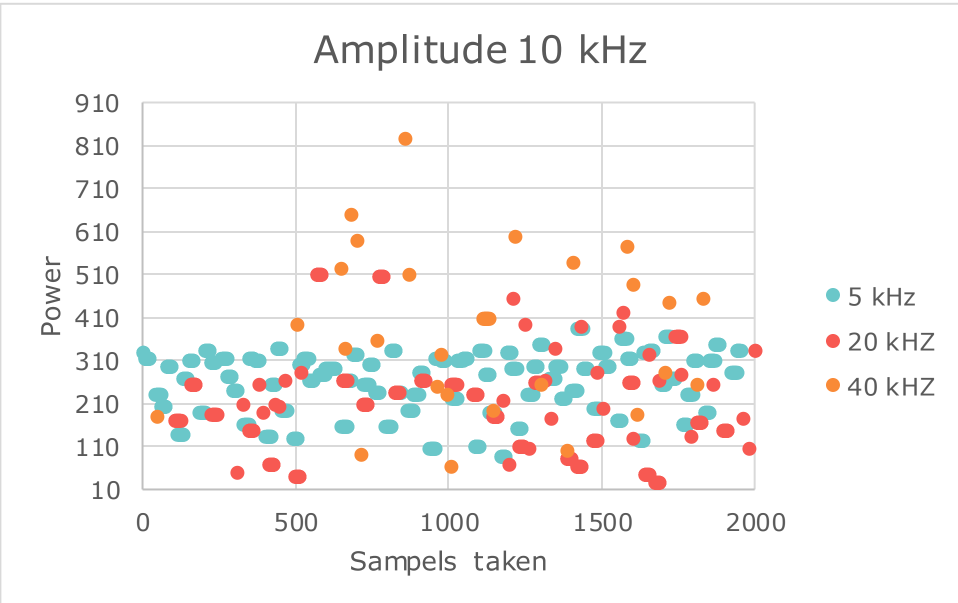


Figure 7-6: ***5, 20 and 40 kHz input signal frequencies power.***

The Power Setpoint limited the detection of the signal. The figure shows the power of frequencies near 10 kHz. It is chosen 9.8, 9.9, 10, 10.1 and 10.2 kHz frequencies to observe when the echo will be sent.

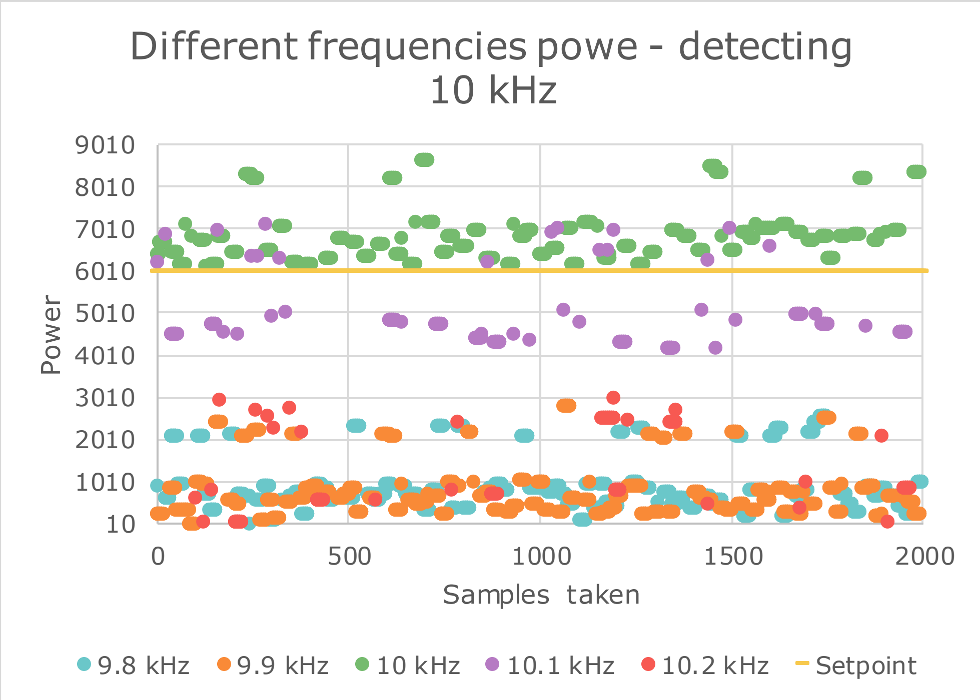


Figure 7-7: ***Power of different frequencies close to 10 kHz.***

It is observed that the limit of activation is from 10 kHz to 10.1 kHz because with these input signal frequencies the power is upper than Power Setpoint. Other frequencies do not have enough power to activate the Echo signal.

7.4. Performance

The modem function correctly establishing the input frequency to 10 kHz with input signal amplitude of the 200 mV. In the following graphics, the modem performance is shown with different input frequencies.

The figure shows 5 ms of a sinusoidal signal, in the blue, activation every 100 ms and there correspond echo, in red, of square signal with a duration of 5 ms.

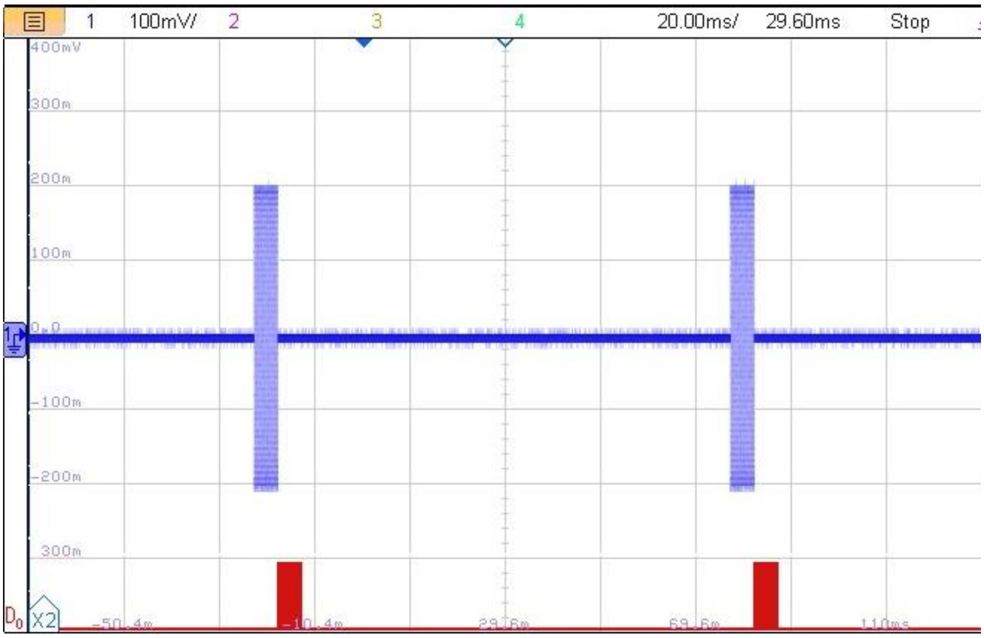


Figure 7-8: ***In blue the input signal of 10 kHz, amplitude of 200 mV and activating the signal every 100 ms. In red the Echo sent by modem. Time 20 ms/div. Voltage 100 mV/div.***

In the figure… the signal is shown to observe in detail.

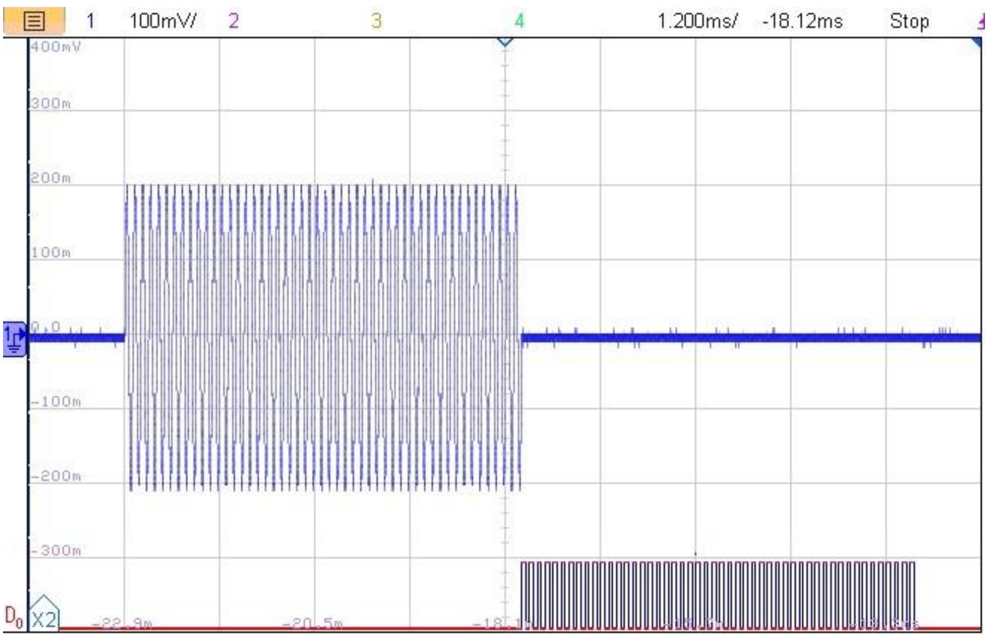


Figure 7-9: ***In blue, input signal of 10 kHz, amplitude 200 mV, sinusoidal and duration of 5 ms. In Red the echo send by modem; square signal of 10 kHz and 5 ms of duration. Time 1.2 ms/div. Voltage 100 mV/div.***

As it mentioned, it has studied timer to send echo in established time but this did not work correctly and it is not included in the modem. Nevertheless, because calculation in real time, the time of calculation is very little, as it could see in the figure. In this figure, it is shown the delay between the ending of the input signal and sending the echo. This time depends on the microcontroller instruction processing time.

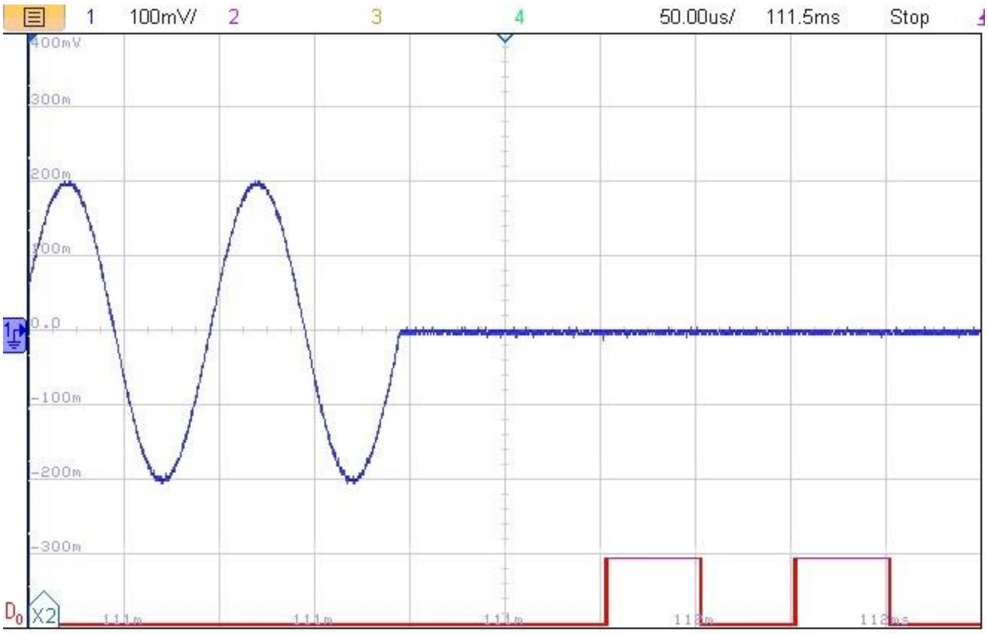


Figure 7-10: ***Observing the delay between input signal and Echo. In blue, input sinusoidal signal of 10 kHz and amplitude of 200 mV. In Red the echo send by modem square signal of 10 kHz. Time 50 us/div. Voltage 100 mV/div.***

It can observe that there are 112 μs of delay between the input signal and the Echo.

7.5. Resources

As is mentioned in objective, one of the main objectives is to develop a compact modem using the minim of the resources. A following it is shown the resources of FPGA that are needed to develop the modem.

Resource	Utilization	Available	Utilization %
LUT	14442	63400	22.78
LUTRAM	36	19000	0.19
FF	5967	126800	4.71
BRAM	18	135	13.33
DSP	3	240	1.25
IO	32	210	15.24
BUFG	3	32	9.38
MMCM	1	6	16.67

Figure 7-11: ***Table of resources used and resources available in Nexys 4DDR board.***

The figure shows the percentage of the FPGA resources used.

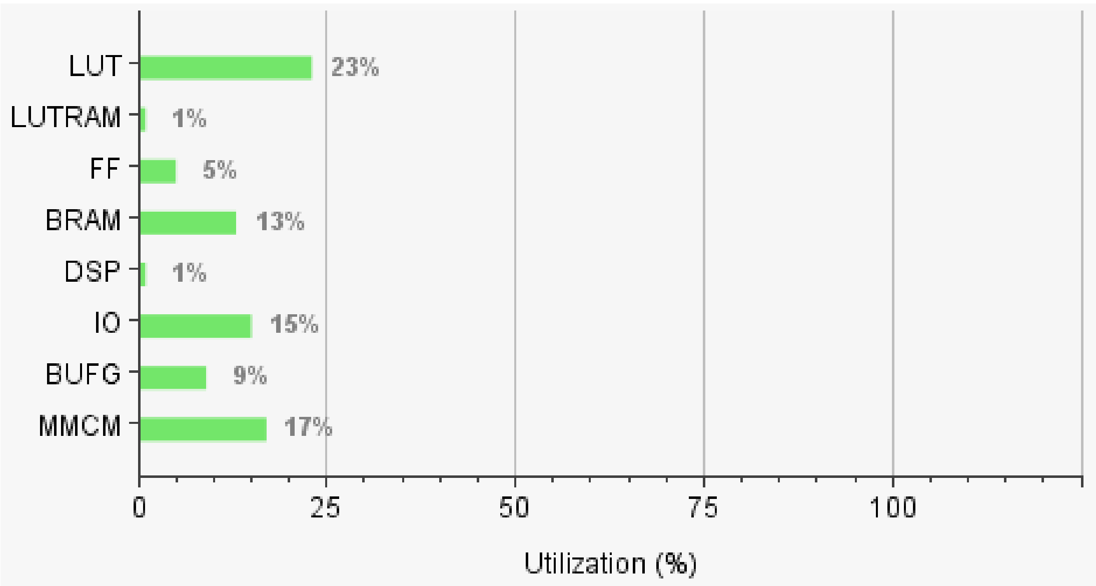


Figure 7-12: ***Graphic showing the percentage of the board resources used.***

It can observe that about 23 per cent of Look Up Tables, and about 5 per cent of Flip Flop are used to develop the modem. And approximately 14 per cent of memory is used. Hence, the modem is developed using very few resources.

8. FUTURE WORK

In this thesis, an acoustic modem using a synthesize microcontroller has been developed. The core has developed in FGPA to observe the functionality of the modem. And during their developing, it has observed some limitations using this specific microcontroller.

It has observed that the microcontroller with Cortex-M3 core has data acquisition limits, which limits the detection of the higher input signal frequencies. This could be improved using a microcontroller with better features and increasing the input Clock frequency.

This investigation work is the previous stage to develop a modem in a silicon chip. But before generating the chip, the Goertzel Algorithm could be used as Digital Filter thus the microcontroller could not need to process the information and detect the tone of the signal. And the microcontroller could be used to do other jobs. Hence, the next stage of this thesis would be developing the silicon chip with Digital filter to improve modem functionality and with filter, the modem could not have the input signal frequency limitations.

This page intentionally left blank

9. CONCLUSION

Developing the modem has consisted of the main objectives set from the beginning. There was a clear idea, develop a modem which will be used to track underwater species. Use of the microcontroller will allow the detection of the communication signal. The processor and other parts will be created using Hardware Description Language and this way the modem will be compact.

Because of being part of a big project, the thesis is limited to create the modem using a microcontroller and assuming that the input signal is electric and adapted to be processed. And the output is a signal that emulates the Echo which will be sent to answer.

The first part was how to get the signal to process with the microcontroller. The idea was, using an Analogue to Digital converter (ADC) the data can be converted in digital. Moreover, it was not necessary to buy a separate component because almost all FPGA boards include an ADC in the board.

The next stage was the connection of these two devices which involves GPIOs, Verilog modules and others IP. The connection has been improved as the thesis is developed.

And the final stage was processing the signal and detect whether the signal corresponds to the specifications or not. Different ideas came out to the surface, such as, use of bandpass filter, or use of matched filter [7]. But finally, it has decided to use the Goertzel Algorithm to detect the tone of the signal. This algorithm is fast and easier to implement in C code.

The design of the system started studying the ADC. It is used XADC predesigned component which performs analogue to digital conversion and with other more functionalities. During this part, some complication has been faced because of the technical problems with the core.

Once the ADC function properly, the next stage was to select the Microcontroller which was described in HDL and could be implemented on FPGA.

At first, the Cortex-M0 and Cortex-M1 was tried to use for modem but it was observed that these have problems to implement in FPGA because of not founding third party parts. Thus, it was decided to implement Cortex-M3, which came as a compact core. Furthermore, the core has AHB to AXI bridge to connect with other Vivado parts with AXI protocol.

Subsequently, the core was implemented in FPGA programming a small pilot program, included with the ARM developer package. To understand how the microcontroller functions, how it connects, and how it is programmed, took some days. When the program functioned correctly, the ADC was connected to the microcontroller.

At this stage, the system was capable to convert the analogue data into digital and the microcontroller got the data to process. When the Goertzel Algorithm began to be implemented, another problem arose. It needed a vector of data to proceed with the calculation. And this could have taken a lot of time for the tone detection. The solution was, modify the Algorithm to work in real time.

It was observed that using Microcontroller Cortex-M3 for real-time detection limits the acquisition frequency of the ADC to 80 kHz. This frequency does not achieve the Nyquist rate to detect 69 kHz. Thus, the detection of the objective input signal, which has 69 kHz frequency, is not possible. However, to demonstrate the correct functionality of the modem, it was proposed the detection of the signal with 10 kHz frequency.

After processing the signal, it was detected the tone of the signal. And when the tone corresponds to 10 kHz frequency, an echo signal is generated with similar characteristics than the input signal.

Completed the modem, it has observed the use of sources to create the modem. And as it could see in the *Chapter 7*, these are very few elements which are used to develop the modem in the FPGA.

Since they are used different programs and there are some procedures to fallow to configure the programs, generate the Digital blocs and create software, a user’s manual is provided in the annexes of this document.

Finally, one of the issues faced at the end of the thesis was to send the echo at the established time. It has used a microcontroller internal timer, configured the timer before the Goertzel Algorithm starts and an interruption occurs when the selected delay time has passed. Thus, to generate the echo two conditions needed to be fulfilled: the input signal corresponds to the desired signal and timer interruption has occurred. It has been studied this functionality nevertheless it has not been achieved due to lack of time.

This page intentionally left blank

10.REFERENCES

[1] Digilent, "Nexys4 DDR™ FPGA Board Reference Manual," 11 Abril 2016. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual. [Accessed 26 May 2019].

[2] B. W. Kernighan and D. M. Ritchie, The C programming language, 2nd Edition, Englewood Cliffs: Prentice-Hall, 1988.

[3] M. Sánchez-Élez, "INTRODUCCIÓN A LA PROGRAMACIÓN EN VHDL," 2014. [Online]. Available: https://eprints.ucm.es/26200/1/intro_VHDL.pdf. [Accessed 22 April 2019].

[4] ARM, "ARM® Cortex®-M3 Processor - Technical Reference Manual," ARM, 24 Febrero 2015. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.100165_0201_00_en/arm_cortexm3_processor_trm_100165_0201_00_en.pdf. [Accessed 18 May 2019].

[5] K. Banks, "The Goertzel Algorithm," Embedded, 28 Agosto 2002. [Online]. Available: https://www.embedded.com/design/configurable-systems/4024443/The-Goertzel-Algorithm. [Accessed 2 Mayo 2019].

[6] Xilinx, "AXI Block RAM (BRAM) Controller v4.0," 5 Octuber 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf. [Accessed 10 May 2019].

[7] J. C. Bancroft, “Introduction to the matched filters,” 2002. [Online]. Available: https://crewes.org/ForOurSponsors/ResearchReports/2002/2002-46.pdf. [Accessed 21 April 2019].

[8] H. J. Landau, “Sampling, data transmission, and the Nyquist rate,” *IEEE Journals,* pp. 1701-1706, 1967.

[9] J. Yiu, The definitive guide to the ARM Cortex-M3, 2nd Edition, Oxfoard: Elsevier, 2010.

[10] E. Monmasson and M. N. Cirstea, “FPGA Design Methodology for Industrial Control Systems—A Review,” vol. 54, 2007.

[11] M. M. Mano and M. D. Ciletti, Digital Design, 5th Edition, Upper Saddle River: Pearson, 2012.

[12] I. M. Rusiñol, "Diseño de sistemas de posicionamiento acústico para la monitorización de especies marinas," UPC, Barcelona, 2016.

[13] J. G. Agudelo, “Contriution to the model and navigation control of an autonomous underwater vehicle,” Julio 2015. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/95737/TJGA1de1.pdf. [Accessed 18 May 2019].

[14] ARM, “ARM® Cortex®-M3 DesignStart™ FPGA-Xilinx edition,” 29 Octubre 2018. [Online]. Available: https://static.docs.arm.com/101483/0000/arm_cortex_m3_designstart_fpga_xilinx_edition_ug_101483_0000_00_en.pdf?_ga=2.220801629.41850125.1559121026-2031830401.1549732769. [Accessed 18 May 2019].

[15] ARM, “ARM®v7-M Architecture - Reference Manual,” ARM, 29 Junio 2018. [Online]. Available: https://static.docs.arm.com/ddi0403/e/DDI0403E_d_armv7m_arm.pdf?_ga=2.186121514.2016275911.1559387162-2031830401.1549732769. [Accessed 15 May 2019].

[16] ARM, “CoreSight™ Components - Technical Reference Manual,” ARM, 10 Julio 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/DDI0314H_coresight_components_trm.pdf. [Accessed 18 May 2019].

[17] ARM, “Cortex™-M1 - Technical Reference Manual,” 7 Mayo 2008. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0413d/DDI0413D_cortexm1_r1p0_trm.pdf. [Accessed 22 May 2019].

[18] ARM, "Cortex™-M3 - Technical Reference Manual," ARM, 13 Junio 2007. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf. [Accessed 18 May 2019].

[19] K. Banks, “The Goertzel Algorithm,” 28 Agosto 2002. [Online]. Available: https://www.embedded.com/design/configurable-systems/4024443/The-Goertzel-Algorithm. [Accessed 2 May 2019].

[20] C. J. Chen, “Modified Goertzel Algorithm in DTMF detection using the TMS320C80,” Texas Instrument, Junio 1996. [Online]. Available: http://www.ti.com/lit/an/spra066/spra066.pdf. [Accessed 25 May 2019].

[21] F. Nacini, “Robohub,” 4 Mayo 2017. [Online]. Available: https://robohub.org/janus-creates-a-new-era-for-digital-underwater-communications/. [Accessed 23 May 2019].

[22] A. Vitali, "The Goertzel algorithm to compute individual terms of the discrete Fourier transform (DFT)," ST microelectronics, Deciembre 2017. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/design_tip/group0/20/06/95/0b/c3/8d/4a/7b/DM00446805/files/DM00446805.pdf/jcr:content/translations/en.DM00446805.pdf. [Accessed 25 May 2019].

[23] Xilinx, "AXI GPIO v2.0," 5 Octuber 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf. [Accessed 10 May 2019].

[24] Xilinx, "AXI Interconnect v2.1," 20 December 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf. [Accessed 10 May 2019].

[25] Xilinx, "Block Memory Generator v8.3," 5 April 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf. [Accessed 10 May 2019].

[26] Xilinx, "Clocking Wizard v5.3," 5 Octuber 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_3/pg065-clk-wiz.pdf. [Accessed 10 May 2019].

[27] Xilinx, "LogiCore Concat v2.1," 6 April 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xilinx_com_ip_xlconcat/v2_1/pb041-xilinx-com-ip-xlconcat.pdf. [Accessed 10 May 2019].

[28] Xilinx, "LogiCORE IP Constant v1.1," 9 April 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xilinx_com_ip_xlconstant/v1_1/pb040-xilinx-com-ip-xlconstant.pdf. [Accessed 10 May 2019].

[29] Xilinx, "LogiCORE IP Slice v1.0," 6 April 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xilinx_com_ip_xlslice/v1_0/pb042-xilinx-com-ip-xlslice.pdf. [Accessed 10 May 2019].

[30] Xilinx, "Processor System Reset Module v5.0," 18 November 2015. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf. [Accessed 10 May 2019].

[31] Xilinx, "Utility Vector Logic for Vivado (v2.0)," 2 November 2015. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/util_vector_logic/v2_0/ds913.pdf. [Accessed 10 May 2019].

[32] Xilinx, "XADC Wizard v3.0," 1 April 2015. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v3_0/pg091-xadc-wiz.pdf. [Accessed 2 March 2019].

[33] R. Prew, "Battle Over the FPGA: VHDL vs Verilog! Who is the True Champ?," Digilent Blog, 11 Abril 2016. [Online]. Available: https://blog.digilentinc.com/battle-over-the-fpga-vhdl-vs-verilog-who-is-the-true-champ/. [Accessed 31 May 2019].

This page intentionally left blank

DEVELOPMENT OF AN ACOUSTIC MODEM USING SYNTHESIZABLE MICROCONTROLLER

BACHELOR'S THESIS

Bachelor's degree in Industrial Electronics and Automatic Control Engineering

Author: M. Fahad Hassan-Mobshar

Director: Jordi Cosp Vilella

Call: June 2019

A. ANNEXE: TABLES

I. ADC Tables

*Table A-1: **XADC I/O ports***

Name	Type	Description
s_drp_daddr_in	Input	Input address port. This port specified the address of the XADC registers. The port has 7-bit width ([6:0]).
s_drp_den_in	Input	Data enable signal. Signal has to be set logic-High before data is carry out or carry in.
s_drp_di_in	Input	Data input port. It is used to save data in XADC registers. The port has 16-bit width ([15:0]).
s_drp_do_out	Output	Data output port. It is used to obtain data from ADC. The port has 16-bit width ([15:0]).
s_drp_drdy_out	Output	Data ready port. This signal goes high when the ADC has carried data in output port.
s_drp_dwe_in	Input	Data write enable signal. This signa has to be logic-High to write data in XADC registers.
Vp_Vn_vn_in	Input	Analogic input Vn negative port.
Vp_Vn_vp_in	Input	Analogic input Vn positive port.

Vaux2_vaxn2	Input	Analogic input Vaux2 negative port.
Vaux2_vaxp2	Input	Analogic input Vaux2 positive port.
Vaux10_vaxn10	Input	Analogic input Vaux10 negative port.
Vaux10_vaxp10	Input	Analogic input Vaux10 positive port.
dclk	Input	Input XADC clock. The clock has 50 MHz frequency.
reset_in	Input	Reset input signal. Active logic-High.
user_temp_ _alarm_out	Output	Temperature alarm. The signal is logic-High when the measured temperature is upper or lower than established temperature limits.
vccint_alarm_out	Output	V_{CCINT} alarm. The signal is logic-High when measured V_{CCINT} is upper or lower than established V_{CCINT} limits.
vccaus_alrm_out	Output	V_{CCAUX} alarm. This signal is logic-High when measured V_{CCAUX} is upper or lower than established V_{CCAUX} limits.
ot_out	Output	Over Temperature. This signal turns logic-High when the temperature exceeds the over temperature (OT) threshold. The default OT threshold is 125 ºC.
alarm_out	Output	Alarm. This signal could be used to flag the occurrence of any other alarm. This signal is active logic-High.
channel_out	Output	Channel selection output. The current ADC channel is placed on these outputs at the end of an ADC conversion. The port has 5-

		bit width ([4:0]). To know more about input MUX, see *Table A-6*.
eoc_out	Output	End of conversion. This signal transition to logic-High at the endo of an ADC conversion.
eos	Output	End of sequence. This signal transitions to logic-High when the measurement data from the las channel in an automatic channel sequence is written to the status registers.
busy	Output	Busy. This signal transitions to logic-High for an extended period during and ADC or sensor calibration.

Table A-2: ***Status registers***

Name	Address	Description
Temperature	00h	The result of temperature sensor measurement is stored at this location. The 12 MSBs correspond to the temperature sensor transfer function shown in *Figure 4-8*.
$\boldsymbol{V_{CCINT}}$	01h	The result of V_{CCINT} supply monitor measurement is stored at this location. The 12 MSBs correspond to the supply sensor transfer function shown in *Figure 4-9.*
$\boldsymbol{V_{CCAUX}}$	02h	The result of V_{CCAUX} data supply monitor measurement is stored at this location. The 12 MSBs correspond to the supply sensor transfer function shown in *Figure 4-9.*
$\boldsymbol{V_P/V_N}$	03h	The result of a conversion the dedicated analogue input channel is stored in this register.

		The 12 MSBs correspond to the transfer function shown in *Figure 4-6* or *Figure 4-7* depending on analogue input mode.
V_{REFP}	04h	The result of a conversion on the reference input V_{REFP} is stored in this register. The 12 MSBs correspond to the ADC transfer function shown in *Figure 4-9.*
V_{REFN}	05h	The result of a conversion on the reference input V_{REFN} is stored in this register. The channel is measured in bipolar mode with a two's complement's output coding as shown in *Figure 4-9.* The supply sensor is also used to measure V_{REFN}, thus 1 LSB = 3 V/4096.
V_{CCBRAM}	06h	The result of the on-chip V_{CCBRAM} supply monitor measurement is stored at this location. The 12 MSBs correspond to the supply sensor transfer function shown in *Figure 4-9.*
Undefined	07h	This location is unused.
Supply A offset	08h	The calibration coefficient for the supply sensor offset using ADC A is stored at this location.
ADC A offset	09h	The calibration coefficient for the ADC A offset is stored at this location.
ADC A gain	0Ah	The calibration coefficient for the ADC A gain error is stored at this location.
Undefined	0Bh to 0Fh	These locations are unused.
$VAUXP_{[15:0]}$ / $VAUXN_{[15:0]}$	10h to 1Fh	The results of the conversions on auxiliary analogue input channels are stored in this

		register. The 12 MSBs correspond to transfer function shown in *Figure 4-6* or *Figure 4-7* depending on analogue input mode.
Max temp	20h	Maximum temperature measurement recorded since power-up or the last XADC reset.
Max V_{CCINT}	21h	Maximum V_{CCINT} measurement recorded since power-up or the last XADC reset.
Max V_{CCAUX}	22h	Maximum V_{CCAUX} measurement recorded since power-up or the last XADC reset.
Max V_{CCBRAM}	23h	Maximum V_{CCBRAM} measurement recorded since power-up or the last XADC reset.
Min temp	24h	Minimum temperature measurement recorded since power-up or the last XADC reset.
Min V_{CCINT}	25h	Minimum V_{CCINT} measurement recorded since power-up or the last XADC reset.
Min V_{CCAUX}	26h	Minimum V_{CCAUX} measurement recorded since power-up or the last XADC reset.
Min V_{CCBRAM}	27h	Minimum V_{CCBRAM} measurement recorded since power-up or the last XADC reset.
Undefined	28h to 2Ah	These locations are unused.
Unassigned	2Bh	
Undefined	2Ch to 2Eh	These locations are unused.
Unassigned	2Fh	

Supply B offset	30h	The calibration coefficient for the supply sensor offset using ADC B is stored at this location.
ADC B offset	31h	The calibration coefficient for the ADC B gain error is stored at this location.
ADC B gain	32h	The calibration coefficient for the ADC B gain error is stored at this location.
Undefined	33h to 3Eh	These locations are unused.
Flag	3Fh	This register contains general status information (see *Flag register*).

Table A-3: ***Configuration register for 0, address*** $40h$

Bit	Name	Description
[4:0]	CH[4:0]	When operating in single channel mode or external multiplexer mode, these bits are used to select the ADC input channel. See *Table A-6*.
[7:5]	0	These bits should always set logic Low.
8	ACQ	When using single channel mode, this bit is set logic 1 to increase the setting time available on external analogue input in continuous sampling mode by six ADCCLK cycles.
9	$E\bar{C}$	This bit is used to select either continuous (logic 0) or event-driven (logic 1) sampling mode for the ADC.

10	$B\overline{U}$	This bit is used in single channel mode to select either unipolar (logic 0) or bipolar (logic 1) operating mode for the ADC analogue input.
11	MUX	This bit should be set to a logic 1 to enable external multiplexer mode.
[13:12]	AVG[1:0]	These bits are used to set the amount of sample averaging on selected channels in both single channel and sequence modes. See *Table A-7*.
14	0	This bit should always set as logic Low.
15	CAVG	This bit is used to disable averaging for the calculation of the calibration coefficients. Averaging is enabled by default (logic 0) and is fixed at 16 samples.

Table A-4: ***Configuration register, address*** $41h$

Bit	Name	Description
0	OT	This bit is used to disable the over-temperature signal by setting this bit to logic 1.
0 to 3, 8	ALM[3:0]	These bits are used to disable individual alarm outputs (logic 1) for temperature, V_{CCINT}, V_{CCAUX}, and V_{CCBRAM}, respectively.
[7:4]	CAL0 to CAL3	These bits enable (logic 1) the application of the calibration coefficients to the ADC and on-chip supply sensor measurements. See *Table A-8*.
[11:9]	0	These bits should always set logic Low.

[16:12]	SEQ[3:0]	These bits enable the channel-sequencer function. See *Table A-9*.

Table A-5: ***Configuration register 3, address*** $42h$

Bit	Name	Description
[3:0]	0	These bits should always set logic Low.
[5:4]	PD[1:0]	The XADC could be power-down using these bits. See *Table A-10*.
[7:6]	0	These bits should always set logic Low.
[15:8]	CD[7:0]	These bits select the division ratio between the DRP clock (DCLK) and lower frequency ADC clock (ADCCLK) used for ADC. See *Table A-11*.

Table A-6: ***ADC channel selection***

ADC Channel	CH[4:0]	Description
0	00h	On-chip temperature
1	01h	V_{CCINT}
2	02h	V_{CCAUX}
3	03h	V_P, V_N – Dedicated analogue input
4	04h	V_{REFP} $(1.25\,V)^{(1)}$
5	05h	V_{REFN} $(0\,V)^{(1)}$

6	06h	V_{CCBRAM}
7	07h	Invalid channel selection
8	08h	Carry out an XADC calibration
9-15	09h to 0Fh	Invalid channel selection
16	10h	VAUXP[0], VAUXN[0] – Auxiliary channel 0
17-31	11h to 1Fh	VAUXP[15:1], VAUXN[15:1] – Auxiliary channels [15:1]

1. *These channel selection options are used for XADC self-check and calibration operations. When these channels are selected, the supply sensor is connected to V_{REFP} or V_{REFN}.*

Table A-7: ***Averaging filter settings***

AVG[1:0]	Function
00b	No averaging
01b	Average 16 samples
10b	Average 64 samples
11b	Average 256 samples

Table A-8: ***Calibration enables***

Name	Description
CAL0	ADCs offset correction enable
CAL1	ADCs offset and gain correction enable
CAL2	Supply sensor offset correction enable

CAL3	Supply sensor offset and gain correction enable

Table A-9: ***Sequencer operation settings***

SEQ[3:2]	SEQ[1:0]	Function
00b	00b	Default mode
00b	01b	Single pass sequence
00b	10b	Continuous sequence mode
00b	11b	Single channel mode (sequencer off)
01b	XXb	Simultaneous sampling mode
10b	XXb	Independent ADC mode
11b	XXb	Default mode

Table A-10: ***Power down selection***

PD[1:0]	Description
00b	Default. All XADC blocks powered up.
01b	Not valid – do not select
10b	ADC B powered down
11b	XADC powered down

Table A-11: ***DCLK division selection***(1)

CD[7:0]	Division

00h	2
01h	2
02h	2
03h	3
04h	4
-	-
FEh	254
FFh	255

1. *Minimum division ratio is 2. DCLK division must be selected to keep the ADC clock in its supported frequency range.*

Table A-12: ***Sequencer on-chip channel selection (for 48h, 4Ah & 4Ch)***

Bit	ADC Channel	Function
0	8	XADC calibration
1 to 7	9 to 15	Invalid channel selection
8	0	On-chip temperature
9	1	V_{CCINT}
10	2	V_{CCAUX}
11	3	V_P, V_N – Dedicated analogue input
12	4	V_{REFP}
13	5	V_{REFN}
14	6	V_{CCBRAM}

15	7	Invalid channel selection

Table A-13: ***Sequencer auxiliary channel selection (for 49h, 4Bh & 4Dh)***

Bit	ADC Channel	Function
0	16	VAUXP[0], VAUXN[0] – auxiliary channel 0
1	17	VAUXP[1], VAUXN[1] – auxiliary channel 1
-	-	-
14	30	VAUXP[14], VAUXN[14] – auxiliary channel 14
15	31	VAUXP[15], VAUXN[15] – auxiliary channel 15

II. Cortex M3 Tables

Table A-14: ***Cortex-M3 Core I/O ports description***

Name	Type	Description
HCLK	Input	Processor Clock
SYSRESETn	Input	Processor Reset signal. When this signal is logic-High the entire processor system is reset with exception of debug logic in the: NVIC, FPB, DWT, ITM and AHB-AP.

IRQ	Input	External interrupt signals. This port adjusts the width automatically depending on the input signals.
NMI	Input	Non-maskable Interrupt input.
CFGITCMEN[1:0]	Input	Alias input enable. If the CFGITCMEN [1] is set, then the internal RAM ITCM is mapped to the upper address alias in the memory map. And if CFGITCMEN [0] is set, then the internal RAM ITCM is mapped to the lower address alias in the memory map. The value of this signal has to be held constant for at least two cycles before SYSRESTn is deserted
DBGRESRTn	Input	SW-DP is reset with this signal. This reset must be synchronized to DBGCLK.
DBGRESTART	Input	External input restart request signal to exit from debug state.
EDBGRQ	Input	External debug request signal.
STCLK	Input	System Tick Clock.
SWCLKTCK	Input	Serial wire or JTAG clock. This signal is the clock for the debug interface domain of the SWJ-DP. In JTAG mode this is equivalent to TCK, and in SW-DP is the Serial Wire Clock. It is asynchronous to all other clocks
SWDITMS	Input	SW Test Mode State (TMS)
nTRST	Input	SWJ-DP reset input port.
TDI	Input	Test Data Input port.

CM3_SYS_AXI3	-	Cortex M3 System AHB to System AXI3 master bus (see for bus signals definitions).
CM3_CODE_AXI3	-	Cortex M3 Combined Code AHB to Code AXI3 master bus (see for bus signals definitions).
SYSRESTETREQ	Output	System reset request output port.
DBGRESTARTED	Output	Debug restart auxiliary signal. This signal is asserted when the SYSRESTREQ bit of the Application Interrupt and Reset control register is set. For example, this could be used as an input to a watchdog timer.
LOCKUP	Output	Indicate the core is locked up. This signal gives immediate indication of seriously errant kernel software. This is the result of the core being locked up due to an unrecoverable exception following the activation of the processor's built in system state protection hardware.
HALTED	Output	In halting debug mode. HALTED remains asserted while the core is in debug.
JTAGNSW	Output	This signal identifies whether the SWJ block is in SW, when this signal is logic-Low, or JTAG mode, when this signal is logic-High.
JTAGTOP	Output	JTAG status output port. This signal indicates the stat of the JTAG-DP TAP controller.
SWDO	Output	Serial wire data output port.
SWDOEN	Output	Serial wire output enable port.
TDO	Output	Test data output port.

nTDOEN	Output	Test data out enable is unused unless SWO block is being used.
TRACENA	Output	Trace Enable port. If the TRCENA bit of the Debug Exception and Monitor Control Register is enabled then the power consumption of Data Watchpoint and Trace (DWT) and Instrumentation Trace Macrocell blocks is minimized.
TRACECLK	Output	Is the reference clock for the Trace Poet Interface Unit (TPIU). It is asynchronous to the other clocks.
TRCEDATA[3:0]	Output	Output data for clocked modes. TRACEDATA changes on both edges of TRCECLK

III. IP Tables

Table A-15: ***AXI Interconnect core Slave I/O signals***

Signal Name	Direction	Width	Description
Snn_ACLK	Input	1	Slave interface clock input
Snn_ARESETN	Input	1	Slave interface reset input (active-Low)
Snn_AXI_araddr	Input	32	Read address channel address
Snn_AXI_arcache	Input	4	Read address channel cache characteristics
Snn_AXI_arburst	Input	2	Read address channel burst type (0-2)

Snn_AXI_arlen	Input	4	Read address channel burst length code (0-16)
Snn_AXI_arlock	Input	2	Read address channel atomic access type (0, 1)
Snn_AXI_arprot	Input	3	Read address channel protection bits
Snn_AXI_arready	Output	1	Read address channel ready
Snn_AXI_arsize	Input	3	Read address channel transfer size code (0-7)
Snn_AXI_aruser	Input	1	User-defined AR channel signals.
Snn_AXI_arvalid	Input	1	Read address channel valid
Snn_AXI_awaddr	Input	32	Write address channel address
Snn_AXI_awburst	Input	2	Write address channel burst type code (0-2)
Snn_AXI_awcahce	Input	4	Write address channel cache characteristics
Snn_AXI_awlen	Input	4	Write address channel burst length
Snn_AXI_awlock	Input	2	Write address channel atomic access type (0,1)

Snn_AXI_awprot	Input	3	Write address channel protection bits
Snn_AXI_awready	Output	1	Write address channel ready
Snn_AXI_awsize	Input	3	Write address channel transfer size code (0-7)
Snn_AXI_awuser	Input	1	User defined aw channel signals
Snn_AXI_awvalid	Input	1	Write address channel valid
Snn_AXI_bready	Input	1	Write response channel ready
Snn_AXI_bresp	Output	2	Write response channel response code (0-3)
Snn_AXI_bvalid	Output	1	Write response channel valid
Snn_AXI_rdata	Output	32	Read data channel data
Snn_AXI_rlast	Output	1	Read data channel last data beat
Snn_AXI_rready	Input	1	Read data channel ready.
Snn_AXI_rresp	Output	2	Read data channel response code (0-3)
Snn_AXI_rvalid	Output	1	Read data channel valid
Snn_AXI_wdata	Input	32	Write data channel data

Snn_AXI_wlast	Input	1	Write data channel last data beat
Snn_AXI_wready	Output	1	Write data channel ready
Snn_AXI_wstrb	Input	4	Write data channel byte strobes
Snn_AXI_wvalid	input	1	Write data channel valid

Table A-16: ***AXI Interconnect core Master I/O signals***

Signal Name	Direction	Width	Description
Mnn_ACLK	Input	1	Master interface clock input
Mnn_ARESETN	Input	1	Master interface reset input (active-Low)
Mnn_AXI_araddr	Output	32	Read address channel address
Mnn_AXI_arready	Input	1	Read address channel ready
Mnn_AXI_arvalid	Output	1	Read address channel valid
Mnn_AXI_awaddr	Output	32	Write address channel address
Mnn_AXI_awready	Intput	1	Write address channel ready

Mnn_AXI_awvalid	Output	1	Write address channel valid
Mnn_AXI_bready	Output	1	Write response channel ready
Mnn_AXI_bresp	Input	2	Write response channel response code (0-3)
Mnn_AXI_bvalid	Input	1	Write response channel valid
Mnn_AXI_rdata	Input	32	Read data channel data
Mnn_AXI_rready	Output	1	Read data channel ready.
Mnn_AXI_rresp	Input	2	Read data channel response code (0-3)
Mnn_AXI_rvalid	Input	1	Read data channel valid
Mnn_AXI_wdata	Output	32	Write data channel data
Mnn_AXI_wready	Input	1	Write data channel ready
Mnn_AXI_wstrb	Output	4	Write data channel byte strobes
Mnn_AXI_wvalid	Output	1	Write data channel valid

Table A-17: ***AXI Global port signals***

Signal Name	Direction	Width	Description

ACLK	Input	1	Clock input
ARESETN	Input	1	Global reset input (active-Low)

Table A-18: ***AXI GPIO I/O signals***

Signal Name	Direction	Description
S_AXI_aclk	Input	AXI Clock
S_AXI_arestn	Reset	AXI Reset, active-Low
S_AXI	Input / Output	AXI slave signals in *Table A-15* these signals are described
ip2intc_irpt	Output	AXI GPIO interrupt. Active-High and level sensitive signal.
gpio_io_i gpio2_io_i	Input	Channel 1 and Channel 2 purpose input pins. Width of this port is configurable, maximum 32-bits.
gpio_io_o gpio2_io_o	Output	Channel 1 and Channel 2 general purpose output. Width of this port is configurable, maximum 32-bits.
gpio_io_t gpio2_io_t	Output	Channel 1 and Channel 2 general purpose 3-state pins. Width of this port is configurable, maximum 32-bits.

Table A-19: ***AXI BRAM Controller I/O signals***

Signal Name	Direction	Description

S_AXI_aclk	Input	AXI Clock
S_AXI_arestn	Input	AXI Reset, active-Low
S_AXI	Input / Output	AXI slave signals in *Table A-15* are described
bram_addr_a	Output	BRAM Port A (write port) address bus. Bus is sized according to data width and based on C_S_AXI_BASEADDR and C_S_AXI_HIGHADDR. In this case this width is 13-bits
bram_clk_a	Output	Port A BRAM clock. Connected to ACLK with same frequency, and same phase.
bram_wrdata_a	Output	BRAM Port A (write port) address bus. Bus is sixed according to data width and based on C_S_AXI_BASEADDR and C_S_AXI_HIGHADDR. In this case the pin has 32-bits width.
bram_rdata_a	Input	BRAM Port A (read port) read data bus. Size of BRAM read data width is equal to size of AXI slave port connection to the AXI BRAM Controller. In this case the pin has 32-bits width.
bram_en_a	Output	BRAM Port A (write port) enable signal. Active High.
bram_rst_a	Output	BRAM Port A reset. Active-High.
bram_we_a	4 bits	BRAM Port A active-High write enable signal.

Table A-20: ***Block Memory Generator core I/O pins***

Signal Name	Direction	Description
addra	Input	Port A Address, addresses the memory space for port A Read and Write operation.
clka	Input	Port A Clock, port A operations are synchronous to this clock. For synchronous operation.
dina	Input	Port A Data Input, data input to be written into the memory through port A.
douta	Output	Port A data output, data output from Read operations through port A.
ena	Input	Port A Clock Enable, enables Read, Write, and reset operations through port A.
rsta	Input	Port A set/Reset, reset the Port A memory output latch or output register.
wea [3:0]	Input	Port A Write Enable, enables Write operations through port A.

Table A-21: ***FF_D_T block I/O ports description***

Signal Name	Direction	Description
CLK	Input	Input Clock signal. Clock frequency 50 MHz.
RST	Input	Input reset signal. Active logic-Low.
DATA_INPUT	Input	Input data port. Data to save in register.

DRDY	Input	ADC data ready signal.
GOERT_STARTED	Input	Goertzel Algorithm started enable. When this signal is logic-High, indicates that the Goertzel algorithm is being calculated.
DATA_OUTPUT	Output	Data output port.
DATA_ENABLE	Output	Data ready enable signal.

Table A-22: ***Square Generator block I/O ports description***

Signal Name	Direction	Description
CLK	Input	Input Clock signal. Clock frequency 50 MHz.
RST	Input	Input reset signal. Active logic-Low.
Enable	Input	Enable to begin the signal generation.
Data_Out	Out	Data out of generated signal.

Table A-23: ***Processor Synchronous Reset Module I/O pins***

Signal Name	Direction	Description
slowest_sync_clk	Input	Slowest Synchronous Clock
ext_reset_in	Input	External Reset Input, Active-High or Low based upon the generic Ext Reset Active Polarity.

aux_reset_in	Input	Auxiliary Reset Input. Active-High or Low based upon the generic Auxiliary Reset Active Polarity.
mb_debug_sys_rst	Input	MDM reset input. Always active-High, minimum width defined by parameter External Reset Active Window Width.
dcm_locked	Input	DCM Lock signal.
mb_reset	Output	MB core reset. Active-High.
bus_struct_reset	Output	Bus Structures reset, Active-High. 1-bit width.
peripheral_reset	Output	Peripheral reset is for all peripherals attached to any bus that is synchronous with the slowest_sync_clk. Active-High.
interconnect_aresetn	Output	Interconnect_aresetne reset, for example, interconnects with active-Low reset inputs.
peripheral_aresetn	Output	The signal peripheral_aresetn is for all peripherals attached to interconnect that is synchronous with the slowest_sync_clk. Active-Low.

Table A-24: ***Clocking Wizard core I/O pins***

Signal Name	Direction	Description

clk_in1	Input	Clock in1, single-ended primary input clock port. Available when single-ended primary clock source is selected.
clk_out1	Output	Clock out1, output clock of the clocking network. This signal is not optional.
locked	Output	When the signal is asserted, indicates that the output clocks are stable and usable by downstream circuity.

This page intentionally left blank

B. ANNEXE: BUDGET

This annexe is about the budget of this project which takes into consideration the cost of the devices, licenses and engineering time required to design, implement and verify the results of the modem in the laboratory.

This chapter is divided into three sections:

- Devices
- Licenses
- Engineering work

I. Devices cost

Firstly, it is used a personal computer to develop the modem, synthesize them, and generate the module files to implement in the FPGA. It is also used to generate the program of the microcontroller and generate the files. Furthermore, it is used to generate the schematics and this report file.

The second device used is the Nexys 4 DDR board to implement the design and observe the functionality of the modem.

It has also used a Rigol DG5101 Function generator to emulate the input analogue signal which could come from the Node. And a Keysight MOSOX3040T oscilloscope is used to observe the Goertzel Algorithm result and the echo generated by the modem.

The cost of these devices is shown in the following table.

Table B-1: ***Devices cost***

Devices	Price (€)	Useful life (yr.)	Time used (yr.)	Price (€)
Computer	900.00	3	0.5	150.00
Nexys 4 DDR	236.86	4	0.5	29.61
RIGOL DG5101	3,930.57	5	0.5	393.06
KEYSIGHT MSOX3034T	5,287.00	5	0.5	528.70
Total	**10,354.43**	**-**	**-**	**1,101.37**

II. Licences cost

For this project, it has been used the following programs

- Vivado: to develop the modem
- Keil uVision: to create the program and generate the Hex file
- Microsoft Office: to write the report

The cost related to the licenses are shown in the following table.

Table B-2: ***Licenses cost***

Programs	Price per year (€/yr.)	Time used (yr.)	Price (€)
Vivado WebPack	0	0.5	0.00
Keil uVision	0	0.5	0.00
Microsoft Office	69.00	0.5	34.5

Total	**69.00**	**-**	**34.5**

III. Engineering work cost

To develop this modem, it has been invested a number of hours by an engineer. The following table shows time dedicated to the different sections and their cost.

The hours spent developing the project are the ones estimated. The amount of the hours is larger because of technical difficulties during the modem development.

The table also includes the price and time dedicated to create different documents such as the report, annexes, user manual, etc.

*Table B-3: **Engineering time cost***

Sections	Time dedicated (h)	Price per hour (€/h)	Price (€)
Developing the ADC in the FPGA.	80	10.00	800.00
Testing the microcontroller with simple application	150	12.00	1,800.00
Connecting the ADC and the microcontroller	250	12.00	3,000.00
Testing and improving the modem.	100	12.00	1,200.00
Drafting the report	100	8.00	800.00
Total	**680**	**-**	**7,600.00**

IV. Total

Finally, the total budget of developing the modem is obtained summing the previous calculated costs. The following table shows the budget of this modem.

*Table B-4: **Total Budget***

	Price (€)
Devices cost	1,101.37
Licences cost	34.50
Engineering work cost	7,600.00
Total	**8,735.87**

C. ANNEXE: USER MANUAL

This chapter describes the previous configuration of programs to edit the modem files. It also describes the steps to follow to generate the modem for FPGA.

I. Configuration

Before start developing the modem in Vivado, first, the configuration is needed. During this hardware and software repositories are going to add in the Viviado project. You can visit the following link to configure. The video describes step by step the configuration of Vivado to use Cortex-M processor.

- Arm Cortex-M DesignStart FPGA: STEP 2 Prepare Vivado for Cortex-M development

II. Generating modem

They are some steps to follow to generate the .bit file for FPGA. These steps depend on whether the software is modified or hardware. If you need to modify only software of the microcontroller continues to step 16 to generate a new bitstream file.

When in the design any minimum change is made, the next steps are followed to generate the bit file.

1. To modify hardware; goes to Vivado project, expand the m1_for_arty_a7_wrapper, double click on the block diagram entry.

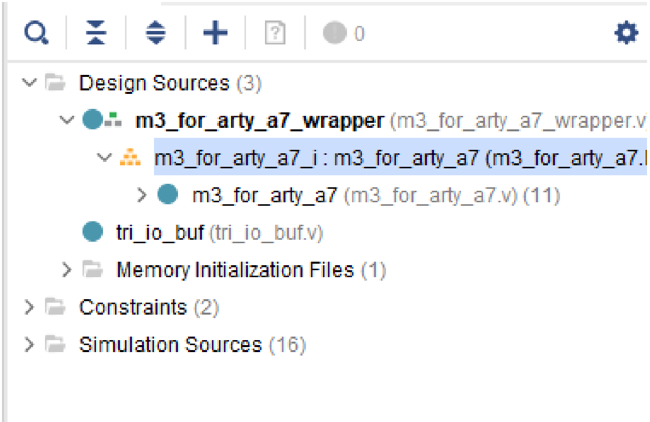


Figure C-1: ***Opening Block diagram***

2. After modifying the design, first, you have to validate the design clicking on *validate design* tick box.

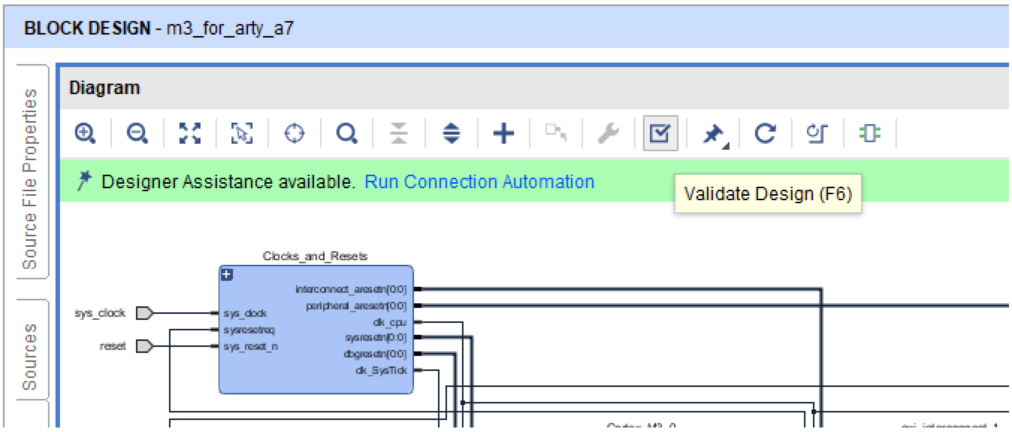


Figure C-2: ***Validating design***

3. If there is no error the design will be validated successfully. Save the block diagram.
4. Next, generate a new bitstream image file. The new bitstream takes around 20 min to build (depends on the PC).
5. When the new bitstream file has generated, we need to generate new hardware file. Go to $file \rightarrow Export \rightarrow Export\ Hardware$ in Vivado to generate new hardware description file.

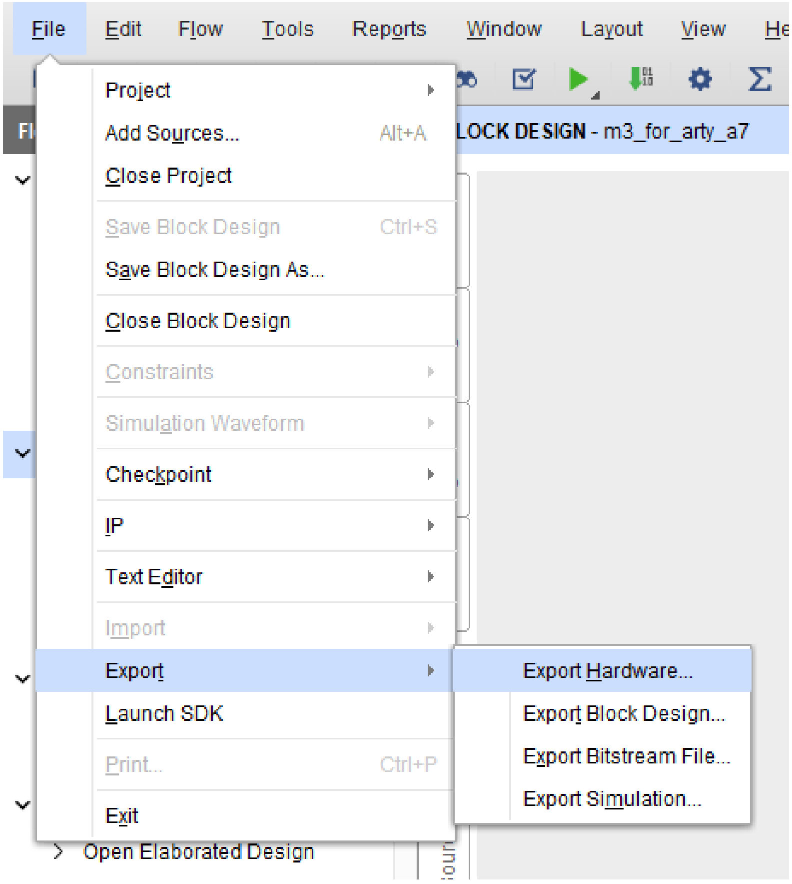


Figure C-3: ***Export Hardware***

6. The next window will appear when you click con *Export Hardware*.

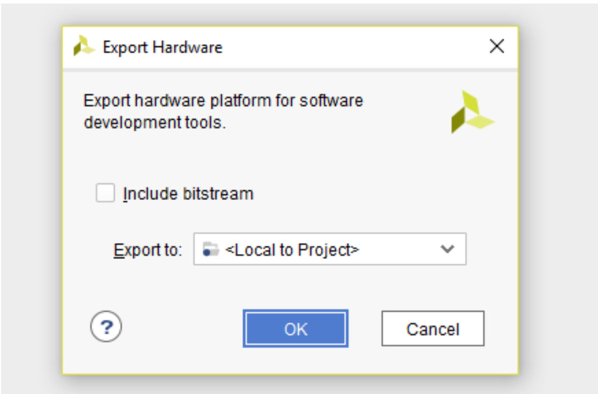


Figure C-4: ***Export Hardware window***

7. In choose location, navigate to the folder *Software*, where the hardware description file will be saved. If appears a warining click in ok, the waring would be because you are going to rebuild the Hardware.

8. Open a new windows folder and navigate to sdk_workspace (…/software/m3_for_arty_a7_1/sdk_workspace). Delete all files and folders because new bsp files will be saved in this folder.

9. Next, you need to launch SDK thus go to $file \rightarrow Launch\,SDK$. The following window will appear.

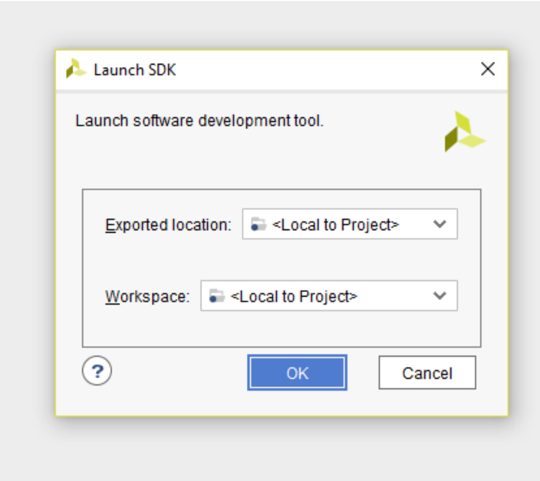


Figure C-5: **Launch SDK**

10.In *Export location,* navigate to the *Software directory*. In Workspace navigate to …/software/m3_for_arty_a7_1/sdk_workspace, and click in ok.

11.When the SDK has launched, go to $Xilinx \rightarrow Repositories$.

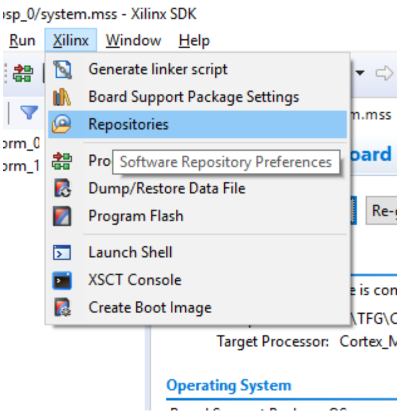


Figure C-6: **SW repositories**

12.There rescan the repositories and click in ok.

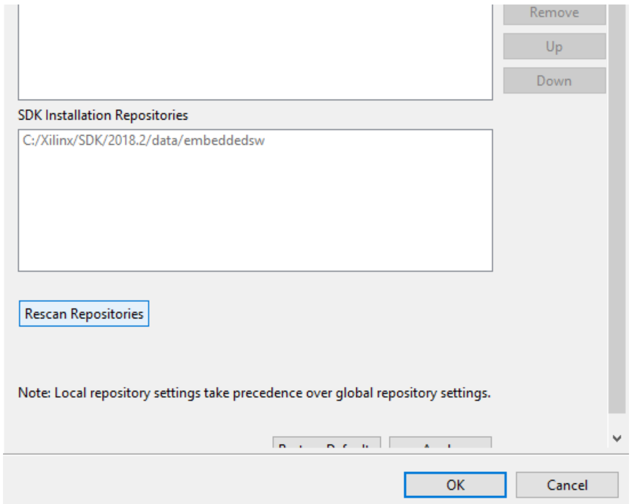


Figure C-7: ***Rescanning SW repositories***

13.Now go to $file \rightarrow new \rightarrow Board\ Support\ Package$ and click on.

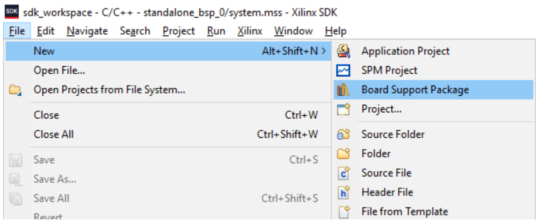


Figure C-8: ***Generating new BSP***

14.Click in finish

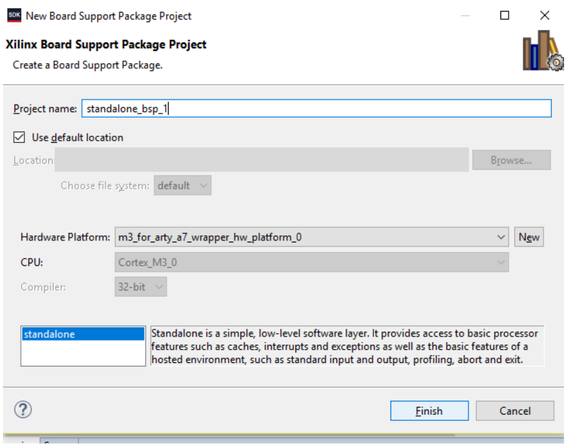


*Figure C-9:****Generating standalone bsp***

15.After a while, you will see in *SDK log* that the bsp has been created.

16.Now open the Keil uVision program from *Software* folder.

17. If you need to modify any software, modify.

18. Click in rebuild and ensure that there are no errors.

19. Open a new windows folder and navigate to Bulid Keil (…/software/m3_for_arty_a7_1/Bulid_Keil). Click on make_hex_a7. This will generate new bram_a7.hex file with the new program.

20. Go back to Vivado and *Run Synthesis*. After the synthesis has finished successfully, generate *Bitstream* file.

21. Finally, when the Bitstream file is successfully generated, you can implement the file in FPGA.

D. ANNEXE: CODE

This chapter shows the software and hardware code of the device. This are only codes which are created by the developer. Other elements are part of Cortex-M3 and Vivado. These parts can be observed in Annexe Schematic.

The chapter is divided into two sections: Hardware Code and Software Code.

I. Hardware code

There are two elements created for modem using Verilog HDL. The two parts generated are FF_D_T block and Square Generator block. Specified

FF_D_T

```
module FF_D_T#(
        parameter DWIDTH = 16
        )
        (
        input       [DWIDTH-1:0]    DATA_INPUT,
        input                       CLK,
        input                       RST,
        input                       DRDY,
        input                       GOERT_STARTED,
        output      [DWIDTH-1:0]    DATA_OUTPUT,
        output                      DATA_ENABLE
    );
```

Code D-1: ***FF-D-T module code (I); I/O declaration.***

```
    reg     [DWIDTH-1:0]    D_OUT_AUX;
    reg                     D_ENABLE_AUX;
    reg                     GS_Z1;
    reg                     GS_Z2;



    always @(posedge CLK) begin
        if (!RST) begin
            D_OUT_AUX <= 0;
        end
        else if (DRDY) begin
            D_OUT_AUX <= DATA_INPUT;
        end
    end

    assign DATA_OUTPUT = D_OUT_AUX;

    always @(posedge CLK) begin
        if (!RST) begin
            D_ENABLE_AUX <= 0;
            GS_Z1 <= 0;
            GS_Z2 <= 0;
        end
        else if (!GS_Z1 && GS_Z2) D_ENABLE_AUX <= 0;
        else if(!GOERT_STARTED && DRDY) D_ENABLE_AUX <=  !(D_ENABLE_AUX);
        GS_Z2 <= GS_Z1;
        GS_Z1 <= GOERT_STARTED;
    end

    assign DATA_ENABLE = D_ENABLE_AUX;
endmodule
```

Code D-2: ***FF-D-T module code (II).***

Square Generator

```
module Square_Generator#(
    parameter   DO_Width = 8,
    parameter   Input_Clock_Frequency = 50, // Clock time in MHz
    parameter   Signal_Time = 5000,       // Time in microseconds
    parameter   Output_Signal_Frequency = 10000 // Frequency in Hz
    )
    (
    input                        CLK,
    input                        RST,
    input                        Enable,
    output  [DO_Width-1:0]       Data_Out
    );

    reg     [DO_Width-1:0]       D_OUT_AUX;
    reg                          EnableZ1;
    reg                          EnableZ;
    reg                          EAux;
    reg     [31:0]               Tem;
    reg     [31:0]               TS;            // to count de perio
    reg                          ES;            // to obtain enable
```

Code D-3: ***Square Generator module code (I); I/O and internal registers declaration.***

```
always @(posedge CLK) begin
    /*
    Enable detection. The aux Enable is active for long time,
    until the complete signal is not transmit
    */
    if (!RST) begin
        EnableZ1 <= 0;
        EnableZ <= 0;
        TS <= 0;
        ES <= 0;
        D_OUT_AUX <= 0;
        Tem <= 0;
    end
    else begin
        EnableZ1 <= EnableZ;
        EnableZ <= Enable;
        if (EnableZ1 == 0 && EnableZ == 1) begin
            EAux <= 1;
        end
        /* Counter to create square signal period */
        if (EAux)  begin
 // couter for signal T: Input_Freq / (Out_Freq * 2); Out_T/2 * 1/Input_T
             if (TS < (In_Clk_Frq*500000/Out_Sig_Frq - 1)) begin
                    TS <= TS+1;
             end
             else if (TS == (In_Clk_Frq*500000/Out_Sig_Frq -1)) begin
                ES <= 1;
                TS <= 0;
             end
        end
```

Code D-4: ***Square Generator module code (II)***

```
        /* Signal generating and send during the signal time   */
         if (EAux)  begin
            if (Tem < (Signal_Time*In_Clk_Frq - 1)) begin
                if (ES) begin
                    D_OUT_AUX <= ~D_OUT_AUX;
                    ES <= 0;
                end
                Tem <= Tem +1;
            end
            else if (Tem == (Signal_Time*In_Clk_Frq - 1)) begin
                EAux <= 0;
                TS <= 0;
                Tem <= 0;
                D_OUT_AUX <= 0;
            end
        end
    end
end

assign Data_Out = D_OUT_AUX;

endmodule
```

Code D-5: ***Square Generator module code (II)***

II. Software code

Five different type of files are needed for code. These files are used to generate the program apart from other files generated by SDK. These are main used program files. The following files are the most important files:

- main.c
- gpio.c

Other important files for program are included in the folder.

Main

```
/*                        Program main file                          */


/* --------Included Headers-------- */
// Xilinx specific headers
#include "xparameters.h"
#include "xgpio.h"

#include "m3_for_arty.h"        // Project specific header
#include "gpio.h"

int main (void)
{
      //    Initialise the GPIO
      InitialiseGPIO();

      // Enable GPIO Interrupts
      NVIC_EnableIRQ(GPIO0_IRQn);
      NVIC_EnableIRQ(GPIO1_IRQn);
      NVIC_EnableIRQ(GPIO_ADC_IRQn);
      EnableGPIOInterrupts();



      while (1) {
      // Wait for interruption


      }
}
```

Code D-6: ***Program code main.c file.***

Gpio C file

```
/*                GPIO: Interruption attender file                */


/*--------Included Headers--------*/
#include <math.h>
#include <limits.h>
#include <time.h>


#include "gpio.h"
#include "xparameters.h"        // Project memory and device map
#include "xgpio.h"              // Xilinx GPIO routines
#include "peripherallink.h"     // IRQ definitions


#define M_PI 3.141592653589793
/*!< SysTick CTRL: Disable Mask */
#define SysTick_CTRL_ENABLE_MskN (0UL << SysTick_CTRL_ENABLE_Pos)


/************************* Variable Definitions ***********************/
/*
 * The following are declared static to this module so they are zeroed and
so they are easily accessible from a debugger
 * Also they are initialised in main, but accessed by the IRQ routines
 */


static XGpio Gpio_ADC_Data_Control; /* The driver instance for GPIO ADC */
static XGpio Gpio_Data;       /* The driver instance for GPIO Device 0 */
static XGpio Gpio_RGBLed_PB;  /* The driver instance for GPIO Device 1 */
```

Code D-7: ***Program code gpio.c file (I).***

```
// Initialise the GPIO and zero the outputs
int InitialiseGPIO( void )
{
    // Define local variables
    int status;
    /** Initialize the GPIO driver so that it's ready to use,
     * specify the device ID that is generated in xparameters.h **/

    status = XGpio_Initialize(&Gpio_ADC_Data_Control, XPAR_ADC_AXI_GPIO_2_DEVICE_ID);
    if (status != XST_SUCCESS)  {
        return XST_FAILURE;
    }
   status = XGpio_Initialize(&Gpio_Data, XPAR_AXI_GPIO_0_DEVICE_ID);
    if (status != XST_SUCCESS)  {
        return XST_FAILURE;
    }
    status = XGpio_Initialize(&Gpio_RGBLed_PB, XPAR_AXI_GPIO_1_DEVICE_ID);
    if (status != XST_SUCCESS)  {
        return XST_FAILURE;
    }
/* GPIOADC */
// In the ADC is needed to select channel data from which is want to see in output channel
// The port 0 of GPIO ADC is selected as output to write the address of desired converted data
    XGpio_SetDataDirection(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0xffff000);
    // the aux10 is selected to see converted data, addr of aux2 is 10010b -> 0x12
    XGpio_DiscreteWrite(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0x01A);
    // Now the port is configured to be input
    // Port0 drives control signals. Set bus to be input.
    // (In this case a normal mode is used and ADC can't be configured.)
    XGpio_SetDataDirection(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0xffffffff);
    // Port1 drives data bus. Set bus to be input.
    // (In this case a normal mode is used and ADC can't be configured.)
    XGpio_SetDataDirection(&Gpio_ADC_Data_Control, ARTY_A7_ADC_DATA, 0xffffffff);
    /* GPIO0 */
    // Port0 drives data out.  Set bottom 16 UART ports to be outputs.
    XGpio_SetDataDirection(&Gpio_Data, ARTY_A7_Output_DATA, 0xFFFF0000);
```

Code D-8: ***Program code gpio.c file (II).***

```
/* GPIO1 */
    // Port0 drives led_rgb.  Set 12 UART ports to be outputs.
    XGpio_SetDataDirection(&Gpio_RGBLed_PB, ARTY_A7_RGB_CHANNEL, 0xffffff00);
     //ARTY_A7_GPIO1->TRI0 = 0xffffff00;
     // Port 1 inputs the push button switches. Set to be inputs
    XGpio_SetDataDirection(&Gpio_RGBLed_PB, ARTY_A7_PB_CHANNEL, 0xffffffff);
     //ARTY_A7_GPIO1->TRI1 = 0xffffffff;

/* Default Values */
     // Default value of Data Out Bus
     XGpio_DiscreteWrite(&Gpio_Data, ARTY_A7_Output_DATA, 0x0000);
     //ARTY_A7_GPIO0 -> DATA0 = 0x0000;
     // Default value of RGB LEDs
     XGpio_DiscreteWrite(&Gpio_RGBLed_PB, ARTY_A7_RGB_CHANNEL, 0x00);
     //ARTY_A7_GPIO1->DATA0 = 0x00;
         return XST_SUCCESS;
     }

/* Set GPIO interrupts */
     void EnableGPIOInterrupts( void ) {
     // ADC signals on Channel 2 to occur interrupt
     XGpio_InterruptEnable(&Gpio_ADC_Data_Control, XGPIO_IR_CH1_MASK);
     // Push buttons on Channel 2 to occur interrupt
      XGpio_InterruptEnable(&Gpio_RGBLed_PB, XGPIO_IR_CH2_MASK);

     // Having enabled the M3 to handle the interrupts, now enable the GPIO to send the interrupts
     XGpio_InterruptGlobalEnable(&Gpio_ADC_Data_Control);
     XGpio_InterruptGlobalEnable(&Gpio_Data);
     XGpio_InterruptGlobalEnable(&Gpio_RGBLed_PB);
     }
/* Define the GPIO interrupt handlers */
     void GPIO0_Handler ( void ){
         // Clear interrupt from GPIO
         XGpio_InterruptClear(&Gpio_Data, XGPIO_IR_MASK);
         // Clear interrupt in NVIC
         NVIC_ClearPendingIRQ(GPIO0_IRQn);
     }
```

Code D-9: ***Program code gpio.c file (III).***

```
void GPIO1_Handler ( void ) {
    int mask, led_val, incr;
    volatile uint32_t gpio_push_buttons;
    volatile uint32_t gpio_leds_rgb;

    // For LEDs, cycle around color each time respective push button is pressed
    // Only change if a pushbutton is pressed.
    // This prevents a double change as the button is released.
    if( XGpio_DiscreteRead(&Gpio_RGBLed_PB, ARTY_A7_PB_CHANNEL) != 0 ) {
// LEDs are on a 3 spacing.  So multiply button press by 2^3 to increment the correct LED
        gpio_push_buttons = XGpio_DiscreteRead(&Gpio_RGBLed_PB, ARTY_A7_PB_CHANNEL);
        gpio_leds_rgb     = XGpio_DiscreteRead(&Gpio_RGBLed_PB, ARTY_A7_RGB_CHANNEL);
        if ( gpio_push_buttons & 0x1 ) {
                mask = 0x7;
                incr = 0x1;
        } else if ( gpio_push_buttons & 0x2 ) {
                mask = (0x7 << 3);
                incr = (0x1 << 3);
        } else if ( gpio_push_buttons & 0x4 ) {
                mask = (0x7 << 6);
                incr = (0x1 << 6);
        } else if ( gpio_push_buttons & 0x8 ) {
                mask = (0x7 << 9);
                incr = (0x1 << 9);
        }
        led_val = gpio_leds_rgb & mask;
        led_val = (led_val+incr) & mask;
        gpio_leds_rgb = (gpio_leds_rgb & ~mask) | led_val;

        XGpio_DiscreteWrite(&Gpio_RGBLed_PB, ARTY_A7_RGB_CHANNEL, gpio_leds_rgb);
    }
    // Clear interrupt from GPIO
    XGpio_InterruptClear(&Gpio_RGBLed_PB, XGPIO_IR_MASK);
    // Clear interrupt in NVIC
    NVIC_ClearPendingIRQ(GPIO1_IRQn);
}
```

Code D-10: ***Program code gpio.c file (IV).***

```
void GPIO_ADC_Handler ( void ){

     volatile int32_t adc_data;                // ADC data bus
     volatile uint32_t adc_signals;            // ADC signals bus
     uint16_t i, j=0;                          // parameters for loop
     uint32_t Frq;                             // Analyzed Frequency
     float NSamples;   // number of samples to detect length of spesific frequency signal
     float z0, z1, z2;        // to save previous and actual values (Goertzel Agorithm)
     float coeff, cos_a, omega; // Coefficient, cos(a) and rotational speed (Goertzel algorithm)
     int k;                                    // Parameter for Goertzel algorithm
     uint32_t Sampling_Frq;                    // ADC sampling rate
     float power;                              // Power get from algorithm in float
     uint16_t power_int;                       // Power in 16bits integer
     uint16_t Power_Setpoint, E_Setpoint;      // Setpoint of power and error

        /*------- Initial Conditions -------*/
        /*      Detected Frequency = 10 kHz
                Number of Samples = 400
                Sampling Frequency = 80 kHz
                Error Setpoint = 3000
                Power Setpoint = 60000        */
/* Note: Sampling Frequency and N° of sample was adjusted to obtain an accurate response */
     Frq = 10000;
     NSamples = 480.0;
     //z0 = 0;
     z1 = 0;
     z2 = 0;
     Sampling_Frq = 77000;
     Power_Setpoint = 6000;
     E_Setpoint = 3000;

// Goertzel algorithm applied to data
     k = (int) (0.5 + ((NSamples * Frq) / Sampling_Frq));
     omega = (2.0 * M_PI * k) / NSamples;
     cos_a = cos(omega);
     coeff = 2.0 * cos_a;
```

Code D-11: ***Program code gpio.c file (V).***

```
  adc_data = ((XGpio_DiscreteRead(&Gpio_ADC_Data_Control, ARTY_A7_ADC_DATA)) & 0xfff0);

    if (adc_data > E_Setpoint){

      for (i = 0; i < NSamples; i++){

      // Get data and signals from GPIO

      adc_data = ((XGpio_DiscreteRead(&Gpio_ADC_Data_Control, ARTY_A7_ADC_DATA)) & 0xfff0);

      adc_signals = XGpio_DiscreteRead(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS);

      // drdy_out signal is bit 8 of adc signal bus

      // doing a bitwise AND operation of adc signals bus with all 0s except the 8th bit,

      //drdy value can be get

        if (adc_signals & 0x0080){

            // The port 0 of GPIO ADC is selected as output to write data

            XGpio_SetDataDirection(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0xfffff000);

            // the Goerz_Started (bit 9) is set to not modify data_enable

            // (aux2 is also selected: 00011010b -> 0x1A) : 0x

            // the write enable is also 0 in previous case

            XGpio_DiscreteWrite(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0x11A);

            z0 = coeff * z1 - z2 + adc_data;

            z2 = z1;

            z1 = z0;

            XGpio_DiscreteWrite(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0x01A);

            // Port0 drives control signals. Set bus to be input.

            XGpio_SetDataDirection(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0xffffffff);

          }

      }

    }

 /* Calculation of the real and imaginary part: real = z1 * cos_a - z2; imag = z1 * sin_a;

    power = sqrt(real^2 + imag^2) = sqrt(z1^2*cos_a^2 - 2*z1*cos_a*z2 + z2^2 + z1^2*sin_a^2)

    power = sqrt(z1^2 - 2*z1*z2*cos_a + z2^2)                                   */

    power = sqrt(z1*z1 + z2*z2 - 2*z1*z2*cos_a);

    // power in integer value. It is divided by 100 to have numbers within 16-bit vector.

    power_int = (uint16_t) (power/100);

    XGpio_DiscreteWrite(&Gpio_Data, ARTY_A7_Output_DATA, power_int);
```

Code D-12: ***Program code gpio.c file (VI).***

```
    // Determine if the input signal frequency correspond to Frq
    if (power_int > Power_Setpoint) {
        XGpio_SetDataDirection(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0xfffff000);
        XGpio_DiscreteWrite(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0x021A);
        for(j = 0; j < 5; j++){}
        XGpio_DiscreteWrite(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0x001A);

        // Port0 drives control signals. Set bus to be input.
        XGpio_SetDataDirection(&Gpio_ADC_Data_Control, ARTY_A7_ADC_SIGNALS, 0xffffffff);
    }

    XGpio_InterruptClear(&Gpio_ADC_Data_Control, XGPIO_IR_MASK);
    // Clear interrupt in NVIC
    NVIC_ClearPendingIRQ(GPIO_ADC_IRQn);

}
```

Code D-13: ***Program code gpio.c file (VII).***

E. ANNEXE: SCHEMATIC

The folder includes in PDF the design of ADC Diagram, Clock and resets Diagram and the complete diagram of the modem. The folder also includes PDF of Nexys 4DDR board schematic.

This page intentionally left blank