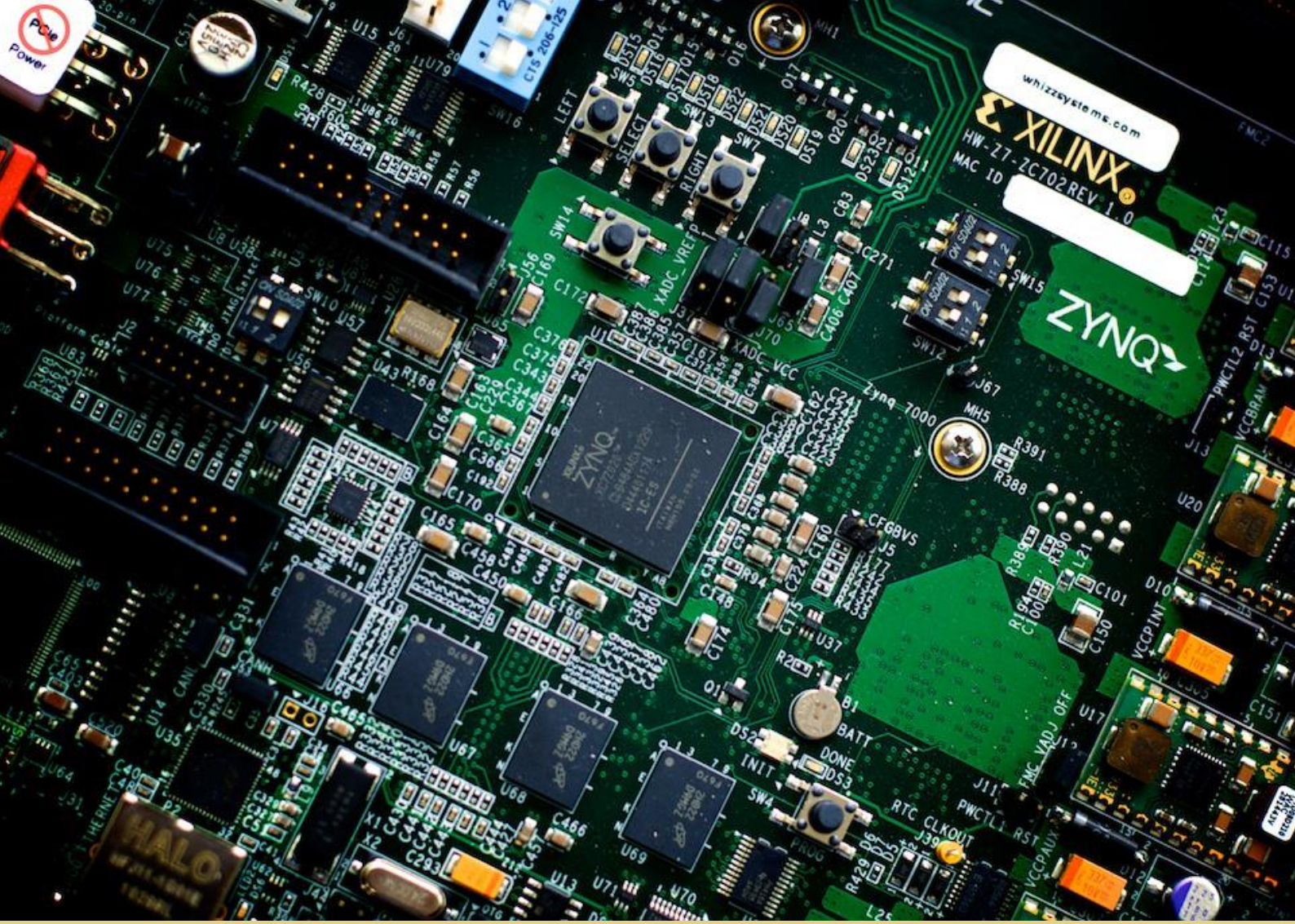


João Alexandre Abreu Ferreira

Real-time implementation of combined Ring-type Magnitude Modulation and LINC techniques on SoC Zynq-7000 Architecture

Dissertação submetida para obtenção do grau de Mestre em Engenharia Eletrotécnica e de Computadores, Área de Especialização em Telecomunicações

Setembro de 2015

· U C ·

Universidade de Coimbra

Real-time implementation of combined Ring-Type Magnitude Modulation and LINC techniques on SoC Zynq-7000 Architecture

João Alexandre Abreu Ferreira

Dissertação para obtenção do Grau de Mestre em

Engenharia Eletrotécnica e de Computadores

Orientador: Doutor Marco Alexandre Cravo Gomes
Co-Orientador: Doutor Vítor Manuel Mendes da Silva

Júri

Presidente: Doutora Maria do Carmo Raposo de Medeiros
Orientador: Doutor Marco Alexandre Cravo Gomes
Vogais: Doutor Mário João Simões Ferreira dos Santos

Setembro de 2015

Agradecimentos

Chegou ao fim mais uma etapa da minha vida. Este trabalho é precisamente o culminar desta viagem que agora dá lugar a outra, a da vida profissional. Com isto, não posso deixar de agradecer à minha família mais próxima, em especial aos meus pais é à ”mana”, que estiveram sempre ao meu lado e me apoiaram ao longo desta jornada.

Agradeço também aos meus amigos e colegas de curso, que me acompanharam na vida de estudante, quer tenha sido para a praxe, jantares, grandes maratonas de estudo ou simples procissões à máquina do café, para 15 minutos de conversa.

Aos professores Marco Gomes e Vítor Silva, pelo tempo que me dedicaram ao longo deste trabalho, especialmente quando parecia não haver soluções à vista, e também a todos os professores do DEEC, pela dedicação e entrega que mostram todos os dias ao leccionar este curso. Também à professora Natália Reis, pela ajuda que me deu para melhorar a escrita em Inglês em todo este trabalho.

Aos meus colegas de laboratório, que estiveram sempre disponíveis para ajudar.

A todos,
Muito Obrigado.

Abstract

The development of low-cost systems is a constant goal of Telecommunications engineering, where bandwidth and transmitting power are resources that need to be optimized, especially in mobile devices. In these cases, in which it is necessary to use power amplification (PA) for radio-frequencies (RF), power efficiency becomes one of the main focuses. In most cases, amplification is achieved through high power amplifiers (HPA) with a high level of linearity, which is a requirement in case the signals to transmit have high peak-to-average-power ratio (PAPR), which is typical in signals with high spectral efficiency. One way to avoid this problem is to use the Linear Amplification with Nonlinear Components (LINC) technique. However, digital LINC implementation has large oversampling requirements, as well as its power efficiency gains are limited at the LINC's combiner stage by the PAPR of the signals to transmit.

In the project GLANCES (UID/EEA/50008/2013) from the Instituto de Telecomunicações, a new transmitter front-end architecture was envisioned. Leveraging on the decomposition of high order constellations as a sum of offset-quadrature phase sift keying (OQPSK) signals, is it possible to use LINC techniques on the separate amplification of each OQPSK component, with considerable gains in efficiency due to the much lower OQPSK signal's PAPR. This performance can even be leverage by a new proposed Ring-Magnitude Modulation (RMM) technique, that warranties a perfect amplitude ring shape of a bandwidth limited OQPSK signal at the input of the LINC amplifier.

This thesis aims to develop a proof of concept of the described system, working in real time and supported by a field programmable gate array (FPGA) architecture: Zynq-7000 system on chip (SoC), to implement LINC techniques together with ring-type magnitude modulation (RMM), with two different *hardware* approaches: 1) a look-up table (LUT) based implementation for each of the LINC and RMM techniques and 2) a LUT based implementation for RMM and a hardware module to perform the LINC decomposition, based in its vectorial form.

Keywords

LINC, RMM, OQPSK, FPGA, SoC, LUT, Real Time, Signal Processing

Resumo

O desenvolvimento de sistemas de baixo custo é um marco constante na ciência das Telecomunicações, onde a largura de banda e a energia dispendida para transmissão são meios escassos que precisam de ser otimizados, principalmente em dispositivos móveis. Nestes casos, em que é necessário recorrer a amplificação de potência para radio-frequências (RF), a eficiência energética torna-se num dos principais focos de atenção. Na maioria dos casos, a amplificação é conseguida através de amplificadores de alta potência (HPA) com elevada linearidade, linearidade essa que é um requisito importante caso o sinal a transmitir tenha uma elevada relação entre a potência de pico e a poteência média (PAPR), o que é habitual em sinais com alta eficiência espectral. Uma das respostas a este problema é uma técnica chamada Amplificação Linear através de Componentes Não-Lineares (LINC). No entanto, a implementação digital LINC tem requisitos elevados de sobreamostragem, e os seus ganhos de potência são limitados no combinador LINC pelo PAPR dos sinais a transmitir. No projeto GLANCES (UID/EEA/50008/2013) do Instituto de Telecomunicações, uma nova arquitectura de transmissores foi desenvolvida. Baseando-se no facto de constelações de maior ordem poderem ser decompostas numa soma de sinais com modulação digital de fase em quadratura OQPSK, mostra-se ser possível usar técnicas LINC na amplificação de cada uma das componentes OQPSK, com ganhos consideráveis que se devem ao PAPR muito mais baixo apresentado por estas constelações. Este desempenho pode ainda ser melhorado por uma técnica inovadora: Modulação de Mangitude em Anel (RMM), que garante a forma anelar da amplitude de um sinal OQPSK de banda limitada à entrada do amplificador LINC. Este projeto de tese pretende desenvolver uma prova de conceito de um sistema de tempo real, suportado por uma arquitetura FPGA Zynq-7000 SoC para implementação conjunta de decomposição LINC e de RMM, com duas aproximações diferentes ao problema, feitas em *hardware*: 1) um sistema baseado em tabelas de consulta (LUT) para cada uma das técnicas LINC e RMM e 2) um sistema baseado em LUT para a implementação da RMM e um módulo de cálculo em hardware para a decomposição LINC, na sua forma vetorial.

Palavras Chave

LINC, RMM, OQPSK, FPGA, SoC, LUT, Tempo Real, Processamento de Sinal

Contents

1 Introduction **1**

1.1 Objectives 5

1.2 Dissertation Outline 6

1.3 Thesis framework and contributions 6

2 LINC systems **7**

2.1 Basic concepts for LINC systems 8

2.1.1 Angle decomposition 9

2.1.2 Vector decomposition 10

2.1.3 LINC branches matching 11

2.2 Digital LINC transmission system 11

2.2.1 Parameter-imposed limits 12

3 Magnitude Modulation **15**

3.1 The Magnitude Modulation Principle 16

3.1.1 Look-Up Table Based Approach 18

3.2 Ring-type Magnitude Modulation applied to OQPSK signals 19

3.2.1 LUT scheme 19

3.2.2 Table size and parameters 20

3.2.3 Symbol storage and search 21

3.2.4 Final acknowledgments 21

4 Architecture Design **23**

4.1 Hardware and Workspace 24

4.1.1 ZC702 24

4.1.2 FMC30RF 25

4.1.3 Other equipment 27

4.2 System Architecture and Implementations 28

4.2.1 Project backbone - Stellar IP 29

4.2.2 RRC filter - Core Generator 32

4.2.3 LINC and RMM - Vivado HLS . . . 34
4.2.4 Complete project - Xilinx ISE . . . 39
4.2.5 Board programming - Xilinx SDK . . . 43
4.2.6 Board communication - Microsoft Visual Studio . . . 44
4.2.7 External acquirements - GNU-Radio . . . 45

5 Implementation and Simulation 47
5.1 ZC702 total resources' capacity . . . 48
5.2 Blocks' specifications and physical occupation . . . 48
5.2.1 Generator with RMM and upscaling . . . 49
5.2.1.A Timing and resources . . . 51
5.2.1.B Bit rate . . . 51
5.2.2 Root-Raised Cosine Filter . . . 52
5.2.2.A Timing and resources . . . 52
5.2.2.B Bit rate . . . 54
5.2.3 LINC decomposer . . . 54
5.2.3.A Timing and resources . . . 54
5.2.3.B Bit rate . . . 55
5.3 Complete system . . . 56
5.3.1 Original system' occupation . . . 56
5.3.2 System #1's occupation and results . . . 57
5.3.3 System #2's occupation and results . . . 58
5.4 Final comparison and observations . . . 59

6 Conclusions 61
6.1 Future work . . . 62

A Implemented ISE VHDL schematic 67

B FIR Filter Implementation 69

C LINC Look-Up Table 73

D C++ Code of the Implemented Blocks in HLS 75
D.1 LINC Calculator . . . 76
D.2 LUT LINC . . . 77
D.3 Generator . . . 78

E VHDL Code of the Implemented Snippets in ISE **81**
E.1 Counter 82
E.2 RMM activator 83
E.3 Output channel selector 83

List of Figures

2.1 LINC decomposition technique using two different representations. . . . 8
2.2 Basic digital LINC transmission system. 12

3.1 Generic SC transmitter scheme. 16
3.2 Magnitude modulation principle. 16
3.3 MPMM principle. 17
3.4 Generic LUT-based Ring-type Magnitude Modulation transmitter scheme [1]. 18
3.5 Considered buffer scheme 21
3.6 Diagrams of the transition path between the constellation's symbols of a ring-type magnitude modulated OQPSK signal without RMM (3.6a) and with RMM (3.6b). 21

4.1 Chosen boards. 25
4.2 High Level Block Diagram of Zynq-7000 XC7Z020 AP SoC. 26
4.3 High Level Block Diagram of FMC30RF. 26
4.4 FMC30RF supplied firmware schematic. 27
4.5 Complete diagram of the hardware setup. 28
4.6 System flow. 29
4.7 Complete software schematic. 29
4.8 Stellar IP workspace. 30
4.9 Generated Stellar IP design. 32
4.10 Vivado HLS workspace. 34
4.11 Vivado HLS main action buttons. 35
4.12 Generator block diagram. 36
4.13 LINC calculator block diagram. 37
4.14 LUT LINC block diagram. 38
4.15 Xilinx ISE workspace. 39
4.16 Graphic implementation of the Counter block. 40
4.17 Implemented code for left button. 41

4.18 Implemented code for right button. 41
4.19 ZC702's assigned buttons for the above mentioned code blocks. 42
4.20 Block-diagram of the implemented parts of the project. 42
4.21 Xilinx SDK environment. 43
4.22 Microsoft Visual Studio environment. 45
4.23 GNU-Radio project. 46

5.1 Histogram of RNG. 50
5.2 Comparison of filters #1 and #2 for the same random sequence. 53
5.3 Output of RRC filter - LUT implementation 57
5.4 Output of LINC left branch - LUT implementation 58
5.5 Output of RRC filter - Calculator implementation 59
5.6 Output of LINC left arm - Calculator implementation 59

A.1 Complete VHDL design, translated to RTL. 68

B.1 RRC filters details 71
B.2 RRC filter implementation details. 71

List of Tables

5.1 Available resources of ZC702's programmable logic (XC7Z020) 48
5.2 Generator spent resources and timing 51
5.3 RRCs' spent resources and timing 53
5.4 LINC calculator spent resources and timing 54
5.5 LINC LUT spent resources and timing 55
5.6 LINC implementations comparison 55
5.7 Original system's occupation 57
5.8 Complete system #1's occupation 57
5.9 Complete system #2's occupation 58
5.10 Occupation and comparison of all systems 60
5.11 Expected vs. final occupation results 60

B.1 FIR filter coefficients for $N_{sym} = 5$ 70
B.2 FIR filter coefficients for $N_{sym} = 7$ 72

C.1 LUT input addresses and corresponding output values 74

List of Acronyms

DAC	digital-to-analog converter
ADC	analog-to-digital converter
FIR	finite impulse response
FPGA	field programmable gate array
HPA	high power amplifier
LSB	least-significant bit
LFSR	linear feedback shift-register
RNG	random number generator
WFM	waveform memory
SDR	software-defined radio
CORDIC	Coordinate Rotation Digital Computer
LINC	linear amplification with nonlinear components
LUT	look-up table
FF	flip-flop
RAM	random access memory
GPIO	general purpose input-output
MM	magnitude modulation
MPMM	multistage polyphase magnitude modulation
NL	non-linear

OQPSK	offset quadrature phase shift keying
PAPR	peak-to-average power ratio
RC	rectangular clipping
RMM	ring-type magnitude modulation
RRC	root raised cosine
PLL	phase-locked loop
VCO	voltage controlled oscillator
HDL	hardware description language
VHDL	Very high speed integrated circuits Hardware Description Language
JTAG	Joint Test Action Group
LED	light-emitting diode
MUX	multiplexer
SC	single-carrier
RF	radio-frequency
SoC	system on chip
ASIC	application-specific integrated circuit
DSP	digital signal processor
MAC	multiply and accumulate
PS	processing system
PL	programmable logic
FMC	FPGA mezzanine card
LPC	low-pin count
FIFO	first-in first-out
HLS	high level synthesis

RTL	register-transfer level
UART	universal asynchronous transmitter/receiver
MIMO	multiple input, multiple output
FFT	fast Fourier transform
USB	universal serial bus

1

Introduction

The need for information in our world grows as a daily basis, and it is mainly supported by Telecommunications. This rising demand is targeting the focus of communication engineers to optimize power and spectrum efficiency and transmission rates, while minimizing systems' costs [2]. In the design of generic wireless radio-frequency (RF) transmitters, one of the components that need most attention is the high power amplifier (HPA), since it is a critical component [3]; the HPAs (that can perform linear or non-linear amplification) have direct influence on the overall cost, efficiency, bandwidth and linearity of the transmitter.

However, although the HPA component has such an importance to the transmission system, it does not mean that all the possible improvements that can be made have to be implemented on the HPA itself; the delivered signal has also an important role on the overall system performance. In fact, the signal's characteristics dictate HPA's features: while constant envelope signals allow for the use of low-cost power efficient non-linear (NL) amplifiers, typical modulated signals, with high spectral efficiency, present large envelope fluctuations (i.e. an high peak-to-average power ratio (PAPR)) and so require the use of expensive linear HPAs with poor power efficiency [3,4] in order to avoid signal distortion at the amplification stage. This way of thinking opens a vast set of solutions on signal conditioning (like PAPR reduction or envelope-control techniques [5, 6]), and linear amplification with nonlinear components (LINC) decomposition [7] fits easily in these requirements.

The LINC [3, 8] technique consists on decomposing a signal in two phase-modulated signals (or branches) that have lower linearity requirements than the original; due to this, the branches can be effectively amplified by NL amplifiers. This decomposition is clearly an advantage over sending just the original signal without any processing because the system can now benefit from low-cost and efficient signal amplification, rather than expensive linear amplifiers. The NL amplifiers are cheaper, have higher amplification capacity than linear amplifiers [4, 9], and are much more efficient. The original signal can be reproduced again by combining the two LINC branches, taking into account that they need to be synchronized and balanced [3, 7].

However, LINC is greedy in terms of spectrum [3], since the decomposition itself mirrors the envelope of the original signal: a highly variable input produces outputs with high phase content, which also contributes to spread the signal's spectrum. On the other hand, decreasing the decomposition angle θ to lower values can indeed increase the LINC combination system efficiency [1]. Besides this, the LINC technique has some nuances when implemented in digital domain: the non-linear phase variation of their branches produces spectral regrowth, which is not desirable; for that reason, a high oversampling rate is included in the system in order to minimize this effect. However, an *a priori*

excursion limiting on the LINC input signal can effectively reduce the high oversampling requirements.

So, the objective at this point is to emulate at most an amplitude-limited signal, in both upper and lower bounds. The modulation that is closer to bring such results *per se* is OQPSK, which has a controlled envelope and smooth transitions, evades zero-crossing and produces an inside gap. Choosing a modulation like OQPSK could be a severe limitation for any current transmission system, since it has low spectral efficiency; however, Dinis *et al.* [10] shows that any signal from a high-order spectral efficient modulation can be seen as the sum of several OQPSK signals, so LINC limitations can actually be easily surpassed without compromising transmission efficiency, and the LINC concept itself can then be extended to high-order constellations. In fact, the proposed schemes are considered as valuable solutions for improving power efficiency of next-generation millimeter-wave communications where the use of multiple NL amplifiers and antennas is foreseen [11].

Every transmission system of this type needs a pulse shaping filter (usually a root raised cosine (RRC) [12]), in order to make the transmitting signal more adequate to the communications channel, e.g., by limiting its bandwidth. An OQPSK, when filtered by such a structure - that contributes to elevate the signal's PAPR - looses a part of its characteristics, namely the shape of the upper and lower bounds of the signal. One way to evade this problem is to apply ring-type magnitude modulation (RMM), which arranges the signal in a way that when it is passed by the RRC, the output assumes a ring-type form, generating a properly conditioned input for LINC. The RMM technique does not affect noticeably the transmission performance nor does it spread the transmitted signal' spectrum, and can effectively reduce the PAPR and lower the bandwidth utilization [1].

So, a LINC system, applied through the amplification of bandwidth-limited OQPSK signals, can clearly benefit from a joint implementation with RMM [13], since RMM effects will improve the conditioning of the input signal to the LINC technique[1] and as so, there will be lower oversampling requirements posed by performing LINC's decomposition at digital domain, as well as improving LINC's efficiency at the combination stage.

But how should this set of techniques be implemented in practice? There are some possible ways such as: application-specific integrated circuit (ASIC)s, digital signal processor (DSP)s, simple field programmable gate array (FPGA)s or even FPGAs with an integrated processor (system on chip (SoC)s). Although all have strong and weak points. For this proof-of-concept, the chosen platform should have the possibility to make fast and accurate calculations, enough memory storage for all the look-up table (LUT) ta-

[1]Nowadays, multilevel LINC transmitters are also being tested with promising results concerning efficiency on signals with high PAPR [14].

bles that may be implemented and, above all, the capacity to transmit two RF complex signals (with an operating frequency approximately at $1.2GHz$ [2]), resulting from LINC decomposition. Based on these requirements, it is time to choose which platform to use.

ASICs are usually very specific, and as so they are optimized in terms of power or resource density [16]. However, this only makes sense when the application is already tested and approved, since a new modification in hardware presents superior costs than in other devices; due to this, ASIC design can be more expensive. In fact, although ASICs are known to be low-cost devices, this advantage only occurs because they are mass produced devices.

DSPs are very efficient and optimized processors when the main purpose is, for example, to implement a finite impulse response (FIR) filter: its optimized multiply and accumulate (MAC) structure provides a good option to choose from. Also, today's DSPs have simple and easy programming interfaces, and many have full MATLAB/Simulink [17] integration, as well as FPGAs do. However, some DSPs may not still be the best in class when compared to designs in FPGA that achieve massive parallelism [18]. Although current DSP technology is heading to parallel implementations of algorithms [19], it is still not enhanced as it is on FPGAs, which can imply lower processing speeds [18].

On the other hand, FPGAs provide the capability of designing a complete system from scratch, tweak it and perfect it according to the user's needs (power optimization, area, speed or timing) more easily than in ASICs [16]; also, they are optimized to process information in parallel at very high rates. In sum, FPGAs' programmable logic (PL) allows many degrees of freedom for the system designer. This is undoubtedly the best choice for a proof-of-concept, whose needs can be broad and varied [16].

For our real-time system, the implementation is not limited to low-level parallel processing, and, for example, a connection to a host PC can be an advantage, in order to implement a master-slave control system. This can be achieved with an embedded processor, which only a SoC can provide: full PL-processing system (PS) integration, where parallel processing is achieved by the PL (FPGA) and sequential computation is done by the PS (embedded chip). Ultimately, it is the best of two worlds in one place.

Consequently, the chosen platform was a SoC FPGA, more precisely a Zynq^{TM}-7000 All Programmable SoC ZC702 [20] Evaluation Kit by Xilinx [21], due to its processing capabilities and connections. However, this board does not include any RF transmitters, which are essential to this proof-of-concept, but the existence of two low-pin count (LPC) FPGA mezzanine card (FMC)[3] connectors can surpass this problem, be-

[2] This operating frequency was specified by project GLANCES [15], where a part of this thesis work is included.

[3] FMC is an ANSI/VITA standard [22] I/O connection designed to increase modularity in FPGA systems. It was created to be used between FPGAs and daughter cards.

cause there are many FMC compatible daughter cards designed to transmit in RF that are ZC702-compatible. Taking this into account, and the RF transmitter frequency proposed ($1.2GHz$), our choice fell on two FMC30RF[4] [23] by 4DSP [24], on its frequency range variant from $1.2GHz$ to $3.0GHz$. This board is compliant with the VITA 57.1 standard [22, 23] and has a DAC transmission (Tx) IQ modulator, with a bandwidth up to $30MHz$ per channel.

Since the chosen environment was a SoC, the signal's transformation will be fully-digital, which is an advantage due to the flexibility of digital systems over analog ones ([3,25,26]). However, the LINC decomposition can be made both digitally or analogically [27, 28].

Some implementations on FPGA were made concerning LINC [29] and magnitude modulation (MM) [30] [5] by using different approaches [31,32]. This work aims to present a full LINC-RMM transmitter, built on a Zynq-7000 architecture and employing different approaches rather than those widely used, with all the parameters optimized to get the most of the implemented techniques, in order to prove that such a system is a suitable candidate for highly efficient and low cost transmissions.

This thesis is also a part of an Instituto de Telecomunicações' investigation project GLANCES [15] which is the result of an intense collaboration between the Multimedia Signal Processing Group from IT-Coimbra and the Wireless Communications Group from IT-Lisbon, and whose final purpose is to investigate and create a power and spectrum efficient broadband wireless system, by studying signal processing methods and techniques like LINC and by building a complete communications system based on that. Besides the implemented transmission system, created in this thesis, an amplification system with LINC branches combiner will be built in hardware, in Lisbon.

1.1 Objectives

This thesis proposes to create a working prototype of a real-time system, built on a Zynq-7000 SoC architecture, that achieves a full RMM and LINC decomposition of any offset quadrature phase shift keying (OQPSK) signal, ready for transmission. Two different approaches to this problem were considered:

1. System with a LUT for both RMM and LINC blocks;

2. System with a LUT for the RMM block and a hardware calculation block for LINC;

[4] Model: FMC30RF 2-1-2-1

[5] This MM implementation was made by using its real-time variant MPMM, and not LUT-variant RMM.

Both systems will be detailed and compared in terms of performance, FPGA resources utilization and speed, in the subsequent chapters.

1.2 Dissertation Outline

This thesis is structured in six chapters. After the introduction, LINC transmission systems will be explored in Chapter 2, and both MM and the used variant RMM will be explained in Chapter 3. Following these, a concise explanation of all the software and hardware architectures with some implementation choices will be given in Chapter 4, followed by a detailed description of the performed implementation and simulations in Chapter 5. Finally, Chapter 6 discusses the main results and conclusions derived from this thesis work and presents some ways to develop any future work on this subject.

1.3 Thesis framework and contributions

This thesis work was carried out under the project GLANCES [15] (Generalized Linear Amplification with Nonlinear Components for Power and Spectral Efficient Broadband Wireless Systems, supported by Instituto de Telecomunicações – IT). Also, it is the proof-of-concept of the developed work by Simões [1], and so, the best approaches considered by Simões will be applied here.

2

LINC systems

The choice of HPAs to a generic wireless transmission system is very dependent of the requirements of power linearity, and it is known that linear HPAs have low energy efficiency [4,9] when compared to NL HPAs. The LINC technique [3,7,8] makes the use of highly efficient NL amplifiers in these transmission systems possible and still achieve linear power amplification.

This chapter is focused on the description of the LINC technique, on the possible ways to decompose a signal according to this method and the basics needed to implement such a system in an FPGA.

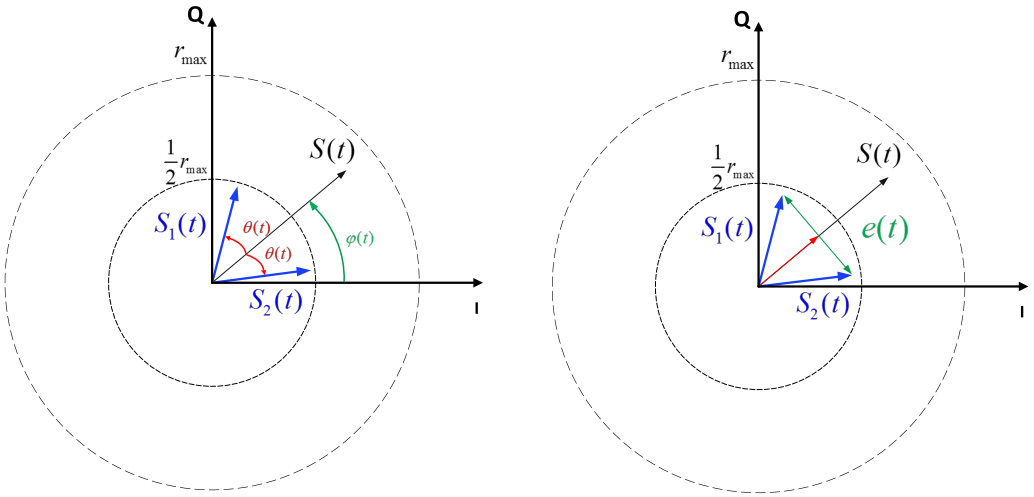
2.1 Basic concepts for LINC systems

The LINC technique consists in decomposing a time-varying signal with non-constant envelope as a sum of two constant envelope phase-modulated signals. This decomposition can be made through two different representations: one is based on angle decomposition, and the other is based on vector decomposition.

Our starting point is the generic representation of a signal, $S(t)$, which is given by:

$$S(t) = r(t)e^{j\phi(t)} \tag{2.1}$$

where both the instantaneous amplitude $r(t)$ and phase $\phi(t)$ vary with time. The time-varying amplitude can still be described as a constant amplitude phase modulated signal with an angle $\theta(t)$:



(a) LINC concept using θ.

(b) LINC concept using $e(t)$.

Figure 2.1: LINC decomposition technique using two different representations.

$$\begin{aligned} r(t) &= r_{max} cos(\theta(t)) \\ &= r_{max} \frac{e^{j\theta(t)} + e^{-j\theta(t)}}{2} \end{aligned} \tag{2.2}$$

where r_{max} is its maximum amplitude and $\theta(t)$ is known as the LINC branches' decomposition angle, as depicted in Figure 2.1. From now on, both the decompositions can be derived from these main equations.

2.1.1 Angle decomposition

The angle decomposition is very straightforward; merging 2.1 with 2.2, thus obtaining:

$$\begin{aligned} S(t) &= r_{max} \left[\frac{e^{j\theta(t)} + e^{-j\theta(t)}}{2} \right] e^{j\phi(t)} \\ &= \frac{r_{max}}{2} \left[e^{j(\theta(t)+\phi(t))} + e^{j(-\theta(t)+\phi(t))} \right] \end{aligned} \tag{2.3}$$

From 2.3 it is possible to infer that $S(t)$ has two distinct phase variations ($\theta(t)$ and $\phi(t)$), making it possible to describe 2.1 as the sum of two constant-envelope signals, $S_1(t)$ and $S_2(t)$:

$$S_1(t) = \frac{r_{max}}{2} e^{j(\theta(t)+\phi(t))} \tag{2.4}$$

$$S_2(t) = \frac{r_{max}}{2} e^{j(-\theta(t)+\phi(t))} \tag{2.5}$$

where:

$$S(t) = S_1(t) + S_2(t) \tag{2.6}$$

as depicted in Figure 2.1a. As this derivation has no restrictions, it was proven that any signal can be described as a sum of two constant-envelope signals.

Computationally, in order to calculate LINC's branches, it is only needed to set a maximum value r_{max} and then compute the angle $\theta(t)$ from 2.2, which is given by:

$$\theta(t) = \arccos\left(\frac{r(t)}{r_{max}}\right) \tag{2.7}$$

Always having in mind that r_{max} should be chosen as the maximum amplitude value that the signal can reach, or else some distortion will occur[1].

[1] A workaround to this problem will be discussed shortly.

However, regarding the implementation in the FPGA of the LINC angle decomposition, it is hard to calculate this solution, due to the computational weight of the arc-cosine function in conjunction with the implicit sines and cosines of Eqs. 2.4 and 2.5. Such a solution would spend at least two instances of CORDIC [33] blocks to perform these desired calculations[2], since one instance only calculates one function.

2.1.2 Vector decomposition

It is also possible to calculate the LINC branches via vector decomposition, where a vector $1 \pm je(t)$ [3] is defined as the transformation factor of $S(t)$ in $S_1(t)$ (plus sign) and $S_2(t)$ (minus sign). Using the trigonometric identities:

$$cos(\theta) + jsin(\theta) = e^{j\theta} \tag{2.8}$$

$$sin(\theta) = cos(\theta)tan(\theta) \tag{2.9}$$

on Eq. 2.4, we obtain:

$$\begin{aligned} S_1(t) &= \frac{r_{max}}{2} e^{j\phi(t)} \Big[cos(\theta(t)) + jsin(\theta(t)) \Big] \\ &= \frac{r_{max}}{2} e^{j\phi(t)} cos(\theta(t)) + j\frac{r_{max}}{2} e^{j\phi(t)} cos(\theta(t))tan(\theta(t)) \\ &= \frac{r_{max} cos(\theta(t)) e^{j\phi(t)}}{2} \Big[1 + jtan(\theta) \Big] \\ &= \frac{S(t)}{2} \Big[1 + jtan(\theta(t)) \Big] \end{aligned} \tag{2.10}$$

A similar derivation can be made for Eq. 2.5, obtaining the following:

$$S_2(t) = \frac{S(t)}{2} \Big[1 - jtan(\theta(t)) \Big] \tag{2.11}$$

Regarding these mathematical expressions, it is easy to understand that $e(t)$ is equivalent to $tan(\theta(t))$. Right now, the vectorial form seems to be no different of the angle form, since it also uses an angle expression to describe LINC decomposition. However, a closer look, with the aid of the mathematical expression:

[2] CORDICs are the reference HDL building blocks to perform angle calculation (such as sines, cosines, tangents and their respective arcs), as well as square roots and polar-to-rectangular conversion and vice-versa.

[3] j the imaginary unit and $e(t)$ an error vector that will be deduced shortly.

$$tan(\theta)^2 = \frac{1}{cos(\theta)^2} - 1 \tag{2.12}$$

concatenated with the Equation 2.7, shows that $e(t)$ can be described as:

$$e(t) = \sqrt{\frac{r_{max}^2}{r(t)^2} - 1} \tag{2.13}$$

dispensing the need of a direct angle calculation. Again, r_{max} should be previously defined in order to calculate this expression.

Since vectorial representation only needs to implement one CORDIC block for the square root (which in Chapter 5 will be compared to a simple memory unit (i.e., a LUT) in terms of resources occupation, for simplicity), it does not use as much FPGA area and resources as angle representation and it also allows to perform calculations with higher precision. For this reason, vectorial representation is the chosen type for our approach.

As it is mathematically proven, $S_1(t)$ and $S_2(t)$ have constant envelopes, and this lead to the possibility of using power-efficient non-linear amplifiers. So, indirectly, LINC decomposition increases the overall efficiency of a wireless transmission system.

2.1.3 LINC branches matching

On the receptor side, both LINC branches can be added to generate an amplified replica of $S(t)$. In theory, the signal $S_{received}(t)$ would be a perfect copy of $S(t)$; however in practice, this combination requires some care: $S_1(t)$ and $S_2(t)$ have to be perfectly synchronized and any unbalances between the transmitting branches can cause small deviations between the received signal and the transmitted signal.

The sets of equations (2.4) - (2.5) and (2.10) - (2.11) are mathematically equivalent; however, one of the sets may outperform the other in a digital implementation of the system, depending on how the two branches are physically generated [3, 25, 26]. As stated earlier, vectorial representation will tend to be the most used because the involved calculations are simpler and faster to execute on an FPGA[4].

2.2 Digital LINC transmission system

As referred to in Chapter 1, this implementation will be fully-digital, which is a common choice [3, 25, 26] due to the offered flexibility. To contextualize, a generic digital LINC transmitter is shown in figure 2.2.

[4]More on this topic will be presented in Chapter 4.

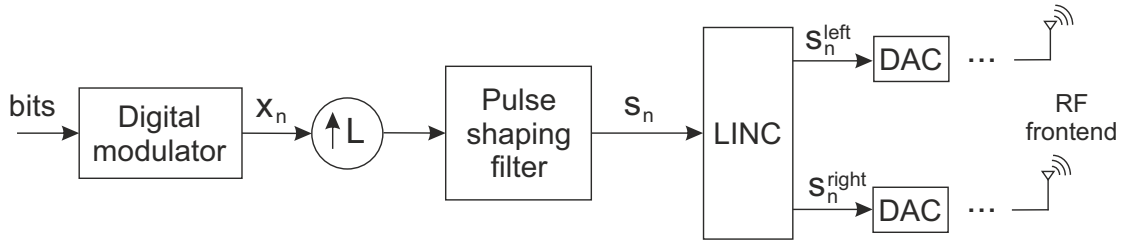


Figure 2.2: Basic digital LINC transmission system.

The entering information is assumed to be binary and it is coded by a digital modulator that shapes the input bits to output symbols (x_n). After this, the signal is oversampled by a factor L and passed through an RRC filter (this process is usually called pulse-shaping, which limits digitally the bandwidth of the signal to transmit). Finally, the digital signal s_n (equivalent to $S(t)$ in analog analysis) is ready to be decomposed by LINC, and then converted to analog to be amplified and transmitted.

The value of the oversampling factor L must be chosen wisely, since it will have direct effect on the signal spectrum of the LINC's components: an arbitrary choice that does not take the LINC's oversampling requirements [8] and the digital-to-analog converter (DAC)'s reconstruction filter into account and it may provoke peak regrowth, which is unwanted. A high oversampling rate avoid these problems, while lowers the needed complexity of this reconstruction filter, and in consequence, its cost [12].

As stated in Simões [1], the value of L was chosen to be 8, along with other important parameters for the design. This topic will be further discussed.

2.2.1 Parameter-imposed limits

When talking about the LINC's discrete-time decomposition, it is needed to take into account the DAC features (reconstruction filter characteristics and resolution) and the HPAs' saturation effect of $\theta(t)$ and $e(t)$. These effects may compel the LINC's input signal s_n to be clipped. In this case, θ and e are given by:

$$\theta(r_n) = \begin{cases} \arccos\left(\frac{r_n}{s_M}\right), & r_n \leq s_M \\ 0, & r_n > s_M \end{cases}, \tag{2.14}$$

$$e(r_n) = \begin{cases} \sqrt{\left(\frac{s_M}{r_n}\right)^2 - 1}, & r_n \leq s_M \\ 0, & r_n > s_M \end{cases}, \tag{2.15}$$

where r_n is the amplitude of the original digital signal, related to sample n and s_M is the LINC maximum amplitude level (or clipping level)[5]. As stated in [8], the polar clipping

[5] s_M is the digital-equivalent of r_{max}, on the previous analog analysis.

operation has a slightly superior performance, when compared to cartesian operation, this being the reason why polar clipping was chosen.
Assuming ideally balanced amplifiers and perfect combining, we can determine the transmitted signal s_c using the following equation (assuming an amplifiers' unit power gain):

$$s_c = s_{n1} + s_{n2} = \begin{cases} s_n, & |s_n| \leq s_M \\ s_M e^{j \arg(s_n)}, & |s_n| > s_M \end{cases} . \tag{2.16}$$

As stated earlier, such a system requires enough bandwidth to accommodate both signal's components, while it needs a perfectly balanced branch amplification in order to cancel the complementary terms of s_{n1} and s_{n2}, because any possible amplitude or phase unbalances between the two branches may result in significant performance degradation [1, 3, 8].

3

Magnitude Modulation

The HPA is a key component of the front-end of any RF transmission system, both for its role and for the great amount of energy it consumes. The spent energy can be lowered if proper conditioning is done to the transmitting signal, reducing the signal's excursion and, in consequence, its PAPR. On an OQPSK transmitting system that includes LINC and a RRC, the RMM implementation also takes a major role in signal conditioning, significantly improving the overall power and spectrum efficiencies. The RMM technique is then presented as a solution to improve system performance, because it offers the possibility to control the amplitude's upper and lower bounds, thus increasing energy efficiency for transmission without noticeable performance reduction, for any given input OQPSK signal.

This Chapter will explain the model of the RMM method for single-carrier (SC), thought to fit in the current transmission system.

3.1 The Magnitude Modulation Principle

Figure 3.1 shows a typical SC transmitter, composed by its basic building blocks.

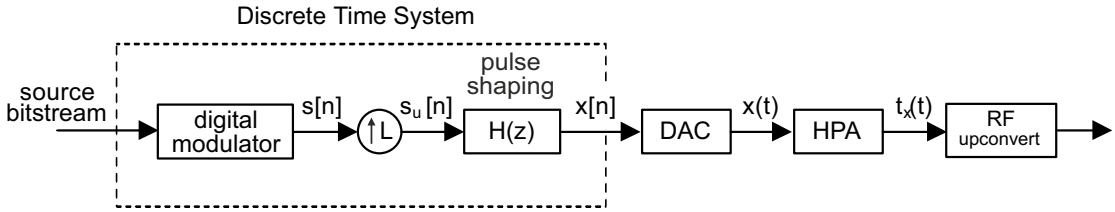


Figure 3.1: Generic SC transmitter scheme [34].

The pulse-shaping filter used to limit bandwidth is the main contributor for the transmitted signal high PAPR. In fact, as mentioned in Chapter 1, when the transmitted bits are mapped in constant-envelope constellations, it is the only contributor for the signal's PAPR. This effect makes room for a symbol readjustment operation to limit the signal's excursion (prior filtering).

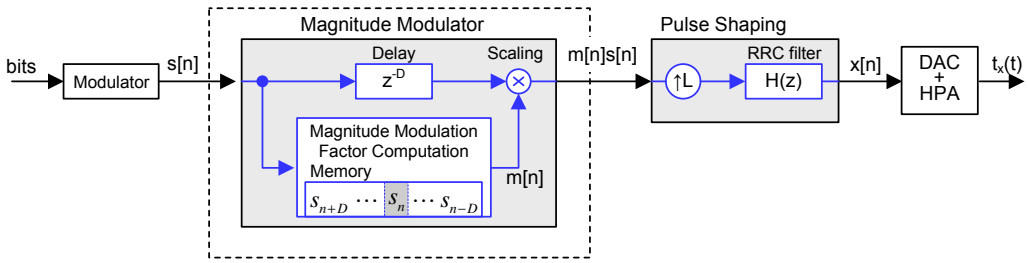


Figure 3.2: Magnitude modulation principle [34, 35].

The principles of MM are shown in more detail in Figure 3.2. The technique conjugates the knowledge of the pulse-shaping impulse response with the awareness of the signal being transmitted, with the magnitude modulation factor $m[n]$ that is applied to each symbol $s[n]$ being computed, taking into account $s[n]$'s closer neighbors.

Currently, there are two major techniques that are used to apply MM: one is the real-time calculator MPMM, which implements the RRC on its polyphase representation. In this method, the MM is as a cascade of basic blocks, as depicted in Figure 3.3, and only the current symbol is considered for a MM calculation. The other case is the pre-calculated LUT-based MM, whose implementation is reduced to a simple memory access[1] and in which a sequence of $2D+1$ symbols is used to calculate only one MM coefficient - the quantity of used symbols to calculate just one coefficient explains the complexity of the process and the difficulty of implementing it in real-time.

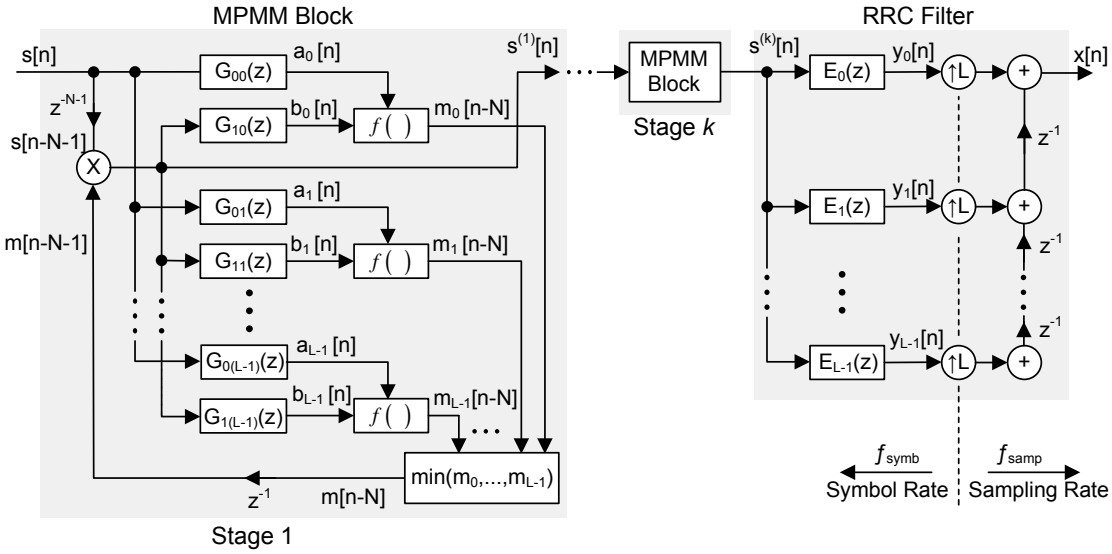


Figure 3.3: MPMM system.

Since the real-time computation of the multistage polyphase magnitude modulation (MPMM) coefficients require high processing rates and FPGA resources, the implemented system would not benefit much nor would it be capable of having a MM calculator over a LUT-based approach, due to the fact that the LUT system is far more efficient in terms of speed and occupied resources.

But, when talking about OQPSK signals, a different MM scheme can be applied, which is more adequate and provides PAPR reduction: it is the RMM. This technique is very similar to the MM-LUT, having just one difference: in this case, a lower bound to the signal's amplitude is also implemented, in order to decrease the decomposition angle

[1] Since the calculation of the MM coefficients is computationally heavy and complex, a real-time version of MM currently makes no sense.

θ and thus, prepare the signal for a more efficient LINC decomposition and addressing its requirements, after pulse-shaping filtering. In our case, this method will be used by employing a static memory (LUT) in the FPGA.

3.1.1 Look-Up Table Based Approach

The LUT-RMM method relies on a previous computation of all the coefficients, so that the true effort of a system which implements this method becomes reduced to a memory access. Figure 3.4 shows the implemented MM block, where 2-bit symbols enter sequentially to the current set.

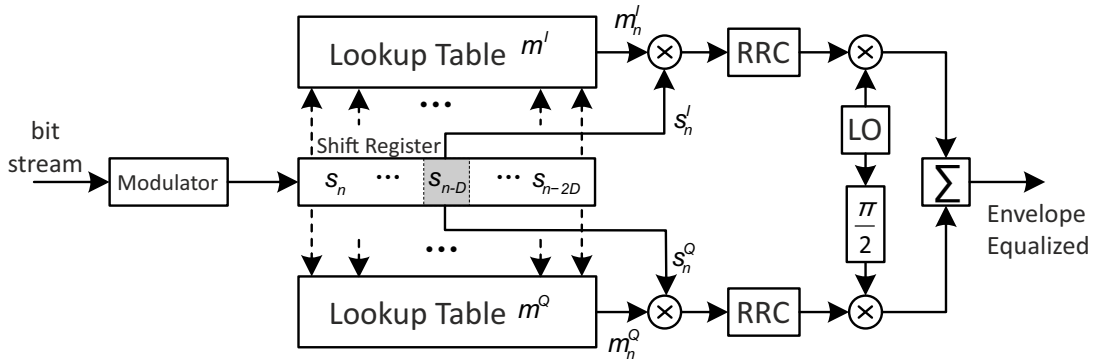


Figure 3.4: Generic LUT-based Ring-type Magnitude Modulation transmitter scheme [1].

The process of creation of the LUTs is described on Algorithm 1:

Algorithm 1 Concise Algorithm for Computation of RMM Coefficients' LUT [34].

do {

STEP 0*:* **Setup input data.**

$$\textit{Input Data} = \begin{cases} \textit{Signal to be MM} & \Leftarrow \text{ first iteration} \\ \textit{MM signal from previous iteration} & \Leftarrow \text{ other iterations} \end{cases}$$

STEP 1*:* **Filter data using a RRC filter.**

STEP 2*:* **Limit the magnitude of the filtered signal to the specified thresholds A_l and A_u.**

STEP 3*:* **Filter the resulting signal using a match RRC filter.**

STEP 4*:* **Sample the resulting signal to obtain the RMM sequence corresponding to the input data.**

} **while** (Signal limitation occurs in *STEP* 2)

STEP 5*:* **Output the RMM coefficients by performing the ratio of RMM signal from the most recent iteration to the input original sequence to be RMM.**

In this algorithm, a channel with noiseless transmission is emulated. The starting MM sequence is an all-ones vector, so that the algorithm's first iteration does not include MM. Then, the input sequence is upsampled, filtered through an RRC and evaluated according to the algorithm's clipping method. After this, the output signal is filtered again in a matched RRC, and the corresponding symbols to the input data are obtained through sampling. Finally, the quotient between the output and the input sequences is computed, and this ratio is stored as a MM coefficient. The iterations are repeated until no more clipping is applied.

3.2 Ring-type Magnitude Modulation applied to OQPSK signals

The original MM procedures only limit the signal's envelope maximum excursions, since their appliance is made in SC transmissions that employ linear HPAs. However, by using LINC techniques on the system, energy efficiency of the front-end is mainly limited by the combiner's efficiency [3, 36]. According to Simões [1], a reduction of the decomposition angle θ can effectively increase the combiner's efficiency and reduce the signal's PAPR. This can be translated to a lower bound limiting on the LINC input signal's amplitude.

The RMM technique appears as an effective excursion control method that can fit those requirements: its lower and upper bounds limiting helps generate a properly conditioned input for LINC, while not diminishing the transmission's performance. However, it would be desirable to have an initial signal that could be as approximate to these requirements as possible, in order to simplify the modulation methods. OQPSK appears as the best candidate, since it has an upper and lower limited envelope, that evades zero-crossing and has smooth transitions.

This technique was thought to fit a transmission system, which includes a pulse-shaping filter (an RRC in this case): the final ring-shaped form is only obtained after filtering, i.e., where the LINC decomposition technique is applied.

3.2.1 LUT scheme

The modulated symbols are stored sequentially in an array, with a total capacity of $2D+1$ symbols. When symbol $s[n]$ is first stored, it is used with the previous $2D$ samples to calculate the amplitude factor of the symbol $s[n-D]$, which is the current middle symbol of the stored set, , i.e. the LUT-MM scheme introduces a processing delay of D symbols. After this computation, all symbols advance one position in the stored array,

including $s[n]$, thus making space for one new symbol, that is $s[n+1]$. Now, the scaling factor obtained is multiplied by $s[n-D+1]$, and the cycle continues endlessly. It is easy to comprehend that for any finite D length and bits to represent a symbol, there will be a limited number of possible cases, and in consequence, a limited number of scaling factors. So, for a constellation with M symbols, this number of factors is equal to M^{2D+1}, which is equal to the number of entries of the necessary LUT to implement.

The value of D is not arbitrary; it should be chosen accordingly to the RRC filter's characteristics, namely roll-off and length, to make sure that all the symbols that contribute for the signal's amplitude are included in the calculations. The filter's length is given by the following expression:

$$length_{RRC} = 2 * N_{sym} * L_{oversampling} + 1 \tag{3.1}$$

where N_{sym} is the number of considered symbols and $L_{oversampling}$ is the oversampling factor[2]. In order to perfectly match the designs of the RRC and the MM LUT, the number of affected symbols must be the same; this means that $N_{sym} = D$. However, an implementation of the RRC filter with a value of N_{sym} superior than the perfectly-matched can also bring similar results [3].

As this technique requires D symbols to be stored *before* they are actually used, the RMM block inserts a time delay that is equivalent to the symbol time T_{symb} multiplied by D symbols.

3.2.2 Table size and parameters

As stated earlier, considering a LUT-MM memory of $2D+1$ symbols and a constellation of size M, the computed MM LUT will have M^{2D+1} entries. In our case, D was chosen to be equal to 3, in order to have a reasonable contrast between the LUT states and not an enormous table, to acknowledge system's timings and resources. Also, the implemented constellation is an OQPSK[4], which leads to an M factor equal to 4; the constellation size can never be neglected since it greatly affects the overall length of the generated LUTs. These values originate two 16384-entries LUTs (one for I and one for Q), where is being employed rectangular clipping (RC).

Regarding the RRC implementation with a specification of having a roll-off factor of 25%, a filter was designed (spreading over 7 past and future symbols) and an oversampling factor $L = 8$.

[2]Note that the length of the RRC filter is always odd.

[3]It is taken as example the study made in Simões work [1], where $D = 5$ and $N_{sym} = 7$, without any loss of performance; more on this topic will be developed on Chapter 5, Sub-chapter 5.2.2.

[4]Refer to project GLANCES [15].

3.2.3 Symbol storage and search

The way symbols enter the shift-register is intrinsically connected to the MM table algorithm. In the current case, and due to the method used to create the LUTs, any new symbol is appended on the right, and all the symbols are composed by their quadrature component (Q value), followed by their in-phase component (I value), in a right-to-left scheme, as stated in figure 3.5. This value is implicitly casted to a 14-bit unsigned integer (i.e., $2D+1$ times 2 bits/symbol), being the bit I_{n+D} the least-significant bit (LSB); this 14-bit number is used as the input of the RMM LUTs, which gives the corresponding coefficients to be multiplied by the current middle symbol.

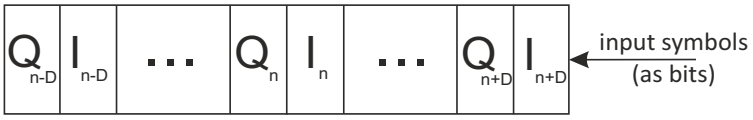
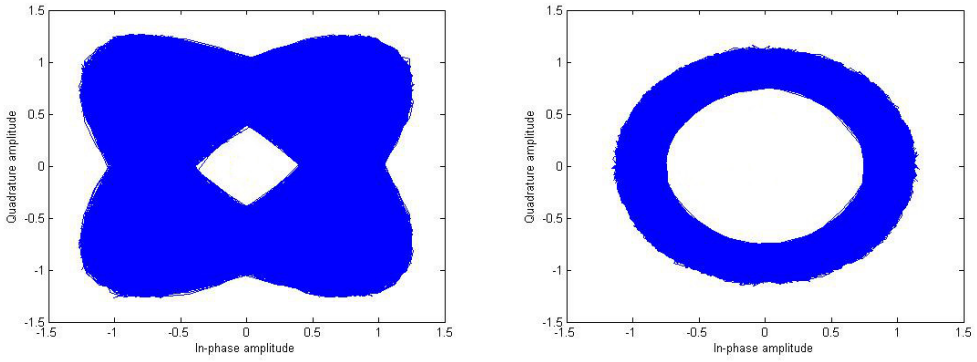


Figure 3.5: Considered buffer scheme.

3.2.4 Final acknowledgments

The expected result of ring-type magnitude modulation on an OQPSK modulated signal is illustrated as an example in Figure 3.6 ([1]). It is shown that the samples that overpass the imposed amplitude threshold are uniformized and that the inner gap is wider.



(a) (b)

Figure 3.6: Diagrams of the transition path between the constellation's symbols of a ring-type magnitude modulated OQPSK signal without RMM (3.6a) and with RMM (3.6b).

According to Simões [1], A_l and A_u will be set to 0.8 and 1.1 (which are the applied thresholds on Figure 3.6b), respectively; this choice is corroborated by Simões work [1] on LINC acceptance of various lower and upper bounds testings.

4

Architecture Design

In the previous chapters, both LINC techniques and RMM were theoretically explained, always having in mind the final purpose of this thesis. However, some developments and equations that seem rather easy to develop in theory, become quite complex to implement on FPGAs (for example, LINC angle representation would require CORDIC blocks to compute sines, cosines and arc-tangents, which are more resources consuming than vector's representation, which only needs a square-root calculation). Also, it is always necessary to know that all the resources are limited, which conditions the development of optimal code to maximize operation speed and minimize the occupied area.

This chapter aims to explain the developed implementation, both in hardware terms (devices used and testing workspace) and in software terms (utilized programs and implementations). This chapter opens the way to Chapter 5, where a more complete overall implementation will be discussed, along with the obtained results.

4.1 Hardware and Workspace

As referred to in Chapter 1, some generic requirements would be necessary (e.g. high processing power and speed), in order to accomplish all the stages of the transmission system correctly. The selected board was a Zynq^{TM}-7000 All Programmable SoC ZC702 [20] Evaluation Kit by Xilinx [21] [1], with two FMC30RF[2] boards [23] by 4DSP [24], on its' frequency range variant from 1.2GHz to 3.0GHz, because the system was thought to be operating with a 1.2GHz carrier RF frequency[3]. A photo of the utilized boards is shown in Figure 4.1.

4.1.1 ZC702

The ZC702 is the main board of the hardware setup: it is here that all implementations are deployed and whose physical resources are used. The ZC702's features include a JTAG interface for communication and bitstream upload, an Ethernet interface via a RJ-45 connector, a USB-to-UART bridge, an I^2C bus [37], two FMC LPC connectors, 1GB DDR3 component memory and general purpose input-output (GPIO); also, the SoC contains an integrated PS and PL (depicted in Figure 4.2), whose both parts run independently and the PS holds two ARM Cortex^{TM}-A9 application processors, internal memories and external memory interface and peripherals such as the ones mentioned above. This thesis will only step through the most relevant board details; however, a more complete description is available in [38].

[1] Zynq SoC device reference: XC7Z020-1CLG484C.

[2] Model: FMC30RF 2-1-2-1.

[3] From now on, the selected FPGA will just be called ZC702, and the daughter cards will be called FMC30RF, for simplicity.

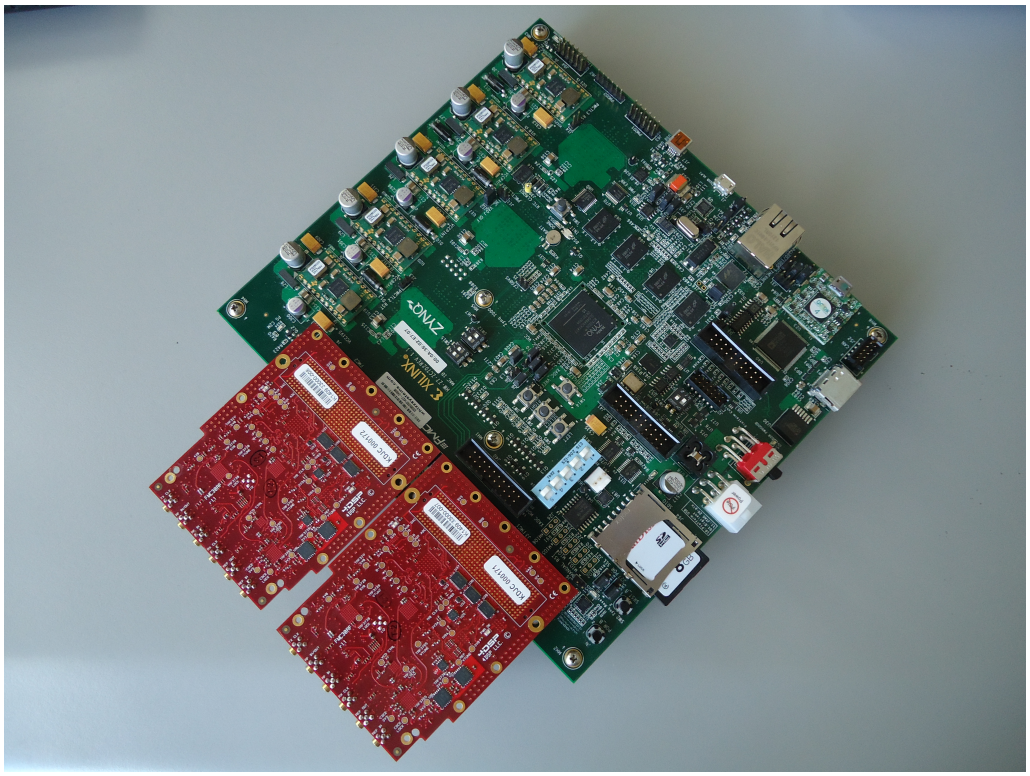


Figure 4.1: Board set.

This board's PL has a $200MHz$ system clock source and programmable user clocks. For the implemented design, a $100MHz$ clock is generated to feed all the hardware components. Also, it has a PS clock source, working at a frequency of $33.3MHz$. Our system will be completely developed inside the PL, since all the created blocks work in VHDL.

4.1.2 FMC30RF

The FMC30RF boards are RF transceivers with a dual DAC transmitter IQ modulator with up to $30MHz$ bandwidth for each channel[4] and a dual analog-to-digital converter (ADC) receiver IQ demodulator, with an on-board PLL/VCO. The DAC offers a transmission rate up to $250Msamples/s$ with a 12-bit precision. A high level block diagram is shown in Figure 4.3, and a broader description of this board is available in [40].

The original firmware schematic of the board is also presented, where the created structures are depicted with more detail, like the reception FIFO or the transmission WFM, which is a memory block used to store the values before transmission. This block works like a FIFO, but with slight differences: here, the values are only refreshed if a direct command is sent to the block. While that does not happen, the WFM keeps send-

[4] Although 4DSP announces up to 60 MHz of bandwidth, this value is a sum of both the I and Q bandwidths, as stated in [39].

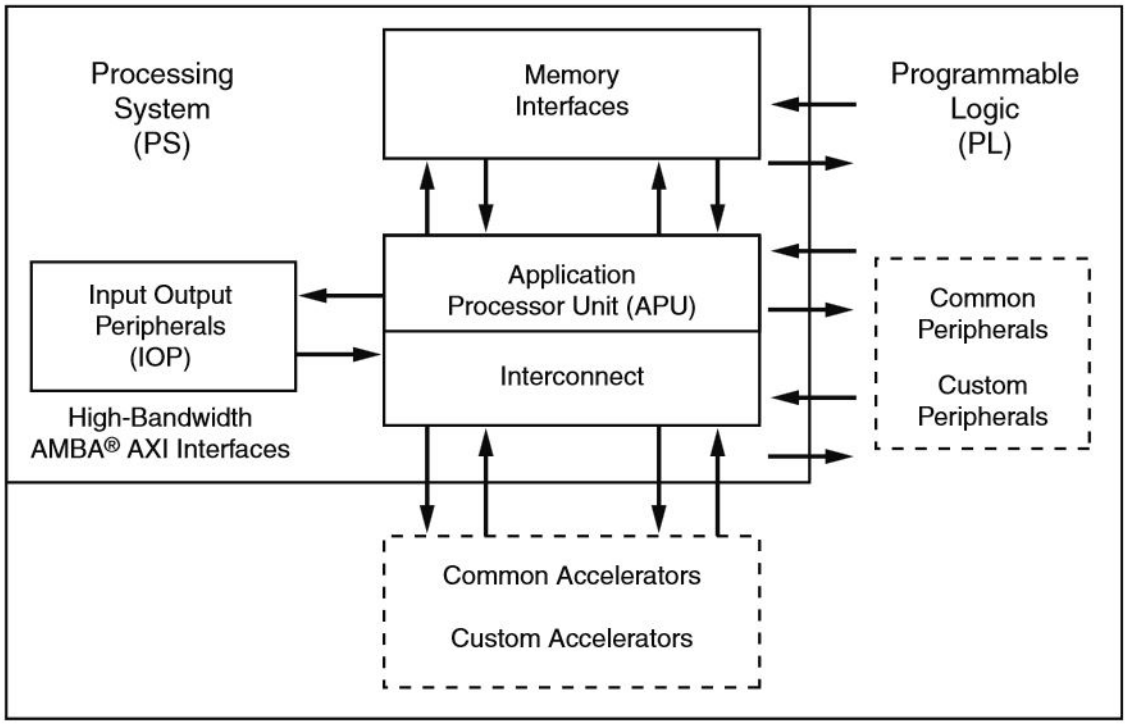


Figure 4.2: High Level Block Diagram of Zynq-7000 XC7Z020 AP SoC.

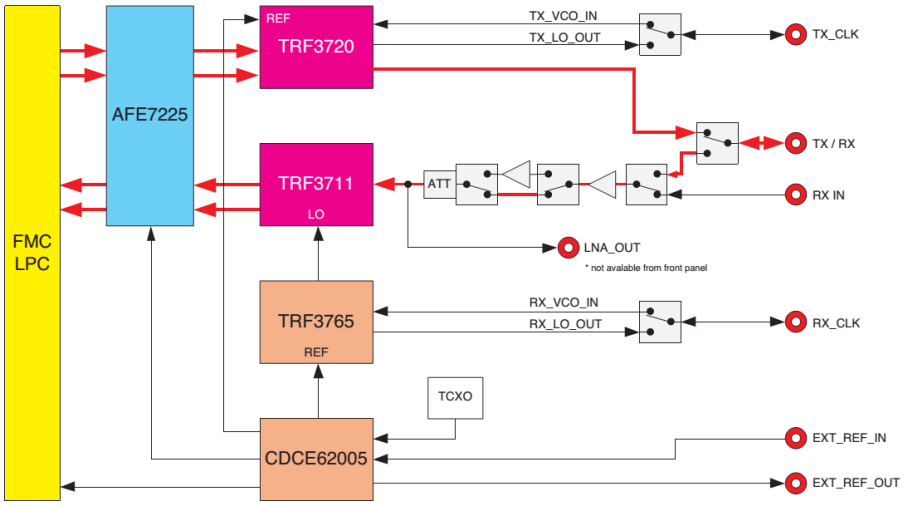


Figure 4.3: High Level Block Diagram of FMC30RF.

ing the same set of data continuously, working as a circular FIFO. This could become a problem if the objective is to generate a transmission channel; however, it has a simple solution, as it will be discussed in Sub-chapter 4.2.5.

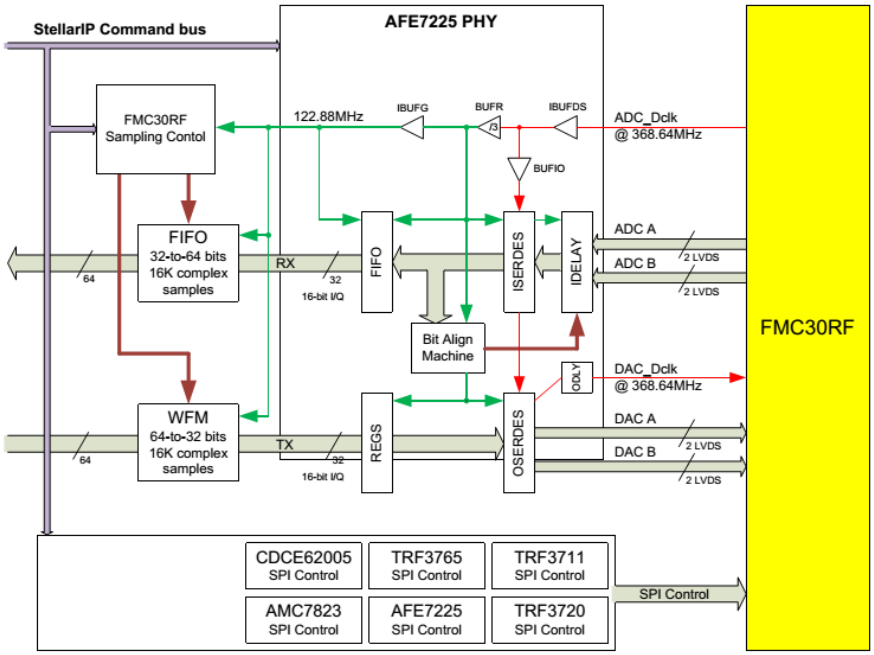


Figure 4.4: FMC30RF supplied firmware schematic.

4.1.3 Other equipment

Besides the above mentioned boards, some equipment was used to test and acquire the output signals, namely an HP 8591E Spectrum Analyzer, which has a range that covers the chosen RF transmission frequency, and a USRP B210 software-defined radio (SDR) board [41], which is a transceiver board that also supports the reception of the desired frequency. Both equipments present evaluations in real-time. Host computer-board interaction was also used to acquire directly the generated data from the RRC and LINC blocks' outputs, which is considered to be more reliable to see if the blocks are working correctly.

The hardware setup is depicted in Figure 4.5. The first equipment was used to analyze the spectrum and test in part the validity of the output signals that came from both FMC30RF boards (LINC branches), and the second one was used to see the same outputs, but in both time and frequency domains and the constellation of the signal. While the spectrum analyzer requires no software, the USRP board needs a host computer with USB[5] that runs GNU-Radio [42]. GNU-Radio is an open-source software that works

[5]Preferably, USB 3.0 or better to provide decent dataflow speeds (USB 2.0 will also work).

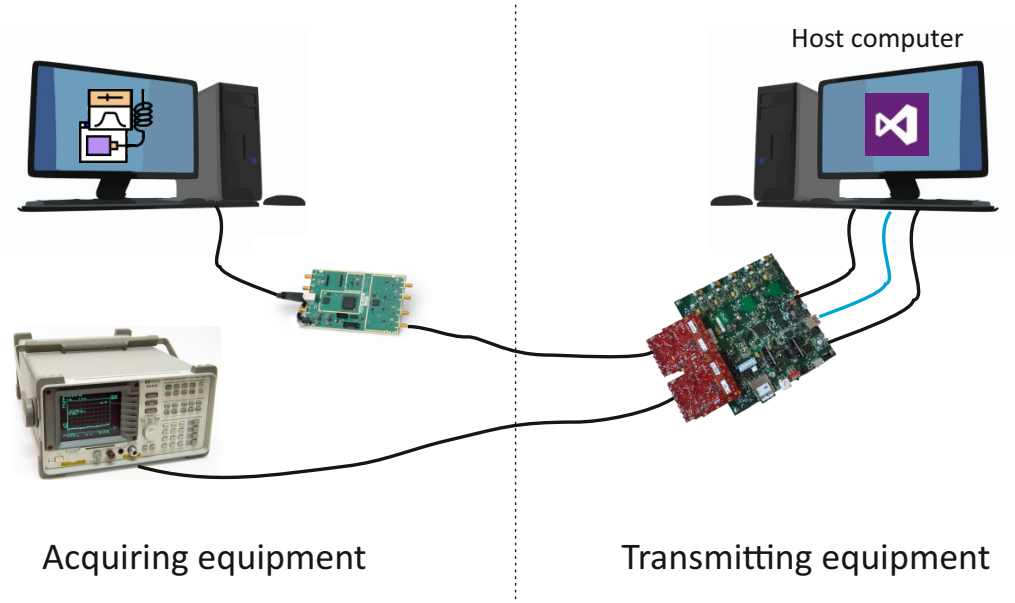


Figure 4.5: Complete diagram of the hardware setup.

based on a programming block environment to manage compatible SDR boards. However, since this material was only used to measure and acquire some signals' elements, a detailed explanation on its working will not be given on this thesis.

4.2 System Architecture and Implementations

The focus of this work is to build in FPGA an efficient OQPSK transmitter system with its power efficiency leveraged through the use of RMM and LINC amplification techniques. Figure 4.6 shows the architecture of this system, that is composed by: a pseudo-random OQPSK symbol generator, the ring-type magnitude modulation block, an upsampler and the pulse-shaping RRC filter, the LINC decomposer and a transmitter front-end that is capable of reproducing the two output branches of LINC. Ideally, a signal sink connected to some of these elements would be an advantage, since it would allow capturing intermediate values, for more accurate testings. This evaluation will be performed by connecting the outputs of the "transmission via RF" block to the acquiring equipment previously mentioned.

Hardware implementation was mainly supported by software tools; like most of the actual platforms on the market, each of the above described hardware parts have a proprietary software included to increase systems design's speed. The manufacturers of FMC30RF offer a software which is compliant with one of the available design suites from Xilinx, so software compatibility should not be a problem. For the different stages

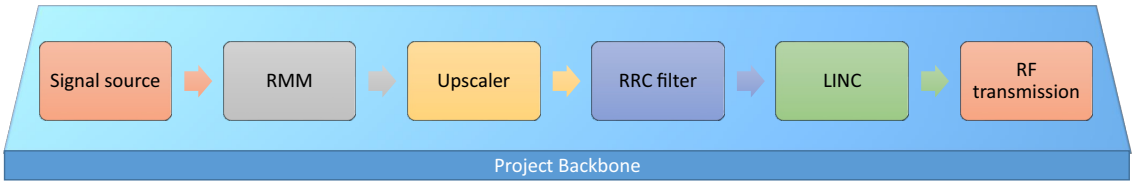


Figure 4.6: Projected data flow.

of this design, the used programming/software languages were VHDL in Xilinx ISE [43], C/C++/System C in Vivado HLS [44] and in Visual Studio [45], and the block-based programming Stellar IP [46]. Since several different programs were used during the designing process, a simple schematic is presented on Figure 4.7 to unveil a small part of all the stages that have been undertaken.

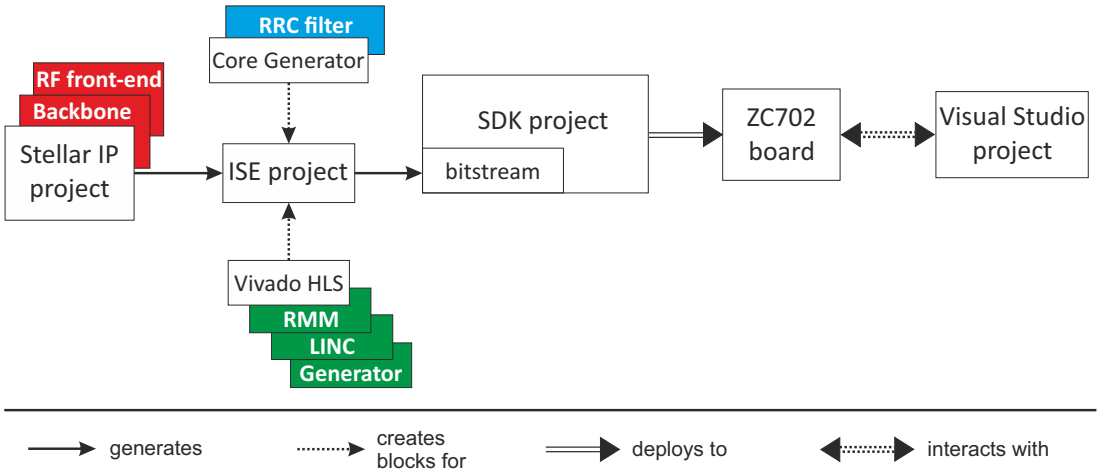


Figure 4.7: Complete software schematic of the design process.

4.2.1 Project backbone - Stellar IP

The starting point of the transmitter' software architecture is the base firmware that came with the FMC30RF boards - a Stellar IP project. This project does not really program anything; it simply creates the backbone of the complete project, and arranges the fundamental building blocks of the design, like the hardware description language (HDL) files which configure the hardware of the FMC boards or the ZC702 main board. This part can be seen as the lowest level in programming terms, since these blocks describe directly the utilized hardware structures. Stellar IP[6] is a proprietary software, produced by 4DSP, that builds a Very high speed integrated circuits Hardware Description Language (VHDL) project from their building blocks environment.

[6]Version used: 1.2.0.0.

Stellar IP

The workspace is depicted in Figure 4.8; it is very clean and simple, and it functions in a pick-and-place way: the system designer only needs to select the necessary components, drop them in the main design window from the left pane of blocks, and finally wire them. However, this simple aspect can be tricky to a newcomer: errors are not well documented, and the tool itself provides almost no feedback on how to solve them. For example, the simple act of putting an extra block and connecting it correctly on any original design will trigger an error, unless some intrinsic parameters are altered before this change.

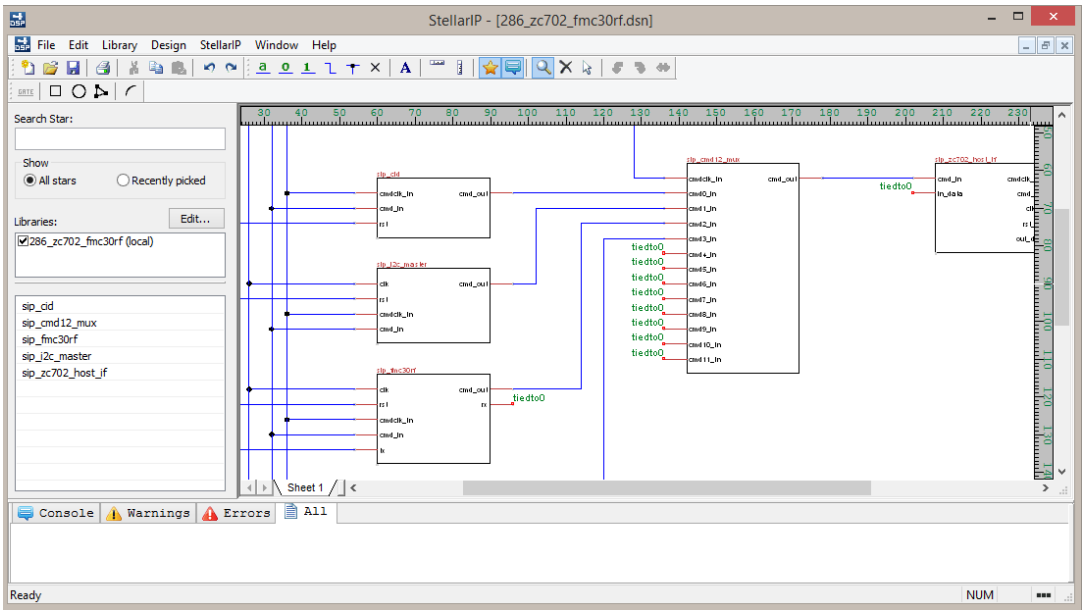


Figure 4.8: Stellar IP workspace.

The original Stellar IP project was prepared to receive the transmitting signal from the host computer to the board via Ethernet, and to send the acquired signal from the board to the host via the same protocol, which is equivalent to say that, originally, the signal source would be the host computer. However, the boards producer states that the original firmware does not support very high downlink speeds and may not be reliable in timings as the designer wishes [47], which is a major weakness if working at high clock frequencies is a necessity. Because of this situation, our design method was rethought and the employed solution was to generate a signal's source in VHDL to be implemented inside the main board, i.e., the ZC702. This would solve any possible timing problems, since it is the board itself that is generating the input, instead of an external source.

In addition, the original firmware was only designed to handle a single FMC daughter card with a single ZC702 board, with one RF reception and one RF transmission channel. But, in our case, two output branches are needed for LINC; a single FMC30RF card is

not able to emit them simultaneously, so two boards are needed. This was indeed the principal and most challenging alteration made on the original design: add a second FMC block and connect it to the rest of the diagram, working with both the transmission and reception channels - this is the design flow of a multiple input, multiple output (MIMO) system.

However, in order for this diagram to work, more than 100% of the available resources of the ZC702 would be needed, which means that the MIMO system would not be feasible. So, a workaround was employed: since the important part of a transmitter is its transmission channels, the receiving ports of both the boards were disconnected from the main board, thus deactivating for now the entire reception part on the block diagram.

Block diagram

Figure 4.9 depicts the final block diagram achieved in Stellar IP for this system. As mentioned, there were two FMC30RF parts (3) used, a ZC702 part (5), and some control parts: a command multiplexer (MUX) (4) where the control signals of all the blocks are multiplexed, in order to connect them to the ZC702; an I^2C [37] master (2) which handles some control signals of the ZC702 board itself; and a block to store all the necessary info about the design like the number of blocks or their memory indexes: the Constellation ID block (1). It is possible to observe in the FMC30RF blocks that the *rx* pin is disconnected, to accomplish the deactivation of the reception part of the design. The *rx* port handles the FMC30RF's gathered values from the ADC.

As it can be seen in Figure 4.9, the transmitting ports of the FMC30RFs are connected to an output port of the ZC702 board, which means that, by now, the main board is considered to be the signal source for our future design. This topic will be further evaluated.

When it is completed, the design can then be generated and a VHDL project is built, to be used in Xilinx ISE software [43] [7].

New structures placing

Once the fundamentals of the transmitter are concluded, the software blocks (RMM, RRC and LINC) can then be developed. The obvious place where these structures should be inserted is between the signal source and the transmitting port of the FMC30RF. But first, more important questions arise: these implementations should be described in VHDL, in order to fit in the existing backbone, but building such complex designs

[7] Although Xilinx has already a newer and more complete suite of design tools, i.e., Vivado Design Suite [48], Stellar IP does not support Vivado project generation for any FMC30RF design, neither the company plans to support it in the near future whatsoever [49].

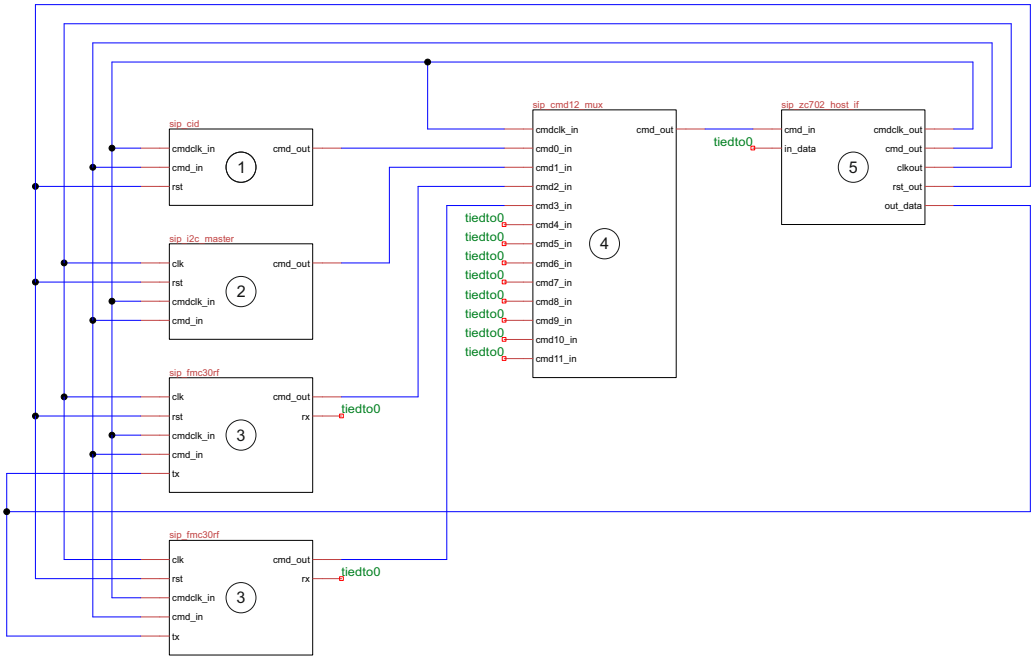


Figure 4.9: Generated Stellar IP design.

in VHDL by hand would be very problematic, even if the designer has solid skills in hardware description languages.

Fortunately, there are some tools that automatically construct VHDL blocks from a set of existing designs from where the user can choose, like Xilinx Core Generator [50], that are very optimized and are widely used (like filters, FFT blocks or FIFOs). In case the designer wants something very specific, and because of that it is not available in these predefined libraries, there are some other tools that can build VHDL structures from code in other languages, such as C or C++; these are the high level synthesis (HLS) tools, and in this case, a wide range of software is available, such as Xilinx Vivado HLS [44], Bluespec [51] or PandA [52].

In our transmitter's case, both these needs apply. Since an RRC filter in our design is needed, a tool, as the first type mentioned, is used, which is the Core Generator [50]. Also, the offered set of designs from Core Generator do not hold blocks that perform neither LINC decomposition nor RMM, so for these implementations, a second-like tool will be used: Vivado HLS [44]. Both these tools are developed by Xilinx, this being the main reason of their choice, to avoid having integration problems.

4.2.2 RRC filter - Core Generator

The pulse-shaping filter, in this case an RRC, is one of the key elements of any RF transceiver, since it limits the signal bandwidth and adapts it to the characteristics of

the communications channel, in the digital domain. Filters are common components of communication systems (and signal processing systems in general), and thus they are already built and optimized to be synthesized in hardware. Indeed, Core Generator [50] has a FIR filter implementation in its library, so it is only necessary to personalize it according to our needs.

In our architecture, an OQPSK-modulated complex signal is provided at the entrance of this block, and so the filter has to employ two single-rate channels (one for the in-phase component and another for the quadrature). In Sub-chapter 3.2.1, an explanation was given on the relation that should exist between the number of taps to implement on the filter and the quantity of affected symbols from the RMM: the RMM LUT tables size should follow the number of the filter's N_{sym}, which in our case is equal to 7. However, Simões [1] used an RMM table with just 5 affected symbols, since a table with 7 affected symbols would have a size of approximately 10^9 entries (refer to Sub-chapter 5.2.2) and is computationally extremely heavy to create. This shrink of the RMM tables was successful, because there was not any apparent loss of performance in his system.

In our case, and since the tables are being implemented in an FPGA with limited resources, even the use of RMM LUT tables with 5 affected symbols was not feasible, since it occupied more than 100% of the ZC702 LUT's resources; this problem motivated a new reduction in the size of the RMM tables, using just 3 symbols, instead of 5 used by Simões [1]. This increased gap between the number of affected symbols by the RRC filter and the RMM LUTs motivated some testing, to understand if an RRC filter with the factor 7 and the RMM implementation with the factor 3 would work well together, or if instead a filter with a lower factor, i.e. 5 affected symbols, would perform better, due to the smaller difference between factors. The resulting comparisons show that the smaller RMM table implementation was successful and that both filters perform similarly; the results are presented on Sub-chapter 5.2.2, whose filters' details are shown in Annex B.

Filter implementation

Therefore, regarding the filter implementation, Simões [1] approach will be followed: the implemented filter will have 113 taps, generated in MATLAB [17] with a roll-off factor of 25%. The FIR block also offers the possibility of upsampling the input sequence by a constant factor, which would be desirable; however, this upsampling does not support OQPSK, since it is not possible to specify a time shift between the implemented single-rate channels, i.e., the in-phase and quadrature components. Because of that, the upsampling is performed outside the RRC filter block.

Finally, at the input, a set of two fixed-point signed values are expected (one for I and

one for Q) with a format of Q4.18[8], and all the calculations inside the block are done in fixed-point arithmetics: the coefficients are stored in a fixed-point format of Q0.16 in a complements of 2 arithmetic. At the output, signed Q7.9 values are expected at each channel, and an output rounding mode of truncating LSB is performed.

4.2.3 LINC and RMM - Vivado HLS

Due to their specificity and novelty (as for RMM), the core blocks of our transmitter, namely LINC decomposition and RMM, are not implemented in the Core Generator libraries; the same happen to the OQPSK signal source and the upsampler, which need to be implemented to surpass the original system limitations. A solution that is followed is to create them in Vivado HLS v2015.2, which is a C-to-HDL synthesizer [44]. Using this type of tools, much time can be spared in system designing over direct VHDL writing, because whether the design is big, complex or hard to describe in hardware language, the software will automatize all the programming sequence and build an accurate machine model, discarding the user of all this work.

Vivado HLS

Vivado HLS workspace is shown in Figure 4.10. On the left pane, there is a file explorer of the open project, and in the middle is the language editor.

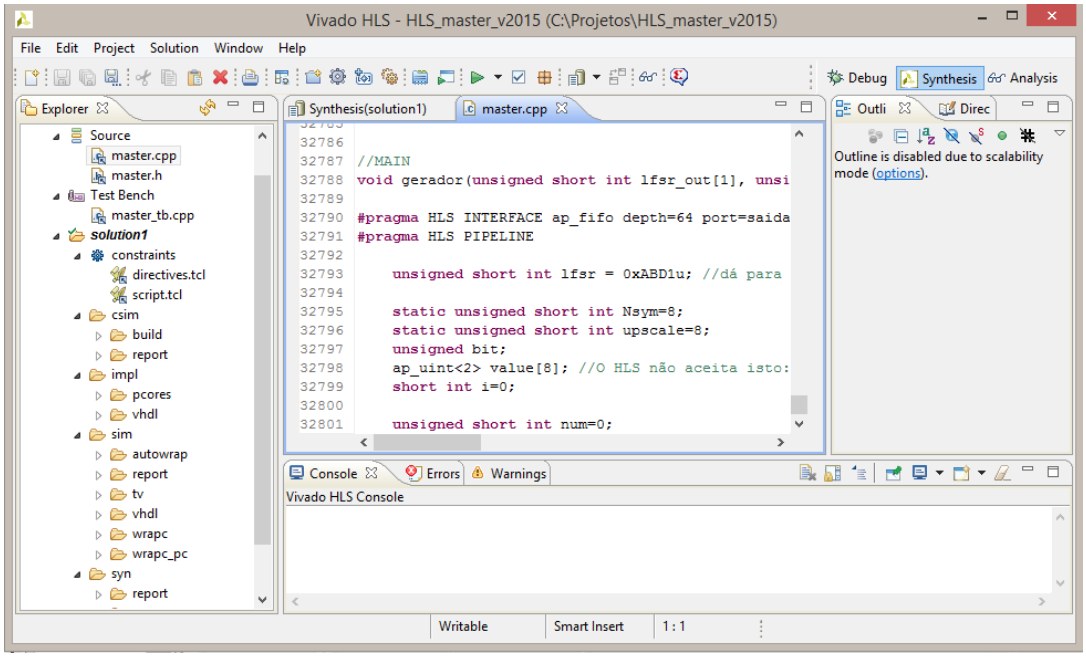


Figure 4.10: Vivado HLS workspace.

[8]Notation: Q $Bits_{Integer}.Bits_{Fractional}$.

On top (refer to Figure 4.11 to better understand this explanation), it is possible to synthesize the code (2), to run a C simulation (1), to co-simulate C and RTL (3) for a more precise report on the behavior of the design, and to export the design (4), so it can be used on ISE [43], for example. All these actions, except the last one mentioned, produce a block and timing report in terms of clock cycles, function of the board operating frequency/period, chosen in the project properties. These reports detail the main resources and the minimum amount of time that the block needs to function properly, and so they are very important to the designer since it is possible to preview if one block will fit the main design constraints or not.

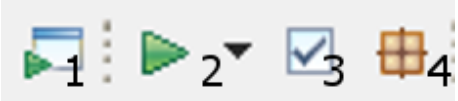


Figure 4.11: Vivado HLS main action buttons.

Implemented blocks in Vivado HLS

In this thesis, three different blocks were built in Vivado HLS [44]. The first two refer to the LINC decomposer, comparing two different approaches in terms of speed and calculations accuracy in the implementation of the vectorial decomposition equations (see Section 2.1.2). The first block performs it by calculating all the stages, while the second uses a LUT where the calculation of the error vector [9] is performed with a memory block.

The third block holds the RMM. However, since it was also necessary to create a signal source and to upsample the RMM output in order to prepare it for filtering, all these functions were combined in just one block. So, this major block will internally generate an OQPSK signal, perform RMM on it and then upsample the resulting signal, making it ready to pass through the RRC. Due to the quantity of different operations that this block has to perform, it will just be called ”Generator”. A block diagram of this system is depicted in Figure 4.12.

Generator block implementation

The Generator uses a fast algorithm (based on the fast C algorithms from [53], which will be discussed in detail in Chapter 5) in C to create *random*[10] numbers of two bits between 0 and 3, since the employed modulation is OQPSK, i.e., one bit for the in-phase part and other for the quadrature part. Next, the generated symbols enter sequentially a

[9] Refer to Eq. 2.13.

[10] Almost none of the random number generator (RNG) algorithms can generate ”true” randomness, unless they use external random variables.

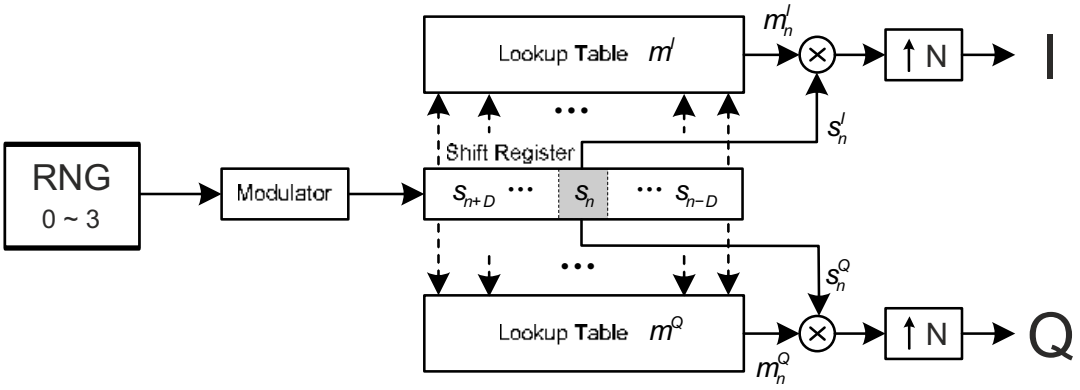


Figure 4.12: Generator block diagram.

zero-initialized register with size of 14 bits (i.e., 7 symbols) [11] and this register is used as an integer to access the RMM-LUTs, in order to find the correct coefficients to apply to the middle symbol of the register. All the RMM coefficients are signed and were stored in a fixed-point arithmetics of type Q2.18[12]; the intermediate calculations are also made in this arithmetic and with a wrapped saturation.

Next, the resulting values are multiplied by a factor of $\sqrt{N}$, in order to maintain the signal's average power after interpolation, and then upsampled by this N factor, in which N is coherent with the number of affected symbols of the RRC filter. In this case, since the upsampling is fixed for all the implementations, it would not make sense to perform a real-time computation of $\sqrt{N}$ because it would consume many resources. Thus, this value was calculated outside the HLS environment and then included as a constant in the code of type Q3.18, to be multiplied by the samples. Finally, floating point values of type Q4.18 are released from this block. As stated in Simões [1] work, N must be chosen to be at least equal to 8 to fit in the oversampling requirements of digital LINC implementation. For our case, the upsampling factor was chosen to be precisely $N = 8$.

In this block, a small detail was also implemented, in order to allow the final user to have more control on what is happening inside the block: a binary input was created to turn on and off the RMM code snippet, i.e., to activate and deactivate the RMM appliance on the signal. This topic will be discussed shortly.

LINC decomposition block implementation - 1st design

Regarding the LINC decomposition block, it takes 2 complex symbols at a time, i.e., four Q16.0 values (one in-phase and one quadrature sample for each complex symbol of

[11] Refer to Sub-section 3.2.2.

[12] Notation: Q $Bits_{Integer}.Bits_{Fractional}$.

the digital input signal at the arbitrary time sample n: $s[n]$ and $s[n+1]$) for each cycle and apply LINC, whose decomposition of one complex value results in two complex samples (one for each branch). It also receives as input the maximum admissible value for LINC, which is an unsigned 32-bit integer. Although the outputs of the RRC block are Q7.9 fixed-point values, they are being implicitly casted as 16-bit integers, which in our case is equivalent to multiply the samples by 2^9. This happens because the output of this block needs to be a 16-bit integer, to connect correctly with the DACs, and since an initial scale is being done.

This block was built this way because the original design structured the arrival of new samples to the DAC in sets of 2 adjacent values, i.e., for each clock cycle, the in-phase and quadrature components of the signal on an arbitrary time n would arrive, followed by their adjacent in-phase and quadrature components of the signal on the time $n+1$. In order to simplify the introduction of new blocks to the original design, it was chosen to use the existing structures, like the signal FIFOs that deliver the signal this way, instead of altering them, since they are compatible with the DACs, which need to receive values in this configuration.

The decomposition made on the LINC calculator implementation is represented by a block diagram on Figure 4.13, which includes the numeric representations of their intern stages. Here, the I and Q samples of the same time slot are acquired, and the computation of the quadratic modulus is performed; if this value surpasses the maximum, then it is reduced to the threshold. Finally, the square root is applied and the output samples are computed. Note that the quadratic modulus is stored inside a C++ double-type variable and the result of the square-root is stored inside a C++ float variable; it allows for a flexible computation and storage of the obtained values, although it may need more resources than a fixed-point architecture. In this case, an upper and lower saturated arithmetic was used.

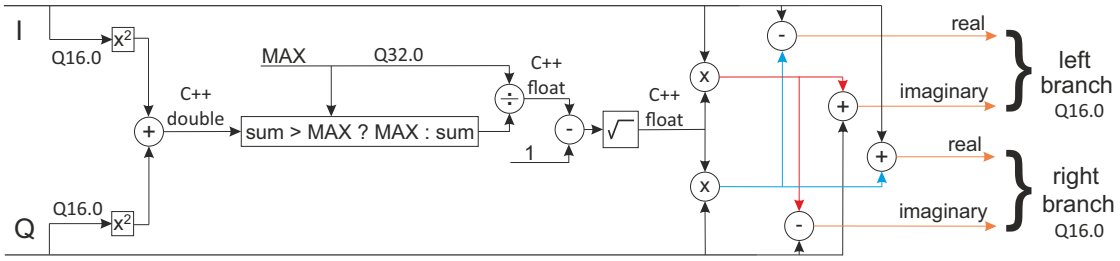


Figure 4.13: LINC calculator block diagram.

LINC decomposition block implementation - 2nd design

The memory block implementation performs a similar set of operations, but instead of having the quadratic modulus assertion with MAX and the square-root calculator, it has a

4096-entries LUT. The index for gathering a value from the LUT is achieved by applying a mask to the quadratic modulus, in order to obtain a 12-bit value. The chosen 12 bits are the 22th to the 11th. This may be an odd choice; however, some analysis on the output values of the RRC, i.e., the input values of the LINC LUT block show that these values never surpass a threshold of approximately 700 units, so a tailored solution was made to cover the most significant bits of this maximum value. This LUT outputs Q6.14[13] fixed-point values, which hold a reasonable precision for the necessary calculations.

Contrarily to the calculator implementation, the LUT LINC block was designed with very tight bit representation, i.e., the numeric formats were closely drawn to minimize bit wasting. Figure 4.14 displays the implemented design.

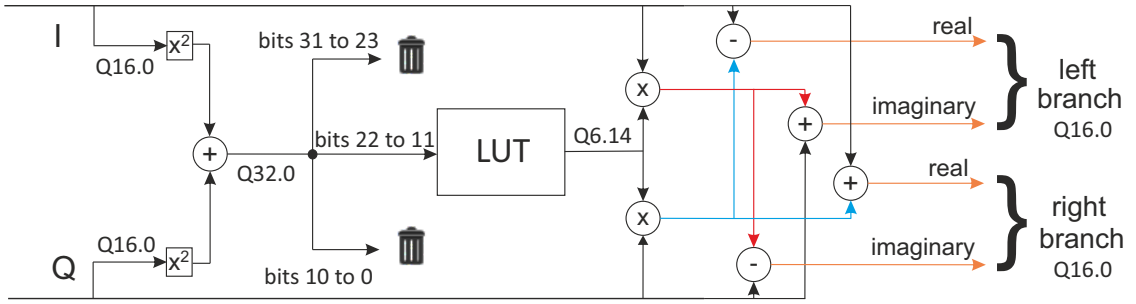


Figure 4.14: LUT LINC block diagram.

Although the structure of this implementation can make it faster than the homologous calculator (due to the substitution of the square-root block by a LUT), it may have less precision [14]. In Annex C, further information is presented pertaining to this LUT.

Vivado HLS code directives

The C++ code of all the blocks designed in Vivado HLS [44] is presented in Annex D. Note that this is a FPGA implementation, whose parallelization is a great advantage. All these blocks were designed to work in pipeline, speeding up the calculation processes. This can be done with easiness just by applying this line of code in the beginning of the C++ code:

```
#pragma HLS PIPELINE
```

With this, Vivado HLS [44] will try to parallelize at most the design, allowing for faster processing rates in these blocks. This directive, along with other Vivado HLS details, such as saturation and arithmetics architecture, were consulted in [54].

With all the blocks finished, it is now necessary to insert them in the backbone design. All the VHDL blocks created in Vivado HLS have extra control ports, apart from the

[13] Notation: Q $Bits_{Integer}$.$Bits_{Fractional}$.

[14] This observation will be further detailed on Chapter 5.

inputs and outputs that are generated from code translating. These extra control ports will have a very important role on fitting the corresponding blocks in the complete design, since they can start the activity, acknowledge the stop and indicate the readiness of the blocks.

4.2.4 Complete project - Xilinx ISE

All the building blocks individually implemented were, in a second phase, connected in the backbone project. ISE 14.7 is the software that can handle the project as a whole, synthesize it and generating a bitstream, which is the core piece of the implementation in ZC702. Its workspace is presented in Figure 4.15. On the upper left pane, the HDL hierarchy is presented, and in the middle left there are the possible activities, like synthesis and FPGA mapping. The right pane holds the text editor, where the HDL files are presented.

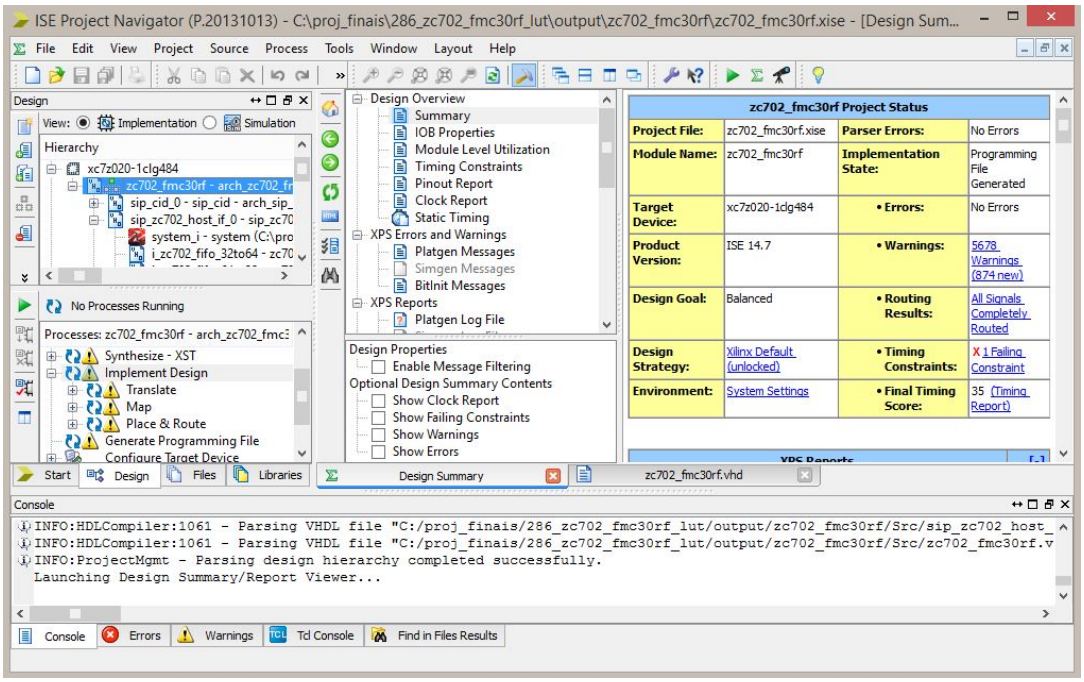


Figure 4.15: Xilinx ISE workspace.

Control signals and timing requirements

When designing, in any HDL, it is necessary to take into account all the control signals and timings; these are not only useful to the system designer, but also needed to implement a correct work dynamics between the code snippets. In our design, the first and second blocks (the Generator and the RRC, respectively) are both external blocks that have their own control signals, and whose data connection between them have different

rates. Bearing this in mind, a Counter was implemented to synchronize the Generator and the RRC filter, due to the disparity of their input and output latencies[15], since the first block outputs 32 values and the second only receives two *per* cycle. The Counter was made to certify that a block only starts working when there is available information from the back; no block will start with erroneous information, and with this, the integrity of the whole system is ensured.

The Counter is clocked at $100MHz$, like all the other blocks, and activates an increment register for each cycle (starting at 0) when an input enable is activated; this effect triggers an enable signal to the output. When the counter reaches 31, the output enabled signal is deactivated and the counter resets. This block was implemented because the structure that sends the enabling trigger has 32 data values to dispatch for each finished operation, and the receiving structure only accepts two values *per* cycle. A graphic implementation of the Counter is shown in Figure 4.16.

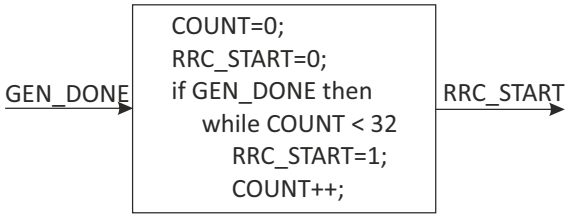


Figure 4.16: Graphic implementation of the Counter block.

User-controllable options

Apart from this, another two fractions of code were designed directly in VHDL. One was built to make it possible to activate or deactivate RMM and it is depicted in Figure 4.17, where the block is represented by a pseudo-code of the implementation. This function is controlled by a ZC702's pushbutton and the RMM initial state is 0 (deactivated). The assigned button is the ZC702's user pushbutton SW5 (left), and is represented in Figure 4.19 with the number 1.

The other code snippet (depicted in Figure 4.18) works as a channel selector and connects the input port of the output data FIFO (which originally stores values that are then sent to the host computer) to one of the following outputs: the out port of the RRC or the output of the LINC left arm. It is also controlled by a button, the ZC702's user pushbutton SW7 (right), which is represented in Figure 4.19 with the number 2.

It was said before than both the receiving ports that connects the input of the FMC30RF to the ZC702 (and from there to the host computer, via Ethernet) were disabled in Stellar

[15]One block's output will be connected to other's input; this implementation serves to clock-enable the receiving block while there is any available output from the other.

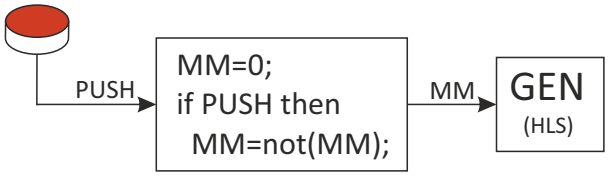


Figure 4.17: Graphic implementation and pseudo-code for RMM activation.

IP, due to the over occupation of the resources. However, the limit occupation is only exceeded if the two receiving channels are enabled. Using just one receiving path will not occupy all the available resources, and so, a shunt was made in one of the paths, in order to be used to give feedback of the data operations inside the main board, like the output of the RRC filter or LINC. This code snippet is the part that handles whether this feedback channel connects, to send information to the host computer.

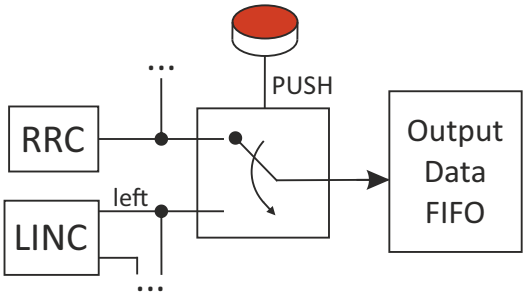


Figure 4.18: Graphic implementation of the output channel selector.

However, due to the fast PL clock speed, one can not precisely know which is the current mode when a pushbutton is pressed, and so, some of the ZC702's available LEDs were used to acknowledge the state change (highlighted in Figure 4.19): the two on top turn on when RMM is activated and turn off when the opposite happens; the second pair activates when the output comes from the RRC and deactivates when the output is the LINC left branch. The other LEDs simply mimic the pushbuttons, and do not maintain their state like the above mentioned LEDs.

Blocks pairing

Finally, a word on the overall implementation. As stated before, all the HLS-based blocks have direct input and output connections to start, acknowledge the end and the readiness of a new operation. These signals are extremely important to put the system correctly working, and then some care was taken on how to connect them all. The used approach was to include built-in FIFOs on the output of all the blocks, except the LINC block (which is the last of the chain), and also to include input FIFOs before the input

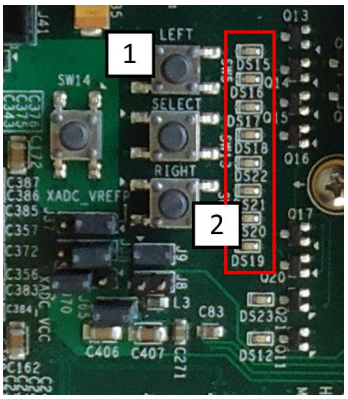


Figure 4.19: ZC702's assigned buttons for the above mentioned code blocks.

ports of every block (except the first block - the Generator), in order to nullify any bit rate differences between the blocks and to guarantee that all the blocks will have available information before they start operating.

A simple block-diagram of the implemented connections between all the created blocks is shown in Figure 4.20.

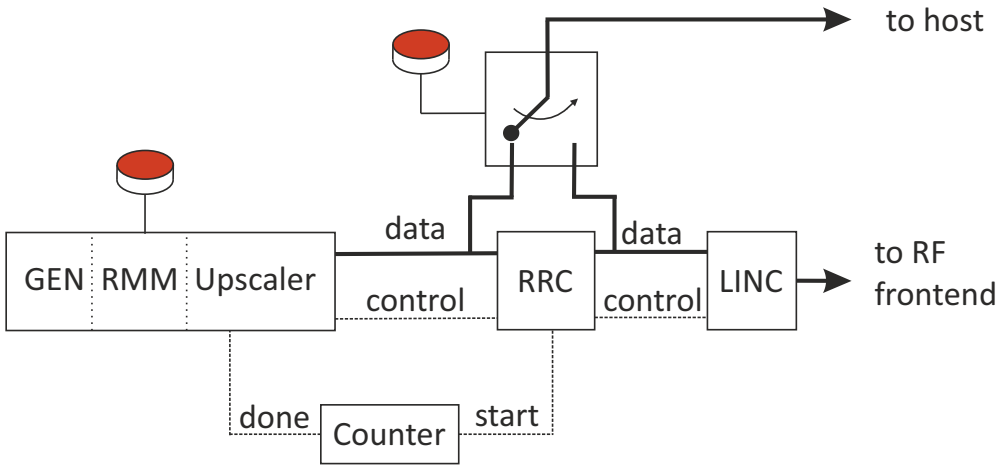


Figure 4.20: Block-diagram of the implemented parts of the project.

The complete design includes not only the Stellar IP backbone project (synthesized to VHDL), but also this Counter and the generated blocks in Vivado HLS and Core Generator[16]. The code for these small snippets is available in Annex E. The complete design can then be synthesized, with all the components mapped in the FPGA, and when these processes are complete, ISE generates a bitstream to program the board; the programming process is then delivered to Xilinx SDK.

[16]Due to the size of the schematic of the design, it will be presented in Annex A.

4.2.5 Board programming - Xilinx SDK

When the synthesis and mapping processes are complete for the PL resources, a bitstream is created to program the ZC702 board's logic. This bitstream is then sent to the SDK to be deployed. This software (presented in Figure 4.21) manages all the board programming, both for PL and for PS, although its main focus is the latter. Here, the coding task is made in C or C++, and the developing environment is based on Eclipse [55].

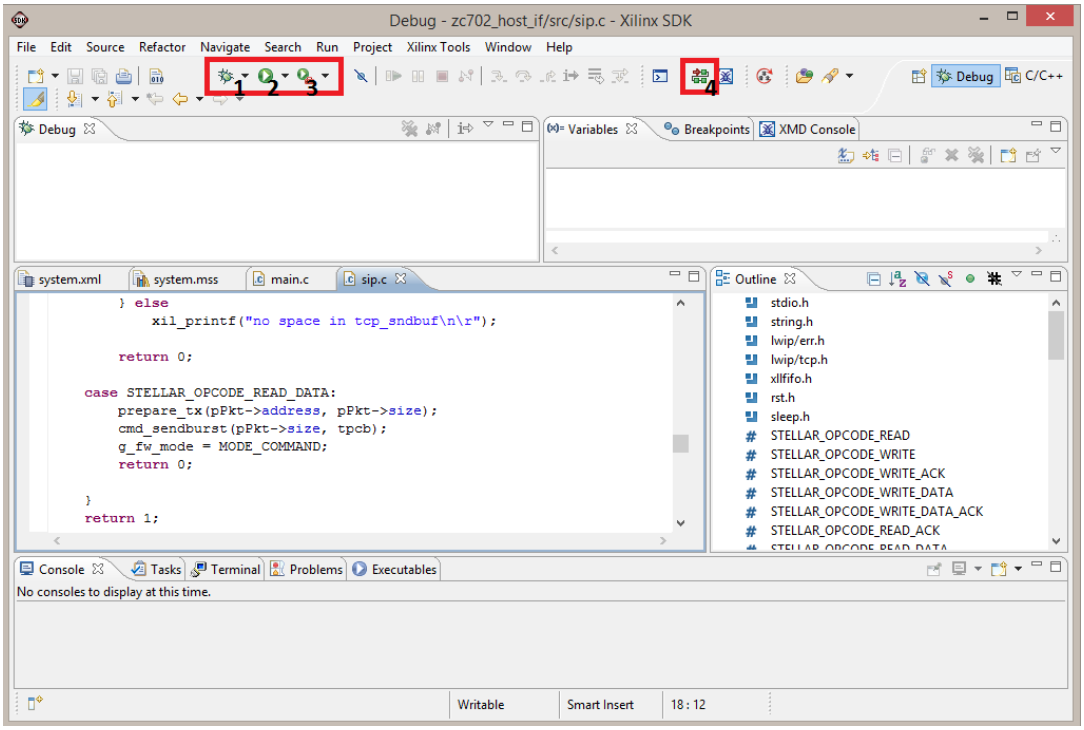


Figure 4.21: Xilinx SDK environment.

In order to program the board correctly, it is necessary to plug in two USB cables from the host computer to the board, one being the USB-to-UART bridge and the other the JTAG programming connector. It is necessary to first deploy the bitstream (4) and only then to program the ARM processor (that can be done in Debug mode (1) or in normal Run mode (2). There is also last build mode to recall the last programming mode (3)). Note that the SDK only programs the board, and the application that runs on the ZC702 board behaves as a *slave* to the host computer.

Initially, our system was designed to perform all the operations mentioned above in the logic part, except the LINC decomposition, that would be processed in the ARM processor, and then returned again to the logic part, to be transmitted. The LINC decomposition was initially implemented in the processor part in C/C++, but eventually this idea was dropped in detriment of a full PL implementation. Since the programming language that was used to compute the LINC decomposition in Vivado HLS was C++, the code that was

previously written for the processor was just used as the base for the Vivado HLS project, with some performance directives added, to be synthesized in VHDL.

At this point, the given software was just altered to constantly refresh the DAC waveform memory (WFM)[17]. Originally, the design was built in a way that the host computer would send the values to the board, and it would need to send a refreshing command for each time new data was sent; while no new data comes to the board, the old data keeps being repeatedly transmitted, which would not emulate a true telecommunications channel. Also, 4DSP support stated that it would not be possible to send large amounts of data in small time intervals [47] (i.e., no real-time transmission of data from the host to the board). For these reasons, and due to the main objectives of this thesis, the signal was generated inside the board (to dispense an overloaded transmission of control signals and data from the host) and the WFM *discrete* state was unlocked to constantly refresh the stored data of 16k values (taking into account the data generation rate - the WFM only cleans the samples when they are sent to the DACs).

4.2.6 Board communication - Microsoft Visual Studio

This is the end of the programming line on the host computer. Microsoft Visual Studio[18] runs an application (also given in Stellar IP software package) in the host computer and works as the *master* of the board. After the board is programmed, it is necessary to connect the host computer and the ZC702 by an Ethernet cable, and define a static IP for both the host and the board; only then will the host program work. The host software allows the user to personalize and setup the board and the daughter cards widely at register level, and to send control, data and probing commands.

Here, the FMC30RF daughter cards are powered up through registers, and all the RF boards' frequencies are set, as the PLL frequency (set to $491.52MHz$) and the DACs operating frequency (set to $122.88MHz$, equal to $freq_{PLL}/4$).

Although the operation states of the components of the FMC30RFs are accessible and tunable through registers, they do not all seem to work properly when changing the registers freely[19], more precisely the frequency registers. In fact, we wanted to be able to change DAC operating frequency, e.g. for loosing design time constraints in low bit-rate transmission. Some tests were done in this regard according to the documentation [56] in order to change the provided design by 4DSP. However, we arrived to the conclusion that the claimed flexibility didn't exist, and a single sub-multiple of the default 122.88MHz

[17] Refer to Figure 4.4.

[18] Version: Visual Studio Express 2012 for Windows.

[19] Freely is stated here as the change of registers inside a range of possible values, which are supposed to be supported, as described in the data sheets [56] of the used components of the FMC30RF.

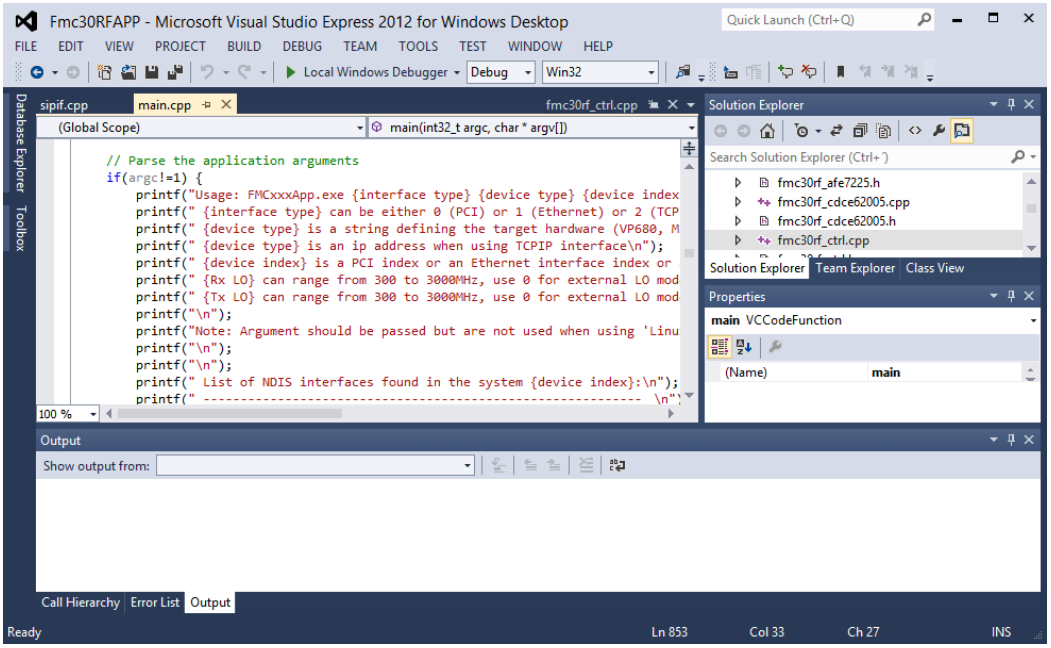


Figure 4.22: Microsoft Visual Studio environment.

was not supported.

Since the WFM was unlocked to constantly gather new data, the host only needs to startup the boards and to activate DAC transmission, simplifying the overall function calls and minimizing packets communication via Ethernet.

4.2.7 External acquirements - GNU-Radio

The system outputs can be measured in varied forms, such as internal board acquirements, where samples are gathered before they being sent to the DACs and directed to the host computer to be analyzed via MATLAB [17], or external signal acquirements, such as the HP spectrum analyzer or GNU-Radio software [42], as stated in Sub-chapter 4.1.3. Although the first cited process is considered to have more precision, GNU-Radio is also good to use, since it offers wide signal analysis methods, like time, spectrum and constellation views.

GNU-Radio and USRP B210 were used to gather the output signals and view them. For that, a software project was made, in block programming, to set correctly the acquiring parameters. Figure 4.23 shows the created block project:

The RF transmission frequency used to acquire the output signal was $1.2GHz$, with a sampling rate of $60M$ samples/s. An input gain of 10 dB is applied in order to see in more detail the gathered data.

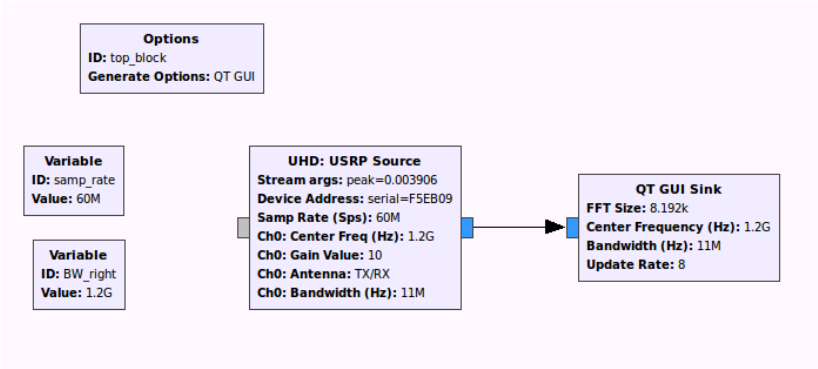


Figure 4.23: GNU-Radio block project.

5

Implementation and Simulation

Some background on the workspace was given in the previous chapter. Now, implementation, simulation details and obtained results concerning hardware, timings and output signals' precision are explored.

5.1 ZC702 total resources' capacity

ZC702 board can be decomposed in its base resources (such as look-up tables, flip-flops, RAM blocks and DSPs), which are the considered elements whose synthesis occupation is compared to in programs like Vivado HLS [44]. Table 5.1 depicts the availability of those main resources on the considered board:

Table 5.1: Available resources of ZC702's programmable logic (XC7Z020)

Resources	units
LUTs	53 200
FFs	106 400
Block RAMs	140
DSP blocks	220

The considered RAM blocks are 36 kB wide and build a total of 560 kB of space; a RAM block can be divided into two 18 kB blocks if needed (this process is automatically handled by Vivado HLS in any C-to-HDL implementation). Also, each DSPs have a 18*25 MAC calculations capacity and its peak performance was calculated to be of 276 GMAC calculations for a symmetric FIR filter [57] (max performance of 1.25 GMACs per DSP block).

These values will serve as a reference for the HLS occupation results and for the final ISE project.

5.2 Blocks' specifications and physical occupation

Three implementations were made in Vivado HLS; two of them are relative to LINC decomposition and the third one is the Generator block (that implements the bit generator with RMM and upscaling), being all them described previously in section 4.2.3. The ZC702 occupancy and some code specifications for each of these blocks will now be discussed.

5.2.1 Generator with RMM and upscaling

This block consists in an approximately uniform bit generator, followed by a RMM block (which can be turned on or off via an input bit) and an upscaler by the factor of 8, which also includes sample multiplication by $\sqrt{8}$.

The bit generator was first designed with a LFSR algorithm, but it was leading to very awkward results in conjunction with the RMM part[1]. Because of this, it was discarded and a new algorithm was thought of for this system, adapted from an example given in [53] for pseudo-random number generation with uniform distribution. It works on a multiply-and-sum base, where a random initial seed is recalculated by such a type of operation in every iteration, as follows:

$$seed_{i+1} = \Big(seed_i * m + s\Big) \% T \tag{5.1}$$

where i represents the current iteration, m is the multiplier constant, s is the sum constant, and T is the maximum value allowed for the seed. $\%$ represents the remainder operation. The random number is then obtained by the operation:

$$rand = low + \frac{\Big((high - low + 1) * seed_i\Big)}{T} \tag{5.2}$$

where low represents the lower bound for the RNG, and $high$ represents the higher bound.

Since the *seed* is always calculated to be a number between 0 and $T-1$, then the division operation will always return a value between 0 and $(high - low)$, which implies that *rand* will take a value between low and $high$.

Please note that this random generation process will not create "true" randomness, because the generation process is fully predictable, i.e., not based on a true random variable. However, for the sake of this implementation, such a process that generates a uniform distribution of numbers will serve the purpose.

In order to be uniform, the constant values must be chosen wisely; they should not be small, and some of them must be relatively primes (more precisely, m should be the closest prime of $\left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right) * T$) [53].

In order to create randomness as long as possible, a set of numbers were chosen to overflow only at 2^{32}, which is the size of a C-type integer [2]. The implemented values are $T = 714025$, $m = 4096$ and $s = 150889$. The initial seed was chosen to be $seed_0 = 357$,

[1] The LFSR algorithm can uniformly generate all symbols from a $n-bit$ vector, except one: the all-zeros vector. In the current case - generation of all of the four OQPSK possible symbols -, this would lead to a very non-uniform system, and for that it was abandoned.

[2] C refers to the programming language C.

and since an OQPSK constellation is being used, with two bits *per* symbol, values are generated within a range that goes from 0 to 3.

An histogram for 64000 samples with the initial seed presented above was calculated in MATLAB [17] and is shown in Figure 5.1.

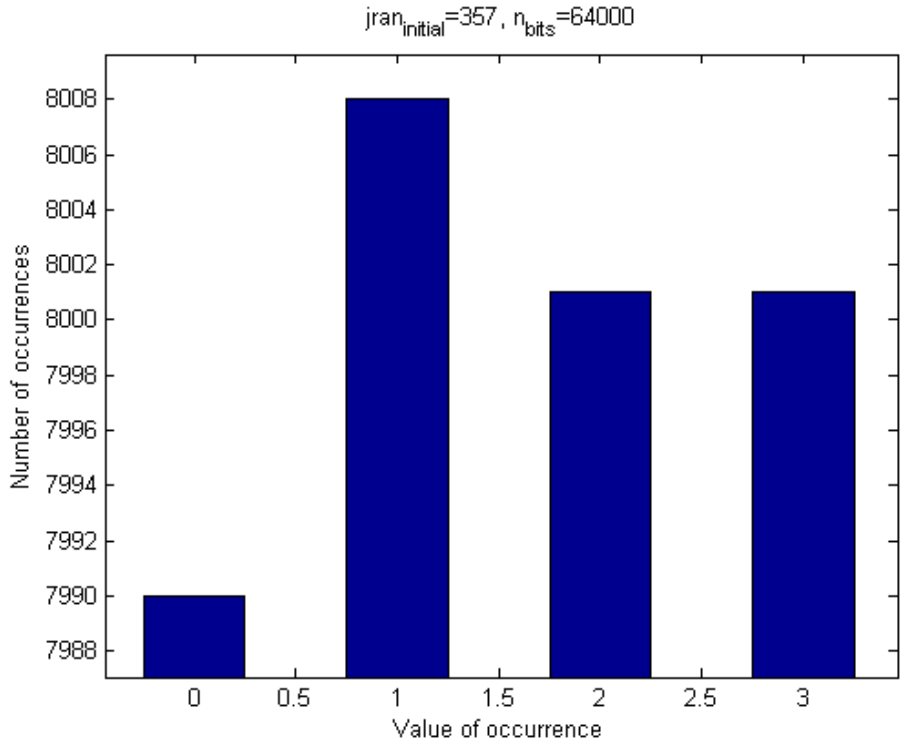


Figure 5.1: Histogram of the random number generator.

The new symbols (constituted by the Q bit, followed by the I bit[3]) are then modulated according to QPSK, ensuring an amplitude value of 1 for each symbol (i.e., $1 \angle\theta$). Due to the implemented algorithms, a 1 bit will lead to $-\frac{1}{\sqrt{2}}$, and a 0 bit will lead to $+\frac{1}{\sqrt{2}}$. Next, the symbols enter sequentially the RMM shift-register and, in this stage, the design is done in a way that RMM can be chosen to be turned on or off, defined by an input single-bit port; if this port is set high, then RMM is applied.

The RMM is applied as stated in Sub-chapter 3.1: for each new symbol, the buffer is implicitly casted to a 14-bit unsigned integer. This value is then used to search in RMM tables for the corresponding multiplication factors for both in-phase and quadrature values of the QPSK-modulated symbol.

Finally, the upsampling is done, and the offset from OQPSK is also implemented. The upsampling factor L is then used, and the output will follow Eq. 5.3. In this case, that $L = 8$, s^Q will be delayed 4 samples relatively to the corresponding s^I value.

[3]The integrity of the system depends on it; refer to Sub-chapter 3.2.3.

$$s[n] = s^I[n] + s^Q[n - k\frac{L}{2}] \tag{5.3}$$

In order to continue the sequence generation correctly, the generator block has input and output ports specially reserved to transport the last states of the RMM shift-register and the last calculated seed. This is a security feature that assures the continuity of this block, in case of stopping.

5.2.1.A Timing and resources

The used resources of this block are presented in Table 5.2. In subtable 5.2a, the maximum time lapse of the block is presented, relatively to the main board clock ($f_{clock} = 100MHz$, which is equivalent to $T_{clock} = 10ns$). Also, the total latency (number of cycles needed for the block to an input be reflected on the output) and the input latency (number of cycles needed for the block to accept new inputs[4]) are shown.

Table 5.2: Generator spent resources and timing

(a)

Timing (ns)	
Estimated	Required
8.47	10.00
Latency (cycles)	
Total	Input
144	32

(b)

Resources	units	% of Total
LUTs	8 601	16.17
FFs	8 570	8.05
Block RAMs	20	14.29
DSP blocks	40	18.18

By observing this data, it is possible to see that the timings are met and the resources do not occupy much space (less than 20 % in the worst case - DSP blocks).

5.2.1.B Bit rate

Regarding the Generator bit rate, the RNG algorithm produces four dibit symbols for each 144 cycles , in order to generate 32 samples at the output (including the upsampling by 8). According to the FPGA clock frequency of $100MHz$ (or mega clock cycles per second) that feeds this block, the 2-bit symbol generation can be translated to a symbolic bit rate of approximately $5.56Mbit/s$ for the RNG algorithm. The full block takes 144 cycles to produce 32 samples (16 samples for I and 16 samples for Q) of 22 bits each, which gives a total bit rate of $488.89Mbit/s$.

[4] If this number is smaller than the total, it indicates that the block is working in pipelining.

However, due to the implemented pipelining, during its activity the Generator produces 32 new outputs every 32 cycles, making an effective throughput of 1 output per cycle. This leads to a total effective bit rate of $2.2 Gbit/s$.

5.2.2 Root-Raised Cosine Filter

The RRC filter was generated in Core Generator [50], and consists in a FIR filter with the MATLAB [17] generated taps for a RRC. As stated earlier in Sub-chapter 3.2.1, the number of affected symbols of the RMM LUT table D should be equal to the number of symbols affected by the RRC N_{sym}; however, in our case, the number that are being used are slightly different, due to implementation problems: D is being used equal to 3 and N_{sym} is being used equal to 7, which could lead to a matching problem. However, Simões [1] used a different ratio on these values ($D = 5$ and $N_{sym} = 7$), without any apparent loss of information. This motivated a comparison, in order to understand if a larger gap between these values would affect noticeably or not the obtained results, so two filters' performances were tested and compared: one with $N_{sym} = 5$ and other with $N_{sym} = 7$. This Sub-chapter aims to compare both situations in results and hardware occupation's terms, unveiling strong points and major drawbacks.

The taps of both RRC filters were generated using MATLAB [17] and can be consulted in Annex B. Case #1 will be described with the parameter $N_{sym} = 5$ and case #2 will be described with the parameter $N_{sym} = 7$. The values of L and $rolloff$ are 8 and 25% respectively, as stated in Sub-chapter 3.2.2.

Both filters' behaviors were simulated in MATLAB [17] with the same uniformly random binary sequence, similar to the sequence implemented in practice, where OQPSK modulation and RMM were applied. The result of each filtering is compared in Figure 5.2:

According to the Figure, the filters present small differences for the same sequence, but these changes do not seem to be significant. Therefore, it can be concluded that in terms of results, both filters can be used.

5.2.2.A Timing and resources

Relative to hardware occupation, both filters were synthesized in Core Generator [50]. The obtained results are displayed in Table 5.3:

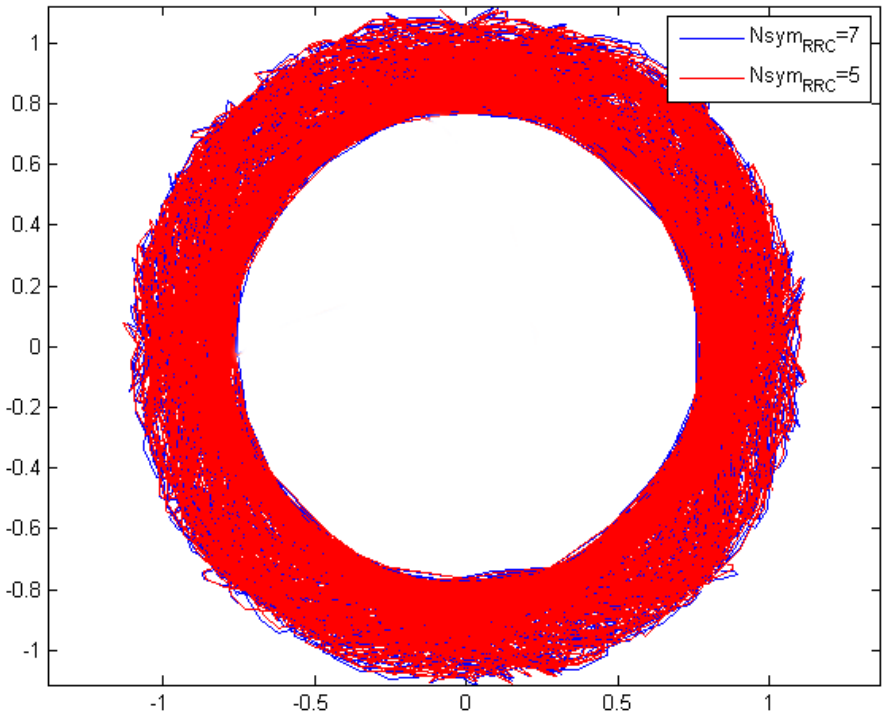


Figure 5.2: Comparison of filters #1 and #2 for the same pseudo-random sequence.

Table 5.3: RRCs' spent resources and timing

(a)

	Timing (ns)	
	Estimated	Required
#1	N.A.	10.00
#2	N.A.	10.00
	Latency (cycles)	
	Total	Input
#1	48	1
#2	64	1

(b) [a]

Resources	#1 units	#1 (%)	#2 units	#2 (%)
Block RAMs	0	00.00	0	00.00
DSP blocks	82	37.27	114	51.82

[a] Note that only block RAMs and DSP blocks are displayed here because Core Generator only supports resource occupation's preview for these two types of units.

This comparison can be of greater help for the system designer since there are some relevant differences between both filters. Whether the designer's concern is more focused on area or speed optimization, then the RRC filter #1 is the better option since it produces accurate results with less FPGA space; however, it is proven that a wider filter (like #2) also provides similar results, although it uses more resources and spends more time. Since the following Simões [1] approach is one of the main focuses of this work, filter #2 will be used in implementation.

5.2.2.B Bit rate

Filter #2's input and output bit rates were computed: since the filter has two channels (one for I and one for Q samples), it can handle 2 input values of 22 bits for each 50 cycles, which translates to an input bit rate of $88Mbit/s$. However, due to the implemented pipeline, which makes it possible to acquire 2 inputs of 22 bits each for each cycle, the operating input rate effectively ascends to $4.4Gbit/s$, which is superior than the Generator's output rate, thus not having any loss of information.

The output rate is simple: it would require 50 cycles to transform an input of 2 values of 22 bits into an output of 2 values with 16 bits each, so the output rate would be $64Mbit/s$, but the pipelining inserts a new arrangement, on which the filter presents 2 new values each cycle. This generates a effective output of $3.2Gbit/s$ (the decrease of the rate between the input and the output can be explained with the decrease of bits in each output sample, relative to the bit number in each input).

5.2.3 LINC decomposer

This block consists in a LINC decomposer of an input signal. Two approaches were taken for this element: the first is a complete LINC calculator - which from now on will be called "LINC calculator" - where all the steps are fully calculated, and the second is a LUT-based method - which will be called "LINC LUT" - that is used to perform the most expensive arithmetic operation in LINC's decomposition, which is (in terms of resources) the square-root[5]. Although two different approaches are being used, the results should be approximately the same, and in both cases, for each complex sample that enters the block, two complex samples are generated, one for the left branch and one for the right branch.

5.2.3.A Timing and resources

The utilized resources of LINC calculator are stated in Table 5.4.

Table 5.4: LINC calculator spent resources and timing

(a)

Timing (ns)	
Estimated	Required
8.62	10.00
Latency (cycles)	
Total	Input
75	1

(b)

Resources	units	% of Total
LUTs	25 595	48.11
FFs	16 266	15.29
Block RAMs	0	00.00
DSP blocks	36	16.36

[5] See Eq. 2.13 from Chapter 2.

Next, the utilized resources of the LINC LUT are presented, in Table 5.5. The LUT was built in MATLAB [17], with fixed-point values Q6.14[6], which can be consulted on Annex C.

Table 5.5: LINC LUT spent resources and timing

(a)

Timing (ns)	
Estimated	Required
8.77	10.00
Latency (cycles)	
Total	Input
4	1

(b)

Resources	units	% of Total
LUTs	552	1.04
FFs	485	0.46
Block RAMs	3	2.14
DSP blocks	8	3.64

By comparing these two approaches, one can easily relate the LUT implementation to a faster and cheaper way of decomposing a signal into LINC, against the Calculator. The only resource where the LUT overpasses the Calculator are the RAM blocks, since this approach uses stored values. Yet, it only uses about $1,8\%$ of the total, which is almost not representative for the final design.

However, using a LUT instead of real-time calculation has drawbacks: the fact that the LUT has limited entries (and, in consequence, limited outputs) and the stored values have non-infinite precision (in this case, minimum variations of 2^{-14}), some imprecisions on the output signals can be inserted.

Table 5.6 displays the percentages of both LINC implementations, and compares them with colors (green highlights the cell that contains the less used quantity of a resource; red highlights the opposite).

Table 5.6: LINC implementations comparison

Resources	% LUT	% Calculator
LUTs	1.04	48.11
FFs	0.46	15.29
Block RAMs	2.14	0.00
DSP blocks	3.64	16.36

5.2.3.B Bit rate

Although both blocks implement LINC decomposition, one spends less time than the other to generate an output. For this reason, bit rate will be computed for both approaches, with a final comparison between them.

[6]Notation: Q $Bits_{Integer}.Bits_{Fractional}$.

For the Calculator block, 75 cycles are needed to transform an input into an output. This block receives 4 values of 16 bits each, which is equal to an input bit rate of $85.33 Mbit/s$; the implemented pipeline makes possible to receive 4 new values each cycle, thus providing a effective bit rate of $6.4 Gbit/s$, when in constant operation. Regarding the output, this block releases 4 16-bit values, which is the same that it receives in the input. Due to this, the output bit rates are equal to the input ones, thus not being necessary to calculate them again.

On the other hand, the LUT block also receives 4 16-bit values, but needs only 4 cycles to calculate an output from an input. In this case, an input bit rate of $1.6 Gbit/s$ is achievable. Due to pipelining, this bit rate can be increased up to $6.4 Gbit/s$ when in constant operation, since it latency passes to 1 cycle. Like in the Calculator block, in the LUT block the number of released bits in the output is equal to the number in the input side. This leads us to the same conclusion of the Calculator: the input and output rates are equal for this block.

It should be noted that, once again, the input bit rate of the generic LINC decomposition block is greater than the output bit rate of the RRC filter, which is the preceding block. Because of this, it can be stated that no information is lost when the system is constantly operating.

5.3 Complete system

The study of all the building blocks mentioned above was introduced previously so the reader could have an idea of which parts are faster in terms of latency or more expensive to implement. In the first part of this final section, the basic occupation of the complete system[7] without all these building blocks will be presented, i.e., just the original transmission system, altered to include two FMC30RF cards at the same time.

5.3.1 Original system' occupation

Table 5.7 depicts the original system' occupation.

By looking at these values, it is easy to see that the original design leaves more space for any extra blocks that could be implemented; however, the block RAMs are mostly used, and only about 49% of space is left unused.

As stated in Chapter 1 (Sub-chapter 1.1), in order to reach the proposed objective of this thesis, two identical systems were created, based on two different approaches:

[7]The final system was built in ISE, which produces more detailed occupation summaries than Vivado HLS or Core Generator; however, to keep it simple, just the previously analyzed data will be transcribed to the tables.

Table 5.7: Original system's occupation

Resources	units	% of Total
LUTs	6 072	11.41
FFs	8 091	7.60
Block RAMs	71	50.71
DSP blocks	0	0.00

- **Complete system** #1: LUTs for each of the RMM and LINC blocks;
- **Complete system** #2: a LUT for the RMM block and a hardware calculation block for LINC;

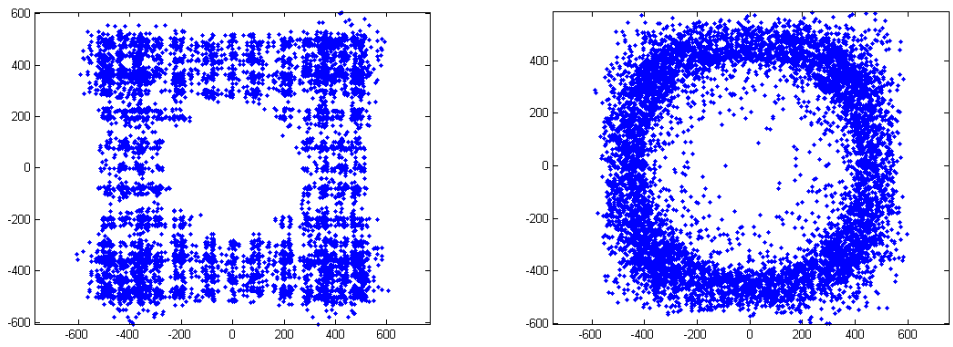
Both systems will now be evaluated in terms of occupation, timings and output signals.

5.3.2 System #1's occupation and results

The complete system #1 is composed by the original design, plus one Generator with RMM and Upscaling, one RRC filter and one LINC Decomposer (LUT form). The implementation details are shown in Table 5.8:

Table 5.8: Complete system #1's occupation

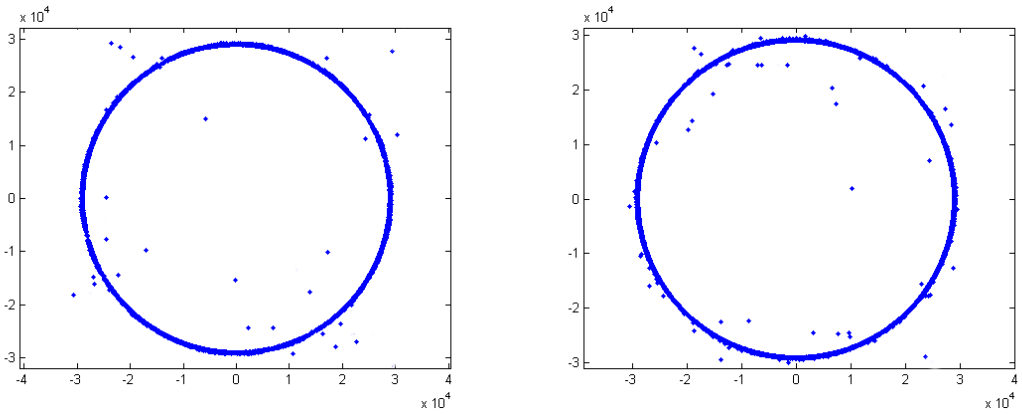
Resources	units	% of Total
LUTs	27 623	51.24
FFs	19 061	17.91
Block RAMs	71	50.71
DSP blocks	146	66.36



(a) without RMM (b) with RMM

Figure 5.3: Output of RRC filter - LUT implementation

Apart from this, the output signals from the RRC and the LINC blocks were gathered, in order to understand if the generation and decomposition were being made correctly. Figure 5.3 shows those results for the RRC out port and Figure 5.4 shows them for the LINC block (it is only shown the left arm because the right LINC arm has a similar behavior).



(a) without RMM (b) with RMM

Figure 5.4: Output of LINC left branch - LUT implementation

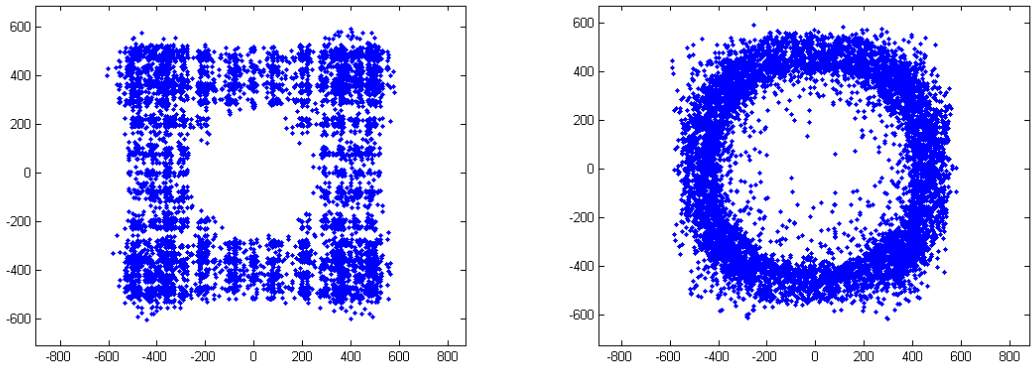
5.3.3 System #2's occupation and results

The complete system #2 is composed by the original design, plus one Bit Generator with RMM and Upscaling, one RRC filter and one LINC Decomposer (Calculator form). The implementation details are shown in Table 5.9:

Table 5.9: Complete system #2's occupation

Resources	units	% of Total
LUTs	38 172	71.75
FFs	29 764	27.97
Block RAMs	71	50.71
DSP blocks	174	79.09

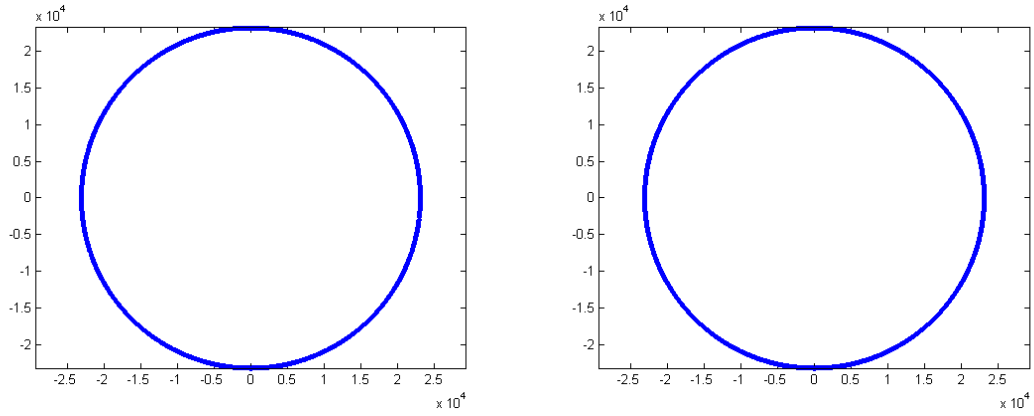
Data was also acquired from transmission tests, from the host-to-board connection. Figure 5.5 shows the gathered outputs from the RRC out port, with and without RMM, for a random bitstream.



(a) without RMM (b) with RMM

Figure 5.5: Output of RRC filter - Calculator implementation

The output of the LINC left branch was also acquired, with and without RMM, for a random bitstream. Results are shown in Figure 5.6.



(a) without RMM (b) with RMM

Figure 5.6: Output of LINC left arm - Calculator implementation

5.4 Final comparison and observations

A simple analysis of this data shows what was expected: the full-LUT system (#1) does not spend as many resources as the calculator system (#2), at the cost of less-precision operations, resulting in a less accurate LINC decomposition.

However, a more detailed comparison with the above presented numbers shows that the predicted occupation for all the HLS-generated blocks does not match with the final occupation numbers, especially when looking to block RAM values, whose expected growth for System #1 and #2 should be 16.43% and 14.29%, respectively.

Table 5.10: Occupation and comparison of all systems

	Original System	System #1		System #2	
Resources	% of Total	% of Total	% of Increase	% of Total	% of Increase
LUTs	11.41	51.24	+39.83	71.75	+60.34
FFs	7.60	17.91	+10.31	27.97	+20.37
Block RAMs	50.71	50.71	+0.00	50.71	+0.00
DSP blocks	0.00	66.36	+66.36	79.09	+79.09

This can be explained with a possible inaccuracy of Vivado HLS to predict the used resources, possibly giving a worst-case estimate, or it can be ISE itself, that optimizes resource utilization when synthesizing the full design: some blocks can being used simultaneously for different operations, for example.

Table 5.11: Expected vs. final occupation results

	System #1		System #2	
Resources	% Final	% Expected	% Final	% Expected
LUTs	51.24	28.62	71.75	75.69
FFs	17.91	16.11	27.97	30.94
Block RAMs	50.71	66.43	50.71	65.00
DSP blocks	66.36	73.64	79.09	86.36

In system #1, it is noticeable the overestimation of block RAMs and DSP blocks, while LUTs and FFs are underestimated. On the other hand, system #2 is overestimated in all parameters.

6

Conclusions

This thesis is a culminating point of the project GLANCES [15], whose one objective was to build a real-time SC transmitter with LINC-RMM in FPGA environment, and also the proof-of-concept of the system developed by Simões [1], whose results were very promising and were successfully replicated in practice.

The focus of this work was, since the beginning, to engineer a complete, optimized and entirely new system that could embrace two types of signal decomposition's state of the art techniques, and implement it, showing their feasibility in real-world problems. The applications of this project are broad, but can be greatly used in mobile systems, since RMM and LINC can effectively pair up to reduce mainly a system's cost and PAPR.

Two implementations were made, concerning this arrangement: one implements real-time LINC decomposition and the other makes use of a precomputed memory block. Both systems have their advantages, one being of the first its precision calculations, making it suited for critical applications. The second is in its quickness and low resource utilization, although its results are less accurate than in the first system, being the latter best suited for low-cost applications. However, in terms of optimization, this system can be considered yet in its first steps, and much more can be done regarding this subject.

6.1 Future work

Both implemented systems work well. However, a more profound study on what computed LUTs work better on the LINC-LUT system would be needed to tune it up, since the low quantity of bits assigned to the integer part of the stored values can greatly affect the overall performance in some cases. Also, a different approach on how to build the LINC-LUT system could bring better results than the ones presented in this thesis.

Regarding the Generator block, a more uniform bit generation algorithm that does not use *time.h* C package can be studied, since the current algorithm still has some deviations, although it serves the purpose.

Also, a deeper analysis of both LINC branches' synchronization would be desirable, since it was always done by software and FPGA signaling, assuming that the refreshing signals would arrive exactly at the same time. As there is a possibility that this synchronization may not being done the right way, the utilization of an external clock source would be advised in this study, as it has never been done through this work.

Besides this, this implementation did not use all FPGA's resources, which mean that there is still space available to implement more systems, or to optimize the current ones.

Finally, the same system could be implemented, but with some parts working in the PS, i.e., inside the ARM processor, and then compare them to the systems implemented here, in terms of resources and timing, to see if there are any relevant gains.

Bibliography

[1] A. Simões, "Ring-type magnitude modulation for LINC: Paving the road for better efficiency," *M.Sc. dissertation, Universidade de Coimbra, Portugal*, 2014.

[2] G. Li, Z. Xu, C. Xiong, C. Yang, S. Zhang, Y. Chen, and S. Xu, "Energy-efficient wireless communications: tutorial, survey, and open issues," *Wireless Communications, IEEE*, vol. 18, no. 6, pp. 28–35, December 2011.

[3] A. Birafane, M. El-Asmar, A. Kouki, M. Helaoui, and F. Ghannouchi, "Analyzing LINC systems," *Microwave Magazine, IEEE*, vol. 11, no. 5, pp. 59–71, Aug 2010.

[4] P. Reynaert and M. Steyaert, *RF Power Amplifiers for Mobile Communications*. Springer, 2006.

[5] M. Hunton, "System and method for post-filtering peak power reduction in communications systems," *US Patent 7 170 952 B2*, 2007.

[6] N. Lashkarian, E. Hemphill, H. Tarn, H. Parekh, and C. Dick, "Reconfigurable digital front-end hardware for wireless base-station transmitters: Analysis, design and FPGA implementation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 8, pp. 1666–1677, Aug. 2007.

[7] D. Cox, "Linear amplification with nonlinear components," *Communications, IEEE Transactions on*, vol. 22, no. 12, pp. 1942–1945, Dec 1974.

[8] R. Dinis and A. Gusmão, "Nonlinear signal processing schemes for OFDM modulations within conventional or LINC transmitter structures," *European Transactions on Telecommunications*, vol. 19, no. 3, pp. 257–271, 2008.

[9] S. Cripps, "RF power amplifiers for wireless communications," *Microwave Magazine, IEEE*, vol. 1, no. 1, pp. 64–64, Mar 2000.

[10] R. Dinis, P. Montezuma, N. Souto, and J. Silva, "Iterative frequency-domain equalization for general constellations," in *Sarnoff Symposium, 2010 IEEE*, April 2010.

[11] P. Bento, M. Gomes, V. Silva, R. Dinis, and P. Montezuma, "A multi-antenna technique for mm-wave communications with large constellations and strongly nonlinear amplifiers," *German Microwave Conference*, pp. 284–287, March 2015.

[12] F. J. Harris, *Multirate Signal Processing for Communication Systems*. Prentice Hall PTR, 2004.

[13] A. Simões, M. Gomes, R. Dinis, V. Silva, and F. Cercas, "Magnitude modulation applied to LINC transmitters: Paving the road for better efficiency," *IEEE 80th Vehicular Technology Conference: VTC2014-Fall*, Sep 2014.

[14] A. Aref, A. Askar, A. Nafe, M. Tarar, and R. Negra, "Efficient amplification of signals with high papr using a novel multilevel linc transmitter architecture," *7th European Microwave Integrated Circuits Conference (EuMIC)*, pp. 655–658, Oct. 2012.

[15] http://www.it.pt/project_detail_p.asp?ID=1939, August 2015.

[16] H. Phoon, M. Yap, and C. Chai, "A highly compatible architecture design for optimum fpga to structured-asic migration," *International Conference on Semiconductor Electronics, IEEE*, pp. 506–510, October 2006.

[17] http://www.mathworks.com/products/matlab/, August 2015.

[18] R. Duren, J. Stevenson, and M. Thompson, "A comparison of fpga and dsp development environments and performance for acoustic array processing," *MWSCAS, IEEE*, pp. 1177–1180, 2007.

[19] R. Sernec, M. Zajc, and J. Tasic, "The evolution of dsp architectures: Towards parallelism exploitation," *MELECON, IEEE*, vol. 2, pp. 782–785, 2000.

[20] http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html, August 2015.

[21] http://www.xilinx.com/, August 2015.

[22] http://www.vita.com/Specifications, August 2015.

[23] http://www.4dsp.com/FMC30RF.php, August 2015.

[24] http://www.4dsp.com/, August 2015.

[25] L. Sundstrom, "The effect of quantization in a digital signal component separator for LINC transmitters," *Vehicular Technology, IEEE Transactions on*, vol. 45, no. 2, pp. 346–352, May 1996.

[26] S. A. Hetzel, A. Bateman, and J. McGeehan, "A LINC transmitter," in *Vehicular Technology Conference, 1991. Gateway to the Future Technology in Motion., 41st IEEE*, May 1991, pp. 133–137.

[27] P. Colantonio, F. Giannini, and M. Rossi, "Rf experimental implementation of linc technique," *European Microwave Conference*, pp. 56–59, Oct. 2007.

[28] L. Panseri, L. Roman'o, S. Levantino, C. Samori, and A. Lacaita, "Low-power all-analog component separator for an 802.11 a/g linc transmitter," *Proceedings of the 32nd European Solid-State Circuits Conference*, pp. 271–274, Sep. 2007.

[29] Y. Tian, O. Hammi, S. Boumaiza, and F. Ghannouchi, "Design and optimization of digital signal components separator of linc using fpga processors," *IEEE International Conference on Signal Processing and Communications*, pp. 836–839, Nov. 2007.

[30] R. Ferrão, M. Gomes, and V. Silva, "International conference on field programmable logic and applications," *IEEE Trans. Commun.*, pp. 1–4, Sep. 2013.

[31] Y. Chabaane, "Développement d'une plateforme matérielle pour l'implémentation des techniques de décomposition de signaux pour les amplificateurs depuissance à deux branches," *M.Sc. dissertation, Université du Québec, Canadá*, 2011.

[32] P. Vizarreta, P. Gilabert, G. Montoro, and J. Berenguer, "Implementación de un transmisor linc en un procesador fpga," *XXV Simposium Nacional de la Unión Científica Internacional de Radio*, pp. 1–4, 2011.

[33] http://www.xilinx.com/products/intellectual-property/cordic.html, August 2015.

[34] M. Gomes, "Magnitude modulation for peak power control in single carrier communication systems," Ph.D. dissertation, Universidade de Coimbra, Portugal, 2010.

[35] M. Gomes, V. Silva, F. Cercas, and M. Tomlinson, "Power efficient back-off reduction through polyphase filtering magnitude modulation," *Communications Letters, IEEE*, vol. 13, no. 8, pp. 606–608, August 2009.

[36] A. Birafane and A. Kouki, "On the linearity and efficiency of outphasing microwave amplifiers," *Microwave Theory and Techniques, IEEE Transactions on*, vol. 52, no. 7, pp. 1702–1708, July 2004.

[37] http://www.i2c-bus.org/, August 2015.

[38] http://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf, August 2015.

[39] http://support.4dsp.com/support/discussions/topics/5000039830, August 2015.

[40] http://www.4dsp.com/pdf/FMC30RF_data_sheet.pdf, August 2015.

[41] http://www.ettus.com/product/details/UB210-KIT, August 2015.

[42] http://gnuradio.org/redmine/projects/gnuradio/wiki, August 2015.

[43] http://www.xilinx.com/products/design-tools/ise-design-suite.html, August 2015.

[44] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, August 2015.

[45] https://www.visualstudio.com/, August 2015.

[46] http://www.4dsp.com/software_stellar.php, August 2015.

[47] http://support.4dsp.com/support/discussions/topics/5000039870, August 2015.

[48] http://www.xilinx.com/products/design-tools/vivado.html, August 2015.

[49] http://support.4dsp.com/support/discussions/topics/5000039678, August 2015.

[50] http://www.xilinx.com/tools/coregen.htm, August 2015.

[51] http://www.bluespec.com/high-level-synthesis-tools.html, August 2015.

[52] http://panda.dei.polimi.it/, August 2015.

[53] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C - The Art of Scientific Computing*. Cambridge University Press, 2002.

[54] http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf, August 2015.

[55] http://www.xilinx.com/tools/sdk.htm, August 2015.

[56] D. L. USA, *FMC30RF Star - A/D Daughter Card*, 2014.

[57] http://www.xilinx.com/support/documentation/selection-guides/zynq7000-product-table.pdf, September 2015.

A

Implemented ISE VHDL schematic

This section displays the complete ISE VHDL schematic in register-transfer level (RTL). There is only one block missing, which is the implemented FIR filter. This filter was designed in Core Generator and was included inside the ISE project; however, the tool cannot represent Core Generator blocks inside RTL designs.

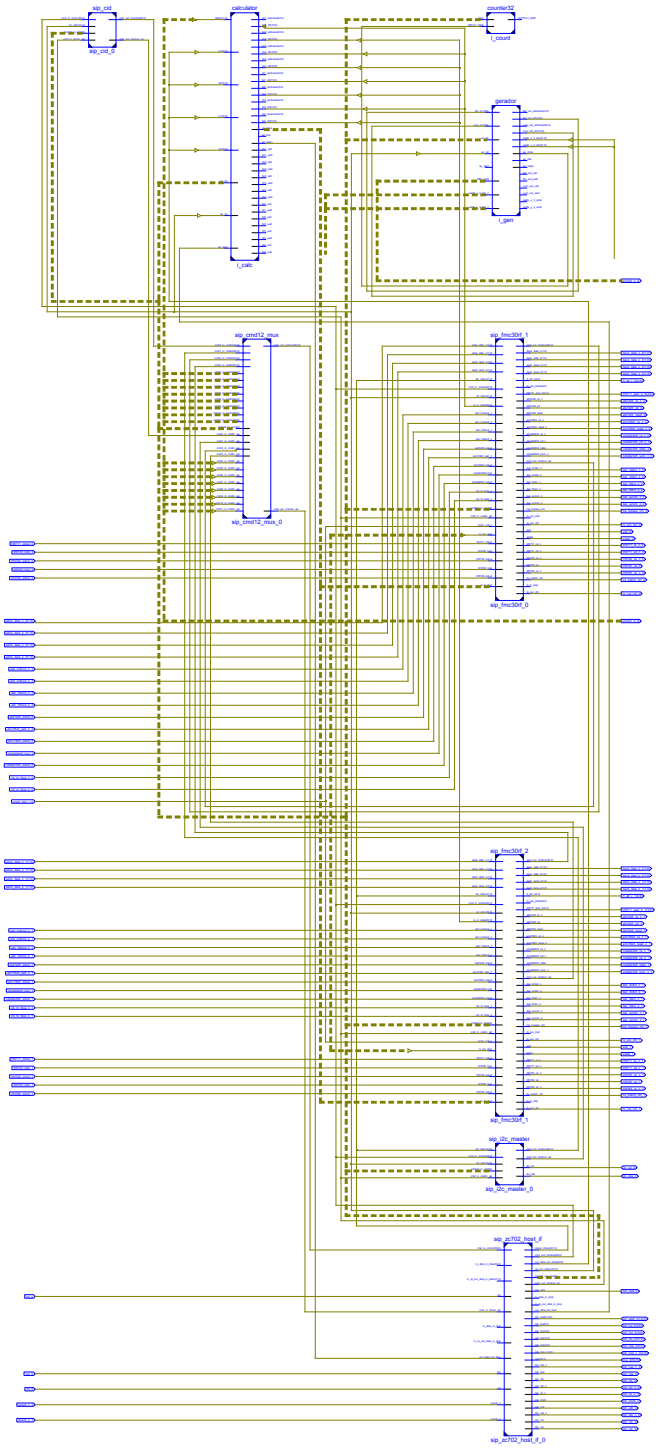


Figure A.1: Complete VHDL design, translated to RTL.

B

FIR Filter Implementation

B. FIR Filter Implementation

This section adds some information about the FIR filter implementation and includes some images to help understanding the design. The first presented filter has $N_{sym} = 5$, and the second has $N_{sym} = 7$; both have even coefficients' symmetry. In both implementations, a fixed-point representation of the coefficients was used, of type Q0.16[1]; bits are being interpreted in 2's complements and none of them represent an integer part of the number.

Table B.1: FIR filter coefficients for $N_{sym} = 5$

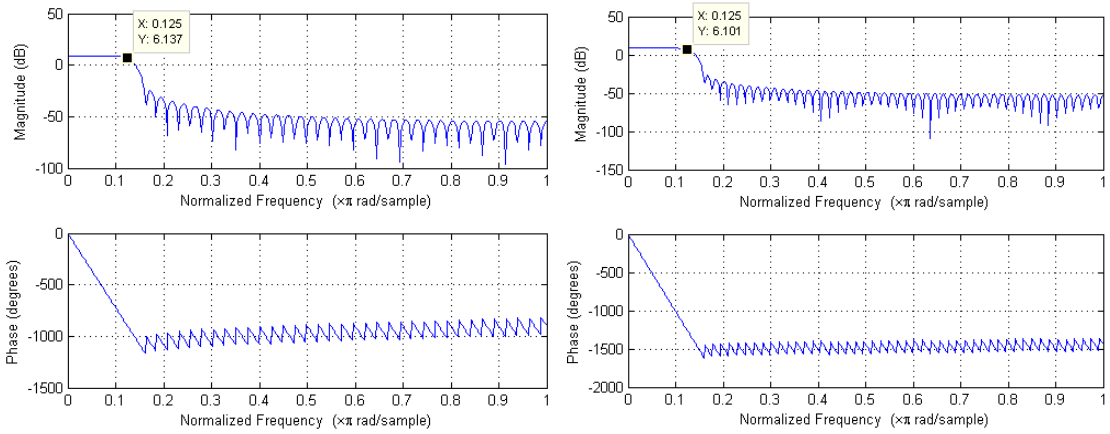
1	-0.0026526	22	0.0306119	43	0.3334593	64	-0.0070360
2	-0.0038362	23	0.0332705	44	0.2827816	65	-0.0132629
3	-0.0040413	24	0.0295055	45	0.2198386	66	-0.0159092
4	-0.0030788	25	0.0187566	46	0.1513058	67	-0.0150943
5	-0.0010370	26	0.0017434	47	0.0840969	68	-0.0115661
6	0.0017042	27	-0.0194438	48	0.0244740	69	-0.0064688
7	0.0045266	28	-0.0414858	49	-0.0227113	70	-0.0010681
8	0.0066916	29	-0.0602093	50	-0.0545421	71	0.0035101
9	0.0075026	30	-0.0711544	51	-0.0702572	72	0.0064799
10	0.0064799	31	-0.0702572	52	-0.0711544	73	0.0075026
11	0.0035101	32	-0.0545421	53	-0.0602093	74	0.0066916
12	-0.0010681	33	-0.0227113	54	-0.0414858	75	0.0045266
13	-0.0064688	34	0.0244740	55	-0.0194438	76	0.0017042
14	-0.0115661	35	0.0840969	56	0.0017434	77	-0.0010370
15	-0.0150943	36	0.1513058	57	0.0187566	78	-0.0030788
16	-0.0159092	37	0.2198386	58	0.0295055	79	-0.0040413
17	-0.0132629	38	0.2827816	59	0.0332705	80	-0.0038362
18	-0.0070360	39	0.3334593	60	0.0306119	81	-0.0026526
19	0.0021290	40	0.3663228	61	0.0230857		
20	0.0128334	41	0.3777046	62	0.0128334		
21	0.0230857	42	0.3663228	63	0.0021290		

The filter is an RRC, that was first designed using MATLAB [17] and whose

[1]Notation: Q $Bits_{Integer}.Bits_{Fractional}$.

coefficients are presented on Table B.1. The used parameters are $N_{sym} = 5$, $L = 8$ and $rolloff = 0.25$, which produced a filter with $length_{RRC} = 2 * N_{sym} * L + 1 = 81$ taps. This filter has a band-pass magnitude of approximately 9 dB, with a -3 dB cutoff frequency at approximately $0.125 * \pi$ radians/s, as seen in Figure B.1a.

The second filter is also an RRC, with MATLAB-designed coefficients [17] in Table B.2. All the parameters are the same as the first filter, except N_{sym}, which is equal to 7. The filter size is then obtained as $length_{RRC} = 2 * N_{sym} * L + 1 = 113$. Bandwidth-related details of this filter are the same of the first one, as it can be seen on Figure B.1b.



(a) $N_{sym} = 5$ (b) $N_{sym} = 7$

Figure B.1: RRC filters details

Both filters receive type Q4.18[2] inputs and transmit Q7.9 type outputs. The tool used to generate this core for the final project was Core Generator. It is very simple to use and provides a graphic relation between the ideal implemented filter and the expected one after software quantization, as depicted in Figure B.2.

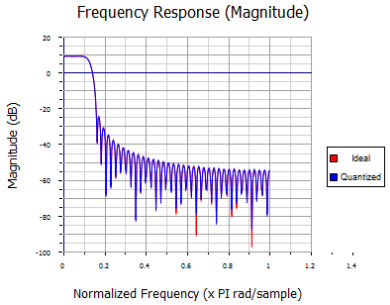


Figure B.2: RRC filter implementation details.

[2]Notation: Q $Bits_{Integer}.Bits_{Fractional}$.

Table B.2: FIR filter coefficients for $N_{sym} = 7$

1	0.0018947	30	-0.0115661	59	0.3334593	88	0.0064799
2	0.0008729	31	-0.0150943	60	0.2827816	89	0.0075026
3	-0.0004197	32	-0.0159092	61	0.2198386	90	0.0066916
4	-0.0017034	33	-0.0132629	62	0.1513058	91	0.0045266
5	-0.0026812	34	-0.0070360	63	0.0840969	92	0.0017042
6	-0.0031078	35	0.0021290	64	0.0244740	93	-0.0010370
7	-0.0028518	36	0.0128334	65	-0.0227113	94	-0.0030788
8	-0.0019343	37	0.0230857	66	-0.0545421	95	-0.0040413
9	-0.0005359	38	0.0306119	67	-0.0702572	96	-0.0038362
10	0.0010359	39	0.0332705	68	-0.0711544	97	-0.0026526
11	0.0024109	40	0.0295055	69	-0.0602093	98	-0.0008833
12	0.0032410	41	0.0187566	70	-0.0414858	99	0.0009840
13	0.0032869	42	0.0017434	71	-0.0194438	100	0.0024862
14	0.0024862	43	-0.0194438	72	0.0017434	101	0.0032869
15	0.0009840	44	-0.0414858	73	0.0187566	102	0.0032410
16	-0.0008833	45	-0.0602093	74	0.0295055	103	0.0024109
17	-0.0026526	46	-0.0711544	75	0.0332705	104	0.0010359
18	-0.0038362	47	-0.0702572	76	0.0306119	105	-0.0005359
19	-0.0040413	48	-0.0545421	77	0.0230857	106	-0.0019343
20	-0.0030788	49	-0.0227113	78	0.0128334	107	-0.0028518
21	-0.0010370	50	0.0244740	79	0.0021290	108	-0.0031078
22	0.0017042	51	0.0840969	80	-0.0070360	109	-0.0026812
23	0.0045266	52	0.1513058	81	-0.0132629	110	-0.0017034
24	0.0066916	53	0.2198386	82	-0.0159092	111	-0.0004197
25	0.0075026	54	0.2827816	83	-0.0150943	112	0.0008729
26	0.0064799	55	0.3334593	84	-0.0115661	113	0.0018947
27	0.0035101	56	0.3663228	85	-0.0064688		
28	-0.0010681	57	0.3777046	86	-0.0010681		
29	-0.0064688	58	0.3663228	87	0.0035101		

C

LINC Look-Up Table

This section adds some information about the LINC LUT implementation. The values were obtained in MATLAB according with the function:

$$output = \sqrt{\frac{r_{max}^2}{input} - 1} \tag{C.1}$$

where r_{max} is the maximum amplitude threshold and $input = I^2 + Q^2$, being I and Q the input in-phase and quadrature components, respectively.

The LUT was designed to have 4096 entries (from index 0 to 2^{12}), with fixed-point values, of type Q6.14[1]. In order to compute the LUT with a simpler approach in MATLAB, r_{max}^2 was assigned to 4095, and the 4096 entries were created by setting $input = i$, $i \in \{0,1,2,...,4094,4095\}$. The beginning and the end of the table are transcribed in Table C.1.

Table C.1: LUT input addresses and corresponding output values

Address	Value	Address	Value
0	11111111111111111111	...	...
1	11111111111100000000	4088	00000000001010100110
2	10110100111101000000	4089	00000000001001110100
3	10010011101110101100	4090	00000000001000111101
4	01111111111011000000	4091	00000000001000000000
5	01110010011001110010	4092	00000000000110111100
6	01101000011011000001	4093	00000000000101101010
7	01100000101010100001	4094	00000000000100000000
...	...	4095	00000000000000000000

[1]Notation: Q $Bits_{Integer}.Bits_{Fractional}$.

D

C++ Code of the Implemented Blocks in HLS

This section presents the developed code for the HLS-designed blocks, in C++, namely the LINC Calculator, the LUT LINC and the Generator, which includes the RMM and the Upsampler.

D.1 LINC Calculator

```
#include "hls_math.h"
#include "ap_int.h"
#include <stdio.h>
#include <stdlib.h>

void calculator(short int x1, short int y1, short int x2, short int y2, unsigned int
     MAX, short int Sr1[1], short int Sr2[1], short int Sc1[1], short int Sc2[1], short
     int Sr3[1], short int Sr4[1], short int Sc3[1], short int Sc4[1]){
#pragma HLS PIPELINE
#pragma HLS INTERFACE ap_none port=x1,y1,x2,y2

   float div[2];
   double u=x1*x1+y1*y1;
   if((x1==0 && y1==0) || u>MAX)
      div[0]=1; //if over the limit, set equal to 1 so the final sqrt() value is 0
   else
      div[0]=(float) MAX/u;

   u=x2*x2+y2*y2;
   if((x2==0 && y2==0) || u>MAX)
      div[1]=1; //if over the limit, set equal to 1 so the final sqrt() value is 0
   else
      div[1]=(float) MAX/u;

   div[0]-=1;
   div[0]=hls::sqrtf(div[0]);

   div[1]-=1;
   div[1]=hls::sqrtf(div[1]);


   Sr1[0] = ((int) (x1-div[0]*y1))>>1; //S1r - left (1st sample)
   Sc1[0] = ((int) (y1+div[0]*x1))>>1; //S1c - left (1st sample)

   Sr2[0] = ((int) (x1+div[0]*y1))>>1; //S2r - right (1st sample)
   Sc2[0] = ((int) (y1-div[0]*x1))>>1; //S2c - right (1st sample)
```

```
   Sr3[0] = ((int) (x2-div[1]*y2))>>1; //S1r - left (2nd sample)
   Sc3[0] = ((int) (y2+div[1]*x2))>>1; //S1c - left (2nd sample)

   Sr4[0] = ((int) (x2+div[1]*y2))>>1; //S2r - right (2nd sample)
   Sc4[0] = ((int) (y2-div[1]*x2))>>1; //S2c - right (2nd sample)

   }
```

D.2 LUT LINC

```
#include "ap_fixed.h"

ap_ufixed<20,6> xk[4096]={
   //values of the implemented LINC LUT
   //which will not be transcribed
};

void tabela(ap_int<16> x1, ap_int<16> y1, ap_int<16> x2, ap_int<16> y2, ap_int<16>
     Sr1[1], ap_int<16> Sc1[1], ap_int<16> Sr2[1], ap_int<16> Sc2[1], ap_int<16> Sr3[1],
     ap_int<16> Sc3[1], ap_int<16> Sr4[1], ap_int<16> Sc4[1]){
#pragma HLS PIPELINE
#pragma HLS INTERFACE ap_none port=x1,x1,y1,y2,Sr1,Sc1,Sr2,Sc2,Sr3,Sc3,Sr4,Sc4

   ap_uint<32> sum=(x1*x1+y1*y1)(31,11);

   ap_int<16> sum_disc=sum & 4095;
   ap_ufixed<20,6, AP_RND_CONV, AP_SAT> div[2];
   div[0]= xk[sum_disc]; //search the LUT

   sum=(x2*x2+y2*y2)(31,11);
   sum_disc=sum & 4095;
   div[1]= xk[sum_disc]; //search the LUT

   //the division by 2 is already considered in the bit separation

   Sr1[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)x1-(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[0]*y1)(24,9); //S1r - left (1st sample)
   Sc1[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)y1+(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[0]*x1)(24,9); //S1c - left (1st sample)

   Sr2[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)x1+(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[0]*y1)(24,9); //S2r - left (1st sample)
```

```
   Sc2[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)y1-(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[0]*x1)(24,9); //S2c - left (1st sample)

   Sr3[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)x2-(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[1]*y2)(24,9); //S1r - right (2nd sample)
   Sc3[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)y2+(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[1]*x2)(24,9); //S1c - right (2nd sample)

   Sr4[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)x2+(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[1]*y2)(24,9); //S2r - right (2nd sample)
   Sc4[0] = 10* ((ap_fixed<25,16, AP_RND_CONV, AP_SAT>)y2-(ap_fixed<25,16, AP_RND_CONV,
        AP_SAT>)div[1]*x2)(24,9); //S2c - right (2nd sample)*/
}
```

D.3 Generator

```
#include "ap_fixed.h"
#include "hls_math.h"

ap_fixed<20,2> table_MM_LUT_i[16384]={
   //values of the implemented RMM I table
   //which will not be transcribed
};

ap_fixed<20,2> table_MM_LUT_q[16384]={
   //values of the implemented RMM Q table
   //which will not be transcribed
};

void gerador(ap_uint<20> jran_out[1], ap_uint<20> jran_in, unsigned short num_out[1],
     unsigned short int num_in, bool MM_onoff, ap_fixed<22,4> saida_x[32],
     ap_fixed<22,4> saida_y[32]){

#pragma HLS INTERFACE ap_fifo depth=64 port=saida_x,saida_y
#pragma HLS PIPELINE

   //JRAN must be initialized between 0 and (im-1) //numerical recipes based
   ap_uint<13> ia=4096;
   ap_uint<4> j=0;
   ap_uint<20> ic=150889, im=714025;
   ap_uint<32> jran=357;//612359;

   static unsigned short int Nsym=8;
   static unsigned short int upscale=8;
```

```
unsigned bit;
ap_uint<2> value[8]; //HLS does not accept value[Nsym]
short int i=0;

unsigned short int num=0;

ap_fixed<20,2> MagMod_i, MagMod_q;

ap_fixed<22,4> *x=&saida_x[0], *y=&saida_y[0];

if(jran_in!=0)
   jran=jran_in;
if(num_in!=0)
   num=num_in;

//rand
for (i=0;i<Nsym/2;i++){
   j= (ap_uint<24>) (4*jran)/im; //floor rounding is intended
   jran=(ap_uint<32>) (jran*ia+ic)%im;
   value[i]= j & 0x0003;
}
jran_out[0]=jran;

//mm+upscale
for(i=0; i<Nsym/2; i++){
   #pragma HLS unroll
   /*
   * NEW BITS ENTER LIKE THIS
   * +-----+--+--+--+--+--+--+
   * | ... |Q0|I0|Q1|I1|Q2|I2| = I1 and Q1 are more recent than I0 e Q0
   * +-----+--+--+--+--+--+--+
   *     <----------------
   */
   num=(unsigned short int) (num<<2) | value[i];
   num=num & 0x3FFF;
   //num=short int = 16bits(=2bytes) => only the right 14 bits
   //so => 0011 1111 1111 1111 = 3FFF

   //Send temp[] array to LUT table
   if(MM_onoff){
      MagMod_i=table_MM_LUT_i[num];
      MagMod_q=table_MM_LUT_q[num];
   }
   else{
      MagMod_i=(ap_fixed<20,2>) 1;
      MagMod_q=(ap_fixed<20,2>) 1;
   }
```

```
    MagMod_i *= (ap_fixed<20,2>)0.707106781;

    if(((num>>6) & 1)==1) //I
    MagMod_i*= (ap_fixed<20,2>) -1;

    MagMod_q*= (ap_fixed<20,2>) 0.707106781;

    if(((num>>7) & 1)==1) //Q
    MagMod_q*= (ap_fixed<20,2>) -1; // MagMod* sqrt(8) *

    //upscaling factor => 8 (OQPSK)
       for(int j=0; j<upscale;j++){
          #pragma HLS unroll
          if(j==0)
             *(x++)=(ap_fixed<22,4>) (MagMod_i*(ap_fixed<21,3>)2.8284271247);
                  //2.82...=sqrt(upscaling factor)
          else
             *(x++)=0;

          if(j==upscale/2)
             *(y++)=(ap_fixed<22,4>) (MagMod_q*(ap_fixed<21,3>)2.8284271247);
                  //2.82...=sqrt(upscaling factor)
          else
             *(y++)=0;
       }
    }
    num_out[0]=num;
}
```

E

VHDL Code of the Implemented Snippets in ISE

This section presents the developed code for the ISE-designed blocks in VHDL, namely the Counter, the RMM Activator and the Output channel selector.

E.1 Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity counter32 is
   Port ( CLK : in STD_LOGIC;
      INPUT_GEN : in std_logic;
      OUTPUT_RRC : out STD_LOGIC
   );
end counter32;

architecture Behavioral of counter32 is

signal counter : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
signal out_time : std_logic := '0';

begin

OUTPUT_RRC <= out_time;

count_process: process(CLK, INPUT_GEN, out_time)
   begin
      if rising_edge(CLK) then
         if out_time='1' then
            counter <= counter + 1;
         elsif counter > 31 then
            counter <= (others => '0');
            out_time <= '0';
         elsif INPUT_GEN='1' then
            counter <= counter + 1;
            out_time <= '1';
         end if;
      end if;
   end process;

end Behavioral;
```

E.2 RMM activator

```
  LEFT_BUTTON: process(sip_zc702_host_if_0_cmdclk_out_cmdclk, gpio_sw_n)
    --MAGNITUDE MODULATION ON/OFF
  begin
    if rising_edge(sip_zc702_host_if_0_cmdclk_out_cmdclk) then
      if gpio_sw_n = '1' then
        MM <= not MM;
        for t in 1 to 1000000 loop
        end loop;
        -- waiting process to reduce button sensitivity
      end if;
    end if;
  end process;
```

E.3 Output channel selector

```
RIGHT_BUTTON: process(sip_zc702_host_if_0_cmdclk_out_cmdclk, gpio_sw_s, connect_stop)
--RRC OUT / LINC OUT
variable onoff : std_logic := '0';
begin
  if rising_edge(sip_zc702_host_if_0_cmdclk_out_cmdclk) then
    if gpio_sw_s = '1' then
      onoff := not onoff;
      pmod2_0_ls <= onoff;
      pmod2_1_ls <= onoff;
      for t in 1 to 1000000 loop
      end loop;
      -- waiting process to reduce button sensitivity
    end if;
    if connect_stop='0' then
      if onoff = '1' then
        connect_dval <= calculator_0_tx_dval;
        connect_data <= calculator_0_tx_data;
      else
        connect_dval <= out_rrc_valid;
        connect_data <= out_rrc_data & out_rrc_data;
      end if;
    else
      connect_dval <= '0';
      connect_data <= (others=>'0');
    end if;
  end if;
end process;
```