Karlsruher Institute of Technology - KIT

Institute for Information Processing
Technology - ITIV

# Performance Driven Optimizations in FPGA Based QAM Systems

Master Thesis

Alberto Sonnino
October 5, 2015

**Head of Institute:** Prof. Dr.-Ing. Dr. h. c. J. Becker
Prof. Dr.-ing E. Sax
Prof. Dr. rer. nat W. Stork

**Supervisors:** M. Tech G. Shalina
Dipl.-Ing P. Figuli
Prof. Dr.-Ing. Dr. h. c. J. Becker

Karlsruhe Institut of Technologies

**Abstract**

*The purpose of this Master Thesis is to optimize the performance of high-speed Quadrature Amplitude Modulation (QAM) implemented on FPGAs by exploiting the advantageous properties of a mixed time and frequency domain approach.*

*Quadrature Amplitude Modulation (QAM) conveys two signals by using two sinusoidal carrier waves. It provides high speed data rate transmission and is widely used in many different today's applications like television, Wi-Max and satellite communication due to its arbitrary high spectral efficiency.*

*The FPGA technology and the primarily role that it is playing in portable and mobile communications is a matter of much discussions nowadays. Its incomparable cost, flexibility and reconfigurability in project designing is the first reason of its success.*

*This paper presents a complete new design approach for a QAM transmitter based on the Xilinx Virtex 7 FPGA Kit. The issue of the speed optimisation for the filtering operation is covered and a new technique consisting in the exploitation of mixed-domain to increase parallelism is the main goal of this work. Xilinx ISim is used for simulations and functional verifications while the Xilinx ISE 14.7 software ensures the synthesis and the FPGA design implementation.*

## Acknowledgements

*Firstly, I would like to express my sincere gratitude to my tutors Dipl.-Ing P. Figuli and M. Tech G. Shalina for the continuous support of my Master thesis, for their patience, motivation, and immense knowledge. their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better supervisors and mentors for my Master thesis.*

*My sincere thanks also goes to my little brother Lorenzo Sonnino who helped me in the development of program utilities that greatly contributed to the creation of this work.*

*I also place on record, my sense of gratitude to one and all, who directly or indirectly, have lent their hand in this venture.*

# Contents


1 Motivation and Introduction . . . 1
2 Today's State-of-the Art . . . 4
3 Fundamentals . . . 6
3.1 Fundamentals - QAM Modulation . . . 6
3.1.1 QAM Modulation - QAM Mapping . . . 6
3.1.2 QAM Modulation - Modulator . . . 8
3.2 Fundamentals - Fourier Transform . . . 9
3.2.1 Fourier Transform - Theoretical Concepts . . . 10
3.2.2 Fourier Transform - Convolution Property . . . 11
3.3 Fundamentals - Filter . . . 11
3.3.1 Filter - Finite Impulse Response Filters . . . 12
3.3.2 Filter - Squared Raised Root Cosine Filter . . . 14
4 Concepts and Methodology . . . 15
4.1 Concepts and Methodology - Design Strategy . . . 15
4.2 Concepts and Methodology - Conceptual Model . . . 16
5 Implementation . . . 21
5.1 Implementation - Design Pattern . . . 21
5.2 Implementation - QAM Mapper . . . 22
5.3 Implementation - Discrete Fourier Transform . . . 25
5.4 Implementation - SRRC Filter . . . 30
5.5 Implementation - Modulator . . . 33
5.6 Implementation - Transmitter . . . 36
6 Experimental Results . . . 43
6.1 Experimental Results - Design Precision . . . 43
6.2 Experimental Results - Design Resources and Performances . . . 44
6.2.1 Design Resources and Performances - DSP & Mults Combination . 44
6.2.2 Design Resources and Performances - Fabric & LUTs Combination 45
6.2.3 Design Resources and Performances - Fabric & Mults Combination 45
7 Conclusion And Further Improvements . . . 47

**Appendices** **49**
A APPENDIX. Fast Fourier Transform . . . i
B APPENDIX. Design Precision Analysis . . . iii
B.1 Design Precision Analysis - DFT and IDFT . . . iii
B.2 Design Precision Analysis - Filter and Modulator . . . vi

# List of Tables


1 QAM Mapping - Specifications . . . . . 22
2 DFT / IDFT - Specifications . . . . . 26
3 Complex Multiplier ports . . . . . 29
4 Adder Subtracter ports . . . . . 30
5 QAM Mapping - Specifications . . . . . 31
6 Multiplier ports . . . . . 33
7 Modulator - Specifications . . . . . 34
8 Modulator Adder Subtracter ports . . . . . 36
9 Transmitter - Specifications . . . . . 37

# List of Figures


1 Standard Communication Chain . . . 2
2 Standard QAM transmitter . . . 3
3 Mixed-domain QAM transmitter . . . 3
4 State-of-the-art summary chart . . . 4
5 Mixed-domain QAM transmitter - QAM modulation . . . 6
6 QAM mapping . . . 7
7 Supported QAM constellations [10] . . . 7
8 Modulator . . . 9
9 Mixed-domain QAM transmitter - DFT and IDFT Blocks . . . 10
10 Mixed-domain QAM transmitter - Filter . . . 12
11 FIR filter diagram . . . 12
12 SRRC Filter's coefficients . . . 14
13 Methodology . . . 15
14 Mixed-domain QAM transmitter - Conceptual model . . . 16
15 Discrete Fourier Transform (DFT) block - Conceptual model . . . 17
16 Squared Root Raised Cosine (SRRC) filter block - Conceptual model . . . 18
17 Inverse Fast Fourier Transform (IDFT) block - Conceptual model . . . 19
18 Transmitter - Conceptual model . . . 20
19 Parallel bus packing . . . 21
20 Implemented parallel system . . . 21
21 Implemented parallel system - QAM mapping Block . . . 23
22 Implemented parallel system - DFT and IDFT blocks . . . 26
23 Coefficient bus packing . . . 27
24 Implemented parallel system - Filter block . . . 31
25 Implemented parallel system - Modulator block . . . 34
26 Transmitter - Implementation . . . 38
27 Fourier QAM Modulation (FQM) Utility . . . 38
28 Transmitter's precision comparison . . . 43
29 Transmitter's error . . . 43
30 DSP - Mults Resources . . . 44
31 DSP - Mults Performances . . . 44
32 Fabric - LUTs Resources . . . 45
33 Fabric - LUTs Performances . . . 45
34 Fabric - LUTs Resources . . . 46
35 Fabric - Mults Performances . . . 46

36 FFT execution (example with $N = 8$) [14] . . . . . ii
37 Test input signal . . . . . iii
38 DFT's precision comparison . . . . . iv
39 IDFT's precision comparison . . . . . v
40 Filter precision comparison . . . . . vi
41 Modulator precision comparison . . . . . vi

# 1 Motivation and Introduction

SNR (Signal-to-Noise Ratio) improvements and higher data rates are required by the unraveling growth in the field of communication technology. This upgrowth has turned the focus towards modulation techniques which can meet the demands of spectrum efficiency with less Intersymbol Interference (ISI).

The development in the modulation technique projects M-ary QAM as one of the efficient digital modulation schemes because of its attractiveness to multiply the data rate for the given bandwidth. Though higher order modulations grant the boon of supporting higher data rates in the demanding field of radio communications, the price is paid in terms of SNR to achieve a tolerable Bit Error Rate (BER).

### Hardware Choice

FPGAs (Field Programmable Gate Arrays), due to their incredible configurability and flexibility, are playing a constantly increasing role in any digital communication environment. Indeed, the growth of these systems does not only claim for high speed hardware but also for a flexible, low-cost and standardised environment.

FPGA systems offer the possibility to easily test, modify and update the entire design and implementation. Towards the projected terabit/s communication in future applications, there are efforts made in exploiting FPGAs. More specifically, the entire system is implemented on a Xilinx Virtex 7 FPGA kit which is one of the most powerful available FPGAs. However, today, these devices are clocked below 1GHz and the improvement of performances is a big challenge on all abstraction layers, from the system architecture down to physical technology.

### Modulation Choice

QAM (Quadrature Amplitude Modulation) is widely used in many digital radio communications and data communications applications. Many forms of QAM are commonly used today for state-of-the art applications when high data rate are required. Indeed, QAM is a high order form of modulation able to carry many bits of information per symbol.

By selecting a higher order format of QAM, the data rate increases but, because of the constellation points getting closer and closer, noise can more easily lead to a misinterpretation of the symbol and the receiver may confuse two adjacent symbols. For this reason, this thesis considers different QAM systems, including 8-QAM, 16-QAM, 32-QAM and 64-QAM.

### Filter Choice

Filters are one of the most crucial step in the transmission of QAM signals since the band limitation of the transmitted signals and the Inter Symbol Interferences (ISI) absolutely need to be considered in any modern communication system.

Finite Impulse Response (FIR) filters can easily be designed to be linear phase and therefore delay the input signal but don't distort its phase. In that purpose, the Squared Root Raised Cosine (SRRC) is one of the most frequently used filter in digital communications thanks to its pulse shaping characteristic, its matched filter properties and its respect of the Nyquist criteria.

Optimizing and discussing the nature of the filter or the choice its parameters is left to related works [7, 8] and is not the purpose of this paper: the pursued goal is to speed up its implementation in such a way that the described filtering process can be generalized to any other kind of filters.

Due to the convolutional form of their impulse response, modern filters cannot be easily parallelized and constitute a significant speed limitation in digital communication technologies. This barrier can be overcome by implementing the filtering operation in frequency domain where the convolution operation becomes a simple multiplication. This fact is the main motivation of this Master Thesis.

### Thesis's Objectives

This Master Thesis focuses on the architecture level and aims at optimizing the performance of QAM modulators, exploiting the degree of parallelism of the underlying FPGA platform as well as mixed-domains (time and frequency) where beneficial.

A standard transmitter chain can be modelled as shown in Fig.1 here below. Indeed, the bit generator produces the input bit stream that is encrypted and encoded (for example, using a Viterbi encoder) to ensure data security and strength against noise. Next, the stream is clustered and furnished to the QAM symbol mapper in order to obtain an in-phase and quadrature component. The filter operation add extra strength against noise by limiting the transmission band and, finally, the signal is modulated at a given frequency and sent to the channel.

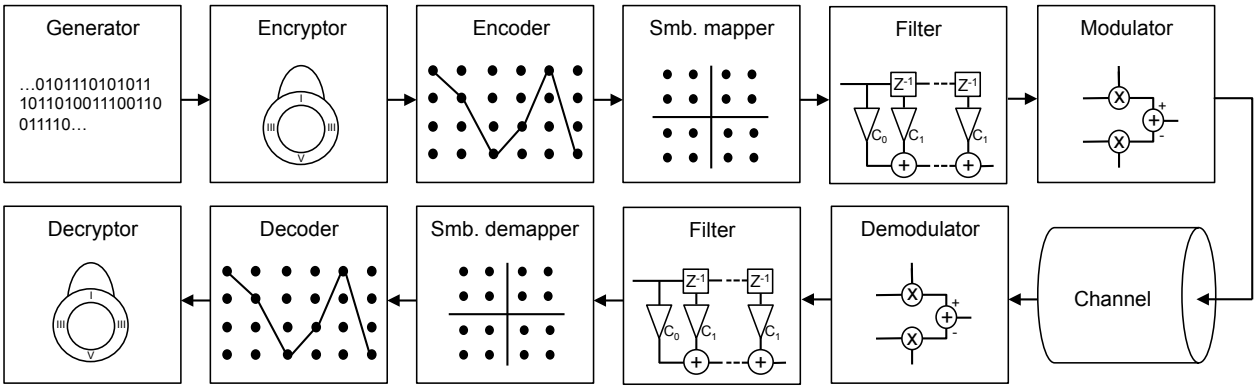



Figure 1: Standard Communication Chain

Subsequently, the receiver performs the inverse operations to retrieve the original message. Indeed, firstly, the demodulation retrieves the signal in baseband and secondly, the other part of the matched filter is applied to suppress the noise added on the undesired frequency locations and the inverse QAM mapper regenerates the bit stream from the in-phase and quadrature component. Finally, the error correcting code operation recovers the orignal message that it then decrypted.

Despite all the block component described above are essential to ensure a good quality communication, this work focusses only on the QAM symbol mapping, the filtering operation and the modulation of the transmitter. More specifically, a standard time-domain QAM transmitter can be modelled as shown in Fig.2.

The first block maps the input signals using a selected QAM constellation and outputs the corresponding real and imaginary QAM symbol.

Next, this complex signal is filtered and mixed with a carrier to achieve the modulation. Detailed explanations of this process are given in section 3.1.2.

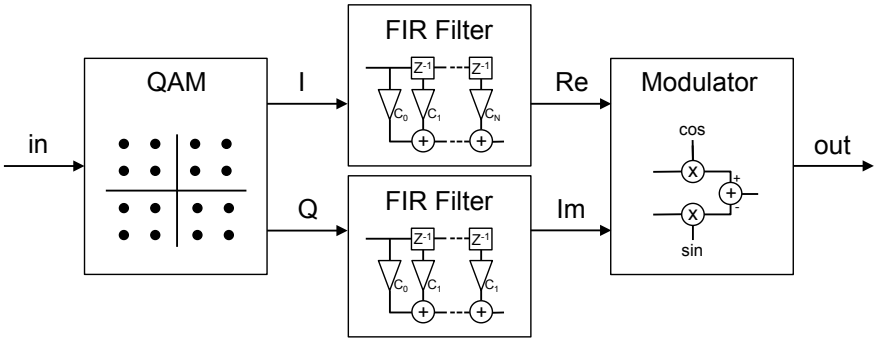



Figure 2: Standard QAM transmitter

As discussed in section 3.3.1, this standard implementation has the disadvantage to require the filter input sequentially due to the convolution nature of the filtering process. For that reason, the filtering operation is the main focus of this work. In order to overcome this inconvenient and process parallel data, the following mixed-domain structure shown in Fig.3 is considered and explained in section 3.

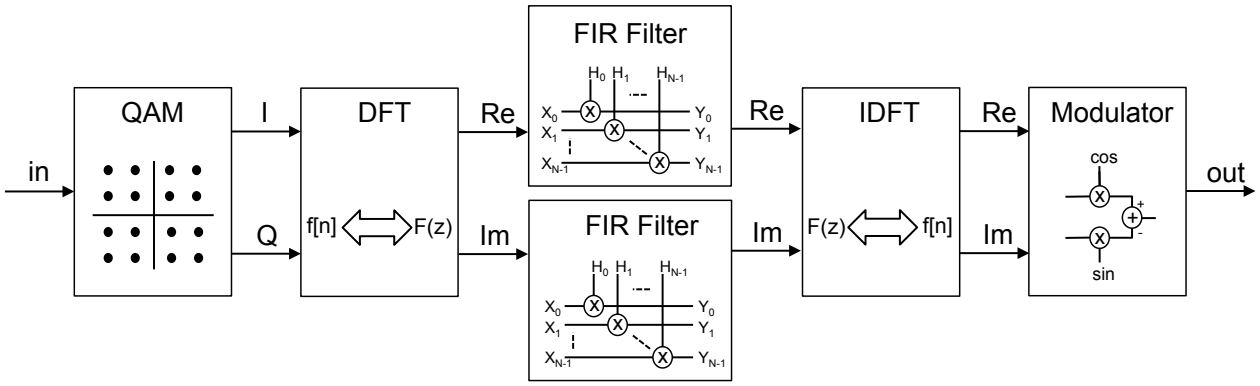



Figure 3: Mixed-domain QAM transmitter

This paper starts describing the current state-of-art QAM communication technologies. Then, the fundamental concepts for a good understanding of this thesis are derived and each block of the diagram illustrated in Fig.3 is discussed in details. The next section focusses on the theoretical aspects and on the methodology used to implement the modulator. Finally, this report ends by illustrating a hardware implementation and the experimental results of the realised QAM transmitter.

# 2 Today's State-of-the Art

This section is devoted to the current state-of-the-art in the target domain. More specifically, other works in the same field as this paper are investigated in order to high-line the today's top technology. Fig.4 displays a qualitative chart summarising the research.

In 2003, Yongbin Wu and Yousef R. Shayan from the Concordia University (Canada) developed an implementation of high-speed transceiver Quadrature Amplitude Modulation (QAM) using a Xilinx Virtex II Field Programmable Gate Array (FPGA). They targeted only the 64-QAM format but they embedded the mixers inside the FPGAs board instead of relying on analog electronic for this process. Moreover, the filter selected for their implementation is a FIR filter, exactly as the one implemented in this project. Although their technology is pretty old compared to today's FPGAs, they could reach a top frequency of 55 MHz [1].

Seven years later, in 2010, three researchers from the University of Hanoi (Vietnam), Xuan-Thang Vu, Nguyen Anh Duc and Trinh Anh Vu, built a similar system and published a complete 16-QAM scheme implemented on two Xilinx Virtex IV FPGA board, one for the receiver and one for the transmitter. Their implementation was realised using the Xilinx System Generator (Sysgen) and they achieved a top frequency of 111.11 MHz [2]. This very same year, Vadim Smolyakov, Dimpesh Patel, Mahdi Shabany and P. Glenn Gulak developed a 64-QAM receiver based on a Xilinx Virtex V FPGA kit operating at a maximum frequency of 125 MHz [3].

In 2012, those results were improved by Siqiang Ma and Yong'en Chen from the University of Shanghai. They built a modular QAM transmitter working with 16-QAM, 32-Q1M, 64-QAM, 128-QAM or 256-QAM at 128.6 MHz. Their implementation was based on a Xilinx Virtex IV kit and they also used a FIR filter [4].

The next year, a collaboration between the University of Paderborn (Germany) and the Bielefeld University (Germany) published 16-QAM based transceiver working at 625 MHz on a Xilinx Virtex VI board. However, their very high clock frequency comes at the price of a low precision. Indeed, they worked on a 6-bit data bus due to the limitation imposed by their Digital to Analog Converter (DAC) [5].

The last presented state-of-the-art result was made this year by three engineers of the company *E2v Semiconductors* (based in U.K.). Exploiting the powerful Xilinx

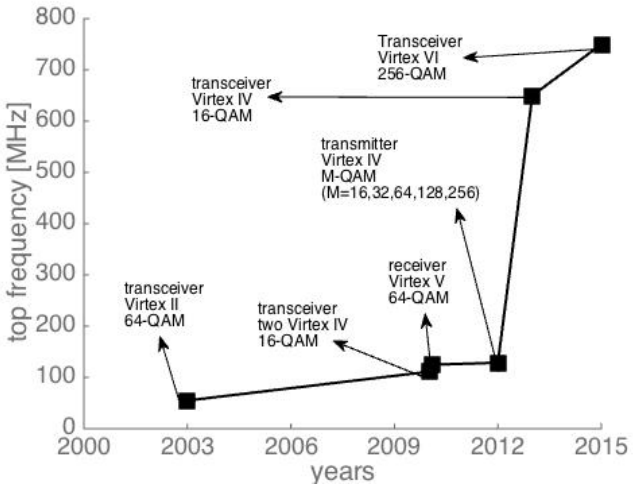



Figure 4: State-of-the-art summary chart

Virtex VI FPGA, they built a 256-QAM transceiver operating at 750MHz. Nevertheless, this impressive result is attenuated by the fact that their system doesn't comprise a filter [6].

As a final note, it has to be mentioned that much better results have been obtained with ASIC technologies or multi-FPGA systems. However, those systems are not considered because this work focuses on single FPGA implementations.

# 3 Fundamentals

As shown in Fig.3 of section 1, the implemented system comprises a QAM symbol mapper, a Fourier transformer to send the signal in the frequency domain, a SRRC filter, an indirect Fourier transform block to retrieve the signal in time domain and, finally, a modulator. In the following subsections, the fundamentals of this system are inspected block by block.

## 3.1 Fundamentals - QAM Modulation

Quadrature Amplitude Modulation (QAM) allows to send two modulated signals into a single channel. Here below, Fig.5 illustrates the QAM modulation processing blocks.

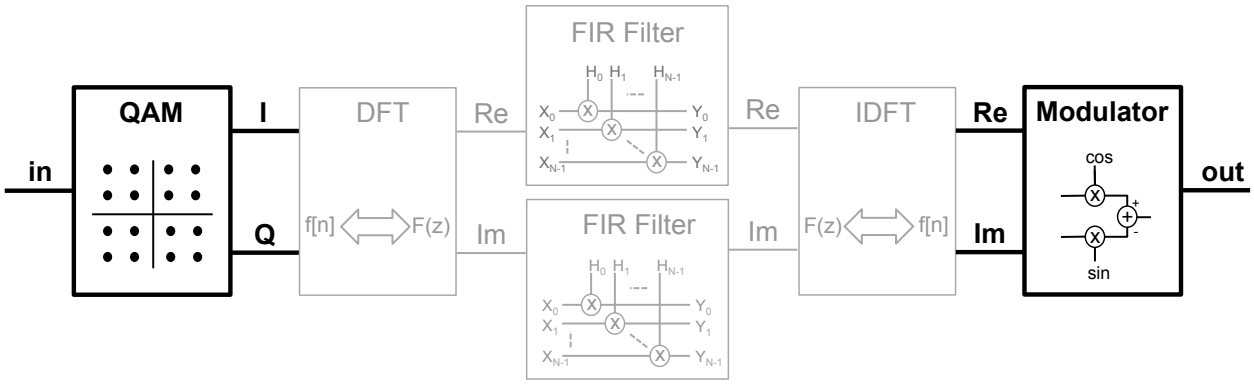



Figure 5: Mixed-domain QAM transmitter - QAM modulation

The targeted mixed-domain QAM transmitter uses two blocks in order to achieve QAM modulation, one at the beginning of the processing chain and one at the end. In this section, the appellation *QAM mapper* and *modulator* designate the former and the latter block, respectively.

### 3.1.1 QAM Modulation - QAM Mapping

As in many modulation schemes, QAM can efficiently be represented using a constellation diagram. The constellation diagrams show the different positions for the states within different forms of QAM.

Many of those diagrams are possibles but this thesis implements the standard rectangular constellation in which the points are arranged in a square grid with equal vertical and horizontal spacing. In addition to require less overhead implementation, the rectangular constellation is the simplest and therefore, it is the only one considered by this paper.

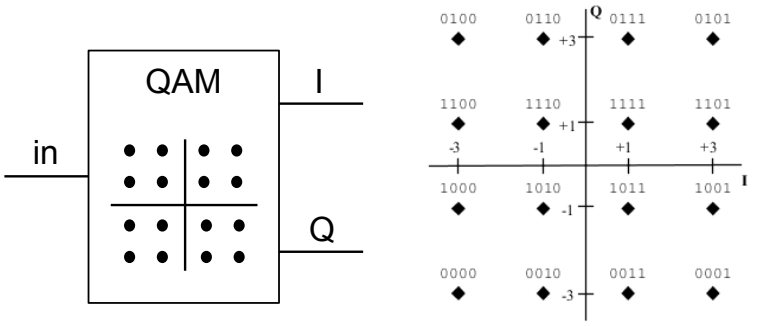
#### QAM Formats

Multiple forms of QAM are possible and some of the more common forms include 16-QAM, 32-QAM, 64-QAM, 128-QAM, and 256-QAM [9]. More generally, we call these forms *M-QAM* formats where $M$ denotes the number of points on the constellation, i.e. the number of distinct states that can

exist. Each symbol of the M-QAM constellation contains exactly $log_2(M)$ bits and, by consequences, the first operation performed on the input stream is its clusterization into the $log_2(M)$ bits and its mapping into the constellation.

Fig.6 shows the QAM mapping block of the mixed-domain QAM modulator introduced in section 1 (Fig.6a) together with a 16-QAM constellation (Fig.6b).

Among the many available QAM format orders, the 16-QAM is preferred due to its relatively low dense constellation, which is therefore the default transmitter operating mode. Indeed, the faster data rates and higher levels of spectral efficiency offered by higher order format come at the price of a lower resistance to noise and higher Inter Symbol Interferences (ISI).

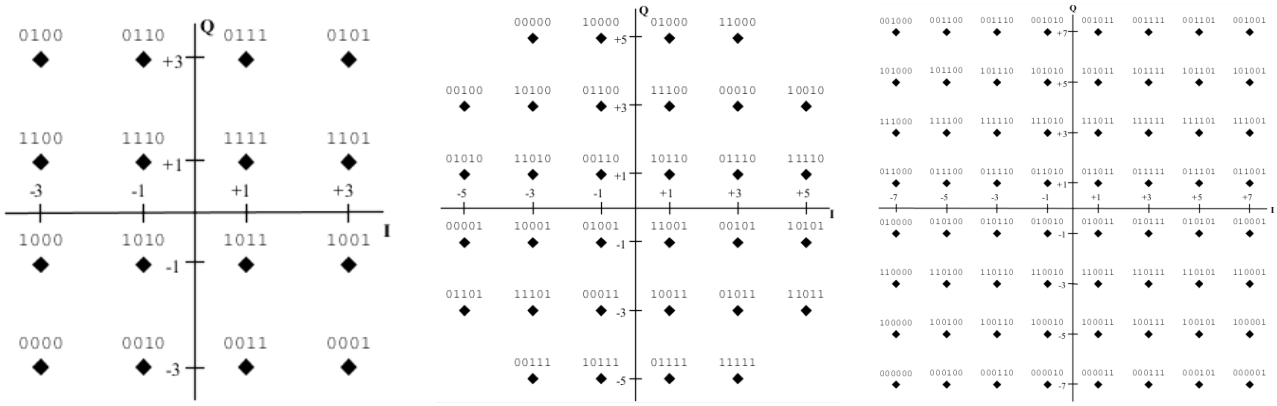



(a) QAM mapper block (b) 16-QAM constellation [10]

Figure 6: QAM mapping

This is due to the fact that the order of the modulation increases with the number of points on the QAM constellation diagram and therefore, the receiver may fail more likely to distinguish them at the reception.

In order to achieve the highest level of modularity, the implement system support 8-QAM, 16-QAM, 32-QAM and 64-QAM. Moreover, the discussion can be easily extended to higher QAM orders. Here below, Fig.7 illustrates the constellation scheme used for the other supported formats.




(a) 8-QAM constellation (b) 32-QAM constellation (c) 64-QAM constellation

Figure 7: Supported QAM constellations [10]

**Gray Code**

There are many ways to associate a symbol (composed of 4 bits, in the case of 16-QAM) to a given constellation position. Certainly, the gray code is the most common choice [11] .

Gray code aims to code the symbols in such a way that every symbol will have exactly one bit difference with each of his neighbours. This allow to reduce the erroneous symbol decision to one bit error and, therefore, improve the SNR of the system. For easy reference, the 16-QAM constellation of Fig. 6b as well as the constellations shown in Fig.7 display gray coded symbols.

**Output Assignment**

Once the input stream is clustered in the $log_2(M)$ bits and mapped to the constellation, two outputs are produced, respectively the In-phase (I) and the Quadrature (Q) components (see Fig.6a).

This output assignment can be done by hardcoding each symbol to the corresponding $I$ and $Q$ value. For example, observing the Fig.7b, the corresponding In-phase and Quadrature components of the 32-QAM symbol [01101] are $I = -5d$ and $Q = -3d$.

Nevertheless, for QAM formats possessing an even number of bits $log_2(M)$ per symbol, the following cleaner mapping technique is applicable.

Let's consider a 4-bit cluster [$b_3$ $b_2$ $b_1$ $b_0$]; i.e., for the 16-QAM modulation, the mapping is defined in the following way:

$$[\ \underbrace{b_3}_{I_1}\ \ \underbrace{b_2}_{Q_1}\ \ \underbrace{b_1}_{I_2}\ \ \underbrace{b_0}_{I_2}\ ] \quad \Rightarrow \quad I = \{I_1, I_2\} \quad \text{and} \quad \mathrm{Q} = \{\mathrm{Q}_1, \mathrm{Q}_2\}$$

In other words, if we consider the following example vector [1011], the in-phase and quadrature components are $I = \{1, 1\}$ and $Q = \{0, 1\}$, respectively.

Finally, a finite value is associated to each possible I and Q as follows:

$$\begin{aligned} 00 &\Rightarrow d \\ 01 &\Rightarrow 3d \\ 10 &\Rightarrow -d \\ 11 &\Rightarrow -3d \end{aligned}$$

where $3d$ is the highest filter value (see section 3.3.1). This discussion can be easily extended to all other order QAM formats possessing an even number of bits per symbol and is exploited in the implication of the system in section 5.2.

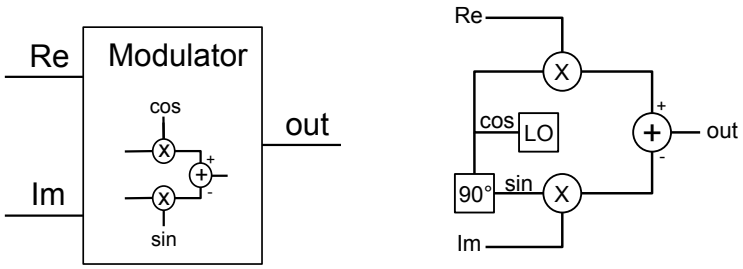
### 3.1.2 QAM Modulation - Modulator

The modulated wave is output by the modulator block shown in Fig.8a at the end of the transmitter.

First, the $I$ input[1] is multiplied by the cosine function generated at a fixed frequency $f_0$ by a local oscillator. Similarly, the $Q$ component is multiplied by a 90° shifted LO signal, which result in a sine wave.

The purpose of this operation is to obtain two orthogonal carriers. Finally, those two terms are subtracted to result in the output of the transmitter. This procedure is illustrated by the diagram depicted in Fig.8b.

In order to avoid aliasing, the carriers have to resampled at leat at twice the carrier's frequency.




(a) Modulator block (b) Modulator scheme

Figure 8: Modulator

Eq.(1) describes mathematically the operation performed by the modulator.

$$\begin{aligned} out(t) &= \mathcal{R}\left\{[I(t) + iQ(t)]e^{2\pi f_0 t}\right\} \\ &= I(t)\cos(2\pi f_0 t) - Q(t)\sin(2\pi f_0 t) \end{aligned} \tag{1}$$

At the receiver, the modulated signals can be demodulated using a coherent demodulator but the demodulation operation won't be discussed in this section since the receiver is not part of this work. Interested readers can find additional information concerning QAM demodulation in reference [12].

### 3.2 Fundamentals - Fourier Transform

The fourier transform is the key element of this mixed-domain modulator. More specifically, the $I$ and $Q$ components generated by the QAM mapping block are transferred in the Fourier domain, where they are filtered. After the filtering operation, the components are taken back in the time domain.

The next two subsections are dedicated to the explanation of these blocks. First, the underlying theoretical concepts necessary to understand the Fourier domain and the transfer between this domain and the time domain are set. Next, the main property making this transform an useful asset for filtering operations is illustrated in details.

Here below, Fig.9 shows the two blocks performing this operation. The first, called *DFT*, sends the signals to the Fourier domain, while the second, named *IDFT*, take them back.

[1] Note that for clarity the $I$ and $Q$ components are sometimes represented on figures by the *Re* and *Im* symbols, respectively.

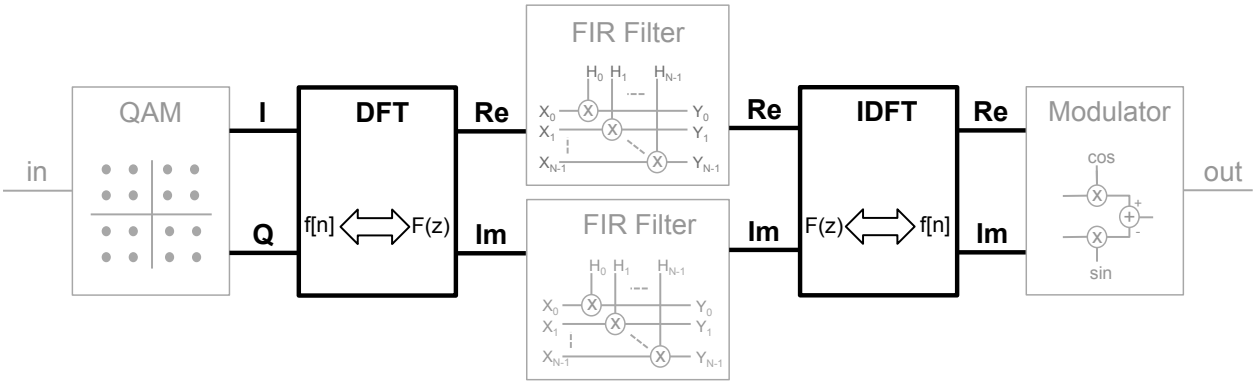



Figure 9: Mixed-domain QAM transmitter - DFT and IDFT Blocks

### 3.2.1 Fourier Transform - Theoretical Concepts

Fourier transform's theory can be widely discussed and interpreted but only the fundamental concepts necessary to a good understanding of the implemented system will be summarized in this section.

#### Time and Frequency Domain

The Fourier transform decomposes a signal into the multiples frequencies that make it up, an alternative representation made of sines and cosines. Indeed, Fourier's theory shows that any waveform, no matter what it describes in the universe, is just the sum of simple sinusoids of different frequencies. Therefore, the Fourier Transform (FT) is the mathematical tool that deconstructs the signal into its sinusoidal components and, similarly, the Inverse Fourier Transform (IFT) is the tool to reverse it [13].

The apellation *time-domain* graph designates the signal's changes over time. In contrast, after applying the Fourier transform, the signal is lying in the Fourier domain and is represented by a *frequency-domain* graph.

Mathematically, for a given time-domain signal $f(t)$, it's corresponding Fourier representation in the frequency domain is given by $F(w)$ in Eq.2 here below:

$$F(w) = \int_{-\infty}^{\infty} f(t)e^{-wit}\,dt \tag{2}$$

where $w$ denotes the signal pulsation. Correspondingly, the inverse Fourier transform to take back the signal $F(w)$ from the frequency domain into the time-domain signal $f(t)$ is derived as follows:

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(w)e^{wit}\,dw \tag{3}$$

**Discrete Fourier Transform (DFT)**

When a continuous signal is not available and a finit list of equally spaced samples of a signal have to be considered, the Discrete Fourier Transform (DFT) is used instead of the Fourier Transform described above. More specifically, the discrete Fourier transform converts the discrete signal into the list of coefficients of a finite combination of complex sinusoids, ordered by their frequencies, that have those same sample values. Eqs.(4) and (5) define the DFT equivalently to Eqs.(2) and (3).

$$X[k] = \sum_0^{N-1} x[n] e^{-2\pi i k n/N} \qquad k \in \mathcal{Z} \tag{4}$$

$$x[n] = \frac{1}{N} \sum_0^{N-1} X[k] e^{2\pi i k n/N} \qquad n \in \mathcal{Z} \tag{5}$$

Despite the fact that performing the DFT for a large number of inputs $N$ requires numerous operations, Appendix A briefly explains an algorithm, called FFT, to perform it efficiently.

### 3.2.2 Fourier Transform - Convolution Property

The interest of this concept is the observation that linear operations performed in one domain (time or frequency) have corresponding operations in the other domain, that are sometimes easier to perform. More specifically, the convolution operation in time domain, becomes a simple multiplication in frequency domain. Therefore, denoting $\mathcal{F}$ the Fourier transform operation and, considering two given time-domain function $f$ and $g$, the following equivalences hold:

$$\begin{aligned} \mathcal{F}\{f \circledast g\} &= \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \\ &= G \cdot F \end{aligned} \tag{6}$$

where the operators $'\circledast'$ and $'\cdot'$ denote the convolution and multiplication, respectively. Then, applying the inverse Fourier transform $\mathcal{F}^{-1}$ on both side of Eq.(6), we obtain:

$$\begin{aligned} f \circledast g &= \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\} \\ &= \mathcal{F}^{-1}\{G \cdot F\} \end{aligned} \tag{7}$$

## 3.3 Fundamentals - Filter

Since more and more channels and high data rates are required for today's communications, the channel's bandwidth has to be narrowed down but this causes the current symbol to overlapp with the previous one, which create ISI. In that purpose, a filter respecting the Nyquist criteria (and then ensuring zero ISI) is essential.

Therefore, this last part of the *Fundamentals* section describes the filtering operation, which is the most delicate operation of this transmitter and the reason of the mixed-domain utilisation.

Many different sorts of filters are available but Finite Impulse Response (FIR) filters are preferred in this work. Indeed, FIR filters can easily be designed to be linear phase and therefore delay the input signal but don't distort its phase.

The next subsection is dedicated to the theoretical explanation of this kind of filters. Subsequently, this section ends explaining the filter chosen for the described QAM transmitter: the Squared Raised Root Cosine (SRRC) filter. As usual, Fig.10 here below high-line the blocks targeted in the following discussion.

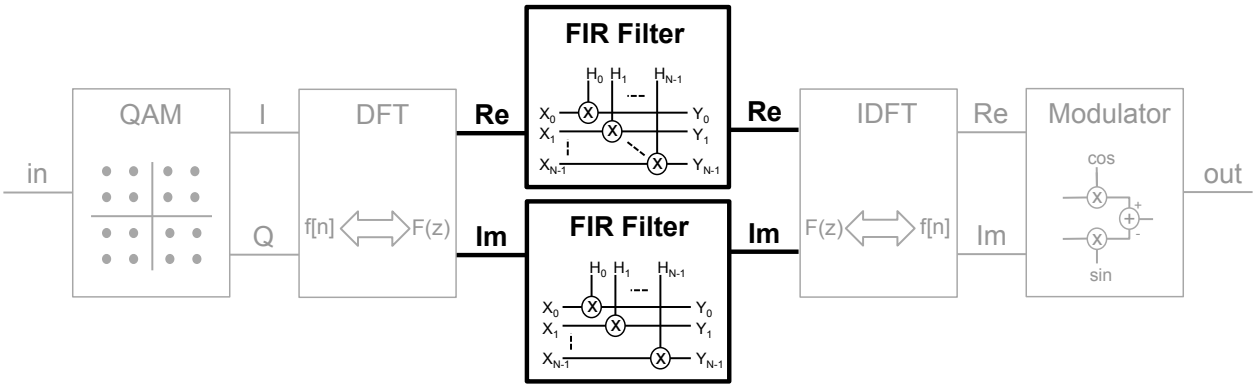



Figure 10: Mixed-domain QAM transmitter - Filter

### 3.3.1 Filter - Finite Impulse Response Filters

The appellation *Finite Impulse Response Filter* comes from the fact that the impulse response of this kind of filters reaches zero in a finite amount of time, contrarily to the other class of filters called infinite Impulse Response Filters (IIR).

Fig.11 shows a standard FIR filter implementation diagram. First, the input is multiplied by the first filter coefficient $c_0$. Secondly, the signal is delayed and multiplied with the next coefficient. This second procedure is repeated $(N-1)$ times (where $N$ is the filter's order). Finally, all the multiplication results are summed up to produce the output signal.

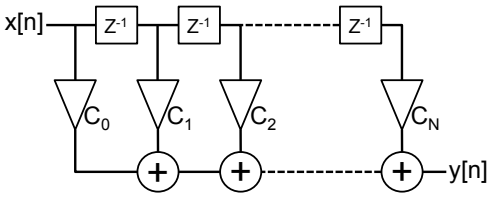



Figure 11: FIR filter diagram

The design of the FIR filter implemented in this work is done by finding the coefficients and filter order that meet certain specifications but this won't be discussed in this paper. Indeed, the filters parameters comes directly from [7].

#### Mathematical Description

As said above, the impulse response $h[n]$ of FIR filters have non zero values only on a given time duration and can therefore be modelled as follows:

$$h[n] = \sum_{i=0}^{N} c[i] \cdot \delta[n-i] = \begin{cases} c_i & \text{for} \quad 0 < i < N \\ 0 & \text{for} \quad otherwise \end{cases} \tag{8}$$

Therefore, a FIR filters of order $N$ is mathematically described as in Eq.(9) here below.

$$\begin{aligned} y[n] &= \sum_{i=-\infty}^{\infty} h[i] \cdot x[n-i] \\ &= \sum_{i=0}^{N} c[i] \cdot x[n-i] \\ &= (x \circledast c)[n] \end{aligned} \tag{9}$$

where $x[n]$, $y[n]$ and $c_i$ (with $i \in [0, N]$) are respectively the input, the output and the filter's coefficients.

### Properties

FIR filters are preferred in this work over the other filter class (the IIR filters) because of their numerous advantages.

First of all, in the case of a FIR filter, the same rounding error appears in every iteration because of the absence of feedback. Therefore, the total error doesn't sum up over each cycle. Secondly, the output of a FIR filter is a sum of a finite number of finite multiples of the input and by consequences, cannot be grater than a fixed multiple of the input value. This ensures the filter stability. And, finally, the last advantageous property of FIR filters is their ability to be designed with linear phase and therefore, they delay the input signal but don't distort its phase.

For completeness, it has to me mentioned that the main drawback of FIR filters respect to IIR filters is the considerable amount of computation power required to realized a FIR filter with similar characteristics to an IIR.

### Fourier Analysis

The filtering operation in time-domain is given below,

$$y[n] = x[n] \circledast h[n] \tag{10}$$

Accordingly to Eq.(7) of section 3.2.1, this operation can be transposed in the frequency domain using the convolution-multiplication symmetry and simply becomes:

$$Y[k] = X[k] \cdot H[k] \tag{11}$$

where $X$, $Y$ and $H$ are the Fourier transform of the input, output and filter's impulse response, respectively. Additionally, observing Eq.(8), we can easily derive the impulse response $H$ of a FIR filter of order $N$ as follow:

$$H[k] = c[k] \cdot \delta[k] = \begin{cases} c_k & \text{for} \quad 0 < k < N \\ 0 & \text{for} \quad otherwise \end{cases} \tag{12}$$

The major interest of a mixed domain transmitter is now evident. Implementing Eq.(11) is much simpler than Eq.(10) and, most importantly, Eq.(11) is parallelizable, while the other is not.

### 3.3.2 Filter - Squared Raised Root Cosine Filter

Matched filters are the optimal linear filter that maximize the SNR in the presence of additive stochastic noise. It works by correlating the received signal with a known template (the expected version of the received signal). Therefore, this kind of filters is used to detect an expected signal and distinguish it from background noises, which is exactly our objective.

More specifically, the chosen matched filter implemented in this paper is the Squared Raised Root Cosine (SRRC) Filter because it is a good compromise between high spectral efficiency and low ISI. Its main goal is to limit the transmitted signal into a defined part of the channel (pulse shaping) in order to prevent interferences with adjacent channels.

The filter is design to achieve a fast decay of sidelobes in the impulse response, narrow transition band, great minimum stopband attenuation, efficient bandwidth utilization and low cost. Unfortunately, improving one of these characteristics will degrade the other one. The specific design of the filter is then realized by balancing the above features by properly choosing the following parameters: oversampling factor, roll-off factor, truncation length [7].

The filter parameters of the selected SRRC filters are directly taken from [7] and are summarized in the table here below. As said in section 3.1.1, the QAM mapping output value $3d$ equals $c_5 = 0.54098593171027443$.

| | | | | | |
|---|---|---|---|---|---|
| $c_0$ | = | 0.022507907903927645 | $c_6$ | = | 0.3076724792547561 |
| $c_1$ | = | 0.028298439380057477 | $c_7$ | = | -0.037500771921555154 |
| $c_2$ | = | -0.076801948979409798 | $c_8$ | = | -0.076801948979409798 |
| $c_3$ | = | -0.037500771921555154 | $c_9$ | = | 0.028298439380057477 |
| $c_4$ | = | 0.3076724792547561 | $c_{10}$ | = | 0.022507907903927645 |
| $c_5$ | = | 0.54098593171027443 | | | |

Figure 12: SRRC Filter's coefficients

# 4 Concepts and Methodology

This section explains the behaviour of the realised transmitter as well as the methodology applied in this work.

First, the strategy applied during the whole project is summarised in a few main steps. Then, the conception of every block constituting the system is explained one by one and finally, an overview of the project's simulation is described.

## 4.1 Concepts and Methodology - Design Strategy

The design strategy adopted during this project can be separated in two main steps. The first step in the conception of this work was the construction of a single channel (without any parallelisation) mixed-domain modulator. In the following sections, the term *simple transmitter* refers to such design. The second main step was the migration to a parallel modulator.

Both of these main steps are divided in a conceptual reference MATLAB simulation phase, a hardware system building phase and an optimization phase (see Fig.13).

- *MATLAB model*: In order to deeply understand the behaviour of the system, a complete MATLAB model has been developed. The first purpose of this model was to give an idea of the expected behaviour of the system. The second purpose was to compare the hardware results with the reference MATLAB model to prove its functionality.

  Each block constituting the system was first realised in MATLAB using the physical and mathematical fundamentals explained in section 3.

1. Simple Transmitter
   1. MATLAB model
   2. Hardware implementation
   3. Optimization
2. Parallel Transmitter
   1. MATLAB model
   2. Hardware implementation
   3. Optimization

Figure 13: Methodology

- *Hardware implementation*: The second step is to transpose the MATLAB model into a working hardware system described in Verilog. The good functioning and the precision of this system could be easily tested by comparing the output of each block with the MATLAB model[2].
- *Optimization*: The third step is the optimisation of the hardware model by analysing the logic requirements as well as the IP core configurations.

The focus of this system is to implement the filter in the frequency domain. In addition, since the parallel transmitter requires the filter coefficients, the DFT coefficients and the carriers to be entered by the user through a file, the Java program *FQM Utility* has been developed in order to auto-generate those files. This program is explained in section 5.6.

[2]The MATLAB model is considered as perfectly precise. In other words, all MATLAB internal rounding errors are ignored.

## 4.2 Concepts and Methodology - Conceptual Model

The conceptual design is explained in this section through the reference MATLAB model since it defines the ideal and expected system's behaviour while the next section is entirely dedicated to the hardware implemented system.

The realised single channel MATLAB model can be depicted as shown in Fig.14. This section aims to explain this conceptual model and to show its results[3].

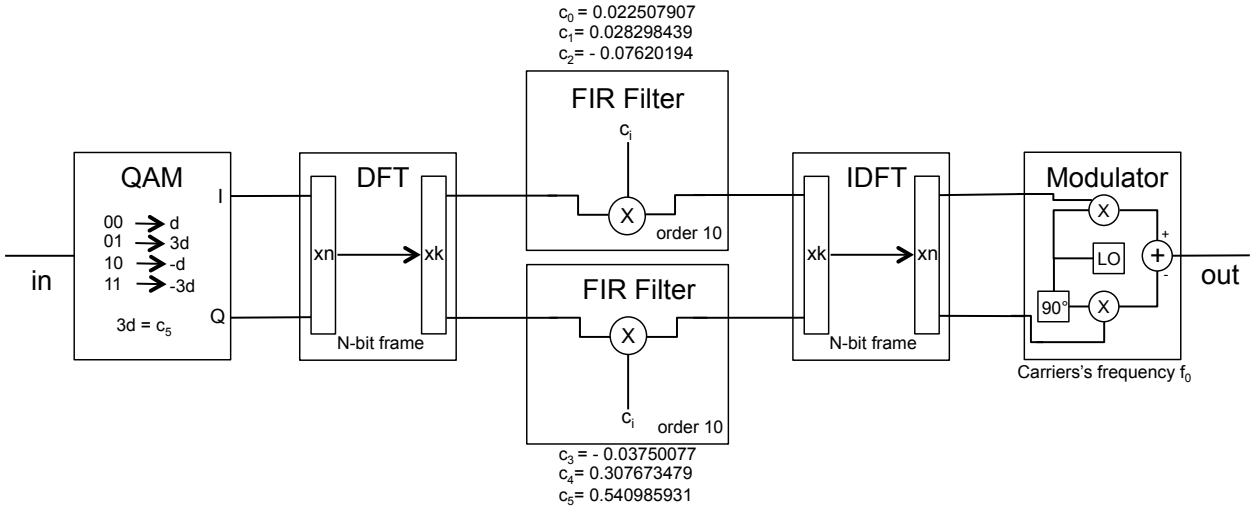



Figure 14: Mixed-domain QAM transmitter - Conceptual model

The input stream goes first through the QAM block in order to generate $I$ and $Q$ components following a selected QAM mapping scheme. This step is not simulated by the conceptual model because of its simplicity. Indeed, the MATLAB model receives as input randomly generated $I$ and $Q$ values.

In addition to these two values, the system also requires the filter's coefficients $c_i$, that are furnished through a file[4] to the program. Depending on the number of filter's coefficients $n$, the DFT size has to be computed as the minimum power of two above $n$. Event though the DFT does not require the transform length to be a power of two, the FFT does (see appendix A) and, since this project has been build to be used with FFT cores (as explained in sections 5.6 and 7), this additional design constraint has been imposed. The DFT coefficients have also to be generated and provided through a file. The two last model's requirement are the sine and cosine carriers values. Those functions could certainly be easily generated by MATLAB but importing them as it is done by the FPGA design massively simplifies the synchronisation between the conceptual model and the hardware implementation. Further explanations are given in section 5.

[3] It is to note that rescaling operations are executed at the output of some block but won't be discussed in this section since it does not affect the system's behaviour. See section 5 for further explanations.

[4] As previously mentioned, the filter's coefficient file, the DFT's coefficient file and the carriers file are auto-generated by the Java program *FQM Utility*.

The next paragraphs illustrate in details the behaviour of the DFT, FIR Filter and IDFT blocks, respectively. Finally, the last paragraph summarises the model by displaying all the signals appearing in each data bus in order to give to the reader a general view on the system.

### Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) block receives as input the $I$ and $Q$ values mapped by the QAM block and outputs the result of their Fourier Transform. In this section, since the system has been simulated with an order 10 filter, the DFT size is the minimum power of two above 11, that is $2^4 = 16$.

Fig.15 shows eighty samples of five different waves. The two waves on the left are the $I$ and $Q$ input values and the two on the right are the real and imaginary DFT's output.

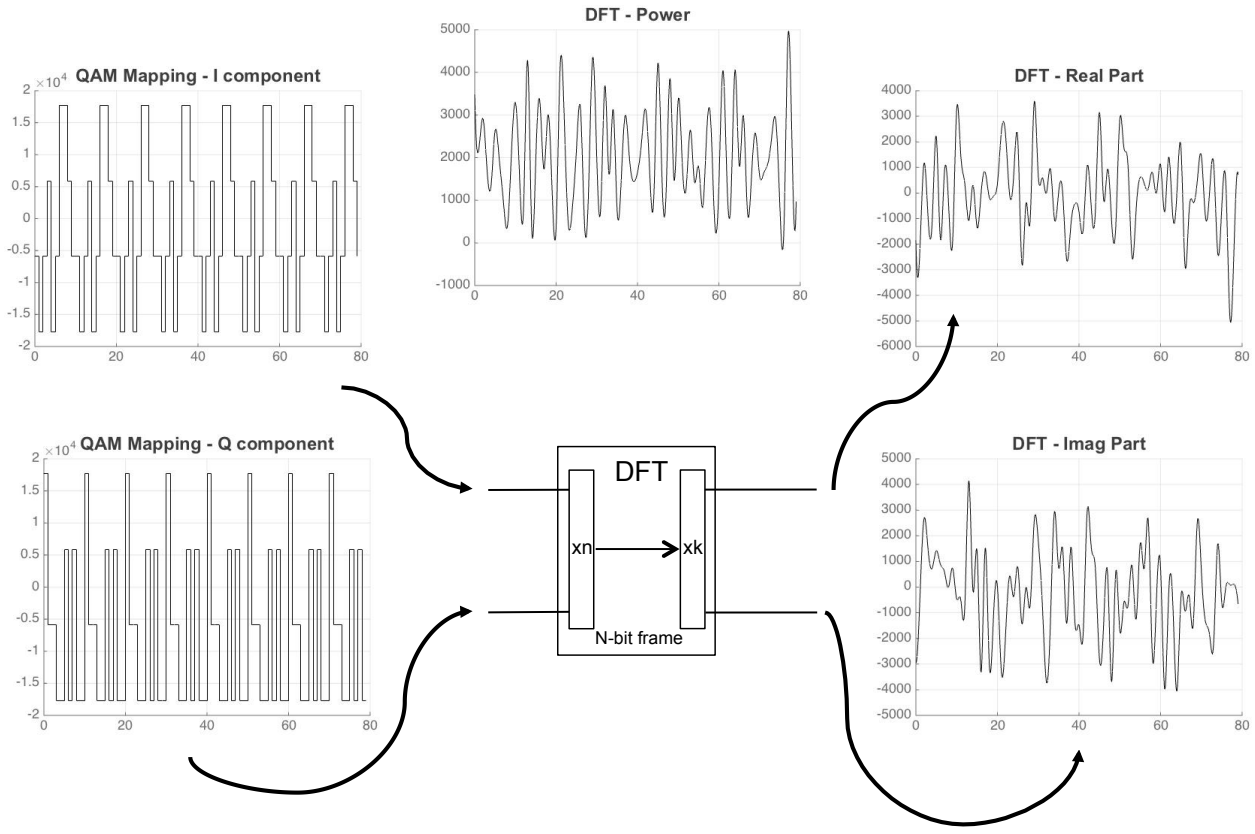



Figure 15: Discrete Fourier Transform (DFT) block - Conceptual model

The fifth graph (in the upper middle of the figure) shows the DFT power output computed as

$$power = \sqrt{xk_{re}^2 + xk_{im}^2} \tag{13}$$

where $xk_{re}$ and $xk_{im}$ are respectively the DFT's real and imaginary output.

**Squared Root Raised Cosine (SRRC) Filter**

This second paragraph illustrates the Squared Root Raised Cosine (SRRC) filter behaviour similarly to the previous paragraph. As mentioned above, the filter's coefficients $c_i$ are also taken as input constants and are therefore not displayed as wave signal in Fig.16.

Indeed, the reference filter's coefficients considered for this simulation are give in Tab.12 of section 3.3.2 and are displayed on the top and bottom of the FIR filter blocks of Fig.14 (note that the filter's coefficients are symmetric and therefore only the first six coefficients are shown in the figure). Nevertheless, in the pursuit of modularity, the system is design to implement any order of any type of FIR filter (see section 5.6 for further explanations).

The filter coefficients have to be padded with zeros in order to have as many coefficients as DFT outputs. In this case, since the DFT size is 16 and we have 11 filter coefficients, 5 zeros have to be added before filtering.

Again, the two waves on the left of Fig.16 are the inputs of the target block and the two signals on right are its outputs. It is to notice that the filter's inputs are exactly the DFT's output described in the previous paragraph. Finally, a formula similar to Eq.(13) is used to compute the filter output power which is displayed on the middle of the figure.

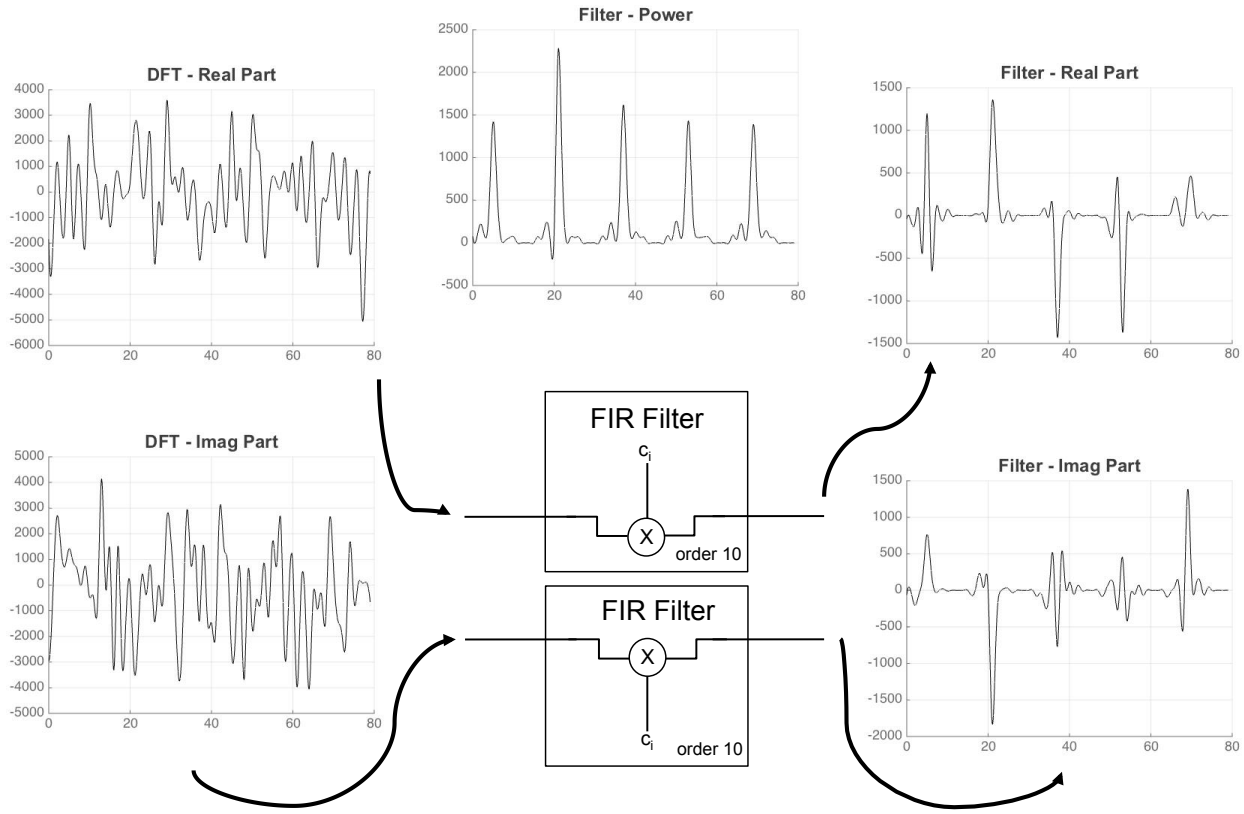



Figure 16: Squared Root Raised Cosine (SRRC) filter block - Conceptual model

The filter's power wave shows a typical pulse shaping behaviour. Indeed, all the signal is concentrated in a portion of the spectrum. We can also observe the relative abrupt transition and the low side lobes of the chosen SRRC filter.

A similar note can be made when observing the filter's output respect to its input. Remembering that the input waves are in the frequency domain, it can be noticed that only some frequencies are kept (where the peak are localised) and all the other are attenuated or suppressed.

### Inverse Discrete Fourier Transform (IDFT)

The Inverse Discrete Fourier Transform (IDFT) block operates similarly to the DFT block described above. Its inputs are the filter's output and the five graphs on Fig.17 are the filter's inputs, the IDFT power, the real IDFT output and the imaginary IDFT output. As a final note, the IDFT's outputs look familiar due to its multiple oscillations recalling a time-domain behaviour.

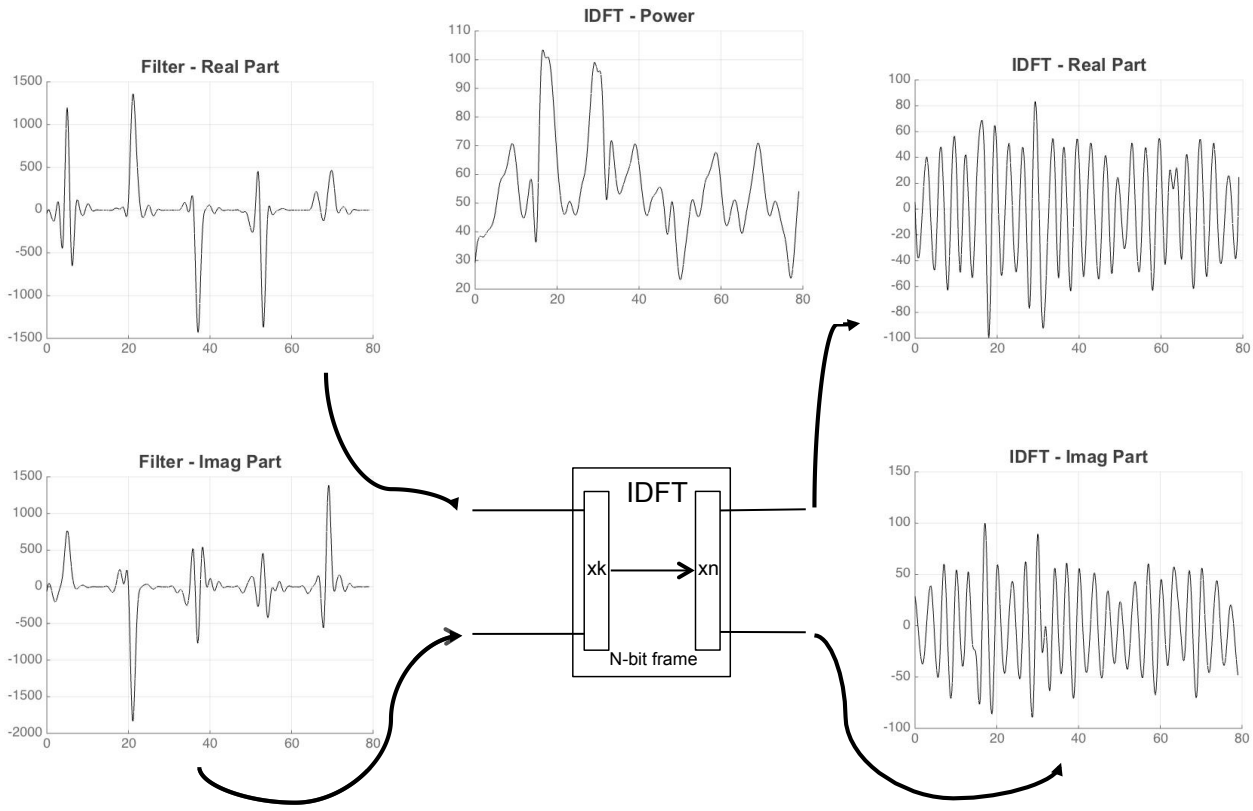



Figure 17: Inverse Fast Fourier Transform (IDFT) block - Conceptual model

The next step is to modulate those waves by mixing them with cosinusoïdal carriers in order to be able to transmit them centred around a given frequency $f_0$. The next paragraph shows the result of such operation.

**Summary**

This last paragraph of the *Concepts and Methodology* section aims to summarise the desired behaviour of the simple mixed-domain transmitter by displaying a typical wave signal for each data bus.

First, a bit stream is entered into the system and the QAM mapping block generates the $I$ and $Q$ signal[5].Secondly, the samples are sent into the frequency domain by the DFT block. Then, the filtering operation is applied and the signal is taken back in the time domain by the IDFT block.

Finally, the modulator block receives the time-domain real and imaginary samples and performs the operation described by Eq.(1) of section 3.1.2. As mentioned before, the sine and cosine waves are actually consider as inputs in the MATLAB file but Fig.18 still displays a $LO$ inside the modulator to high-line the system behaviour.

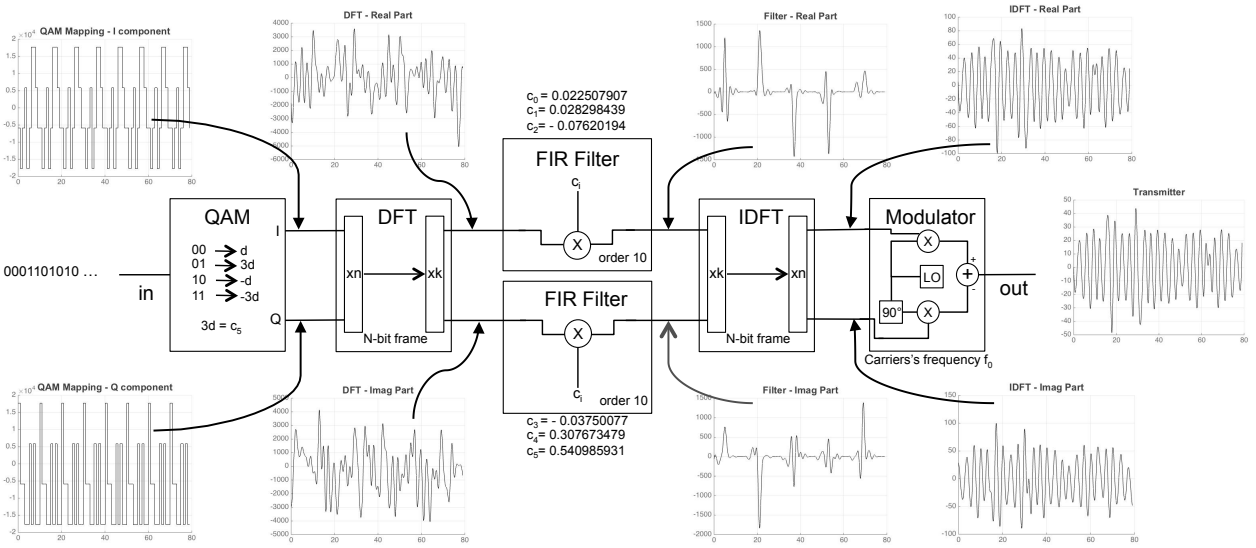



Figure 18: Transmitter - Conceptual model

Observing the output of the transmitter on the right most graph, we notice its familiarity with a standard time-domain modulated wave.

[5]As mentioned above, this block is not simulated in MATLAB due to its simplicity but is still depicted in Fig.18 to overview the behaviour of the complete system.

# 5 Implementation

This section is entirely dedicated to the hardware implementation of the QAM transmitter. This section starts by stating the general design pattern and then explains as clearly as possible all the implemented Verilog module. The first explains the implementation of the QAM mapper while the second is dedicated to the DFT and IDFT modules. The third illustrates the modulator and finally, the last part of this section explain the transmitter module; i.e. the top module coordinating all the others.

More specifically, the illustration of each part is divided into four paragraphs. First, a table illustrates the block's parameters, its input and output ports, as well as the IP cores and dependencies required by the module. The second paragraphs explains in details the module's implementation and the third focuses on some major aspects of the Verilog code. Finally, the last section is dedicated to the IP core configuration and block's dependencies.

## 5.1 Implementation - Design Pattern

The implementation of the parallel transmitter is entirely done by using handwritten Verilog code, Xilinx IP cores and auto-generated Verilog modules (using the Java *FQM Utility*, see section 5.6).

Despite the parallelism, the designed module's ports are still very similar to the simple transmitter. Indeed, all the parallel inputs and outputs are packed into the same bus as shown in Fig.19, where each $data_i$ is a 16-bit bus

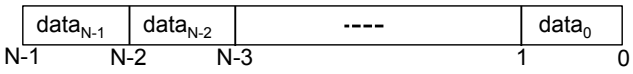



Figure 19: Parallel bus packing

Again, this section explains in details the implementation of the system depicted in Fig.20 block by block using the pattern introduced in the beginning of section 5.

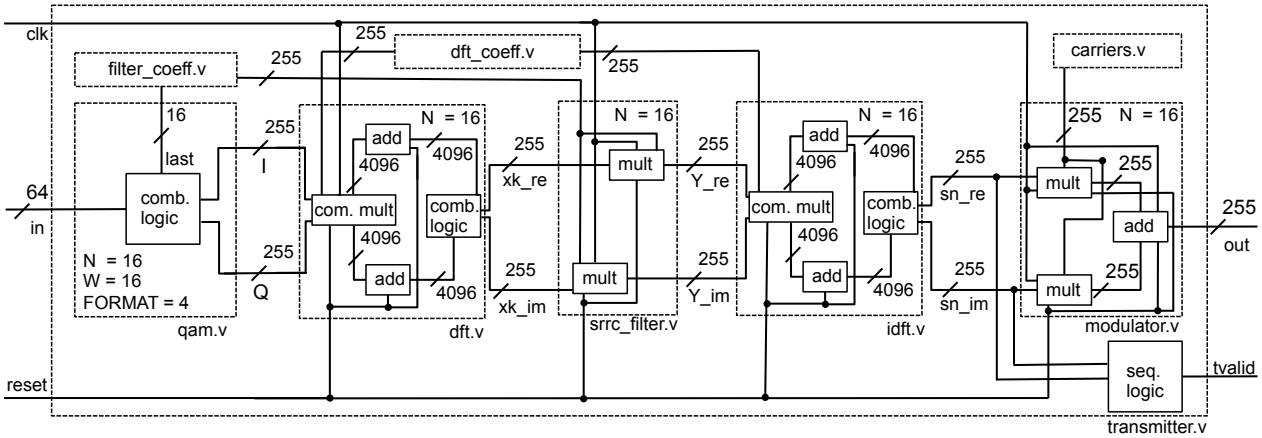



Figure 20: Implemented parallel system

## 5.2 Implementation - QAM Mapper

The first block performs the QAM mapping operation. It receives a `N` clustered input bit stream and delivers the `N` corresponding in-phase and quadrature QAM mapped signal.

### Specifications

`qam.v` – Receives `N` packed input buses of `W` bits and outputs `N` `W`-bit in-phase and quadrature packed QAM mapped signals.

| | | |
|---|---|---|
| **Latency** | - | |
| **Parameters** | `N` | Number of parallel inputs |
| | `W` | Bus width |
| | `FORMAT` | QAM order |
| **Inputs** | `in` | Clustered input stream |
| | `last` | Last constellation point |
| **Outputs** | `I` | In-phase component |
| | `Q` | Quadrature component |
| **IP Cores** | - | |
| **Dependencies** | `qam8.v` | Manage the 8-QAM modulation |
| | `qam16.v` | Manage the 16-QAM modulation |
| | `qam32.v` | Manage the 32-QAM modulation |
| | `qam64.v` | Manage the 64-QAM modulation |

Table 1: QAM Mapping - Specifications

### Module Explanations

In order to achieve the highest level of modularity for each block, this module can receive an arbitrary width input bus. This has the advantage of allowing the block to be easily used in another design.

However, due to the IP core implementation constraints, the transmitter bus width is fixed to 16-bit.

This module, shown in Fig.21, is set up using the three Verilog parameters `N`, `W` and `FORMAT`. Respectively, the first indicates the number of parallel inputs, the second defines the bus width and the third sets up the QAM modulation format.

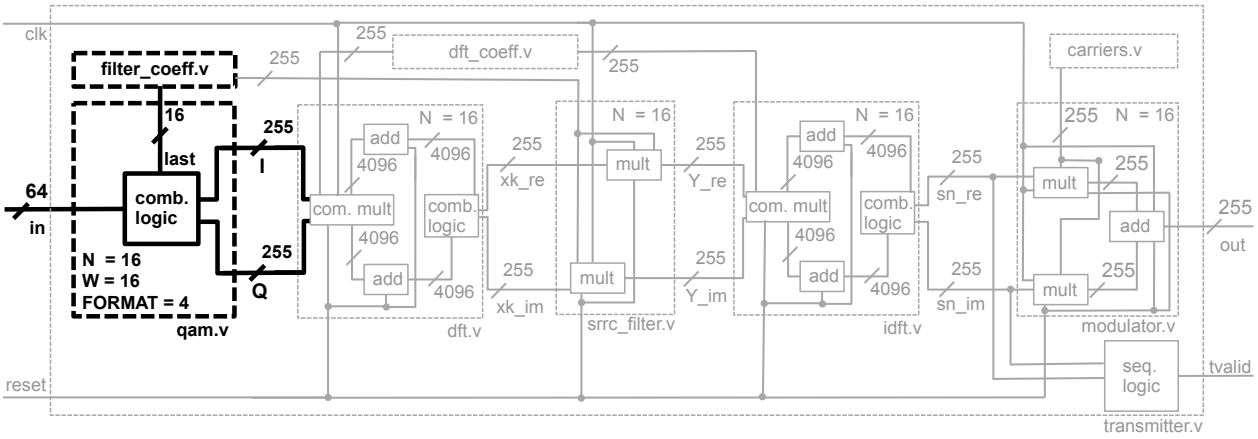



Figure 21: Implemented parallel system - QAM mapping Block

The available modulation formats are 8-QAM, 16-QAM (default), 32-QAM and 64-QAM. It's to note that the width of the input bus `in` depends on the specified modulation format (Fig.21 displays the default input size; i.e. $4 * 16 = 64$). Indeed, the input stream has to be cluttered by sets of 3, 4, 5, or 6 bits depending on the selected format (see section 3.1 for further explanations about the QAM cluster's width). Certainly, more modulation formats could have been implemented as 128-QAM or 256-QAM but their implementation would be similar the the already implemented units and doesn't add any additional interest to this paper.

### Hardware Code

This paragraph illustrates the Verilog code implementing the QAM-mapping module. This module is extremely simple because the complexity is equally distributed between all its dependencies.

As shown here below, the main goal of this unit is to select the appropriate QAM format by analysing the `FORMAT` parameter and by generating the corresponding dependency.

```
1  /*****************************************************************************
2  * QAM Mapping
3  *****************************************************************************/
4  generate // generate only the required module
5     case(FORMAT)
6      3'b011:    // 8-QAM
7        qam8 #(
```

```
8             .N(N),        // Number of parallel inputs
9             .W(W)         // Bus Witdh
10          ) qam8_isnt(
11            .in(in),      // Clustered input stream
12            .last(last),  // Last constellation point
13            .I(I),        // In-phase component
14            .Q(Q)         // Quadrature component
15          );
16        3'b100: qam16 #(N, W) qam16_isnt(in, last, I, Q); // 16-QAM
17        3'b101: qam32 #(N, W) qam32_isnt(in, last, I, Q); // 32-QAM
18        3'b110: qam64 #(N, W) qam64_isnt(in, last, I, Q); // 64-QAM
19        default:   // unsupported or undefined QAM format
20          begin assign I = 'bx; assign Q = 'bx; end
21      endcase
22    endgenerate
23    /*************************************************************************/
```

If the format is unrecognised, the modules don't generate any dependencies and simply outputs undefined signals.

### Cores Configuration and Dependencies

Since this module don't use any IP cores, this paragraph is entirely dedicated to the dependencies of the block. As mentioned in the previous paragraph, all the responsibilities are delegated to the block's dependencies. Indeed, the Verilog modules `QAM8`, `QAM16`, `QAM32` and `QAM64` handle the 8-QAM, 16-QAM, 32-QAM and 64-QAM modulation by its own.

The code below only illustrates the implementation of the 16-QAM format but the discussion can be extended to the other QAM modules. The signals `p1` and `last` represent respectively the first and the last constellation point. For an easier understanding, this code has to be read in parallel with Fig.6b and the 16-QAM explanation given in section 3.1.1.

```
1     /*************************************************************************
2     * 16-QAM Mapping
3     *************************************************************************/
4     genvar i;
5     generate for(i=0; i < N; i=i+1)
6       always@(*) begin
7         case({in[4*i+3],in[4*i+1]}) // assign I value
8           2'b00:   I[W*i+(W-1):W*i] =  p1;
9           2'b01:   I[W*i+(W-1):W*i] =  last;
10          2'b10:   I[W*i+(W-1):W*i] = -p1;
11          2'b11:   I[W*i+(W-1):W*i] = -last;
12          default: I[W*i+(W-1):W*i] = 'dx;
13        endcase
```

```
14        case({in[4*i+2],in[4*i]})   // assign Q value
15          2'b00:   Q[W*i+(W-1):W*i] =  p1;
16          2'b01:   Q[W*i+(W-1):W*i] =  last;
17          2'b10:   Q[W*i+(W-1):W*i] = -p1;
18          2'b11:   Q[W*i+(W-1):W*i] = -last;
19          default: Q[W*i+(W-1):W*i] = 'dx;
20        endcase
21      end
22    endgenerate
23    /*****************************************************************************/
```

This code is extremely simple since it consists in purely combinatorial logic. Nevertheless, the generate loop allows to easily meet the modulatory specification by repeating the code for each one of the parallel input N and to pack the date shown in Fig.19.

## 5.3 Implementation - Discrete Fourier Transform

This section is probably the most complex of the chapter. It is dedicated to the blocks performing the Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (IDFT). Those blocks are described in the same section because of their enormous similarity. The DFT block (resp. IDFT block) receives a signal in the time domain (resp. frequency domain) and outputs its corresponding transformation in the frequency domain (resp. time domain).

### Specifications

---

`dft.v` / `idft.v` – Receives `N` 16-bit packed input busses and outputs `N` 16-bit packed busses corresponding to the input's tranform.

| | | |
|---|---|---|
| **Latency** | 4 cycles | |
| **Parameters** | `N` | The transform length |
| **Inputs** | `clk` | Clock |
| | `reset` | Reset |
| | `xn_re` | DFT real input |
| | `xn_im` | DFT imaginary input |
| | `ccos` | Cosine DFT coefficients |
| | `csin` | Sine DFT coefficients |

| | | |
|---|---|---|
| **Outputs** | `xk_re` | DFT real output |
| | `xk_im` | DFT imaginary output |
| **IP Cores** | `Complex Multilier` | Version 5.0 |
| | `Adder Subtracter` | Version 11.2 |
| **Dependencies** | - | |

Table 2: DFT / IDFT - Specifications

### Module Explanations

The only input parameter of this module is `N` that can be either viewed as the transform length or the number of parallel input busses.

Indeed, the signals precision is fixed to 16 bits because the IP cores require this information to be manually hardcoded inside the Xilinx configuration interface and therefore, cannot be changed programmatically.

In addition to the clock and the reset signal, this module requires the time domain complex input separated in its real (`xn_re`) and imaginary (`xn_im`) part. Moreover, the cosine and sine coefficients weight (respectively called `ccos` and `csin`) to apply during the transformation process have also to be provided. Fig.22 high-lines the DFT and IDFT block implementation.

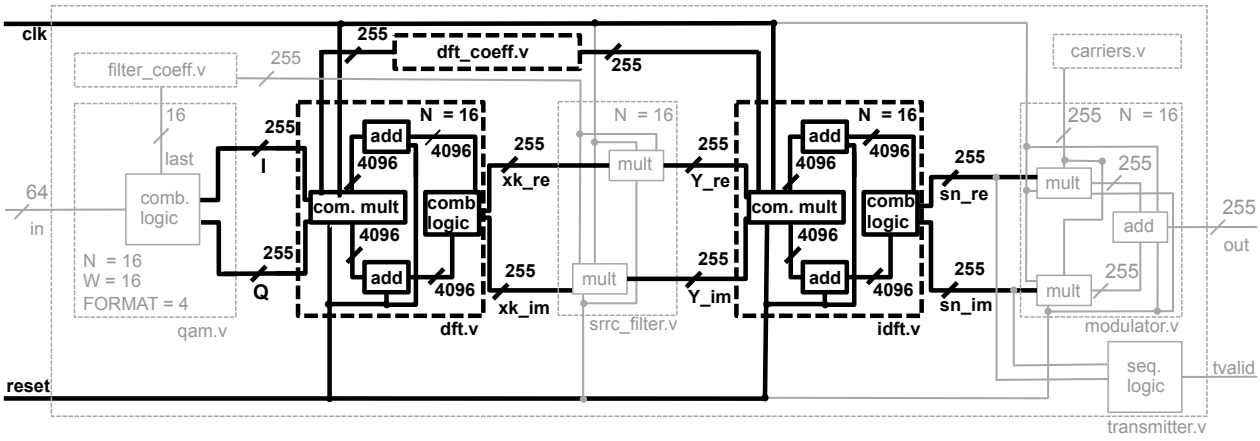



Figure 22: Implemented parallel system - DFT and IDFT blocks

The coefficients have to be provided using the reverse packed pattern respect that the module's port. In order words, as shown in Fig.23, the first data set comes at position $N-1$, the second comes at position $N-2$, etc.

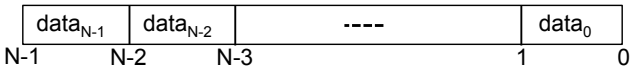



Figure 23: Coefficient bus packing

Unfortunately, the FFT Xilinx IP core used for the single channel transmitter requires a serial input and mandatorily delivers a serial output and, by consequence, cannot be used in the parallel design. Therefore, in order to achieve the aimed parallelisation, the DFT had to be completely implemented assembling adders and multipliers[6]. More specifically, Eq.(4) of section 3.2.1 has been implemented in hardware. First, all the product $(x[n]e^{-2\pi ikn/N})$ are computed and stored locally. Afterwards, the products are summed up to complete the equation. This process is implemented `N` time, to compute each $X[k]$.

An other alternative could have been to implement this equation using `N` Multiply and Accumulate (MAC) logic cores (one for each $X[k]$) in order to spare adders and multipliers. However, once more, the Xlinx MAC IP Core requires a serial input and has a latency depending on the number of terms `N`. By consequence, the transmitter latency would have depended on the number of input parameters, which is the reason that this option has not been considered.

As a final note, it is to remark that both the DFT and IDFT blocks rescale the output by $2^{-17}$ to fit in the desired 16-bit bus and avoid the possible overflow due to the 33-bit result produced by the complex multipliers.

### Hardware Code

This paragraph illustrate the Verilog code implementing the DFT module. The IDFT module doesn't deserve its own explanation due to its similarity with the DFT[7].

As shown here below, the code is composed of two generate loops: the inner one loops over each input $n$ and the outer one loops to generate to output results $k$.

```
1   genvar k,n,i;
2    generate
3     for(k=0; k < N; k=k+1) begin
4       // Compute complex products
5       for(n=0; n < N; n=n+1) begin
6         complex_mult complex_mult_fft_inst (
7           .aclk(clk),                // input aclk
8           .aresetn(~reset),          // input aresetn
```

[6]Nevertheless, after discussion with the Xilinx development team, it has been certified that the next version of the core will allow parallel inputs and outputs due to the huge demand in that area.

[7]Indeed, the unique difference between these two modules is the minus sign placed in front of the sine coefficients provided to the complex multipliers.

```
9           .s_axis_a_tvalid(1'b1),     // input s_axis_a_tvalid
10          .s_axis_a_tdata({
11            xn_im[16*n+(16-1):16*n],// DFT imaginary input
12            xn_re[16*n+(16-1):16*n] // DFT real input
13          }),
14          .s_axis_b_tvalid(1'b1),     // input s_axis_b_tvalid
15          .s_axis_b_tdata({ // Sine and cosin DFT coefficients
16            -csin[16*(N-1-n)+(16-1)+N*16*(N-1-k):16*(N-1-n)+N*16*(N-1-k)],
17            ccos[16*(N-1-n)+(16-1)+N*16*(N-1-k):16*(N-1-n)+N*16*(N-1-k)]
18          }),
19          .s_axis_ctrl_tvalid(1'b1), // input s_axis_ctrl_tvalid
20          .s_axis_ctrl_tdata(8'b0),  // input [7 : 0] s_axis_ctrl_tdata
21          .m_axis_dout_tvalid(),     // output m_axis_dout_tvalid
22          .m_axis_dout_tdata({
23            t_im[16*n+(16-1)+N*16*k:16*n+N*16*k], // DFT imaginary output
24            t_re[16*n+(16-1)+N*16*k:16*n+N*16*k]  // DFT real output
25          })
26        );
27      end
```

The first operation performed by the DFT module is to multiply the inputs with the corresponding trigonometric coefficients. All the produced results are stored in two temporary signals called `t_re` or `t_im` depending on their real or imaginary nature.

Secondly, the temporary results are added two by two in oder to produce the DFT sums. The final output can be found on each $N-2$ last positions of the signals `a_re` and `a_im`.

```
1   // Sum up complex products
2   adder_fft adder_fft_inst_re (
3     .a(t_re[16*0+(16-1)+N*16*k:16*0+N*16*k]), // input [15 : 0] a
4     .b(t_re[16*1+(16-1)+N*16*k:16*1+N*16*k]), // input [15 : 0] b
5     .sclr(reset),                             // input sclr
6     .s(a_re[16*0+(16-1)+N*16*k:16*0+N*16*k])  // output [15 : 0] s
7   );
8   adder_fft adder_fft_inst_im (
9     .a(t_im[16*0+(16-1)+N*16*k:16*0+N*16*k]), // input [15 : 0] a
10    .b(t_im[16*1+(16-1)+N*16*k:16*1+N*16*k]), // input [15 : 0] b
11    .sclr(reset),                             // input sclr
12    .s(a_im[16*0+(16-1)+N*16*k:16*0+N*16*k])  // output [15 : 0] s
13  );
14  for(i=2; i < N; i=i+1) begin
15    adder_fft adder_fft_inst_re_loop (
16        .a(t_re[16*i+(16-1)+N*16*k:16*i+N*16*k]),         // input [15 : 0] a
17        .b(a_re[16*(i-2)+(16-1)+N*16*k:16*(i-2)+N*16*k]), // input [15 : 0] b
18        .sclr(reset),                                     // input sclr
```

```
19          .s(a_re[16*(i-1)+(16-1)+N*16*k:16*(i-1)+N*16*k])  // output [15 : 0] s
20      );
21      adder_fft adder_fft_inst_im_loop (
22          .a(t_im[16*i+(16-1)+N*16*k:16*i+N*16*k]),         // input [15 : 0] a
23          .b(a_im[16*(i-2)+(16-1)+N*16*k:16*(i-2)+N*16*k]), // input [15 : 0] b
24          .sclr(reset),                                     // input sclr
25          .s(a_im[16*(i-1)+(16-1)+N*16*k:16*(i-1)+N*16*k])  // output [15 : 0] s
26      );
27  end
```

As a final remark, the small mask is applied to the output to delete the the first four output caused by the complex multipliers latency.

### Cores Configuration and Dependencies

Contrarily to the previous block, the DFT and the IDFT blocks don't have any dependencies but take advantage of two Xilinx IP cores: the `Complex Multiplier v5.0` and the `Adder Subtracter v11.2`. The `Complex Multiplier v5.0` core is used $N^2$ times and is configured to work with 16-bit inputs. Tab.3 here below summarizes the core's ports.

`dft.v` / `idft.v` — `Complex Multiplier v5.0`

| | | |
|---|---|---|
| **Latency** | 4 cycles | |
| **Inputs** | `aclk` | Clock |
| | `aresetn` | Negative reset |
| | `s_axis_a_tvalid` | Valid flag for first input |
| | `s_axis_a_tdata` | First input (imaginary and real part) |
| | `s_axis_b_tvalid` | Valid flag for first input |
| | `s_axis_b_data` | Second input (imaginary and real part) |
| | `s_axis_ctrl_tvalid` | Valid flag for control input |
| | `s_axis_ctrl_tdata` | Control Signal |
| **Outputs** | `m_axis_dout_tvalid` | Valid flag for output |
| | `m_axis_dout_tdata` | Output (imaginary and real part) |

Table 3: Complex Multiplier ports

After performance's analysis, both cores have been configured to prefer Mults or DSP before fabric (see section 6). The *symmetric rounding* option allows to rescale the output to 16 bits instead of 33. This is done by entering a byte of zeros in the `s_axis_ctrl_tdata` core's input. Since this core implements the *AXI4-Stream* compliant, the real and imaginary part of all inputs and outputs are concatenated in the same bus, starting by the imaginary part.

The `Adder Subtracter v11.2` core is used $2N(N-1)$ times and is configured to work in *adder* mode with 16-bit inputs. This fair amount of core's instances is required by the fact that both the imaginary and the real part have to be processed. Because these additions need to be synchronised, the core is manually configured to have zero latency. Tab.4 here below summarizes the core's ports. Two processing options are available: DSP48 or Fabric. Both options will be analysed in section 6.

`dft.v` / `idft.v` — `Adder Subtracter v11.2`

| | | |
|---|---|---|
| **Latency** | - | |
| **Inputs** | `a` | First input |
| | `b` | Second First input |
| | `sclr` | Synchronous clear |
| **Outputs** | `s` | Output |

Table 4: Adder Subtracter ports

## 5.4 Implementation - SRRC Filter

This block filter the signal in frequency domain by using the mathematical property described by Eq.(7) in section 3.2.1.

### Specifications

`srrc_filter.v` — Performs a parallel filtering operation in frequency domain from the given complex inputs and filter coefficients.

| | | |
|---|---|---|
| **Latency** | 4 cycles | |
| **Parameters** | `N` | Number of parallel inputs |
| **Inputs** | `clk` | Clock |
| | `reset` | Reset |
| | `X_re` | Real inputs |
| | `X_im` | Imaginary inputs |
| | `H` | Filter coefficients |
| **Outputs** | `Y_re` | Real outputs |
| | `Y_im` | Imaginary outputs |
| **IP Cores** | `Multiplier` | Version 11.2 |
| **Dependencies** | - | |

Table 5: QAM Mapping - Specifications

### Module Explanations

Exactly as the previous blocks, this module is set up through the unique parameter `N` defining the number of parallel inputs. As usual, Fig.24 here below recall the filter block inside the design.

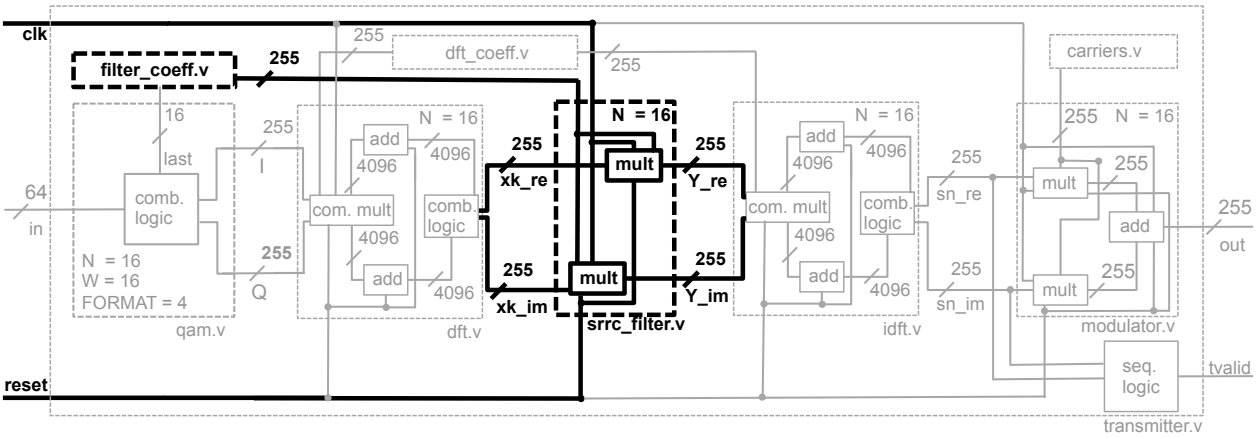



Figure 24: Implemented parallel system - Filter block

The implementation is very straightforward, all signals are presented in their packed format and the coefficients have to be provided following the inverse packed pattern, identically to the trigonometric coefficients of the DFT and IDFT.

Similarly to the `Complex multiplier` core, the `Multiplier` cores rescale the output by $2^{-16}$ to fit in the desired 16-bit bus and avoid the possible overflow due to the standard 32-bit result.

### Hardware Code

This paragraph illustrates the Verilog code implementing the filter module. This module is very simple as everything is handled by the Xilinx IP cores.

As shown here below, a generate loop is applied over each real and imaginary input to multiply it by the corresponding filter coefficient.

```
1  /*****************************************************************************
2  * Filter and Resize by 2*(-16)
3  *****************************************************************************/
4  genvar i;
5  generate
6    for(i=0; i < N; i=i+1) begin
7      mult mult_filter_inst_re (
8        .clk(clk),                              // input clk
9        .a(X_re[16*i+(16-1):16*i]),             // input [15 : 0] a
10       .b(H[16*(N-1-i)+(16-1):16*(N-1-i)]),    // input [15 : 0] b
11       .sclr(reset),                           // input sclr
12       .p(Y_re[16*i+(16-1):16*i])              // output [15 : 0] p
13     );
14     mult mult_filter_inst_im (
15       .clk(clk),                              // input clk
16       .a(X_im[16*i+(16-1):16*i]),             // input [15 : 0] a
17       .b(H[16*(N-1-i)+(16-1):16*(N-1-i)]),    // input [15 : 0] b
18       .sclr(reset),                           // input sclr
19       .p(Y_im[16*i+(16-1):16*i])              // output [15 : 0] p
20     );
21   end
22 endgenerate
23 /*****************************************************************************/
```

### Cores Configuration and Dependencies

The only core in this block is the `Multiplier v11.2`. Because of the need to process real and imaginary inputs, $2N$ core's instances are needed. Again, Tab.6 here below summarizes the core's ports.

`srrc_filter.v` – `Multiplier v11.2`

| | | |
|---|---|---|
| **Latency** | 4 cycles | |
| **Inputs** | `clk` | Clock |
| | `a` | First First input |
| | `b` | Second First input |
| | `sclr` | Synchronous clear |
| **Outputs** | `p` | Output |

Table 6: Multiplier ports

The core is configures to receive 16-bit inputs and to produce 16-bit symmetrically rounded outputs. Moreover, this core allows the user to select if the multiplication has to occurs through Mults or LUTs. As usual, both options are analysed in section 6.

## 5.5 Implementation - Modulator

The modulator performs the operation described by Eq.(1) of section 3.1.2: it multiplies the real input by the cosine carrier and the imaginary input by the sine carrier, then those product are subtracted.

**Specifications**

`modulator.v` – Modulate the N 16-bit inputs packed in parallel by the provided carriers and output N 16-bit modulated output packed.

| | | |
|---|---|---|
| **Latency** | 5 cycles | |
| **Parameters** | `N` | Number of parallel inputs |

| | | |
|---|---|---|
| **Inputs** | `clk` | Clock |
| | `reset` | Reset |
| | `I` | Real inputs |
| | `Q` | Imaginary inputs |
| | `cos` | In-phase carrier |
| | `sin` | Quadrature carrier |
| **Outputs** | `dout` | Output |
| **IP Cores** | `Multiplier` | Version 11.2 |
| | `Adder Subtracter` | Version 11.2 |
| **Dependencies** | - | |

Table 7: Modulator - Specifications

### Module Explanations

The module is parametrised with the usual parameter `N` defining the number of parallel inputs and is depicted in Fig.25. Since, it uses the same multipliers as the SRRC filter, the discussion about the precision and the rescaling remains the same.

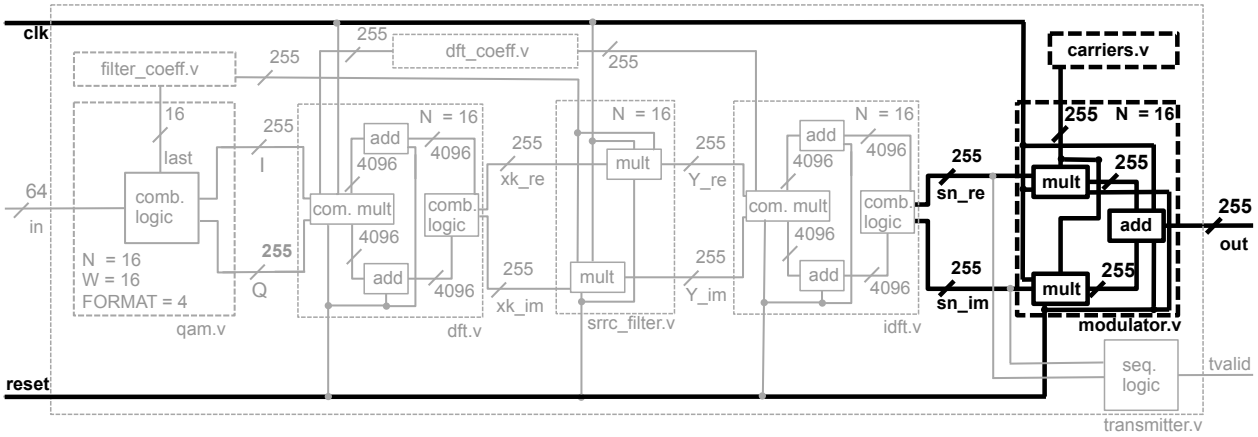



Figure 25: Implemented parallel system - Modulator block

Again, all signals are presented in their packed format and the carriers have to be provided following the inverse packed pattern, identically to the trigonometric coefficients of the DFT / IDFT and to the filter's coefficients.

### Hardware Code

The Verilog code implementing this module exactly the same as the filter, except that the `adder subtracter` core is instantiated in order to subtract the two multiplier's outputs. Again, a generate loop ensure the process of all the inputs in parallel.

```
1  /*****************************************************************************
2  * Multiplier and Subtracter - Rescaling 2^(-16)
3  *****************************************************************************/
4  genvar i;
5  generate
6    for(i=0; i < N; i=i+1) begin
7      // Multiplier
8      mult mult_mod_inst_re (
9        .clk(clk),                                  // input clk
10       .a(I[16*i+(16-1):16*i]),                    // input [15 : 0] a
11       .b(cos[16*(N-1-i)+(16-1):16*(N-1-i)]),      // input [15 : 0] b
12       .sclr(reset),                               // input sclr
13       .p(p1[16*i+(16-1):16*i])                    // output [15 : 0] p
14     );
15     mult mult_mod_inst_im (
16       .clk(clk),                                  // input clk
17       .a(Q[16*i+(16-1):16*i]),                    // input [15 : 0] a
18       .b(sin[16*(N-1-i)+(16-1):16*(N-1-i)]),      // input [15 : 0] b
19       .sclr(reset),                               // input sclr
20       .p(p2[16*i+(16-1):16*i])                    // output [15 : 0] p
21     );
22     // Subtracter
23     Subtracter Subtracter_mod_inst (
24       .a(p1[16*i+(16-1):16*i]),                   // input [15 : 0] a
25       .b(p2[16*i+(16-1):16*i]),                   // input [15 : 0] b
26       .clk(clk),                                  // input clk
27       .sclr(reset),                               // input sclr
28       .s(dout[16*i+(16-1):16*i])                  // output [15 : 0] s
29     );
30   end
31  endgenerate
32  /*****************************************************************************/
```

#### Cores Configuration and Dependencies

The two cores instantiated in this module are previously used the `Multiplier v11.2` and `Adder Subtracter v11.2`. The multiplier configuration is exactly the same as in the filter module and is therefore not discussed in this paragraph.

Nevertheless, the `Adder Subtracter` core is not configured in the same way as it is in the DFT / IDFT module. Indeed, in addition to be configured as subtracter instead as adder, since there is no more needs for synchronisation, the core latency is automatically set to two clock cycle in the pursuit of performance's optimization. As usual, Tab.8 here below summarizes the ports of this core.

`modulator.v` — `Adder Subtracter v11.2`

| | | |
|---|---|---|
| **Latency** | 2 cycles | |
| **Inputs** | `clk` | Clock |
| | `a` | First First input |
| | `b` | Second First input |
| | `sclr` | Synchronous clear |
| **Outputs** | `p` | Output |

Table 8: Modulator Adder Subtracter ports

observing the hardware code of the previous section, we notice that $2N$ `Multiplier` cores and $N$ `Adder Subtracter` cores are needed.

### 5.6 Implementation - Transmitter

This last module called *Transmitter* is the top entity of the design. It assembles all the previously described blocks. By consequence, even though this module doesn't directly possess any IP cores, it needs as dependencies all the previously blocks as well as the the filter's coefficients, the DFT's coefficient and the carriers.

This last part of the section is slightly different from the other because in addition to describer the module as usual, it also explains step by step how the design can be used by an adequately informed the user.

**Specifications**

| | | | |
|---|---|---|---|
| `transmitter.v` | – | Receives N 16-bit packed inputs and output N 16-bit QAM modulated packed waves. | |
| **Latency** | 17 cycles | | |
| **Parameters** | `N` | Number of parallel inputs | |
| | `FORMAT` | QAM order | |
| **Inputs** | `clk` | Clock | |
| | `reset` | Reset | |
| | `in` | Clustered input stream | |
| **Outputs** | `tvalid` | Output's valid flag | |
| | `out` | Output | |
| **IP Cores** | - | | |
| **Dependencies** | `qam.v` | QAM module | |
| | `dft.v` | DFT module | |
| | `srrc_filter.v` | SRRC filter module | |
| | `idft.v` | IDFT module | |
| | `modulator.v` | Modulator module | |
| | `dft_coeff.v` | DFT / IDFT coefficients | |
| | `filter_coeff.v` | Filter coefficients | |
| | `carriers.v` | Carriers | |

Table 9: Transmitter - Specifications

**Module Explanations**

Two input parameters define the behaviour of the transmitter. The first parameter is `N`, representing the number of parallel inputs and the second is `FORMAT` to specify the desired modulation format

among the four available QAM formats (see section 5.2). Fig.26 shows an abstract view of the system focussing on its input and output ports. The inputs are the clock, the reset signal and the clustered binary stream.

As mentioned before, the input stream has to be clustered by the number of bits defined by the Verilog parameter `FORMAT`. In other words, if a 16-QAM modulation is desired, the user needs to specify `FORMAT=4` and enter `N` bit streams, clustered by 4 and packed in a single bus of width `(FORMAT*N)`.

The outputs are the modulated signal, delivered in his packed representation and the `tvalid` flag, indicating when the output data are valid. The output width is `(16*N)` since the signal precision is fixed to 16 bits.

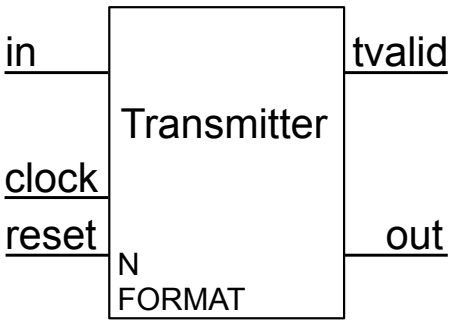



Figure 26: Transmitter - Implementation

In order to use the design properly, the user should start by running the Java application *FQM Utility* (see Fig.27).

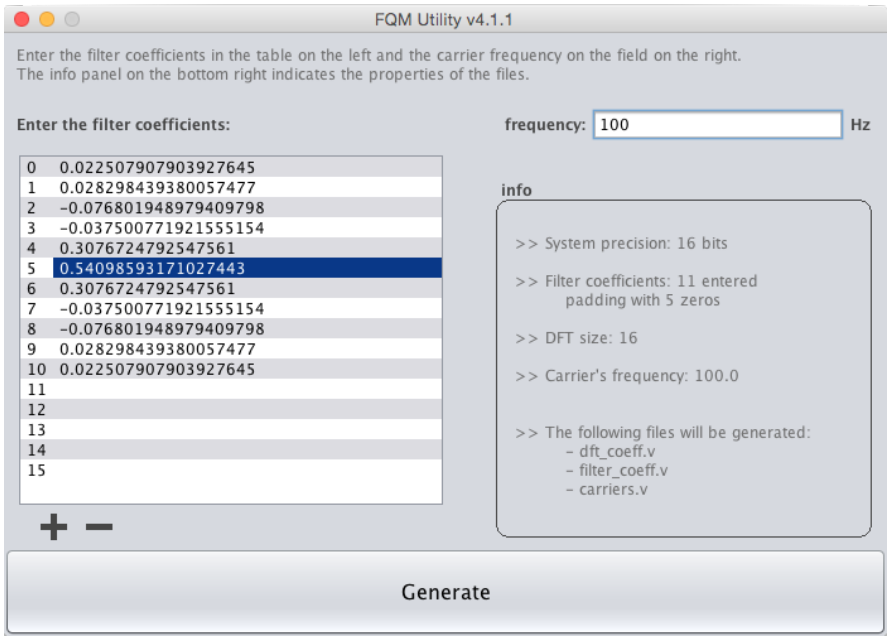



Figure 27: Fourier QAM Modulation (FQM) Utility

First, the desired filter coefficients have to entered in table on the left. Fifteen rows are displayed by defaults but if more that fifteen coefficients are needed, the "+" button on the bottom of the table adds extra rows. Only the filled rows are considered; i.e. all the empty rows are considered inexistent. The "-" button suppresses a selected row.

Secondly, the carrier frequency has to be entered in Hertz in the text field on the top right of the interface.

The information field on the bottom right of the interface updates the entered parameters in real time. The first row indicates the system precision and the second indicates the number of filter coefficients entered as well as the number of zeros that will be padded in order to reach `N`; i.e.the smallest power of two greater than the number of coefficients. The third line represents `N`, which is then the DFT size[8] or the number of parallel inputs. Then, the carrier's frequency is displayed and, finally, the last lines of the info field are the Verilog units that will be generated.

Afterwards, to run the transmitter design, the user must include those three files into the project, and run the top module `transmitter.v` with the same parameter `N` as displayed by the `FQM Utility` and with the desired `FORMAT` parameter.

### Hardware Code

The following code instantiates the three units (`dft_coeff.v`, `filter_coeff.v`, `carriers.v`) generated by the *FQM Utilities*.

```
1  /*****************************************************************************
2   * Config.
3   *****************************************************************************/
4   // Load DFT Coefficients
5   dft_coeff dft_coeff_inst(
6    .ccos(ccos),                  // Cosine DFT coefficients
7    .csin(csin)                   // Sine DFT coefficients
8   );
9   // Load filter coefficients
10  filter_coeff filter_coeff_inst(
11   .H(H),                        // Filter coefficients
12   .H_max(H_max)                 // Highest filter coefficient
13  );
14  // Load carriers
15  carriers carriers_inst(
16   .clk(clk),                          // Clock
17   .reset(reset | (~|{sn_re,sn_im})), // Reset
18   .cos(cos),                          // Cosine carrier
19   .sin(sin)                           // Sine carrier
20  );
21  /*****************************************************************************/
```

[8] Event if the DFT algorithm doesn't require the number of inputs to be a power of two, the FFT does (see appendix A). As mentioned before, Xilinx will soon provide the next generation of the FFT IP core able to process parallel data and therefore, this design is built to include this core as soon as it is distributed.

Next, all the previously discussed unit are instantiated and despite being repetitif, the code below clearly shows the the connection between them. For easy understanding, this code has to be read in parallel with Fig.20 given in the beginning of this section.

```
1  /*****************************************************************************
2  * QAM mapping
3  *****************************************************************************/
4  QAM #(
5   .N(N),           // Number of parallel inputs
6   .W(16),          // Bus Witdh
7   .FORMAT(FORMAT) // QAM order
8  ) QAM_isnt(
9   .in(in),         // Clustered input stream
10  .last(H_max),    // Last constellation point
11  .I(I),           // In-phase component
12   .Q(Q)           // Quadrature component
13 );
14
15
16 /*****************************************************************************
17 * DFT
18 *****************************************************************************/
19 dft #(
20  .N(N)            // Transform length
21 ) dft_inst(
22  .clk(clk),       // Clock
23  .reset(reset),   // Reset
24  .xn_re(I),       // DFT real input
25  .xn_im(Q),       // DFT imaginary input
26  .ccos(ccos),     // Cosine DFT coefficients
27  .csin(csin),     // Sine DFT coefficients
28   .xk_re(xk_re), // DFT real input
29   .xk_im(xk_im)  // DFT imaginary input
30 );
31
32
33 /*****************************************************************************
34 * SRRC filter
35 *****************************************************************************/
36 SRRC_filter #(
37  .N(N)             // Number of parallel inputs
38 ) SRRC_filter_inst (
39  .clk(clk),       // Clock
40  .reset(reset),   // Reset
```

```
41      .H(H),           // Filter coefficients
42        .X_re(xk_re),  // Filter's real input
43      .X_im(xk_im),    // Filter's imaginary input
44      .Y_re(Y_re),     // Filter's real output
45      .Y_im(Y_im)      // Filter's imaginary output
46    );
47
48
49    /*****************************************************************************
50    * IDFT
51    *****************************************************************************/
52    idft #(
53      .N(N)           // Transform length
54    ) idft_inst(
55      .clk(clk),      // Clock
56      .reset(reset),  // Reset
57      .xn_re(Y_re),   // DFT real input
58      .xn_im(Y_im),   // DFT imaginary input
59      .ccos(ccos),    // Cosine DFT coefficients
60      .csin(csin),    // Sine DFT coefficients
61       .xk_re(sn_re), // DFT real input
62       .xk_im(sn_im)  // DFT imaginary input
63    );
64
65
66    /*****************************************************************************
67    * Modulator
68    *****************************************************************************/
69    modulator #(
70      .N(N)          // Number of parallel inputs
71    ) modulator_inst (
72      .clk(clk),     // Clock
73      .reset(reset), // Reset
74      .I(sn_re),     // In-phase input
75      .Q(sn_im),     // In-Quadrature input
76      .sin(sin),     // In-phase carrier
77      .cos(cos),     // Quadrature carrier
78      .dout(out)     // Output
79    );
80    /*****************************************************************************/
```

### Cores Configuration and Dependencies

Since the transmitter unit doesn't posses any IP cores and almost all the dependencies consist in the previous blocks, the only units explained in this paragraph are the modules auto-generated by the Java program.

Nevertheless, going through the multiples dependencies of this block, we can compute the total number of IP cores used by the complete system: this design requires $2N^2$ `Complex Multiplier`, $4N^2 - 2N$ `Adder Subtracter` and $4N$ `Multipliers`.

The modules `dft_coeff.v` and `filter_coeff.v` are pretty simple while they consist in outputting hardcoded values. Nevertheless, the module `carriers` is a little more complex since it contains the trigonometric carriers and has to output them synchronously with the data flow. The Verilog code below shows an example of a this unit generated for a carrier frequency of 100Hz and `N=16`.

```
1  // log2(400 / 16) = 5
2  reg [4:0] count;
3
4  always@(posedge clk, posedge reset)
5     if(reset | (count == 'd24)) count <= 'd0;
6     else                        count <= count + 'd1;
7
8  always@(count, reset)
9     case(count)
10      5'd0: begin
11        cos = 256'b010000000000000000 ...
12        sin = 256'b000000000000000000 ...
13      end
14      5'd1: begin
15        cos = 256'b00111101111111010011110110111 0...
16        sin = 256'b00001111111101011000100001111...
17      end
18
19     ...
20    endcase
```

In this particular case, the number of sample is $(4 * 100 \bmod 16) = 400$. Each clock cycle, `(16*N)` packed trigonometric carriers are output and once the period is over, the counter is reinitialised.

As explained in section 3.1.2, the sampling rate has to be at least twice the carrier's frequency. Nevertheless, in this implementation, the number of samples is computed to be the smallest number being four time the carrier's frequency and divisible by `N`. This property is primary to satisfy the Nyquist criterion and ensure continuity when the period is repeated, since `N` samples have to be processed each clock cycle.

# 6 Experimental Results

This last section is devoted to the experimental results obtained from the parallel QAM transmitter's implementation described in the previous sections.

This section starts by comparing the obtained modulator's output with the MATLAB reference simulation described in section 4.2. As explained in section 5, diverse design choices have to be made while configuring the IP cores. More specifically, the `Adder Subtracter` cores have to be configured to used either the FPGA fabric or the DSP and, the `Multiplier` IP cores offer a similar choice between Mults or LUTs. Therefore, the last part of this section investigates the all the pertinent configuration possibilities.

## 6.1 Experimental Results - Design Precision

This section aims to compare the transmitter's output data with the reference MATLAB model in order to observe the system's precision. As mentioned before, the MATLAB simulation is considered as perfect in this work; i.e. all the internal MATLAB rounding errors are ignored.

Considering a set of 16 parallel random inputs, Fig.28 here below displays the amplitude of the system's output computed by MATLAB and by the implemented transmitter. Both results are plotted on the same graph but only one curved is visible due to their proximity. However, zooming on the plot, we can still observe a small error. In that purpose, Fig.29 is then devoted to plot this error as an absolute value[9].

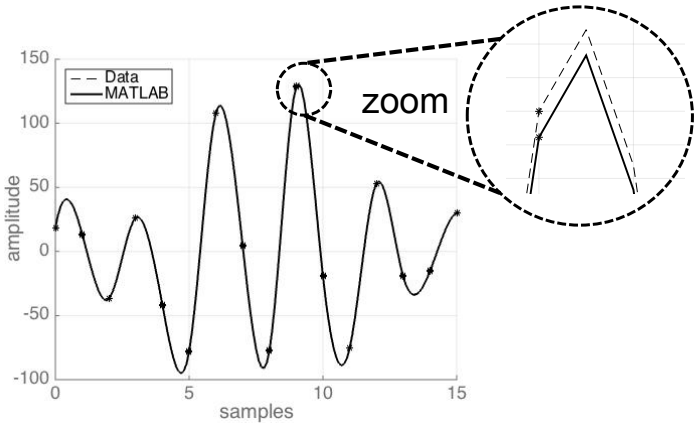



Figure 28: Transmitter's precision comparison

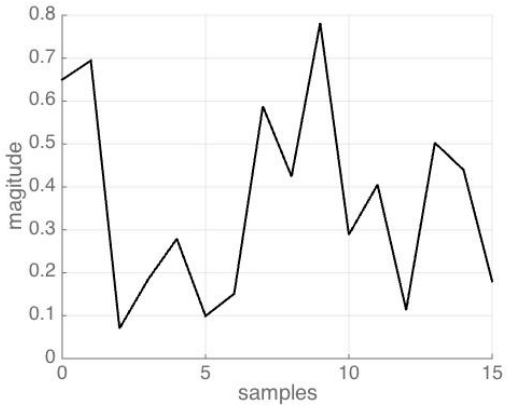



Figure 29: Transmitter's error

From these figures, we can deduce that the implemented system has less than 1% of error respect to the MATLAB model. For completeness, many other sets of random input samples have been tested and the precision still appears to be very similar to the one exposed above.

[9]It would have been more appropriate to plot the error in relative value but unfortunately, due to the numerous zeros, such a plot would have present singularities.

For further analysis, appendix B presents a comparison between the MATLAB reference model and the realised system for each block of the system.

## 6.2 Experimental Results - Design Resources and Performances

All the simulations in this section have been made by selecting the parameter `N=16` and a carrier frequency of 100Hz. Certainly, multiple other simulations could have been easily made but since the place and route's time exceeds thirty hours, they are not part of this work.

This section starts by showing the resources requirements and the performances reached by the optimization of the QAM system configuring the adders in order to use DSP instead of the fabric and the multipliers in order to use Mults instead of LUTs. Secondly, the combination Fabric - LUTs is inspected and finally, the configuration Fabric - Mults is analysed.

### 6.2.1 Design Resources and Performances - DSP & Mults Combination

Fig.31 plots the performances achieved when all adders are configured to use DSPs instead of the fabric and Mults instead of LUTs, respect to the entered time constraints.

We can observe that the maximum achievable clock frequency after the place and route operation is 28.57 MHz, which is pretty low. Moreover, after the synthesis, the best achievable clock frequency is only 28.77 MHz.

| | |
|---|---|
| Slice Registers | 5% |
| Slice LUTs | 1% |
| LUTs Used as Logic | 1% |
| Occupied Slices | 14 % |
| Unused Flip Flop | 6% |
| Unused LUTs | 82% |
| Fully Used LUT-FF pairs | 11% |
| Bounded IOBs | 46% |
| BUFG-BUFGCTRLs | 6% |
| DSP48E1s | 92% |

Figure 30: DSP - Mults Resources

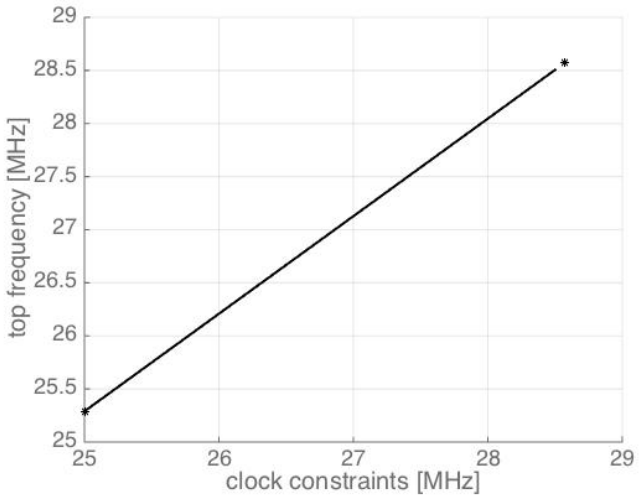



Figure 31: DSP - Mults Performances

It is therefore clear that selecting the DSP option in the adder's configuration utility is not optimal. This can be explained observing Tab.30. Indeed, the $10^{th}$ row of this table shows that 92% of the DSP48E1 block are used. This certainly causes a problem for an optimal place and route operation

and therefore, the design results very slow[10]. Indeed, routing such a huge amount of DSP48E1 requires much effort.

The above simulations have been made with all the supported QAM format but since those modules don't require many resources and are mainly combinatorial, the results stay identical.

### 6.2.2 Design Resources and Performances - Fabric & LUTs Combination

Similarly, to the previous section, Fig.33 displays the system's performances when all multipliers use LUTs instead of Mults. Since the use of DSP inside the adders appeared very inefficient in this design, the Fabric has been used instead.

| | |
|---|---|
| Slice Registers | 8% |
| Slice LUTs | 12% |
| LUTs Used as Logic | 11% |
| Occupied Slices | 22 % |
| Unused Flip Flop | 22% |
| Unused LUTs | 33% |
| Fully Used LUT-FF pairs | 44% |
| Bounded IOBs | 46% |
| BUFG-BUFGCTRLs | 6% |
| DSP48E1s | 54% |

Figure 32: Fabric - LUTs Resources

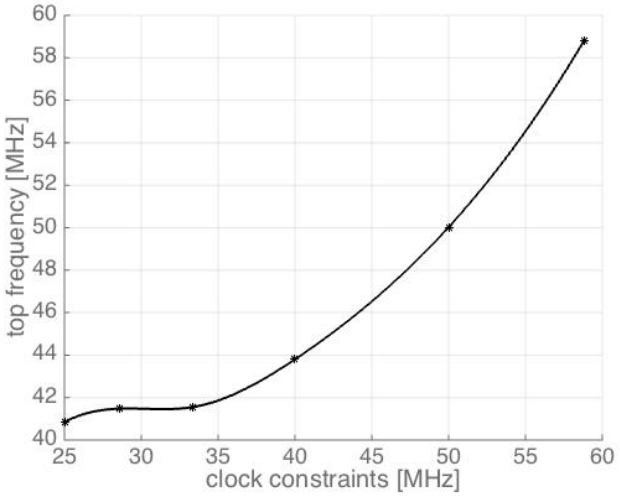



Figure 33: Fabric - LUTs Performances

This result is far better than the previous one. Indeed, the best achievable clock frequency after synthesis is 62.26MHz. However, after the place and route operation, this speed drops to 58.82 MHz.

Similarly to the previous experiment, is has been found out that the QAM format doesn't change significantly the experimental results.

### 6.2.3 Design Resources and Performances - Fabric & Mults Combination

Finally, the best results are obtained by configuring the adders in order to use the fabric and the multipliers to use the Mults. For that reason, this section contains deeper explanation regarding the designed system.

[10]It is to note that the resources utilisation shown in Tab.31 slightly vary from with the time constraints. However, only the resources of the optimal design in term of performances are presented.

At first sight, a speed of 62.59MHz is achievable after the logic synthesis and the place and route allows the clock to reach exactly 62.5MHz. This results is slightly better than the previous case.

Nevertheless, since the systems receive `N=16` parallel inputs, the effective reached speed is exactly:

$$16 * 62.5 = 1GHz$$

From this observation, since each M-QAM format's symbol contains $log_2(M)$ bits, we can derive the throughput for each one of the supported modulation formats:

$$\begin{aligned} 8-\text{QAM} &: 3 * 16 * 62.5 = 3Gb/s \\ 16-\text{QAM} &: 4 * 16 * 62.5 = 4Gb/s \\ 32-\text{QAM} &: 5 * 16 * 62.5 = 5Gb/s \\ 64-\text{QAM} &: 6 * 16 * 62.5 = 6Gb/s \end{aligned}$$

| | |
|---|---|
| Slice Registers | 5% |
| Slice LUTs | 7% |
| LUTs Used as Logic | 6% |
| Occupied Slices | 17% |
| Unused Flip Flop | 28% |
| Unused LUTs | 51% |
| Fully Used LUT-FF pairs | 19% |
| Bounded IOBs | 46% |
| BUFG-BUFGCTRLs | 6% |
| DSP48E1s | 57% |

Figure 34: Fabric - LUTs Resources

From the information displayed in Tab.34, we observe that the simulated design with `N=16` requires more than 50% of the DSP48E1s logic units. Therefore, running a place and route with `N=32` will mandatorily fail.

Finally, Fig.35 displays the maximum achievable clock speed respect to the entered time constraints for the optimised design and consists in the final result of this work.

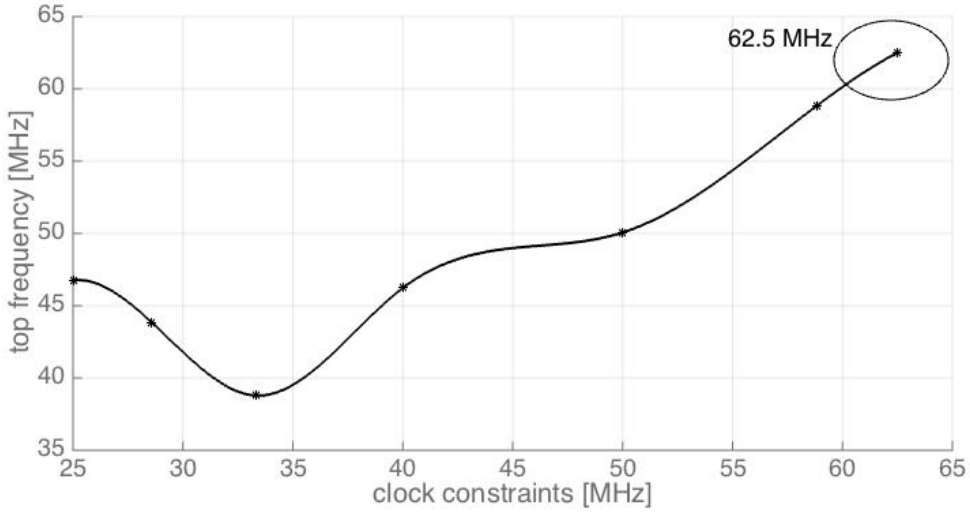



Figure 35: Fabric - Mults Performances

# 7 Conclusion And Further Improvements

This paper described a complete new way to optimise the performance of high-speed Quadrature Amplitude Modulation (QAM) implemented on FPGAs by exploiting the advantageous properties of a mixed time and frequency domain approach.

### In this Work

While standard transmitters operating entirely in time domain need to process serial data due to the convolutional nature of the filtering operation, this mixed-domain transmitter has the theoretical capability to work with an arbitrary number of parallel inputs $N$.

The design has been simulated, synthesised, routed and tested on a Xilinx Virtex 7 FPGA kit with a precision of 16 bits, for $N = 16$ parallel inputs and for multiple QAM formats; i.e. 8-QAM, 16-QAM, 32-QAM and 64-QAM. However, the concept can be generalised to more parallel inputs and other modulation formats.

After a long place and route operation, a top frequency of 62.5 MHz has been reached while processing 16 parallel inputs with a carrier frequency of 100 Hz. Therefore, this implementation offers an effective speed of exactly 1 GHz. This result is remarkable when compared to the current state-of-the-art for the target technology. In particular, when comparing this modulator with the work referenced in [5] reaching 625 MHz but completely deprived of modularity or the system realised this year by *E2v Semiconductors* achieving 750 MHz but without filter.

In addition to the high achieved performances, the realised system is extremely generic. Indeed, an arbitrary number of filter coefficients can be considered and the number of parallel inputs $N$ as well as the QAM modulation format are initially choose by the user though the core parametrisation.

### Next Works

Nevertheless, the results achieved in this work are only preliminaries. Indeed, multiples way could optimise and improve the current system.

First of all, the Discrete Fourier Transform (DFT) can be replaced by the Fast Fourier Transform (FFT), as depicted in appendix A. More specifically, a deep improvement could consist either in implementing a FFT core or in waiting for the new Xilinx FFT IP core. Nevertheless, even if replacing the DFT by the FFT is a huge improvement for large input sizes $N$, this system has been tested for a relative small number of parallel inputs ($N = 16$) and therefore, in this particular case, the FFT wouldn't improve the performances dramatically.

Secondly, the current system uses 57% of the DSP48E1s when ran with $N = 16$ and, by consequence, it cannot work for much larger $N$. Therefore, a possible further optimization could consist in reducing the amount of DSP48E1s used by the design.

Finally, a further optimised system could support other QAM (and non-QAM) modulation formats.

# Appendices

# A APPENDIX. Fast Fourier Transform

This first appendix illustrates an algorithm to efficiently perform the DFT operation. Even though many algorithms to implement the Fourier Transform exists, the most preferred and generally used is called Fast Fourier Transform (FFT). FFT is an algorithm to efficiently implement the Discrete Fourier Transformation (see section 3.2.1). Indeed, the computational complexity of DFT is $\mathcal{O}(N^2)$ while FFT has a complexity of $\mathcal{O}(Nlog_2(N))$.

FFT exploits the the symmetry of the exponent $e^{-j2\pi kn/N}$ (see Eq.(4)) and aims to build smaller DFT from a big one. In that purpose, considering $N = 2^m$ (with $m \in \mathcal{N}$) and defining

$$W_N = e^{-j2\pi/N} \tag{14}$$

we can separate $x[n]$ into an even and an odd-indexed subsequence as follows:

$$X[k] = \sum_{0}^{N-1} x[n]W_N^{kn} \tag{15}$$

$$= \sum_{neven} x[n]W_N^{kn} + \sum_{nodd} x[n]W_N^{kn} \tag{16}$$

Next, we define even and odd indices as $2r$ and $r = 2r + 1$, respectively ($r = 0, 1, \ldots, N/2 - 1$) and we rewrite Eq.(21) as:

$$X[k] = \sum_{r=0}^{N/2-1} x[2r]W_N^{k(2r)} + \sum_{r=0}^{N/2-1} x[2r+1]W_N^{k(2r+1)} \tag{17}$$

$$= \sum_{r=0}^{N/2-1} x[2r](W_N^2)^{kr} + W_N^k \sum_{r=0}^{N/2-1} x[2r+1](W_N^2)^{kr} \tag{18}$$

Finally, noticing that

$$W_N^2 = e^{-j2\pi/(N/2)} = W_{N/2} \tag{19}$$

we obtain:

$$X[k] = \underbrace{\sum_{r=0}^{N/2-1} x[2r](W_{N/2})^{kr}}_{X_e[k]} + \underbrace{W_N^k \sum_{r=0}^{N/2-1} x[2r+1](W_{N/2})^{kr}}_{X_o[k]} \tag{20}$$

$$= X_e[k] + W_N^k X_o[k] \tag{21}$$

where $X_e[k]$ and $X_o[k]$ are respectively the $N/2$ DFT of the even and the odd samples. Therefore, Eq.(21) express $X[k]$ as the sum of two $N/2$ points DFT.

The DFT has now been split in two, the next step is then continuing to split the DFT for $p = log_2(N)$ times. Fig.36 depict an example of Fast Fourier Transform execution for the particular case of $N = 8$ [14]. It is to note that the time-domain inputs has to be inserted in bit reversed order. To take back the signal from the frequency domain to the time-domain, the reverse algorithm can be applied.

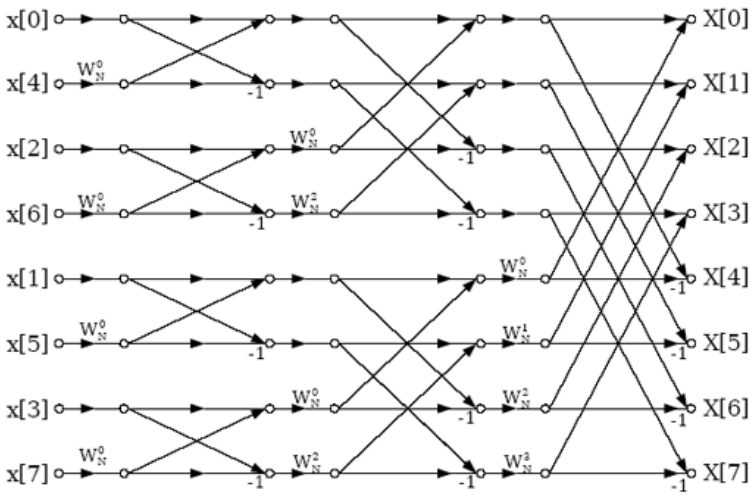
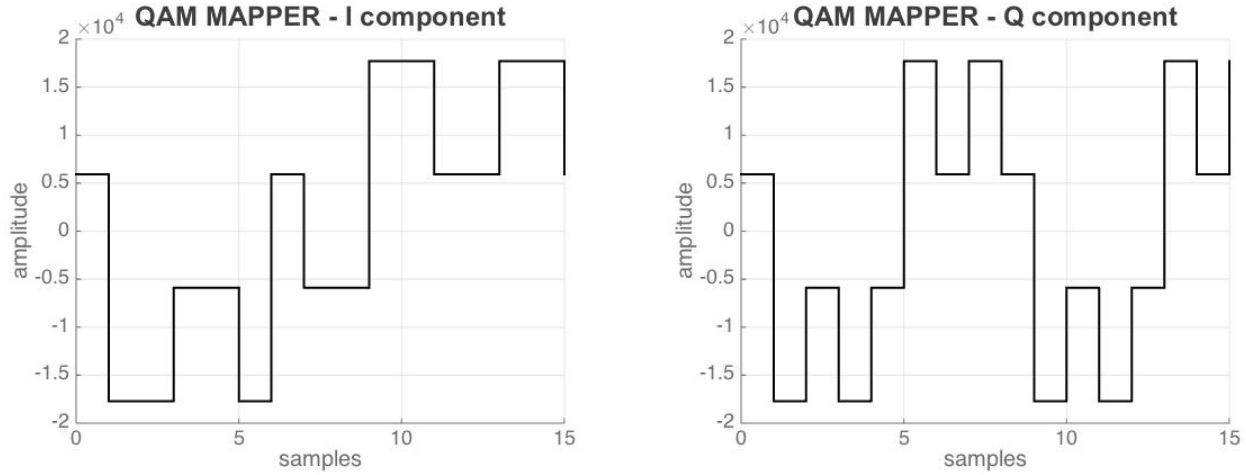



Figure 36: FFT execution (example with $N = 8$) [14]

This last algorithm is called Inverse Fast Fourier Transform (IFFT) and its derivation is not explained because of its similarity with the FFT algorithm.

# B APPENDIX. Design Precision Analysis

This appendix displays some graphs comparing the reference MATLAB simulation with the actual implemented system in the goal of proving the precision of the realised modulator. All graphs have been generated considering the filter coefficients given in Tab.12 of section 3.3.2, a carrier frequency of 100 Hz and the set of 16 random parallel inputs being mapped as shown here below.




(a) I test compnent

(b) Q test compnent

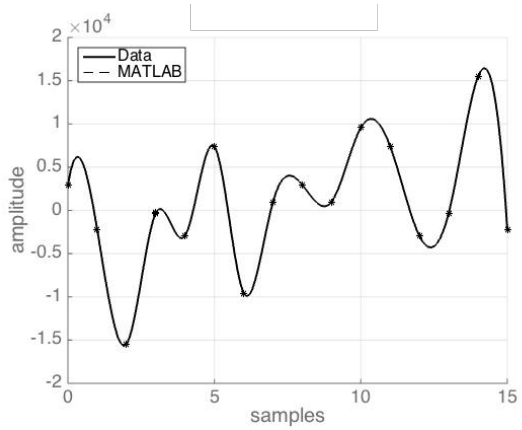
Figure 37: Test input signal

This appendix is divided in two parts. The first part illustrates the precision of the Discrete Fourier Transform (DFT) and Indirect Discrete Fourier Transform (IDFT) blocks while the second is devoted to the filter's and modulator's precision.

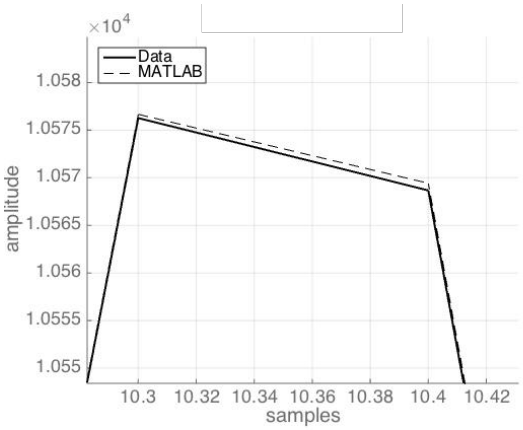
## B.1 Design Precision Analysis - DFT and IDFT

This first section of the appendix presents the amplitude of the DFT's and IDFT's output computed through MATLAB and through the implemented transmitter in Figs.38 and 39, respectively. Each of these figures is divided into six plots.

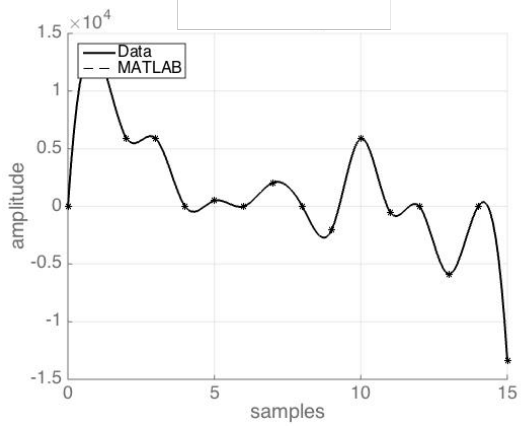
The upper two are dedicated to the real part. Both the reference MATALB result and the hardware system's result are plotted on the same graph and shown in the figure on the left. However, only one curved is visible in these figures due the graph's proximity. For that reason, the figure on the right presents a zoom of the main peak of the function in order to visually discriminate the curves. The two graphs in the middle have exactly the same purpose of the two upper but handle the imaginary part. Again, the graph on the left plots both the MATLAB and the data values on the same figure and, the right plot zooms on it. Finally, the two lower plots are the magnitude of the error; i. e. the difference in absolute value between the reference MATLAB simulation and the implemented hardware system.



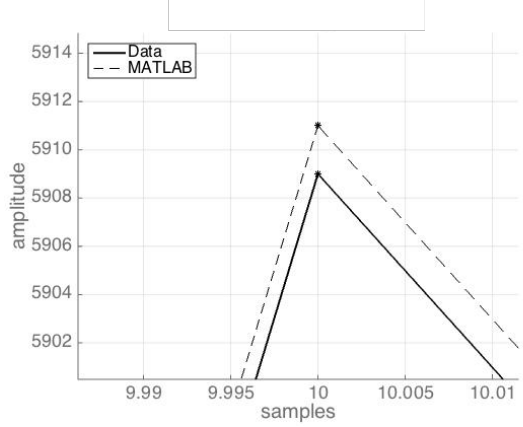

(a) DFT's real part



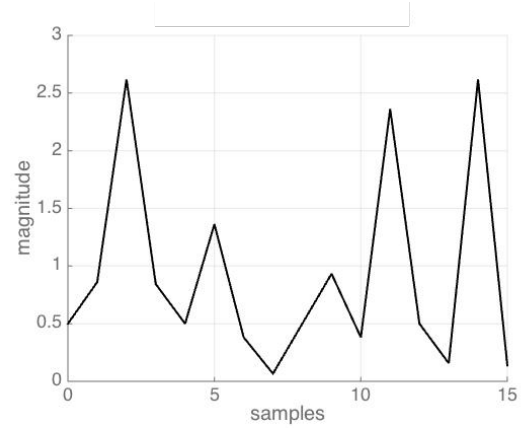

(b) DFT's real part - Zoom



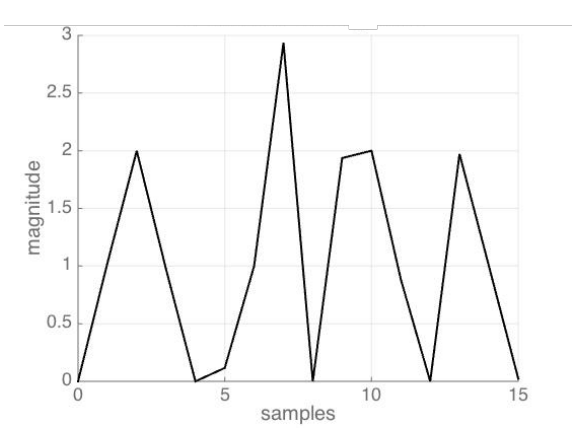

(c) DFT's imaginary part




(d) DFT's imaginary part - Zoom

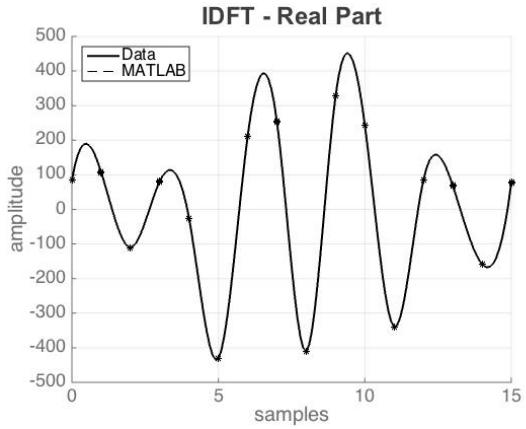



(e) DFT's real part error

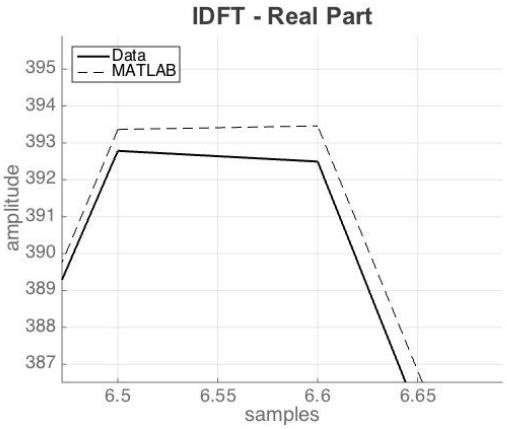



(f) DFT's imag. part error

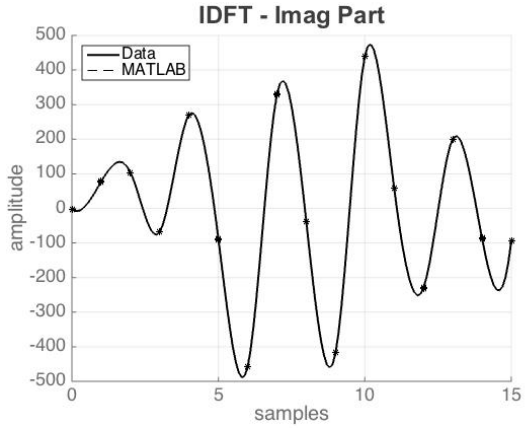
Figure 38: DFT's precision comparison



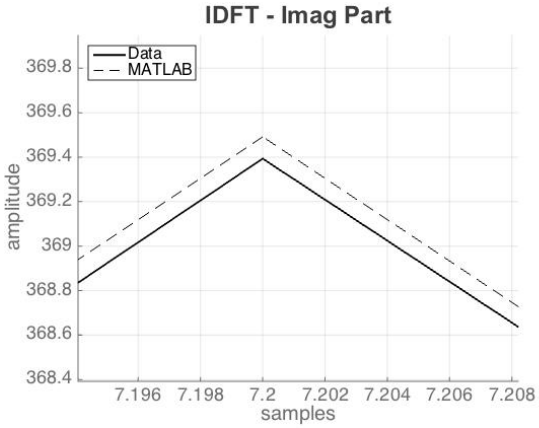

(a) IDFT's real part



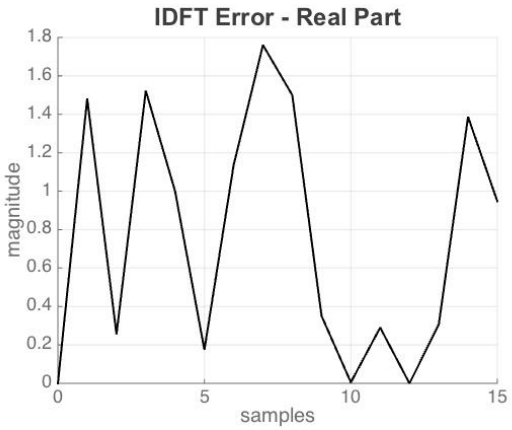

(b) IDFT's real part - Zoom



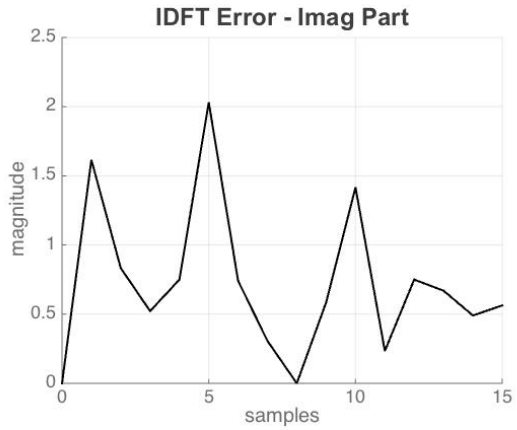

(c) IDFT's imaginary part




(d) IDFT's imaginary part - Zoom




(e) IDFT's real part error



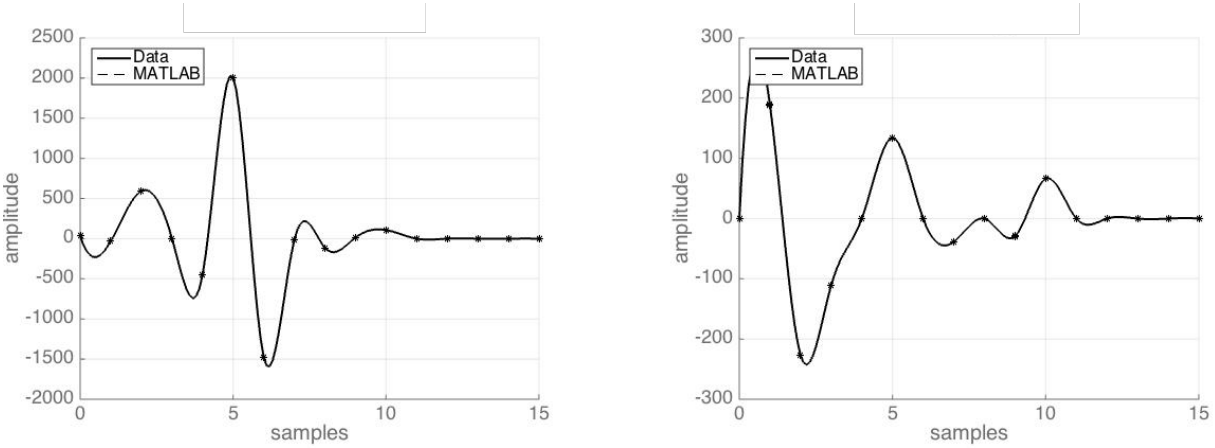

(f) IDFT's imag. part error

Figure 39: IDFT's precision comparison

## B.2 Design Precision Analysis - Filter and Modulator

This last part of the appendix is dedicated to the precision's illustration of the filter and the modulator blocks. Fig.40 shows the filter's real and imaginary part while Fig.41 shows the modulator's output. Once more, the very high system's precision causes the curves to be almost indistinguishable.

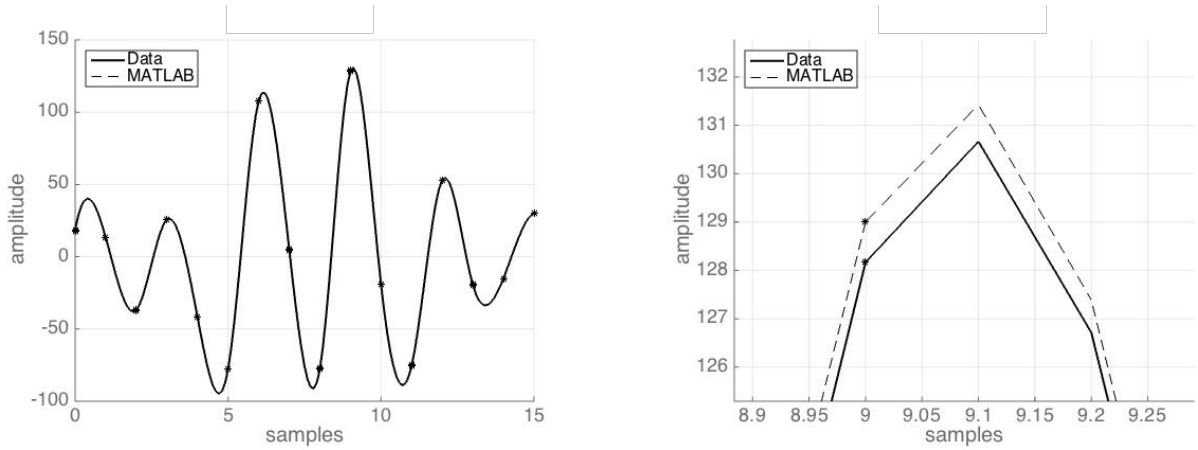
The fact that the filter operation is realised in the Fourier domain (see section 4), implies that this block is only composed of multipliers. Therefore the block's precision is exactly the same as the precision of the Xilinx IP core `Multiplier v11.2` implementing it and its analysis is beyond the scope of this work. For that reason, a zoomed versions of the filter's graphs is not provided.




(a) Filter real part

(b) Filter imaginary part

Figure 40: Filter precision comparison




(a) Modulator output

(b) Modulator output - Zoom

Figure 41: Modulator precision comparison

# Bibliography


[1] Yongbin Wu and Yousef R. Shayan,
*Implementation of High-Speed Multi-Level QAM Modems Based On Xiilnx Virtex-II FPGA*,
Departement of Electrical and Computer Engineering
Concordia University, Montreal, Quebec, Canada

[2] Xuan-Thang Vu, Nguyen Anh Duc and Trinh Anh Vu,
*16-QAM Transmitter and Receiver Design Based on FPGA*,
Electronics &Telecommunication Faculty
Hanoi University of Technology, Hanoi, Vietnam

[3] Vadim Smolyakov, Dimpesh Patel, Mahdi Shabany and P. Glenn Gulak,
*A WiMAX/LTE Compliant FPGA Implementation of a High-Throughput Low-Complexity 4x4 64-QAM Soft MIMO Receiver*,
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada

[4] Siqiang MA and Yong'en CHEN,
*FPGA Implementation of High-throughput Complex Adaptive Equalizer for QAM Receiver*,
Communication Software and ASIC Design Center
Tongji University, Shanghai, China

[5] A. Al-Bermani, C. Wordehoff, O. Jana, K. Puntsria, M. F. Panhwara, U. Ruckert and R. Noé,
*The Influence of Laser Phase noise on Carrier Phase Estimation of a Real- Time 16-QAM Transmission with FPGA Based Coherent Receiver*,
University of Paderborn, Paderborn, Germany and Bielefeld University, Bielefeld, Germany

[6] M. Stackler, A. Glascott-Jones, N. Chantier,
*A high speed transmission system using QAM and direct conversion with high bandwidth converters*,
E2v Semiconductors

[7] Shalina Percy George Ford, Peter Figuli and Juergen Becker,
*Parametric Design Space Exploration for Optimizing QAM Based High-speed Communication*,
IEEE/CIC International Conference on Communications in China, 2015

[8] M. Ferrario, A. Spalvieri and R.Valtolina,
*Design of transmit fir filters for fdm data transmission systems*,
Communications, IEEE Transactions on, vol. 52, no. 2, Feb 2004

[9] Ian Poole,
*Comparison of 8-QAM, 16-QAM, 32-QAM, 64-QAM, 128-QAM, 256-QAM, Types*,
Radio-Electronics,
*http://www.radio-electronics.com/info/rf-technology-design/quadrature-amplitude-modulation-qam/8qam-16qam-32qam-64qam-128qam-256qam.php*

[10] NI AWR Design Environment 12,
*Visual System Simulator System Block Catalog*,
2015

[11] Online Electrical Engineering,
*Gray Code — Binary to Gray Code and that to Binary Conversion*,
*http://www.electrical4u.com/gray-code-binary-to-gray-code-and-that-to-binary-conversion/*

[12] Ravindra H. Sharma, Dr. Kiritkumar R. Bhatt,
*A Review on Implementation of QAM on FPGA*,
International Journal of Innovative Research in Computer and Communication Engineering on Vol. 3, Issue 3, March 2015

[13] P.J. Bevel,
*The Fourier Transform*,
Copyright 2010 TheFourierTransform.com,
*http://www.thefouriertransform.com*

[14] Douglas L. Jones,
*Decimation-in-time (DIT) Radix-2 FFT*,
*http://cnx.org/contents/ce67266a-1851-47e4-8bfc-82eb447212b4@7/Decimation-in-time-DIT-Radix-2*