UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Alcatel·Lucent Bell Labs Research

Implementation of DSP algorithms in VHDL for high-speed optical communications

A Degree's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Sergi Caelles Prat

In partial fulfilment

of the requirements for the degree in

Science and Telecommunication Engineering

Advisors:

Dr. Joan M. Gené Bernaus

Dr. Sebastian Randel

Barcelona, July 2014

Abstract

The amount of traffic that backbone networks have to handle increases every year at about 30 to 60 %. A solution that has been used for the last two decades is the wavelength-division multiplexing (WDM) technique and more recently incorporating complex constellations in each wavelength, but there is not a lot of room for improvement in these systems. The main reason is because current experiments are close to the nonlinear Shannon limit for single-mode fibre (SMF), a problem known as the capacity crunch.

Researchers have to find other ways to keep increasing the bandwidths to meet the growth in the traffic demands. The approach where researchers are squeezing their minds is the space-division multiplexing (SDM). Among other problems that the researchers face in this new technology, there is the crosstalk among parallel SDM. The current approach to overcome this problem is using multiple-input-multiple-output (MIMO) techniques in the digital signal processing (DSP) part of the system. Actually, the DSP has been used since the development of coherent systems in 2010 made it possible, e.g., in the elimination of the crosstalk among different polarizations when we are transmitting using polarization division multiplexing (PDM).

In this thesis, which has been developed during the first four months of a seven month internship in Bell Labs located at Crawford Hill (USA), different algorithms that will be placed in the DSP part are presented. The main feature of the different designs is the possibility of using them in real time. Therefore, they are implemented in VHDL to be able to be run in an FPGA.

The algorithms that have been developed are a digital phase shifter in order to interpolate digital signals, a 64-point discrete Fourier transform and a frequency domain filter. These are the first steps for the final implementation of a complete real time MIMO receiver. This will be an important achievement as currently there are no publications that use a real time MIMO receiver in the DSP part of an optical system. All the different implementations have been tested with satisfactory results. However, some more optimizations will have to be done in the future to squeeze the complete MIMO receiver in a relatively large FPGA.

Resum

El tràfic que han de suportar les xarxes troncals s'incrementa cada any entre un 30 i un 60 %. La solució actual i que s'ha estat fent servir en les últimes dues dècades és la "wavelength-division multiplexed" (WDM) a la qual recentment s'hi ha incorporat constel•lacions complexes a cada longitud d'ona, però malauradament no hi ha molt marge de millora en aquesta direcció. El principal motiu és que els experiments actuals estan molt propers al límit no lineal de Shannon per fibres òptiques mono mode, aquest problema és coneix com el "capacity crunch".

Els investigadors han de trobar noves maneres d'incrementar la capacitat per poder assolir l'increment de tràfic a les xarxes. La solució en la qual s'estan centrant les seves forces s'anomena "spatial-division multiplexing" (SDM). Entre molts altres problemes que s'han de fer front amb aquesta nova tecnologia , hi ha la diafonia entre els SDM paral•lels. La solució actual a aquest problema és l'ús de la tècnica "multiple-input-multiple-output" (MIMO) a la part de processat del senyal (DSP) del receptor. De fet, el DSP s'ha fet servir des de l'aparició del receptors coherents a l'any 2010, per exemple, es fa servir per eliminar la diafonia que hi ha entre les diverses polaritzacions quan estem transmetent amb "polarization division multiplexing" (PDM).

En aquesta treball de fi de grau, el qual ha estat desenvolupat durant els quatre primers mesos dels set en que consisteix l'estada als Bell Labs situats a Crawford Hill (USA), diversos algoritmes que funcionaran a la part de DSP seran presentats. La principal característica dels diversos dissenys és la possibilitat de poder ser implementats en temps real, per aquest fet, estaran implementats en VHDL per a poder fer-los funcionar en una FPGA.

Els principals algoritmes que s'han desenvolupat són un "digital phase shifter" per a poder interpolar senyals digitals, una implementació de 64 mostres de la transformada discreta de Fourier i un filtre en freqüència. Aquets són els primers passos per a poder assolir l'objectiu final, la implementació d'un receptor MIMO complet. Aquest és un fet rellevant, ja que, actualment en cap publicació es fa servir a la part de DSP d'un sistema de fibra òptica un receptor MIMO que funcioni en temps real. Tots els algoritmes han estat testejats amb resultats satisfactoris. Tot i així, s'hauran de realitzar més optimitzacions per fer encabir el receptor MIMO complert en una FPGA.

Resumen

El tráfico que tienen que soportar las redes troncales aumenta cada año entre un 30 y un 60 %. La solución actual y que se ha estado utilizando en las dos últimas décadas es la "wavelength-division multiplexed" (WDM) a la cual recientemente se ha incorporado constelaciones complejas en cada una de la longitudes de onda, pero desgraciadamente no se puede continuar en esta dirección. El principal motivo es que los experimentos actuales están muy próximos al límite no lineal de Shannon para fibras ópticas mono modo, este problema se conoce como el "capacity crunch".

Los investigadores deben encontrar un nuevo modo de incrementar la capacidad para poder compensar el incremento de tráfico en las redes. La solución en la cual están centrando sus esfuerzos se denomina "spatial-division multiplexing" (SDM). Entre los problemas que se han de hacer frente con esta nueva tecnología, está la diafonía entre los SDM paralelos. La solución actual a este problema es el uso de la técnica "multiple-input-multiple-output" (MIMO) en la parte del procesado de señal (DSP) del receptor. De hecho, el DSP se ha utilizado desde la aparición de los receptores coherentes en el año 2010, por ejemplo, se utiliza para eliminar la diafonía que existe entre las diversas polarizaciones cuando estamos transmitiendo con "polarization division multiplexing" (PDM).

En este trabajo de fin de grado, el cual ha sido desarrollado durante los cuatro primeros meses de los siete que durará la estancia en los Bells Labs situados en Crawford Hill (USA), diversos algoritmos que funcionan en la parte de DSP serán presentados. La principal característica de los varios diseños es la posibilidad de poder ser implementados en tiempo real, para ello, estarán implementados en VHDL para que puedan funcionar en una FPGA.

Los principales algoritmos que se han de desarrollado son un "digital phase shifter" para poder interpolar la señal digital, una implementación de 64 muestras de la transformada discreta de Fourier y un filtro en frecuencia. Estos son los primeros pasos para poder alcanzar el objetivo final, la implementación del receptor MIMO completo. Cabe destacar la importancia de esta implementación, ya que, actualmente en ninguna publicación se usa en la parte de DSP de un sistema de fibra óptica un receptor MIMO que funcione en tiempo real. Todos los algoritmos se han testeado con resultados satisfactorios. A pesar de ello, se tendrán que realizar más optimizaciones para poder meter todo el receptor MIMO en una FPGA.

Acknowledgements

In the first place, I would like to thank several people from Bell Labs where I have been undergoing my internship. Firstly, my supervisor, Sebastian Randel, as he has always guided me when I did not know exactly how to continue something and he has provided me topics to research and develop, also, Steve Corteselli who has always helped when I had troubles writing VHDL code. Last but not least, I would like to thank Peter Winzer as he was the person who made the internship possible and all the other colleagues that helped me or that involved me in the different activities done by the company.

I would also like to thank my colleague and roommate, Dario Pilori, as he has always explain me any doubt that I had in anything, and my other roommate, Sai Chen, for the great moments that we have had the three of us experiencing the American lifestyle.

From UPC Barcelona, I would like to acknowledge Joan M. Gené Bernaus for giving me the possibility of having this great experience in the US and helping me through the whole process of getting there.

And finally, I would like to thank my parents, brother and family for the trust they have always bestowed on me.

Revision history and approval record

Revision	Date	Purpose
0	17/06/2014	Document creation
1	19/06/2014	Document revision
2	22/06/2014	Document revision
3	02/07/2014	Document revision
4	10/07/2014	Final document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Sergi Caelles Prat	scalles@gmail.com
Joan M. Gené Bernaus	joan.gene@upc.edu
Sebastian Randel	sebastian.randel@alcatel-lucent.com

Written by:		Reviewed and approved by:	
Date	11/07/2014	Date	11/07/2014
Name	Sergi Caelles Prat	Name	Joan M. Gené Bernaus
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

1. Introduction

In the typical sources that our society has always relied on like water, fossil fuels or electricity, nowadays there is another one which importance is growing, the speed at which the data can be transmitted. Since the 1970s when the optical fibre replaced the coaxial cable for long-haul transport, a lot of research and development has been done in the optical field. Already in the 1980s, the single-mode fibre (SMF) was found to be a better solution as the absence of modal dispersion allowed a wider bandwidth. From that days to nowadays, almost all the research has focused in increasing its capacity using the different dimensions available in the fibre.

However, in recent experiments the limit in the capacity of the SMF is being reached. In 2010, Essiambre and his co-authors [1] found that the capacity limit for the SMF was below the general capacity limit that was proved by Shannon [2] in 1948. The main problem that reduces the Shannon capacity is the non-linear effects that appear in the fibre when the power transmitted increases in order to achieve longer distances. This problem has been stated as the "capacity crunch" and it is a major problem that the researchers will have to face in the following years.

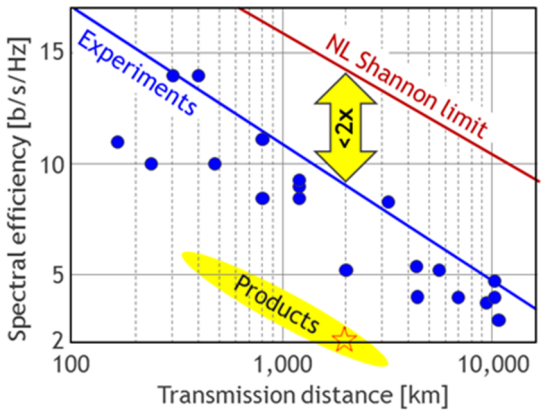


Figure 1. Comparison of the current experiments with the NL Shannon limit [3].

The amount of traffic that the backbones networks have to handle increases every year at about 30 to 60 % [4]. The increasing number of applications relying on machine-to-machine traffic and cloud computing could accelerate even more the capacity demand. In order to keep increasing the bandwidths to meet the increase in traffic demands, the researchers are currently focusing in exploiting the SDM.

Actually, all the different dimensions that we can find in a fiber are exploited except the SDM. The time domain is exploited sending one symbols after the other, pulse shaping may be used to compress the spectrum and multilevel modulation to carry more bits in a symbol, but not much more things can be done. The quadrature dimension is also used with two-dimensional symbol alphabets, the size of them can be increased but the limit is being reached. In the frequency domain, multiple communication signals are being transmitted in parallel on distinct carriers on the same transmission medium, known as WDM, but the bandwidth available is limited for physical constraints as the signals are mainly transmitted at the C-band. The polarization is also being used when we are using coherent optical communications to realize simultaneous transmission of multiple streams. The spatial dimension is the only one that is not completely exploited yet, the improvements

could be in a variety of techniques from the deployment of parallel optical fibers to the usage of multimode fibers to transmit data in the different modes or the usage of multicore fibers.

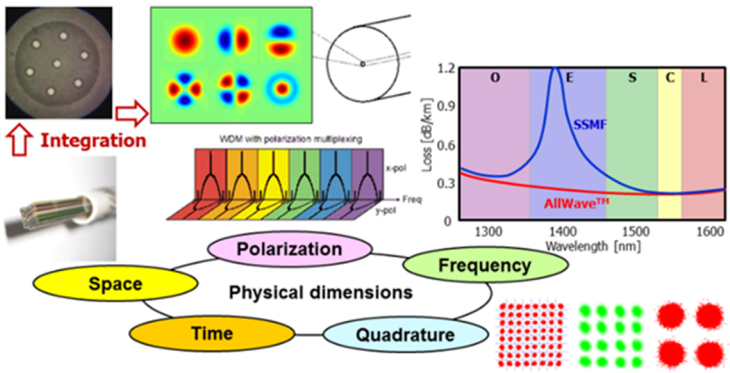


Figure 2. Different dimensions available on the fibre. [3]

Taking a look at the evolution of the aggregate per-fibre capacity along the years, we can see another proof that the SDM is the future as capacity increases in WDM have slowed down to about 20 % per year since 2002.

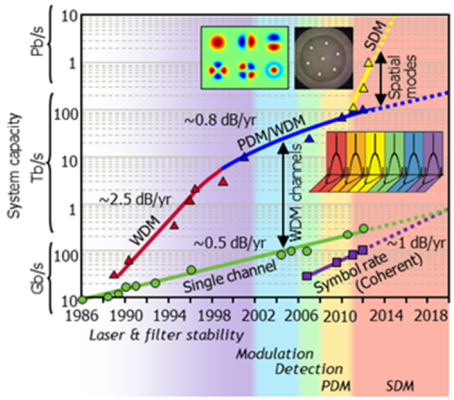


Figure 3. Evolution of the system capacity with the different technologies. [3]

1.1. Spatial Division Multiplexing

The most trivial solution using SDM is by using M parallel optical lines systems, this solution is scalable but it is not economically sustainable as the capacity will be M times bigger at M times the cost and energy consumption. The SDM technology is then expected to scale the capacity with a similar cost and energy as WDM. The researches will face a bunch of problems that they will have to solve like the compatibility, the integration and the crosstalk.

Regarding the compatibility, operators will want to make use of the current infrastructure so hybrid network architectures that use parallel fiber stands on some spans and possibly new SDM-specific fiber on other spans must be supported.

The integration is a key aspect to follow the historic trend and reduce the energy consumption per transmitter by about 20% per year [5]. The integration can take place in different parts of the systems, on a system level, on a network level, on a transponder level, on amplifier level and on a fiber and splice/connection level.

The integration of hardware at various levels will inevitably result in crosstalk among parallel SDM paths and consequently in transmission penalties. If the crosstalk exceeds a tolerable limit, a mitigation of it will have to be implemented. This can be done by adapting multiple-input multiple-output (MIMO) techniques that have been original developed for wireless systems.

1.2. MIMO DSP

First of all, the DSP is possible due to the development of the coherent systems in 2010. Currently, the MIMO systems that are being used are 2x2 MIMO and are used in the structure of the PDM coherent receiver. In-phase (I) and quadrature (Q) components of the received x' and y' polarizations are first converted into the digital domain typically through over-sampled analog-to-digital convertors and then expressed as complex sample streams, $I_{x'}+jQ_{x'}$ and $I_{y'}+jQ_{y'}$. After applying front-end corrections that compensate for various imperfections of the optoelectronic receiver, chromatic dispersion (CD) of the transmission link is compensated by digitally applying the inverse CD filter function. The following block that we find at the system is the butterfly filter, in the particular implementation for the PDM system, this will be given by the 2x2 Jones matrix and the matrix elements are implemented as individual filters, typically with a few tens of filter taps. The coefficients will be update using an adaptive algorithm such as blind constant modulus (CMA) or various forms of blind decision-direct algorithms like the least-mean square (LMS) algorithm.

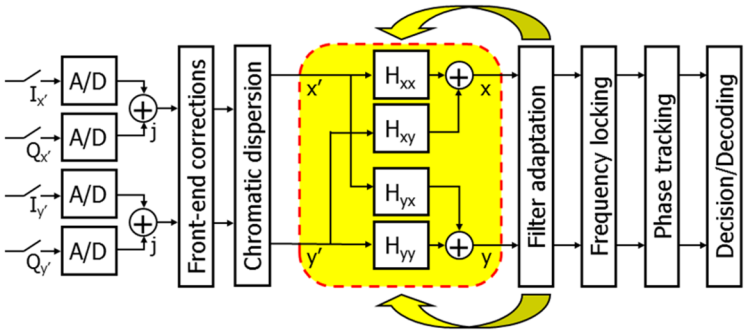


Figure 4. MIMO coherent receiver structure. [3]

In the generalization of this structure to the MxM MIMO receiver, the only block that will change will be the butterfly equalizer. The equalizer will have to deal with an MxM channel based on an appropriate channel estimation or filter adaption algorithm. In the former case, the channel matrix is estimated and inverted to obtain the equalizer coefficients based on the zero forcing criterion or the minimum-mean-square-error (MMSE) criterion. In the latter case, gradient-based coefficient adaption algorithms are used to directly adapt the filter coefficients to approach the MMSE solution.

The work that has been developed in this thesis is present in some of the blocks that are showed in the previous figure. All the different designs have been implemented in VHDL in order to be able to run them in an FPGA in real time. First of all, an implementation of an interpolation is presented and is called digital phase shifter. The corresponding block will be the front-end corrections as the main purpose is to compensate the skew between the ADCs. The second algorithm implemented is the Fast Fourier Transform for 64-point input. This block is not meaningful by itself, but it is a powerful tool that is used in a lot of different algorithms. The last design implemented is a frequency domain filter (FDF) that will use the FFT blocks already developed, different version are presented in order to suit different needs. This is the first step to build the whole system of the implementation of the MxM MIMO receiver.

2. Digital phase shifter

2.1. Definition of the system

The digital phase shifter is a block in the DSP part that will be able to delay the signal in the digital domain with a fractional delay between 0 and 1 sample. Delays of an integer number of samples are really easy to implement with a simple buffer so the challenge was to implement non-integer delays. This fractional delay can be useful in the transmitter/receiver to compensate delays between the in-phase and quadrature components as they are generated with different DAC/ADC or in general for any other synchronization issue. It has to be taken into account that in the experiments high data rates are used and a small delay between the different components might result in an important problem.

2.2. State of the art

In order to implement the digital phase shifter, an interpolation of the signal is required to compute intermediate values between signal samples. There are many different algorithms to perform it. When deciding which one to use several characteristics have to be taken into account like the simplicity and the performance, but the most important one is that it has to be synthesizable in hardware as the final algorithm will be running in an FPGA. The decision of which algorithm should be used was made prior to this project and the main characteristics of the chosen algorithm are presented here. A cubic Lagrange interpolator using the Farrow structure was the selected algorithm, a brief introduction is given and more information can be found at [6]. The Farrow structure is suitable for a hardware implementation and the resource usage of cubic order should be low enough to fit in a FPGA.

If we use the classical Lagrange interpolation approach, the solution can be stated as follows. Given a function x(t) defined for $t_{-(N-1)},\ldots,t_0,\ldots,t_N$ there exist a Lagrange polynomial of degree 2N-1:

$$P(t) = \sum_{n=-(N-1)}^{N} \lambda_n \left[(t - t_{-(N-1)}) \cdots (t - t_{n-1})(t - t_{n+1}) \cdots (t - t_N) \right] x(t_n) \qquad (2.1)$$

with

$$\lambda_n = \frac{1}{(t_n - t_{-(N-1)}) \cdots (t_n - t_{n-1})(t_n - t_{n+1}) \cdots (t_n - t_N)} \qquad (2.2)$$

The polynomial P(t) is a linear combination of the values of $x(t_n)$,

$$P(t) = \sum_{n=-(N-1)}^{N} q_n(t) x(t_n) \qquad (2.3)$$

We are only interested in values $0 \leq t \leq 1$ so we can set $t=\mu$ in the definition of the Lagrange coefficients $q_n(t)$. Then the expression of $q_n(t)$ will be:

$$q_n(\mu) = \sum_{m=0}^{2N-1} d_m(n)\, \mu^m \qquad (2.4)$$

which is a polynomial in μ. If we add this to (2.1) the result will be:

$$P(\mu) = \sum_{m=0}^{2N-1} \mu^m \left[\sum_{n=-(N-1)}^{N} d_m(n)\, x(t_n) \right] \tag{2.5}$$

Then, it can be noticed that P(μ) can be computed as the output of a Farrow structure. The final expressions for the Lagrange coefficients if we particularize for the cubic case will be:

$$q_{-1}(\mu) = -\frac{1}{6}\mu^3 + \frac{1}{2}\mu^2 - \frac{1}{3}\mu \tag{2.6}$$

$$q_0(\mu) = \frac{1}{2}\mu^3 - \mu^2 - \frac{1}{2}\mu + 1 \tag{2.7}$$

$$q_1(\mu) = -\frac{1}{2}\mu^3 + \frac{1}{2}\mu^2 + \mu \tag{2.8}$$

$$q_2(\mu) = \frac{1}{6}\mu^3 - \frac{1}{6}\mu \tag{2.9}$$

If we represent the Farrow structure as a FIR filter:

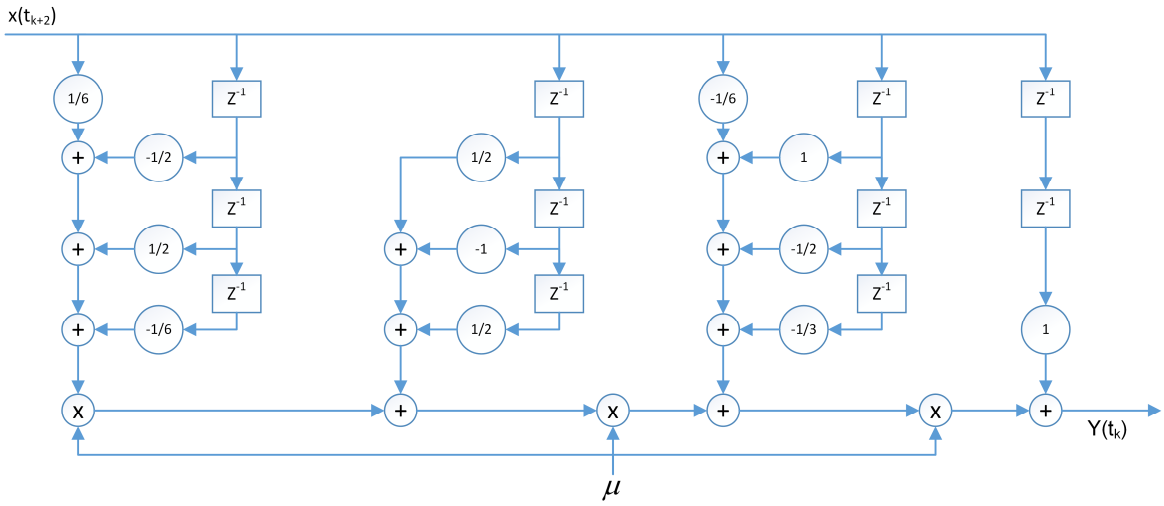


Figure 5. Farrow structure

The final result that will be used in the implementation step is depicted in the following table. In each column, we can find the different samples that will be multiplied by each power of the μ.

μ^3	μ^2	μ	1
$-\frac{1}{6}x(t_{n-1})$	$\frac{1}{2}x(t_{n-1})$	$-\frac{1}{3}x(t_{n-1})$	
$\frac{1}{2}x(t_n)$	$-x(t_n)$	$-\frac{1}{2}x(t_n)$	$x(t_n)$
$-\frac{1}{2}x(t_{n+1})$	$\frac{1}{2}x(t_{n+1})$	$x(t_{n+1})$	
$\frac{1}{6}x(t_{n+2})$		$-\frac{1}{6}x(t_{n+2})$	

Table 1. Samples corresponding to each power of the fractional delay

2.3. Methodology

The procedure for the implementation of the algorithm was the following. First, a Matlab code was developed to verify that the performance was the expected one. Then, another code in Matlab that simulated the hardware design was also developed to emulate some specific parameters of the hardware. Finally, the implementation in VHDL was made as well as a simulation to check if the results were the same as the ones obtained with the second Matlab code.

2.3.1. Matlab

The first Matlab approach, as said before, was developed just to have the algorithm working and test that the results were the expected. The function made for this purpose was called *deskew(x,d,cy)*. The input parameters are *x* which is the signal where the fractional delay has to be applied, *d* that is the fractional delay that has to be applied to the signal and *cy* to specify if the algorithm will work in a cyclic or non-cyclic way. In the non-cyclic approach, the samples that are unknown are simply padded with zeros, whether in the cyclic the last samples of the input are used for the first unknown samples.

Some examples of the obtained results are:

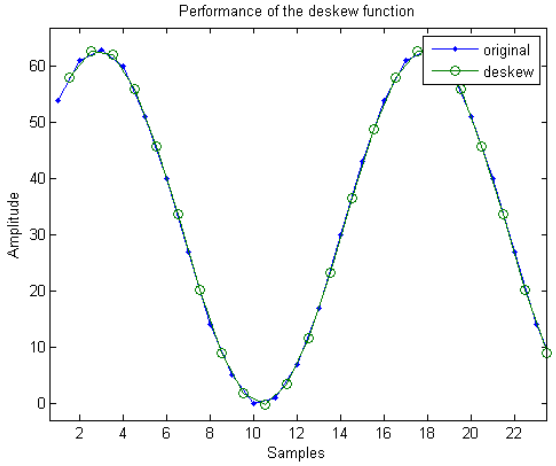


Figure 6. Performance of the deskew function with a sine wave

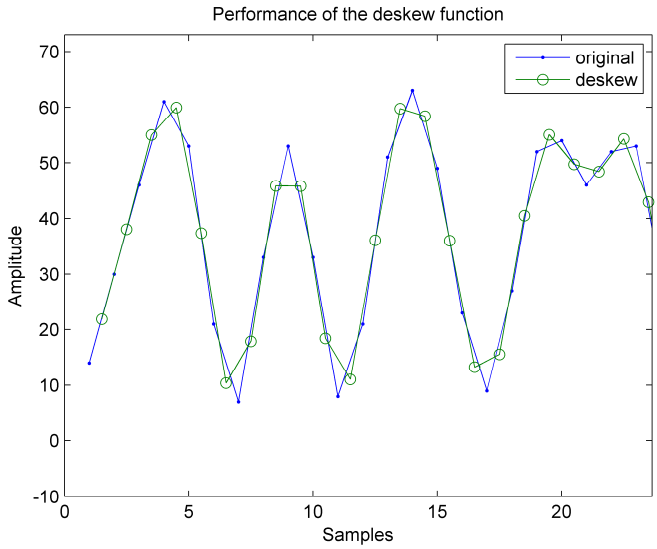


Figure 7. Performance of the deskew function with a QPSK signal

The implementation of the algorithm was verified doing some test with common functions like a sine wave, a pulses wave and real data generated from a QPSK signal. All the results were satisfactory so the next step was to build the algorithm in a more hardware friendly style.

In order to develop the second Matlab code, the fixed point toolbox for Matlab was used to simulate the behaviour of the hardware. Several important things had to be kept in mind when the algorithm was developed:

- Avoid divisions because they are expensive when we are talking of hardware resources usage.
- The interpolation of a signal may produce values higher than the current values of the signal, for example when the input is a sequence of pulses which amplitude is the maximum value of the input range. This can cause a clipping in the final result due to the range of the output signal must be the same as the input and have to represent the signal with the same number of bits.
- During all the operations, the range of the signal will increase and so will do the number of bits necessary to represent it. In order to minimize the resources usage we will have to figure out which is the number of bits that are used in each step of the algorithm.

To solve the first problem, all the equation of the Lagrange coefficients were multiplied by 6 removing in this way the fractional numbers. This can be seen as a gain that will not modify the signal and that at the end can be removed. Moreover, the delay in the original expressions for the Lagrange coefficients is also a fractional value between 0 and 1. Taking into account that a precision of 64 different values between 0 and 1 was considered enough, the delay has 6 bits and the values of the μ in our implementation will go from 0 to 63. In order to avoid the division per 64 in the expressions, all of them will be multiplied by 2^{3*6}. The final expressions for the Lagrange coefficients are:

$$q_{-1}(\mu) = 2^{3*6}\left[-\left(\frac{\mu}{64}\right)^3 + 3\left(\frac{\mu}{64}\right)^2 - 2\left(\frac{\mu}{64}\right)\right] = [-(\mu)^3 + 2^6 * 3(\mu)^2 - 2^{2*6} * 2(\mu)] \quad (2.10)$$

$$q_0(\mu) = 2^{3*6}\left[3\left(\frac{\mu}{64}\right)^3 - 6\left(\frac{\mu}{64}\right)^2 - 3\left(\frac{\mu}{64}\right) + 6\right] = [3(\mu)^3 - 2^6 * 6(\mu)^2 - 2^{2*6} * 3(\mu) + 2^{3*6} * 6] \quad (2.11)$$

$$q_1(\mu) = 2^{3*6}\left[-3\left(\frac{\mu}{64}\right)^3 + 3\left(\frac{\mu}{64}\right)^2 + 6\left(\frac{\mu}{64}\right)\right] = [-3(\mu)^3 + 2^6 * 3(\mu)^2 + 2^{2*6} * 6(\mu)] \quad (2.12)$$

$$q_2(\mu) = 2^{3*6}\left[\left(\frac{\mu}{64}\right)^3 - \left(\frac{\mu}{64}\right)\right] = [(\mu)^3 - 2^{2*6}(\mu)] \quad (2.13)$$

After all these manipulations of the original expressions, a gain of $6*64^3$ has been introduced and will be removed in the final step. Due to these modifications, the word length used in all the operations will be longer but we will avoid to work with fractional numbers which would have used more resources of the FPGA. Furthermore, the multiplications by a power of two are simple shifts of the bits to the left (if the MSB is the leftmost) and adding as many zero as the power of two on the right.

Regarding the second point, the clipping is handled in the last step when the relevant bits of the final sample have to be chosen. In order to remove the gain introduced we will divide for $2^{3*6}\cdot 2^4$. The original gain introduced was smaller than this value, but this was chosen mainly for two purposes. First of all, because if the division is by a power of two we only have to take the bits of the word starting on the power of two and taking from that to the number of bits that we want in the MSB direction. In our case, the beginning will be at the bit position 21 and the end at the 27 as the input and the output are 6 bits symbols. Secondly, dividing by a greater value than the gain we introduce, it will result in a smaller range in the output signal reducing in this way the possibility of clipping.

Finally, in order to figure out the number of bits necessaries in each of the operations, a first version of the code with a word length long enough to avoid any problems was developed. Then, the function was run with an input matrix with all the possible combinations and in each step of the algorithm, the greatest value was found. The final result will be depicted in a schematic in the VHDL section.

There is a last thing that has to be taken into account. In order to avoid problems with the DC, this is subtracted at the beginning of the algorithm changing the range of the input signal from 0…63 to -32…31. To compensate this modification, in the last operation before performing the division of the gain, the maximum negative values that was found when looking for the word length of each step is added to guarantee that the final result is positive.

The result is the function *intdelay(x,d)* whose output is the interpolated signal. The inputs are the *x* which is the signal where the interpolation is applied and the *d* which is the delay that we want for our signal. In this function the cyclic approach of the first Matlab code is implemented by default.

In Figure 8 and Figure 9 we can see two examples of the final result of the two different functions that have been implemented in Matlab, the general implementation of the algorithm and the hardware implementation. In the first one a QPSK and in the second one a sine wave are the signals interpolated. The behaviour of the *deskew* function was checked before so the main purpose of this plots is to check the *intdelay* function. The final result is the desired one as the output signal will have the same values with the only difference that the values outputting the *intdelay* are compressed to avoid the previous mentioned clipping that we can have at the output.

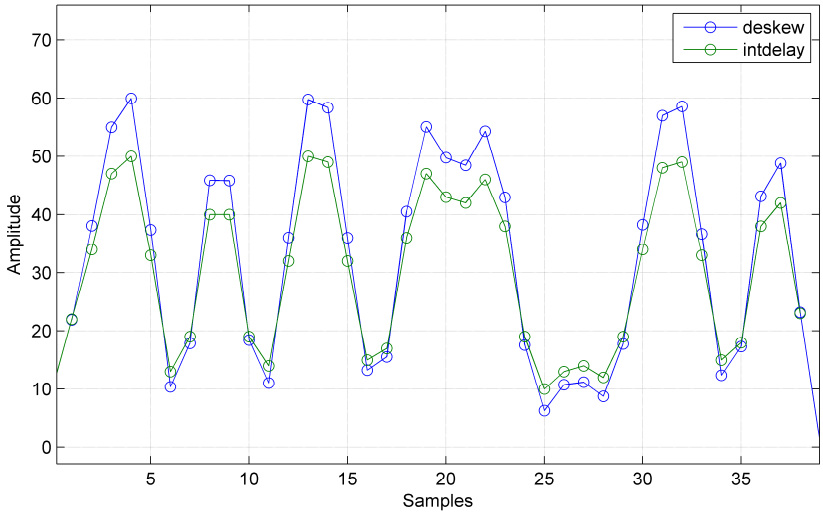


Figure 8. Comparison of the deskew and intdelay functions with a QPSK signal

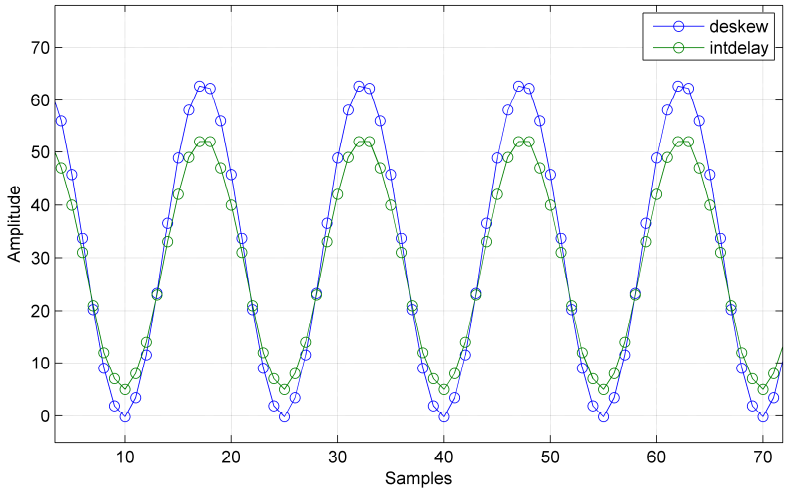


Figure 9. Comparison of the deskew and intdelay functions with a sine wave

2.3.2. VHDL

The implementation with VHDL once the second Matlab code was working properly should be more or less straightforward, but there are a few new things that have to be considered. First of all, in hardware the future samples are unknown so a small modification has to be introduced. Secondly, the operations cannot be done all at the same time like in Matlab, they have to be broken down in small parts to perform each of them every clock.

To solve the first problem, another interpretation was made. At the input the current sample was interpreted as the sample t_{n+2} of the algorithm. As a result, the output sample will be the one that would correspond to two clocks ago. In other words, the interpolated sample is delayed two clocks in respect to the input one. The solution of the second problem was to perform some operations in parallel and add delays to the signals that were not being used in the operation; the final result will be shown later.

The requirement was to make a block where in each clock at the input there are 256 consecutive samples of 6 bit each and at the output we should have the same, in a more graphical way:

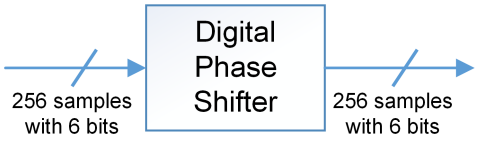


Figure 10. General schematic of the digital phase shifter

The approach was to generate two blocks, one small that performs the cubic interpolation with an input of four samples and another huge block that will combine the small ones to perform the interpolation in all the different samples that input the system in each clock.

The entity of the small block is:

```
entity phase_shifter_v4 is
        Port ( z0 : in STD_LOGIC_VECTOR (5 downto 0);
               z1 : in STD_LOGIC_VECTOR (5 downto 0);
               z2 : in STD_LOGIC_VECTOR (5 downto 0);
               z3 : in STD_LOGIC_VECTOR (5 downto 0);
               d : in STD_LOGIC_VECTOR (5 downto 0);
               clk : STD_LOGIC;
               y : out STD_LOGIC_VECTOR (5 downto 0));
end phase_shifter_v4 ;
```

where *z0*, *z1*, *z2*, *z3* are three consecutives samples (*z0* is the newest one and *z3* is the oldest one), *d* is the fractional delay with an integer representation between 0 and 63, *clk* the clock and *y* the output. To have a better idea of how the system looks like this is a schematic that represents the whole system as well as the number of bits that each signal has in every step:

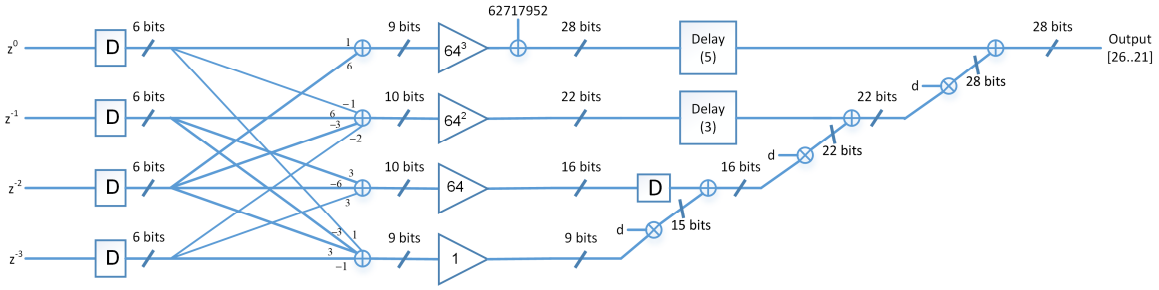


Figure 11. Schematic of the small block of the digital phase shifter

The final result has a latency of 8 samples, but this fact is no relevant for our application.

In order to test if the final results of the system implemented in VHDL are correct, a signal is generated and input in the Vivado simulator, ISIM, and in the Matlab function *intdelay*. In order to have the correct behaviour, both approaches should give the same output. In Figure 12, there are the results obtained using the Matlab function *indelay* and in Figure 13, the results obtained with ISIM. Therefore, as the results are the same, we can state that the VHDL implementation is correct.

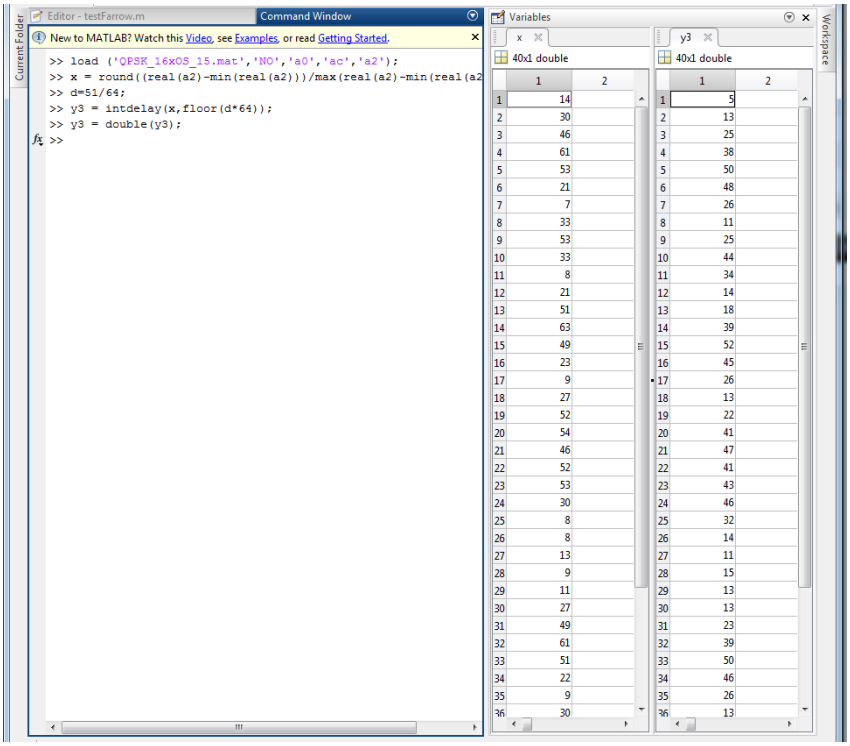


Figure 12. Results in Matlab of the intdelay function

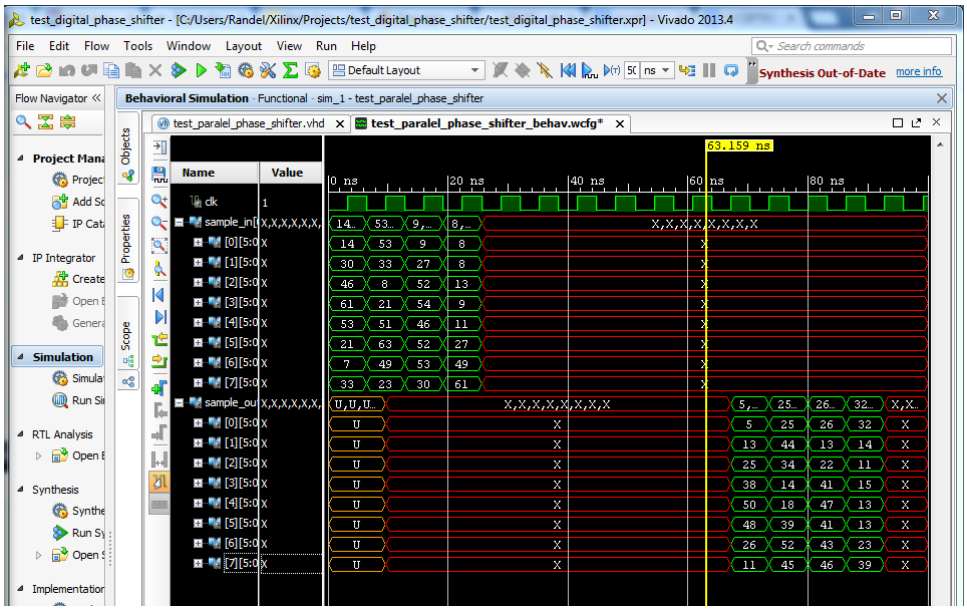


Figure 13. Results in Vivado of the small block of the digital phase shifter

Once the small block has been done, it is time to assemble the big block. The entity of it is:

```
entity paralel_phase_shifter is
    GENERIC(NUMBER_OF_SAMPLES: POSITIVE :=256 );
    Port ( clk : in STD_LOGIC;
           samples_in : in sample_array(0 to NUMBER_OF_SAMPLES-1);
           delay : in STD_LOGIC_VECTOR (6 downto 0);
           samples_out : out sample_array(0 to NUMBER_OF_SAMPLES-1));
end paralel_phase_shifter;
```

where *NUMBER_OF_SAMPLES* is, as the name says, the number of samples that will input the system in each clock, *clk* is the clock of the system, *samples_in* are the samples that input the system, *delay* is the desired fractional delay that will be applied to the input and *samples_out* are the samples once the system has made the interpolation. The schematic of the combinations of the small blocks used in the big block is the following:

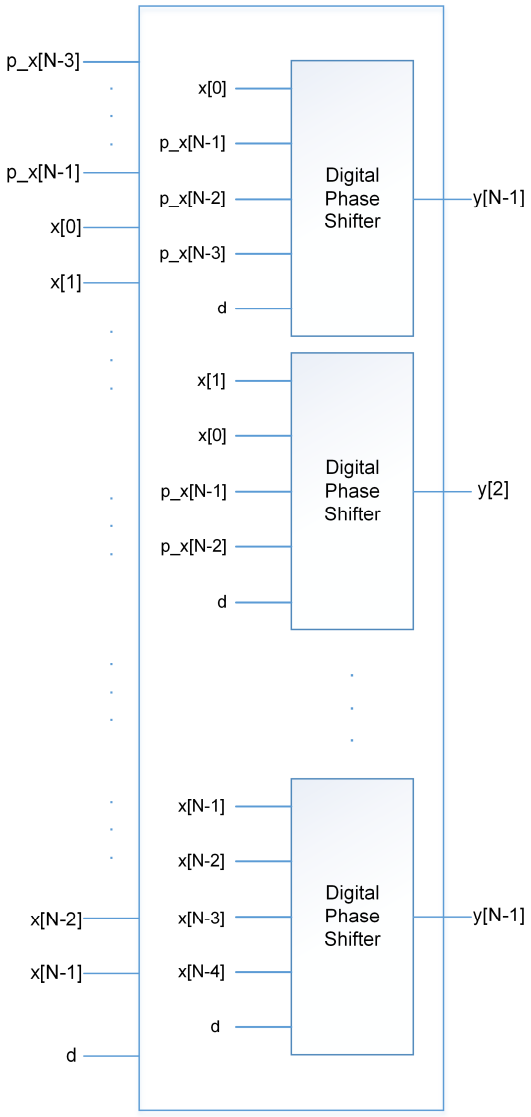


Figure 14. Schematic of the big block of the digital phase shifter

2.4. Results

Once the VHDL code was working properly, the only remaining thing was the integration with the code that controls the FPGA. This part was not done by me as the development of the whole code is being done by another researcher, Stephen Corstelli. So the code was provided to him and he integrated it with my cooperation. After that, the code was program in the FPGA and tested with different signals and different delays. In the test that is shown below a low pass filtered square signal with four samples per each period running at 30Gsamples/s was used. In all the different figures, except the upper left that is the original, a different delay was applied.

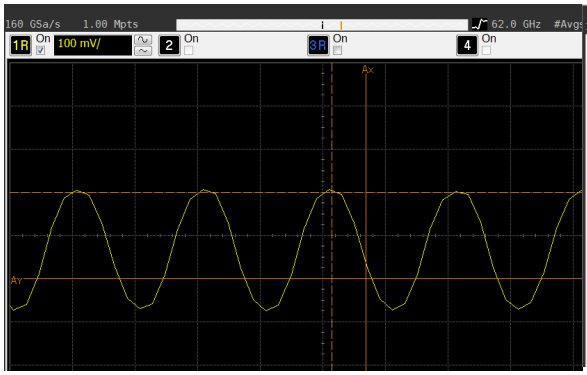


Figure 15. Screenshot of the oscilloscope with a delay=0

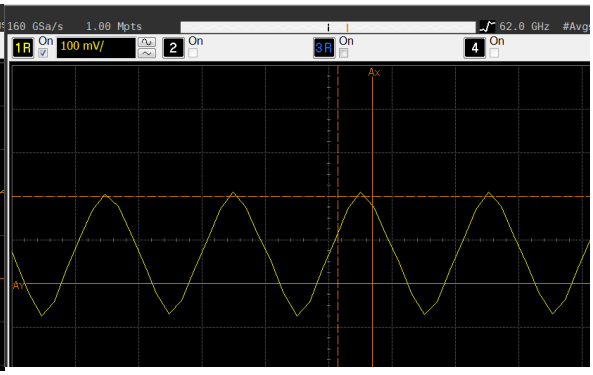


Figure 16 . Screenshot of the oscilloscope with a delay=15

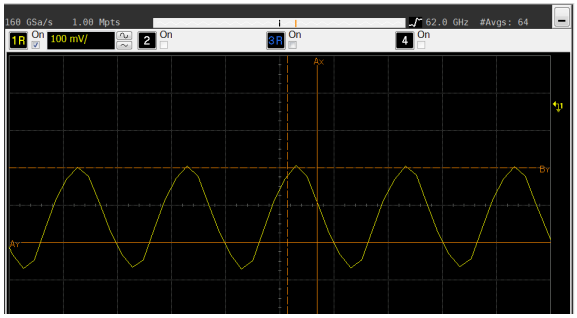


Figure 17 Screenshot of the oscilloscope with a delay=32

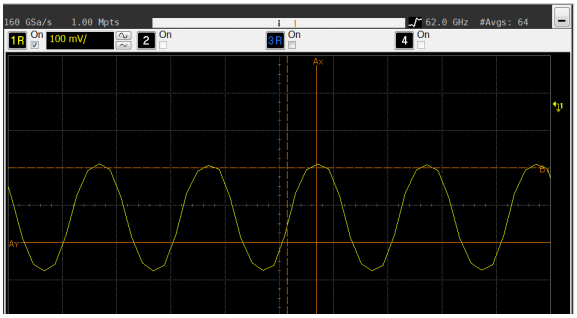


Figure 18 Screenshot of the oscilloscope with a delay=63

In the previous snapshots, we can see that the behaviour of the system is satisfactory, the signal peak in the middle of the screen is moved to the right when we increase the delay. In some of the delays, the sine wave is not completely preserved, but the result is good enough. We also have to take into account that we are not using a really high interpolation order as we are using a cubic order and the performance cannot be really high.

Another important result is the usage of the resources available. In order to obtain it, we have to target an FPGA, in our case the code was implemented in the FPGA model: xc7vx680t-2. The final usage of the total resources taking into account not only the digital phase shifter code, but also the code that need the FPGA to run properly is presented in the Table 2. A description of the meaning of the different parameters that are evaluated in the table can be found in the appendices.

Logic Utilization	Used	Available	Utilization
Slice LUTs	145105	433200	33%
LUT as Logic	107305	433200	24%
LUT as Memory	35873	174200	20%
Slice Registers	157608	866400	18%
DSPs	2048	3600	56%

Table 2. Utilization resources of the digital phase shifter

As a conclusion, the resources utilized of this FPGA are relatively low and other DSP algorithms could also be implemented before or after the digital phase shifter in the same FPGA. The only requirement for the other algorithms will be that they don't use a lot of DSP cells as the digital phase shifter uses 2/3 of them. The objective of developing an interpolator that fits in an FPGA is therefore accomplished. The integration with the experiments that are currently being undergone in the lab will be the next step in the future as this is the reason why this module was developed.

3. DFT

3.1. Description of the system

The FFT is a basic algorithm when working with signals that has been used in a lot of different applications. With the increasing of the usage of DSP in the optical field, this algorithm could be a really useful tool to help other algorithms to work properly. The implementation of the DFT algorithm in an FPGA is a necessary tool to be closer to have the final designs working in real time.

Before starting to develop our implementation, a research of the options already available was made. The best option found was an IP core that is provided by Xilinx in their design software suite, Vivado 2014.1 [7]. Although this option has a lot of parameters to tweak, it has a major drawback that the input has to be sequential. This represented a major problem because a parallelization of the input is necessary due to the high speed that the FPGA will have to be working. Therefore, the DFT block should work with all the N parallel samples inputting at the same time.

3.2. State of the art

A lot of research in different methods to implement a fast version of the DFT has been done since in 1965 a publication of James W. Cooley and John W. Tukey [8] popularized the idea of the FFT. In that publication, they explained that the DFT has a lot of symmetries and there is no need to compute all the operations present in it. In our case, the Split-radix algorithm was the selected algorithm because it is relatively easy to implement in hardware and it also uses less resources than some of the most common ones like Radix-2 or Radix-4. Actually, the Split-radix is based in this two and uses the best of both of them. In the following section a brief explanation of the three of them is given.

3.2.1. DIT Radix-2

One of the first approaches is to take advantage of the symmetries of the DFT reducing in this way the total number of multiplications that are needed. Let's start with the general expression for the DFT:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\left(j\frac{2\pi nk}{N}\right)} \tag{3.1}$$

This expression can be rearranged if we separate the odd and even samples of the summation:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)\, e^{-\left(j\frac{2\pi nk}{N/2}\right)} + e^{-\left(j\frac{2\pi k}{N}\right)} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)\, e^{-\left(j\frac{2\pi nk}{N/2}\right)} \tag{3.2}$$

The exponential of the second summation it is called the twiddle factor $W_N^k = e^{-\left(j\frac{2\pi k}{N}\right)}$. Then we can express the equation as two smaller DFTs:

$$X(k) = DFT_{\frac{N}{2}}[x(0), x(2), \dots, x(N-2)] + W_N^k DFT_{\frac{N}{2}}[x(1), x(3), \dots, x(N-1)] \tag{3.3}$$

The algorithm is called Decimation in Time (DIT) because the samples are rearranged in time and then in the frequency domain the samples are in the correct order. The other

possible option is the Decimation in Frequency (DIF) where the opposite procedure is applied. In the previous expression, it can be noticed that the DFTs produce only N/2 samples. However, we have to take into account the periodicity with N/2 samples of these small DFTs, these can be reused and the only parameter that will have to be changed to compute all the samples of X(k) is the twiddle factor. A graphical representation of the combination of all the samples will be:

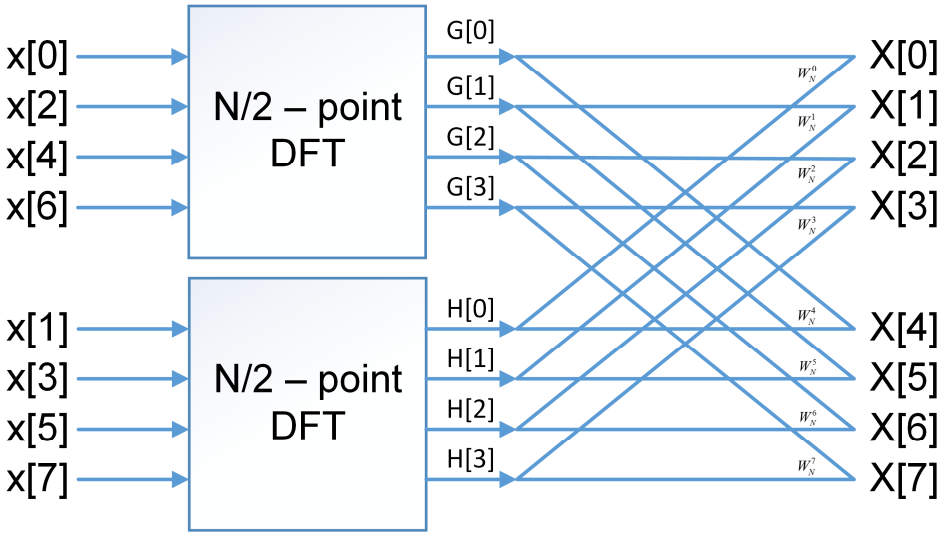


Figure 19. Schematic of the DIT Radix-2 algorithm

Once reached this step, there are further modifications to reduce the number of multipliers used by the twiddle factors. The part of the algorithm where the samples of the small DFTs are combined and the twiddle factors are used, it is called the butterfly. The further modification consists in merging the twiddle factors at the beginning instead of doing the multiplication just before the addition. The final change in the butterfly will be:

Figure 20. Evolution of the Radix-2 butterfly

The algorithm is recursively applied in the smaller DFTs until a length-2 DFT is reached. For example the final structure of a length-8 DFT is:

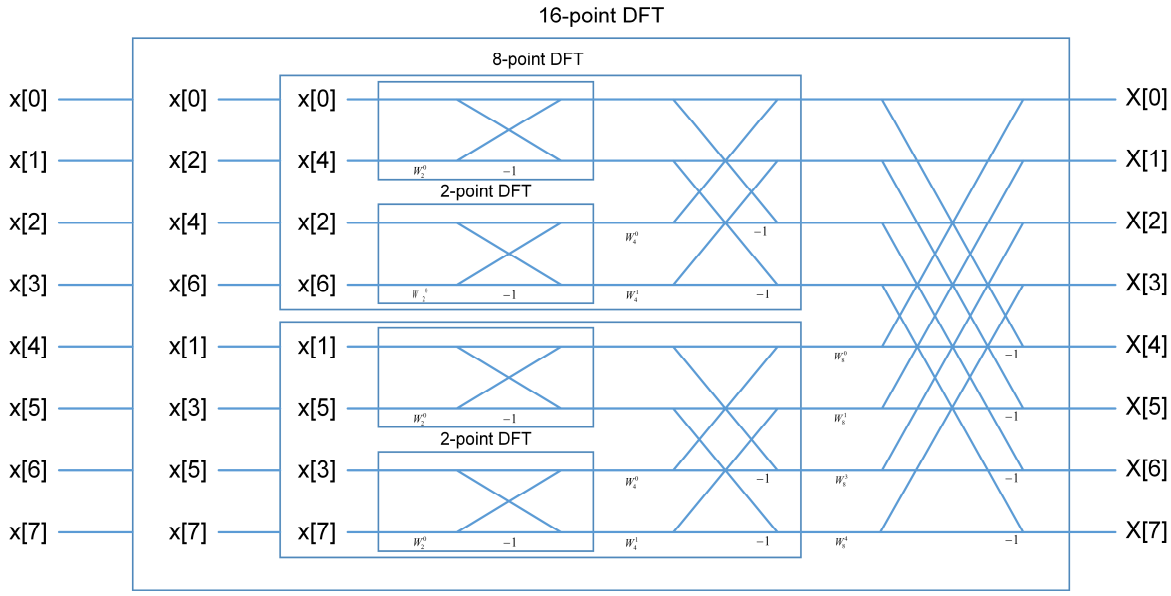


Figure 21. Example of the structure of an 8-point DFT implemented with Radix-2

The initial computational cost if we compute directly the DFT is:

- N^2 complex multipliers
- $N^2 - N$ complex adders

After using the Radix-2 algorithm the computational cost will be:

- $\frac{N}{2} log_2(N)$ complex multipliers
- $N log_2(N)$ complex adders

A more detailed explanation of the algorithm can be found in [9].

3.2.2. DIT Radix-4

The main idea of the Radix-4 algorithm is the same as in Radix-2, separate the DFT in smaller DFTs and take advantage of the symmetries. However, instead of partitioning it in two parts, it is partition in four. The partitions are:

$$X(k) = \sum_{n=0}^{\frac{N}{4}-1} x(4n) e^{-\left(j\frac{2\pi(4n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1) e^{-\left(j\frac{2\pi(4n+1)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+2) e^{-\left(j\frac{2\pi(4n+2)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+3) e^{-\left(j\frac{2\pi(4n+3)k}{N}\right)} \quad (3.4)$$

The result with the corresponding twiddle factors using for them the same definition as in Radix-2 is:

$$X(k) = DFT_{\frac{N}{4}}[x(4n)] + W_N^k DFT_{\frac{N}{4}}[x(4n+1)] + W_N^{2k} DFT_{\frac{N}{4}}[x(4n+2)] + W_N^{3k} DFT_{\frac{N}{4}}[x(4n+3)] \quad (3.5)$$

Depicted below is the structure of the algorithm, the twiddle factors are not included due to a lack of space:

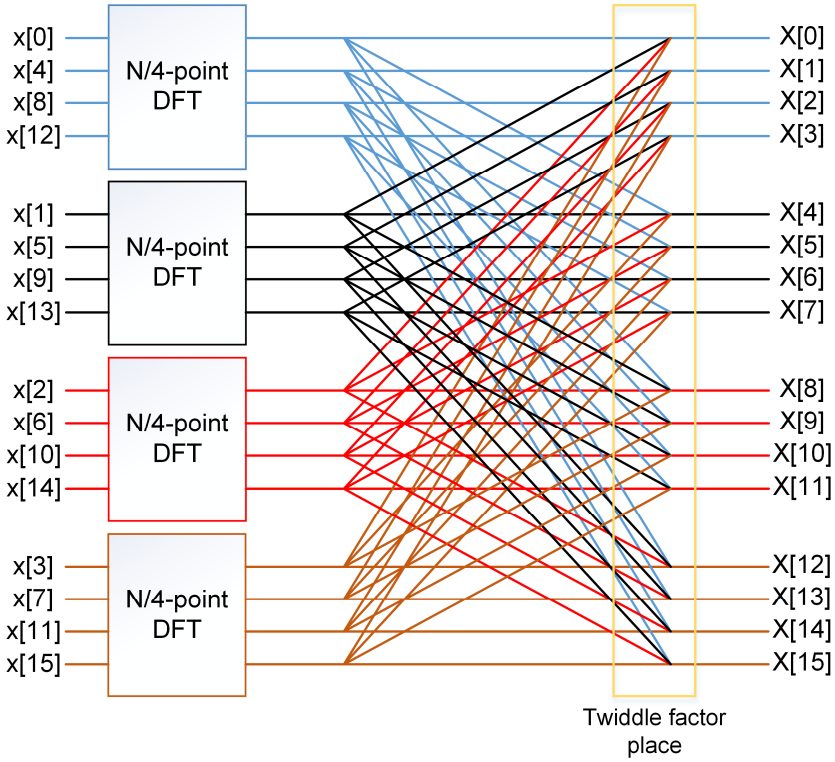


Figure 22. Schematic of the DIT Radix-4

The same modification of doing the multiplications at the beginning of the butterfly is also applied. Moreover, the resulting butterfly is a multiplication by the twiddle factors followed by a length-4 DFT. It has to be noticed that to perform the length-4 DFT only 8 adders are required and no multipliers are needed. The butterfly will look like:

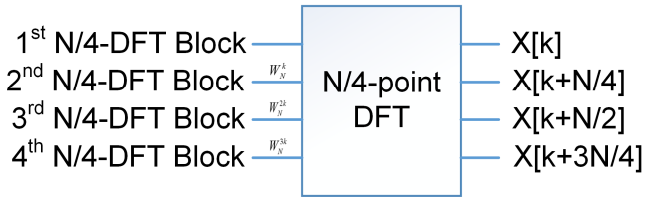


Figure 23. Final butterfly of the Radix-4

As in the Radix-2, the periodicity with N/4 of the short DFTs is used, so their output for a frequency sample k are reused to compute X(k), X(k+N/4), X(k+N/2) and X(k+N/3/4). It is this reuse that give as the efficiency

The final computational cost of the Radix-4 algorithm is better that the Radix-2 as it uses a 75% of the multipliers used in Radix-2:

- $\frac{3N}{8} log_2(N)$complex multipliers
- $Nlog_2(N)$ complex adders

Further information about the algorithm can be found in [10].

3.2.3. Split-radix

The Split-radix mixes Radix-2 and Radix-4 decompositions, using in this way less multipliers than the two former algorithms. The first clear description was made by Duhamel and Hollman [11] although Yavne [12] was the first to derive it, but in a really atypical way. Johnson and Frigo [13] have introduced some modifications that improve just a little bit the algorithm, but they difficult the understanding so the original algorithm is the one that will be explain and implemented.

The decompositions of the DFT are:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-\left(j\frac{2\pi(2n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1)e^{-\left(j\frac{2\pi(4n+1)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+3)e^{-\left(j\frac{2\pi(4n+3)k}{N}\right)} \quad (3.6)$$

The expression with the corresponding twiddle factors is:

$$X(k) = DFT_{\frac{N}{2}}[x(2n)] + W_N^k DFT_{\frac{N}{4}}[x(4n+1)] + W_N^{3k} DFT_{\frac{N}{4}}[x(4n+3)] \quad (3.7)$$

The schematic of the algorithm is:

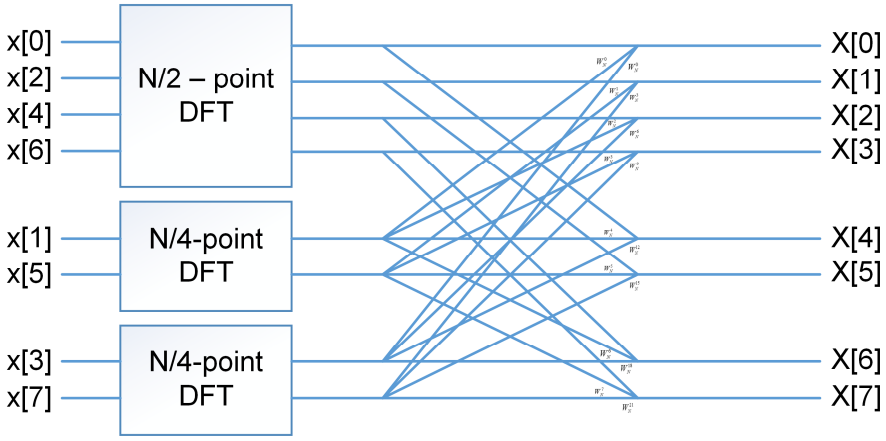


Figure 24. Schematic of the Split-radix

The butterfly can be modified in order to use less multipliers, the steps of the modifications of a part of it are:

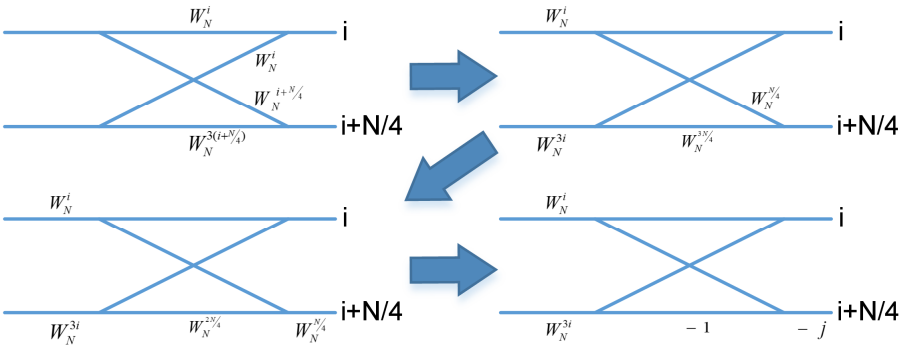


Figure 25. Evolution of part of the butterfly of the Split-radix

Then, the final complete butterfly for the Split-radix is:

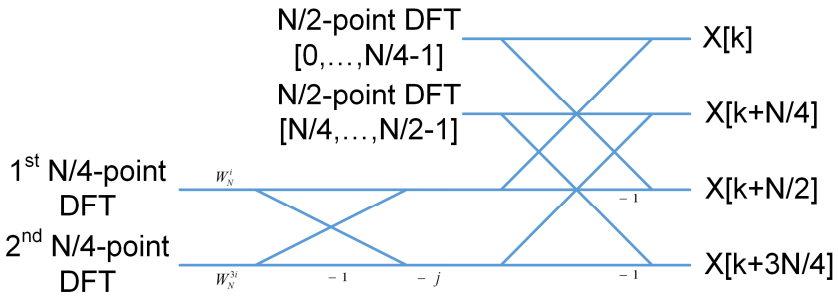


Figure 26. Final butterfly of the Split-radix

The new butterfly differs from the previous ones because it is not any more symmetrical and it has an L shape. As in Radix-4, the periodicity in the shorter DFTs is used, the first N/4 samples of the N/2 DFT and all the samples of the N/4 DFTs are used to compute X(k) and X(k+N/2). The second set of N/4 samples of the N/2 DFT and again all the samples of the two N/4 DFTs are used for the X(k+N/4) and X(k+3N/4).

An example of the final butterfly with an 8-point DFT:

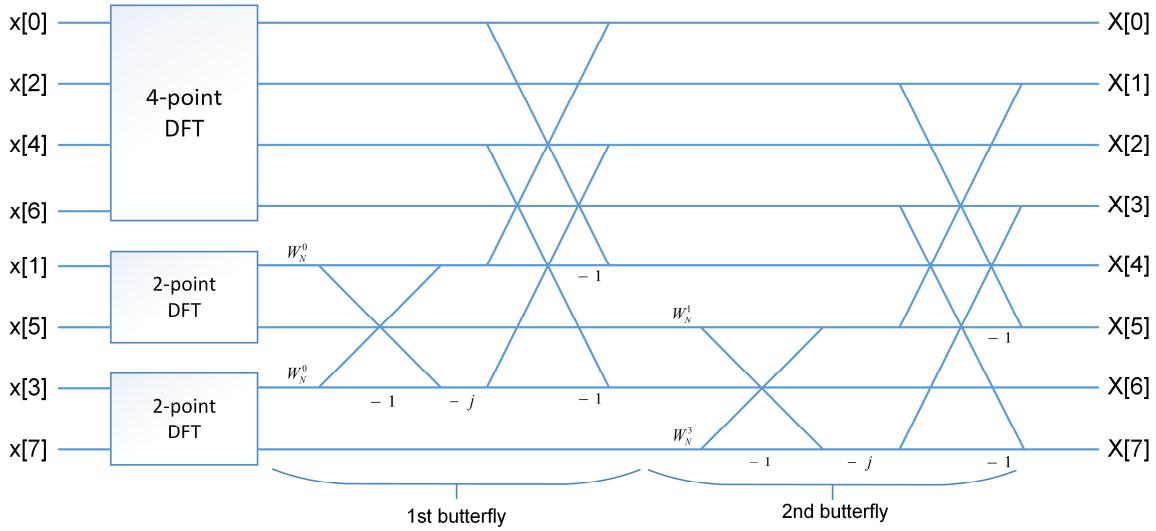


Figure 27. Example of an 8-point DFT implemented with Split-radix

Finally, the computational cost with the Split-radix will be, as stated before, lower than the two previous algorithms:

- $\frac{N}{3} log_2(N)$ complex multipliers
- $N log_2(N)$ complex adders

As a conclusion, the number of complex multipliers used in the Split-radix are only a 67% of the ones that were used in Radix-2.

More information about the algorithm can be found in [14].

3.2.4. CORDIC algorithm

All the algorithms that have been presented use multipliers. This mathematical operation when it is implemented in hardware it usually requires a lot of resources and this is not always desired. The CORDIC algorithm substitutes the multiplications with a sequence of additions and subtractions. It is suitable when we want to rotate the phase of a complex number, in our case we have a complex number and we want to multiply it by the twiddle factor so the CORDIC algorithm suits perfectly.

The basic principles of the algorithm are the following. We have the original input $C = I_c + jQ_c$ and we want to multiply it by $R = I_r + jQ_r$ which will give us $Y = I_y + jQ_y$. If we want to add R's phase to C the result is $I_y = I_c I_r - Q_c Q_r;\ Q_y = Q_c I_r + I_c Q_r$ whether if we want to subtract it, the result is $I_y = I_c I_r + Q_c Q_r;\ Q_y = Q_c I_r - I_c Q_r$.

If we set the rotation complex number as $R = 1 \pm jL$ with L as a decreasing power of two starting at 2^0 the phase of R will be arctan(L) or –arctan(L). So to add a phase, we will use the expression of R with a plus and to subtract, the same expression with the minus. The final result will be $I_y = I_c \mp Q_c 2^{-k}; Q_y = Q_c \pm I_c 2^{-k}$.

In this expression, it can be clearly seen that no multipliers are needed as the operations with a power of 2 can be done with a shifting of the binary numbers. Also, the possible phases that we will be able to add or subtract are fixed and the first 5 are represented in the following table:

k	L	atan(L)
0	1	45°
1	0.5	26.5605°
2	0.25	14.03624°
3	0.125	7.12502°
4	0.0625	3.57633°

Table 3. Evolution of the phase of R with k

Once we have the basic principles, the algorithm is more or less straightforward. The graphic flow is presented in Figure 28.

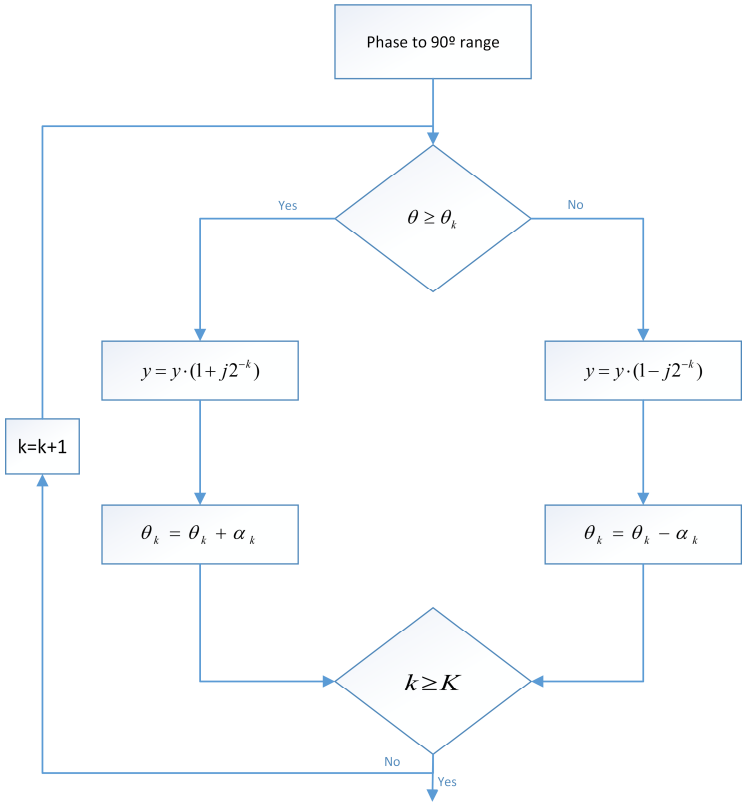


Figure 28. General graphic flow for the CORIC algorithm

The algorithm work for phases that are less than 90 degrees, but a shift of 90 degrees is really easy to implement with a swapping of the real and imaginary parts with the corresponding changes in the signs.

Although that using the CORDIC algorithm seems to have almost no drawbacks, the algorithm introduces a gain in each of the iterations that have to be taken into account. Fortunately, this gain does not depend on the input and it is fixed depending on the number of iterations, so it will not be too difficult to remove. The evolution of the gain for the first 5 iterations is:

K	CORDIC gain
0	1.414213
1	1.581138
2	1.629800
3	1.642484
4	1.645688

Table 4. Evolution of the CORIC gain with K

Moreover, if we continue the table, we will be able to observe that the gain more or less converges at 1.647 starting at the 7th iteration.

More information about the CORDIC algorithm can be found in [15].

3.3. Implementation

In order to reach the final goal of implementing a 64-point DFT using the Split-radix in VHDL, several steps were done to assure the correct performance of the final result. In a first stage, all the algorithms were developed in Matlab and were integrated once the separate behaviour was the expected. Once the Matlab implementation was finished, the next step was the VHDL coding. This was also done in several steps in order to warranty that the final result was correct.

3.3.1. Matlab

The approach to develop and test the algorithms was similar to the one used in the digital phase shifter. First of all, a Matlab version with floating point was written to check that the expected results were obtained. For this purpose, three functions were created *fft_radix2(x), fft_radix4(x) and fft_split_radix(x).* The three algorithms were implemented, although we are only interested in the Split-radix, because it was easier to develop in order of ascending difficulty. The functions were implemented in a recursively way, so if the DFT needs an inner DFT of half the samples for example, the same function is called again with half the samples as the input.

Once all the algorithms were working properly, the CORDIC algorithm was implemented. The resultant function was *cordic(x,theta,K)* where *x* is the input data, *theta* is the phase rotation that we want to apply and *K* is the number of iterations that we want to perform. In order to compare the results with the complex exponential, the CORDIC gain was removed at the end of the function. Some tests using a complex input and a random phase were performed and the results were compared to a multiplication of the input with a complex exponential of the same phase and with modulus one. When the results were satisfactory, the CORDIC algorithm was included in the Split-radix function and some simulations were done.

In order to have an idea of how the number of iterations in the CORDIC algorithm improve the final result, the Figure 29 was done. In order to compute the SNR, we have to consider the original signal as the Matlab function to perform the FFT using floating point and the signal with the noise as the signal resultant of the Split-radix function with the CORDIC implemented. Then, the SNR will measure the noise introduced by the numerical error when we used the CORDIC algorithm.

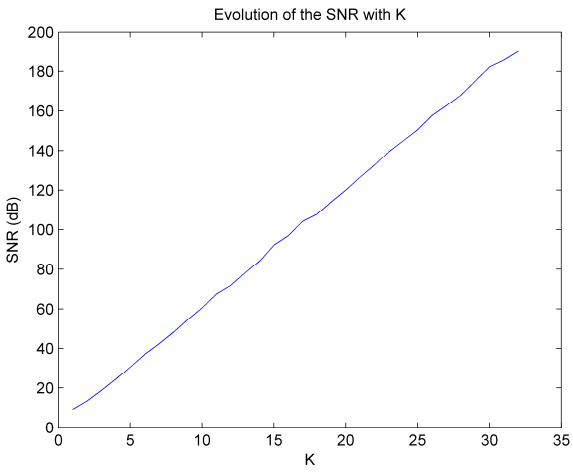


Figure 29. Evolution of the SNR with K

The necessary number of iterations will depend in each application, but usually 10 iterations with a SNR of 60 dB should be enough. In order to have an example with a real application, the FFT algorithm was applied as a part of a real DSP block, the result is showed in the Figure 30. The figure was obtained using the Split-radix algorithm with CORDIC in a real DSP block that my supervisor was working on. The y-axis represents the SNR penalty for a BER of $3.8 \cdot 10^{-3}$. The SNR penalty is a useful parameter to understand the performance of the system, it measure how many dB we are below the theoretical value of the system for a given BER. The BER chosen is the hard FEC limit to achive almost error free transmissions. In the x-axis, there is the number of iterations of the CORDIC:

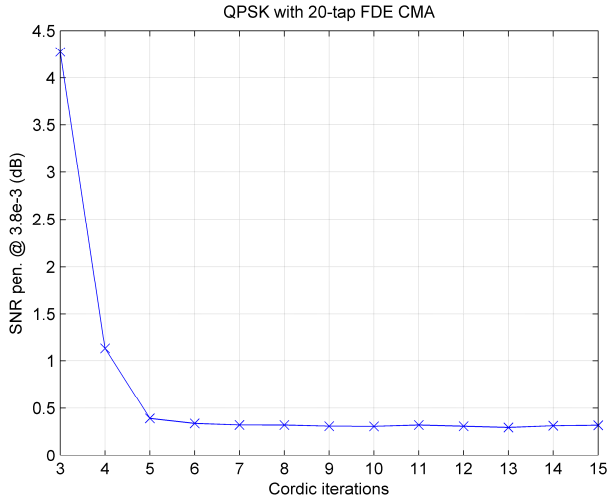


Figure 30. Evolution of the SNR penalty at 3.8e-3 with the CORDIC iterations

As a conclusion, in the final implementation in hardware only 6 iterations will be needed as we can appreciate in the plot that the improvement for a higher number of iterations is almost insignificant.

The next step was to implement the algorithm with a closer relationship with the hardware. As before, the fix point toolbox of Matlab was used. The functions were redesign to be easier to implement in hardware and the fixed point was introduced. The first approach was to not use the CORDIC algorithm and use the complex exponential multiplications. In this way, the deterioration of the outcome due to the fixed point could be clearly appreciated and not mixed with the one produced by the CORDIC algorithm. Once again, the result was compared to the default FFT function of Matlab. The Figure 31 shows the SNR taking into account, as in the Figure 29, that the noise will be the numerical error introduced by our function in comparison to the Matlab default FFT function, instead of the number of iterations, now we will plot the number of bits that are used in the decimal part of the numbers.

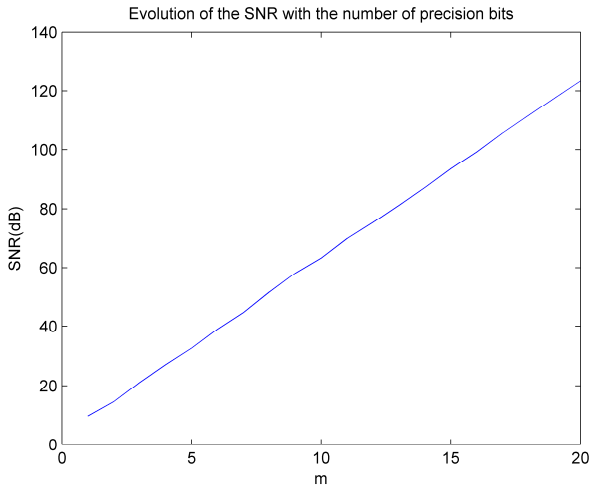


Figure 31. Evolution of the SNR with the precision bits used

The necessary SNR will be specific of the implementation that we are doing and it is difficult to determine an exact value, for example in a general implementation with high precision a SNR of about 60 dB should be enough and therefore 10 bits in the precision part of the number will have to be used.

The algorithm with fixed point and without CORDIC was also included in the same DSP block that the algorithm with CORDIC was tested before, as a result the Figure 32 was obtained. The y-axis is the same as the previous one, but the x-axis now represents the number of bits used at the right of the dot in the fixed point numbers.

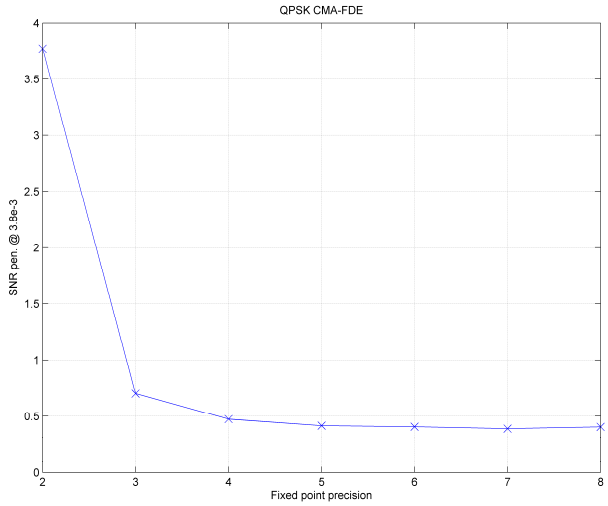


Figure 32. Evolution of the SNR penalty at 3.8e-3 with the precision bits used

Once again, we can notice that in a specific implementation the accuracy can be lower than the one that we could expect. In this specific system then, the required number of precision bits will be 5 because if we take more bits the improvement that we will see in the result is almost insignificant.

A simulation including a sweeping of both parameters, the number of precision bits and the number of CORDIC iterations, could not be perform due to the time that the simulation would take (with the trials that have been done, the duration of the simulation has been

estimated with at least two weeks). The main problem for this was the fixed point design of Matlab that is pretty slow. As a result, the final parameters that we have to continue with the VHDL code will be the two that have been found separately, 6 iterations for the CORDIC algorithm and 5 bits of precision. If in the future these parameters have to be modified, it will not be a major problem to update the VHDL code, specially the number of precision bits is really easy to be changed.

3.3.2. VHDL

Different implementations in VHDL were done in order to develop the project in incremental stages. In this project, three of them will be presented and all of them are completely functional. The first uses multipliers, the second one uses the CORDIC algorithm with a generic implementation and the last one uses the CORDIC algorithm with a hard coded implementation.

For the development of all the different implementations, apart from the packages which are already available in Vivado 2014.1, the package from David Bishop [16] to deal with fixed point in VHDL was used. This package is widely accepted in the industry and compatible with VHDL-93.

In each implementation, first the overall system is presented and then all the necessary blocks to build it are breakdown. The first section contains some blocks that are used in almost all the designs. Furthermore, all the blocks have a configurable part, called generic, where the number of bits at the input of each block and the number of precision bits used are defined. Another important point, it is that the bit growth inside the algorithm is not taken into account in the preliminary designs, it is only specified in the last implementation.

At the end of each implementation, the approximated usage of the resources is shown to have an idea of the drawbacks of each design and why several implementations were done. The resource utilization is obtained using the FPGA model: xc7v2000t-2flg1925. This is the FPGA model where the algorithm will be implemented at the end. In order to have a general framework to compare the resource utilization among the different implementations, the parameters must be the same. The number of bits at the left of the dot will be 10 and the number of bits at the right (precision bits) will be 5.

Common blocks

4-point DFT

The block to compute the 4-point DFT will be used several times in all the different designs. The input parameters of the block are:

```
entity fft_4 is
    GENERIC(NUM_BITS: POSITIVE := 10;
            PRECISION: POSITIVE := 5);
    Port ( clk : in STD_LOGIC;
           x_real: in fftsamples(0 to 3); --The input must be a signed fix point number
with #NUM_BITS bits before the dot and #PRECISION bits after it.
           x_imag: in fftsamples(0 to 3);
           y_real: out fftsamples(0 to 3); --The output will be a signed fix point number
with #NUM_BITS bits before the dot and #PRECISION bits after it.
           y_imag: out fftsamples(0 to 3));
end fft_4;
```

*The fftsamples are defined in another package as TYPE fftsamples IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0)

In the block, the operation that is performed is equivalent to:

$$Y = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \cdot X \tag{3.8}$$

This will not be expressed directly as a matrix in the code because it is not possible in VHDL. Instead, the addition of the real and imaginary parts will be done separately in different lines of code.

The block first stores the inputs, then in the following clock it does the computations expressed in the previous matrix and in the last clock it outputs the result. Consequently, this block will have a latency of three clocks. We have to take into account that the latency is not important for the delay that will introduce between the input and the output, but to equalize the delay of the different branches when we build the whole system.

2-point DFT

The 2-point DFT will also be used in all the following designs. The definition of the block is:

```
entity fft_2 is
    GENERIC(NUM_BITS: POSITIVE := 10;
            PRECISION: POSITIVE := 5);
    Port ( clk : in STD_LOGIC;
           x_real: in fftsamples(0 to 1); --The input must be a signed fix point number
with #NUM_BITS bits before the dot and #PRECISION bits after it.
           x_imag: in fftsamples(0 to 1);
           y_real: out fftsamples(0 to 1); --The output will be a signed fix point number
with #NUM_BITS bits before the dot and #PRECISION bits after it.
           y_imag: out fftsamples(0 to 1));
end fft_2;
```

*The fftsamples are defined in another package as TYPE fftsamples IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);

In the block, the operation performed is:

$$Y = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot X \tag{3.9}$$

The procedure followed by the block is the same that the 4-point DFT. Therefore, the latency of the block will also be three clocks.

Multiplier design

In the first design of the Split-radix algorithm, the twiddle operations were implemented using multipliers. The idea was to have a first structure that could be improved updating some of the different blocks and where it could also be test if the algorithm was implemented correctly.

The input for all the DFT blocks will be pretty similar. In the following code the X will represent the number of points that the DFT has:

```
entity fft_X_sr is
    GENERIC(NUM_BITS: POSITIVE := 10;
        PRECISION: POSITIVE := 5);
    Port ( clk : in STD_LOGIC;
           x_real: in fftsamples(0 to X-1);
           x_imag: in fftsamples(0 to X-1);
           y_real: out fftsamples(0 to X-1);
           y_imag: out fftsamples(0 to X-1));
end fft_X_sr;
```

*The fftsamples are defined in another package as TYPE fftsamples IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);

The real and imaginary part of the twiddle factors are computed in advance and are integrated in the code as a constant. For example in the 8-point FFT:

```
    type twiddle_array is array (0 to 1) of sfixed(2 downto -PRECISION);
    CONSTANT twiddle_real: twiddle_array := (to_sfixed(0.707106781186548, 2, -
PRECISION),to_sfixed(-0.707106781186548, 2, -PRECISION));
    CONSTANT twiddle_imag: twiddle_array := (to_sfixed(-0.707106781186548, 2, -
PRECISION),to sfixed(-0.707106781186548, 2, -PRECISION));
```

The Figure 33 depicts the general schematic of the 64-point DFT where each block that is implemented in the system is represented with a box. The blocks can be matched to the different parts in the description of the Split-radix that has been done in the State of the art of this section. In parenthesis there is the latency of each block that has to be taken into account when we built the whole system in order to equalize it in all the branches. Finally, the delay blocks are simple memories.

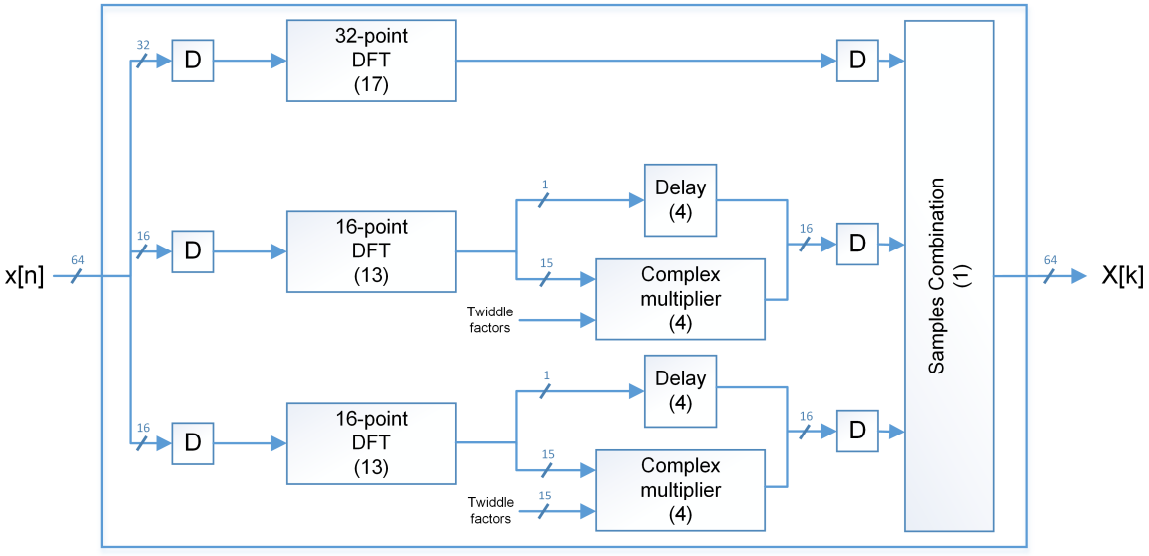


Figure 33. Schematic of the 64-point DFT point implemented with multipliers

The block “Samples combination" is not a block by itself and it is implemented inside the general FFT block. In this part the butterfly of the Split-radix, except the multiplications by the twiddle factors that are done previously, is performed. The “Complex multiplier” block is then the only that will be detailed below.

Complex multiplier

The inputs for the multiplier block are:

```
entity complex_multiplier is
    GENERIC(NUM_BITS: POSITIVE := 10;
            PRECISION: POSITIVE := 5);
    Port (  clk: in STD_LOGIC;
            a_real: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            a_imag: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            b_real: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            b_imag: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            y_real: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            y_imag: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0));
end complex_multiplier;
```

In this block, instead of using the four multiplications that are usually needed to perform a complex multiplication, three are used. First of all, the inputs are combined in the following way:

$$p1 = a_{real} \cdot b_{real} \tag{3.10}$$

$$p2 = a_{imag} \cdot b_{imag} \tag{3.11}$$

$$p3 = (a_{real} + a_{imag}) \cdot (b_{real} + b_{imag}) \tag{3.12}$$

After that, using only additions or subtractions we can obtain the final result:

$$y_{real} = p1 - p2 \tag{3.13}$$

$$y_{imag} = p3 - p2 - p1 \tag{3.14}$$

Resource utilization

The resources taking into account the specifications detailed previously are:

Logic Utilization	Used	Available	Utilization
Slice LUTs	42207	1221600	3.45%
LUT as Logic	39027	1221600	3.19%
LUT as Memory	3180	344800	0.92%
Slice Registers	30368	2443200	1.24%
DSPs	216	2160	10,00%

Table 5. Resource utilization of 64 point DFT block with the multipliers implementation

The Slice LUTs utilization is low, but the main problem is the high utilization of the DSP slices (a definition of the meaning of these parameters can be found at the appendices). If we want to use several FFT blocks combined with other blocks that may use also multipliers, we will run out of DSP slices really quickly.

CORDIC design with generic implementation

In the second implementation, the multipliers used by the twiddle factors were replaced by the CORDIC algorithm blocks. In the implementation of this part a major problem was faced, the CORDIC gain. In order to remove it, a division was done in Matlab. However, in hardware a division is not the best solution and the gain should be handled in another way. The implemented idea multiplies the branches that do not undergo the CORDIC algorithm by the CORDIC gain to equalize all the signals. It also has to be pointed out that not always is necessary this multiplication and it will depend on in each of the X-point DFT blocks how many times the gain has been compensated in the used blocks.

The entity of the blocks is similar to the next one if we substitute the X for the number of points of the current DFT:

```
entity fft_X_sr_cord is
    GENERIC(NUM_BITS: POSITIVE := N_BITS;--Number of bits at the left of the comma
        PRECISION: POSITIVE := PREC;--Number of bits at the right of the comma
        CORD_ITERATIONS: POSITIVE := CORD_IT;--Number of iterations of the cordic
        PRECISION_THETA: POSITIVE :=PREC_PHASE);--Number of bits at the right of the
comma used in all the operations related with the phase
    Port ( clk : in STD_LOGIC;
           x_real: in fftsamples(0 to X-1);
           x_imag: in fftsamples(0 to X-1);
           y_real: out fftsamples(0 to X-1);
           y_imag: out fftsamples(0 to X-1));
end fft_64_sr_cord;
```

*The fftsamples are defined in another package as TYPE fftsamples IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);

Now, the different phases that will be used for the CORDIC algorithm are stored as a constant in each of the DFT blocks. For example the constants in the 32-point DFT block are:

```
    TYPE twiddle_array is array (0 to N/4-2) of sfixed(6 downto -PRECISION_THETA);
    --Twiddle factors for the first 32 FFT block
    CONSTANT twiddle1: twiddle_array := (to_sfixed(-11.25, 6, -
PRECISION_THETA),to_sfixed(-22.5, 6, -PRECISION_THETA),to_sfixed(-33.75, 6, -
PRECISION_THETA),
        to_sfixed(-45, 6, -PRECISION_THETA),to_sfixed(-56.25, 6, -PRECISION_THETA),
to_sfixed(-67.50, 6, -PRECISION_THETA),to_sfixed(-78.75, 6, -PRECISION_THETA));

    --Twiddle factors for the second 32 FFT block
    CONSTANT twiddle2: twiddle_array := (to_sfixed(-33.75, 6, -
PRECISION_THETA),to_sfixed(-67.50, 6, -PRECISION_THETA),to_sfixed(-101.25, 6, -
PRECISION_THETA),
        to_sfixed(-135.00, 6, -PRECISION_THETA),to_sfixed(-168.75, 6, -
PRECISION_THETA),to_sfixed(-202.50, 6, -PRECISION_THETA),to_sfixed(-236.25, 6, -
PRECISION_THETA));
```

The 64-point DFT and the 8-point DFT blocks are shown in Figure 34 and Figure 35 to have an idea of the result.

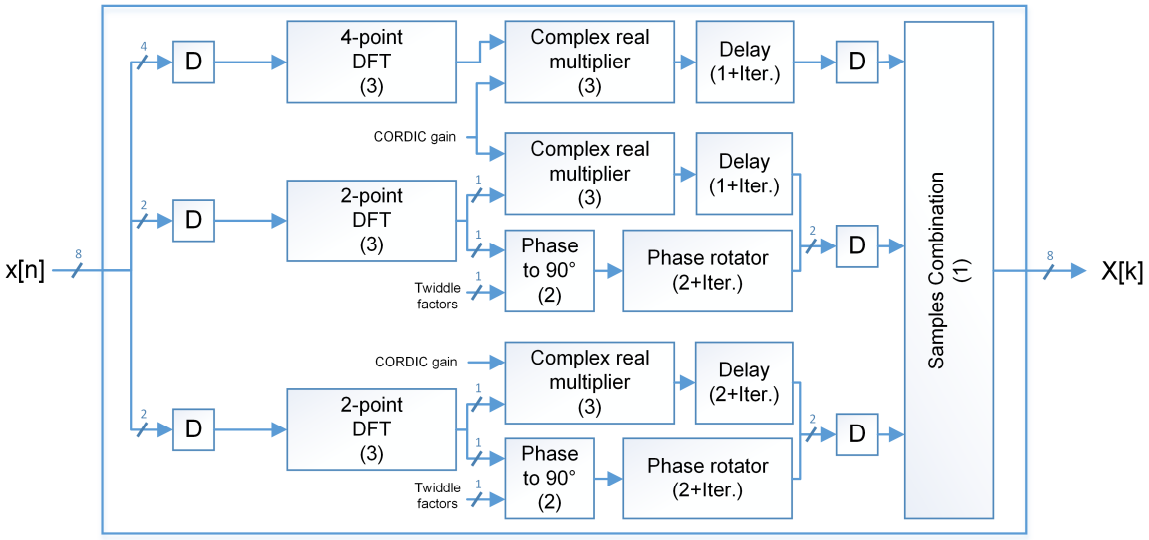


Figure 34. Schematic of the 8-point DFT point with the general implementation of the CORDIC

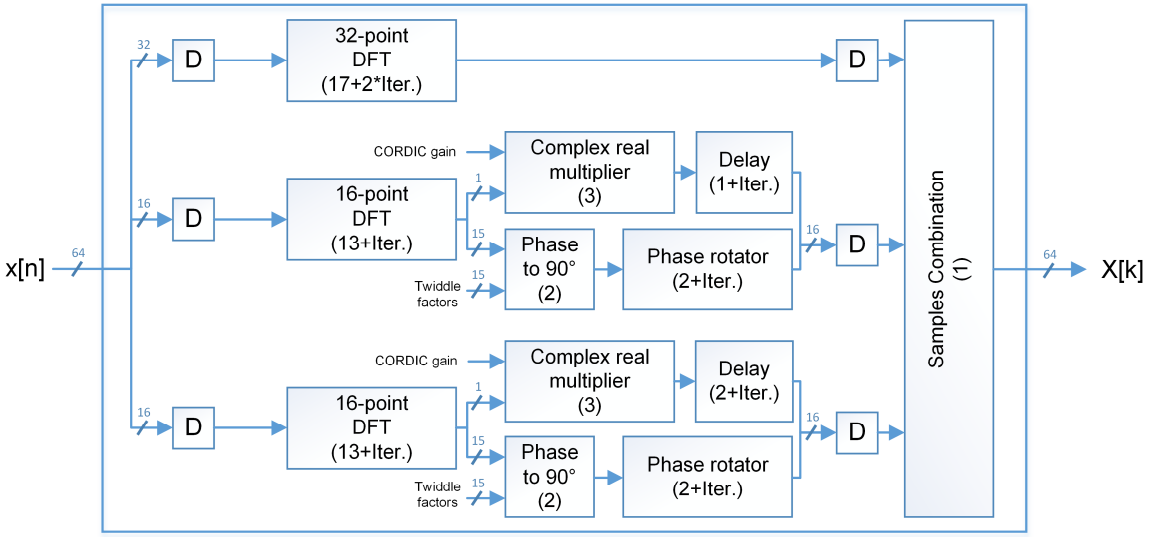


Figure 35. Schematic of the 64-point DFT point with the general implementation of the CORDIC

As before the block “Samples combination” is integrated in the code of each DFT block and it is the butterfly used in the Split-radix algorithm without the multiplication of the twiddle factors. Then, the blocks that will be detailed are the “Phase to 90°”, the “Phase rotator” and the “Complex real multiplier”.

Phase to 90°

At the input of the CORDIC algorithm the phase must always me be within the -90° to 90° range. The main purpose of this block is to convert the phase of the twiddle factor to the correct range, to do so it rotates the samples associated with each twiddle factor by the corresponding multiple of 90 degrees.

The entity of the block is:

```
entity phase_to_90 is
    GENERIC(NUM_BITS: POSITIVE := 10;--Number of bits at the left of the comma
            PRECISION: POSITIVE := 5;--Number of bits at the right of the comma
            PRECISION_THETA: POSITIVE :=4);--Number of bits at the right of the comma
used in all the operations related with the phase
    Port ( clk: in STD_LOGIC;
           x_real: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
           x_imag: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
           phase_in: in STD_LOGIC_VECTOR(PRECISION_THETA+6 downto 0);
           phase_out: out STD_LOGIC_VECTOR(PRECISION_THETA+6 downto 0);-- From -90 to 90
           y_real: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
           y_imag: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0));

end phase_rotator_continuous;
```

Phase rotator

This block performs the CORDIC iterations. In the hardware implementation is difficult to implement a loop, to solve this problem instead of doing all the loop iterations each clock, in each clock an iteration is performed for all the samples that input or are already in the block. This will introduce a delay between the input and the output of at least the number of iterations, but this is not a problem in our application. The entity of the block will be:

```
entity phase_rotator_continuous is
    GENERIC(NUM_BITS: POSITIVE := 10;--Number of bits at the left of the comma
            PRECISION: POSITIVE := 5;--Number of bits at the right of the comma
            ITERATIONS: POSITIVE := 7;--Number of iterations that the algorithm will do
            PRECISION_THETA: POSITIVE :=10);--Number of bits at the right of the comma
used in all the operations related with the phase
    Port ( clk: in STD_LOGIC;
           x_real: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
           x_imag: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
           phase: in STD_LOGIC_VECTOR(PRECISION_THETA+6 downto 0);--Phase from -90 to 90
           y_real: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
           y_imag: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0));
end phase_rotator_continuous;
```

Complex real multiplier

In order to equalize the gain in all the branches, we need to perform some multiplications. With this approach we still use some multipliers, but much less than in the previous results. This block will be the responsible to perform the multiplication of the complex signal with the gain which is a real value. The entity is:

```
entity complex_real_multiplier is
    GENERIC(NUM_BITS: POSITIVE := N_BITS;
            PRECISION: POSITIVE := PREC);
    Port (  clk: in STD_LOGIC;
            a_real: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            a_imag: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            b_real: in STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            y_real: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0);
            y_imag: out STD_LOGIC_VECTOR(NUM_BITS+PRECISION-1 downto 0));
end complex_real_multiplier;
```

Resource utilization

In the resources used by this second implementation, we can see that the Slice LUTs used has increased and the DSP bocks usage has decrease although it is still not zero. In further implementation, the objective will be to avoid using any DSP block and minimize the Slice LUTs resources.

Logic Utilization	Used	Available	Utilization
Slice LUTs	79136	1221600	6,47%
LUT as Logic	75868	1221600	6,21%
LUT as Memory	3268	344800	0,94%
Slice Registers	43906	2443200	1,79%
DSPs	112	2160	5,18%

Table 6. Resource utilization of the 64-point DFT block with the general implementation of the CORDIC

CORDIC with hard coded implementation

In the final implementation, the minimization of the resources was the most important point. Once that the system was working properly, all the blocks were analyzed to determine if further simplifications could be done. The modifications will be detailed in each of the blocks. Also, it was found a way to remove the CORDIC gain without doing the division. The final result for the 16 point DFT is:

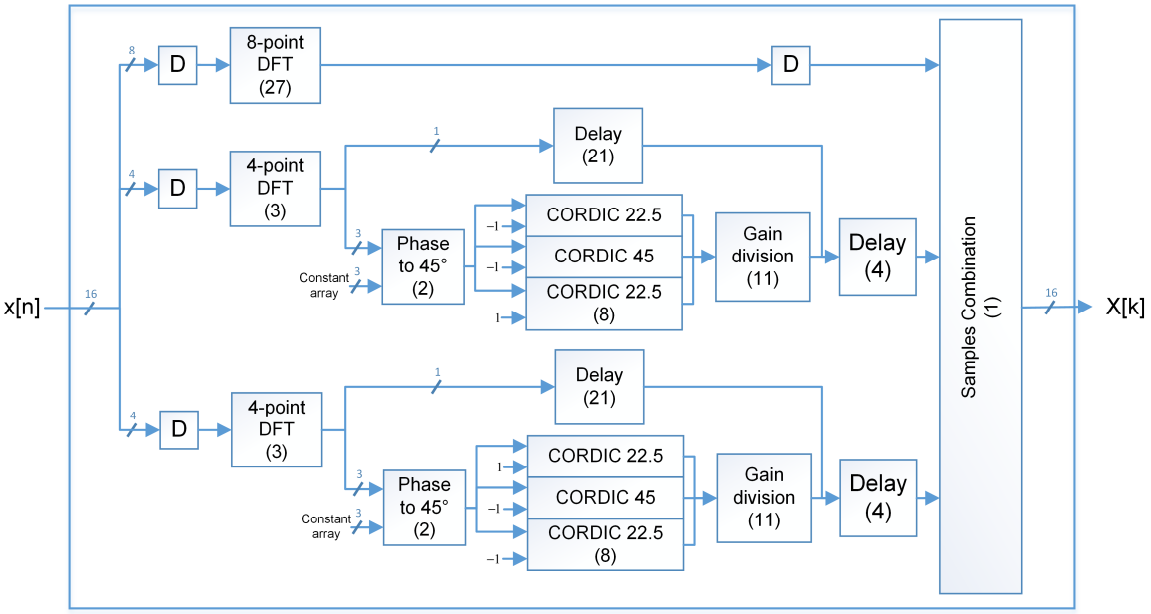


Figure 36 Schematic of the 16-point DFT point with the hard coded CORIDC implementation

The block "Samples combination" is the same as in previous designs. The really important blocks that have been developed specifically and have been hard coded are the "Phase to 45°", the different CORDIC blocks for each particularized phase and the "Gain division" block.

Gain division

Instead of doing a real division, an approximation with a succession of shifting with powers of two will be done. This operation will be much simple than using a division or multiplier. The approximation done is:

$$1-\frac{1}{2}+\frac{1}{4}-\frac{1}{8}-\frac{1}{16}+\frac{1}{32}+\frac{1}{64}-\frac{1}{128}+\frac{1}{256}+\frac{1}{512} \qquad (3.15)$$

In this way, the CORDIC gain is removed and we avoid the problem of equalization with the branches that do not undergo the CORDIC block. We also get rid of all the multipliers that we were using for this purpose in previous designs.

Phase to 45°

The rotation of 90 ° that this block was doing is hard coded. In the DFT block there will be a constant array that will determine for each of the twiddle factors which is the rotation. For example in the 32-point DFT block:

```
TYPE twiddel_rang is array (0 to N/4-2) of STD_LOGIC_VECTOR(1 downto 0);
CONSTANT twid_ran1: twiddel_rang:=("00","00","00","00","01","01","01");
CONSTANT twid_ran2: twiddel_rang:=("00","01","01","01","10","10","11");
```

Where "00" is a rotation of 0°, "01" a rotation of 90°, "10" a rotation of 180° and "11" a rotation of 270°.

CORDIC

The CORDIC algorithm was hard coded. It was found that the different twiddle factors only produces 8 different phases after applying the proper rotation of 90°, 180° or 270° to the original phases. All the different 8 phases will only be present in the 64 DFT block, in the smaller DFTs only some of them are used. These 8 phases can have positives or negatives values, and they are [5.625; 11.25; 16.875; 22.5; 28.125; 33.75; 39.375; 45]. Below, an example of the 32-point DFT block with the original phases and the ones obtained after the modifications is shown.

W_N^k		
Initial phase	**Number of 90° rotations**	**Final phase**
-11.25	0	-11.25
-22.5	0	-22.5
-33.75	0	-33.75
-45	0	-45
-56.25	1	33.75
-67.5	1	22.5
-78.75	1	11.25

Table 7. Final result of the first twiddle factor

W_N^{3k}		
Initial phase	**Number of 90° rotations**	**Final phase**
-33.75	0	-33.75
-67.5	1	22.5
-101.25	1	-11.25
-135	1	-45
-168.75	2	11.25
-202.5	2	-22.5
-236.25	3	33.75

Table 8. Final result of the second twiddle factor

The design then consists on building a block for each of the 8 different phases, the sign of the phase will be handled inside the blocks and is also stored as a constant array in each DFT block. For example, in the 32-point DFT block:

```
TYPE twiddle_sign_array is array (0 to N/4-2) of STD_LOGIC;
-- Sign of the resultant phase after the addition of the corresponding 90 degrees
CONSTANT twid_sign1: twiddle_sign_array := ('1','1','1','1','0','0','0');
-- Sign of the resultant phase after the addition of the corresponding 90 degrees
CONSTANT twid_sign2: twiddle_sign_array := ('1','0','1','1','0','1','0');
```

In the constant array, a negative phase is represented by a '1' and the positive one with a '0'.

In these blocks, the iterations of the CORDIC algorithm for that specific phase will be coded and no other operations apart from the essential ones will be required. The difference among the blocks is the result of the operation $\theta \geq \theta_k$ where θ is the target phase and θ_k is the phase in each iteration. As an example, in the tables that are below it is presented the result of this operation in two different phases.

5.625°	
Iteration	$\boldsymbol{\theta \geq \theta_k}$
1	Yes
2	No
3	No
4	Yes
5	No
6	No

Table 9. Evolution of the CORDIC for the 5.625° phase

28.125°	
Iteration	$\boldsymbol{\theta \geq \theta_k}$
1	Yes
2	No
3	Yes
4	No
5	Yes
6	No

Table 10. Evolution of the CORDIC for the 25.125° phase

Bit growth of the system

Once the final design was done, some other optimizations should be performed. The number of bits necessary in each stage of the DFTs blocks is different and it grows when we perform DFTs with more points. In order to determine the number of bits necessary in each stage, the different operations were evaluated taking into account the highest value that could produce. The result might not be optimal, but it should be really close to it. In fact, in some books [17] it is specified that the relation between the bit growth and the number of point of the FFT is $log_2(N) + 1$, but no demonstrations are provided and therefore in the current implementations our approximation will be used. In the schematics below, there is an example of the bit growth in the 32-point DFT block.

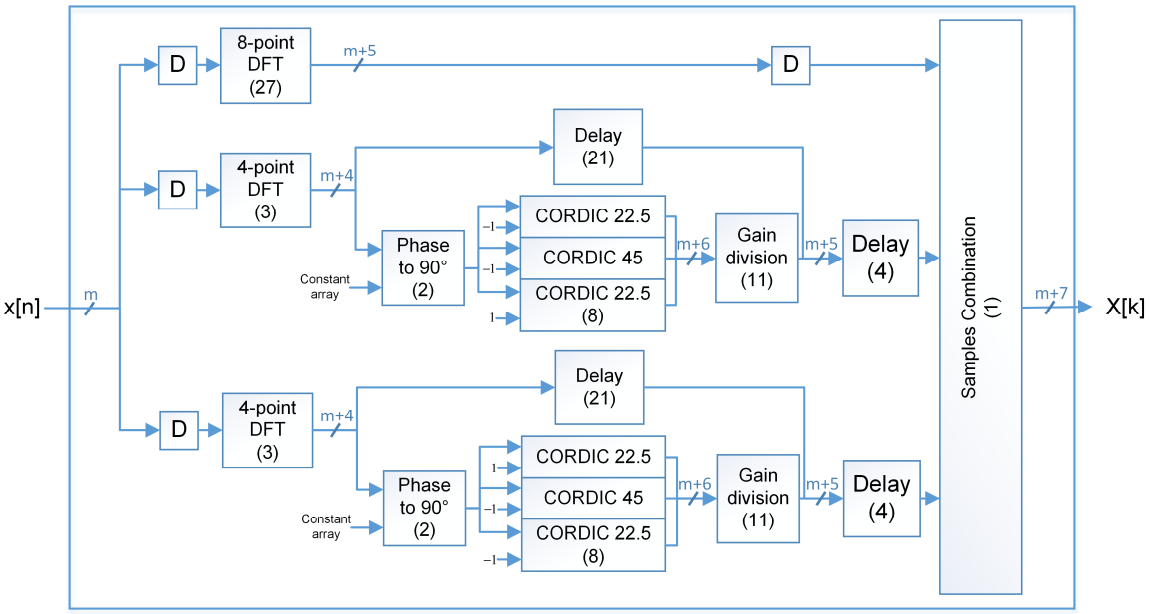


Figure 37. Bit growth in the 32-point DFT with the CORDIC hard coded

Resource utilization

In the final resource utilization, the utilization of Slice LUTs is more or less the same as the previous design, but now we have the advantage of not using any multiplier at all. This should bring the possibility to use several DFT blocks in an implementation of a system. It has to be taken into account that in order to be able to compare the system with the previous one, the resource utilization was done before the design of the bit growth.

Logic Utilization	Used	Available	Utilization
Slice LUTs	78452	1221600	6.42%
LUT as Logic	72494	1221600	5.93%
LUT as Memory	5958	344800	1.73%
Slice Registers	66384	2443200	2.71%
DSPs	0	2160	0%

Table 11. Resource utilization 64-point DFT block with the CORIC hard coded

At the end, we have reached the objective of not using multipliers at all, but using a considerable amount of Slice LUTs. Depending on the type of project where we will use the DFT, we can use one of the different implementation. There is also the possibility to make a mix design where in some stages the multipliers are used and in some other the CORDIC algorithm hard coded, these will depend on the availability of resources of each design.

IDFT

The IFFT blocks were developed once the final design of the DFTs was finished. The last design of the several implementations will be used as a base, so the IDFT will be using the CORDIC algorithm with the hard coded implementation. The structure is the same, so in order not to repeat everything again only the main changes will be detailed:

- Modification of the 4-point DFT block.
- Modification of the phase to 45°.
- Modification of the butterfly of each block which will involve a change in the constant phases and in the combination of the samples.

4-point IDFT

In this block there is a small change in the combination of the samples. The new relationship between the input and the output is:

$$Y = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{bmatrix} \cdot X \qquad (3.16)$$

Phase to 45°

The only modification in this block is the sign in which the input will be rotated. Instead of the previous values, now the rotations values can be -90°, -180° or -270°.

Modifications of the butterfly

The butterfly has to be modified a little bit in order to perform the inverse transform. The twiddle factors will now have different values, but we will also be able to apply the corresponding transformations to get the same 8 phases that we had in the DFT blocks. For example, the following tables represent the new twiddle factors and the final phase shift that we will apply in the 32-point DFT.

W_N^{-k}		
Initial phase	**Number of -90° rotations**	**Final phase**
11.25	0	11.25
22.5	0	22.5
33.75	0	33.75
45	0	45
56.25	1	-33.75
67.5	1	-22.5
78.75	1	-11.25

Table 12. Final result of the first twiddle factor

W_N^{-3k}		
Initial phase	**Number of -90° rotations**	**Final phase**
33.75	0	33.75
67.5	1	-22.5
101.25	1	11.25
135	1	45
168.75	2	-11.25
202.5	2	22.5
236.25	3	-33.75

Table 13. . Final result of the second twiddle factor

In order to apply these transformations, the phase to 45° will also be used. The main difference will be the constant array that stores the shifts necessary for each twiddle factor. For example in the 32-point IDFT block, the constant array of the necessary shifts is:

```
TYPE twiddle_sign_array is array (0 to N/4-2) of STD_LOGIC;
CONSTANT twid_ran1: twiddel_rang:=("00","00","00","00","01","01","01");
CONSTANT twid_ran2: twiddel_rang:=("00","01","01","01","10","10","11");
```

The constant arrays that store the sign values of the final phases will also be changed. Finally, the combination of the samples will also be modified. The final butterfly for the inverse transform using the Split-radix algorithm is shown in Figure 38.

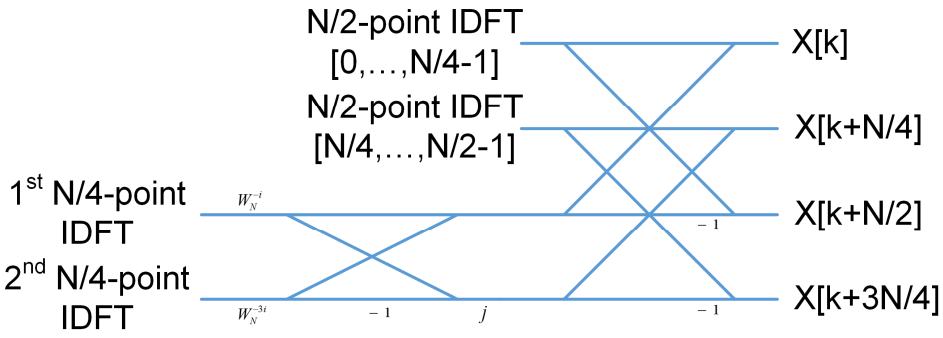


Figure 38. Final butterfly of the Split-radix for the IDFT

3.4. Results

In all the systems that were developed, simulations were done in order to verify that their behaviour was correct. The ISIM simulator that is included in Vivado 2014.1 was used. Then the results of the simulation were compared to the ones obtained in Matlab with the same inputs. For example, these are two screenshots of the results obtained using both programs:

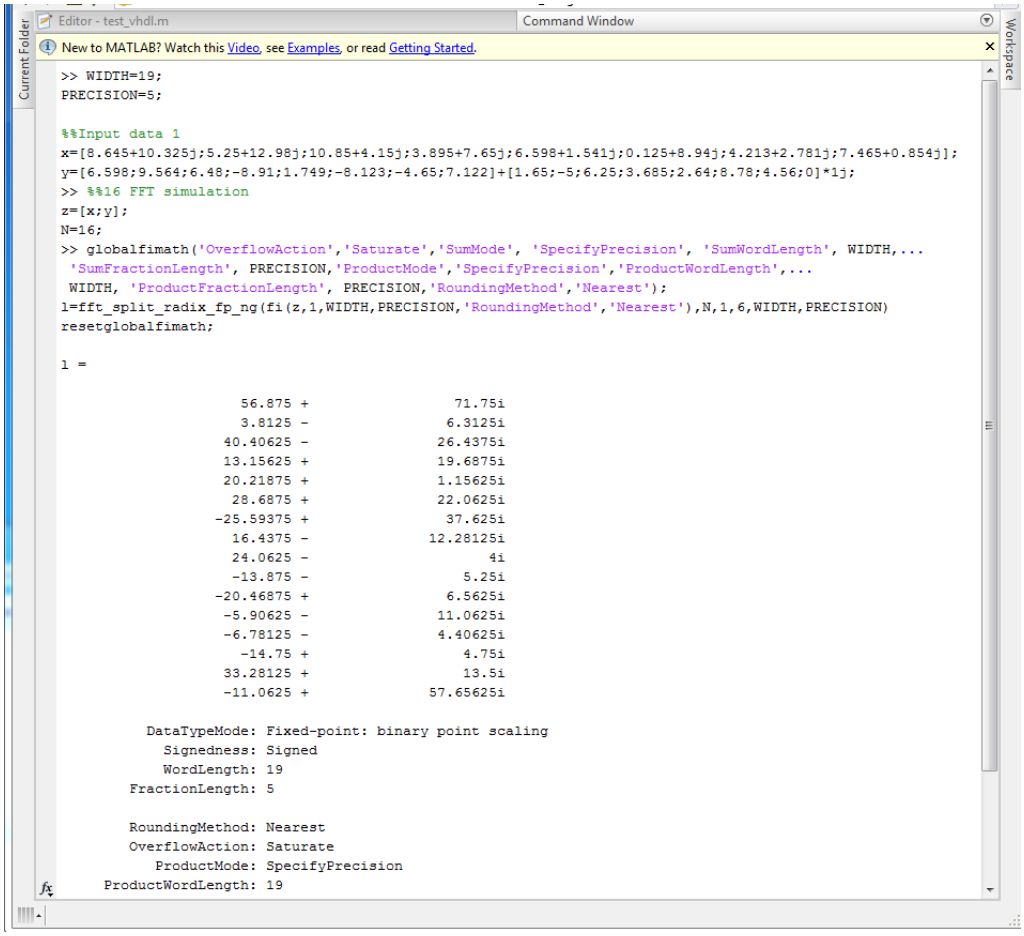


Figure 39. Result of the Split-radix function in Matlab

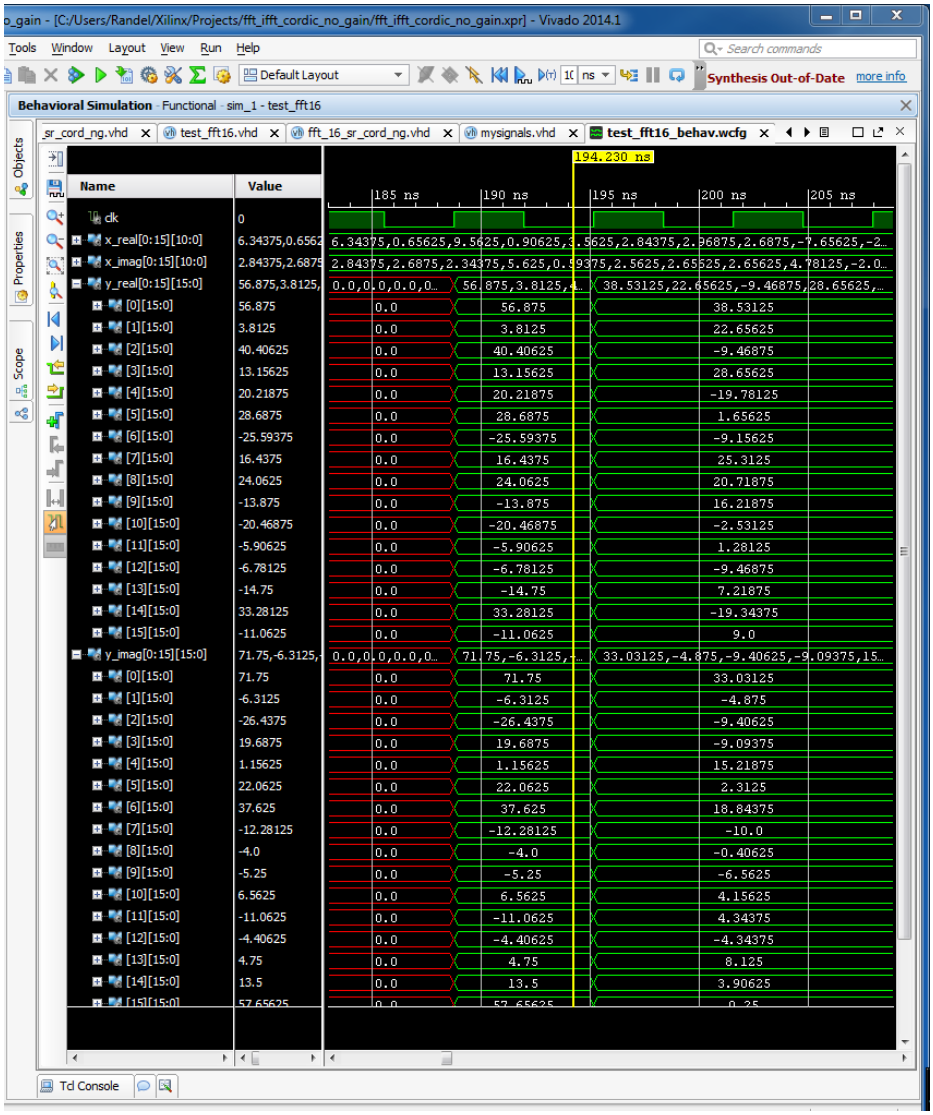


Figure 40. Result of the 64-point DFT block in Vivado

After checking that with several inputs the results were the same in both programs, the solution achieved was considered to be correct. In this case no simulations with a real FPGA were done because for our point of view the results should be the same and it is not really meaningful to see the results in an oscilloscope. In future developments this block will be integrated in bigger designs and then the performance of the overall system will be programmed in an FPGA and tested.

4. Frequency domain filter

The frequency domain filter (FDF) is a useful type of filtering when we want to avoid the complexity of the convolution in time domain. The general idea is to switch to the frequency domain, apply just multiplications there and then come back to the time domain. The main problem that we could encounter with this solution is the aliasing corruption of the samples, but the overlap-save method will be applied to solve it.

First of all, the theory applied in the system will be presented and after that, two major implementations will be detailed. In the first one, all the signals are real and some tricks are applied in order to reduce the resources usage. In the second one, all the signals will be complex and the full implementation has to be done. In this section, the Matlab stage to check the correct implementation of the algorithms will not be done, mainly for two reasons, first because the main blocks are the DFT blocks that have already been tested and second because another approach with a VHDL simulation through Matlab will be developed.

4.1. State of the art

4.1.1. Overlap-save method

The main purpose of the overlap-save method is to perform the calculation of linear convolutions. In the time domain the operation that we want to perform is:

$$y = x * h = \mathrm{F}^{-1}\{\mathrm{F}\{x\} \cdot \mathrm{F}\{h\}\} \tag{4.1}$$

where *x*, *h* and *y* are vectors of the same size.

When we used the overlap-save algorithm, the input signal will be divided into blocks to obtain x_N. These blocks will have a size of N=L+M-1. Then, the filter in time domain will be $h_N=[h_L,0_{M-1}]$, so it will have L taps and will be padded with M-1 zeros. The number of point that will be used in the FFT transform will be N. The result will be:

$$y_N = IFFT\{FFT\{x_N\} \cdot FFT\{h_N\}\} \tag{4.2}$$

The result of the previous equation will present a major problem of aliasing:

$$y_N(n) = \begin{cases} aliasing\ corruption & n = 0..M-2 \\ y_N(n) & n = M-1,\dots,N-1 \end{cases} \tag{4.3}$$

Therefore, the first M-1 samples must be discarded due to the aliasing corruption that introduce the use of the circular convolution. In order to solve this problem, the overlap-save will be applied in the design of the samples that will be in each block of *x*. If in each bock, we use L samples from the previous one, then, we will be able to discard the aliasing sample that we obtain at the output. Through the concatenation of the different samples that will output each block and that are not affected by the aliasing, we will be able to get correct results.

In the Figure 41, a graphical example on how the blocks are assembled and how the signal is discarded at the output is represented.

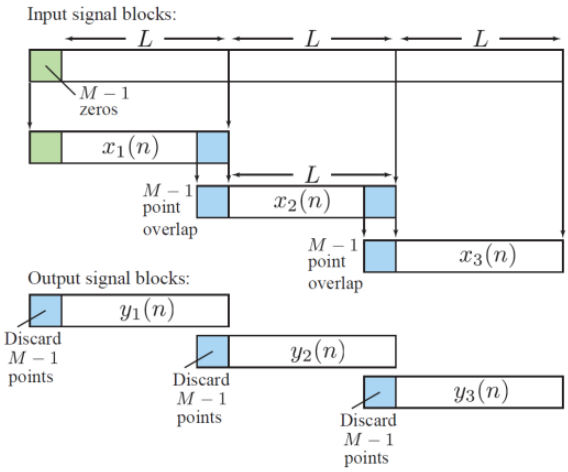


Figure 41. Schematic of the Overlap-save algorithm

4.2. Implementation

The VHDL designs of the different approaches to implement the FDF are presented here. First of all, a description of the general block is done. Later, the resource utilization is given to compare the systems among them. The FPGA that is targeted in this case is the xc7v2000tflg1925-2. We have to take into account that all the implementations are completely functional and features like the bit growth are present in all them.

As in the previous section, in order to develop the different implementations, in addition of the packages already provided by Vivado 2014.1, the package from David Bishop [16] to deal with fixed point in VHDL was used.

The main characteristic of the systems is that in each clock 128 new samples with 6 bits will input the system and therefore 128 samples with 6 will have also to output the system. The general block of the system is:

Figure 42. General block of the FDF

4.2.1. Real input signals implementation

The whole schematic of the system will be the following:

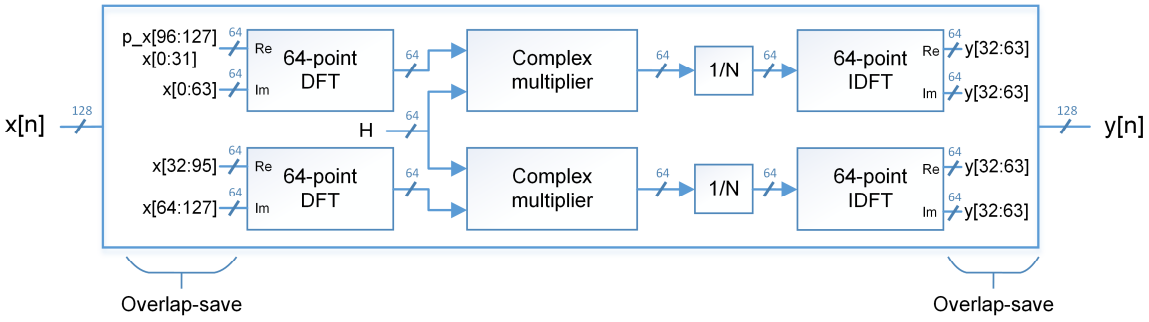


Figure 43. Schematic of the real input implementation of the FDF

The implementation of the overlap-save method is done directly in the inputs of the different DFT blocks with an overlapping percentage of 50%. Most of the blocks have already been presented in previous chapters, the "Complex multiplier" is the same used in the CORDIC generic implementation and the 64-point DFT and IDFT blocks use the CORDIC hard coded design. The last block that has not been presented before is the 1/N block. The behaviour of the block is pretty simple, a shifting of the signal to the right (if the MSB is on the left) is done, in our case it will be shifted 6 positions as the N is 64. This division is performed before the IDFT and not after in order to reduce the word length of the signal in the IDFT block.

The entity of the system will be:

```
entity cma_real is
    GENERIC( NUM_BIT : POSITIVE := N_BITS_IN;
             PRECISION : POSITIVE := PREC;
             NUM_BIT_FILT : POSITIVE := N_BIT_FILT);
    Port ( clk : in STD_LOGIC;
             x : in samples(0 to 127);
             h_real : in samples_filt(0 to 63);
             h_imag : in samples_filt(0 to 63);
             y : out samples_out(0 to 127));
end cma_real;
```

*The signal type are described in another package as

TYPE samples IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BITS_IN-1 DOWNTO 0);

TYPE samples_filt IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BIT_FILT+PREC-1 DOWNTO 0);

TYPE samples_out IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BITS_IN-1 DOWNTO 0);

The trick of this design is to use only two DFT blocks instead of four. Then, one signal is going to be in the real part and another one in the imaginary part. Once in the frequency domain, we will multiply by the coefficients of the filter provided that they are real in the time domain and therefore have Hermitian symmetry in the frequency domain. Finally, we will apply the IDFT, then in the real part of the transform we will have the result of the signal that we inputted in the real input of the DFT block, the same works for the imaginary part. The mathematics demonstration that what we are doing is right is the following:

$$IDFT[DFT[x_1 + jx_2] \cdot (H_r + jH_i)] = IDFT[DFT[x_1] \cdot (H_r + jH_i)] + j\big[IDFT[DFT[x_2] \cdot (H_r + jH_i)]\big] \quad (4.4)$$

We will be able to separate the signal after the process because $IDFT[DFT[x_1] \cdot (H_r + jH_i)]$ and $[IDFT[DFT[x_2] \cdot (H_r + jH_i)]]$ will produce a real result and therefore the signals will not be mixed. The reason is because $DFT[x_1]$ has Hermitian symmetry as x_1 is real and $(H_r + jH_i)$ has also Hermitian symmetry because the impulse response in time fomain is also real. Then, the product of two Hermitian signals produce another Hermitian signal and if we apply the IDFT of a Hermitian signal the result will be real.

Resource utilization

The final resource implementation will be considerably high, but we have to take into account that the complexity of the system built is also important. Later when the implementations with the complex signals will be detailed, we will be able to compare the utilization and find out that we use much less resources in the real implementation than in the complex one.

Logic Utilization	Used	Available	Utilization
Slice LUTs	660907	1221600	54,1%
LUT as Logic	598803	1221600	49,01%
LUT as Memory	62104	344800	18,01%
Slice Registers	551864	2443200	22,58%
DSPs	768	2160	35,55%

Table 14. Resource utilization of the real input implementation

4.2.2. Complex signals implementation

The real implementation although it achieves a really good result, it will not be suitable in all of the cases. Then, the complex implementation will be required despite the higher usage of resources. In order to implement it, several designs were made to figure out which one gives to us the best performance in relationship with the resources used.

64-point DFT block implementation

The main advantage of using a DFT with as many point as possible is the number of taps that the filter can have, in this implementation it can have 33 taps. There are two slightly different implementations that share the same structure, which is presented below.

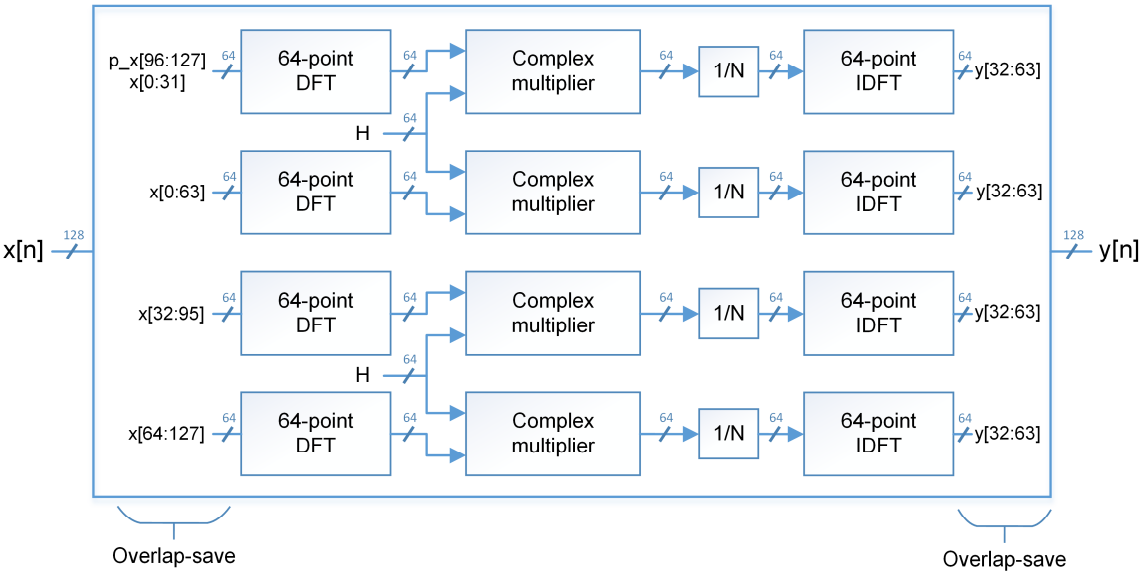


Figure 44. Schematic of the complex implementation of the FDF using 64-point DFT blocks

Mainly, all the blocks that are used have already been described in another section. The "Complex multiplier" was used in the implementation of the DFT with multipliers and the 1/N block was presented in the previous implementation. Regarding the DFT and IDFT blocks, here is where there is a slight difference between the two different implementations. The difference is only in the 64-point DFT block as the smaller DFT blocks used inside it

use the CORDIC hard coded implementation. In the first design, the CORDIC hard coded is used while in the second one an implementation with multipliers is done. The main reason to use multipliers in this last stage is because the usage of the different resources was not equal and a lot of DSP slices were left, this can be noticed in the resource utilization part.

The entity will also be the same for both implementations:

```
entity cma_imag_mult is
    GENERIC( NUM_BIT : POSITIVE := N_BITS_IN;
             PRECISION : POSITIVE := PREC;
             NUM_BIT_FILT : POSITIVE := N_BIT_FILT);
    Port ( clk : in STD_LOGIC;
             x_real : in samples(0 to 127);
             x_imag : in samples(0 to 127);
             h_real : in samples_filt(0 to 63);
             h_imag : in samples_filt(0 to 63);
             y_real : out samples_out(0 to 127);
             y_imag : out samples_out(0 to 127));
end cma_imag_mult;
```

*The signal type are described in another package as

TYPE samples IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BITS_IN-1 DOWNTO 0);

TYPE samples_filt IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BIT_FILT+PREC-1 DOWNTO 0);

TYPE samples_out IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BITS_IN+N_BIT_FILT+11-1 DOWNTO 0);

Resource utilization

The following table shows the result of the implementation with the 64-point DFT using the CORDIC algorithm hard coded. As said before, we can appreciate a huge usage of the Slice LUTs whereas we use less than half of the DSP cells. This may be a problem in the final design as the DSP code is only a part of the whole program that will run the FPGA and some additional resources have to be left for this purpose.

Logic Utilization	Used	Available	Utilization
Slice LUTs	941892	1221600	77,1%
LUT as Logic	867652	1221600	71,02%
LUT as Memory	74240	344800	21,53%
Slice Registers	789340	2443200	32,3%
DSPs	768	2160	35,55%

Table 15. Resource utilization of the FDF implemented with 64-point DFT with CORDIC hard coded

In order to fix the problem of the previous design, this second was made. In this one the 64-point DFT block will be implemented using multipliers, the inner DFT blocks will remain using the CORDIC algorithm hard coded. In the table below, we can now appreciate that a

much more equilibrated usage of the resources is done. As a result, in this implementation there will be plenty of space for the other programmes that will run the FPGA.

Logic Utilization	Used	Available	Utilization
Slice LUTs	701927	1221600	57,45%
LUT as Logic	637275	1221600	52,16%
LUT as Memory	64652	344800	18,75%
Slice Registers	533708	2443200	21,84%
DSPs	1488	2160	68,88%

Table 16. Resource utilization of the FDF implemented with 64-point DFT with multipliers

32-point DFT block implementation

The second implementation uses 32-point DFT blocks instead the 64-point. This will come with the drawback that now the filter will only be able to have 17 taps, but the usage of the resources will be much lower. The general schematic for this implementation is:

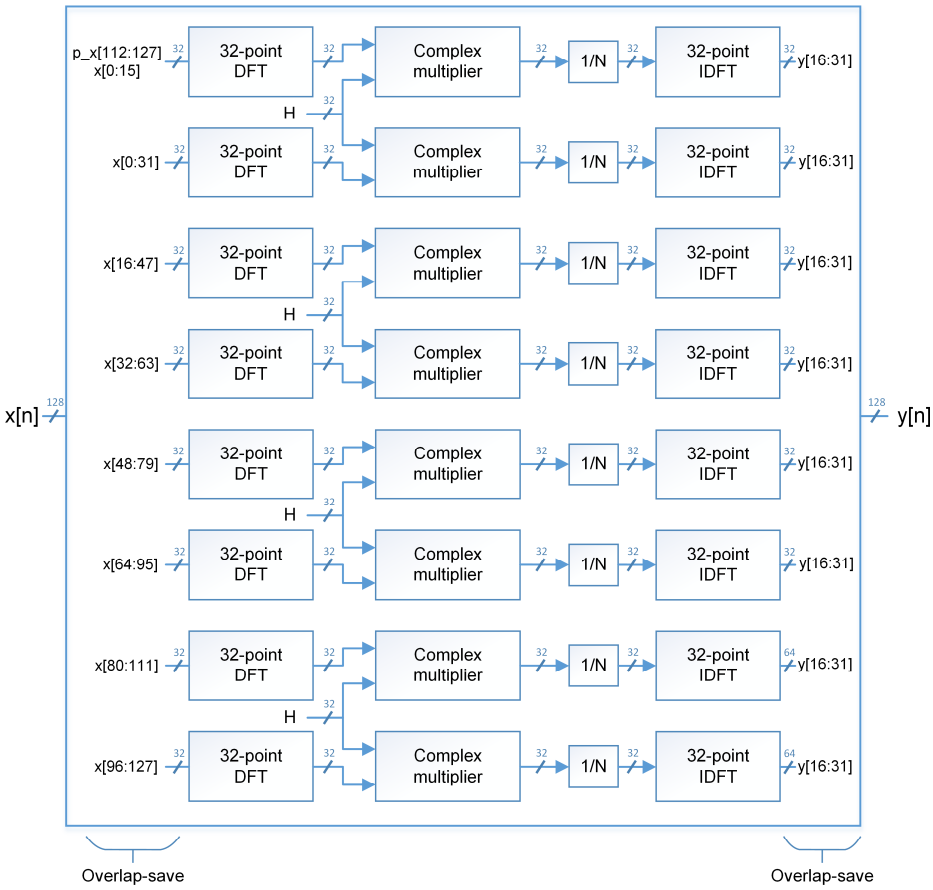


Figure 45. Schematic of the complex implementation of the FDF using 32-point DFT blocks

All of the blocks used have already been presented before. The only clarification that has to be made is that the DFT and IDFT blocks are implemented with the CORDIC hard coded.

The entity of the block will look like this:

```
entity cma_32 is
    GENERIC( NUM_BIT : POSITIVE := N_BITS_IN;
             PRECISION : POSITIVE := PREC;
             NUM_BIT_FILT : POSITIVE := N_BIT_FILT);
    Port ( clk : in STD_LOGIC;
             x_real : in samples(0 to 127);
             x_imag : in samples(0 to 127);
             h_real : in samples_filt(0 to 31);
             h_imag : in samples_filt(0 to 31);
             y_real : out samples_out_32(0 to 127);
             y_imag : out samples_out_32(0 to 127));
end cma_32;
```

*The signal type are described in another package as

TYPE samples IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BITS_IN-1 DOWNTO 0);

TYPE samples_filt IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BIT_FILT+PREC-1 DOWNTO 0);

TYPE samples_out_32 IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(N_BITS_IN+N_BIT_FILT+9-1 DOWNTO 0);

Resource utilization

In this final implementation, the number of resources used will be much lower. In the current targeted FPGA probably this implementation will not be suitable as the performance is lower than the previous ones due to the filter have less taps. However, the FPGA that is currently being targeted is considerably big and this implementation will be useful if in the future more smaller FPGA have to be used.

Logic Utilization	Used	Available	Utilization
Slice LUTs	660907	1221600	54,1%
LUT as Logic	598803	1221600	49,01%
LUT as Memory	62104	344800	18,01%
Slice Registers	551864	2443200	22,58%
DSPs	768	2160	35,55%

Table 17 Resource utilization of the FDF implemented with 32-point DFT with CORDIC hard coded

4.3. Results

4.3.1. Integration of the ISIM simulation in Matlab

In order to simulate the VHDL in Matlab, several solutions were taking into account:

-System C: The VHDL code can be integrated in System C that is a branch of C. Then in Matlab the C code can be compiled as a function. This option was discarded due to there

are many different languages involved and the many problem that one could face to integrate everything.

-Matlab with HDL verifier: Matlab has a toolbox in order to deal with VHDL. Although this may be the best option, it was not the first solution due to the high price of the toolbox.

-Use Vivado in batch mode with Matlab: This seems to be the most straightforward option and it is the one that was implemented. The main advantage is that Vivado can be used in batch mode with all the options that are available in the GUI. The only problem that we will face is that the simulations could not be very fast and could take several minutes.

The procedure is the following, first the data that we have to simulate is written in a text file. Then, Vivado is executed in Matlab with the system command line and a TCL script in Vivado is run. This script configures all the parameters of the simulation and executes the VHDL file where the simulation is defined. This VHDL file reads the data from the text file and after that it writes the result in another text file. Finally, Matlab reads the text file with the results.

4.3.2. Results of the simulations

In order to perform the simulations, the procedure described in the previous point was used. Although that different systems have been implemented, most of them share the same blocks and therefore if one of them is working, the others will work as well. All the systems that have been explained were tested, but only the results of one of them will be presented for the reason explained before. The chosen design is the complex implementation with 64-point with the DFT blocks completely implemented with the CORDIC hard coded.

The frequency domain filter will be used as a transmit filter in order to check the correct behaviour. First, a QSPK signal is generated, then the coefficients of the transmit filter with the *rcosdegin* function are created and transformed to the frequency domain after the corresponding zero stuffing. After that, all this data is given to the VHDL block that will output the result. At the end, the SNR and the BER are computed to have an idea of the numerical noise that the VHDL block has introduced to the system. The result is a SNR value of 36.45 dB and a BER value of 0. If instead of using the VHDL code, we apply the filter using the Matlab convolution function, the SNR is 65 dB and the BER will also be 0. Therefore, the value of the SNR with the VHDL implementation is much lower due to the previous mentioned numerical error introduced by the CORDIC and the fix point design. However, in order to have a decent system the required SNR will be at least 35 dB and as we are above this value, the performance of the system meets the requirements. In the Figure 46, where the constellation after applying the filter is shown, it can be clearly appreciated the effect of the numerical error as we have multiple point in each symbol.

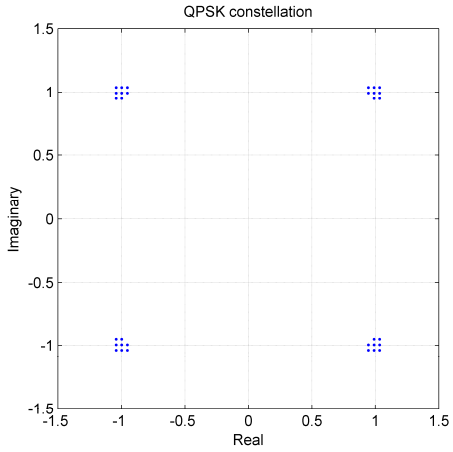


Figure 46. QPSK constellation resultant of the test

The design of the system with the previous parameters (6 CORDIC iterations and 5 bits in the precision part) was successful, but in order to check which will be the result when some of this parameters are modified, some more simulations were done. The parameter modified was the number of precision bits use in the system. If we take a look at the Table 18, we can determine that the minimum number of bits necessary to achieve the required SNR of 35 dB is the value that was used in the initial design. In the pictures that follow the table we can see how the constellation gets worst or improve when we reduced or increase the number of bits in the precision part.

Number of precision bits	BER	SNR (dB)
1	0	17.02
2	0	26.79
3	0	29.42
4	0	33.61
5	0	36.45
6	0	39.47

Table 18. Comparison of the SNR and BER with different number of precision bits

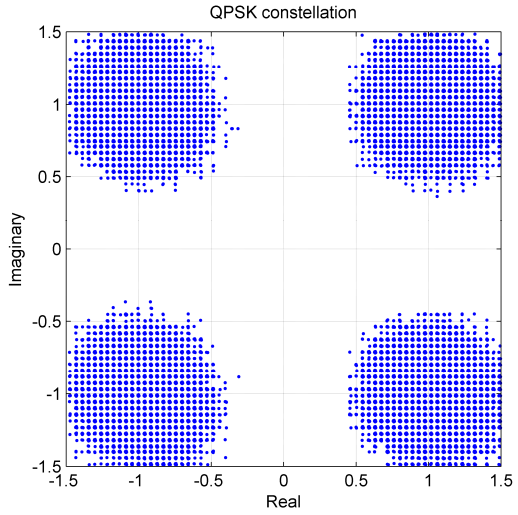


Figure 47. QPSK constellation with 1 precision bit

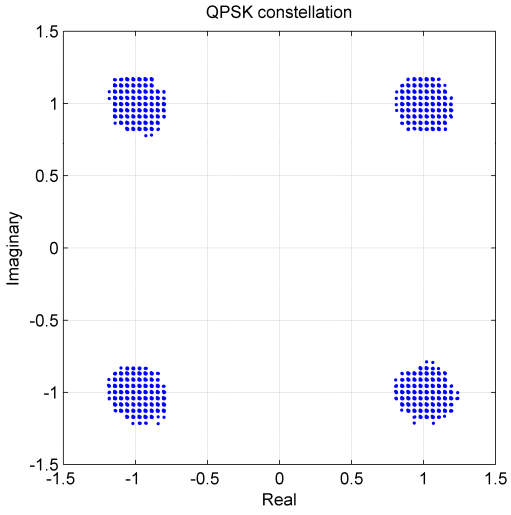


Figure 48. QPSK constellation with 2 precision bits

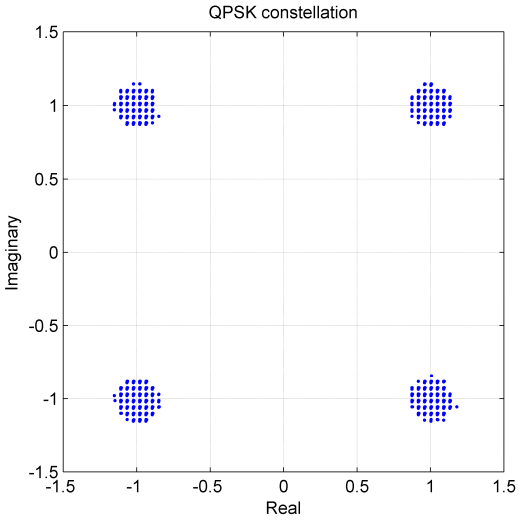


Figure 49. QPSK constellation with 3 precision bits

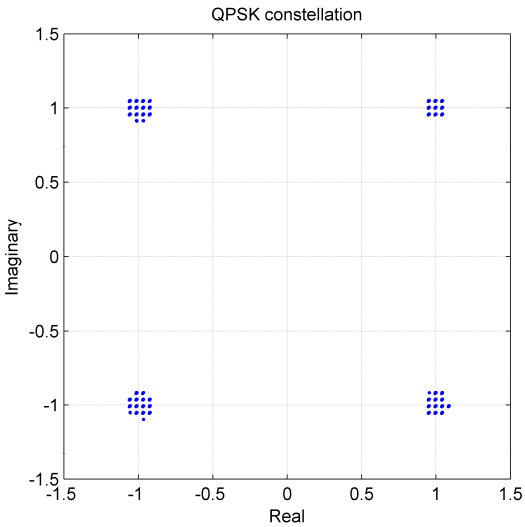


Figure 50. QPSK constellation with 4 precision bits

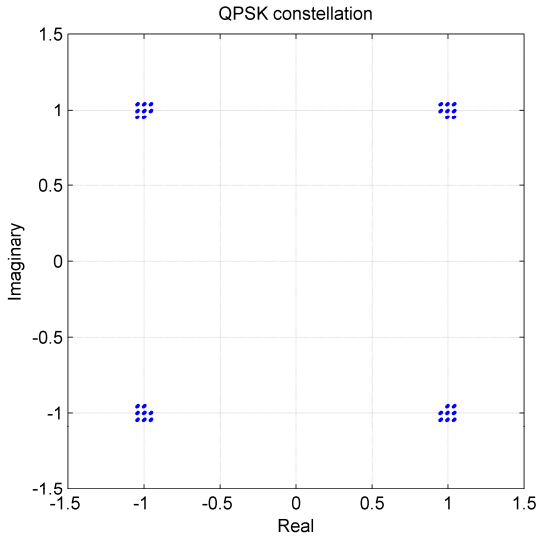


Figure 51. QPSK constellation with 6 precision bits

The conclusion once we analyse the results is that the implementation in VHDL is correct and that the initial value for the parameters that was used in all the designs is the correct one as it satisfy the requirements. Moreover, we can also notice the easiness when we want to change some of the parameters. This can be a really useful utility if in the future the minimum requirements are modified.

5. Budget

The goal of this thesis is to develop software and therefore a prototype has not been developed. For this reason, in this section it will be included a roughly approximation of all the resources used during the thesis. As it is a research center is really difficult to compute the amortization or any other economical parameters and this part will also not be included. The research does not provide a direct input of money for a company as not all the topics that are under research will finish in a commercial product.

In the next tables, the cost commercial cost of the different FPGA that has been targeted during the thesis are shown as well as the prices for the software used, Matlab and Xilinx software. An approximation of the salary of an intern is also included. It has also to be taken into account that there are a lot of different things that have been used trough the development of the thesis like the computer or the oscilloscope that are not included because are not directly related with the thesis. We have also to bear in mind that the cost of the software or hardware is not exclusive for this project and that is also used in other experiments and projects.

FPGA	Cost
xc7v2000tflg1925-2	$25,249
xc7vx690tffg1927-2	$7,209

Matlab	Cost
Individual license	$2,150
Fixed point toolbox	$2,650

Xilinx software	Cost
Vivado System Edition	$4,795

Salary	Cost
$10 per hour	$6,956 (4 months)

6. Conclusions and future development

In this thesis, different DSP algorithms for optical high speed communications have been design and implemented. The first block coded is the digital phase shifter, in this block the cubic Lagrange interpolation is used to interpolate the digital signal in order to be able to delay a signal a fractional number of samples. The main implementation of this algorithm will be to compensate the skew between the in-phase and quadrature components of a complex signal.

The second block implemented is the 64-point DFT using the Split-radix algorithm. Several implementations are made and depending which are the resources available in the final design where the DFT blocks will be used, one of them will be selected. This block is not useful by itself, but it is widely used inside the implementation of other algorithms.

The last block implemented is a FDF, this one also has several implementations and the previous DFT blocks are used. As before, different options are develop to be able to suit different implementation and adapt to the resources available in different FPGA. One of the conclusion that we can get after the different test that have been performed, it is that in most cases the precision in the computation of the DFT is much less than the one that one could expect.

All these algorithms are the base to build a real time MIMO receiver that can be used in the DSP of different systems such as when we are using SDM with a multimode fibre or when we are transmitting using PDM. This would be one of the first MIMO receivers that could work in real time and it will be an important step in order to do experiments that are more close to the final commercial product.

It has been proved that all the presented design work perfectly and therefore the following steps will be the integration of them in real experiments in order to finish the tweaking of the different parameters such as the precision bits or the number of iterations of the CORDIC algorithm.

As a future work, these are some of the main points that will be done during the rest of the stay and that will try to reduce the resource utilization:

- A research in other possible algorithms to perform the interpolation in order to try to improve the results of the digital phase shifter.

- In the implementation of the Split-radix, the number of points in the DFT is limited to powers of two. In order to have a wider choice in the number of points of the DFT, some research should be done about the prime factor DFT. This will also allow different percentages in the overlap-save implementation in the FDF.

- In the current blocks that compute the DFT, the computation of the bit growth is done for a uniform distributed input. However, in a real case scenario probably the input will not be distributed in this way and we will be able to drop some MSB or LSB during the steps of the computation of the DFT. Different real signals could be inputted to the system and the different values at the different stages could be considered to reduce the bit growth.

- A similar approach like the one presented in the last point could be done in order to optimize the bit growth in the different implementations of the FDF.

Finally, once all the optimisations have been performed as well as the implementation of the non-power of two DFT blocks, a system that will feature a 6x6 MIMO receiver will be built. This system will be the final design and it will use some of the blocks that have already been developed. The system is depicted in Figure 52, it will use a 33 % overlap and the filters will have 9 taps. We will also have a block where the odd and even samples (O/E) are separated due to the signal is two times oversampled. The block OL will perform the overlap and the block RL will perform the discard overlap.

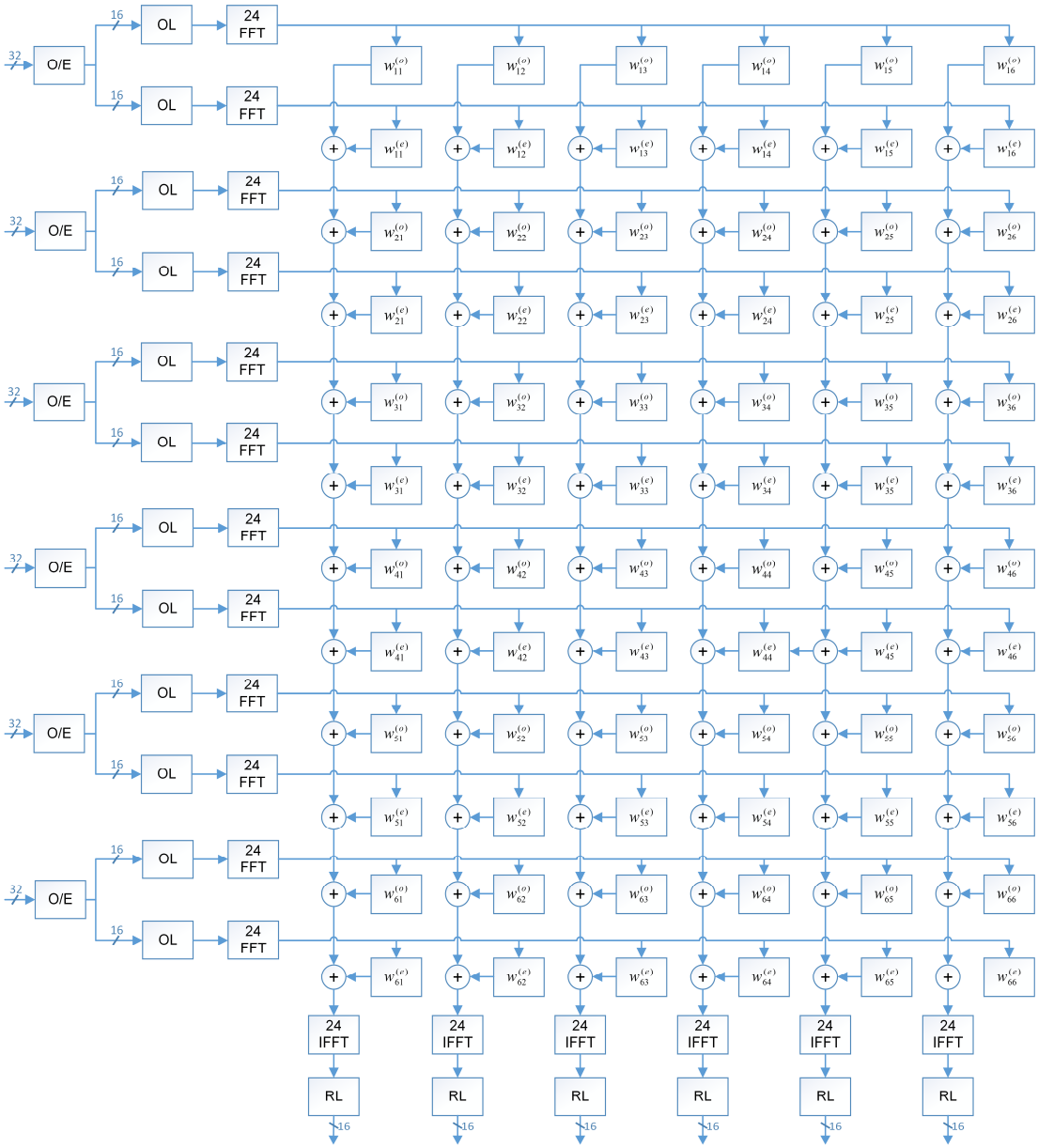


Figure 52. Schematic of a 6x6 MIMO receiver

Once this system will be built the main goal of the internship will be achieved. If there is still some time left, the design of the coefficient adaptation part of the receiver will be started.

As a conclusion, all the work that has been presented in this thesis will be really important for the implementation of the final design. First of all, because some of the blocks that have been developed will be included in it. Furthermore, all the knowledge about the FPGA field that has been acquired will be key factor in order to be able to squeeze all final design in one FPGA board

Bibliography

[1] R.-J Essiamdre et al. Capacity limits of optical fiber networks, J. Lightwave Technol. 28 (4) (2010) 662-701.

[2] C. E. Shannon, “A mathematical theory of communication", The Bell Systems Technical Journal, vol. 27, pp. 379{423, 623{656, July-October 1948.

[3] Ivan P. Kaminow, Tingye Li, AlanE. Willner, *Optical Fiber Telecommunication VIB*, 6th ed. Academic Press, USA, 2013

[4] R. W. Tkach, Scaling optical communications for the next decade and beyond, Bell Labs Tech. J. 14 (4) (2010) 3-9.

[5] Tucker, R. S. IEEE J. Sel. Top. Quantum Electron. 17, 245–260 (2011).

[6] Heincrich Meyr, Marc Moeneclaey, Stefan A. Fechtel, *Digital communications receivers* 1st ed.Wiley-interscience Publication, 1998

[7] Vivado Design Suite “LogiCORE IP Fast Fourier Transform v9.0” [Online] Available:

http://japan.xilinx.com/support/documentation/ip_documentation/xfft/v9_0/pg109-xfft.pdf [Accessed: 07 July 2014].

[8] Cooley, J. W. and Tukey, J. W. (1965). “An Algorithm for the Machine Calculation of Complex Fourier Series”. Math. Computat, 19, 297–301.

[9] D. Jones, "Decimation-in-tome (DIT) Radix-2 FFT" OpenStax-CNX. September 15, 2006. Available: http://cnx.org/content/m12016 [Accessed: 15 June 2014]

[10] D. Jones, "Radix-4 FFT Algorithms" OpenStax-CNX. September 18, 2006. Available: http://cnx.org/content/m12027 [Accessed: 15 June 2014]

[11] P. Duhamel and H. Hollman. (1984, Jan 5). “Split-radix FFT algorithms”. Electronics Letters, 20, 14-16.

[12] R. Yavne. (1968). “An economical method for calculating the discrete Fourier transform”. Proc. AFIPS Fall Joint Computer Conf., 33, 115-125.

[13] S.G Johnson and M. Frigo. (2006). A modified split-radix FFT with fewer arithmetic operations. IEEE Transactions on Signal Processing, 54.

[14] D. Jones, "Split-radix FFT Algorithms." OpenStax-CNX. November 2, 2006. Available: http://cnx.org/couglasontent/m12031/1.5/ [Accessed: 15 June 2014]

[15] Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, September 1959

[16] David Bishop. “Fixed point design for VHDL”. [Online] Available: http://www.vhdl.org/fphdl/ [Accessed: 25 June 2014]

[17] Edmund Lai. *Practical Digital Signal Processing*, 1st ed. Newnes: 2003. Page 72

Appendices

Interpretation of the resource utilization tables

In different sections of the thesis, the following table is presented to give an idea of which is the resource utilization of each design. In this appendices, a brief explanation of the meaning of the different parameters that are presented will be detailed. The definitions that will be presented correspond to the Virtex 7 family (the different Xilinx families have different definitions of all these categories) as all the FPGA used in the thesis correspond to this family.

Logic Utilization	Used	Available	Utilization
Slice LUTs			
LUT as Logic			
LUT as Memory			
Slice Registers			
DSPs			

A Slice is 4 LUTs (lookup tables) and 8 flip-flops.

A LUT is either a 6-input lookup table with 1 output, or two 5-input lookup tables with separate outputs but common inputs. The output(s) of the LUT(s) in a Slice can connect to the 8 flip-flips in the Slice or be brought out of the Slice unregistered.

All Slices can be used for logic, but the LUTs in some Slices can be used as memory, which is why there are entries for "LUT as Logic" and "LUT as memory". A LUT in a Slice used as memory implements 64 bits (since there are 6 inputs to the LUT).

The "Slice Registers" refers to how many of the 8 Slice flip-flops are used in the design (the LUTs can be used in a Slice without using the flip-flops).

A DSP block is an adder, a 25x18 multiplier, and a 48 bit accumulator, built in hard logic.

As a conclusion, when we want to compare the results of the different implementations, the same FPGA has to be targeted and then with an easy inspection of the utilization we will be able to figure out which design uses less resources.

FPGA used in the project

In this project two different FPGA are targeted, in this section a table with the main characteristics and the resource available in each of them is presented. Both of them correspond to the Virtex 7 family produced by Xilinx and they are one of biggest FPGA available in the market. The two FPGA that are being used are highlighted in yellow in the table.

Virtex-7 FPGA Feature Summary

Table 6: **Virtex-7 FPGA Feature Summary**

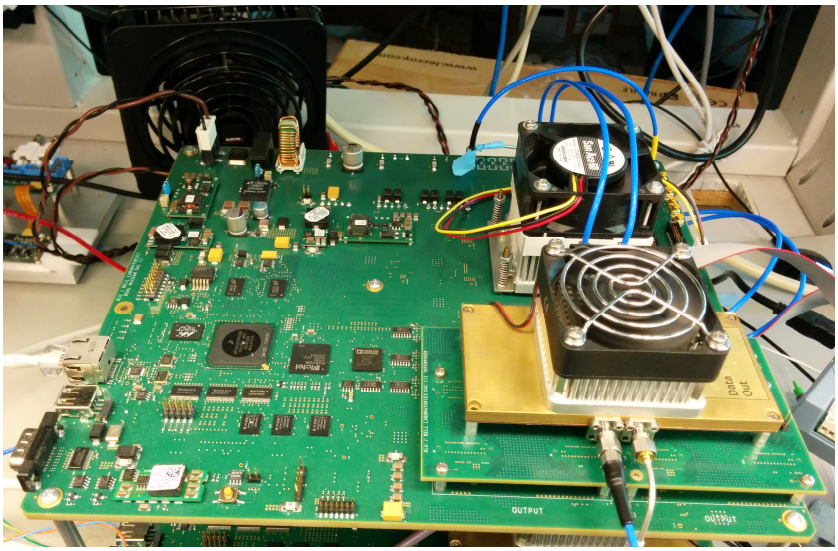
Device[1]	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices[3]	Block RAM Blocks[4]			CMTs[5]	PCIe[6]	GTX	GTH	GTZ	XADC Blocks	Total I/O Banks[7]	Max User I/O[8]	SLRs[9]
		Slices[2]	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)									
XC7V585T	582,720	91,050	6,938	1,260	1,590	795	28,620	18	3	36	0	0	1	17	850	N/A
XC7V2000T	1,954,560	305,400	21,550	2,160	2,584	1,292	46,512	24	4	36	0	0	1	24	1,200	4
XC7VX330T	326,400	51,000	4,388	1,120	1,500	750	27,000	14	2	0	28	0	1	14	700	N/A
XC7VX415T	412,160	64,400	6,525	2,160	1,760	880	31,680	12	2	0	48	0	1	12	600	N/A
XC7VX485T	485,760	75,900	8,175	2,800	2,060	1,030	37,080	14	4	56	0	0	1	14	700	N/A
XC7VX550T	554,240	86,600	8,725	2,880	2,360	1,180	42,480	20	2	0	80	0	1	16	600	N/A
XC7VX690T	693,120	108,300	10,888	3,600	2,940	1,470	52,920	20	3	0	80	0	1	20	1,000	N/A
XC7VX980T	979,200	153,000	13,838	3,600	3,000	1,500	54,000	18	3	0	72	0	1	18	900	N/A
XC7VX1140T	1,139,200	178,000	17,700	3,360	3,760	1,880	67,680	24	4	0	96	0	1	22	1,100	4
XC7VH580T	580,480	90,700	8,850	1,680	1,880	940	33,840	12	2	0	48	8	1	12	600	2
XC7VH870T	876,160	136,900	13,275	2,520	2,820	1,410	50,760	18	3	0	72	16	1	6	300	3

Notes:
1. EasyPath™-7 FPGAs are also available to provide a fast, simple, and risk-free solution for cost reducing Virtex-7 T and Virtex-7 XT FPGA designs
2. Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs.
3. Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.
4. Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.
5. Each CMT contains one MMCM and one PLL.
6. Virtex-7 T FPGA Interface Blocks for PCI Express support up to x8 Gen 2. Virtex-7 XT and Virtex-7 HT Interface Blocks for PCI Express support up to x8 Gen 3, with the exception of the XC7VX485T device, which supports x8 Gen 2.
7. Does not include configuration Bank 0.
8. This number does not include GTX, GTH, or GTZ transceivers.
9. Super logic regions (SLRs) are the constituent parts of FPGAs that use SSI technology. Virtex-7 HT devices use SSI technology to connect SLRs with 28.05 Gb/s transceivers.

This table was obtained from the Xilinx website, more information and the whole document can be accessed at:

http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

In order to have an idea about how the FPGA looks like once that has been assembled in the corresponding board, below there is a picture of the whole board that contains it.



Glossary

CD	Chromatic Dispersion
CMA	Constant Modulus Algorithm
CORDIC	Coordinate Rotation Digital Computer
DAC	Digital to Analogue Converter
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
FDF	Frequency Domain Filter
FEC	Forward Error Correction
FFT	Fast Fourier Transform
FPGA	Field Programmable Gates Array
IDFT	Inverse Discrete Fourier Transform
LMS	Least-Mean Square
LUT	Look Up Table
MIMO	Multiple-Input Multiple-Output
MMSE	Minimum Mean Square Error
PDM	Polarization Division Multiplexing
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
SDM	Space-Division Multiplexing
SMF	Single Mode Fibre
SNR	Signal to Noise Ration
VHDL	Very High Description Language
WDM	Wavelength-Division Multiplexing