HARDWARE IMPLEMENTATION OF OFDM ON FPGA FOR SDR APPLICATIONS

By

NC Saad ur Rehman Fazal
PC Mariam Khizar
NC Usama Karim

Submitted to the Faculty of Electrical Engineering, Military College of Signals National University of Sciences and Technology, Rawalpindi in partial fulfillment for the requirements of a B.E Degree in Telecom Engineering
JULY 2011

ABSTRACT

The project is to develop a digital communication system that would implement OFDM on FPGA. The system would be able to transmit and receive voice or data between two separate FPGA kits. It handles all the baseband processing of an OFDM system and an RF Front-end could be connected for wireless transmission.

The aim of the project is to establish a system that would be able to communicate at higher data rates. The FPGA is specifically being used for its fast, real-time processing capabilities along with the efficient high speed RAM made up of flip-flops and logic gates. A prototype code of the OFDM system has been established using Verilog HDL. The prototype includes various OFDM blocks with specifications of 802.11a standard. It incorporates a 16-point FFT block and a 2/3 convolutional encoder. The FPGA used for implementation is Xilinx Spartan 3E 1600K.

The developed prototype has been tested on software by simulating it on the Xilinx ISE suite11.1 and the results were in accordance with the theoretically calculated ones. A bit stream is introduced as an input signal in place of audio data for testing purposes. The developed code was also synthesized using the Xilinx environment to make sure that the required resources do not exceed the available ones. The Code was tested successfully on the hardware after the implementation of the code on FPGA and the data has been transmitted over a wired medium. Voice is fed as an input at the transmitter end and recovered back at the receiver end.

DECLARATION

Declaration: - No portion of the work presented in this dissertation has been submitted in support of another award or qualification either at this institution or elsewhere

Dedicated to

Almighty Allah,

Faculty for their help

and Our Parents for their support and care.

ACKNOWLEDGMENTS

We would like to take this opportunity to express our deepest gratitude to our supervisor, Dr. Adnan Rashdi for his guidance, help and encouragement throughout the period of completing our project.

We would like to thank to Ma`am Saba Zia of SEECS, NUST for all the help and guidance especially in using Xilinx Spartan development board and configuration of the related software.

We would also like to sincerely thank all our friends and all those, whoever has helped us either directly or indirectly, in the completion of our final year project and thesis.

TABLE OF CONTENTS

LIST OF FIGURES

KEY TO SYMBOLS

ADC	Analog to Digital Converter
ADSL	Asymmetric Digital Subscriber Line
ASIC	Application-specific integrated circuit
AWGN	Additive white Gaussian noise
BPSK	Binary Phase Shift Keying
CPLD	Complex Programmable Logic Device
DAC	Digital to analog Converter
DAB	Digital Audio Broadcast
DVB	Digital Video Broadcast
FDMA	Frequency Division Multiple Access
FEC	Forward Error Correction
FFT	Fast Fourier Transform
HDL	Hardware description Language
IOB	Input Output Blocks
IFFT	Inverse Fast Fourier Transform
ICI	Inter Carrier Interference
ISI	Inter Symbol Interference
LUT	Look up Tables
MMAC	Multimedia Mobile Access Communication
Mux	Multiplexer
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
SPI	Serial Peripheral Interface

UART	Universal asynchronous receiver/transmitter
UCF	User Constraints File
USRP	Universal Software Radio Peripheral
VLSI	Very Large Scale Integration

Chap 1

Introduction

1.1. Overview

This chapter gives the basic information about the project. The chapter covers the material on project background, project objectives, project scope and the thesis outline. The problem statement of the project will also be carried out in this chapter.

1.2. Project Background

With the rapid growth of digital communication in recent years, the need for high-speed data transmission has been increased. The mobile telecommunications industry faces the problem of providing the technology that be able to support a variety of services ranging from voice communication with a bit rate of a few kbps to wireless multimedia in which bit rate up to 100 Mbps. Many systems have been proposed and OFDM system has gained much attention for different reasons.

Although OFDM was first developed in the 1960s, only in recent years, it has been recognized as an outstanding method for high-speed cellular data communication where its implementation relies on very high-speed digital signal processing. This method has

only recently become available with reasonable prices versus performance of hardware implementation.

Since OFDM is carried out in the digital domain, there are several methods to implement the system. One of the methods to implement the system is using ASIC (Application Specific Integrated Circuit). ASICs are the fastest, smallest, and lowest power way to implement OFDM into hardware. The main problem using this method is inflexibility of design process involved and the longer time to market period for the designed chip.

Another method that can be used to implement OFDM is general purpose Microprocessor or Micro Controller. Power PC 7400 and DSP Processor is an example of microprocessor that is capable to implement fast vector operations. This processor is highly programmable and flexible in term of changing the OFDM design into the system. The disadvantages of using this hardware are, it needs memory and other peripheral chips to support the operation. Besides that, it uses the most power usage and memory space, and would be the slowest in term of time to produce the output compared to other hardware.

Field-Programmable Gate Array (FPGA) is an example of VLSI circuit which consists of a “sea of NAND gates” whereby the function are customer provided in a “wire list”. This hardware is programmable and the designer has full control over the actual design implementation without the need (and delay) for any physical IC fabrication facility. An FPGA combines the speed, power, and density attributes of an ASIC with the programmability of a general purpose processor will give advantages to the OFDM system. An FPGA could be reprogrammed for new functions by a base station to meet

future needs particularly when new design is going to fabricate into chip. This will be the best choice for OFDM implementation since it gives flexibility to the program design besides the low cost hardware component compared to others.

1.3. Project Objective

The aim for this project is to design a baseband OFDM processing including FFT (Fast Fourier Transform) and IFFT (Inverse Fast Fourier Transform), mapping (modulator), Guard Insertion and scrambling and Convolutional Encoding using hardware programming language (Verilog HDL). These designs were developed using Verilog HDL programming language in design entry software provided along with the FPGA kits available i.e. Xilinx Spartan 3E. Software used is Xilinx ISE v11.1.

The design is then implemented in the Digilent FPGA development board with Xilinx Spartan 3E FPGA. Description on the development board will be carried out at methodology chapter.

Several tools involved in the process of completing the design in real hardware which can be divided into two categories, software tools and hardware tools. The softwares used in this project are MATLAB 7.9.1 and Simulink for Simulation purposes and Xilinx ISE v11.1 including Xilinx Plan Ahead and iSim modules. While the hardware used is Xilinx Spartan 3E FPGA with 1600K Gates (xc3s1600e-5fg320) for real-time hardware implementation.

1.4. Project Scope

The work of the project will be focused on the design of the processing block which is 16 point IFFT and FFT function. The design also includes mapping block, Convolutional Encoding, Scrambling and Guard Insertion block sets. All design needed to be verified to ensure that there are no errors in Verilog programming before being simulated. Design process will be described on the methodology chapter.

The second scope is to implement the design into FPGA hardware development board. This process is implemented if all designs are correctly verified and simulated using particular software. Implementation includes hardware programming on FPGA or downloading hardware design into FPGA and software programming.

Creating test bench program also include in the scope of the project. Test bench is a program developed using Verilog programming and is intended as the input interface for user as well as to control data processing performed by the hardware. Creating this software required in understanding the operation of the FFT and IFFT computation process. Further chapter will discuss the method on developing the program from mathematical algorithm into behavioral synthesis.

The above is to verify the result of the output for each module which has been developed. Test bench program is used to deliver the computation result if input value is provided by

the user. These computation values should be verified and tested to ensure the correctness of the developed module. ‘iSim’ Software is used to compare the computation performed by the FPGA hardware with the software. There are several test performed to the design modules and the test process also will be discuss in the methodology chapter.

1.5. Organization of Report

The report about the project covers all the necessary information required to understand the proposed digital communication system. It also emphasizes on why there was a need to develop it and how did we achieve it.

The first chapter contains the introductory notes about the project. It describes the objectives and the scope of the project. The need for a modern high speed communication system is introduced in the project background.

The second chapter gives an overview of the proposed OFDM system and discusses all the relevant theoretical information available for the system and the FPGAs. We will discuss the specifications of the kit used in the project later on in the report. The relevant applications of OFDM and FPGA are also given in this chapter to emphasize on their importance.

The third chapter deals with the development of the project according to the objectives established and what techniques were used to achieve those objectives. The procedure followed to implement the system is discussed in this chapter. The blocks of OFDM used

and implemented are explained in detail and later on the technique to code them in Verilog and then burn them on FPGA are also discussed. Later on the use of DAC and A/D in the FPGA kit is also mentioned.

The fourth chapter deals with the analysis of the project. To quantify the completion at every stage the results were compared and analyzed. The simulation in MATLAB environment and then in Xilinx ISE environment are presented in this chapter. The results of signal outputs in the timing diagram and the synthesis reports for every module are shown and explained in the chapter.

Chapter 2

Design and Development

2.1. Digital Communication System Structure

A digital communication system involves the transmission of information in digital form from one point to another point as shown in Figure 1.1

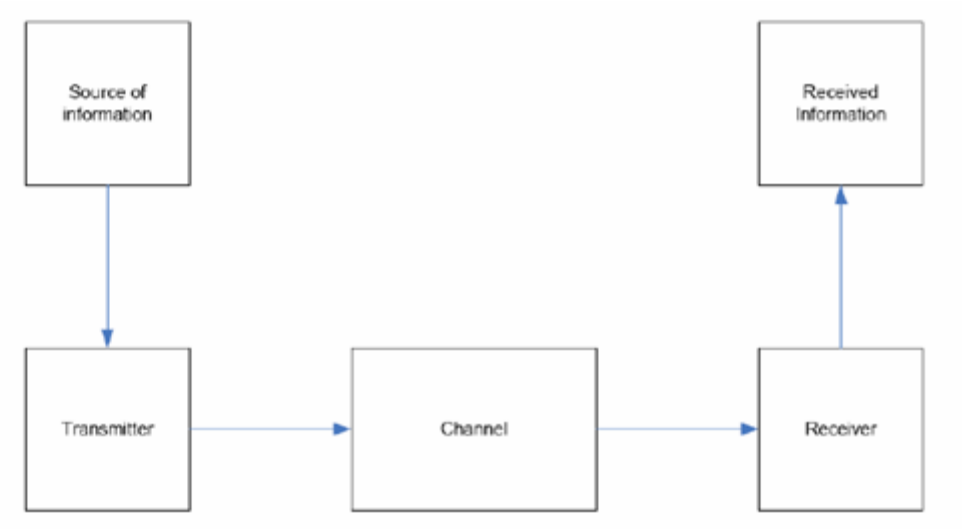

Figure 2-1: A Typical Digital Transmission System

Regardless of the form of communication method, the three basic elements in a communication system consist of transmitter, channel and receiver[1].

The source of information is the messages that are to be transmitted to the other end in the receiver. A transmitter can consist of source encoder, channel encoder and

modulation. Source encoder employed an efficient representation of the information such that resources can be conserved. A channel encoder may include error detection and correction code. The aim is to increase the redundancy in the data to improve the reliability of transmission. A modulation process convert the base band signal into band pass signal before transmission.

During transmission, the signal experiences impairment which attenuates the signals amplitude and distort signals phase. Also, the signals transmitting through a channel also impaired by noise, which is assumed to be Gaussian distributed component.

In the receiver end, the reversed order of the steps in the transmitter is performed. Ideally, the same information must be decoded in the receiving end.

2.2. Multichannel Transmission

OFDM started in the mid 60's, Chang [2] proposed a method to synthesis band limited signals for multi channel transmission. The idea is to transmit signals simultaneously through a linear band limited channel without inter channel (ICI) an inter symbol interference (ISI).

After that, Saltzberg [3] performed an analysis based on Chang's work and he conclude that the focus to design a multi channel transmission must concentrate on reducing crosstalk between adjacent channels rather than on perfecting the individual signals.

In 1971, Weinstein and Ebert [4] made an important contribution to OFDM. Discrete Fourier transform (DFT) method was proposed to perform the base band modulation and demodulation. DFT is an efficient signal processing algorithm. It eliminates the banks of sub carrier oscillators. They used guard space between symbols to combat ICI and ISI problem. This system did not obtain perfect orthogonality between sub carriers over a dispersive channel.

It was Peled and Ruiz [5] in 1980 who introduced cyclic prefix (CP) that solves the orthogonality issue. They filled the guard space with a cyclic extension of the OFDM symbol. It is assume the CP is longer than impulse response of the channel.

2.3. Basic Principles of OFDM

Orthogonal Frequency Division Multiplexing (OFDM) is a multi-carrier transmission technique, which divides the available spectrum into many carriers, each one being modulated by a low rate data stream.

OFDM is similar to FDMA in that the multiple user access is achieved by subdividing the available bandwidth into multiple channels that are then allocated to users. However, OFDM uses the spectrum much more efficiently by spacing the channels much closer together. This is achieved by making all the carriers orthogonal to one another, preventing interference between the closely spaced carriers.

2.3.1. Orthogonality Defined

Orthogonality is defined for both real and complex valued functions. The functions $\phi_m(t)$ and $\phi_n(t)$ are said to be orthogonal with respect to each other over the interval $a < t < b$ if they satisfy the condition:

$$\int_a^b \varphi_m(t)\,\varphi_m^*(t)dt = 0, \quad \text{Where } n \neq m$$

OFDM splits the available bandwidth into many narrowband channels (typically 100-8000), each with its own sub-carrier[7]. These sub-carriers are made orthogonal to one another, meaning that each one has an integer number of cycles over a symbol period. Thus the spectrum of each sub-carrier has a "null" at the centrefrequency of each of the other sub-carriers in the system, as demonstrated in Figure 2.0 below.

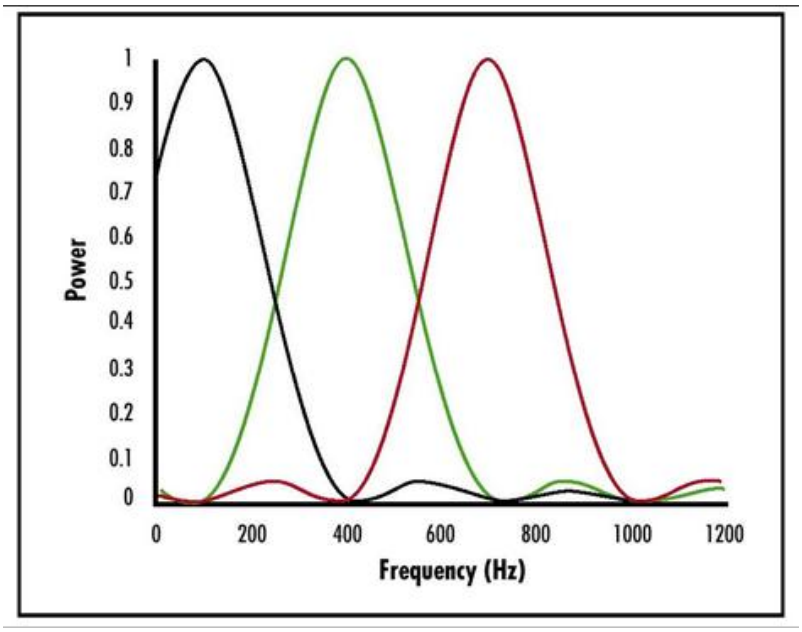

Figure 2-2: Orthogonality of sub-carriers

This results in no interference between the sub-carriers, allowing then to be spaced as close as theoretically possible. Because of this, there is no great need for the users of the channel to be time-multiplexed, and there is no overhead associated with switching between users. This overcomes the problem of overhead carrier spacing required in FDMA.

2.3.2. OFDM Carriers

As fore mentioned, OFDM is a special form of Multi Carrier Modulation (MCM) and the OFDM time domain waveforms are chosen such that mutual orthogonality is ensured even though sub-carrier spectra may over-lap. With respect to OFDM, it can be stated that orthogonality is an implication of a definite and fixed relationship between all carriers in the collection.

It means that each carrier is positioned such that it occurs at the zero energy frequency point of all other carriers. The sinc function, illustrated in Figure 2-3 exhibits this property and it is used as a carrier in an OFDM system.f_u is the sub-carrier spacing

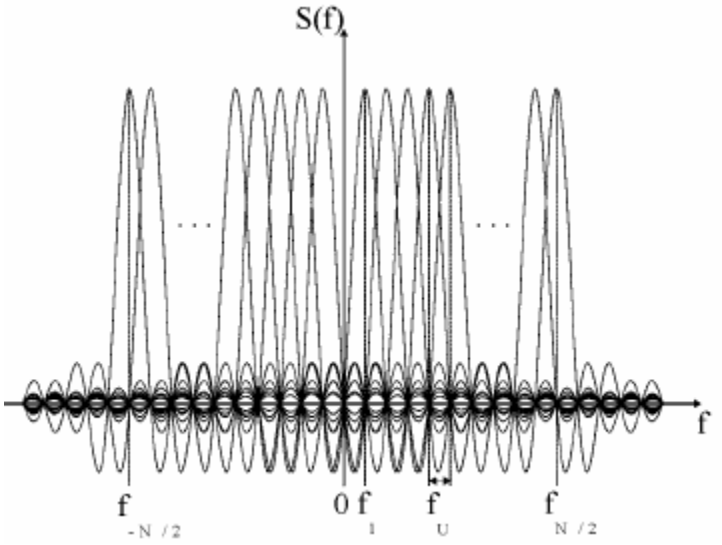

Figure 2-3: OFDM sub carriers in the frequency domain

2.3.3. Generation of OFDM Signals

To implement the OFDM transmission scheme, the message signal must first be digitally modulated. The carrier is then split into lower-frequency sub-carriers that are orthogonal to one another. This is achieved by making use of a series of digital signal processing operations.

The message signal is first modulated using a modulation scheme such as BPSK, QPSK, or some form of QAM (16QAM or 64QAM for example). In BPSK, each data symbol modulates the phase of a higher frequency carrier. Figure 2.2 shows the time domain representation of 8 symbols (01011101) modulated within the carrier using BPSK. In the frequence domain, the effect of the phase shifts in the carrier is to expand the bandwidth occupied by the BPSK signal to a Sinc function. The zeros (or "nulls") of the sinc frequency occur at intervals of the symbol frequency.

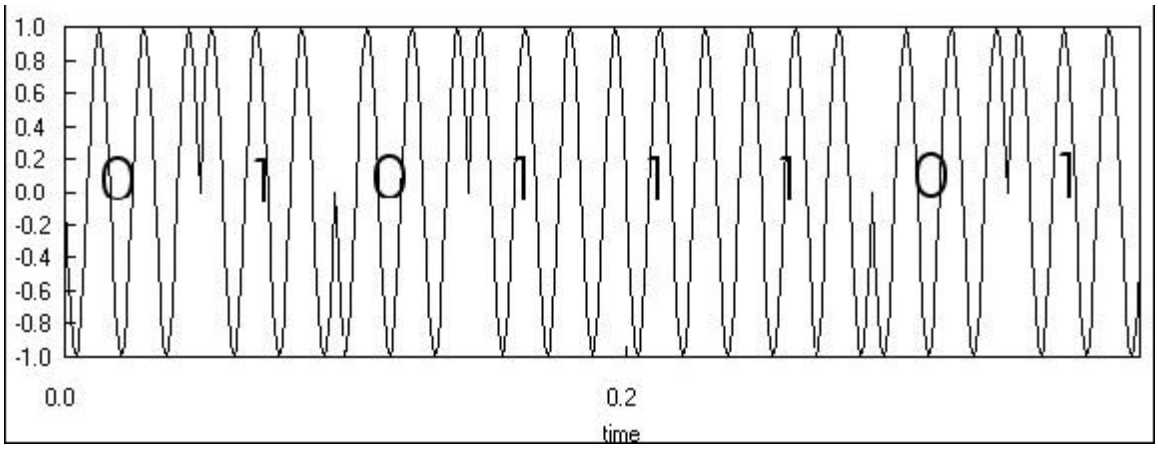

Figure 2-4**:** Binary Phase-Shift Key (BPSK) representation of "01011101"

Originally, multicarrier systems were implemented through the use of separate local oscillators to general each individual sub-carrier. This was both inefficient and costly. With the advent of cheap powerful processors, the sub-carriers can now be generated using Fast Fourier Transforms (FFT). The FFT is used to calculate the spectral content of the signal. It moves a signal from the time domain where it is expressed as a series of time events to the frequency domain where it is expressed as the amplitude and phase of a particular frequency. The inverse FFT (IFFT) performs the reciprocal operation.

The underlying principle here is that FFT can keep tones orthogonal to one another if the tones have an integer number of cycles in a symbol period. In the example below we see signals with 1, 2 and 4 cycles respectively that form an orthogonal set.

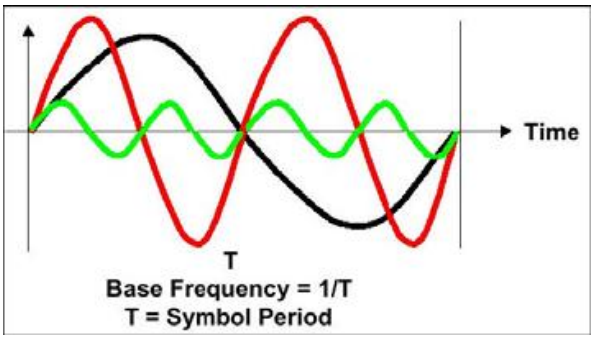

Figure 2-5**:** A set of orthogonal signals

To convert the sub-carriers to a set of orthogonal signals, the data is first combined into frames of a suitable size for an FFT or IFFT. A FFT should be always in the length of 2N (where N is an integer). Next, an N-point IFFT is performed and the data stream is the output of the transmitter. Thus when the signals of the IFFT output are transmitted sequentially, each of the N channel bits appears at a different sub-carrier frequency.

By using an IFFT process, the spacing of the sub carriers is chosen in such a way that at the frequency where the received signal is evaluated, all other signals is zero. In order for this orthogonality, the receiver and the transmitter must be perfectly synchronized. This means they both must assume exactly the same modulation frequency and the same time-scale for transmission. At the receiver, the exact inverse operations are performed to recover the data. Since the FFT is performed in this stage, the data is back in the frequency domain. It is then demodulated according to the block diagram below.

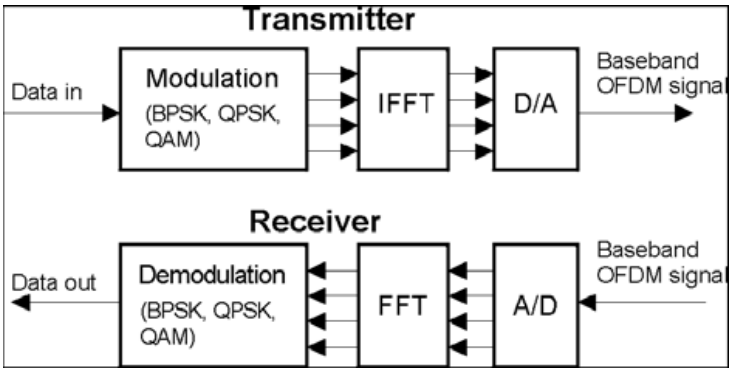

Figure 2-6**:** Block diagram for OFDM communications

2.3.4. Guard Period

One of the most important properties of OFDM transmission is its robustness against multi path delay. This is especially important if the signal's sub-carriers are to retain their orthogonality through the transmission process. The addition of the guard period between the transmitted symbols can be used to accomplish this. The guard period allows time for multipath signals from the previous symbol to dissipate before information from the current symbol is recorded.

The most effective guard period is a 'cyclic prefix' which is appended at the front of every OFDM symbol. The cyclic prefix is a copy of the last part of the OFDM symbol, and is of equal or greater length than the maximum delay spread of the channel (see Figure 2.5). Although the insertion of the cyclic prefix imposes a penalty on bandwidth efficiency, it is often the best compromise between performance and efficiency in the presence of inter-symbol interference.

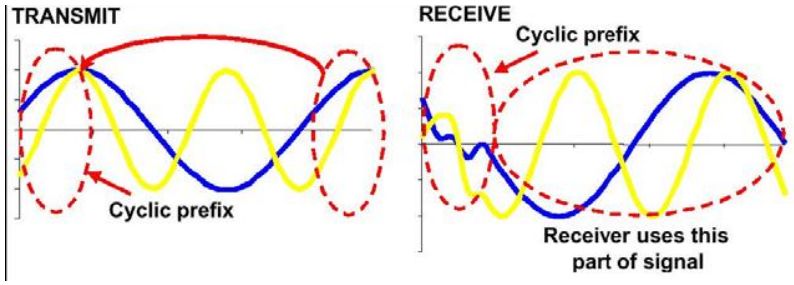

Figure 2-7: Implementation of cyclic prefix

2.4. Advantages of OFDM

OFDM has several advantages compared to other type of modulation technique implemented in wireless system. Below are some of the advantages that describe the uniqueness of OFDM compared to others:

2.4.1. Bandwidth Efficiency

A key aspect of all high speed communication system lies in its bandwidth efficiency. This is especially important for wireless communications where all current and future devices are expected to share an already crowded range of carrier frequencies. In OFDM, the frequency band containing the message is sub-divided into parallel bit streams of lower frequency carriers. These sub-carriers are designed to be orthogonal to one another, such that they can be separated out at the receiver without interference from neighboring carriers[7]. In this manner, OFDM is able to space the channels much closer together, which allows for more efficient use of the spectrum than through simple frequency division multiplexing.

The advantage of orthogonality in OFDM does not happen in FDMA where up to 50% of the total bandwidth is wasted due to the extra spacing between channels.

2.4.2. OFDM overcomes the effect of ISI

The limitation of sending in high bitrate is the effect of inter-symbol interference (ISI). As communication systems increase their information transfer speed, the time for each transmission becomes shorter. Since the delay time caused by multi-path remains constant, ISI becomes a limitation in sending high data rate communication. OFDM avoids this problem by sending many low speed transmissions simultaneously. For example figure 2.6 below shows two ways to transmit the same four pieces of binary data.

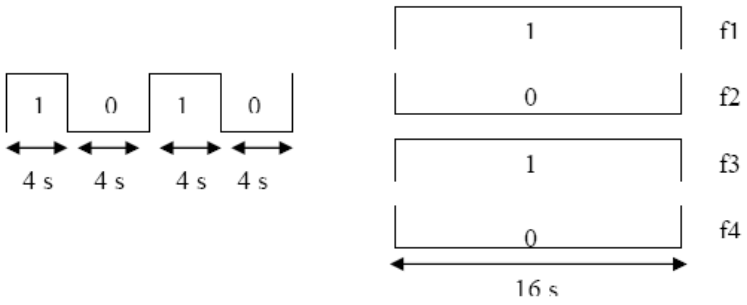

Figure 2-8: Two ways to transmit the same four pieces of binary data

Suppose that this transmission takes four seconds. Then, each piece of data in the left picture has duration of four second. When transmit these data, OFDM would send the

four pieces simultaneously as shown on the right. In this case, each piece of data has duration of 16 seconds. This longer duration leads to fewer problems with ISI.

2.4.3. OFDM combats the effect of frequency selective fading and burst error

OFDM is used to spread out a frequency selective fade over many symbols. This effectively randomizes burst errors caused by a deep fade or impulse interference, so that instead of several adjacent symbols being completely destroyed, many symbols are only slightly distorted. This allows successful reconstruction of a majority of them even without forward error correction (FEC). Because of this dividing, an entire channel bandwidth into many narrow sub-bands, the frequency response over each individual sub-band is relatively flat. Since each sub-channel covers only a small fraction of original bandwidth, equalization is potentially simpler than in a serial system.

2.5. The weakness of OFDM

Although OFDM is excellent in combating fading effect, it does not mean that OFDM is free from any weaknesses. Below are some of the weaknesses for the OFDM system.

2.5.1. Peak-to-Mean Power Ratio

OFDM signal has varying amplitude as shown by figure 2.7. It is very important that the amplitude variations be kept intact as they define the content of the signal. If the amplitude is clicked or modified, then an FFT of the signal would no longer result in the original frequency characteristics and the modulation may be lost.

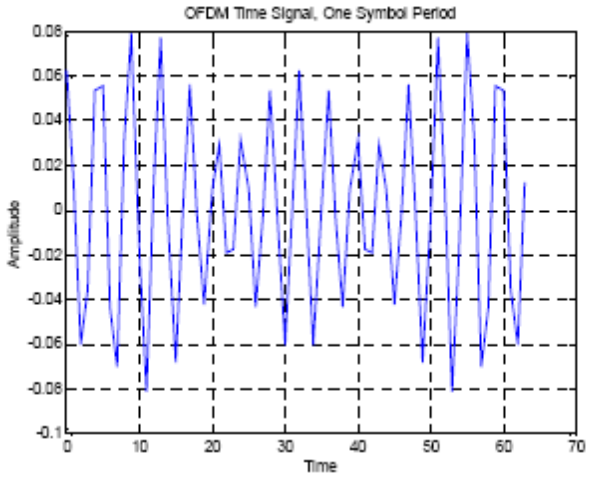

Figure 2-9: Show amplitude varying in OFDM

This is one of the drawbacks of OFDM, the fact that it requires linear amplification. In addition, very large amplitude peaks may occur depending on how the sinusoids line up, so the peak-to-average power ratio is high. This means that the linear amplifier has to have a large dynamic range to avoid distorting the peaks. The result is a linear amplifier with a constant, high bias current resulting in very poor power efficiency.

2.5.2. Synchronization

The limitation of OFDM in many applications is that it is very sensitive to frequency errors caused by frequency differences between the local oscillators in the transmitter and receiver. Carrier frequency offset causes a number of impairments including attenuation and rotation of each of the sub carriers and inter-carrier interference (ICI) between sub-carriers. In the mobile radio environment, the relative movement between transmitter and receiver causes Doppler frequency shifts[7].

In addition, the carriers can never be perfectly synchronized. These random frequency errors in OFDM system distort orthogonality between sub carriers and thus inter-carrier interference (ICI) occurs.

To optimize the performance of an OFDM link, time and frequency synchronization between the transmitter and receiver is of absolute importance. This is achieved by using known pilot tones embedded in the OFDM signal or attaching fine frequency timing tracking algorithms within the OFDM signal`s cyclic extension (guard interval).

2.6. Applications of OFDM

OFDM has been chosen for several current and future communications systems all over the world. It is well suited for systems in which the channel characteristics make it difficult to maintain adequate communications link performance. In addition to high-speed wireless applications, wired systems such as asynchronous digital subscriber line

(ADSL) and cable modem utilize OFDM as its underlying technology to provide a method of delivering high-speed data.

Recently, OFDM has also been adopted into several European wireless communications applications such as the digital audio broadcast (DAB) and terrestrial digital video broadcast (DVB-T) systems.

2.6.1. Digital Broadcasting

Standardized in 1995, Digital Audio Broadcasting (DAB) was the first standard to use OFDM. DAB uses a single frequency network, but the efficient handling of multi path delay spread results in improved CD quality sound, new data services, and higher spectrum efficiency. A broadcasting industry group also created Digital Video Broadcasting (DVB) in 1993.

DVB produced a set of specifications forthe delivery of digital television over cable, DSL and satellite. In 1997 the terrestrial network, Digital Terrestrial Television Broadcasting (DTTB), was standardized. DTTB utilizes OFDM in up to 2,000 and 8,000 sub-carrier modes[8].

2.6.2. Terrestrial Digital Video Broadcasting

A pan-broadcasting-industry group created Digital Video Broadcasting (DVB) in 1993. DVB produced a set of specifications for the delivery of digital television over cable, DSL and satellite. In 1997 the terrestrial network, Digital Terrestrial Television

Broadcasting (DTTB), was standardized. DTTB utilizes OFDM in the 2,000 and 8,000 sub carrier modes.

2.6.3. IEEE 802.11a/HiperLAN2 and MMAC Wireless LAN

OFDM in the new 5GHz band is comprised of 802.11a, HiperLAN2, and WLAN standards. In July 1998, IEEE selected OFDM as the basis for the new 802.11a 5GHz standard in the U.S. targeting a range of data rates up to 54 Mbps. In Europe, ETSI BRAN is now working on three extensions for OFDM in the HiperLAN standard: (i) HiperLAN2, a wireless indoor LAN with a QoS provision; (ii) HiperLink, a wireless indoor backbone; and (iii) HiperAccess, an outdoor, fixedwireless network providing access to a wired infrastructure. In Japan, consumer electronics companies and service providers are cooperating in the MMAC project to define new wireless standards similar to those of IEEE and ETSI BRAN.

2.6.4. Mobile Wireless Communication

OFDM's capability to work around interfering signals gives it potential to threaten existing CDMA (2.5G and 3G) wireless technology. This is what is allowing the technology to push forward in Europe. In densely populated areas where buildings, vehicles and people can scatter the path of a signal, broadcasters as well as high-speed data providers are anxious to eliminate multi-path effects.

According to industry analysts, telecom providers may also be lured to OFDM technology because it could end up causing only a fraction of what it costs to implement 3G wireless technologies.

2.7. FPGA

A Field-programmable Gate Array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare). FPGAs can be used to implement any logical function that an ASIC could perform. The FPGA Kit used for the project is given below.

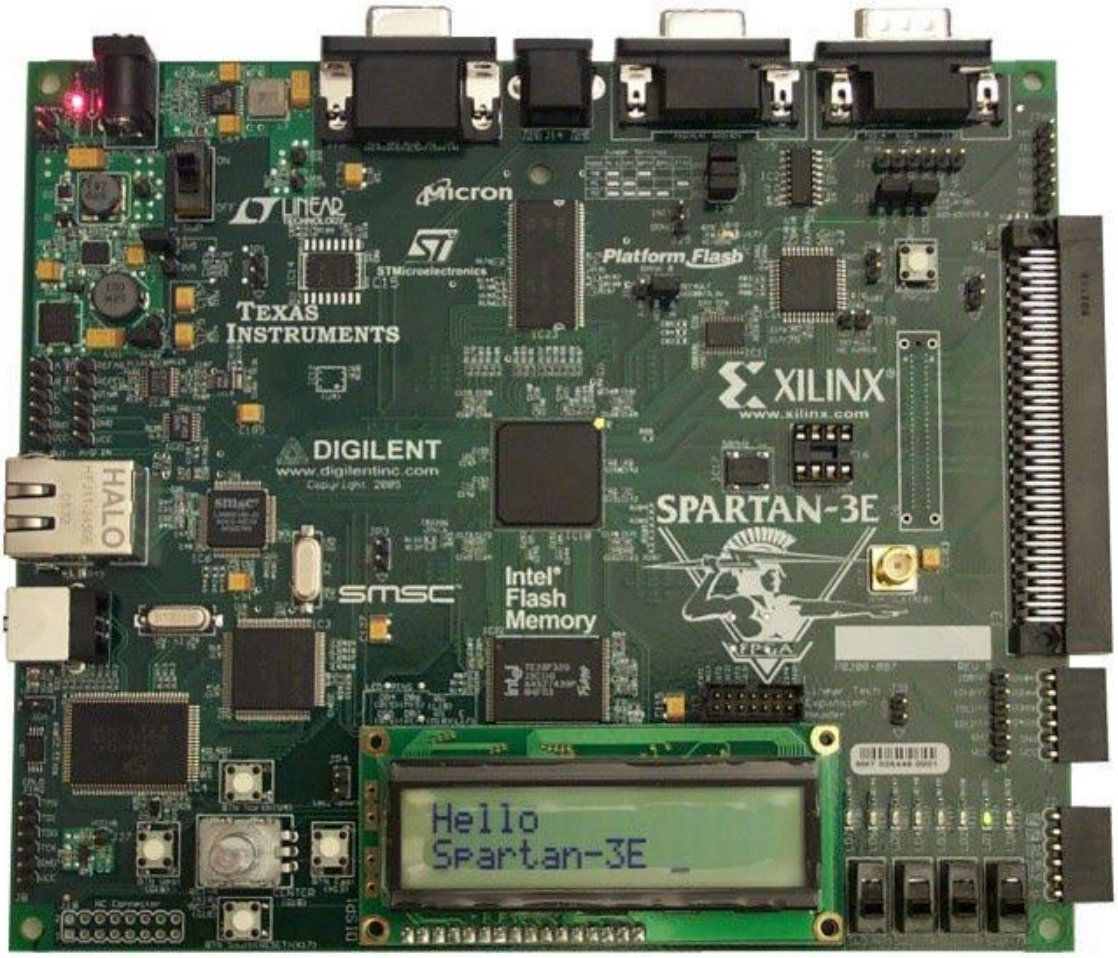

Figure 2-10 : Xilinx Spartan 3E Kit

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—somewhat like many (changeable) logic gates that can be inter-wired in (many) different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

2.7.1. History and Modern Developments

Xilinx Co-Founders, Ross Freeman and Bernard Vonderschmitt, invented the first commercially viable field programmable gate array in 1985 – the XC2064.The XC2064 had programmable gates and programmable interconnects between gates, the beginnings of a new technology and market. The XC2064 boasted a mere 64 configurable logic blocks (CLBs), with two 3-input lookup tables (LUTs). More than 20 years later, Freeman was entered into the National Inventors Hall of Fame for his invention.

FPGAs got a glimpse of fame in 1997, when Adrian Thompson, a researcher working at the University of Sussex, merged genetic algorithm technology and FPGAs to create a sound recognition device. Thomson's algorithm configured an array of 10 x 10 cells in a Xilinx FPGA chip to discriminate between two tones, utilizing analogue features of the digital chip.

Current advances in field-programmable gate array (FPGA) technology have enabled high-speed processing in a compact footprint, while retaining the flexibility and programmability of software radio technology. FPGAs are popular for high-speed, compute-intensive, reconfigurable applications (fast Fourier transform (FFT), finite impulse response (FIR) and other multiply-accumulate operations). Reconfigurable cores are available from FPGA and board vendors and enable implementation of modulator, demodulator and CODEC functionality in the FPGA. System designers are increasingly looking for front-end acquisition/converter products with integrated FPGA to offload the baseband processing and reduce data transfer rates.

The first kit was believed to have 9,000 gates and the kits being manufactured nowadays have more than a Million gates with the reduced size and energy consumption. The FPGA kits nowadays allow the user to process the data very easily and quickly through many input and output options along with the flexibility of simulating the code by using the company provided software along the kits. Xilinx provides the complete working environment in the form of Xilinx ISE Suite along with the FPGA kits which includes all the necessary tools to simulate and run the code on FPGAs.

In recent years semiconductor process technology has progressed to the point where FPGAs can be installed in both product prototypes and mass produced products, thanks to higher capacities and lower prices. In particular, we are now seeing many cases where FPGAs are adopted in products that are produced at low volumes and products where there is a need for long-term supply, and the demand to install a full-scale graphics sub-system on these FPGAs is growing every year.

2.7.2. Architecture

The most common FPGA architecture consists of an array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.

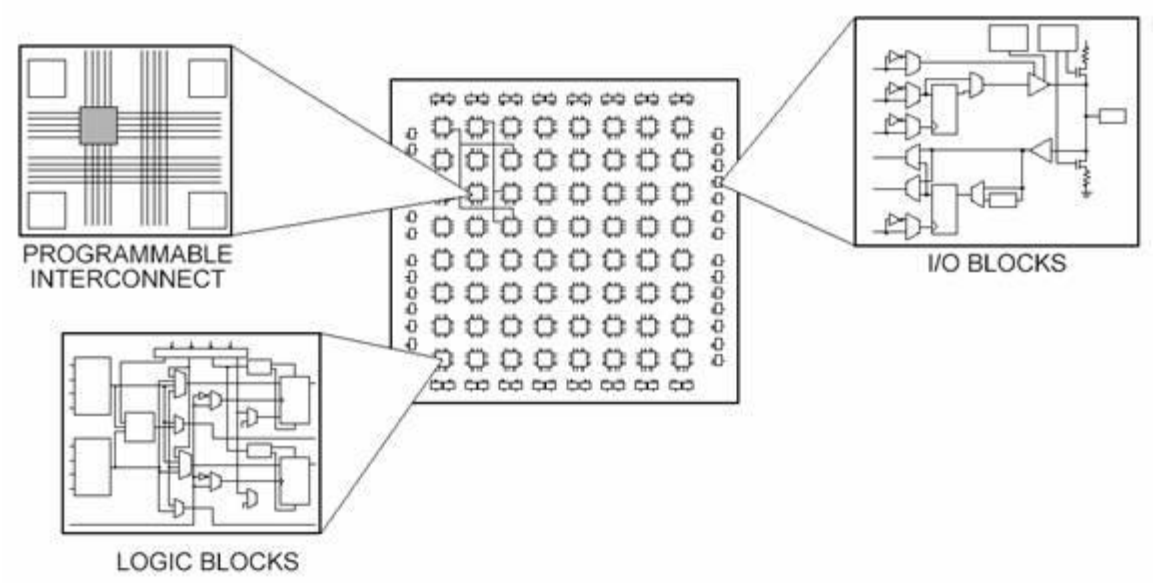

Figure 2-11: FPGA Chip Blocks

An application circuit must be mapped into an FPGA with adequate resources. While the number of CLBs/LABs and I/Os required is easily determined from the design, the number of routing tracks needed may vary considerably even among designs with the same amount of logic. For example, a crossbar switch requires much more routing than a systolic array with the same gate count. Since unused routing tracks increase the cost (and decrease the performance) of the part without providing any benefit, FPGA manufacturers try to provide just enough tracks so that most designs that will fit in terms of LUTs and IOs can be routed. This is determined by estimates such as those derived from Rent's rule or by experiments with existing designs.

In general, a logic block (CLB or LAB) consists of a few logical cells (called ALM, LE, Slice etc). A typical cell consists of a 4-input Lookup table (LUT), a Full adder (FA) and a D-type flip-flop, as shown below. The LUTs are in this figure split into two 3-input

LUTs. In normal mode those are combined into a 4-input LUT through the left mux. In arithmetic mode, their outputs are fed to the FA. The selection of mode is programmed into the middle mux. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space.

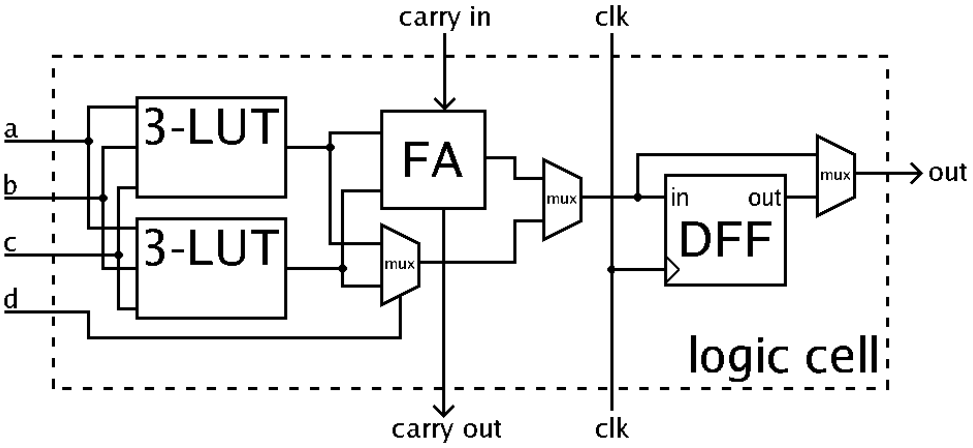

Figure 2-12: Simplified example illustration of a logic cell

ALMs and Slices usually contain 2 or 4 structures similar to the example figure, with some shared signals. CLBs/LABs typically contain a few ALMs/LEs/Slices.

In recent years, manufacturers have started moving to 6-input LUTs in their high performance parts, claiming increased performance.

Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed. For this example architecture, the locations of the FPGA logic block pins are shown below.

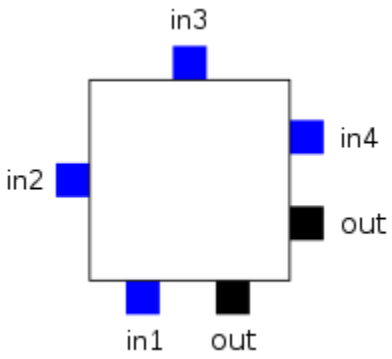

Figure 2-13: Logic Block Pin Locations

Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block.

Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

Similarly, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic-block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

Whenever a vertical and a horizontal channel intersect, there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that

allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure below illustrates the connections in a switch box.

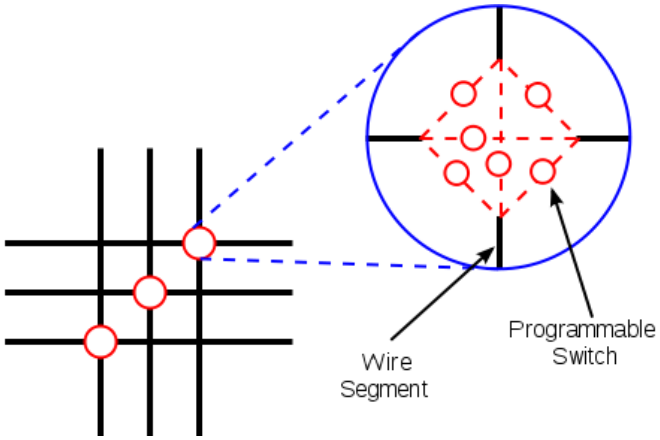

Figure 2-14: Switch box topology

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed IO logic and embedded memories.

FPGAs are also widely used for systems validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, reducing the time-to-market.

To shrink the size and power consumption of FPGAs, vendors such as Tabula and Xilinx have introduced new 3D or stacked architectures.

Following the introduction of its 28nm 7-series FPGAs, Xilinx revealed that that several of the highest-density parts in those FPGA product lines will be constructed using multiple dice in one package, employing technology developed for 3D construction and stacked-die assemblies. The technology stacks several (three or four) active FPGA dice side-by-side on a silicon interposer – a single piece of silicon that carries passive interconnect.

2.7.3. FPGA OR DSP?

FPGAs have evolved from being flexible logic design platforms to signal processing engines. They are now an essential component of software radio due to their flexibility and real-time processing capabilities. Increasingly, system designers are porting more and more signal processing functionalities in FPGAs. The flexibility of having the ability to integrate logic design with signal processing is pushing designers to replace traditional digital signal processors (DSPs) with FPGAs.

FPGAs are inherently suited for high-speed parallel multiply and accumulate functions. Current generation FPGAs can perform 18 × 18 multiplication operation at speeds in excess of 200 MHz. This makes FPGAs an ideal platform for operations such as FFT, FIR, digital down-converters (DDC), digital up-converters (DUC), correlators and pulse compression (for radar processing).

It does not imply, however, that all DSP functionalities may be implemented in FPGAs. Floating point operations are difficult to implement in FPGAs due to the large amount of real estate needed in the device. Also, processing involving matrix inversion (or division) is also more suited to a DSP/GPP platform. FPGAs and DSP will thus coexist for a long time, and a flexible platform will include a mix of both.

2.7.4. Applications of FPGA

Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.

FPGAs especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture. One such area is code breaking, in particular brute-force attack, of cryptographic algorithms.

FPGAs are increasingly used in conventional high performance computing applications where computational kernels such as FFT or Convolution are performed on the FPGA instead of a microprocessor.

The inherent parallelism of the logic resources on an FPGA allows for considerable computational throughput even at a low MHz clock rates. The flexibility of the FPGA allows for even higher performance by trading off precision and range in the number format for an increased number of parallel arithmetic units. This has driven a new type of processing called reconfigurable computing, where time intensive tasks are offloaded from software to FPGAs.

The adoption of FPGAs in high performance computing is currently limited by the complexity of FPGA design compared to conventional software and the turn-around times of current design tools.

Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

2.7.5. Applications in Digital Signal Processing

Digital signal processing has traditionally been done using enhanced microprocessors. While the high volume of generic product provides a low cost solution, the performance falls seriously short for many applications. Until recently, the only alternatives were to develop custom hardware (typically board level or ASIC designs), buy expensive fixed function processors (e.g. an FFT chip), or use an array of microprocessors.

Recent increases in Field Programmable Gate Array performance and size offer a new hardware acceleration opportunity. FPGAs are an array of programmable logic cells interconnected by a matrix of wires and programmable switches. Each cell performs a simple logic function defined by a user's program. An FPGA has a large number (64 to over 20,000) of these cells available to use as building blocks in complex digital circuits. Custom hardware has never been so easy to develop.

The ability to manipulate the logic at the gate level means you can construct a custom processor to efficiently implement the desired function. By simultaneously performing all of the algorithm’s sub-functions, the FPGA can outperform a DSP by as much as 1000:1.

2.7.6. Applications in Software Defined Radios

SDR baseband processing often requires both processors and FPGAs. In such applications, the processor handles system control and configuration functions, while the FPGA implements the computationally intensive signal-processing data path and control,

minimizing the latency in the system. When it is necessary to switch from one standard to another, the processor can switch dynamically between major sections of software, while the FPGA can be completely reconfigured, as necessary, to implement the data path for the particular standard.

FPGAs can be used as co-processors to interface with DSPs and general-purpose processors, thereby providing higher system performance and lower system costs. Having the freedom to choose where to implement baseband-processing algorithms adds another dimension to the flexibility when implementing SDR algorithms.

Chap 3

Development and Design

3.1. Overview

In this chapter we will discuss the methodology and approach leading to the development of this project. We will start the chapter by looking on the different modules of OFDM implemented in our Communication system necessary for quality transmission and how they all sum up to form a fast and reliable digital communication system. Later on, we will discuss the approaches to implement the communication and the reason why we chose the FPGA for the implementation of OFDM. We will sum up this chapter by discussing the tools and softwares used to develop the prototype.

3.2. The OFDM Model

OFDM is implemented according to the various standards and techniques required in the communication system. The OFDM model we used in our project was according to the IEEE 802.11 standard. The modulation technique used is QPSK. 16-point FFT and IFFT are employed and the convolutional encoding along with the scrambling precedes the FFT block. Guard insertion and Guard Removal also are a part of the prototype developed. The block diagram of the developed system is given below:

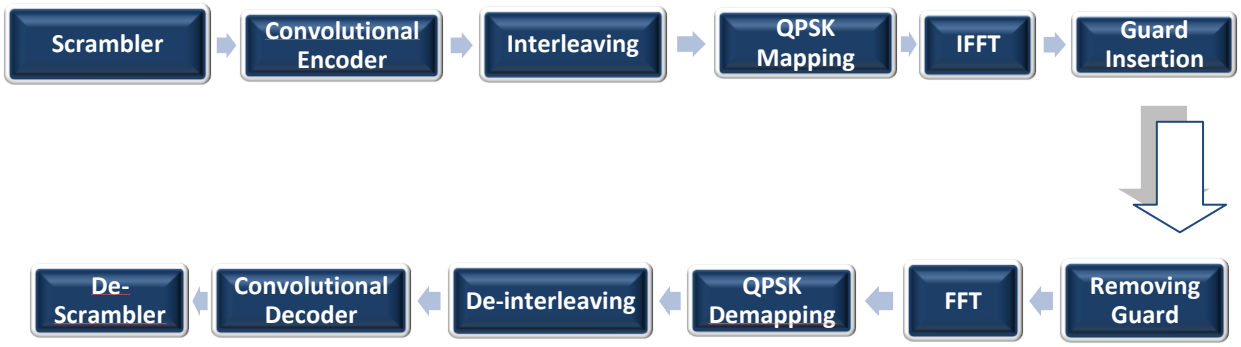

Figure 3-1: Block Diagram of the OFDM System

Now we will discuss the proposed OFDM communication system in detail.

3.2.1. Scrambler and De-Scrambler

A scrambler is a device that transposes or inverts signals or otherwise encodes a message at the transmitter to make the message unintelligible at a receiver not equipped with an appropriately set descrambling device. Also referred to as a randomizer, it is a device that manipulates a data stream before transmitting. The manipulations are reversed by a descrambler at the receiving side.

Scrambler is basically built from modulo-2 and shift operations at appropriate clock cycles. At first the temporary bit is calculated at negative edge of the cycle and then pushed to the register along with the shift to right at the positive edge of the clock. Scrambler structure is basically similar to that of Linear Feedback Shift Register (LFSR). In our design, we have used a frame synchronous scrambler which takes data in octets and are placed in the serial bitstream. The scrambler used employs the generator

polynomial $S(x) = x^7 + x^4+1$. The same scrambler is used to scramble transmit data and to de-scramble received data. While transmitting, the initial state of the scrambler will be set over random non-zero state. The LFSR scrambler used is shown below in the figure.

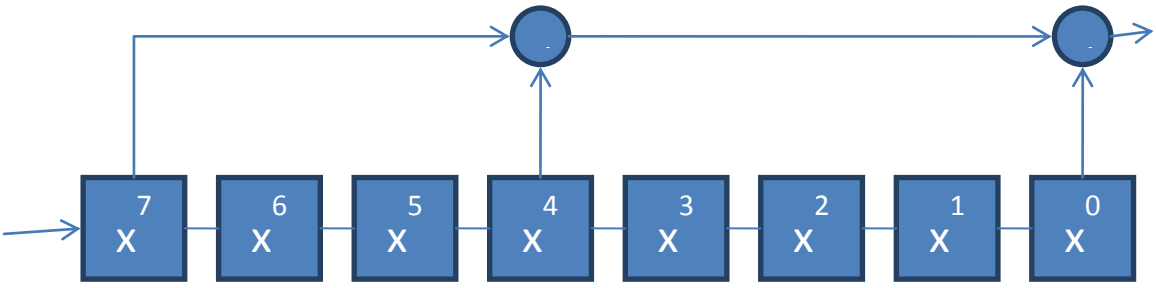

Figure 3-2: Scrambler of the standard generator polynomial

There are two main reasons why scrambling is used:

- It facilitates the work of a timing recovery circuit, an automatic gain control and other adaptive circuits of the receiver (eliminating long sequences consisting of '0' or '1' only).
- It eliminates the dependence of a signal's power spectrum upon the actual transmitted data, making it more dispersed to meet maximum power spectral density requirements (because if the power is concentrated in a narrow frequency band, it can interfere with adjacent channels due to the cross modulation and the inter-modulation caused by non-linearities of the receiving tract).

The descrambler works in the same way and brings back the original sequence that was scrambled.

3.2.2. Convolutional Encoder and Decoder

Convolutional Encoder is implemented in the OFDM system for channel coding, which is an essential part of any good digital communication system. Its purpose here is to provide forward error correction. It helps provide redundant bits on channel to incorporate channel encoding.

In this design, the encoder uses the generator polynomials $g_0=x^5+x^4+x^2+x$ and $g_1=x^5+x^2+x+1$ of rate (R) = 1/2. The data rates are improved by applying puncturing. It is the procedure of omitting some of the encoded bits in the transmitter. On the receiving side, the decoder inserts the dummy zeros in place of the omitted bits. Convolutional encoder's local controller controls the working of the encoder with puncturing module in conjunction.

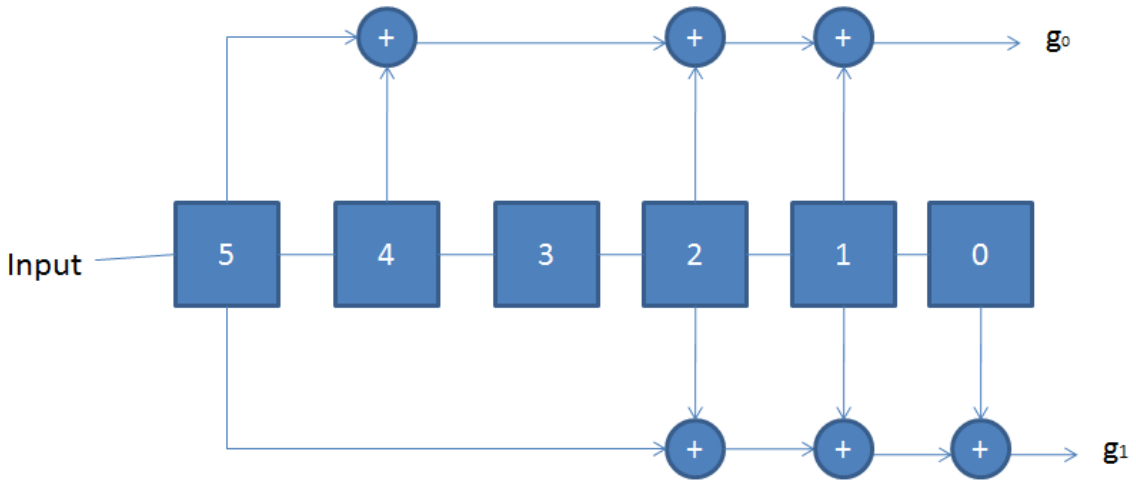

Figure 3-3 : Convolutional Encoder with the given Generator Polynomial

The convolutional encoding rate used in the project is according to the IEEE Standard 802.11a which specifies the rate of 1/2 by basic convolutional encoder and 2/3 and 3/4 with punctured codes. If puncture enable is set high, controller performs basic coding and sends the data to puncturing module on the next clock cycle and punctures the incoming

code according to the rate flag. When the rate flag is set high, 3/4 convolutional encoding is performed, else it is set for rate 2/3.

3.2.3. Interleaving and De-Interleaving

Interleaving is a way to arrange data in a non-contiguous way to increase performance. Many communication channels are not memory-less: errors typically occur in bursts rather than independently. If the number of errors within a code word exceeds the error-correcting code's capability, it fails to recover the original code word. Interleaving ameliorates this problem by shuffling source symbols across several code words, thereby creating a more uniform distribution of errors.

Symbol interleaver / de-interleaver can mitigate the effects of burst noise. Typically, these functions are needed for transport channels that require a bit error ratio (BER) in the order of 10^{-6}. The encoded data bits are interleaved by the interleaver block with a block size corresponding to number of bits in OFDM symbol. The interleaver is defined by a two-step permutation.

The interleaver is defined by a two-step permutation. The first permutation ensures that adjacent coded bits are mapped onto nonadjacent subcarriers. The second permutation ensures that adjacent coded bits are mapped alternately onto less and more significant bits of the constellation, thereby avoiding long runs of low reliability bits.
k will be used to denote the index of the coded bit before the first permutation; i will denote the index after the first and before the second permutation; j will denote the index after the second permutation, just prior to modulation mapping.
The first permutation is defined by

$i = (N_{CBPS} /16) (k \bmod 16) + \text{floor}(k/16)$ $k = 0, 1, ..., N_{CBPS} - 1$

The function floor (.) denotes the largest integer not exceeding the parameter. The second permutation is defined by

$j = s \times \text{floor}(i/s) + (i + N_{CBPS} - \text{floor}(16 \times i/ N_{CBPS})) \bmod s$ $i = 0, 1, ... N_{CBPS} - 1$

where*s* is determined by the number of coded bits per subcarrier N_{DBPS}

according to

$s = \max (N_{DBPS} /2, 1)$

3.2.4. Modulation Mapping and De-mapping

The encoded and interleaved binary serial input data shall be divided into groups of 2 bits each and converted into complex numbers representing QPSK constellation points. The modulation mapper takes two bits as input and after mapping, outputs a 28-bit stream with most significant 14 bits representing real, and the rest 14 representing imaginary part of the signal. Unsigned fractional bits for each of the 2-bit input were first calculated using MATLAB and then the code was developed accordingly.

3.2.5. FFT / IFFT

In our design, pipelined FFT module has been designed and radix-2 FFT has been implemented. We have implemented 16-point FFT; it consists of 8 butterfly structures and 4 total stages. The FFT block takes 28 bit N complex data points as serial input where N represents the number of points.

In the design, serially pipelined FFT has been employed. Parallel pipelined FFT processors are employed to meet the growing demand of high processing rate. Highly parallel implementations obtain high computation rates but require the simultaneous distribution of all data samples. The high rate of distribution of data required to keep the processors busy is impossible to achieve especially in real time applications involving word-serial data. This problem coupled with limited input/output resources in FPGAs makes the parallel algorithm inefficient.

The serially pipelined FFT computes one transform in O(N) processing cycles, producing the output sequentially at the input data rate. So the cascaded FFT is ideally suited for real-time signal processing. Cascaded FFT uses registers organized as shift-registers between butterfly computation units.

FFT is the most important process in implementation of OFDM. The FFT operates by decomposing an *N* point time domain signal into *N* time domain signals each composed of a single point. The second step is to calculate the *N* frequency spectra corresponding to these *N* time domain signals. Lastly, the *N* spectra are synthesized into a single frequency spectrum. This way FFT is basically the process that divides the input signal into N number of orthogonal frequencies, which is the basic and unique property of OFDM technique.

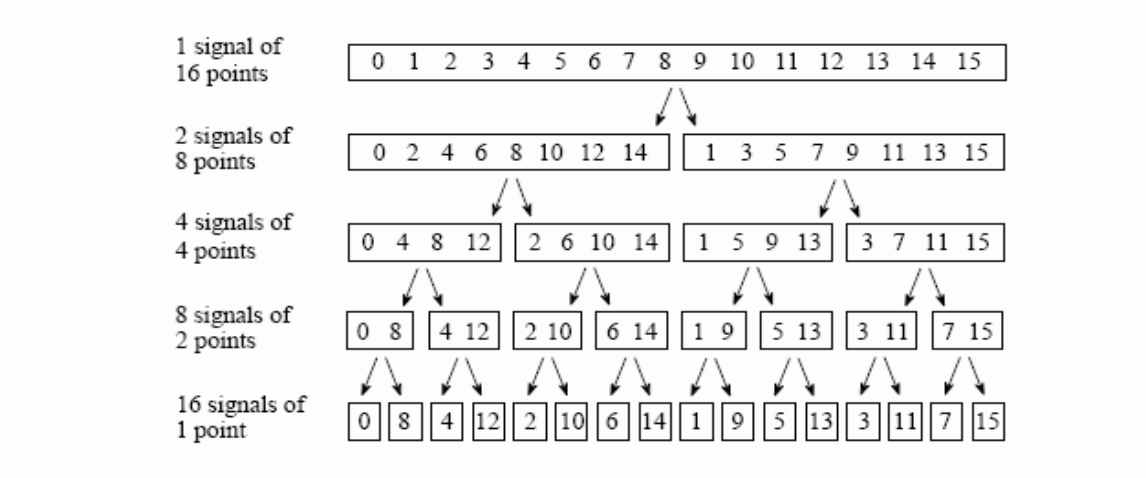

Figure 3-4: FFT 16-point signal broken up into 16 different signals

In our system, 16 point signal is decomposed through four separate stages. The first stage breaks the 16 point signal into two signals each consisting of 8 points. The second stage decomposes the data into four signals of 4 points. This pattern continues until there are *N* signals composed of a single point. An interlaced decomposition is used each time a signal is broken in two, that is, the signal is separated into its even and odd numbered samples. There are Log_2N stages required in this decomposition, i.e., a 16 point signal (2^4) requires 4 stages.

The structure is replicated at each clock cycle for N/2 butterfly operations and is executed in parallel for all stages to get serially pipelined data output of complex 16-points. The source controller initializes stage and butterfly controller according to N=16 and forwards the control to the data address generator. Address generator will produce addresses considering radix-2 structure, for the 2 complex points to be sent to Data Memory.

These complex points are then forwarded to the radix block where two complex values for the next stage are produced. In the radix block, the second complex point is multiplied

by the twiddle factor generated by the twiddle address generator and then increments the butterfly number for that stage. This cycle continues till N/2 butterflies. In pipelined implementation, this process is carried out for $\log_2 N$ i.e., 4 cycles and transformed data is presented on the output pins after 16 cycles.

3.2.6. Guard Insertion and Removal

Guard insertion is required at the transmitter end to avoid inter-symbol interference. Shifting the time TGUARD creates the "circular Prefix" used in OFDM to avoid ISI from any previous frame.

The reasons to use a cyclic prefix for the guard interval are:

- To maintain the receiver carrier synchronization ; some signals instead of a long silence must always be transmitted;
- Cyclic convolution can still be applied between the OFDM signal and the channel response to model the transmission system.

The figure below shows the effect of adding a guard interval after every OFDM symbol. We can see that the tolerance on timing the samples is considerably more relaxed.

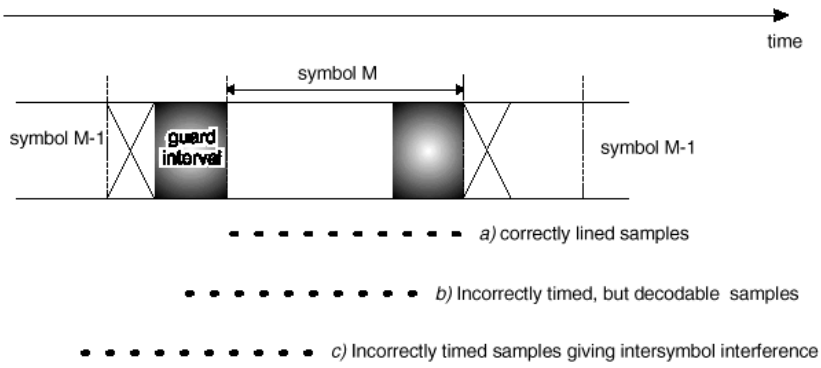

Figure 3-5: The effect on Timing Tolerance of adding a Guard Interval

Each symbol now is made up of two parts. The whole signal is now contained in the active symbol (highlighted for the symbol M in the figure below) the last part of which is also repeated at the start of the symbol and is called the guard interval.

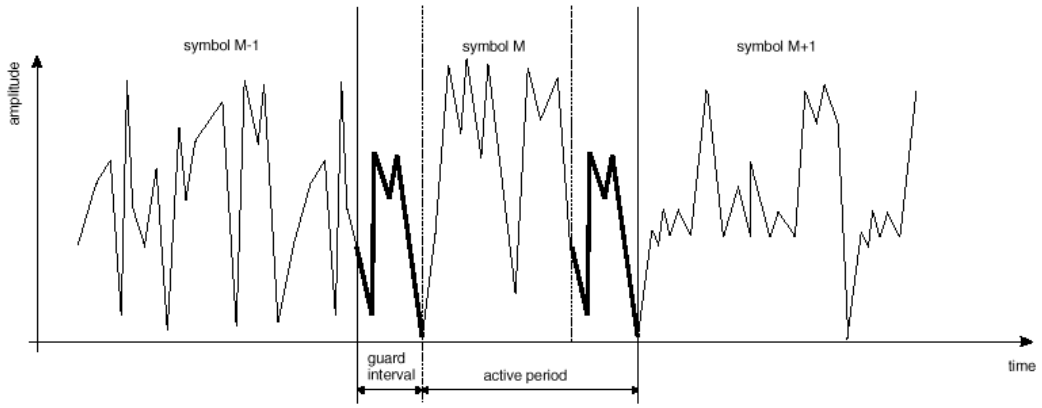

Figure 3-6: Example of the guard interval

3.3. Implementation on FPGA:

After the OFDM structure is created on software, it has to be implemented on FPGA for real-time transmission. The architecture would be a simplex system where one FPGA kit would act as transmitter and the other would act as the receiver. All the processing would be done on FPGA and the data would be transmitted between the two kits serially over the UART port.

3.3.1. Integration of Transmitter and Receiver:

Initially all the modules of OFDM were individually coded in Verilog and then the modules for transmitter were integrated to create a stand-alone system. The transmitter end takes voice (analog) input, performs the formerly mentioned functions and then sends the digital data to the receiver over the wired connection where the original signal is recovered.

3.3.2. Burning of Code on FPGA:

Before implementing the code on hardware, it is first synthesized on the software Xilinx ISE v11.1. Synthesis results give a summary of all the processing and memory resources that would be required to successfully implement the code on FPGA. This result also verifies if there are any processes in the code that cannot be implemented on hardware in form of gates and latches. After successful synthesis, I/O pins have to be configured. There are various input, output and I/O pins available on FPGA kit. In addition, an FX2 interface device is also available with Spartan 3e with an additional 40 differential I/O

pins that can also be used. UCF file is created for the code on which every input and output is assigned the location of a pin on the FPGA.

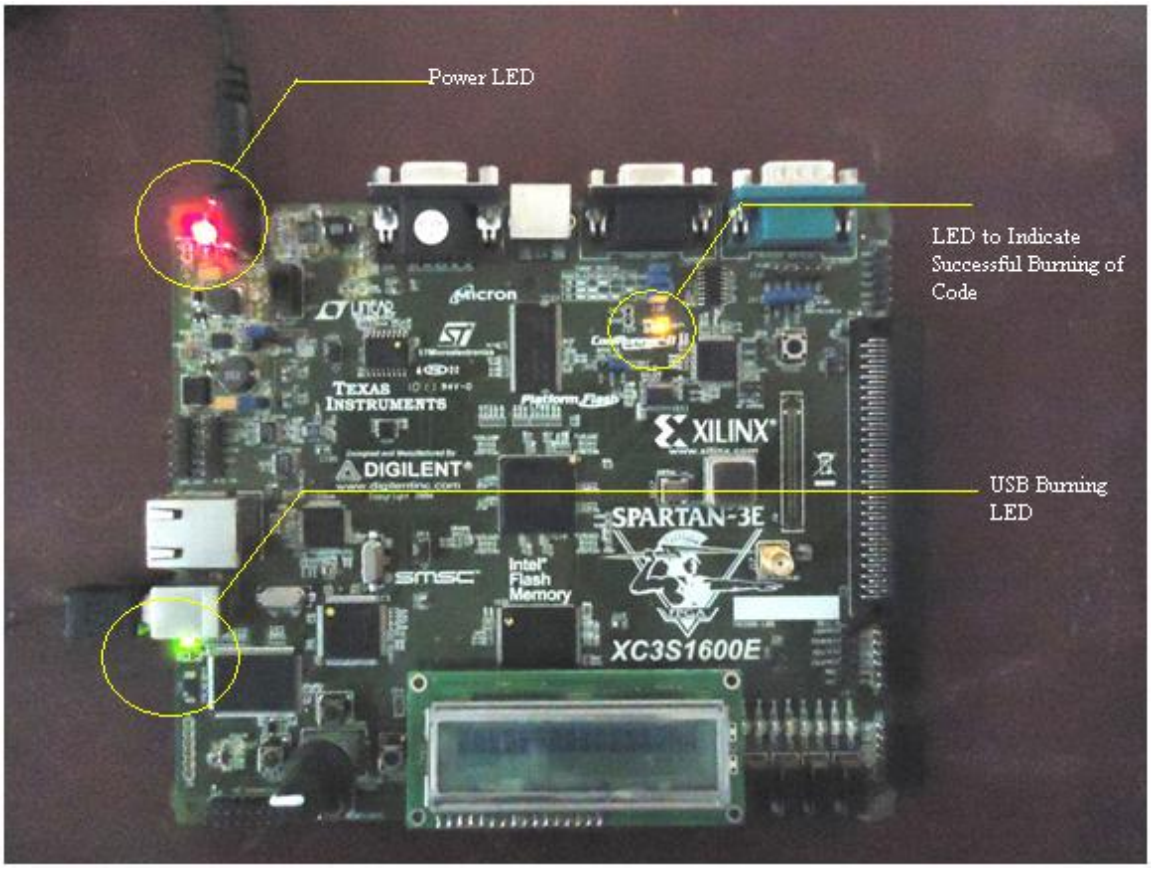

Figure 3-7 : FPGA Code Burning Process

When the code is burnt on the FPGA, then all the modules will get their inputs correspondingly from the physical pins that had been assigned in the UCF file. The software used for burning the code on FPGA Cores in the Xilinx Environment is '*iMPACT*' application of the Xilinx ISE v11.1.

3.3.3. Audio input and DAC / A/D:

On the transmitter end, voice analog signal is fed to the ADC circuit on the FPGA kit. This circuit consists of a pre-amplifier and an analog-to-digital converter IC. This circuit interacts with the FPGA over a Serial Peripheral Interface (SPI) where FPGA acts as the master and ADC acts as slave. After converting the analog signal to digital format, it is sent to the corresponding input port from where it enters the transmitter. After the insertion of guard interval, the digital data packet leaves the transmitter and is sent to the receiver over the wired channel through the serial UART port.

Initially all the modules of OFDM were individually coded in Verilog and then the modules for transmitter were integrated to create a stand-alone system. The transmitter end takes voice (analog) input, performs the formerly mentioned functions and then sends the digital data to the receiver over the wired connection where the original signal is recovered.

Before implementing the code on hardware, it is first synthesized on the software Xilinx ISE 11. Synthesis results give a summary of all the processing and memory resources that would be required to successfully implement the code on FPGA. This result also verifies if there are any processes in the code that cannot be implemented on hardware in form of gates and latches. After successful synthesis, I/O pins have to be configured. There are various input, output and I/O pins available on FPGA kit. In addition, an FX2 interface device is also available with Spartan 3e with an additional 40 differential I/O pins that can also be used. UCF file is created for the code on which every input and output is assigned the location of a pin on the FPGA. When the code is burnt on the FPGA, then all

the modules will get their inputs correspondingly from the physical pins that had been assigned in the UCF file.

The Xilinx Spartan 3E has various serial and parallel input ports but it does not contain and audio input/output jack. For this reason we have developed an external circuit where we have used an external audio jack which converts audio signals into voltage and sends this analog signal to the ADC of the transmitter.

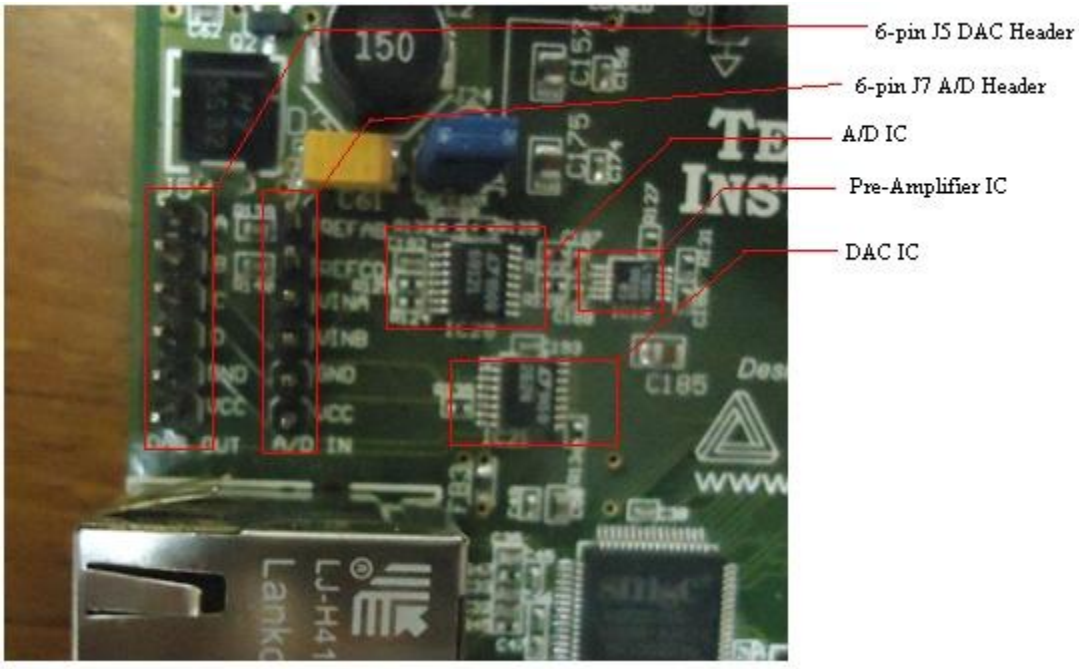

Figure 3-8 : DAC and A/D Ports in the FPGA Kit

On the transmitter end, voice analog signal is fed to the ADC circuit on the FPGA kit. This circuit consists of a pre-amplifier and an analog-to-digital converter IC. This circuit interacts with the FPGA over a Serial Peripheral Interface (SPI) where FPGA acts as the master and ADC acts as slave. After converting the analog signal to digital format, it is sent to the corresponding input port from where it enters the transmitter. After the

insertion of guard interval, the digital data packet leaves the transmitter and is sent to the receiver over the wired channel through the serial UART port. DAC works on the receiver end. The received data is processed according to the OFDM standard and the output bits from the descrambler enter the DAC, 12 bits per frame, these bits are then mapped to by the DAC to reconstruct the analog signal.

In this chapter we discussed the implementation of the proposed OFDM system on the FPGA kit. The designs of the blocks of OFDM were explained according to the need and the standard followed i.e. IEEE 802.11a after estimating the memory resources available on the FPGA kits available. The basic techniques and knowledge of Verilog coding was used to implement the OFDM system. We also established a process for synthesizing and burning the Verilog code of the given system using the ISE v11.1 .

Chap 4

Analysis and Evaluation

4.1. Overview

In this chapter we will demonstrate the results of the project. We will start over with the simulation results in Simulink and then in Xilinx environment through System Generator Blockset. We later on implemented the system on FPGA in Real-Time environment through coding in Verilog and creating the modules in Xilinx ISE v11.1. All the results of the Verilog code were verified using iSim application of the software. The implementation was finally analyzed through voice transmission through the use of DAC and A/D. the separate results for A/D and DAC were compared with that of the original voice.

4.2. Simulink Results

The OFDM code was implemented in simulink before going to the Xilinx and FPGA environment to validate the design of the communication system. The system model is shown below in the figure

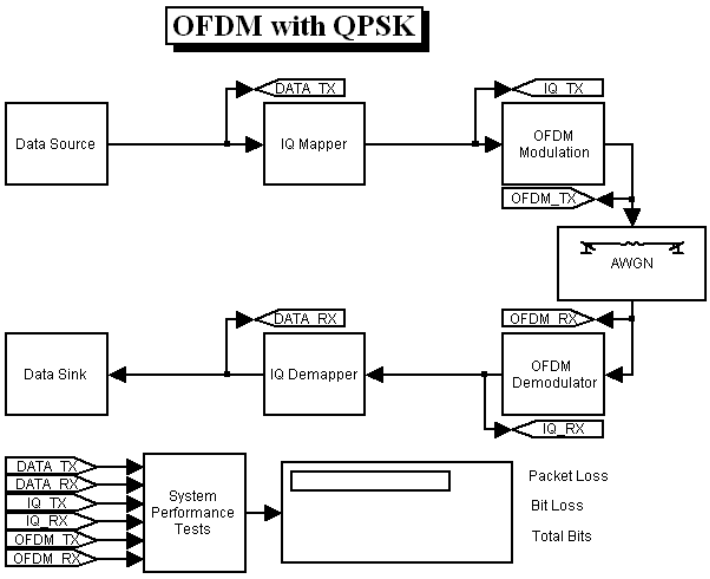

Figure 4-1 : Simulink OFDM Model

The Bernoulli Binary Generator was used as the data source followed by the simulink block for QPSK mapping. The iFFT and cyclic prefix addition were performed in the OFDM modulation block and then sent on the AWGN model. Then the FFT block demodulated the OFDM signal and then the QPSK demapper and then the final data sink where the signal was analyzed. The result of the simulation is shown below in the figures.

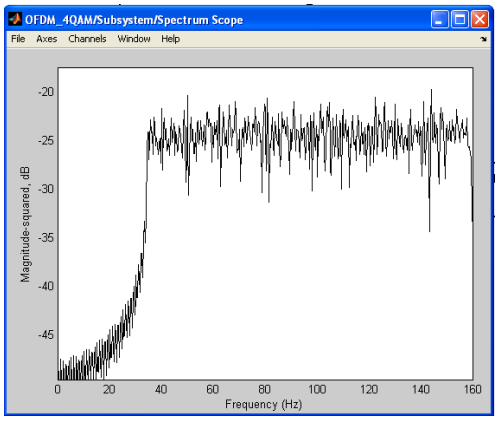

Figure 4-2 : Magnitude of the FFT Signal along the frequency

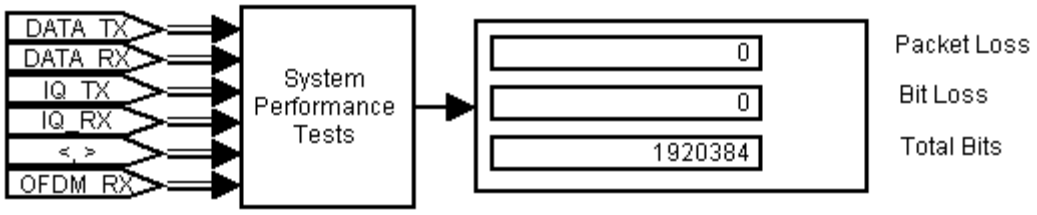

Figure 4-3 : Simulation Performance

The figure shows that at the end of simulation 1920384 bits were sent from the Bernoulli Generator and all of them were correctly recovered at the Data Sink. If the AWGN channel modeling is done more aggressively then the bit loss starts to increase slightly.

4.3. ISE Results

The OFDM system was implemented in verilog and tested in the ISE software before transferring to the FPGA for real-time transmission. The analysis in the ise software enabled us to verify the verilog code syntax and its synthesis. The ise gave warnings and errors about any mistakes in the written Verilog code which made them detectable and easily removable. The test-bench creation allowed for the analysis of the signal processing in the FPGA kit according to the clock of the kit. The results calculated theoretically were compared with the results shown for each module in the timing diagrams. The memory resources used and the output signals for each module is explained in the figures given below. We will explain each module with the repesctive figures and the timing diagrams obtained in the iSim Application of the software.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	13	14752	0%
Number of Slice Flip Flops	20	29504	0%
Number of 4 input LUTs	18	29504	0%
Number of bonded IOBs	9	250	3%
Number of GCLKs	1	24	4%

Figure 4-4 : Scrambler Synthesis Report in ISE

The synthesis report includes the calculation of the expected use of the memory of FPGA kit by the module synthesized. It tells about the number of slices used along with all the inputs and outputs required by the module and how much we actually have available in the FPGA kit.

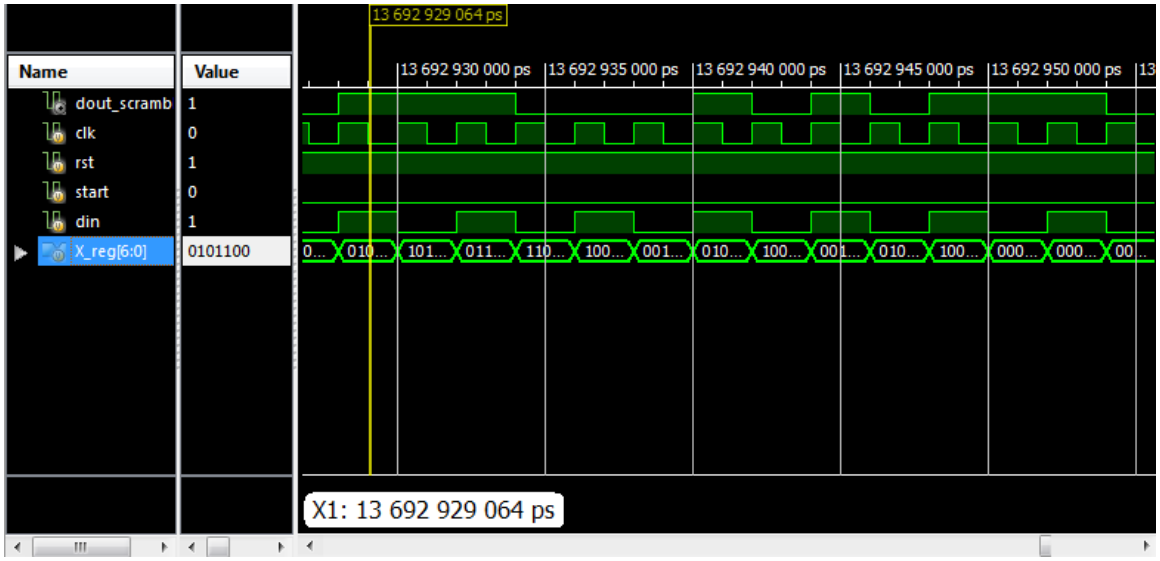

Figure 4-5 : Output of Scrambler

Output of the scrambler shows the timing diagram of the signals runnig in the scrambler module. The first signal is the output of the scrambler that is passed on to the encoder module. Then the clock and reset signals are shown along with the start and input signals.

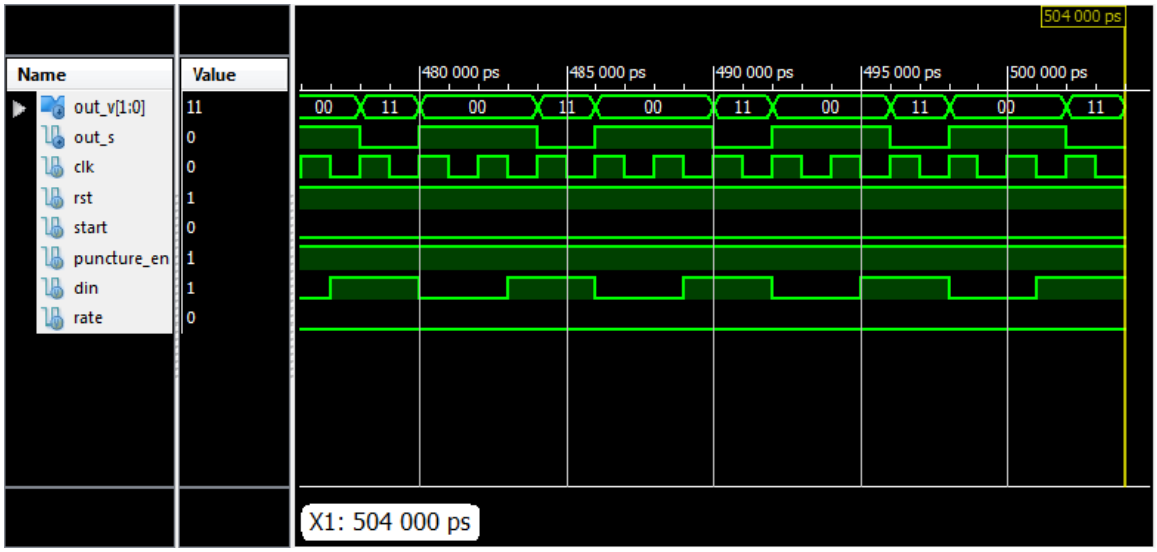

Figure 4-6 : Output of Encoder

Output of the encoder shows the output bits when compared to the initial bits given to the encoder module. Puncture enabling signal can also be seen in the figure.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	476	14752	3%
Number of Slice Flip Flops	150	29504	0%
Number of 4 input LUTs	1039	29504	3%
Number of bonded IOBs	16	250	6%
Number of GCLKs	1	24	4%

Figure 4-7 : Interleaver Synthesis Report in ISE

Interleaver module synthesis report indicated that only 3% of the slices available in the Spartan 3E-1600 kit are being used. Input/output blocks have a usage of 6% of the total available.

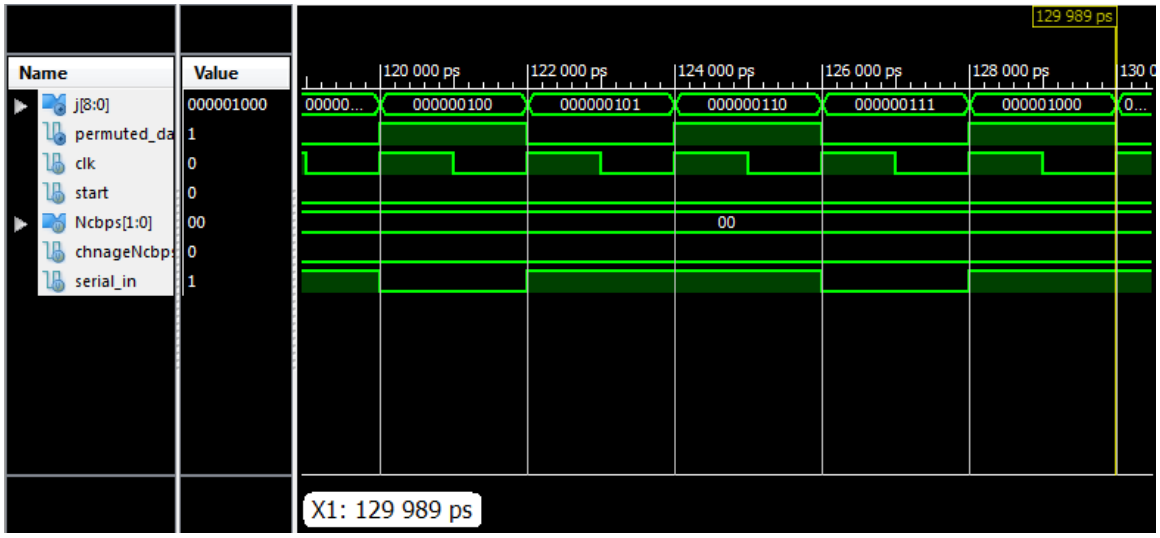

Figure 4-8 : Output of Interleaver

Interleaver output is shown to us which is in accordance with the double permutation algorithm we implemented for the interleaving purposes. Permuted data signal and the Ncbps Signal can also be seen.

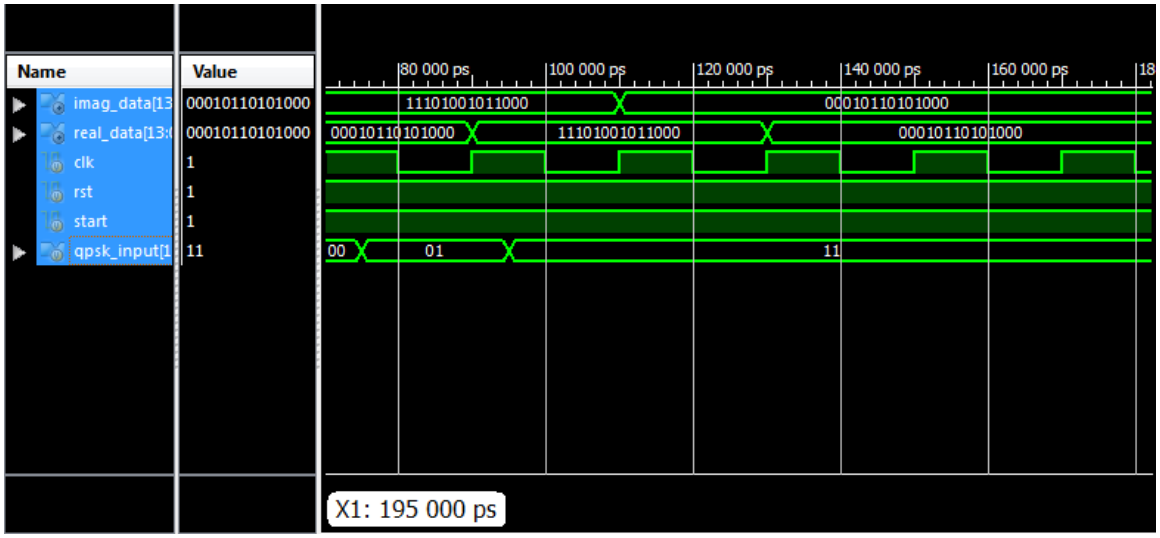

Figure 4-9 : Output of QPSK Mapper

QPSK mapping results in the imaginary and real data comparing to the given input data.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	23	14752	0%
Number of Slice Flip Flops	30	29504	0%
Number of 4 input LUTs	24	29504	0%
Number of bonded IOBs	33	250	13%
Number of GCLKs	1	24	4%

Figure 4-10 : QPSK Mapper Synthesis Report in ISE

QPSK mapper synthesis tells us that it needs only 23 slices which is negligible to the amount of slices available. IOBs needed are 13% of the total.

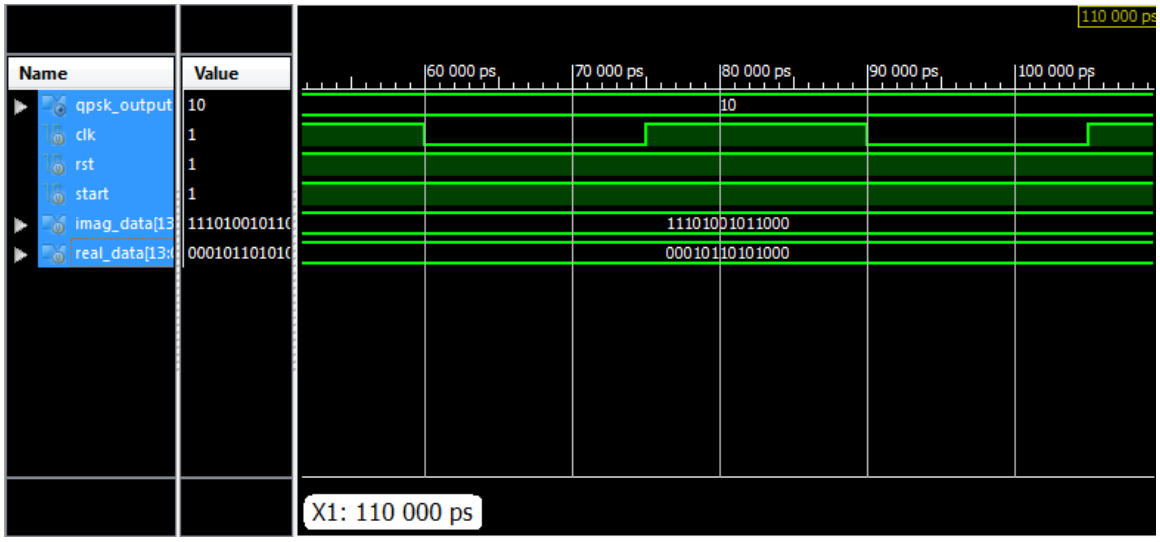

Figure 4-11 : Output of QPSK Demapper

The imaginary and real data can be seen in the top of the figure and the output demodulated signal can be seen at the end.

In this chapter we discussed the simulations and their results. The analysis helped us in checking the outputs of our coding and implementation of the system so that we could ensure the proper communication and working before we moved forward to real-time transmission and reception of the system. The real-time transmission was carried out by voice communication through OFDM.

5. Future Work

The prototype developed is implemented on the kit Xilinx Spartan 3E with 1600K gates. OFDM is continuously evolving with better modulation techniques and higher data rates and the upcoming 4G technologies in the market are employing OFDM wireless Communication standards. The implementation can be carried out on newer kits with available USB interface which can support the SDR applications through connection with USRP and much higher data rates could be achieved with better accuracy and robustness.

16-point FFT is currently being implemented in the system and can be further upgraded to 128-point FFT for higher data rates and support for newer standards of communication. The transceiver designed could be used for video transmission through wireless medium by interfacing with IP Camera and the relevant OS.

6. Conclusion

The proposed system allows faster and more robust communication as compared to older systems and the use of FPGA makes the real-time implementation which results in lesser delay. The system was implemented by Verilog HDL coding and the hardware used is the Xilinx Spartan 3E kit. The A/D and DAC present in the kit were used to test the system through voice communication.

The Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (FFT) have been chosen to implement the design instead of the Discrete Fourier Transform and Inverse Discrete Fourier Transform because they offer better speed with less computational time.

In conclusion, the main objective of this project has been successfully accomplished and the result obtained from this project is valid.

BIBLIOGRAPHY

[1] Dusan Matiae, "*OFDM as a possible modulation technique for multimedia applications in the range of mm waves*," TUD-TVS, 30-10-1998.

[2] R. W. Chang, "*Synthesis of Bandlimited Orthogonal Signals for Multichannel Data Transmission*," Bell System Tech. J., pp. 1775-1796, Dec, 1966.

[3] B. R. Saltzberg, "*Performance of an Efficient Parallel Data Transmission Sytem*," IEEE Trans. Comm. , pp 805-811, Dec, 1967.

[4] S. B. Weinstein and P.M. Ebert, "*Data transmission by frequency division multiplexing using the discrete fourier transform*," IEEE Transactions on Communication Technology", vol. COM-19, pp. 628-634, October 1971.

[5] Joaquin Garcia, Rene Cumplido *"On the design of an FPGA-Based OFDM modulator for IEEE 802.11a"* 2nd International Conference on Electrical and Electronics Engineering (ICEEE) and XI Conference on Electrical Engineering (CIE 2005)

[6] Zhi Yong Li "*OFDM transceiver design with FPGA and DEMO on DE2-70 board*" 2008

[7] Loo, Kah Cheng "*Design of an OFDM transmitter and receiver using FPGA*" 2006

[8] Maryse Wouters, Geert Vanwijnsberghe, Peter Van Wesemael, Tom Huybrechts, Steven Thoen "*Real Time Implementation on FPGA of an OFDM based Wireless LAN modem extended with Adaptive Loading*", 2010

[9] Chia-Sheng Tsai and Po-Chang Huang "*Concatenated Codes Design for OFDM based Wireless Local Area Networks*" Tatung University, Taipei, Taiwan, 2009

[10] Merlyn, M. "*FPGA implementation Of FFT processor with OFDM transceiver*" Signal and Image Processing (ICSIP), 2010 International Conference

[11] Wang Qiang Tao Cheng Huang Wei "*Efficient Implementation of Synchronization in OFDM System Based on FPGA*" Advanced Communication Technology, The 9th International Conference 12-14 Feb. 2007.

[12] Saba Zia "Performance Comparison and Evaluation of 802.11A and its Implementation in Reconfigurable Environment", 2009

Appendix – A

“Verilog Code Segments”

Scrambler:

```
module scrambler(//inputs
                        clk,
                        rst,
                        start,
                        din,
                        //outputs
                        dout_scrambler
                                                        );

inputclk  ;
inputrst  ;
input start;
input din  ;
//outputs
outputdout_scrambler;

//registers
regdout_scrambler;
reg      temp;
reg [6:0] X_reg   ;

always @(posedgeclk)
begin
if(start)
                begin
X_reg<=7'b1111111;
end

else
begin
        X_reg<={X_reg[5:0],temp};
end
end

always @(negedgeclk)
begin
temp<=X_reg[3] ^ X_reg[6];
end
```

```
always @(posedgeclk)
begin
dout_scrambler<=din ^ temp;
end

endmodule
```

Convolutional Encoder:

```
moduleconvEncoder(
                        //inputs
                        clk,
                        rst,
                //      start,
                        din,
                        encoding, //from controller
                        puncturing, //from punture
                        //outputs
                        dout
                        );
inputclk   ;
inputrst   ;
//input start ;
input din   ;
input encoding;
input puncturing;
output [1:0] dout       ;

//registers
reg [1:0] dout    ;
reg [5:0] X_reg   ;
//assign X_reg = 6'b100100 ;

always @(posedgeclk or negedgerst)
begin
        if(!rst)
        begin
dout<=0;
X_reg<=0;
```

```
        end

        else
        begin
                if( encoding || puncturing)
                begin
        dout[0]     <=X_reg[0] ^ X_reg[1] ^ X_reg[3] ^ X_reg[4] ^ din;
        dout[1]     <=X_reg[0] ^ X_reg[3] ^ X_reg[4] ^ X_reg[5] ^ din;
X_reg<={din,X_reg[5:1]};

end

else
begin
dout<=dout;
X_reg<=X_reg;
end
end
end

endmodule
```

Interleaver:

```
modulecomplete_interleaver(
                        //inputs
clk      ,
start    ,
Ncbps    ,
chnageNcbps,
serial_in ,
                        //outputs
j        ,
                        permuted_data2
                        );
//inputs
inputclk       ;
input start    ;
input [1:0]Ncbps     ;
inputchnageNcbps;
```

```
inputserial_in  ;
//outputs
output [8:0] j       ;
output permuted_data2 ;
always @ (posedgeclk)
begin
        if (start_first_permutation || chnageNcbps)         //incorporated new Ncbps signal
here so if Ncbps changes, values of k, addrb_0 and flag can be reset
        begin
        addrb_0         <=0;
        permuted_data1_0  <=serial_in;
        addrb_1         <=0;
        permuted_data1_1  <=serial_in;
        k               <=0;
        block_length_flag<=0;

        end
        else
        begin
                case(Ncbps)
        2'b00:

        if(!block_length_flag)
        begin
                if(addrb_0==47 && k==47) //check this condition again since it gives
zero address again here for this memory and stores serial_in that is a new value for zero.
                begin                   //may be it would work because read cycle occurs after
one clock cycle so it reads previous value.
                addrb_0         <=0;
                permuted_data1_0<=serial_in;
                k           <=0;
                block_length_flag<=1;
        end

                else if(addrb_0==46 && k==31)
                begin
                addrb_0         <=2;
                permuted_data1_0<=serial_in;
                k           <=k + 1;
                block_length_flag<=block_length_flag;
```

```
end

        else if(addrb_0==45 && k==15)
        begin
        addrb_0        <=1;
        permuted_data1_0<=serial_in;
        k              <=k + 1;
                        block_length_flag<=block_length_flag;
end
        else
        begin
                addrb_0        <=addrb_0 + 3;
      permuted_data1_0<=serial_in;
      k              <=k + 1;
                block_length_flag<=block_length_flag;
        end

end

else
begin
        if(addrb_1==47 && k==47)
        begin
        addrb_1        <=0;
        permuted_data1_1<=serial_in;
        k              <=0;
        block_length_flag<=0;
end

        else if(addrb_1==46 && k==31)
        begin
        addrb_1        <=2;
        permuted_data1_1<=serial_in;
        k              <=k + 1;
        block_length_flag<=block_length_flag;
end

        else if(addrb_1==45 && k==15)
        begin
        addrb_1        <=1;
```

```
        permuted_data1_1<=serial_in;
        k           <=k + 1;
                        block_length_flag<=block_length_flag;
end
        else
        begin
                addrb_1         <=addrb_1 + 3;
      permuted_data1_1<=serial_in;
      k           <=k + 1;
                block_length_flag<=block_length_flag;
        end
end

2'b01:
if(!block_length_flag)
begin


if(addrb_0==95 && k==95)
        begin
        addrb_0         <=0;
        permuted_data1_0<=serial_in;
        k           <=0;
                        block_length_flag<=1;
end
else if(addrb_0==94 && k==79)
        begin
        addrb_0         <=5;
        permuted_data1_0<=serial_in;
        k           <=k + 1;
                        block_length_flag<=block_length_flag;
end


else if(addrb_0==93 && k==63)
        begin
        addrb_0         <=4;
        permuted_data1_0<=serial_in;
        k           <=k + 1;
                        block_length_flag<=block_length_flag;
```

```
        end
        else if(addrb_0==92 && k==47)
                begin
                addrb_0         <=3;
                permuted_data1_0<=serial_in;
                k               <=k + 1;
                                block_length_flag<=block_length_flag;
        end

                else if(addrb_0==91 && k==31)
                begin
                addrb_0         <=2;
                permuted_data1_0<=serial_in;
                k               <=k + 1;
                                block_length_flag<=block_length_flag;
        end

                else if(addrb_0==90 && k==15)
                begin
                addrb_0         <=1;
                permuted_data1_0<=serial_in;
                k               <=k + 1;
                                block_length_flag<=block_length_flag;
        end
                else
                begin
                addrb_0         <=addrb_0 + 6;
          permuted_data1_0<=serial_in;
          k             <=k + 1;
                            block_length_flag<=block_length_flag;
                end

end
else
begin


        if(addrb_1==95 && k==95)
                begin
                addrb_1         <=0;
```

```
        permuted_data1_1<=serial_in;
        k       <=0;
                block_length_flag<=0;
end
else if(addrb_1==94 && k==79)
        begin
        addrb_1     <=5;
        permuted_data1_1<=serial_in;
        k       <=k + 1;
                block_length_flag<=block_length_flag;
end


else if(addrb_1==93 && k==63)
        begin
        addrb_1     <=4;
        permuted_data1_1<=serial_in;
        k       <=k + 1;
                block_length_flag<=block_length_flag;
end
else if(addrb_1==92 && k==47)
        begin
        addrb_1     <=3;
        permuted_data1_1<=serial_in;
        k       <=k + 1;
                block_length_flag<=block_length_flag;
end

        else if(addrb_1==91 && k==31)
        begin
        addrb_1     <=2;
        permuted_data1_1<=serial_in;
        k       <=k + 1;
                block_length_flag<=block_length_flag;
end

        else if(addrb_1==90 && k==15)
        begin
        addrb_1     <=1;
        permuted_data1_1<=serial_in;
```

```
            k           <=k + 1;
                        block_length_flag<=block_length_flag;
        end

always @ (posedgeclk)
begin
        if (start_second_permutation || chnageNcbps)    //add Ncbps signal to reset all
these if Ncbps changes during the block execution
        begin
        addra_1         <=0;
            addra_0         <=0;
        permuted_data2<=permuted_data1_1;
        j           <=0;
        count<=0;
        flag<=0;
        block_length_flag<=0;
        end
        else
        begin
            case(Ncbps)
        2'b00:

        if(!block_length_flag)
        begin
            if(addra_1==47 && j==47)
            begin
            addra_1         <=0;
            permuted_data2<=permuted_data1_1;
            j           <=0;
            count<=count;
            flag<= flag;
            block_length_flag<=1;


        end

            else
            begin
                    addra_1         <=addra_1 + 1;
              permuted_data2<=permuted_data1_1;
```

```
      j          <=j+ 1;
count<=count;
flag<= flag;
block_length_flag<=block_length_flag;

          end
end
else
begin
          if(addra_0==47 && j==47)
          begin
          addra_0        <=0;
          permuted_data2<=permuted_data1_0;
          j          <=0;
          count<=count;
          flag<= flag;
          block_length_flag<=0;


end

          else
          begin
                    addra_0        <=addra_0 + 1;
      permuted_data2<=permuted_data1_0;
      j          <=j+ 1;
count<=count;
flag<= flag;
block_length_flag<=block_length_flag;

          end
end
2'b01:
if(!block_length_flag)
begin
if(addra_1==95 && j==95)
          begin
          addra_1        <=0;
          permuted_data2<=permuted_data1_1;
          j          <=0;
```

```
        count<=count;
        flag<= flag;
        block_length_flag<=1;
end

        else
        begin
                addra_1        <=addra_1 + 1;
      permuted_data2<=permuted_data1_1;
      j          <=j+ 1;
count<=count;
flag<= flag;
block_length_flag<=block_length_flag;
        end
end
else
begin
if(addra_0==95 && j==95)
        begin
        addra_0        <=0;
        permuted_data2<=permuted_data1_0;
        j          <=0;
        count<=count;
        flag<= flag;
        block_length_flag<=0;
end

        else
        begin
                addra_0        <=addra_0 + 1;
      permuted_data2<=permuted_data1_0;
      j          <=j+ 1;
count<=count;
flag<= flag;
block_length_flag<=block_length_flag;
        end
end

2'b10:
if(!block_length_flag)
```

```
begin
if(addra_1==191 )
        begin
        addra_1        <=0;
        permuted_data2<=permuted_data1_1;
        j          <=0;
        count<=0;
flag<=0;
block_length_flag<=1;

end


else
        if((count==23) && (addra_1!=191) )
        begin

                        addra_1        <=addra_1 + 1;
              permuted_data2<=permuted_data1_1;
              j          <=j+2;
        count<=0;
        flag<=0;
        block_length_flag<=block_length_flag;
        end

else if(count>=11 && count<=22)
        begin
                if(!flag)
                begin
                        addra_1        <=addra_1 + 1;
              permuted_data2<=permuted_data1_1;
              j          <=addra_1+ 2;
        count<=count + 1;
        flag<=1;
        block_length_flag<=block_length_flag;
                end

                else
                begin
                        addra_1        <=addra_1 + 1;
```

```
                permuted_data2<=permuted_data1_1;
                j           <=addra_1;
            count<=count + 1;
            flag<=0;
            block_length_flag<=block_length_flag;
                    end

            end

            else
            begin
                    addra_1         <=addra_1 + 1;
          permuted_data2<=permuted_data1_1;
          j           <=j+ 1;
count<=count + 1;
flag<=flag;
block_length_flag<=block_length_flag;
            end
end
else
begin
if(addra_0==191 )
            begin
            addra_0         <=0;
            permuted_data2<=permuted_data1_0;
            j           <=0;
            count<=0;
flag<=0;
block_length_flag<=0;

end

else
            if((count==23) && (addra_0!=191) )
            begin

                        addra_0         <=addra_0 + 1;
                permuted_data2<=permuted_data1_0;
                j           <=j+2;
            count<=0;
```

```
        flag<=0;
        block_length_flag<=block_length_flag;
        end

else if(count>=11 && count<=22)
        begin
            if(!flag)
            begin
                addra_0     <=addra_0 + 1;
          permuted_data2<=permuted_data1_0;
          j         <=addra_0+ 2;
        count<=count + 1;
        flag<=1;
        block_length_flag<=block_length_flag;
            end
            else
            begin
                addra_0     <=addra_0 + 1;
          permuted_data2<=permuted_data1_0;
          j         <=addra_0;
        count<=count + 1;
        flag<=0;
        block_length_flag<=block_length_flag;
            end
```

Modulation Mapper:

```
moduleqpsk_main(
clk, rst, start, qpsk_input,   //inputs
imag_data, real_data, modulated        //outputs
                                       );

////////////////////////////////////qpsk ram ////////////
always @ (posedgeclk)
begin
case (qpsk_input)

                        2'b00:data_frm_qpsk<=28'b11101001011000_11101001011000 ;
//-1,-1  //now +1=.707 and -1= -.707
                    2'b01:data_frm_qpsk<=28'b00010110101000_11101001011000 ;  //1 ,-
1
          2'b11:data_frm_qpsk<=28'b00010110101000_00010110101000 ;  //1 , 1
          2'b10:data_frm_qpsk<=28'b11101001011000_00010110101000 ;  //-1, 1

        endcase

end
////////////////////////////////mapper/////////
always @ (posedgeclk or negedgerst)

if (!rst)
begin
imag_data<=0;
real_data<=0;
end
else
begin
        if (start)
        begin
        real_data<= data_frm_qpsk [13:0];
        imag_data<= data_frm_qpsk [27:14] ;
        modulated<= data_frm_qpsk;

        end
end
endmodule
```

Fast Fourier Transform:

```
moduledataAddressGeneratorIn(
                //inputs
clk                 ,
rst                 ,
total_stages        ,       //output from FFT initialization
gen_data_address    ,
stage_number        ,
butterfly_number    ,
                //outputs
data_address_gen_done   ,
                data_address1           ,
                data_address2
                );
always @(posedgeclk or negedgerst)
begin
        if(!rst)
        begin
        data_address_gen_done<=0;
        data_address1       <=0;
   data_address2       <=0;
generating<=0;
        end
else
        begin
                if(gen_data_address || generating)
                begin
                data_address_gen_done<=1;
                data_address1       <=butterfly_number - 1;
      data_address2       <=(temp_reg<<(total_stages -
stage_number))+(butterfly_number-1);
generating<=1;
                end
                else
                begin
                data_address_gen_done<=0                    ;
                data_address1       <=data_address1       ;
      data_address2       <=data_address2       ;
generating<=generating;
end
```

```
end
end
endmodule
module radix2cell_datapath(
		//inputs
clk,
rst,
		start_radix2,
ADDING_state,
			PRODUCING_ADDITIONS_state,
		data_in_line1re,
		data_in_line2re,
		data_in_line1im,
		data_in_line2im,
		//outputs
		data_out_line1re,
		data_out_line2re,
		data_out_line1im,
		data_out_line2im,
		radix2_done
		);

always @(posedgeclk or negedgerst)
begin
	if(!rst)
	begin
   data_out_line1re<=0;
   data_out_line2re<=0;
   data_out_line1im<=0;
   data_out_line2im<=0;
   radix2_done=0;
	end
else
	begin
		if(start_radix2 || ADDING_state ||PRODUCING_ADDITIONS_state)
		begin
			data_out_line1re<=data_in_line1re + data_in_line2re;
			data_out_line2re<=data_in_line1re - data_in_line2re;
			data_out_line1im<=data_in_line1im + data_in_line2im;
			data_out_line2im<=data_in_line1im - data_in_line2im;
```

```
          radix2_done=1;
                end
else


begin
                        data_out_line1re<=0;
                        data_out_line2re<=0;
                        data_out_line1im<=0;
                        data_out_line2im<=0;
          radix2_done=0;
                end
          end
end
endmodule

moduletwiddleFactorMultiplication( //inputs
clk                    ,
rst                    ,
twiddle_factor_cos          , //ouput from twiddle factor mem controller
                data_out_line2re            , //from radix cell...to be multiplied with cos
twiddle
start_twiddle_multiplication ,
MULTIPLYING_state      ,
          PRODUCING_PRODUCTS_state,

                //outputs
                mult_data_in_line2re
                );
always @( negedgeclk or negedgerst)
begin

          if(!rst)
          begin
          twiddle_factor_cos_reg<=0;
       data_out_line2re_reg          <=0;
       mult_data_in_line2re_reg_MSB    <=0;


          end
```

```
        else
        begin

        if(twiddle_factor_cos[5] && !data_out_line2re[13] )
                begin
                        if((data_out_line2re==0) || (twiddle_factor_cos==0))
                        begin
                                twiddle_factor_cos_reg<=~(twiddle_factor_cos[4:0]) +
12'd1;
            data_out_line2re_reg        <=data_out_line2re[12:0]  ;
            mult_data_in_line2re_reg_MSB <=0;

                        end
                        else
                        begin
twiddle_factor_cos_reg<=~(twiddle_factor_cos[4:0]) + 12'd1;
            data_out_line2re_reg        <=data_out_line2re[12:0]  ;
            mult_data_in_line2re_reg_MSB <=1;
end
end

                else if(!twiddle_factor_cos[5] && data_out_line2re[13] )
                begin
                        if((data_out_line2re==0) || (twiddle_factor_cos==0))
                        begin
                                twiddle_factor_cos_reg<=twiddle_factor_cos[4:0];
            data_out_line2re_reg        <=~(data_out_line2re[12:0]) + 12'd1  ;
            mult_data_in_line2re_reg_MSB <=0;

                        end
                        else
                begin
twiddle_factor_cos_reg<=twiddle_factor_cos[4:0];
            data_out_line2re_reg        <=~(data_out_line2re[12:0]) + 12'd1  ;
            mult_data_in_line2re_reg_MSB <=1;
end
end
                else if(twiddle_factor_cos[5] && data_out_line2re[13])
                begin
twiddle_factor_cos_reg<=~(twiddle_factor_cos[4:0]) + 12'd1;
```

```
            data_out_line2re_reg       <=~(data_out_line2re[12:0]) + 12'd1  ;
            mult_data_in_line2re_reg_MSB<=0;
end
                else
                begin
twiddle_factor_cos_reg<=twiddle_factor_cos[4:0];
            data_out_line2re_reg       <=data_out_line2re[12:0]  ;
            mult_data_in_line2re_reg_MSB<=0;
end


end
end


always @( negedgeclk or negedgerst)
begin

        if(!rst)
        begin
                 partial_prod0              <=0;
      partial_prod1             <=0;
      partial_prod2             <=0;
      partial_prod3             <=0;
      partial_prod4             <=0;
      mult_data_in_line2re_reg_MSB0 <=0;
        end
        else
        begin
                if (start_twiddle_multiplication || MULTIPLYING_state ||
PRODUCING_PRODUCTS_state)
                begin
```

```
        partial_prod0       <={5'b00000,(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[12]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[11]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[10]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[9]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[8]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[7]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[6]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[5]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[4]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[3]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[2]),(twiddle_factor_cos_reg[0] &
data_out_line2re_reg[1]),(twiddle_factor_cos_reg[0] & data_out_line2re_reg[0])};
        partial_prod1       <={4'b0000,(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[12]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[11]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[10]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[9]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[8]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[7]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[6]),(twiddle_factor_cos_reg[1]
&data_out_line2re_reg[5]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[4]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[3]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[2]),(twiddle_factor_cos_reg[1] &
data_out_line2re_reg[1]),(twiddle_factor_cos_reg[1] & data_out_line2re_reg[0]),1'b0};
        partial_prod2       <={3'b000,(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[12]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[11]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[10]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[9]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[8]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[7]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[6]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[5]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[4]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[3]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[2]),(twiddle_factor_cos_reg[2] &
data_out_line2re_reg[1]),(twiddle_factor_cos_reg[2] & data_out_line2re_reg[0]),2'b00};
```

```
            partial_prod3               <={2'b00,(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[12]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[11]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[10]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[9]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[8]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[7]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[6]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[5]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[4]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[3]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[2]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[1]),(twiddle_factor_cos_reg[3] &
data_out_line2re_reg[0]),3'b000};
            partial_prod4               <={1'b0,(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[12]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[11]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[10]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[9]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[8]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[7]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[6]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[5]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[4]),(twiddle_factor_cos_reg[4]
&data_out_line2re_reg[3]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[2]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[1]),(twiddle_factor_cos_reg[4] &
data_out_line2re_reg[0]),4'b0000};
    mult_data_in_line2re_reg_MSB0 <=mult_data_in_line2re_reg_MSB;
end
else
begin
      partial_prod0             <=0;
      partial_prod1             <=0;
      partial_prod2             <=0;
      partial_prod3             <=0;
      partial_prod4             <=0;
      mult_data_in_line2re_reg_MSB0 <=0;
end
```

```
        end

end

always @( negedgeclk or negedgerst)
begin

        if(!rst)
        begin
            mult_data_in_line2re_reg<=0;
            mult_data_in_line2re_reg_MSB1<=0;
        end
        else
                if (MULTIPLYING_state || PRODUCING_PRODUCTS_state)
        begin
         mult_data_in_line2re_reg     <=partial_prod0 + partial_prod1 + partial_prod2 +
partial_prod3 + partial_prod4;
         mult_data_in_line2re_reg_MSB1<=mult_data_in_line2re_reg_MSB0;
        end
        else
        begin
          mult_data_in_line2re_reg     <=mult_data_in_line2re_reg     ;
          mult_data_in_line2re_reg_MSB1<=mult_data_in_line2re_reg_MSB1;
        end
end

always @( negedgeclk or negedgerst)
begin

        if(!rst)
        begin
            mult_data_in_line2re_reg_MSB2<=0;
            mult_data_in_line2re_trunc    <= 0;
        end
        else
                                if (MULTIPLYING_state ||
PRODUCING_PRODUCTS_state)

        begin
```

```
            mult_data_in_line2re_reg_MSB2    <= mult_data_in_line2re_reg_MSB1;

        if(mult_data_in_line2re_reg[4:0]<16)
            mult_data_in_line2re_trunc <=mult_data_in_line2re_reg[17:5];
        else
            mult_data_in_line2re_trunc <=mult_data_in_line2re_reg[17:5]  + 13'd1;
        end

        else
        begin
            mult_data_in_line2re_trunc    <= 0;
            mult_data_in_line2re_reg_MSB2<=0;

        end

end

always @( negedgeclk or negedgerst)
begin
        if(!rst)
        begin
      mult_data_in_line2re_delay[12:0]  <=0;
      mult_data_in_line2re_delay[13]      <= 0;

        end
        else
                                        if (MULTIPLYING_state ||
PRODUCING_PRODUCTS_state)
        begin
                mult_data_in_line2re_delay[13]<=mult_data_in_line2re_reg_MSB2;

if(mult_data_in_line2re_reg_MSB2)
            mult_data_in_line2re_delay[12:0]  <=~(mult_data_in_line2re_trunc) +
13'd1;
        else
            mult_data_in_line2re_delay[12:0]  <=mult_data_in_line2re_trunc;
        end
        else
        begin
```

```
                mult_data_in_line2re_delay[12:0]  <=0;
                mult_data_in_line2re_delay[13]<=0;
end
end

always @( posedgeclk or negedgerst)
begin
        if(!rst)
        begin
      mult_data_in_line2re[12:0]  <=0;
      mult_data_in_line2re[13]      <= 0;

        end
        else
                                        if (MULTIPLYING_state ||
PRODUCING_PRODUCTS_state)
        begin
                mult_data_in_line2re[13]  <=mult_data_in_line2re_delay[13];
                mult_data_in_line2re[12:0]<=mult_data_in_line2re_delay;

        end
        else
        begin
         mult_data_in_line2re[13]   <=0;
          mult_data_in_line2re[12:0]<=0;

end


end
endmodule
```

Guard Insertion:

```
moduleGI_write(
clk,
        BitReversedPoint,
        start_GI,
        //outputs
        addrb_0,
        addrb1_0,
        DataPoint_0,
        GIpoint_0,
        addrb_1,
        addrb1_1,
        DataPoint_1,
        GIpoint_1,
        addrb_2,
        addrb1_2,
        DataPoint_2,
        GIpoint_2,
        addrb_3,
        addrb1_3,
        DataPoint_3,
        GIpoint_3
        );

inputclk;
input [27:0] BitReversedPoint          ;
inputstart_GI ;
//outputs
output [5:0]     addrb_0            ;
output [3:0]     addrb1_0             ;
output [27:0] DataPoint_0         ;
output [27:0] GIpoint_0          ;

output [5:0]     addrb_1            ;
output [3:0]     addrb1_1             ;
output [27:0] DataPoint_1         ;
output [27:0] GIpoint_1          ;

output [5:0]     addrb_2            ;
output [3:0]     addrb1_2             ;
```

```
output [27:0] DataPoint_2          ;
output [27:0] GIpoint_2          ;

output [5:0]     addrb_3             ;
output [3:0]     addrb1_3             ;
output [27:0] DataPoint_3          ;
output [27:0] GIpoint_3          ;

//registers
reg [5:0]     addrb_0             ;
reg [3:0]     addrb1_0             ;
reg [27:0] DataPoint_0          ;
reg [27:0] GIpoint_0          ;

reg [5:0]     addrb_1             ;
reg [3:0]     addrb1_1             ;
reg [27:0] DataPoint_1          ;
reg [27:0] GIpoint_1          ;

reg [5:0]     addrb_2             ;
reg [3:0]     addrb1_2             ;
reg [27:0] DataPoint_2          ;
reg [27:0] GIpoint_2          ;

reg [5:0]     addrb_3             ;
reg [3:0]     addrb1_3             ;
reg [27:0] DataPoint_3          ;
reg [27:0] GIpoint_3          ;

reg [1:0] mem_count;

reg [5:0]     addrb         ;



always @(posedgeclk)
begin
        if(start_GI)
        begin
        mem_count<=0;
```

```
            addrb_0    <=0;
            addrb_1    <=0;
            addrb_2    <=0;
            addrb_3    <=0;
            DataPoint_0<=0;
            DataPoint_1<=0;
            DataPoint_2<=0;
            DataPoint_3<=0;
          addrb<=0;
          end
          else
          case(mem_count)
                  0:
          if(addrb==15)
          begin
               addrb_0<=addrb;
          addrb<=0;
mem_count<=1;
        DataPoint_0<=BitReversedPoint;
            addrb_1    <=addrb_1;
            addrb_2    <=addrb_2;
            addrb_3    <=addrb_3;
      DataPoint_1<=DataPoint_1;
      DataPoint_2<=DataPoint_2;
      DataPoint_3<=DataPoint_3;


          end
          else
          begin
               addrb_0<=addrb;
          addrb<=addrb + 1;
mem_count<=mem_count;
        DataPoint_0<=BitReversedPoint;
            addrb_1    <=addrb_1;
            addrb_2    <=addrb_2;
            addrb_3    <=addrb_3;
      DataPoint_1<=DataPoint_1;
      DataPoint_2<=DataPoint_2;
      DataPoint_3<=DataPoint_3;
```

```
end

    1:
if(addrb==15)
          begin
               addrb_1<=addrb;
          addrb<=0;
mem_count<=2;
        DataPoint_1<=BitReversedPoint;
        addrb_0    <=addrb_0;
              addrb_2    <=addrb_2 ;
              addrb_3    <=addrb_3 ;
              DataPoint_0<=DataPoint_0;
        DataPoint_2<=DataPoint_2;
        DataPoint_3<=DataPoint_3;
          end
          else
          begin
               addrb_1<=addrb;
          addrb<=addrb + 1;
mem_count<=mem_count;
        DataPoint_1<=BitReversedPoint;
        addrb_0    <=addrb_0;
              addrb_2    <=addrb_2;
              addrb_3    <=addrb_3;
              DataPoint_0<=DataPoint_0;
        DataPoint_2<=DataPoint_2;
        DataPoint_3<=DataPoint_3;


end
    2:
if(addrb==15)
          begin
               addrb_2<=addrb;
          addrb<=0;
mem_count<=3;
        DataPoint_2<=BitReversedPoint;
        addrb_0    <=addrb_0;
              addrb_1    <=addrb_1;
```

```
            addrb_3    <=addrb_3;
            DataPoint_0<=DataPoint_0;
        DataPoint_1<=DataPoint_1;
        DataPoint_3<=DataPoint_3;
          end
          else
          begin
               addrb_2<=addrb;
          addrb<=addrb + 1;
mem_count<=mem_count;
        DataPoint_2<=BitReversedPoint;
        addrb_0    <=addrb_0;
            addrb_1    <=addrb_1;
            addrb_3    <=addrb_3;
            DataPoint_0<=DataPoint_0;
        DataPoint_1<=DataPoint_1;
        DataPoint_3<=DataPoint_3;
end
    3:
if(addrb==15)
          begin
               addrb_3<=addrb;
          addrb<=0;
mem_count<=0;
        DataPoint_3<=BitReversedPoint;
        addrb_0    <=addrb_0;
            addrb_1    <=addrb_1;
            addrb_2    <=addrb_2;
            DataPoint_0<=DataPoint_0;
        DataPoint_1<=DataPoint_1;
        DataPoint_2<=DataPoint_2;
          end
          else
          begin
               addrb_3<=addrb;
          addrb<=addrb + 1;
mem_count<=mem_count;
        DataPoint_3<=BitReversedPoint;
        addrb_0    <=addrb_0;
            addrb_1    <=addrb_1;
```

```
            addrb_2    <=addrb_2;
            DataPoint_0<=DataPoint_0;
        DataPoint_1<=DataPoint_1;
        DataPoint_2<=DataPoint_2;
end
endcase
end

//GI band memory storage

always @(negedgeclk)
begin
          if(start_GI)
          begin
            addrb1_0      <=0;
            addrb1_1      <=0;
            addrb1_2      <=0;
            addrb1_3      <=0;
            GIpoint_0     <=0;
            GIpoint_1     <=0;
            GIpoint_2     <=0;
            GIpoint_3     <=0;
          end
          else
          case(mem_count)
                  0:
          if(addrb_0>=0 && addrb_0<=15)
          begin
               addrb1_0    <=addrb_0;
        GIpoint_0<=DataPoint_0;
          end
          else
          begin
               addrb1_0<=addrb1_0;
        GIpoint_0<=GIpoint_0;
end
    1:
if(addrb_1>=0 && addrb_1<=15)
          begin
               addrb1_1    <=addrb_1;
```

```
        GIpoint_1<=DataPoint_1;
          end
          else
          begin
               addrb1_0<=addrb1_0;
        GIpoint_0<=GIpoint_0;
end
    2:
if(addrb_2>=0 && addrb_2<=15)
          begin
               addrb1_2    <=addrb_2;
        GIpoint_2<=DataPoint_2;
          end
          else
          begin
               addrb1_2<=addrb1_2;
        GIpoint_2<=GIpoint_2;
end
    3:
if(addrb_3>=0 && addrb_3<=15)
          begin
               addrb1_3    <=addrb_3;
        GIpoint_3<=DataPoint_3;
          end
          else
          begin
               addrb1_3<=addrb1_3;
        GIpoint_3<=GIpoint_3;
end
endcase
end
endmodule
```

Appendix – B

“Spartan 3E Starter Board Data Sheets”

Spartan-3E FPGA Features and Embedded Processing Functions

The Spartan-3E Starter Kit board highlights the unique features of the Spartan-3E FPGA family and provides a convenient development board for embedded processing applications. The board highlights these features:

- Spartan-3E FPGA specific features
- Parallel NOR Flash configuration
- MultiBoot FPGA configuration from Parallel NOR Flash PROM
- SPI serial Flash configuration
- Embedded development
- MicroBlaze™ 32-bit embedded RISC processor
- PicoBlaze™ 8-bit embedded controller
- DDR memory interfaces

Key Components and Features

The key features of the Spartan-3E Starter Kit board are:

- Xilinx XC3S500E Spartan-3E FPGA
- Up to 232 user-I/O pins
- 320-pin FBGA package
- Over 10,000 logic cells
- Xilinx 4 Mbit Platform Flash configuration PROM
- Xilinx 64-macrocell XC2C64A CoolRunner™ CPLD
- 64 MByte (512 Mbit) of DDR SDRAM, x16 data interface, 100+ MHz
- 16 MByte (128 Mbit) of parallel NOR Flash (Intel StrataFlash)
- FPGA configuration storage
- MicroBlaze code storage/shadowing
- 16 Mbits of SPI serial Flash (STMicro)
- FPGA configuration storage
- MicroBlaze code shadowing
- 2-line, 16-character LCD screen
- PS/2 mouse or keyboard port
- VGA display port
- 10/100 Ethernet PHY (requires Ethernet MAC in FPGA)
- Two 9-pin RS-232 ports (DTE- and DCE-style)
- On-board USB-based FPGA/CPLD download/debug interface
- 50 MHz clock oscillator
- SHA-1 1-wire serial EEPROM for bitstream copy protection
- Hirose FX2 expansion connector
- Three Digilent 6-pin expansion connectors
- Four-output, SPI-based Digital-to-Analog Converter (DAC)
- Two-input, SPI-based Analog-to-Digital Converter (ADC) with programmable-gain pre-amplifier
- ChipScope™ SoftTouch debugging port
- Rotary-encoder with push-button shaft
- Eight discrete LEDs
- Four slide switches
- Four push-button switches

• SMA clock input
• 8-pin DIP socket for auxiliary clock oscillator

Design Trade-Offs

A few system-level design trade-offs were required in order to provide the Spartan-3E Starter Kit board with the most functionality.

Configuration Methods Galore!

A typical FPGA application uses a single non-volatile memory to store configuration images. To demonstrate new Spartan-3E FPGA capabilities, the starter kit board has three different configuration memory sources that all need to function well together. The extra configuration functions make the starter kit board more complex than typical Spartan-3E FPGA applications.
The starter kit board also includes an on-board USB-based JTAG programming interface. The on-chip circuitry simplifies the device programming experience. In typical applications, the JTAG programming hardware resides off-board or in a separate programming module, such as the Xilinx Platform USB cable.

Voltages for all Applications

The Spartan-3E Starter Kit board showcases a triple-output regulator developed by Texas Instruments, the TPS75003 specifically to power Spartan-3 and Spartan-3E FPGAs. This regulator is sufficient for most stand-alone FPGA applications. However, the starter kit board includes DDR SDRAM, which requires its own high-current supply. Similarly, the USB-based JTAG download solution requires a separate 1.8V supply.

RS-232 Serial Ports

Overview

As shown in Figure 7-1, the Spartan®-3E FPGA Starter Kit board has two RS-232 serial ports: a female DB9 DCE connector and a male DTE connector. The DCE-style port connects directly to the serial port connector available on most personal computers and workstations via a standard straight-through serial cable. Null modem, gender changers, or crossover cables are not required. Use the DTE-style connector to control other RS-232 peripherals, such as modems or printers, or perform simple loopback testing with the DCE connector. Note that Figure 7-1 shows the view looking “out” the DTE connector.

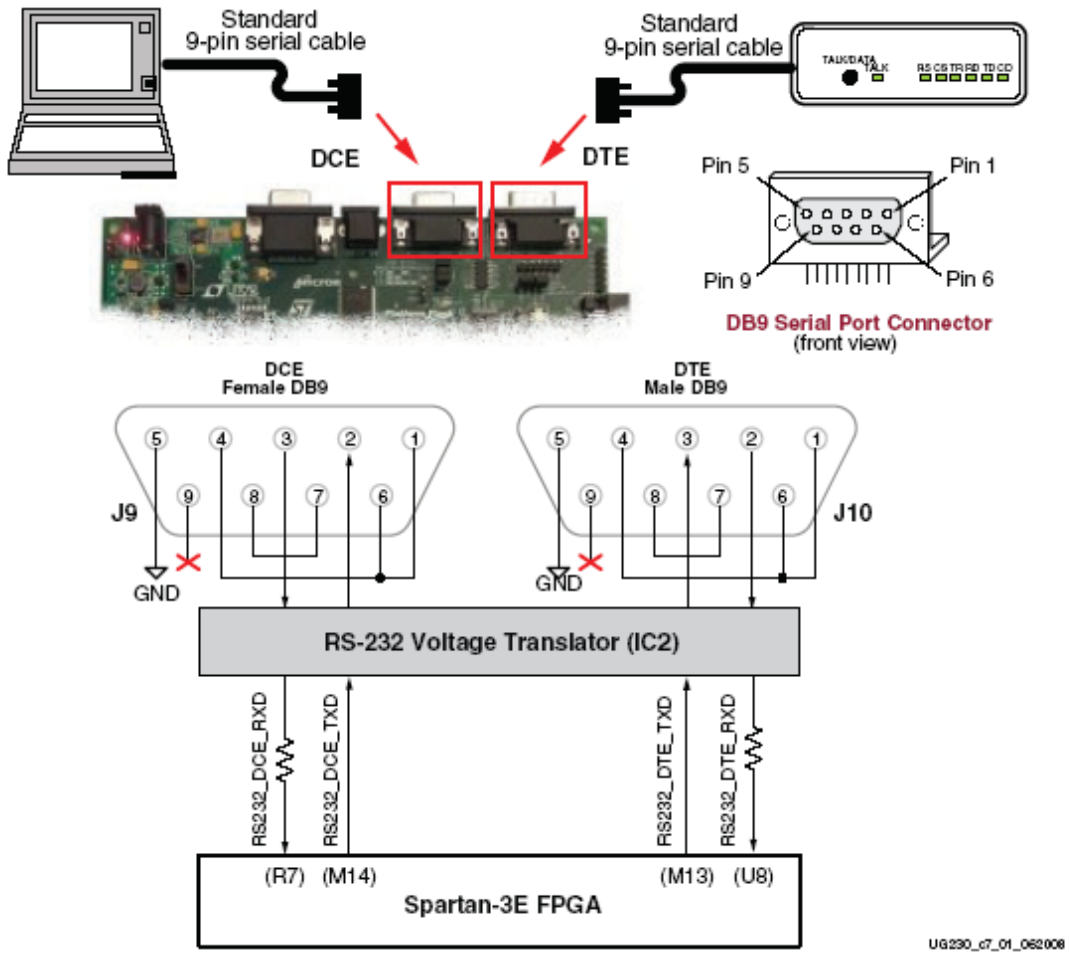

Figure 7-1: **RS-232 Serial Ports**

DB9 Serial Port Connector
(front view)
Standard
9-pin serial cable
Standard
9-pin serial cable
RS CS TR RD TD CD
TALK/DATA
TALK
RS-232 Peripheral
UG230_c7_01_062008

Figure 7-1 shows the connection between the FPGA and the two DB9 connectors. The FPGA supplies serial output data using LVTTL or LVCMOS levels to the Maxim device, which in turn, converts the logic value to the appropriate RS-232 voltage level. Likewise, the Maxim device converts the RS-232 serial input data to LVTTL levels for the FPGA. A series resistor between the Maxim output pin and the FPGA's RXD pin protects against accidental logic conflicts.
Hardware flow control is not supported on the connector. The port's DCD, DTR, and DSR signals connect together, as shown in Figure 7-1. Similarly, the port's RTS and CTS signals connect together.

UCF Location Constraints

The data below provide the UCF constraints for the DTE and DCE RS-232 ports, respectively, including the I/O pin assignment and the I/O standard used.

```
NET "RS232_DTE_RXD" LOC = "U8" | IOSTANDARD = LVTTL ;
NET "RS232_DTE_TXD" LOC = "M13" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW = SLOW ;
NET "RS232_DCE_RXD" LOC = "R7" | IOSTANDARD = LVTTL ;
NET "RS232_DCE_TXD" LOC = "M14" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW = SLOW ;
```