DESIGN AND DEVELOPMENT OF DIGITAL BASEBAND PROCESSING MODULE FOR SDR

By

Bakhtawar Hasan

Nabila Shahid

Rabeea Suhail

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

JUNE 2012

ABSTRACT

DESIGN AND DEVELOPMENT OF DIGITAL BASEBAND PROCESSING MODULE FOR SDR

Design and development of a standalone Digital Baseband Processing Module has been carried out according to IEEE 802.11a standard. We have considered utilizing the existing resources for developing the Digital baseband module to provide technology self-reliance and import substitution.

Implementation of Orthogonal Frequency Division Multiplexing (OFDM) which is a multi carrier modulation technique has been performed that provides high bandwidth efficiency as the carriers are orthogonal to each other and share data among themselves. The complete Verilog code has been developed for the system according to 802.11a standard. The kit that has been used for Hardware Programming is Xilinx Virtex5-LX50T. Prior to downloading onto the Hardware, the code has been thoroughly synthesized and checked using in Xilinx ISE suite v12.3 and the simulation results were validated by comparing with MATLAB simulation.

The performance of the baseband processing module has been tested by giving live audio signals as input and recovering the same signal at the other end successfully.

Dedicated to

Almighty Allah for His blessings,

Teachers and friends for their help

And Our Parents for their support and prayers

ACKNOWLEDGEMENT

We would like to express our gratitude towards our advisor Dr Adnan Rashdi for all his help, invaluable guidance, critics and generous support throughout our final year project.

Special acknowledgements to all those teachers at NUST and other Universities for helping us. Their interest in this project was very beneficial and helped design many vital parts of the project.

We would also like to 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 TABLES

Table No. **Page No.**

LIST OF FIGURES

Figure No. **Page No.**

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	Carrier Interference
ISI	Inter Symbol Interference
LUT	Look up Tables
OFDM	Orthogonal Frequency Division Multiplexing
QPSK	Quadrature Phase Shift Keying
UCF	User Constraints File

CHAPTER 1

INTRODUCTION

1.1. OVERVIEW

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

1.2. PROJECT BACKGROUND

The design of the project has been carried according to IEEE 802.11a using OFDM technique. 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 alI the carriers orthogonal to one another, preventing interference between the closely spaced carriers.

There are various methods to implement such digital communication systems. One of the methods to implement the system is using ASIC (Application Specific Integrated Circuit) and another method is to use a general purpose Microprocessor or Micro Controller. The main problem using ASICs is inflexibility of design process involved and the longer time to market period for the designed chip. The disadvantage of using Micro Controller is that it needs memory and other peripheral chips to support the operation.

Due to the limitations of above mentioned equipments, Field-Programmable Gate Array (FPGA) has been used as the hardware platform. We have employed Xilinx Virtex-5 FPGA. FPGAs are chips, which are programmed by the user to perform the desired functionality.

The chips may be programmed either *Once:* Antifuse technology, e.g. devices manufactured by Quicklogic; *Several times:* Flash based, e.g. devices manufactured by Actel; *Dynamically:* SRAM based, e.g. device manufactured by Actel, Altera, Atmel, Cypress, Lucent, Xilinx.

The project implementation consists of an ADC module which takes analog data as an input, and after digitizing, sends it to the Baseband Processing module using USB port. The Baseband Processing Module processes it and sends it to the DAC module, which converts the signal back into analog form which can be interpreted easily.

1.3. PROJECT OBJECTIVE

The objective of this project is to cut dependence on foreign products and to develop baseband processing module indigenously with no proprietary issues and to carry out an efficient implementation of the OFDM system (i.e. transmitter and receiver) by the implementation of baseband processing using “Field Programmable Gate Array (FPGA)”. FPGA has been chosen as the target platform because OFDM has large arithmetic processing requirements which can become prohibitive if implemented in software on a Digital Signal Processor (DSP).

1.4 DIGITAL COMMUNICATION SYSTEM

OFDM is a digital modulation technique; therefore an introduction to digital communication systems is being provided. A digital communication system involves the transmission of information in digital form from one point to another. Figure 1.1 shows a typical digital communication system.

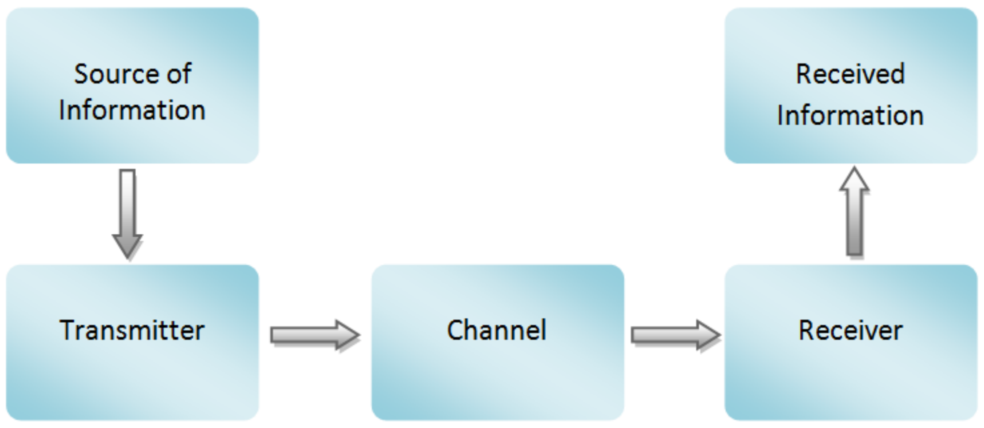


Figure 1.1 A Typical Digital Communication System

The three basic elements in a communication system are transmitter, channel and receiver. 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 coder and modulation. Source encoder provides an efficient representation of the information through which the resources are conserved. A channel coder may include error detection and correction code. A modulation process then converts the base band signal into band pass signal before transmission. The designed digital baseband processing module consists of a scrambler, convolutional encoder, interleaver, ifft and guard insertion at the transmitter end and reverse at the receiver end. All of these modules are implemented, and the codes in Verilog are burnt on the kit for baseband processing.

1.5 ORTHOGONAL FREQUENCY DIVISION MULTIPLEXING

OFDM is a multi-carrier digital modulation technique that has been recognized as an excellent method for high speed bi-directional wireless data communication. OFDM effectively squeezes multiple modulated carriers tightly together, reducing the required bandwidth but keeping the modulated signals orthogonal so they do not interfere with each other. OFDM is similar to FDM but much more spectrally efficient by spacing the sub-channels much closer together (until they are actually overlapping). This is done by finding frequencies that are orthogonal, which means that they are perpendicular in a mathematical sense, allowing the spectrum of each sub-channel to overlap another without interfering with it. In Figure 1.2 the effect of this is seen, as the required bandwidth is greatly reduced by removing guard bands (which are present in FDM) and allowing signals to overlap.

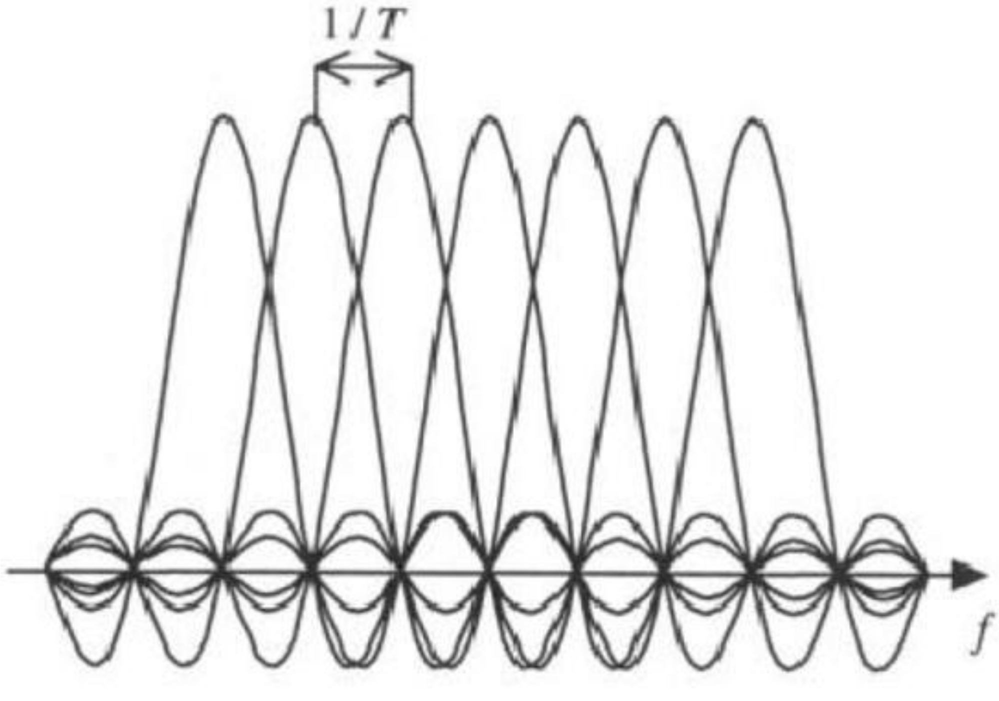


Figure 1.2   System Overlap in OFDM [2]

1.6 COMPLETE OFDM SYSTEM

The complete OFDM system Block diagram showing all the modules involved can be seen in Figure 1.3.

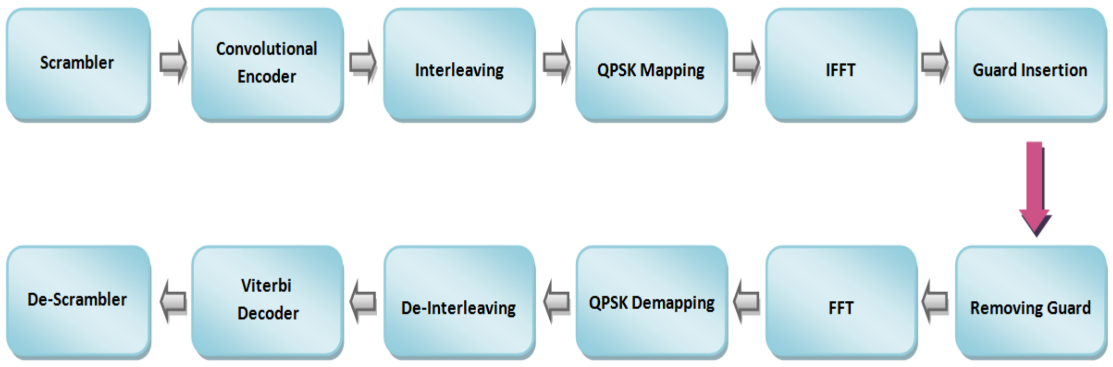


Figure 1.3 Block Diagram of OFDM System

1.6.1 SCRAMBLER / DESCRAMBLER

Data bits are given to the transmitter as input which pass through a scrambler that randomizes the bit sequence. It is done to make the input sequence more disperse so that the dependence of input signal's power spectrum on the actual transmitted data can be eliminated. At the receiver end descrambling is the last step.

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).

1.6.2 CONVOLUTIONAL ENCODER /DECODER

Convolutional Encoder is implemented in the OFDM to provide forward error correction. It provides redundant bits on channel to incorporate channel encoding. As per the specifications of IEEE 802.11a, the generator polynomials used are g0=x5+x4+x2+x and g1=x5+x2+x+1. The data rates are improved further 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. The convolutional encoding rate used in the project is according to the IEEE Standard 802.11a which specifies the rate of ½.

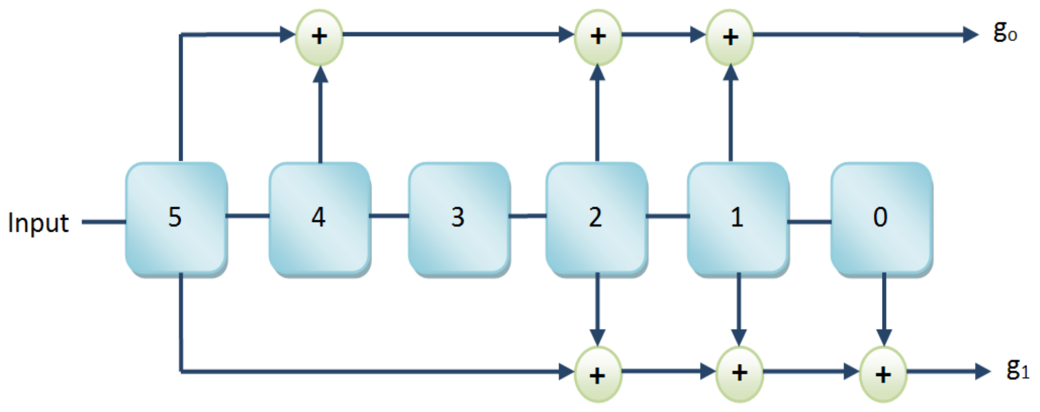


Figure 1.4 Convolutional Encoder

1.6.3 INTERLEAVING / DEINTERLEAVING

Interleaving is done to protect the data from burst errors during transmission. Conceptually, the in-coming bit stream is re-arranged so that adjacent bits are no more adjacent to each other. The data is broken into blocks and the bits within a block are rearranged. Talking in terms of OFDM, the bits within an OFDM symbol are rearranged in such a fashion so that adjacent bits are placed on non-adjacent sub-carriers. As far as De-Interleaving is concerned, it again rearranges the bits into original form during reception.

1.6.4 CONSTELLATION MAPPER / DE-MAPPER

The Constellation Mapper basically maps the incoming (interleaved) bits onto different sub-carriers. Different modulation techniques can be employed (such as QPSK, BPSK, QAM etc.) for different sub-carriers. The De-Mapper simply extracts bits from the modulated symbols at the receiver.

1.6.5 INVERSE FAST FOURIER TRANSFORM / FAST FOURIER TRANSFORM

This is the most important block in the OFDM communication system. It is IFFT that basically gives OFDM its orthogonality[2] . The IFFT transform a spectrum (amplitude and phase of each component) into a time domain signal. It converts a number of complex data points into the same number of points in time domain. Similarly, FFT at the receiver side performs the reverse task i.e. conversion from time domain back to frequency domain.

1.6.6 GUARD INSERTION / REMOVAL

In order to preserve the sub-carrier orthogonality and the independence of subsequent OFDM symbols, a cyclic guard interval is introduced. The guard period is specified in terms of the fraction of the number of samples that make up an OFDM symbol. The cyclic prefix contains a copy of the end of the forthcoming symbol. Addition of cyclic prefix results in circular convolution between the transmitted signal and the channel impulse response. Frequency domain equivalent of circular convolution is simply the multiplication of transmitted signal’s frequency response and channel frequency response, therefore received signal is only a scaled version of transmitted signal (in frequency domain), hence distortions due to severe channel conditions are eliminated[2]. Removal of cyclic prefix is then done at the receiver end and the cyclic prefix–free signal is passed through the various blocks of the receiver.

1.7 FPGA DESIGN FLOW

According to modern standards, a logic circuit with 20000 gates is common. In order to implement large circuits, it is convenient to use a type of programmable chip that has a large logic capacity. A field programmable gate arrays (FPGA) is a programmable logic device that supports implementation of relatively large logic circuits [2]. FPGA is different from other logic technologies like complex programmable logic device (CPLD) and simple programmable logic device (SPLD) because FPGA does not contain AND or OR planes.

Instead, FPGA consists of logic blocks for implementing required functions. An FPGA contains 3 main types of resources: logic blocks, I/O blocks for connecting to the pins of the package, and interconnection wires and switches. The logic blocks are arranged in a two-dimensional array, and the interconnection wires are organized as horizontal and vertical routing channels between rows and columns of the logic blocks [3].

The routing channels contain wires and programmable switches that allow the logic blocks to be interconnected in many ways. FPGA can be used to implement logic circuits of more than a few hundred thousand equivalent gates in size [3]. Equivalent gates is a way to quantify a circuit’s size by assuming that the circuit is to be built using only simple logic gate and then estimating how many of these gates are needed. Figure 1.5 gives a clear picture of the FPGA design flow.

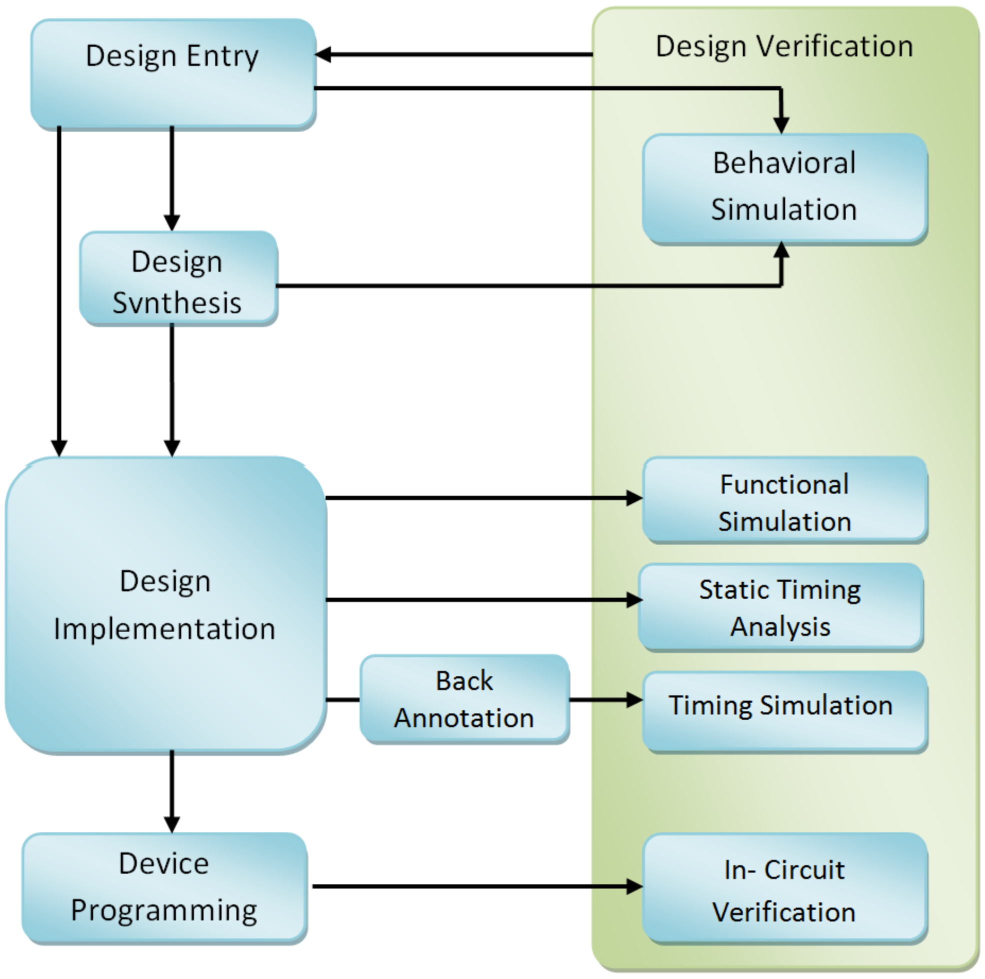


Figure 1.5 The FPGA Design Flow

1.8 PROJECT SPECIFICATIONS

The complete OFDM system, comprising of the transmitter and the receiver, has been implemented on a single FPGA board. The overall specification factors include a FPGA board: Xilinx Virtex 5 FPGA; Data Input and output: FPGA kit's USB ports; Software model of the OFDM system created in MATLAB; Verilog used as the hardware description language; ISim and ModelSim 6.1 used for simulation of the design; Xilinx ISE Design Suite v12.3 used to map the design to targeted device (Virtex 5).

Top level architecture of the proposed OFDM system is shown in Figure 1.6. It is very challenging on how software algorithm may be mapped to hardware logic. A variable may correspond to a wire or a register depending on its application and sometimes an operator can be mapped to hardware like adders, latches, multiplexers etc.

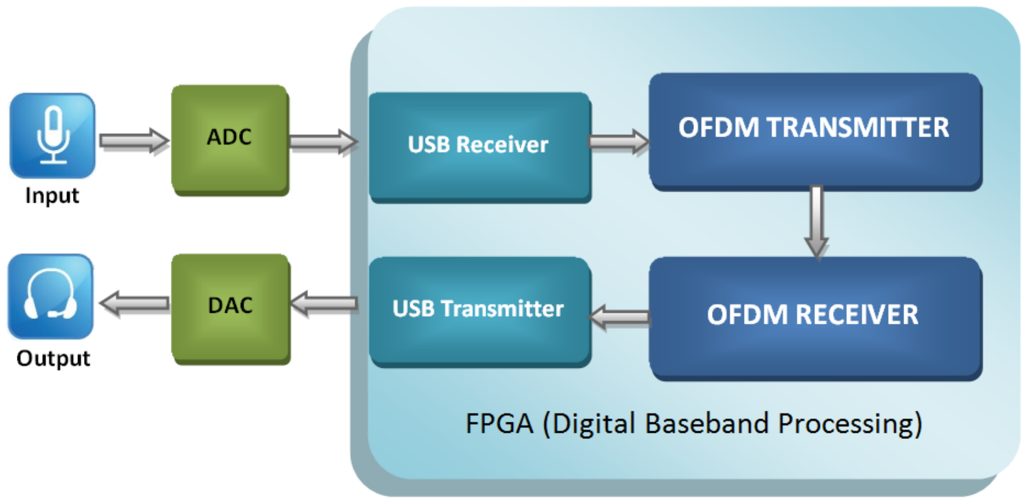


Figure 1.6 Top Level Architecture of the Proposed OFDM System

1.9 PROJECT DESIGN FLOW

The design procedure involves creation of a top level design of the complete system; Determining the basic operation of each block and creating the appropriate logic; I/O integration of the various logic blocks; Description of design functionality using Verilog hardware description language; ISim and Modelsim is used to simulate the design functionality and to report errors in desired behavior of the design; FPGA bitstream file is fed to the hardware; Input is given to the system after digitizing through the ADC and the desired output is interpreted using DAC.

Figure 1.7 shows the complete project design flow sequentially in the form of different stages alongwith the softwares and hardwares employed at these stages.

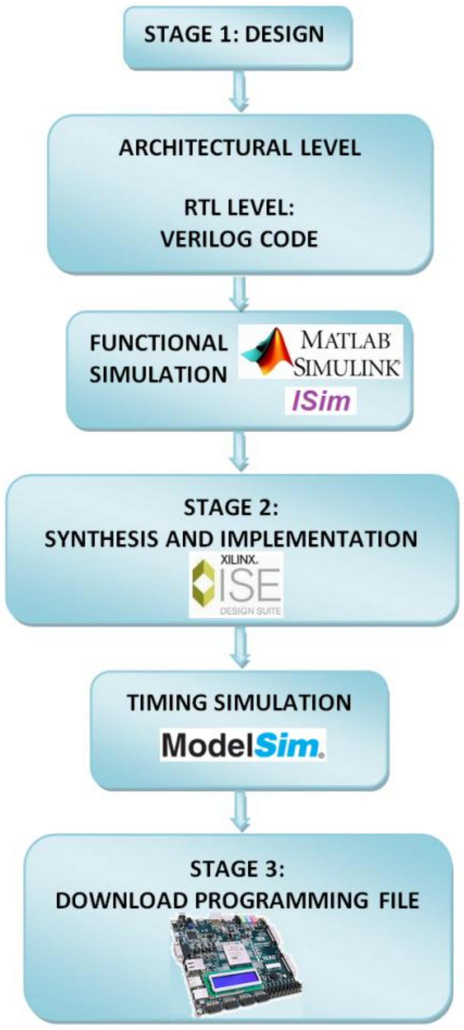


Figure 1.7 Project Design Flow

1.10. PROJECT SCOPE

The work of the project is entirely focused on the design and development of a baseband processing module. Coding is done in Verilog and OFDM blocks are implemented on FPGA kit. Forward Error Correction and noise immunity can well define the scope of this project. The results of programmed hardware have been tested by comparing with simulation model of Matlab Similink.

CHAPTER 2

LITERATURE SURVEY

2.1 EVOLUTION OF OFDM

OFDM can be viewed as a collection of transmission techniques. When this technique is applied in wireless environment, it is referred to as OFDM. In the wired environment, such as asymmetric digital subscriber lines (ADSL), it is referred as discrete multi tone (DMT). In OFDM, each carrier is orthogonal to all other carriers. However, this condition is not always maintained in DMT [11]. OFDM is an optimal version of multi carrier transmission schemes.

Frequency Division Multiplexing (FDM) is also a form of the multi-channel transmission. The use of Frequency Division Multiplexing (FDM) goes back over a long period of time, where more than one low rate signal, such as telegraph, was carried over a relatively wide bandwidth channel using a separate carrier frequency for each signal [2]. To facilitate separation of the signals at the receiver, the carrier frequencies were spaced sufficiently far apart so that the signal spectra did not overlap. Empty spectral regions between the signals assured that they could be separated with readily realizable filters. The resulting spectral efficiency was therefore quite low.

2.2 THE OFDM SYSTEM

A detailed explanation of the OFDM system was given in the previous chapter, in which different building blocks of an OFDM communication system were discussed. In 1971 Discrete Fourier Transform (DFT) was used in baseband modulation/demodulation in order to achieve orthogonality. Since DFT has heavy computational requirements, therefore, Fast Fourier

Transform (FFT) was utilized. For an N point discrete Fourier Transform the required number of computations is N^2, but that for FFT is Nlog (N), which is much lesser than DFT. In this way the problem of bandwidth inefficiency due to the placement of guard bands between sub-channels was solved and a new technique “Orthogonal Frequency Division Multiplexing” came into being.

As OFDM is a multi-carrier modulation technique, therefore, the input data is split and mapped onto different sub-carriers. Each carrier is modulated using one of the single carrier modulation techniques discussed above.

The OFDM system successfully avoids any inter-channel interference (ICI) because the carriers are kept orthogonal. In addition, a cyclic prefix (CP) is added before the start of each transmitted symbol to act as a guard period preventing inter-symbol interference (ISI), provided that the delay spread in the channel is less than the guard period [5]. This guard period is specified in terms of the fraction of the number of samples that make up a symbol.

2.3 ADVANTAGES AND DISADVANTAGES OF OFDM

Another advantage of OFDM is its resilience to Multipath, which is the effect of multiple reflected signals hitting the receiver. This results in interference and frequency-selective fading which OFDM is able to overcome by utilizing its parallel, slower bandwidth nature. This makes OFDM ideal to handle the harsh conditions of the mobile wireless environment. The introduction of cyclic prefix made OFDM system resistant to time dispersion [6]. OFDM symbol rate is low since a data stream is divided into several parallel streams before transmission. This makes the fading slow enough for the channel to be considered as constant during one OFDM symbol interval.

Cyclic prefix is a crucial feature of OFDM used to combat the inter-symbol interference (ISI) and inter-channel-interference (ICI) introduced by the multi-path channel through which the signal is propagated [2]. The basic idea is to replicate part of the OFDM time domain waveform from the back to the front to create a guard period. The duration of the guard period should be longer than the worst-case delay spread of the target multi-path environment. The use of a cyclic prefix instead of a plain guard interval, simplifies the channel equalization in the demodulator.

In wire system, OFDM system can offer an efficient bit loading technique [2]. It enables a system to allocate different number of bits to different sub channels based on their individual SNR. Hence, an efficient transmission can be achieved.

One of the major disadvantages of OFDM is its requirement for high peak-to average power ratio (PAPR) [2]. This put high demand on linearity in amplifiers.

Second, the synchronization error can destroy the orthogonality and cause interference. Phase noise error and Doppler shift can cause degradation to OFDM system [2]. A lot of effort is required to design accurate frequency synchronizers for OFDM.

OFDM’s high spectral efficiency and resistance to Multipath make it an extremely suitable technology to meet the demands of wireless data traffic. This has made it not only ideal for such new technologies like WiMAX and Wi-Fi but also currently one of the prime technologies being considered for use in future fourth generation (4G) networks.

2.4 APPLICATIONS OF OFDM

Initially, OFDM applications are scarce because of their implementation complexity. Now, OFDM has been adopted as the new European digital audio broadcasting (DAB) standard and for terrestrial digital video broadcasting (DVB) [7].

In fixed-wire applications, OFDM is employed in asynchronous digital subscriber line (ADSL) and high bit-rate digital subscriber line (HDSL) systems. It has been proposed for power line communications systems as well due to its resilience to dispersive channel and narrow band interference. It has been employed in WiMAX as well.

2.5 VERILOG HARDWARE DESCRIPTION LANGUAGE

Verilog HDL is one of the two most common Hardware Description Languages (HDL) used by integrated circuit (IC) designers. The other one is VHDL.

HDL allows the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are technology-independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

Verilog can be used to describe designs at four levels of abstraction [8]; Algorithmic level (much like c code with if, case and loop statements), Register transfer level (RTL uses registers connected by Boolean equations), Gate level (interconnected AND, NOR etc.), Switch level (the switches are MOS transistors inside gates).

The language also defines constructs that can be used to control the input and output of simulation. More recently Verilog is used as an input for synthesis programs which will generate a gate-level description (a netlist) for the circuit. Some Verilog constructs are not synthesizable. Also the way the code is written will greatly affect the size and speed of the synthesized circuit.

2.6 SYNTHESIS PROCESS IN VERILOG HDL

Synthesis is to construct a gate-level net list from a model of a circuit described in Verilog. The synthesis process is described in Figure 2.1.

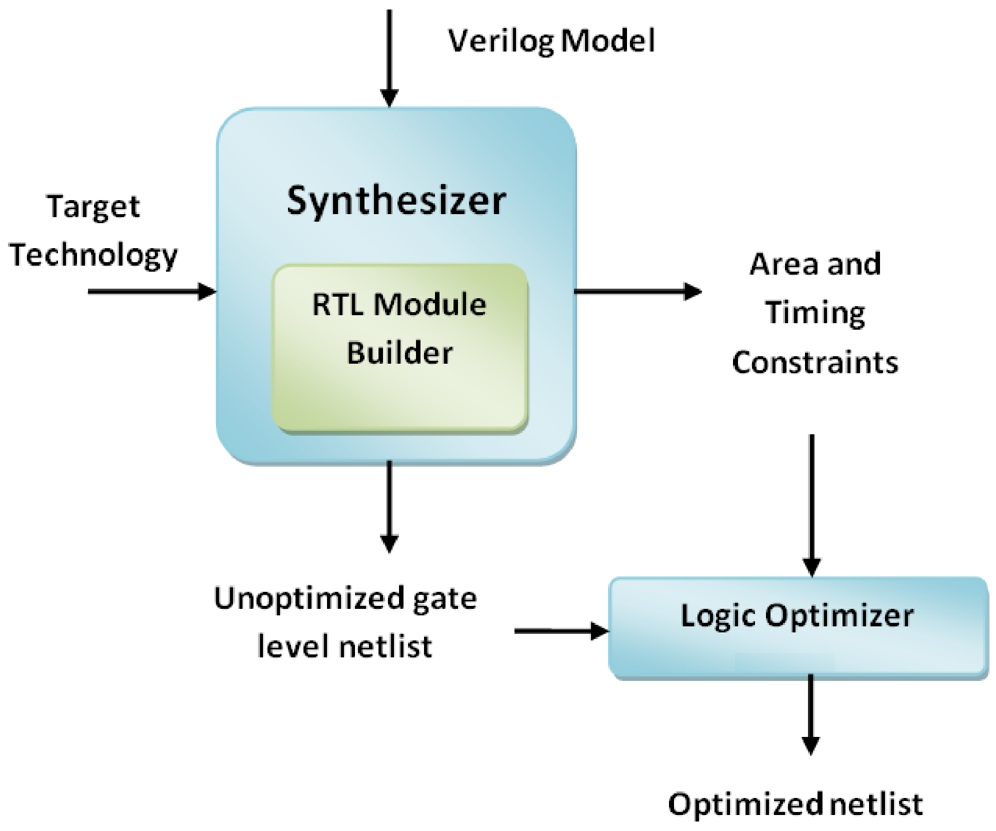


Figure 2.1 Synthesis Process in Verilog Environment

A synthesis program may generate an RTL net list, which consists of register-transfer level blocks such as flip-flops, arithmetic-logic-units and multiplexers interconnected by wires. All these are performed by RTL module builder. This builder is to build or acquire from a library predefined components, each of the required RTL blocks in the user specified target technology.

The synthesis process may produce an unoptimized gate level net list. A logic optimizer can use the produced net list and the constraint specified to produce an optimized gate level net list. This net list can be programmed directly into a FPGA chip.

CHAPTER 3

TRANSMITTER DESIGN AND IMPLEMENTATION

3.1 INTRODUCTION

The proposed OFDM system consists of an OFDM baseband transmitter and an OFDM baseband receiver. This chapter gives details on the complete architecture of the proposed design and elaborates further on the design and implementation of the transmitter portion of the project.

The transmitter gets its input from the Adept USB2. An input stream is sent as input to the transmitter that modulates the incoming stream by splitting it and putting it onto separate sub-carriers (64 in our case). The modulated data after passing through various blocks is given as input to the receiver and also sent back to the DAC (via USB port) for demonstration purposes.

3.2 OFDM HARDWARE ARCHITECTURE

Implementation of the proposed system has been done on Xilinx Virtex 5 LX50T. The USB port receiving module takes the serial stream from ADC and extracts the 8 bit payload by removing the start and stop bits. Figure 3.1 shows the format of data stream in serial communications.

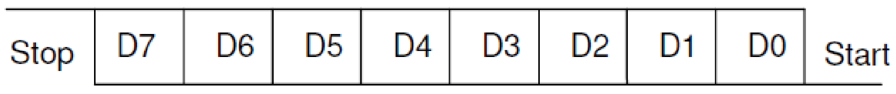


Figure 3.1 Serial Communication Format (8 Bit Data + Start Bit + Stop Bit)

The 1-byte data from the USB receiver is stored in a FIFO register. Data from the FIFO is given (bit by bit) to the transmitter module.

Figure 3.2 shows the various building blocks of the transmitter and depicts the hardware architecture of the project highlighting only the transmitter portion.

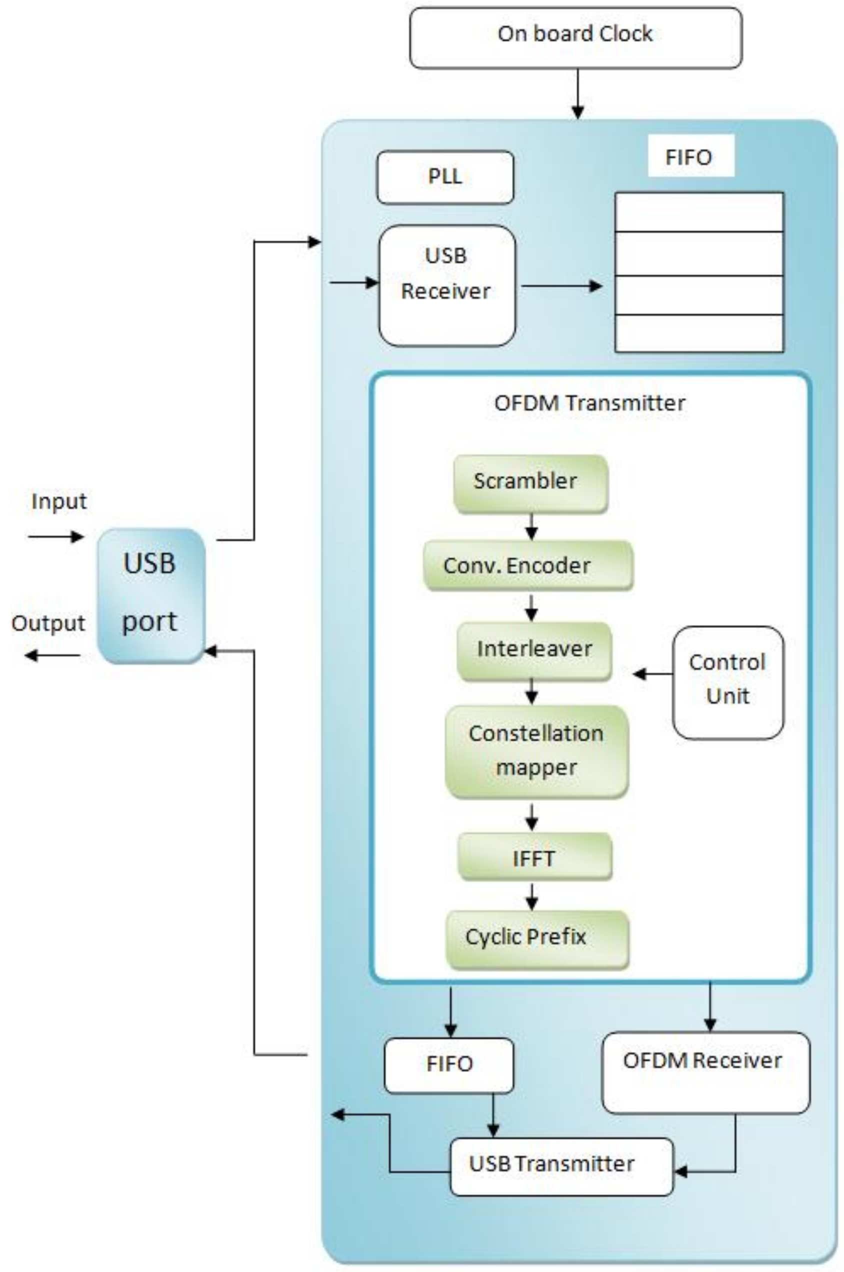


Figure 3.2 Complete Architecture of the Proposed OFDM Transmitter

The modulated output from the transmitter is fed into another FIFO, and then taken out into the transmitter (byte by byte) that prepares the data for serial transmission over the USB interface by adding start and stop bits. The baud rate on which the serial port is operating is 115.2 kbps.

There is a 50 MHz on-board clock source which in conjunction with the PLL core (provided with the Xilinx ISE software) can be used to produce any clock frequency. The output of the PLL then provides clock(s) to all the modules.

Figure 3.3 shows an I/O view of the proposed system and Table 3.1 gives a description of the input and output signals of the OFDM system.

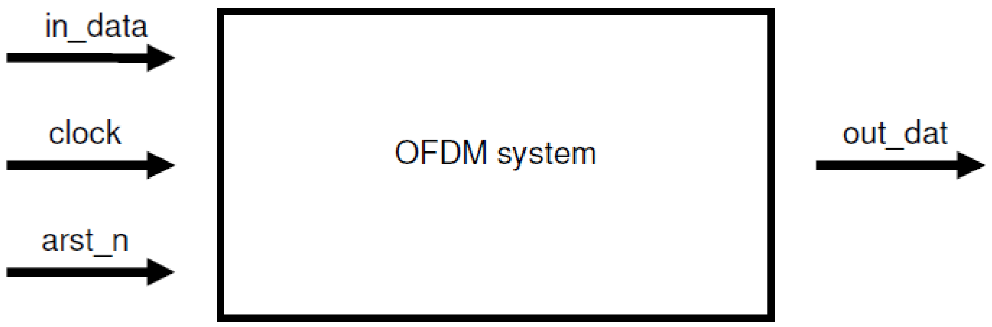


Figure 3.3 I/O View of the OFDM System

Table 3.1 shows the overall OFDM system’s input and output signals including the description of each.

Table 3.1 OFDM System Signal Description

Signal name	Type	Width	Description
in_data	Input	1	Data input to the OFDM system
Clock	Input	1	Clock signal (via 50 MHz on-board clock)
arst_n	Input	1	Asynchronous reset (asserted at negative edge)
out_data	Output	1	Demodulated output data

3.3 THE TRANSMITTER

Among the various components of the OFDM transmitter, the control unit synchronizes the operation of all the blocks in order to avoid any timing mismatches. Each one of these blocks will be discussed in detail in the subsequent sections.

The transmitter gets its input from the FIFO register one bit per clock cycle. This implies that the input to the transmitter is I bit wide. It is only when the FIFO is full that the transmitter starts extracting data from it. Similarly when the FIFO gets empty the transmitter stops taking data from it. Therefore, the transmitter makes use of certain control and status signals provided by the FIFO to determine when to ask the FIFO for data and when to stop taking input data.

In a similar fashion, the output of the transmitter is also stored in a FIFO register. In order for this FIFO to determine when to start storing output data from the transmitter, the transmitter provides a status signal that tells this FIFO that data is present on the output lines.

Figure 3.4 shows the I/O diagram for the transmitter and Table 3.2 gives the description of the signals in and out of the transmitter.

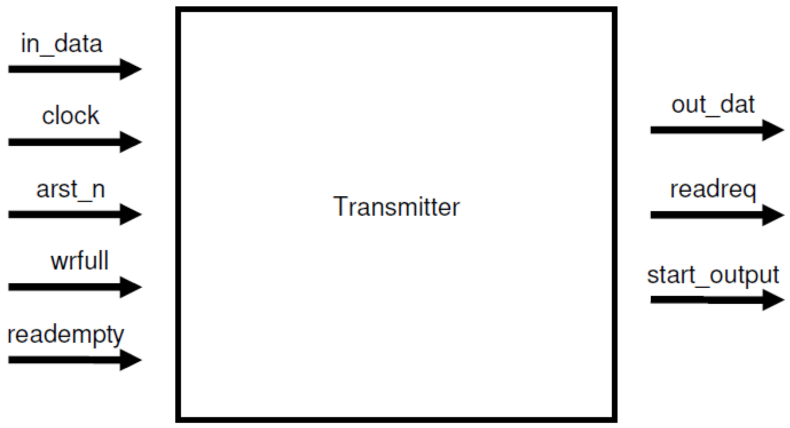


Figure 3.4 I/O Diagram of the Transmitter

Table 3.2 shows the transmitter's input and output signals alongwith their width and description.

Table 3.2 Transmitter Signal Description

Signal name	Type	Width	Description
in_data	Input	1	Data input to the transmitter
Clock	Input	1	Clock – 20 MHz (Output of PLL)
arst_n	Input	1	Asynchronous reset (asserted at negative edge)
Wrfull	Input	1	FIFO status signals - asserted when FIFO is full
Readempty	Input	1	FIFO status signal – asserted when FIFO is empty
out_data	Output	48	Modulated data coming out of the transmitter
Readreq	Output	1	FIFO control signal – requests data from FIFO (transmitter asserts this signal when the FIFO is full)
start_output	Output	1	Asserted when there is data present on the out_data lines

3.4 FIFO

First In First Out is a popular data structure (also known as queue) that is used for buffering in order to provide flow control. The FIFO architecture has been obtained from Xilinx's IP Core Generator Wizard. This parameterized IP Core Generator allows creating FIFOs of any width and depth with various options of control and status signals. Using technology specific modules allows for quick prototyping of the design. The appropriate parameters have been provided and the IP Core has been interfaced with the design. The hardware implementation details of all the blocks are discussed accordingly.

3.5 SCRAMBLER

A scrambler (often referred to as a randomizer) is a device that manipulates a data stream before transmitting. The purpose of scrambling is to eliminate the dependence of a signal's power spectrum upon the actual transmitted data and making it more disperse to meet maximum power spectral density requirements, because if the power is concentrated in a narrow frequency band, it can interfere with adjacent channels [10].

3.5.1 DESIGN OF SCRAMBLER

Figure 3.5 shows the input/output parameters of the Scrambler. Input bus is 1 bit wide and arst_n is the asynchronous reset input. A negative edge on the arst_n input resets the Scrambler. A bit is latched in at the positive edge of the clock. See Table 3.3 for a description of the signals.

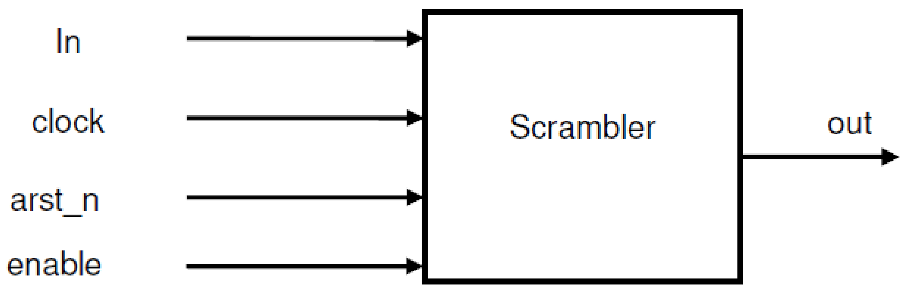


Figure 3.5 Scrambler I/O Diagram

Table 3.3 Scrambler Signals Description

Signal name	Type	Width	Description
in	Input	1	Input data to the transmitter
clock	Input	1	Positive edge clock
arst_n	Input	1	Asynchronous reset (Negative edged)
enable	Input	1	If high, input is present on the line in
out	Output	1	Output scrambled data

Scramblers can be implemented using a Linear Feedback Shift Register (LFSR) [10]. An LFSR is a simple register composed of memory elements (flip-flops) and modulo-2 adders (i.e. XOR gates). Feedback is taken from two or more memory elements, which are XOR-ed and fed back to the first stage (memory element) of the Linear Feeback Shift Register (LFSR). In the proposed design, a standard 7 bit scrambler has been used to randomize the incoming bits. An initial seed value is stored in the LFSR when arst_n is asserted; this value may be any random bit string except for all zeroes or all ones. If the initial seed contains all zeroes or all ones then the LFSR is locked in a state where every output value is same i.e. either one or zero.

Figure 3.6 shows the logic diagram of the scrambler showing its basic construction. It comprises of a feedback output, which is actually the modulo-2 added result of the contents of memory elements 4 and 7, then it is XORed with the input and the result obtained is designated as the output and is also shifted into the first stage using LFSR. These memory elements are actually flip-flops (D-flip flops are used here); with the output of each flip flop acting as the input for the next flip flop.

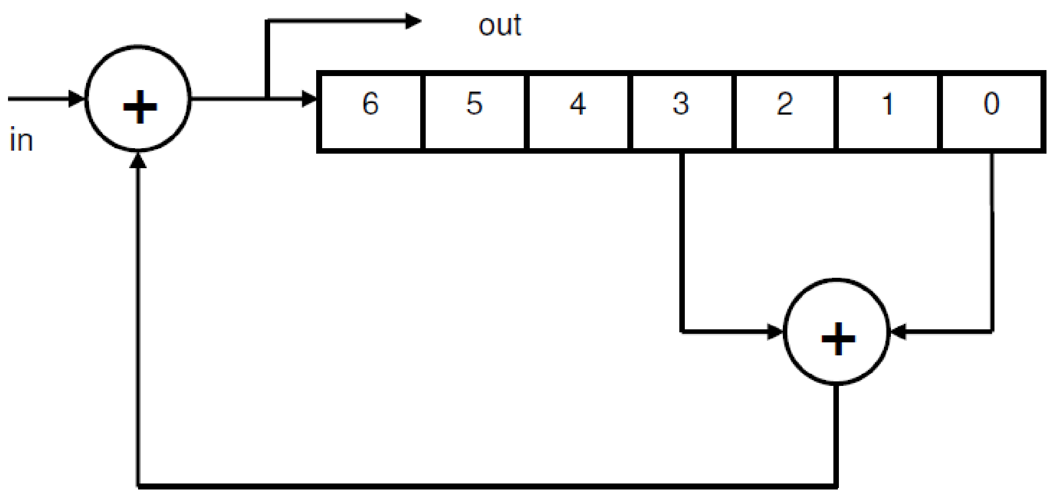


Figure 3.6 Scrambler Logic Diagram

Figure 3.7 shows, in detail, the circuit diagram of the scrambler. We can see that the reset (arst_n) is asserted on the negative edge, this is shown by the bubble at the reset pins of the flip-flops.

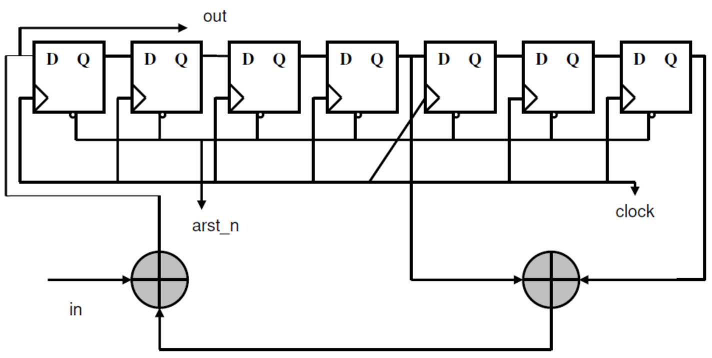


Figure 3.7 Circuit Diagram of Scrambler

3.6 CONVOLUTIONAL ENCODER

Convolutional coding is part of the Forward Error Correction (FEC) done in communication systems. The purpose of forward error correction (FEC) is to improve the capacity of a channel by adding some carefully designed redundant information to the data being transmitted through the channel [11]. The process of adding this redundant information is known as channel coding [11]. Convolutional codes operate on serial data, one or a few bits at a time. There are a variety of useful Convolutional codes, and a variety of algorithms for decoding the received coded information sequences to recover the original data. Convolutional codes are usually described using two parameters: the code rate and the constraint length. The code rate, m/n, is expressed as a ratio of the number of bits into the Convolutional encoder (m) to the number of channel symbols output by the Convolutional encoder (n) in a given encoder cycle. The constraint length parameter, K, denotes the "length" of the Convolutional encoder, i.e. how many k-bit stages are available to feed the combinatorial logic that produces the output symbols. Convolutional codes are often used to improve the performance of digital radio, mobile phones, and satellite links. In the proposed design a Convolutional encoder with a code rate of ½ has been chosen i.e. m=1 and n=2. A constraint length of 7 is kept because it is standard and its decoding can be efficiently done using the popular "Viterbi Decoding Algorithm".

3.6.1 ENCODER DESIGN

Figure 3.8 shows the I/O parameters of the Convolutional Encoder. Input bus is 1 bit wide and arst_n is the asynchronous reset input. A negative edge on the arst_n input resets the encoder. A bit is latched in at the positive edge of the clock. For every input bit there is a two bit wide output designated by even and odd. Table 3.4 gives description of the I/O signals of the Convolutional Encoder.

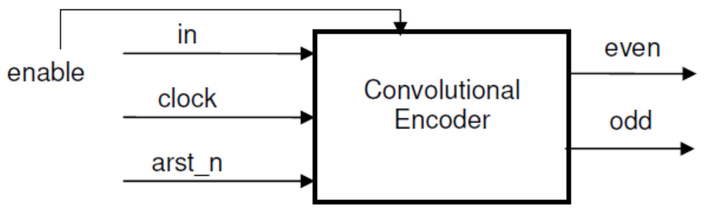


Figure 3.8 Convolutional Encoder I/O Diagram

Table 3.4 Signals Description for the Convolutional Encoder

Signal name	Type	Width	Description
in	Input	1	Input data to the Convolutional Encoder
clock	Input	1	Positive edge clock
arst_n	Input	1	Asynchronous reset (Negative edged)
Enable	Input	1	If high, input is present on the line in
even	Output	1	Least significant bit of the output
odd	Output	1	Most significant bit of the output

Convolutional Encoder can be implemented using either a shift register or by using "Algorithmic State Machine" [16]. However, a shift register gives an easy to implement and area efficient solution. For the configuration of m=1, n=2 and k (constraint length) =7, Figure 3.9 shows how the Convolutional encoder is implemented in the proposed design using a shift register. Initially all zeroes are stored in the register. When the first input bit arrives it is shifted into the register from left and the 2 bit output appears on the lines designated as even and odd.

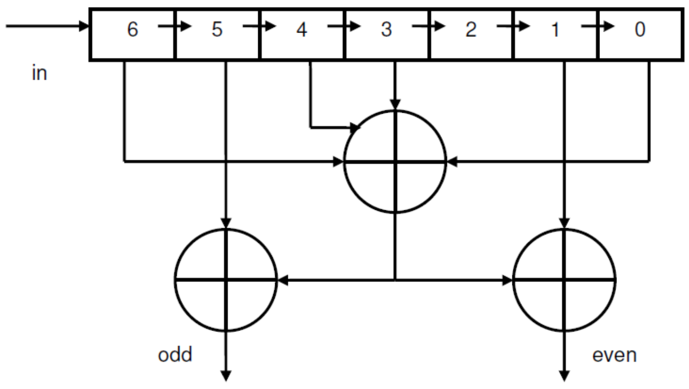


Figure 3.9 Convolutional Encoder Circuit Diagram

The even output is generated by adding the contents of 1st, 0, 3rd, 4th and 6th stages of the shift register, whereas the odd output is generated by adding the 5th, 0, 3rd, 4th and 6th stages of the register. This addition is modulo-2 addition carried out through XOR gates (modulo-2 addition is basically a XOR operation). Just like the Scrambler the memory elements here are D-flip-flops as well.

3.7 INTERLEAVER

Interleaving is mainly used in digital data transmission technology, to protect the transmission against burst errors. These errors overwrite a lot of bits in a row, but seldom occur. The device that performs interleaving is known as Interleaver.

Conceptually, the in-coming bit stream is re-arranged so that adjacent bits are no more adjacent to each other. Actually the data is broken into blocks and the bits within a block are re-arranged. In the proposed design, a block consists of 64 symbols (128 bits). Number of bits in each symbol depends upon the corresponding single-carrier modulation technique to be applied to produce that symbol.

Two memory elements (usually RAMs) are used. In the first RAM the incoming block of bits is stored in sequential order. This data from the first RAM is read out randomly (using an algorithm) so that the bits are re-arranged and stored in the second RAM and then read out.

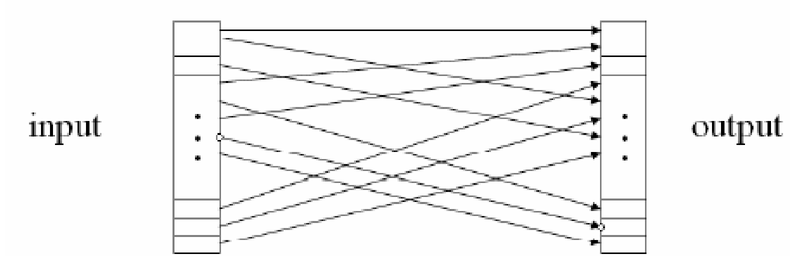


Figure 3.10 Interleaving Concept

As mentioned above that the incoming bit stream is broken into blocks, when interleaving in the OFDM system the block size should be equal to the size of an OFDM symbol. Since there are 64 sub-carriers and each sub-carrier is modulated using QPSK, therefore in one OFDM symbol there would be 128 bits. Hence, the job of the interleaver would be to re-arrange the bits within the OFDM symbol.

3.7.1 INTERLEAVER DESIGN

The function that the interleaver has to perform is to read 128 bits, re-arrange them and read them out. This can be accomplished by using RAMs for temporarily storing the bits and then the bits can be read out from the RAMs in the desired order. Remember that the block before the interleaver is the Convolutional Encoder that gives an output of two bits. Therefore the input bus of the interleaver should be two bits wide. This can be seen in Figure 3.11 which shows the top level architecture of the interleaver.

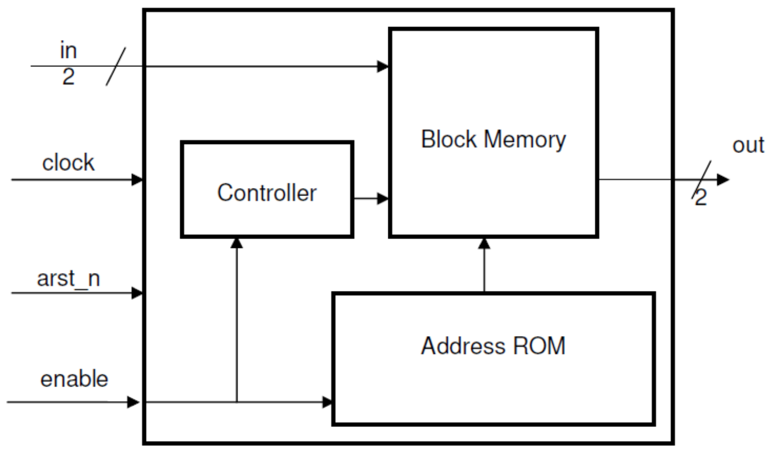


Figure 3.11 Interleaver I/O Diagram (A Top Level Architecture)

Table 3.5 shows the input and output signals of the interleaver also describing their function and width.

Table 3.5 Signals Description for Interleaver

Signal name	Type	Width	Description
in	Input	2	Input data to the Interleaver
clock	Input	1	Positive edge clock
arst_n	Input	1	Asynchronous reset (Negative edged)
enable	Input	1	If high, input is present on the line in
out	Output	2	Output of the Interleaver

Note that the input and output buses are two bits wide. The three building blocks of the interleaver are; block memory, controller and address ROM. The block memory contains the memory elements necessary to store the incoming block of data. There are a total of four memory elements; each is a 64x1 RAM. Four RAMs are used in order to achieve pipelined operation. Two of these RAMs are used for writing a block while another block is being read out from the other two RAMs. In this way the RAMs are alternately switched between reading and writing modes. Hence, reading and writing is done simultaneously without any latency. The configuration of each of these RAMs is such that two bits are written at a time in two memory locations and one bit is read at a time. Recall that input to the interleaver is two bit wide, therefore that takes care of it. Two memories each 64x1 is used instead of a single memory 128x1 because two bits are to be read at a time. While writing a block of data (i.e. 128 bits), 16 bits are alternately written into the 64x1 RAMs. That is to say that first 16 bits are written to the first RAM, next 16 to the second RAM, next 16 again to the first RAM and so on. This is done in order to keep the two bits that have to be read (in desired order) in separate RAMs.

The job of the controller is to guide the incoming block of data to the correct memory blocks, to switch the RAMs between reading and writing modes, and to switch between the two RAMs for 16 alternate bits in writing mode. This is done by using counters. The address ROM is basically a 64x6 ROM that stores read addresses for the RAMs. Note that a single ROM is enough for the four RAMs. This is because only two RAMs at a time are in the read mode and the two bits that are read out of the two RAMs are in the same memory locations as per the design. Each location of the ROM is 6 bits wide because a 6-bit address is required to read from a RAM having 64 locations.

Figure 3.12 shows the circuit diagram of interleaver. Counter1 and Counter2 provide for the write addresses for the four RAMs 1A, 2A, 1B and 2B. Counter C is a 3-bit counter that controls switching between either RAM 1A and RAM 2A or RAM 1B and RAM 2B depending upon which RAMs are in write mode. Counter1 and Counter2 are 5-bit counters after every 8th count control switches to either Counter1 or Counter2; this is controlled by Counter C.

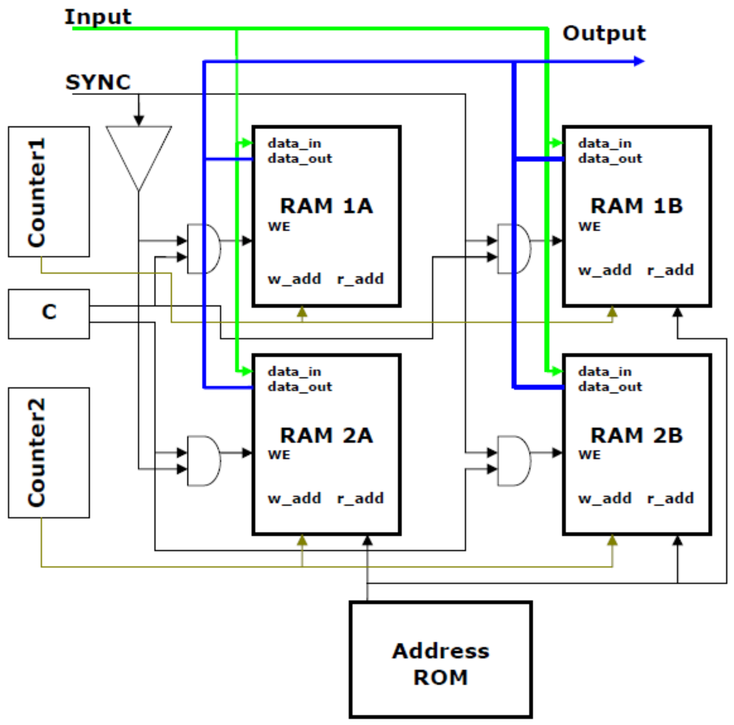


Figure 3.12 Interleaver Circuit Diagram

The SYNC signal decides which RAMs must write and which should read. When SYNC is 0 RAM 1A and RAM 2A are in write mode and RAM 1B and RAM 2B in read mode, opposite is the case when SYNC is high. For the first data block SYNC remains 0 and therefore the block is written to RAM 1A and

RAM 2A. When the last bit of the block is written SYNC goes high and RAM 1A and RAM 2A go in read mode, whereas RAM 1B and RAM 2B go in write mode and the next block is written to these blocks. At the same time the previous is read out of RAMs 1A and 2A in the desired order.

3.8 CONSTELLATION MAPPER

Constellation Mapper maps the incoming bits onto separate sub-carriers. In the proposed design there are 64 sub-carriers and each of them is modulated using QPSK, therefore the function of Constellation Mapper would be to map every two bits on a single carrier, because in QPSK two bits make up one symbol.

Figure 3.13 shows the constellation diagram of QPSK. Mapping of bits on constellation points is done in accordance with gray code so that adjacent constellation points may have just one bit different. Table 3.7 shows the data bits and the corresponding constellation points.

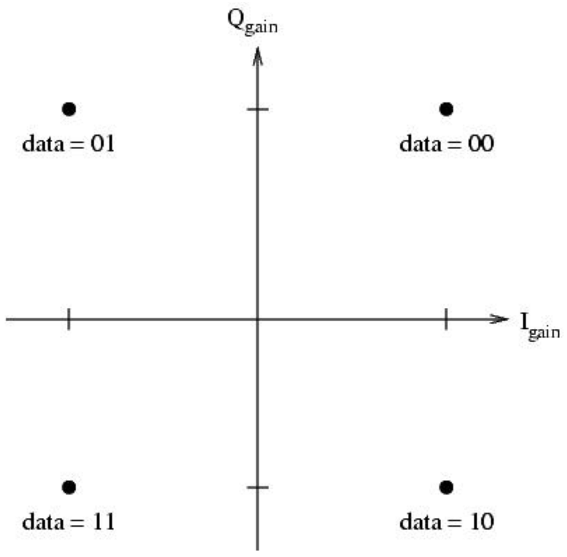


Figure 3.13 QPSK Constellation Mapper

Table 3.6 shows the data bits and the corresponding constellation points to which they are mapped.

Table 3.6 Mapping of Bits to Constellation Points

Data bits	Constellation point
00	0.707 + j0.707
01	- 0.707 + j0.707
10	0.707 - j0.707
11	- 0.707 - j0.707

The block before Constellation Mapper is the Interleaver which gives an output of two bits per clock cycle. Therefore, two bits are mapped to a constellation point every clock cycle.

3.8.1 DESIGN OF CONSTELLATION MAPPER

A ROM is used to store the constellation points. Each constellation point is represented by 48 bits in binary. In these 48 bits, the most significant 24 bits represent the real part and the least significant 24 bits represent the imaginary part. In both the real and imaginary parts the most significant 8 bits are the integer part and the least significant 16 bits represent the fractional part. 2's complement notation has been used to represent negative numbers. The size of ROM is 4x48.

The incoming input bits (2 bits) act as address for the ROM. Table 3.7 shows the ROM contents at each address location. Each of these values in the ROM is a constellation point corresponding to the data bits which here act as addresses for the ROM.

Table 3.7 Contents of the ROM (In Constellation Mapper)

Address (Binary)	Contents (HEX)
00	00B50400B504
01	FF4AFC00B504
10	00B504FF4AFC
11	FF4AFCFF4AFC

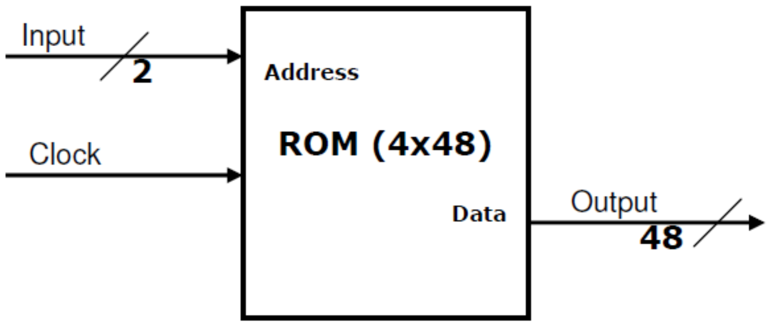


Figure 3.14 Constellation Mapper

Figure 3.14 shows the circuit of a constellation Mapper. It contains nothing but a ROM. Note that the input is two bits wide and the output is 48 bits wide. For a description of the I/O signals of the constellation mapper see Table 3.8.

Table 3.8 Signals Description for Interleaver

Signal name	Type	Width	Description
in	Input	2	Input data to the constellation mapper (acting as address for the above shown ROM)
clock	Input	1	Positive edge clock
out	Output	48	Output of the constellation mapper (representing 48 bit complex number)

3.9 INVERSE FAST FOURIER TRANSFORM

In 1971 Discrete Fourier Transform (DFT) was used in baseband modulation/demodulation in order to achieve orthogonality [14]. Since DFT has heavy computational requirements, therefore, Fast Fourier Transform (FFT) was utilized. For an N point discrete Fourier Transform the required number of computations is N(N-1), but that for FFT/IFFT is Nlog (N), which is much lesser than DFT.

The FFT/IFFT operates on finite sequences. Waveforms which are analog in nature must be sampled at discrete points before the FFT/IFFT algorithm can be applied.

The Discrete Fourier Transform (DFT) operates on sample time domain signal which is periodic. The equation for DFT is:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi k/N} \qquad 3.11$$

X(k) represents the DFT frequency output at the k-the spectral point where k ranges from 0 to N-1. The quantity N represents the number of sample points in the DFT data frame. The quantity x(n) represents the nth time sample, where n also ranges from 0 to N-1. In general equation, x(n) can be real or complex.

The corresponding inverse discrete Fourier transform (IDFT) of the sequence X(k) gives a sequence x(n) defined only on the interval from 0 to N-1 as follows:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} x(k)e^{\frac{j2\pi k}{N}} \qquad 3.12$$

The DFT equation can be re-written into:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad 3.13$$

The quantity W_N^{nk} can be defined as:

$$W_N^{nk} = e^{-j2\pi k/N} \qquad 3.14$$

This quantity is called Twiddle Factor. It is the sine and cosine basis function written in polar form [15]. Examination of the equations reveals that the computation of each point of DFT requires the following: (N-1) complex multiplication, (N-1) complex addition (first term in sum involves $e^{j0} = 1$). Thus, to compute N points in DFT require N(N-1) complex multiplication and N(N-1) complex addition.

As N increases, the number of multiplications and additions required is significant because the multiplication function requires a relatively large amount of processing time even using computer. Thus, many methods for reducing the number of multiplications have been investigated over the last 50 years [16].

3.9.1 RADIX-2^2 ALGORITHM

When the number of data points N in the FFT/IFFT is a power of 4 (i.e., N = 4^v), we can, of course, always use a radix-2 algorithm for the computation. However, for this case, it is more efficient computationally to employ a radix-r FFT algorithm. In the decimation-in-frequency algorithm, the outputs or the frequency domain points are regrouped or subdivided. Consider the FFT equation:

$$X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi k/N} \qquad 3.15$$

As an example we consider N=16. We split or decimate the N-point input sequence into four subsequences, x(4n), x(4n+1), x(4n+2), x(4n+3), n = 0, 1, ... , N/4-1. Therefore, we get X(k), X(k+N/4), X(k+N/2) and X(k+3N/4). This process is called decimation in frequency. This decimation continues until each DFT becomes a 4 point DFT. Each 4 point DFT is known as a butterfly when we represent it graphically. Figure 3.15 shows a radix-4 FFT butterfly.

Since in the proposed design there are 64 sub-carriers so the input to FFT would be 64 complex numbers, hence a 64 point FFT would be required.

For a 4^n point FFT n stages are required and N/4 4 point DFTs per stage. Therefore in our case there would be 3 stages ($64 = 4^3$) and 16 4 point DFTs per stage or we can say 16 butterflies pre stage.

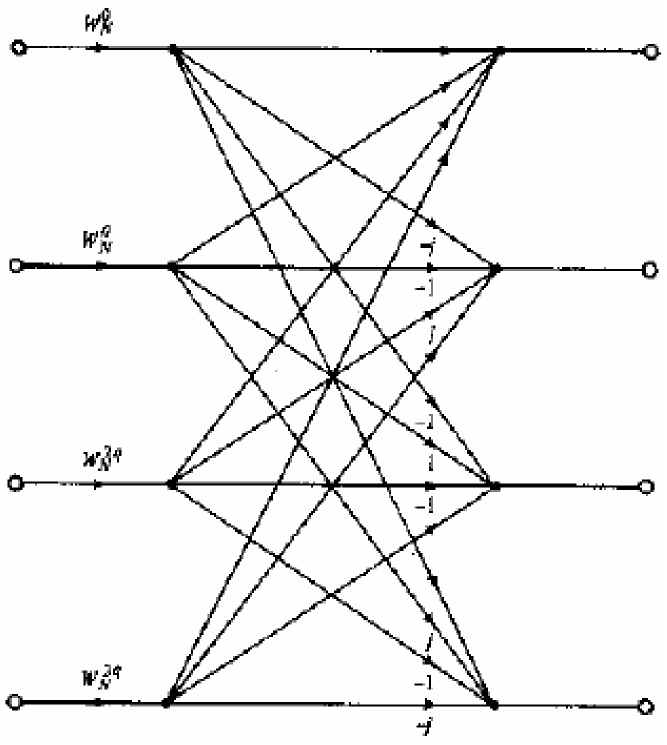


Figure 3.15 Radix-4 FFT Butterfly

In the decimation-in-frequency FFT algorithm, the outputs are decimated; therefore, inputs to the FFT are given in the actual order [17]. In this way we get the output in a rearranged order.

In the proposed design radix-2^2 DIT FFT algorithm is targeted because its butterfly is simple like that of radix 2 and no. of complex multiplications are less like radix 4. Figure 3.16 shows a radix 2 butterfly, its simplicity speaks for itself.

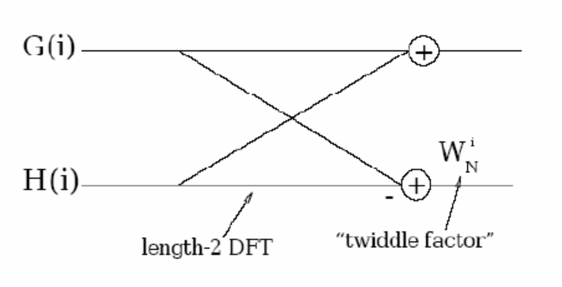


Figure 3.16 Radix-2 FFT Butterfly

In the radix-2^2 algorithm, a radix-4 butterfly is created using two radix-2 butterflies. The benefit of using the radix 2 algorithm is the ease of controlling the butterfly due to its simplicity and the decreased number of stages and complex multipliers.

3.9.2 IFFT DESIGN

From here on whenever I mention FFT, it will incorporate both IFFT and FFT. Basically there are two ways to implement FFT in hardware, one is using pipelined architecture and the other is using memory-based architecture. The former requires less hardware resources and hence occupies less area, but requires greater number of clock cycles. On the other hand in the memory-based architecture more hardware resources are required but it takes less number of clock cycles. In the proposed design pipelined architecture has been chosen in order to make the FFT design area efficient.

Additionally, fixed point FFT implementation has been carried out to avoid any overflows resulting from the complex multiplications.

Figure 3.17 shows the I/O diagram of IFFT and description of the I/O parameters is given in Table 3.9.

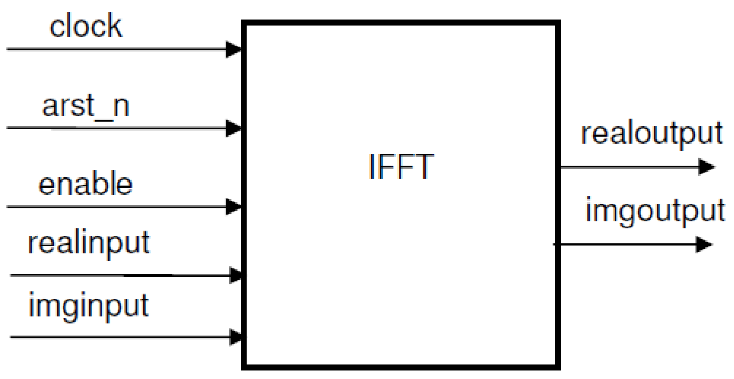


Figure 3.17 IFFT I/O Diagram

Table 3.9 Signals Description for IFFT

Signal name	Type	Width	Description
arst_n	Input	1	Asynchronous reset (negative edged)
clock	Input	1	Positive edged clock
enable	Input	1	When high data is present on the realinput and imginput lines
realinput	Input	24	Real part of the input complex number
imginput	Input	24	Imaginary part of the input complex number
realoutput	Output	24	Real part of the output complex number
imgoutput	Output	24	Imaginary part of the output complex number

Complex data is fed in one data-point per clock cycle. The enable signal is asserted the clock cycle previous to presenting the first data-point.

Figure 3.18 is a block diagram of a 64-point Radix-22 fixed-point FFT example. The module consists of six radix-2 butterflies, shift registers associated with each butterfly, two complex multipliers, two twiddle factor generators, and a controller that provides the control signals. The feedback shift registers vary in length from 1 to 32-bits, and are labeled accordingly.

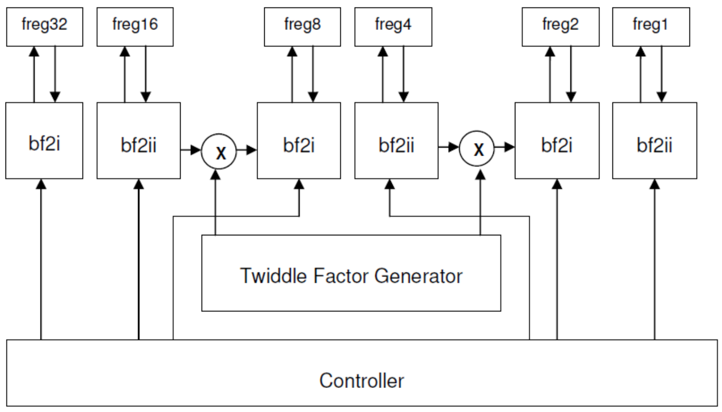


Figure 3.18 Architecture of a 64-Point-2^2 FFT

Each group of two butterflies, consisting of a bf2i and a bf2ii, together emulate a radix-4 butterfly. Figure 3.19 shows the internals of each and how they are connected together.

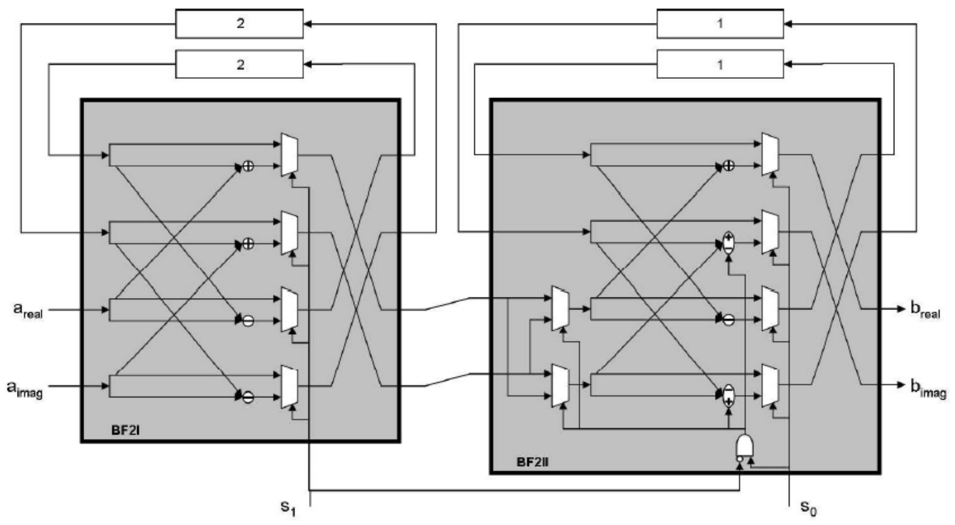


Figure 3.19 Bf2i and Bf2ii Radix 2 Butterflies

These modules operate on a principal known as Single-path Delay Feedback. The FFT Radix-2 butterfly must have two inputs in order to produce the next FFT intermediate value, but the data in our scenario is available only in a serial mode. The SDF mechanism provides a solution where the first input is delayed until the second input is presented, after which the calculation can proceed. Both the bf2i and bf2ii modules accomplish this by multiplexing the first input to a shift register of sufficient length so that that data-point is present at the butterfly input when the second data-point appears. A counter provides the control signals for these multiplexers, which are internal to the butterfly modules.

The counter additionally provides signals to the bf2ii for switching the adder operations, and swapping the real and complex input wires. These mechanisms effect a multiplication of the input by j.

In order to avoid overflow, the data set is scaled down as it propagates through the pipeline. The FFT operation consists of a long series of summations, and thus either the dynamic range of the numerical presentation must be large (floating-point of block floating-point), or the numerical data must be scaled down. Since the module is fixed point, the latter strategy is used.

3.10 CYCLIC PREFIX ADDER

Cyclic prefix is basically a replica of a fractional portion of the end of an OFDM symbol that is placed at the beginning of the symbol. It completely removes inter-symbol interference that can occur due to Multipath. Cyclic prefix is effective only if its duration is greater than the delay spread.

3.10.1 DESIGN OF CYCLIC PREFIX ADDER

The architecture of cyclic prefix adder simply consists of an address ROM that stores addresses, a RAM to store incoming data in sequential order and a counter that provides read addresses to the RAM. Figure 3.20 shows the top-level architecture of the cyclic prefix adder.

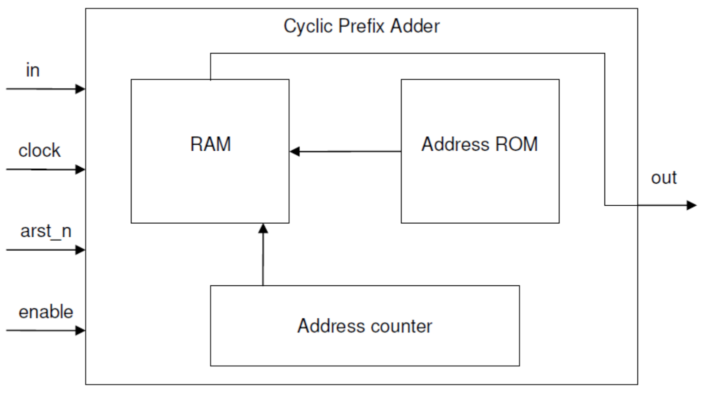


Figure 3.20 Top Level Architecture of Cyclic Prefix Adder

Refer to Table 3.10 for the I/O signal description for the Constellation mapper.

Table 3.10 Signals Description for Constellation Mapper

Signal name	Type	Width	Description
arst_n	Input	1	Asynchronous reset (negative edged)
clock	Input	1	Positive edged clock
enable	Input	1	Enable for Constellation Mapper
in	Input	48	Input complex number
out	Output	48	Output complex number

In the proposed design, the last eight symbols (complex numbers) of the OFDM symbol are replicated at the beginning of the symbol, therefore a total of 72 (64 + 8) symbols are actually transmitted.

CHAPTER 4

RECEIVER DESIGN AND IMPLEMENTATION

4.1 INTRODUCTION

This chapter gives detailed description about the implementation of the receiver part of the project. The receiver has been implemented on the same Virtex-5.

The OFDM receiving unit receives its input directly from the transmitter whenever its output is available. The receiver follows an exact reverse procedure of which was followed in the transmitter. It receives the complex (modulated) output points and performs demodulation and recovers the original bits sent to the transmitter. These bits are forwarded to the peripheral device (which includes DAC) to produce the same audio signal that was given at input for demonstration purpose.

4.2 THE RECEIVER

The receiver part consists of various blocks that perform the reverse operation as compared to the transmitter so that the same information is received at the output end as is given to the input end.

We can see that there are no control or status signals to or from a FIFO; the reason is that the modulated data, from the transmitter, is directly fed to the receiver as input.

I/O diagram of the receiver module showing the simplified flow is given in Figure 4.1. Clock signal is there to provide synchronization between the transmitter and receiver module.

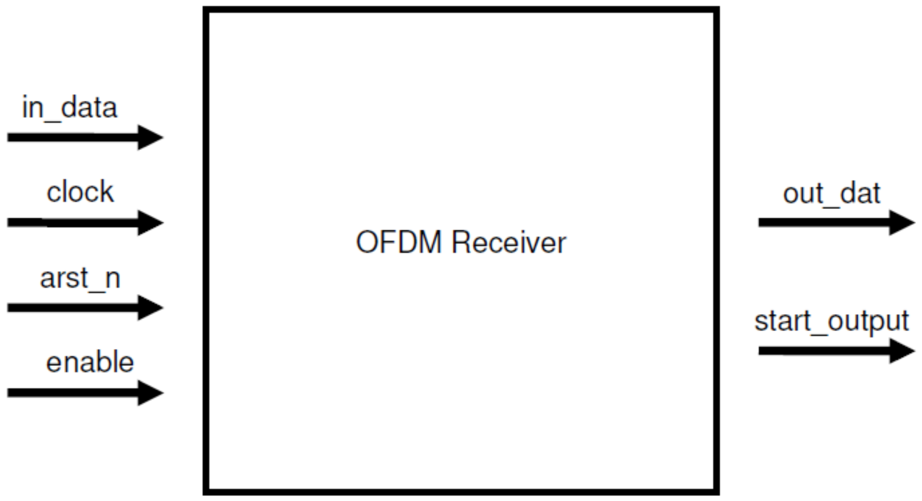


Figure 4.1 I/O Diagram of the OFDM Receiver

Description of the input and output signals involved in OFDM Receiver and their width is given in Table 4.1.

Table 4.1 OFDM Receiver Signal Description

Signal name	Type	Width	Description
in_data	Input	48	Data input to the Receiver
Clock	Input	1	Clock – 20 MHz (output of PLL)
arst_n	Input	1	Asynchronous reset (asserted at negative edge)
enable	Input	1	When asserted data is present on the in_data lines
out_data	Output	1	Demodulated data coming out of the receiver
start_output	Output		Asserted when there is data present on the out_data lines

Figure 4.2 shows the hardware architecture of the complete OFDM system highlighting the receiver part this time. The various blocks that constitute the receiver are shown. The receiver, just like the transmitter, operates at a clock frequency of 20 MHz provided by the on-board PLL.

Now the rest of the chapter is dedicated to the detailed description and design of the blocks inside the OFDM receiver as shown in Figure 4.2.

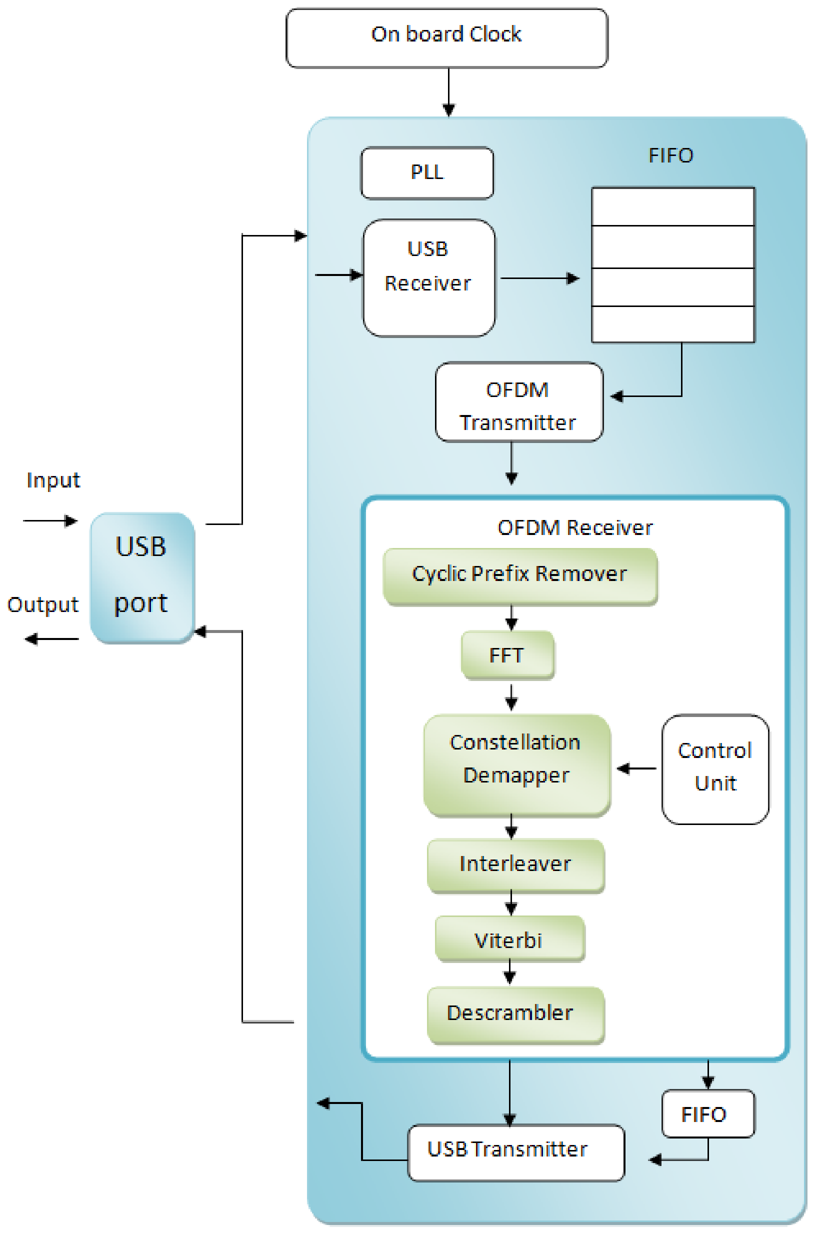


Figure 4.2 Complete Architecture of the Proposed OFDM System Receiver

4.3 CYCLIC PREFIX REMOVER

The cyclic prefix was added at the transmitting end in order to avoid inter-symbol interference, therefore during reception it must be eliminated for any further processing of the received signal. This is done by simply skipping the first eight sub-carriers in the received OFDM symbol. In hardware this is implemented in the control unit. The control unit only enables the next block (FFT) when the first eight bits of the received OFDM symbols have been skipped.

4.4 FAST FOURIER TRANSFORM

Details on FFT/IFFT algorithm and hardware implementation were given in the previous chapter. The only difference being that if it was given for IFFT (although FFT was mentioned at some places). In order to implement FFT in hardware the algorithm is same, only the difference is that the divider is removed and the real and imaginary parts at the input are swapped i.e. real becomes imaginary and imaginary becomes real. Same goes for the output i.e. real and imaginary parts at the output are swapped as well. Figure 4.3 depicts the scenario.

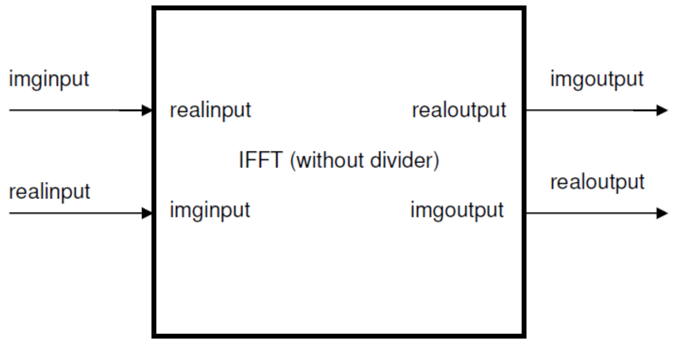


Figure 4.3 FFT

4.5 CONSTELLATION DE-MAPPER

The function of the constellation demapper is to map the QPSK symbols (complex numbers) coming from the output of FFT to the data points shown in the constellation diagram shown in Figure 4.4. Basically it is the inverse procedure of what was done in the constellation mapper at the transmitter.

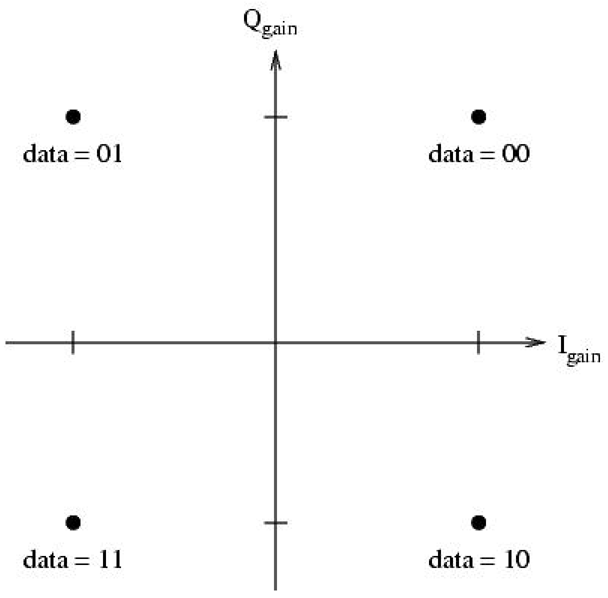


Figure 4.4 QPSK Constellation Diagram

4.5.1 DESIGN OF CONSTELLATION DE-MAPPER

The mapping of data points to QPSK symbols (as done in the transmitter) is shown in Table 4.2.

Table 4.2 Data Points Mapped to Constellation Points

Address (Binary)	Constellation Points	Constellation points (HEX)
00	0.707 + j0.707	00B50400B504
01	-0.707 + j0.707	FF4AFC00B504
10	0.707 – j0.707	00B504FF4AFC
11	-0.707 – j0.707	FF4AFCFF4AFC

Therefore, basically the incoming constellation points are mapped onto the data points as shown in this Table. Figure 4.5 shows the I/O diagram of the constellation demapper and Table 4.3 shows the description of the signals.

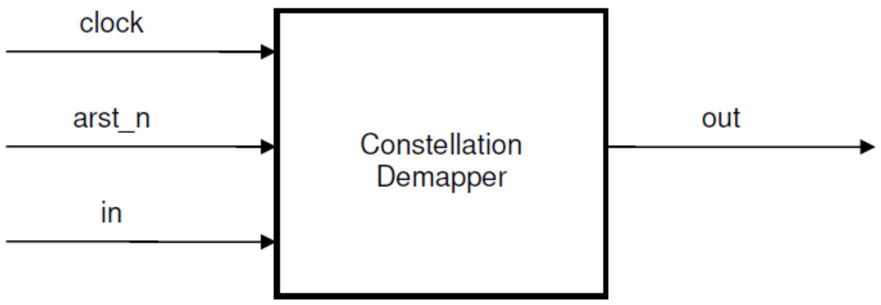
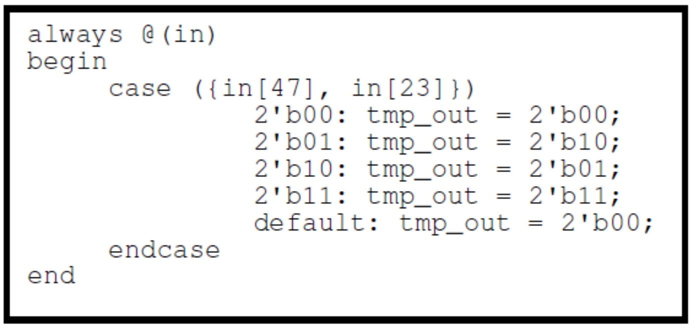


Figure 4.5 I/O Diagram of the Constellation Demapper

Table 4.3 Signals Description for Constellation De-Mapper

Signal name	Type	Width	Description
in	Input	48	Input constellation points
clock	Input	1	Positive edge clock
out	Output	2	Output data points corresponding to Table 4.2
arst_n	Input	1	Asynchronous reset (Negative edged)

Instead of going into the hardware architecture, the design is shown using the Verilog code in Figure4.6. A simple switch-case structure is used to construct the design.



```
always @(in)
begin
     case ({in[47], in[23]})
               2'b00: tmp_out = 2'b00;
               2'b01: tmp_out = 2'b10;
               2'b10: tmp_out = 2'b01;
               2'b11: tmp_out = 2'b11;
               default: tmp_out = 2'b00;
     endcase
end
```

Figure 4.6 Verilog Code Showing the Implementation of Constellation Demapper

4.6 DE-INTERLEAVER

In the previous chapter interleaving was defined as a process in which bits, within a block of 128 bits, are re-arranged in order to avoid burst errors. De-interleaving performs the inverse task. It re-arranges the interleaved bits into their original order.

Recall the row-column method of interleaving discussed in the previous chapter. Deinterleaving is done the same way, the difference being that the number of rows and the number of columns for de-interleaving are interchanged. For example if we perform interleaving on a block of 16 bits using a matrix with 8 rows and 2 columns, then the interleaved pattern can be de-interleaved using a matrix with 2 rows and 8 columns.

Hence the only difference in the hardware architectures of interleaver and de-interleaver is the contents of the address ROM, which actually provides the read addresses to the RAM that stores the data to be de-interleaved.

4.7 VITERBI DECODER

The Viterbi Decoder decodes Convolutional codes. We have used the Xilinx's Viterbi Decoder IP core in our design. Xilinx's Viterbi IP core is a parameterized IP core that is synthesizable and allows for parallel as well as hybrid implementation of the Viterbi decoder.

4.8 DESCRAMBLER

This block performs the reverse mechanism as compared to the scrambler. The same logic has been implemented in reverse order and has been tested in simulation as well as hardware.

4.8.1 DESCRAMBLER DESIGN

Figure 4.7 shows the input/output parameters of the Descrambler. A bit is latched in at the positive edge of the clock.

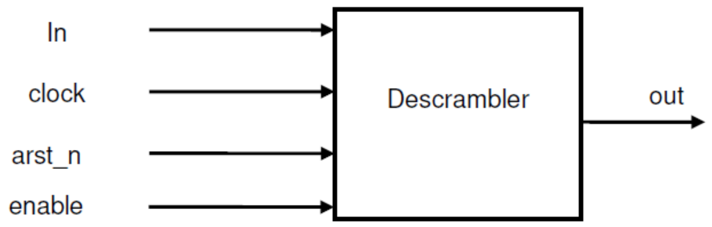


Figure 4.7 De-Scrambler I/O Diagram

See Table 4.4 for a description of the signals in descrambler's design.

Table 4.4 De-scrambler Signals Description

Signal name	Type	Width	Description
in	Input	1	Input data to the Descrambler
clock	Input	1	Positive edge clock
arst_n	Input	1	Asynchronous reset (Negative edged)
enable	Input	1	If high, input is present on the line in
out	Output	1	Output data

Figure 4.8 shows the logic diagram of the de-scrambler. Note that the structure is quite similar to that of the scrambler but is reversed.

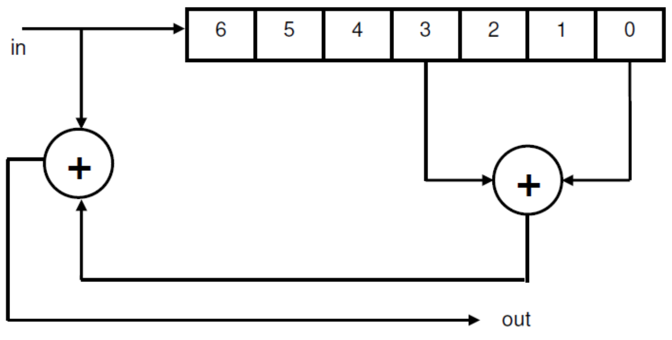


Figure 4.8 De-Scrambler Logic Diagram

CHAPTER 5

ANALYSIS AND CONCLUSION

5.1 INTRODUCTION

This chapter discusses the simulation results obtained from the Xilinx ISim and ModelSim as Simulator with random input samples and also the important synthesis results obtained from Xilinx ISE v12.3. The accuracy of the output of the transmitter has been verified by comparing with the output from MATLAB simulation shown in figure 5.1.

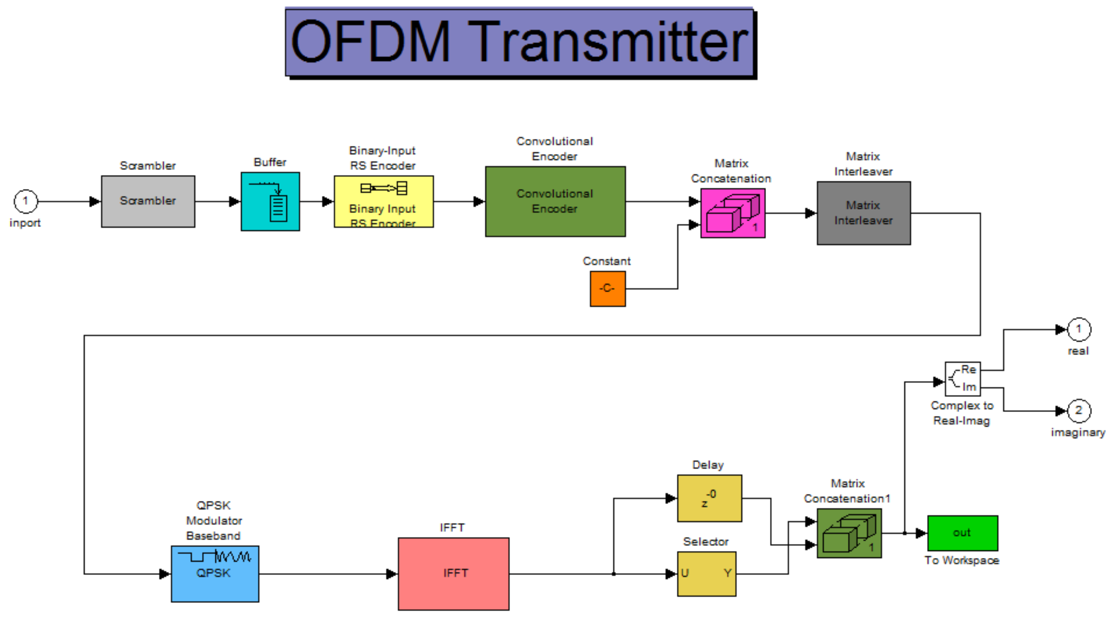


Figure 5.1 MATLAB Model Used for Verification

The result is divided into 2 different sections, for OFDM Transmitter and OFDM Receiver. The output from each of the modules is shown and followed by the overall output.

5.2 SIMULATION OF OFDM TRANSMITTER

5.2.1 SCRAMBLER

To verify proper functioning of the Scrambler was initially fed with a seed value of 1110101. Figure 5.2 shows the simulation results of the scrambler.

Input : 0110101000

Output : 1101110001

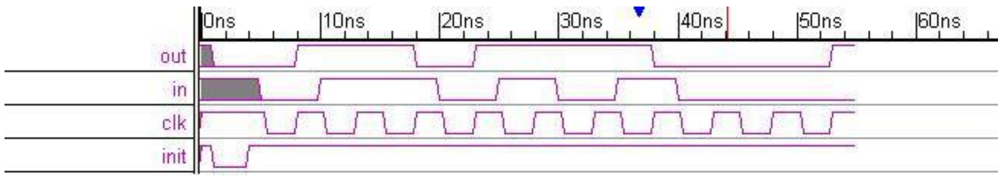


Figure 5.2 Scrambler Simulation Results

5.2.2 CONVOLUTIONAL ENCODER

After simulation of the Verilog code for convolutional encoder block, the following waveform was generated. It can be seen that first of all a low pulse was given to the arst_n (reset) input in order to initialize the shift register with all zeroes. For a 7 bit input a 14 bit output is generated. Figure 5.3 shows the resultant waveforms after the simulation of the Convolutional Encoder.

Input : 1011101

Output : 11010001011100

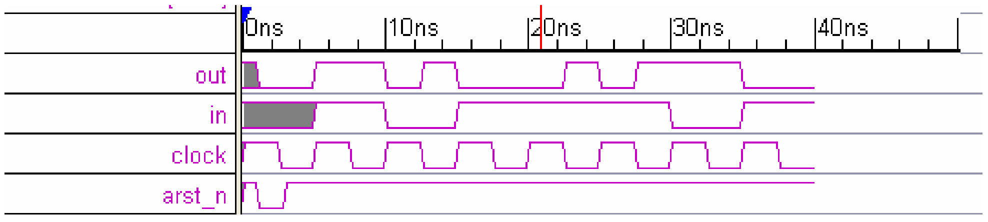


Figure 5.3 Simulation Waveform of the Convolutional Encoder

5.2.3 INTERLEAVER

The waveform for the interleaver goes up to 128 clock cycles. Therefore, it is not shown here. For an input block of data containing alternate 1s and 0s the output was 000000001111111100000000111111100000000……..so on. This clearly shows how bit positions have been changed.

5.2.4 CONSTELLATION MAPPER

Figure 5.4 shows that when an input of 10 was given to the Constellation Mapper the output was 00b504ff4afch which is found to be correct after comparing with the corresponding table.

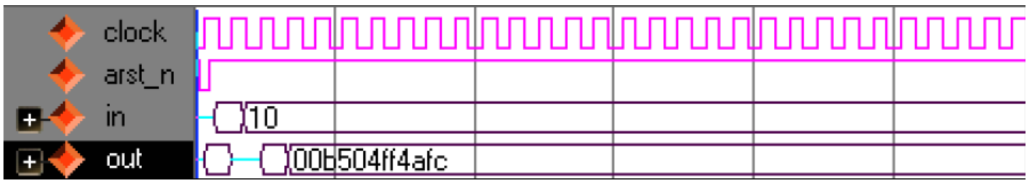


Figure 5.4 Constellation Mapper Simulation Results

5.2.5 IFFT

The IFFT was tested by giving the following 64 complex data points, h00b504000000, h030000000000, h00b504000000,…, h00b504000000 which is equivalent to 0.707, 3, 0.707,…, 0.707. Figure 5.5 shows the simulation results of the IFFT module.

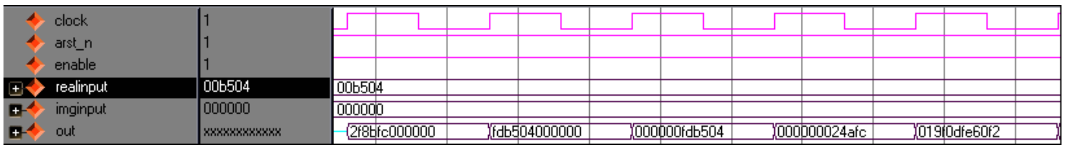


Figure 5.5 IFFT Simulation Results

The outputs were, h2f8bc000000, h5db504000000, h0000005db504 and so on. On verification with MATLAB the results turned out to be correct.

5.2.6 CYCLIC PREFIX ADDER

The inputs given to the cyclic prefix adder were

47'h000000100101, 47'h000010100001, 47'h001110100101,

7'h110010100101, 47'h000010100101, 47'h010101000101,

47'h011110100101, 47'h000011100101. . .47'h000011100101

The outputs turned out to be

47'h000011100101, 47'h000011100101, 47'h000011100101,

47'h000011100101, 47'h000011100101, 47'h000011100101,

47'h000011100101, 47'h000011100101, 47'h000000100101,

47'h000010100001, 47'h001110100101, 47'h110010100101,

47'h000010100101, 47'h010101000101, 47'h011110100101,

47'h000011100101. . . 47'h000011100101

Note that the first eight outputs are actually the last eight inputs and the rest of the output points are same as the inputs. The waveform shown in Figure 5.6 confirms the result.

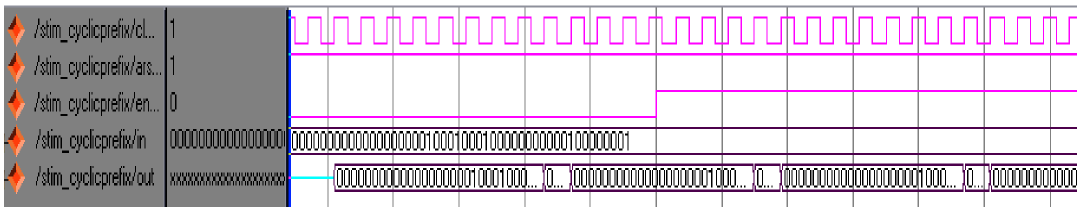
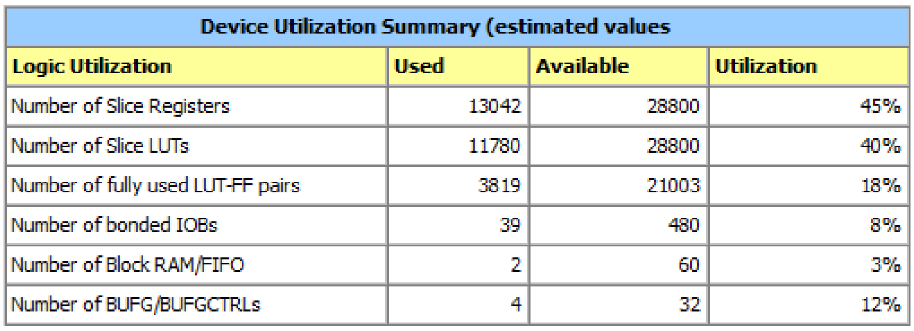


Figure 5.6 Cyclic Prefix Adder Simulation Result

5.3 SYNTHESIS OF OFDM TRANSMITTER

Table 5.1 shows the synthesis results for the OFDM transmitter using Xilinx ISE v 12.3.

Table 5.1 Device Utilization Summary for the OFDM Transmitter



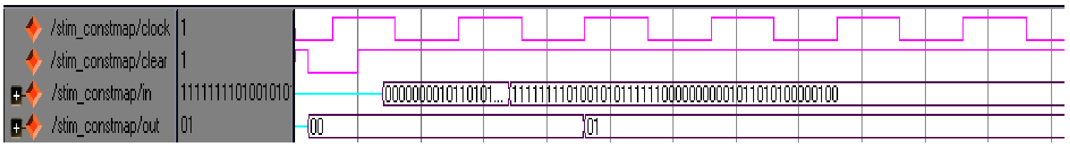
Device Utilization Summary (estimated values			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	13042	28800	45%
Number of Slice LUTs	11780	28800	40%
Number of fully used LUT-FF pairs	3819	21003	18%
Number of bonded IOBs	39	480	8%
Number of Block RAM/FIFO	2	60	3%
Number of BUFG/BUFGCTRLs	4	32	12%

5.4 SIMULATION OF OFDM RECEIVER

The Cyclic Prefix Remover simply removes the cyclic portion added at the transmitting end, and the simulation of the next block FFT is also similar to IFFT so it is not shown.

5.4.1 CONSTELLATION DE-MAPPER

The constellation demapper maps the incoming QPSK constellation points to actual data points. On giving the inputs as h00b50400b504 (which is 0.707 + j0.707) and hFF4AFC00B504 (which is -0.707 + j 0.707), The outputs turned out to be 00 and 01. As shown in Figure 5.7, the results are in accordance with calculations.



```
/stim_constmap/clock  1
/stim_constmap/clear  1
/stim_constmap/in     1111111101001010   000000001011010l...   111111110100101011111100000000001011010100000100
/stim_constmap/out    01                 00                    01
```

Figure 5.7 Constellation De-Mapper Simulation Results

5.4.2 DE-INTERLEAVER

Similar to the interleaver the simulation waveform of de-interleaver extends to 128 cycles which has been verified through simulation.

5.4.3 DE-SCRAMBLER

The inverse of scrambling is done by the De-Scrambler. For the input b1111111110000000000, the output was b1101111111111000010 which is shown in figure 5.8. The output has been verified using MATLAB.

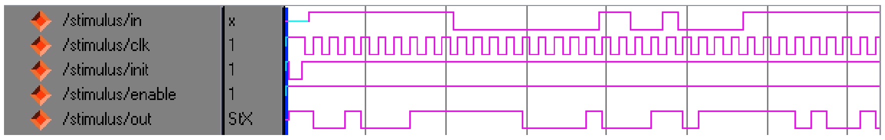
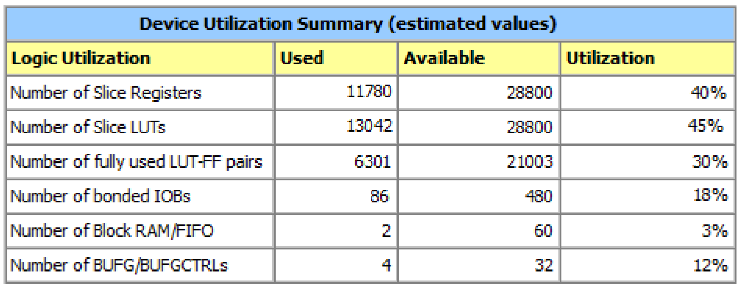


Figure 5.8 Descrambler Simulation Results

5.5 SYNTHESIS OF OFDM RECEIVER

Table 5.2 shows the device utilization summary for the OFDM receiver. It gives the percentage of utilized resources on the hardware.

Table 5.2 Device Utilization Summary for the OFDM Receiver



Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	11780	28800	40%
Number of Slice LUTs	13042	28800	45%
Number of fully used LUT-FF pairs	6301	21003	30%
Number of bonded IOBs	86	480	18%
Number of Block RAM/FIFO	2	60	3%
Number of BUFG/BUFGCTRLs	4	32	12%

5.6 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 Virtex 5 LX50T FPGA. The ADC and DAC present in the overall system architecture were used to test the system through voice communication.

As OFDM has been implemented as the modulation technique, 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.

APPENDIX A

RTL CODE IN VERILOG FOR OFDM TRANSMITTER

```
//*********************************************
// OFDM System - OFDM Transmitter and
// Receiver
//*********************************************

module OFDMSystem (
        input in,
        input clock,
        input arst_n,
        output TxD,
        output start_output
        );

        wire clock1, clock2;
        wire out_fifo, readreq;
        wire wrempty;
        wire readfull, readempty;

        wire [7:0] in_data;
        wire wrfull;
        wire [7:0] q;
        wire [47:0] out_data;

        wire idle, RxD_data_ready;

        wire rdempty1, rdfull1, wrempty1, wrfull1;
        wire [9:0] rdusedw;
        wire [6:0] wrusedw;

        wire TxD_busy, startserialtrans;

        reg start_serialtrans, start_trans;
        reg [2:0] skipbytecount;

        //*****************************************
        //PLL
        //*****************************************

        PLL pll(
                clock,
                clock1,
                clock2
                );

        //***************************************
        //FIFO
        //***************************************
```

```
fifo input_data_fifo (
        in_data,
        clock1,
        readreq,
        clock2,
        RxD_data_ready,
        out_fifo,
        readempty,
        readfull,
        wrempty,
        wrfull
        );

//**************************************
// OFDM Transmitter module
//**************************************

OFDM_transmitter transmitter (
        clock1,
        arst_n,
        out_fifo,
        out_data,
        wrfull,
        readreq,
        readempty,
        start_output
        );

//***************************************
// RS-232 Asyncronous Receiver
//***************************************

async_receiver SerialReceiver(
        clock2,
        arst_n,
        in,
        RxD_data_ready,
        in_data,
        idle
        );

//***************************************
// RS-232 Asyncronous Transmitter
//***************************************

async_transmitter serialtrans(
        clock2,
        arst_n,
        start_trans,
        q,
        TxD,
        TxD_busy
        );

//*****************************************
// FIFO for storing transmitter's output
//*****************************************

trans_out_fifo fifo(
        {out_data,16'd0},
        clock2,
        (start_serialtrans & ~TxD_busy & !start_trans),
        clock1,
        start_output,
        q,
        rdempty1,
        rdfull1,
        rdusedw,
        wrempty1,
        wrfull1,
        wrusedw
        );

//******************************************************

always @(posedge clock2 or negedge arst_n)
begin
        if(!arst_n)
                    start_serialtrans <= 1'b0;
        else if(wrusedw==71)
                    start_serialtrans <= 1'b1;
        else if(rdempty1)
                    start_serialtrans <= 1'b0;
end

//******************************************************

always @(posedge clock2 or negedge arst_n)
begin
        if(!arst_n)
                    start_trans <= 1'b0;
        else if(start_serialtrans && !TxD_busy && skipbytecount!=0 &&
skipbytecount!=1)
                    start_trans <= 1'b1;
        else if(TxD_busy)
                    start_trans <= 1'b0;
end

//******************************************************

always @(posedge clock2 or negedge arst_n)
```

```
	begin
		if(!arst_n)
			skipbytecount <= 3'd0;
		else if(start_serialtrans && !TxD_busy && !start_trans)
			skipbytecount <= skipbytecount + 3'd1;
		else if(!start_serialtrans)
			skipbytecount <= 3'd0;
	end

endmodule


//****************************************
// OFDM Transmitter top-level module
//****************************************

	module OFDM_transmitter (
		input clock,
		input arst_n,
		input in_data,
		output [47:0] out_data,
		input wrfull,
		output readreq,
		input readempty,
		output start_output
		);

		//intermediate outputs of the various blocks

		wire scrambler_out, rs_out, even_conv, odd_conv;
		wire [1:0] interleaver_out;
		wire [47:0] constmap_out, ifft_out, cyclic_out;

		//The control word

		wire [6:0] controlword;

		//**********************************
		//Control Unit
		//**********************************

		ControlUnit controlunit (
			clock,
			arst_n,
			wrfull,
			readempty,
			readreq,
			controlword
			);
```

```
//***********************************
//Scrambler
//***********************************

Scrambler scrambler (
            clock,
            arst_n,
            controlword[6],
            in_data,
            scrambler_out
            );

//*************************************
//Reed Solomon [RS (15,9)] encoder
//*************************************

ReedSolomon_Encoder rs_enc (
            clock,
            scrambler_out,
            controlword[5],
            arst_n,
            rs_out
            );

//*************************************
//Convolution Encoder (k=7, m=1, n=2)
//*************************************

convolution conv_enc (
            clock,
            arst_n,
            rs_out,
            controlword[4],
            even_conv,
            odd_conv
            );

//*********************************
//Interleaver
//*********************************

Interleaver interleaver (
            clock,
            arst_n,
            {odd_conv, even_conv},
            controlword[3],
            interleaver_out
            );

//**************************************
//Constellation Mapper (QPSK)
```

```
		//**************************************

		const_mapper constmapper (
			clock,
			arst_n,
			interleaver_out,
			constmap_out
			);

		//****************************************
		//IFFT (64-point)
		//****************************************

		ifft IFFT (
			clock,
			arst_n,
			controlword[2],
			constmap_out[47:24],
			constmap_out[23:0],
			ifft_out[47:24],
			ifft_out[23:0]
			);

		//*****************************************
		//Cyclic Prefix Adder (1/8)
		//*****************************************

		cyclic_prefix CyclicPrefixAdder (
			clock,
			arst_n,
			controlword[1],
			ifft_out,
			cyclic_out
			);

		//*****************************************
		//End of blocks
		//*****************************************

		assign start_output = controlword[0];
		assign out_data = cyclic_out;

endmodule


//**************************************
// Control Unit
//**************************************

module ControlUnit (
	input clock,
input arst_n,
input readfull,
input readempty,
output reg readreq,
output reg [6:0]controlword
);

reg [5:0] counter_en_scrambler;
reg [5:0] counter_en_rs;
reg [5:0] counter_en_convencoder;
reg [7:0] counter_en_interleaver;
reg [7:0] counter_en_ifft, counter_en_cyclic;
reg [6:0] out_en_counter;

reg temp, temp1;
reg [7:0] dummy_counter;
reg [6:0] dummy_counter1;

//************************************
//Control signal for Scrambler
//************************************

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                controlword[6] <= 1'b0;
        else if(counter_en_scrambler == 6'd35)
                controlword[6] <= 1'b0;
        else if(readreq)
                controlword[6] <= 1'b1;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                counter_en_scrambler <= 6'b000000;
        else if(counter_en_scrambler == 6'd35)
                counter_en_scrambler <= 6'b000000;
        else if(controlword[6])
                counter_en_scrambler <= counter_en_scrambler + 6'd1;
end

//****************************************
//Control signal for Reed Solomon Encoder
//****************************************

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                controlword[5] <= 1'b0;
        else if(counter_en_rs == 6'd60)
```

```
                controlword[5] <= 1'b0;
        else if(controlword[6])
                controlword[5] <= 1'b1;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                counter_en_rs <= 6'b000000;
        else if(counter_en_rs == 6'd60)
                counter_en_rs <= 6'b000000;
        else if(controlword[5])
                counter_en_rs <= counter_en_rs + 6'd1;
end

//*****************************************
//Control signal for convolutional encoder
//*****************************************

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                controlword[4] <= 1'b0;
        else if(counter_en_convencoder == 6'd59)
                controlword[4] <= 1'b0;
        else if(controlword[5])
                controlword[4] <= 1'b1;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                counter_en_convencoder <= 6'b000000;
        else if(counter_en_convencoder == 6'd59)
                counter_en_convencoder <= 6'b000000;
        else if(controlword[4])
                counter_en_convencoder <= counter_en_convencoder + 6'd1
end

//*************************************
//Control signal for the Interleaver
//*************************************

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                controlword[3] <= 1'b0;
        else if(counter_en_interleaver == 8'd63)
                controlword[3] <= 1'b0;
        else if(controlword[4])
                controlword[3] <= 1'b1;
```

```
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                counter_en_interleaver <= 8'b00000000;
        else if(counter_en_interleaver == 8'd63)
                counter_en_interleaver <= 8'b00000000;
        else if(controlword[3])
                counter_en_interleaver <= counter_en_interleaver + 8'd1;
end

//**********************************
//Control signal for ifft
//**********************************

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                temp <= 1'b0;
        else if(counter_en_ifft == 8'd80)
                temp <= 1'b0;
        else if(counter_en_interleaver == 8'd63)
                temp <= 1'b1;
        end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                controlword[2] <= 1'b0;
        else if(counter_en_ifft == 8'd67)
                controlword[2] <= 1'b0;
        else if(counter_en_ifft == 8'd4)
                controlword[2] <= 1'b1;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                counter_en_ifft <= 8'b00000000;
        else if(counter_en_ifft == 8'd80)
                counter_en_ifft <= 8'b00000000;
        else if(temp)
                counter_en_ifft <= counter_en_ifft + 8'd1;
end

//**************************************
//Control signal for cyclic prefix
//**************************************

always @(posedge clock or negedge arst_n)
```

```
begin
	if(!arst_n)
		controlword[1] <= 1'b0;
	else if(counter_en_ifft == 8'd80)
		controlword[1] <= 1'b1;
	else if(counter_en_cyclic == 8'd65)
		controlword[1] <= 1'b0;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		counter_en_cyclic <= 8'b00000000;
	else if(counter_en_cyclic == 8'd65)
		counter_en_cyclic <= 8'b00000000;
	else if(controlword[1])
		counter_en_cyclic <= counter_en_cyclic + 8'd1;
end

//****************************************
//Output control signal
//****************************************

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		dummy_counter1 <= 7'b0000000;
	else if(dummy_counter1 == 7'd66)
		dummy_counter1 <= 7'b0000000;
	else if(controlword[1])
		dummy_counter1 <= dummy_counter1 + 7'd1;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		controlword[0] <= 1'b0;
	else if(dummy_counter1 == 8'd66)
		controlword[0] <= 1'b1;
	else if(out_en_counter == 7'd71)
		controlword[0] <= 1'b0;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		out_en_counter <= 7'b0000000;
	else if(out_en_counter == 7'd71)
		out_en_counter <= 7'b0000000;
	else if(controlword[0])
		out_en_counter <= out_en_counter + 7'd1;
```

```
        end

        //*****************************
        //Read request
        //*****************************

        always @(posedge clock or negedge arst_n)
        begin
                if(!arst_n)
                        readreq <= 1'b0;
                else if(readfull)
                        readreq <= 1'b1;
                else if(readempty)
                        readreq <= 1'b0;
        end

endmodule


//*******************************************
//Scrambler module
//*******************************************

module Scrambler (
        input clock,        //positive edged clock signal
        input arst_n,        //Asynchronous negitive edged reset
        input enable,
        input in,           //Input to the Scrambler
        output out          //Output of the Scrambler
        );

        reg [6:0] LFSR;   //the 7-bit Linear Feedback Shift
                          //Register

        wire actual_in;

        reg [5:0] count;
        reg zero;

        assign actual_in = (!zero) ? in:1'b0;

        //******************************************
        //This always block shifts the LFSR register
        //one position towards right and shifts the
        //input XORed with the feedback in the left
        //most position of LFSR. If arst_n is asserted
        //then seed value is fed to the LFSR
        //*******************************************

        always @(negedge arst_n or posedge clock)
        begin
```

```
            if(!arst_n)
                  LFSR <= 7'b1110101;
            else if(!enable)
                  LFSR <= 7'b1110101;
            else if(enable)
                  LFSR <= {actual_in ^ LFSR[0] ^ LFSR[3], LFSR[6], LFSR[5], LFSR[4],
                  LFSR[3], LFSR[2], LFSR[1]};
      end

      always @(posedge clock or negedge arst_n)
      begin
            if(!arst_n)
                  count <= 6'd0;
            else if(enable)
                  count <= count + 6'd1;
            else if(!enable)
                  count <= 6'd0;
      end

      always @(posedge clock or negedge arst_n)
      begin
            if(!arst_n)
                  zero <= 1'b0;
            else if(count==31)
                  zero <= 1;
            else if(!enable)
                  zero <= 1'b0;
      end

      assign out = LFSR[6];

endmodule


//*********************************************
// Bit-Serial RS(15,9) Encoder
//*********************************************

module ReedSolomon_Encoder (
      input clock,
      input in_data,
      input enable,
      input arst_n,
      output reg out
      );

      reg [3:0] Reg0, Reg1, Reg2, Reg3, Reg4;

      wire [1:0] count_out;
      wire R5out, out_data;
      wire [3:0] GF_out;
      wire red, clearA, reset;

      assign reset = ~clearA & arst_n;

      GF_multiply_sum multiply_sum (
            clock,
            (~(count_out[0] & count_out[1])) & arst_n,
            {(R5out ^ (in_data & ~red)) & ~red,
            Reg4[0],
            Reg3[0],
            Reg2[0],
            Reg1[0],
            Reg0[0]},
            enable,
            GF_out
            );

      COUNTER_2_BIT counter2bit (
            clock,
            arst_n,
            enable,
            count_out
            );

      Redundancy redundancy (
            clock,
            arst_n,
            enable,
            red,
            clearA
            );

      s_reg_par_load_4 Reg5 (
            ~(count_out[0] & count_out[1]),
            GF_out,
            1'b0,
            clock,
            reset,
            enable,
            R5out
            );

      MUX_2_1 mux (
            (in_data & ~red),
            (R5out ^ (in_data & ~red)),
            red,
            out_data
            );

      always @(posedge clock or negedge reset)
      begin
```

```
          if(!reset)
                    Reg4 <= 4'b0000;
          else if(enable)
                    Reg4 <= {(R5out ^ (in_data & ~red)) & ~red, Reg4[3:1]};
          else if(!enable)
                    Reg4 <= 4'b0000;
end

always @(posedge clock or negedge reset)
begin
          if(!reset)
                    Reg3 <= 4'b0000;
          else if(enable)
                    Reg3 <= {Reg4[0], Reg3[3:1]};
          else if(!enable)
                    Reg3 <= 4'b0000;
end

always @(posedge clock or negedge reset)
begin
          if(!reset)
                    Reg2 <= 4'b0000;
          else if(enable)
                    Reg2 <= {Reg3[0], Reg2[3:1]};
          else if(!enable)
                    Reg2 <= 4'b0000;
end

always @(posedge clock or negedge reset)
begin
          if(!reset)
                    Reg1 <= 4'b0000;
          else if(enable)
                    Reg1 <= {Reg2[0], Reg1[3:1]};
          else if(!enable)
                    Reg1 <= 4'b0000;
end

always @(posedge clock or negedge reset)
begin
          if(!reset)
                    Reg0 <= 4'b0000;
          else if(enable)
                    Reg0 <= {Reg1[0], Reg0[3:1]};
          else if(!enable)
                    Reg0 <= 4'b0000;
end

//Registering the output

always @(posedge clock)
```

```
	begin
		out <= out_data;
	end

endmodule


//****************************************
//Convolutional Encoder module
//****************************************

module convolution(
	input clock,
	input arst_n,
	input in,
	input enable,
	output even,
	output odd
	);

	reg [0:6] Reg;
	wire actual_in;

	assign actual_in = enable?in:1'b0;

	always @(negedge arst_n or posedge clock)
	begin
		if(!arst_n)
			Reg = 7'b0000000;
		else if(!enable)
			Reg = 7'b0000000;
		else if(enable)
			Reg ={Reg[1],Reg[2],Reg[3],Reg[4],Reg[5],Reg[6],actual_in};
	end

	assign even = Reg[6] ^ Reg[1] ^ Reg[3] ^ Reg[4] ^ Reg[0];
	assign odd = Reg[6] ^ Reg[5] ^ Reg[3] ^ Reg[4] ^ Reg[0];

endmodule


//*********************************************
//Interleaver: Performs Interleaving within a block of 128 bits
//*********************************************

module Interleaver (
	input clock,
	input arst_n,
	input [1:0] in,
	input enable,
	output [1:0] out
```

```
);

reg [4:0] Counter1;
reg [4:0] Counter2;
reg [2:0] C;
reg [5:0] add_counter;
reg [5:0] sync_counter;
reg SYNC, SYNC1;
reg en1, enable2, enable_1, addcounten;
reg [1:0] reg_in, reg_in1;

wire sig;
wire out1A, out2A, out1B, out2B;
wire [1:0] out1;
wire [1:0] out2;
wire [5:0] rd_address;

assign sig = C[0] & C[1] & C[2];
assign out1 = {out1A, out2A};
assign out2 = {out1B, out2B};
assign out = (SYNC1) ? out1 : out2;

sync_dpram_64x1_rrwrou RAM1A (
        clock,
        reg_in1,
        rd_address,
        Counter1,
        !en1 & !SYNC,
        out1A
        );

sync_dpram_64x1_rrwrou RAM2A (
        clock,
        reg_in1,
        rd_address,
        Counter2,
        en1 & !SYNC,
        out2A
        );

sync_dpram_64x1_rrwrou RAM1B (
        clock,
        reg_in1,
        rd_address,
        Counter1,
        !en1 & SYNC,
        out1B
        );

sync_dpram_64x1_rrwrou RAM2B (
        clock,
          reg_in1,
          rd_address,
          Counter2,
          en1 & SYNC,
          out2B
          );

ROM_64_6 ROM (
          add_counter,
          clock,
          rd_address
          );

always @(posedge clock or negedge arst_n)
begin
          if(!arst_n)
                    Counter1 <= 5'b00000;
          else if(!en1 & enable2)
                    Counter1 <= Counter1 + 5'd1;
          else if(!enable2)
                    Counter1 <= 5'd0;
end

always @(posedge clock or negedge arst_n)
begin
          if(!arst_n)
                    Counter2 <= 5'b00000;
          else if(en1 & enable2)
                    Counter2 <= Counter2 + 5'd1;
          else if(!enable2)
                    Counter2 <= 5'd0;
end

always @(posedge clock or negedge arst_n)
begin
          if(!arst_n)
                    C <= 3'd0;
          else if(enable2)
                    C <= C + 3'd1;
          else if(!enable2)
                    C <= 3'd0;
end

always @(posedge clock or negedge arst_n)
begin
          if(!arst_n)
                    add_counter <= 6'b000000;
          else if(addcounten)
                    add_counter <= add_counter + 6'd1;
          else if(!addcounten)
                    add_counter <= 6'b0;
```

```
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		en1 <= 0;
	else if(sig)
		en1 <= ~en1;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		sync_counter <= 6'b000000;
	else if(enable2)
		sync_counter <= sync_counter + 6'd1;
	else if(!enable2)
		sync_counter <= 6'b000000;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		SYNC <= 1'b0;
	else if(sync_counter == 6'b111111)
		SYNC <= ~SYNC;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		SYNC1 <= 1'b0;
	else
		SYNC1 <= SYNC;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		enable_1 <= 1'b0;
	else
		enable_1 <= enable;
end

always @(posedge clock or negedge arst_n)
begin
	if(!arst_n)
		enable2 <= 1'b0;
	else
		enable2 <= enable_1;
end
```

```
	always @(posedge clock)
	begin
		reg_in <= in;
		reg_in1 <= reg_in;
	end

	always @(posedge clock or negedge arst_n)
	begin
		if(!arst_n)
			addcounten <= 1'b0;
		else if(sync_counter==61)
			addcounten <= 1'b1;
		else if(add_counter==63)
			addcounten <= 1'b0;
	end

endmodule


//*******************************************
//Constellation Mapper - Maps bits onto QPSK Symbols
//*******************************************

module const_mapper (
	input clock,
	input arst_n,
	input [1:0] in,
	output [47:0] data_out
	);

	ROM_48_4 ROM (
		in,
		clock,
		data_out
		);

endmodule


//***********************************
//ifft
//***********************************

module ifft (
	input clock,
	input arst_n,
	input enable,
	input [23:0] realinput,
	input [23:0] imginput,
	output [23:0] realoutput,
	output [23:0] imgoutput
	);

	wire [23:0] swappedrealin, swappedimgin;
	wire [23:0] temprealoutput, tempimgoutput;
	wire [47:0] out1;
	wire [6:0] rem1, rem2;

	assign swappedrealin = imginput;
	assign swappedimgin = realinput;

	fft_processor fft (
		clock,
		arst_n,
		enable,
		swappedrealin,
		swappedimgin,
		out1[47:24],
		out1[23:0]
		);

	//*************************************
	//Instantiation of the dividers
	//*************************************

	divider_ifft divider1 (
		clock,
		7'd64,
		temprealoutput,
		realoutput,
		rem1
		);

	divider_ifft divider2 (
		clock,
		7'd64,
		tempimgoutput,
		imgoutput,
		rem2
		);

	assign temprealoutput = out1[23:0];
	assign tempimgoutput = out1[47:24];

endmodule


//*************************************
// Addition of cyclic prefic
//*************************************
```

```
module cyclic_prefix (
        input clock,
        input arst_n,
        input enable,
        input [47:0] in,
        output [47:0] out
        );

        reg [5:0] wraddcounter, rdadd;
        reg [47:0] temp_in, temp_in1;
        reg [6:0] temp_counter;
        reg [5:0] sync_counter;
        reg read_counter_enable;
        reg sync, sync1, enable1, enable2;

        wire [47:0] out1, out2;
        wire [5:0] wradd;

        assign out = sync1?out1:out2;

        //ROM for write address

        ROM_64x6_cyclicprefix rdaddROM (
                wraddcounter,
                clock,
                wradd
                );

        //Instantiation of RAMs

        RAM_64x48 RAM1 (
                ~arst_n,
                clock,
                temp_in1,
                1'b1,
                rdadd,
                wradd,
                ~sync,
                out1
                );

        RAM_64x48 RAM2 (
                ~arst_n,
                clock,
                temp_in1,
                1'b1,
                rdadd,
                wradd,
                sync,
                out2
                );
```

```
//Always blocks

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                wraddcounter <= 6'b000000;
        else if(!enable)
                wraddcounter <= 6'b000000;
        else if(enable)
                wraddcounter <= wraddcounter + 6'd1;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                rdadd <= 6'd56;
        else if(temp_counter == 7'd71)
                rdadd <= 6'd56;
        else if(read_counter_enable)
                rdadd <= rdadd + 6'd1;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                temp_counter <= 7'b0000000;
        else if(read_counter_enable)
                temp_counter <= temp_counter + 7'd1;
        else if(temp_counter == 7'd71)
                temp_counter <= 7'd0;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                read_counter_enable <= 1'b0;
        else if(sync_counter == 6'd62)
                read_counter_enable <= 1'b1;
        else if(temp_counter == 7'd70)
                read_counter_enable <= 1'b0;
end

always @(posedge clock or negedge arst_n)
begin
        if(!arst_n)
                sync_counter <= 6'b000000;
        else if(enable2)
                sync_counter <= sync_counter + 6'd1;
        else if(!enable2)
                sync_counter <= 6'b000000;
```

```
	end

	always @(posedge clock or negedge arst_n)
	begin
		if(!arst_n)
			enable1 <= 1'b0;
		else
			enable1 <= enable;
	end

	always @(posedge clock or negedge arst_n)
	begin
		if(!arst_n)
			enable2 <= 1'b0;
		else
			enable2 <= enable1;
	end

	always @(posedge clock or negedge arst_n)
	begin
		if(!arst_n)
			sync <= 1'b0;
		else if(sync_counter == 6'b111111)
			sync <= ~sync;
	end

	always @(posedge clock or negedge arst_n)
	begin
		if(!arst_n)
			sync1 <= 1'b0;
		else
			sync1 <= sync;
	end

	always @(posedge clock)
	begin
		temp_in <= in;
		temp_in1 <= temp_in;
	end

endmodule


//************************************
// RS-232 RX module
//************************************

module async_receiver (
	input clk,
	input arst_n,
	input RxD,
output reg RxD_data_ready,
output reg [7:0] RxD_data,
output RxD_idle
);

parameter Baud = 115200;

// We also detect if a gap occurs in the received stream of characters
// Baud generator (we use 8 times oversampling)

parameter Baud8 = Baud*8;
parameter Baud8GeneratorAccWidth = 16;
wire Baud8Tick;

reg [Baud8GeneratorAccWidth:0] Baud8GeneratorAcc;

always @(posedge clk or negedge arst_n)
begin
        if(!arst_n)
                Baud8GeneratorAcc <= 17'd0;
        else
                Baud8GeneratorAcc
                <=Baud8GeneratorAcc[Baud8GeneratorAccWidth-1:0] + 16'd2416;
end

assign Baud8Tick = Baud8GeneratorAcc[Baud8GeneratorAccWidth];

reg [1:0] RxD_sync_inv;

always @(posedge clk or negedge arst_n)
begin
        if(!arst_n)
                RxD_sync_inv <= 2'd0;
        else if(Baud8Tick)
                RxD_sync_inv <= {RxD_sync_inv[0], ~RxD};
end

reg [1:0] RxD_cnt_inv;
reg RxD_bit_inv;

//Filtering the data so that short spikes on
//RxD are not mistaken as start bits

always @(posedge clk or negedge arst_n)
begin
        if(!arst_n)
                RxD_cnt_inv <= 2'd0;
        else if(Baud8Tick)
                begin
                        if( RxD_sync_inv[1] && RxD_cnt_inv!=2'b11)
                                RxD_cnt_inv <= RxD_cnt_inv + 2'h1;
                        else if(~RxD_sync_inv[1] && RxD_cnt_inv!=2'b00)
                                RxD_cnt_inv <= RxD_cnt_inv - 2'h1;
                        if(RxD_cnt_inv==2'b00)
                                RxD_bit_inv <= 1'b0;
                        else if(RxD_cnt_inv==2'b11)
                                RxD_bit_inv <= 1'b1;
                end

end

reg [3:0] state;
reg [3:0] bit_spacing;

// "next_bit" controls when the data sampling occurs
// with a clean connection, values from 8 to 11 work

wire next_bit = (bit_spacing==4'd8);

always @(posedge clk)
begin
        if(state==0)
                bit_spacing <= 4'b0000;
        else if(Baud8Tick)
                bit_spacing <= {bit_spacing[2:0] + 4'b0001} |
                {bit_spacing[3], 3'b000};
end

always @(posedge clk)
begin
        if(Baud8Tick)
                case(state)
                        4'b0000: if(RxD_bit_inv) state <= 4'b1000; // start bit found?
                        4'b1000: if(next_bit) state <= 4'b1001; // bit 0
                        4'b1001: if(next_bit) state <= 4'b1010; // bit 1
                        4'b1010: if(next_bit) state <= 4'b1011; // bit 2
                        4'b1011: if(next_bit) state <= 4'b1100; // bit 3
                        4'b1100: if(next_bit) state <= 4'b1101; // bit 4
                        4'b1101: if(next_bit) state <= 4'b1110; // bit 5
                        4'b1110: if(next_bit) state <= 4'b1111; // bit 6
                        4'b1111: if(next_bit) state <= 4'b0001; // bit 7
                        4'b0001: if(next_bit) state <= 4'b0000; // stop bit
                        default: state <= 4'b0000;
                endcase
end

always @(posedge clk)
begin
        if(Baud8Tick && next_bit && state[3])
                RxD_data <= {~RxD_bit_inv, RxD_data[7:1]};
end
```

```
	always @(posedge clk)
	begin
		RxD_data_ready <= (Baud8Tick && next_bit && state==4'b0001 &&
	~RxD_bit_inv);
		// ready only if the stop bit is received
	end

	reg [4:0] gap_count;

	always @(posedge clk)
	begin
		if (state!=0)
			gap_count<=5'h00;
		else if(Baud8Tick & ~gap_count[4])
			gap_count <= gap_count + 5'h01;
	end

	assign RxD_idle = gap_count[4];

endmodule
```

APPENDIX B

RTL CODE IN VERILOG FOR OFDM RECEIVER

```
//*******************************************
// OFDM Receiver module
//*******************************************

module OFDM_receiver (
        input clock,
        input arst_n,
        input enable,
        input [47:0] in_data,
        output out_data
        );

        wire [5:0] controlword;
        wire [47:0] fft_output;
        wire [1:0] demap_output, deinterleaver_output;

        reg source_rdy, sink_val;
        reg eras_sym;

        wire sink_rdy, source_val, decbit;
        wire [7:0] normalizations;

        reg rs_source_ena, rs_sink_val, rs_sink_eop, rs_sink_sop;
        reg [3:0] rsin;

        wire rs_decfail, rs_sink_ena, rs_source_val;
        wire rs_source_sop, rs_source_eop;
        wire [2:0] num_err_sym;
        wire [3:0] rsout;

        //*******************************************
        // Control Unit
        //*******************************************

        Receiver_control_unit control_unit (
                clock,
                arst_n,
                enable,
                controlword
                );

        //*******************************************
        // Fast Fourier Trasnform
        //*******************************************

        fft_processor fft (
                clock,
		arst_n,
		controlword[5],
		in_data[47:24],
		in_data[23:0],
		fft_output[47:24],
		fft_output[23:0]
		);

//****************************************
// Constellation De-Mapper
//****************************************

const_demapper constellation_demapper (
		clock,
		arst_n,
		fft_output,
		demap_output
		);

//****************************************
// De-interleaver
//****************************************

DeInterleaver interleaver (
		clock,
		arst_n,
		demap_output,
		controlword[4],
		deinterleaver_output
		);

//****************************************
// Viterbi Decoder
//****************************************

viterbi viterbidecoder (
		.clk(clock),
		.decbit(decbit),
		.eras_sym(eras_sym),
		.normalizations(normalizations),
		.reset(~arst_n),
		.rr(deinterleaver_output),
		.sink_rdy(sink_rdy),
		.sink_val(sink_val),
		.source_rdy(source_rdy),
		.source_val(source_val)
		);

//***************************************
// Reed-Solomon Decoder
//***************************************
```

```
	rsdec rsdecoder (
		.bypass(1'b0),
		.clk(clock),
		.decfail(decfail),
		.num_err_sym(num_err_sym),
		.reset(~arst_n),
		.rsin(rsin),
		.rsout(rsout),
		.sink_ena(rs_sink_ena),
		.sink_eop(rs_sink_eop),
		.sink_sop(rs_sink_sop),
		.sink_val(rs_sink_val),
		.source_ena(rs_source_ena),
		.source_eop(rs_source_eop),
		.source_sop(rs_source_sop),
		.source_val(rs_source_val)
		);

	//*****************************************
	// Descrambler
	//*****************************************

	Descrambler descrambler(
		clock,
		arst_n,
		enable,
		in,
		out
		);

endmodule


//*******************************************
//Module - Descrambler
//*******************************************

module Descrambler (
	input clock,
	input arst_n,
	input enable,
	input in,
	output reg out
	);

	reg [6:0] LFSR;

	always @(posedge clock or negedge arst_n)
	begin
		if(!arst_n)
```

```
                LFSR <= 7'b1110101;
            else if(!enable)
                LFSR <= 7'b1110101;
            else if(enable)
                LFSR <= {in, LFSR[6], LFSR[5], LFSR[4], LFSR[3], LFSR[2], LFSR[1]};
        end

        always @(posedge clock or negedge arst_n)
        begin
            if(!arst_n)
                out <= 1'b0;
            else
                out <= in ^ LFSR[0] ^ LFSR[3];
        end

endmodule
```

REFERENCES

[1] Jeffrey G. Andrews and Rias Muhammad, *Fundamentals of WIMAX*. Prentice Hall Communications Engineering, 2006.

[2] Ahmed R. S. Bahai and Burton R. Saltzberg, *Multi Carrier Digital Communications*. Kluwer Academic Publishers, 2002.

[3] Aseem Pandey, Shyam Ratan Agrawalla & Shrikant Manivannan, "*VLSI Implementation of OFDM*", Wipro Technologies, September 2002.

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

[5] "*Orthogonal Frequency Division Multiplexing Tutorial*", Intuitive guide to Principles of Communications, http://www.complextoreal.com

[6] Magis Networks White paper, "*Orthogonal Frequency Division Multiplexing (OFDM) Explained*," Inc. 2001

[7] "*Orthogonal Frequency-Division Multiplexing (OFDM)*", the International Union of Radio Science (URSI), Lulea University of Technology, 2002

[8] Michael D. Ciletti, *Advanced Digital Design with the Verilog HD Xilinx Design Series*. Prentice Hall, 2002.

[9] Lattice Semiconductor white paper, "*Implementing WiMAX OFDM Timing and Frequency Offset Estimation in Lattice FPGAs*," 2005.

[10] J. L. Holsinger, "*Digital communication over fixed time-continuous channels with memory, with special application to telephone channels*," PhD thesis, Massachusetts Institute of Technology, 1964.

[11] "*A Tutorial on Convolutional Coding with Viterbi Decoding*", Spectrum Applications, http://home.netcom.com/~chip.f/viterbi/tutorial.html.

[12] S. B. Weinstein and P. M. Ebert, "*Data transmission by frequency-division multiplexing using the discrete Fourier transform*", IEEE Trans. Communications, COM-19(5): 628-634, Oct. 1971.

[13] "*Interleaver*", Wikipedia the free encyclopedia, http://en.wikipedia.org/wiki/Interleaver

[14] "*DFT*", Wikipedia the free Encyclopedia, en.wikipedia.org/wiki/Discrete_Fourier_transform

[15] L. J. Cimini “*Analysis and simulation of a digital mobile channel using orthogonal frequency division multiplexing*.” IEEE Transactions on Communications, 33(7):665–675, July 1985.

[16] S. Weinstein and P. Ebert “*Data transmission by frequency-division multiplexing using the discrete Fourier transform*.” IEEE Transactions on Communications, 19(5):628–634, October 1971.

[17] “*Fast Fourier Transform*”, Molfram MathWorld, mathworld.wolfram.com/FastFourierTransform.html