P-3046

FPGA IMPLEMENTATON OF OFDM

A PROJECT REPORT

Submitted by

JEEVA. P	71206106021
ARUNRAJ. S	71206106011
GURUMOORTHY. D	71206106019
SURESH. K	71206106053

In partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY, COIMBATORE

ANNA UNIVERSITY : 600 025

APRIL 2010

ANNA UNIVERSITY : CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report **"FPGA IMPLEMENTATION OF OFDM"** is the bonafide work of **"JEEVA.P, ARUNRAJ.S, GURUMOORTHY.D, SURESH.K"** who carried out this project work under my supervision.

[illegible] 1/24/10

SIGNATURE

Dr. RAJESWARI MARIAPPAN, Ph.D.,

HEAD OF THE DEPARTMENT

Electronics & Communication

Engineering,

Kumaraguru College of Technology,

Coimbatore-641006

[illegible]

SIGNATURE

Ms.K.KAVITHA, M.E.,(Ph.D.,)

SENIOR LECTURER

SUPERVISOR

Electronics & Communication

Engineering,

Kumaraguru College of

Technology,

Coimbatore-641006

The candidates with university register number 71206106021, 71206106011, 71206106019, 71206106053 were examined by us in project Viva Voce examination held on 15-04-2010.

[illegible] 15.04.2010

INTERNAL EXAMINER

[illegible]

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We acknowledge our thanks to **Dr.J.SHANMUGAM Ph.D.,DIRECTOR,** Kumaraguru College of Technology, Coimbatore, for providing us an opportunity to take up this project.

We acknowledge our thanks to **Dr.S.RAMACHANDRAN Ph.D., PRINCIPAL**, Kumaraguru College of Technology, Coimbatore, for providing us an opportunity to take up this project.

We take this opportunity to thank, **Dr.RAJESWARI MARIAPPAN Ph.D., HEAD OF THE DEPARTMENT**, Electronics and Communication Engineering, Kumaraguru College of Technology, Coimbatore, for her extended support in completion of the project.

We wish to express our sincere thanks and deep sense of gratitude to our Project coordinator, **Ms.A.VASUKI M.E.,(Ph.D.,) ASST PROFESSOR,** Department of Electronics and Communication Engineering, Kumaraguru College of Technology, Coimbatore, for her wise counsel with timely and inspiring directions during this project work.

We take this opportunity to express our profound thanks to our guide **Ms.K.KAVITHA M.E.,(Ph.D.,),SENIOR LECTURER,** Department of Electronics and Communication Engineering, Kumaraguru College of Technology, Coimbatore, for her valuable guidance and support for the successful completion of the project work.

Finally, we express our thanks to our parents and friends for their encouragement in completion of this project work successfully.

ABSTRACT

The OFDM concept has become very popular in the recent days due to its advantages compared to the other conventional modulation techniques.

Moreover, the 4G mobile standards that are being developed by the scientists also employ OFDM. It is also popular for wide-band communications today by way of low cost digital signal processing components that can efficiently calculate the FFT. So the design of reliable, low-cost OFDM systems for bringing into practical use becomes the need of the hour.

The project aims at implementing OFDM in the FPGA. As OFDM is a complex concept with many modules integrated into it, a three stepped approach is followed to achieve the final goal. In the first step, the entire OFDM system with its modules is analyzed by using MATLAB.A Rayleigh channel is simulated and noise components are added and the BER performance of OFDM with BPSK modulation is studied.

In the second step, VHDL is used to design the QAM & FFT blocks of the OFDM system. A radix-2 butterfly processor which incorporates a complex conjugate multiplier is also designed. This module serves as a basic block for developing higher radix FFTs. The VHDL programs are functionally simulated using Modelsim software.

In the third step, FPGA implementation of OFDM system consisting of the QAM and FFT blocks is carried out. The synthesis is done using XILINX-ISE. The design is implemented in XILINX SPARTAN 2 chip.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 NEED FOR THE PROJECT

In recent days, the need for wireless communications under mobile conditions has increased. For instance, the very high throughput (VHT) study group wants to go beyond the IEEE 802.11n standard. This study group suggests that MIMO-OFDM technology will be a potential solution to increase the data-rate & throughput. The combination of MIMO transmission, OFDM technology and the STBC scheme comprises a promising solution for next generation wireless communications. The aim of our project is to implement the OFDM concept in FPGA.

With the foray of mobile 3G technology all over the world, scientists are now concentrating on developing 4G technology. OFDM is seen as a viable method to be used by the 4G standards. The OFDM has many advantages compared to the conventional modulation techniques that make it very popular in the recent times. It is robust against inter-symbol interference (ISI), narrow band co-channel interference and fading caused by multi-path propagation. It has high spectral efficiency as compared to conventional modulation schemes, spread spectrum, etc. It also has low sensitivity to time synchronization errors.

OFDM is popular for wide-band communications today by way of low cost digital signal processing components that can efficiently calculate the FFT. The OFDM has found applications in the wireless LAN (WLAN) radio interfaces IEEE 802.11a, g & n, digital radio systems (DAB/EUREKA), digital TV systems (DVB-T & ISDB-T), mobile TV systems, FLASH-OFDM & Wireless MAN.

Hence it is the need of the hour to develop reliable, low-cost OFDM systems that can serve better than the communication systems available today and bring those systems into practical use. Our project is just a small step towards exploring this OFDM concept from various aspects

1.2 OVERVIEW OF THE PROJECT

The concept of OFDM is a complex concept that comprises many modules and entities within itself. Our aim is to go for the FPGA implementation of OFDM. In order to achieve our final goal, we follow a three-stepped approach.

1. Analysis of BER performance of OFDM system with the help of MATLAB.

2. a)VHDL coding of the QAM and FFT blocks of OFDM system and functional simulation of the system using Modelsim software.

   b)VHDL coding of the Butterfly processor module along with the complex conjugate multiplier and functional simulation using Modelsim.

3. FPGA implementation of the OFDM system consisting of the QAM and FFT blocks.

1.3 VLSI DESIGN:

VLSI stands for "Very Large Scale Integration". This is the field which involves packing more and more logic devices into smaller and smaller areas. VLSI, circuits that would have taken many boards of space can now be put into small space few millimeters across! VLSI circuits are everywhere... our computers, our car, our brand new state-of-the-art digital camera, the cell phone, and what we have. All this involves a lot of expertise on many fronts within the same field which we will look at in later section.

DEALING WITH VLSI CIRCUIT

The way the normal blocks like latches and gates are implemented is different, but the behavior remains the same. All the miniaturization involves new things to consider. A lot of thought has to go into actual implementations as well as design.

Circuit delays: Large complicated circuits running at very high frequencies have one big problem to tackle-the problem to delays in propagation of signals through gates and wires. Even for areas a few micrometers across. The operation speed is very large that as the delays add up, they can actually become comparable to the clock speed.

1. **Power**: Another effect of high operation frequencies is increased consumption of power. This has two-fold effect-devices consumes batteries faster, and heat dissipation increases. Coupled with the fact the surface areas have decreased, heat posses a major threat to the stability of the circuit itself.

2. **Layouts:** Laying out the circuit components is task common to all branches of electronics. What's so special in our case is that there are many possible ways to do this; there can be multiple layers of different materials on the same silicon, there can be different arrangements of the smaller parts for the same component and soon. The choice between the two is determined by the way we choose the layout the circuit components. Layout can also affect the fabrication of VLSI chips, making it either easy or difficult to implement the components on the silicon.

CHAPTER 2

CONCEPT OF OFDM

2.1 INTRODUCTION TO OFDM

Orthogonal frequency-division multiplexing (OFDM), essentially identical to **coded OFDM (COFDM)** and **discrete multi-tone modulation (DMT)**, is a frequency-division multiplexing (FDM) scheme utilized as a digital multi-carrier modulation method. A large number of closely-spaced orthogonal sub-carriers are used to carry data. The data is divided into several parallel data streams or channels, one for each sub-carrier. Each sub-carrier is modulated with a conventional modulation scheme (such as quadrature amplitude modulation or phase-shift keying) at a low symbol rate, maintaining total data rates similar to conventional single-carrier modulation schemes in the same bandwidth.

OFDM has developed into a popular scheme for wideband digital communication, whether wireless or over copper wires, used in applications such as digital television and audio broadcasting, wireless networking and broadband internet access.

The primary advantage of OFDM over single-carrier schemes is its ability to cope with severe channel conditions (for example, attenuation of high frequencies in a long copper wire, narrowband interference and frequency-selective fading due to multipath) without complex equalization filters. Channel equalization is simplified because OFDM may be viewed as using many slowly-modulated narrowband signals rather than one rapidly-modulated wideband signal. The low symbol rate makes the use of a guard interval between symbols affordable, making it possible to handle

Time-spreading and eliminate inter-symbol interference (ISI). This mechanism also facilitates the design of single frequency networks (SFNs), where several adjacent transmitters send the same signal simultaneously at the same frequency, as the signals from multiple distant transmitters may be combined constructively, rather than interfering as would typically occur in a traditional single-carrier system.

2.2 ORTHOGONALITY

In OFDM, the sub-carrier frequencies are chosen so that the sub-carriers are **orthogonal** to each other, meaning that **cross-talk** between the sub-channels is eliminated and inter-carrier guard bands are not required. This greatly simplifies the design of both the **transmitter** and the **receiver**; unlike conventional **FDM**, a separate filter for each sub-channel is not required.

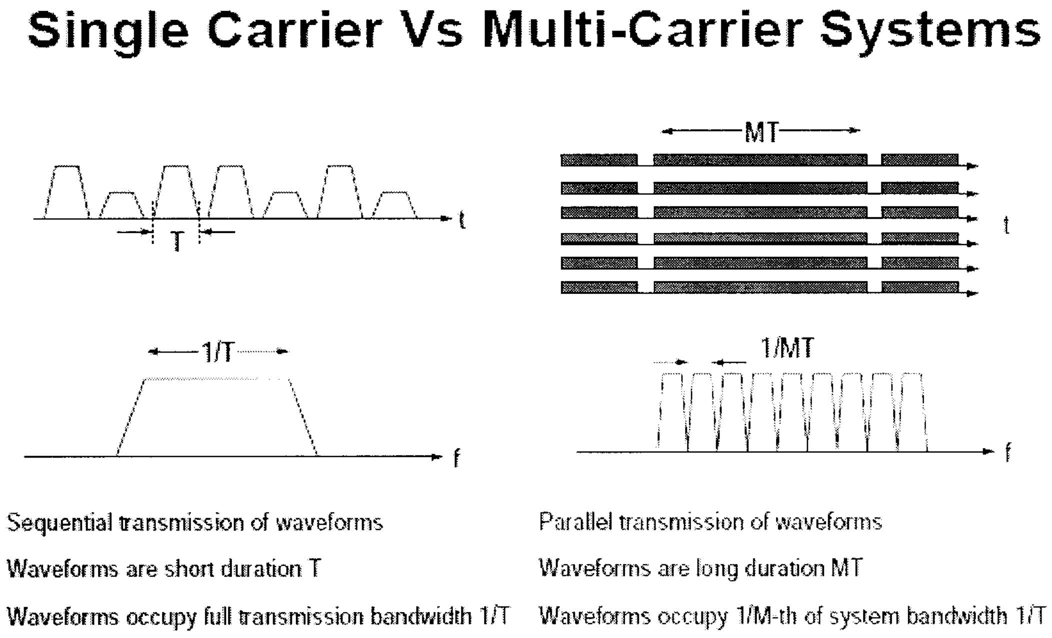
The orthogonality requires that the sub-carrier spacing is del(f) = K/ Tu Hertz, where *Tu* seconds is the useful symbol duration (the receiver side window size), and *k* is a positive integer, typically equal to 1. Therefore, with *N* sub-carriers, the total passband bandwidth will be B= N·del(f) *(*Hz).

The orthogonality also allows high spectral efficiency, with a total symbol rate near the Nyquist rate for the equivalent baseband signal (i.e. near half the Nyquist rate for the double-side band physical passband signal). Almost the whole available frequency band can be utilized. OFDM generally has a nearly 'white' spectrum, giving it benign electromagnetic interference properties with respect to other co-channel users.

OFDM requires very accurate frequency synchronization between the receiver and the transmitter; with frequency deviation the sub-carriers will no longer be orthogonal, causing inter-carrier interference (ICI) (i.e., cross-talk between the sub-carriers). Frequency offsets are typically caused by mismatched transmitter and receiver oscillators, or by Doppler shift due to movement. While Doppler shift alone may be compensated for by the receiver, the situation is worsened when combined with multipath, as reflections will appear at various frequency offsets, which is much harder to correct.

2.3 SINGLE CARRIER VS MULTI-CARRIER SYSTEMS.

The diagram below explains the time/ frequency domain pulse waveforms for the single carrier & multi-carrier systems. As seen clearly, the parallel pulses occupy only a fraction of the system bandwidth, thus the available frequency spectrum is made use of more efficiently in the parallel transmission.



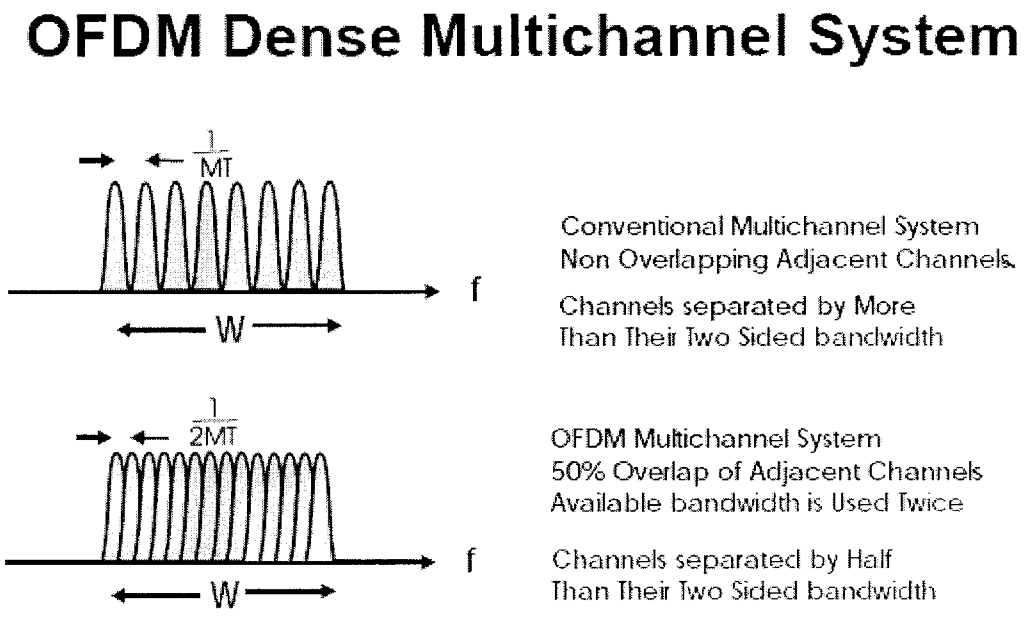
Single Carrier Vs Multi-Carrier Systems

Sequential transmission of waveforms	Parallel transmission of waveforms
Waveforms are short duration T	Waveforms are long duration MT
Waveforms occupy full transmission bandwidth 1/T	Waveforms occupy 1/M-th of system bandwidth 1/T

FIG 2.1 Single carrier vs multicarrier system

2.4 OFDM DENSE MULTI-CHANNEL SYSTEM:

In an OFDM dense multi-channel system, 50% overlap of adjacent channels takes place compared to the Non overlapping adjacent channels that are found in the conventional multichannel system. The orthogonality of the adjacent pulses play a major role here as otherwise ISI will be more for overlapping adjacent channels. Thus the available bandwidth is used twice and also the spectral efficiency increases.



OFDM Dense Multichannel System
1/MT
f
W
Conventional Multichannel System
Non Overlapping Adjacent Channels.
Channels separated by More Than Their Two Sided bandwidth
1/2MT
f
W
OFDM Multichannel System
50% Overlap of Adjacent Channels
Available bandwidth is Used Twice
Channels separated by Half Than Their Two Sided bandwidth

FIG 2.2 OFDM dense multichannel system

2.5 TRANSMITTER BLOCK DIAGRAM

An OFDM carrier signal is the sum of a number of orthogonal sub-carriers, with baseband data on each sub-carrier being independently modulated commonly using some type of quadrature amplitude modulation (QAM) or phase-shift keying (PSK). This composite baseband signal is typically used to modulate a main RF carrier. s[n] is a serial stream of binary digits. By inverse multiplexing, these are first demultiplexed into N parallel streams, and each one mapped to a (possibly complex) symbol stream using some modulation constellation (QAM, PSK). The constellations may be different, so some streams may carry a higher bit-rate than others.An inverse FFT is computed on each set of symbols, giving a set of complex time-domain samples. These samples are then quadrature-mixed to passband in the standard way. The real and imaginary components are first converted to the analogue domain using digital-to-analogue converters (DACs); the analogue signals are then used to modulate cosine and sine waves at the carrier frequency, fc, respectively. These signals are then summed to give the transmission signal, s(t).

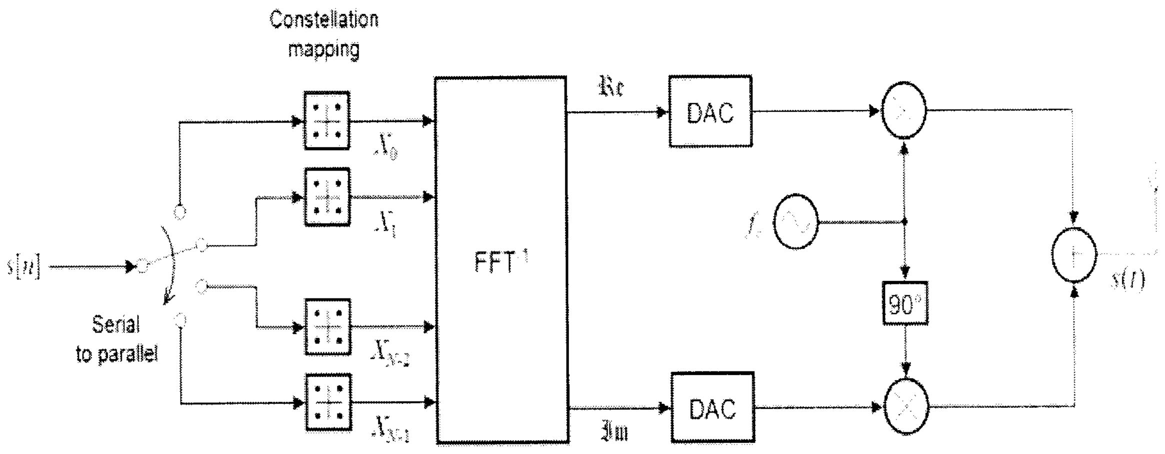


FIG 2.3 OFDM transmitter

2.6 RECEIVER BLOCK DIAGRAM

The receiver picks up the signal r(t), which is then quadrature-mixed down to baseband using cosine and sine waves at the carrier frequency. This also creates signals centered on 2fc, so low-pass filters are used to reject these. The baseband signals are then sampled and digitised using analogue-to-digital converters (ADCs), and a forward FFT is used to convert back to the frequency domain.

This returns N parallel streams, each of which is converted to a binary stream using an appropriate symbol detector. These streams are then re-combined into a serial stream, , s^[n] which is an estimate of the original binary stream at the transmitter.

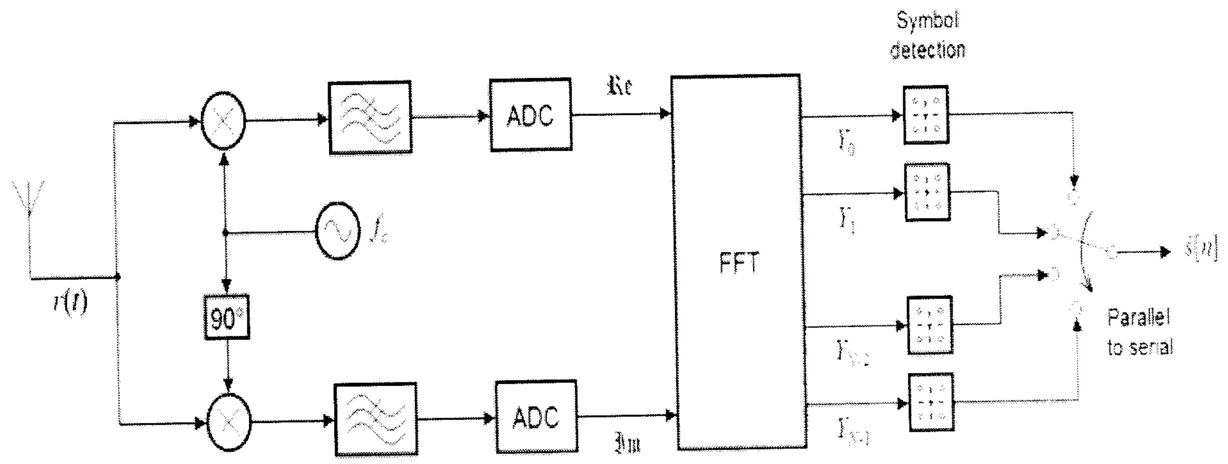


FIG 2.4 OFDM receiver

2.7 MATHEMATICAL DESCRIPTION OF OFDM

If N sub-carriers are used, and each sub-carrier is modulated using M alternative symbols, the OFDM symbol alphabet consists of M^N combined symbols.

The low-pass equivalent OFDM signal is expressed as:

$$\nu(t) = \sum_{k=0}^{N-1} X_k e^{j2\pi kt/T}, \quad 0 \le t < T,$$

where {Xk}are the data symbols, N is the number of sub-carriers, and T is the OFDM symbol time. The sub-carrier spacing of 1/T makes them orthogonal over each symbol period; this property is expressed as:

$$\frac{1}{T}\int_0^T \left(e^{j2\pi k_1 t/T}\right)^* \left(e^{j2\pi k_2 t/T}\right) dt$$
$$= \frac{1}{T}\int_0^T e^{j2\pi(k_2-k_1)t/T} dt = \delta_{k_1 k_2}$$

where (.)* denotes the complex conjugate operator and del function is the Kronecker delta.

Kronecker Delta is a function of two variables, usually integers, which is 1 if they are equal, and 0 otherwise. So, for example,

δ1,2 = 0, but
δ3,3 = 1

It is written as the symbol δij, and treated as a notational shorthand rather than as a function.

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}$$

P-3046

To avoid intersymbol interference in multipath fading channels, a guard interval of length Tg is inserted prior to the OFDM block. During this interval, a cyclic prefix is transmitted. The OFDM signal with cyclic prefix is thus:

$$\nu(t) = \sum_{k=0}^{N-1} X_k e^{j2\pi kt/T}, \quad -T_g \leq t < T$$

The low-pass signal above can be either real or complex-valued. Real-valued low-pass equivalent signals are typically transmitted at baseband—wireline applications such as DSL use this approach. For wireless applications, the low-pass signal is typically complex-valued; in which case, the transmitted signal is up-converted to a carrier frequency fc. In general, the transmitted signal can be represented as:

$$\begin{aligned} s(t) &= \Re\left\{\nu(t)e^{j2\pi f_c t}\right\} \\ &= \sum_{k=0}^{N-1} |X_k| \cos\left(2\pi[f_c + k/T]t + \arg[X_k]\right) \end{aligned}$$

2.8 ADVANTAGES AND DISADVANTAGES OF OFDM

ADVANTAGES:

- Robust against narrow-band co-channel interference.
- Robust against intersymbol interference (ISI) and fading caused by multipath propagation.
- High spectral efficiency as compared to conventional modulation

schemes,spread spectrum, etc.

- Efficient implementation using Fast Fourier Transform (FFT).
- Low sensitivity to time synchronization errors.
- Tuned sub-channel receiver filters are not required (unlike conventional FDM).
- Facilitates single frequency networks (SFNs); i.e., transmitter macrodiversity.
- Efficiently deals with channel delay spread
- Enchanced channel capacity
- Can easily adapt to severe channel conditions without complex equalization.

DISADVANTAGES:

- Sensitive to frequency synchronization problems.
- High peak-to-average-power ratio (PAPR), requiring linear transmitter circuitry, which suffers from poor power efficiency.
- Loss of efficiency caused by cyclic prefix/guard interval.
- Almost half the spectral efficiency offered by vestigial sideband modulation; e.g., used in the ATSC digital TV system.
- Sensitive to small carrier frequency offsets
- Sensitive to high frequency phase noise

2.9 APPLICATIONS OF OFDM

A) CABLE:

- PEP via telephone lines.

- ADSL and VDSL broadband access via POTS copper wiring.
- Power line communication (PLC).
- Multimedia over Coax Alliance (MoCA) home networking.
- ITU-T G.hn, a standard which provides high-speed local area networking over existing home wiring (power lines, phone lines and coaxial cables).
- DVB-C2, an enhanced version of the DVB-C digital cable TV standard.

B) WIRELESS:

- The wireless LAN (WLAN) radio interfaces IEEE 802.11a, g, n and HIPERLAN/2.
- The digital radio systems DAB/EUREKA 147, DAB+, Digital Radio Mondiale, HD Radio, T-DMB and ISDB-TSB.
- The terrestrial digital TV systems DVB-T and ISDB-T.
- The terrestrial mobile TV systems DVB-H, T-DMB, ISDB-T and MediaFLO forward link.
- The cellular network's FLASH-OFDM.
- The mobile broadband 3GPP Long Term Evolution air interface named High Speed OFDM Packet Access (HSOPA).
- The wireless MAN/fixed broadband wireless access (BWA) standard IEEE 802.16 (or WiMAX).
- The mobile broadband wireless access (MBWA) standards IEEE 802.20, IEEE 802.16e (Mobile WiMAX) and WiBro.

CHAPTER 3

STUDY OF BER PERFORMANCE OF OFDM SYSTEM USING MATLAB

3.1 MATLAB:

MATLAB stands for "Matrix Laboratory" and is a numerical computing environment and fourth-generation programming language. Developed by The MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, and Fortran. In 2004, MathWorks claimed that MATLAB was used by more than one million people across the industry and the academic world.

MATLAB, the application, is built around the MATLAB language. The simplest way to execute MATLAB code is to type it in at the prompt, >> , in the Command Window, one of the elements of the MATLAB Desktop. In this way, MATLAB can be used as an interactive mathematical shell. Sequences of commands can be saved in a text file, typically using the MATLAB Editor, as a script or encapsulated into a function, extending the commands available.

Variables are defined with the assignment operator, =. MATLAB is a weakly dynamically typed programming language. It is a weakly typed language because types are implicitly converted. It is a dynamically typed language because variables can be assigned without declaring their type, except if they are to be treated as symbolic objects, and that their type can change. Values can come from constants, from computation involving values of other variables, or from the output of a function.

MATLAB is a "Matrix Laboratory", and as such it provides many convenient ways for creating vectors, matrices, and multi-dimensional arrays. In the MATLAB vernacular, a vector refers to a one dimensional ($1\times N$ or $N\times 1$) matrix, commonly referred to as an array in other programming languages. A matrix generally refers to a 2-dimensional array, i.e. an $m\times n$ array where m and n are greater than or equal to 1. Arrays with more than two dimensions are referred to as multidimensional arrays.

Matrices can be defined by separating the elements of a row with blank space or comma and using a semicolon to terminate each row. The list of elements should be surrounded by square brackets: []. Parentheses: ( ) are used to access elements and subarrays (they are also used to denote a function argument list).

Unlike many other languages, where the semicolon is used to terminate commands, in MATLAB the semicolon serves to suppress the output of the line that it concludes.

MATLAB can call functions and subroutines written in the C programming language or Fortran. A wrapper function is created allowing MATLAB data types to be passed and returned. The dynamically loadable object files created by compiling such functions are termed "MEX-files"

MATLAB is a proprietary product of The MathWorks, so users are subject to vendor lock-in. Although MATLAB Builder can deploy MATLAB functions as library files which can be used with .NET or Java application building environment, future development will still be tied to the MATLAB language.

3.2 PURPOSE OF MATLAB ANALYSIS:

The computation of Bit Error Rate (BER) is useful to find out the performance of any modulation scheme and hence it serves as a precursor to the analysis and design of various modulation techniques and systems. Here also, in order to study the performance of OFDM we go for Matlab simulation which allows us to analyze the BER performance using BPSK in OFDM modulation. The use of MATLAB also helps us to realize the Rayleigh fading channel and also to introduce noise components in order to simulate a real-time system so that its BER can be calculated with greater accuracy.

3.3 COMPUTING THE BER FOR BPSK IN OFDM MODULATION IN THE PRESENCE OF RAYLEIGH FADING CHANNEL

Initial Assumptions:

1. The FFT is taken as 64 point-FFT.

2. Out of the 64 lines, 52 are used as the data sub-carriers used in the BPSK modulation.

3. The number of bits per OFDM symbol is taken to be same as the number of sub-carriers for BPSK, hence it is also 52.

4. The no. of symbols is taken as 10^4.

5. The bit-energy to noise-density ratio is taken as 0:35

6. The bit-energy to noise ratio is converted to the form of symbol to noise ratio.

Transmitter side:

7. The input bits are generated as a random stream of 1's & 0's with equal probability.

8. Then Binary Phase Shift Keying is done by assigning a value of -1 to all input stream 0's and +1 to all input stream 1's.

9.The bits are then grouped into multiple symbols.

10. The modulated symbols are then assigned to the sub-carriers (from -26 to -1, +1 to +26) thus making the total no. of sub-carriers as 52.

11. The power of transmit symbol is normalized to 1.

12. The cyclic prefixes are then appended. Cyclic Prefixes are used in OFDM in order to combat multipath by making channel estimation easy.

13. In order to account for the multipath interference that may occur in transmission, a 10-tap multipath channel is simulated. Here the channel is modeled as Rayleigh Channel in order to account for non line-of-sight propagation.

14. The frequency response of the channel is then computed and stored for use at the receiver.

15. Then each individual symbol is convolved with the random channel.

16. The multiple symbols are then concatenated to form a long vector.

17. Gaussian noise of unit variance and zero mean is then introduced.

18. When noise is added, some energy will be wasted due to the cyclic prefix. To account for this, the term sq.root (80/64) is multiplied with the signal obtained at Receiver.

Receiver Side:

19. Firstly, the received vectors are formatted into symbols.

20. The cyclic prefixes are removed.

21. Then the conversion to the frequency domain is done.

22. The process of equalization by the known channel frequency response is then done. Equalization is carried out in order to flatten the frequency response characteristics of the system.

23. The required 52 data sub-carriers are then extracted.

24. BPSK demodulation is then carried out.

25. The modulated values are then converted into bits.

26. Finally, the No. of errors are counted in order to estimate the BER of the system.

PLOTTING THE GRAPH:

27. The bit-energy to noise-density ratio is taken as the parameter for X-axis with units in decibels.

28. The bit-error-rate is taken as the parameter in the Y-axis.

29. The graph is plotted both for the Rayleigh theory & values obtained by Rayleigh simulation.

30. The BER performance of the system can be studied from the graph.

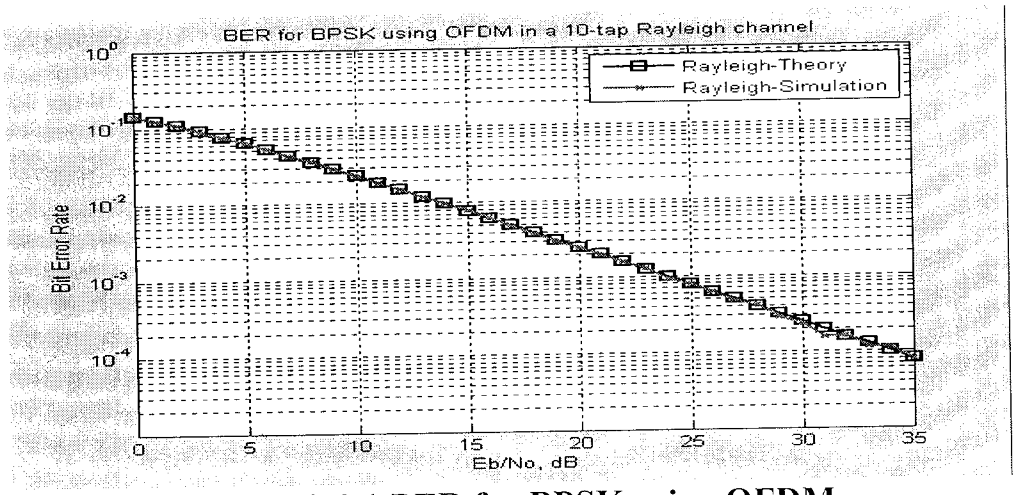


FIG 3.1 BER for BPSK using OFDM

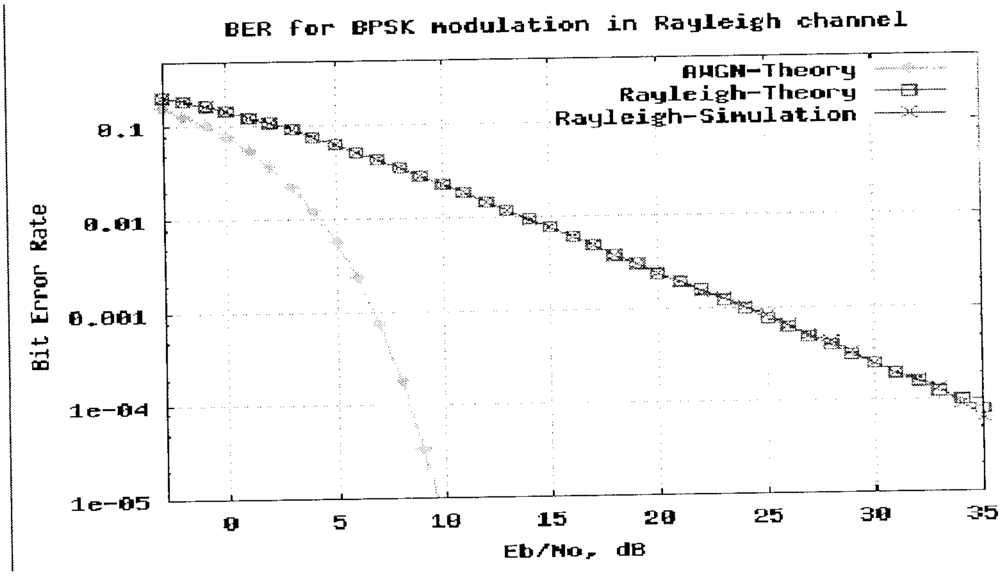


FIG 3.2 BER for BPSK without OFDM

3.4 COMPUTING THE BER FOR BPSK MODULATION IN THE PRESENCE OF A RAYLEIGH FADING CHANNEL

1. The no. of bits/ symbols is taken as 10^6.

TRANSMITTER SIDE:

2. The input bits are generated as a random stream of 1's & 0's with equal probability.

3. Then Binary Phase Shift Keying is done by assigning a value of -1 to all input stream 0's and +1 to all input stream 1's.

4. White Gaussian noise with 0db variation is then introduced.

5. In order to account for the multipath interference that may occur in transmission, a multipath channel is simulated. Here the channel is modeled as Rayleigh Channel in order to account for non line-of-sight propagation.

6. The channel and noise addition is then carried out.

7. Equalization is also done to flatten the frequency response characteristics.

RECEIVER SIDE:

8. Hard decision decoding is performed at the receiver.

9. Finally, the errors are counted in order to calculate the BER & study the performance.

CHAPTER 4

DESIGN OF THE OFDM SYSTEM

4.1 DESIGN BLOCK DIAGRAM

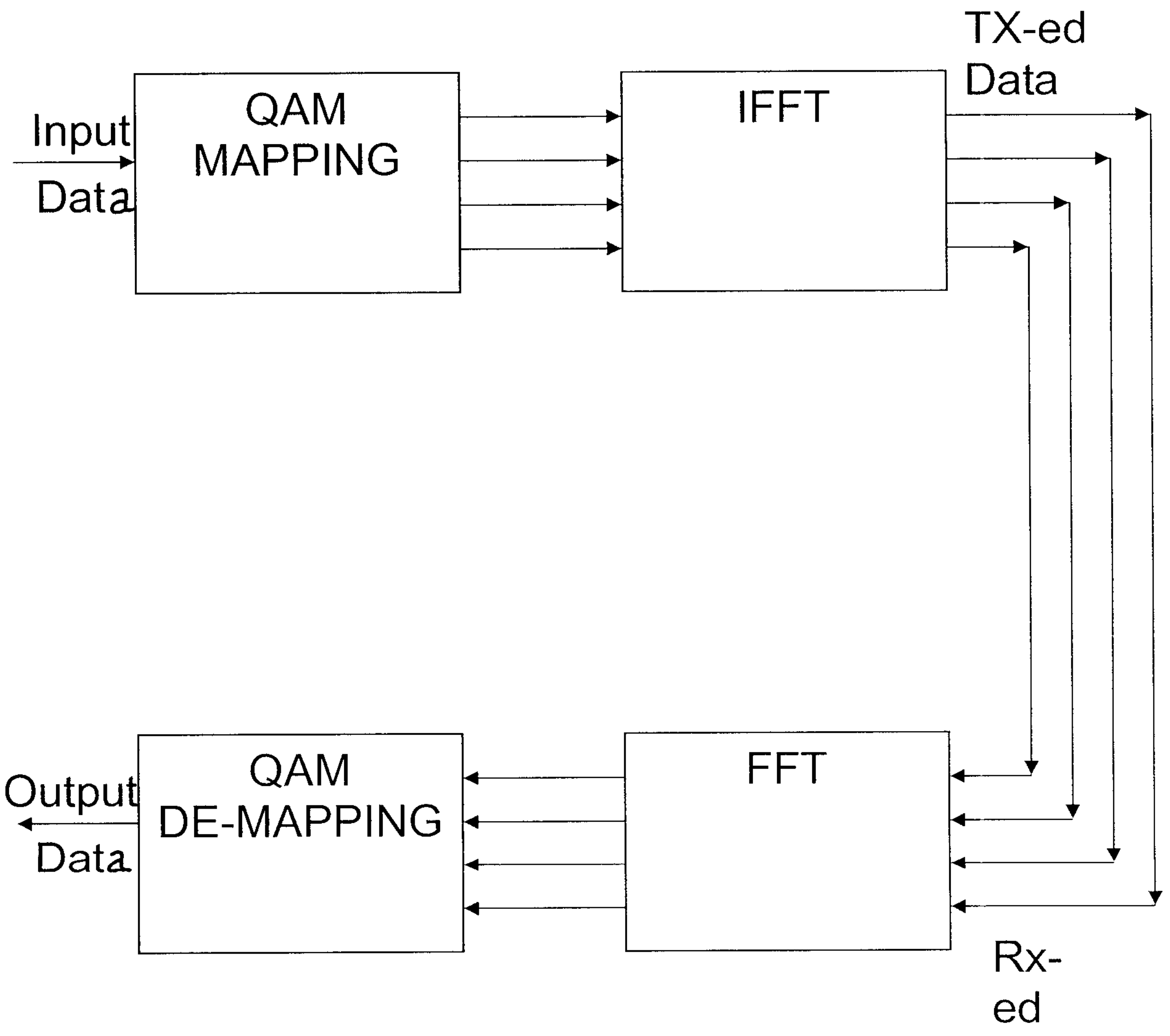


FIG 4.1 Design block diagram

4.2 QUADRATURE AMPLITUDE MODULATION:

QAM is both an analog and a digital modulation scheme. It conveys two analog message signals, or two digital bit streams, by changing (modulating) the amplitudes of two carrier waves, using the amplitude-shift keying (ASK) digital modulation scheme or amplitude modulation (AM) analog modulation scheme. These two waves, usually sinusoids, are out of phase with each other by 90° and are thus called quadrature carriers or quadrature components — hence the name of the scheme. The modulated waves are summed, and the resulting waveform is a combination of both phase-shift keying (PSK) and amplitude-shift keying (ASK), or in the analog case of phase modulation (PM) and amplitude modulation. In the digital QAM case, a finite number of at least two phases, and at least two amplitudes are used. PSK modulators are often designed using the QAM principle, but are not considered as QAM since the amplitude of the modulated carrier signal is constant.

Digital formats of QAM are often referred to as "Quantised QAM" and they are being increasingly used for data communications often within radio communications systems. Systems ranging from cellular technology through wireless systems including WiMAX, and Wi-Fi 802.11 use a variety of forms of QAM, and the use of QAM will only increase within the field of radio communications.

As with many digital modulation schemes, the constellation diagram is a useful representation. In QAM, the constellation points are usually arranged in a square grid with equal vertical and horizontal spacing, although other configurations are possible (e.g. Cross-QAM). Since in digital telecommunications the data are usually binary, the number of points in the grid is usually a power of 2 (2, 4, 8 ...). Since QAM is usually square, some of these are rare—the most common forms are 16-QAM, 64-QAM, 128-QAM and 256-QAM. By moving to a higher-order constellation, it is possible to transmit more bits per symbol. However, if the mean energy of the constellation is to remain the same (by way of making a fair comparison), the points must be closer together and are thus more susceptible to noise and other corruption; this results in a higher bit error rate and so higher-order QAM can deliver more data less reliably than lower-order QAM, for constant mean constellation energy.

If data-rates beyond those offered by 8-PSK are required, it is more usual to move to QAM since it achieves a greater distance between adjacent points in the I-Q plane by distributing the points more evenly. The complicating factor is that the points are no longer all the same amplitude and so the demodulator must now correctly detect both phase and amplitude, rather than just phase.

64-QAM and 256-QAM are often used in digital cable television and cable modem applications. In the US, 64-QAM and 256-QAM are the mandated modulation schemes for digital cable (see QAM tuner) as standardised by the SCTE in the standard ANSI/SCTE 07 2000. Many marketing people will refer to these as QAM-64 and QAM-256. In the UK, 16-QAM and 64-QAM are currently used for digital terrestrial television (Freeview and Top Up TV) and 256-QAM is planned for Freeview-HD.

Communication systems designed to achieve very high levels of spectral efficiency usually employ very dense QAM constellations. One example is the ITU-T G.hn standard for networking over existing home wiring (coaxial cable, phone lines and power lines), which employs constellations up to 4096-QAM (12 bits/symbol).

CONSTELLATION DIAGRAM FOR 16-pt QAM:

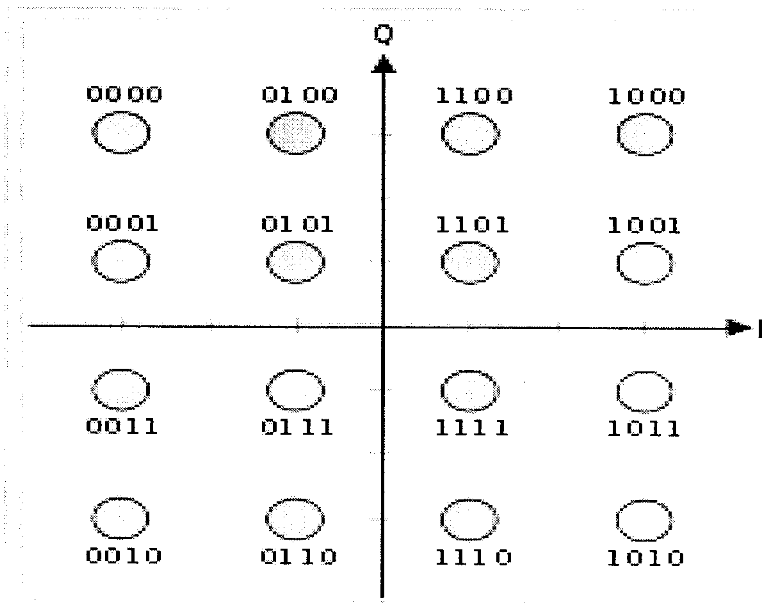


FIG 4.2 Constellation diagram for 16-pt QAM

VARIOUS TYPES OF MODULATION TECHNIQUES USED IN OFDM SYSTEMS:

OFDM Systems

- OFDM modulation consists of multiplexing QAM data symbols over a large number of orthogonal carriers

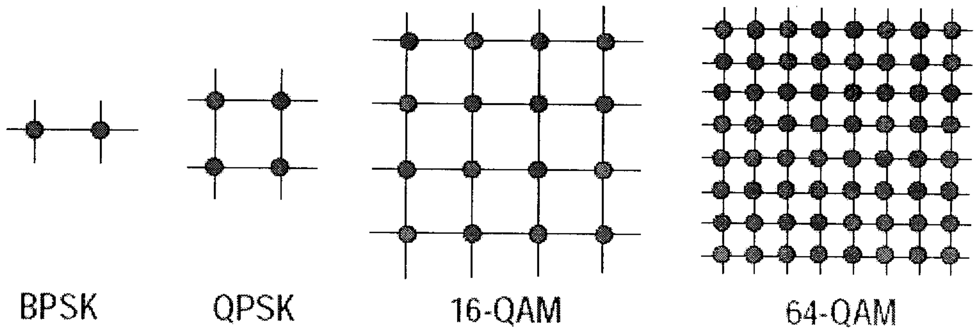


FIG 4.3 Types of OFDM systems

4.3 QAM - ADVANTAGES AND DISADVANTAGES:

Although QAM appears to increase the efficiency of transmission by utilising both amplitude and phase variations, it has a number of drawbacks. The first is that it is more susceptible to noise because the states are closer together so that a lower level of noise is needed to move the signal to a different decision point. Receivers for use with phase or frequency modulation are both able to use limiting amplifiers that are able to remove any amplitude noise and thereby improve the noise reliance. This is not the case with QAM. The second limitation is also associated with the amplitude component of the signal. When a phase or frequency modulated signal is amplified in a transmitter, there is no need to use linear amplifiers, whereas when using QAM that contains an amplitude component, linearity must be maintained. Unfortunately linear amplifiers are less efficient and consume more power, and this makes them less attractive for mobile applications.

4.4 FAST FOURIER TRANSFORMS

A fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. There are many distinct FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory. The fast Fourier Transform is a highly efficient procedure for computing the DFT of a finite series and requires less number of computations than that of direct evaluation of DFT. It reduces the computations by taking advantage of the fact that the calculation of the coefficients of the DFT can be carried out iteratively. Due to this, FFT computation technique is used in digital spectral analysis, filter simulation, autocorrelation and pattern recognition.

The FFT is based on decomposition and breaking the transform into smaller transforms and combining them to get the total transform. FFT reduces the computation time required to compute a discrete Fourier transform and improves the performance by a factor of 100 or more over direct evaluation of the DFT.

A DFT decomposes a sequence of values into components of different frequencies. This operation is useful in many fields but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly: computing a DFT of N points in the obvious way, using the definition, takes O(N 2) arithmetical operations, while an FFT can compute the same result in only O(N log N) operations.

The difference in speed can be substantial, especially for long data sets where N may be in the thousands or millions—in practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to N/log(N). This huge improvement made many DFT-based algorithms practical; FFTs are of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers.

The most well known FFT algorithms depend upon the factorization of N, but (contrary to popular misconception) there are FFTs with O(N log N) complexity for all N, even for prime N. Many FFT algorithms only depend on the fact that $e^{-\frac{2\pi i}{N}}$ is an Nth primitive root of unity, and thus can be applied to analogous transforms over any finite field, such as number-theoretic transforms.

The Fast Fourier Transform algorithms exploit the 2 basic properties of the twiddle factor - the symmetry property & periodicity property and reduces the number of complex multiplications required to perform DFT.

FFT algorithms are based on the fundamental principle of decomposing the computation of discrete Fourier Transform of a sequence of length N into successively smaller discrete Fourier transforms. There are basically two classes of FFT algorithms.

A) decimation-in-time

B) decimation-in-frequency.

In decimation-in-time, the sequence for which we need the DFT is successively divided into smaller sequences and the DFTs of these subsequences are combined in a certain pattern to obtain the required DFT of the entire sequence. In the decimation-in-frequency approach, the frequency samples of the DFT are decomposed into smaller and smaller subsequences in a similar manner.

4.5 Various FFT Algorithms:

The different types of FFT algorithms are classified by the different index maps of the input & output sequences.

1) Cooley- Tukey FFT Algorithm

   Classified into Decimation in Time (DIT) & Decimation In Frequency (DIF) Algorithms.

2) Good – Thomas FFT Algorithm (Prime-factor FFT algorithm)

3) Winograd FFT Algorithms

4) Bruun's FFT algorithm

4.6 Cooley-Tukey FFT Algorithm:

The Cooley-Tukey is the most universal of all FFT algorithms. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N1N2$ into many smaller DFTs of sizes $N1$ and $N2$, along with $O(N)$ multiplications by complex roots of unity traditionally called twiddle factors (after Gentleman and Sande, 1966).

This method (and the general idea of an FFT) was popularized by a publication of J. W. Cooley and J. W. Tukey in 1965, but it was later discovered (Heideman & Burrus, 1984) that those two authors had independently re-invented an algorithm known to Carl Friedrich Gauss around 1805 (and subsequently rediscovered several times in limited forms).

The most well-known use of the Cooley–Tukey algorithm is to divide the transform into two pieces of size N / 2 at each step, and is therefore limited to power-of-two sizes, but any factorization can be used in general (as was known to both Gauss and Cooley/Tukey). These are called the radix-2 and mixed-radix cases, respectively. Although the basic idea is recursive, most traditional implementations rearrange the algorithm to avoid explicit recursion. Also, because the Cooley–Tukey algorithm breaks the DFT into smaller DFTs, it can be combined arbitrarily with any other algorithm for the DFT such as the Prime-factor FFT algorithm, Bruun's FFT algorithm, Rader's FFT algorithm or Bluestein's FFT algorithm.

4.7 Illustration of the Cooley-Tukey algorithm for the radix-2 decimation in Time case:

A radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the Cooley–Tukey algorithm, although highly optimized Cooley–Tukey implementations typically use other forms of the algorithm. Radix-2 DIT divides a DFT of size N into two interleaved DFTs (hence the name "radix-2") of size $N/2$ with each recursive stage.

The discrete Fourier transform (DFT) is defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk},$$

where k is an integer ranging from 0 to $N - 1$.

Radix-2 DIT first computes the DFTs of the even-indexed inputs x_{2m} ($x_0, x_2, \ldots, x_{N-2}$) and of the odd-indexed inputs x_{2m+1} ($x_1, x_3, \ldots, x_{N-1}$), and then combines those two results to produce the DFT of the whole sequence. This idea can then be performed recursively to reduce the overall runtime to O(N log N). This simplified form assumes that N is a power of two; since the number of sample points N can usually be chosen freely by the application, this is often not an important restriction.

The Radix-2 DIT algorithm rearranges the DFT of the function xn into two parts: a sum over the even-numbered indices $n = 2m$ and a sum over the odd-numbered indices $n = 2m + 1$

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}.$$

One can factor a common multiplier $e^{-\frac{2\pi i}{N}k}$ out of the second sum, as shown in the equation below. It is then clear that the two sums are the DFT of the even-indexed part $x2m$ and the DFT of odd-indexed part $x2m + 1$ of the function xn. Denote the DFT of the ***E***ven-indexed inputs $x2m$ by Ek and the DFT of the Odd-indexed inputs $x2m + 1$ by Ok and we obtain:

`

$$X_k = \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk}}_{\text{DFT of even-indexed part of } x_m} + e^{-\frac{2\pi i}{N}k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk}}_{\text{DFT of odd-indexed part of } x_m} = E_k + e^{-\frac{2\pi i}{N}k} O_k.$$

However, these smaller DFTs have a length of *N*/2, so we need compute only *N*/2 outputs: thanks to the periodicity properties of the DFT, the outputs for $N/2 \leq k < N$ from a DFT of length *N*/2 are identical to the outputs for $0 \leq k < N/2$. That is, *Ek* + *N* / 2 = *Ek* and *Ok* + *N* / 2 = *Ok*. The phase factor exp[− 2π*ik* / *N*] (called a twiddle factor) obeys the relation: exp[− 2π*i*(*k* + *N* / 2) / *N*] = *e* − π*i*exp[− 2π*ik* / *N*] = − exp[− 2π*ik* / *N*], flipping the sign of the *Ok* + *N* / 2 terms. Thus, the whole DFT can be calculated as follows:

$$X_k = \begin{cases} E_k + e^{-\frac{2\pi i}{N}k} O_k & \text{if } k < N/2 \\ E_{k-N/2} - e^{-\frac{2\pi i}{N}(k-N/2)} O_{k-N/2} & \text{if } k \geq N/2. \end{cases}$$

This result, expressing the DFT of length *N* recursively in terms of two DFTs of size *N*/2, is the core of the radix-2 DIT fast Fourier transform. The algorithm gains its speed by re-using the results of intermediate computations to compute multiple DFT outputs. Note that final outputs are obtained by a +/− combination of *Ek* and *Ok*exp(− 2π*ik* / *N*), which is simply a size-2 DFT; when this is generalized to larger radices below, the size-2 DFT is replaced by a larger DFT (which itself can be evaluated with an FFT).

This process is an example of the general technique of divide and conquer algorithms; in many traditional implementations, however, the explicit recursion is avoided, and instead one traverses the computational tree in breadth-first fashion.

The above re-expression of a size-N DFT as two size-N/2 DFTs is sometimes called the Danielson–Lanczos lemma, since the identity was noted by those two authors in 1942(influenced by Runge's 1903 work). They applied their lemma in a "backwards" recursive fashion, repeatedly doubling the DFT size until the transform spectrum converged.

4.8 Butterfly structures for FFT:

In the context of fast Fourier transform algorithms, a butterfly is a portion of the computation that combines the results of smaller discrete Fourier transforms (DFTs) into a larger DFT, or vice versa (breaking a larger DFT up into subtransforms). The name "butterfly" comes from the shape of the data-flow diagram in the radix-2 case, as described below. The same structure can also be found in the Viterbi algorithm, used for finding the most likely sequence of hidden states.

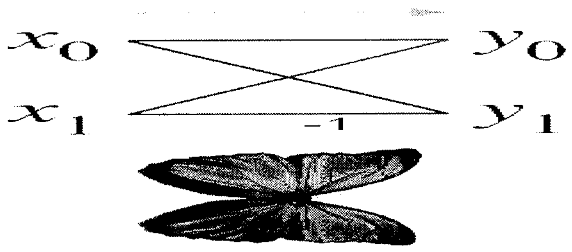


FIG 4.4 Radix-2 butterfly structure

Most commonly, the term "butterfly" appears in the context of the Cooley–Tukey FFT algorithm, which recursively breaks down a DFT of composite size n = rm into r smaller transforms of size m where r is the "radix" of the transform.

These smaller DFTs are then combined with size-r butterflies, which themselves are DFTs of size r (performed m times on corresponding outputs of the sub-transforms) pre-multiplied by roots of unity (known as twiddle factors). This is the "decimation in time" case; one can also perform the steps in reverse, known as "decimation in frequency", where the butterflies come first and are post-multiplied by twiddle factors.

The figure illustrates the Data-flow diagram connecting the inputs x (left) to the outputs y that depend on them (right) for a "butterfly" step of a radix-2 Cooley–Tukey FFT. This diagram resembles a butterfly (as in the Morpho butterfly shown for comparison), hence the name.

In the case of the radix-2 Cooley–Tukey algorithm, the butterfly is simply a DFT of size-2 that takes two inputs $(x0, x1)$ (corresponding outputs of the two sub-transforms) and gives two outputs $(y0, y1)$ by the formula.

$$y_0 = x_0 + x_1$$

$$y_1 = x_0 - x_1.$$

where k is an integer depending on the part of the transform being computed.

More specifically, a decimation-in-time FFT algorithm on $n = 2^p$ inputs with respect to a primitive n-th root of unity $\omega = \exp(2\pi i / n)$ relies on $O(n \log n)$ butterfly form.

4.9 Radix-2 FFTs:

A) Decimation-in-time Algorithm:

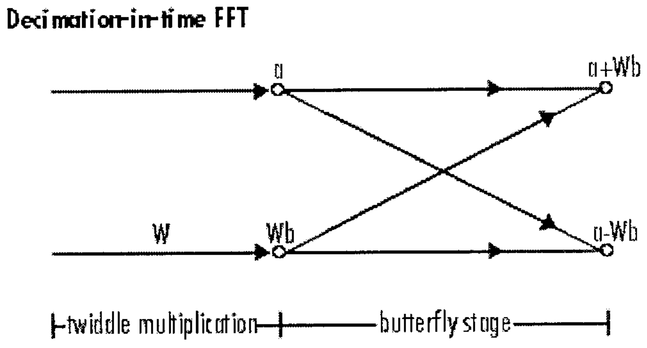


FIG 4.5 Decimation in time FFT

In the DIT algorithm, the twiddle multiplication is performed before the butterfly stage whereas for the DIF algorithm, the twiddle multiplication comes after the Butterfly stage.

B) Decimation-in-Frequency Algorithm:

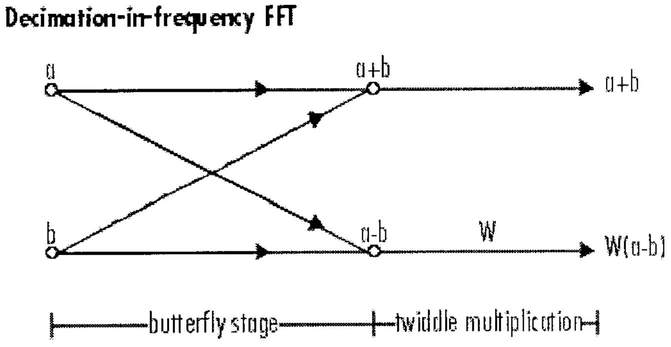


FIG 4.6 Decimation in frequency FFT

4.10 HARDWARE DESCRIPTION LANGUAGES:

Hardware Description Language (HDL) is a language that can describe the behavior and structure of electronic system, but it is particularly suited as a language to describe the structure and the behavior of the digital electronic hardware design, such as ASICs and FPGAs as well as conventional circuits. HDL can be used to describe electronic hardware at many different levels of abstraction such as Algorithm, Register transfer level (RTL) and Gate level. Algorithm is un synthesizable, RTL is the input to the synthesis, and Gate Level is the input from the synthesis. It is often reported that a large number of ASIC designs meet their specification first time, but fail to work when plunged into a system. HDL allows this issue to be addressed in two ways, a HDL specification can be executed in order to achieve a high level of confidence in its correctness before commencing design and may simulate one specification for a part in the wider system context(Eg:- Printed Circuited Board Simulation). This depends upon how accurately the specialization handles aspects such as timing and initialization.

ADVANTAGES OF HDL:

A design methodology that uses HDLs has several fundamental advantages over traditional Gate Level Design Methodology. The following are some of the advantages:

- One can verify functionality early in the design process and immediately simulate the design written as a HDL description. Design simulation at this high level, before implementation at the Gate Level allows testing architectural and designing decisions.

- FPGA synthesis provides logic synthesis and optimization, so one can automatically convert a Verilog HDL description to gate level implementation in a given technology.
- HDL descriptions provide technology independent documentation of a design and its functionality. A HDL description is more easily read and understood than a net-list or schematic description.
- HDLs typically support a mixed level description where structural or net-list constructs can be mixed with behavioral or algorithmic descriptions. With this mixed level capabilities one can describe system architectures at a high level or gate level implementation.

4.11 VHDL:

VHDL (VHSIC hardware description language; VHSIC: very-high-speed integrated circuit) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

VHDL is a fairly general-purpose language, and it doesn't require a simulator on which to run the code. There are many VHDL compilers, which build executable binaries. It can read and write files on the host computer, so a VHDL program can be written that generates another VHDL program to be incorporated in the design being developed. Because of this general-purpose nature, it is possible to use VHDL to write a testbench that verifies the functionality of the design using files on the host computer to define stimuli, interacts with the user, and compares results with those expected.

It is relatively easy for an inexperienced developer to produce code that simulates successfully but that cannot be synthesized into a real device, or is too large to be practical. One particular pitfall is the accidental production of transparent latches rather than D-type flip-flops as storage elements.

VHDL is not a case sensitive language. One can design hardware in a VHDL IDE (such as Xilinx ISE or Altera Quartus) to produce the RTL schematic of the desired circuit. After that, the generated schematic can be verified using simulation software (such as ModelSim) which shows the waveforms of inputs and outputs of the circuit after generating the appropriate testbench. To generate an appropriate testbench for a particular circuit or VHDL code, the inputs have to be defined correctly. For example, for clock input, a loop process or an iterative statement is required.

The key advantage of VHDL when used for systems design is that it allows the behavior of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).

Another benefit is that VHDL allows the description of a concurrent system (many parts, each with its own sub-behavior, working together at the same time). VHDL is a Dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.

A final point is that when a VHDL model is translated into the "gates and wires" that are mapped onto a programmable logic device such as a CPLD or FPGA, then it is the actual hardware being configured.

4.12 VHDL description of 4-point FFT & IFFT Blocks

1) The Decimation-In-Time Algorithm is used to design the 4-point FFT and IFFT structures.

2) The entire structure is developed by taking into account the operation of complex numbers. Hence we use 8- input values to individually give the values for real part & imaginary parts.

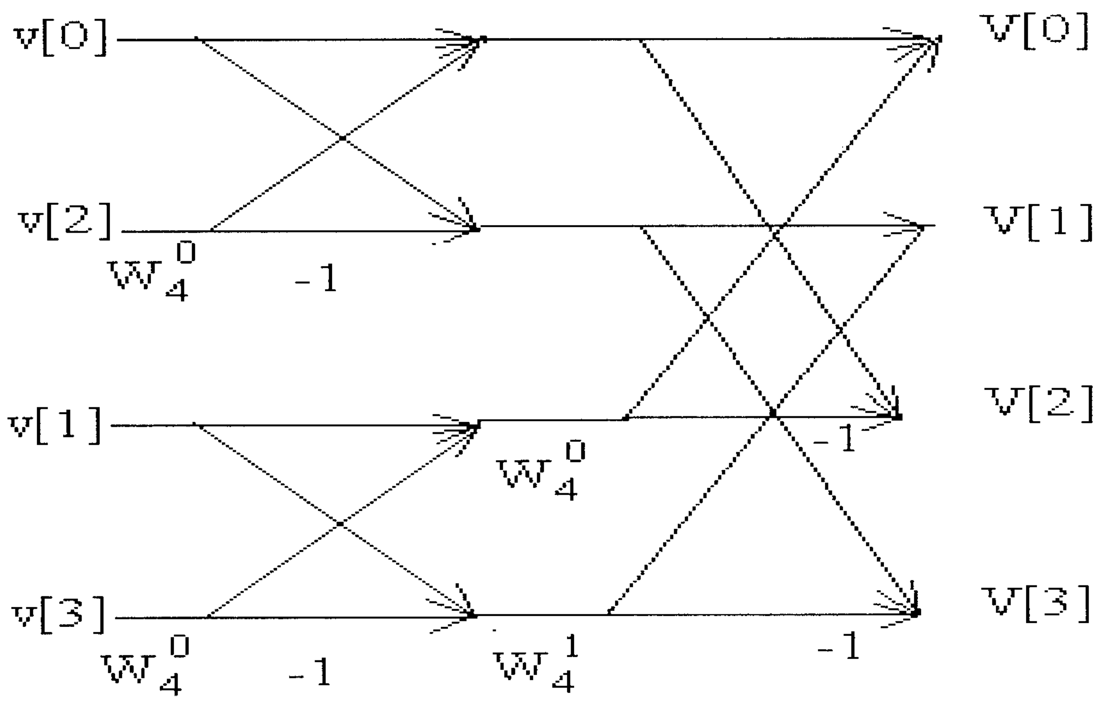


FIG 4.7 4-point Decimation-In-Time FFT Algorithm

4.13 IFFT ALGORITHM:

1) The FFT algorithm can be itself be used to compute the IFFT

2) First take the complex conjugate of the input values & multiply by N

3) This value should in turn be given to the input of the FFT Block

4) The complex conjugate is taken for the values obtained at the output of the FFT algorithm and then multiplied with N.

5) This gives the IFFT output values.

4.14 DESIGN OF A GENERAL RADIX-2 FFT USING VHDL:

As we move to higher-point FFTs, the structure for computing the FFT becomes more complex and the need for an efficient complex multiplier to be incorporated within the butterfly structure arises. Hence we propose an algorithm for an efficient complex multiplier that overcomes the complication of using complex numbers throughout the process.

A radix-2 FFT can be efficiently implemented using a butterfly processor which includes, besides the butterfly itself , an additional complex multiplier for the twiddle factors.

A radix-2 butterfly processor consists of a complex adder, a complex subtraction, and a complex multiplier for the twiddle factors. The complex multiplication with the twiddle factor is often implemented with four real multiplications and 2 add / subtract operations. However it is also possible to build the complex multiplier with only 3 real multiplications and 3 add / subtract operations because one operand is precomputed.

Normal Complex Operation:

$$(X+jY)\,(C+jS) = CX + jSX + jCY - YS$$

$$= CX - YS + j\,(SX + CY)$$

Real Part $R = CX - YS$

Imaginary Part $I = SX + CY$

4.15 EFFICIENT COMPLEX MULTIPLIER:

Consider the complex twiddle factor multiplication

$(X+jY)\,(C+jS) = R+jI$

C & S are the real & imaginary parts of the twiddle factor which are precomputed & stored in a table.

Also 2 other co-efficients are stored. C+S, C-S

With these 3 precomputed factors we first compute

$E = X-Y$, and then $Z = C*E = C*(X-Y)$

We can thus compute the final product using

$R = (C-S)Y * Y+Z$ $\qquad$ $I = (C+S) * X-Z$

WORKING OF THE ALGORITHM:

$$R = (C-S)Y + C(X-Y)$$

$$= CY-SY + CX - CY = CX - SY$$

$$I = (C+S)X - C(X-Y)$$

$$= CX + SX - CX + CY = CY + SX$$

The algorithm uses 3 multiplications, 1 addition, & 2 subtractions at the cost of an additional 3rd table.

Using the twiddle factor multiplier that has been developed, it is possible to design a butterfly processor for a radix-2 Cooley-Tukey FFT. Hence this basic structure of radix-2 FFT can be used as a building block to construct higher N-point FFTs. This structure has been developed as an extension to provide for the computation of higher value index FFTs.

CHAPTER 5

FPGA IMPLEMENTATION

5.1 Introduction

A **field-programmable gate array (FPGA)** is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare). FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, partial re-configuration of the portion of the design[1] and the low non-recurring engineering costs relative to an ASIC design (not withstanding the generally higher unit cost), offer advantages for many applications.

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

5.2 FPGA ARCHITECTURE:

The most common FPGA architecture consists of an array of configurable logic blocks (CLBs), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.

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

A classic FPGA logic block consists of a 4-input lookup table (LUT), and a flip-flop, as shown below. In recent years, manufacturers have started moving to 6-input LUTs in their high performance parts, claiming increased performance.f

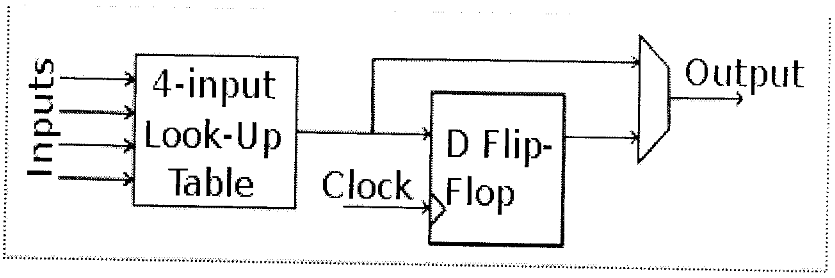


FIG 5.1 Typical logic block in FPGA

There is only one output, which can be either the registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed.

For this example architecture, the locations of the FPGA logic block pins are shown below.

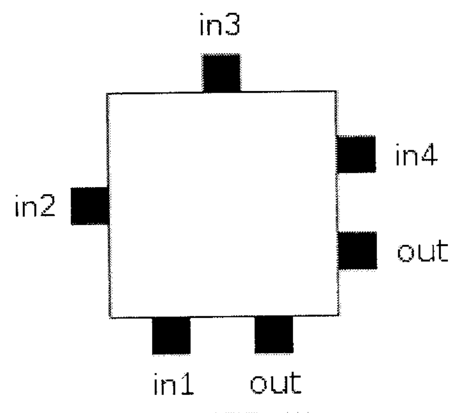


FIG 5.2 Logic Block Pin Locations

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

Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it. Similarly, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it. Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.Whenever a vertical and a horizontal channel intersect, there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure below illustrates the connections in a switch box.

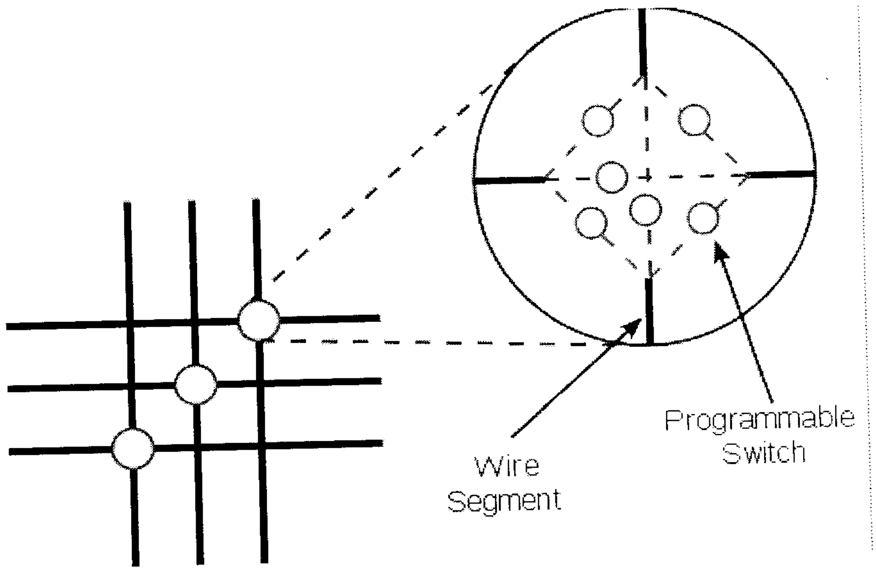


FIG 5.3 Switch box topology

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

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

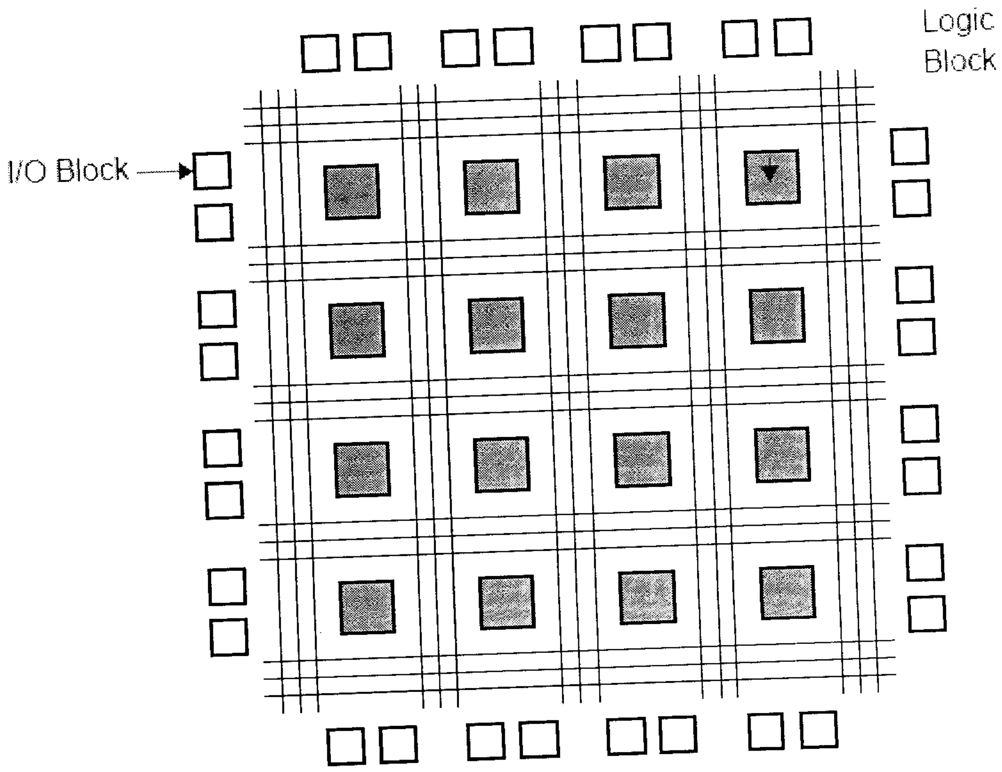


FIG 5.4 Structure of an FPGA

5.3 FPGA DESIGN AND PROGRAMMING

Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA.

Going from schematic/HDL source files to actual configuration. The source files are fed to a software suite from the FPGA/CPLD vendor that through different steps will produce a file. This file is then transferred to the FPGA/CPLD via a serial interface (JTAG) or to an external memory device like an EEPROM.

The most common HDLs are VHDL and Verilog, although in an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level through the introduction of alternative languages.

To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process. These predefined circuits are commonly called IP cores, and are available from FPGA vendors and third-party IP suppliers (rarely free, and typically released under proprietary licenses). Other predefined circuits are available from developer communities such as OpenCores (typically released under free and open source licenses such as the GPL, BSD or similar license).

In a typical design flow, an FPGA application developer will simulate the design at multiple stages throughout the design process. Initially the RTL description in VHDL or Verilog is simulated by creating test benches to simulate the system and observe results. Then, after the synthesis engine has mapped the design to a netlist, the netlist is translated to a gate level description where simulation is repeated to confirm the synthesis proceeded without errors. Finally the design is laid out in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist.

5.4 Basic process technology types

- SRAM - based on static memory technology. In-system programmable and re-programmable. Requires external boot devices. CMOS.
- Antifuse - One-time programmable. CMOS.
- PROM - Programmable Read-Only Memory technology. One-time programmable because of plastic packaging.
- EPROM - Erasable Programmable Read-Only Memory technology. One-time programmable but with window, can be erased with ultraviolet (UV) light. CMOS.
- EEPROM - Electrically Erasable Programmable Read-Only Memory technology. Can be erased, even in plastic packages. Some, but not all, EEPROM devices can be in-system programmed. CMOS.
- Flash - Flash-erase EPROM technology. Can be erased, even in plastic packages. Some, but not all, flash devices can be in-system programmed. Usually, a flash cell is smaller than an equivalent EEPROM cell and is therefore less expensive to manufacture. CMOS.
- Fuse - One-time programmable. Bipolar.

5.5 Major manufacturers

Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. Together, they control over 80 percent of the market, with Xilinx alone representing over 50 percent.

Xilinx also provides free Windows and Linux design software, while Altera provides free Windows tools; the Linux tools are only available via a rental scheme.

Other competitors include Lattice Semiconductor (SRAM based with integrated configuration Flash, instant-on, low power, live reconfiguration), Actel (antifuse, flash-based, mixed-signal), SiliconBlue Technologies (low power), Achronix (RAM based, 1.5 GHz fabric speed), and QuickLogic (handheld focused CSSP, no general purpose FPGAs).

In March 2010, two FPGA companies that had previously worked in stealth mode, announced their new FPGA technology: Tabula and Tier Logic.

5.6 DESIGN FLOW

The steps are listed below.

1. Design Entry: Enter the design into an ASIC design system either using a Hardware Description Language Or schematic entry.
2. Logic synthesis: Use an HDL and a logic synthesis tool to produce a netlist, a description of the logic cells and their connections.
3. System partitioning: divide large system into ASIC-sized pieces.
4. Pre-layout simulation: check to see if the design functions properly.

5. Floor planning: Arrange the blocks of the net-list on the chip.
6. Placement: Decide the locations of cells in the block.
7. Looping: Make the connections cells and the blocks.
8. Extraction: Determine the resistance and capacitance of the interconnect.
9. Post-layout simulation: check to see if the design still works with the added loads of the interconnect.

5.7 SYNTHESIS

A process that starts from high-level logic abstraction and automatically creates a net list (connection details between individual components)using a library. The synthesizing unit includes the selection of device, speed-grade for its internal operation. After synthesis the output generated is "net-list" file named with the extension of ".xnf".

5.8 SIMULATION

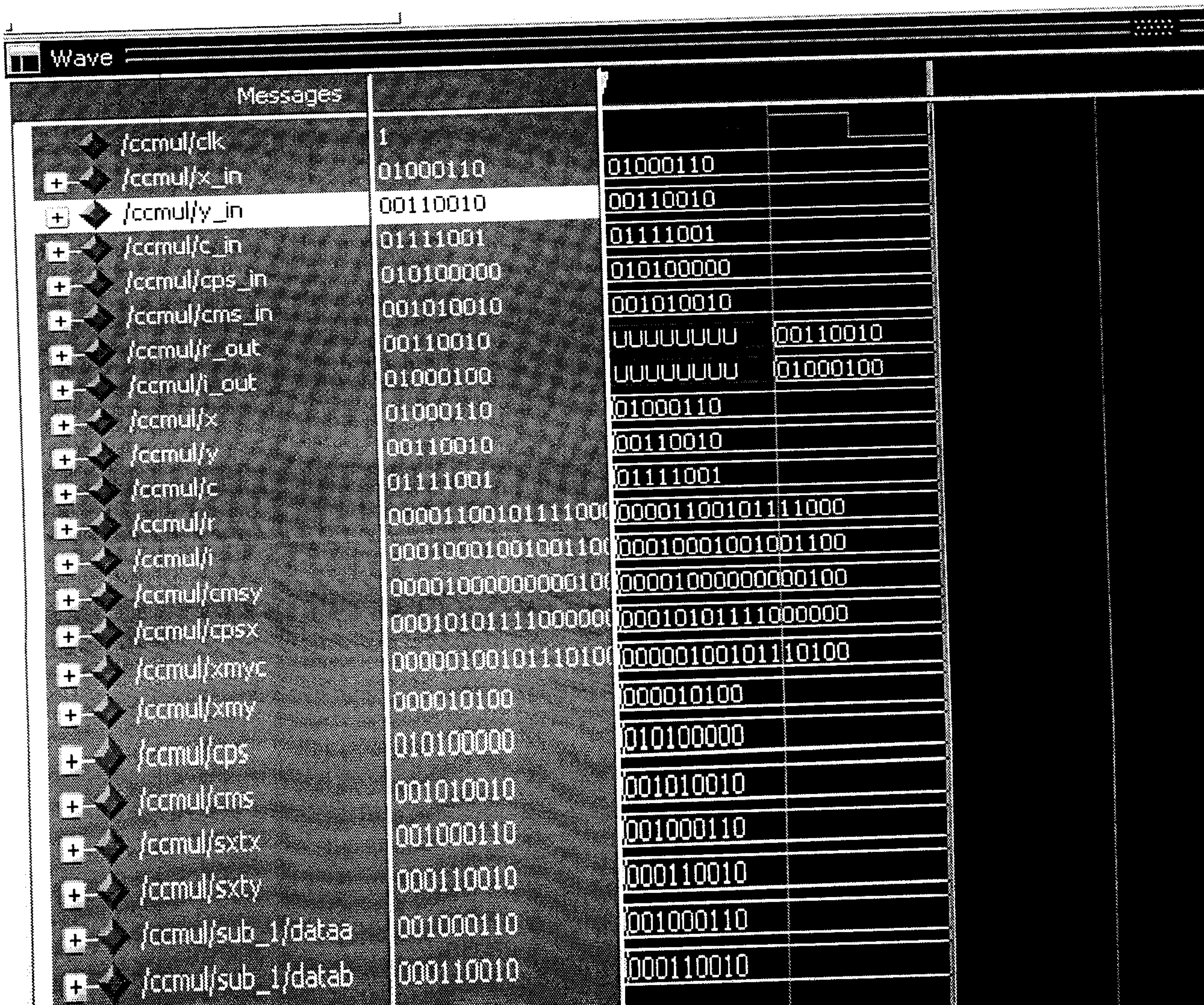
This simulation is otherwise called as waveform editor. Digital oscilloscopes were too costly even with low channel capacity and it is not possible to view all signals getting transferred from one component to another .This problem is faithfully solved by this simulation part of the tool. Here one could select all signals right from the initial part of the design to end pin of the package and this can be visualized. Simulation takes part in input of net-list file and creates a file with the extension ".edn". This simulation in addition to the wave form presentation, it adds some features like simulation and timing simulation.

CHAPTER 6

RESULTS AND DISCUSSION

6.1 SIMULATION RESULTS OBTAINED IN MODELSIM

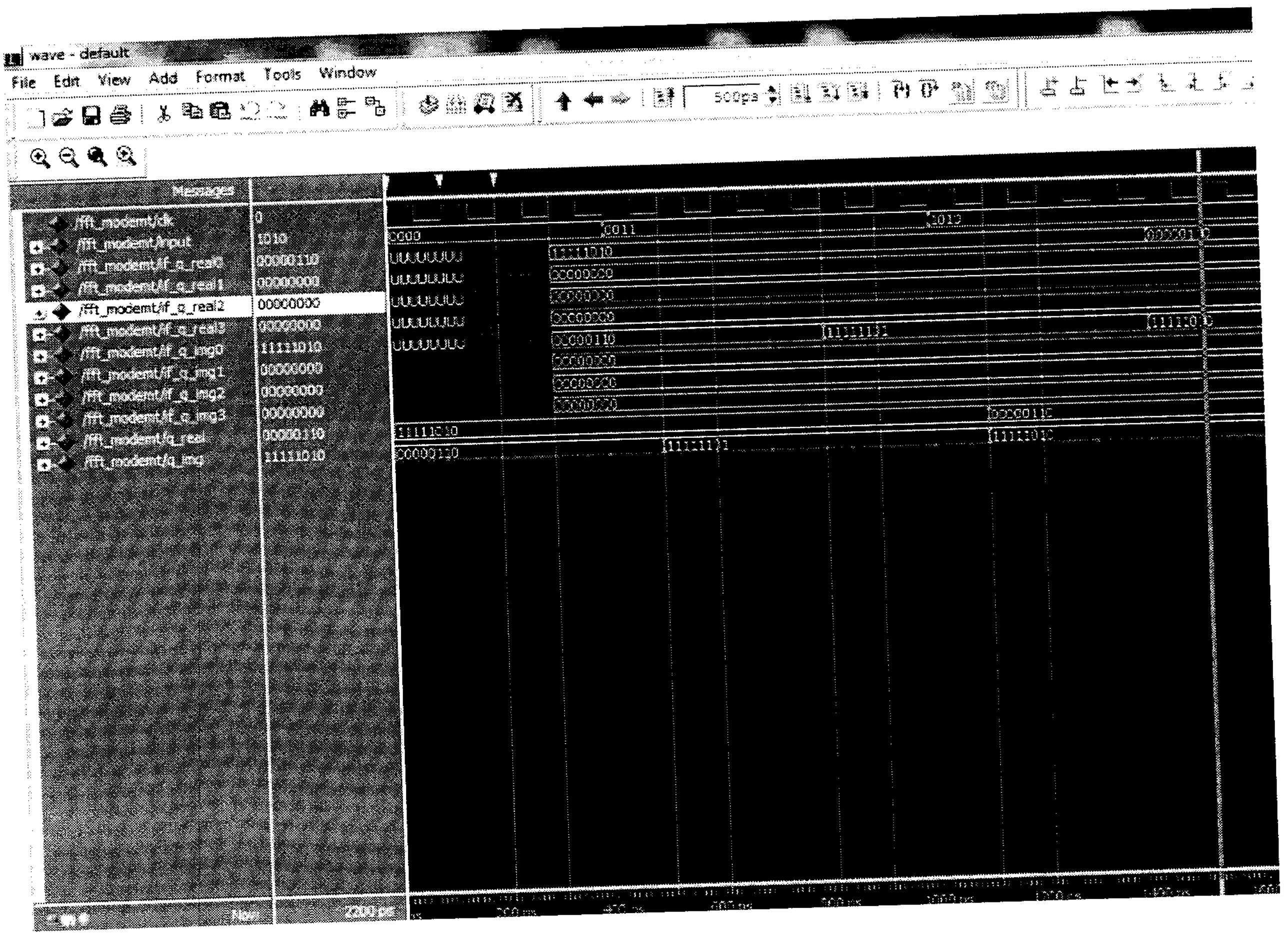
6.1.1 SIMULATION RESULT OBTAINED FOR COMPLEX CONJUGATE MULTIPLIER: (using ALTERA Modelsim)



```
Wave
Messages
/ccmul/clk              1
/ccmul/x_in             01000110            01000110
/ccmul/y_in             00110010            00110010
/ccmul/c_in             01111001            01111001
/ccmul/cps_in           010100000           010100000
/ccmul/cms_in           001010010           001010010
/ccmul/r_out            00110010            UUUUUUUU    00110010
/ccmul/i_out            01000100            UUUUUUUU    01000100
/ccmul/x                01000110            01000110
/ccmul/y                00110010            00110010
/ccmul/c                01111001            01111001
/ccmul/r                0000110010111100    0000110010111000
/ccmul/i                0001000100100110    0001000100100110
/ccmul/cmsy             0000100000000010    0000100000000100
/ccmul/cpsx             0001010111100000    0001010111000000
/ccmul/xmyc             0000010010111010    0000010010111010
/ccmul/xmy              000010100           000010100
/ccmul/cps              010100000           010100000
/ccmul/cms              001010010           001010010
/ccmul/sxtx             001000110           001000110
/ccmul/sxty             000110010           000110010
/ccmul/sub_1/dataa      001000110           001000110
/ccmul/sub_1/datab      000110010           000110010
```

6.1.2 SIMULATION RESULT OBTAINED FOR OFDM TRANSMITTER:

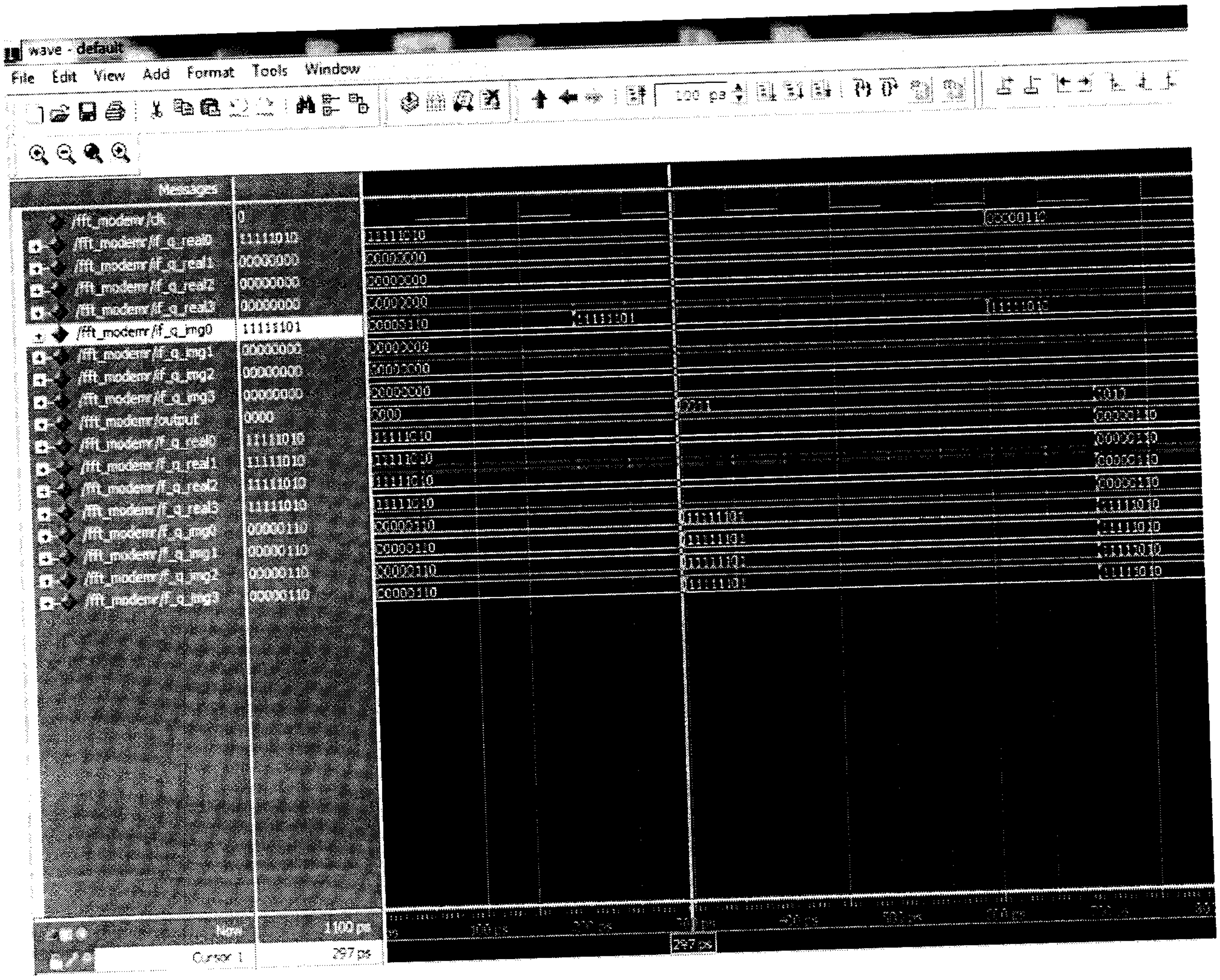
(Comprising the QAM & Inverse FFT Blocks)- using XILINX Modelsim



```
/fft_modemt/clk          0
/fft_modemt/input        1010
/fft_modemt/if_q_real0   00000110
/fft_modemt/if_q_real1   00000000
/fft_modemt/if_q_real2   00000000
/fft_modemt/if_q_real3   00000000
/fft_modemt/if_q_img0    11111010
/fft_modemt/if_q_img1    00000000
/fft_modemt/if_q_img2    00000000
/fft_modemt/if_q_img3    00000000
/fft_modemt/q_real       00000110
/fft_modemt/q_img        11111010
```

6.1.3 SIMULATION RESULT OBTAINED FOR OFDM RECEIVER:

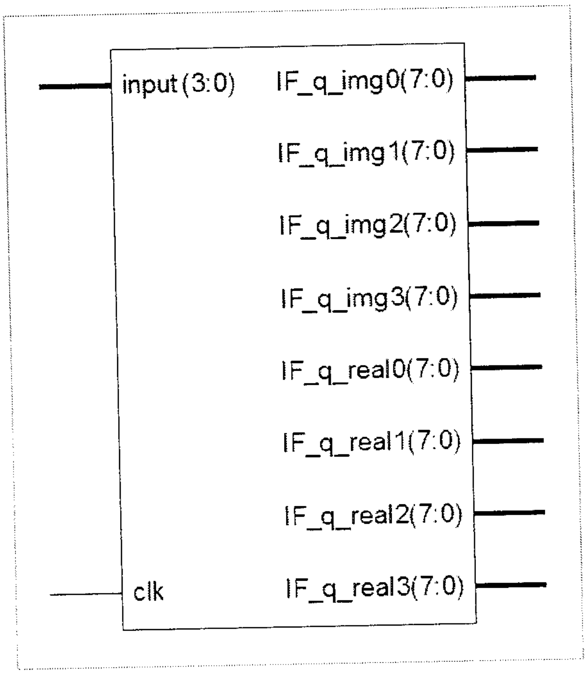
(Comprising the Inverse-QAM & FFT Blocks) - Using XILINX Modelsim



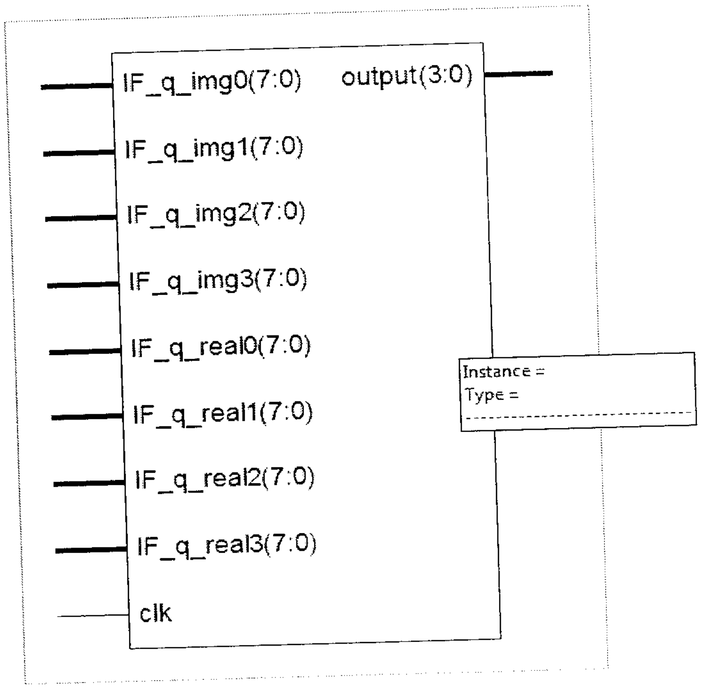
```
/fft_modemr/clk          0
/fft_modemr/if_q_real0   11111010
/fft_modemr/if_q_real1   00000000
/fft_modemr/if_q_real2   00000000
/fft_modemr/if_q_real3   00000000
/fft_modemr/if_q_img0    11111101
/fft_modemr/if_q_img1    00000000
/fft_modemr/if_q_img2    00000000
/fft_modemr/if_q_img3    00000000
/fft_modemr/output       0000
/fft_modemr/f_q_real0    11111010
/fft_modemr/f_q_real1    11111010
/fft_modemr/f_q_real2    11111010
/fft_modemr/f_q_real3    11111010
/fft_modemr/f_q_img0     00000110
/fft_modemr/f_q_img1     00000110
/fft_modemr/f_q_img2     00000110
/fft_modemr/f_q_img3     00000110

Now        1100 ps
Cursor 1   297 ps
```

6.2 RTL SCHEMATIC

6.2.1 RTL SCHEMATIC FOR OFDM TRANSMITTER



6.2.2 RTL SCEMATIC FOR OFDM RECEIVER



6.3 SYNTHESIS REPORT

6.3.1 SYNTHESIS REPORT OBTAINED FOR OFDM TRANSMITTER

```
Release 8.1i - xst I.24

Copyright (c) 1995-2005 Xilinx, Inc.  All rights reserved.
--> Parameter TMPDIR set to ./xst/projnav.tmp
CPU : 0.00 / 0.16 s | Elapsed : 0.00 / 0.00 s

--> Parameter xsthdpdir set to ./xst
CPU : 0.00 / 0.16 s | Elapsed : 0.00 / 0.00 s

--> Reading design: FFT_modemt.prj

TABLE OF CONTENTS
  1) Synthesis Options Summary
  2) HDL Compilation
  3) HDL Analysis
  4) HDL Synthesis
     4.1) HDL Synthesis Report
  5) Advanced HDL Synthesis
     5.1) Advanced HDL Synthesis Report
  6) Low Level Synthesis
  7) Final Report
     7.1) Device utilization summary
     7.2) TIMING REPORT

=========================================================================
*                Synthesis Options Summary                    *
=================================================================
===============
---- Source Parameters
Input File Name                    : "FFT_modemt.prj"
Input Format                     : mixed
Ignore Synthesis Constraint File   : NO

---- Target Parameters
Output File Name                     : "FFT_modemt"
Output Format                     : NGC
Target Device                     : xc2s200-6-pq208

---- Source Options
Top Module Name                        : FFT_modemt
Automatic FSM Extraction              : YES
FSM Encoding Algorithm                : Auto
FSM Style                        : lut
RAM Extraction                       : Yes
RAM Style                          : Auto
ROM Extraction                       : Yes
Mux Style                        : Auto
Decoder Extraction                    : YES
Priority Encoder Extraction            : YES
Shift Register Extraction            : YES
Logical Shifter Extraction            : YES
XOR Collapsing                      : YES
ROM Style                          : Auto
Mux Extraction                       : YES
```

```
Resource Sharing                  : YES
Multiplier Style                  : lut
Automatic Register Balancing      : No

---- Target Options
Add IO Buffers                    : YES
Global Maximum Fanout             : 100
Add Generic Clock Buffer(BUFG)    : 4
Register Duplication              : YES
Slice Packing                     : YES
Pack IO Registers into IOBs       : auto
Equivalent register Removal       : YES

---- General Options
Optimization Goal                 : Speed
Optimization Effort               : 1
Keep Hierarchy                    : NO
RTL Output                        : Yes
Global Optimization               : AllClockNets
Write Timing Constraints          : NO
Hierarchy Separator               : /
Bus Delimiter                     : <>
Case Specifier                    : maintain
Slice Utilization Ratio           : 100
Slice Utilization Ratio Delta     : 5

---- Other Options
lso                               : FFT_modemt.lso
```

```
`

Read Cores                    : YES
cross_clock_analysis             : NO
verilog2001                   : YES
safe_implementation              : No
Optimize Instantiated Primitives   : NO
tristate2logic                 : Yes
use_clock_enable                 : Yes
use_sync_set                   : Yes
use_sync_reset                  : Yes

=========================================================================
===============


=========================================================================
===============
*                    HDL Compilation                    *
=========================================================================
===============
Compiling vhdl file "C:/Xilinx/transmit/newQAM_16.vhd" in Library work.
Entity <qam_16> compiled.
Entity <qam_16> (Architecture <bhv>) compiled.
Compiling vhdl file "C:/Xilinx/transmit/IFFT_4.vhd" in Library work.
Entity <IFFT_4> compiled.
Entity <IFFT_4> (Architecture <bhv>) compiled.
Compiling vhdl file "C:/Xilinx/transmit/FFT_modemt.vhd" in Library work.
Entity <FFT_modemt> compiled.
Entity <FFT_modemt> (Architecture <bhv>) compiled.
```

```
=========================================================================
*                       HDL Analysis                                    *
=========================================================================
Analyzing Entity <FFT_modemt> (Architecture <bhv>).
Entity <FFT_modemt> analyzed. Unit <FFT_modemt> generated.

Analyzing Entity <qam_16> (Architecture <bhv>).
INFO:Xst:1561 - "C:/Xilinx/transmit/newQAM_16.vhd" line 83: Mux is complete
: default of case is discarded
Entity <qam_16> analyzed. Unit <qam_16> generated.

Analyzing Entity <IFFT_4> (Architecture <bhv>).
Entity <IFFT_4> analyzed. Unit <IFFT_4> generated.


=========================================================================
*                       HDL Synthesis                                   *
=========================================================================

Synthesizing Unit <IFFT_4>.
    Related source file is "C:/Xilinx/transmit/IFFT_4.vhd".

    Found 8-bit register for signal <iy1_img>.
    Found 8-bit register for signal <iy0_real>.
    Found 8-bit register for signal <iy2_img>.
    Found 8-bit register for signal <iy1_real>.
Found 8-bit register for signal <iy2_real>.
Found 8-bit register for signal <iy3_img>.
Found 8-bit register for signal <iy3_real>.
Found 8-bit register for signal <iy0_img>.
Found 8-bit adder for signal <$n0000> created at line 66.
Found 8-bit subtractor for signal <$n0001> created at line 47.
Found 8-bit subtractor for signal <$n0002> created at line 43.
Found 8-bit adder for signal <$n0003> created at line 69.
Found 8-bit adder for signal <$n0004> created at line 38.
Found 8-bit subtractor for signal <$n0005> created at line 46.
Found 8-bit adder for signal <$n0006> created at line 72.
Found 8-bit adder for signal <$n0007> created at line 41.
Found 8-bit adder for signal <$n0008> created at line 37.
Found 8-bit subtractor for signal <$n0009> created at line 44.
Found 8-bit adder for signal <$n0010> created at line 40.
Found 8-bit adder for signal <$n0011> created at line 47.
Found 8-bit subtractor for signal <$n0012> created at line 47.
Found 8-bit subtractor for signal <$n0013> created at line 43.
Found 8-bit adder for signal <$n0014> created at line 43.
Found 8-bit adder for signal <$n0015>.
Found 8-bit adder for signal <$n0016>.
Found 8-bit adder for signal <$n0017> created at line 46.
Found 8-bit subtractor for signal <$n0018> created at line 46.
Found 8-bit subtractor for signal <$n0019> created at line 41.
Found 8-bit subtractor for signal <$n0020> created at line 41.
Found 8-bit adder for signal <$n0021>.
Found 8-bit adder for signal <$n0022>.
Found 8-bit subtractor for signal <$n0023> created at line 44.
```

```
    Found 8-bit adder for signal <$n0024> created at line 44.
    Found 8-bit subtractor for signal <$n0025> created at line 40.
    Found 8-bit subtractor for signal <$n0026> created at line 40.
    Found 8-bit register for signal <k0_img>.
    Found 8-bit register for signal <k0_real>.
    Found 8-bit register for signal <k1_img>.
    Found 8-bit register for signal <k1_real>.
    Found 8-bit register for signal <k2_img>.
    Found 8-bit register for signal <k2_real>.
    Found 8-bit register for signal <k3_img>.
    Found 8-bit register for signal <k3_real>.
    Found 8-bit register for signal <z0_img>.
    Found 8-bit register for signal <z0_real>.
    Found 8-bit register for signal <z1_img>.
    Found 8-bit register for signal <z1_real>.
    Found 8-bit register for signal <z2_img>.
    Found 8-bit register for signal <z2_real>.
    Found 8-bit register for signal <z3_img>.
    Found 8-bit register for signal <z3_real>.
    Summary:
            inferred 192 D-type flip-flop(s).
            inferred  27 Adder/Subtractor(s).
Unit <IFFT_4> synthesized.


Synthesizing Unit <qam_16>.
    Related source file is "C:/Xilinx/transmit/newQAM_16.vhd".
    Found 16x16-bit ROM for signal <$n0004>.
```

```
   Found 8-bit register for signal <qam_real>.
   Found 8-bit register for signal <qam_img>.
   Summary:
            inferred   1 ROM(s).
            inferred  16 D-type flip-flop(s).
Unit <qam_16> synthesized.


Synthesizing Unit <FFT_modemt>.
   Related source file is "C:/Xilinx/transmit/FFT_modemt.vhd".
Unit <FFT_modemt> synthesized.


=========================================================================
==============
HDL Synthesis Report

Macro Statistics
# ROMs                                   : 1
 16x16-bit ROM                           : 1
# Adders/Subtractors                     : 27
 8-bit adder                             : 15
 8-bit subtractor                        : 12
# Registers                              : 26
 8-bit register                          : 26

=========================================================================
==============
=====================================================================
===============
*                  Advanced HDL Synthesis                       *
=====================================================================
===============


Synthesizing (advanced) Unit <IFFT_4>.
        Found 2-bit shift register for signal <k0_real<4>>.
        Found 2-bit shift register for signal <k0_real<5>>.
        Found 2-bit shift register for signal <k0_img<4>>.
        Found 2-bit shift register for signal <k0_img<5>>.
        Found 3-bit shift register for signal <iy0_real<0>>.
        Found 3-bit shift register for signal <iy0_real<1>>.
        Found 3-bit shift register for signal <iy0_real<2>>.
        Found 3-bit shift register for signal <iy0_real<3>>.
        Found 2-bit shift register for signal <k3_img<0>>.
        Found 2-bit shift register for signal <k3_img<1>>.
        Found 2-bit shift register for signal <k3_img<2>>.
        Found 2-bit shift register for signal <k3_img<3>>.
        Found 2-bit shift register for signal <k3_img<4>>.
        Found 2-bit shift register for signal <k3_img<5>>.
        Found 2-bit shift register for signal <k1_real<4>>.
        Found 2-bit shift register for signal <k1_real<5>>.
        Found 2-bit shift register for signal <k1_img<0>>.
        Found 2-bit shift register for signal <k1_img<1>>.
        Found 2-bit shift register for signal <k1_img<2>>.
        Found 2-bit shift register for signal <k1_img<3>>.
        Found 2-bit shift register for signal <k1_img<4>>.
        Found 2-bit shift register for signal <k1_img<5>>.
        Found 3-bit shift register for signal <iy0_img<0>>.
        Found 3-bit shift register for signal <iy0_img<1>>.
        Found 3-bit shift register for signal <iy0_img<2>>.
        Found 3-bit shift register for signal <iy0_img<3>>.
        Found 2-bit shift register for signal <k2_real<4>>.
        Found 2-bit shift register for signal <k2_real<5>>.
        Found 2-bit shift register for signal <k2_img<0>>.
        Found 2-bit shift register for signal <k2_img<1>>.
        Found 2-bit shift register for signal <k2_img<2>>.
        Found 2-bit shift register for signal <k2_img<3>>.
        Found 2-bit shift register for signal <k2_img<4>>.
        Found 2-bit shift register for signal <k2_img<5>>.
        Found 2-bit shift register for signal <k3_real<4>>.
        Found 2-bit shift register for signal <k3_real<5>>.
        Found 3-bit shift register for signal <iy1_real<0>>.
        Found 3-bit shift register for signal <iy1_real<1>>.
        Found 3-bit shift register for signal <iy1_real<2>>.
        Found 3-bit shift register for signal <iy1_real<3>>.
        Found 3-bit shift register for signal <iy2_real<0>>.
        Found 3-bit shift register for signal <iy2_real<1>>.
        Found 3-bit shift register for signal <iy2_real<2>>.
        Found 3-bit shift register for signal <iy2_real<3>>.
        Found 3-bit shift register for signal <iy3_real<0>>.
        Found 3-bit shift register for signal <iy3_real<1>>.
        Found 3-bit shift register for signal <iy3_real<2>>.
        Found 3-bit shift register for signal <iy3_real<3>>.
Unit <IFFT_4> synthesized (advanced).
```

```
INFO:Xst:2261 - The FF/Latch <iy3_real_5> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy3_real_7>
INFO:Xst:2261 - The FF/Latch <iy3_real_4> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy3_real_6>
INFO:Xst:2261 - The FF/Latch <iy1_real_5> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy1_real_7>
INFO:Xst:2261 - The FF/Latch <iy0_real_5> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy0_real_7>
INFO:Xst:2261 - The FF/Latch <iy1_real_4> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy1_real_6>
INFO:Xst:2261 - The FF/Latch <iy0_real_4> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy0_real_6>
INFO:Xst:2261 - The FF/Latch <iy0_img_5> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy0_img_7>
INFO:Xst:2261 - The FF/Latch <iy2_real_5> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy2_real_7>
INFO:Xst:2261 - The FF/Latch <iy0_img_4> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy0_img_6>
INFO:Xst:2261 - The FF/Latch <iy2_real_4> in Unit <IFFT_4> is equivalent to
the following FF/Latch, which will be removed : <iy2_real_6>


=================================================================
===============
Advanced HDL Synthesis Report


Macro Statistics
# ROMs                                  : 1
 16x16-bit ROM                             : 1
# Adders/Subtractors                       : 27
 8-bit adder                            : 15
 8-bit subtractor                    : 12
# Registers                          : 50
 Flip-Flops                          : 50
# Shift Registers                    : 48
 2-bit shift register                : 28
 3-bit shift register                : 20

=========================================================================
==============

=========================================================================
==============
*                  Low Level Synthesis                   *
=========================================================================
==============
INFO:Xst:2261 - The FF/Latch <dut1/qam_img_3> in Unit <FFT_modemt> is
equivalent to the following 4 FFs/Latches, which will be removed :
<dut1/qam_img_4> <dut1/qam_img_5> <dut1/qam_img_6> <dut1/qam_img_7>


Optimizing unit <FFT_modemt> ...


Optimizing unit <IFFT_4> ...
Loading device for application Rf_Device from file 'v200.nph' in environment
C:\Xilinx.


Mapping all equations...
Building and optimizing final netlist ...
INFO:Xst:2261 - The FF/Latch <dut1/qam_real_3> in Unit <FFT_modemt> is
equivalent to the following 4 FFs/Latches, which will be removed :
<dut1/qam_real_4> <dut1/qam_real_5> <dut1/qam_real_6> <dut1/qam_real_7>
Found area constraint ratio of 100 (+ 5) on block FFT_modemt, actual ratio is 4.
```

```
FlipFlop dut1/qam_img_3 has been replicated 4 time(s)
FlipFlop dut1/qam_real_3 has been replicated 3 time(s)

=====================================================================
================
*                    Final Report                    *
=====================================================================
================
Final Results
RTL Top Level Output File Name     : FFT_modemt.ngr
Top Level Output File Name         : FFT_modemt
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 69

Cell Usage :
# BELS                             : 308
#      GND                         : 1
#      INV                         : 9
#      LUT1                        : 2
#      LUT2                        : 9
#      LUT2_D                      : 2
#      LUT2_L                      : 78
#      LUT3                        : 6
#      LUT4                        : 9
#      MUXCY                       : 101
#     VCC                    : 1
#     XORCY                  : 90
# FlipFlops/Latches          : 97
#     FD                     : 86
#     FDR                    : 4
#     FDRS                   : 6
#     FDS                    : 1
# Shift Registers            : 48
#     SRL16                  : 48
# Clock Buffers              : 1
#     BUFGP                  : 1
# IO Buffers                 : 68
#     IBUF                   : 4
#     OBUF                   : 64
==========================================================================================
===============

Device utilization summary:
---------------------------

Selected Device : 2s200pq208-6

 Number of Slices:                 114  out of   2352    4%
 Number of Slice Flip Flops:        97  out of   4704    2%
 Number of 4 input LUTs:           154  out of   4704    3%
    Number used as logic: 106
    Number used as Shift registers: 48
 Number of bonded IOBs:             69  out of    144   47%
```

```
=========================================================================
==============
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
    FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE
TRACE REPORT
    GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
-----------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 145   |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -6

   Minimum period: 10.840ns (Maximum Frequency: 92.251MHz)
   Minimum input arrival time before clock: 4.401ns
   Maximum output required time after clock: 6.959ns
   Maximum combinational path delay: No path found

Timing Detail:
```

```
`

--------------
All values displayed in nanoseconds (ns)

==========================================================================
===============
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 10.840ns (frequency: 92.251MHz)
  Total number of paths / destination ports: 7976 / 142
-------------------------------------------------------------------------
Delay:              10.840ns (Levels of Logic = 13)
  Source:           dut1/qam_img_0 (FF)
  Destination:      dut2/Mshreg_k3_img_5 (FF)
  Source Clock:     clk rising
  Destination Clock: clk rising

  Data Path: dut1/qam_img_0 to dut2/Mshreg_k3_img_5
                           Gate     Net
    Cell:in->out     fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FD:C->Q              8   1.085   1.845  dut1/qam_img_0 (dut1/qam_img_0)
     LUT2_D:I1->LO        1   0.549   0.000  dut2/IFFT_4__n0017<0>lut (N30)
       MUXCY:S->O             1       0.659      0.000      dut2/IFFT_4__n0017<0>cy
(dut2/IFFT_4__n0017<0>_cyo)
       MUXCY:CI->O            1       0.042      0.000      dut2/IFFT_4__n0017<1>cy
(dut2/IFFT_4__n0017<1>_cyo)
       MUXCY:CI->O            1       0.042      0.000      dut2/IFFT_4__n0017<2>cy
(dut2/IFFT_4__n0017<2>_cyo)
       MUXCY:CI->O            1       0.042      0.000      dut2/IFFT_4__n0017<3>cy
(dut2/IFFT_4__n0017<3>_cyo)
       MUXCY:CI->O            1       0.042      0.000      dut2/IFFT_4__n0017<4>cy
```

```
(dut2/IFFT_4__n0017<4>_cyo)
      XORCY:CI->O          1    0.420    1.035   dut2/IFFT_4__n0017<5>_xor
(dut2/_n0017<5>)
    (dut2/IFFT_4__n0018<5>_cyo)
      XORCY:CI->O          1    0.420    1.035   dut2/IFFT_4__n0018<6>_xor
(dut2/_n0018<6>)
      LUT2_L:I0->LO        1    0.549    0.000   dut2/IFFT_4__n0005<6>lut
(dut2/N141)
      MUXCY:S->O           0    0.659    0.000   dut2/IFFT_4__n0005<6>cy
(dut2/IFFT_4__n0005<6>_cyo)
      XORCY:CI->O          1    0.420    0.000   dut2/IFFT_4__n0005<7>_xor
(dut2/_n0005<7>)
    SRL16:D                   0.788          dut2/Mshreg_k3_real_5
    ----------------------------------------
    Total                  10.840ns (6.925ns logic, 3.915ns route)
                              (63.9% logic, 36.1% route)

=========================================================================
==============

Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
 Total number of paths / destination ports: 19 / 16
-------------------------------------------------------------------------
Offset:            4.401ns (Levels of Logic = 2)
 Source:            input<2> (PAD)
 Destination:        dut1/qam_real_1 (FF)
 Destination Clock: clk rising

Data Path: input<2> to dut1/qam_real_1
                        Gate     Net
 Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
```

```
  ----------------------------------------  ------------
  IBUF:I->O          3  0.776  1.332  input_2_IBUF (input_2_IBUF)
  INV:I->O           1  0.549  1.035  Mrom_data_dut1/Mrom__n000421_INV_0
(N12)
  FDS:D                 0.709         dut1/qam_real_1
  ----------------------------------------
  Total                 4.401ns (2.034ns logic, 2.367ns route)
                            (46.2% logic, 53.8% route)

=========================================================================
===============
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 64 / 64
-------------------------------------------------------------------------
Offset:            6.959ns (Levels of Logic = 1)
  Source:            dut2/iy0_real_5 (FF)
  Destination:       IF_q_real0<7> (PAD)
  Source Clock:      clk rising

  Data Path: dut2/iy0_real_5 to IF_q_real0<7>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FD:C->Q             2   1.085   1.206  dut2/iy0_real_5 (dut2/iy0_real_5)
     OBUF:I->O               4.668          IF_q_real0_7_OBUF (IF_q_real0<7>)
    ----------------------------------------
    Total                    6.959ns (5.753ns logic, 1.206ns route)
                                (82.7% logic, 17.3% route)
```

```
=========================================================================
=============
CPU : 6.10 / 6.30 s | Elapsed : 7.00 / 7.00 s


Total memory usage is 132364 kilobytes

Number of errors   :   0 (   0 filtered)
Number of warnings :  48 (   0 filtered)
Number of infos    :  13 (   0 filtered)
```

6.3.2 SYNTHESIS REPORT OBTAINED FOR OFDM RECEIVER

```
        Release 8.1i - xst I.24
Copyright (c) 1995-2005 Xilinx, Inc.  All rights reserved.
--> Parameter TMPDIR set to ./xst/projnav.tmp
CPU : 0.00 / 0.32 s | Elapsed : 0.00 / 1.00 s

--> Parameter xsthdpdir set to ./xst
CPU : 0.00 / 0.32 s | Elapsed : 0.00 / 1.00 s

--> Reading design: FFT_modemr.prj

TABLE OF CONTENTS
  1) Synthesis Options Summary
  2) HDL Compilation
  3) HDL Analysis
  4) HDL Synthesis
     4.1) HDL Synthesis Report
```

```
5) Advanced HDL Synthesis
   5.1) Advanced HDL Synthesis Report
6) Low Level Synthesis
7) Final Report
   7.1) Device utilization summary
   7.2) TIMING REPORT


==========================================================================
==============

                 Synthesis Options Summary                    *
==========================================================================
==============
---- Source Parameters
Input File Name                    : "FFT_modemr.prj"
Input Format                      : mixed
Ignore Synthesis Constraint File   : NO

---- Target Parameters
Output File Name                    : "FFT_modemr"
Output Format                      : NGC
Target Device                      : xc2s200-6-pq208

---- Source Options
Top Module Name                     : FFT_modemr
Automatic FSM Extraction            : YES
FSM Encoding Algorithm              : Auto
FSM Style                          : lut
```

```
RAM Extraction                     : Yes
RAM Style                          : Auto
ROM Extraction                     : Yes
Mux Style                          : Auto
Decoder Extraction                 : YES
Priority Encoder Extraction        : YES
Shift Register Extraction          : YES
Logical Shifter Extraction         : YES
XOR Collapsing                     : YES
ROM Style                          : Auto
Mux Extraction                     : YES
Resource Sharing                   : YES
Multiplier Style                   : lut
Automatic Register Balancing       : No

---- Target Options
Add IO Buffers                     : YES
Global Maximum Fanout              : 100
Add Generic Clock Buffer(BUFG)     : 4
Register Duplication               : YES
Slice Packing                      : YES
Pack IO Registers into IOBs        : auto
Equivalent register Removal        : YES

---- General Options
Optimization Goal                  : Speed
Optimization Effort                : 1
Keep Hierarchy                     : NO
```

```
RTL Output                         : Yes
Global Optimization                    : AllClockNets
Write Timing Constraints                : NO
Hierarchy Separator                   : /
Bus Delimiter                      : <>
Case Specifier                     : maintain
Slice Utilization Ratio                : 100
Slice Utilization Ratio Delta           : 5

---- Other Options
lso                            : FFT_modemr.lso
Read Cores                         : YES
cross_clock_analysis                   : NO
verilog2001                        : YES
safe_implementation                    : No
Optimize Instantiated Primitives   : NO
tristate2logic                    : Yes
use_clock_enable                      : Yes
use_sync_set                       : Yes
use_sync_reset                      : Yes

=========================================================================
==============


=========================================================================
==============
*                        HDL Compilation                        *
=========================================================================
`

==============
Compiling vhdl file "C:/Xilinx/recieve/FFT_4.vhd" in Library work.
Entity <FFT_4> compiled.
Entity <FFT_4> (Architecture <bhv>) compiled.
Compiling vhdl file "C:/Xilinx/recieve/newDEQAM_16.vhd" in Library work.
Entity <deqam_16> compiled.
Entity <deqam_16> (Architecture <bhv>) compiled.
Compiling vhdl file "C:/Xilinx/recieve/FFT_modemr.vhd" in Library work.
Entity <FFT_modemr> compiled.
Entity <FFT_modemr> (Architecture <bhv>) compiled.

============================================================
==============
*                    HDL Analysis                    *
============================================================
==============
Analyzing Entity <FFT_modemr> (Architecture <bhv>).
Entity <FFT_modemr> analyzed. Unit <FFT_modemr> generated.

Analyzing Entity <FFT_4> (Architecture <bhv>).
Entity <FFT_4> analyzed. Unit <FFT_4> generated.

Analyzing Entity <deqam_16> (Architecture <bhv>).
Entity <deqam_16> analyzed. Unit <deqam_16> generated.


============================================================
==============
*                    HDL Synthesis                    *
```

```
`

===========================================================================
===============

Synthesizing Unit <deqam_16>.
    Related source file is "C:/Xilinx/recieve/newDEQAM_16.vhd".
    Found 4-bit register for signal <qam_out>.
    Summary:
            inferred   4 D-type flip-flop(s).
Unit <deqam_16> synthesized.


Synthesizing Unit <FFT_4>.
    Related source file is "C:/Xilinx/recieve/FFT_4.vhd".
    Found 8-bit register for signal <y1_img>.
    Found 8-bit register for signal <y0_real>.
    Found 8-bit register for signal <y2_img>.
    Found 8-bit register for signal <y1_real>.
    Found 8-bit register for signal <y3_img>.
    Found 8-bit register for signal <y2_real>.
    Found 8-bit register for signal <y0_img>.
    Found 8-bit register for signal <y3_real>.
    Found 8-bit subtractor for signal <$n0000> created at line 24.
    Found 8-bit adder for signal <$n0001> created at line 22.
    Found 8-bit subtractor for signal <$n0002> created at line 27.
    Found 8-bit adder for signal <$n0003> created at line 25.
    Found 8-bit adder for signal <$n0004> created at line 30.
    Found 8-bit subtractor for signal <$n0005> created at line 28.
    Found 8-bit adder for signal <$n0006> created at line 21.
    Found 8-bit subtractor for signal <$n0007> created at line 31.
    Found 8-bit subtractor for signal <$n0008> created at line 24.
    Found 8-bit adder for signal <$n0009> created at line 24.
    Found 8-bit adder for signal <$n0010>.
    Found 8-bit adder for signal <$n0011>.
    Found 8-bit subtractor for signal <$n0012> created at line 27.
    Found 8-bit adder for signal <$n0013> created at line 27.
    Found 8-bit subtractor for signal <$n0014> created at line 25.
    Found 8-bit subtractor for signal <$n0015> created at line 25.
    Found 8-bit subtractor for signal <$n0016> created at line 30.
    Found 8-bit subtractor for signal <$n0017> created at line 30.
    Found 8-bit subtractor for signal <$n0018> created at line 28.
    Found 8-bit adder for signal <$n0019> created at line 28.
    Found 8-bit adder for signal <$n0020>.
    Found 8-bit adder for signal <$n0021>.
    Found 8-bit adder for signal <$n0022> created at line 31.
    Found 8-bit subtractor for signal <$n0023> created at line 31.
    Summary:
            inferred  64 D-type flip-flop(s).
            inferred  24 Adder/Subtractor(s).
Unit <FFT_4> synthesized.


Synthesizing Unit <FFT_modemr>.
    Related source file is "C:/Xilinx/recieve/FFT_modemr.vhd".

Unit <FFT_modemr> synthesized.

WARNING:Xst:524 - All outputs of the instance <dut3> of the block <FFT_4> are
unconnected in block <FFT_modemr>.
```

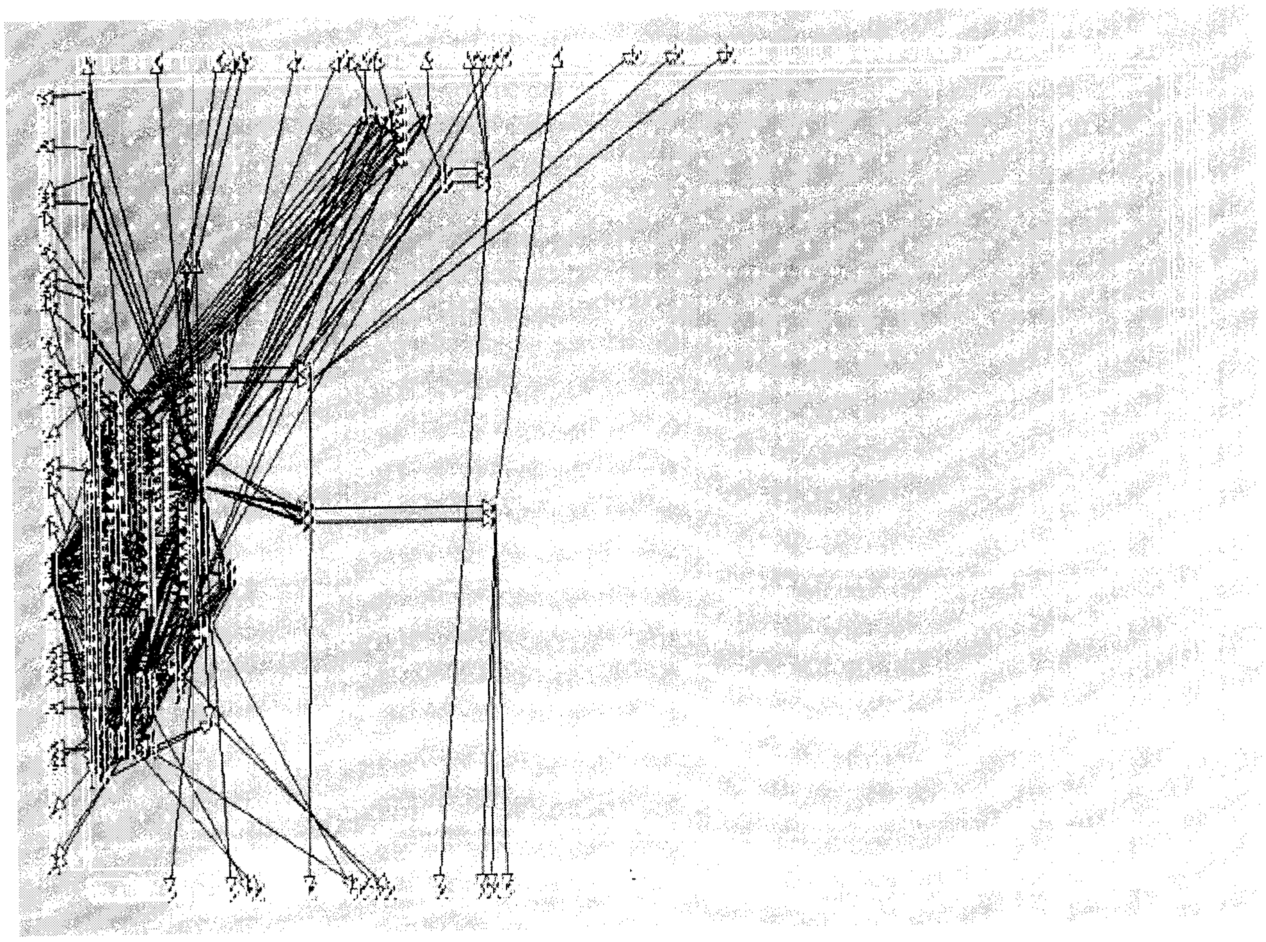
```
This instance will be removed from the design along with all underlying logic

=========================================================================
==============
HDL Synthesis Report

Macro Statistics
# Registers                             : 1
 4-bit register                         : 1

=========================================================================
==============

=========================================================================
==============
*                    Advanced HDL Synthesis                     *
=========================================================================
==============


=========================================================================
==============
Advanced HDL Synthesis Report

Macro Statistics
# Registers                             : 68
 Flip-Flops                             : 68

=========================================================================
==============
`

=================================================================
===============
*                    Low Level Synthesis                    *
=================================================================
===============

Optimizing unit <FFT_modemr> ...

Optimizing unit <FFT_4> ...

Optimizing unit <deqam_16> ...
Loading device for application Rf_Device from file 'v200.nph' in environment
C:\Xilinx.

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block FFT_modemr, actual ratio is 0.

=================================================================
===============
*                    Final Report                    *
=================================================================
===============
Final Results
RTL Top Level Output File Name     : FFT_modemr.ngr
Top Level Output File Name         : FFT_modemr
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO
```

```
Design Statistics
# IOs                        : 69

Cell Usage :
# BELS                        : 20

#     LUT2                    : 4
#     LUT3                    : 3
#     LUT4                    : 12
#     MUXF5                     : 1
# FlipFlops/Latches               : 4
#     FD                    : 3
#     FDR                     : 1
# Clock Buffers                 : 1
#     BUFGP                   : 1
# IO Buffers                  : 20
#     IBUF                    : 16
#     OBUF                     : 4
=================================================================
==============

Device utilization summary:
---------------------------

Selected Device : 2s200pq208-6

 Number of Slices:                  11  out of   2352     0%
 Number of 4 input LUTs:                19  out of   4704     0%
Number of bonded IOBs:              21  out of   144    14%
  IOB Flip Flops: 4
Number of GCLKs:                     1  out of     4    25%


=========================================================================
==============
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
    FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE
TRACE REPORT
   GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                           | BUFGP              | 4    |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -6

   Minimum period: No path found
   Minimum input arrival time before clock: 8.883ns
   Maximum output required time after clock: 6.788ns
  Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
  Total number of paths / destination ports: 117 / 5
-------------------------------------------------------------------------
Offset:              8.883ns (Levels of Logic = 6)
  Source:            IF_q_img0<4> (PAD)
  Destination:       dut4/qam_out_3 (FF)
  Destination Clock: clk rising

  Data Path: IF_q_img0<4> to dut4/qam_out_3
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             2   0.776   1.206  IF_q_img0_4_IBUF (IF_q_img0_4_IBUF)
     LUT2:I0->O            1   0.549   1.035  dut4/Ker7_SW0 (N11)
     LUT4:I3->O            4   0.549   1.440  dut4/Ker7 (dut4/N7)
     LUT4:I0->O            1   0.549   0.000  dut4/Ker0_F (N156)

     MUXF5:I0->O           2   0.315   1.206  dut4/Ker0 (dut4/N0)
     LUT2:I1->O            1   0.549   0.000  dut4/_n0001<2>1 (N155)
     FDR:D                     0.709          dut4/qam_out_2
    ----------------------------------------
    Total                      8.883ns (3.996ns logic, 4.887ns route)
```

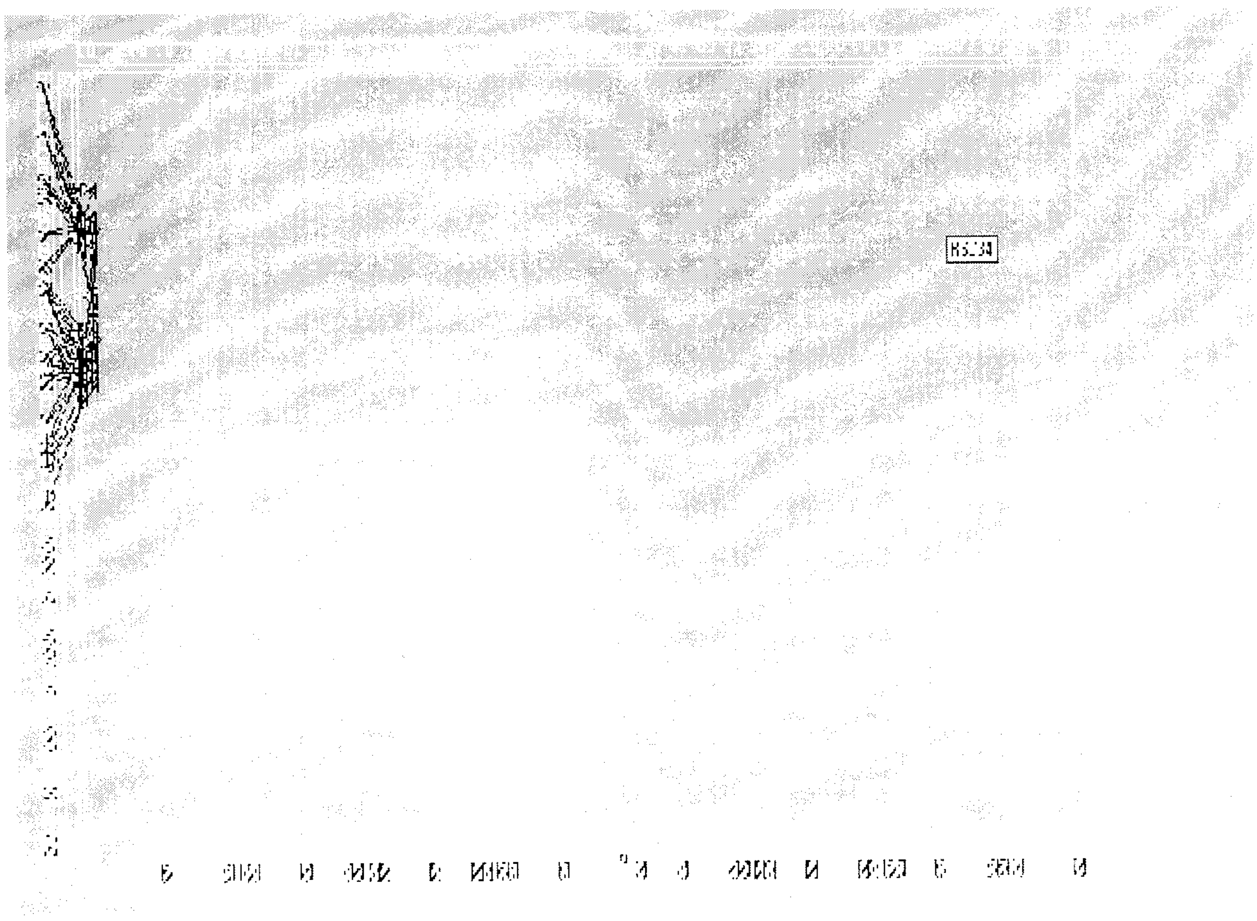
```
                         (45.0% logic, 55.0% route)
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
 Total number of paths / destination ports: 4 / 4
-------------------------------------------------------------------------
Offset:            6.788ns (Levels of Logic = 1)
 Source:            dut4/qam_out_3 (FF)
 Destination:        output<3> (PAD)
 Source Clock:       clk rising

 Data Path: dut4/qam_out_3 to output<3>
                         Gate     Net
   Cell:in->out     fanout   Delay   Delay  Logical Name (Net Name)
   ----------------------------------------  ------------
    FD:C->Q             1   1.085   1.035  dut4/qam_out_3 (dut4/qam_out_3)
    OBUF:I->O                4.668         output_3_OBUF (output<3>)
   ----------------------------------------
   Total                   6.788ns (5.753ns logic, 1.035ns route)
                            (84.8% logic, 15.2% route)

=========================================================================
===============
CPU : 4.82 / 5.21 s | Elapsed : 5.00 / 6.00 s


-->
Total memory usage is 132108 kilobytes
Number of errors   :    0 (   0 filtered)
Number of warnings :    9 (   0 filtered)
Number of infos    :    0 (   0 filtered)
```

6.4 FLOOR PLANNING RESULTS:

6.4.1 FLOOR PLAN OBTAINED FOR OFDM TRANSMITTER:



6.4.2 FLOOR PLAN OBTAINED FOR OFDM RECEIVER:



CHAPTER 7

CONCLUSION

The project analyses the OFDM system under various considerations. First, the MATLAB simulation of the OFDM system provides an insight into the various processes that are carried out in the process of OFDM. The Rayleigh-Fading channel for BPSK system has been modeled and the noise components also have been introduced. Finally, the BER performance of the OFDM system is estimated and analyzed.

Next our aim of developing VHDL programs for the OFDM system comprising of the QAM, Inverse FFT, FFT & Inverse QAM blocks has also been achieved.

The 16-point QAM with rectangular constellation is employed along with the 4-point Decimation-In-Time FFT & Inverse FFT blocks. The transmitter module output is given as input to the receiver module and the transmitted data is obtained back at the receiver accurately. The VHDL code for the complex conjugate multiplier to be incorporated within the butterfly processor has also been functionally simulated using Modelsim.

Finally, the developed OFDM system is implemented in the FPGA. The RTL schematic, synthesis report and floorplans for the transmitter and receiver modules are obtained successfully. Thus with the description of the system in VHDL, the hardware implementation of the system is arrived.

CHAPTER 8

EXTENSION OF THE PROJECT

As the aim of our project is achieved, useful extensions are suggested for the project that will make the OFDM concept to be implemented practically.

1)The OFDM Transmitter & Receiver cores are constructed with only the QAM & FFT blocks for coding and implementation. In order to make it practically applicable, other blocks such as the Frequency Interleaver, de-interleaver, STBC Coder & Decoder along with Channel Estimation can be included.

2) The outputs of the Transmitter block are directly coupled to the Receiver Block. However in practical, there exists a wireless path in between the Transmitter and Receiver. Hence an RF module can be added for demonstration or in case of practical systems, additional modifications are required for amplifying the received signal. Error correction & detection may also be included to make the system more reliable.

3)In the FFT block, 4-point FFT is used. If more complex systems are designed, the need for multiplication of complex twiddle factors arises. An algorithm for the complex conjugate multiplier is proposed. As an extension to this, higher order FFTs can be developed to account for complex systems.

APPENDIX - A

MATLAB PROGRAMS

A.1 Script for computing the BER for BPSK modulation in a Rayleigh fading channel

```
clear
N = 10^6 % number of bits or symbols
% Transmitter
ip = rand(1,N)>0.5; % generating 0,1 with equal probability
s = 2*ip-1; % BPSK modulation 0 -> -1; 1 -> 0
Eb_N0_dB = [-3:35]; % multiple Eb/N0 values
for ii = 1:length(Eb_N0_dB)
n = 1/sqrt(2)*[randn(1,N) + j*randn(1,N)]; % white Gaussian noise,0dB variance
h = 1/sqrt(2)*[randn(1,N) + j*randn(1,N)]; % Rayleigh channel
% Channel and noise Noise addition
y = h.*s + 10^(-Eb_N0_dB(ii)/20)*n;
% equalization
yHat = y./h;
% receiver - hard decision decoding
ipHat = real(yHat)>0;
% counting the errors
nErr(ii) = size(find([ip- ipHat]),2);
end
simBer = nErr/N; % simulated ber
theoryBerAWGN = 0.5*erfc(sqrt(10.^(Eb_N0_dB/10))); % theoretical ber
EbN0Lin = 10.^(Eb_N0_dB/10);
theoryBer = 0.5.*(1-sqrt(EbN0Lin./(EbN0Lin+1)));
```

```
% plot
close all
figure
semilogy(Eb_N0_dB,theoryBerAWGN,'cd-','LineWidth',2);
hold on
semilogy(Eb_N0_dB,theoryBer,'bp-','LineWidth',2);
semilogy(Eb_N0_dB,simBer,'mx-','LineWidth',2);
axis([-3 35 10^-5 0.5])
grid on
legend('AWGN-Theory','Rayleigh-Theory', 'Rayleigh-Simulation');
xlabel('Eb/No, dB');
ylabel('Bit Error Rate');
title('BER for BPSK modulation in Rayleigh channel');
```

A.2 Script for computing the BER for BPSK with OFDM modulation in the presence of Rayleigh fading channel

```
clear all
nFFT        = 64; % fft size
nDSC        = 52; % number of data subcarriers
nBitPerSym  = 52; % number of bits per OFDM symbol
nSym        = 10^4; % number of symbols
EbN0dB      = [0:35]; % bit to noise ratio
EsN0dB          = EbN0dB + 10*log10(nDSC/nFFT) + 10*log10(64/80); %
converting to     symbol to noise ratio
for ii = 1:length(EbN0dB)
% Transmitter
```

```
ipBit = rand(1,nBitPerSym*nSym) > 0.5; % random 1's and 0's
ipMod = 2*ipBit-1; % BPSK modulation 0 --> -1, 1 --> +1
ipMod = reshape(ipMod,nBitPerSym,nSym).'; % grouping into multiple symbolsa
% Assigning modulated symbols to subcarriers from [-26 to -1, +1 to +26]
xF = [zeros(nSym,6) ipMod(:,[1:nBitPerSym/2]) zeros(nSym,1)
ipMod(:,[nBitPerSym/2+1:nBitPerSym]) zeros(nSym,5)] ;
% Taking FFT, the term (nFFT/sqrt(nDSC)) is for normalizing the power of
transmit symbol to 1
xt = (nFFT/sqrt(nDSC))*ifft(fftshift(xF.')).';
% Appending cylic prefix
xt = [xt(:,[49:64]) xt];
% multipath channel
nTap = 10;
ht = 1/sqrt(2)*1/sqrt(nTap)*(randn(nSym,nTap) + j*randn(nSym,nTap));
% computing and storing the frequency response of the channel,for at recevier
hF = fftshift(fft(ht,64,2));
% convolution of each symbol with the random channel
for jj = 1:nSym
xht(jj,:) = conv(ht(jj,:),xt(jj,:));
end
xt = xht;
% Concatenating multiple symbols to form a long vector
xt = reshape(xt.',1,nSym*(80+nTap-1));
% Gaussian noise of unit variance, 0 mean
nt = 1/sqrt(2)*[randn(1,nSym*(80+nTap-1)) + j*randn(1,nSym*(80+nTap-1))];
% Adding noise, the term sqrt(80/64) is to account for the wasted energy due to
```

cyclic prefix

```
yt = sqrt(80/64)*xt + 10^(-EsN0dB(ii)/20)*nt;
% Receiver
yt = reshape(yt.',80+nTap-1,nSym).'; % formatting the received vector into
symbols
yt = yt(:,[17:80]); % removing cyclic prefix
% converting to frequency domain
yF = (sqrt(nDSC)/nFFT)*fftshift(fft(yt.')).';
% equalization by the known channel frequency response
yF = yF./hF;
% extracting the required data subcarriers
yMod = yF(:,[6+[1:nBitPerSym/2] 7+[nBitPerSym/2+1:nBitPerSym] ]);
% BPSK demodulation
% +ve value --> 1, -ve value --> -1
ipModHat = 2*floor(real(yMod/2)) + 1;
ipModHat(find(ipModHat>1)) = +1;
ipModHat(find(ipModHat<-1)) = -1;
% converting modulated values into bits
ipBitHat = (ipModHat+1)/2;
ipBitHat = reshape(ipBitHat.',nBitPerSym*nSym,1).';
% counting the errors
nErr(ii) = size(find(ipBitHat - ipBit),2);
end
simBer = nErr/(nSym*nBitPerSym);
EbN0Lin = 10.^(EbN0dB/10);
theoryBer = 0.5.*(1-sqrt(EbN0Lin./(EbN0Lin+1)));
```

```
close all; figure
semilogy(EbN0dB,theoryBer,'bs-','LineWidth',2);
hold on
semilogy(EbN0dB,simBer,'mx-','LineWidth',2);
axis([0 35 10^-5 1])
grid on
legend('Rayleigh-Theory', 'Rayleigh-Simulation');
xlabel('Eb/No, dB')
ylabel('Bit Error Rate')
title('BER for BPSK using OFDM in a 10-tap Rayleigh channel')
```

APPENDIX - B

VHDL CODING:

B.1 VHDL CODE FOR THE COMPLEX CONJUGATE MULTIPLIER TO BE IMPLEMENTED WITH THE BUTTERFLY PROCESSOR

```
library lpm;
  use lpm.lpm_components.ALL;
  library ieee;
     use ieee.std_logic_1164.ALL;
     use ieee.std_logic_arith.ALL;
     entity ccmul is
        generic(w2:integer:=17;
        w1:integer:=9;
        w:integer:=8);
        port(clk:std_logic;
           x_in,y_in,c_in
                       : in STD_LOGIC_VECTOR(w-1 downto 0);
           cps_in,cms_in
                     : in STD_LOGIC_VECTOR(w1-1 downto 0);
          r_out,i_out
                     : out STD_LOGIC_VECTOR(w-1 downto 0));
```

```
end ccmul;
architecture flex of ccmul is
 signal x,y,c:std_logic_vector(w-1 downto 0);
 signal r,i,cmsy,cpsx,xmyc : std_logic_vector(w2-1 downto 0);
 signal xmy,cps,cms,sxtx,sxty:std_logic_vector(w1-1 downto 0);
      begin
   x<=x_in;
   y<=y_in;
   c<=c_in;
   cps<=cps_in;
   cms<=cms_in;
   process
     begin
       wait until clk='1';
       r_out <=r(w2-3 downto w-1);
       i_out <=i(w2-3 downto w-1);
     end process;
     sxtx <=x(x'high)&x;
     sxty <=y(y'high)&y;
     sub_1:lpm_add_sub
     generic            map            (            LPM_WIDTH
=>w1,LPM_DIRECTION=>"SUB",LPM_REPRESENTATION=>"SIGNED")
          port map (dataa=>sxtx,datab=>sxty,result=>xmy);
                         mul_1:lpm_mult
            generic
map(LPM_WIDTHA=>w1,LPM_WIDTHB=>w,LPM_WIDTHP=>w2,LPM_WI
DTHS=>w2,LPM_REPRESENTATION=>"SIGNED")
            port map (dataa=>xmy,datab=>c,result=>xmyc);
              mul_2:lpm_mult
            generic
map(lpm_widtha=>w1,lpm_widthb=>w,lpm_widthp=>w2,lpm_widths=>w2,lpm_
representation=>"SIGNED")
            port map(dataa=>cms,datab=>y,result=>cmsy);
              mul_3:lpm_mult
            generic
map(lpm_widtha=>w1,lpm_widthb=>w,lpm_widthp=>w2,lpm_widths=>w2,lpm_
representation=>"SIGNED")
            port map(dataa=>cps,datab=>x,result=>cpsx);
             sub_2:lpm_add_sub
         generic
map(lpm_width=>w2,lpm_direction=>"SUB",lpm_representation=>"SIGNED")
         port map(dataa=>cpsx,datab=>xmyc,result=>i);
           add_1:lpm_add_sub
```

```
            generic
map(lpm_width=>w2,lpm_direction=>"ADD",lpm_representation=>"SIGNED")
            port map( dataa=>cmsy,datab=>xmyc,result=>r);
            end flex;
```

B.2 VHDL CODE FOR OFDM TRANSMITTER MODULE:

```
library ieee;
use ieee.std_logic_1164.all;
entity FFT_modemt is
port (clk:in std_logic;
    input  :in std_logic_vector(3 downto 0);

IF_q_real0,IF_q_real1,IF_q_real2,IF_q_real3,IF_q_img0,IF_q_img1,IF_q_img2,I
F_q_img3:out std_logic_vector(7 downto 0));
    end FFT_modemt;
architecture bhv of FFT_modemt is
component qam_16
port (clk:in std_logic;
    qam_in :in std_logic_vector(3 downto 0);
    qam_real:out std_logic_vector(7 downto 0);
    qam_img:out std_logic_vector(7 downto 0));
end component;
```

```
component IFFT_4
port(clk:std_logic;
    ix0_real,ix1_real,ix2_real,ix3_real:in std_logic_vector(7 downto 0);
    ix0_img,ix1_img,ix2_img,ix3_img    :in std_logic_vector(7 downto 0);
    iy0_real,iy1_real,iy2_real,iy3_real:out std_logic_vector(7 downto 0);
    iy0_img,iy1_img,iy2_img,iy3_img    :out std_logic_vector(7 downto 0));
end component;
signal q_real,q_img:std_logic_vector(7 downto 0);
begin
dut1:qam_16  port map(clk,input,q_real,q_img);
dut2:IFFT_4 port map (clk,q_real,q_real,q_real,q_real,q_img,q_img,q_img,q_img,

IF_q_real0,IF_q_real1,IF_q_real2,IF_q_real3,IF_q_img0,IF_q_img1,
IF_q_img2,IF_q_img3);
end bhv;
```

B.3 VHDL CODE FOR Inverse FFT BLOCK:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity  IFFT_4 is
```

```
  port(clk:std_logic;
      ix0_real,ix1_real,ix2_real,ix3_real:in std_logic_vector(7 downto 0);
      ix0_img,ix1_img,ix2_img,ix3_img    :in std_logic_vector(7 downto 0);
      iy0_real,iy1_real,iy2_real,iy3_real:out std_logic_vector(7 downto 0);
      iy0_img,iy1_img,iy2_img,iy3_img    :out std_logic_vector(7 downto 0));
 end IFFT_4;
 architecture bhv of IFFT_4 is
 signal z0_real,z1_real,z2_real,z3_real:std_logic_vector(7 downto 0);
 signal z0_img,z1_img,z2_img,z3_img    :std_logic_vector(7 downto 0);
 signal k0_real,k1_real,k2_real,k3_real:std_logic_vector(7 downto 0);
 signal k0_img,k1_img,k2_img,k3_img    :std_logic_vector(7 downto 0);
 begin
 process(clk,ix0_real,ix1_real,ix2_real,ix3_real,ix0_img,ix1_img,ix2_img,ix3_img)
 begin
 if clk'event and clk='1' then
--z0_real<= ix0_real + ix1_real + ix2_real + ix3_real;
--z0_img<= "00000000";
--
--z1_real<=ix0_real - ix2_real;
--z1_img<=ix1_img - ix3_img;--
--z2_real<=ix0_real - ix1_real + ix2_real -  ix3_real;
```

```
--z2_img<="00000000";--

--z3_real<=ix0_real - ix2_real;

--z3_img<=ix3_img - ix3_img;

z0_real<= ix0_real + ix1_real + ix2_real + ix3_real;

z0_img<=  ix1_img + ix2_img + ix3_img + ix0_img;---- (-)

z1_real<=ix0_real - ix2_real - ix1_img + ix3_img;

z1_img<= ix3_real - ix0_img - ix1_real + ix2_img;--

z2_real<=ix0_real - ix1_real + ix2_real -  ix3_real;

z2_img<= ix1_img - ix2_img + ix3_img -ix0_img ;

z3_real<=ix0_real + ix1_img - ix2_real - ix3_img;

z3_img<=  ix1_real + ix2_img - ix3_real - ix0_img;

k0_real<="00" & (z0_real(7 downto 2));

k0_img<="00" & z0_img(7 downto 2);

k1_real<="00" & z1_real(7 downto 2);

k1_img<="00" & z1_img(7 downto 2);

k2_real<="00" & z2_real(7 downto 2);

k2_img<="00" & z2_img(7 downto 2);

k3_real<="00" & z3_real(7 downto 2);

k3_img<="00" & z3_img(7 downto 2);

iy0_real<= k0_real(5 downto 4) & k0_real(5 downto 0);

iy0_img<=  k0_img(5 downto 4) & k0_img(5 downto 0);
```

```
iy1_real<=k1_real(5 downto 4) & k1_real(5 downto 0);
iy1_img<="00000001" + not (k1_img(5 downto 4) & k1_img(5 downto 0));
iy2_real<=k2_real(5 downto 4) & k2_real(5 downto 0);
iy2_img<="00000001" + not(k2_img(5 downto 4) & k2_img(5 downto 0));
iy3_real<=k3_real(5 downto 4) & k3_real(5 downto 0);
iy3_img<="00000001" + not(k3_img(5 downto 4) & k3_img(5 downto 0));
end if;
end process;
end bhv;
```

B.4 VHDL CODE FOR QAM BLOCK:

```
library ieee;
use ieee.std_logic_1164.all;
entity qam_16 is
port (clk:in std_logic;
    qam_in :in std_logic_vector(3 downto 0);
    qam_real:out std_logic_vector(7 downto 0);
    qam_img:out std_logic_vector(7 downto 0));
end qam_16;
architecture bhv of qam_16 is
begin
```

`

```
process(clk,qam_in)
begin
if clk='1' and clk'event then
case qam_in is
when "0000" =>
qam_real<="11111010";-- -6
qam_img <="00000110";--+6j
when "0001"=>qam_real<="11111010";-- -6
qam_img <="00000011";--- +3j
when "0010"=>
qam_real<="11111010";-- -6
qam_img <="11111010";-- -6j
when "0011"=>
qam_real<="11111010";-- -6
qam_img <="11111101";-- -3j
when "0100" =>
qam_real<="11111101";-- -3
qam_img <="00000110";--+6j
when "0101" =>
qam_real<="11111101";-- -3
qam_img <="00000011";--- +3j
```

```
when "0110"=>
qam_real<="11111101";-- -3
qam_img <="11111010";-- -6j
when "0111"=>
qam_real<="11111101";-- -3
qam_img <="11111101";-- -3j
when "1000"=>
qam_real<="00000110";--+6
qam_img <="00000110";--+6j
when "1001"=>
qam_real<="00000110";--+6
qam_img <="00000011";--- +3j
when "1010"=>
qam_real<="00000110";--+6
qam_img <="11111010";-- -6j
when "1011" =>
qam_real<="00000110";--+6
qam_img <="11111101";-- -3j
when "1100" =>
qam_real<="00000011";--- +3
qam_img <="00000110";--+6j
```

`

```
when "1101" =>

qam_real<="00000011";--- +3

qam_img <="00000011";--- +3j

when "1110" =>

qam_real<="00000011";--- +3

qam_img <="11111010";-- -6j

when "1111" =>

qam_real<="00000011";--- +3

qam_img <="11111101";-- -3j

when others=>

qam_real<="00000000";--0

qam_img <="00000000";--j0

end case;

end if;

end process;

end bhv;
```

B.5 VHDL CODE FOR OFDM RECEIVER SIDE:

```
library ieee;

use ieee.std_logic_1164.all;
```

```
entity FFT_modemr is
port (clk:in std_logic;

IF_q_real0,IF_q_real1,IF_q_real2,IF_q_real3,IF_q_img0,IF_q_img1,IF_q_img2,
IF_q_img3:in std_logic_vector(7 downto 0);
    output :out std_logic_vector(3 downto 0));
end FFT_modemr;
architecture bhv of FFT_modemr is
component  FFT_4
port(clk:std_logic;
   x0_real,x1_real,x2_real,x3_real:in std_logic_vector(7 downto 0);
   x0_img,x1_img,x2_img,x3_img    :in std_logic_vector(7 downto 0);
   y0_real,y1_real,y2_real,y3_real:out std_logic_vector(7 downto 0);
   y0_img,y1_img,y2_img,y3_img     :out std_logic_vector(7 downto 0));
end component;
component deqam_16
port (clk:in std_logic;
   qam_out :out std_logic_vector(3 downto 0);
   qam_real:in std_logic_vector(7 downto 0);
   qam_img :in std_logic_vector(7 downto 0));
end component;
```

```
signal  F_q_real0,F_q_real1,F_q_real2,F_q_real3,F_q_img0,F_q_img1,F_q_img2,
F_q_img3:std_logic_vector(7 downto 0);
begin
dut3:FFT_4                                                                    port
map(clk,IF_q_real0,IF_q_real1,IF_q_real2,IF_q_real3,IF_q_img0,IF_q_img1,
IF_q_img2,IF_q_img3,
            F_q_real0,F_q_real1,F_q_real2,F_q_real3,F_q_img0,F_q_img1,
F_q_mg2,F_q_img3);
dut4:deqam_16 port map(clk,output,IF_q_real0,IF_q_img0);
end bhv;
```

B.6 VHDL CODE FOR FFT BLOCK:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity  FFT_4 is
port(clk:std_logic;
   x0_real,x1_real,x2_real,x3_real:in std_logic_vector(7 downto 0);
   x0_img,x1_img,x2_img,x3_img    :in std_logic_vector(7 downto 0);
   y0_real,y1_real,y2_real,y3_real:out std_logic_vector(7 downto 0);
   y0_img,y1_img,y2_img,y3_img    :out std_logic_vector(7 downto 0));
end FFT_4;
```

```
architecture bhv of FFT_4 is
begin
process(clk,x0_real,x1_real,x2_real,x3_real,x0_img,x1_img,x2_img,x3_img)
begin
if clk'event and clk='1' then
y0_real<= x0_real + x1_real + x2_real + x3_real;
y0_img<= x0_img + x1_img + x2_img + x3_img;
y1_real<=x0_real - x2_real + x1_img - x3_img;
y1_img<= x0_img - x1_real - x2_img + x3_real;
y2_real<=x0_real - x1_real + x2_real -  x3_real;
y2_img<=x0_img - x1_img + x2_img - x3_img;
y3_real<=x0_real - x1_img - x2_real + x3_img;
y3_img<=x0_img + x1_real - x2_img - x3_real;
end if;
end process;
end bhv;
```

B.7 VHDL CODE FOR INVERSE QAM BLOCK:

```
library ieee;
use ieee.std_logic_1164.all;
entity deqam_16 is
```

```
port (clk:in std_logic;
     qam_out :out std_logic_vector(3 downto 0);
     qam_real:in std_logic_vector(7 downto 0);
     qam_img :in std_logic_vector(7 downto 0));
end deqam_16;
architecture bhv of deqam_16 is
signal real_img:std_logic_vector(15 downto 0);
begin
real_img<=qam_real & qam_img;
process(clk,real_img)
begin
if clk='1' and clk'event then
case real_img is
when "1111101000000011" =>
--qam_real<="11111010";-- -6
--qam_img <="00000011";--+3j
qam_out<="0001";
when "1111101000000110"=>
--qam_real<="11111010";--- -6
--qam_img <="000001106";--- +j6
qam_out<="0000";
```

```
`

when "1111101011111010"=>
--qam_real<="11111010";-- -6
--qam_img <="11111010";-- -6j
qam_out<="0010";
when "1111101011111101"=>
--qam_real<="11111010";--- -6
--qam_img <="11111101";--- -3j
qam_out<="0011";
qam_out<="0100";
when "1111110100000011" =>
--qam_real<="11111101";-- -3
--qam_img <="00000011";--- +3j
qam_out<="0101";
when "1111110111111010"=>
--qam_real<="11111101";-- -3
--qam_img <="11111010";-- -6j
qam_out<="0110";
when "1111110111111101"=>
--qam_real<="11111101";-- -3
--qam_img <="11111101";-- -3j
qam_out<="0111";
```

```
when "0000011000000110"=>
--qam_real<="00000110";--+6
--qam_img <="00000110";--+6j
qam_out<="1000";
when "0000011000000011"=>
--qam_real<="00000110";--+6
--qam_img <="00000011";--- +3j
qam_out<="1001";
when "0000011011111010"=>
--qam_real<="00000110";--+6
--qam_img <="11111010";-- -6j
qam_out<="1010";
when "0000011011111101" =>
--qam_real<="00000110";--+6
--qam_img <="11111101";-- -3j
qam_out<="1011";
when "0000001100000110" =>
--qam_real<="00000011";--- +3
--qam_img <="00000110";--+6j
qam_out<="1100";
when "0000001100000011" =>
```

```
--qam_real<="00000011";--- +3
--qam_img <="00000011";--- +3j
qam_out<="1101";
when "0000001111111010" =>
--qam_real<="00000011";--- +3
--qam_img <="11111010";-- -6j
qam_out<="1110";
when "0000001111111101" =>
--qam_real<="00000011";--- +3
--qam_img <="11111101";-- -3j
qam_out<="1111";
when others=>
--qam_real<="00000000";--0
--qam_img <="00000000";--j0
qam_out<="0000";
end case;
end if;
end process;
end bhv;
```

APPENDIX - C

REFERENCES

BOOKS:

1. John G. Proakis, Dimitris G. Manolakis, "Digital Signal Processing - Principles, Algorithms, and Applications" - Prentice Hall of India.

2. U. Meyer-Baese, "Digital Signal Processing with Field Programmable Gate Arrays"- Springer.

3. Simon Haykins, "Communication Systems" - Tata McGraw Hill.

4. J. Bhasker, "A VHDL Primer" - Pearson Education.

5. Weste & Eshraghian, "Principles of CMOS VLSI Design" - Tata McGraw Hill.

PAPERS:

1. Ming-Fu Sun, Student Member, IEEE, Ta-Yang Juan, Kan-Si Lin, and Terng-Yin Hsu, Member, IEEE, "Adaptive Frequency-Domain Channel Estimator in 4×4 MIMO-OFDM Modems" -[IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 17, NO. 11, NOVEMBER(2009)]

2. Dr Chris Dick, DSP Chief Architect, Director, Signal Processing Systems Engineering, "FPGA Implementation of an OFDM PHY"-[Proceeding of the SDR 03 Technical Conference and Product Exposition]

WEBSITES:

1. www.dsplog.com

2. www.altera.com

3. www.fpgacentral.com