

# IMPLEMENTATION OF THE FFT PROCESSOR USING FPGA

## A DISSERTATION

*Submitted in partial fulfilment of the  
requirements for the award of the degree*

*of*

MASTER OF TECHNOLOGY

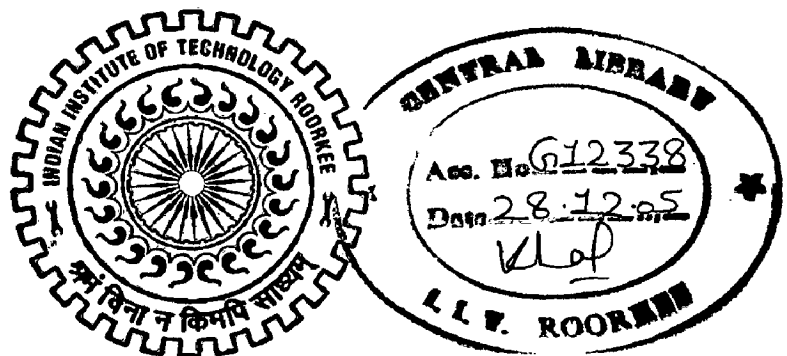
*in*

ELECTRICAL ENGINEERING

(With Specialization in System Engineering and Operations Research)

*By*

**M.NAVEEN KUMAR REDDY**



DEPARTMENT OF ELECTRICAL ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE  
ROORKEE-247 667 (INDIA)

JUNE, 2005

## CANDIDATE'S DECLARATION

---

I hereby declare that the work presented in this dissertation entitled "**IMPLEMENTATION OF THE FFT PROCESSOR USING FPGA**" submitted in partial fulfillment of the requirements for the award of degree of **Master of Technology in Electrical Engineering** with specialization in **System Engineering and Operations Research**, in the Department of Electrical Engineering, Indian Institute of Technology Roorkee, Roorkee, is an authentic record of my own work carried out from July 2004 to June 2005 under the guidance of **Prof.M.K.Vasantha**, Professor and **Dr. Indra Gupta**, Asstt Professor, Department of Electrical Engineering, Indian Institute of Technology Roorkee, Roorkee.

I have not submitted the matter embodied in this report for the award of any other degree or diploma.

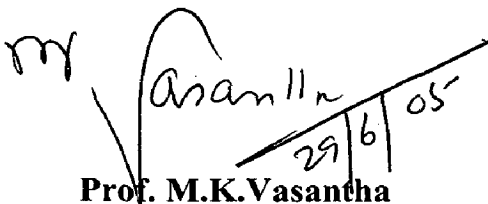
Date: 29 June 2005  
Place: Roorkee

M. Naveen Reddy  
(M.Naveen Kumar Reddy)

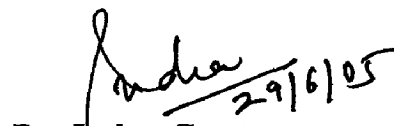
## CERTIFICATE

---

This is to certify that the above statement made by the candidate is true to the best of my knowledge and belief.

  
Prof. M.K.Vasantha  
Professor

Department of Electrical Engineering  
Indian Institute of Technology Roorkee  
Roorkee.

  
Dr. Indra Gupta  
Asstt Professor

Department of Electrical Engineering  
Indian Institute of Technology Roorkee  
Roorkee.

## ACKNOWLEDGEMENTS

---

I express my heartfelt gratitude to **Prof.M.K.Vasantha**, Professor and **Dr.Indra Gupta**, Asstt Professor, Department of Electrical Engineering, Indian Institute of Technology Roorkee, Roorkee for their valuable guidance, support encouragement and immense help.

I consider myself extremely fortunate for having got the opportunity to learn and work under their able supervision. I have deep sense of admiration for their innate goodness and inexhaustible enthusiasm. The valuable hours of discussions and suggestions that I had with them have undoubtedly helped in supplementing my thoughts in the right direction for attaining the desired objective. Working under their guidance will always remain a cherished experience in my memory and I will adore it throughout my life.

My heartfelt gratitude and indebtedness goes to all the teachers of SEOR group who, with their encouraging and caring words, constructive criticism and suggestions, have contributed directly or indirectly in a significant way towards completion of this report.

I am highly grateful to **Mr.Rahul Dubey**, Research scholar, Department of Electrical Engineering, Indian Institute of Technology Roorkee, and Roorkee for his keen interest and generous encouragement during this venture.

I am especially thankful to **Major Seby Thomas** for his feedback in the class. His ideas and discussions not only helped in technical areas but also in personality development.

A word of thanks also goes to **Mr.Vijender singh** and **Mr.Vishal Saxena**, research scholar, Electrical Engineering department for always being there to help at the hour of need.

I would also like to extend our sincere appreciation to **Mr. Kalyan Singh** and **Mr.Joshi**, Laboratory staff of Micro Processor & Computer Lab for providing the required facilities and co-operation during this work.

Special, sincere and heartfelt gratitude goes to my parents and my friends whose sincere prayers, best wishes, support and encouragement have been a constant source of assurance, guidance, strength and inspiration to me.

*M. Naveen*  
( **M.Naveen Kumar Reddy** )

## ABSTRACT

---

The Fast Fourier Transform (FFT) is a computationally intensive digital signal processing function widely used in applications such as imaging, software defined radio, wireless communication, instrumentation and machine inspection. Historically, this has been a relatively difficult function to implement optimally in hardware, leading many software designers to use digital signal processors (DSPs) in soft implementations. Unfortunately, because of the function's computationally intensive nature, such an approach typically requires multiple DSPs within the system to support the processing requirements. This is costly from a device and board real-estate perspective.

Field-programmable gate array (FPGA) have become an extremely cost-effective means of off-loading computationally intensive algorithms to improve overall system performance. The FFT processor implementation on FPGA that utilizes dedicated hardware multiplier resources can cost effectively achieve application-specific integrated circuit (ASIC)-like performance while reducing development time, cost and risks.

In this thesis 16-point FFT processor has been designed and implemented. The design is based on a decimation-in-frequency radix-4 algorithm and employs in-place computation to optimize memory usage. In order to operate the processor, data must first be loaded into the internal RAM. The processor is then instructed to compute the FFT, overwriting the input data in the RAM with the results. Upon completion of the FFT, the results may be read out from the RAM via the output data port. The design specifications for the FFT processor are laid down using radix-4 algorithm. It is capable of computing one butterfly computation every 40ns thus it can compute 16-complex point FFT in 1300ns including data input and output processes. The chip is operating with a clock frequency of 100MHz. The FFT processor is designed and tested according to the design specifications with the help of ISE (Integrated Software Environment) provided by Xilinx. The designed FFT processor has been implemented in Xilinx Spartan-II FPGA.

# CONTENTS

---

<b>CH No.</b>	<b>Topic</b>	<b>Page No.</b>
	<b>CANDIDATES'S DECLARATION</b>	i
	<b>ACKNOWLEDGEMENTS</b>	ii
	<b>ABSTRACT</b>	iii
	<b>CONTENTS</b>	iv
	<b>LIST OF FIGURES</b>	vii
	<b>LIST OF TABLES</b>	viii
	<b>ABBREVIATIONS AND ACRONYMS</b>	ix
<b>1.</b>	<b>INTRODUCTION TO FFT PROCESSOR</b>	
	1.1 overview	1
	1.2 Scope of The Report	3
	1.3 Thesis of Organization	3
<b>2.</b>	<b>FFT PROCESSOR ALGORITHMS</b>	
	2.1 Introduction	5
	2.2 Discrete Fourier Transform	8
	2.3 Fast Fourier Transform	8
	2.4 FFT Algorithms	9
	2.4.1 Radix-2 FFT Algorithm: DIT FFT	9
	2.4.2 Radix-2 FFT Algorithm: DIF FFT	11
	2.4.3 Radix-4 FFT Algorithm: DIF FFT	14
		20
	2.5 Inverse Fast Fourier Transform	
<b>3.</b>	<b>ARCHITECTURE OF SPARTAN-II FPGA AND VHDL</b>	
	3.1 Field Programmable Gate Arrays	21
	3.2 Brief Description of Xilinx FPGAs	22
	3.2.1 Features	23
	3.2.2 General Overview	24
	3.2.3 Architectural Description Spartan-II Array	25
	3.2.4 Input/Output Block	26
	3.2.5 I/O Banking	29
	3.2.6 Configurable Logic Block	30
	3.2.7 Block RAM	33

3.3 Hardware Description Languages.	33
3.3.1 Verilog HDL	34
3.3.2 VHDL	35
<b>4. DESIGN AND IMPLEMENTATION OF FFT PROCESSOR</b>	
4.1 Introduction	38
4.2 Algorithm Choice	38
4.3 FFT Processor	39
4.3.1 Data Input	39
4.3.2 FFT Computation	39
4.3.3 Data Output Process	40
4.4 Architecture	40
4.4.1 Coefficient ROM	40
4.4.2 Block RAM	40
4.4.3 Radix-4 Butterfly Implementation Details	40
4.4.4 Address generation unit	44
4.5 FFT Processor architecture	46
<b>5. DESIGN FLOW AND FINAL IMPLEMENTATION ON FPGA</b>	
5.1 Introduction	49
5.2 Specification	50
5.3 Design Entry	50
5.4 Simulation	50
5.5 User Constrain File	50
5.6 Synthesis	50
5.7 Implementation	51
5.8 Place and Route	51
5.9 FPGA Configuration	53
<b>6. RESULTS AND DISCUSSIONS</b>	
6.1 Simulation Results	54
6.2 MATLAB Results	54
6.3 Synthesis Results	61

<b>7.</b>	<b>CONCLUSIONS AND FUTURE SCOPE OF WORK</b>	
	7.1 Conclusions	62
	7.2 Future Scope of Work	62
	<b>REFERENCES</b>	63
	<b>APPENDIX</b>	
	Software Code ( CD Attached)	

## LIST OF FIGURES

Fig No	Title	Page No
2.1	(a) Analysis (b) Synthesis of the White Light Using Glass Prisms	5
2.2	Rectangular Pulse	7
2.3	Sin Function	7
2.4	8-Point FFT with 3-Stage DIT Radix-2 Structure	10
2.5	Basic Butterfly Computation in the DIT FFT Algorithm	11
2.6	$N=8$ -Point Decimation-in-Frequency FFT Algorithm.	12
2.7	Basic Butterfly Computation in the DIF-FFT Algorithm	13
2.8	Basic Butterfly Computation in a Radix-4 FFT Algorithm.	16
2.9	16-point, Radix-4 DIT FFT Algorithm	17
3.1	Basic Spartan-II Family FPGA Block Diagram	27
3.2	Spartan-II Input/Output Block (IOB)	28
3.3	Spartan-II I/O Banks	29
3.4	Spartan-II CLB Slice (two identical slices in each CLB)	32
4.1	Three sub-processes of the FFT Algorithm	39
4.2	Block Diagram Representation of the FFT processor	41
4.3	Block Diagram of Radix-4 Component	42
4.4	Asynchronous, Complex Multiplier	42
4.5	Complex Multiplier block diagram	43
4.6	Butterfly Processing Element Architecture for 1 <sup>st</sup> stage	45
4.7	Butterfly Processing Element Architecture for 2 <sup>nd</sup> stage	45
4.8	Flow chart for address generation unit	47
4.9	16-point FFT architecture	48
5.1	The High-Level Design Flow	49
5.2	Place and Route Data Flow	52
6.1	The input is stored in RAM	55
6.2	The inputs which are applied to 16-point FFT processor	56
6.3	Final output of the 16-point FFT processor	57
6.4	The matlab results of 16-point FFT	58
6.5	RTL View of 16-Point FFT Processor	61



## LIST OF TABLES

---

Table No	Title	Page No
2.1	Relation between Linear and Bit -Reverse Order	13
2.2	The comparison of radix-4 with radix-2 algorithms for N-point FFT	17
2.3	The quaternary system for decimal numbers 0 through 15.	18
2.4	The digit-reversed positions for a 16-point sequence	19
3.1	Spartan-II FPGA Family Members	24
3.2	Spartan-II Block RAM Amounts	33
6.1	Hardware utilization for 1 <sup>st</sup> stage of radix-4 FFT algorithm.	59
6.2	Hardware utilization for 2 <sup>nd</sup> stage of radix-4 FFT algorithm.	59
6.3	Hardware utilization for 16-point FFT processor	60

## ABBREVIATION AND ACRONYMS

---

AGU	Address Generation Unit
ASIC	Application Specific Integrated Circuits
CLB	Configurable Logic Block
DFT	Discrete Fourier Transform
DIF	Decimation-In-Frequency
DIT	Decimation-In-Time
DLL	Delay-Locked Loop
DSP	Digital Signal Processing
EDIF	Electronic Design Interchange Format
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
IOB	Input/Output Block
ISE	Integrated Software Environment
LC	Logic Cell
LUT	Look-Up Tables
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SDF	Standard Delay Format
UCF	User Constrain File
VLSI	Very Large Scale Integrated Circuit
VHDL	Very High Speed Integrated Circuits HDL
Floor Planning	Process of choosing the best grouping and Connectivity of logic in a design
Net list	Text description of circuit connectivity

## INTRODUCTION TO FFT PROCESSOR

---

### 1.1 Overview

The Discrete Fourier Transform (DFT) is one of the most fundamental operations in digital signal processing [1]. The Discrete Fourier transforms play an important role in many digital signal processing applications including acoustics, optics, telecommunications, speech, signal, image processing[17], linear filtering, quantum mechanics, noise reduction ,and image reconstruction . The Discrete Fourier Transform is a very popular technique used for converting signals in time domain to the frequency domain[9]. The DFT operation can be represented by the following expression.

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \quad \text{for } k=0, 1 \dots N-1 \quad (1.1)$$

Where  $X(k)$  = frequency transform of signal

$x(n)$  = data points in time domain

$N$  = number of data points

$W_N$  = twiddle factor

If the DFT operation is computed using the above expression then the complexity of the algorithm is  $O(N^2)$  for  $N$  data points and it will take a long time to compute the DFT. The Fast Fourier Transform (FFT) algorithm reduces the complexity of computing the Discrete Fourier Transform[2].

In 1965, Cooley and Tukey introduced the fast fourier transform, which efficiently and significantly reduces the computational cost of calculating  $N$ -point DFT from  $O(N^2)$  to  $O(N \log_2 N)$ . A large number of FFT algorithms have been developed by Cooley and Tukey [4]. Among these, the radix-2, radix-4, split-radix and FHT algorithms are the ones that have been mostly used for practical applications due to their simple structure, with a constant butterfly geometry, and the possibility of performing them “in place”. Most of the research to date for the implementation and benchmarking of FFT algorithms have been performed using general purpose processors [3,4], Digital Signal Processors(DSPs) and dedicated FFT processor ICs [5,6]. However, as Field Programmable Gate Arrays (FPGAs) have grown in

capacity, improved in performance, and decreased in cost, they have become a viable solution for performing computationally intensive tasks (i.e. computation of FFT), with the ability to tackle applications for custom chips and programmable DSP devices [7,8]. In this thesis, since we mainly focus on the fast structures of the DFT, the terms DFT and FFT will be used interchangeably. The order of the multiplicative complexity is commonly used to measure and compare the efficiency of the algorithms since multiplications are intrinsically more complicated among all operations [5]. It is well-known in the field of VLSI that among the digital arithmetic operations (addition, multiplication, shifting and addressing, etc.), multiplication is the operation that consumes most of the time and power required for the entire computation and, therefore, causes the resulting devices to be large and expensive. Therefore, reducing the number of multiplications in digital chip design is usually a desirable task [14].

FPGAs have become an attractive option for implementing signal processing applications because of their high processing power and customizability. The inclusion of new features in the FPGA fabric, such as a large number of embedded multipliers, adds to this attractiveness. FPGAs can now be considered for computationally demanding applications such as those in signal processing. Traditionally, the performance metrics for signal processing and indeed, most processing in general, have been latency and throughput.

Over last years, the interest in high speed multimedia communication systems has grown enormously [18]. These systems, including digital television, high speed wired data connections, and wireless local area networks, adopt multicarrier modulation scheme such as orthogonal frequency division multiplexing and discrete multi-tone to enhance data transmission rate [13]. In this scheme, sub carriers are generated by using fast Fourier transform (FFT). Therefore, it is very important to design the high speed FFT processor [11,15,19]. For the pipeline architecture, it is desirable to use the radix-4 algorithm because of the double processing rate compared to the radix-2 algorithm. However, the radix-4 algorithm requires more nontrivial multipliers than the radix-2 algorithm. Therefore, the radix-4 based algorithm, which reduces the number of nontrivial multipliers, is desired. The radix-2 algorithm, which can minimize the number of nontrivial multipliers required to implement the FFT Processor was presented previously. The same computational complexity as the split radix algorithm can be obtained but with a much spatially regular signal flow graph

(SFG) [10]. Only one full complex multiplication is required for every three columns, and the other two columns contain either pure trivial factor  $-j$  or a combination of  $-j$  and the special twiddle factor  $W_8$ , enabling an implementation with two additive operations and two constant scaling. However, based on the radix-2 butterfly unit, it has a lower processing rate than the radix-4 based algorithm. In this thesis, we propose an efficient FFT algorithm and present the results on its pipeline implementation. Pipelining the FFT process allows parallel processing and divides the complexity of computing the DFT into a number of stages. The result obtained from one stage is immediately available to the next stage without any delay. The proposed algorithm results in a reduced number of nontrivial multipliers like the radix-2 algorithm, but the processing rate is twice as fast as the radix-2 algorithm because it is based on the radix-4 butterfly unit [6].

## 1.2 Scope of The Report

The report is concentrated on 16-point FFT processors, and what architectures and algorithms are most suitable for dedicated FFT processors. The first part of the report gives a review on the theory behind the DFT and FFT algorithm. Some terminologies like radix butterflies, algorithms, architectures, etc., are introduced in this part. The second part of the report describes the main goal of this master's thesis project, i.e. to design and implement a FFT processor for transform length of 16-point. These transform lengths reduces the amount of algorithms, architectures, and so on, that could be taken into account when designing a processor according to these criteria. Some parts of the theory are therefore very briefly described compared to others, because of its limited usefulness in the considered area. What trade-offs have to be made? What architecture and algorithm should be used? What types of simulations should be done? How is testing performed? These are some of the questions that will be discussed in this report.

## 1.3 Organization of Thesis

Chapter-2: This chapter reviews the brief discussion on frequency analysis and the detail notes on DFT, FFT and FFT algorithms.

Chapter-3: This chapter focuses on Xilinx FPGA family on which FFT processor is implemented and also explains HDL languages.

Chapter-4: This chapter presents the complete architecture and implementation of fft processor.

Chapter-5: This chapter describes the process of implementation of FFT processor on FPGA.

Chapter-6: This chapter discusses the simulation results and these results are compared with matlab results and it also provides the synthesis report.

Chapter-7: This chapter concludes the total work and proposes the future scope of work.

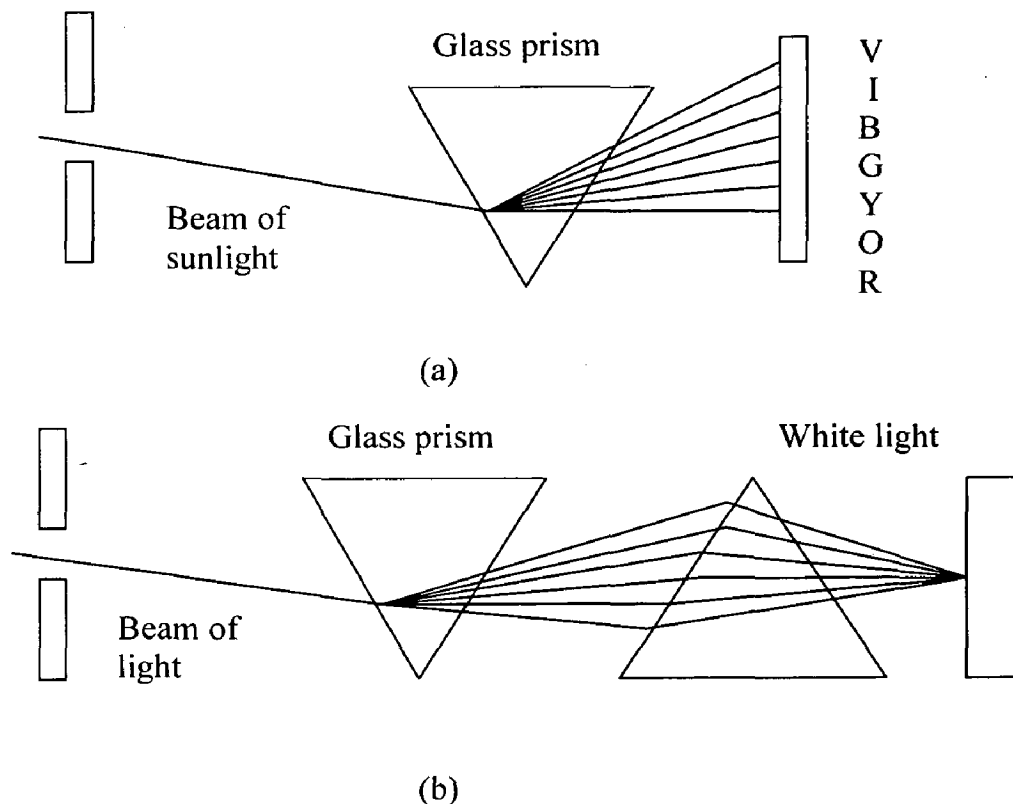
---

**FFT PROCESSOR ALGORITHMS**


---

**2.1 Introduction**

It is well known that a prism can be used to break up white light into the colors of the rainbow.



**Fig 2.1(a) Analysis (b) Synthesis of the White Light Using Glass Prisms.**

When a white light is passed through a prism, it will separate white light into different colors. Next place another prism upside-down with respect to the first and observe that the colors blended back into white light [1].

Frequency analysis of a signal involves the resolution of the signal into its frequency (sinusoidal) components. Instead of light, our signal wave forms are basically functions of time. The role of the prism is played by the Fourier analysis i.e. Fourier series and Fourier transform [1].

If we decompose a waveform into sinusoidal components, in much the same way that a prism separates white light into different colors, the sum of those

sinusoidal components results in the original waveform. on the other hand, If any of these components is missing, the result is a different signal[1].

## Fourier Series

The basic mathematical representation of periodic signals is the Fourier series, which is a linear weighted sum of harmonically related sinusoids or complex exponentials [1].

Linear combination of harmonically related complex exponentials of the form

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{j2\pi k F_0 t} \quad (2.1)$$

$$c_k = \frac{1}{T_p} \int_{T_p} x(t) e^{-j2\pi k F_0 t} dt \quad (2.2)$$

Is a periodic signal with fundamental period  $T_p=1/F_0$ . Hence we can think of the exponential signals

$$e^{j2\pi k F_0 t} \quad k=0, \pm 1, \pm 2, \dots$$

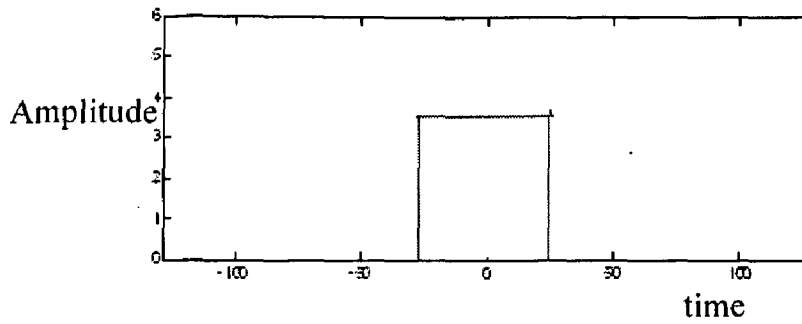
As the basic “building blocks”, from which we can construct periodic signals of various type by proper choice of the fundamental frequency and the coefficients  $\{c_k\}$ .  $F_0$  determines the fundamental period of  $x(t)$  and the coefficients  $\{c_k\}$  specify the shape of the waveform[1].

## Fourier Transform

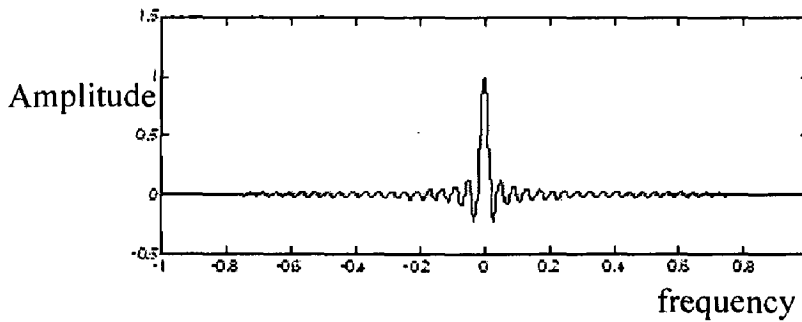
The Fourier transform is one of several mathematical tools that is useful in the analysis and design of LTI systems. These signal representations basically involve the decomposition of the signals in terms of sinusoidal (or complex exponential) components [7]. With such decomposition, a signal is said to be represented in the frequency domain. Most signals of practical interest can be decomposed into a sum of sinusoidal signal components for the class of finite energy signals, the decomposition is called the Fourier Transform [1].

Fourier transform of continuous a periodic signal is defined as.





**Fig 2.2 Rectangular Pulse**



**Fig 2.3 Sin Function**

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft} dt \quad (2.3)$$

$$x(t) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft} df \quad (2.4)$$

Frequency analysis of discrete-time signals is usually and most conveniently performed on a digital signal processor, which may be a general-purpose digital computer or specially designed hardware. To perform a frequency analysis on a discrete-time signal  $\{x(n)\}$ , we convert the time-domain sequence to an equivalent frequency-domain representation. We know that such a representation is given by the Fourier Transform  $X(w)$  of the sequence  $\{x(n)\}$ . However,  $X(w)$  is a continuous function of frequency and therefore, it is not a computationally convenient representation of the sequence  $\{x(n)\}$ .

In this section we consider the representation of a sequence  $\{x(n)\}$  by samples of its spectrum  $X(w)$ . Such a frequency-domain representation leads to the Discrete Fourier Transform, which is a powerful computational tool for performing frequency analysis of discrete-time signals [6].

## 2.2 Discrete Fourier Transform

The discrete Fourier transform is the counterpart of the Fourier transform in the discrete time domain. The definition of the DFT is given by the expression:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \quad \text{for } k=0, 1, \dots, N-1 \quad (2.5)$$

and the inverse DFT( IDFT) is expressed as:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)W_N^{-kn} \quad \text{for } n=0, 1, \dots, N-1 \quad (2.6)$$

where  $W_N^k = e^{-j2\pi k/N}$  is a sequence of twiddle factors of the DFT and is equally spaced around the unit cycle. In these equations  $x(n)$  is the sample value in the time domain and  $X(k)$  is the sample value in the frequency domain. If the sampling rate of a signal is  $F$ , the sequence of time-domain sampling locations becomes

$$0, 1/F, 2/F, 3/F, \dots, (N-1)/F$$

Thus, corresponding to these time-domain samples, the sequence of frequency-domain sampling locations will be

$$0, F/N, 2F/N, 3F/N, \dots, (N-1)F/N.$$

These equations show that the complexity of a direct computation of DFTs and IDFTs is  $O(N^2)$ , hence the long transforms considered to be very costly in a straight forward computation. The FFT algorithm deals with these complexity problems by exploiting regularities in the DFT algorithm [2].

## 2.3 Fast Fourier Transform

At the outset it should be pointed out that the FFT is not a different transform from the DFT, but rather it represents a means for computing the DFT with a considerable reduction in the number of computations.

Fast Fourier Transform, as the name suggests, is a fast and efficient way of computing the DFT. This algorithm was independently presented by Cooley-Tukey in 1965. A direct computation of the DFT or IDFT requires  $N^2$  complex multiplications and  $N(N-1)$  complex additions. FFT removes the redundant multiplication and addition/subtraction operations seen in the naive approach of direct computation of DFT. The DFT can be computed in  $O(N \log_2 N)$  multiplications by using the FFT[3].

## 2.4 FFT Algorithms:

The following algorithms are used for computing the DFT efficiently.

- Divide-and-Conquer Approach.
- Radix -2
- Radix -4
- Split-Radix

And also exists radix-16, 32, 2<sup>N</sup>...etc. In the following section, we presents

- (1) Radix-2 decimation-in-time algorithm (DIT).
- (2) Radix-2 decimation-in-frequency algorithm (DIF).
- (3) Radix-4 decimation-in-frequency algorithm (DIF).

### 2.4.1 Radix-2 FFT Algorithm: DIT FFT

The fast Fourier transform algorithm achieves its computational efficiency through a divide and conquer strategy. The essential idea is a grouping of the time and frequency samples such that the DFT summation over N values can be expressed as a combination of DFT summations over N/2 samples. When N is a power of two, this process of grouping can be repeatedly applied until the DFT summation has been reduced to a combination of DFT summation over only two samples [1]. For example, when N is a power of two, equation 2.5 can be decomposed as follows [1]:

$$\begin{aligned}
 X(k) &= \sum_{n=0}^{N/2-1} x(2n)W_N^{2kn} + \sum_{n=0}^{N/2-1} x(2n+1)W_N^{(2n+1)k} \\
 &= \sum_{n=0}^{N/2-1} x(2n)W_N^{2kn} + W_N^k \sum_{n=0}^{N/2-1} x(2n+1)W_N^{(2n)k}
 \end{aligned} \tag{2.7}$$

Define two (N/2)-point sequences (h (n)) and (g (n)) as the even and odd elements of (x(n)), respectively. Then,

$$h(n) = x(2n) \tag{2.8}$$

$$g(n)=x(2n+1) \tag{2.9}$$

The discrete Fourier transform of the two (N/2)-point sequences can be defined as follows

$$H(k) = \sum_{n=0}^{N/2-1} h(n)W_{N/2}^{kn} \tag{2.10}$$

$$G(k) = \sum_{n=0}^{N/2-1} g(n)W_{N/2}^{kn} \quad (2.11)$$

Thus, the discrete Fourier transform of  $x(n)$  can be expressed in terms of even and odd elements as[1]:

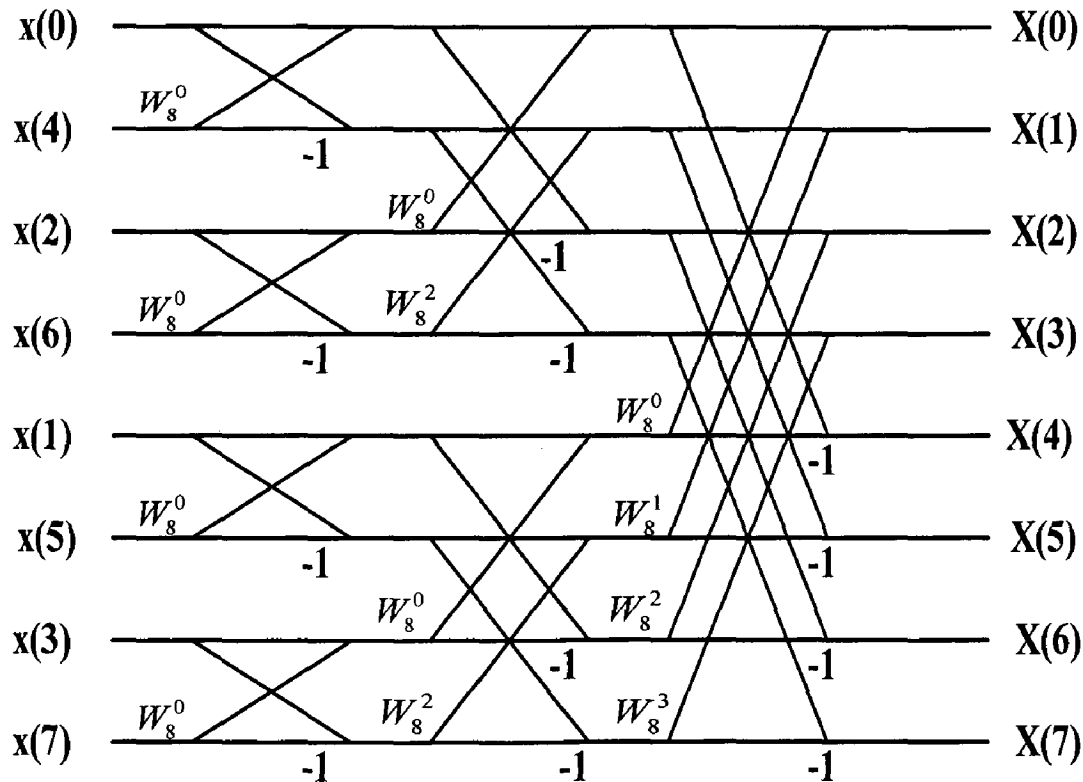
$$X(k) = \sum_{n=0}^{N/2-1} h(n)W_{N/2}^{kn} + W_N^k \sum_{n=0}^{N/2-1} g(n)W_{N/2}^{kn} \quad (2.12)$$

In terms of  $H(k)$  and  $G(k)$ , we have:

$$X(k) = H(k) + W_N^k G(k) \quad (2.13)$$

For the coefficient at  $(K + N/2)$ , we obtain:

$$X(k+N/2) = H(k) - W_N^k G(k) \quad (2.14)$$

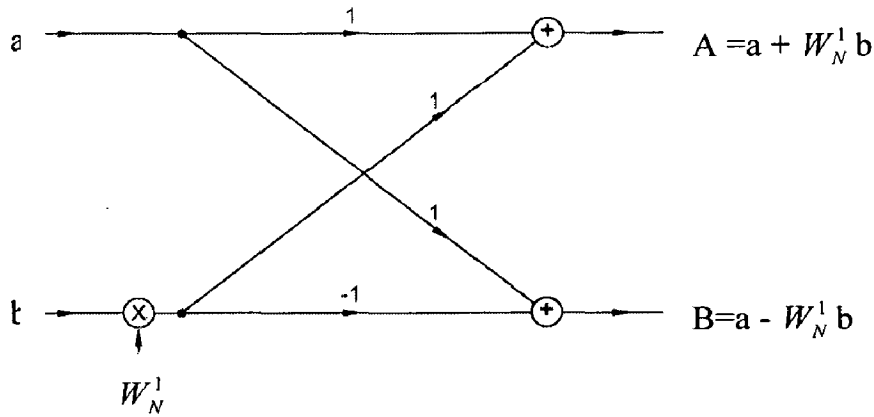


**Fig 2.4 8-Point FFT with 3-Stage Decimation-in-Time Radix-2 Structure**

Therefore,  $N$ -Point discrete Fourier transform can be represented by two  $(N/2)$ -point discrete Fourier transform. Since  $N$  is a power of two, the above partitioning scheme can be iteratively applied to the sequences  $(h(n))$  and  $(g(n))$  by  $N/2$  elements. These partitions can be carried out until the two-point DFT is reached. The process is

known as decimation-in-time, depicted in fig 2.4 for  $N = 8$ . Figure 2.5 shows the basic butterfly computation in the decimation-in-time FFT algorithm [1].

The structure shown in fig. 2.4 is called decimation-in-time (DIT). Its basic module is a radix-2 butterfly shown in fig.2.5 in which two points 'a' and 'b' are computed to give two output points 'A' and 'B' via the operations represented by equations, 2.13 and 2.14. In addition, the input is in bit-reverse order and output is in linear order [2].



**Fig 2.5 Basic Butterfly Computation in the Decimation-In-Time FFT Algorithm**

### 2.4.2 Radix-2 FFT Algorithm: DIF FFT

We begin by splitting the DFT into two summations, one of which involves the sum over the first  $N/2$  data points and the second sum involves the last  $N/2$  data points, thus we obtain

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W_N^{kn} + \sum_{n=N/2}^{N-1} x(n)W_N^{kn} = \sum_{n=0}^{(n/2)-1} x(n)W_N^{kn} + W_N^{Nk/2} \sum_{n=0}^{(n/2)-1} x\left(n + \frac{N}{2}\right)W_N^{kn} \quad (2.15)$$

Since  $W_N^{kN/2} = (-1)^k$ , the above expression can be rewritten as

$$X(k) = \sum_{n=0}^{(n/2)-1} \left[ x(n) + (-1)^k x\left(n + \frac{N}{2}\right) \right] W_N^{kn} \quad (2.16)$$

Now, let us split (decimate)  $X(k)$  into the even- and odd-numbered samples. Thus we obtain final expression as

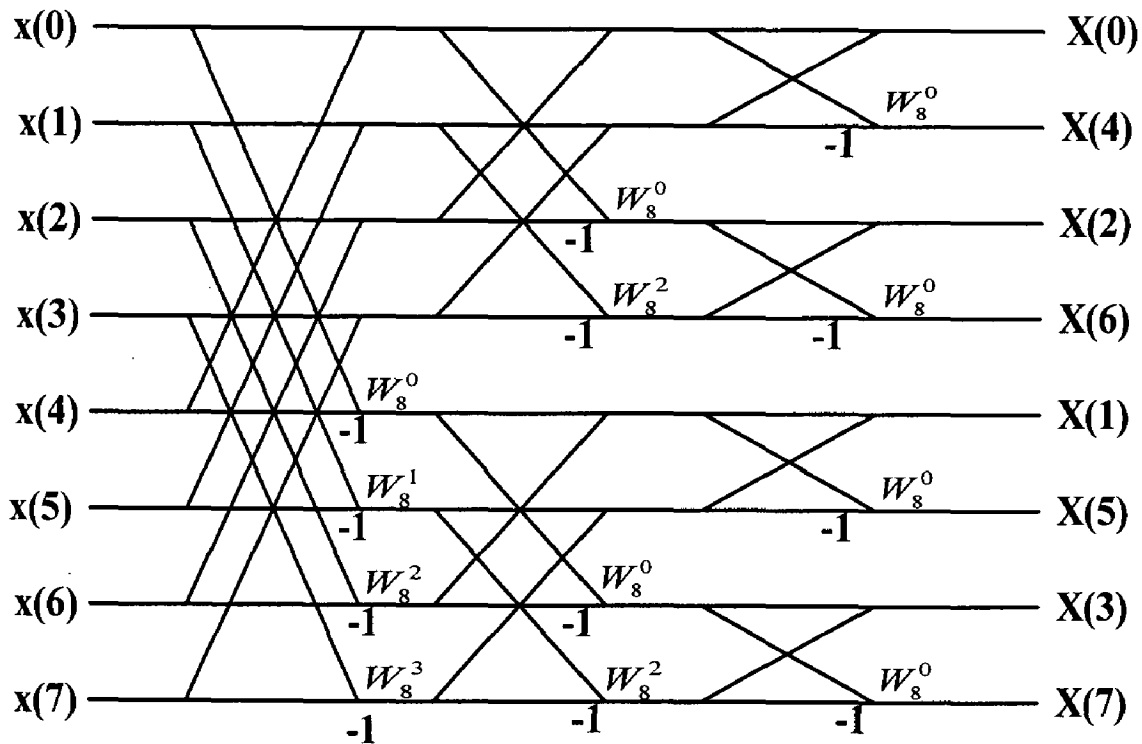
$$X(2k) = \sum_{n=0}^{(N/2)-1} h(n)W_{N/2}^{kn} \quad \text{for } k=0, 1, 2, \dots, \frac{N}{2}-1 \quad (2.17)$$

$$X(2k+1) = \sum_{n=0}^{(N/2)-1} g(n)W_{N/2}^{kn} \quad \text{for } k=0, 1, 2, \dots, \frac{N}{2}-1 \quad (2.18)$$

Where,

$$h(n) = x(n) + x\left(n + \frac{N}{2}\right) \quad \text{for } n=0, 1, 2, \dots, \frac{N}{2}-1$$

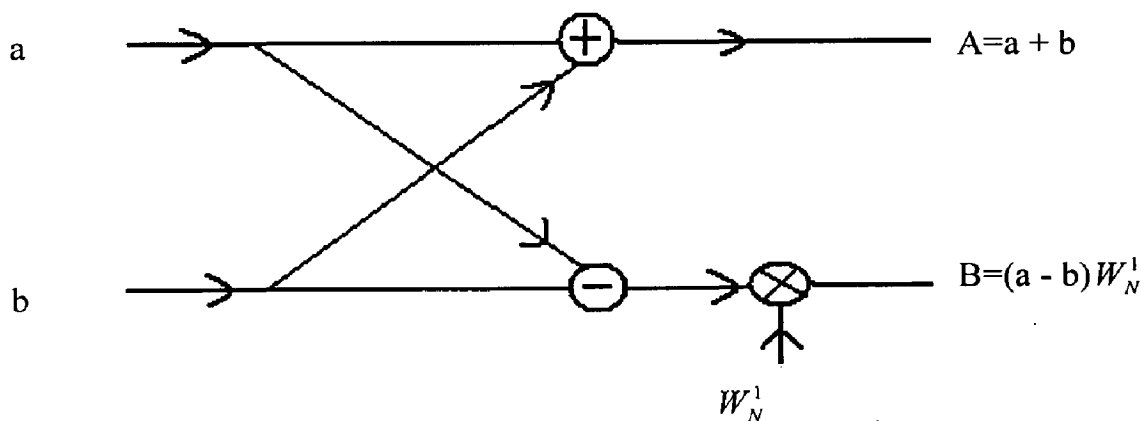
$$g(n) = \left[ x(n) - x\left(n + \frac{N}{2}\right) \right] W_N^n \quad (2.19)$$



**Fig 2.6 N=8-Point Decimation-in-Frequency FFT Algorithm.**

This computation procedure can be repeated through decimation of the  $N/2$ -point DFTs,  $X(2k)$ ,  $X(2k+1)$ . This process is known as decimation-in-frequency (DIF). The computation of the  $N$ -point DFT via DIF FFT algorithm requires same number of complex multiplications and complex additions as in the decimation-in-time algorithm. 8-point ( $N=8$ ) DIF algorithm is given in fig.2.6. we observe that the basic computation in this figure involves the butterfly operation illustrated in fig.2.7 and the input data  $x(n)$  occurs in natural order, but the output DFT occurs in bit-reversed order[3].

We observed from the above discussion is each radix-2 butterfly requires one complex multiplication and two complex additions. Observing fig.2.6 and fig.2.7, it can be seen that there are  $\log_2 N$  radix-2 butterfly stages for N-point FFT and each stage has  $N/2$  radix-2 butterflies. Therefore, there are totally  $(N/2) \log_2 N$  radix-2 butterflies in an N-point FFT. Both structures employ the in-place algorithm. In-place means that the computed outputs can be placed on the same storage as the inputs. Moreover, the addressing for the input and output data can be shared [1].



**Fig 2.7 Basic Butterfly Computation in the DIF-FFT Algorithm**

**Table 2.1 Relation between Linear and Bit -Reverse Order**

LINEAR		BIT-REVERSE	
DECIMAL	DIGIT	DIGIT	DECIMAL
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

### 2.4.3 Radix-4 FFT Algorithm: DIF FFT

Whereas a radix-2 FFT divides an N-point sequence successively in half until only two-point DFTs remain, a radix-4 FFT divides an N-point sequence successively in quarters until only four-point DFTs remain. An N-point sequence is divided into four N/4-point sequences; each N/4-point sequence is broken into four N/16-point sequences, and so on, until only four-point DFTs are left. The four-point DFT is the core calculation (butterfly) of the radix-4 FFT, just as the two-point DFT is the butterfly for a radix-2 FFT [1].

A radix-4 FFT essentially combines two stages of a radix-2 FFT into one, so that half as many stages are required. Although addressing of data and twiddle factors is more complex, a radix-4 FFT requires fewer calculations than a radix-2 FFT. Like the radix-2 FFT, the radix-4 FFT requires data scrambling and/or unscrambling. However, radix-4 FFT sequences are scrambled and unscrambled through digit reversal, rather than bit reversal as in the radix-2 FFT. Digit reversal is described later in this section [1].

The radix-4 DIF FFT expresses the DFT equation as four summations, and then divides it into four equations, each of which computes every fourth output sample. The following equations illustrate radix-4 decimation in frequency.

$$\begin{aligned}
 X(k) &= \sum_{n=0}^{N-1} x(n)W_N^{nk} \\
 &= \sum_{n=0}^{(N/4)-1} x(n)W_N^{nk} + \sum_{n=N/4}^{(N/2)-1} x(n)W_N^{nk} + \sum_{n=N/2}^{(3N/4)-1} x(n)W_N^{nk} + \sum_{n=3N/4}^{(N)-1} x(n)W_N^{nk} \\
 &= \sum_{n=0}^{(N/4)-1} x(n)W_N^{nk} + \sum_{n=0}^{(N/4)-1} x(n+N/4)W_N^{(n+N/4)k} \\
 &\quad + \sum_{n=0}^{(N/4)-1} x(n+N/2)W_N^{(n+N/2)k} + \sum_{n=0}^{(N/4)-1} x(n+3N/4)W_N^{(n+3N/4)k} \\
 &= \sum_{n=0}^{(N/4)-1} [x(n) + W_N^{k(N/4)}x(n+N/4) + W_N^{k(N/2)}x(n+N/2) + W_N^{k(3N/4)}x(n+3N/4)]W_N^{nk}
 \end{aligned} \tag{2.20}$$

The three twiddle factor coefficients can be expressed as follows:

$$W_N^{k(N/4)} = (e^{-j2\pi/N})^{k(N/4)} = (e^{-j\pi/2})^k = (\cos(\pi/2) - j\sin(\pi/2))^k = (-j)^k \tag{2.21}$$

Similarly



$$W_N^{k(N/2)} = (-1)^k \quad (2.22)$$

$$W_N^{k(3N/4)} = j^k \quad (2.23)$$

Equation 2.20 can thus be expressed as

$$X(k) = \sum_{n=0}^{(N/4)-1} [x(n) + (-j)^k x(n + N/4) + (-1)^k x(n + N/2) + (j)^k x(n + 3N/4)] W_N^{nk} \quad (2.24)$$

Four sub-sequences of the output (frequency) sequence are created by setting  $k=4r$ ,  $k=4r+1$ ,  $k=4r+2$  and  $k=4r+3$ :

$$X(4r) = \sum_{n=0}^{(N/4)-1} [(x(n) + x(n + N/4) + x(n + N/2) + x(n + 3N/4)) W_N^0] W_{N/4}^{nr} \quad (2.25)$$

$$X(4r+1) = \sum_{n=0}^{(N/4)-1} [(x(n) - jx(n + N/4) - x(n + N/2) + jx(n + 3N/4)) W_N^n] W_{N/4}^{nr} \quad (2.26)$$

$$X(4r+2) = \sum_{n=0}^{(N/4)-1} [(x(n) - x(n + N/4) + x(n + N/2) - x(n + 3N/4)) W_N^{2n}] W_{N/4}^{nr} \quad (2.27)$$

$$X(4r+3) = \sum_{n=0}^{(N/4)-1} [(x(n) + jx(n + N/4) - x(n + N/2) - jx(n + 3N/4)) W_N^{3n}] W_{N/4}^{nr} \quad (2.28)$$

For  $r = 0$  to  $(N/4)-1$ .

$X(4r)$ ,  $X(4r+1)$ ,  $X(4r+2)$ , and  $X(4r+3)$  are  $N/4$ -point DFTs. Each of their  $N/4$  points is a sum of four input samples ( $x(n)$ ,  $x(n+N/4)$ ,  $x(n+N/2)$  and  $(n+3N/4)$ ), each multiplied by either  $+1$ ,  $-1$ ,  $j$ , or  $-j$ . The sum is multiplied by a twiddle factor ( $W_N^0, W_N^n, W_N^{2n}$ , or  $W_N^{3n}$ ) [1].

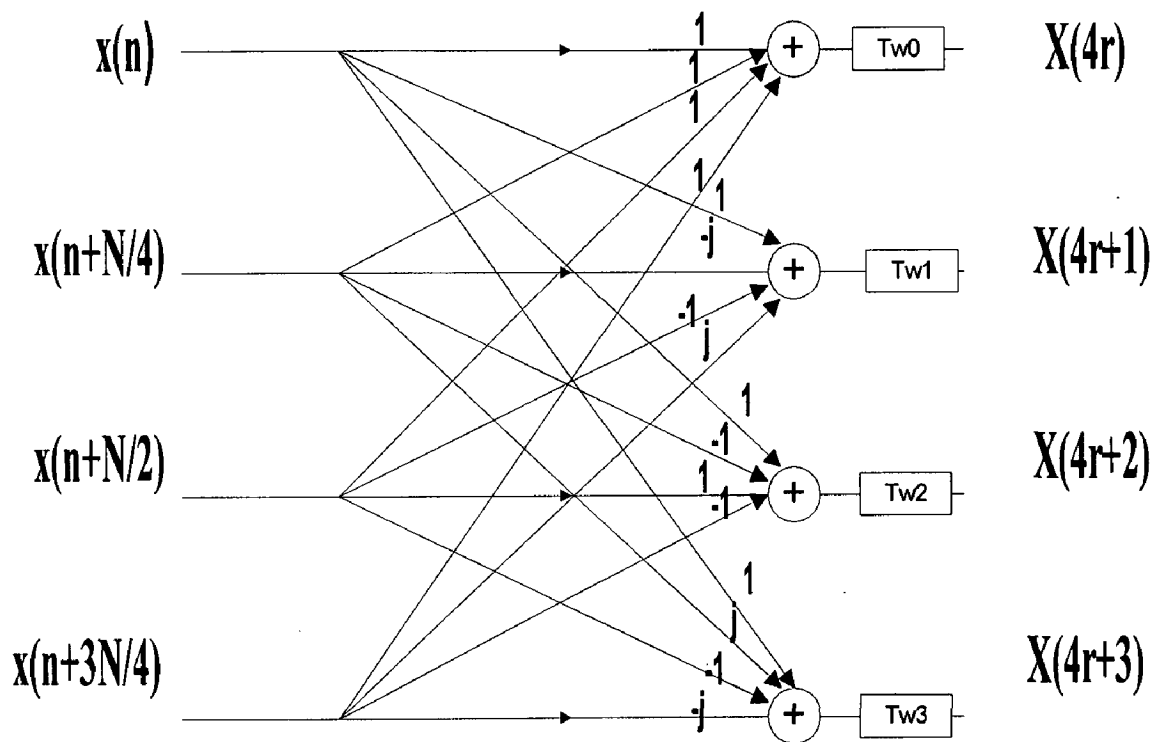
The four one-point DFT equations make up the butterfly calculation of the radix-4 FFT. A radix-4 butterfly is shown graphically in fig 2.8.

The output of each leg represents one of the four equations which are combined to make a four-point DFT. These four equations correspond to equations,(2.25) through (2,28), for one point rather than  $N/4$  points. The radix-4 butterfly expressed in matrix form as

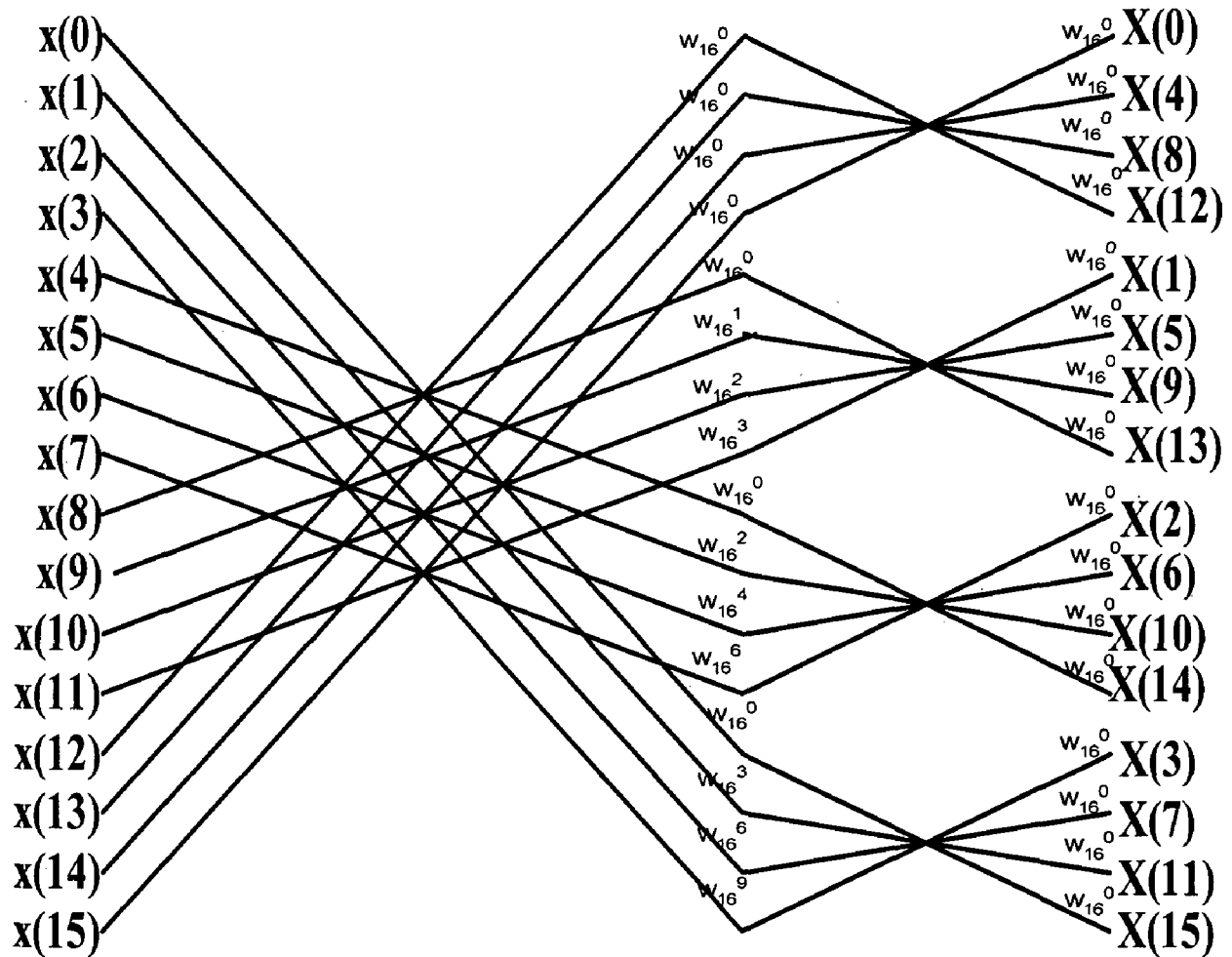
$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \begin{bmatrix} W_N^0 x(0) \\ W_N^q x(1) \\ W_N^{2q} x(2) \\ W_N^{3q} x(3) \end{bmatrix}$$

Where  $q = 0,1,2,3$ .

The 16-point radix-4 decimation-in-frequency FFT algorithm is shown in fig 2.9 .Its input in normal order and its output is in digit-reversed order. The computational complexity of radix-2 and radix-4 algorithm for N-point FFT is depicted in Table.2.2.



**Fig 2.8 Basic Butterfly Computation in a Radix-4 FFT Algorithm.**



**Fig 2.9 16-point, Radix-4 DIT FFT Algorithm**

The radix-4 butterfly is consequently larger and more complicated than a radix-2 butterfly; however, fewer butterflies are needed. Specifically,  $N/4$  butterflies are used in each of  $(\log_4 N)$  stages, which is one quarter the number of butterflies in a radix-2 FFT[16].

**Table 2.2 The comparison of radix-4 with radix-2 algorithms for N-point FFT**

	<b>Complex Multiplications</b>	<b>Complex Additions</b>	<b>Number of Stages</b>	<b>No. of Butterflies (for each Stage)</b>
<b>Radix-2</b>	$(N/2)\log_2 N$	$N \log_2 N$	$\log_2 N$	$N/2$
<b>Radix-4</b>	$(3N/4) \log_4 N$	$3N \log_4 N$	$\log_4 N$	$N/4$

## Digit Reversal

Whereas bit reversal reverses the order of bits in binary (base 2) numbers, digit reversal reverses the order of digits in quarternary (base 4) numbers. Every two bits in the binary number system correspond to one digit in the quarternary number system. (For example, binary 1110 = quarternary 32.) The quarternary system is illustrated in below table for decimal numbers 0 through 15[16].

**Table 2.3 The quarternary system for decimal numbers 0 through 15.**

Decimal	Binary	Quarternary
0	0000	00
1	0001	01
2	0010	02
3	0011	03
4	0100	10
5	0101	11
6	0110	12
7	0111	13
8	1000	20
9	1001	21
10	1010	22
11	1011	23
12	1100	30
13	1101	31
14	1110	32
15	1111	33

The radix-4 DIF FFT successively divides a sequence into four subsequences, resulting in an output sequence in digit-reversed order. A digit-reversed sequence is unscrambled by digit-reversing the data positions. For example, position 12 in quarternary (six in decimal) becomes position 21 in quarternary (nine in decimal) after digit reversal. Therefore, data in position six is moved to position nine when the

digit reversed sequence is unscrambled. The digit-reversed positions for a 16-point sequence (samples X(0) through X(15)) are shown in Table 2.4.

In an N-point radix-4 FFT, only the number of digits needed to represent N locations is reversed. Two digits are needed for a 16-point FFT, three digits for a 64-point FFT, and five digits for a 1024-point FFT[16].

**Table 2.4 The digit-reversed positions for a 16-point sequence**

Sample, Sequential Order	Sequential Location		Digit-Reversed Location		Sample, Digit-Reversed Order
	Decimal	Quarternary	Decimal	Quarternary	
X(0)	0	00	0	00	X(0)
X(1)	1	01	4	10	X(4)
X(2)	2	02	8	20	X(8)
X(3)	3	03	12	30	X(12)
X(4)	4	10	1	01	X(1)
X(5)	5	11	5	11	X(5)
X(6)	6	12	9	21	X(9)
X(7)	7	13	13	31	X(13)
X(8)	8	20	2	02	X(2)
X(9)	9	21	6	12	X(6)
X(10)	10	22	10	22	X(10)
X(11)	11	23	14	32	X(14)
X(12)	12	30	3	03	X(3)
X(13)	13	31	7	13	X(7)
X(14)	14	32	11	23	X(11)
X(15)	15	33	15	33	X(15)

## 2.5. Inverse Fast Fourier Transform

The inverse FFT (IFFT) defined by eqs., (2.29) can be changed to the following form:

$$x(n) = \frac{1}{N} \left[ \sum_{k=0}^{N-1} X^*(k) W_N^{kn} \right] \quad \text{for } n=0, 1, \dots, N-1 \quad (2.29)$$

where the notation of the superscript \* denotes the conjugated data. If the input frequency samples are conjugated, the bracketed term in eqs., (2.29) is exactly an FFT operation. Thus, the computation of the IFFT is the same as the FFT except the input data sequence of the first stage and output data sequence of the last stage are conjugated. The results computed by the FFT instructions are reference values and not exact values shown by eqs., (2.29). If users like to get exact values, the results have to be multiplied by a factor. This factor can be derived from the length of data N and the total number of scaling for the block data [1].

---

**ARCHITECTURE OF SPARTAN-II FPGA AND VHDL**

---

### 3.1 Field Programmable Gate Arrays

A field programmable gate array (FPGA) is an inexpensive hardware component, which allows the user to program its functionality quickly and inexpensively. This allows for cheaper prototyping and shorter time to-market of hardware designs. FPGAs have a lower gate density than full custom (customized VLSI chips) and semi custom (mask programmed gate arrays) design methodologies. FPGAs were first introduced in the mid-1980s to replace multi-chip glue logic circuits with a single reconfigurable solution [12]. FPGAs have far outgrown their sole use as a replacement for simple glue logic circuits [13]. Presently, FPGA applications include signal and image processing, graphic accelerators, military target correlation/recognition, cryptography, reconfigurable computing, and on-chip coprocessors. FPGAs are utilized in four major design areas: rapid prototyping, emulation, pre-production, and full-production [14]. FPGAs are the direct result of the convergence of two distinct technologies: Programmable Logic Devices (PLDs) and Application Specific Integrated Circuits (ASICs) [15]. A simple PLD consists of arrays of AND and OR gates that can be used to create basic circuit designs. ASICs are custom-made chips generally used in high volume applications because non-recurring engineering costs (NREs) are much higher than in an FPGA design cycle. FPGAs are sized from thousands of gates to tens-of-million gates and are available in a variety of sizes with different packaging, internal logic blocks, and process technologies [25].

Internal FPGA architectures are commonly constructed using a symmetric tile structure containing a network of switchboxes, logic blocks, wire channels, and input-output blocks. A switchbox (SB) is a location in the FPGA fabric that provides a method to connect internal wires together. The switchbox allows horizontal wire segments to switch to vertical wire segments and vice versa. The switchbox also allows horizontal wire segments to connect to other horizontal wire segments as well as connecting vertical wires to other vertical wires. The size and contents within a logic block vary greatly depending on the manufacture and target market. For

example, FPGAs targeted towards cost-effective solutions typically contain simpler logic blocks than an FPGA targeted for high-performance applications. Although the contents within logic blocks can vary for different architectures, there are two basic building blocks found in a logic block: memory elements and function generators. Memory elements provide designers with the ability to temporarily store information until desired conditions are met. Function generators can be configured to produce any function up to the number of inputs into the function generator. Depending on the architecture, some function generators can operate in different modes such as random access memory (RAM), read only memory (ROM), or more complex modes like shift registers. FPGAs are configured through a bitstream that is loaded into the device. A bitstream is a file created by the FPGA manufacturer that configures the switchboxes, logic blocks, and other internal FPGA logic[25].

FPGAs have redefined the boundaries of digital electronics allowing designers to build systems piecewise. Multiple designers can rapidly test and verify the functionality of each individual piece of a system to ensure proper functionality prior to merging the entire system together. With increasing interest in reconfigurable computing, FPGAs are recognized as the most viable, cost effective solution. Whether a design is statically or dynamically reconfigurable, FPGAs provide rapid programmability, and a short time to market design cycle. Many companies have marketed FPGAs, the major companies being Xilinx, Actel and Altera. Reprogrammable FPGAs use EPROM, EEPROM or static RAM technology. Xilinx FPGAs, which use static RAM technology, are the FPGAs used in this thesis[25].

### 3.2 Brief Description of Xilinx FPGAs

The Spartan-II 2.5V Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The six-member family offers densities ranging from 15,000 to 200,000 system gates, as shown in Table 1. System performance is supported up to 200 MHz. Spartan-II devices deliver more gates, I/Os, and features per dollar than other FPGAs by combining advanced process technology with a streamlined Virtex-based architecture. Features include block RAM (to 56K bits), distributed RAM (to 75,264 bits), 16 selectable I/O standards, and four Delay-Locked Loops (DLLs). Fast,



predictable interconnect means that successive design iterations continue to meet timing requirements. The Spartan-II family is a superior alternative to maskprogrammed ASICs. The FPGA avoids the initial cost, lengthy development cycles, and inherent risk of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary (impossible with ASICs)[25].

### 3.2.1 Features:

#### (1) Second generation ASIC replacement technology

- ❖ Densities as high as 5,292 logic cells with up to 200,000 system gates
- ❖ Streamlined features based on Virtex architecture
- ❖ Unlimited reprogrammability
- ❖ Very low cost
- ❖ Advanced 0.18 micron process

#### (2) System level features

- ❖ Select RAM hierarchical memory:
- ❖ 16 bits/LUT distributed RAM
- ❖ Configurable 4K bit block RAM
- ❖ Fast interfaces to external RAM
- ❖ Fully PCI compliant
- ❖ Low-power segmented routing architecture
- ❖ Full read back ability for verification/observability
- ❖ Dedicated carry logic for high-speed arithmetic
- ❖ Efficient multiplier support
- ❖ Cascade chain for wide-input functions
- ❖ Abundant registers/latches with enable, set, reset
- ❖ Four dedicated DLLs for advanced clock control
- ❖ Four primary low-skew global clock distribution nets
- ❖ IEEE 1149.1 compatible boundary scan logic

#### (3) Versatile I/O and packaging

- ❖ Pb-free package options
- ❖ Low-cost packages available in all densities

- ❖ Family footprint compatibility in common packages
  - ❖ 16 high-performance interface standards
  - ❖ Hot swap Compact PCI friendly
  - ❖ Zero hold time simplifies system timing
- (4) Fully supported by powerful Xilinx development system
- ❖ Foundation ISE Series: Fully integrated software
  - ❖ Alliance Series: For use with third-party tools
  - ❖ Fully automatic mapping, placement, and routing

### 3.2.2 General Overview

The Spartan-II family of FPGAs have a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable Input/Output Blocks (IOBs). There are four Delay-Locked Loops (DLLs), one at each corner of the die. Two columns of block RAM lie on opposite sides of the die, between the CLBs and the IOB columns. These functional elements are interconnected by a powerful hierarchy of versatile routing channels. This is shown in Fig.3.1[25]

**Table 3.1 Spartan-II FPGA Family Members**

Spartan-II Device	Logic Cells	System Gates (Logic and RAM)	CLB Array (R x C)	Total CLBs	Maximum Available User I/O	Total Distributed RAM Bits	Total Block RAM Bits
<b>XC2S15</b>	<b>432</b>	<b>15,000</b>	<b>8 x 12</b>	<b>96</b>	<b>86</b>	<b>6,144</b>	<b>16K</b>
<b>XC2S30</b>	<b>972</b>	<b>30,000</b>	<b>12 x 18</b>	<b>216</b>	<b>92</b>	<b>13,824</b>	<b>24K</b>
<b>XC2S50</b>	<b>1,728</b>	<b>50,000</b>	<b>16 x 24</b>	<b>384</b>	<b>176</b>	<b>24,576</b>	<b>32K</b>
<b>XC2S100</b>	<b>2,700</b>	<b>100,000</b>	<b>20 x 30</b>	<b>600</b>	<b>176</b>	<b>38,400</b>	<b>40K</b>
<b>XC2S150</b>	<b>3,880</b>	<b>150,000</b>	<b>24 x 36</b>	<b>864</b>	<b>260</b>	<b>55,296</b>	<b>48K</b>
<b>XC2S200</b>	<b>5,292</b>	<b>200,000</b>	<b>28 x 42</b>	<b>1,176</b>	<b>284</b>	<b>75,264</b>	<b>56K</b>

Spartan-II FPGAs are customized by loading configuration data into internal static memory cells. Unlimited reprogramming cycles are possible with this approach. Stored values in these cells determine logic functions and interconnections implemented in the FPGA. Configuration data can be read from an external serial PROM (master serial mode), or written into the FPGA in slave serial, slave parallel, or Boundary Scan modes [25].

Spartan-II FPGAs are typically used in high-volume applications where the versatility of a fast programmable solution adds benefits. Spartan-II FPGAs are ideal for shortening product development cycles while offering a cost-effective solution for high volume production [25].

Spartan-II FPGAs achieve high-performance, low-cost operation through advanced architecture and semiconductor technology. Spartan-II devices provide system clock rates up to 200 MHz. Spartan-II FPGAs offer the most cost-effective solution while maintaining leading edge performance. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan-II FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and distributed form), DLL clock drivers, programmable set and reset on all flip-flops, fast carry logic, and many other features [25].

### 3.2.3 Architectural Description Spartan-II Array

The Spartan-II user-programmable gate array, shown in Figure 1, is composed of five major configurable elements.

- ❖ IOBs provide the interface between the package pins and the internal logic
- ❖ CLBs provide the functional elements for constructing most logic
- ❖ Dedicated block RAM memories of 4096 bits each
- ❖ Clock DLLs for clock-distribution delay compensation and clock domain control
- ❖ Versatile multi-level interconnect structure

As can be seen in fig.3.1, the CLBs form the central logic structure with easy access to all support and routing structures. The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip. Values stored in static memory cells control all the configurable logic elements and

interconnect resources. These values load into the memory cells on power-up, and can reload if necessary to change the function of the device. Each of these elements will be discussed in detail in the following sections [25].

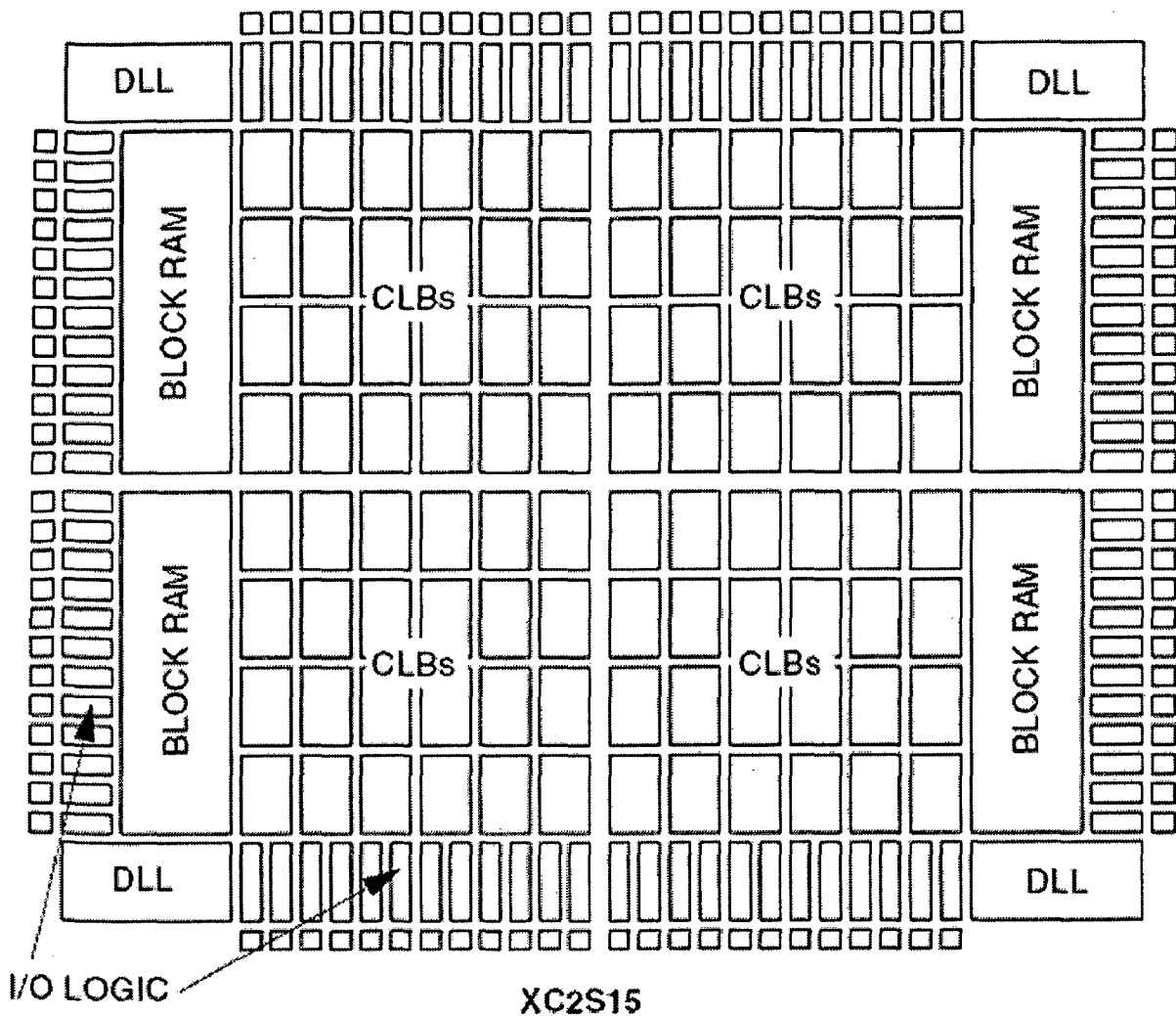
### 3.2.4 Input/Output Block

The Spartan-II IOB, as seen in fig.4.2, features inputs and outputs that support a wide variety of I/O signaling standards. These high-speed inputs and outputs are capable of supporting various state of the art memory and bus interfaces. The three IOB registers function either as edge-triggered D-type flip-flops or as level-sensitive latches. Each IOB has a clock signal (CLK) shared by the three registers and independent Clock Enable (CE) signals for each register [25].

In addition to the CLK and CE control signals, the three registers share a Set/Reset (SR). For each register, this signal can be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear A feature not shown in the block diagram, but controlled by the software, is polarity control. The input and output buffers and all of the IOB control signals have independent polarity controls [25].

Optional pull-up and pull-down resistors and an optional weak-keeper circuit are attached to each pad. Prior to configuration all outputs not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak-keeper circuits are inactive, but inputs may optionally be pulled up. The activation of pull-up resistors prior to configuration is controlled on a global basis by the configuration mode pins. If the pull-up resistors are not activated, all the pins will float. Consequently, external pull-up or pull-down resistors must be provided on pins required to be at a well-defined logic level prior to configuration [25].

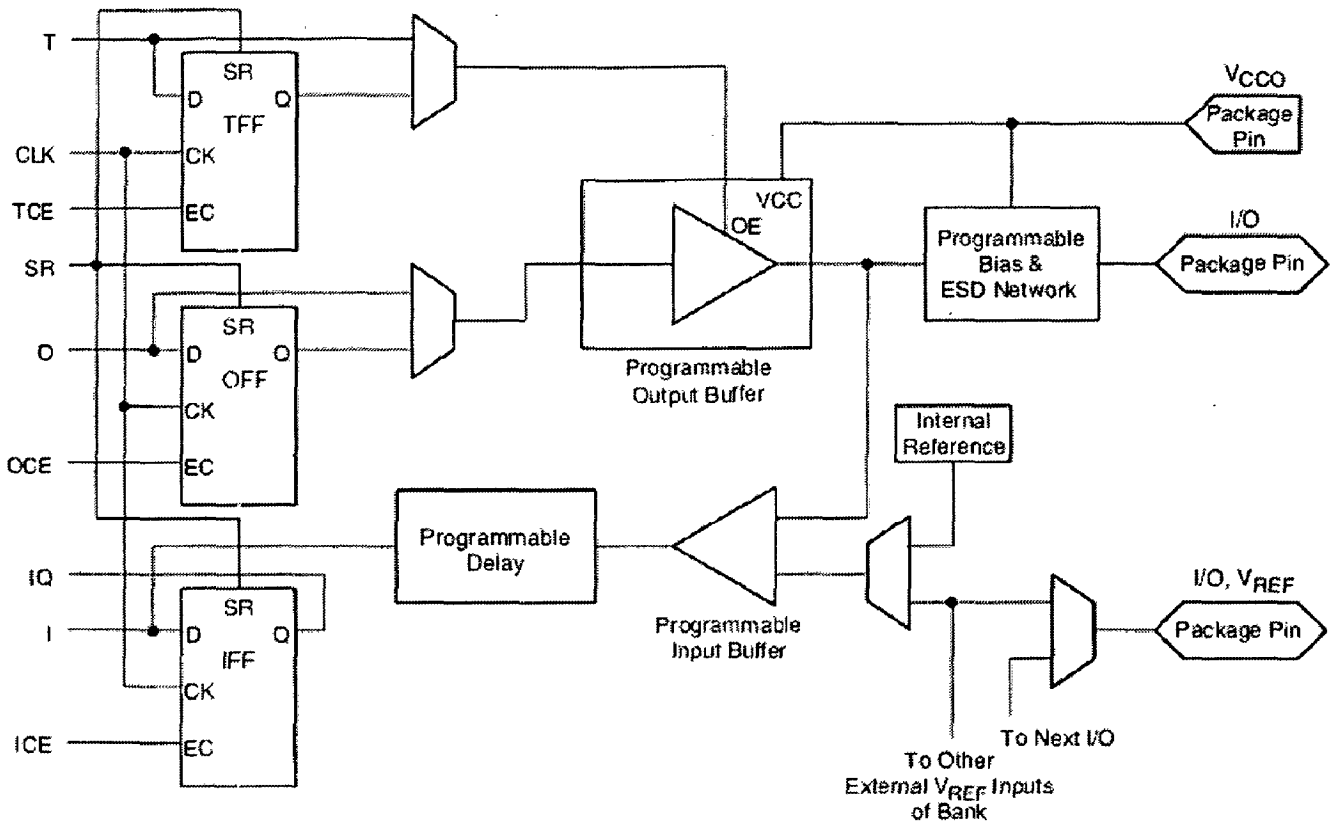
All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients. Two forms of over-voltage protection are provided, one that permits 5V compliance, and one that does not. For 5V compliance, a zener-like structure connected to ground turns on when the output rises to approximately 6.5V. When 5V compliance is not required, a conventional clamp diode may be connected to the output supply voltage, VCCO. The type of over-voltage protection can be selected independently for each pad [25].



**Fig 3.1 Basic Spartan-II Family FPGA Block Diagram**

### Input Path

A buffer in the Spartan-II IOB input path routes the input signal either directly to internal logic or through an optional input flip-flop. An optional delay element at the D-input of this flip-flop eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of the FPGA, and when used, assures that the pad-to-pad hold time is zero. Each input buffer can be configured to conform to any of the low-voltage signaling standards supported. In some of these standards the input buffer utilizes a user-supplied threshold voltage, VREF. The need to supply VREF imposes constraints on which standards can be used in close proximity to each other. There are optional pull-up and pull-down resistors at each input for use after configuration.



**Fig 3.2 Spartan-II Input/Output Block (IOB)**

## Output Path

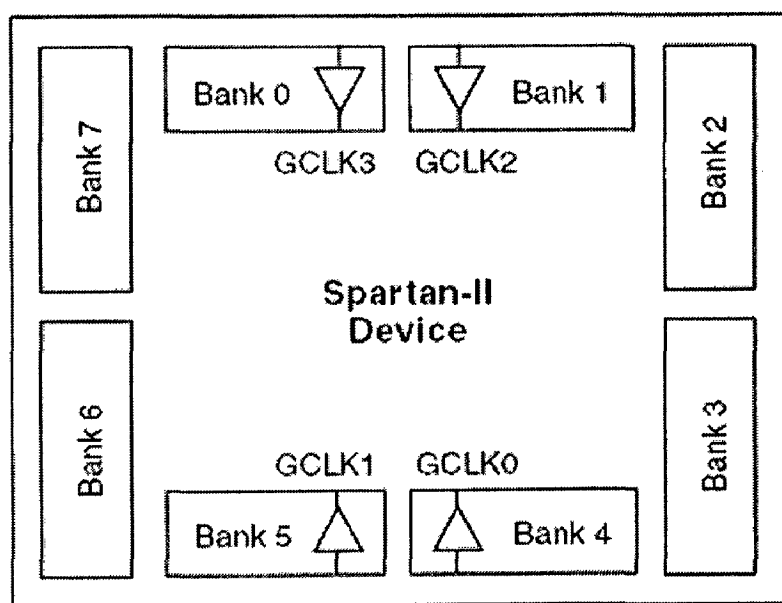
The output path includes a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop. The 3-state control of the output can also be routed directly from the internal logic or through a flip-flop that provides synchronous enable and disable. Each output driver can be individually programmed for a wide range of low-voltage signaling standards. Each output buffer can source up to 24 mA and sink up to 48 mA. Drive strength and slew rate controls minimize bus transients [25].

In most signaling standards, the output high voltage depends on an externally supplied VCCO voltage. The need to supply VCCO imposes constraints on which standards can be used in close proximity to each other. An optional weak-keeper circuit is connected to each output. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low to match the input signal. If the

pin is connected to a multiple-source signal, the weak keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way helps eliminate bus chatter. Because the weak-keeper circuit uses the IOB input buffer to monitor the input level, an appropriate VREF voltage must be provided if the signaling standard requires one. The provision of this voltage must comply with the I/O banking rules [25].

### 3.2.5 I/O Banking

Some of the I/O standards described above require VCCO and/or VREF voltages. These voltages are externally connected to device pins that serve groups of IOBs, called banks. Consequently, restrictions exist about which I/O standards can be combined within a given bank. Eight I/O banks result from separating each edge of the FPGA into two banks as shown in Figure 3). Each bank has multiple VCCO pins which must be connected to the same voltage. Voltage is determined by the output standards in use [25].



**Fig 3.3 Spartan-II I/O Banks**

Some input standards require a user-supplied threshold voltage,  $V_{REF}$ . In this case, certain user-I/O pins are automatically configured as inputs for the  $V_{REF}$  voltage. About one in six of the I/O pins in the bank assume this role.  $V_{REF}$  pins within a bank are interconnected internally and consequently only one  $V_{REF}$  voltage can be used within each bank. All  $V_{REF}$  pins in the bank, however, must be

connected to the external voltage source for correct operation. In a bank, inputs requiring  $V_{REF}$  can be mixed with those that do not but only one  $V_{REF}$  voltage may be used within a bank. Input buffers that use  $V_{REF}$  are not 5V tolerant. The  $V_{CCO}$  and  $V_{REF}$  pins for each bank appear in the device pinout tables. Within a given package, the number of  $V_{REF}$  and  $V_{CCO}$  pins can vary depending on the size of device. In larger devices, more I/O pins convert to  $V_{REF}$  pins. Since these are always a superset of the  $V_{REF}$  pins used for smaller devices, it is possible to design a PCB that permits migration to a larger device. All  $V_{REF}$  pins for the largest device anticipated must be connected to the  $V_{REF}$  voltage, and not used for I/O [25].

### 3.2.6 Configurable Logic Block

The basic building block of the Spartan-II CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and storage element. Output from the function generator in each LC drives the CLB output and the D input of the flip-flop. Each Spartan-II CLB contains four LCs, organized in two similar slices ; a single slice is shown in Figure 4. In addition to the four basic LCs, the Spartan-II CLB contains logic that combines function generators to provide functions of five or six inputs[25].

### Look-Up Tables

Spartan-II function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16x1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16x2-bit or 32x1-bit synchronous RAM, or a 16x1-bit dual-port synchronous RAM.

The Spartan-II LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-mode data. This mode can also be used to store data in applications such as Digital Signal Processing [25].

### Storage Elements

Storage elements in the Spartan-II slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by function generators within the slice or directly from slice inputs, bypassing



the function generators. In addition to Clock and Clock Enable signals, each slice has synchronous set and reset signals (SR and BY). SR forces a storage element into the initialization state specified for it in the configuration. BY forces it into the opposite state. Alternatively, these signals may be configured to operate asynchronously. All control signals are independently invertible, and are shared by the two flip-flops within the slice [25].

### Additional Logic

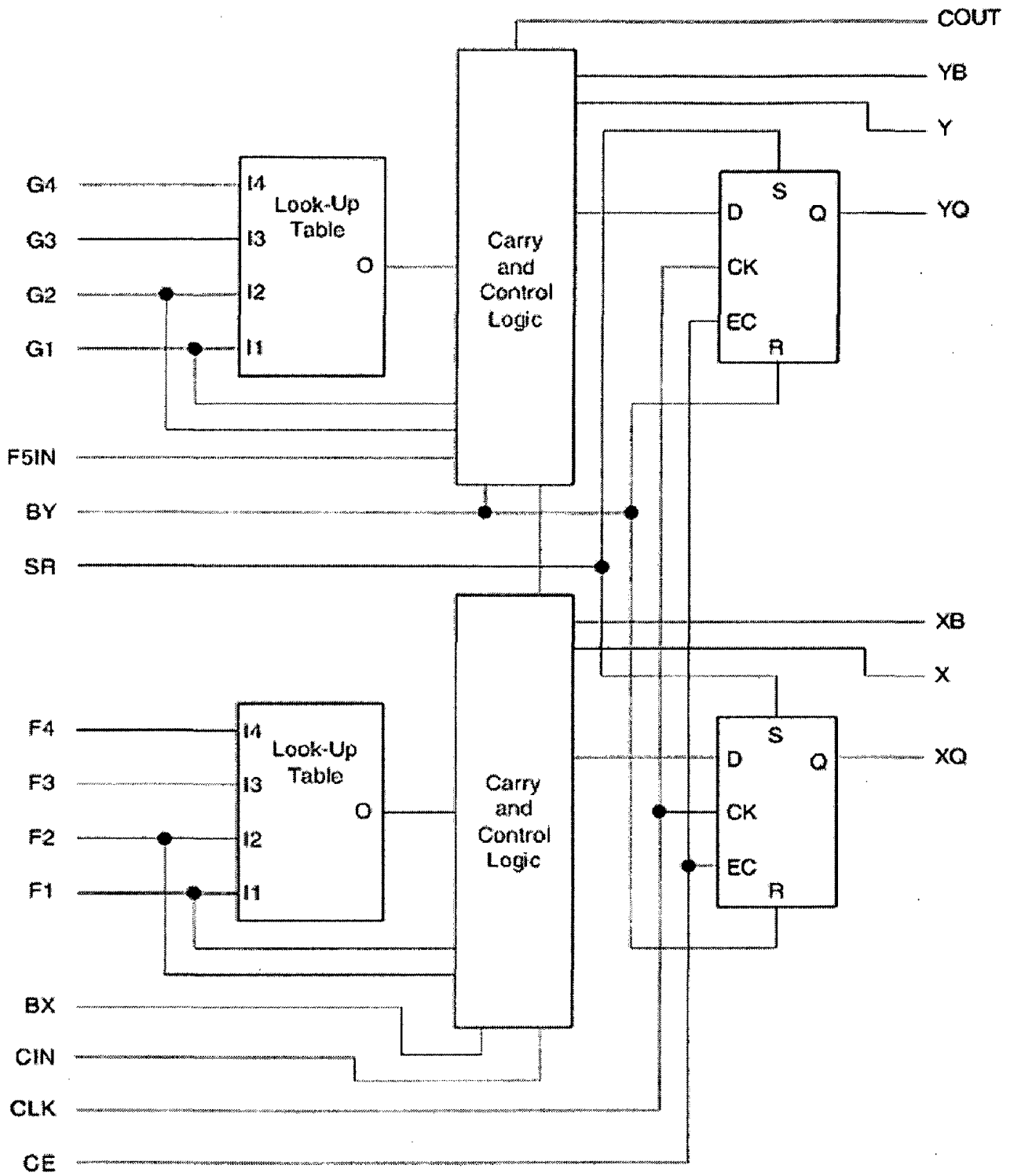
The F5 multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine inputs. Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs. Each CLB has four direct feed through paths; one per LC. These paths provide extra data input lines or additional local routing that does not consume logic resources[25].

### Arithmetic Logic

Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Spartan-II CLB supports two separate carry chains, one per slice. The height of the carry chains is two bits per CLB. The arithmetic logic includes an XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementation. The dedicated carry path can also be used to cascade function generators for implementing wide logic functions [25].

### BUFTs

Each Spartan-II CLB contains two 3-state drivers (BUFTs) that can drive on-chip busses. Each Spartan-II BUFT has an independent 3-state control pin and an independent input pin.



**Fig 3.4 Spartan-II CLB Slice (two identical slices in each CLB)**

### 3.2.7 Block RAM

Spartan-II FPGAs incorporate several large block RAM memories. These complement the distributed RAM Look-Up Tables (LUTs) that provide shallow memory structures implemented in CLBs.

Block RAM memory blocks are organized in columns. All Spartan-II devices contain two such columns, one along each vertical edge. These columns extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Spartan-II device eight CLBs high will contain two memory blocks per column, and a total of four blocks. Each block RAM cell is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion. The Spartan-II block RAM also includes dedicated routing to provide an efficient interface with both CLBs and other block RAMs[25].

**Table.3.2 Spartan-II Block RAM Amounts**

<b>Spartan-II Device</b>	<b>N0. of Blocks</b>	<b>Total Block RAM Bits</b>
<b>XC2S15</b>	<b>4</b>	<b>16K</b>
<b>XC2S30</b>	<b>6</b>	<b>24K</b>
<b>XC2S50</b>	<b>8</b>	<b>32K</b>
<b>XC2S100</b>	<b>10</b>	<b>40K</b>
<b>XC2S150</b>	<b>12</b>	<b>48K</b>
<b>XC2S200</b>	<b>14</b>	<b>56K</b>

### 3.3 Hardware Description Languages

In electronics, a **hardware description language** or **HDL** is a standard text-based format for describing either the behavior or the structure, or both, of an electronic circuit.

HDLs have two purposes. First, they are used to write a model for the expected behavior of a circuit before that circuit is designed and built. The model is

fed into a computer program, called a simulator that allows the designer to verify that his solution behaves correctly. Second, they are used to write a detailed description of a circuit that is fed into another computer program called a logic compiler. The output of the compiler is used to configure a programmable logic device that has the desired function. Often, the HDL code that has been simulated in the first step is re-used and compiled in the second step.

An HDL is analogous to a software programming language, but with subtle differences. Both types of language are processed by a compiler. An HDL compiler often works in several stages, first producing a logic description file in a proprietary format, then converting that to a logic description file in the industry-standard EDIF format, then converting that to a JEDEC (Joint Electron Device Engineering Council)-format file[21].

HDLs used by logic compilers include:

- ❖ **Verilog HDL**
- ❖ **VHDL**
- ❖ **AHDL** (a proprietary language used by Altera)
- ❖ **CUPL** (a proprietary language used by Logical Devices, Inc.)

The current trend is to move away from proprietary HDLs and towards the two leading standards, VHDL and Verilog HDL [22].

### **.3.3.1 VERILOG HDL**

Verilog HDL is a hardware description language used to design and document electronic systems. Verilog HDL allows designers to design at various levels of abstraction. Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. . At that time, Verilog was not standardized and the language modified itself in almost all the revisions that came out within 1984 to 1990. In 1990, Cadence recognized that if Verilog remained a closed language, the pressures of standardization would eventually cause the industry to shift to VHDL. Consequently, Cadence organized Open Verilog International (OVI), and in 1991 gave it the documentation for the Verilog Hardware Description Language. This was the event which "opened" the language.

In 1994, the IEEE 1364 working group was formed to turn the OVI LRM into an IEEE standard. This effort was concluded with a successful ballot in 1995, and Verilog became an IEEE standard in December, 1995 [20].

## Features of Verilog HDL

In Verilog HDL, a component is represented by a design module. The module declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component ports. The module body provides an "internal" view; it describes the behavior or the structure of the component.

The connections between components are specified within component instantiation statements. These statements specify an instance of a component occurring within another component or the circuit. Each component instantiation statement is labeled with an identifier. The Verilog language provides a large set of built-in logic gates which can be instantiated to build larger logic circuits. The set of logical functions described by the built-in gates include AND, OR, XOR, NAND, NOR and NOT.

### 3.3.2 VHDL

VHDL is an acronym which stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. The language has been known to be somewhat complicated. The acronym does have a purpose, though; it is supposed to capture the entire theme of the language that is to describe hardware much the same way we use schematics [21].

VHDL is being used for documentation, verification, and synthesis of large digital designs. This is actually one of the key features of VHDL, since the same VHDL code can theoretically achieve all three of these goals, thus saving a lot of effort. In addition to being used for each of these purposes, VHDL can be used to take three different approaches to describing hardware. These three different approaches are the structural, data flow, and behavioral methods of hardware description. Most of the time a mixture of the three methods is employed. The following sections introduce you to the language by examining its use for each of these three methodologies. There are also certain guidelines that form an approach to using VHDL for synthesis, which is not addressed by this tutorial [22].

VHDL was established as the IEEE 1076 standard in 1987. In 1993, the IEEE 1076 standard was updated and an additional standard, IEEE 1164 was adopted. In 1996, IEEE 1076.3 became the VHDL synthesis standard [23].

## Features of VHDL

VHDL is a worldwide standard for the description and modeling of digital hardware. VHDL gives the designer many different ways to describe hardware. The language offers: familiar programming tools for complex and simple problems, sequential and concurrent modes of execution to meet a large variety of design needs, packages and libraries to support design management and component reuse [20].

VHDL has ample features appropriate for describing the behavior of electronic components ranging from simple logic gates to complete Microprocessors, high performance Digital Signal Processors and custom chips. Features of VHDL allow timing aspects of circuit behavior (such as rise and fall times of signals, delays through gates, and functional operation) to be precisely described [21].

## Packages

Packages are intended to hold commonly-used declarations such as constants, type declarations and global subprograms. Packages can be included within the same source file as other design units (such as entities and architectures) or may be placed in a separate source file and compiled into a named library. This latter method is useful in using the contents of a package throughout a large design or in multiple projects. The IEEE 1164 standard provides a standard package named *std\_logic\_1164* that includes declarations for the type's *std\_logic*, *std\_ulogic*, *std\_logic\_vector* and *td\_ulogic\_vector*, as well as many useful functions related to those data types [22].

## Design Libraries

A design library is an implementation-dependent storage facility for previously analyzed design units. This resulted in many different implementations in synthesis and simulation tools. In general, however, design libraries are used to collect commonly-used design units (typically packages and package bodies) into uniquely-named areas that can be referenced from multiple source files in your design [23].

## Components

Components are used to connect multiple VHDL design units (entity/architecture pairs) together to form a larger, hierarchical design. Using hierarchy can dramatically simplify the design description and can make it much easier to re-use portions of the design in other projects. Components are also useful while making the use of third-party design units, such as simulation models for standard parts, or synthesizable core models obtained from a company specializing in such models [21].

## Configurations

Configurations are features of VHDL that allow large, complex design descriptions to be managed during simulation. (Configurations are not generally supported in synthesis.) One example of how to use configurations is to construct two versions of a system-level design, one of which makes use of high-level behavioral descriptions of the system components, while a second version substitutes in a post-synthesis timing model of one or more components.

For large projects involving many engineers and many design revisions, configurations can be used to manage versions and specify how a design is to be configured for system simulation, detailed timing simulation, and synthesis. Because simulation tools allow configurations to be modified and recompiled without the need to recompile other design units, it is easy to construct alternate configurations of a design very quickly without recompiling the entire design [22].

**DESIGN AND IMPLEMENTATION OF FFT PROCESSOR**

---

**4.1 Introduction**

The summary of different approaches to the FFT problems is discussed in chapter-2. The selection of an adequate algorithm for implementing the architecture of the FFT processor to be used for a specific application, is discussed in present chapter. Once the algorithm is selected for the desired application, the next step is its implementation.

In the present work the algorithm is selected taking into consideration the two parameters namely speed and complexity. The selected algorithm is suited for higher speed with lesser complexity. The algorithm is implemented using VHDL.

**4.2 Algorithm Choice**

While selecting the algorithm the basic point of consideration is that its architecture and corresponding design should be simple and easily understandable. In order to reduce the complexity of algorithm it is proposed to restrict it to 16-point. Therefore it is required to observe the existing algorithms, which are applicable for 16-point.

The radix-2 FFT algorithm has many good features. For example, it has low quantization noise level, and it is also easily parameterizable to the different FFT lengths. Radix-r does not seem to be as good a choice as the radix-2 one, because it has higher quantization noise, and that it is not as easily parameterizable to the different FFT lengths. To be able to parameterize this algorithm to the general power-of-2 FFT lengths, different radix-r stages have to be used in the pipeline, resulting in a mixed radix implementation.

Since minimizing the number of multipliers is important, a good choice of algorithm would be the split radix one. It has a lower number of multipliers than all the above ones, but this algorithm results in a complex design, which will be harder to parameterize. The control of this type of processor would also be more complex.

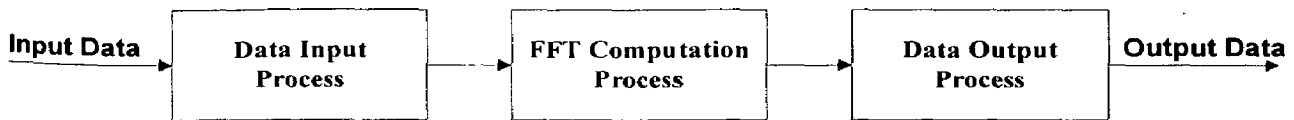


The radix-4 algorithm is the most attractive algorithm. It has low number of multipliers, simple control structure and architecture.

Prime factor algorithms can not be used, because the right FFT lengths can not be calculated using this algorithm. These are the reasons that the radix-4 FFT algorithm is used in the FFT implementation in present work.

### 4.3 FFT Processor

In this master thesis, it is to implement proposed a 16-point FFT processor. In the proposed architecture the complete operation of the FFT processor is divided into three sub processes. Namely: Data Input, FFT Computation and Data Output Processes. This is depicted in Figure 4.1[18].



**Fig 4.1 Three sub-processes of the FFT Algorithm**

#### 4.3.1 Data Input

The process cycle starts with The Data Input process. The input data assumed to be complex. So each input has real part and imaginary part. In this process, 16 complex inputs are taken at a time and stored in the blockRAM in FPGA (section 3.6.9). The real and imaginary part of the input data is stored separately in block RAM.

#### 4.3.2 FFT Computation

The FFT computation process is main block of the processor. It computes the FFT operation, which converts the time domain samples into frequency domain. It takes the input data which is stored in the blockRAM, applies the radix-4 algorithm on that data and the result is again stored into the same memory. The detailed description is given in following sections.

### 4.3.3 Data Output Process

The data output process is the last stage of the FFT processor. This process provides the results of the FFT Computation process to the outside world.

## 4.4 Architecture

The schematic diagram of the FFT processor architecture is shown in fig 4.2. It consists of a processing element, a blockRAM, a coefficient ROM and an address generation unit. In the designed FFT processor the processing element is a radix-4 butterfly (which is referred as the butterfly processing element).

In the designed FFT processor data pathways are in the form of 12-bit signed fractions. Coefficients are stored as 12-bit signed fixed-point words. Different elements of designed FFT processor are discussed in following sections.

### 4.4.1 Coefficient ROM

From equation 2.1,  $W_N$  called twiddle factor coefficient which is a complex number. Twiddle factor coefficients for 16-point FFT are generated using MATLAB. These coefficients are very small, hence these are scaled by a factor of 1024 and stored in ROM. The real and imaginary parts are separately stored in ROM. The result is then derived by finally scaling the stored output value with the  $1/1024$ .

### 4.4.2 Block RAM

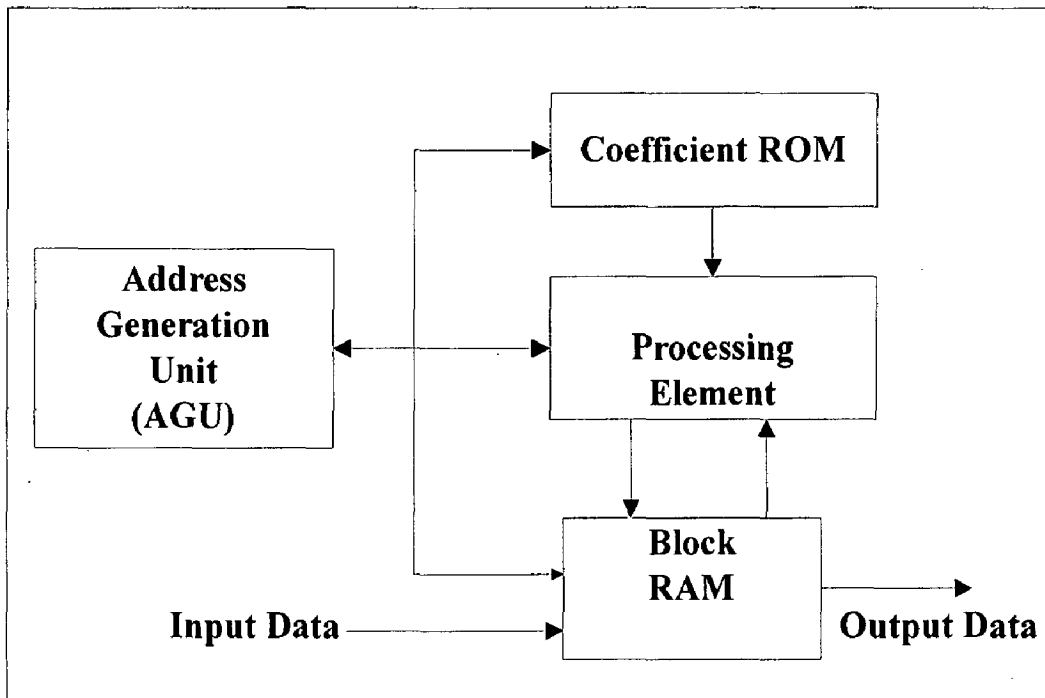
Block RAM is on-chip memory of FPGA. The size of memory depends on version of FPGA. Block RAM is used to store the input data, which is a complex data; hence the real and imaginary parts are stored separately in the memory. The same block RAM is used to store the intermediate results (i.e the output coming from the radix-4 butterfly) by replacing existing data because of the initial data is never used for further computation. The final output is again stored in same Block RAM.

### 4.4.3 Radix-4 Butterfly Implementation Details

The butterfly is the basic operator of the FFT. It takes four data words and computes 4-point DFT. The basic butterfly unit is shown in figure 2.8. The boxes in Figure 2.8 represent multiplications and the circles represent sums. The  $T_{wn}$  values ( $T_{w0}$ ,  $T_{w1}$ ,  $T_{w2}$ , and  $T_{w3}$ ) are commonly referred to as twiddle factors. These

values are determined by the number of samples in the input. J. G. Proakis and D. G. Manolakis have discussed the FFT algorithms and twiddle factors in detail[44]. Figure 4.3 shows the Radix 4 Butterfly in a simplified component form.

Since higher order FFTs use many butterfly stages, each having a different set of twiddle factors, the twiddle factors are not fixed, but are received as inputs. The inputs and outputs are assumed to be complex so each input and output has a real port and an imaginary port. In this implementation, the data is in signed (2's complement) form.

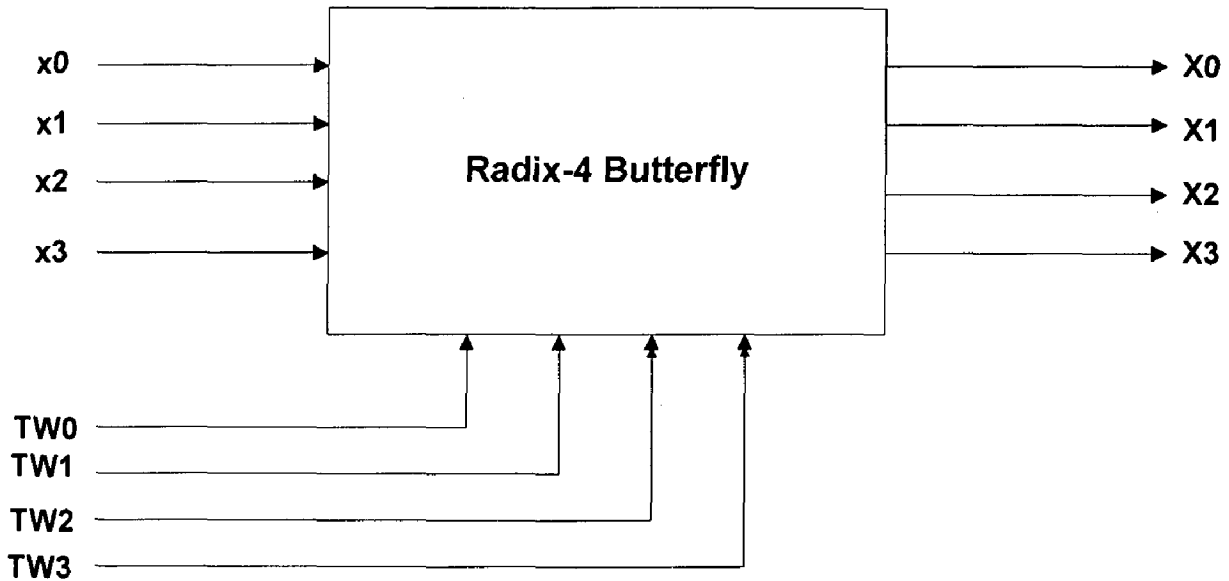


**Fig 4.2 Block Diagram Representation of the FFT processor**

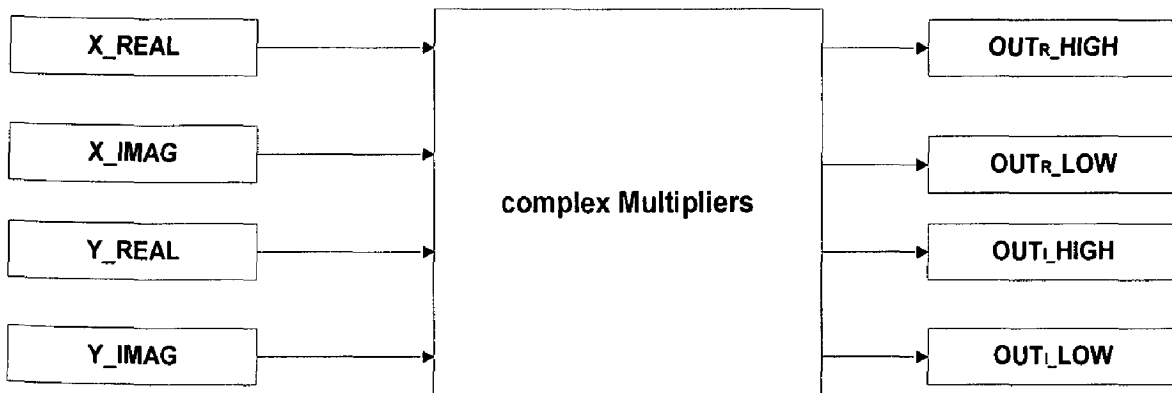
### Complex Multiplier

Each multiplication in the radix-4 butterfly is a complex multiplication. An asynchronous complex multiplier component is designed to perform the complex multiplication. It is shown in Figure 4.4. The schematic diagram of complex multiplier is shown in figure 4.4.

The complex multiplier uses four  $N$  bit real multipliers (signed). The multiplier has four inputs consisting of the real part and imaginary part of two complex numbers. The input width,  $N$ , is variable. The output is a complex number with  $2*N$  bits for the real part and  $2*N$  bits for the imaginary part.



**Fig 4.3 Block Diagram of Radix-4 Component**



**Fig 4.4 Asynchronous, Complex Multiplier**

Given two complex numbers  $x$  and  $y$  where  $x = x_R + jx_I$  and  $y = y_R + jy_I$ ,

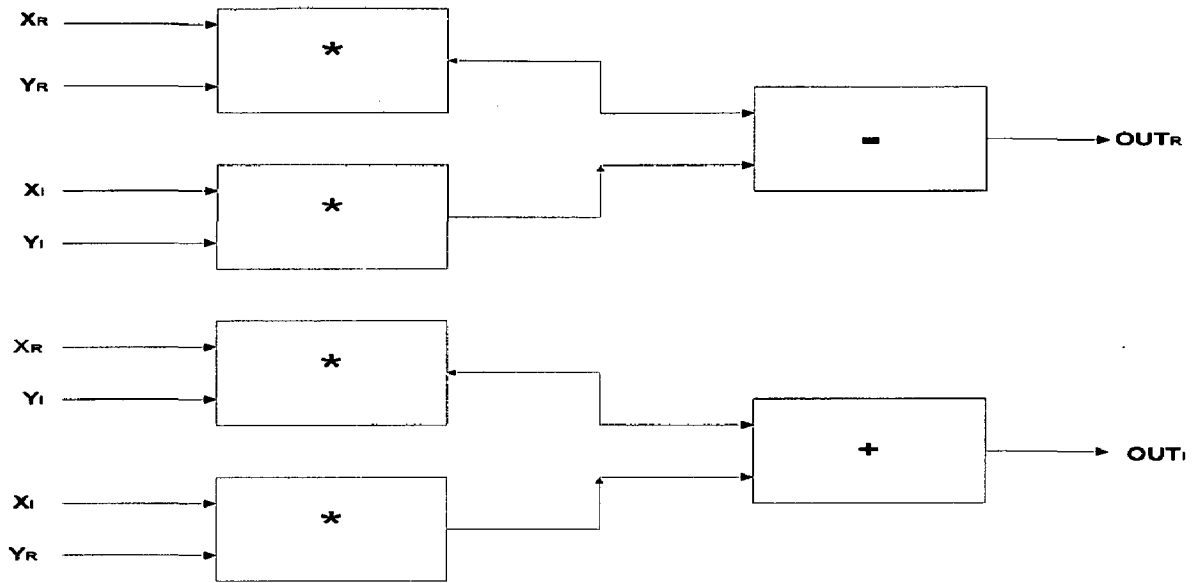
$$\text{Then } x * y = ((x_R * y_R) - (x_I * y_I)) + j((x_R * y_I) + (x_I * y_R)) \quad (4.1)$$

and

$$Out_R = ((x_R * y_R) - (x_I * y_I)) \quad (4.2)$$

$$Out_I = ((x_R * y_I) + (x_I * y_R)). \quad (4.3)$$

Figure 4.5 further elaborates the complex multiplier.



**Fig 4.5 Complex Multiplier block diagram**

### Radix-4 Butterfly Component

For the Design of N-point FFT processor, the number of stages required and Number of butterflies for each stage are derived using following equations.

$$\text{Number of stages} = \log_4 N \quad (4.4)$$

$$\text{Number of butterflies for each stage} = N/4 \quad (4.5)$$

Therefore 2 stages are required for computing 16-pt FFT. In each stage radix-4 algorithm is required to be invoked four times. A 16-point radix-4 decimation-in-frequencies FFT algorithm is shown in figure 2.9.

By observing figure 2.9 It contain four different set of twiddle factors in the first stage. Hence It has 16 input ports and eight output ports. Eight inputs are for the real and imaginary parts of four data samples. Another eight inputs are for the real and imaginary parts of the four twiddle factors. The eight outputs are for the real and imaginary parts of the four results. The input width is variable. To avoid errors, all data and twiddle inputs must be the same width. The output width is twice the input width.

The radix-4 butterfly requires four complex multiplications for a total of 16 real multipliers. If the inputs are 12 bits each, then one real multiplier requires 544 CLBs. A full implementation of a radix-4 butterfly would require 8,704 CLBs. For this reason a single complex multiplier is used and it is time multiplexed over four

clock cycles to achieve the four complex multiplications. Thus a single butterfly operation takes four clock cycles. We can get one output for every clock cycle.

It is possible for an overflow to occur when the results of the four multiplications are added together. To keep this implementation simple, no overflow checks are performed. The detailed architecture for butterfly processing element is depicted in Figure 4.6.

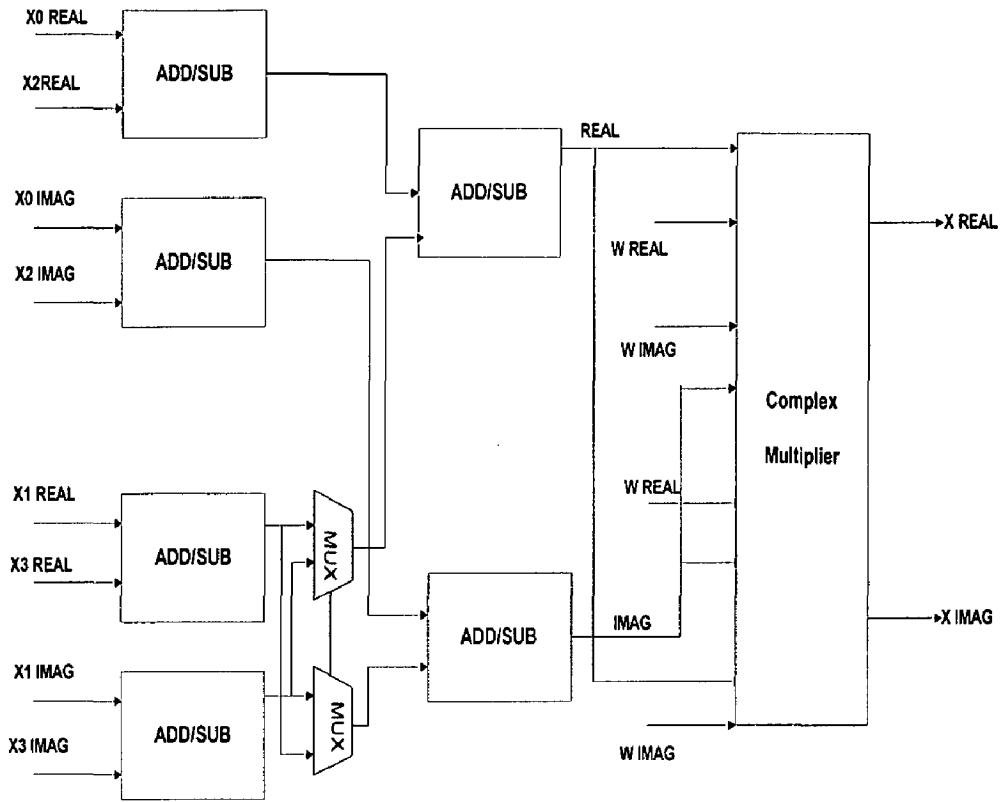
To simplify the architecture the number of multipliers are to be reduced. For this purpose, two different radix-4 butterfly architecture for the two stages are developed.

In 2<sup>nd</sup> stage, four butterflies are required with same set of twiddle factors. The set of twiddle factor are  $(W_{16}^0, W_{16}^0, W_{16}^0, W_{16}^0)$ . the real part of  $W_{16}^0$  is one and imaginary part is zero. Hence, the architecture of butterfly element in the 2<sup>nd</sup> stage can be developed without using complex multipliers. This architecture is very simple than that of 1<sup>st</sup> stage. It will improve the speed and reduce the complexity of the circuit. The detailed architecture of the 2<sup>nd</sup> stage butterfly processing element after simplification is depicted in Figure 4.7.

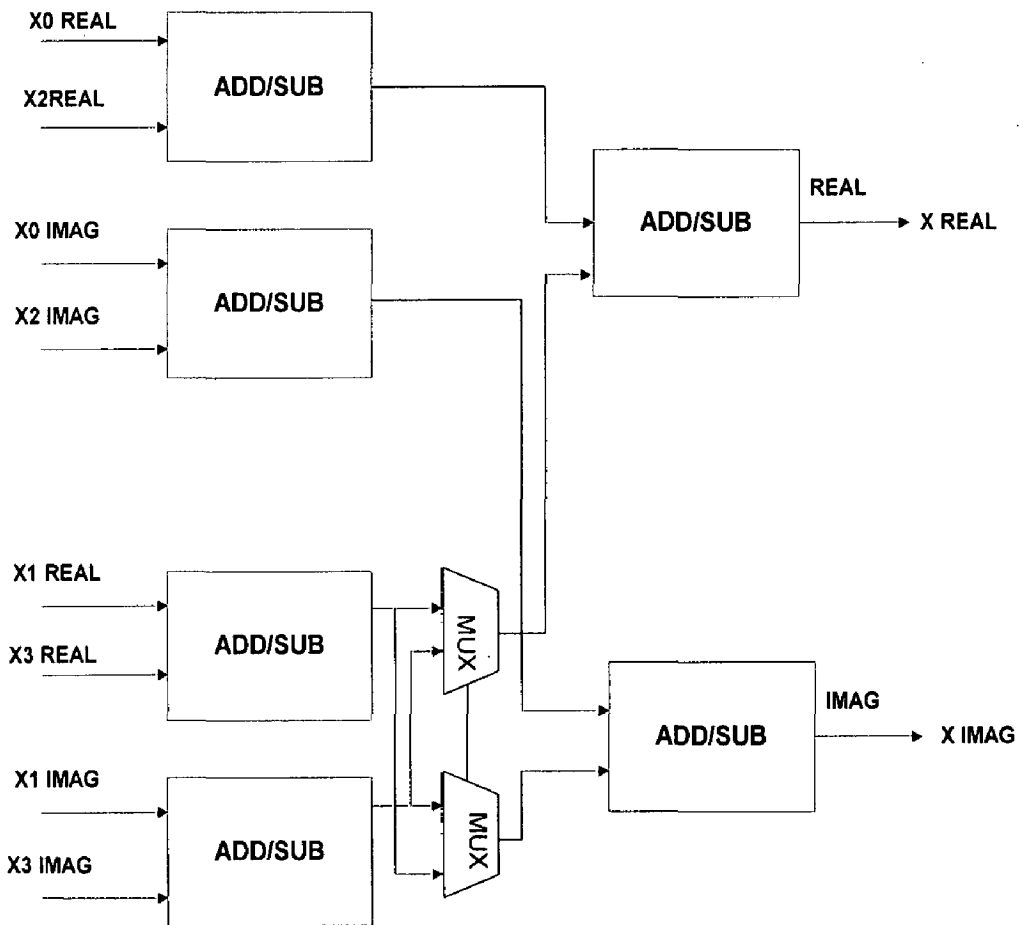
#### 4.4.4 Address Generation Unit (AGU)

The AGU plays main role in the design of the processor. It will synchronize the all input data which is required for processing. The address generation procedure is different for two stages of radix-4 FFT algorithm. It will generates address of four input at a time and also generates corresponding twiddle factors required for that input set. The 16-point of input data is stored in address from 0 to 15 in RAM. The points required to generate in the order of (0,4,8,12),(1,5,9,13),(2,6,10,14) and (3,6,11,15) . it will be observed from the fig 2.9. The above input order can obtain by using digit reverse technique, which is explained in above section. I developed a simple architecture for digit reverse, which will swapped the bits 1, 2 with 3, 4 after generating from counter. Doing this four bit patterns are transformed as below.

$$\begin{aligned}
 (0, 1, 2, 3) & \rightarrow (0, 4, 8, 12) \\
 (4, 5, 6, 7) & \rightarrow (1, 5, 9, 13) \\
 (8, 9, 10, 11) & \rightarrow (2, 6, 10, 14) \text{ and} \\
 (12, 13, 14, 15) & \rightarrow (3, 7, 11, 15).
 \end{aligned}$$



**Fig 4.6 Butterfly Processing Element Architecture (BPEA) for 1<sup>st</sup> stage**



**Fig 4.7 Butterfly Processing Element Architecture (BPEA) for 2<sup>nd</sup> stage**

For 2<sup>nd</sup> stage of FFT ,the points required for radix-4 algorithm is in sequential order like (0,1,2,3),(4,5,6,7),(8,9,10,11) and (12,13,14,15).Hence simple counter logic will generates the points in sequential order. The flow chart of above discussion is shown in fig4.8

#### 4.5 FFT Processor Architecture

The FFT architecture using radix-4 butterfly unit designed in previous section is developed as shown in fig 4.9. It is observed from above discussion is there are two stages are required for 16-point FFT. In each stage radix-4 algorithm is required to be invoked four times.

The addressing scheme is required for generating addresses which derived from Finite State Machine. The Twiddle factors required for performing the operation are stored in the Read Only Memory. The Real part and Imaginary parts are separately stored in two different ROMs. Coefficients are to be first scaled with the 1024 and stored in the ROM to avoid overflow. The result is then derived by finally scaling the stored value with the 1/1024.

Initially, 16 points are stored in the RAM. The data is a complex data, hence the real and corresponding imaginary part of data has to be taken at a time. For each radix-4 butterfly operation, address of four input points and corresponding twiddle factors are generated by using FSM. Initially, the radix-4 algorithm is applied for the four points of 0,4 8,12 and the result is stored in RAM . The operation is repeated for three different points (1,5,9,13), (2,6,10,14) and (3,7,11,15). With this operation, first stage has been completed. In the second stage it requires four points 0,1,2,3 and proceeds to consider the points in order (4,5,6,7), (8,9,10,11) and (12,13,14,15).

Here again the appropriate addresses are generated by the finite state machine (AGU). The final values resulting from the second stage of operation are stored in the RAM, as in the first stage, the real and imaginary values are stored separately.

**This architecture is developed by using VHDL. The complete source code and relevant documentation are provided in CD (attached at the end of report).**

The architecture explained in this chapter is implemented using FPGA. The processor for implementation is described in next chapter.



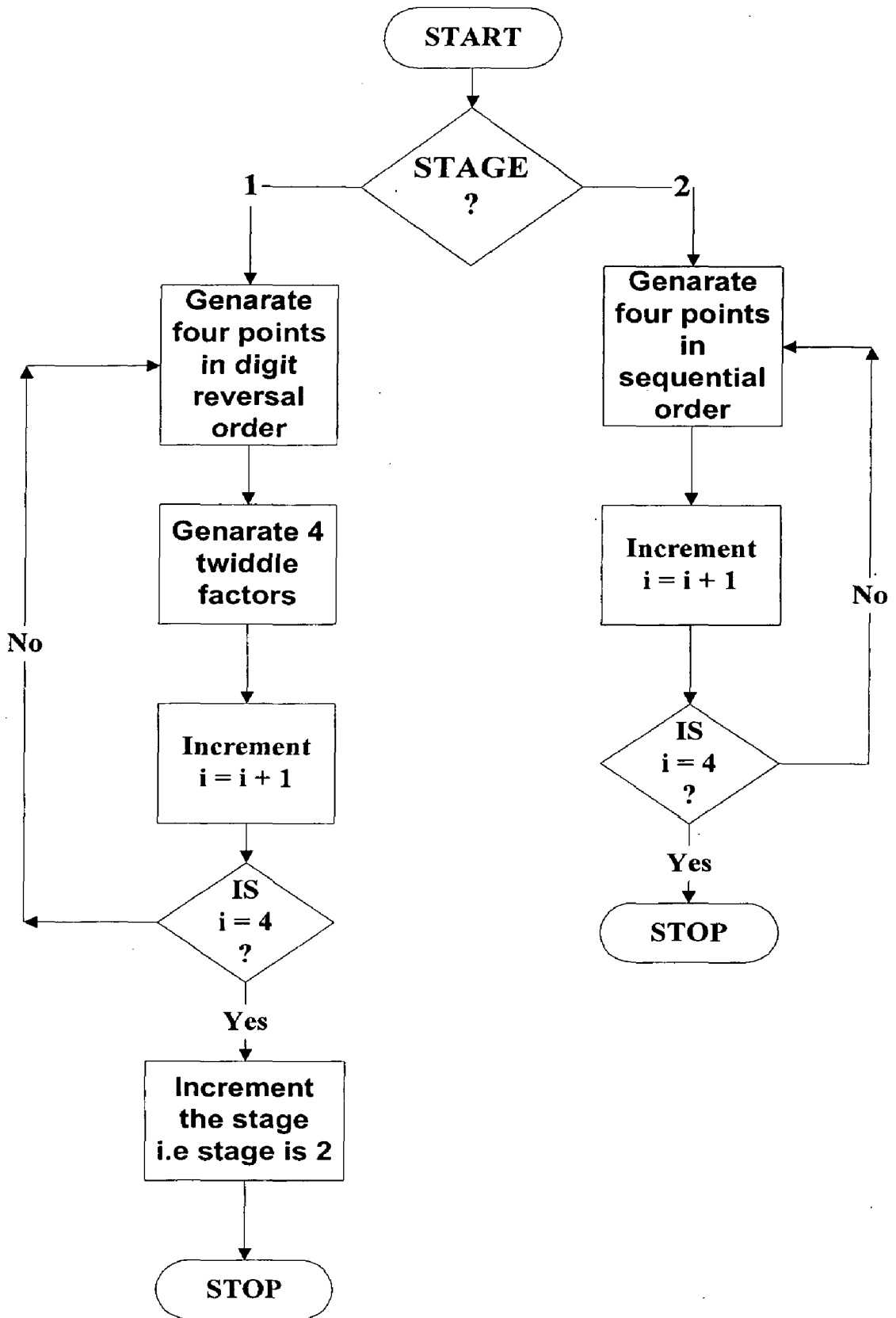
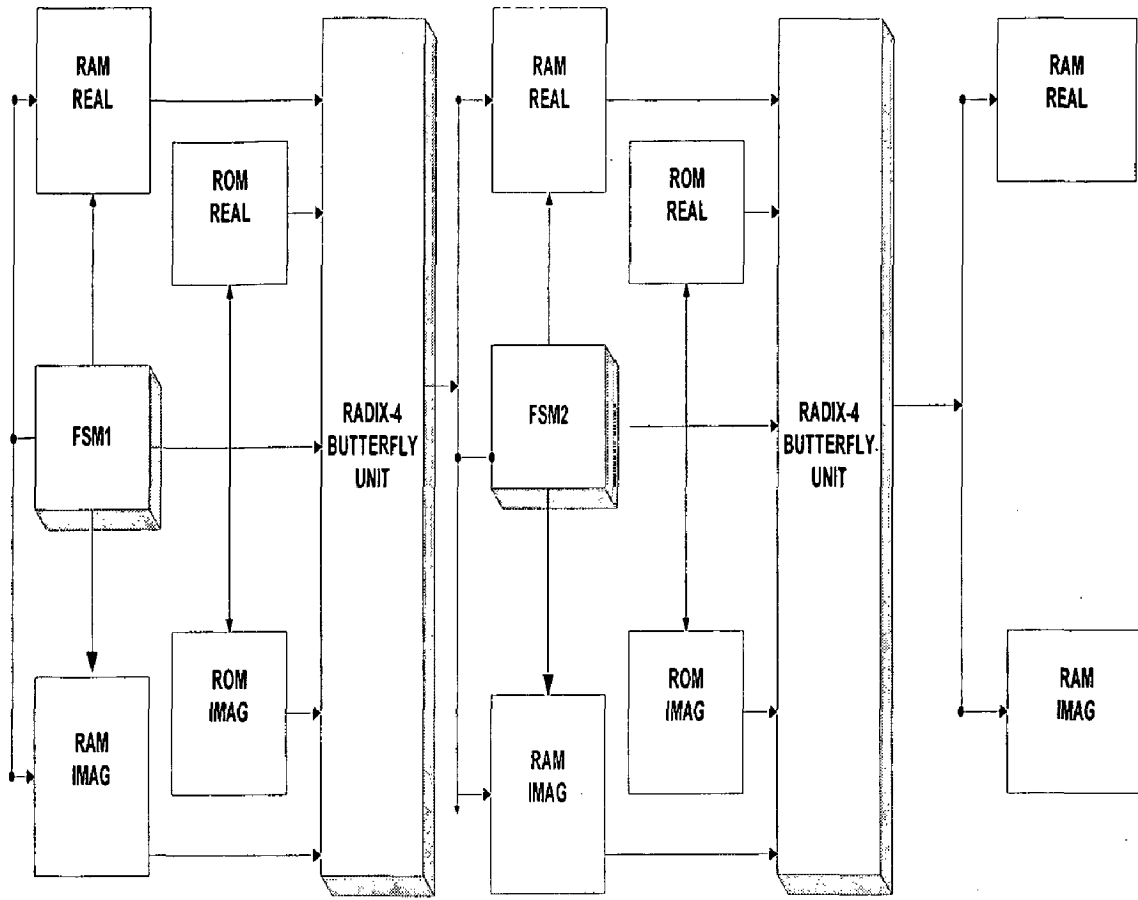


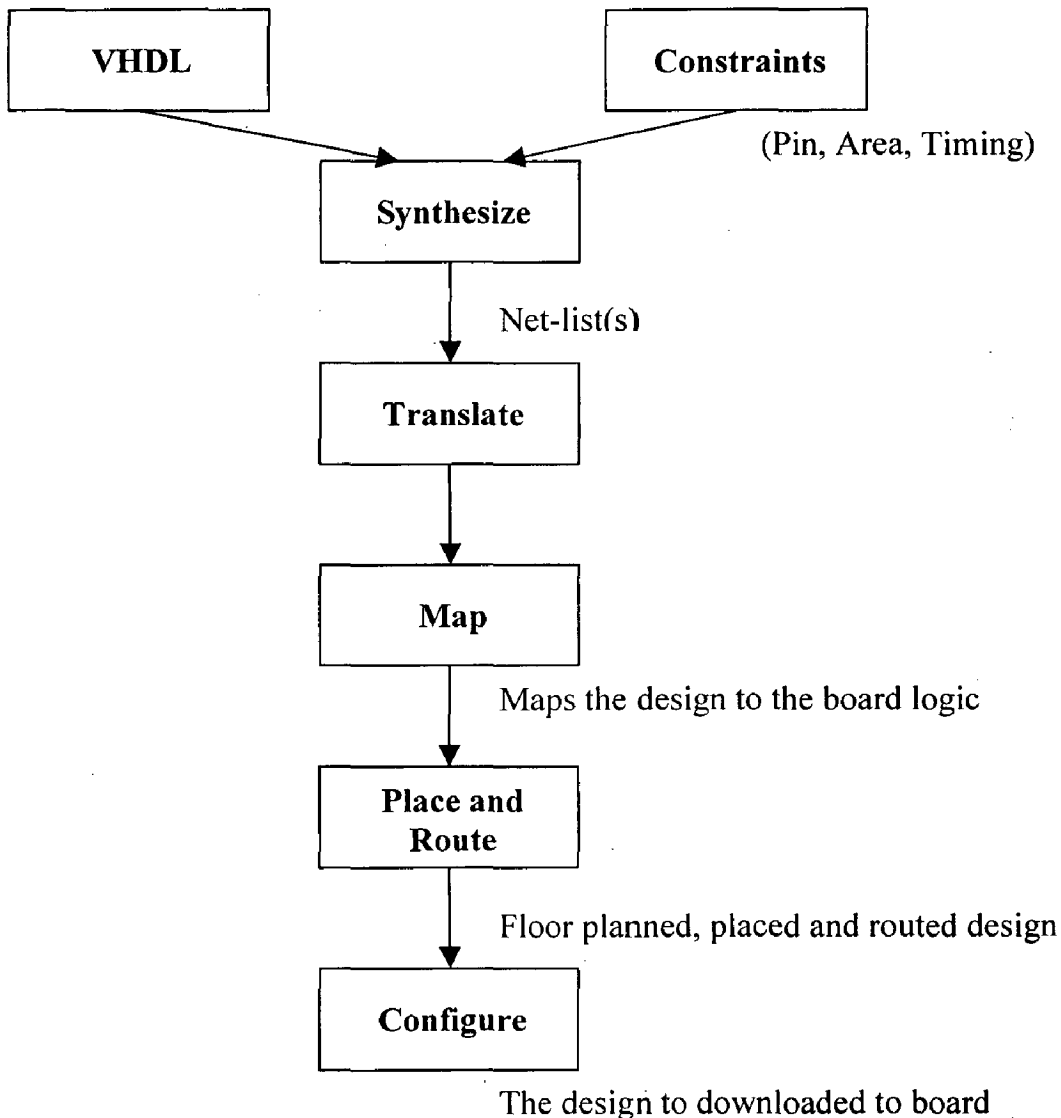
Fig 4.8 Flow chart for address generation unit



**Fig 4.9 16-point FFT architecture**

**DESIGN FLOW AND FINAL IMPLEMENTATION ON FPGA****5.1 Introduction**

This chapter describes the design flow used to create complex FPGA and ASIC devices. The designer starts with a design specification, creates an RTL description, verifies that description, synthesizes the description to gates, uses place and route tools to implement the design in the chip, and then verifies that the final result is correct in terms of function and timing. The design flow is shown in figure 5.1.



**Fig 5.1 The High-Level Design Flow**

## 5.2 Specification

All designs should start with a detailed specification of the exact tasks the application should do and include details on how fast the tasks must be completed.

## 5.3 Design Entry

In general design entry would be done through any hardware description language (HDL) such as VHDL or Verilog. In this thesis, VHDL is used for design entry. One of the best uses of VHDL today is to synthesize ASIC and FPGA devices.

## 5.4 Simulation

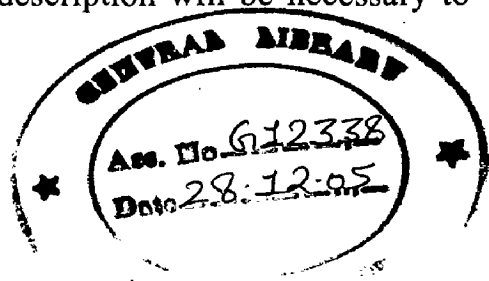
Simulation is the representation of the structure and behavior of a digital logic system through the use of a computer. A simulator interprets the HDL description and produces readable output, such as a timing diagram, that predicts how the hardware will behave before it is actually fabricated. Simulation allows the detection of functional errors in a design without having to physically create the circuit. The stimulus that tests the functionality of the design is called a test bench. Thus, to simulate a digital system, the design is first described in HDL and then verified by simulating the design and checking it with a test bench, which is also written in HDL.

## 5.5 User Constraint File

The UCF file maps signals in VHDL code to pins on the FPGA board. The signal name in your .vhd file must match the net name in the UCF file. If the names do not match, change the name in your .vhd file, not the net name in the .UCF file. This UCF file and .vhd files are the input to the Synthesis process.

## 5.6 Synthesis

After the hardware has been written, simulated and debugged, it needs to be synthesized. In some cases, rewriting the hardware description will be necessary to



make the hardware partitions synthesizable. If any code is rewritten, the hardware must be simulated again to make sure it still meets the requirements of the specifications

Synthesis is an automatic method of converting a higher level of abstraction to a lower level of abstraction. There are several synthesis tools available currently. In this thesis, ISE tool which is provided by Xilinx was used for synthesis.

The current synthesis tool converts the Register Transfer Level (RTL) descriptions to gate level netlists. These gate level netlists consists of interconnected gate level macro cells. Models for the gate level cells are contained in technology libraries for each type of technology supported. The netlists, which are generated from synthesis tool are device independent, so its contents do not depend on the particulars of the FPGA. It is usually stored in a standard format called the Electronic Design Interchange Format (EDIF)[24].

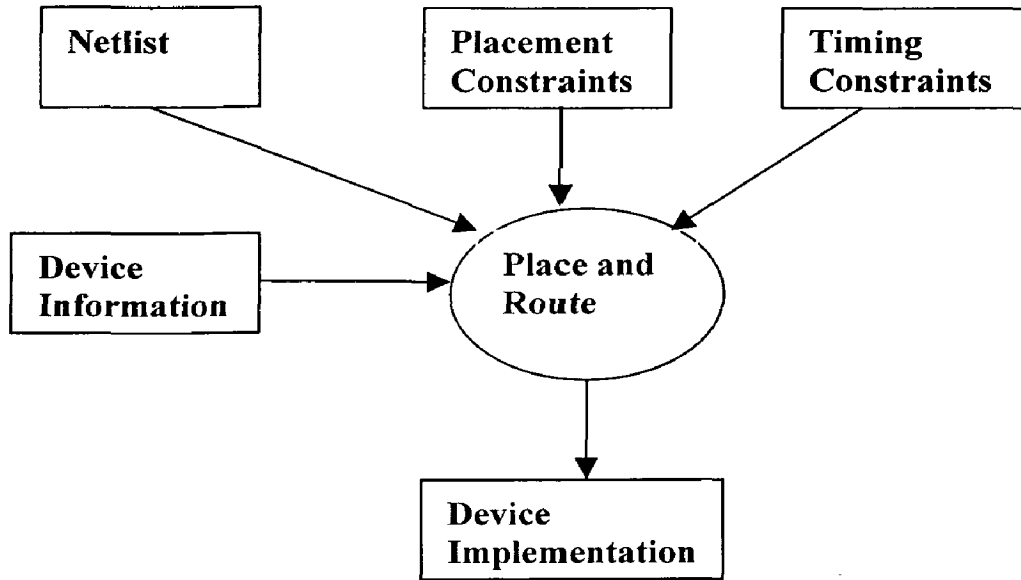
## 5.7 Implementation

In the Design Implementation stage, the netlist produced by the design entry program is converted into the bitstream file which configures the FPGA. The first step Maps the design onto the FPGA resources; the second step Places or assigns logic blocks created in the mapping process in specific locations in the FPGA. The third step Routes the interconnect paths between the logic blocks. The output is a Logic Cell Array File (LCA) for the particular FPGA; this process is explained in detail in section 5.7. This LCA file is then converted into a bitstream file for configuring the FPGA[24].

## 5.8 Place and Route

Place and route tools are used to take the design netlist and implement the design in the target technology device. The place and route tools place each primitive from the netlist into an appropriate location on the target device and then route signals between the primitives to connect the device according to the netlist. Place and route tools are typically very architecture and device dependent. These tools are tuned to take advantage of each architectural and routing advantage the device contains. FPGA vendors provide these tools because the differences in architectures are large enough

that writing a common tool for all architectures would be very difficult. fig 5.2 shows a dataflow diagram of the place and route tools[24].



**Fig 5.2 Place and Route Data Flow**

Input to the place and route tools are the netlist in EDIF or another netlist format, and possibly timing constraints. The format of the netlist input file varies from manufacture to manufacturer. Some tools use EDIF[24].

Another input to some place and route tools is the timing constraints, which give the place and route tools an indication about which signals have critical timing associated with them and to route these nets in the most timing efficient manner. These nets are typically identified during the static timing analysis process during synthesis. These constraints tell the place and route tool to place the primitives in close proximity to one another and to use the fastest routing. The closer the cells are, the shorter the routed signals will be and the shorter the time delay[24].

Some place and route tools allow the designer to specify the placement of large parts of the design. This process is also known as floorplanning. Floorplanning allow the user to pick locations on the chip for large blocks of the design so that routing wires are as short as possible. The designer lays out blocks on the chip as general areas. The floorplanner feeds this information to the place and route tools so that these blocks are placed properly. After the cells are placed, the router makes the appropriate connections[24].

After all the cells are placed and routed, the output of the place and route tools consists of data files that can be used to implement the chip. In the case of FPGAs, these files describe all of the connections needed to make the FPGA macro cells implement the functionality required. Antifuse FPGAs use this information to burn the appropriate fuses, while reprogrammable devices download this information to the device to turn on the appropriate transistor connections.

The other output from the place and route software is a file used to generate the timing file. This file describes the actual timing of the programmed FPGA device or the final ASIC device. This timing file, as much as possible, describes the timing extracted from the device when it is plugged into the system for testing. The most common format of this file for most simulators is SDF (Standard Delay Format). Sometimes, proprietary formats are generated and later translated to SDF. SDF is used to back-annotate the post route timing information from place and route tools into the post layout timing simulation[24].

## 5.9 FPGA Configuration

Configuration is a process in which the circuit design (bitstream file) is downloaded into the FPGA. The method of configuring the FPGA determines the type of bitstream file. FPGAs can be configured by a PROM. The serial PROM is the most common. The FPGA can either actively read its configuration data out of external serial or byte parallel PROM (master mode), or the configuration data can be written into the FPGA (slave and peripheral mode).

## RESULTS AND DISCUSSIONS

---

### 6.1 Simulation Results

To make FPGA to work to the needs of the user, design needs to be simulated and different signals timing of execution needs to be checked. In the present design, 16-point FFT processor is simulated and results had been shown below.

### 6.2 Matlab Results

Matlab is used to verify the results which are obtain from the simulation. Matlab tool will gives the perfect solution for any application. The matlab results are shown in fig6.4.

#### **Discussion:**

The input to the FFT processor is complex. Hence the real and imaginary parts are indicated separately. The signals X0\_real\_s and X0\_imag\_s are the real and imaginary parts of complex data X0.it is shown in fig 6.1.

Each radix-4 butterfly will take four complex data and the radix-4 butterfly is invoked four times in each stage of FFT. Input to the first stage of radix-4 butterflies shown in fig6.2.

The signals address\_s\_1 and address\_s\_2 shows the address of input data which is generated by address generation unit. The signal address\_s\_1 indicates address of the input for 1<sup>st</sup> stage of radix-4 butterflies and address\_s\_2 denotes the address of the input for 2<sup>nd</sup> stage of radix-4 butterflies.

The final output of simulation is represented in fig 6.3 and these results are compared with the matlab results shown in fig 6.4.



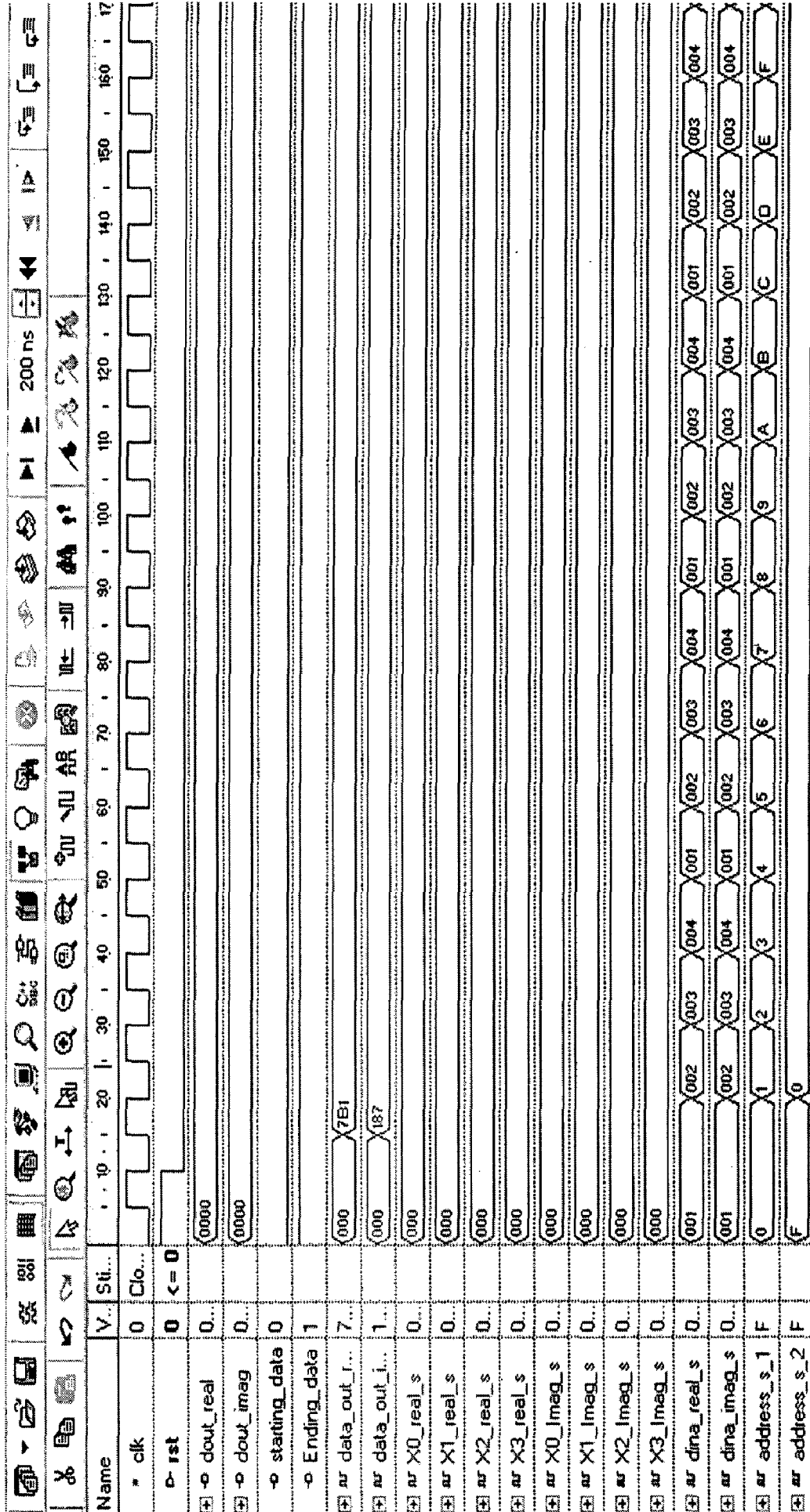


Fig 6.1 The input is stored in RAM. The scale above the waveforms show the system clock cycles.

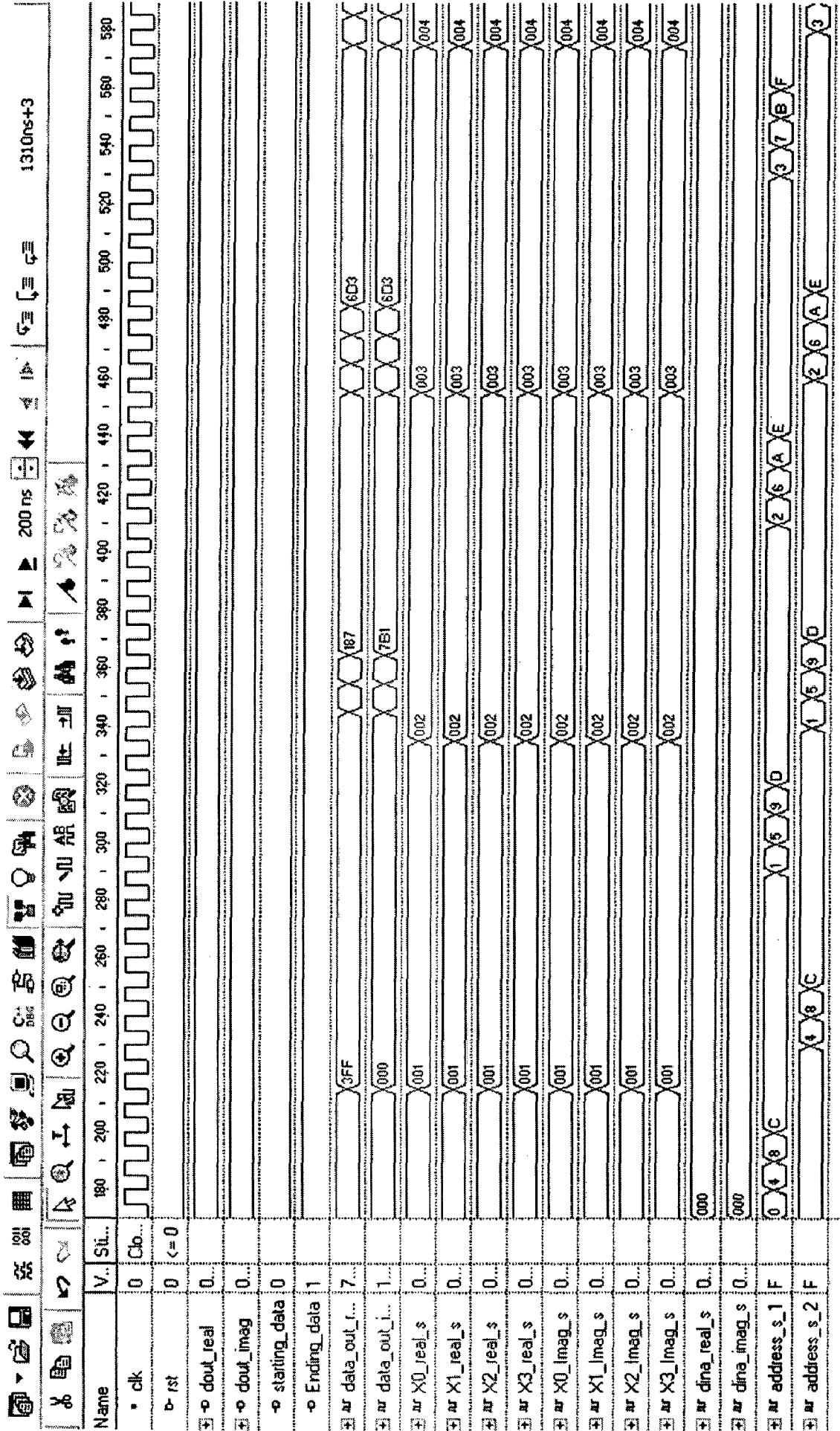


Fig 6.2 The inputs which are applied to 16-point FFT processor.

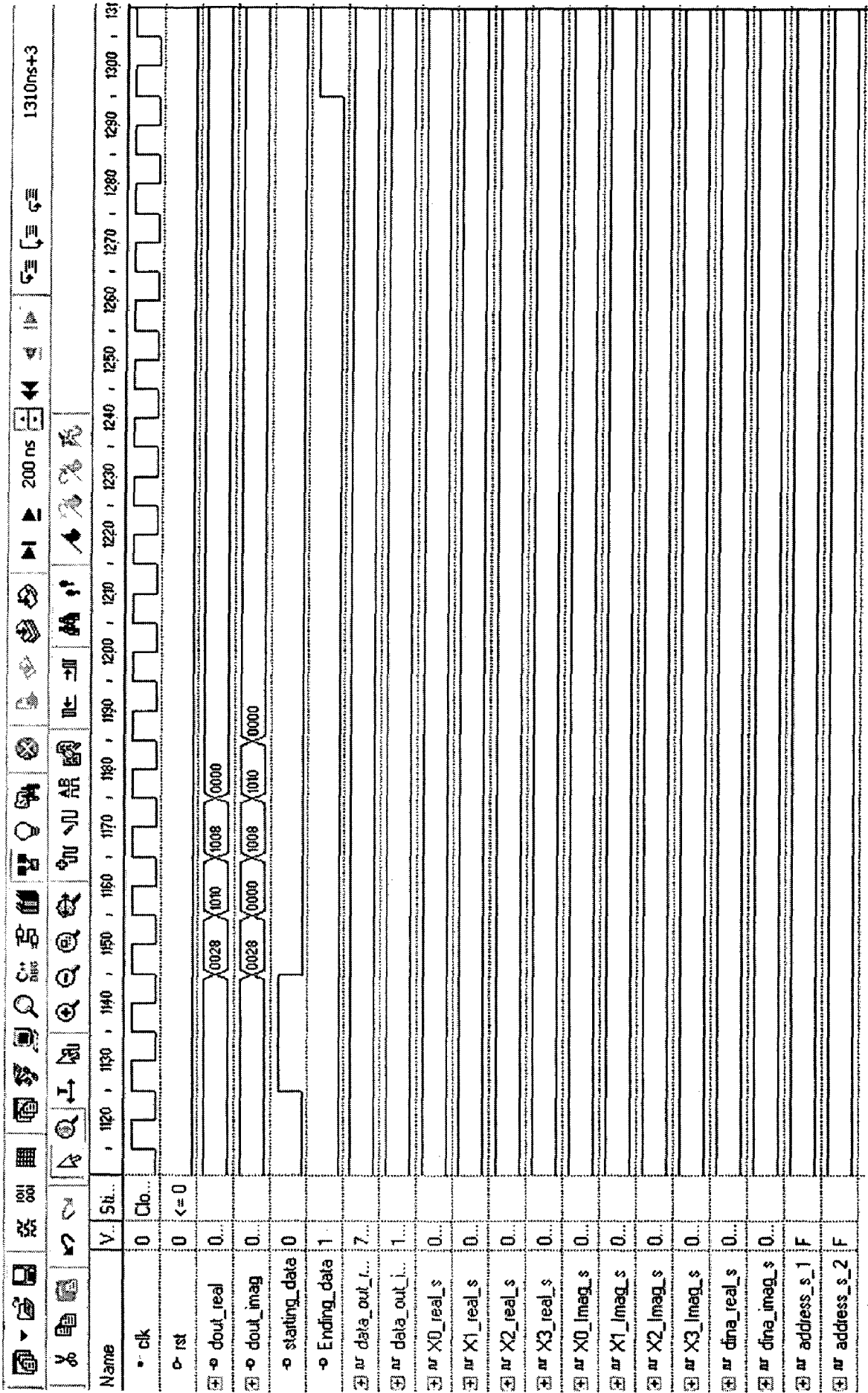


Fig 6.3 Final output of the 16-point FFT processor



## 6.3 Synthesis Results

Synthesis results give the mapping report and how the proposed architecture had been placed on the FPGA, and how the CLBs are connected in FPGA and what are the pins of FPGA are connected as user IOs It also gives the RTL schematic view.

### Mapping Report for radix-4 butterfly

This report is simply the outcome of how exactly designed code is represented and how much of the resources it utilized. Following tables 6.1 and 6.3 gives the device utilization for 1<sup>st</sup> stage of radix-4 FFT and 2<sup>nd</sup> stage of radix-4 FFT algorithm.

**Selected Device: 2s200pq208-5**

**Table 6.1: Hardware utilization for 1<sup>st</sup> stage of radix-4 FFT algorithm.**

	Utilized Number	Total Number	%Of Utilization
<b>SLICES</b>	466	2352	19
<b>FLIP FLOPS</b>	9	4704	0
<b>4-input LUTs</b>	856	4704	18
<b>Bonded IOBs</b>	118	144	81
<b>GCLKs</b>	1	4	25

**Table 6.2 Hardware utilization for 2<sup>nd</sup> stage of radix-4 FFT algorithm.**

	Utilized Number	Total Number	%Of Utilization
<b>SLICES</b>	75	2352	3
<b>FLIP FLOPS</b>	9	4704	0
<b>4-input LUTs</b>	136	4704	2
<b>Bonded IOBs</b>	116	144	80
<b>GCLKs</b>	1	4	25

## Discussion

From the above results it is observed that, the device utilization for 2<sup>nd</sup> stage of radix-4 FFT is less than that of 1<sup>st</sup> stage of radix-4 FFT because of no multipliers are utilized in 2<sup>nd</sup> stage.

## Mapping Report for the 16-point FFT processor

The table 6.3 gives the amount of device utilized for the implementation of 16-point FFT processor

**Table 6.3 Hardware utilization for 16-point FFT processor**

	<b>Utilized Number</b>	<b>Total Number</b>	<b>%Of Utilization</b>
<b>SLICES</b>	<b>1271</b>	<b>2352</b>	<b>54</b>
<b>FLIP FLOPS</b>	<b>487</b>	<b>4704</b>	<b>10</b>
<b>4-input LUTs</b>	<b>2006</b>	<b>4704</b>	<b>42</b>
<b>Bonded IOBs</b>	<b>30</b>	<b>144</b>	<b>20</b>
<b>BRAMs</b>	<b>4</b>	<b>14</b>	<b>28</b>
<b>GCLKs</b>	<b>1</b>	<b>4</b>	<b>25</b>

Top level schematic drawing is shown in figure 6.5

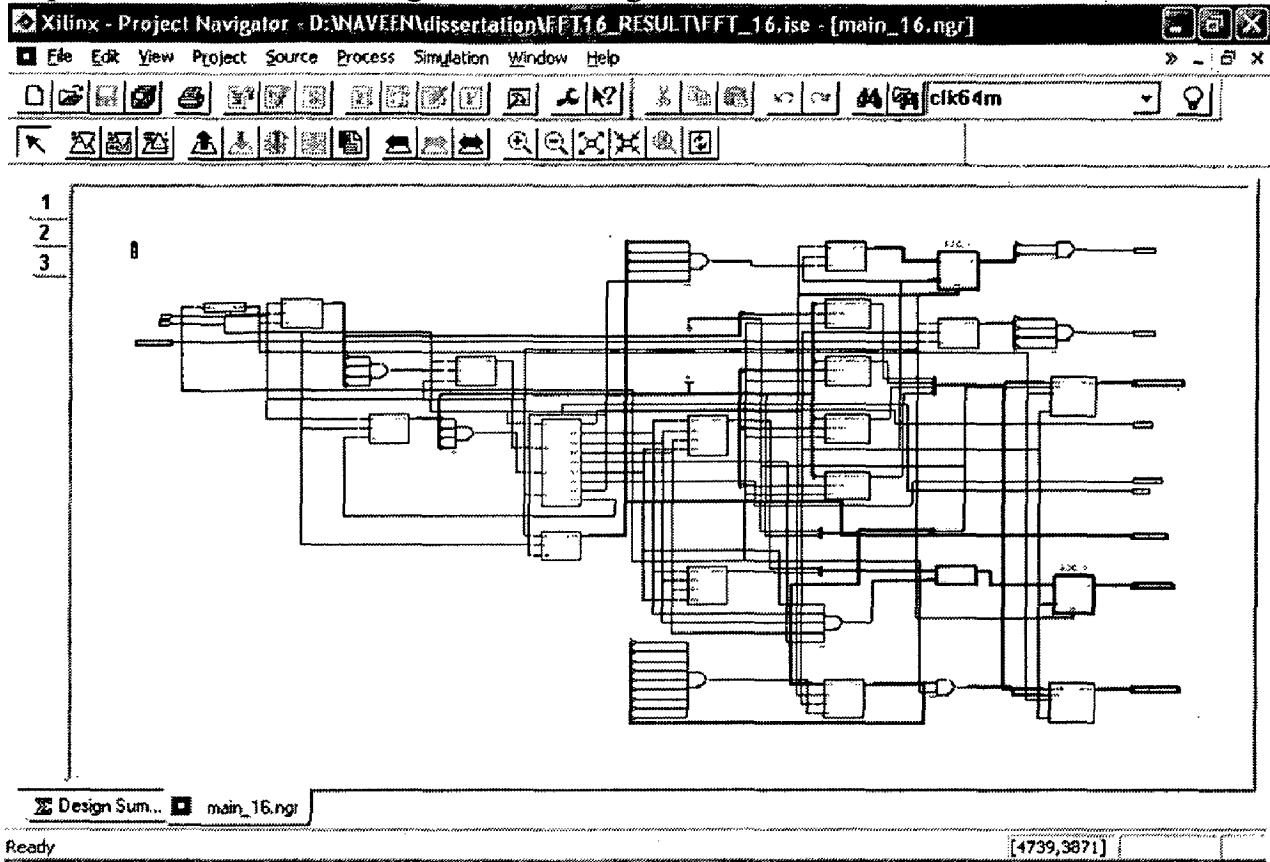
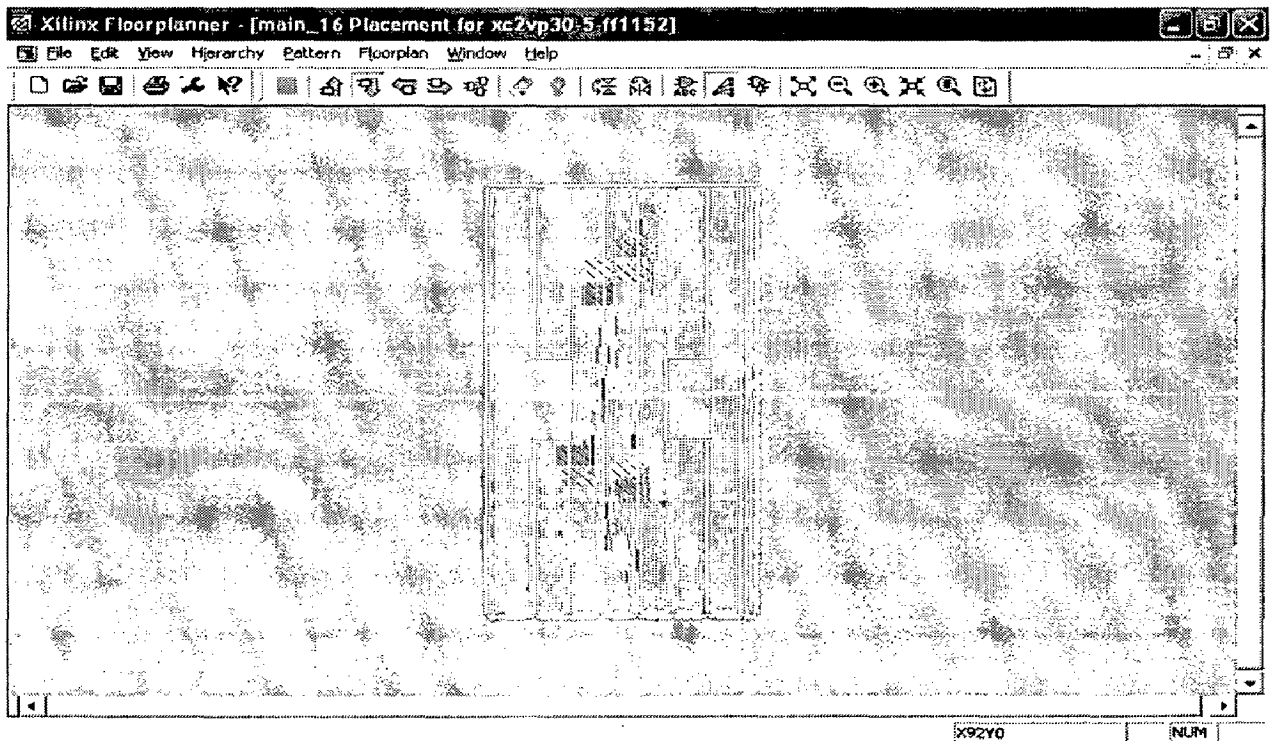
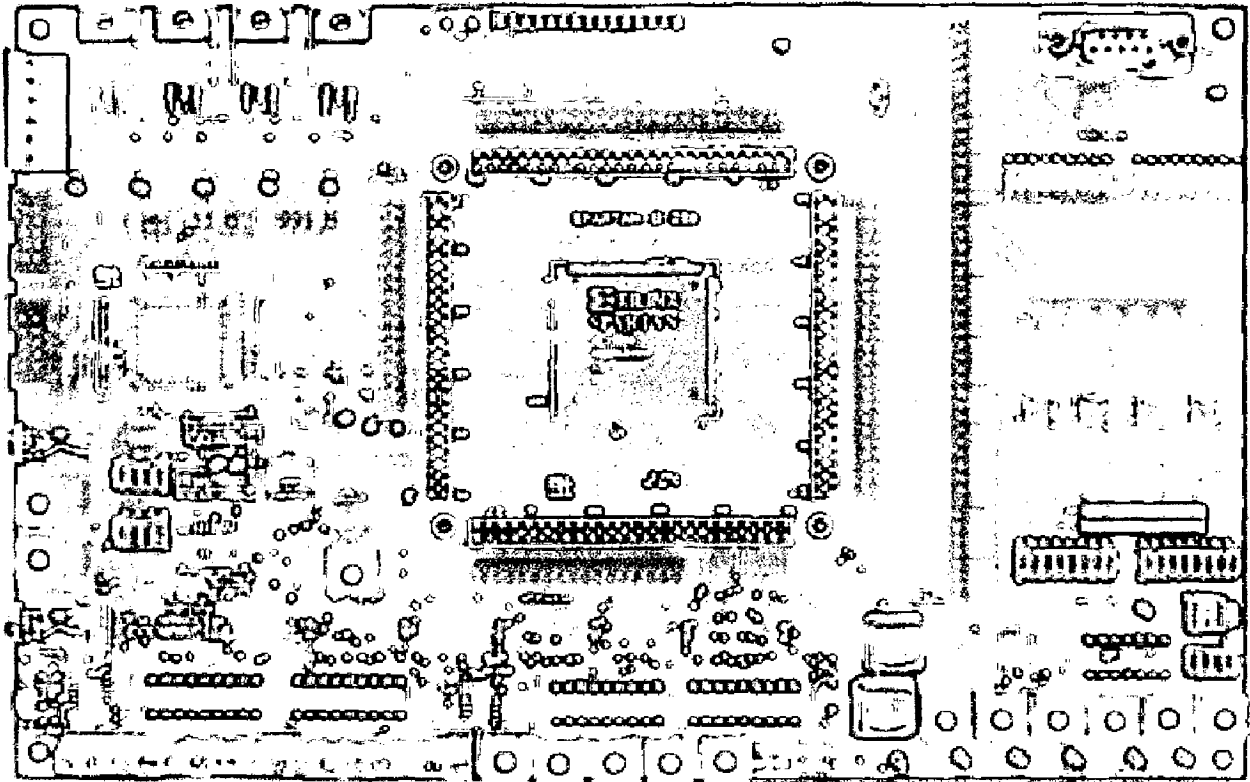


Fig .6.5 RTL view of 16-point FFT processor.

**Floor Plan:** How exactly gates were places on the floor of the chip?





**Xilinx SPARTAN-II FPGA**



**CONCLUSION AND FUTURE SCOPE OF WORK**

---

**7.1 Conclusions**

FFT processor architecture optimized for speed and area has been designed. The algorithm used was a modified version of the DIF-FFT radix-4 with the inputs in natural order and the outputs in bit reversed order. The architecture consisted of a radix-4 butterfly, BlockRAM, coefficient ROM, and address generation unit. Separate memories are used for storing the data and the coefficients. Although the processor designed is quite small and fast there are some improvements that can be made. Most of the cells used to build the FFT processor have been optimized for speed rather than area and power consumption. These blocks can be redesigned for reduced area and power consumption. The FFT processor is capable of computing 16 point complex FFT in 1300ns including data input and output processes. The chip is operating with a clock frequency of 100MHz.

**7.2 Future Scope of Work**

- The present work is designed for 16-points of input data. The same architecture can be applied for 32, 256, 1024 etc., points of input data.
- The main objective of work is concentrated on performance of the processor so while designing the processor the power consumption is not considered as a constraint. The same circuit can be better implemented keeping in view of power consumption using CMOS technique.
- The process is designed for computing the stored input data. The architecture can be extended to facilitate real time data by using pipeline architecture.
- The control circuit which takes care of overflow of data is being excluded in the designed architecture to reduce the complexity as the performance of the processor is main objective. The control circuit can be included in architecture to control overflow of data. .

## REFERENCES

---

1. J. G. Proakis and D. G. Manolakis, "Digital Signal Processing: Principles, Algorithms and Applications", 3<sup>rd</sup> ed. Englewood Cliffs, NJ:Prentice-Hall, 2002.
2. A. V. Oppenheim and R. W. Schaffer, "Discrete-Time Signal Processing". Englewood Cliffs, NJ: Prentice-Hall, 1989.
3. J. S. Walker, "Fast Fourier Transforms", 2nd ed. Boca Raton, FL: CRC,1996.
4. J.W. Cooley and J. W. Tukey, "An Algorithm for the Machine Computation of the Complex Fourier Series," Math.of Computation, Vol. 19, April 1965, pp. 297-301.
5. J. A. C. Bingham, "Multi-carrier modulation for data transmission: an idea whose time has come," IEEE Communications Magazines, Vol. 28, No. 5, pp. 5-14, May. 1990.
6. E. O. Brigham, "The Fast Fourier Transform", Prentice-Hall, 1974.
7. S. Winograd, "On computing the discrete Fourier transform," Proc. Nat.Acad. Sci., U S., Vol. 73, pp. 1005-1006, Apr. 1976.
8. L. R. Rabiner and B. Gold, "Theory and Application of Digital Signal Processing", Prentice-Hall, 1975.
9. H. Lim and E. E. Swartzlander, "Multidimensional systolic arrays for the implementation of discrete Fourier transforms," IEEE Trans. On Signal Processing, Vol. 47, pp. 1359-1370, May 1999.
10. P. Duhamel and H. Hallmann, "Split radix FFT algorithm," IEEElectronics Letters, Vol. 20, No. 1, pp. 14-16, Jan. 1984.
11. S. Bertazzoni, G. C. Cardarilli, M. Iannuceelli, M. Salmeri, A. Salsano, and O. Simonelli, "16-point high speed (i) FFT for OFDM modulation," ISCAS'98, Vol. 5, pp. 210-212, 1998
12. Shousheng He and Mats Torkelson, "A new approach to pipeline FFT processor," Parallel Processing Symposium, pp. 766-770, 1996.
13. Shousheng He and Mats Torkelson, "Designing pipeline FFT processor for OFDM (de)modulation," ISSSE'98, Vol. 2, pp. 945-950, 1998.
14. M. T. Heideman and C. S. Burrus, "On the number of multiplications necessary to compute a length 2 DFT," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-34, pp. 91-95, Feb. 1986.

15. N. Weste, M. Bickerstaff, T. Arivoli, P. J. Ryan, J. W. Dalton, D. J. Skellern, and T. M. Percival, "A 50MHz 16-point FFT processor for WLAN applications," IEEE Custom Integrated Circuits Conference, pp. 457-460, 1997.
16. J. Melander, L. Wanhammar, T. Widhe, "Design of efficient radix-4 butterfly PEs for VLSI", in Proc. IEEE International Symposium on Circuits and Systems, ISCAS, June 1997.
17. R.C. Gonzales and R. E. Woods. "Digital Image Processing", Addison-Wesley, 2002.
18. Ediz Çetin, Richard C. S. Morling and Izzet Kale, "An Integrated 256-point complex FFT Processor for Real-time Spectrum Analysis and measurement", IEEE Proceedings of Instrumentation and Measurement Technology Conference, vol. 1, pp. 96-101, Ottawa, Canada, May 19-21, 1997.
19. ZAHEER M. ALI, "A High-speed FFT Processor", IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. COM-26, NO. 5, MAY 1978.
20. J. Bhaskar, "VHDL Primer", 3rd Edition, Pearson education Asia, 2001.
21. Douglas L. Perry, "VHDL Programming by Example", 4th Edition, Tata McGraw Hill, 2002.
22. Ben Cohen, Kluwer, "VHDL Coding Styles and Methodologies", 2nd Edition, Academic Publishers. 2000.
23. Stephen M. Triberger, "Field Programmable Gate Array Technology", Kluwer Academic Publishers.
24. Xilinx Application Note, Spartan-II Series and Xilinx ISE 7.1i Design Environment, <http://www.xilinx.com>, 2001.
25. Spartan-II Platform FPGA Handbook October 24, 2002.  
<http://www.xilinx.com/bvdocs/userguides/ug012.pdf>