FAST FOURIER TRANSFORM IMPLEMENTATION USING FIELD PROGRAMMABLE GATE ARRAY TECHNOLOGY FOR ORTHOGONAL FREQUENCY DIVISION MULTIPLEXING SYSTEMS

By

RAMA KRISHNA LOLLA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

To
My Family
&
BABA

ACKNOWLEDGMENTS

I would like to extend my thanks to Dr. Fred J. Taylor for his suggestions at all the stages of the project. This project would not have taken shape without his guidance.

I would like to thank my advisors, Dr. John G. Harris and Dr. John M. Shea, for their timely suggestions. I am also thankful to my colleagues in the High Speed Digital Architecture Laboratory for their support.

I would also like to acknowledge the continuous support my family has given me during the course of my work.

TABLE OF CONTENTS

page

LIST OF TABLES

Table page

LIST OF FIGURES

Figure page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

FAST FOURIER TRANSFORM IMPLEMENTATION USING FIELD PROGRAMMABLE GATE ARRAY TECHNOLOGY FOR ORTHOGONAL FREQUENCY DIVISION MULTIPLEXING SYSTEMS

By

Rama Krishna Lolla

December 2002

Chair: Dr. Fred J. Taylor
Major Department: Electrical and Computer Engineering

Orthogonal Frequency Division Multiplexing (OFDM) is an emerging multi-carrier technique, which uses Fast Fourier Transforms (FFTs) to modulate the data onto sets of orthogonal frequencies. The core operation in the OFDM systems is the FFT unit that consumes a large amount of resources and power. The goal of this thesis was to study better implementation structures for the FFT. The Radix-2 and Radix-4 implementations of the Cooley-Tukey algorithm and the Chirp-z algorithm were implemented using the Field Programmable Gate Array (FPGA) technology. Twos complement numbering system was used in the designs, and their performance was judged on the basis of their implementation complexity and amount of power consumed for implementation.

CHAPTER 1
INTRODUCTION

Orthogonal Frequency Division Multiplexing (OFDM) is an emerging Multi-carrier technique, which uses FFTs to modulate the data onto sets of orthogonal frequencies. Orthogonality enables the frequencies to overlap while still maintaining statistical independence. The transmitter uses an IFFT to convert the “frequency domain” data into the “time domain” and the received signals are converted back into the “frequency domain” by using an FFT at the receiver. An IFFT is similar in structure to the FFT, the differences being the twiddle factors in each being the complex conjugates of other [1]. This core operation is often the limiting technology when it comes to the power consumed for its implementation. The objective of this thesis is to study the implementations of Cooley-Tukey and Chirp-z FFT algorithms onto FPGA technology to arrive at a low power, low latency configuration.

OFDM Overview

OFDM efficiently overcomes the problems that plague most wireless channels. Multi-path propagation is a serious hazard that introduces delay spread accounting for multiple copies of the transmitted signal to reach the receiver. This causes energy of one symbol of information to spill onto several successive symbols. This phenomenon is called Inter Symbol Interference (ISI). OFDM reduces ISI through several simultaneous transmissions, thus making it possible to have an increase in the transmission time for each symbol. OFDM moves the equalization operation to the frequency domain instead of time domain as in the case of single carrier systems.

The OFDM implementation also has excellent ICI performance. Using the FFT, which uses bands of frequencies that are the harmonics of the fundamental frequency band, does this. This ensures minimum cross talk between the sub-carriers thereby reducing the ICI. This does not require the phase lock of the local oscillators. These properties of an OFDM system are the much sought after solutions to combat the delay spread in the wireless environment [1].

FFT Algorithms Explored

The Cooley-Tukey algorithm formulates an efficient way to reduce the usage the number of complex multiplications. The algorithm allows for configuring the design in more than one way based on the fundamental repetitive unit used for implementing the longer point FFTs [2-5]. Two such configurations are explored in this thesis. They are the Radix-2 (grouping in units of 2) and Radix-4 (grouping in units of 4) Cooley-Tukey implementations.

The Chirp-Z algorithm provides for greater frequency resolution that is independent of the sampling rate. Spectral resolution is greatly improved by mapping the contour closer to the poles in the z-domain. In the limiting case, when the contour chosen is a unit circle, the results are a perfect match with the Cooley-Tukey algorithm. Its implementation has blocks of circular convolutions that are usually implemented in CCDs [2-5]. An attempt has been made in this thesis work to assemble the circular convolution blocks onto FPGAs.

The modules in this work were compiled onto Altera FLEX10KE family of devices using Altera MAX+PLUS II software. These FPGAs give maximum flexibility by allowing modifications in the designing as well as the testing phase. The FPGAs can

often be used to gain quicker control over the market at a cheaper price. The internal blocks of the FPGA are usually standardized for each family of device [6-9].

Thesis Organization

Chapter 2 describes the OFDM scheme and the ways it reduces the effects of ISI and ICI. Chapter 3 discusses the details of the Cooley-Tukey and the Chirp-z algorithms for the implementation of FFTs. Chapter 4 discusses the FPGA technology and its comparisons with other technologies. Chapter 5 concludes the thesis work with a detailed description of the actual work done, the results obtained and the inferences drawn.

CHAPTER 2
OFDM THEORY AND IMPLEMENTATION

Description of the Wireless Channel

The wireless communication channel introduces some non-linearity in the signal. These nonlinear effects can be modeled in most cases as an filter. The receiver is assumed to receive multiple copies of the transmitted sequence with different amplitudes and phases. This is mainly due to the different signal paths (Figure 2.1) and their associated path losses. The gains distributed along the multiple paths determine the coefficients of the filtering model of the channel. In addition there are often variations along a path either due to mobile environment or climatic changes. These effects can be mitigated to some degree, by increasing the transmitted power. The more the power generated at the transmitter, the lesser the error probability. A typical power spectrum consists of some peaks and troughs signifying the distribution of power at various frequency bands. An ideal communication strategy assigns signals to those channels or bands having the highest gain. This is called Water Filling Strategy for power allocation [1].

Multi-path propagation effects cause the signal to spread in time into successive signal values. This results in Inter-Symbol-Interference (ISI). Here the energy of one symbol overlaps onto the successive symbols. This is a major concern in the case of single carrier systems. This overlap causes constructive interference at some instances and destructive interference at others. Multi-carrier systems attempt to resolve this issue by dividing the high-speed data stream into several simultaneous transmissions (thus keeping the overall

data rate constant) so that each individual transmission has an increased transmission time. This reduces the probability of each symbol to be misread at the receiver.

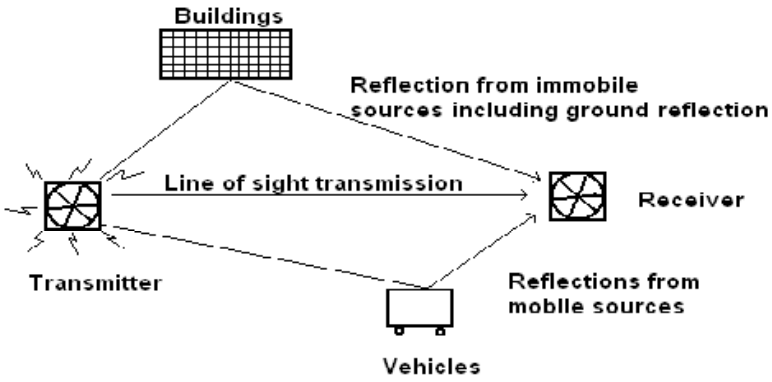

Figure 2.1 Multipath Propagation

Orthogonal Frequency Division Multiplexing (OFDM) is one of the prominent and effective multi carrier transmission techniques. Frequency division multiplexing involves transmitting of various symbols of information at various frequency bands with some “guard band” to separate the carriers. OFDM is an advanced technique that eliminates the use of the guard band even while retaining proper decipherability at the receiver.

Parallel transmission is accomplished by implementing some type of quadrature amplitude multiplexing (QAM) and then transmitting the data after performing an inverse Fourier transform. That is an inverse Fourier transform is taken at the transmitter with the received sequence processed by a FFT. The idea behind taking the inverse Fourier transform at the transmitter can be motivated as follows. Orthogonality is desired in a transmitted array data sequence having known frequency, amplitude and phase. If we assume that if the data is already assumed to be in “frequency domain” at the transmitter, the IFFT produces a “time domain” array of signals subject to some form frequency, amplitude and phase restrictions. At the receiver, the “frequency domain” data is regained by taking the FFT of the received “time-domain” sequence [1].

History of OFDM

An impractical analog implementation of a Fourier transform would involve using oscillators at the required frequencies. The oscillator drift in analog components is a major reason for the initial failure of this line of thought. This would cause the carriers to lose orthogonality and result in a phenomenon called Inter Carrier interference (ICI). The digital implementations became available, the frequency drift problems were mitigated and OFDM research once again resumed [1]. The digital implementations almost meet the orthogonality criteria by remaining at a constant frequency. The block diagram of the OFDM link is shown in Figure (2.2).

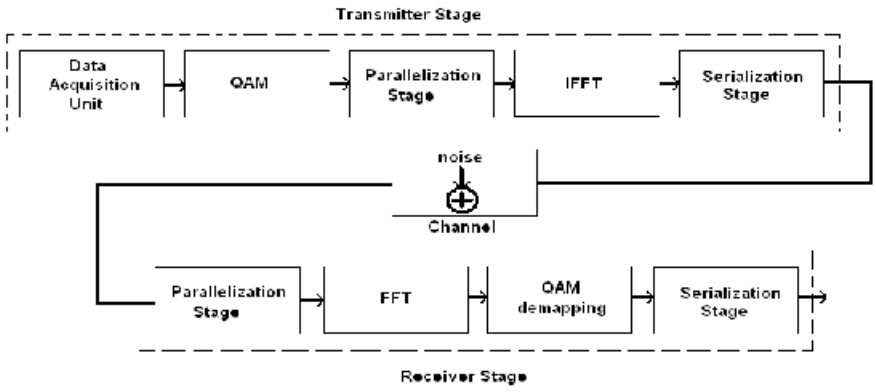

Figure 2.2 General Block Diagram of an OFDM communication system

The quadrature amplitude multiplexing (QAM) stage produces in-phase and quadrature components that can be fed into the Fourier transform stage as the real and imaginary parts of the complex input respectively. The Fourier transform can be thought of as a tool to simultaneously transmit an array of symbols at various frequencies giving the effect of a filter bank. Each sub-channel acts as a single carrier system and can be treated as such allowing for some statistical dependence on one another. Single carrier systems usually have an equalizer in the time domain to nullify the effects of ISI. Multi-

carrier systems like OFDM have the equalizing phenomenon in the frequency domain to combat the ISI. ISI can be effectively eliminated by adding a “guard interval” at the end of each symbol, the length of the guard time being greater than the maximum tolerable delay spread. This reduces the spreading effect of signals onto successive symbols. Efficient utilization of bandwidth is evident in the simultaneous transmissions at a different range of frequencies.

For an N-point FFT/IFFT channel, with the N simultaneous transmission the symbol time can be increased by a factor of N, thus reducing the ISI in the same proportion. The length of the FFT is however constrained by the exponentially increasing complexity of design of the transmitter and receiver modules. So the choice of the length of FFT is usually a trade off between the complexity of implementation and decipherability of data at the receiver.

Orthogonality is a concept that statistically quantifies the independence among the components of the skeleton structure for describing any system. The projections of any signal along the components of the orthogonal system could sufficiently represent the system. In actual implementations, the information is sent in the shape of sinc (sin x / x) pulses so that the frequency domain representation would be rectangular pulses. The sinc pulses have nulls at periodic intervals and if the subcarriers are placed at that spacing, then the maxima of each subcarrier would occur only when all other subcarrier contributions are zero. The inherent orthogonality in an OFDM system allows the spectrum to overlap without causing any interference problems. This eliminates the usage of any steep bandpass filters that are required for other frequency division multiplexing systems implying lesser implementation complexity. The lack of orthogonality causes

some amount of cross talk between the subcarriers and this phenomenon is called the Inter Carrier Interference (ICI). This ICI is to be minimized to establish a good communication link. Orthogonality is introduced into the system by ensuring that when one the output corresponding to a particular sub-carrier is at its peak, there is minimum (ideally zero) contribution from the remaining sub-carriers. The sub-carrier spacing is thus determined by the null-null spacing of each transmission. This forces the correlation between the sub-carriers to zero.

Thus the OFDM system attempts to reduce the problems of ISI and ICI with very low implementation complexity. This may not be effectively removed in the single carrier systems even after equalization. The implementation of the OFDM however brings into focus the issues of power usage. It is observed that the major power sink in the transmitter/receiver design was the IFFT/FFT and for low power applications, this issue must be dealt with extensively. Fortunately, the IFFT/FFT implementations for an arbitrary N-point implementation vary only in their twiddle factors in most cases. There are many ways of the FFT implementation. The underlying concept of the DFT and two algorithms (Cooley-Tukey and Chirp-z algorithms) to implement the FFT were studied as a part of this thesis work and they were compared in terms of power and latency issues.

CHAPTER 3
ALGORITHM THEORY AND DESCRIPTION

The algorithms used for the purpose of this thesis work were the Cooley Tukey and the Chirp-z Transform Algorithms. The main purpose of this study was to innovate a better FFT implementation structure for OFDM applications. The following are descriptions of the Cooley Tukey algorithm and the Chirp-z algorithm.

Cooley-Tukey Algorithm

Formally a discrete Fourier transform (DFT) is given by equations (3.1) and (3.2) as

Analysis Equation:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k] e^{j(2\pi kn)/N} \qquad \forall n \in [0, N-1] \tag{3.1}$$

Synthesis Equation:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j(2\pi kn)/N} \qquad \forall k \in [0, N-1] \tag{3.2}$$

where, x[n] is the nth sample of an N-element time series and correspondingly, X[k] is the k^{th} harmonic of an N-point discrete Fourier transform of x[n]. The summations in both the synthesis and analysis equations exist only if all the values of x and X are bounded. The DFT assumes periodicity. The multiplying exponential coefficients are of unit magnitude and can effectively be represented as equally spaced points along a unit circle mapped in the z-domain according to their phase. More detailed description of the properties of the DFT equations described above can be found in [2-5]. The implementation of the equations (3.1) and (3.2) in their canonic form would require

N*(N-1) complex additions and N^2 complex multiplications. This can result in a high implementation complexity and latency.

Cooley and Tukey published their simplifications to this set of equations in 1965 that took form of the algorithm described below [5]. A complexity reduction is achieved by breaking down long DFTs into collections of smaller FFTs. The algorithm is described as follows.

Let N be the number of points in the input sequence. Consider representing N as the composite number,

$$\mathrm{N} = \mathrm{N}_1 * \mathrm{N}_2 \tag{3.3}$$

The time domain index n, and the frequency domain index k are also resolved as

$$\mathrm{n} = \mathrm{n}_2\ \mathrm{N}_1 + \mathrm{n}_1 \qquad \forall\, \mathrm{n}_1 \in [0, \mathrm{N}_1 - 1] \text{ and } \mathrm{n}_2 \in [0, \mathrm{N}_2 - 1] \tag{3.4}$$

$$\mathrm{k} = \mathrm{k}_1 \mathrm{N}_2 + \mathrm{k}_2 \qquad \forall\, \mathrm{k}_1 \in [0, \mathrm{N}_1 - 1] \text{ and } \mathrm{k}_2 \in [0, \mathrm{N}_2 - 1] \tag{3.5}$$

Substituting these in the DFT Equation (3.2)

$$X[\,\mathrm{k}_1\mathrm{N}_2 + \mathrm{k}_2\,] = \sum_{n=0}^{N-1} x[\mathrm{n}_2\ \mathrm{N}_1 + \mathrm{n}_1] e^{-j(2\pi(\mathrm{n}_2\,\mathrm{N}_1 + \mathrm{n}_1)(\mathrm{k}_1\mathrm{N}_2 + \mathrm{k}_2)/N}} \tag{3.6}$$

From Equation (3.6), the exponential is of the form

$$W_N^{(k_1N_2+k_2)(n_2N_1+n_1)} = W_N^{(k_1n_2N_1N_2)} * W_N^{(k_1n_1N_2)} * W_N^{(k_2n_2N_1)} * W_N^{(k_2n_1)} \tag{3.7}$$

Using the relations,

$$W_N^{\ m} = W_{N/m} \qquad W_N^{\,k_1n_2N_1N_2} = e^{-j2\pi k_1 n_2} = 1 \qquad \exists n_2, k_1$$

Equation (3.7) becomes,

$$W_N^{(k_1N_2+k_2)(n_2N_1+n_1)} = W_{N_1}^{(k_1n_1)} * W_{N_2}^{(k_2n_2)} * W_N^{(k_2n_1)} \tag{3.8}$$

Substituting Equation (3.8) in Equation (3.6),

$$X(k_1 N_2 + k_2) = \sum_{n_1=0}^{N_1-1}\left[\left(\sum_{n_2=0}^{N_2-1} x[n_2 N_1 + n_1] * W_{N_2}^{\,k_2 n_2}\right) * W_N^{\,k_2 n_1}\right] * W_{N_1}^{\,k_1 n_1} \qquad (3.9)$$

The inner sum is clearly a N_2-point DFT for fixed n_1 and the outer sum is an N_1-point DFT for fixed k_2. There is also a gluing factor, known as the "twiddle factor", $W_N^{\,k_2 n_1}$ which is multiplied by the inner sum of products term for the fixed values of k_2 and n_1. Tables (3.1) and (3.2) shows the unresolved indices (n, k) in terms of the resolved indices ((n_1, n_2), (k_1, k_2)).

Table 3.1 Time-domain index n resolved in terms of n1 and n2

n_1	n_2→	0	1	2		N_2-1
0		0	N_1	2* N_1		(N_2-1)*N_1
1		1	N_1+1	2* N_1+1		(N_2-1)* N_1+1
2		2	N_1+2	2* N_1+2		(N_2-1)* N_1+2
N_1-1		N_1-1	2* N_1-1	3* N_1-1		N-1

Table 3.2 Resolution of the frequency domain index k

k_1	k_2→	0	1	2		N_2-1
0		0	1	2		N_2-1
1		N_2	N_2+1	N_2+2		2*N_2-1
2		2*N_2	2*N_2+1	2*N_2+2		3*N_2-1
N_1-1		(N_1-1)*N_2	(N_1-1)*N_2+1	(N_1-1)*N_2+2		(N-1)

Thus the algorithm describes a means by which sets of N_2-point DFTs (for fixed n_1) interface sets of N_1-point DFTs (for fixed k_2). The algorithm is interpreted in the Figure (3.1).

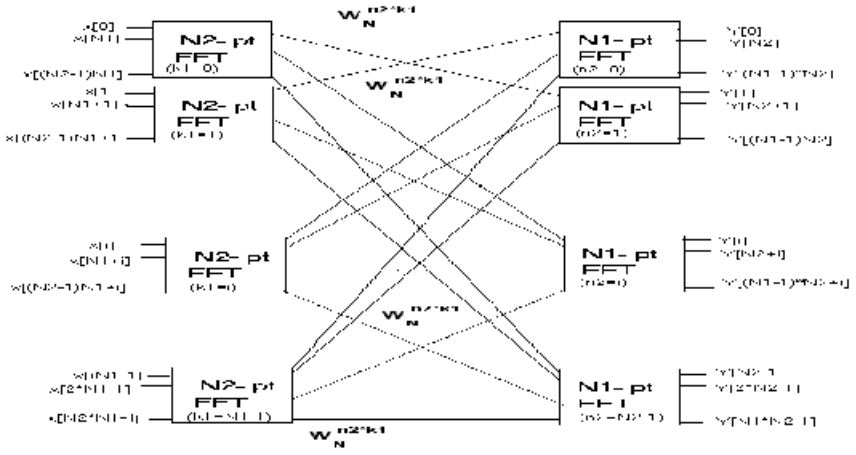

Figure 3.1 Cooley-Tukey Algorithm Implementation

Complexity Analysis

The complexity of the entire N-point DFT implementation can be modeled as the complex multiplication and addition count associated with N_1 N_2-point FFT units and N_2 N_1-point FFT units and N twiddle factors [5].

$$MultiplierComplexity = N_1\left[(N_2)^2\right] + N_2\left[(N_1)^2\right] + N$$

$$\Rightarrow MultiplierComplexity = N(N_1 + N_2 + 1) \tag{3.10}$$

$$AdditionComplexity = N_1\left[N_2 * (N_2 - 1)\right] + N_2\left[N_1 * (N_1 - 1)\right]$$

$$= N_1 N_2^2 - N_1 N_2 + N_2 N_1^2 - N_1 N_2$$

$$\Rightarrow AdditionComplexity = N(N_2 + N_1 - 2) \tag{3.11}$$

If N can be resolved into a highly composite number:

$$N = N_1 * N_2 * N_3 * \ldots * N_n \tag{3.12}$$

then the multiplier complexity is approximately $N*(N_1 + N_2 + N_3 + \ldots + N_n)$

This is much lesser than the original N^2 for a direct implementation shown in Equation (3.1). It is common knowledge that a multiplier unit is more complex than a simple adder. So the complexity of the DFT units is expressed often in terms of the multiplier complexity alone.

Radix-2 Algorithm

When N is of the form $N=2^n$, it can be factored as N=2 x 2 x 2 x ….x 2. Thus all the individual blocks that are implemented would only be 2-point FFT blocks, which require no multiplications at all. All the butterfly coefficients would then be implemented as a part of the gluing logic that connects the individual blocks. If the first level of factorization is N=2 x N/2, i.e., $N_1 = 2$ and $N_2 = N/2$, then the frequency domain and time domain indices (k, n) can be modified as

$$n = 2n_2 + n_1 \qquad n_1 \in [0,1], n_2 \in [0, (N/2 - 1)] \tag{3.13}$$

$$k = (N_1/2)k_1 + k_2 \qquad k_1 \in [0,1], k_2 \in [0, (N/2 - 1)] \tag{3.14}$$

Substituting from Equations (3.13) and (3.14) in Equation (3.9),

$$X[k_1(N/2) + k_2] = \sum_{n_1=0}^{1}\left[\left(\sum_{n_2=0}^{\frac{N}{2}-1} x[2n_2 + n_1] * W_{\frac{N}{2}}\right) * W_N^{k_2 n_1}\right] * W_2^{k_1 n_1}$$

$$= \sum_{n_2=0}^{N/2-1} x[2n_2] * W_{N/2}^{k_2 n_2} + (-1)^{k_1} * W_N^{k_2} * [\sum_{n_2=0}^{N/2-1} x[2n_2 + 1] * W_{N/2}^{k_2 n_2}]$$

$$\Rightarrow X[k_1(N/2) + k_2] = X_0'[k_2] + (-1)^{k_1} * W_N^{k_2} * X_1'[k_2] \tag{3.15}$$

where the terms X_i' are (N/2)-point DFT units. The first term is a grouping of even indexed terms in the time domain and the second term is a grouping of odd-indexed terms

in the time domain. We can further express the above result to extract the even and odd indexed frequency domain indices as

$$X[k_2] = X_0^{'}[k_2] + W_N^{k_2} * X_1^{'} \qquad 0 \le k_2 \le N/2 - 1 \tag{3.16}$$

$$X[k_2 + N/2] = X_0^{'}[k_2] - W_N^{k_2} * X_1^{'} \qquad 0 \le k_2 \le N/2 - 1 \tag{3.17}$$

A block representation of a basic radix-2 implementation unit is shown in the Figure (3.2).

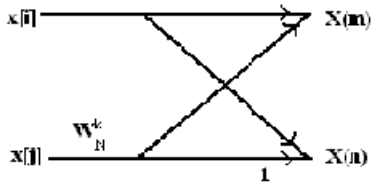

Figure 3.2 Radix-2 repetitive unit

This one level of reduction would reduce the implementation complexity to sum of multiplier complexities of two N/2-point DFT units and N/2 multiplications and N additions. If $N=2^m$ and we factorize N m times, then the multiplier complexity is given by

$$MultiplierComplexity = (N/2) * \log_2(N) \tag{3.18}$$

and the addition complexity is given by

$$AdditionComplexity = N * \log_2(N) \tag{3.19}$$

Further if we observe the twiddle factors to be multiplied, we see that there are some factors which are only multiplications with $0,(-1)^k,(-j)^k$. A detailed diagram of a radix-2 8-point implementation is shown in Figure (3.3). The resulting algorithm is called a radix-2 fast Fourier transform (FFT).

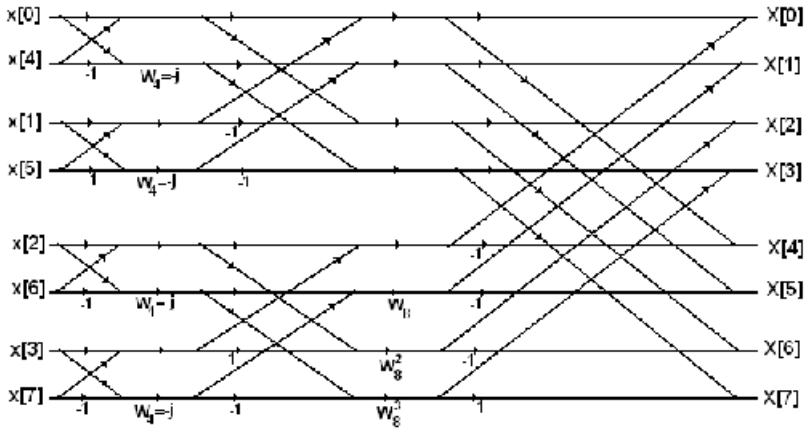

Figure 3.3 Implementation of a Radix-2 8-point FFT unit

Radix-4 Algorithm

The radix-4 FFT algorithm goes a step further in reducing the complexity of a DFT implementation. When N is a power of 4, i.e., N= 4 x 4 x 4 x … x 4, N can be factorized as N= 4 x (N/4). Here N_1=4 and N_2 = N/4.The time and frequency domain indices (n,k) can therefore be expressed as

$$n = 4n_2 + n_1 \qquad n_1 \in [0,3], n_2 \in [0,(N/4 - 1)] \tag{3.20}$$

$$k = (N/4)k_1 + k_2 \qquad n_1 \in [0,3], n_2 \in [0,(N/4 - 1)] \tag{3.21}$$

$$X[k_1 * (N/4) + k_2] = \sum_{n_1=0}^{3}\left[\left(\sum_{n_2=0}^{N/4-1} x[4n_2 + n_1] * W_{N/4}^{k_2 n_2}\right) * W_N^{k_2 n_1}\right] * W_4^{k_1 n_1} \tag{3.22}$$

$$\begin{aligned} X[k_1 * (N/4) + k_2] &= \sum x[4n_2] * W_{N/4}^{k_2 n_2} + (-j)^{k_1} * W_N^{k_2} \sum x[4n_2 + 1] * W_{N/4}^{k_2 n_2} \\ &+ (-1)^{k_1} * W_N^{2k_2} \sum x[4n_2 + 2] * W_{N/4}^{k_2 n_2} + (j)^{k_1} * W_N^{3k_2} \sum x[4n_2 + 3] * W_{N/4}^{k_2 n_2} \end{aligned} \tag{3.23}$$

$$\begin{aligned} X[k_1 * (N/4) + k_2] &= X_0'[k_2] + (-j)^{k_1} * W_N^{k_2} * X_1'[k_2] + (-1)^{k_1} * W_N^{2k_2} * X_2'[k_2] \\ &+ (j)^{k_1} * W_N^{3k_2} * X_3'[k_2] \end{aligned} \tag{3.24}$$

The $X_i^{'}$ s in Equation 3.20 are all N/4 point FFT units of grouped terms of type (4m+i). Thus the complexity reduces to sum of complexity of 4 N/4 point FFT units and 3N/4 complex multiplications and 3N complex additions. Expressing the right hand side of the equations for the varying k_1 values can further reduce this.

$$X[k_2] = \left(X_0^{'}[k_2] + W_N^{2k_2} * X_2^{'}[k_2]\right) + \left(W_N^{k_2} * X_1^{'}[k_2] + W_N^{3k_2} * X_3^{'}[k_2]\right) \quad (3.25)$$

$$X[(N/4) + k_2] = \left(X_0^{'}[k_2] - W_N^{2k_2} * X_2^{'}[k_2]\right) - j\left(W_N^{k_2} * X_1^{'}[k_2] - W_N^{3k_2} * X_3^{'}[k_2]\right) \quad (3.26)$$

$$X[(N/2) + k_2] = \left(X_0^{'}[k_2] + W_N^{2k_2} * X_2^{'}[k_2]\right) - \left(W_N^{k_2} * X_1^{'}[k_2] + W_N^{3k_2} * X_3^{'}[k_2]\right) \quad (3.27)$$

$$X[(3N/4) + k_2] = \left(X_0^{'}[k_2] - W_N^{2k_2} * X_2^{'}[k_2]\right) + j\left(W_N^{k_2} * X_1^{'}[k_2] - W_N^{3k_2} * X_3^{'}[k_2]\right) \quad (3.28)$$

From Equations (3.25), (3.26), (3.27), (3.28), we observe that the number of complex additions reduces to 2N from 3N. So if we go for $\log_2(N)$ stage implementations(and factorizations), then we see that

$$MultiplierComplexity = (3N/8) * \log_2(N) \quad (3.29)$$

and

$$AdditionComplexity = N \log_2(N) \quad (3.30)$$

This implementation can also be seen as a repetition of a fundamental unit, which is the radix-4 block shown in Figure (3.4).

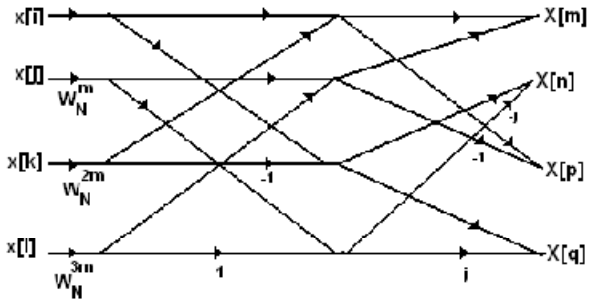

Figure 3.4 Radix-4 basic block

We observe that the basic repetitive block in the radix-4 algorithm does not have any actual multiplications just like the radix-2 block. The resulting algorithm is called a radix-4 FFT.

Chirp-z Algorithm

One interesting feature of the Cooley-Tukey algorithm implementation is that the frequency resolution is always related to the number of points at the input of the FFT unit. The only way to increase the spectral resolution is to increase the number of data points at the input of the DFT unit. Also, this algorithm only maps onto equally spaced locations on the unit circular contour in the z-domain. As a result, the FFT can only provide constant bandwidth analysis in the context of N equal frequency bands of constant gain. An alternative way of implementation is to map onto a contour close to the poles in the z-domain so that the spectral resolution is improved.

The Chirp-z transform algorithm avoids these problems by giving the freedom to choose a range of frequencies to be analyzed independent of the sampling rate and define the frequency response resolution to be determined by the chosen contour in the z-domain. If the contour is chosen to be the unit circle, the Chirp-z transform and Cooley-Tukey FFT algorithm produce the same result. The downside to the Chirp-z implementation is its higher implementation complexity and slower performance in comparison to the Cooley-Tukey FFT algorithm. The algorithm modifies the z-domain mapping to represent the FFT of the signal.

If x(n) is a N-point sequence, the z-transform is defined as

$$X(z_k) = \sum_{n=0}^{N-1} x[n] * z_k^{-n} \qquad k \in [0, L-1] \tag{3.31}$$

Here L represents the number of frequency domain outputs; clearly this is independent of the sampling rate. The z- domain contour is chosen starting at a point closer to the poles of the system, which is to be resolved and also it is a continuous track which could also be a unit circle as in the case of the DFT. Though we have only an N-point sequence x[n] in the time domain, we can have an L-point sequence X[z] in the frequency domain.

If $z_0 = r_0 e^{j\theta_0}$ is the origin of the contour, then the contour spirals either inwards or outwards based on the value of R according to the equation,

$$z_k = z_0 \left(R * e^{j\phi}\right)^k \qquad k \in [0, L-1]$$

This results in

$$\Rightarrow z_k = r_0 e^{j\theta_0} \left(R * e^{j\phi_0}\right)^k \qquad k \in [0, L-1] \tag{3.32}$$

Here (r_0,?$_0$) represent the origin of the contour spiral, ϕ_0 represents how the successive stages follow on the contour and R determines the convergence or divergence of the contour. If R<1 the contour spirals inwards towards the origin and if R>1 the contour spirals away from the origin. If R=1, then the contour is a circle of radius r_0.

Substituting Equation (3.32) in Equation (3.31),

Equation 3.33

$$X[z_k] = \sum_{n=0}^{N-1} x[n] * \left(r_0 e^{j\theta_0} \left(R * e^{j\phi_0}\right)^k\right)^{-n}$$

$$= \sum_{n=0}^{N-1} x[n] * \left(r_0 e^{j\theta_0}\right)^{-n} * \left(R * e^{j\phi_0}\right)^{-nk} \tag{3.33}$$

If we define $V = R * e^{j\phi_0}$,

$$X[z_k] = \sum_{n=0}^{N-1} x[n] * (r_0 e^{j\theta_0})^{-n} * V^{-nk} \tag{3.34}$$

To simplify this further, we use the relation

$$nk = \tfrac{1}{2}\left[k^2 + n^2 - (k-n)^2\right] \tag{3.35}$$

in Equation (3.34).

$$X[z_k] = \sum_{n=0}^{N-1} x[n] * \left(r_0 e^{j\theta_0}\right)^{-n} * V^{-n^2/2} * V^{-k^2/2} * V^{(k-n)^2/2}$$

$$\Rightarrow X[z_k] = V^{-k^2/2} \sum_{n=0}^{N-1} \left[x[n]\left(r_0 e^{j\theta_0}\right)^{-n} * V^{-n^2/2}\right] * V^{(k-n)^2/2} \tag{3.36}$$

Defining the grouped term to represent a new sequence g (n)

$$g(n) = x[n]\left(r_0 e^{j\theta_0}\right)^{-n} * V^{-n^2/2} \qquad n \in [0, N-1] \tag{3.37}$$

In the case of a circular mapping (R=1),

$$z_k = r * e^{-j2\pi k/N} \tag{3.38}$$

the z_k are equally spaced points along a circle of radius r.

Then

$$X[z_k] = \sum_{n=0}^{N-1} x(n) * r^{-n} * e^{-j2\pi kn/N}$$

$$= \sum_{n=0}^{N-1} [x(n) * r^{-n}] * e^{-j2\pi kn/N}$$

Here the modified sequence is $y(n)=x(n)*r^{-n}$ and it is sufficient to calculate the DFT of the modified sequence.

Returning to the more general case, consider a sequence h(n) defined as

$$h(n) = V^{n^2/2} \tag{3.39}$$

Substituting from Equation (3.37) and Equation (3.39) in Equation (3.36),

$$X(z_k) = V^{-k^2/2} \sum_{n=0}^{N-1} g(n)h(k-n) \tag{3.40}$$

Defining the convolution sum as another sequence y(k), we have

$$y(k) = \sum_{n=0}^{N-1} g(n)h(k-n) \tag{3.41}$$

Thus Equation (3.40) becomes

$$X(z_k) = \frac{y(k)}{h(k)} \tag{3.42}$$

Here the sequence y(n) is a convolution between a sequence g(n) of length N and second sequence h(n) of infinite length. Taking a M-point segment of this infinite length sequence for practical purposes, y(n) would be a sequence of length L given by

$$L = M + N - 1 \tag{3.43}$$

The convolution filter h(k) is usually implemented using charge coupled devices (CCD) or surface acoustic wave (SAW) devices. Since we try to obtain a frequency resolution of L, the length of the convolution filter h(n) is considered to be

$$M = L - N + 1 \Rightarrow M = L - (N-1) \tag{3.44}$$

which implies that

$$-(N-1) \le n \le (L-1)\,. \tag{3.45}$$

The computational complexity of the Chirp-z algorithm is thus dependent on M requiring M*$\log_2$M complex multiplications. Compared to N*L, the complexity comparison can be argued as follows. When L is small, direct computation is more efficient but when L is large, the Chirp-z transform is better.

To calculate a DFT, set the contour parameters as:

r_0=R=1, $?_0$=0, F_0=2p/N and L=N.

So, h(n) from Equation (3.39) simplifies into

$$\begin{aligned} &h(n) = \cos(\tfrac{\pi n^2}{N}) + j\sin(\tfrac{\pi n^2}{N}) \\ &\Rightarrow h(n) = h_r(n) + jh_i(n) \end{aligned} \tag{3.46}$$

and

$$h(-n) = V^{-n^2/2} = \cos(\tfrac{\pi n^2}{N}) - j\sin(\tfrac{\pi n^2}{N}) \qquad (3.47)$$

These coefficients are implemented in a ROM for the pre-multiplications and post-multiplications. The algorithm is implemented as shown in the following figure (Figure 3.5).

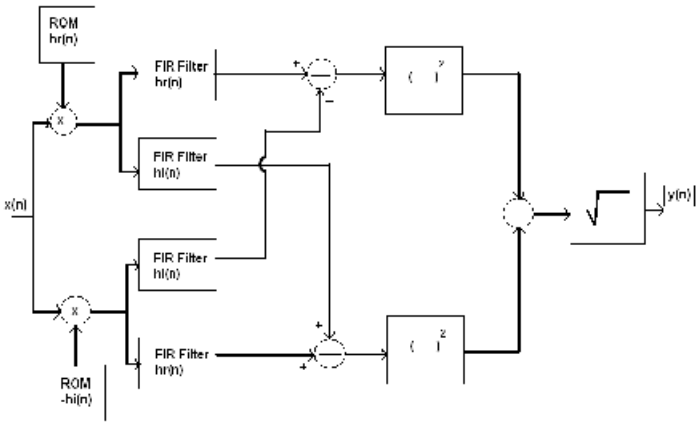

Figure 3.5 Chirp -z implementation

The sequence h(n) has n^2 complex exponential values that can be thought of as a continuously increasing frequency term as ?n = $n^2F_0/2$ = $(nF_0/2)n$. This signal, shown in Figure 3.6, has an increasing frequency and sounds like the chirp of a bird.

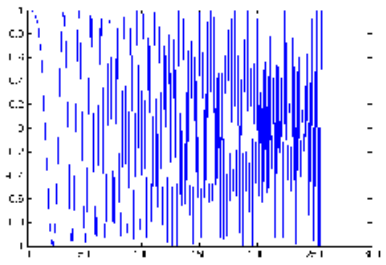

Figure 3.6 Chirp Signal

There are some interesting properties of this chirp signal which enhance the applications of this Chirp-z algorithm for the computation of the DFT. The phase of the

signal in Figure 3.6 is parabolic as shown is shown in Figure 3.7. The figure shows a linear region as well as the curvature in the in the phase. So the phase can be expressed in the form Phase(n) = a*n+ß*n^2. Here a determines the linear region and ß determines the curvature in the Figure 3.7.

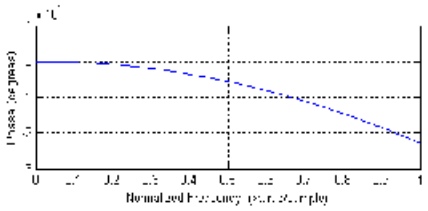

Figure 3.7 Phase response of the Chirp Signal shown in Figure 3.6

An interesting point is that a chirp signal can be completely recovered from an impulse signal after passing through a system with unit magnitude and phase shown in the Figure 3.7 and vice versa. The reverse system would require a system with unit magnitude response and increasing phase (opposite to the original system). This property of the chirp signal encourages its use in radar systems, which require short pulses with higher energy.

Relation between the tolerances of the chirp system and the DFT algorithm is obtained by seeing the equation describing the incremental evolution factor F. Defining f_2 and f_1 to be the maximum and minimum operating frequencies,

$$\phi_0 = \left(\frac{2\pi}{F_s}\right)\left(\frac{(f_2 - f_1)}{N}\right) \qquad (3.48)$$

The frequency resolution in a conventional DFT system is given by

$$\phi_{DFT} = \left(\frac{2\pi}{N}\right) \qquad (3.48)$$

From Equations 3.48 and 3.49, we see that

$$\phi_0 = \phi_{DFT}\left(\frac{f_2 - f_1}{F_s}\right) \qquad (3.48)$$

This implies that the Chirp-z algorithm has a lesser frequency tolerance for a given N. This also indicates that the number of points required to achieve a particular spectral resolution is always smaller when using the Chirp-z algorithm.

CHAPTER 4
FIELD PROGRAMMABLE GATE ARRAYS

The field programmable gate arrays (FPGAs) are a class of programmable devices which house large circuits with gate count exceeding 20,000 gates, a count that is too large to be fit onto a CPLD. CPLDs have blocks of AND gates interfacing blocks of OR gates. Unlike the CPLDs, the FPGAs have logic blocks interconnected with sets of programmable switches. The structure of a general FPGA is shown in the figure(4.1).

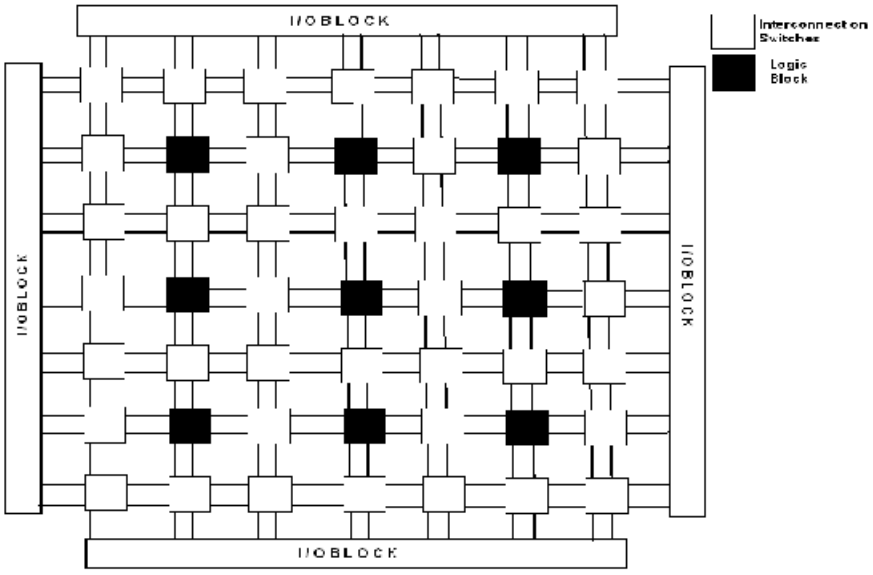

Figure 4.1 General structure of an FPGA

There are three types of blocks in the figure, viz., I/O blocks, logic blocks and interconnection switches. The logic blocks, all identical and usually standardized for each family of devices, are arranged in a neat arrangement of a matrix. The I/O blocks usually interface the internal circuitry to the external pins. The interconnecting switches connect the I/O blocks and the logic blocks. These switches, shown in figure (4.2), are

programmable and form a connection between a horizontal and vertical line based on the value of the SRAM cell (‘0’ for no connection, Vv?V_h, and ‘1’ for a formed connection, V_v=V_h).

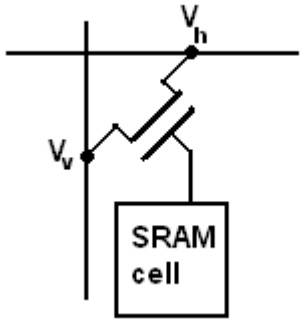

Figure 4.2 Programmable Interconnection Switch

Each logic segment of a user program must be small enough to fit into a logic block. Each logic block is usually an implementation of either look-up-tables (LUT), multiplexers or general gates. An LUT implementation of a three input function f=X_1X_2+ X_1X_3+ X_2X_3 is shown in figure (4.3) and the truth table is implemented in table (4.1).

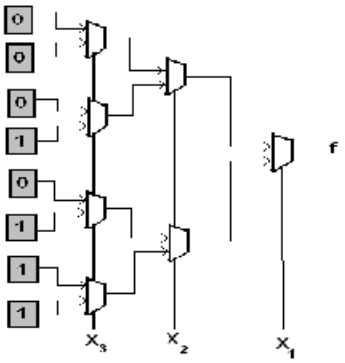

Figure 4.3 A 3-input LUT implementation

Table 4.1 Truth table of the function implemented in Figure (4.3)

X_1	X_2	X_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

In Figure (4.3), each multiplexer is controlled by a single input based on what the multiplexer decides to pass a particular input to the output. Usually the number of inputs to the LUT is about five, which then would require 32 input blocks. Since the FPGAs are volatile, they must be reprogrammed every time they are powered up. An alternative solution is to have a RAM/ROM memory block that automatically supplied these requisite values at power on. Each of the memory cells holds a value of either a ‘1’ or a ‘0’. The values that are fed into the SRAM cells are calculated using a simple protocol. From the truth table, if the entries are placed in ascending order, and if the first level of multiplexers is controlled by the least significant bit (LSB), and the last level of multiplexers is controlled by the most significant bit (MSB), the SRAM values would be in the same order as the output values of the truth table. When a circuit is implemented in an FPGA, the logic blocks are programmed to realize the necessary functions and the programmable switches are also programmed to make the suitable interconnections.

Power Calculations in FPGAs

Power dissipation in digital circuits is often the limiting factor in the utility of a particular circuit in an application. For the purpose of this thesis, Altera FLEX10KE FPGAs were used and their power dissipation is given by the following equation [8].

$$Power = (I_{CCINT} * V_{CCINT}) + \sum_{n=1}^{d} P_{DCn} + 0.5 * OUT * C_{AVE} * V_O * f_{MAX} * tog_{IO} * V_{CCIO} \quad (4.1)$$

where I_{CCINT}= no-load current in the device
V_{CCINT}= no-load voltage V_{CC}
d= number of DC outputs
P_{DCn}=DC output of output n
f_{MAX}= Maximum frequency of operation
tog_{IO}=average number of I/O pins toggling at each block
V_{CCIO}=DC power supply value.
OUT=Number of output and bi-directional pins
C_{AVE}=Average capacitance of the FPGA device
Vo= Voltage level of the high output state

Costs Involved in FPGA Fabrication

The actual cost of FPGA fabrication is the engineering costs, and tool (software/hardware) price. The engineering cost for an FPGA fabrication is much less than that of the ASIC counterpart. But the actual comparison of the FPGA costs is evident when it is compared to the manufacturing cost of ASIC devices. The ASIC devices on the other hand have high NRE costs and longer times to market the product. This actually is the major advantage for the FPGA. The break even number of the FPGA design can be found as follows.

FPGA cost = Engineering costs & tools + total sales for the all items sold

On the other hand,

ASIC cost = NRE + Engineering cost & tools + total sales for all items sold + Re-spin cost + Inventory costs + Accounting for future price reductions.

When the additional costs of the ASICs are considered, the FPGAs are a much better choice for even a moderate amount of sales.

Comparison to other Technologies

Programmability is one very good advantage of the FPGAs that is absent in the ASIC implementations of digital circuits. ASICs cannot be changed at will unlike the FPGAs. The design and testing cycles in an FPGA are much shorter than an ASIC and hence can be marketed much faster. However, for high volume productions, ASIC implementations are much cheaper. Since optimizations can be done up to the gate level in ASIC implementations, they are more power efficient.

CHAPTER 5
IMPLEMENTATION DETAILS AND RESULTS

This chapter describes inferences drawn from the results and the work done for obtaining the results, and also a description of the tools used.

Description of the Work

The thesis work is based on modules built for the purposes of this thesis rather than the standard modules provided along with Altera tools. This was done to obtain an in-depth understanding of the pipelining and FPGA concepts in general.

The Cooley-Tukey and Chirp-Z algorithms were implemented using a fixed-point 2's complement integer arithmetic in VHDL and Verilog. The Cooley-Tukey FFTs have been fit into the Altera FLEX10KE family of devices but the Chirp-z FFTs were too cumbersome (24 multipliers of the type used in this work for a 4-point implementation) to fit onto the FPGAs. Matlab models were built for those designs which could be fit onto Altera FPGAs. These modules were used for observing the performance of the FFTs in varying noise environments. The fixed-point implementation allows power efficient high-speed operations at low cost. This is very much suitable for the mobile/portable applications [10]. This implementation on the other hand loses precision thus decreasing the dynamic range and increasing the round off noise. It is important to note that the complexity of design increases exponentially as the internal bus width is increased.

From the packaging point of view, the pin count is also a major issue. If the complex input were of 16-bit width, then a parallel input of an N-point FFT (N>4) would be a virtual impossibility. An alternative work around for this problem is to provide only 2 16-

bit inputs that would take in real and complex data inputs simultaneously at clocked intervals. The main drawback in this implementation is that the FFT would assume a serial-type form and hence the actual operation of the FFTs is done at N times lesser speed.

The effect of this bus width is visible at the output also. Noise is introduced in the system due to insufficient representation of all the numbers in the system. This type of noise is called round-off noise and it propagates in the system along successive stages. Initially the system was attempted to be with only an internal bus width (all the twiddle factors, outputs of all stages) of 16-bit width only. This requires rounding off the output of a 16-by-16-bit multiplier from 32 bits to 16-bit width (thereby losing 16 bits of precision), and a 16-bit adder (losing ½ bit precision loss at each unit). Tables 5.1 and 5.2 reveal the implementation details of the case where the round-off errors have not been eliminated. The deficiencies in the preliminary design were removed after employing the following techniques.

1. Each multiplier output is not truncated till it reaches one more level (Figure (5.1)).
2. The conventional multiplier was replaced with a pipelined multiplier (Figure 5.2), which greatly reduced the speed-bottleneck and increased the maximum operating frequency of the system.
3. All 16-bit adder/subtraction units were replaced with 32-bit units.
4. A serial-to –parallel converter is placed at the input unit and a parallel-to-serial converter is placed after the output unit to reduce the pin counts.

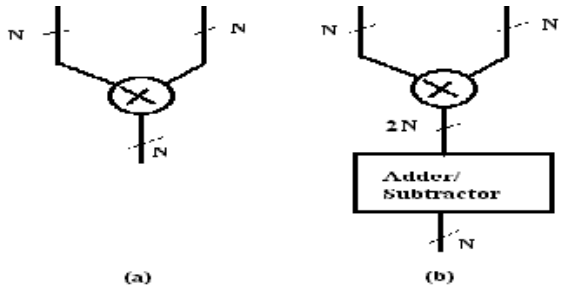

Figure 5.1 Implementation of Multipliers (a) shows the initial truncating configuration and Figure (b) shows the truncation operation after one more level of processing

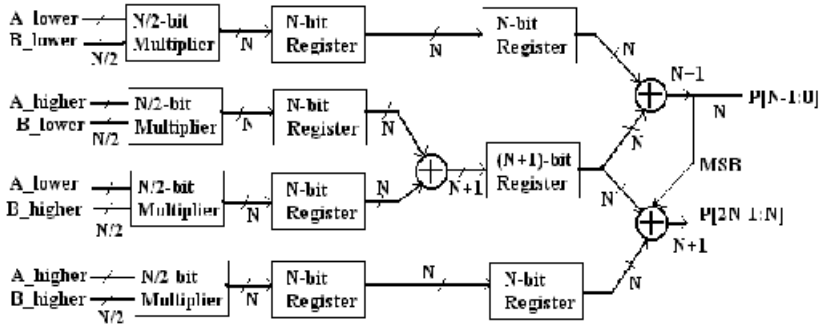

Figure 5.2 N-by-N-bit Pipelined Multiplier

In general, the internal bus width was standardized to 32-bit width. These adjustments greatly increased the complexity in design. The pipelined multiplier required 3847 logic blocks while the conventional Baugh-Wooley multiplier required only 2160 logic blocks for its implementation. Due to the pipelined multiplier implementation, there was an observed increase in the maximum possible operating frequency from about 7 MHz to about 58MHz and 30 MHz for the Radix-2 and Radix-4 Cooley-Tukey 8-point FFT implementations respectively. The pipelined multiplier was a 7-stge case involving the usage of smaller 4-by-4 point multiplications. Tables 5.3 and 5.4 quantify the implementation issues of the case where round off errors were eliminated to a great extent.

Description of Tools Used

The Altera MAX+PLUS II software was used to synthesize the VHDL/Verilog code. The software contained tools to compile, simulate and edit the floor plan of the design. The compiler was equally optimized for area and speed. The designs were fit into the FLEX10KE device family. The FLEX10KE family of devices has the following features:

- High gate density implementation
- Accommodates designs of about 200,000 typical gates
- 4096 SRAM bits per Embedded Array Block (EAB).
- Multi-volt I/O pins (2.5V, 3.3 V or 5.0 V devices)
- Built-in Joint Test Action Group (JTAG) Boundary Scan Test (BST) circuitry available without consuming additional device logic.
- Built in low-skew clock distribution trees.
- Flexible fast track interconnects
- Powerful I/O pins

Each FPGA has an embedded array (EAB) and a logic array (LAB) which are useful for efficient implementations. The embedded array is used in implementing a variety of memory functions, complex logic functions, microcontroller applications and data transform. The logic array is used to implement a multitude of general logic functions. Each LAB has 8 logic elements (LEs) and a local interconnect and each LE has a four-input look-up-table (LUT), a programmable flip-flop and a dedicated signal path for carry and cascaded functions. The FPGA device has the ability of be configured either serially or in parallel synchronously/asynchronously. The average capacitance of the device remains unchanged for any operating frequency.

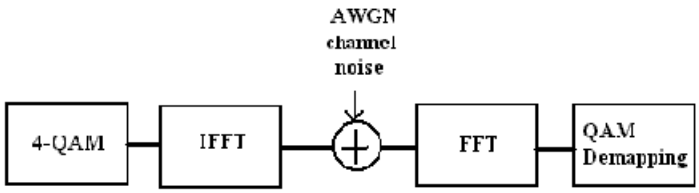

Figure 5.3 Model used in the Thesis work

Figure (5.3) shows the model used in the thesis work. The IFFT is only a special case of the FFT implementation. Here smaller Point IFFT units replace the smaller Point FFTs and the twiddle factors were replaced by their complex conjugates. The FFTs were fit into the FLEX10KE family of devices. Up to 4-point FFTs could be accommodated using a single device for both the radix-2 and radix-4 cases but their 8-point implementations required multiple devices. This reduced the percentage utilization of the logic blocks to a great extent. Matlab models were built for the FFTs for the Cooley-Tukey and Chirp-z algorithms. The model was used for performance evaluations under varying noise conditions. The results for the implementation are in the following results and inferences section.

Results and Conclusions

Table 5.1 Radix–2 Cooley-Tukey implementation with round off errors.

N	Multiliers	LC Count	Delay(ns)	Variance	Precision*	Variance	Precision*
2	0	150	29.7	1.01E-09	15	2.46E-10	16
4	0	609	43.2	2.57E-09	15	1.66E-10	16
8	3	4151	154.4	2.94E-07	10	5.13E-09	13

Table 5.2 Radix-4 Cooley Tukey implementation with round off errors.

N	Multiliers	LC count	Delay(ns)	FFT Variance	FFT Precision*	IFFT Variance	IFFT Precision*
4	0	723	40.9	2.13E-09	14	1.40E-10	16
8	3	4114	145.6	3.02E-07	10	5.53E-09	13

Table 5.3 Radix-2 Cooley Tukey implementation without round off errors.

N	Multiliers	LC count	Delay(ns)	FFT Variance	FFT Precision**	IFFT Variance	IFFT Precision**
2	0	244	49.9	6.20E-10	15	1.46E-10	16
4	0	795	67.1	2.53E-09	14	1.53E-10	16
8	3	5744	169.1	1.68E-07	11	2.53E-09	14

Table 5.4 Radix-4 Cooley-Tukey implementation without round off errors.

N	Multiliers	LC count	Delay(ns)	FFT Variance	FFT Precision*	IFFT Variance	IFFT Precision*
4	0	1381	81.1	1.01E-19	31	7.09E-21	33
8	3	5699	171.1	1.72E-07	11	2.62E-09	14

The results in Tables 1-4 can be argued as follows. The IFFT equation (Equation 3.1) has a factor (1/N) in it. Since this can be achieved by a simple shifting operation, the IFFT has more precision than the FFT in all the Cooley-Tukey implementations. The number of logic blocks required for the implementation of the no round-off errors case is considerably more than when the round-off errors were present. The increase in the amount of hardware has a direct impact on the propagation delays. But the usage of a multi-stage multiplier allows pipelined implementation and this increases the throughput of the entire system. The Baugh-Wooley complex multiplier has about 2160 logic blocks as compared to the 3847 logic blocks of the complex pipelined multiplier. As the length of the FFT increases, the number of the multipliers is also on the rise and that increases the implementation complexity of the system. Attempts to fit a 16-point FFT unit in both the cases (with and without the round-off errors) proved futile. Usage of library-parameterized modules (LPMs) is a possible solution to this problem. But since the purpose of this thesis was to also judge the performance of the systems in terms of latency, the usage of a pipelined multiplier became a necessity. It is common knowledge

that greater the number of stages in a design implementation, lesser is the amount of precision preserved. The Radix-4 FFT implementation has fewer stages of implementation compared to the Radix-2 implementation and hence greater is its precision. The theoretical precision of a fixed-point implementation is given by

$$\Pr ecision = \log_2\left(\sqrt{Variance}\right) \tag{5.1}$$

and is calculated in bits. The tables reveal that the observed precision is very close in almost all cases to the theoretical case.

Power Calculations

The power dissipation in an FPGA is the aggregate sum of all the internal power dissipation and the power dissipation due to the I/O. It is given by the formula

$$\begin{aligned} PowerEstimate &= P_{INT} + P_{IO} \\ &= \{I_{CCINT} * V_{CCINT}\} + \{P_{ACOUT} + P_{DCOUT}\} \\ &= \{(I_{CCstandby} + I_{CCactive}) * V_{CCINT}\} \\ &+ \left\{ \begin{array}{l} (0.5 * OUT * C_{Average} * V_O * f_{MAX} * tog_{IO} * V_{CCIO}) \\ + \left(\sum_{n=1}^{h} h * OUT * \left(\frac{V_{CC}^2}{R_{ioh}}\right)\right) + \left(\sum_{n=1}^{l} l * OUT * \left(\frac{V_{CC}^2}{R_{iol}}\right)\right) \end{array} \right\} \end{aligned} \tag{5.2}$$

Here

h =percentage of high dc outputs

l = percentage of low dc outputs

OUT= number of total output and bi-directional pins in an FPGA device

f_{MAX} = Maximum possible operating frequency

V_{CC} = Supply voltage (could be 2.5, 3.3 or 5.0 volts).

R_{ioh}=Pull-down resistance + Resistance calculated from the slope of the IOH characteristics in the device datasheet.

R_{iol}=Pull-up resistance + Resistance calculated from the slope of the IOL characteristics in the device datasheet.

tog_{IO}= percentage of switching expected in the outputs (usually assumed as 12.5%)

Vo=the output high voltage at a particular value of V_{CC} (3.8,3.3, 2.5V for 5.0, 3.3, 2.5 V V_{CCIO} respectively)

$C_{Average}$= Average capacitance of the family of devices specified in the data sheet.

Usually the higher point FFTs extend over more than one device and an accurate power calculation would take into account power for each of these devices. The power calculations for the biggest design implemented, i.e., the Radix-2 8-point FFT is arrived at as follows. The implementation distributes itself onto three different devices, EPF10K130EFC672-1(a), EPF10K200EBC600-1(b) and EPF10K100EF484-1(c). The power calculation must take into consideration the requirements of the three chips separately.

Table 5.5 Power calculations for Radix-2 8-point FFT.

	O/P pins	Logic blocks	Vcc/ Vccio	K	$I_{CCActive}$ (mA)	Iccsup (mA)	P_{INT} (mW)	P_{dc} (mW)	P_{ac} mW
a	144	1765	2.5V	4.6	0.0593	150	167.099	32.89	768
b	181	1966	2.5V	4.8	0.0689	250	191.178	41.34	966
c	112	2013	2.5V	4.5	0.0662	125	184.265	25.58	598

f_{max}= 58.47 Mhz, V_{CCINT} = 2.5V V_{CCio} = 2.5V

$I_{CCstandby}$ = 7.5 mA R_{ioh}=1400 ohms R_{iol}=1007 ohms

Assuming 50% high o/p and 50 % low op and 1k pull-up and pull-down resistors, we have the total power as 2975.253 mW for a radix-2 8-point implementation with a 2.5V V_{CC}.

Noise Tolerance

The performance of each of the designs was tested under varying noise conditions. The radix-2 and radix-4 perform identical to each other for an equal bus width and number of points (Figures 5.4 and 5.5). Increasing the bus width brought minor improvement into the system but when compared to the floating point implementation (Figure 5.6), the improvement is insignificant. So increasing the bus width to more than 16-bit width in these implementations would not be of much value. The changing of the radix from noise performance is much better in the case of 32-bit bus width.

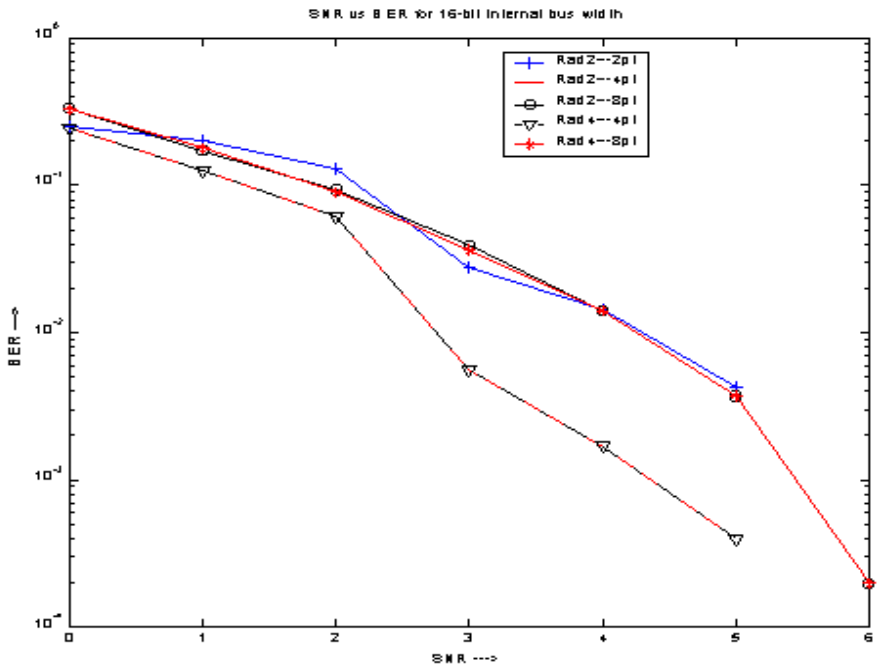

Figure 5.4 BER variations against SNR for an internal bus width of 16.

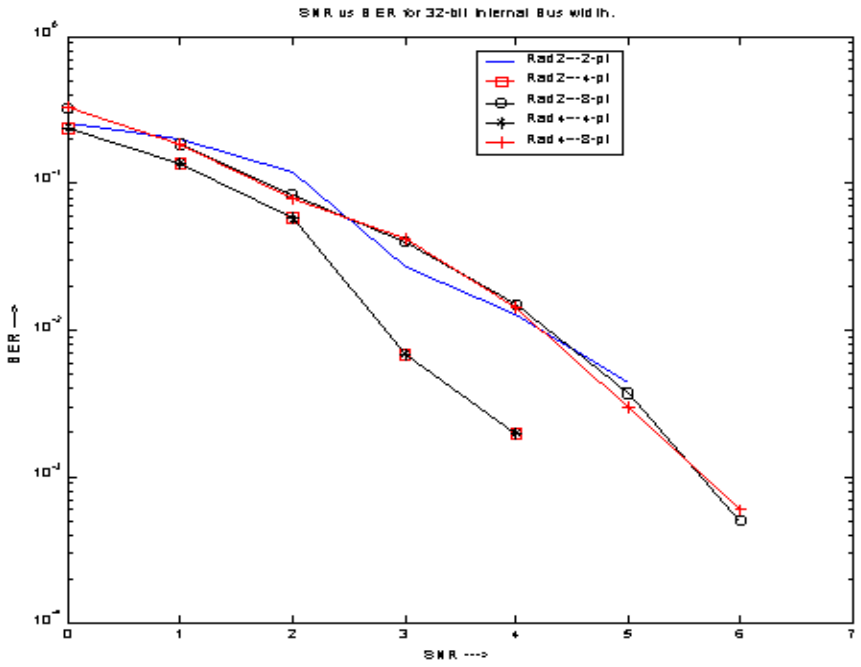

Figure 5.5 BER variations against SNR for an internal bus width of 32.

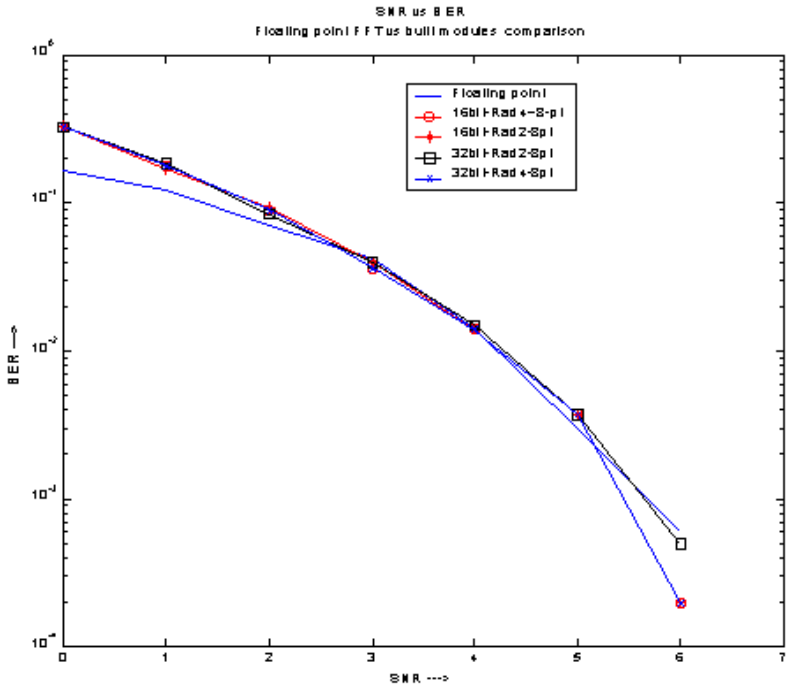

Figure 5.6 BER variations against SNR: Comparison of floating point results with modeled Radix –2 and Radix –4 8 point FFTs with 16- and 32-bit internal bus width.

The Cooley Tukey algorithm was more feasible for implementation into the FPGAs when compared to the Chirp-z implementation due to the excessive amount of multipliers involved in the lower point FFTs. The Chirp-z algorithm usually has the FIR convolution blocks implemented onto CCDs or SAW devices which reduces the number of multipliers to a minimal number. The implementation complexity of the Chirp-z algorithm is 6N complex multiplications compared to the $N*\log_2 N$ for the radix-2 Cooley –Tukey implementation which would mean the break-even point for the algorithms to be sixty-four. So any implementation greater than 64-point FFT would be better off using the Chirp-z algorithm. Since the standard library parameterized modules were not used in the design, even 4-point FFT could not be configured using the Chirp-z algorithm. For the smaller point FFTs, if the data could be clocked out after the ROM coefficient pre-multiplications to the external device (CCDs or SAW devices) for the circular convolution, the implementation complexity in that case would have been only 2N complex multiplications on the FPGA. This would involve converting the digital data into analog form at the output of the FPGA (input of the CCD) and a re-conversion into digital form after a certain time for clocking the input into the FPGAs. This analog implementation of the circular convolution also preserves the precision to a great extent. Though this type of implementation is theoretically feasible and also efficient in terms of precision, its practical implementation would be avoided since it involves the intermediate digital to analog conversion and also the analog to digital conversion. Most applications use longer point FFTs and so the Chirp-z algorithm is a much better choice in those cases. The Cooley Tukey algorithm implementation is also highly desirable for the reduction of complexity associated with it.

Directions of Future Work

This thesis used fast multipliers that were designed for being able to provide a comparison between the general multipliers and the multi-stage fast multipliers. This restricted the maximum possible FFT length to eight. The simulation software provides multiplier modules that are highly optimized for each family of the FPGA devices. Usage of such modules may reduce the power consumption and latency. Since these multipliers are usually much smaller, a higher point FFT implementation using greater number of such multipliers is possible and further research can be done with these multiplier modules.

The Chirp-z algorithm provides for more efficient implementations with much lesser hardware. More work can be done using such an implementation. Also the amount of precision loss in such a design would be independent of the length of the FFT. The increased spectral resolution provided by this algorithm with much lesser hardware would be an incentive to pursue the study of this algorithm implementations on FPGAs.

APPENDIX A
16-BIT COOLEY-TUKEY IMPLEMENTATION

```
%%%%%%%%%  bwcell1.m  %%%%%%%%
function [s]=bwcell1(a,b)
% Cell 1 of the Baugh Wooley Multiplier
s=a & b;
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  bwcell2.m  %%%%%%%%
function [S,cout]=bwcell2(a,b,Sin,Cin)
% Cell 2 of the Baugh Wooley Multiplier
d=a & b;
[S,cout]=fulladder(d,Sin,Cin);
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  bwcell3.m  %%%%%%%%
function [s]=bwcell3(a,b)
% Cell 3 of the Baugh Wooley Multiplier
s=a & ~b;
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  bwcell4.m  %%%%%%%%
function [s,cout]=bwcell4(a,b,Sin,Cin)
% Cell 4 of the Baugh Wooley Multiplier
c=~a & b;
[s,cout]=fulladder(c,Sin,Cin);
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  bwcell5.m  %%%%%%%%
function [s,cout]=bwcell5(x,y)
% Cell 5 of the Baugh Wooley Multiplier
b=x & y;
c=~x;
d=~y;
[s,cout]=fulladder(b,c,d);
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  bwm16a.m  %%%%%%%%
function [p]=bwm16a(x,y)
% This function multiplies two 16-bit signed numbers and gives a 16-bit
signed number
% as output.
```

```
% The logic implemented is the Baugh Wooley Multiplier

t=zeros(16);
c=zeros(16);

% First row
for i=16:-1:2
    t(1,i) = bwcell1( x(i),y(16) );
end
t(1,1)=bwcell3(x(1),y(16));

% row 2
for i=16:-1:2
    [t(2,i),c(2,i)]=bwcell2(x(i),y(15),t(1,i-1),0);
end
t(2,1)=bwcell3(x(1),y(15));

% Row three to row 15

for j=3:15
    for i=16:-1:2
        [t(j,i),c(j,i)]=bwcell2(x(i),y(17-j),t(j-1,i-1),c(j-1,i));
    end
    t(j,1)=bwcell3(x(1),y(17-j));
end
% row 16

for i=16:-1:2
    [t(16,i),c(16,i)]=bwcell4(x(i),y(1),t(15,i-1),c(15,i));
end

[t(16,1),c(16,1)]=bwcell5(x(1),y(1));

% Last Row
[temp,cout(16)]=fulladder(x(1),y(1),t(16,16));
for i=16:-1:2
    [p(i),cout(i-1)]=fulladder(c(16,i),t(16,i-1),cout(i));
end
[p(1),nc]=fulladder(1,c(16,1),cout(1));

clear c
clear t
clear cout
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  com.m  %%%%%%%%
function [b]=com(a)

% This function calculates the twos complement of a number
tmp1=~a;
tmp2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1];
[b,tmp]=ad16c(tmp1,tmp2);
clear tmp1
clear tmp2
clear a
```

```
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  cplxmul16.m  %%%%%%%%
function [pr,pi]=cplxmul16(xr,xi,yr,yi)

% This function multiplies two 16-bit signed complex inputs and gives a
16-bit signed
% complex output.

t0=bwm16a(xr,yr);
t1=bwm16a(xr,yi);
t2=bwm16a(xi,yr);
t3=bwm16a(xi,yi);


pr=sub16(t0,t3);
pi=add16(t2,t1);
return
%%%%%%%%%  END  %%%%%%%%


%%%%%%%%%  fulladder.m  %%%%%%%%
function [s,cout]=fulladder(a,b,c)
tmp1=(a & ~b) | (~a & b);
s=(tmp1 & ~c) | (~tmp1 & c);
tmp2= a & b;
tmp3= b & c;
tmp4= c & a;
cout= (tmp2 | tmp3) | tmp4;
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  getformat.m  %%%%%%%%
function [format]=getformat(a)
% function gets the input vector a into a format that allows only real
transmissions
% to be possible in the time domain data.

lena=length(a);

format=[real(a(1)); a(2:lena); imag(a(1)); conj(a(2:lena))];
return
%%%%%%%%%  END  %%%%%%%%


%%%%%%%%%  hexa2bin.m  %%%%%%%%
% Function returns the string(NOT ARRAY) containing the binary
equivalent of the input hexadecimal number
% for example
% if b='f74b'
% then hexa2bin(b) would be equal to '1111011101001011'
% and if b='074b'
% then hexa2bin(b) would be equal to '11101001011'
% NOTE THAT THE INITIAL ZEROS ARE NOT PRESENT IN THE RESULT
function [s]=hexa2bin(a)
s=dec2bin(hex2dec(a));
```

```
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  hex2decim.m  %%%%%%%%
intebits=6;
fracbits=16-intebits;
hexad=1010111100000000;
a=[0];
for i=1:16
    digi=mod(hexad,10);
    hexad=floor(hexad/10);
    a=[digi a];
end
a=a(1:16);
decim=0;
factor=2^(-fracbits);
for i=16:-1:2
   decim=decim+a(i)*factor;
   factor=factor*2;
end
decim
if a(1)==1
   decim=decim-2^(intebits-1);
end
decim
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  iterative.m  %%%%%%%%
%  This program does the performance analysis of the program for FFT in
terms
%  of the errors and its types, execution times, etc, over a number of
iterations.

E_mean=[];
E_var=[];
mag_err=[];
sign_err=[];
magsign_err=[];
E_no=[];
ratio_vector=[];
N=8; % index of the fft to be analysed

for i=1:1000
    % i is the iteration index
   % i
    % Initialization of variables for each iteration
    Error_mean=0;
    Error_variance=0;
    Avg_no_of_magnitude_errors=0;
    Avg_no_of_sign_errors=0;
    Avg_no_of_mag_and_sign_errors=0;

    % preparation of inputs

    inr=rand(1,N);
    ini=rand(1,N);
```

```
    cplx=inr+j*ini;

    cplxfft=fft(cplx);
    test=ramasfft8(cplx);

    right=0;
    signerr=0;
    magerr=0;
    magsignerr=0;

    Error_mean1=sqrt(mean((test-reshape(cplxfft,[N,1])).^2));
    E_mean=[E_mean Error_mean1];

end

Error_mean=mean(abs(E_mean))
Error_variance=var(abs(E_mean))
%%%%%%%%%  END


 %%%%%%%%%  rad2ct16.m  %%%%%%%%
function
[or0,oi0,or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi
8,or9,oi9,or10,oi10,or11,oi11,or12,oi12,or13,oi13,or14,oi14,or15,oi15]=
rad2ct16(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii
7,ir8,ii8,ir9,ii9,ir10,ii10,ir11,ii11,ir12,ii12,ir13,ii13,ir14,ii14,ir1
5,ii15)
% this function calculates the 8-point radix-2 cooley tukey fft
transform
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 2)  & (N2 = 8)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1)
%    k2,n2 => (0,1,2,...7)
%

% RESULTS :
%   EXECUTION TIME :
%   FRACTIONAL PRECISION :
%   ERROR OVER A RUN OF 100 TIMES :
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY :
%       SIGN ERRORS ONLY  :
%       BOTH MAGNITUDE AND SIGN :


[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3,tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad2c
t8(
ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6,ir8,ii8,ir10,ii10,ir12,ii12,ir14,ii14);
[tr8,ti8,tr9,ti9,tr10,ti10,tr11,ti11,tr12,ti12,tr13,ti13,tr14,ti14,tr15
,ti15]=rad2ct8(ir1,ii1,ir3,ii3,ir5,ii5,ir7,ii7,ir9,ii9,ir11,ii11,ir13,i
i13,ir15,ii15);
```

```
tra0=shrega(tr0);
tia0=shrega(ti0);
tra1=shrega(tr1);
tia1=shrega(ti1);
tra2=shrega(tr2);
tia2=shrega(ti2);
tra3=shrega(tr3);
tia3=shrega(ti3);
tra4=shrega(tr4);
tia4=shrega(ti4);
tra5=shrega(tr5);
tia5=shrega(ti5);
tra6=shrega(tr6);
tia6=shrega(ti6);
tra7=shrega(tr7);
tia7=shrega(ti7);
tra8=shrega(tr8);
tia8=shrega(ti8);

w16_r1=[0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0];
w16_i1=[1 1 1 0 0 1 1 1 1 0 0 0 0 0 1 1];
w16_r2=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w16_i2=[1 1 0 1 1 1 0 0 0 0 1 0 0 0 0 0];
w16_r3=[0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 0];
w16_i3=[1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 0];
w16_r4=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w16_i4=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1];

[tra9,tia9]=cplxmul16(tr9,ti9,w16_r1,w16_i1);
[tra10,tia10]=cplxmul16(tr10,ti10,w16_r2,w16_i2);
[tra11,tia11]=cplxmul16(tr11,ti11,w16_r3,w16_i3);
[tra12,tia12]=cplxmul16(tr12,ti12,w16_r4,w16_i4);
[tra13,tia13]=cplxmul16(tr13,ti13,w16_i1,w16_i3);
[tra14,tia14]=cplxmul16(tr14,ti14,w16_i2,w16_i2);
[tra15,tia15]=cplxmul16(tr15,ti15,w16_i3,w16_i1);


[or0,oi0,or8,oi8]=rad2ct2(tra0,tia0,tra8,tia8);
[or1,oi1,or9,oi9]=rad2ct2(tra1,tia1,tra9,tia9);
[or2,oi2,or10,oi10]=rad2ct2(tra2,tia2,tra10,tia10);
[or3,oi3,or11,oi11]=rad2ct2(tra3,tia3,tra11,tia11);
[or4,oi4,or12,oi12]=rad2ct2(tra4,tia4,tra12,tia12);
[or5,oi5,or13,oi13]=rad2ct2(tra5,tia5,tra13,tia13);
[or6,oi6,or14,oi14]=rad2ct2(tra6,tia6,tra14,tia14);
[or7,oi7,or15,oi15]=rad2ct2(tra7,tia7,tra15,tia15);
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad2ct16a.m  %%%%%%%%
function
[or0,oi0,or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi
8,or9,oi9,or10,oi10,or11,oi11,or12,oi12,or13,oi13,or14,oi14,or15,oi15]=
rad2ct16a(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,i
i7,ir8,ii8,ir9,ii9,ir10,ii10,ir11,ii11,ir12,ii12,ir13,ii13,ir14,ii14,ir
15,ii15)
% this function calculates the 8-point radix-2 cooley tukey fft
transform
```

```
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 4)  & (N2 = 4)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1,2,3)
%    k2,n2 => (0,1,2,3)

% RESULTS :
%   EXECUTION TIME :
%   FRACTIONAL PRECISION :
%   ERROR OVER A RUN OF 100 TIMES :
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY :
%       SIGN ERRORS ONLY  :
%       BOTH MAGNITUDE AND SIGN :

[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad2ct4(
ir0,ii0,ir4,ii4,ir8,ii8,ir12,ii12);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad2ct4(ir1,ii1,ir5,ii5,ir9,ii9,ir13,
ii13);
[tr8,ti8,tr9,ti9,tr10,ti10,tr11,ti11]=rad2ct4(
ir2,ii2,ir6,ii6,ir10,ii10,ir14,ii14);
[tr12,ti12,tr13,ti13,tr14,ti14,tr15,ti15]=rad2ct4(ir3,ii3,ir7,ii7,ir11,
ii11,ir15,ii15);

tra0=shrega(tr0);
tia0=shrega(ti0);
tra1=shrega(tr1);
tia1=shrega(ti1);
tra2=shrega(tr2);
tia2=shrega(ti2);
tra3=shrega(tr3);
tia3=shrega(ti3);
tra4=shrega(tr4);
tia4=shrega(ti4);
tra8=shrega(tr8);
tia8=shrega(ti8);
tra12=shrega(tr12);
tia12=shrega(ti12);

w16_r1=[0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0];
w16_i1=[1 1 1 0 0 1 1 1 1 0 0 0 0 0 1 1];
w16_r2=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w16_i2=[1 1 0 1 1 1 0 0 0 0 1 0 0 0 0 0];
w16_r3=[0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 0];
w16_i3=[1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 0];
w16_r4=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w16_i4=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1];

[tra5,tia5]=cplxmul16(tr5,ti5,w16_r1,w16_i1);
[tra6,tia6]=cplxmul16(tr6,ti6,w16_r2,w16_i2);
[tra7,tia7]=cplxmul16(tr7,ti7,w16_r3,w16_i3);
[tra9,tia9]=cplxmul16(tr9,ti9,w16_r2,w16_i2);
[tra10,tia10]=cplxmul16(tr10,ti10,w16_r4,w16_i4);
```

```
[tra11,tia11]=cplxmul16(tr11,ti11,w16_i2,w16_i2);
[tra13,tia13]=cplxmul16(tr13,ti13,w16_r3,w16_i3);
[tra14,tia14]=cplxmul16(tr14,ti14,w16_i2,w16_i2);
[tra15,tia15]=cplxmul16(tr15,ti15,w16_i3,w16_r3);

[or0,oi0,or4,oi4,or8,oi8,or12,oi12]=rad2ct4(tra0,tia0,tra4,tia4,tra8,ti
a8,tra12,tia12);
[or1,oi1,or5,oi5,or9,oi9,or13,oi13]=rad2ct4(tra1,tia1,tra5,tia5,tra9,ti
a9,tra13,tia13);
[or2,oi2,or6,oi6,or10,oi10,or14,oi14]=rad2ct4(tra2,tia2,tra6,tia6,tra10
,tia10,tra14,tia14);
[or3,oi3,or7,oi7,or11,oi11,or15,oi15]=rad2ct4(tra3,tia3,tra7,tia7,tra11
,tia11,tra15,tia15);

return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad2ct2.m  %%%%%%%%
function [Or1,Oi1,Or2,Oi2]=rad2ct2(ir1,ii1,ir2,ii2)
% This function calculates the 2 point FFT of the two 16-bit inputs and
gives
% two 16 bit outputs.
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    out1=in1+in2;
%    out2=in1-in2;

% RESULTS :
%   EXECUTION TIME : 27.4 ns(DELAY in MAXPLUS)  WHEN FITTED INTO
"EP1K10FC256-1" DEVICE OF ACEX 1K FAMILY
%   FRACTIONAL PRECISION : 1 less than input precision
%   ERROR OVER A RUN OF 100 TIMES :  mean := -6.8272e-05 - 6.0807e-05i
and  variance := 4.0445e-09
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY : 0
%       SIGN ERRORS ONLY  :   0.02
%       BOTH MAGNITUDE AND SIGN : 0

Or1=add16(ir1,ir2);
Or2=sub16(ir1,ir2);
Oi1=add16(ii1,ii2);
Oi2=sub16(ii1,ii2);
return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad2ct2ifft.m  %%%%%%%%
function [Or1,Oi1,Or2,Oi2]=rad2ct2ifft(ir1,ii1,ir2,ii2)
% This function calculates the 2 point FFT of the two 16-bit inputs and
gives
% two 16 bit outputs.
Or1=add16(ir1,ir2);
Or2=sub16(ir1,ir2);
Oi1=add16(ii1,ii2);
Oi2=sub16(ii1,ii2);
return
```

```
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad2ct32.m  %%%%%%%%
function
[or0,oi0,or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi
8,or9,oi9,or10,oi10,or11,oi11,or12,oi12,or13,oi13,or14,oi14,or15,oi15,o
r16,oi16,or17,oi17,or18,oi18,or19,oi19,or20,oi20,or21,oi21,or22,oi22,or
23,oi23,or24,oi24,or25,oi25,or26,oi26,or27,oi27,or28,oi28,or29,oi29,or3
0,oi30,or31,oi31]=rad2ct32(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,
ii5,ir6,ii6,ir7,ii7,ir8,ii8,ir9,ii9,ir10,ii10,ir11,ii11,ir12,ii12,ir13,
ii13,ir14,ii14,ir15,ii15,ir16,ii16,ir17,ii17,ir18,ii18,ir19,ii19,ir20,i
i20,ir21,ii21,ir22,ii22,ir23,ii23,ir24,ii24,ir25,ii25,ir26,ii26,ir27,ii
27,ir28,ii28,ir29,ii29,ir30,ii30,ir31,ii31)
%Function to calculate the 16 point radix 4 fft
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 4)  & (N2 = 4)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1,2,3)
%    k2,n2 => (0,1,2,3)

% RESULTS :
%   EXECUTION TIME :
%   FRACTIONAL PRECISION :
%   ERROR OVER A RUN OF 100 TIMES :
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY :
%       SIGN ERRORS ONLY  :
%       BOTH MAGNITUDE AND SIGN :


[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3,tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad4c
t8(ir0,ii0,ir4,ii4,ir8,ii8,ir12,ii12,ir16,ii16,ir20,ii20,ir24,ii24,ir28
,ii28);
[tr8,ti8,tr9,ti9,tr10,ti10,tr11,ti11,tr12,ti12,tr13,ti13,tr14,ti14,tr15
,ti15]=rad4ct8(ir1,ii1,ir5,ii5,ir9,ii9,ir13,ii13,ir17,ii17,ir21,ii21,ir
25,ii25,ir29,ii29);
[tr16,ti16,tr17,ti17,tr18,ti18,tr19,ti19,tr20,ti20,tr21,ti21,tr22,ti22,
tr23,ti23]=rad4ct8(ir2,ii2,ir6,ii6,ir10,ii10,ir14,ii14,ir18,ii18,ir22,i
i22,ir26,ii26,ir30,ii30);
[tr24,ti24,tr25,ti25,tr26,ti26,tr27,ti27,tr28,ti28,tr29,ti29,tr30,ti30,
tr31,ti31]=rad4ct8(ir3,ii3,ir7,ii7,ir11,ii11,ir15,ii15,ir19,ii19,ir23,i
i23,ir27,ii27,ir31,ii31);

tra0=shrega(tr0);
tia0=shrega(ti0);
tra1=shrega(tr1);
tia1=shrega(ti1);
tra2=shrega(tr2);
tia2=shrega(ti2);
tra3=shrega(tr3);
tia3=shrega(ti3);
tra4=shrega(tr4);
tia4=shrega(ti4);
```

```
tra5=shrega(tr5);
tia5=shrega(ti5);
tra6=shrega(tr6);
tia6=shrega(ti6);
tra7=shrega(tr7);
tia7=shrega(ti7);
tra8=shrega(tr8);
tia8=shrega(ti8);
tra16=shrega(tr16);
tia16=shrega(ti16);
tra24=shrega(tr24);
tia24=shrega(ti24);

w32_1r=[0 0 1 1 1 1 1 0 1 1 0 0 0 1 0 1];
w32_1i=[1 1 1 1 0 0 1 1 1 0 0 0 0 1 0 0];
w32_2r=[0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0];
w32_2i=[1 1 1 0 0 1 1 1 1 0 0 0 0 0 1 1];
w32_3r=[0 0 1 1 0 1 0 1 0 0 1 1 0 1 1 0];
w32_3i=[1 1 0 1 1 1 0 0 0 1 1 1 0 0 1 0];
w32_4r=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w32_4i=[1 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0];
w32_5r=[0 0 1 0 0 0 1 1 1 0 0 0 1 1 1 0];
w32_5i=[1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 0];
w32_6r=[0 0 0 1 0 1 1 1 1 0 0 0 1 1 1 0];
w32_6i=[1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 0];
w32_7r=[0 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1];
w32_7i=[1 1 0 0 0 0 0 1 0 0 1 1 1 0 1 1];
w32_8r=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w32_8i=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1];


[tra9,tia9]=cplxmul16(tr9,ti9,w32_1r,w32_1i);
[tra10,tia10]=cplxmul16(tr10,ti10,w32_2r,w32_2i);
[tra11,tia11]=cplxmul16(tr11,ti11,w32_3r,w32_3i);
[tra12,tia12]=cplxmul16(tr12,ti12,w32_2r,w32_2i);
[tra13,tia13]=cplxmul16(tr13,ti13,w32_4r,w32_4i);
[tra14,tia14]=cplxmul16(tr14,ti14,w32_6r,w32_6i);
[tra15,tia15]=cplxmul16(tr15,ti15,w32_7r,w32_7i);
[tra17,tia17]=cplxmul16(tr17,ti17,w32_2r,w32_2i);
[tra18,tia18]=cplxmul16(tr18,ti18,w32_4r,w32_4i);
[tra19,tia19]=cplxmul16(tr19,ti19,w32_6r,w32_6i);
[tra20,tia20]=cplxmul16(tr20,ti20,w32_8r,w32_8i);
[tra21,tia21]=cplxmul16(tr21,ti21,w32_2i,w32_6i);
[tra22,tia22]=cplxmul16(tr22,ti22,w32_4i,w32_4i);
[tra23,tia23]=cplxmul16(tr23,ti23,w32_6i,w32_2i);
[tra25,tia25]=cplxmul16(tr25,ti25,w32_3r,w32_3i);
[tra26,tia26]=cplxmul16(tr26,ti26,w32_6r,w32_6i);
[tra27,tia27]=cplxmul16(tr27,ti27,w32_1i,w32_7i);
[tra28,tia28]=cplxmul16(tr28,ti28,w32_4i,w32_4i);
[tra29,tia29]=cplxmul16(tr29,ti29,w32_7i,w32_1i);
[tra30,tia30]=cplxmul16(tr30,ti30,w32_6i,w32_6r);
[tra31,tia31]=cplxmul16(tr31,ti31,w32_3i,w32_3r);


[or0,oi0,or8,oi8,or16,oi16,or24,oi24]=rad2ct2(tra0,tia0,tra8,tia8,tra16
,tia16,tra24,tia24);
```

```
[or1,oi1,or9,oi9,or17,oi17,or25,oi25]=rad2ct2(tra1,tia1,tra9,tia9,tra17
,tia17,tra25,tia25);
[or2,oi2,or10,oi10,or18,oi18,or26,oi26]=rad2ct2(tra2,tia2,tra10,tia10,t
ra18,tia18,tra26,tia26);
[or3,oi3,or11,oi11,or19,oi19,or27,oi27]=rad2ct2(tra3,tia3,tra11,tia11,t
ra19,tia19,tra27,tia27);
[or4,oi4,or12,oi12,or20,oi20,or28,oi28]=rad2ct2(tra4,tia4,tra12,tia12,t
ra20,tia20,tra28,tia28);
[or5,oi5,or13,oi13,or21,oi21,or29,oi29]=rad2ct2(tra5,tia5,tra13,tia13,t
ra21,tia21,tra29,tia29);
[or6,oi6,or14,oi14,or22,oi22,or30,oi30]=rad2ct2(tra6,tia6,tra14,tia14,t
ra22,tia22,tra30,tia30);
[or7,oi7,or15,oi15,or23,oi23,or31,oi31]=rad2ct2(tra7,tia7,tra15,tia15,t
ra23,tia23,tra31,tia31);

return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad2ct4.m  %%%%%%%%
function
[or1,oi1,or2,oi2,or3,oi3,or4,oi4]=rad2ct4(ir1,ii1,ir2,ii2,ir3,ii3,ir4,i
i4)
%  This function computes the Radix 2 4-point Cooley Tukey FFT given
the four points
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 4)  & (N2 = 4)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1,2,3)
%    k2,n2 => (0,1,2,3)

% RESULTS :
%   EXECUTION TIME : 40.0 ns Delay when fitted into  "EP1k100FC484-
1"device of ACEX 1K family
%   FRACTIONAL PRECISION : 2 bits less than input precision
%   ERROR OVER A RUN OF 100 TIMES :  -1.1922e-04 -1.2181e-04i with a
variance of 0.3048
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY : 0
%       SIGN ERRORS ONLY  :  0.03
%       BOTH MAGNITUDE AND SIGN :  3.97


[tr1,ti1,tr2,ti2]=rad2ct2(ir1,ii1,ir3,ii3);
[tr3,ti3,tr4,ti4]=rad2ct2(ir2,ii2,ir4,ii4);

[or1,oi1,or3,oi3]=rad2ct2(tr1,ti1,tr3,ti3);
[or2,oi2,or4,oi4]=rad2ct2(tr2,ti2,ti4,com(tr4));

return

%%%%%%%%%  END  %%%%%%%%%%%%%%%%%  rad2ct4ifft.m  %%%%%%%%
```

```
function
[or1,oi1,or2,oi2,or3,oi3,or4,oi4]=rad2ct4ifft(ir1,ii1,ir2,ii2,ir3,ii3,i
r4,ii4)
%  This function computes the Radix 2 4-point Cooley Tukey FFT given
the four points

[tr1,ti1,tr2,ti2]=rad2ct2(ir1,ii1,ir3,ii3);
[tr3,ti3,tr4,ti4]=rad2ct2(ir2,ii2,ir4,ii4);

[or1,oi1,or3,oi3]=rad2ct2(tr1,ti1,tr3,ti3);
[or2,oi2,or4,oi4]=rad2ct2(tr2,ti2,com(ti4),tr4);

return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad2ct8.m  %%%%%%%%
function
[or0,oi0,or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7]=rad2c
t8(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
% this function calculates the 8-point radix-2 cooley tukey fft
transform
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 4)  & (N2 = 4)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1,2,3)
%    k2,n2 => (0,1,2,3)

% RESULTS :
%   EXECUTION TIME : 106.8ns when fitted into "EP1K30QC208-
1","EP1K50FC484-1","EP1K100FC484-1" Decives of ACEX1K family
%   FRACTIONAL PRECISION : 5 fractional bits less than the input
%   ERROR OVER A RUN OF 100 TIMES :  -0.0010 - 0.0010 i
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY : 0
%       SIGN ERRORS ONLY  :    1.26
%       BOTH MAGNITUDE AND SIGN :  1.17


[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad2ct4(
ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad2ct4(ir1,ii1,ir3,ii3,ir5,ii5,ir7,i
i7);




tra0=shrega(tr0);
tia0=shrega(ti0);
tra1=shrega(tr1);
tia1=shrega(ti1);
tra2=shrega(tr2);
tia2=shrega(ti2);
tra3=shrega(tr3);
```

```
tia3=shrega(ti3);
tra4=shrega(tr4);
tia4=shrega(ti4);


w8_r1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w8_i1=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
w8_r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w8_i2=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w8_r3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
w8_i3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];

[tra5,tia5]=cplxmul16(tr5,ti5,w8_r1,w8_i1);

[tra6,tia6]=cplxmul16(tr6,ti6,w8_r2,w8_i2);

[tra7,tia7]=cplxmul16(tr7,ti7,w8_r3,w8_i3);

[or0,oi0,or4,oi4]=rad2ct2(tra0,tia0,tra4,tia4);

[or1,oi1,or5,oi5]=rad2ct2(tra1,tia1,tra5,tia5);

[or2,oi2,or6,oi6]=rad2ct2(tra2,tia2,tra6,tia6);

[or3,oi3,or7,oi7]=rad2ct2(tra3,tia3,tra7,tia7);

return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad2ct8ifft.m  %%%%%%%%
function
[or0,oi0,or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7]=rad2c
t8ifft(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
% this function calculates the 8-point radix-2 cooley tukey ifft
transform

[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad2ct4ifft(
ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad2ct4ifft(
ir1,ii1,ir3,ii3,ir5,ii5,ir7,ii7);

tra0=shrega(tr0);
tia0=shrega(ti0);
tra1=shrega(tr1);
tia1=shrega(ti1);
tra2=shrega(tr2);
tia2=shrega(ti2);
tra3=shrega(tr3);
tia3=shrega(ti3);
tra4=shrega(tr4);
tia4=shrega(ti4);

w8_r1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w8_i1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 0];
w8_r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w8_i2=[0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1];
w8_r3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
```

```
w8_i3=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 0];


[tra5,tia5]=cplxmul16(tr5,ti5,w8_r1,w8_i1);
[tra6,tia6]=cplxmul16(tr6,ti6,w8_r2,w8_i2);
[tra7,tia7]=cplxmul16(tr7,ti7,w8_r3,w8_i3);

[or0,oi0,or4,oi4]=rad2ct2ifft(tra0,tia0,tra4,tia4);
[or1,oi1,or5,oi5]=rad2ct2ifft(tra1,tia1,tra5,tia5);
[or2,oi2,or6,oi6]=rad2ct2ifft(tra2,tia2,tra6,tia6);
[or3,oi3,or7,oi7]=rad2ct2ifft(tra3,tia3,tra7,tia7);

return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad4ct4.m  %%%%%%%%
function
[or0,oi0,or1,oi1,or2,oi2,or3,oi3]=rad4ct4(ir0,ii0,ir1,ii1,ir2,ii2,ir3,i
i3)
% This function calculates the raxix 4 4-point Cooley Tukey FFT
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 4)  & (N2 = 4)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1,2,3)
%    k2,n2 => (0,1,2,3)

% RESULTS :
%   EXECUTION TIME : 44.1ns when fitted into "EP1K100FC484-1" device of
ACEX1K family
%   FRACTIONAL PRECISION : 2 less than Input
%   ERROR OVER A RUN OF 100 TIMES :  -1.1553e-04 - 1.1447e-04
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY : 0
%       SIGN ERRORS ONLY  : 0.0450
%       BOTH MAGNITUDE AND SIGN : 3.955


t0=add16(ir0,ir1);
t1=add16(ir2,ir3);
t2=add16(ii0,ii1);
t3=add16(ii2,ii3);

t4=sub16(ir0,ir2);
t5=sub16(ii0,ii2);
t6=sub16(ii1,ii3);
t7=sub16(ir1,ir3);

t8=add16(ir0,ir2);
t9=add16(ir1,ir3);
t10=add16(ii0,ii2);
t11=add16(ii1,ii3);

oi1=sub16(t5,t7);
```

```
oi2=sub16(t10,t11);
or2=sub16(t8,t9);
or3=sub16(t4,t6);

or0=add16(t0,t1);
oi0=add16(t2,t3);
or1=add16(t4,t6);
oi3=add16(t5,t7);

return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad4ct4ifft.m  %%%%%%%%
function
[or0,oi0,or3,oi3,or2,oi2,or1,oi1]=rad4ct4(ir0,ii0,ir1,ii1,ir2,ii2,ir3,i
i3)
% This function calculates the raxix 4 4-point Cooley Tukey FFT
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 4)  & (N2 = 4)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1,2,3)
%    k2,n2 => (0,1,2,3)

% RESULTS :
%   EXECUTION TIME : 44.1ns when fitted into "EP1K100FC484-1" device of
ACEX1K family
%   FRACTIONAL PRECISION : 2 less than Input
%   ERROR OVER A RUN OF 100 TIMES :  -1.1553e-04 - 1.1447e-04
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY : 0
%       SIGN ERRORS ONLY  : 0.0450
%       BOTH MAGNITUDE AND SIGN : 3.955


t0=add16(ir0,ir1);
t1=add16(ir2,ir3);
t2=add16(ii0,ii1);
t3=add16(ii2,ii3);

t4=sub16(ir0,ir2);
t5=sub16(ii0,ii2);
t6=sub16(ii1,ii3);
t7=sub16(ir1,ir3);

t8=add16(ir0,ir2);
t9=add16(ir1,ir3);
t10=add16(ii0,ii2);
t11=add16(ii1,ii3);

oi1=sub16(t5,t7);
oi2=sub16(t10,t11);
or2=sub16(t8,t9);
or3=sub16(t4,t6);
```

```
or0=add16(t0,t1);
oi0=add16(t2,t3);
or1=add16(t4,t6);
oi3=add16(t5,t7);

return
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  rad4ct8.m  %%%%%%%%
function
[or0,oi0,or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7]=rad4c
t8(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
%
% GOVERNING EQUATIONS AND ANALYSIS
%
%    N = 16 ; =>  (N1 = 4)  & (N2 = 4)
%    k = k1 + 2 * k2;
%    n = 8 * n1 + n2;
%    k1,n1 => (0,1,2,3)
%    k2,n2 => (0,1,2,3)

% RESULTS :
%   EXECUTION TIME : 124.0 ns  when fitted into devices of the ACEX 1K
family
%   FRACTIONAL PRECISION : 5 fractional bits less than the input
%   ERROR OVER A RUN OF 100 TIMES :   -9.6769e-04 - 9.6433e-04i with a
variance of 0.1246
%   AVERAGE NUMBER OF ERRORS :
%   ERROR TYPES--->
%       MAGNITUDE ERRORS ONLY : 0
%       SIGN ERRORS ONLY  :     1.12
%       BOTH MAGNITUDE AND SIGN :  1.35


% N1=4 PoiNT FFT

[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad4ct4(ir0,ii0,ir2,ii2,ir4,ii4,ir6,i
i6);

[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad4ct4(ir1,ii1,ir3,ii3,ir5,ii5,ir7,i
i7);

% MultiPLiCAtiON PHASE

  w8_1r=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
  w8_1i=[1 1 0 1 0 0 1 0 1 0 1 1 1 1 1 1];
  w8_2r=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
  w8_2i=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
  w8_3r=[1 1 0 1 0 0 1 0 1 0 1 1 1 1 1 1];
  w8_3i=[1 1 0 1 0 0 1 0 1 0 1 1 1 1 1 1];

 tr0m=shrega(tr0);
 tr1m=shrega(tr1);
 tr2m=shrega(tr2);
 tr3m=shrega(tr3);
 tr4m=shrega(tr4);
```

```
 ti0m=shrega(ti0);
 ti1m=shrega(ti1);
 ti2m=shrega(ti2);
 ti3m=shrega(ti3);
 ti4m=shrega(ti4);


    [tr5m,ti5m]=cplxmul16(tr5,ti5,w8_1r,w8_1i);
    [tr6m,ti6m]=cplxmul16(tr6,ti6,w8_2r,w8_2i);
    [tr7m,ti7m]=cplxmul16(tr7,ti7,w8_3r,w8_3i);


% N2=2 PoiNT FFT
    [or0,oi0,or4,oi4]=rad2ct2(tr0m,ti0m,tr4m,ti4m);
    [or1,oi1,or5,oi5]=rad2ct2(tr1m,ti1m,tr5m,ti5m);
    [or2,oi2,or6,oi6]=rad2ct2(tr2m,ti2m,tr6m,ti6m);
    [or3,oi3,or7,oi7]=rad2ct2(tr3m,ti3m,tr7m,ti7m);

return
%%%%%%%%  END  %%%%%%%%

%%%%%%%%  ramasfft.m  %%%%%%%%
function [b]=ramasfft(a)
% Program to interface my fft engine in place of the standard FFT
function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

N=8;       %%% ! ! !  MODIFY ! ! !   %%%%%
shift=8;   %%% ! ! !  MODIFY ! ! !   %%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    or=zeros(N,1);
    oi=or;
    a_re=real(a);
    a_im=imag(a);

    a_r=zeros(N,16);
    a_i=zeros(N,16);
    o_r=a_r;
    o_i=a_i;

    for i=1:N
        a_r(i,:)=convert(a_re(i),2);
        a_i(i,:)=convert(a_im(i),2);
    end
%  2-point
[or1,oi1,or2,oi2]=rad2ct2(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:));
  %  4-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4]=rad2ct4(a_r(1,:),a_i(1,:),a_r(2,:),a_
i(2,:),a_r(3,:),a_i(3,:),a_r(4,:),a_i(4,:));

[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8]=rad2c
t8(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:),a_r(3,:),a_i(3,:),a_r(4,:),a_i(4
,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7,:),a_i(7,:),a_r(8,:),a_i(
8,:));
```

```
%  16-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8,or9,oi
9,or10,oi10,or11,oi11,or12,oi12,or13,oi13,or14,oi14,or15,oi15,or16,oi16
]=rad2ct16(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:),a_r(3,:),a_i(3,:),a_r(4,
:),a_i(4,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7,:),a_i(7,:),a_r(8
,:),a_i(8,:),a_r(9,:),a_i(9,:),a_r(10,:),a_i(10,:),a_r(11,:),a_i(11,:),
a_r(12,:),a_i(12,:),a_r(13,:),a_i(13,:),a_r(14,:),a_i(14,:),a_r(15,:),a
_i(15,:),a_r(16,:),a_i(16,:));
%  32-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8,or9,oi
9,or10,oi10,or11,oi11,or12,oi12,or13,oi13,or14,oi14,or15,oi15,or16,oi16
,or17,oi17,or18,oi18,or19,oi19,or20,oi20,or21,oi21,or22,oi22,or23,oi23,
or24,oi24,or25,oi25,or26,oi26,or27,oi27,or28,oi28,or29,oi29,or30,oi30,o
r31,oi31,or32,oi32]=rad2ct16(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:),a_r(3,
:),a_i(3,:),a_r(4,:),a_i(4,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7
,:),a_i(7,:),a_r(8,:),a_i(8,:),a_r(9,:),a_i(9,:),a_r(10,:),a_i(10,:),a_
r(11,:),a_i(11,:),a_r(12,:),a_i(12,:),a_r(13,:),a_i(13,:),a_r(14,:),a_i
(14,:),a_r(15,:),a_i(15,:),a_r(16,:),a_i(16,:),a_r(17,:),a_i(17,:),a_r(
18,:),a_i(18,:),a_r(19,:),a_i(19,:),a_r(20,:),a_i(20,:),a_r(21,:),a_i(2
1,:),a_r(22,:),a_i(22,:),a_r(23,:),a_i(23,:),a_r(24,:),a_i(24,:),a_r(25
,:),a_i(25,:),a_r(26,:),a_i(26,:),a_r(27,:),a_i(27,:),a_r(28,:),a_i(28,
:),a_r(29,:),a_i(29,:),a_r(30,:),a_i(30,:),a_r(31,:),a_i(31,:),a_r(32,:
),a_i(32,:));
    o_r1=arr2dec(or1,shift);
    o_i1=arr2dec(oi1,shift);
    o_r2=arr2dec(or2,shift);
    o_i2=arr2dec(oi2,shift);
    o_r3=arr2dec(or3,shift);
    o_i3=arr2dec(oi3,shift);
    o_r4=arr2dec(or4,shift);
    o_i4=arr2dec(oi4,shift);
    o_r5=arr2dec(or5,shift);
    o_i5=arr2dec(oi5,shift);
    o_r6=arr2dec(or6,shift);
    o_i6=arr2dec(oi6,shift);
    o_r7=arr2dec(or7,shift);
    o_i7=arr2dec(oi7,shift);
    o_r8=arr2dec(or8,shift);
    o_i8=arr2dec(oi8,shift);

   % 2-point    b=[o_r1+j*o_i1;o_r2+j*o_i2];
   % 4-point       b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4];


b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4;o_r5+j*o_i5;o_r6+j*o
_i6;o_r7+j*o_i7;o_r8+j*o_i8];


   % 16-point
b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4;o_r5+j*o_i5;o_r6+j*o
_i6;o_r7+j*o_i7;o_r8+j*o_i8;o_r9+j*o_i9;o_r10+j*o_i10;o_r11+j*o_i11;o_r
12+j*o_i12;o_r13+j*o_i13;o_r14+j*o_i14;o_r15+j*o_i15;o_r16+j*o_i16];

   %
b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4;o_r5+j*o_i5;o_r6+j*o
_i6;o_r7+j*o_i7;o_r8+j*o_i8;o_r9+j*o_i9;o_r10+j*o_i10;o_r11+j*o_i11;o_r
12+j*o_i12;o_r13+j*o_i13;o_r14+j*o_i14;o_r15+j*o_i15;o_r16+j*o_i16;o_r1
```

```
7+j*o_i17;o_r18+j*o_i18;o_r19+j*o_i19;o_r20+j*o_i20;o_r21+j*o_i21;o_r22
+j*o_i22;o_r23+j*o_i23;o_r24+j*o_i24;o_r25+j*o_i25;o_r26+j*o_i26;o_r27+
j*o_i27;o_r28+j*o_i28;o_r29+j*o_i29;o_r30+j*o_i30;o_r31+j*o_i31;o_r32+j
*o_i32];
return
%%%%%%%%  END  %%%%%%%%

%%%%%%%%  ramasifft.m  %%%%%%%%
function [b]=ramasfft(a)
% Program to interface my fft engine in place of the standard FFT
function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

N=8;       %%% ! ! !  MODIFY ! ! !   %%%%%
shift=5;   %%% ! ! !  MODIFY ! ! !   %%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    or=zeros(N,1);
    oi=or;
    a_re=real(a);
    a_im=imag(a);

    a_r=zeros(N,16);
    a_i=zeros(N,16);
    o_r=a_r;
    o_i=a_i;

    for i=1:N
        a_r(i,:)=convert(a_re(i),2);
        a_i(i,:)=convert(a_im(i),2);
    end
%  4-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4]=rad2ct4ifft(a_r(1,:),a_i(1,:),a_r(2,:
),a_i(2,:),a_r(3,:),a_i(3,:),a_r(4,:),a_i(4,:));

[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8]=rad2c
t8ifft(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:),a_r(3,:),a_i(3,:),a_r(4,:),a
_i(4,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7,:),a_i(7,:),a_r(8,:),
a_i(8,:));
%  16-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8,or9,oi
9,or10,oi10,or11,oi11,or12,oi12,or13,oi13,or14,oi14,or15,oi15,or16,oi16
]=rad2ct16ifft(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:),a_r(3,:),a_i(3,:),a_
r(4,:),a_i(4,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7,:),a_i(7,:),a
_r(8,:),a_i(8,:),a_r(9,:),a_i(9,:),a_r(10,:),a_i(10,:),a_r(11,:),a_i(11
,:),a_r(12,:),a_i(12,:),a_r(13,:),a_i(13,:),a_r(14,:),a_i(14,:),a_r(15,
:),a_i(15,:),a_r(16,:),a_i(16,:));
%  32-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8,or9,oi
9,or10,oi10,or11,oi11,or12,oi12,or13,oi13,or14,oi14,or15,oi15,or16,oi16
,or17,oi17,or18,oi18,or19,oi19,or20,oi20,or21,oi21,or22,oi22,or23,oi23,
or24,oi24,or25,oi25,or26,oi26,or27,oi27,or28,oi28,or29,oi29,or30,oi30,o
r31,oi31,or32,oi32]=rad2ct32ifft(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:),a_
r(3,:),a_i(3,:),a_r(4,:),a_i(4,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a
_r(7,:),a_i(7,:),a_r(8,:),a_i(8,:),a_r(9,:),a_i(9,:),a_r(10,:),a_i(10,:
```

```
),a_r(11,:),a_i(11,:),a_r(12,:),a_i(12,:),a_r(13,:),a_i(13,:),a_r(14,:)
,a_i(14,:),a_r(15,:),a_i(15,:),a_r(16,:),a_i(16,:),a_r(17,:),a_i(17,:),
a_r(18,:),a_i(18,:),a_r(19,:),a_i(19,:),a_r(20,:),a_i(20,:),a_r(21,:),a
_i(21,:),a_r(22,:),a_i(22,:),a_r(23,:),a_i(23,:),a_r(24,:),a_i(24,:),a_
r(25,:),a_i(25,:),a_r(26,:),a_i(26,:),a_r(27,:),a_i(27,:),a_r(28,:),a_i
(28,:),a_r(29,:),a_i(29,:),a_r(30,:),a_i(30,:),a_r(31,:),a_i(31,:),a_r(
32,:),a_i(32,:));
    o_r1=arr2dec(or1,shift);
    o_i1=arr2dec(oi1,shift);
    o_r2=arr2dec(or2,shift);
    o_i2=arr2dec(oi2,shift);
    o_r3=arr2dec(or3,shift);
    o_i3=arr2dec(oi3,shift);
    o_r4=arr2dec(or4,shift);
    o_i4=arr2dec(oi4,shift);
    o_r5=arr2dec(or5,shift);
    o_i5=arr2dec(oi5,shift);
    o_r6=arr2dec(or6,shift);
    o_i6=arr2dec(oi6,shift);
    o_r7=arr2dec(or7,shift);
    o_i7=arr2dec(oi7,shift);
    o_r8=arr2dec(or8,shift);
    o_i8=arr2dec(oi8,shift);

   % 4-point    b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4];


b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4;o_r5+j*o_i5;o_r6+j*o
_i6;o_r7+j*o_i7;o_r8+j*o_i8];


   % 16-point
b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4;o_r5+j*o_i5;o_r6+j*o
_i6;o_r7+j*o_i7;o_r8+j*o_i8;o_r9+j*o_i9;o_r10+j*o_i10;o_r11+j*o_i11;o_r
12+j*o_i12;o_r13+j*o_i13;o_r14+j*o_i14;o_r15+j*o_i15;o_r16+j*o_i16];

   %
b=[o_r1+j*o_i1;o_r2+j*o_i2;o_r3+j*o_i3;o_r4+j*o_i4;o_r5+j*o_i5;o_r6+j*o
_i6;o_r7+j*o_i7;o_r8+j*o_i8;o_r9+j*o_i9;o_r10+j*o_i10;o_r11+j*o_i11;o_r
12+j*o_i12;o_r13+j*o_i13;o_r14+j*o_i14;o_r15+j*o_i15;o_r16+j*o_i16;o_r1
7+j*o_i17;o_r18+j*o_i18;o_r19+j*o_i19;o_r20+j*o_i20;o_r21+j*o_i21;o_r22
+j*o_i22;o_r23+j*o_i23;o_r24+j*o_i24;o_r25+j*o_i25;o_r26+j*o_i26;o_r27+
j*o_i27;o_r28+j*o_i28;o_r29+j*o_i29;o_r30+j*o_i30;o_r31+j*o_i31;o_r32+j
*o_i32];
return
%%%%%%%%%  END  %%%%%%%%


%%%%%%%%%  shrega.m  %%%%%%%%
function [S]=shrega(a)
% this function shifts the register contents to the right
% while taking care of SIGN EXTENTION
S=[a(1) a(1) a(1) a];
S=S(1:16);
return
%%%%%%%%%  END  %%%%%%%%
```

```
%%%%%%%%%  snrvary1.m  %%%%%%%%
% This program reads data from a text file 'intext.txt' and sends the
file data to the IFFT engine
% and may add noise to the transmitted data before receiving it and
passing  it to the FFT engine
% to decipher its output. The output is then written to a file called
'outtext.txt'.

% Gather FIDs for input and output files
infid= fopen('outtext.txt','r');
outfid=fopen('outtext2.txt','w');

% Noise Measure
snratio=[-20 -15 -10 -5 -4 -3 -2 -1 0];
lengsnr=length(snratio);
% Read Data from input file
[indata,incount]=fread(infid,'bit1');

%indata=randint(50,1);
%incount=50;
for g=1:lengsnr
g
% Initialize Outout data
outdata=[];
outsymbol=[];

N=8;

datalen=incount;
rema=mod(incount,2*N);
if (rema~=0)
    indata=[indata;zeros(2*N-rema,1)];
    datalen=datalen+2*N-rema
end
%indata=randint(datalen/N,1);
batches=datalen/N
symbollist=[];

% Encoding the input Data...
disp('Encoding the Input Data ...');

for i=0:batches-1
   symbollist=[symbollist; getsymbol(indata(N*i+1:N*(i+1)))];
end
disp('Encoding the Input Data .... Done');


for i=0:batches/2-1
    z=getformat(symbollist(N*i+1:N*(i+1/2)));
    transmit=ramasifft8(z);
    powx=sum(abs(transmit.*transmit));    %  Power of the transmitted
window of data

    pownoise=powx * 10 ^ ( snratio(g) / 10 );   % Noise power
calculation

    noise=rand(N,1);
```

```
    inputnoise=pownoise * noise;

    receiverinput=transmit+inputnoise;

    receive=ramasfft8(receiverinput);
    p=deformat(receive);
    outsymbol=[outsymbol; p];
    z=getformat(symbollist(N*(i+1/2)+1:N*(i+1)));
    transmit=ramasifft8(z);
    powx=sum(abs(transmit.*transmit));    %  Power of the transmitted
window of data

    pownoise=powx * 10 ^ ( snratio(g) / 10 );   % Noise power
calculation

    noise=rand(N,1);

    inputnoise=pownoise * noise;

    receiverinput=transmit+inputnoise;

    receive=ramasfft8(receiverinput);
    p=deformat(receive);
    outsymbol=[outsymbol; p];
end

% finding the nearest point in the given constellation
disp('Fitting the received vector in the constellation');
lenoutsymlist=length(outsymbol);
symout=zeros(lenoutsymlist,1);
for k=1:lenoutsymlist
   symout(k)=getpoint4qam(outsymbol(k));
end


% Decoding the input Data...
disp('Decoding the Output Data ...');
for i=0:batches/2-1
   y=symout(N*i+1:N*(i+1));
   outdata=[outdata; getnumber(y)];
end
disp('Decoding the Output Data .... Done');

[numberoferrors(g),ber(g)]=biterr(abs(indata),abs(outdata));
end


count=fwrite(outfid,outdata,'bit1');
st=fclose('all');
errdata=indata-outdata;
plot(abs(indata),'ro');
plot(abs(indata),'r');
hold on
plot(abs(outdata),'kd');
plot(abs(outdata),'k');
plot(abs(errdata),'*');
```

```
plot(abs(errdata));
hold off
%%%%%%%%%  END  %%%%%%%%

%%%%%%%%%  sub16.m  %%%%%%%%
function [S]=sub16(a,b)
% This function calculates the difference between two 16-bit numbers
and gives the
% result as a 16 bit word avoiding an overflow. So one bit of precision
is lost in the
% process.

T=com(b);
S=add16(a,T);
return
%%%%%%%%%  END  %%%%%%%%
```

APPENDIX B
32-BIT COOLEY-TUKEY AND CHIRP-Z IMPLEMENTATION

```
%%%%%%%%  add40.m  %%%%%%%%%
function [C]=add40(A,B)
% function [C]=add40(A,B)
% Adds two 32 bit numbers to produce a 40 bit output
%A,B --> in 32 bits
%C --> out 40 bits
Cin=0;
A=[A zeros(1,8)];
B=[B zeros(1,8)];
[tmp1,c0]=ad16c(A(25:40),B(25:40));
[tmp2,c1]=ad12c(A(13:24),B(13:24),c0);
[tmp3,c2]=ad12c(A(1:12),B(1:12),c1);
test=xor(A(1) ,B(1));
testbar=~test;
term1=tmp3(1) & test;
term2=c2 & testbar;
msb=term1 | term2;
C=[msb tmp3 tmp2 tmp1];
C=C(1:40);
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  bm4.m  %%%%%%%%%
function [P1]=bm4(X1,Y1)
X=X1(4:-1:1);
Y=Y1(4:-1:1);
%port(X,Y:in std_logic_vector(4 downto 1);
 %     P:out std_logic_vector(8 downto 1));
   LOW=0;
   for i= 1:4
        for j=1:4
          t(i+4*j-4)=X(i) & Y(j);
        end
     end
   P(1)=t(1);
   [P(2),c(1)]=fulladder(t(5),LOW,t(2));
   [a(1),c(2)]=fulladder(t(6),LOW,t(3));
   [a(2),c(3)]=fulladder(t(7),LOW,t(4));
   [P(3),c(4)]=fulladder(t(9),c(1),a(1));
```

```
  [a(3),c(5)]=fulladder(t(10),c(2),a(2));
  [a(4),c(6)]=fulladder(t(11),c(3),t(8));
  [P(4),c(7)]=fulladder(t(13),c(4),a(3));
  [a(5),c(8)]=fulladder(t(14),c(5),a(4));
  [a(6),c(9)]=fulladder(t(15),c(6),t(12));
  [P(5),c(10)]=fulladder(LOW,c(7),a(5));
  [P(6),c(11)]=fulladder(c(10),c(8),a(6));
  [P(7),P(8)]=fulladder(c(11),c(9),t(16));
  P1=P(8:-1:1);
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  bm8.m  %%%%%%%%%
function[P1]=bm8(A1,B1)
%port(X,Y:= std_logic_vector(7 downto 0);
 %     P:out std_logic_vector(15 downto 0));
 X=A1(8:-1:1);
 Y=B1(8:-1:1);
  LOW=0;
    for i = 1:8
      for j = 1:8
        t(i+8*j-8)=X(i) & Y(j);
      end
    end
  P(1)=t(1);
  [P(2),c(1)]=fulladder(t(9),LOW,t(2));
  for i=2:7
     [a(i-1),c(i)]=fulladder(t(i+1),LOW,t(i+8));
  end
  [P(3),c(8)]=fulladder(t(17),c(1),a(1));
  for i=1:5
     [a(i+6),c(i+8)]=fulladder(t(i+17),a(i+1),c(i+1));
  end
  [a(12),c(14)]=fulladder(t(16),c(7),t(23));
  [P(4),c(15)]=fulladder(t(25),c(8),a(7));
  for i=1:5
     [a(i+12),c(i+15)]=fulladder(t(i+25),a(i+7),c(i+8));
  end
  [a(18),c(21)]=fulladder(t(24),c(14),t(31));
  [P(5),c(22)]=fulladder(t(33),c(15),a(13));
  for i=1:5
     [a(i+18),c(i+22)]=fulladder(t(i+33),a(i+13),c(i+15));
  end
  [a(24),c(28)]=fulladder(t(32),c(21),t(39));
  [P(6),c(29)]=fulladder(t(41),c(22),a(19));
  for i=1:5
```

```
      [a(i+24),c(i+29)]=fulladder(t(i+41),a(i+19),c(i+22));
   end
   [a(30),c(35)]=fulladder(t(40),c(28),t(47));
   [P(7),c(36)]=fulladder(t(49),c(29),a(25));
   for i=1:5
      [a(i+30),c(i+36)]=fulladder(t(i+49),a(i+25),c(i+29));
   end
   [a(36),c(42)]=fulladder(t(48),c(35),t(55));
   [P(8),c(43)]=fulladder(t(57),c(36),a(31));
   for i=1:5
      [a(i+36),c(i+43)]=fulladder(t(i+55),a(i+31),c(i+36));
   end
   [a(42),c(49)]=fulladder(t(56),c(42),t(63));
   [P(9),c(50)]=fulladder(LOW,c(43),a(37));
   [P(10),c(51)]=fulladder(c(50),c(44),a(38));
   [P(11),c(52)]=fulladder(c(51),c(45),a(39));
   [P(12),c(53)]=fulladder(c(52),c(46),a(40));
   [P(13),c(54)]=fulladder(c(53),c(47),a(41));
   [P(14),c(55)]=fulladder(c(54),c(48),a(42));
   [P(15),P(16)]=fulladder(c(55),c(49),t(64));
   P1=P(16:-1:1);
 return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  bwcell1.m  %%%%%%%%%
function [P]=bwcell1(X,Y)
  P=X & Y;
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  bwcell2.m  %%%%%%%%%
function [SUMOUT,COUT]=bwcell2(X,Y,SUMIN, CIN)
   D=X & Y;
[SUMOUT,COUT]=fulladder(D,SUMIN,CIN);
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  bwcell3.m  %%%%%%%%%
function [s]=bwcell3(a,b)
% Cell 3 of the Baugh Wooley Multiplier
s=a & ~b;
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  bwcell4.m  %%%%%%%%%
function [s,cout]=bwcell4(a,b,Sin,Cin)
```

```
% Cell 4 of the Baugh Wooley Multiplier
c=~a & b;
[s,cout]=fulladder(c,Sin,Cin);
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  bwcell5.m  %%%%%%%%%
function [s,cout]=bwcell5(x,y)
% Cell 5 of the Baugh Wooley Multiplier
b=x & y;
c=~x;
d=~y;
[s,cout]=fulladder(b,c,d);
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  circconv.m  %%%%%%%%%
function z=circconv(a,b)
n=length(a);
z=zeros(size(a));
if (n~=length(b))
   ERROR('Unequal Vector lengths in the circular convolution argument');
end
x=1;
y=1;
for i=1:n
   z(i)=0;
   for j=1:n
      if(x>n)
         x=x-n;
      end
      if (y<1)
         y=y+n;
      end
      z(i)=z(i)+a(x)*b(y);
      x=x+1;
      y=y-1;
   end
   y=y+1;
end
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  com32.m  %%%%%%%%%
function [b]=com32(a)
%function [b]=com32(a)
```

```
% returns the 16-bit complement of a number represented by the array 'a'
% for example,
% if a=[1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 1 1 0 1 1 0 1 1 0 1 1 0 0 0 1 0 1]
% then b =[0 1 1 0 0 1 1 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 1 1 1 0 1 1]
  tmp=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1];
  tmp1=~a;
  [tmp3]= add40(tmp1,tmp);
  b=tmp3(2:33);
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  cplxmul16.m  %%%%%%%%%
function [pr,pi]=cplxmul16( xr,xi,yr,yi,clk,resetn)
  tmp0=pipemul16 (xr,yr,clk,resetn);
  tmp1=pipemul16 (xr,yi,clk,resetn);
  tmp2=pipemul16 (xi,yr,clk,resetn);
  tmp3=pipemul16 (xi,yi,clk,resetn);
  pra=sub40(tmp0,tmp3);
  pia=add40(tmp2,tmp1);
  pr=pra(1:32);
  pi=pia(1:32);
return
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  czt.m  %%%%%%%%%
N=4;
ROM1=zeros(size(cplx));
ROM0=zeros(size(cplx));
sqr=zeros(size(cplx));
hi=zeros(size(cplx));
hr=zeros(size(cplx));
for i=1:N
   ROM0(i)=cos(pi*(i-1)*(i-1)/N);
   ROM1(i)=-sin(pi*(i-1)*(i-1)/N);
   hr(i)=ROM0(i);
   hi(i)=-ROM1(i);
end
g0= cplx .* ROM0;
g1= cplx .* ROM1;
hi
c=1;
o1=circconv(hr,g0);
o2=circconv(hi,g0);
o3=circconv(hr,g1);
o4=circconv(hi,g1);
a0=o1-o4;
```

```
a1=o2+o3;
s1=a0 .* a0;
s2=a1 .* a1;
a3=s1+s2;
sqr=sqrt(a3);
%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  deformat.m  %%%%%%%%%

function [deformed]=deformat(formatin)
% Function reads the out put of the FFT at the receiver and deformats it into
% its constituent symbols.

len=length(formatin);

deformed=[real(formatin(1))+j*imag(formatin(len/2))];

for i=2:len/2
   deformed=[deformed; formatin(i)];
end
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  fircos2.m  %%%%%%%%%

function [z0,z1,z2,z3]=fircos2(x0,x1,x2,x3);


 y0 = [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];       % 4000h --> 1
 y1 = [0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0];       % 3b20  --> 0.9239

p0=bwm16a(x0,y0);
p1=bwm16a(x1,y1);
z0=add16(p0,p1);

q0=bwm16a(x0,y1);
q1=bwm16a(x1,y0);
z1=add16(q0,q1);

return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  fircos4.m  %%%%%%%%%
```

```
function [z0,z1,z2,z3]=fircos4(x0,x1,x2,x3);

 %y0 = [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];       % 4000h --> 1
 y1 = [0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];       % 2d41  --> 0.7071
 y3 = [1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];       % c000  --> -1
 y2 = [0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];       % 2d41  --> 0.7071

p0=shrega(x0);
p1=bwm16a(x1,y3);
p2=bwm16a(x2,y2);
p3=bwm16a(x3,y1);


pr01=add16(p0,p1);
pr23=add16(p2,p3);

z0=add16(pr01,pr23);


q0=bwm16a(x0,y1);
q1=shrega(x1);
q2=bwm16a(x2,y3);
q3=bwm16a(x3,y2);

qr01=add16(q0,q1);
qr23=add16(q2,q3);
z1=add16(qr01,qr23);


r0=bwm16a(x0,y2);
r1=bwm16a(x1,y1);
r2=shrega(x2);
r3=bwm16a(x3,y3);

rr01=add16(r0,r1);
rr23=add16(r2,r3);
z2=add16(rr01,rr23);

s0=bwm16a(x0,y3);
s1=bwm16a(x1,y2);
s2=bwm16a(x2,y1);
s3=shrega(x3);

sr01=add16(s0,s1);
sr23=add16(s2,s3);
z3=add16(sr01,sr23);
```

```
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  fircos8.m  %%%%%%%%%

function [z0,z1,z2,z3,z4,z5,z6,z7]=fircos8(x0,x1,x2,x3,x4,x5,x6,x7);


 y0 = [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      % 4000h --> 1
 y1 = [0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0];      % 3b20  --> 0.9239
 y3 = [1 1 0 0 0 1 0 0 1 1 0 1 1 1 1 1];     % c4df  --> -0.9239
 y2 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      %  0000 --> 0
 y4 = [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      % 4000h --> 1
 y5 = [1 1 0 0 0 1 0 0 1 1 0 1 1 1 1 1];     % c4df  --> -0.9239
 y6 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      %  0000 --> 0
 y7 = [0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0];      % 3b20  --> 0.9239

p0=bwm16a(x0,y0);
p1=bwm16a(x1,y7);
p2=bwm16a(x2,y6);
p3=bwm16a(x3,y5);
p4=bwm16a(x4,y4);
p5=bwm16a(x5,y3);
p6=bwm16a(x6,y2);
p7=bwm16a(x7,y1);


pr01=add16(p0,p1);
pr23=add16(p2,p3);
pr45=add16(p4,p5);
pr67=add16(p6,p7);
pri03=add16(pr01,pr23);
pri47=add16(pr45,pr67);
z0=add16(pri03,pri47);

q0=bwm16a(x0,y1);
q1=bwm16a(x1,y0);
q2=bwm16a(x2,y7);
q3=bwm16a(x3,y6);
q4=bwm16a(x4,y5);
q5=bwm16a(x5,y4);
q6=bwm16a(x6,y3);
q7=bwm16a(x7,y2);

qr01=add16(q0,q1);
```

```
qr23=add16(q2,q3);
qr45=add16(q4,q5);
qr67=add16(q6,q7);
qri03=add16(qr01,qr23);
qri47=add16(qr45,qr67);
z1=add16(qri03,qri47);


r0=bwm16a(x0,y2);
r1=bwm16a(x1,y1);
r2=bwm16a(x2,y0);
r3=bwm16a(x3,y7);
r4=bwm16a(x4,y6);
r5=bwm16a(x5,y5);
r6=bwm16a(x6,y4);
r7=bwm16a(x7,y3);

rr01=add16(r0,r1);
rr23=add16(r2,r3);
rr45=add16(r4,r5);
rr67=add16(r6,r7);
rri03=add16(rr01,rr23);
rri47=add16(rr45,rr67);
z2=add16(rri03,rri47);


s0=bwm16a(x0,y3);
s1=bwm16a(x1,y2);
s2=bwm16a(x2,y1);
s3=bwm16a(x3,y0);
s4=bwm16a(x4,y7);
s5=bwm16a(x5,y6);
s6=bwm16a(x6,y5);
s7=bwm16a(x7,y4);

sr01=add16(s0,s1);
sr23=add16(s2,s3);
sr45=add16(s4,s5);
sr67=add16(s6,s7);
sri03=add16(sr01,sr23);
sri47=add16(sr45,sr67);
z3=add16(sri03,sri47);


t0=bwm16a(x0,y4);
t1=bwm16a(x1,y3);
```

```
t2=bwm16a(x2,y2);
t3=bwm16a(x3,y1);
t4=bwm16a(x4,y0);
t5=bwm16a(x5,y7);
t6=bwm16a(x6,y6);
t7=bwm16a(x7,y5);

tr01=add16(t0,t1);
tr23=add16(t2,t3);
tr45=add16(t4,t5);
tr67=add16(t6,t7);
tri03=add16(tr01,tr23);
tri47=add16(tr45,tr67);
z4=add16(tri03,tri47);

u0=bwm16a(x0,y5);
u1=bwm16a(x1,y4);
u2=bwm16a(x2,y3);
u3=bwm16a(x3,y2);
u4=bwm16a(x4,y1);
u5=bwm16a(x5,y0);
u6=bwm16a(x6,y7);
u7=bwm16a(x7,y6);
ur01=add16(u0,u1);
ur23=add16(u2,u3);
ur45=add16(u4,u5);
ur67=add16(u6,u7);
uri03=add16(ur01,ur23);
uri47=add16(ur45,ur67);
z5=add16(uri03,uri47);

v0=bwm16a(x0,y6);
v1=bwm16a(x1,y5);
v2=bwm16a(x2,y4);
v3=bwm16a(x3,y3);
v4=bwm16a(x4,y2);
v5=bwm16a(x5,y1);
v6=bwm16a(x6,y0);
v7=bwm16a(x7,y7);
vr01=add16(v0,v1);
vr23=add16(v2,v3);
vr45=add16(v4,v5);
vr67=add16(v6,v7);
vri03=add16(vr01,vr23);
vri47=add16(vr45,vr67);
z6=add16(vri03,vri47);
```

```
w0=bwm16a(x0,y7);
w1=bwm16a(x1,y6);
w2=bwm16a(x2,y5);
w3=bwm16a(x3,y4);
w4=bwm16a(x4,y3);
w5=bwm16a(x5,y2);
w6=bwm16a(x6,y1);
w7=bwm16a(x7,y0);

wr01=add16(w0,w1);
wr23=add16(w2,w3);
wr45=add16(w4,w5);
wr67=add16(w6,w7);
wri03=add16(wr01,wr23);
wri47=add16(wr45,wr67);
z7=add16(wri03,wri47);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  firsin2.m  %%%%%%%%%

function [z0,z1,z2,z3]=firsin2(x0,x1,x2,x3);


 y0 = [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];       % 4000h --> 1
 y1 = [0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0];       % 3b20  --> 0.9239

p0=bwm16a(x0,y0);
p1=bwm16a(x1,y1);
z0=add16(p0,p1);

q0=bwm16a(x0,y1);
q1=bwm16a(x1,y0);
z1=add16(q0,q1);

return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  firsin4.m  %%%%%%%%%

function [z0,z1,z2,z3]=firsin4(x0,x1,x2,x3);

 y1 = [0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];       % 2d41h  --> 0.7071
 y3 = [0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];       %  2d41 --> 0.7071
```

```
p0=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
p1=bwm16a(x1,y3);
p2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
p3=bwm16a(x3,y1);


pr01=add16(p0,p1);
pr23=add16(p2,p3);

z0=add16(pr01,pr23);


q0=bwm16a(x0,y1);
q1=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
q2=bwm16a(x2,y3);
q3=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];

qr01=add16(q0,q1);
qr23=add16(q2,q3);
z1=add16(qr01,qr23);


r0=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
r1=bwm16a(x1,y1);
r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
r3=bwm16a(x3,y3);

rr01=add16(r0,r1);
rr23=add16(r2,r3);
z2=add16(rr01,rr23);

s0=bwm16a(x0,y3);
s1=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
s2=bwm16a(x2,y1);
s3=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];

sr01=add16(s0,s1);
sr23=add16(s2,s3);
z3=add16(sr01,sr23);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  firsin8.m  %%%%%%%%%
```

```
function [z0,z1,z2,z3,z4,z5,z6,z7]=firsin8(x0,x1,x2,x3,x4,x5,x6,x7);

y0 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      % 0000h --> 0
y1 = [0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 1];      % 187d  --> 0.3827
y3 = [1 1 1 0 0 1 1 1 1 0 0 0 0 0 1 0];      % e782  --> -0.9239
y2 = [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      %  4000 --> 1
y4 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      % 0000h --> 0
y5 = [1 1 1 0 0 1 1 1 1 0 0 0 0 0 1 0];      % e782  --> -0.9239
y6 = [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];      %  4000 --> 1
y7 = [0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 1];      % 187d  --> 0.3827

p0=bwm16a(x0,y0);
p1=bwm16a(x1,y7);
p2=bwm16a(x2,y6);
p3=bwm16a(x3,y5);
p4=bwm16a(x4,y4);
p5=bwm16a(x5,y3);
p6=bwm16a(x6,y2);
p7=bwm16a(x7,y1);


pr01=add16(p0,p1);
pr23=add16(p2,p3);
pr45=add16(p4,p5);
pr67=add16(p6,p7);
pri03=add16(pr01,pr23);
pri47=add16(pr45,pr67);
z0=add16(pri03,pri47);

q0=bwm16a(x0,y1);
q1=bwm16a(x1,y0);
q2=bwm16a(x2,y7);
q3=bwm16a(x3,y6);
q4=bwm16a(x4,y5);
q5=bwm16a(x5,y4);
q6=bwm16a(x6,y3);
q7=bwm16a(x7,y2);

qr01=add16(q0,q1);
qr23=add16(q2,q3);
qr45=add16(q4,q5);
qr67=add16(q6,q7);
qri03=add16(qr01,qr23);
qri47=add16(qr45,qr67);
z1=add16(qri03,qri47);
```

```
r0=bwm16a(x0,y2);
r1=bwm16a(x1,y1);
r2=bwm16a(x2,y0);
r3=bwm16a(x3,y7);
r4=bwm16a(x4,y6);
r5=bwm16a(x5,y5);
r6=bwm16a(x6,y4);
r7=bwm16a(x7,y3);

rr01=add16(r0,r1);
rr23=add16(r2,r3);
rr45=add16(r4,r5);
rr67=add16(r6,r7);
rri03=add16(rr01,rr23);
rri47=add16(rr45,rr67);
z2=add16(rri03,rri47);


s0=bwm16a(x0,y3);
s1=bwm16a(x1,y2);
s2=bwm16a(x2,y1);
s3=bwm16a(x3,y0);
s4=bwm16a(x4,y7);
s5=bwm16a(x5,y6);
s6=bwm16a(x6,y5);
s7=bwm16a(x7,y4);

sr01=add16(s0,s1);
sr23=add16(s2,s3);
sr45=add16(s4,s5);
sr67=add16(s6,s7);
sri03=add16(sr01,sr23);
sri47=add16(sr45,sr67);
z3=add16(sri03,sri47);


t0=bwm16a(x0,y4);
t1=bwm16a(x1,y3);
t2=bwm16a(x2,y2);
t3=bwm16a(x3,y1);
t4=bwm16a(x4,y0);
t5=bwm16a(x5,y7);
t6=bwm16a(x6,y6);
t7=bwm16a(x7,y5);
```

```
tr01=add16(t0,t1);
tr23=add16(t2,t3);
tr45=add16(t4,t5);
tr67=add16(t6,t7);
tri03=add16(tr01,tr23);
tri47=add16(tr45,tr67);
z4=add16(tri03,tri47);

u0=bwm16a(x0,y5);
u1=bwm16a(x1,y4);
u2=bwm16a(x2,y3);
u3=bwm16a(x3,y2);
u4=bwm16a(x4,y1);
u5=bwm16a(x5,y0);
u6=bwm16a(x6,y7);
u7=bwm16a(x7,y6);
ur01=add16(u0,u1);
ur23=add16(u2,u3);
ur45=add16(u4,u5);
ur67=add16(u6,u7);
uri03=add16(ur01,ur23);
uri47=add16(ur45,ur67);
z5=add16(uri03,uri47);

v0=bwm16a(x0,y6);
v1=bwm16a(x1,y5);
v2=bwm16a(x2,y4);
v3=bwm16a(x3,y3);
v4=bwm16a(x4,y2);
v5=bwm16a(x5,y1);
v6=bwm16a(x6,y0);
v7=bwm16a(x7,y7);
vr01=add16(v0,v1);
vr23=add16(v2,v3);
vr45=add16(v4,v5);
vr67=add16(v6,v7);
vri03=add16(vr01,vr23);
vri47=add16(vr45,vr67);
z6=add16(vri03,vri47);
w0=bwm16a(x0,y7);
w1=bwm16a(x1,y6);
w2=bwm16a(x2,y5);
w3=bwm16a(x3,y4);
w4=bwm16a(x4,y3);
w5=bwm16a(x5,y2);
w6=bwm16a(x6,y1);
```

```
w7=bwm16a(x7,y0);

wr01=add16(w0,w1);
wr23=add16(w2,w3);
wr45=add16(w4,w5);
wr67=add16(w6,w7);
wri03=add16(wr01,wr23);
wri47=add16(wr45,wr67);
z7=add16(wri03,wri47);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  getformat.m  %%%%%%%%%

function [format]=getformat(a)
% function gets the input vector a into a format that allows only real transmissions
% to be possible in the time domain data.

lena=length(a);

format=[real(a(1)); a(2:lena); imag(a(1)); conj(a(2:lena))];
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  icztsim.m  %%%%%%%%%

function sqr=icztsim(cplx)
N=length(cplx);
ROM1=zeros(size(cplx));
ROM0=zeros(size(cplx));
sqr=zeros(size(cplx));
hi=zeros(size(cplx));
hr=zeros(size(cplx));

for i=1:N
   ROM0(i)=cos(pi*(i-1)*(i-1)/N);
   ROM1(i)=sin(pi*(i-1)*(i-1)/N);
   hr(i)=ROM0(i);
   hi(i)=ROM1(i);
end
g0= cplx .* ROM0;
g1= cplx .* ROM1;

c=1;
```

```
o1=circconv(hr,g0);
o2=circconv(hi,g0);
o3=circconv(hr,g1);
o4=circconv(hi,g1);

a0=o1-o4;
a1=o2+o3;

s1=a0 .* a0;
s2=a1 .* a1;

a3=s1+s2;
sqr=sqrt(a3);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  muln.m  %%%%%%%%%

function [C]=muln(A,B)
%muln
%----

%A,B  -->  inputs (n downto 1)
%clk --> input 1 bit

%C  --> output(2n downto 1)
n=length(A);
C=zeros(2*n,1);

if n==2

  C(1) =A(1) & B(1);
  x=A(2) & B(2);
  y= A(1) & B(2);
  z= A(2) & B(1);
  C(2)= xor(y,z);
  n=y & z;
  C(3)=xor(n,x);
  C(4)=n & x;
else
  p=muln(A(n:-1:n/2+1),B(n:-1:n/2+1));
  q=muln(A(n:-1:n/2+1),B(n/2:-1:1));
  r=muln(A(n/2:-1:1),B(n:-1:n/2+1));
  s=muln(A(n/2:-1:1),B(n/2:-1:1));
```

```
  tmp=[zeros(1,n/2); s(n:-1:n/2+1)];
  tmp2=tmp+r+r;
  tmp3=[zeros(1,n/2) tmp2(n:-1:n/2+1)];
  tmp4=tmp3+p;
  C=[tmp4 tmp2(n/2:-1:1) s(n/2:-1:1)];
end;
return;

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  multiplier.m  %%%%%%%%%

function [C]=muln(A,B)
%muln
%----
%A,B  -->  inputs (n downto 1)
%clk --> input 1 bit
%C  --> output(2n downto 1)
n=length(A);
C=zeros(2*n,1);

if n==2
then
  C(1) =A(1) and B(1);
  x=A(2) and B(2);
  y= A(1) and B(2);
  z= A(2) and B(1);
  C(2)= y xor z;
  n=y and z;
  C(3)=n xor x;
  C(4)=n and x;
  return;
else
  p=muln(A(n:n/2+1),B(n:n/2+1);
  q=muln(A(n:n/2+1,B(n/2:1));
  r=muln(A(n/2:1),B(n:n/2+1);
  s=muln(A(n/2:1),B(n/2:1));
  tmp=[zeros(1,n/2) s(n:n/2+1)];
  tmp2=tmp+r+r;
  tmp3=[zeros(1,n/2) tmp2(n:n/2+1)];
  tmp4=tmp3+p;
  C=[tmp4 tmp2(n/2:1) s(n/2:1)];
  return;
end if;

%%%%%%%%  END  %%%%%%%%%
```

```
%%%%%%%%  pipemul16.m  %%%%%%%%%

function result=pipemul16(xa1,xb1,clk,resetn)
LOW=0;
ya=com16(xa1);
yb=com16(xb1);
if xa1(1)==1
   ta1=ya(9:16);
   ta2=ya(1:8);
else
   ta1=xa1(9:16);
   ta2=xa1(1:8);
end
if xb1(1)==1
   tb1=yb(9:16);
   tb2=yb(1:8);
else
   tb1=xb1(9:16);
   tb2=xb1(1:8);
end
tmp0_1=pipemul8(ta1,tb1,clk,resetn);
tmp0_2=pipemul8(ta2,tb1,clk,resetn);
tmp0_3=pipemul8(ta1,tb2,clk,resetn);
tmp0_4=pipemul8(ta2,tb2,clk,resetn);
tmp1_1=reg16(tmp0_1,resetn);
tmp1_2=reg16(tmp0_2,resetn);
tmp1_3=reg16(tmp0_3,resetn);
tmp1_4=reg16(tmp0_4,resetn);
tmp2_2=add1617(tmp1_2,tmp1_3);
tmp3_1=reg16(tmp1_1,resetn);
tmp3_2=reg17(tmp2_2,resetn);
tmp3_3=reg16(tmp1_4,resetn);
tc1=[tmp3_2(10:17) zeros(1,8)];
tc2=[zeros(1,7) tmp3_2(1:9)];
tmp4_1=add1617(tmp3_1,tc1);
tmp4_2=reg16(tc2,resetn);
tmp4_3=reg16(tmp3_3,resetn);
tmp5=ad16ca(tmp4_2,tmp4_3,tmp4_1(1));
p=[tmp5(2:17) tmp4_1(2:17)];
p1=com32(p);
if xa1(1)==xb1(1)
   result=p;
else
   result=p1;
end
```

```
return


%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  pipemul8.m  %%%%%%%%%

function p=pipemul8(xa1,xb1,clk,resetn)
LOW=0;

tb1=xb1(5:8);
tb2=xb1(1:4);
ta1=xa1(5:8);
ta2=xa1(1:4);
tmp0_1=bm4(ta1,tb1);
tmp0_2=bm4(ta2,tb1);
tmp0_3=bm4(ta1,tb2);
tmp0_4=bm4(ta2,tb2);
tmp1_1=reg8(tmp0_1,resetn);
tmp1_2=reg8(tmp0_2,resetn);
tmp1_3=reg8(tmp0_3,resetn);
tmp1_4=reg8(tmp0_4,resetn);
tmp2_2=add89(tmp1_2,tmp1_3);
tmp3_1=reg8(tmp1_1,resetn);
tmp3_2=reg9(tmp2_2,resetn);
tmp3_3=reg8(tmp1_4,resetn);

tc1=[tmp3_2(6:9) zeros(1,4)];
tc2=[zeros(1,3) tmp3_2(1:5)];

tmp4_1=add89(tmp3_1,tc1);
tmp4_2=reg8(tc2,resetn);
tmp4_3=reg8(tmp3_3,resetn);
tmp5=ad8ca(tmp4_2,tmp4_3,tmp4_1(1));

p=[tmp5(2:9) tmp4_1(2:9)];
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct2.m  %%%%%%%%%

function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  This function calculates the 2-point FFT of two 32 bit complex inputs
%  the result is a 2-point 16-bit complex output
```

```
tmpre0=add40(ir0,ir1);
tmpre1=sub40(ir0,ir1);
tmpi0=add40(ii0,ii1);
tmpi1=sub40(ii0,ii1);

  ore0=tmpre0(1:16);
  oi0=tmpi0(1:16);
  ore1=tmpre1(1:16);
  oi1=tmpi1(1:16);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct2ifft.m  %%%%%%%%%

function [ore0,oi0,ore1,oi1]=rad2ct2ifft(ir0,ii0,ir1,ii1)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  This function calculates the 2-point iFFT of two 32 bit complex inputs
%  the result is a 2-point 16-bit complex output

tmpre0=add40(ir0,ir1);
tmpre1=sub40(ir0,ir1);
tmpi0=add40(ii0,ii1);
tmpi1=sub40(ii0,ii1);

  ore0=tmpre0(1:16);
  oi0=tmpi0(1:16);
  ore1=tmpre1(1:16);
  oi1=tmpi1(1:16);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct4.m  %%%%%%%%%

function [ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3]=rad2ct4(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  This function calculates the 2-point FFT of two 32 bit complex inputs
%  the result is a 2-point 16-bit complex output
% If the input is in the notation 32.32-x(32 bit input, x integer bits) then the output would
be in the form
% 32.32-x-2 i.e., there would be a 4 bit shift in the output implying 2 bit precision is lost.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%    Statistics    %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Statistics collected over 100 iterations

%Error_mean                 =  -0.0001 + 1.0655i
%Error_variance             =   5.4781
%Avg_no_of_magnitude_errors  =   0
%Avg_no_of_sign_errors       =   0.1300
%Avg_no_of_mag_and_sign_errors =   0
%Avg_no_of_right_results     =   3.8700
%Avg_ratio                  =   1.0001


  zero=0;
  [tmpr0,tmpi0,tmpr2,tmpi2]=rad2ct2(ir0,ii0,ir2,ii2);
  [tmpr1,tmpi1,tmpr3,tmpi3]=rad2ct2(ir1,ii1,ir3,ii3);

  tr0=[tmpr0 zeros(1,16)];
  tr1=[tmpr1 zeros(1,16)];
  tr2=[tmpr2 zeros(1,16)];
  tr3=[tmpr3 zeros(1,16)];
  ti0=[tmpi0 zeros(1,16)];
  ti1=[tmpi1 zeros(1,16)];
  ti2=[tmpi2 zeros(1,16)];
  ti3=[tmpi3 zeros(1,16)];

  [ore0,oi0,ore2,oi2]=rad2ct2(tr0,ti0,tr1,ti1);
  [tr3c]=com32(tr3);
  [ore1,oi1,ore3,oi3]=rad2ct2(tr2,ti2,ti3,tr3c);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct4ifft.m  %%%%%%%%%

function [ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3]=rad2ct4ifft(ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  This function calculates the 2-point FFT of two 32 bit complex inputs
%  the result is a 2-point 16-bit complex output
  zero=0;
  [tmpr0,tmpi0,tmpr2,tmpi2]=rad2ct2(ir0,ii0,ir2,ii2);
  [tmpr1,tmpi1,tmpr3,tmpi3]=rad2ct2(ir1,ii1,ir3,ii3);

  tr0=[tmpr0 zeros(1,16)];
```

```
  tr1=[tmpr1 zeros(1,16)];
  tr2=[tmpr2 zeros(1,16)];
  tr3=[tmpr3 zeros(1,16)];
  ti0=[tmpi0 zeros(1,16)];
  ti1=[tmpi1 zeros(1,16)];
  ti2=[tmpi2 zeros(1,16)];
  ti3=[tmpi3 zeros(1,16)];

  [ore0,oi0,ore2,oi2]=rad2ct2(tr0,ti0,tr1,ti1);
  [ti3c]=com32(ti3);
  [ore1,oi1,ore3,oi3]=rad2ct2(tr2,ti2,ti3c,tr3);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct8.m  %%%%%%%%%

function
[ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3,ore4,oi4,ore5,oi5,ore6,oi6,ore7,oi7]=rad2ct8(ir0,ii0,i
r1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  this function calculates the 2-point FFt of eight 32 bit complex inputs
%  the result is a 8-point 16-bit complex output

%Error_mean =  -0.2844 + 0.7659i
%Error_variance =    6.0012
%Avg_no_of_magnitude_errors =     0
%Avg_no_of_sign_errors =    0.0100
%Avg_no_of_mag_and_sign_errors =    7.9900
%Avg_no_of_right_results =     0
%Avg_ratio =    2.0006

[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad2ct4(ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad2ct4(ir1,ii1,ir3,ii3,ir5,ii5,ir7,ii7);

[tra0]=shrega(zeropad(tr0));
[tia0]=shrega(zeropad(ti0));
[tra1]=shrega(zeropad(tr1));
[tia1]=shrega(zeropad(ti1));
[tra2]=shrega(zeropad(tr2));
[tia2]=shrega(zeropad(ti2));
[tra3]=shrega(zeropad(tr3));
[tia3]=shrega(zeropad(ti3));
[tra4]=shrega(zeropad(tr4));
[tia4]=shrega(zeropad(ti4));
```

```
W8_r1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
W8_i1=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
W8_r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
W8_i2=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1];
W8_r3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
W8_i3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];

tr5=tr5(1:16);
ti5=ti5(1:16);
tr6=tr6(1:16);
ti6=ti6(1:16);
tr7=tr7(1:16);
ti7=ti7(1:16);
clk=1;
resetn=1;

[tra5,tia5]=cplxmul16(tr5,ti5,W8_r1,W8_i1,clk,resetn);
[tra6,tia6]=cplxmul16(tr6,ti6,W8_r2,W8_i2,clk,resetn);
[tra7,tia7]=cplxmul16(tr7,ti7,W8_r3,W8_i3,clk,resetn);

[ore0,oi0,ore4,oi4]=rad2ct2(tra0,tia0,tra4,tia4);
[ore1,oi1,ore5,oi5]=rad2ct2(tra1,tia1,tra5,tia5);
[ore2,oi2,ore6,oi6]=rad2ct2(tra2,tia2,tra6,tia6);
[ore3,oi3,ore7,oi7]=rad2ct2(tra3,tia3,tra7,tia7);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct8fft.m  %%%%%%%%%

function
[ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3,ore4,oi4,ore5,oi5,ore6,oi6,ore7,oi7]=rad2ct8fft(ir0,ii
0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  this function calculates the 2-point FFt of eight 32 bit complex inputs
%  the result is a 8-point 16-bit complex output

%Error_mean =  -0.2844 + 0.7659i
%Error_variance =    6.0012
%Avg_no_of_magnitude_errors =     0
%Avg_no_of_sign_errors =    0.0100
%Avg_no_of_mag_and_sign_errors =    7.9900
%Avg_no_of_right_results =     0
%Avg_ratio =    2.0006
```

```
check(ir0);
check(ii0);

check(ir1);
check(ii1);
check(ir2);
check(ii2);
check(ir3);
check(ii3);
check(ir4);
check(ii4);
check(ir5);
check(ii5);
check(ir6);
check(ii6);
check(ir7);
check(ii7);




[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad2ct4fft(ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad2ct4fft(ir1,ii1,ir3,ii3,ir5,ii5,ir7,ii7);

[tra0]=shrega(zeropad(tr0));
[tia0]=shrega(zeropad(ti0));
[tra1]=shrega(zeropad(tr1));
[tia1]=shrega(zeropad(ti1));
[tra2]=shrega(zeropad(tr2));
[tia2]=shrega(zeropad(ti2));
[tra3]=shrega(zeropad(tr3));
[tia3]=shrega(zeropad(ti3));
[tra4]=shrega(zeropad(tr4));
[tia4]=shrega(zeropad(ti4));

W8_r1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
W8_i1=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
W8_r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
W8_i2=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1];
W8_r3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
W8_i3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];

tr5=tr5(1:16);
ti5=ti5(1:16);
tr6=tr6(1:16);
```

```
ti6=ti6(1:16);
tr7=tr7(1:16);
ti7=ti7(1:16);
clk=1;
resetn=1;

[tra5,tia5]=cplxmul16(tr5,ti5,W8_r1,W8_i1,clk,resetn);
[tra6,tia6]=cplxmul16(tr6,ti6,W8_r2,W8_i2,clk,resetn);
[tra7,tia7]=cplxmul16(tr7,ti7,W8_r3,W8_i3,clk,resetn);

[ore0,oi0,ore4,oi4]=rad2ct2fft(tra0,tia0,tra4,tia4);
[ore1,oi1,ore5,oi5]=rad2ct2fft(tra1,tia1,tra5,tia5);
[ore2,oi2,ore6,oi6]=rad2ct2fft(tra2,tia2,tra6,tia6);
[ore3,oi3,ore7,oi7]=rad2ct2fft(tra3,tia3,tra7,tia7);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct8ifft.m  %%%%%%%%%


function
[ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3,ore4,oi4,ore5,oi5,ore6,oi6,ore7,oi7]=rad2ct8ifft(ir0,ii
0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
%  this function calculates the 2-point FFt of eight 32 bit complex inputs
%  the result is a 8-point 16-bit complex output

%Error_mean =  -0.2844 + 0.7659i
%Error_variance =    6.0012
%Avg_no_of_magnitude_errors =     0
%Avg_no_of_sign_errors =    0.0100
%Avg_no_of_mag_and_sign_errors =    7.9900
%Avg_no_of_right_results =     0
%Avg_ratio =    2.0006

[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad2ct4ifft(ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad2ct4ifft(ir1,ii1,ir3,ii3,ir5,ii5,ir7,ii7);

[tra0]=shrega(zeropad(tr0));
[tia0]=shrega(zeropad(ti0));
[tra1]=shrega(zeropad(tr1));
[tia1]=shrega(zeropad(ti1));
[tra2]=shrega(zeropad(tr2));
[tia2]=shrega(zeropad(ti2));
[tra3]=shrega(zeropad(tr3));
[tia3]=shrega(zeropad(ti3));
```

```
[tra4]=shrega(zeropad(tr4));
[tia4]=shrega(zeropad(ti4));

w8_r1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w8_i1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w8_r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w8_i2=[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w8_r3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
w8_i3=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];

tr5=tr5(1:16);
ti5=ti5(1:16);
tr6=tr6(1:16);
ti6=ti6(1:16);
tr7=tr7(1:16);
ti7=ti7(1:16);
clk=1;
resetn=1;

[tra5,tia5]=cplxmul16(tr5,ti5,w8_r1,w8_i1,clk,resetn);
[tra6,tia6]=cplxmul16(tr6,ti6,w8_r2,w8_i2,clk,resetn);
[tra7,tia7]=cplxmul16(tr7,ti7,w8_r3,w8_i3,clk,resetn);

[ore0,oi0,ore4,oi4]=rad2ct2ifft(tra0,tia0,tra4,tia4);
[ore1,oi1,ore5,oi5]=rad2ct2ifft(tra1,tia1,tra5,tia5);
[ore2,oi2,ore6,oi6]=rad2ct2ifft(tra2,tia2,tra6,tia6);
[ore3,oi3,ore7,oi7]=rad2ct2ifft(tra3,tia3,tra7,tia7);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad4ct4.m  %%%%%%%%%

function [ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3]=rad4ct4( ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3)

tmp0 =add40(ir0,ir1);
tmp1 =add40(ir2,ir3);
tmp2 =add40(ii0,ii1);
tmp3 =add40(ii2,ii3);
tmp8 =add40(ir0,ir2);
tmp9 =add40(ir1,ir3);
tmp10= add40(ii0,ii2);
tmp11= add40(ii1,ii3);
tmp4 =sub40(ir0,ir2);
tmp5 =sub40(ii0,ii2);
tmp7 =sub40(ir1,ir3);
```

```
tmp6 =sub40(ii1,ii3);
oi1 =sub40(tmp5,tmp7);
oi2 =sub40(tmp10,tmp11);
ore2= sub40(tmp8,tmp9);
ore3= sub40(tmp4,tmp6);
ore0= add40(tmp0,tmp1);
oi0 =add40(tmp2,tmp3);
ore1= add40(tmp4,tmp6);
oi3 =add40(tmp5,tmp7);

return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad4ct4ifft.m  %%%%%%%%%

function [ore0,oi0,ore3,oi3,ore2,oi2,ore1,oi1]=rad4ct4ifft( ir0,ii0,ir1,ii1,ir2,ii2,ir3,ii3)

tmp0 =add40(ir0,ir1);
tmp1 =add40(ir2,ir3);
tmp2 =add40(ii0,ii1);
tmp3 =add40(ii2,ii3);
tmp8 =add40(ir0,ir2);
tmp9 =add40(ir1,ir3);
tmp10= add40(ii0,ii2);
tmp11= add40(ii1,ii3);
tmp4 =sub40(ir0,ir2);
tmp5 =sub40(ii0,ii2);
tmp7 =sub40(ir1,ir3);
tmp6 =sub40(ii1,ii3);
oi1 =sub40(tmp5,tmp7);
oi2 =sub40(tmp10,tmp11);
ore2= sub40(tmp8,tmp9);
ore3= sub40(tmp4,tmp6);
ore0= add40(tmp0,tmp1);
oi0 =add40(tmp2,tmp3);
ore1= add40(tmp4,tmp6);
oi3 =add40(tmp5,tmp7);

return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct8.m  %%%%%%%%%
```

```
function
[ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3,ore4,oi4,ore5,oi5,ore6,oi6,ore7,oi7]=rad2ct8(ir0,ii0,i
r1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  this function calculates the 2-point FFt of eight 32 bit complex inputs
%  the result is a 8-point 16-bit complex output

%Error_mean =  -0.2844 + 0.7659i
%Error_variance =    6.0012
%Avg_no_of_magnitude_errors =     0
%Avg_no_of_sign_errors =    0.0100
%Avg_no_of_mag_and_sign_errors =    7.9900
%Avg_no_of_right_results =     0
%Avg_ratio =    2.0006

[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad4ct4(ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad4ct4(ir1,ii1,ir3,ii3,ir5,ii5,ir7,ii7);

[tra0]=shrega(zeropad(tr0));
[tia0]=shrega(zeropad(ti0));
[tra1]=shrega(zeropad(tr1));
[tia1]=shrega(zeropad(ti1));
[tra2]=shrega(zeropad(tr2));
[tia2]=shrega(zeropad(ti2));
[tra3]=shrega(zeropad(tr3));
[tia3]=shrega(zeropad(ti3));
[tra4]=shrega(zeropad(tr4));
[tia4]=shrega(zeropad(ti4));

W8_r1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
W8_i1=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
W8_r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
W8_i2=[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1];
W8_r3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
W8_i3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];

tr5=tr5(1:16);
ti5=ti5(1:16);
tr6=tr6(1:16);
ti6=ti6(1:16);
tr7=tr7(1:16);
ti7=ti7(1:16);
clk=1;
resetn=1;

[tra5,tia5]=cplxmul16(tr5,ti5,W8_r1,W8_i1,clk,resetn);
```

```
[tra6,tia6]=cplxmul16(tr6,ti6,W8_r2,W8_i2,clk,resetn);
[tra7,tia7]=cplxmul16(tr7,ti7,W8_r3,W8_i3,clk,resetn);

[ore0,oi0,ore4,oi4]=rad2ct2(tra0,tia0,tra4,tia4);
[ore1,oi1,ore5,oi5]=rad2ct2(tra1,tia1,tra5,tia5);
[ore2,oi2,ore6,oi6]=rad2ct2(tra2,tia2,tra6,tia6);
[ore3,oi3,ore7,oi7]=rad2ct2(tra3,tia3,tra7,tia7);
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  rad2ct8ifft.m  %%%%%%%%%

function
[ore0,oi0,ore1,oi1,ore2,oi2,ore3,oi3,ore4,oi4,ore5,oi5,ore6,oi6,ore7,oi7]=rad2ct8ifft(ir0,ii
0,ir1,ii1,ir2,ii2,ir3,ii3,ir4,ii4,ir5,ii5,ir6,ii6,ir7,ii7)
%function [ore0,oi0,ore1,oi1]=rad2ct2(ir0,ii0,ir1,ii1)
%  this function calculates the 2-point FFt of eight 32 bit complex inputs
%  the result is a 8-point 16-bit complex output

%Error_mean =  -0.2844 + 0.7659i
%Error_variance =    6.0012
%Avg_no_of_magnitude_errors =     0
%Avg_no_of_sign_errors =    0.0100
%Avg_no_of_mag_and_sign_errors =    7.9900
%Avg_no_of_right_results =     0
%Avg_ratio =    2.0006

[tr0,ti0,tr1,ti1,tr2,ti2,tr3,ti3]=rad4ct4ifft(ir0,ii0,ir2,ii2,ir4,ii4,ir6,ii6);
[tr4,ti4,tr5,ti5,tr6,ti6,tr7,ti7]=rad4ct4ifft(ir1,ii1,ir3,ii3,ir5,ii5,ir7,ii7);

[tra0]=shrega(zeropad(tr0));
[tia0]=shrega(zeropad(ti0));
[tra1]=shrega(zeropad(tr1));
[tia1]=shrega(zeropad(ti1));
[tra2]=shrega(zeropad(tr2));
[tia2]=shrega(zeropad(ti2));
[tra3]=shrega(zeropad(tr3));
[tia3]=shrega(zeropad(ti3));
[tra4]=shrega(zeropad(tr4));
[tia4]=shrega(zeropad(ti4));

w8_r1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w8_i1=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];
w8_r2=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
w8_i2=[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
```

```
w8_r3=[1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0];
w8_i3=[0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1];

tr5=tr5(1:16);
ti5=ti5(1:16);
tr6=tr6(1:16);
ti6=ti6(1:16);
tr7=tr7(1:16);
ti7=ti7(1:16);
clk=1;
resetn=1;

[tra5,tia5]=cplxmul16(tr5,ti5,w8_r1,w8_i1,clk,resetn);
[tra6,tia6]=cplxmul16(tr6,ti6,w8_r2,w8_i2,clk,resetn);
[tra7,tia7]=cplxmul16(tr7,ti7,w8_r3,w8_i3,clk,resetn);

[ore0,oi0,ore4,oi4]=rad2ct2ifft(tra0,tia0,tra4,tia4);
[ore1,oi1,ore5,oi5]=rad2ct2ifft(tra1,tia1,tra5,tia5);
[ore2,oi2,ore6,oi6]=rad2ct2ifft(tra2,tia2,tra6,tia6);
[ore3,oi3,ore7,oi7]=rad2ct2ifft(tra3,tia3,tra7,tia7);
return

%%%%%%%%  END  %%%%%%%%%


%%%%%%%%  ramasczt4.m  %%%%%%%%%

function [b]=ramasczt4(a)
% Program to interface my fft engine in place of the standard FFT function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

N=length(a);       %%% ! ! !  MODIFY ! ! !   %%%%%
shift=3;   %%% ! ! !  MODIFY ! ! !   %%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   a_re=real(a);
   a_im=imag(a);

   a_r=zeros(N,32);
   a_i=zeros(N,32);

   for i=1:N
      a_r(i,:)=convert(a_re(i),32,2);
      a_i(i,:)=convert(a_im(i),32,2);
   end
```

```
   [or1,oi1,or2,oi2,or3,oi3,or4,oi4]=czt4(a_r(1,:),a_r(2,:),a_r(3,:),a_r(4,:));

   o_r1=arr2dec(or1,2+shift);
   o_i1=arr2dec(oi1,2+shift);
   o_r2=arr2dec(or2,2+shift);
   o_i2=arr2dec(oi2,2+shift);
   o_r3=arr2dec(or3,2+shift);
   o_i3=arr2dec(oi3,2+shift);
   o_r4=arr2dec(or4,2+shift);
   o_i4=arr2dec(oi4,2+shift);

   b=[o_r1+j*o_i1; o_r2+j*o_i2; o_r3+j*o_i3; o_r4+j*o_i4];
   return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  ramasfft8.m  %%%%%%%%%


function [b]=ramasfft8(a)
% Program to interface my fft engine in place of the standard FFT function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

N=length(a);       %%% ! ! !  MODIFY ! ! !   %%%%%
shift=7;   %%% ! ! !  MODIFY ! ! !   %%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   a_re=real(a);
   a_im=imag(a);

   a_r=zeros(N,32);
   a_i=zeros(N,32);

   for i=1:N
      a_r(i,:)=convert(a_re(i),32,2);
      a_i(i,:)=convert(a_im(i),32,2);
   end

[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8]=czt8(a_r(1,:),a_r(2,:),a_r(
3,:),a_r(4,:),a_r(5,:),a_r(6,:),a_r(7,:),a_r(8,:));

   o_r1=arr2dec(or1,2+shift);
   o_i1=arr2dec(oi1,2+shift);
   o_r2=arr2dec(or2,2+shift);
   o_i2=arr2dec(oi2,2+shift);
```

```
   o_r3=arr2dec(or3,2+shift);
   o_i3=arr2dec(oi3,2+shift);
   o_r4=arr2dec(or4,2+shift);
   o_i4=arr2dec(oi4,2+shift);
   o_r5=arr2dec(or5,2+shift);
   o_i5=arr2dec(oi5,2+shift);
   o_r6=arr2dec(or6,2+shift);
   o_i6=arr2dec(oi6,2+shift);
   o_r7=arr2dec(or7,2+shift);
   o_i7=arr2dec(oi7,2+shift);
   o_r8=arr2dec(or8,2+shift);
   o_i8=arr2dec(oi8,2+shift);
   b=[o_r1+j*o_i1; o_r2+j*o_i2; o_r3+j*o_i3; o_r4+j*o_i4; o_r5+j*o_i5; o_r6+j*o_i6;
o_r7+j*o_i7;
o_r8+j*o_i8];%o_r9+j*o_i9;o_r10+j*o_i10;o_r11+j*o_i11;o_r12+j*o_i12;o_r13+j*o_i1
3;o_r14+j*o_i14;o_r15+j*o_i15;o_r16+j*o_i16;o_r17+j*o_i17;o_r18+j*o_i18;o_r19+j*
o_i19;o_r20+j*o_i20;o_r21+j*o_i21;o_r22+j*o_i22;o_r23+j*o_i23;o_r24+j*o_i24;o_r2
5+j*o_i25;o_r26+j*o_i26;o_r27+j*o_i27;o_r28+j*o_i28;o_r29+j*o_i29;o_r30+j*o_i30;
o_r31+j*o_i31;o_r32+j*o_i32];
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  ramasfft.m  %%%%%%%%%

function [b]=ramasfft(a)
% Program to interface my fft engine in place of the standard FFT function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

N=length(a);       %%% ! ! !  MODIFY ! ! !   %%%%%
shift=2;   %%% ! ! !  MODIFY ! ! !   %%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   a_re=real(a);
   a_im=imag(a);

   a_r=zeros(N,32);
   a_i=zeros(N,32);

   for i=1:N
      a_r(i,:)=convert(a_re(i),32,2);
      a_i(i,:)=convert(a_im(i),32,2);
   end

[or1,oi1,or2,oi2,or3,oi3,or4,oi4]=rad2ct4(a_r(1,:),a_i(1,:),a_r(2,:),a_i(2,:),a_r(3,:),a_i(3,:),
```

```
a_r(4,:),a_i(4,:));%,a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7,:),a_i(7,:),a_r(8,:),a_i(8,:));%,a_
r(9,:),a_i(9,:),a_r(10,:),a_i(10,:),a_r(11,:),a_i(11,:),a_r(12,:),a_i(12,:),a_r(13,:),a_i(13,:),a_
r(14,:),a_i(14,:),a_r(15,:),a_i(15,:),a_r(16,:),a_i(16,:),a_r(17,:),a_i(17,:),a_r(18,:),a_i(18,:),
a_r(19,:),a_i(19,:),a_r(20,:),a_i(20,:),a_r(21,:),a_i(21,:),a_r(22,:),a_i(22,:),a_r(23,:),a_i(23
,:),a_r(24,:),a_i(24,:),a_r(25,:),a_i(25,:),a_r(26,:),a_i(26,:),a_r(27,:),a_i(27,:),a_r(28,:),a_i(
28,:),a_r(29,:),a_i(29,:),a_r(30,:),a_i(30,:),a_r(31,:),a_i(31,:),a_r(32,:),a_i(32,:));
   % 8-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8]=rad2ct8(a_r(1,:),a_i(1,:),
a_r(2,:),a_i(2,:),a_r(3,:),a_i(3,:),a_r(4,:),a_i(4,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7,:),a
_i(7,:),a_r(8,:),a_i(8,:));%,a_r(9,:),a_i(9,:),a_r(10,:),a_i(10,:),a_r(11,:),a_i(11,:),a_r(12,:),a
_i(12,:),a_r(13,:),a_i(13,:),a_r(14,:),a_i(14,:),a_r(15,:),a_i(15,:),a_r(16,:),a_i(16,:),a_r(17,:
),a_i(17,:),a_r(18,:),a_i(18,:),a_r(19,:),a_i(19,:),a_r(20,:),a_i(20,:),a_r(21,:),a_i(21,:),a_r(2
2,:),a_i(22,:),a_r(23,:),a_i(23,:),a_r(24,:),a_i(24,:),a_r(25,:),a_i(25,:),a_r(26,:),a_i(26,:),a_
r(27,:),a_i(27,:),a_r(28,:),a_i(28,:),a_r(29,:),a_i(29,:),a_r(30,:),a_i(30,:),a_r(31,:),a_i(31,:),
a_r(32,:),a_i(32,:));
   % 16-point
[or1,oi1,or2,oi2,or3,oi3,or4,oi4,or5,oi5,or6,oi6,or7,oi7,or8,oi8,or9,oi9,or10,oi10,or11,oi1
1,or12,oi12,or13,oi13,or14,oi14,or15,oi15,or16,oi16]=rad2ct16(a_r(1,:),a_i(1,:),a_r(2,:),a
_i(2,:),a_r(3,:),a_i(3,:),a_r(4,:),a_i(4,:),a_r(5,:),a_i(5,:),a_r(6,:),a_i(6,:),a_r(7,:),a_i(7,:),a_r
(8,:),a_i(8,:),a_r(9,:),a_i(9,:),a_r(10,:),a_i(10,:),a_r(11,:),a_i(11,:),a_r(12,:),a_i(12,:),a_r(1
3,:),a_i(13,:),a_r(14,:),a_i(14,:),a_r(15,:),a_i(15,:),a_r(16,:),a_i(16,:));%,a_r(17,:),a_i(17,:
),a_r(18,:),a_i(18,:),a_r(19,:),a_i(19,:),a_r(20,:),a_i(20,:),a_r(21,:),a_i(21,:),a_r(22,:),a_i(2
2,:),a_r(23,:),a_i(23,:),a_r(24,:),a_i(24,:),a_r(25,:),a_i(25,:),a_r(26,:),a_i(26,:),a_r(27,:),a_
i(27,:),a_r(28,:),a_i(28,:),a_r(29,:),a_i(29,:),a_r(30,:),a_i(30,:),a_r(31,:),a_i(31,:),a_r(32,:),
a_i(32,:));

   o_r1=arr2dec(or1,2+shift);
   o_i1=arr2dec(oi1,2+shift);
   o_r2=arr2dec(or2,2+shift);
   o_i2=arr2dec(oi2,2+shift);
   o_r3=arr2dec(or3,2+shift);
   o_i3=arr2dec(oi3,2+shift);
   o_r4=arr2dec(or4,2+shift);
   o_i4=arr2dec(oi4,2+shift);


    b=[o_r1+j*o_i1 o_r2+j*o_i2 o_r3+j*o_i3 o_r4+j*o_i4];% o_r5+j*o_i5 o_r6+j*o_i6
o_r7+j*o_i7
o_r8+j*o_i8];%o_r9+j*o_i9;o_r10+j*o_i10;o_r11+j*o_i11;o_r12+j*o_i12;o_r13+j*o_i1
3;o_r14+j*o_i14;o_r15+j*o_i15;o_r16+j*o_i16;o_r17+j*o_i17;o_r18+j*o_i18;o_r19+j*
o_i19;o_r20+j*o_i20;o_r21+j*o_i21;o_r22+j*o_i22;o_r23+j*o_i23;o_r24+j*o_i24;o_r2
5+j*o_i25;o_r26+j*o_i26;o_r27+j*o_i27;o_r28+j*o_i28;o_r29+j*o_i29;o_r30+j*o_i30;
o_r31+j*o_i31;o_r32+j*o_i32];
  % 8 point b=[o_r1+j*o_i1 o_r2+j*o_i2 o_r3+j*o_i3 o_r4+j*o_i4 o_r5+j*o_i5
o_r6+j*o_i6 o_r7+j*o_i7
o_r8+j*o_i8];%o_r9+j*o_i9;o_r10+j*o_i10;o_r11+j*o_i11;o_r12+j*o_i12;o_r13+j*o_i1
```

```
3;o_r14+j*o_i14;o_r15+j*o_i15;o_r16+j*o_i16;o_r17+j*o_i17;o_r18+j*o_i18;o_r19+j*
o_i19;o_r20+j*o_i20;o_r21+j*o_i21;o_r22+j*o_i22;o_r23+j*o_i23;o_r24+j*o_i24;o_r2
5+j*o_i25;o_r26+j*o_i26;o_r27+j*o_i27;o_r28+j*o_i28;o_r29+j*o_i29;o_r30+j*o_i30;
o_r31+j*o_i31;o_r32+j*o_i32];
  % 16 point  b=[o_r1+j*o_i1 o_r2+j*o_i2 o_r3+j*o_i3 o_r4+j*o_i4 o_r5+j*o_i5
o_r6+j*o_i6 o_r7+j*o_i7 o_r8+j*o_i8 o_r9+j*o_i9 o_r10+j*o_i10 o_r11+j*o_i11
o_r12+j*o_i12 o_r13+j*o_i13 o_r14+j*o_i14 o_r15+j*o_i15
o_r16+j*o_i16];%o_r17+j*o_i17;o_r18+j*o_i18;%o_r19+j*o_i19;o_r20+j*o_i20;o_r21
+j*o_i21;o_r22+j*o_i22;o_r23+j*o_i23;o_r24+j*o_i24;o_r25+j*o_i25;o_r26+j*o_i26;o
_r27+j*o_i27;o_r28+j*o_i28;o_r29+j*o_i29;o_r30+j*o_i30;o_r31+j*o_i31;o_r32+j*o_i
32];
 %   c=fft(a);

 %b-c'
 return

 %%%%%%%%  END  %%%%%%%%%

%%%%%%%%  reconvert.m  %%%%%%%%%


function [decim]=reconvert(hexad1,bits,intebits);
% function converts a 16-bit binary STRING(NOT ARRAY) into its equivalent binary
number
% for example
% if b=0000011101001011  (BCD)
% then reconvert(b,2) would give a result of 0.1140
fracbits=bits-intebits;
a=[0];
digi=zeros(1,bits);
a=destring(hexad1);
decim=0;
factor=2^(-fracbits);
for j=bits:-1:2
   decim=decim+a(j)*factor;
   factor=factor*2;
end
if a(1)==1
   decim=decim-2^(intebits-1);
end
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  reg16.m  %%%%%%%%%
```

```
function [q]=reg16(d,resetn)
% function [q]=reg16(d,resetn)
% Function simulates the behaviour of a 16-bit register
% when resetn='0' then q=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] irrespective of d
% when resetn='1' then q=d;
if resetn==0
   q=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
else
   q=d;
end;
return

%%%%%%%%  END  %%%%%%%%%
%%%%%%%%  romcoef.m  %%%%%%%%%

for i=1:N
%    figure(i);
   plot(incrfac',abs(cumul(:,i)),'r');
end
hold off
N=4;
for n=1:N
   c(n)=cos(pi*(n-1)*(n-1)/N);
   disp(c(n));
   disp(stringize(convert(c(n),32,2)))
end
disp('sincoef');
for n=1:N
   s(n)=sin(pi*(n-1)*(n-1)/N);
   disp(s(n));
   disp(stringize(convert(s(n),32,2)))
end

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  roufil.m  %%%%%%%%%

function z=roufil(a,b)

z=[];
a
b
lena=length(a);
lenb=length(b);

for i=1:lena
```

```
   i
   tmp=a .* b
   z=[z sum(tmp)]
   a=[a(lena) a(1:lena-1)]
end
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  shrega.m  %%%%%%%%%

function [R0M]=shrega(TR)
% Program to right shift a decimal point in a register by two places to the right with sign
extension and to extend the word to 32 bit length
R0M=[TR(1) TR(1) TR(1) TR(1:29)];
return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  snrvary1.m  %%%%%%%%%


% This program reads data from a text file 'intext.txt' and sends the file data to the IFFT
engine
% and may add noise to the transmitted data before receiving it and passing  it to the FFT
engine
% to decipher its output. The output is then written to a file called 'outtext.txt'.

% Gather FIDs for input and output files
infid= fopen('outtext.txt','r');
outfid=fopen('out.txt','w');

% Noise Measure
%snratio=[-20];
%

snratio=[-20:0];
lengsnr=length(snratio);
% Read Data from input file
[indata,incount]=fread(infid,'bit1');
N=4;
for g=1:lengsnr

% Initialize Outout data
outdata=[];
outsymbol=[];
```

```
snrvalue=snratio(g)
datalen=incount;
rema=mod(incount,2*N);
if (rema~=0)
   indata=[indata;zeros(2*N-rema,1)];
   datalen=datalen+2*N-rema;
end
%indata=randint(datalen/N,1);
batches=datalen/N;
symbollist=[];

% Encoding the input Data...

for i=0:batches-1
  symbollist=[symbollist; getsymbol(indata(N*i+1:N*(i+1)))];
end

for i=0:batches/2-1
 %  i
   z=getformat(symbollist(N*i+1:N*(i+1/2)));
   transmit=ramasifft4(z);
   powx=sum(abs(transmit.*transmit));    %  Power of the transmitted window of data

   pownoise=powx * 10 ^ ( snratio(g) / 10 );   % Noise power calculation

   noise=rand(N,1);

   inputnoise=pownoise * noise;

   receiverinput=transmit+inputnoise;

   receive=ramasfft4(receiverinput);
   p=deformat(receive);
   outsymbol=[outsymbol; p];
   z=getformat(symbollist(N*(i+1/2)+1:N*(i+1)));
   transmit=ramasifft4(z);
   powx=sum(abs(transmit.*transmit));    %  Power of the transmitted window of data

   pownoise=powx * 10 ^ ( snratio(g) / 10 );   % Noise power calculation

   noise=rand(N,1);

   inputnoise=pownoise * noise;
```

```
   receiverinput=transmit+inputnoise;

   receive=ramasfft4(receiverinput);
   p=deformat(receive);
   outsymbol=[outsymbol; p];
end

% finding the nearest point in the given constellation
lenoutsymlist=length(outsymbol);
symout=zeros(lenoutsymlist,1);
for k=1:lenoutsymlist
  symout(k)=getpoint4qam(outsymbol(k));
end


% Decoding the input Data...

for i=0:batches/2-1
  y=symout(N*i+1:N*(i+1));
  outdata=[outdata; getnumber(y)];
end

lenoutdata=length(outdata);
minim=min(lenoutdata,datalen);
[numberoferrors(g),ber(g)]=biterr(abs(indata(1:minim)),abs(outdata(1:minim)));
end

save fft_snr_vary_r4c4

count=fwrite(outfid,outdata,'bit1');
st=fclose('all');
%plot(points,ber);


%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  sub16.m  %%%%%%%%%

function [S]=sub16(a,b)
% This function calculates the difference between two 16-bit numbers and gives the
% result as a 16 bit word avoiding an overflow. So one bit of precision is lost in the
% process.

T=com(b);
test=xor(a(1),b(1));
testbar=~test;
```

```
[tmp,cout]=ad16c(T,a);

term1=test & cout;
term2=testbar & tmp(1);

shiftbit=term1 | term2;
S=[shiftbit tmp];
n=length(S);
S=[S zeros(1,40-n)];
S=S(1:40);

return

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  sub40.m  %%%%%%%%%

function [S]=sub40(a,b)
% This function calculates the difference between two 32-bit numbers and gives the
% result as a 32 bit word avoiding an overflow. So no precision is lost in the
% process.

T=com32(b);
S=add40(a,T);
returnN=8;
inr=rand(1,N);
ini=rand(1,N);

cplx=inr+j*ini;

cplxfft=fft(cplx);
test=ramasfft(cplx);

cplxout=fft(cplxifft);
testout=ramasfft(test);

plot(abs(testout),'kd');
hold on;
plot(abs(testout),'k');

plot(abs(cplxout),'r');
hold off;
mean(test-cplxfft)
max(test-cplxfft)
mean(testout-cplxout)
```

```
max(testout-cplxout)for w=0:0.05:1
   wp=convert(w,16,2);
   wn=convert(-w,16,2);
   w
   corrcoef(wp,wn)
end

%%%%%%%%  END  %%%%%%%%%

%%%%%%%%  zeropad %%%%%%%%%

   function [z]=zeropad(a)
% pads the given input to 32 bits by appending zeros to fill the element
z=[a zeros(1,32-length(a))];
return
%%%%%%%%  END  %%%%%%%%%
```

LIST OF REFERENCES

[1] Prasad, Ramjee; Richard Van Nee, OFDM Wireless Multimedia Communications, Artech House, Boston, 2000.

[2] Chu, Eleanor; Alan George, Inside the FFT Blackbox, CRC Press, Boca Raton, 2000.

[3] Proakis, John G.; Dimitris G. Manolakis, Digital Signal Processing, Prentice Hall of India Private Limited, New Delhi, 2000.

[4] Burrus, C.S. and T.W.Parks, DFT/FFT and Convolution Algorithms Theory and Implementation, John Wiley & Sons, New York, 1985.

[5] Taylor, Fred and Jon Mellot, Hands-On Digital Signal Processing, McGraw-Hill, New York, 1998.

[6] Brown, Stephen and Zvonko Vranesic, Fundamentals of Digital Logic with VHDL Design, McGraw-Hill, New York, 2000

[7] Bellaouar, Abdellatif and Mohamed Ielmasry, Low Power Digital VLSI Design: Circuits and Systems, Kluwer Publishers, Norwell, 1995.

[8] Altera Corporation, “Altera Corporation: The Programmable Solutions Company,” 1995-2002, link: www.altera.com, July 3, 2002

[9] Xilinx Inc, “Xilinx: Programmable Logic Devices, FPGA & CPLD,” 1994-2002, link: www.xilinx.com, July 3, 2002.

[10] Salomon, O.; J. M. Green; H. Klar, “General Algorithms for a Simplified Addition of 2’s Complement Numbers,” IEEE Journal of Solid State Circuits, Vol. 30, No.7, July 1995, pp. 839-844.

[11] Kraniauskas, Peter, ”A Plain Man’s Guide To The FFT,” IEEE Signal Processing Magazine, April 1994, pp. 24-35.

[12] Ochiai, Hideki; Hideki Imai, “On Clipping for Peak Power Reduction of OFDM Signals,” Global Telecommunications Conference, 2000. GLOBECOM ’00. IEEE, Vol. 2, 2000, pp. 7311-735.

[13] Zhao, Yuping; Sven-Gustav Haggman, “BER Analysis of OFDM Communication Systems with Intercarrier Interference,” International Conference on Communication Technology, ICCT ’98 October 22-24, 1998, Beijing, China, pp. S38-02-1 – S38-02 –5.

[14] Wu, Yiyan; William Y. Zou, “Orthogonal Frequency Division Multiplexing: A Multi-Carrier Modulation Scheme,” IEEE Transactions on Consumer Electronics, Vol. 41, No.3, August 1995, pp. 392-399.

[15] Li, Xiaodong; Leonard J. Cimini, “Effects of Clipping and Filtering on the Performance of OFDM,” IEEE Communications Letters, Vol. 2, No. 5, May 1998, pp. 131-133.

[16] Oliver, William D., ”The Singing Tree: A Novel Interactive Musical Interface,” Master’s Thesis, Massachusetts Institute of Technology, 1997.

[17] Oraintara, Soontorn; Ying-Jui Chen; Truong Q. Nguyen, “Integer Fast Fourier Transform,” IEEE Transactions On Signal Processing, Vol. 50, No. 3, March 2002, pp. 607-618.

[18] Waggener, Bill, Pulse Code Modulation Techniques with Applications in Communications and Data Recording, Van Nostrand Reinhold, New York, 1995.

BIOGRAPHICAL SKETCH

Rama Krishna Lolla was born on February 15, 1979, at Machillipatnam, Andhra Pradesh, India. He attended Abhyudaya Cooperative Junior College in Hyderabad, Andhra Pradesh, India, and graduated in 1996. He received his Bachelor of Engineering from Birla Institute of Technology, Ranchi, India, in 2000.