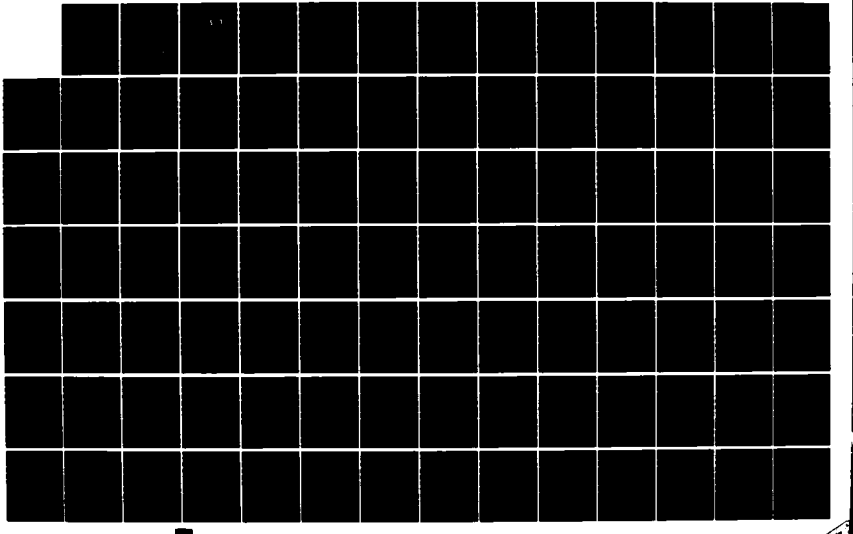
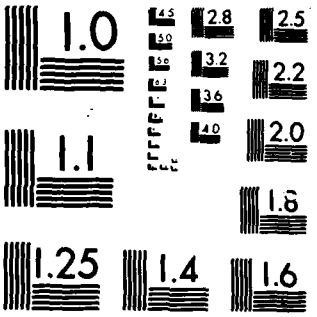
AD-A163 963 ARCHITECTURE AND NUMERICAL ACCURACY OF HIGH-SPEED DFT 1/3
(DISCRETE FOURIER T..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K TAYLOR
UNCLASSIFIED DEC 85 AFIT/GE/ENG/85D-47 F/G 9/1 NL





ARCHITECTURE AND NUMERICAL ACCURACY OI HIGH-SPEED DFT PROCESSING SYSTEMS

Kent Taylor, B.S.E.E.

Captain, USAF

December 1985

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/85D-47

ARCHITECTURE AND NUMERICAL ACCURACY OF HIGH-SPEED DFT PROCESSING SYSTEMS

Kent Taylor, B.S.E.E.

Captain, USAF

December 1985

AFIT GE/ENG/85D-47

ARCHITECTURE AND NUMERICAL ACCURACY OF HIGH-SPEED DFT PROCESSING SYSTEMS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Kent Taylor, B.S.E.E.

Captain, USAF

December 1985

Acknowledgement

I would like to thank my thesis advisor, Captain Richard Linderman, for his guidance and motivation during this research. Lieutenant Colonel Harold Carter and Captain Glenn Prescott provided much needed help in computer programming and digital signal processing applications. My fellow thesis students, Captain Paul Rossbach, Captain James Collins, and Captain Paul Coutee, helped with the many details of research, especially the designs of the VLSI circuits.

Most of all, I wish to thank my wife Anna for her support and understanding throughout this effort.

Kent Taylor

Table of Contents

List of Figures

List of Tables

Abstract

This research examines a very large-scale integrated (VLSI) circuit implementation of the Winograd and Good-Thomas algorithms for computing discrete Fourier Transforms (DFTs) with composite blocklengths. The theoretical background for calculating DFTs in general is developed, before the algorithms of interest are presented in detail. Once the validity of the algorithms is established, a VLSI architecture, which exploits the parallelism and pipelining inherent in the algorithms, is discussed. Winograd processors use both the small and large Winograd algorithms to compute DFTs with blocklengths of 15, 16, and 17. Longer blocklength DFTs (240, 255, 272, and 4080) are computed using a pipeline of Winograd processors, dual-port memories, and an interface processor; the pipeline uses the Good-Thomas Prime Factor Algorithm (PFA). Fault tolerance was included in the initial design of the VLSI architecture. Watchdog processors check both data and addresses of active Winograd processors, while parity checking circuits incorporated in the Winograd processors augment memory error-correction coding (ECC).

The numerical accuracy of the VLSI circuit was determined using a software simulation. The signal-to-noise ratio (SNR) was used as the accuracy metric. The signal was the output of a standard module, which used double-precision arithmetic, while the noise was the difference between the standard and the simulation module. The simulation module used integer arithmetic to exactly mimic operation of the VLSI circuit. The outputs of the standard module were also compared with a direct evaluation of the DFT to verify the standard module did compute a DFT. Results of the comparison between the standard and simulation modules for single-factor DFTs (15, 16, and 17) indicate the VLSI circuit can produce information accurate enough for synthetic aperture radar and other demanding applications.

Architecture and Numerical Accuracy of High-Speed DFT Processing Systems

Chapter 1

Introduction

1.1. Background.

Current radar and image recognition systems require real-time computation of discrete Fourier transforms (DFTs). These applications need spectral information, given by the DFT, on a large number of sample points to obtain the necessary spectral resolution for precise calculation of operations such as correlation and convolution. The DFT is used since only a finite number of sampled values are available, rather than the original analog signal. The DFT should not be computed directly because the number of operations would be proportional to the square of the number of sample points (i.e., $O(N^2)$). Instead, the class of algorithms known as fast Fourier transforms (FFTs) is usually employed, since the number of operations is only proportional to the logarithm of the number of sample points times the number sample points (i.e., $O(N\log N)$). The first popular FFT algorithm, the Cooley-Tukey radix-two algorithm [5], is still widely used today. The algorithm, developed in 1965, takes advantage of symmetry properties within the DFT computation to reduce the number of operations.

Until the advent of very large-scale integrated (VLSI) circuits, most DFT calculations were performed by general-purpose computers or by banks of circuit boards containing medium- and large-scale integrated (MSI and LSI) circuits. The general-purpose computers were used for most applications because they had better accuracy and throughput, unless size and power requirements forced the use of integrated circuits. The loss in throughput and or accuracy when integrated circuits were used meant some

jobs had to be processed off-line, rather than in real time (e.g., synthetic aperture radar and image processing from space vehicles). It is now possible, using VLSI circuits, to put all the required arithmetic and control circuitry necessary to perform DFT computations onto a single chip. However, several single-chip processors may be needed for those applications which require long blocklengths and/or high throughput. In the case of synthetic aperture radar, the throughput and blocklength constraints are so severe, off-line optical processing continues to be used [18].

One method of increasing the throughput is to reduce the number of operations required to compute the DFT. Winograd has shown a class of algorithms (know as Winograd Fourier Transform Algorithms; WFTA) to use the fewest number of multiplications in computing the DFT [29]. Reducing the number of multiplications gives a larger increase in performance than reducing the number of additions because the multiplications are more complicated, requiring several additions to yield the product. Thus, designing an integrated circuit which implements a WFTA would be likely to have high throughput. The remaining question is whether the circuit can provide the necessary accuracy for the radar and image processing applications.

1.2. Problem Statement.

The research presented in this thesis has two goals: 1) develop an architecture for a VLSI circuit which computes DFTs using the WFTA; and 2) determine the numerical accuracy of the VLSI circuit using a software program to simulate the numerical operation of the circuit.

1.3. Scope.

This thesis report is the first of a series of four reports on the research of VLSI circuits implementing a WFTA. This report will focus on the numerical simulation of the

VLSI circuit and the development of the circuit from a system-level viewpoint. The other three thesis reports will cover the following areas:

1) VLSI arithmetic circuitry;

2) VLSI control circuitry;

3) VLSI circuit simulation.

A summary of information contained in the other three reports is presented in the following paragraphs.

1.3.1. VLSI Arithmetic Circuitry. This area of the research is highlighted in the thesis report of Captain Paul Coutee [6]. Captain Coutee discusses the modules necessary to realize the multipliers and the adder/subtractor elements in the VLSI circuit. The design of the modules, including optimizing the area, is presented in detail. Also, the requirements of the arithmetic circuitry, from both a systems viewpoint and a circuit viewpoint, are developed.

1.3.2. VLSI Control Circuitry. Captain Paul Rossbach [22] describes the modules necessary to realize the control portion of the VLSI circuit, including the signals which allow the circuit to compute the DFT and the generation of addresses for both input and output data. A ring counter and a programmed-logic array (PLA) are used to generate the the control signals. The address circuitry uses a read-only memory (ROM) to store the addresses for the input and output data. A special algorithm to reduce the number of transistors in the ROM is presented in detail.

1.3.3. VLSI Circuit Simulation. A functional simulation of the VLSI circuitry is presented in the thesis report of Captain James Collins [4]. Captain Collins discusses the language requirements necessary to simulate operation of a 16-point Winograd

processor in detail, implementing such functions as parity checking and generation, rounding, and scaling. A simulation in the 'C' computer language is given in detail, showing the flow of information through the processor. Also, a description of the 16-point processor is presented using the VHSIC Hardware Description Language (VHDL).

1.4. General Approach.

The general approach to developing the VLSI circuits which perform the WFTA will be to first introduce the theory of how Winograd's algorithms allow DFT computations to be calculated using convolution algorithms. Then, the VLSI circuits will be presented, with block diagrams of individual processors and DFT systems incorporating processors, memories, and host interfaces. Finally, the numerical simulation of the circuits is discussed, including differences between the simulation module and the standard module and results of the simulation.

1.5. Overview of Remaining Chapters.

The remaining chapters in this thesis report will follow the general approach outlined in the previous paragraph. Chapter 2 contains the necessary theory to understand how the algorithms presented in this report can compute DFTs. First, the Cooley-Tukey fast Fourier Transform (FFT) algorithm is given, since it introduces the concept of algorithms which use fewer operations than the direct evaluation of the DFT. Next, the use of Winograd's short convolution algorithm is presented, showing a method of quickly computing a cyclic convolution directly. Then, Rader's prime algorithm can be used to change a DFT calculation into a cyclic convolution computation. The combination of these two ideas gives rise to Winograd's small DFT algorithm. For longer DFT blocklengths, Winograd's large DFT algorithm or the Good-Thomas Prime Factor Algorithm (PFA) may be used with the small Winograd modules.

Chapter 3 has the information on the architecture of the VLSI circuit. Beginning with the Winograd processors, the circuits are presented from a system-level viewpoint, including block diagrams and control signal descriptions. The DFT processor, a system of Winograd processors and associated memory and interface chips, is discussed next. Finally, the characteristics of both the individual processors and the DFT system are presented. Fault tolerance is discussed from a design viewpoint (i.e., incorporating fault tolerance into the original design of the circuits, rather than as an additional item for later development).

Chapter 4 contains the information on the numerical performance of the architecture presented in Chapter 3. First, the metric used to determine the numeric accuracy, the signal-to-noise ratio, is given, as well as some of the main sources of noise (noise being the difference between the results from the standard and simulation modules). Second, the details of the programming are given, especially the requirements of the language to be used and the differences between the standard and simulation modules. Lastly, the results of the simulation are presented, with the average signal-to-noise ratios given for three DFT blocklengths.

Chapter 5 has the results, conclusions, and recommendations for the this report. Future research will be centered on the fabrication and testing of the VLSI circuits; however, some theoretical work should be accomplished (i.e., validation of the 17-point algorithm) and efforts should be made to determine the effects of coefficient wordlength and different types of input data on the numerical accuracy. The appendices contain a development of the 15-point DFT using the large Winograd algorithm, the simulation programs, and the simulation results.

Chapter 2

Theory

2.1. Overview

The material in this chapter describes the algorithms used to compute the Discrete Fourier Transform (DFT) and presents the theoretical background necessary to understand why these algorithms were chosen. The algorithms used are Winograd's Small DFT algorithm, Winograd's Large DFT algorithm, and the Good-Thomas Prime Factor Algorithm (PFA). Winograd's Small DFT algorithm allows efficient computation of DFTs with short blocklengths (e.g., 3, 5, 16, and 17). These short blocklength DFTs are used in Winograd's Large DFT algorithm and the Good-Thomas PFA to compute DFTs with longer blocklengths (e.g., 4080) and non-prime blocklengths (e.g., 15). The information in this chapter is presented in the following order. First, the Cooley-Tukey algorithm for computing DFTs, known as the fast Fourier Transform (FFT), is given. Then, the method of computing DFTs from cyclic convolutions is described. Next, the theory of the Winograd modules is discussed. Finally, the Good-Thomas Prime Factor Algorithm is explained.

2.2. The Cooley-Tukey FFT Algorithm.

The DFT is a means of describing the discrete frequency components of a finite sequence of values [17]. The DFT may be viewed as the Fourier Transform of an infinite, periodic sequence (where the original sequence is one period of the infinite sequence) or as sampled values of the Z transform of the original finite sequence. The DFT can be expressed using a summation form, as shown in (2-1).

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{kn} \qquad k = 0, 1, \cdots N-1 \tag{2-1}$$

where $W_N = e^{-j(2\pi/N)}$

This direct evaluation of the DFT requires $4N^2$ real multiplications and $N(4N-2)$ real additions, if the N input points are complex samples.. As N grows, the number of operations grows as N^2. Most algorithms which reduce the number of operations in DFT computations take advantage of two properties of the coefficients used in the direct form of the DFT. The first property is conjugate symmetry, as shown in (2-2).

$$W_N^{k(N-n)} = (W_N^{kn})^* \tag{2-2}$$

Conjugate symmetry reduces the number of operations by about one-half. Also, certain values of the product kn yield coefficients of 0 or 1, which are called trivial coefficients. The other property which FFT algorithms exploit is periodicity, as shown in (2-3).

$$W_N^{kn} = W_N^{k(n+N)} = W_N^{(k+N)n} \tag{2-3}$$

Periodicity provides a greater reduction in the number of operations than does the symmetry property.

The FFT algorithm achieves the dramatic decrease in the number of operations by decomposing the DFT computation into successively smaller computations. The decomposition may be performed in either the time (decimation in time) or the frequency (decimation in frequency) domain. Usually, the decimation is performed when N is an even integer. Then, the single summation in (2-1) can be separated into two summations, one for even integers and one for odd integers (2-4).

$$X(k) = \sum_{n\ even} x(n)\, W_N^{nk} + \sum_{n\ odd} x(n)\, W_N^{nk} \tag{2-4}$$

The symmetry and periodicity properties described previously can be applied to show the two summations in (2-4) each correspond to an $N/2$-point DFT. Thus, the $N/2$-point DFT is computed once, then combined with itself to give an N-point DFT. The decimation process is repeated until only 2-point transforms are computed. The number of decimations is: $log_2 N$; therefore, the number of multiplications required by the decimation-in-time FFT algorithm to compute the DFT is: $N/2$.

Even though the number of operations has decreased by $2N/\log_2 N$, the coefficients (W_N) must be computed for each of the $N/2$ points. Also, only two data words may be read into the arithmetic circuitry for each multiplication/addition operation (often called a "butterfly"). These two characteristics usually provide the most severe limitation of the FFT algorithms, that of input/output (I/O) bandwidth. The I/O bandwidth relates to the rate at which data may be read into and out of memory. Thus, even with the reduction in the number of operations given by the FFT algorithms, there are still some problems which limit the usefulness of the algorithms.

2.3. Discrete Fourier Transforms From Cyclic Convolutions.

One of the most frequent applications of the DFT is to compute the convolution of two finite sequences. The convolution is a result of passing an input signal through a linear filter. The mathematical representation of a filtering operation is expressed in the time domain as a convolution of the input signal with the impulse response of the filter. Often, it is easier to perform the computations in the frequency domain, where the filtering operation is a multiplication of the spectrum of the input signal with the frequency response of the filter. The resulting product can be inverse-transformed for time-domain analysis. Since only a finite number of sampled values of the original analog signal are available, the DFT is used, rather than the continuous-time Fourier transform. This method of computing convolutions is chosen because computation of the DFT usually

involves fewer operations (multiplications and additions) than direct evaluation of the convolution itself. However, this relationship may be reversed; a cyclic convolution may be used to compute a DFT if the convolution algorithm requires fewer operations than the DFT algorithm. Winograd [29] has shown such algorithms do exist.

2.3.1. Winograd's Short Convolution Algorithm. The following development of methods to compute cyclic convolutions was adapted from Blahut [3] and Winograd [29]. A cyclic convolution may be written as:

$$s(x) = g(x)\, d(x) \qquad mod\,[\, m(x)\,] \tag{2-5}$$

where the degree of g(x) and d(x) is (N - 1) and $m(x) = x^N - 1$.

The coefficients of s(x) are expressed as:

$$s_i = \sum_{k=0}^{N-1} g_{((i-k))}\, d_k \qquad i = 0, 1, \cdots N-1 \tag{2-6}$$

where the double parentheses denote modulo N arithmetic.

This may be broken into several smaller computations by factoring m(x) into K relatively prime polynomials (i.e., those without any common factors). The residues of the g(x) and d(x) for each of the K factors are computed.

$$g^{(k)}(x) = \mathbf{R}\ \ g(x)\ \ using\ m^{(k)}(x) \tag{2-7a}$$

$$d^{(k)}(x) = \mathbf{R}\ \ d(x)\ \ using\ m^{(k)}(x) \tag{2-7b}$$

where $\mathbf{R}\ \ x\ \ $ represents taking the residue of x.

Then, the kth residue of s(x) may be found by:

$$s^{(k)}(x) = g(x)\, d(x) \quad mod\, [\, m^{(k)}(x)\,] \tag{2-8}$$
$$= \mathbf{R}\,[\, g^{(k)}(x)\;\; d^{(k)}(x)\,] \quad mod\;\, [\, m^{(k)}(x)\,]$$

Finally, s(x) is found by combining the K residues:

$$s(x) = \sum_{k=0}^{K-1} s^{(k)}(x)\, a^{(k)}(x) \tag{2-9}$$

The Winograd short convolution algorithm may be expressed using matrix notation [3]; this allows a more compact representation of the Winograd convolution.

$$\mathbf{s} = \mathbf{C}\,[\,(\mathbf{A}\,\mathbf{g}) \cdot (\mathbf{B}\,\mathbf{d})\,] \tag{2-10}$$
$$= \mathbf{C}\,[\,\mathbf{G} \cdot \mathbf{D}\,]$$
$$= \mathbf{C}\,\mathbf{S}$$

where

$\mathbf{g} \equiv g(x)$ in (2-5)

$\mathbf{d} \equiv d(x)$ in (2-5)

$\mathbf{A} \equiv M(N) \times N$ matrix of the residues of g(x)

$\mathbf{B} \equiv M(N) \times N$ matrix of the residues of d(x)

$\mathbf{C} \equiv N \times M(N)$ matrix of coefficients for the residues of S

$\mathbf{S} \equiv M(N)$ vector of the residues of s(x)

$M(N) \equiv$ the number of multiplications required for an N-point convolution

The form of (2-10) may be massaged to yield three matrices on the right hand side of the equation: a pre-addition matrix, a post-addition matrix, and a diagonal multipli-

cation matrix. The pre-addition and post-addition matrices are composed of zeroes and ones (the ones may be positive or negative). The multiplication matrix is a diagonal matrix of coefficients. These coefficients may be computed and stored before they are needed. The advantages of the matrix form of the short Winograd convolution algorithm will become clear when the small Winograd DFT module is discussed.

One of the reasons for computing the cyclic convolution directly is to reduce the number of operations, especially the number of multiplications. Using the direct method of computing the cyclic convolution (2-5), one needs $2N^2$ real multiplications (assuming complex input data), whereas Winograd's algorithm requires $2\sum_{k=0}^{K-1}\{deg\ m^{(k)}(x)\}^2$ multiplications. For example, the linear convolution of a real 3-point vector with a real 2-point vector requires six multiplications using the direct method and five multiplications using Winograd's method [3]. Although the savings of the number of multiplications in the example is not great, it illustrates the idea. Thus, by using Winograd's small convolution algorithm, the number of multiplications is reduced. However, we still must show the number of operations required to compute the DFT using a cyclic convolution is less than the number of operations required to compute the DFT using one of the Fast Fourier Transform (FFT) algorithms.

2.3.2. Rader's Prime Algorithm. Rader's prime algorithm is the link between cyclic convolutions and DFTs [20]. An N-point DFT may be computed using a cyclic convolution of length (N-1) and index scrambling if N is a prime number. First, find a primitive element π of a finite field of N elements (this finite field is referred to as a Galois field and is denoted by GF(N)). Each integer in GF(N) can be written as a unique power of π. Now, the DFT

$$V_k = \sum_{i=0}^{N-1} \omega^{ik} v_i \quad k = 1, 2, \cdots N-1 \tag{2-11}$$

can be rewritten by breaking out the zero frequency and zero time components:

$$V_k = v_0 + \sum_{i=1}^{N-1} \omega^{ik} v_i \qquad k = 0, 1, \cdots N-1 \tag{2-12a}$$

$$V_0 = \sum_{i=0}^{N-1} v_i \tag{2-12b}$$

This "breaking out" was done since zero cannot be expressed as a power of the primitive element π. Let r(i), defined on {1, 2, ... N-1}, be a unique mapping of i, also defined on {1, 2, ... N-1}, such that:

$$\pi^{r(i)} = i$$

Thus, r(i) is simply a permutation of i. Now, (2-12b) can be expressed as:

$$V_{\pi^{r(k)}} = v_0 + \sum_{i=1}^{N-1} \omega^{\pi^{r(i)+r(k)}} v_{\pi^{r(i)}} \tag{2-13}$$

or, substituting l = r(k) and j = N - 1 - r(i):

$$V'_l = v_0 + \sum_{j=1}^{N-1} \omega^{\pi^{l+j}} v'_j \tag{2-14}$$

where V' and v' are scrambled output and input sequences, respectively.

This last equation (2-14) can be recognized as a cyclic convolution between v'_j and ω^{π^j} (reference (2-5)). The cyclic convolution can be evaluated using the methods described in paragraph 2.3.1. This procedure, Rader's prime algorithm coupled with Winograd's

small convolution algorithm, gives rise to the Winograd Small DFT algorithm.

2.4. Winograd DFT Modules.

Information from the previous section showed how to compute DFTs of prime length from cyclic convolutions. The reason for doing so was to reduce the number of operations, especially the number of multiplications, involved in the DFT computation. This section deals with the methods for computing DFTs of lengths 15, 16, and 17, using the general approach described in paragraph 2.2. First, Winograd's small DFT algorithm will be described for DFTs whose blocklength is either a prime or a power of a prime. Then, the fitting together of small Winograd modules to form a large Winograd module will be discussed. Together, both types of Winograd modules will be used in the Good-Thomas PFA to create DFTs with still longer blocklengths (reference paragraph 2.4.).

2.4.1. Winograd's Small DFT Algorithm. Constructing a DFT using Winograd's small DFT algorithm requires knowledge of Rader's prime algorithm and Winograd's short convolution algorithm. This small DFT algorithm provides an efficient mechanism for calculating DFTs with short blocklengths; in this case, the blocklengths of interest are 3, 5, 16, and 17. For the DFTs whose blocklength is a prime number (e.g., 3, 5, and 17), Winograd's small DFT algorithm has three steps:

1) Change the DFT to a cyclic convolution using Rader's prime algorithm.

2) Compute the cyclic convolution using Winograd's short convolution algorithm.

3) Incorporate the scrambling required by Rader's prime algorithm by permuting the rows of the post-addition matrix and the columns of the pre-addition matrix.

The 16-point DFT cannot be computed using this method since 16 is a power of 2. Winograd's small DFT algorithm requires three steps for DFTs whose blocklength is a

power of 2:

1) Compute a $2^{(m-1)}$-point DFT on the even indices.

2) Compute a $2^{(m-2)}$-point DFT, preceded by $2^{(m-2)} - 1$ complex multiplications, on the odd indices.

3) Compute two polynomial products modulo $x^{n/8} + 1$ (an irreducible polynomial).

Both of the above algorithms are presented in much greater detail in Blahut [3] and in McClellan and Rader [16]. The equations describing the Winograd small DFT algorithm for DFTs of lengths 3, 5, 16, and 17 can be found in several references [3, 7, 16, 25, 29]. Table 2-1 shows the number of operations required for each DFT, using the small Winograd DFT algorithm; entries are given for Cooley-Tukey radix-2 algorithm for DFTs of lengths 4, 8, and 16 for comparison.

The entries in Table 2-1 are given for complex input data. The entries for the Winograd algorithm include trivial multiplications (by $\pm$ 1 or $\pm$ j). The Winograd entries in Table 2-1 were tabulated in Winograd [29] and Blahut [3]. The Cooley-Tukey entries were tabulated in Blahut [3]. As seen in the direct comparisons for the blocklengths of 4, 8, and 16, the small Winograd DFT algorithm requires a similar number of additions as the Cooley-Tukey radix-2 FFT, but substantially fewer multiplications. It is this savings in multiplications which brought about the interest in using

Comparison of Short Blocklength DFT Algorithms

DFT Size	Small Winograd		Cooley-Tukey Radix-2	
	multiplies	adds	multiplies	adds
3	6	12	(a)	(a)
4	8	16	16	24
5	12	34	(a)	(a)
8	16	52	48	72
16	36	148	128	192
17	72	314	(a)	(a)

(a) DFT size is not a power of two.

Table 2-1

the small Winograd DFT algorithm. These savings in the number of multiplications will become even greater when the large Winograd DFT algorithm is used.

2.4.2. Winograd's Large DFT Algorithm. Winograd's small DFT algorithm yields computationally efficient DFT modules; however, most applications require blocklengths which are much longer than those given by Winograd's small DFT algorithm. Winograd's large DFT algorithm does this by combining relatively prime small Winograd modules into an operation which yields a blocklength equal to the product of the blocklengths of the small Winograd modules. There are five steps in constructing a large Winograd module from small Winograd modules:

1) Scramble the inputs using the Chinese Remainder Theorem;

2) Nest the pre-additions;

3) Nest the multiplications;

4) Nest the post-additions;

5) Unscramble the outputs using the Chinese Remainder Theorem.

2.4.2.1. Index Mapping. Understanding the Chinese Remainder Theorem (C.R.T.) is essential to building a large Winograd module. The C.R.T requires the factors which comprise the composite blocklength to be mutually prime. As a brief example, consider a DFT of length fifteen (15). Fifteen can be written as a product of two mutually prime factors (three and five); thus:

$$N = 15 \tag{2-15}$$
$$= 3 \times 5 = n_1 \times n_2$$

where n_1 and n_2 are the two factors.

The input mapping is described by the equations:

$$k_1 = (N' \times k) \bmod n_1$$

$$k_2 = (N'' \times k) \bmod n_2$$

$$(N' \times n_1 \times k) + (N'' \times n_2 \times k) = 1$$

where

k_1 indicates the row of the input array

k_2 indicates the column of the input array

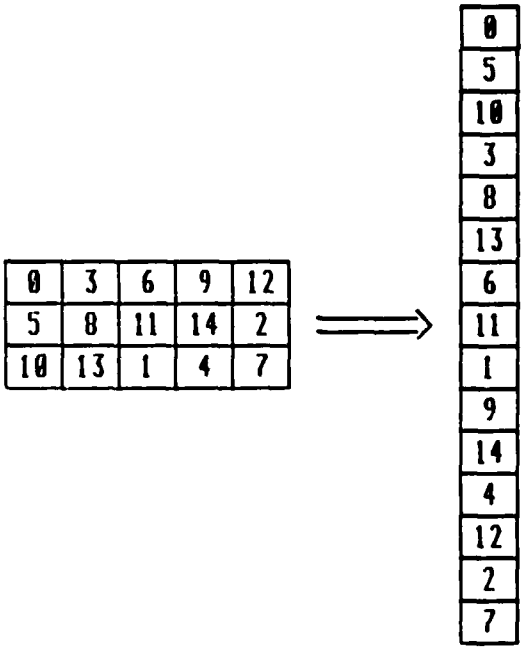


Figure 2-1 Input Mapping for 15-Point Large Winograd Module

For the output mapping, a two-dimensional array is constructed, using n_1 rows and n_2 columns.

$$i_1 = n \bmod n_1$$

$$i_2 = n \bmod n_2$$

where

i_1 indicates the row

i_2 indicates the column

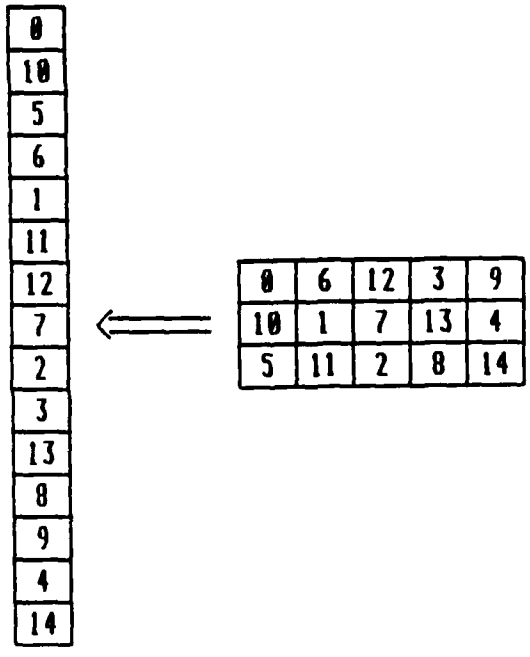


Figure 2-2 Output Mapping for 15-Point Large Winograd Module

These two-dimensional maps are changed back into one dimension by stacking the columns (reference Figures 2-1 and 2-2).

2.4.2.2. Nesting of Additions and Multiplications. The nesting of the multiplications and additions in the large Winograd DFT algorithm is accomplished using the associative property of Kronecker products. The following example indicates the use of Kronecker products:

$$N = N_1 \times N_2$$

$$\mathbf{W} = \text{matrix representation of DFT of N}$$

$$= \mathbf{C}\,\mathbf{B}\,\mathbf{A}\,\underline{x} = \mathbf{W}\,\underline{x}$$

$$\mathbf{W}_1 = \text{matrix representation of DFT of N sub 1}$$

$$= \mathbf{C}_1\,\mathbf{B}_1\,\mathbf{A}_1$$

$$\mathbf{W}_2 = \text{matrix representation of DFT of N sub 2}$$

$$= \mathbf{C}_2\,\mathbf{B}_2\,\mathbf{A}_2$$

where

$\mathbf{A}, \mathbf{A}_1, \mathbf{A}_2 \equiv$ pre-addition matrices

$\mathbf{B}, \mathbf{B}_1, \mathbf{B}_2 \equiv$ multiplicative matrices

$\mathbf{C}, \mathbf{C}_1, \mathbf{C}_2 \equiv$ post-addition matrices

$\underline{x} \equiv$ input vector (N elements)

The output vector, $\underline{X}$, may be written using Kronecker products:

$$\underline{X} = (\mathbf{W}_1 \otimes \mathbf{W}_2)\,\underline{x}$$

$$= (\mathbf{C}_1\,\mathbf{B}_1\,\mathbf{A}_1) \otimes (\mathbf{C}_2\,\mathbf{B}_2\,\mathbf{A}_2)\,\underline{x}$$

$$= (\mathbf{C}_1 \otimes \mathbf{C}_2)\,(\mathbf{B}_1 \otimes \mathbf{B}_2)\,(\mathbf{A}_1 \otimes \mathbf{A}_2)\,\underline{x}$$

One of the advantages of using the large Winograd DFT algorithm is that all like operations are combined into a single matrix (i.e., the pre-additions are combined into one matrix of pre-additions and the same for the multiplications and post-additions). Combining the multiplications into a single matrix saves a considerable number of operations [3, 14]. Since the multiplication matrices are diagonal, the total number of multiplications required is simply the product of the number of multiplications required for each small Winograd module which comprises the large Winograd module (i.e., the number of multiplications is equal to the number of diagonal elements in $\mathbf{B}_1$ times the number of diagonal elements in $\mathbf{B}_2$). Figure 2-3 shows the combination of the pre-additions and the post-additions into single matrices, while Figure 2-4 shows the nesting of the multiplications inside the additions.

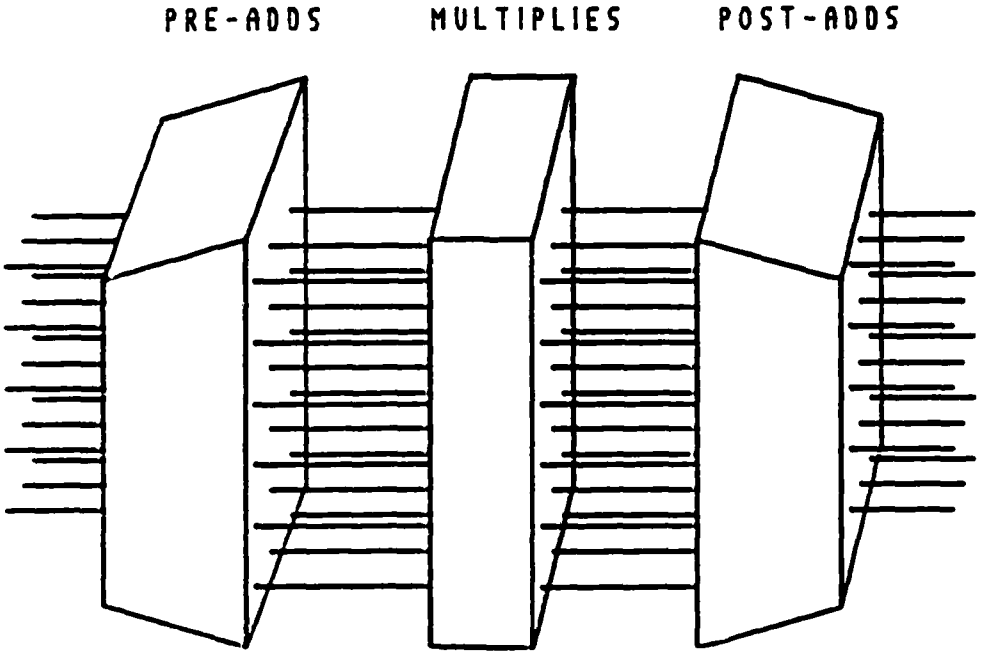


Figure 2-3 Nesting of Additions in 15-Point Large Winograd Module

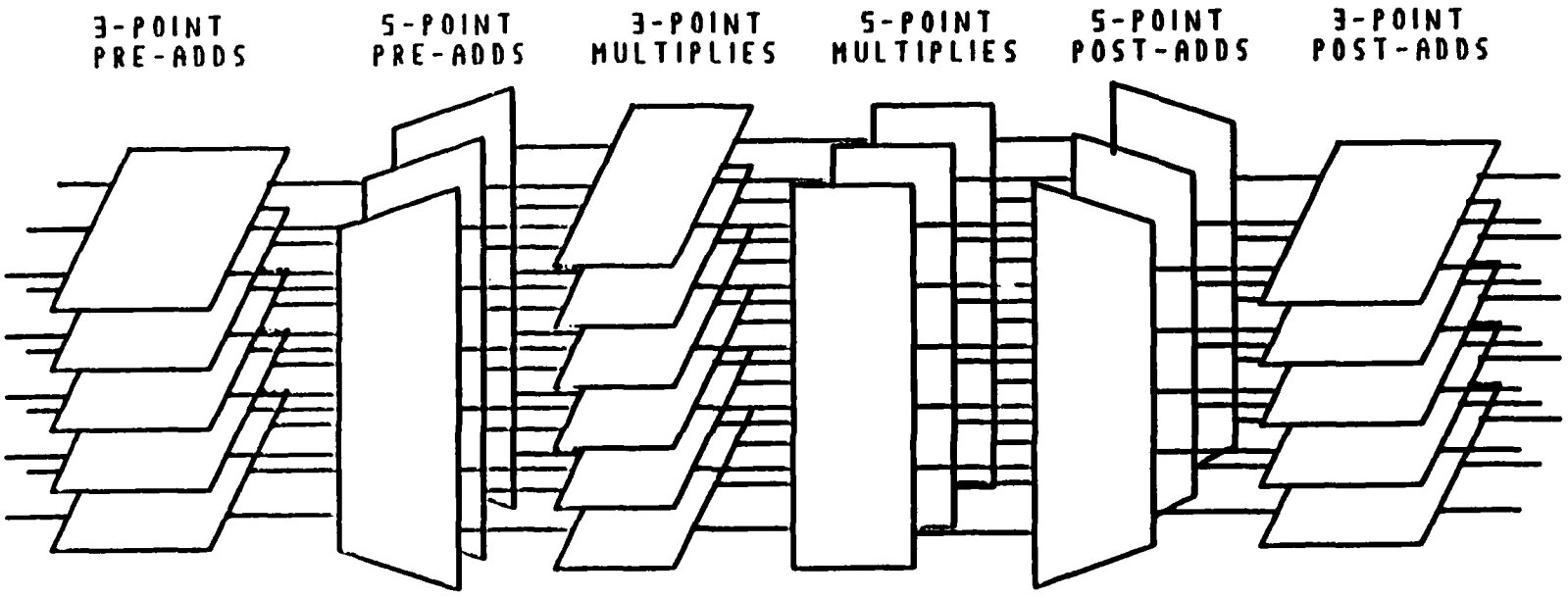


Figure 2-4 Nesting of Multiplications

2.5. Good-Thomas Prime Factor Algorithm.

The Winograd modules, discussed in the previous section, provide an efficient means of computing short- and medium-length DFTs. One of the drawbacks of using the large Winograd modules is the matrices become large and unwieldy, imposing serious memory constraints for DFT computation (e.g., a 255-point transform computed using the large Winograd algorithm requires a multiplication matrix of size 1280). The Good-Thomas prime factor algorithm (PFA) requires more overall multiplications than the large Winograd algorithm. But, since the DFT is broken into several component parts, the total number of multiplications is resolved into several manageable subproblems (e.g., a 255-point DFT computed using the Good-Thomas PFA requires multiplication matrices of size 18 and 36 for the 15-point module and the 17-point module, respectively; reference Table 2-2 for total number of operations). Also, the structure of the Good-Thomas PFA lends itself to pipelined architectures in circuit design (reference Chapter 3), which means the pre-addition matrix may be working on one problem while the multiplication and post-addition matrices are working on the previous problem (this gives better utilization of arithmetic resources). Putting together a PFA algorithm involves:

1) Creating input and output maps of the indices;

2) Choosing efficient DFT modules whose blocklengths are mutually prime (i.e., no common factors).

The index mapping is identical to that used in the large Winograd algorithm since the mapping routine for the PFA also uses the Chinese Remainder Theorem (C.R.T.). The index mapping is performed prior to the first module and after the last module. For example, a 1080-point transform has factors 15, 16, and 17. One possible sequence of operations is described below:

1) Perform the input mapping;

2) Compute 255 16-point DFTs;

3) Compute 272 15-point DFTs;

4) Compute 240 17-point DFTs;

5) Perform the output mapping.

Steps 2, 3, and 4 can rearranged to fit any order.

The short DFT modules may be designed using any algorithm (e.g., Cooley-Tukey radix-2, Winograd small, Winograd large, etc.); the only constraint is the blocklengths of the modules must have no common factors. Winograd modules are well-suited for use within the Good-Thomas PFA since small Winograd modules have blocklengths which are either prime or a power of a prime, while large Winograd modules also require mutually prime factors. The idea behind the Good-Thomas PFA is to change a one-dimensional DFT into an p-dimensional DFT, where p is the number of mutually prime factors (N_1, N_2, ... N_p) which compose the overall blocklength, N. In the two-factor case (i.e., $N = N_1 \times N_2$), N_1 N_2-point DFTs are computed, then N_2 N_1-point DFTs. The order of computing the DFTs does not alter the total number of operations (additions and multiplications). The number of multiplications will be greater than or equal to more than the number of multiplications required by a large Winograd module of equal length. However, the number of multiplications for each block is fewer than the number required for a large Winograd DFT (if small Winograd DFT modules are used in the PFA). Table 2-2 shows a comparison between the Good-Thomas PFA (using small Winograd modules) and the large Winograd DFT algorithm; the entries for Table 2-2 are for complex input data.

The number of multiplications does not include trivial multiplications (by ± 1 or $\pm j$). Entries for Table 2-2 were computed using equations from Kolba and Parks [14]. While the number of multiplications needed for the PFA is approximately twice that required for the large Winograd, the number of additions required for the Good-Thomas PFA is less. Other algorithms, such as the Cooley-Tukey radix-2, require many more multiplications than the PFA (e.g., a 256-point DFT for complex inputs using the

Comparison of Long Blocklength DFT Algorithms				
DFT	Good-Thomas PFA		Large Winograd	
Size	multiplies	adds	multiplies	adds
15	50	81	34	81
240	1100	4812	632	5136
255	1900	7464	1280	8406
272	1640	7540	1280	8168
4080	31148	157164	23312	189048

Table 2-2

Cooley-Tukey radix-2 requires 2048 real multiplications). Figure 2-5 shows a 15-point DFT computed using the PFA; five 3-point DFTs are calculated, then three 5-point DFTs.

2.6. Summary

The material in this chapter has shown how to reduce the number of operations necessary to compute a DFT. First, the class of FFT algorithms was examined. The FFT algorithms reduced the number of operations by exploiting the conjugate symmetry and periodicity properties of the DFT computation. However, the I/O bandwidth of the FFT was poor due to frequent memory references for both data and coefficients. Thus, a different class of algorithms, developed by Winograd, was analyzed. The Winograd algorithms, which have been shown to use the fewest number of multiplications for DFT computation [29], were developed using cyclic convolutions to compute DFTs with short blocklengths. For longer blocklengths, or for those blocklengths which are not prime, the large Winograd and Good-Thomas PFA were shown to be efficient algorithms for DFT computation. The material in the next section shows how to implement these algorithms in a VLSI architecture.

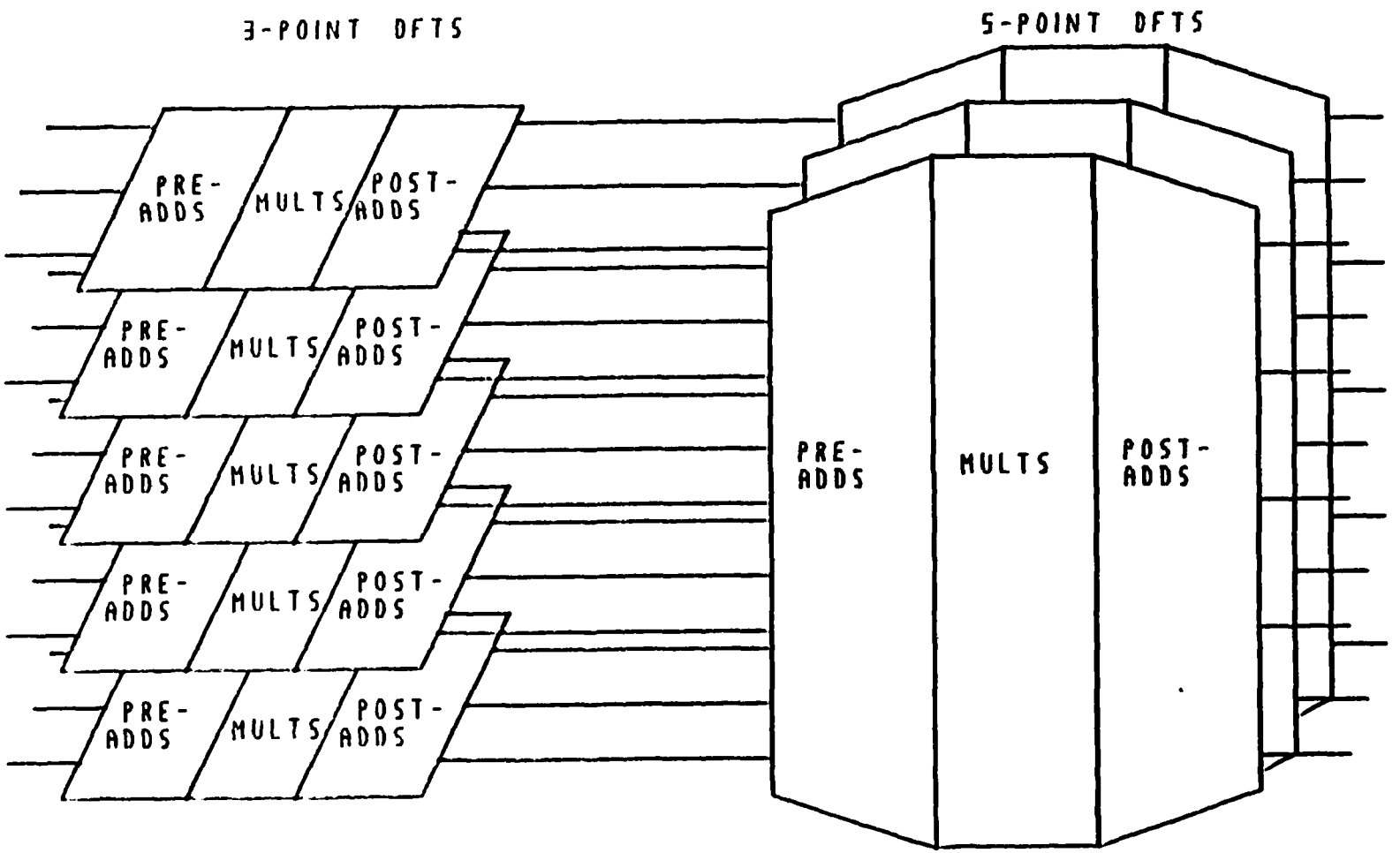


Figure 2-5 15-Point DFT Using Good-Thomas PFA

Chapter 3
VLSI Implementation

3.1. Overview

The material from the previous chapter showed the algorithms used to compute the DFT. Now, we need an architecture which can implement those algorithms in a very large-scale integrated (VLSI) circuit. VLSI circuits lend themselves to regular, parallel structures [15]. The matrix form of the Winograd algorithms map easily into VLSI circuits, with regular structures of adder/subtractors and multipliers (although determining the routing between the elements is not a trivial task). Also, the structure of the Winograd algorithm lends itself to pipelining, the ability to work on more than one problem at a given time. The information in this chapter is provided for both the 16-point Winograd processor and the 4080-point PFA processor. First, the overall layout, signal flow, and physical characteristics for both circuits are given. Then, fault tolerance is presented from a system-design viewpoint. Finally, the computational throughput of the PFA system is described.

3.2. Winograd Processors.

The signal flow, circuit layouts, and physical characteristics will be discussed for the 16-point Winograd processor (the 16-point processor being chosen as being representative of the Winograd processors). The differences for the 15-point and 17-point chips relate to the internal data representation (30 bits for the 15-point and 34 bits for the 17-point versus 32 bits for the 16-point) and the number of sign extensions required to prevent arithmetic overflow (4 sign extensions in the 15-point and 5 in the 17-point versus 3 sign extensions in the 16-point). The internal data representation will change the time to compute a DFT (reference paragraph 3.5.). The need for sign extensions to

prevent arithmetic overflow is discussed in more detail in paragraph 4.2.2.

The data flow for the 16-point chip is shown in Figure 3-1.

As seen in Figure 3-1, the circuits of the 16-point processor may be grouped into three categories:

1) Input/Output (I/O);

2) Control;

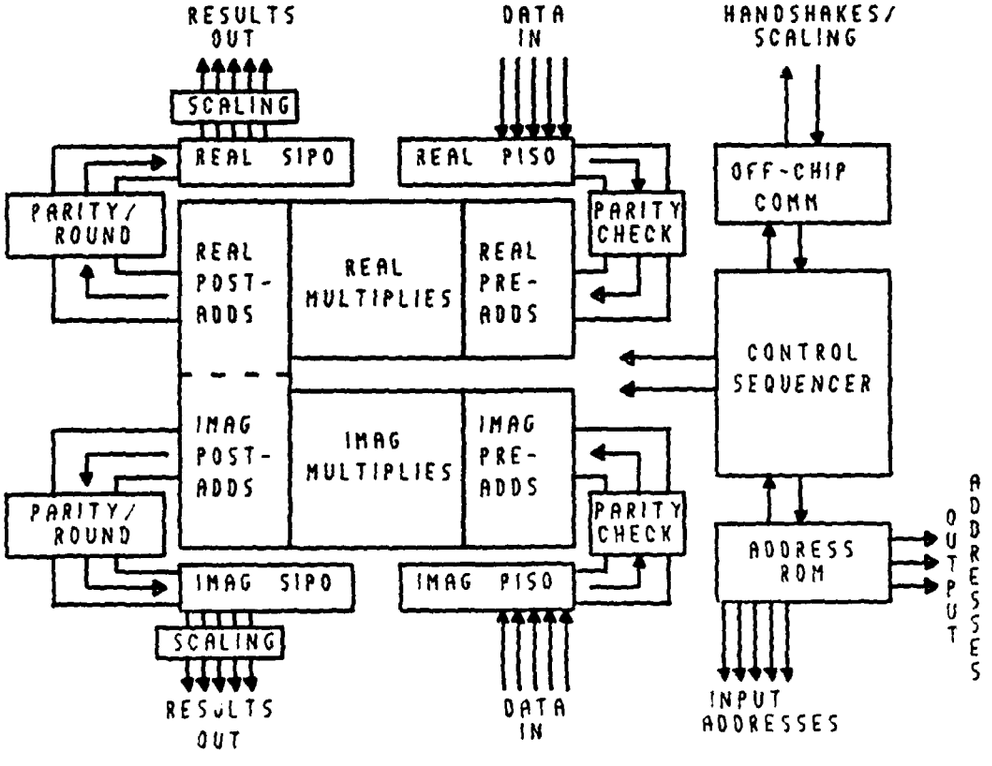


Figure 3-1 16-Point Winograd Processor

3) Arithmetic.

The Parallel In/Serial Out (PISO) registers take inputs from the data memory, while the Serial In/Parallel Out (SIPO) registers send results to the data memory; there are two sets of each type of register, one set for real data/results and one set for imaginary data/results. Also, there is a 4-word buffer which holds the 12-bit addresses for the data memory. There are two types of control circuits; one type is for on-chip control of timing and address generation, while the other is for off-chip communication (scaling control, handshake signalling, etc.). The arithmetic circuits reflect the three components of the Winograd modules: a pre-addition matrix, a post-addition matrix, and a multiplication matrix. Adder/subtractors and multipliers compose most of the arithmetic circuitry; there are also several reset circuits for clearing intermediate results before a new 16-point DFT is started. The signal flow through the 16-point processor is described below:

1) The host processor sends an OPERATE signal to the 16-point processor, so the processor may begin computing DFTs;

2) Sixteen 24-bit data words are loaded from the input memory to the real and imaginary PISO registers;

3) After all sixteen data words are loaded, the PISO registers latch the words into the serial output portion of each register;

4a) The input data words then are serially shifted into the arithmetic circuitry, after passing through a parity check cell which flags any parity errors on the input data;

4b) While the first set of data words are being shifted out serially, the next set of data words are being loaded, in parallel, into the PISO register from the data memory (reference paragraph 3.2.1.).

5) The data pass through the matrix of pre-adders, multipliers, and post-adders (this is where the 16-point DFT is computed);

6) Before being sent to their respective SIPO registers, the real and imaginary results are sent through the parity/rounding cell (which rounds the 32-bit arithmetic results to 23-bit output results and computes a parity bit to be appended to the 23-bit result);

7) After all twenty-three data bits and the single parity bit have been shifted into the SIPO registers, the result words are latched into the parallel portion of each register;

8) The 24-bit results are shifted out of the SIPO register, through the scaling cell (which checks the most significant seven bits of each output data word to find the smallest number of sign extensions for all 4080 words), and out to the output memory.

9) This 16-point processor continues to compute DFTs until it has exhausted all the addresses for that particular DFT blocklength (i.e., 255 16-point DFTs will be computed in the 4080-point PFA system).

Each type of circuitry for the Winograd processors is discussed in greater detail in the following paragraphs.

3.2.1. Input/Output Circuitry. There are two types of I/O circuitry, one for data and one for addresses. The data I/O circuitry are the PISO and SIPO registers. Each type of data register has master-slave two flip-flops in each cell. One flip-flop is used to store the input data, while the other is used to store the output data. For example, the input flip-flops in each cell of the PISO array hold the input data as each 24-bit word is shifted into the PISO register. When the latch signal is activated, the bits in the input flip-flops are copied into the output flip-flops. Then, the outputs may be shifted out (serial shift out) while a new set of inputs is being shifted in (parallel shift in). Figures 3-2a, 3-2b, and 3-2c show the sequence of shifting in, latching, and shifting out for the PISO register. The operation of the SIPO register is similar, except the input shifting is done serially while the output shifting is done in an parallel fashion.

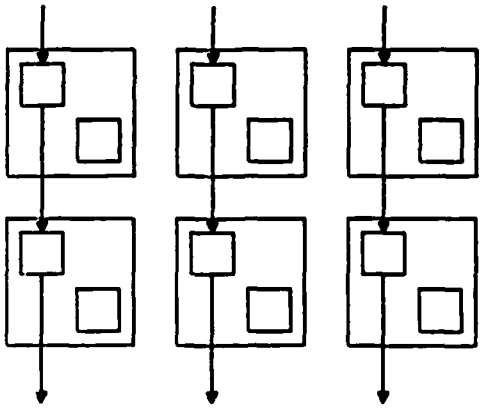


Figure 3-2a PISO Cell--Parallel Shift In

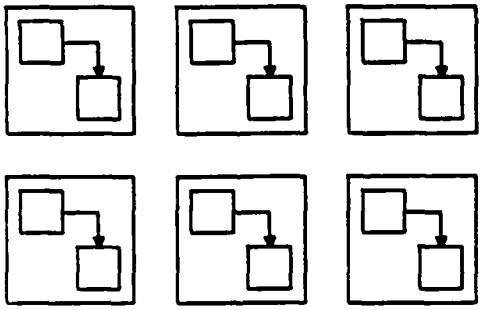


Figure 3-2b PISO Cell--Latch

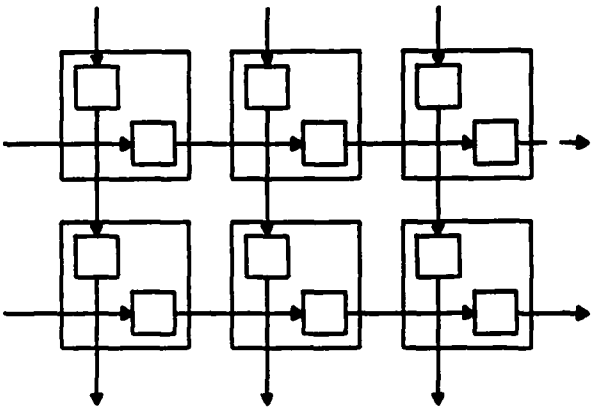


Figure 3-2c PISO Cell--Parallel Shift In, Serial Shift Out

The address buffer holds four 12-bit addresses; the addresses are stored sequentially in the buffer (i.e., first, second, third, and fourth). Since the longest blocklength is 4080, twelve bits suffice for the addressing ($2 sup 12~ =~ 4096$). The data address is loaded onto the address bus during one clock cycle; the data words from the memory are loaded onto the data bus on the next clock cycle (reference paragraph 3.2.3 for memory requirements). A more complete description of the address generation process is given in paragraph 3.2.2.

3.2.2. Control Circuitry. There are two type of control circuitry: one is for off-chip communication, while the other is for on-chip communication. Off-chip communication consists of three handshake signals and a three-bit scaling factor. The three handshake signals are OPERATE, DONECOMP, and DONEIN. The OPERATE signal is originated by the interface chip and is sent to the Winograd processor; it starts the processor computing DFTs. The DONEIN signal is originated by a Winograd processor and is sent to the interface processor; it indicates the Winograd processor has finished its pass

through the input data. The DONECOMP signal is originated by the Winograd processor and is sent to the interface processor; it indicates the Winograd processor has finished its DFT computations. Each Winograd processor computes an output scaling factor which indicates the largest magnitude of any data word in a particular DFT computation. Refer to paragraph 3.3. for more information on the handshake signals.

The scaling factor is a three-bit number passed from one processor to the interface processor. The scaling factor is a direct indication of the smallest number of sign extensions for the given set of 4080 data words. The scaling factor is computed as the data words are being transmitted from the SIPO register to the data memory; the scaling factor is latched into a special register when a particular Winograd processor has finished its pass through the data. When the interface processor receives the DONECOMP signal from a Winograd processor, it reads the scale factor from the register and sends it to the next processor in the system (reference paragraph 3.3.). The host may or may not provide a scaling factor (depending on system configuration); if the host does not provide the scaling factor, the first Winograd processor assumes a scale factor of zero.

There are two types of on-chip control circuitry; one is used to implement the processor timing diagram, while the other is used for address generation. The processor timing diagram reflects the temporal relationship of internal control signals necessary for the arithmetic circuitry to correctly perform a 16-point DFT calculation (e.g.,shifting, latching, etc.) and to enable special circuits (parity, rounding, etc.).

The parity check cell between the PISO register and the arithmetic circuitry is enabled for twenty-three clock cycles. This allows the parity check cell to compute the parity on the input data word and compare its result to the input parity bit. There is an arithmetic round signal which enables the multipliers to round the 60-bit results ($32 + 28 = 60$) to thirty-two bits [6]. The round circuitry in the P R cell between the arithmetic circuitry and the SIPO register is active for twenty-four clock cycles. This allows the round circuitry to operate on the most significant twenty-four bits of data.

rounding the result to twenty-three data bits for the SIPO register. The parity circuitry in the P/R cell monitors the output of the round circuitry in the P/R cell; after the twenty-three data bits have been sent to the SIPO register from the round circuitry, the parity circuit appends a parity bit (computed from the 23 data bits).

The shifting of output results from the SIPO register is straightforward and occurs at the same relative time each 32-cycle period; however, the shifting of input data words depends on the input scaling factor (from the previous processor). If scaling was not implemented, five zeroes would be inserted at the least significant end of the incoming data word and four sign extensions would be appended at the most significant end. The four sign extensions allow for arithmetic growth in the DFT computation and for the extra sign extension required by the multipliers [6]. Only three sign extensions are required for arithmetic growth since the two results from the pre-addition matrix which have more than eight terms are multiplied by one (i.e., trivial multiplications); thus, only three, rather than four, sign extensions are required to allow for arithmetic growth. The reasons for using sign extensions are explained in more detail in paragraph 4.2.2.

The zeroes change the arithmetic inputs to thirty-two bits (to balance the 32-cycle period for reading in new data). If the input scaling factor is greater than or equal to four, no additional sign extensions are required (since the incoming data words have at least four sign extensions); thus, nine zeroes may be inserted at the least significant end of each data word ($9 + 23 = 32$). If the input scaling factor is less than four, additional sign extensions are needed; the number of sign extensions required is: 4 - *inscale* and the number of zeroes inserted is: 5 − *inscale* (where inscale is the three-bit input scaling factor). The effect of inserting the zeroes at the least significant end is to delay shifting the 23-bit incoming data word out of the PISO register into the arithmetic circuitry.

The address generation circuitry computes the input and output addresses. The addresses are stored in a read-only memory (ROM). The order of the addresses is governed by Chinese Remainder Theorem (reference paragraph 2.3.2.1.). The address

buffer holds four addresses since the access time of the ROM is too long to retrieve a single address in two clock cycles. A pointer to the first of a group of four addresses is loaded into the ROM address bus; the four addresses are loaded from the ROM into the address buffer. The pointer is incremented to the next set of four addresses and the address generation process is repeated. Since different configurations may be used for the overall DFT processor (i.e., compute 240-, 255-, or 272-point DFTs rather than 4080-point DFTs), any of the Winograd processors may be required to implement the PFA input mapping; thus, each Winograd processor uses this method of address generation. The output addresses are identical to the input addresses (i.e., the result is stored at the same relative location in the output memory as the corresponding input was taken from the input memory). Thus, the output addresses are merely delayed versions of the input addresses. Rossbach [22] has more details on the control circuitry.

3.2.3. Arithmetic Circuitry. The adder/subtractors in the arithmetic circuitry emulate the pre-addition and post-addition matrices of the Winograd modules. Since the addition matrices contain both positive and negative entries, the elements which reflect the operation of the addition matrices must support both addition and subtraction. Multiplication by imaginary coefficients is reflected in the post-addition matrix (reference paragraph 2.3.1.). So, the results from the real and imaginary multiplication circuits must be combined. This is indicated in Figure 3-1 by the dotted line separating the real and imaginary post-addition circuits. The multipliers are bit-serial and employ a modified Booth's algorithm [6]. Each multiplier cell represents two bits of the coefficient. Thus, fourteen multiplier cells must be used for 28-bit coefficients. Since the coefficients are known ahead of time, they may be hardwired into the multiplier, rather than being stored in a coefficient memory. This reduces the time and area required to perform a multiplication. The multipliers round the results from sixty bits ($28 + 32 = 60$) to thirty-two (32) bits. The use of rounding rather than truncation provides a better

signal-to-noise margin (reference paragraph 4.2.2.). Coutee [6] has more details on the operation of the arithmetic circuitry.

3.2.4. Physical Characteristics of 16-Point Processor. There will be three phases of fabrication and packaging for the Winograd processors. In the first phase, test macrocells for parts of the arithmetic and control circuitry were designed using scalable 3-μm design rules for complementary metal-oxide semiconductor (CMOS) technology. These test cells were packaged in 32-pin dual in-line packages (DIPs). Test cells fabricated and tested to date have worked at a clock rate of 50 MHz [6, 22].

After the design of the VLSI circuits is verified, the Winograd processor will be fabricated using a 1.25-μm CMOS process. These single-chip processor should be packaged using pin-grid arrays or chip carriers [8] to support the 144-pin requirements (96 data, 24 address, 12 control, and 12 power). Each processor should occupy one square inch of surface area on a circuit board (assuming a 144-pin package).

The final phase of fabrication will bring the processors and required peripheral devices (interface processor and memories; reference paragraph 3.3.) into a hybrid circuit package. The reasons for using the hybrid circuit are reduced surface area, increased reliability (elimination of error-prone wire bonds), and increased I/O bandwidth (shorter paths between processors and memories means faster transitions are possible). The target clock rate for the 1.25-μm chips is 70 MHz.

Table 3-1 contains a summary of Winograd processor characteristics.

3.3. 4080-Point PFA Processor. The Winograd processors described in the previous section provide a means of computing DFTs. But, there must be other devices which enable the data to be sent to the Winograd processors and which store the results. Also, the individual Winograd processors compute short blocklength DFTs; useful applications require blocklengths of

Winograd Processor Characteristics	
Characteristic	Value
Technology	CMOS
Packaging	Pin-Grid Array Chip Carrier
Size	1 sq. in. (a)
Pin Count	144
Clock Rate	70 MHz
Power	1 W (a) 100 mW (c)

(a) estimated
(b) active mode; estimated
(c) standby mode; estimated

Table 3-1

256 (image processing) and 4096 (synthetic aperture radar and matched filtering). Thus, a system which computes these longer blocklength DFTs needs to be addressed. The 4080-point PFA system was chosen as a representative system. Systems for computing DFTs with blocklengths of 240, 255, and 272 points will use one less Winograd processor, memory controller, and data memory. The layout of the 4080-point PFA processor is presented in Figure 3-3.

There are separate chips for each Winograd processor, dual-port memories, and the interface processor. The Winograd processors operate autonomously, computing DFTs using either the small Winograd algorithm (16- and 17-point) or the large Winograd algorithm (15-point). The dual-port memories allow one Winograd processor (or the host processor) to fill one half of the memory while its neighboring processor reads from the other half of the memory. The interface processor provides overall control of the pipeline: asynchronous control signals from the memory controllers, the Winograd processors, and the host flow through the interface processor.

The signal flow through the processor for one DFT is described below:

1) The host processor fills left half of input memory with data;

2) When the host finishes filling left half of input memory, it sends a handshake signal to the first memory controller, indicating the left half of the input memory is

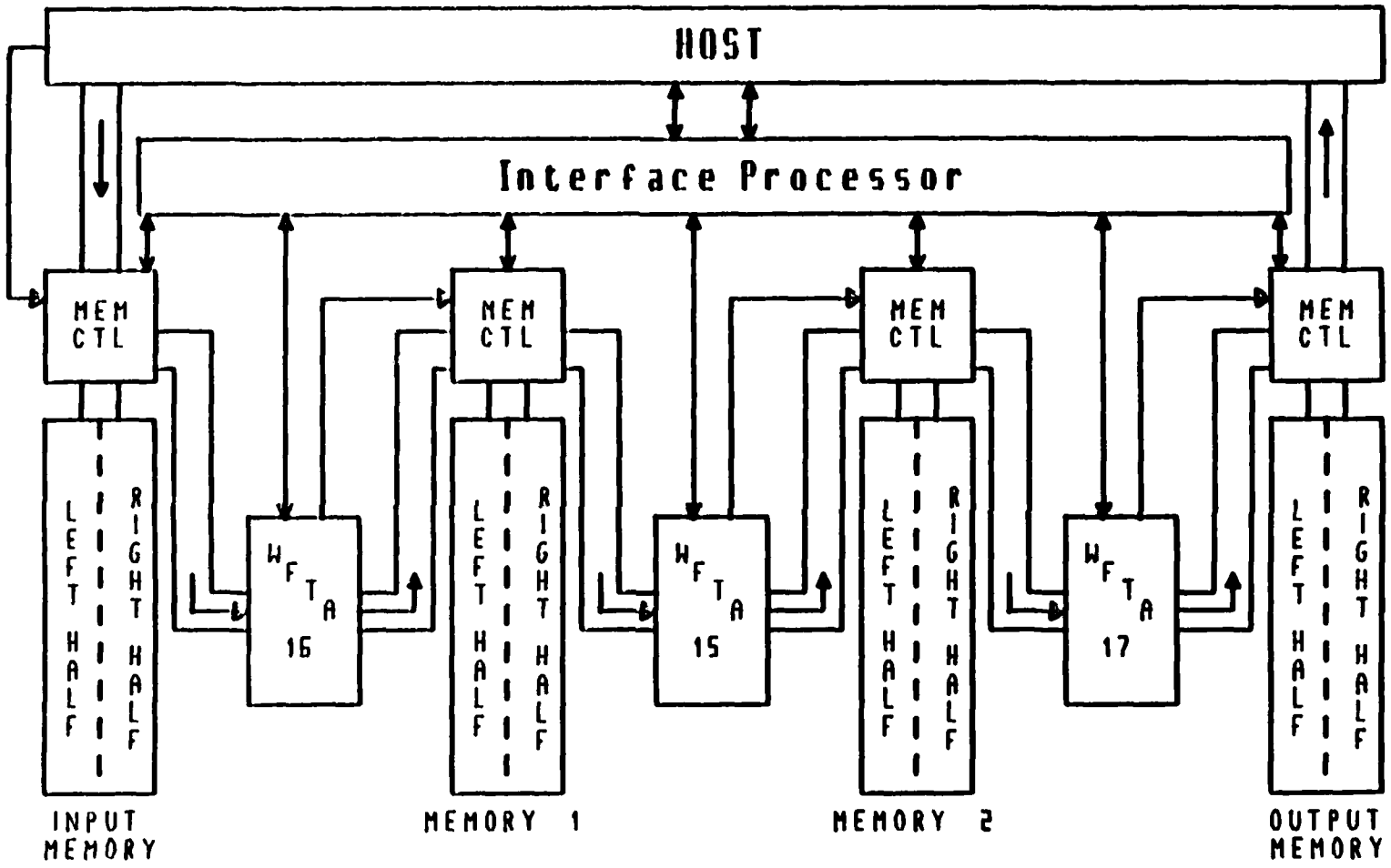


Figure 3-3 4000-Point PFA Processor

filled:

3a) The first memory controller switches the control logic for the input memory such that the host is writing to the right half of the input memory and the 16-point processor is reading from the left half of the input memory;

3b) After the memory controller has switched the logic for the input memory, the host sends an OPERATE signal to the interface processor, which relays the OPERATE signal to the 16-point processor (this initiates the 16-point pass through the data);

4) As the 16-point chip is computing 255 (15×17) DFTs, it sends the outputs to the left half of memory 1;

5) When the host has finished filling the right half of the input memory with new input data, it sends a handshake to the first memory controller, indicating the right half of the input memory is ready;

6) When the 16-point processor has finished reading in 4080 data words, it sends a DONEIN signal to the interface processor, which relays the DONEIN signal to the first memory controller (this allows the first memory controller to switch the control logic of the input memory such that the host is writing to the left half of the input memory and the 16-point processor is reading from the right half of the input memory);

7) When the 16-point processor has finished computing 255 16-point DFTs, it sends a DONECOMP signal to the interface processor, which relays the DONECOMP signal to the second memory controller (this allows the second memory controller to switch the control logic of memory 1 such that the 16-point processor is writing to the right half of memory 1 and the 15-point processor is reading from the left half of memory 1) and to the host (indicating the 16-point processor has finished its pass through the data);

8a) When the host has received the DONECOMP signal from the 16-point processor (via the interface processor) and the host has finished filling the right half of the input memory, the host sends an OPERATE signal to the interface processor, which

relays the OPERATE signal to the 16-point processor (this allows the 16-point processor to begin its pass through the next set of 4080 data words);

8b) When the host has received the DONECOMP signal from the 16-point processor (via the interface processor), the host sends an OPERATE signal to the interface processor, which relays the OPERATE signal to the 15-point processor (this allows the 15-point processor to begin its pass through the first set of 4080 data words);

9) As the 15-point processor is computing 272 (16×17) DFTs, it sends the outputs to the left half of memory 2;

10) When the 16-point processor has finished 255 16-point DFTs, it sends a DONECOMP signal to the interface processor (this allows the interface processor to send a handshake to the second memory controller, indicating the right half of memory 1 is ready);

11) When the 15-point processor has finished reading in 4080 data words, it sends a DONEIN signal to the interface processor, which relays the DONEIN signal to the second memory controller (this allows the second memory controller to switch the control logic of memory 1 such that the 16-point processor is writing to the left half of the input memory and the 15-point processor is reading from the right half of memory 1);

12) When the 15-point processor has finished computing 272 15-point DFTs, it sends a DONECOMP signal to the interface processor, which relays the DONECOMP signal to the third memory controller (this allows the third memory controller to switch the control logic of memory 2 such that the 15-point processor is writing to the right half of memory 2 and the 17-point processor is reading from the left half of memory 2) and to the host (indicating the 15-point processor has finished its pass through the data);

13a) When the host has received the DONECOMP signal from the 15-point processor (via the interface processor) and the 16-point processor has finished filling the right half of memory 1, the host sends an OPERATE signal to the interface processor, which relays the OPERATE signal to the 15-point processor (this allows the 15-point processor

to begin its pass through the next set of 4080 data words);

13b) When the host has received the DONECOMP signal from the 15-point processor (via the interface processor), the host sends an OPERATE signal to the interface processor, which relays the OPERATE signal to the 17-point processor (this allows the 17-point processor to begin its pass through the first set of 4080 data words);

14) As the 17-point processor is computing 240 (15×16) DFTs, it sends the outputs to the left half of the output memory;

15) When the 15-point processor has finished 272 15-point DFTs, it sends a DONECOMP signal to the interface processor (this allows the interface processor to send a handshake to the second memory controller, indicating the right half of memory 2 is ready);

16) When the 17-point processor has finished reading in 4080 data words, it sends a DONEIN signal to the interface processor, which relays the DONEIN signal to the third memory controller (this allows the third memory controller to switch the control logic of memory 2 such that the 15-point processor is writing to the left half of memory 2 and the 17-point processor is reading from the right half of memory 2);

17) When the 17-point processor has finished computing 240 17-point DFTs, it sends a DONECOMP signal to the interface processor, which relays the DONECOMP signal to the fourth memory controller (this allows the fourth memory controller to switch the control logic of the output memory such that the 17-point processor is writing to the right half of the output memory and the host is reading from the left half of the output memory) and to the host (indicating the 17-point processor has finished its pass through the data);

18a) When the host has received the DONECOMP signal from the 17-point processor (via the interface processor) and the 15-point processor has finished filling the right half of memory 2, the host sends an OPERATE signal to the interface processor, which relays the OPERATE signal to the 17-point processor (this allows the 17-point processor

to begin its pass through the next set of 4080 data words);

18b) When the host has received the DONECOMP signal from the 17-point processor (via the interface processor), it begins to read the results from the output memory.

The salient features of the PFA system are the asynchronous control signals for host-processor and processor-processor communications and the data flow between the processors. The interface between the host and the Winograd processors is simple, requiring only three handshake signals. The Winograd processor operate autonomously, communicating with the host (via the interface processor) only at the beginning and end of each DFT. The DFT problems are pipelined both within the Winograd processors and in the PFA system, providing high throughput (reference paragraph 3.5.). Another virtue of the system as shown in Figure 3-3 is it may be reconfigured easily if necessary (reference paragraph 3.5.).

3.4. Data Memory Requirements.

The data memory is implemented off-chip from the Winograd processors; there was insufficient area on-chip for the WFTA circuitry and memory circuitry [6, 22]. The memory must be organized into banks of 4080 24-bit words. The most significant bit will be a parity bit (odd parity), while the remaining twenty-three (23) bits comprise a two's-complement representation of the data. The parity bit is used to provide an independent check on the memory circuits; odd parity is used so the "stuck at zero" or "stuck at one" states (caused by memory power failure) may be detected. The memory controllers monitor the DONE signals from the processors to the host. The controller switches the status of the left and right halves of the memory (reference paragraph 3.2.2.). Read Write signals are generated by the on-chip Winograd processor control circuitry. The access time for the data memories must be less than 28 ns (two clock cycles; reference paragraph 3.5.). Currently, off-the-shelf 256K-bit memories have access times

of approximately 40 ns [30]. If the trends in memory design hold to the same pattern as the past ten years, memories with the required capacity and access times should be available in 1987 [11, 19, 26]. Current 256K-bit memories use on-chip error-correcting codes (ECC) to provide additional fault detection capability [23, 30]. Also, spare rows or columns are provided for internal reconfiguration if necessary [23].

Data memory characteristics are summarized in Table 3-2.

3.5. Fault Tolerance.

Fault tolerance is the ability to operate in the presence of faults and provide useful results to users [24]. Many ideas have been expressed on the issue of fault tolerance; however, most deal with the concepts of fault avoidance, fault detection, and system recovery [24, 27]. Before discussing these three concepts and their application to our system design in the following paragraphs, some definition of terms are presented.

Error: the resource (system) assumes an undesirable state

2.

Failure: the user (host) perceives the resource ceases to deliver an expected service

2.

Data Memory Characteristics	
Characteristic	Value
Technology	CMOS
Packaging	DIP Pin-Grid Array
Size	4.5 sq. in. (a)
Pin Count	48
Access Time	25 ns
Power	100 mW (b)

(a) area for 48-pin DIP
(b) average for 256 Kbit chips

Table 3-2

Fault: the hypothesized cause of an error or failure [2].

Reliability: how often a component fails to perform its function [28].

Availability: the probability a system is operational at a given time [28].

3.5.1. Fault Avoidance. Fault avoidance increases reliability by lessening the probability of failures and errors. Component design and environmental hardening are the two most popular methods used for fault avoidance [24]. Component design reduces errors by careful signal routing and increased circuit integration, while environmental hardening seeks to protect the system from outside interference. Some examples of fault avoidance are single-chip microprocessors (component design) and crystal ovens (environmental hardening). In both cases, emphasis is placed on removing the causes of faults, rather than the effects of the faults.

The Winograd processors employ component design to achieve fault avoidance; however, the processors must be fully operational before fault avoidance is realized. To this end, chip testability is of prime importance [6, 22]. The individual chips must be certified as fully operational with no faults before they can be used in a system. The testability of the Winograd processors is enhanced with multiplexers which allow the pins to operate in two modes (Test and Operate). The Test mode allows test vectors to be written to and outputs to be read from most internal circuits. Individual memory chips also must be capable of being tested. Since commercially available memory chips will be used, this feature is assumed to exist.

3.5.2. Fault Detection. Many techniques are available for fault detection; two of the most popular are information coding (also called error-correction coding or ECC) and consistency checking [12, 23]. Information coding may be used to detect and correct bit errors. Parity checking is a form of information coding which provides bit error detection. Although simple and easy to implement, parity checking has a limitation in that it may not detect multiple-bit errors. A more powerful ECC is the single-error correction, double-error detection (SEC-DED) Hamming code. This code may be used with parity checking to further reduce bit error probabilities. In the PFA system, the parity checking of the Winograd processors provides an independent check on the data memories. If one memory chip should suffer catastrophic failure, the on-chip ECC would be useless. The Winograd parity checking allows detection of this case.

Consistency checking usually involves extra processors which operate in parallel with active processors. These extra processors (sometimes referred to as watchdog processors) compare the results from their computations and the results from the active processor to determine if an error has occurred [12]. The Winograd processors have the capability of operating actively or in the watchdog mode. The interface processor sends control signals to the Winograd processors which configure the processors as either active or watchdog. There is one active Winograd processor and two watchdog processors. When the interface processor sends the OPERATE signal to the active processor, it also sends OPERATE to the two watchdog processors. The watchdog processors receive the same data as the active processor and compute the same DFT; however, the watchdogs only monitor the output data lines of the active processor. An error signal is sent to the interface processor if either one or both watchdog processors detect an error on the data lines. Also, the watchdog processors monitor the input and output address lines of the active processor and flag any addressing errors. Thus, the watchdog processors detect both data and address errors of the active processor. Figure 3-4 shows the configuration of the active and watchdog processors.

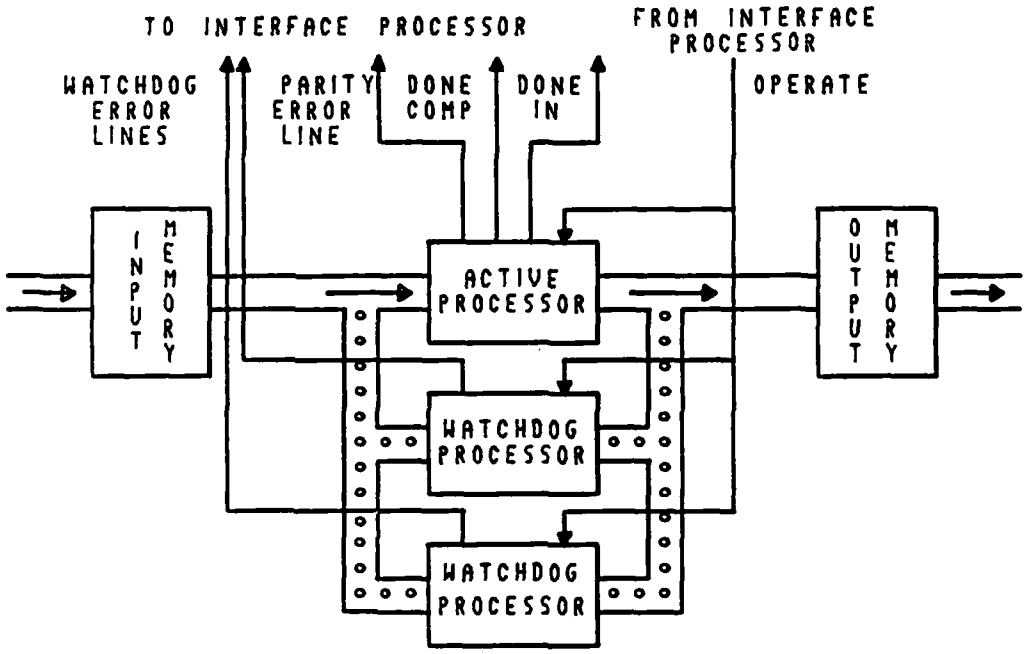


Figure 3-4 Active/Watchdog Processors

3.5.3. System Recovery. System recovery is the process of removing the effects of detected faults from the system. This may accomplished through space redundancy (extra devices) or time redundancy (computation retry) [24]. Space redundancy may be realized using multi-processor voting (using the majority "vote" of an odd number of processors) or through systolic arrays of autonomous processors [10]. Time redundancy requires enough time be scheduled to recompute the problem, lowering system throughput. The problem may be recomputed using different components, a different algorithm, or both [24]. One of the risks of computational retry is entering an infinite loop when a hard fault exists (a hard fault requires hardware replacement to resume

proper operation). Space and time redundancy may be combined into a hybrid realization of system recovery. Spare resources are switched from standby to active mode to replace devices which have exhibited a given number of faults.

System recovery is implemented by the interface processor. The interface processor monitors the watchdog error lines and the parity error lines from the active Winograd processor. Parity errors are recorded and may be used for memory reconfiguration (see below). If both watchdog processors detect the same error condition, the interface processor may remove the active processor from the system, designate one of the watchdog processors as the active processor, and continue operation with one active processor and one watchdog processor (rather than two watchdog processors and one active processor). If only one watchdog processor detects an error, the interface processor may restart the computation (issue another OPERATE signal) or it may remove the watchdog processor and continue operation with only one active processor and one watchdog processor. Data memories may employ space redundancy, as well as parity and error-correction coding. Extra memory locations may be used when on-chip ECC indicates several successive errors [23] or when the interface processor has recorded several parity errors from the same memory chip. A summary of the fault tolerant features incorporated into the design of our VLSI circuit is given is Table 3-3.

3.6. Computational Throughput.

The Winograd processors should employ a clock rate of 70 MHz. A 70 MHz clock rate means a clock cycle time of 14.3 ns. Thus, each 32-cycle period has 460 ns (457.6 ns). The latency through the 16-point Winograd processor is 117 clock cycles; thus, the 16-point processor takes:

$2 \times 117 = 234$ (first and last sets of 16 data words)

$253 \times 32 = 8096$ (255 - 2 = 253)

Fault Tolerant Features	
Characteristic	Type or Value
Fault Avoidance	Component Design (a) (b)
Fault Detection	ECC (b) Consistency Checking (a)
System Recovery	Space Redundancy (a) (b) Time Redundancy (a)
(a) Winograd processors (b) data memories	

Table 3-3

$8096 + 234 = 8330$ clock cycles

8330×14.3 ns $= 119.12$ μs

Similarly, the 15-point processor takes 119.06 μs and the 17-point processor takes 119.18 μs to complete their passes through the data. Thus, the latency through the 4080-point PFA pipeline is:

$119.12 + 119.06 + 119.18 = 357.36$ μs

Adding 2.64 μs for overhead (handshaking, etc.) brings the latency through the 4080-point PFA pipeline to 360 μs. However, once the pipeline is filled, DFT results will appear at the output memory every 120 μs (8333 results per second). This throughput will allow a 4080-point PFA array element (consisting of Winograd processors, dual-port memories, and and interface processor) to meet many of the high data rate applications, such as synthetic aperture radar (reference Chapter 4). The number of arithmetic operations computed for each Winograd processor and the 4080-point system are given in Table 3-4.

3.7. Summary.

The material in this section has shown how the algorithms presented in Chapter 2 mapped into a VLSI architecture. The Winograd processors have three types of circuitry: control, input output, and arithmetic. There will be three phases of fabrication for

Rate of Arithmetic Operations		
Processor	MMPS(a)	MAPS(b)
15	79	378
16	43	323
17	144	646
4080	266	1347

(a) Millions of Multiplications per Second
(b) Millions of Additions per Second

Table 3-4

the Winograd processors; the ultimate design will use 1.25 μm CMOS, with the processor and all peripheral devices mounted on a hybrid circuit package. The 4080-point PFA system demonstrated how longer blocklength DFTs could be computed using the Winograd processors, data memories, and an interface processor. The computational throughput of the 4080-point PFA system, assuming the Winograd processors use an internal clock rate of 70 MHz, will be 8300 DFTs per second (i.e., a new DFT result every 120 μs). Thus, the ability to compute DFTs using VLSI hardware has been shown to be feasible. The next chapter will show the circuits also provide the required numerical accuracy for many applications.

Chapter 4

Numerical Performance of Winograd Processors

4.1. Overview.

The previous chapter discussed the transition from the mathematical algorithms presented in Chapter 2 to VLSI architecture which implemented the algorithms. The material in this chapter will describe how well the VLSI circuits perform the DFT computation, using numerical accuracy as the metric. A comparison will be made using software programs to simulate the operation of the VLSI circuits and to compute a standard result. The simulation programs use integer arithmetic to achieve the same results as the VLSI Winograd processors, while the standard programs use double-precision real arithmetic to provide "correct" results (within the limits of the computer system). The flow of information in this chapter will be to first present material on the metric used for *numerical accuracy*. Next, *the programs used for the comparison are discussed*. Finally, the results of the comparison are given.

4.2. Signal-to-Noise Ratio and Noise Sources.

The metric used to measure the numerical accuracy between results from the simulation and standard programs is the signal-to-noise ratio (SNR). The SNR has been used for many years to measure the quality of communication systems The SNR is the ratio of the power in the signal to the power in the noise (interfering signal). Since the power in the signal may be many orders of magnitude greater than the power in the noise, a logarithmic form of the SNR is often used; this form of the SNR converts the dimensionless ratio into a number with the units of decibels (dB). The formula for converting the ratio (signal power to noise power) to the logarithmic form is shown below:

$$SNR_{dB} = 10 \log_{10} (ratio) \qquad (4-1$$

The power in the signal is the the sum of the power at each frequency component; the power at each component is found by squaring the magnitude of the voltage signal at that particular component. A similar computation is used to compute the power of the noise. In this case, there is only one signal source, the "correct" result computed by the standard portion of the program. However, there are several sources of noise, mostly caused by the scaling required to prevent arithmetic overflow and the finite-length coefficients used by the multipliers in the Winograd processors. These noise sources are now discussed in greater detail.

4.2.1. Scaling. The inputs to the arithmetic circuitry must be scaled down to avoid arithmetic overflow and because the multipliers expect each input to have two sign extensions [6]. Whenever a DFT is computed, the transform results experience arithmetic growth; this is most easily seen by observing Parseval's relationship:

$$\sum_{j=0}^{N-1} \left| v_j \right|^2 = \frac{1}{N} \sum_{k=0}^{N-1} \left| V_k \right|^2 \tag{4-2}$$

where

v_j represents the input sequence;

V_k represents the transformed sequence;

N is the DFT blocklength.

The VLSI circuit uses a blocked floating-point number representation. This is similar to an integer representation, but a number of bits are set aside for an exponent. Thus, a number is represented by a fixed length mantissa (23 bits) and an exponent (a positive 3-bit number). For a 15-point DFT, the transform results may grow by a factor of fifteen. This can be seen by looking at the pre-addition matrix of the 15-point DFT (reference Appendix A). Since fifteen can be represented by four bits ($2^4 = 16$), the 15-

point Winograd processor appends five sign extensions to the inputs to the arithmetic circuitry, four for arithmetic growth and one for the multipliers. The 16-point DFT requires four sign extensions and the 17-point DFT requires six sign extensions (reference paragraph 3.2.3.). The 16-point WFTA chip only needs three sign extensions to avoid arithmetic overflow since the two inputs to the multipliers which have more than eight terms are multiplied by trivial coefficients. Multiplication by trivial coefficients only requires a shift, rather than passing the inputs through the multiplier circuitry. The extra sign extensions are not required for all inputs, they are insurance against arithmetic overflow. Thus, the numerical accuracy of the results suffer since all the sign extensions are not used. On the average, two of the sign extensions will not be used. Since random numbers are used, the output spectrum will be fairly flat, providing the worst case for scaling. If the variance of the inputs (σ_x^2) is assumed to be $\frac{1}{2}$, then the variance of the outputs (σ_Y^2) can be found using (4-2):

$$\sum_{0}^{15} \left[\frac{1}{2}\right]^2 = \frac{1}{16} \sum_{0}^{15} \sigma_Y^2 \tag{4-3}$$

or, collecting terms and noting σ_Y^2 is the same for each term on the right-hand side of (4-3):

$$\frac{16}{4} = \sigma_Y^2 = 4 \tag{4-4}$$

Thus, the outputs are four times as large as the inputs, on the average. This uses only two of the sign extensions provided to prevent arithmetic overflow; the other sign extensions are basically wasted, as far as numerical accuracy is concerned. Therefore, there will be two wasted sign extensions for the 15-point and 16-point simulations and three wasted sign extensions for the 17-point simulation. This translates into a 12 dB loss in SNR for the 15-point and 16-point simulations (1 bit — 6 dB [7]). Scaling provides the

largest source of noise in the Winograd processors.

4.2.2. Arithmetic Roundoff. Another source of noise is created when the 32-bit results from the arithmetic circuitry must be shortened to twenty-three bits. There are two choices; the results may either be truncated or rounded. Truncation, where the least significant nine bits are simply ignored, gives an error of ½ bit (on the average). Rounding, where the twenty-third bit is rounded up if the twenty-fourth bit is a one, gives an average error of ¼ bit. The noise power provided by this source is:

$$\sum_{0}^{15} \left[\frac{1}{4}\right]^2 = 16 \times \left[\frac{1}{16}\right] = 1 \qquad (4\text{-}5)$$

$$10 \times \log_{10} 1 = 0\ dB$$

Thus, rounding provides better arithmetic performance than truncation. However, rounding is more difficult to implement, requiring a latch and an adder, while truncation requires no special circuitry. In the VLSI implementation, the rounding circuitry added no significant delay to the propagation of the arithmetic results to the SIPO register; thus, rounding was chosen instead of truncation. Rounding is also done when the 60-bit products are changed to thirty-two bits in the multipliers; however, this source of noise is almost insignificant since the results are rounded to twenty-three bits at the SIPO register. Figure 4-1 depicts rounding at the SIPO.

4.2.3. Finite-Length Coefficients. The Winograd processors use fixed coefficients, which are hardwired into the multipliers. Coefficients which are not a power of two or combinations of powers of two are not exactly represented. This represents a noise source which could be very significant, since many multiplications are performed in the DFT computation. To reduce the effects of using finite-length coefficients, the coefficient

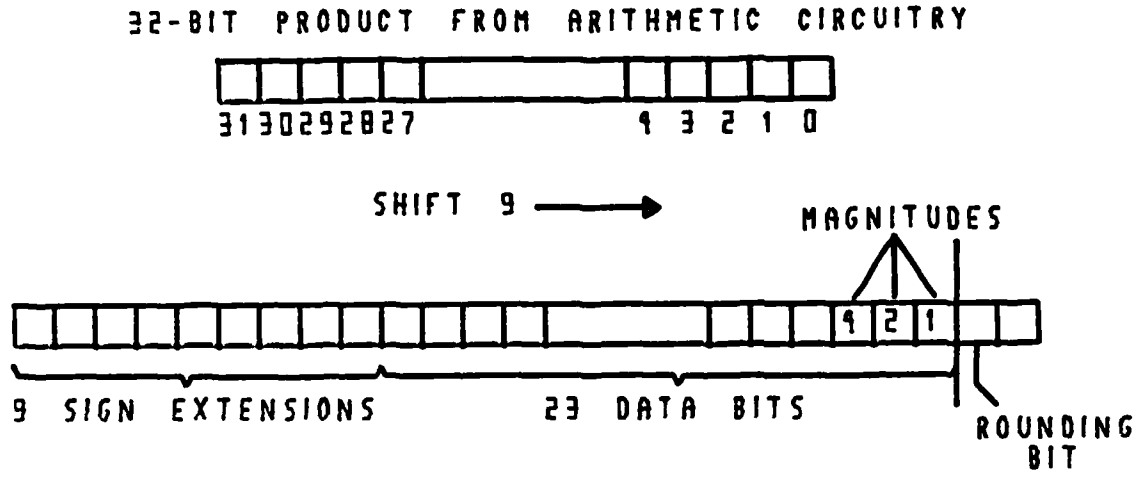


Figure 4-1 SIPO Rounding

wordlength is increased; however, this also increases the latency through the arithmetic circuitry. Thus, there is a tradeoff between the coefficient wordlength and the latency through the arithmetic circuitry. The latency through the arithmetic circuitry does not significantly affect the overall throughput of the Winograd processors. Thus, the coefficient wordlength could be increased arbitrarily to give better numerical performance; but, the area required by the multipliers would grow beyond the size of the VLSI chip (multipliers are the most costly in terms of area [6]). Therefore, to reduce the effects of the finite-length coefficients, yet maintain reasonable chip size, the wordlength was chosen to be twenty-eight bits.

4.3. Simulation Programs.

The goal of the simulation programs is twofold; one goal is to exactly simulate the operation of the VLSI implementation, while the other goal is to provide a precise result to compare with the simulation result. To simulate the VLSI circuit operation, bit-level manipulations (shifting and masking) are required, as well as 32-bit integer and 64-bit

real arithmetic. The standard module was compared with a direct implementation of the DFT, using (2-1). The computer language chosen for the simulation programs was 'C'; 'C' possessed the numerical requirements discussed above and it was available on the VAX 11-780 Scientific Support Computer (SSC) at the Air Force Institute of Technology. The information on the simulation programs will be presented in the following order:

1) Number representation;

2) Differences between standard and simulation modules;

Listings of the simulation programs are given in Appendix B. The flowgraph of the 4080-point simulation program is shown in Figure 4-2.

4.3.1. Number Representation. The VLSI circuits use a two's-complement notation for all numbers; the most significant bit indicates the sign of the number, while the other bits indicate the magnitude. *Positive numbers have zero as the most significant bit;* the remaining bits represent the magnitude in the obvious manner. Negative numbers are represented by complementing a positive number with the same magnitude, then adding one. Internal results are thirty-two bits for the 16-point algorithm (30 bits for the 15-point and 34 bits for the 17-point), coefficients are twenty-eight bits, and input data are twenty-three bits. The simulation modules represent these different wordlengths by declaring all variables to be long integers (32 bits on the VAX SSC), then masking the most significant bits to obtain the desired wordlength. The standard modules represent all variables as double-precision real numbers.

4.3.2. Differences Between Standard and Simulation Modules. There are four major differences between the standard and simulation modules of the simulation programs. The simulation module requires a special multiply function and bit-level manipulations.

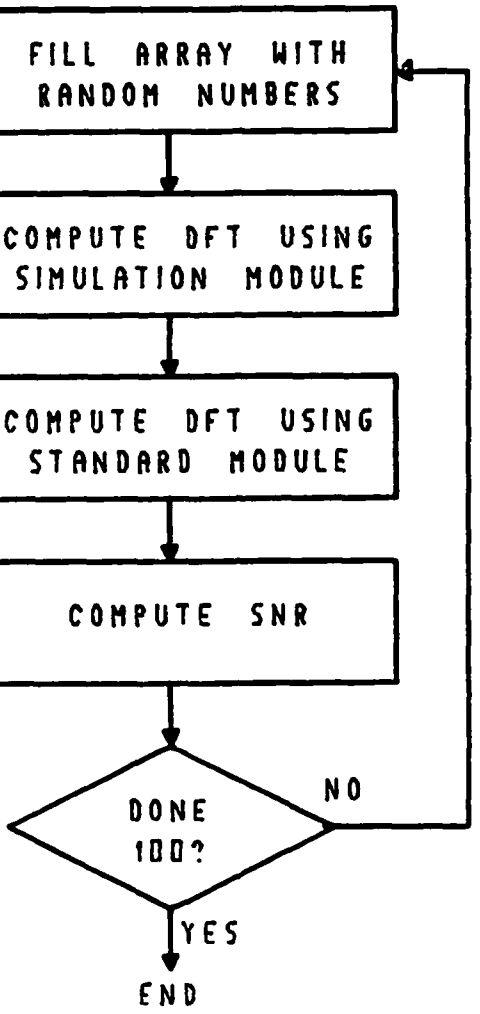


Figure 4-2 Simulation Program Flowchart

The simulation coefficients should reflect the choice of 28-bit integers for the VLSI circuit. The effects of the scaling and the multiplication require a scaling of the standard outputs for SNR computations.

4.3.2.1. Simulation Multiply Routine. The simulation module cannot multiply two long integers using the intrinsic multiply function of 'C'; the multipliers in the VLSI circuit round the results to thirty-two bits, rather than truncating them as the intrinsic multiply function does. However, more importantly, the products from the VLSI multipliers are sixty bits long; using the intrinsic multiply function in 'C' gives 64-bit products since 'C' expects long integers to be thirty-two bits. Thus, a special multiply function, which gives 32-bit results rounded down from 60-bit products is used. The algorithm used in the function is presented in Aho, et. al [1]. The simulation multiplication routine is shown in Figure 4-3.

4.3.2.2. Simulation Scaling and Rounding. The bit-level manipulations required by the simulation modules are manifested in the scaling and rounding routines (reference paragraphs 4.2.1. and 3.2.1.). Scaling is implemented as a logical shifting operation; only the zero filling is done, since the 'C' language on the VAX SSC (also using two's-complement notation) automatically supplies the sign extensions. Rounding is done by masking the bit used for rounding decisions (using the & operator in 'C'), then incrementing the result if the output of the masking operation is a logical one. The inputs to the standard and simulation modules were outputs of the random number generator intrinsic to 'C', masked to twenty-three bits. The period of the generator (2^{32}) was sufficient to ensure random inputs. The range of the inputs was $\pm$ 1 to remove the DC bias.

4.3.2.3. Simulation Coefficients. The decimal coefficients of the standard Winograd modules had to be changed to integer representation for the simulation module. The VLSI circuit used 28-bit coefficients; thus, the simulation had to translate the decimal coefficients into a usable form. First, the decimal coefficients were scaled down so all the coefficients had a magnitude less than one. For the 15-point coefficients, the coefficients

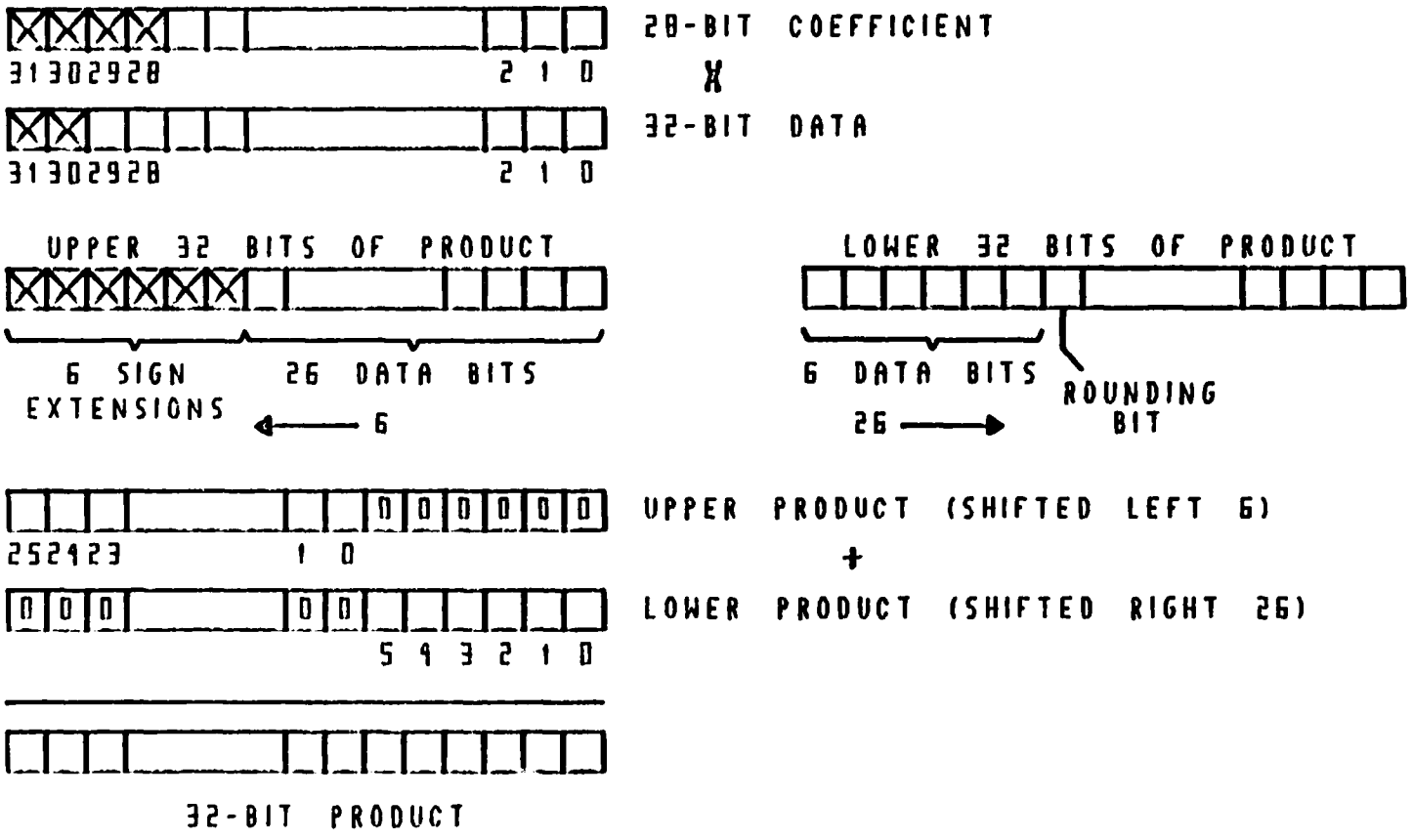


Figure 4-3 Simulation Multiplying

were divided by four since the largest coefficient was 2.308. Similarly, the 16-point coefficients were divided by two (largest coefficient was 1.383) and the 17-point coefficients were divided by eight (largest was 4.081). After the coefficients were scaled such that they were all between $\pm$ 1, were multiplied by 2^{27}, since 28-bit, two's complement numbers are bounded by $2^{27} - 1$ and $-(2^{27})$. The results from the 15-point and the 17-point modules had to be multiplied by factors of two and four, respectively, to provide a common reference with the 16-point module.

4.3.2.4. Simulation Versus Standard Outputs. Since the simulation modules incorporated scaling and integer multiplication, the outputs from the simulation module were of a different order of magnitude than the results from the standard module. The inputs to the simulation module were shifted left four places (scaled up by $2^4 = 16$). The simulation multiplier results were twice as large as they should have been since the simulation module shifted the results to the right twenty-six places, while the decimal coefficients of the standard module were multiplied by 2^{27} to represent the 28-bit coefficients used in the VLSI circuits. The results of the arithmetic circuitry were shifted right by 9 places (scaled down by $2^9 = 512$). Thus, the net effect of the input scaling, integer multiplication, and SIPO rounding was:

$$\frac{input \times 16 \times 2}{512} = \frac{input}{16} \tag{4-6}$$

Thus, the standard results were sixteen times as large as the outputs from the simulation module. The standard results were divided by sixteen, rather than scaling the simulation inputs up by sixteen. This was done since the VLSI circuit will compute the DFT according to the algorithm used by the simulation module; thus, the numerical accuracy should be measured using those results, rather than a shifted version of the results.

4.4. Simulation Results.

There were two types of simulation results, one when the results from the standard modules were compared with results from the direct DFT and the other when the standard results were compared with the simulation results. In both cases, the signal-to-noise ratio in dB was used.

4.4.1. Standard Versus Direct DFT. The results from the standard Winograd modules were compared with results from the direct DFT for six different blocklengths. For each blocklength, 100 different set of random-number inputs were used and the average SNR computed for both the real and imaginary outputs. Table 4-1 shows the results for this portion of the testing.

The decrease from approximately 275 dB for the single-factor DFTs to 175 dB for the double-factor DFTs was probably due to the arithmetic roundoff in the direct DFT. Since the direct DFT performed many more operations than the Winograd modules, the cumulative roundoff from the computer increased, possibly overwhelming the noise due to the difference between the results from the two routines. Still, the results show the standard modules were computing the DFT, rather than some other mathematical function.

Standard Versus Direct DFT Results (dB)		
DFT Size	Real SNR	Imag SNR
15	269.7	270.0
16	270.0	269.9
17	269.8	269.7
240	175.9	176.0
255	175.5	175.6
272	175.1	175.0

Table 4-1

4.4.2. Standard Versus Simulation Results. For this area of testing, the DFT blocklengths were 15, 16, and 17. Again, 100 cases of random-number inputs were used to compute an average SNR for each blocklength. Results are shown in Table 4-2.

The results from this comparison agree well with the theoretical expectations given in paragraph 4.2. Recall from paragraph 4.2., the loss due to scaling was 12 dB for the 15-point and 16-point simulations and 18 dB for the 17-point simulation. For all three simulations, the loss due to SIPO rounding was approximately zero dB. Since the outputs were twenty-three bits, the output signal power should have been:

$$23\ bits \times 6\ dB/bit\ =\ 138\ dB$$

Thus, the theoretical output SNR for the 15-point and 16-point simulation should have been 126 dB and the output SNR for the 17-point simulation should have been 120 dB. These results compare well with results from other VLSI circuits, notably the CUSP chip (94 db SNR for a 16-point DFT using a 20-bit number representation [21]).

4.4.3. Application Accuracy Requirements. The preceding paragraphs showed how well the Winograd processors performed the DFT computation. But, do these results indicate our VLSI circuit can be used for the applications which need high accuracy? A typical synthetic aperture radar (SAR) example may provide an answer. (reference Table 4-3 for SAR parameters taken from Hovanessian [9]).

Standard Versus Simulation Results (dB)		
DFT Size	Real SNR	Imag SNR
15	125.9	126.4
16	127.6	127.6
17	120.8	120.7

Table 4-2

SAR Parameters	
Parameter	Value
PRF	1225 pps
Pulse Width	33.8 μs
Transmit Frequency	1275 MHz
Vehicle Velocity	7.14 km/s
Vehicle Altitude	375 km
Antenna Width	2 m
Range Beamwidth	6.73 °
Range Width	100 km
Azimuth Length	10.9 km

Table 4-3

The time to fly one range beamwidth is 1.53 seconds. The sampling rate is 12 megawords per second (i.e., 12 million 23-bit complex words are collected from the sensor each second). Since a 4080 $\times$ 4080 image is collected every 1.53 seconds, only one PFA array element (reference Chapter 3) is required to keep pace with the sensor data output (reference paragraph 3.6.). The 23-bit number presentation provides a potentially greater accuracy than current 16- or 20-bit systems. For a 16-bit system, the best SNR which can be expected (using the 6 dB/bit rule) is 96 dB. Thus, even without taking into account noise, the VLSI architecture given in Chapter 3 has a potentially better numerical accuracy than a 16-bit system. Although comparable, or even better, accuracy can be obtained using general-purpose computers, the PFA array element can be placed on the vehicle for real-time signal processing. Also, the PFA array element can be used to code the image data sent back to the user (using transform coding techniques [8]).

4.5. Summary.

In this section, the methods of finding the numerical accuracy of the VLSI circuits, using software simulations, have been described. First, the metric for determining the accuracy, the signal-to-noise ratio, was presented. Then, two of the major sources of

noise, scaling and SIPO rounding, were discussed. Next, the simulation programs were given, with special attention paid to the differences between the standard and simulation modules. The results of the standard module compared favorably with outputs from the direct computation of the DFT, showing the standard module did indeed compute the DFT. Finally, the standard and simulation results were compared; excellent agreement with theoretical expectations was noted. Also, the numerical accuracy of the simulation results compares favorably with other VLSI DFT processors and shows the VLSI chips can satisfy the accuracy requirements of SAR signal processing.

Chapter 5

Results, Conclusions, and Recommendations

5.1. Overview.

The previous chapters have shown the transition from mathematical algorithms to a VLSI implementation of those algorithms and how well the VLSI implementation performs, using numerical accuracy as the metric. The material in this chapter gives results of the research, presents conclusions of the research, and discusses recommendations for future research.

5.2. Results.

There were three results from this research: 1) determining the accuracy of the *simulation programs*; 2) *verifying the accuracy of the standard* against which the simulation results were measured; and 3) showing the VLSI architecture of the Winograd and PFA processors to be viable. The metric used to determine the numerical accuracy was the signal-to-noise (SNR) between the outputs from a standard module and the difference between standard and simulation modules. The SNR for the 15-point and 16-point modules averaged 127 dB (11 dB down from the standard), while the SNR for the 17-point module averaged 121 dB (18 dB down from the standard). These results were in excellent agreement with theoretical expectations (i.e., losses due to arithmetic rounding and scaling).

The outputs from the standard module were checked against results from a direct computation of the DFT. Comparison of the Winograd modules (15, 16, and 17) with the direct DFT showed an SNR in excess of 270 dB. The SNR of the direct DFT and the 2-factor PFA modules (240, 255, and 272) was 170 dB. These results indicate the standard modules were computing the DFT, rather than some other mathematical

operation.

The VLSI Winograd processors operate autonomously, requiring simple handshaking with an interface processor. The bit-serial architecture of the arithmetic circuitry allowed fast internal clock rates (targeted 70 MHz for 1.25 μs CMOS circuits) and high throughput (over 8300 4080-point DFTs computed per second if PFA pipeline architecture used). Fault tolerance was integral to the design effort, with watchdog processors and parity checking enabling the detection of errors in the data, addressing, and memory circuits. As seen in Chapters 3 and 4, the throughput and numerical accuracy of the proposed VLSI chips meets existing SAR requirements.

5.3. Conclusions.

There are conclusions to be drawn from each area of the research (theory, VLSI implementation, and numerical simulation).

5.3.1. Theory:

1) Winograd modules provide for the most efficient realization of the Good-Thomas PFA in terms of number of multiplications (i.e., using Winograd modules guarantees the fewest number of multiplications for a given DFT blocklength used in the Good-Thomas PFA);

2) The use of the Good-Thomas PFA allows for smaller multiplication matrices, compared to using the large Winograd algorithm; thus, even though the large Winograd algorithm actually has fewer multiplications, the Good-Thomas PFA may be preferred because the multiplication matrices are smaller (easier to store in a memory and implement in a VLSI circuit).

5.3.2. VLSI Implementation:

1) Using the Good-Thomas PFA with Winograd modules allows an efficient use of area on the chip; the VLSI circuit described in Chapter 3 can perform more multiplications per square centimeter of silicon for a given DFT blocklength than any other VLSI circuit implementing a DFT algorithm;

2) The use of Winograd processors with dual-ported memories allows for several DFT blocklengths to be computed using the same building blocks (e.g., a 255-point DFT can be performed by combining the 15-point and 17-point processors with the appropriate memories; the addressing and control circuitry exist on the processors themselves to perform the different DFT blocklengths[22]);

3) The design of watchdog and active processors allows for a fault tolerant architecture which can withstand several faults and continue to deliver correct results. The interface processor can reconfigure the processors, taking the faulty processor off-line and replacing it with one of the watchdog processors (other fault tolerant characteristics of the VLSI design is use of parity in the data representation and the use of on-chip error-correction coding (ECC) on the data memories).

5.3.3. Numerical Simulation:

1) The most significant source of noise in the simulation is the scaling of the inputs to prevent arithmetic overflow;

2) The outputs from the standard module do compute the DFT;

3) The outputs from the standard module must be scaled for comparison with outputs from the simulation module;

4) According to the simulation results, the VLSI circuit should provide acceptable numerical accuracy for DFT blocklengths of 15, 16, and 17.

5.4. Recommendations.

The recommendations for future research fall into three categories: theory, VLSI implementation, and numerical simulation.

5.4.1. Theory. The recommendation for this area of the research is to verify the 17-point small Winograd algorithm. The algorithm used in the standard and simulation modules was adapted from Burrus and Johnson[13]; that algorithm was a modified Winograd algorithm. The modifications were matrix manipulations which changed some of the arithmetic operations.

5.4.2. VLSI Implementation:

1) Verify the control signals and their timing for all three Winograd processors;

2) Design the Winograd processors for fabrication;

3) Design a test plan (including test procedures and test vectors) for the fabricated processors.

5.4.3. Numerical Simulation:

1) Compute the numerical accuracy for the following DFT blocklengths: 240, 255, 272, and 4080.

2) Determine the effects of different coefficient wordlengths (20, 24, 28, and 32) on the numerical accuracy for each DFT blocklength to determine if a shorter coefficient wordlength can be used;

3) Determine the numerical accuracy of each DFT blocklength for several different inputs, such as sine waves, pulses, etc.

Bibliography

1. Aho, Alfred V., et al., "," in *The Design and Analysis of Computer Algrotihms*, Addison-Wesley Publishing Co., Reading, MA (1974).

2. Avizienis, Algirdas and John P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer* **8**(17) pp. 67-80 (August 1984).

3. Blahut, Richard E., "," in *Fast Algorithms for Digital Signal Processing*, Addison-Wesley Publishing Co., Reading, MA (1985).

4. Collins, James M., "Simulation and Modeling of a VLSI Winograd Fourier Transform Processor," *MS Thesis*, (GE/ENG/85D-9)Wright-Patterson AFB, OH, (December 1985).

5. Cooley, J and J. Tukey, "An Algorithm for the Machine Computation of Complex Fourier Series," *Mathematics of Computation*, (19) pp. 297-301 (1965).

6. Coutee, Paul W., "Arithmetic Circuitry for High Speed VLSI Winograd Fourier Transform Processors," *MS Thesis*, (GE/ENG/85D-11)Wright-Patterson AFB, OH, (December 1985).

7. Elliot, Douglas and K. Rao, "," in *Fast Transforms: Algorithms, Analyses, and Applications*, Academic Press, Inc., Orlando, FL (1983).

8. Farrell, James J. III., "Large-Scale Cost-Effective Packaging," *IEEE Micro* **3**(5) pp. 5-10 (June 1985).

9. Hovanessian, S. A., "," in *Introduction to Synthetic Array and Imaging Radars*, Artech House, Inc., Dedham, MA (1980).

10. Ihara, Hirokazu and Kinji Mori, "Autonomous Decentralized Computer Control Systems," *Computer* **8**(17) pp. 57-66 (August 1984).

11. Ishimoto, Syoji and et al., "A 256K Dual Port Memory," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 38-39 Lewis Winner, (February 1985).

12. Johnson, Dave, "The Intel 432--A VLSI Architecture for Fault-Tolerant Computer Systems," *Computer* **8**(17) pp. 40-48 (August 1984).

13. Johnson, Howard W. and Charles S. Burrus, "Large DFT Modules: 11, 13, 17, 19 and 25," *E. E. Technical Report #8105*, Department of Electrical Engineering, Rice University, (December 1981).

14. Kolba, Dean P. and Thomas W. Parks, "A Prime Factor FFT Algorithm Using High-Speed Convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing* **4**(25) pp. 281-294 (August 1977).

15. Kung, S. Y., "VLSI Array Processors," *IEEE ASSP Magazine* **3**(2) pp. 4-22 (July 1985).

16. McClellan, James H. and Charles M. Rader, "," in *Number Theory in Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ (1979).

17. Oppenheim, Alan V. and Ronald W. Schafer, "," in *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ (1975).

18. Oppenheim, Alan V., "," in *Applications of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ (1978).

19. Posa, John G., "The Megabit RAM: Made in Japan?," *High Technology* **6**(5) pp. 37-41 (June 1985).

20. Rader, Charles M., "Discrete Fourier Transforms When the Number of Data Samples is Prime," *Proceedings of the IEEE* **6**(56) pp. 1107-1108 (June 1968).

21. Reusens, Peter P., Richard W. Linderman, and Walter H. Ku, "Digital Signal Processing Capabilities of CUSP, A High Performance Bit-Serial VLSI Processor," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* pp. 16.1.1-16.1.4 (April 1983).

22. Rossbach, Paul C., "Control Circuitry for High Speed VLSI Winograd Fourier Transform Processors," *MS Thesis*, (GE/ENG/85D-35)Wright-Patterson AFB, OH, (December 1985).

23. Sarrazin, David and Miroslaw Malek, "Fault Tolerant Semiconductor Memories," *Computer* **8**(17) pp. 49-56 (August 1984).

24. Sieworek, Daniel, "Architecture of Fault Tolerant Computers," *Computer* **8**(17) pp. 9-18 (August 1984).

25. Silverman, Harvey F., "An Introduction to Programming the Winograd Fourier Tranform Algorithm (WFTA)," *IEEE Transactions on Acoustics, Speech, and Signal Processing* **2**(25) pp. 152-165 (April 1977).

26. Taylor, Ron and Mark Johnson, "A 1Mb CMOS DRAM with a Divided Bitline Matrix Architecture," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 353-354 Lewis Winner, (February 1985).

27. Toy, Wing and Michele Morganti, "Fault Tolerant Computing," *Computer* **8**(17) pp. 6-7 (August 1984).

28. Wallace, John and Walter Barnes, "Designing for Ultrahigh Availability: The UNIX RTR Operating System," *Computer* **8**(17) pp. 31-39 (August 1984).

29. Winograd, Shmuel, "On Computing the Discrete Fourier Transform," *Mathematics of Computation* **32**(141) pp. 175-199 (January 1978).

30. Yamada, Iunzo and et al., "A Submicron VLSI Memory with a 4b-at-a-Time Built-in ECC Circuit," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 104-105 Lewis Winner, (February 1984).

Appendix A

A 15-Point DFT Using Winograd's Large DFT Algorithm

A 15-point DFT is developed using Winograd's large DFT algorithm (reference paragraph 2.3.2.). The methods requires the use of 3-point and 5-point small Winograd modules; the equations describing these modules can be found in McClellan and Rader [] or Winograd []. The development of the 15-point DFT will take two steps:

1) Show the DFT can be written as a Kronecker product of the 3-point and 5-point small Winograd modules;

2) Expand the Kronecker product of the 3-point and 5-point small Winograd modules back into a DFT form.

To begin, the matrix representation of a 15-point DFT is given in Figure A-1.

The ω terms in the matrix (Figure A-1) are a mathematical shorthand for expressing the complex exponential terms in the DFT. The arguments of the exponentials are multiples of $(2\pi/15)$; if the DFT is thought of as sampled values of the z-transform, the unit circle is segmented into fifteen equal increments over the range $(0, 2\pi)$. The values of the ω terms will be used later for comparison with the results from the Kronecker product of the 5-point and 3-point modules; the values of the ω terms are given below:

ω^0: cos (0) = 1.0 jsin (0) = j0.0

ω^1: cos (-2π/15) = 0.9135 jsin (-2π/15) = -j0.4067

ω^2: cos (-4π/15) = 0.6691 jsin (-4π/15) = -j0.7431

ω^3: cos (-6π/15) = 0.3090 jsin (-6π/15) = -j0.9511

ω^4: cos (-8π/15) = -0.1095 jsin (-8π/15) = -j0.9945

ω^5: cos (-10π/15) = -0.5000 jsin (-10π/15) = -j0.8660

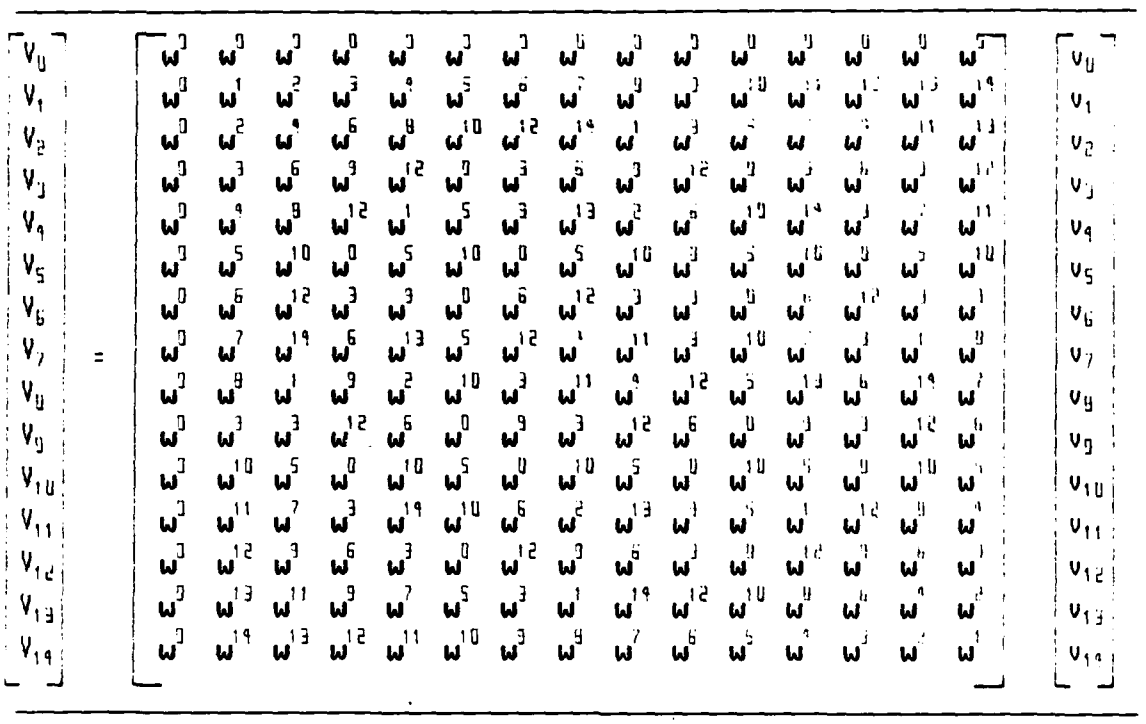


Figure A-1 Matrix Representation of DFT

ω^6: cos (-12π/15) = -0.8090 jsin (-12π/15) = -j0.5878

ω^7: cos (-14π/15) = -0.9781 jsin (-14π/15) = -j0.2079

ω^8: cos (-16π/15) = -0.9781 jsin (-16π/15) = j0.2079

ω^9: cos (-18π/15) = -0.8090 jsin (-18π/15) = j0.5878

ω^{10}: cos (-20π/15) = -0.5000 jsin (-20π/15) = j0.8660

ω^{11}: cos (-22π/15) = -0.1095 jsin (-22π/15) = j0.9945

ω^{12}: cos (-24π/15) = 0.3090 jsin (-24π/15) = j0.9511

ω^{13}: cos (-26π/15) = 0.6691 jsin (-26π/15) = j0.7431

ω^{14}: cos (-28π/15) = 0.9135 jsin (-28π/15) = j0.4067

The rows and columns of the matrix in Figure A-1 must be scrambled according to the rules shown in Figures 2-2 and 2-1, respectively. The row scrambling corresponds to

the output mapping, while the column scrambling corresponds to the input mapping. The column scrambling is shown in Figure A-2 and the row scrambling is shown in Figure A-3.

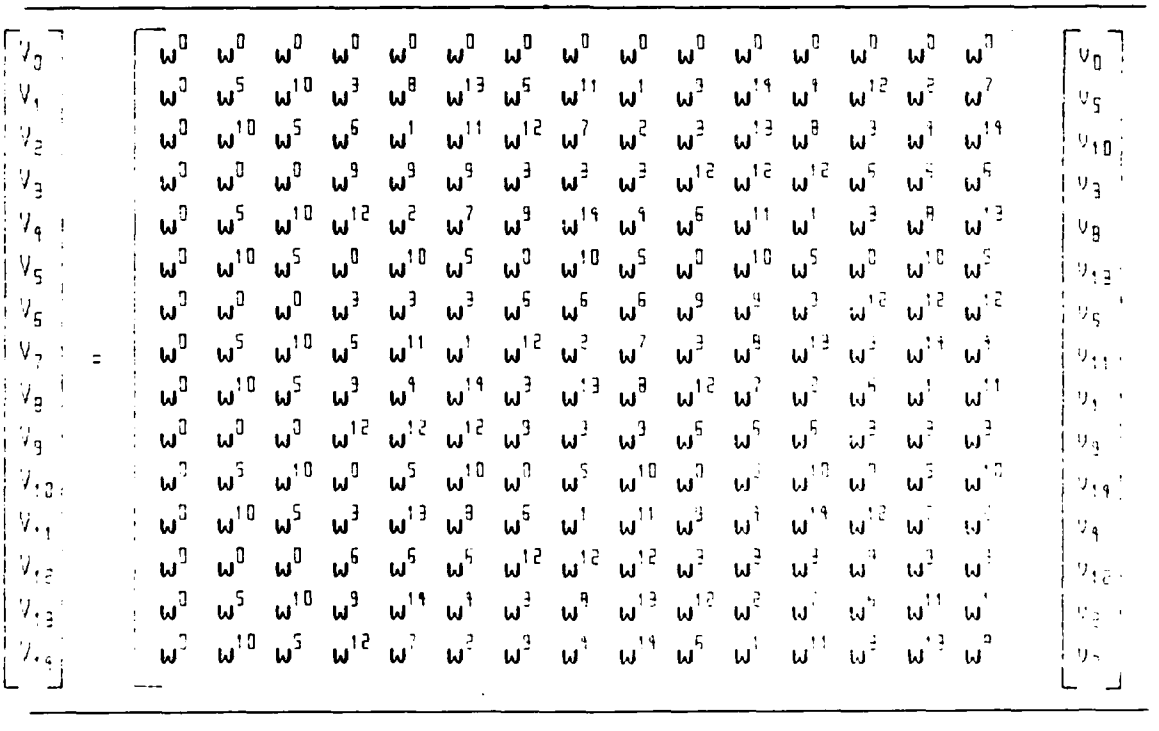


Figure A-2 Column Scrambling of DFT Matrix

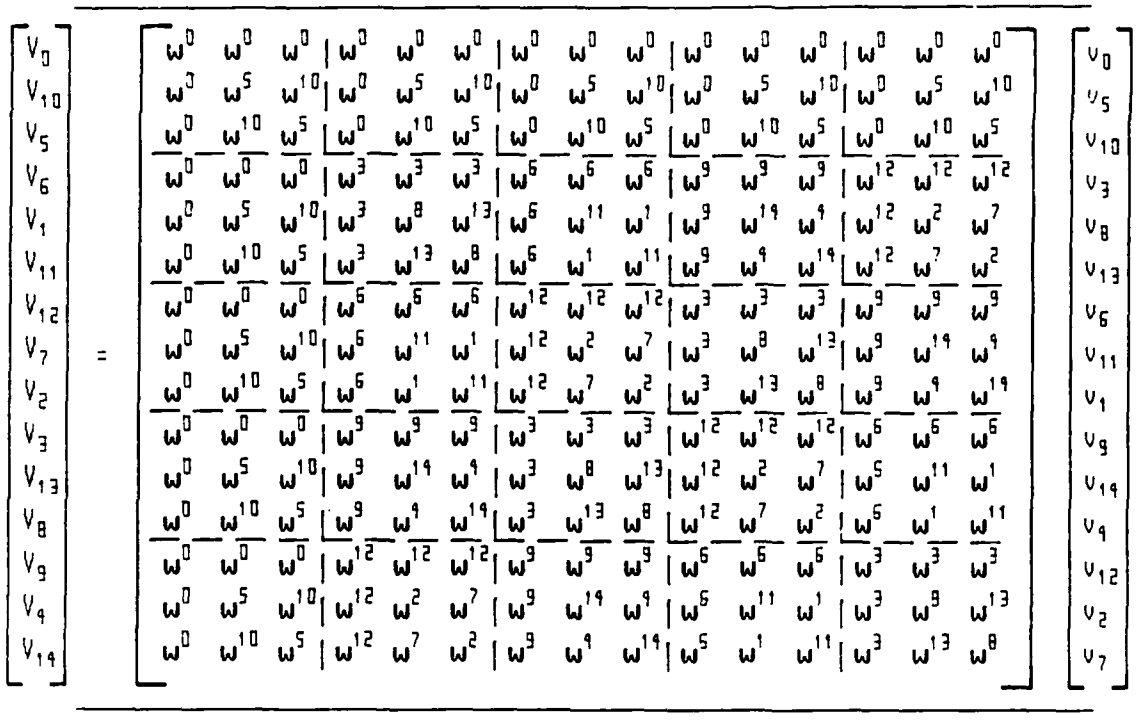


Figure A-3 Row Scrambling of DFT Matrix

Looking at Figure A-3, one can see a pattern of sub-matrices within the DFT matrix; these sub-matrices are the result of the Kronecker product of the 3-point and 5-point small Winograd modules. The Kronecker product of the 3-point and 5-point Winograd modules is shown in Figure A-4.

Thus, the 15-point DFT can be written as the Kronecker product of the 5-point and 3-point small Winograd modules. The matrix representations for the 5-point and 3-point modules for shown in Figures A-5a and A-5b, respectively.

The output vector, **V**, may be expressed, using the associative property of Kronecker products for matrices, as follows:

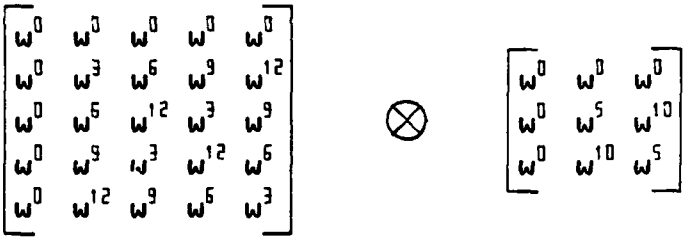


Figure A-4 Kronecker Product of 5-Point and 3-Point
Small Winograd Modules

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & j & 1 & 0 & j \\ 1 & 1 & 0 & -1 & -j & -j \\ 1 & 1 & 0 & -1 & j & j \\ 1 & 1 & -j & 1 & 0 & -j \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1.25 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1.53 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.56 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.36 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.59 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 1 & -1 \end{bmatrix}$$

Figure A-5a Matrix Representation of 5-Point
Small Winograd Module

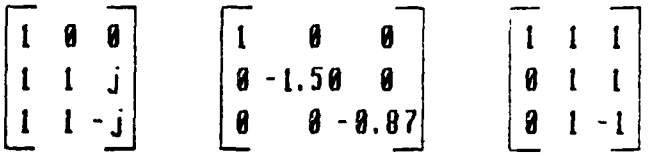


Figure A-5b Matrix Representation of 3-Point
Small Winograd Module

$$\begin{aligned} \mathbf{V} &= (\mathbf{V}_1 \otimes \mathbf{V}_2)\,\underline{x} \\ &= [(\mathbf{C}_1\,\mathbf{B}_1\,\mathbf{A}_1) \otimes [(\mathbf{C}_2\,\mathbf{B}_2\,\mathbf{A}_2)\,\underline{x} \\ &= [(\mathbf{C}_1 \otimes \mathbf{C}_2)(\mathbf{B}_1 \otimes \mathbf{B}_2)(\mathbf{A}_1 \otimes \mathbf{A}_2)\,\underline{x} \end{aligned}$$

Thus, the pre-addition matrix of the 15-point DFT (Figure A-6a) is the Kronecker product of the 5-point and 3-point pre-addition matrices (Figure A-6b). The multiplicative and pre-addition matrices for the 15-point DFT (figures A-7b and A-8b, respectively) are also Kronecker products of their 5-point and 3-point counterparts (figures A-7b and A-8b, respectively).

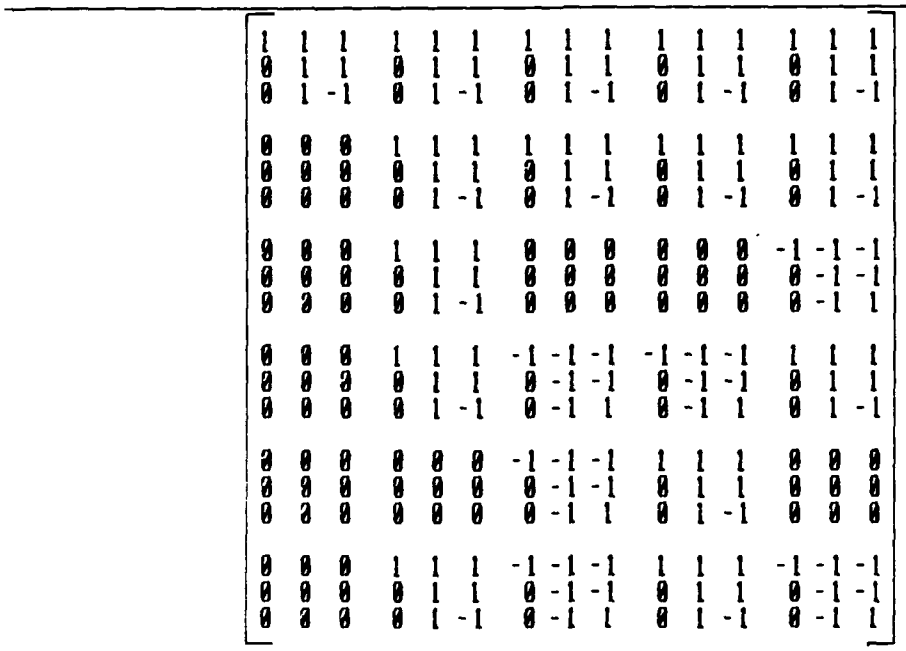


Figure A-6a Pre-Addition Matrix for 15-Point DFT

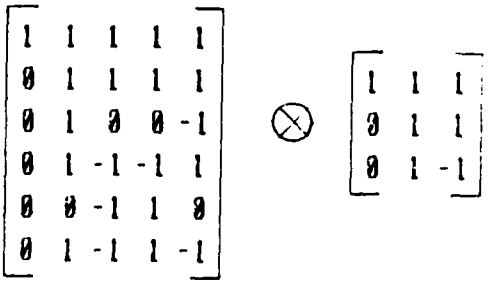


Figure A-6b Kronecker Product of 5-Point and 3-Point Pre-Addition Matrices

1 x 1 = 1	1 x -1.5 = -1.5	1 x -0.87 = -0.866
-1.25 x 1 = -1.25	-1.25 x -1.5 = 1.875	-1.25 x -0.87 = 1.083
-1.53 x 1 = -1.539	-1.53 x -1.5 = 2.300	-1.53 x -0.87 = 1.333
0.559 x 1 = 0.599	0.559 x -1.5 = -0.839	0.559 x -0.87 = -0.484
0.363 x 1 = 0.363	0.363 x -1.5 = -0.545	0.363 x -0.87 = -0.315
0.587 x 1 = 0.588	0.587 x -1.5 = -0.882	0.587 x -0.87 = -0.509

Figure A-7a Multiplicative Coefficients for 15-Point DFT

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1.25 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1.53 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.56 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.36 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.59 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1.50 & 0 \\ 0 & 0 & -0.87 \end{bmatrix}$$

Figure A-7b Kronecker Product of 5-Point and 3-Point Multiplicative Matrices

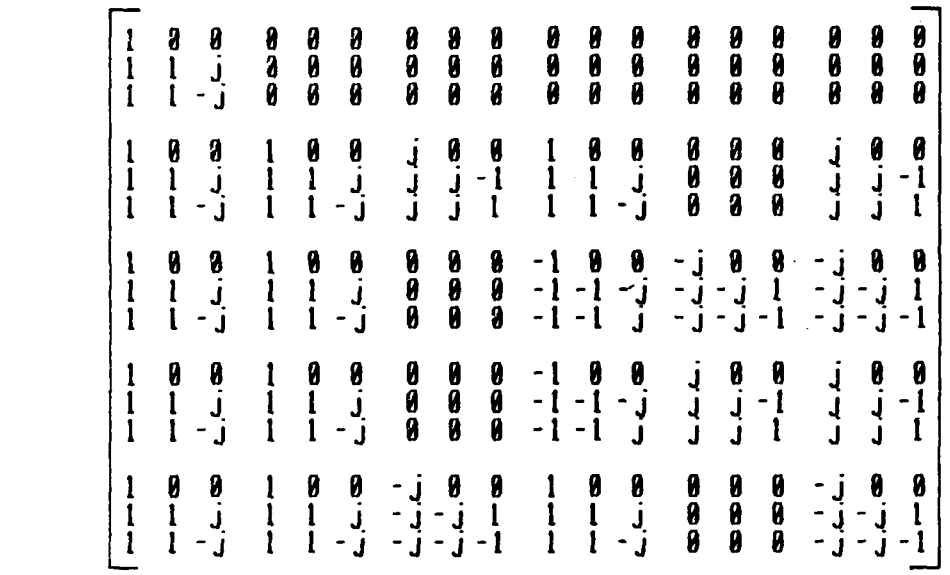


Figure A-8a Post-Addition Matrix for 15-Point DFT

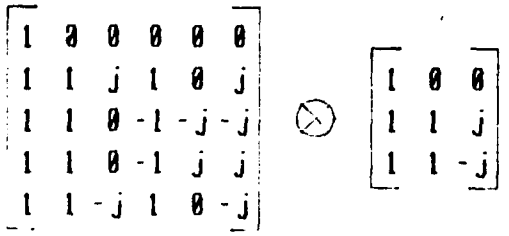


Figure A-8b Kronecker Product of 5-Point and 3-Point Post-Addition Matrices

To determine if the Kronecker product of the 5-point and 3-point modules does result in a 15-point DFT, take the product of all three matrices to see if the resulting matrix is identical to the scrambled DFT matrix (reference Figure A-3). First, take the product of the pre-addition and multiplicative matrices.

Multiply the product shown in Figure A-9 by the post-addition matrix reference Figure A-8a).

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	-1.50	-1.50	0	-1.50	-1.50	0	-1.50	-1.50	0	-1.50	-1.50	0	-1.50	-1.50
0	-0.87	0.87	0	-0.87	0.87	0	-0.87	0.87	0	-0.87	0.87	0	-0.87	0.87
0	0	0	-1.25	-1.25	-1.25	-1.25	-1.25	-1.25	-1.25	-1.25	-1.25	-1.25	-1.25	-1.25
0	0	0	0	1.88	1.88	0	1.88	1.88	0	1.88	1.88	0	1.88	1.88
0	0	0	0	1.08	-1.08	0	1.08	-1.08	0	1.08	-1.08	0	1.08	-1.08
0	0	0	-1.53	-1.53	-1.53	0	0	0	0	0	0	1.53	1.53	1.53
0	0	0	0	2.31	2.31	0	0	0	0	0	0	0	-2.31	-2.31
0	0	0	0	1.33	-1.33	0	0	0	0	0	0	0	-1.33	1.33
0	0	0	0.56	0.56	0.56	-0.56	-0.56	-0.56	-0.56	-0.56	-0.56	0.56	0.56	0.56
0	0	0	0	-0.89	-0.89	0	0.89	0.89	0	0.89	0.89	0	-0.89	-0.89
0	0	0	0	-0.98	0.98	0	0.98	-0.98	0	0.98	-0.98	0	-0.98	0.98
0	0	0	0	0	0	-0.36	-0.36	-0.36	0.36	0.36	0.36	0	0	0
0	0	0	0	0	0	0	0.59	0.59	0	-0.59	-0.59	0	0	0
0	0	0	0	0	0	0	0.31	-0.31	0	-0.31	0.31	0	0	0
0	0	0	0.59	0.59	0.59	-0.59	-0.59	-0.59	0.59	0.59	0.59	-0.59	-0.59	-0.59
0	0	0	0	-0.88	-0.88	0	0.88	0.88	0	-0.88	-0.88	0	0.88	0.88
0	0	0	0	-0.51	0.51	0	0.51	-0.51	0	-0.51	0.51	0	0.51	-0.51

Figure A-9 Product of Pre-Addition and Multiplicative Matrices

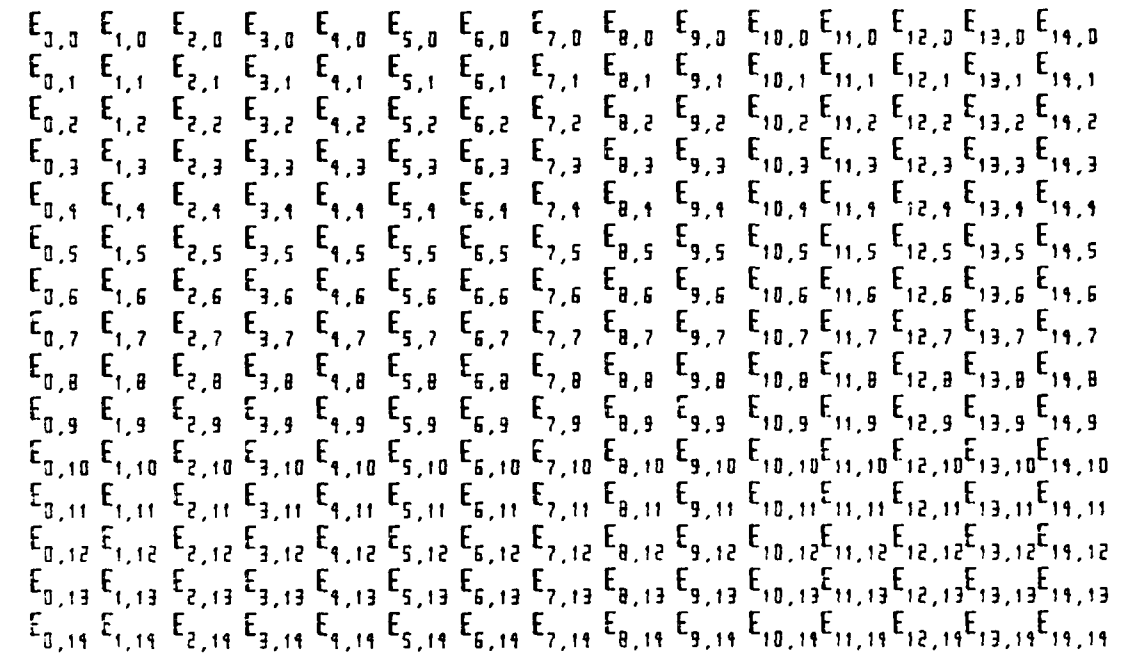


Figure A-10 Product of Pre-Addition, Multiplicative, and Post-Addition Matrices

The expressions for the elements of the matrix shown in Figure A-10 are given on the following pages:

$$E_{0,0} = E_{0,1} = E_{0,2} = E_{0,3} = E_{0,4} = 1 = \omega^0$$

$$E_{0,5} = E_{0,6} = E_{0,7} = E_{0,8} = E_{0,9} = 1 = \omega^0$$

$$E_{0,10} = E_{0,11} = E_{0,12} = E_{0,13} = E_{0,14} = 1 = \omega^0$$

$$E_{1,0} = E_{1,3} = E_{1,6} = E_{1,9} = E_{1,12} = 1 = \omega^0$$

$$E_{1,1} = E_{1,4} = E_{1,7} = E_{1,10} = E_{1,13} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^3$$

$$E_{1,2} = E_{1,5} = E_{1,8} = E_{1,11} = E_{1,14} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{2,0} = E_{2,3} = E_{2,6} = E_{2,9} = E_{2,12} = 1 = \omega^{0}$$

$$E_{2,1} = E_{2,4} = E_{2,7} = E_{2,10} = E_{2,13} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{2,2} = E_{2,5} = E_{2,8} = E_{2,11} = E_{2,14} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^{5}$$

$$E_{3,0} = E_{3,1} = E_{3,2} = 1 = \omega^{0}$$

$$E_{3,3} = E_{3,4} = E_{3,5} = 1 - 1.25 - j1.53 + 0.56 + j0.59 = 0.31 - j0.95 = \omega^{3}$$

$$E_{3,6} = E_{3,7} = E_{3,8} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{3,9} = E_{3,10} = E_{3,11} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j\,0.59 = \omega^{9}$$

$$E_{3,12} = E_{3,13} = E_{3,14} = 1 - 1.25 + j1.53 + 0.56 - j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{4,0} = 1 = \omega^{0}$$

$$E_{4,1} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^{5}$$

$$E_{4,2} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{4,3} = 1 - 1.25 - j1.53 - 0.56 + j0.59 = 0.31 - j0.95 = \omega^{3}$$

$$E_{4,4} = 1 - 1.5 - j0.87 - 1.25 + 1.88 - j1.08 - j1.53 + j2.31 - 1.33 - 0.56 - 0.84 - j0.48 - j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^{8}$$

$$E_{4,5} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - j1.53 - j2.31 - 1.33 - 0.56 - 0.84 - j0.48 - j0.59 - j0.88 - 0.51 = 0.67 - j0.74 = \omega^{13}$$

$$E_{4,6} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{4,7} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 - 0.84 - j0.48 - j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^{11}$$

$$E_{4,8} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48 - j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^{1}$$

$$E_{4,9} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^{9}$$

$$E_{4,10} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48 + j0.59 - j0.88 + 0.51 = 0.92 + j0.41 = \omega^{14}$$

$$E_{4,11} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48 + j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^{4}$$

$$E_{4,12} = 1 - 1.25 + j1.53 + 0.56 - j0.59 = 0.31 - j0.95 = \omega^{12}$$

$$E_{4,13} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + j1.53 - j2.31 + 1.33 + 0.56 - 0.84 - j0.48 - j0.59 + j0.88 - 0.51 = 0.67 - j0.74 = \omega^{2}$$

$$E_{4,14} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + j1.53 - j2.31 - 1.33 + 0.56 - 0.84 + j0.48 - j0.59 + j0.88 + 0.51 = -0.98 - j0.21 = \omega^{7}$$

$$E_{5,0} = 1 = \omega^{0}$$

$$E_{5,1} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{5,2} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^{5}$$

$$E_{5,3} = 1 - 1.25 - j1.53 + 0.56 + j0.59 = 0.31 - j0.95 = \omega^{3}$$

$$E_{5,4} = 1 - 1.5 + j0.87 - 1.25 - 1.88 + j1.08 - j1.53 + j2.31 + 1.33 + 0.56 - 0.84 + j0.48 + j0.59 - j0.88 - 0.51 = 0.67 + j0.74 = \omega^{13}$$

$$E_{5,5} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - j1.53 + j2.31 + 1.33 + 0.56 - 0.84 - j0.48 + j0.59 - j0.88 - 0.51 = -0.98 + j0.21 = \omega^{8}$$

$$E_{5,6} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{5,7} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 - j0.48$$
$$- j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^{1}$$

$$E_{5,8} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48$$
$$- j0.59 + j0.88 - 0.51 = -0.10 + j0.99 = \omega^{11}$$

$$E_{5,9} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^{9}$$

$$E_{5,10} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 - j0.48$$
$$+ j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^{4}$$

$$E_{5,11} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48$$
$$+ j0.59 - j0.88 + 0.51 = 0.92 + j0.41 = \omega^{14}$$

$$E_{5,12} = 1 - 1.25 + j1.53 + 0.56 - j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{5,13} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + j1.53 - j2.31 - 1.33$$
$$+ 0.56 - 0.84 + j0.48 - j0.59 + j0.88 + 0.51 = -0.98 - j0.21 = \omega^{7}$$

$$E_{5,14} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + j1.53 - j2.31 + 1.33$$
$$+ 0.56 - 0.84 - j0.48 - j0.59 + j0.88 - 0.51 = 0.67 - j0.74 = \omega^{2}$$

$$E_{6,0} = E_{6,1} = E_{6,2} = 1 = \omega^{0}$$

$$E_{6,3} = E_{6,4} = E_{6,5} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{6,6} = E_{6,7} = E_{6,8} = 1 - 1.25 + 0.56 + j\,0.36 + j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{6,9} = E_{6,10} = E_{6,11} = 1 - 1.25 + 0.56 - j\,0.36 - j0.59 = 0.31 - j0.95 = \omega^{3}$$

$$E_{6,12} = E_{6,13} = E_{6,14} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^{9}$$

$$E_{7,0} = 1 = \omega^{0}$$

$$E_{7,1} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^{5}$$

$$E_{7,2} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{7,3} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{7,4} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - 0.56 + 0.84 + j0.48$$
$$- j0.59 + j0.88 - 0.51 = -0.10 + j0.99 = \omega^{11}$$

$$E_{7,5} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 - 0.56 + 0.84 - j0.48$$
$$- j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^{1}$$

$$E_{7,6} = 1 - 1.25 + j0.36 + 0.56 + j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{7,7} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48$$
$$+ j0.36 - j0.54 + 0.31 + j0.59 - j0.88 + 0.51 = 0.67 - j0.74 = \omega^{2}$$

$$E_{7,8} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$+ j0.36 - j0.54 - 0.31 + j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^{7}$$

$$E_{7,9} = 1 - 1.25 - 0.56 - j0.36 - j0.59 = 0.31 - j0.95 = \omega^{3}$$

$$E_{7,10} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48$$
$$- j0.36 + j0.54 - 0.31 - j0.59 + j0.88 - 0.51 = -0.98 - j0.21 = \omega^{8}$$

$$E_{7,11} = 1 - 1.5 - j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$- j0.36 - j0.54 + 0.31 - j0.59 + j0.88 + 0.51 = 0.67 - j0.74 = \omega^{13}$$

$$E_{7,12} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{9}$$

$$E_{7,13} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 - 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = 0.92 - j0.41 = \omega^{14}$$

$$E_{7,14} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 - 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^{4}$$

$$E_{8,0} = 1 = \omega^{0}$$

$$E_{8,1} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{8,2} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^{5}$$

$$E_{8,3} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{8,4} = 1 - 1.5 + j0.87 - 1.25 - 1.88 + j1.08 - 0.56 + 0.84 - j0.48$$
$$- j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^{1}$$

$$E_{8,5} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - 0.56 + 0.84 + j0.48$$
$$- j0.59 + j0.88 - 0.51 = -0.10 + j0.99 = \omega^{11}$$

$$E_{8,6} = 1 - 1.25 + 0.56 + j0.36 + j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{8,7} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$+ j0.36 - j0.54 - 0.31 + j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^{7}$$

$$E_{8,8} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48$$
$$+ j0.36 - j0.54 - 0.31 + j0.59 - j0.88 - 0.51 = 0.67 - j0.74 = \omega^{2}$$

$$E_{8,9} = 1 - 1.25 + 0.56 - j0.36 - j0.59 = 0.31 - j0.95 = \omega^{3}$$

$$E_{8,10} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$- j0.36 - j0.54 + 0.31 - j0.59 + j0.88 + 0.51 = 0.67 + j0.74 = \omega^{13}$$

$$E_{8,11} = 1 - 1.5 - j0.87 - 1.25 - 1.88 + j1.08 + 0.56 - 0.84 - j0.48$$
$$- j0.36 - j0.54 - 0.31 - j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^{8}$$

$$E_{8,12} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{9}$$

$$E_{8,13} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 - 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^{4}$$

$$E_{8,14} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 - 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = -0.92 - j0.41 = \omega^{14}$$

$$E_{9,0} = E_{9,1} = E_{9,2} = 1 = \omega^0$$

$$E_{9,3} = E_{9,4} = E_{9,5} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^9$$

$$E_{9,6} = E_{9,7} = E_{9,8} = 1 - 1.25 + 0.56 + j\,0.36 - j0.59 = 0.31 - j0.95 = \omega^3$$

$$E_{9,9} = E_{9,10} = E_{9,11} = 1 - 1.25 + 0.56 - j\,0.36 + j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{9,12} = E_{9,13} = E_{9,14} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^6$$

$$E_{10,0} = 1 = \omega^0$$

$$E_{10,1} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^5$$

$$E_{10,2} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{10,3} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^9$$

$$E_{10,4} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - 0.56 + 0.84 + j0.48 + j0.59 - j0.88 + 0.51 = 0.92 + j0.41 = \omega^{14}$$

$$E_{10,5} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 - 0.56 + 0.84 - j0.48 - j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^4$$

$$E_{10,6} = 1 - 1.25 + 0.56 - j0.36 - j0.59 = 0.31 - j0.95 = \omega^3$$

$$E_{10,7} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48 - j0.36 - j0.54 - 0.31 - j0.59 + j0.88 - 0.51 = -0.98 - j0.21 = \omega^8$$

$$E_{10,8} = 1 - 1.5 - j0.87 - 1.25 + 1.88 - j1.08 - 0.56 - 0.84 - j0.48 - j0.36 - j0.54 - 0.31 - j0.59 - j0.88 - 0.51 = 0.67 - j0.74 = \omega^{13}$$

$$E_{10,9} = 1 - 1.25 - 0.56 + j0.36 - j0.59 = 0.31 - j0.95 = \omega^{12}$$

$$E_{10,10} = 1 - 1.5 - j0.87 - 1.25 + 1.88 - j1.08 - 0.56 - 0.84 - j0.48 - j0.36 - j0.54 - 0.31 - j0.59 - j0.88 - 0.51 = 0.67 - j0.74 = \omega^2$$

$$E_{10,11} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$+ j0.36 - j0.54 - 0.31 + j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^{7}$$

$$E_{10,12} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{10,13} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - 0.56 + 0.84 + j0.48$$
$$- j0.59 + j0.88 - 0.51 = -0.10 + j0.99 = \omega^{11}$$

$$E_{10,14} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 - 0.56 + 0.84 - j0.48$$
$$- j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^{1}$$

$$E_{11,0} = 1 = \omega^{0}$$

$$E_{11,1} = 1 - 1.5 - j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{11,2} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^{5}$$

$$E_{11,3} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^{9}$$

$$E_{11,4} = 1 - 1.5 + j0.87 - 1.25 - 1.88 + j1.08 - 0.56 + 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^{4}$$

$$E_{11,5} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 + 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = 0.92 - j0.41 = \omega^{14}$$

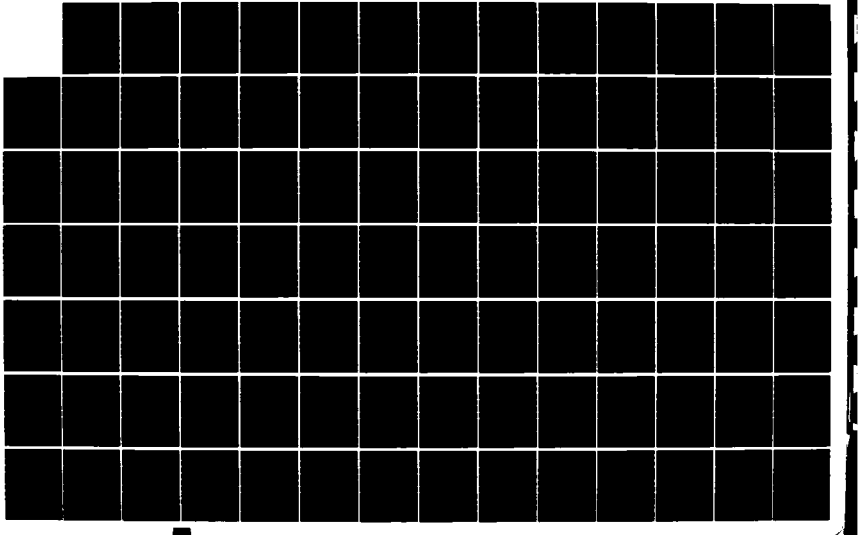
$$E_{11,6} = 1 - 1.25 + 0.56 - j0.36 - j0.59 = 0.31 - j0.95 = \omega^{3}$$

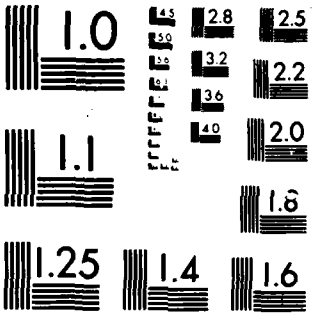
$$E_{11,7} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 - 0.84 - j0.48$$
$$- j0.36 - j0.54 - 0.31 - j0.59 - j0.88 - 0.51 = 0.67 - j0.74 = \omega^{13}$$

$$E_{11,8} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - 0.56 - 0.84 - j0.48$$
$$- j0.36 - j0.54 - 0.31 - j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^{8}$$

$$E_{11,9} = 1 - 1.25 - 0.56 + j0.36 - j0.59 = 0.31 - j0.95 = \omega^{12}$$

AD-A163 963 ARCHITECTURE AND NUMERICAL ACCURACY OF HIGH-SPEED DFT 2/3
(DISCRETE FOURIER T.. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K TAYLOR
UNCLASSIFIED DEC 85 AFIT/GE/ENG/85D-47 F/G 9/1 NL





$$E_{11,10} = 1 - 1.5 - j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$- j0.36 - j0.54 - 0.31 + j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^7$$

$$E_{11,11} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48$$
$$+ j0.36 - j0.54 + 0.31 + j0.59 - j0.88 + 0.51 = 0.67 - j0.74 = \omega^2$$

$$E_{11,12} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^6$$

$$E_{11,13} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 - j0.48$$
$$- j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^1$$

$$E_{11,14} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 + j0.48$$
$$- j0.59 + j0.88 - 0.51 = -0.10 + j0.99 = \omega^{11}$$

$$E_{12,0} = E_{12,1} = E_{12,2} = 1 = \omega^0$$

$$E_{12,3} = E_{12,4} = E_{12,5} = 1 - 1.25 + j1.53 + 0.56 - j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{12,6} = E_{12,7} = E_{12,8} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^9$$

$$E_{12,9} = E_{12,10} = E_{12,11} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j\ 0.59 = \omega^6$$

$$E_{12,12} = E_{12,13} = E_{12,14} = 1 - 1.25 - j1.53 - 0.56 - j0.59 = 0.31 - j0.95 = \omega^3$$

$$E_{13,0} = 1 = \omega^0$$

$$E_{13,1} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^5$$

$$E_{13,2} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{13,3} = 1 - 1.25 + j1.53 - 0.56 - j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{13,4} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - j1.53 - j2.31 + 1.33$$
$$+ 0.56 - 0.84 - j0.48 - j0.59 + j0.88 - 0.51 = 0.67 - j0.74 = \omega^2$$

$$E_{13,5} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + j1.53 - j2.31 - 1.33$$
$$+ 0.56 - 0.84 + j0.48 - j0.59 + j0.88 + 0.51 = -0.98 - j0.21 = \omega^7$$

$$E_{13,6} = 1 - 1.25 - 0.56 + j0.59 = -0.81 - j0.59 = \omega^{9}$$

$$E_{13,7} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48 + j0.59 - j0.88 + 0.51 = 0.92 + j0.41 = \omega^{14}$$

$$E_{13,8} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48 + j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^{4}$$

$$E_{13,9} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^{6}$$

$$E_{13,10} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 + 0.56 - 0.84 - j0.48 - j0.59 + j0.88 - 0.51 = -0.10 + j0.99 = \omega^{11}$$

$$E_{13,11} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48 - j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^{1}$$

$$E_{13,12} = 1 - 1.25 - j1.53 + 0.56 + j0.59 = 0.31 - j0.95 = \omega^{3}$$

$$E_{13,13} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - j1.53 + j2.31 - 1.33 + 0.56 - 0.84 - j0.48 + j0.59 - j0.88 + 0.51 = -0.98 + j0.21 = \omega^{8}$$

$$E_{13,14} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 - j1.53 + j2.31 - 1.33 + 0.56 - 0.84 + j0.48 + j0.59 - j0.88 - 0.51 = 0.67 + j0.74 = \omega^{13}$$

$$E_{14,0} = 1 = \omega^{0}$$

$$E_{14,1} = 1 - 1.5 + j0.87 = -0.5 + j0.87 = \omega^{10}$$

$$E_{14,2} = 1 - 1.5 - j0.87 = -0.5 - j0.87 = \omega^{5}$$

$$E_{14,3} = 1 - 1.25 + j1.53 - 0.56 - j0.59 = 0.31 + j0.95 = \omega^{12}$$

$$E_{14,4} = 1 - 1.5 + j0.87 - 1.25 - 1.88 + j1.08 + j1.53 - j2.31 - 1.33 + 0.56 - 0.84 + j0.48 - j0.59 + j0.88 + 0.51 = -0.98 - j0.21 = \omega^{7}$$

$$E_{14,5} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - j1.53 - j2.31 - 1.33$$
$$- 0.56 - 0.84 - j0.48 - j0.59 - j0.88 - 0.51 = 0.67 - j0.74 = \omega^2$$

$$E_{14,6} = 1 - 1.25 - 0.56 + j0.59 = -0.81 + j0.59 = \omega^9$$

$$E_{14,7} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$+ j0.59 - j0.88 - 0.51 = -0.10 - j0.99 = \omega^4$$

$$E_{14,8} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 + 0.56 - 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = 0.92 + j0.41 = \omega^{14}$$

$$E_{14,9} = 1 - 1.25 - 0.56 - j0.59 = -0.81 - j0.59 = \omega^6$$

$$E_{14,10} = 1 - 1.5 + j0.87 - 1.25 + 1.88 - j1.08 + 0.56 - 0.84 + j0.48$$
$$- j0.59 + j0.88 + 0.51 = 0.92 - j0.41 = \omega^1$$

$$E_{14,11} = 1 - 1.5 - j0.87 - 1.25 + 1.88 + j1.08 - 0.56 - 0.84 - j0.48$$
$$- j0.59 - j0.88 - 0.51 = -0.10 + j0.99 = \omega^{11}$$

$$E_{14,12} = 1 - 1.25 - j1.53 + 0.56 + j0.59 = 0.31 - j0.95 = \omega^3$$

$$E_{14,13} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - j1.53 - j2.31 - 1.33$$
$$- 0.56 - 0.84 - j0.48 - j0.59 - j0.88 - 0.51 = 0.67 - j0.74 = \omega^{13}$$

$$E_{14,14} = 1 - 1.5 - j0.87 - 1.25 - 1.88 - j1.08 - j1.53 - j2.31 - 1.33$$
$$- 0.56 - 0.84 - j0.48 - j0.59 - j0.88 - 0.51 = -0.98 - j0.21 = \omega^8$$

Figure A-11 shows the product of the pre-addition, multiplicative, and post-addition matrices, substituting the ω terms for the E elements shown in Figure A-10.

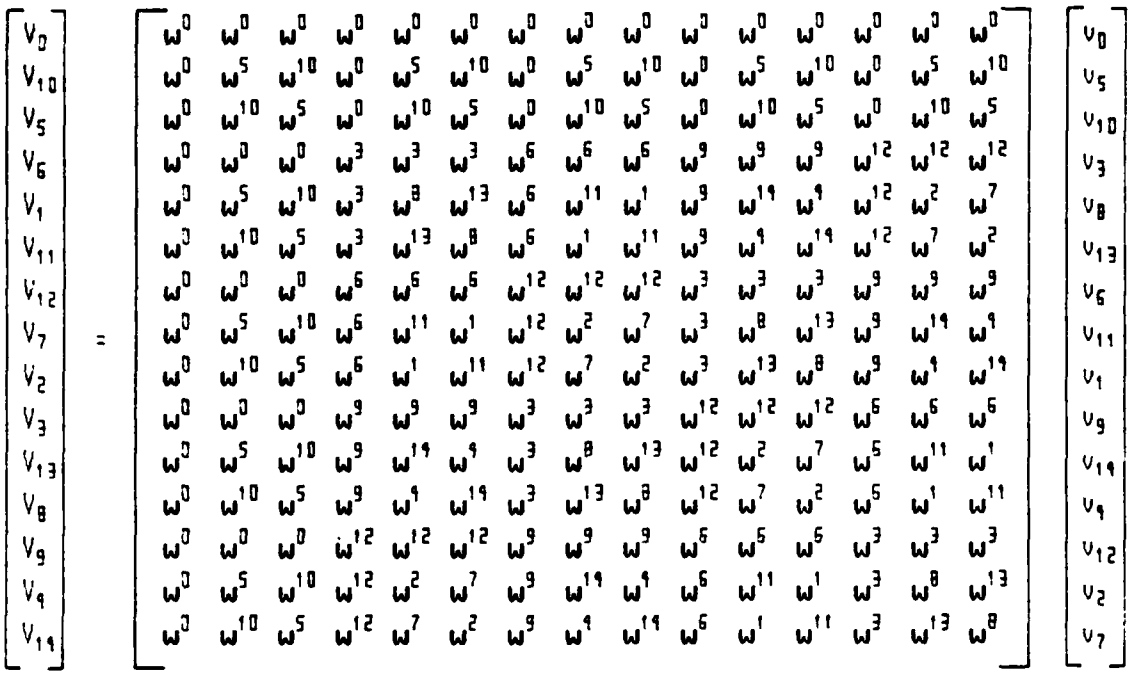


Figure A-11 Product of Pre-Addition, Multiplicative, and Post-Addition Matrices Using ω Terms

The matrix shown in Figure A-11 is identical to the matrix shown in Figure A-3. Thus, the Kronecker product of the 5-point and 3-point Winograd modules does give a 15-point DFT.

Appendix B

Simulation Program Listings

The following program listings indicate the code required for the simulation and standard modules of the simulation programs. The standard modules are the wino files; these files compute Winograd algorithms using double-precision arithmetic. The simulation modules are the sim files; theses files compute Winograd algorithms using integer arithmetic. Only the sim16.c listing is shown; the sim15.c and sim17.c listings are identical to their wino counterparts except for the multiplication of the pre-addition results with the coefficients and the coefficients themselves. The manner in which multiplication is done is illustrated in the sim16.c file. The coefficients for the 15- and 17-point simulation modules are given following the sim16.c listing. The multiply.c listing shows the special multiply routine used by the simulation module; this routine accepts a 28-bit input (data) and a 32-bit input (coefficient) and returns a 32-bit result. The diff programs show how the standard and simulation modules are combined to give comparison results from a single program. The stddiff programs show how the standard modules are combined with the direct DFT.

B.1 SIM16.C

```
/*************************************************************************
*
*       Module:  sim16.c
*
*       Author:  Kent Taylor
*
*       Date:    13 September 1985
*
*       Purpose:  To perform a 16-point DFT on complex input data.
*
*       Inputs:  x, y, h
*
*               x is the array of real input values
*               y is the array of imaginary input data
*               h is the index array
*
*       Outputs:  x, y
*
*************************************************************************/
#define         CS160   67108864
#define         CS161   47453133
#define         CS162   25681450
#define         CS163   87681956
#define         CS164   36319055
#define         CS165   62000506
sim16 (x, y, h)
long x[], y[];
int h[];
{
  long r100, r101, r102, r103, r104, r105, r106, r107, r108, r109;
  long r110, r111, r112, r113, r114, r115;
  long r200, r201, r202, r203, r204, r205, r206, r207, r208, r209;
  long r210, r211, r212, r213;
  long r300, r301, r302, r303, r304, r305, r306, r307;
  long r400, r401;
  long t02, t03, t04, t05, t06, t07, t08, t09;
  long t10, t11, t12, t13, t14, t15, t16, t17, t18, t19;
  long t100, t101, t102, t103, t104, t105, t106, t107, t108, t109, t110, t111;
```

```
long t200, t201, t202, t203, t204, t205, t206, t207;
long s100, s101, s102, s103, s104, s105, s106, s107, s108, s109;
long s110, s111, s112, s113, s114, s115;
long s200, s201, s202, s203, s204, s205, s206, s207, s208, s209;
long s210, s211, s212, s213;
long s300, s301, s302, s303, s304, s305, s306, s307;
long s400, s401;
long u00, u01, u02, u03, u04, u05, u06, u07, u08, u09;
long u10, u11, u12, u13, u14, u15, u16, u17, u18, u19;
long u100, u101, u102, u103, u104, u105, u106, u107, u108, u109, u110, u111;
long u200, u201, u202, u203, u204, u205, u206, u207;
/*************************************************************************
*
*       r variables are the real pre-add equations
*       s variables are the imaginary pre-add equations
*
*************************************************************************/
r100  = x[h[0]] + x[h[8]];
r101  = x[h[0]] - x[h[8]];
r102  = x[h[1]] + x[h[9]];
r103  = x[h[1]] - x[h[9]];
r104  = x[h[2]] + x[h[10]];
r105  = x[h[2]] - x[h[10]];
r106  = x[h[3]] + x[h[11]];
r107  = x[h[3]] - x[h[11]];
r108  = x[h[4]] + x[h[12]];
r109  = x[h[4]] - x[h[12]];
r110  = x[h[5]] + x[h[13]];
r111  = x[h[5]] - x[h[13]];
r112  = x[h[6]] + x[h[14]];
r113  = x[h[6]] - x[h[14]];
r114  = x[h[7]] + x[h[15]];
r115  = x[h[7]] - x[h[15]];
s100  = y[h[0]] + y[h[8]];
s101  = y[h[0]] - y[h[8]];
s102  = y[h[1]] + y[h[9]];
s103  = y[h[1]] - y[h[9]];
s104  = y[h[2]] + y[h[10]];
s105  = y[h[2]] - y[h[10]];
s106  = y[h[3]] + y[h[11]];
```

```
s107 = y[h[3]] - y[h[11]];
s108 = y[h[4]] + y[h[12]];
s109 = y[h[4]] - y[h[12]];
s110 = y[h[5]] + y[h[13]];
s111 = y[h[5]] - y[h[13]];
s112 = y[h[6]] + y[h[14]];
s113 = y[h[6]] - y[h[14]];
s114 = y[h[7]] + y[h[15]];
s115 = y[h[7]] - y[h[15]];
r200 = r100 + r108;
r201 = r100 - r108;
r202 = r112 + r104;
r203 = r112 - r104;
r204 = r102 + r110;
r205 = r102 - r110;
r206 = r106 + r114;
r207 = r106 - r114;
r208 = r103 + r115;
r209 = r103 - r115;
r210 = r111 + r107;
r211 = r111 - r107;
r212 = r105 + r113;
r213 = r105 - r113;
s200 = s100 + s108;
s201 = s100 - s108;
s202 = s112 + s104;
s203 = s112 - s104;
s204 = s102 + s110;
s205 = s102 - s110;
s206 = s106 + s114;
s207 = s106 - s114;
s208 = s103 + s115;
s209 = s103 - s115;
s210 = s111 + s107;
s211 = s111 - s107;
s212 = s105 + s113;
s213 = s105 - s113;
r300 = r200 + r202;
r301 = r200 - r202;
r302 = r206 + r204;
```

```
r303 = r206 - r204;
r304 = r205 - r207;
r305 = r205 - r207;
r306 = r209 + r211;
r307 = r208 + r210;
s300 = s200 + s202;
s301 = s200 - s202;
s302 = s206 + s204;
s303 = s206 - s204;
s304 = s205 + s207;
s305 = s205 - s207;
s306 = s209 + s211;
s307 = s208 + s210;
r400 = r300 + r302;
r401 = r300 - r302;
s400 = s300 + s302;
s401 = s300 - s302;
x[h[0]] = mult (r400, C160);
x[h[8]] = mult (r401, C160);
y[h[0]] = mult (s400, C160);
y[h[8]] = mult (s401, C160);
*************************************************************************
*
*       After the pre-adds, the sums are multiplied by the proper
*       coefficients.
*
*************************************************************************
t02 = mult (r301, C160);
t03 = mult (r201, C160);
t04 = mult (r101, C160);
t05 = mult (r305, C161);
t06 = mult (r213, C161);
t07 = mult (r306, C162);
t08 = mult (r209, C163);
t09 = -(mult (r211, C164));
t10 = mult (r303, C160);
t11 = mult (r203, C160);
t12 = -(mult (r109, C160));
t13 = -(mult (r304, C161));
t14 = -(mult (r212, C161));
```

```
t15 = -(mult (r307, C165));
t16 = mult (r208, C164);
t17 = -(mult (r210, C163));
u02 = mult (s301, C160);
u03 = mult (s201, C160);
u04 = mult (s101, C160);
u05 = mult (s305, C161);
u06 = mult (s213, C161);
u07 = mult (s306, C162);
u08 = mult (s209, C163);
u09 = -(mult (s211, C164));
u10 = mult (s303, C160);
u11 = mult (s203, C160);
u12 = -(mult (s109, C160));
u13 = -(mult (s304, C161));
u14 = -(mult (s212, C161));
u15 = -(mult (s307, C165));
u16 = mult (s208, C164);
u17 = -(mult (s210, C163));
*****************************************************************************
*
*       t variables are the real post-add equations
*       u variables are the imaginary post-add equations
*
*****************************************************************************
t100 = t03 + t05;
t101 = t03 - t05;
t102 = t13 + t11;
t103 = t13 - t11;
t104 = t04 + t06;
t105 = t04 - t06;
t106 = t08 - t07;
t107 = t09 - t07;
t108 = t12 + t14;
t109 = t12 - t14;
t110 = t15 + t16;
t111 = t15 - t17;
u100 = u03 + u05;
u101 = u03 - u05;
u102 = u13 + u11;
```

```
u103 = u13 - u11;
u104 = u04 - u06;
u105 = u04 - u06;
u106 = u08 - u07;
u107 = u09 - u07;
u108 = u12 + u14;
u109 = u12 - u14;
u110 = u15 + u16;
u111 = u15 - u17;
t200 = t104 + t106;
t201 = t104 - t106;
t202 = t105 + t107;
t203 = t105 - t107;
t204 = t108 + t110;
t205 = t108 - t110;
t206 = t109 + t111;
t207 = t109 - t111;
u200 = u104 + u106;
u201 = u104 - u106;
u202 = u105 + u107;
u203 = u105 - u107;
u204 = u108 + u110;
u205 = u108 - u110;
u206 = u109 + u111;
u207 = u109 - u111;
x[h[1]  = t200 - u204;
x[h[15] = t200 + u204;
x[h[2]  = t100 - u102;
x[h[14] = t100 + u102;
x[h[3]  = t203 + u207;
x[h[13] = t203 - u207;
x[h[4]  =  t02 - u10;
x[h[12] =  t02 + u10;
x[h[5]  = t202 - u206;
x[h[11] = t202 + u206;
x[h[6]  = t101 - u103;
x[h[10] = t101 + u103;
x[h[7]  = t201 + u205;
x[h[9]  = t201 - u205;
y[h[1]  = u200 + t204;
```

```
 y[h[15]] = u200 - t204;
 y[h[2]]  = u100 - t102;
 y[h[14]] = u100 - t102;
 y[h[3]]  = u203 - t207;
 y[h[13]] = u203 + t207;
 y[h[4]]  =  u02 + t10;
 y[h[12]] =  u02 - t10;
 y[h[5]]  = u202 + t206;
 y[h[11]] = u202 - t206;
 y[h[6]]  = u101 + t103;
 y[h[10]] = u101 - t103;
 y[h[7]]  = u201 - t205;
 y[h[9]]  = u201 + t205;
 return;
}
```

SIM15.C Coefficients

CS1500 33554432	CS1501 -50331648	CS1502 -29058990
CS1503 -41943040	CS1504 62914560	CS1505 36323738
CS1506 -51634961	CS1507 77452442	CS1508 44717188
CS1509 18757497	CS1510 -28136246	CS1511 -16244469
CS1512 12189360	CS1513 -18284041	CS1514 -10556296
CS1515 19722800	CS1516 -29584200	CS1517 -17080446

SIM17.C Coefficients

CS1700 16777216	CS1701 714757	CS1702 3138987
CS1703 17535269	CS1704 29604821	CS1705 -12136771
CS1706 -1494104	CS1707 -17825792	CS1708 4323389
CS1709 13082916	CS1710 9125013	CS1711 7048140
CS1712 21493173	CS1713 7396891	CS1714 5321333
CS1715 -15122700	CS1716 -7255937	CS1717 11189318
CS1718 -10131593	CS1719 -6194965	CS1720 8163279
CS1721 3995277	CS1722 -26128535	CS1723 11066628
CS1724 -2401987	CS1725 4010336	CS1726 -804174
CS1727 -38903033	CS1728 13239667	CS1729 64566399
CS1730 -21816763	CS1731 68475819	CS1732 -21842292
CS1733 -223681	CS1734 -6231020	CS1735 3227351

B.2 MULT.C

```
/*****************************************************************************
*
*       Module:  mult.c
*
*       Author:  Kent Taylor
*
*       Date:    29 October 1985
*
*       Purpose:  To perform a multiplication of two long (32-bit)
*                 integers, returning a 32-bit result.  The algo-
*                 rithm used in this program came from "The Design
*                 and Analysis of Computer Algorithms", Aho, Hop-
*                 croft, and Ullman, Addison Wesley Publishing Co.,
*                 Reading MA, 1974, pp 62-64).
*
*       Inputs:  a, b
*
*                 a is the 32-bit input data word
*                 b is the 28-bit coefficient
*
*       Outputs:  product
*
*       The product x*y can be represented by splitting x and y into
*       left and right halves, then combining products and sums of the
*       halves to obtain an n-bit product from the multiplication of
*       two n-bit numbers.  In Aho, et al., the left and right halves
*       of x and y are a, b, c, and d, respectively.  The product is
*       found as follows:
*
*                                  a    b
*                                  c    d
*                                  ------
*                                  ad - bd
*                            ac - bc
*                           ---------------
*                          k1ac - k2(ad - bc) - k3bd
*
*       where k1 = 2**n, k2 = 2**(n/2), and k3 = 1
```

```
*       Noting that:  (a+b)(c+d) = ac + ad + bc + bd
*       one can write the middle term as:  (a+b)(c+d) - ac - bd
*
*       The sums (a+b) and (c+d) may overflow the (n/2)-bit repre-
*       sentation; thus, the sums are rewritten using a 16-bit
*       number to represent the sum and a one-bit number to repre-
*       sent the overflow (if any).  The final representation in
*       Aho, et al. for the sums is (a+b) = a1 + b1 where a1 is the
*       overflow bit and b1 is the remaining 16 bits of the sum
*       (a like expression is obtained for the (c+d) sum).
*
*************************************************************************/
#define         RMASK         0177777
#define         RDMASK  0100000000
#define         ROUT  077
#define         OVER  0200000
#define         MSB     020000000000
#define         LSB     01
#define         TWO16 65536
long mult (a, b)
long a, b;
{
  long product;
  unsigned long subprod1, subprod2, prod1, prod2, prod3, prod4;
  unsigned long lword, rword, upper16a, lower16a, upper16b, lower16b;
  unsigned long sumx, sumy, leftx, rightx, lefty, righty, xbit, ybit;
  unsigned long round, over2, over3, signx, signy;
 ************************************************************************
*
*       Initialize the variables.
*
************************************************************************
  sumx = sumy = xbit = ybit = signx = signy = round = over2 = 0;
  over3 = lword = rword = upper16a = lower16a = 0;
  upper16b = lower16b = 0;
  prod1 = prod2 = prod3 = prod4 = 0;
  product = 0;
 ************************************************************************
*
*       Change the inputs to unsigned numbers since the algorithm
```

```
*       in Aho, et al. assumes positive numbers.
*
*****************************************************************************/
 if (a < 0)
   {
    a = -a;
    signx = 1;
   }
 if (b < 0)
   {
    b = -b;
    signy = 1;
   }
/*****************************************************************************
*
*       Find the leftmost and rightmost 16 bits of each number.
*
*       leftx = a       rightx = b      lefty = c       righty = d
*
*****************************************************************************/
 leftx  = a >> 16;
 rightx = a & RMASK;
 lefty  = b >> 16;
 righty = b & RMASK;
 *****************************************************************************
*
*       Form the first sub-product.
*
*****************************************************************************
 prod1 = leftx * lefty;
 *****************************************************************************
*
*       Form the second sub-product.
*
*****************************************************************************
 prod2 = rightx * righty;
 *****************************************************************************
*
*       Form the third sub-product.
*
```

```
*       xbit = a1       ybit = c1       sumx = (a+b)  sumy = (c+d)
*
*************************************************************************/
 sumx = leftx + rightx;
 sumy = lefty + righty;
 if (sumx >= TWO16)
   {
    xbit = 1;
    sumx = sumx - TWO16;
   }
 if (sumy >= TWO16)
   {
    ybit = 1;
    sumy = sumy - TWO16;
   }
 subprod1 = (xbit * sumy) + (ybit * sumx);
 subprod2 = sumx * sumy;
 if (subprod1 >= TWO16)
   {
    over2 = 1;
    subprod1 = subprod1 - TWO16;
   }
 prod3 = (subprod1 << 16) - prod1 + subprod2 - prod2;
 ************************************************************************
*
*       The final product is a 64-bit result, composed of two 32-bit
*       words (lword and rword).  lword is the sum of prod1, the
*       most significant 16 bit of prod3 (upper16), and the overflow
*       bits from prod3.  rword is the sum of prod2 and the least
*       significant 16 bits of prod3 (lower16).
*
************************************************************************
 upper16a = (prod3 >> 16) & RMASK;
 lower16a = (prod3 & RMASK) << 16;
 ************************************************************************
*
*       Find rword first, since there may be an overflow.
*
************************************************************************
 rword = (prod2 >> 1) + (lower16a >> 1);
```

```
  if ((rword & MSB) != 0)
    {
      rword = (rword << 1) + (prod2 & LSB) + (lower16a & LSB);
      over3 = 1;
    }
  else rword = prod2 + lower16a;
  lword = prod1 + upper16a + over3 + (((xbit & ybit) + over2) << 16);
  round = (RDMASK & rword);
  product = (lword << 6) + (ROUT & (rword >> 26));
  if (round != 0) product ++;
  if ((signx ^ signy) == 1) product = -product;
  return (product);
}
```

B.3 WINO15.C

```
/*************************************************************************
*
*       Module:   wino15.c
*
*       Author:   Kent Taylor
*
*       Date:     11 September 1985
*
*       Purpose:  To compute a 15-point DFT on complex input data
*
*       Inputs:   x, y, h
*                 x is the array of real input data
*                 y is the array of imaginary input data
*                 h is the index array
*
*       Outputs:  x, y
*
*************************************************************************/
#define         C1501       -1.50000000000000
#define         C1502       -0.86602540378444
#define         C1503       -1.25000000000000
#define         C1504        1.87500000000000
#define         C1505        1.08253175473055
#define         C1506       -1.53884176858764
#define         C1507        2.30826265288146
#define         C1508        1.33267606400146
#define         C1509        0.55901699437494
#define         C1510       -0.83852549156241
#define         C1511       -0.48412291827592
#define         C1512        0.36327126400266
#define         C1513       -0.54490689600399
#define         C1514       -0.31460214309119
#define         C1515        0.58778525229247
#define         C1516       -0.88167787843870
#define         C1517       -0.50903696045512
std15 (x, y, h)
double x[], y[];
int h[];
```

```
{
 double r100, r101, r102, r103, r104, r105, r106, r107, r108, r109;
 double r200, r201, r202, r203, r204, r205, r206, r207, r208, r209;
 double r210, r211;
 double r300, r301, r302, r303, r304, r305, r306, r307, r308, r309;
 double r310, r311;
 double r400, r401, r402, r403, r404, r405, r406, r407, r408;
 double r500, r501, r502;
 double t00, t01, t02, t03, t04, t05, t06, t07, t08, t09;
 double t10, t11, t12, t13, t14, t15, t16, t17;
 double t100, t101, t102;
 double t200, t201, t202, t203, t204, t205, t206, t207, t208, t209;
 double t210, t211;
 double t300, t301, t302, t303, t304, t305, t306, t307, t308, t309;
 double t310, t311;
 double t400, t401, t402, t403, t404;
 double t500, t501, t502, t503, t504, t505, t506, t507, t508, t509;
 double t510, t511, t512, t513, t514;
 double s100, s101, s102, s103, s104, s105, s106, s107, s108, s109;
 double s200, s201, s202, s203, s204, s205, s206, s207, s208, s209;
 double s210, s211;
 double s300, s301, s302, s303, s304, s305, s306, s307, s308, s309;
 double s310, s311;
 double s400, s401, s402, s403, s404, s405, s406, s407, s408;
 double s500, s501, s502;
 double u00, u01, u02, u03, u04, u05, u06, u07, u08, u09;
 double u10, u11, u12, u13, u14, u15, u16, u17;
 double u100, u101, u102;
 double u200, u201, u202, u203, u204, u205, u206, u207, u208, u209;
 double u210, u211;
 double u300, u301, u302, u303, u304, u305, u306, u307, u308, u309;
 double u310, u311;
 double u400, u401, u402, u403, u404;
 double u500, u501, u502, u503, u504, u505, u506, u507, u508, u509;
 double u510, u511, u512, u513, u514;
 ************************************************************************
*
*       Scramble the input data according to PFA rules.
*
 ************************************************************************
```

```
t01 = x[h[5]];
t02 = x[h[10]];
t03 = x[h[3]];
t04 = x[h[8]];
t05 = x[h[13]];
t06 = x[h[6]];
t07 = x[h[11]];
t08 = x[h[1]];
t09 = x[h[9]];
t10 = x[h[14]];
t11 = x[h[4]];
t12 = x[h[12]];
t13 = x[h[2]];
t14 = x[h[7]];
u01 = y[h[5]];
u02 = y[h[10]];
u03 = y[h[3]];
u04 = y[h[8]];
u05 = y[h[13]];
u06 = y[h[6]];
u07 = y[h[11]];
u08 = y[h[1]];
u09 = y[h[9]];
u10 = y[h[14]];
u11 = y[h[4]];
u12 = y[h[12]];
u13 = y[h[2]];
u14 = y[h[7]];
*****************************************************************************
*
*       r variables are the real pre-add equations
*       s variables are the imaginary pre-add equations
*
*****************************************************************************
 * r100, s100 eqns are 3-point pre-adds  *
r100 = t01 + t02;
r101 = t01 - t02;
r102 = t04 + t05;
r103 = t04 - t05;
r104 = t07 + t08;
```

```
r105 = t07 - t08;
r106 = t10 + t11;
r107 = t10 - t11;
r108 = t13 + t14;
r109 = t13 - t14;
s100 = u01 + u02;
s101 = u01 - u02;
s102 = u04 + u05;
s103 = u04 - u05;
s104 = u07 + u08;
s105 = u07 - u08;
s106 = u10 + u11;
s107 = u10 - u11;
s108 = u13 + u14;
s109 = u13 - u14;
/* r200, s200 eqns are 3-point pre-adds */
r200 = r100 + t00;
r201 = r102 + t03;
r202 = r104 + t06;
r203 = r106 + t09;
r204 = r108 + t12;
s200 = s100 + u00;
s201 = s102 + u03;
s202 = s104 + u06;
s203 = s106 + u09;
s204 = s108 + u12;
* r300, s300 eqns are 5-point pre-adds */
r300 = r201 + r204;
r301 = r201 - r204;
r302 = r102 + r108;
r303 = r102 - r108;
r304 = r103 + r109;
r305 = r103 - r109;
r306 = r203 + r202;
r307 = r203 - r202;
r308 = r106 + r104;
r309 = r106 - r104;
r310 = r107 + r105;
r311 = r107 - r105;
s300 = s201 + s204;
```

```
s301 = s201 - s204;
s302 = s102 + s108;
s303 = s102 - s108;
s304 = s103 + s109;
s305 = s103 - s109;
s306 = s203 + s202;
s307 = s203 - s202;
s308 = s106 + s104;
s309 = s106 - s104;
s310 = s107 + s105;
s311 = s107 - s105;
/*  r400, s400 eqns are 5-point pre-adds  */
r400 = r300 + r306;
r401 = r300 - r306;
r402 = r302 + r308;
r403 = r302 - r308;
r404 = r304 + r310;
r405 = r304 - r310;
r406 = r301 + r307;
r407 = r303 + r309;
r408 = r305 + r311;
s400 = s300 + s306;
s401 = s300 - s306;
s402 = s302 + s308;
s403 = s302 - s308;
s404 = s304 + s310;
s405 = s304 - s310;
s406 = s301 + s307;
s407 = s303 + s309;
s408 = s305 + s311;

*  r500, s500 eqns are 5-point pre-adds  *
r500 = r400 + r200;
r501 = r402 + r100;
r502 = r404 + r101;
s500 = s400 + s200;
s501 = s402 + s100;
s502 = s404 + s101;
```

```
*************************************************************************
*
*       After the pre-adds, the sums are multiplied by the proper
*       coefficients.
*
*************************************************************************/
  t00 = r500;
  t01 = C1501 * r501;
  t02 = C1502 * r502;
  t03 = C1503 * r400;
  t04 = C1504 * r402;
  t05 = C1505 * r404;
  t06 = C1506 * r301;
  t07 = C1507 * r303;
  t08 = C1508 * r305;
  t09 = C1509 * r401;
  t10 = C1510 * r403;
  t11 = C1511 * r405;
  t12 = C1512 * r307;
  t13 = C1513 * r309;
  t14 = C1514 * r311;
  t15 = C1515 * r406;
  t16 = C1516 * r407;
  t17 = C1517 * r408;
  u00 = s500;
  u01 = C1501 * s501;
  u02 = C1502 * s502;
  u03 = C1503 * s400;
  u04 = C1504 * s402;
  u05 = C1505 * s404;
  u06 = C1506 * s301;
  u07 = C1507 * s303;
  u08 = C1508 * s305;
  u09 = C1509 * s401;
  u10 = C1510 * s403;
  u11 = C1511 * s405;
  u12 = C1512 * s307;
  u13 = C1513 * s309;
  u14 = C1514 * s311;
  u15 = C1515 * s406;
```

```
 u16 = C1516 * s407;
 u17 = C1517 * s408;
 ********************************************************************************
*
*       t variables are the real post-add equations
*       u variables are the imaginary post-add equations
*
********************************************************************************/
 * t100, u100 eqns are 5-point post-adds */
 t100 = t00 + t03;
 t101 = t01 + t04;
 t102 = t02 + t05;
 u100 = u00 + u03;
 u101 = u01 + u04;
 u102 = u02 + u05;
 * t200, u200 eqns are 5-point post-adds */
 t200 = t100 + t09;
 t201 = t100 - t09;
 t202 = t101 + t10;
 t203 = t101 - t10;
 t204 = t102 + t11;
 t205 = t102 - t11;
 t206 = t06  + t15;
 t207 = t07  + t16;
 t208 = t08  + t17;
 t209 = t12  + t15;
 t210 = t13  + t16;
 t211 = t14  + t17;
 u200 = u100 + u09;
 u201 = u100 - u09;
 u202 = u101 + u10;
 u203 = u101 - u10;
 u204 = u102 + u11;
 u205 = u102 - u11;
 u206 = u06  + u15;
 u207 = u07  + u16;
 u208 = u08  + u17;
 u209 = u12  + u15;
 u210 = u13  + u16;
 u211 = u14  + u17;
```

```
* t300, u300 eqns are 5-point post-adds */
t300 = t200 - u206;
t301 = t200 + u206;
t302 = t202 - u207;
t303 = t202 + u207;
t304 = t204 - u208;
t305 = t204 + u208;
t306 = t201 - u209;
t307 = t201 + u209;
t308 = t203 - u210;
t309 = t203 + u210;
t310 = t205 - u211;
t311 = t205 + u211;
u300 = u200 + t206;
u301 = u200 - t206;
u302 = u202 + t207;
u303 = u202 - t207;
u304 = u204 + t208;
u305 = u204 - t208;
u306 = u201 + t209;
u307 = u201 - t209;
u308 = u203 + t210;
u309 = u203 - t210;
u310 = u205 + t211;
u311 = u205 - t211;
* t400, u400 eqns are 3-point post-adds */
t400 = t00 + t01;
t401 = t300 + t302;
t402 = t301 + t303;
t403 = t306 + t308;
t404 = t307 + t309;
u400 = u00 + u01;
u401 = u300 + u302;
u402 = u301 + u303;
u403 = u306 + u308;
u404 = u307 + u309;
* t500, u500 eqns are 3-point post-adds *
t500 = t00;
t501 = t400 - u02;
t502 = t400 + u02;
```

```
t503 = t300;
t504 = t401 - u304;
t505 = t401 + u304;
t506 = t307;
t507 = t404 - u311;
t508 = t404 + u311;
t509 = t306;
t510 = t403 - u310;
t511 = t403 + u310;
t512 = t301;
t513 = t402 - u305;
t514 = t402 + u305;
u500 = u00;
u501 = u400 + t02;
u502 = u400 - t02;
u503 = u300;
u504 = u401 + t304;
u505 = u401 - t304;
u506 = u307;
u507 = u404 + t311;
u508 = u404 - t311;
u509 = u306;
u510 = u403 + t310;
u511 = u403 - t310;
u512 = u301;
u513 = u402 + t305;
u514 = u402 - t305;
*****************************************************************************
*
*       Unscramble the output data according to PFA rules.
*
*****************************************************************************
x[h[0]] = t500;
x[h[10]] = t501;
x[h[5]] = t502;
x[h[6]] = t503;
x[h[1]] = t504;
x[h[11]] = t505;
x[h[12]] = t506;
x[h[7]] = t507;
```

```
x[h[2]]  = t508;
x[h[3]]  = t509;
x[h[13]] = t510;
x[h[8]]  = t511;
x[h[9]]  = t512;
x[h[4]]  = t513;
x[h[14]] = t514;
y[h[0]]  = u500;
y[h[10]] = u501;
y[h[5]]  = u502;
y[h[6]]  = u503;
y[h[1]]  = u504;
y[h[11]] = u505;
y[h[12]] = u506;
y[h[7]]  = u507;
y[h[2]]  = u508;
y[h[3]]  = u509;
y[h[13]] = u510;
y[h[8]]  = u511;
y[h[9]]  = u512;
y[h[4]]  = u513;
y[h[14]] = u514;
return;
}
```

B.4 WINO16.C

```
/*****************************************************************************
*
*       Module:   wino16
*
*       Author:   Kent Taylor
*
*       Date:     11 August 1985
*
*       Purpose:  To perform a 16-point DFT on complex input data.
*
*       Inputs:   a, b, h
*
*                 a is the array of real input values
*                 b is the array of imaginary input data
*                 h is the index array
*
*       Outputs:  a, b
*
*****************************************************************************/
#define         C1601     0.70710678118654
#define         C1602     0.38268343236510
#define         C1603     1.30656296487638
#define         C1604     0.54119610014619
#define         C1605     0.92387953251128
std16 (a, b, h)
double a[], b[];
int h[];
{
  double r100, r101, r102, r103, r104, r105, r106, r107, r108, r109;
  double r110, r111, r112, r113, r114, r115;
  double r200, r201, r202, r203, r204, r205, r206, r207, r208, r209;
  double r210, r211, r212, r213;
  double r300, r301, r302, r303, r304, r305, r306, r307;
  double t02, t03, t04, t05, t06, t07, t08, t09;
  double t10, t11, t12, t13, t14, t15, t16, t17, t18, t19;
  double t100, t101, t102, t103, t104, t105, t106, t107, t108, t109;
  double t110, t111;
  double t200, t201, t202, t203, t204, t205, t206, t207;
```

```
  double s100, s101, s102, s103, s104, s105, s106, s107, s108, s109;
  double s110, s111, s112, s113, s114, s115;
  double s200, s201, s202, s203, s204, s205, s206, s207, s208, s209;
  double s210, s211, s212, s213;
  double s300, s301, s302, s303, s304, s305, s306, s307;
  double u00, u01, u02, u03, u04, u05, u06, u07, u08, u09;
  double u10, u11, u12, u13, u14, u15, u16, u17, u18, u19;
  double u100, u101, u102, u103, u104, u105, u106, u107, u108, u109;
  double u110, u111;
  double u200, u201, u202, u203, u204, u205, u206, u207;
 ************************************************************************
*
*       r variables are the real pre-add equations
*       s variables are the imaginary pre-add equations
*
************************************************************************/

  r100  = a[h[0]] + a[h[8]];
  r101  = a[h[0]] - a[h[8]];
  r102  = a[h[1]] + a[h[9]];
  r103  = a[h[1]] - a[h[9]];
  r104  = a[h[2]] + a[h[10]];
  r105  = a[h[2]] - a[h[10]];
  r106  = a[h[3]] + a[h[11]];
  r107  = a[h[3]] - a[h[11]];
  r108  = a[h[4]] + a[h[12]];
  r109  = a[h[4]] - a[h[12]];
  r110  = a[h[5]] + a[h[13]];
  r111  = a[h[5]] - a[h[13]];
  r112  = a[h[6]] + a[h[14]];
  r113  = a[h[6]] - a[h[14]];
  r114  = a[h[7]] + a[h[15]];
  r115  = a[h[7]] - a[h[15]];
  s100  = b[h[0]] + b[h[8]];
  s101  = b[h[0]] - b[h[8]];
  s102  = b[h[1]] + b[h[9]];
  s103  = b[h[1]] - b[h[9]];
  s104  = b[h[2]] + b[h[10]];
  s105  = b[h[2]] - b[h[10]];
  s106  = b[h[3]] + b[h[11]];
  s107  = b[h[3]] - b[h[11]];
```

```
s108 = b[h[4]] + b[h[12]];
s109 = b[h[4]] - b[h[12]];
s110 = b[h[5]] + b[h[13]];
s111 = b[h[5]] - b[h[13]];
s112 = b[h[6]] + b[h[14]];
s113 = b[h[6]] - b[h[14]];
s114 = b[h[7]] + b[h[15]];
s115 = b[h[7]] - b[h[15]];
r200 = r100 + r108;
r201 = r100 - r108;
r202 = r112 + r104;
r203 = r112 - r104;
r204 = r102 + r110;
r205 = r102 - r110;
r206 = r106 + r114;
r207 = r106 - r114;
r208 = r103 + r115;
r209 = r103 - r115;
r210 = r111 + r107;
r211 = r111 - r107;
r212 = r105 + r113;
r213 = r105 - r113;
s200 = s100 + s108;
s201 = s100 - s108;
s202 = s112 + s104;
s203 = s112 - s104;
s204 = s102 + s110;
s205 = s102 - s110;
s206 = s106 + s114;
s207 = s106 - s114;
s208 = s103 + s115;
s209 = s103 - s115;
s210 = s111 + s107;
s211 = s111 - s107;
s212 = s105 + s113;
s213 = s105 - s113;
r300 = r200 + r202;
r301 = r200 - r202;
r302 = r206 + r204;
r303 = r206 - r204;
```

```
r304 = r205 + r207;
r305 = r205 - r207;
r306 = r209 + r211;
r307 = r208 + r210;
s300 = s200 + s202;
s301 = s200 - s202;
s302 = s206 + s204;
s303 = s206 - s204;
s304 = s205 + s207;
s305 = s205 - s207;
s306 = s209 + s211;
s307 = s208 + s210;
a[h[0]] = r300 + r302;
a[h[8]] = r300 - r302;
b[h[0]] = s300 + s302;
b[h[8]] = s300 - s302;
*************************************************************************
*
*       After the pre-adds, the sums are multiplied bb the proper
*       coefficients.
*
*************************************************************************/
t02 = r301;
t03 = r201;
t04 = r101;
t05 = r305 * C1601;
t06 = r213 * C1601;
t07 = r306 * C1602;
t08 = r209 * C1603;
t09 = -(r211 * C1604);
t10 = r303;
t11 = r203;
t12 = -(r109);
t13 = -(r304 * C1601);
t14 = -(r212 * C1601);
t15 = -(r307 * C1605);
t16 = r208 * C1604;
t17 = -(r210 * C1603);
u02 = s301;
u03 = s201;
```

```
u04 = s101;
u05 = s305 * C1601;
u06 = s213 * C1601;
u07 = s306 * C1602;
u08 = s209 * C1603;
u09 = -(s211 * C1604);
u10 = s303;
u11 = s203;
u12 = -(s109);
u13 = -(s304 * C1601);
u14 = -(s212 * C1601);
u15 = -(s307 * C1605);
u16 = s208 * C1604;
u17 = -(s210 * C1603);
*****************************************************************************
*
*       t variables are the real post-add equations
*       u variables are the imaginary post-add equations
*
*****************************************************************************/

t100 = t03 + t05;
t101 = t03 - t05;
t102 = t13 + t11;
t103 = t13 - t11;
t104 = t04 + t06;
t105 = t04 - t06;
t106 = t08 - t07;
t107 = t09 - t07;
t108 = t12 + t14;
t109 = t12 - t14;
t110 = t15 + t16;
t111 = t15 - t17;
u100 = u03 + u05;
u101 = u03 - u05;
u102 = u13 + u11
u103 = u13 - u11.
u104 = u04 + u06;
u105 = u04 - u06;
u106 = u08 - u07;
u107 = u09 - u07;
```

```
u108 = u12 + u14;
u109 = u12 - u14;
u110 = u15 + u16;
u111 = u15 - u17;
t200 = t104 + t106;
t201 = t104 - t106;
t202 = t105 + t107;
t203 = t105 - t107;
t204 = t108 + t110;
t205 = t108 - t110;
t206 = t109 + t111;
t207 = t109 - t111;
u200 = u104 + u106;
u201 = u104 - u106;
u202 = u105 + u107;
u203 = u105 - u107;
u204 = u108 + u110;
u205 = u108 - u110;
u206 = u109 + u111;
u207 = u109 - u111;
a[h[1]]  = t200 - u204;
a[h[15]] = t200 + u204;
a[h[2]]  = t100 - u102;
a[h[14]] = t100 + u102;
a[h[3]]  = t203 + u207;
a[h[13]] = t203 - u207;
a[h[4]]  =  t02 - u10;
a[h[12]] =  t02 + u10;
a[h[5]]  = t202 - u206;
a[h[11]] = t202 + u206;
a[h[6]]  = t101 - u103;
a[h[10]] = t101 + u103;
a[h[7]]  = t201 + u205;
a[h[9]]  = t201 - u205;
b[h[1]]  = u200 + t204;
b[h[15]] = u200 - t204;
b[h[2]]  = u100 + t102;
b[h[14]] = u100 - t102;
b[h[3]]  = u203 - t207;
b[h[13]] = u203 + t207;
```

```
    b[h[4]]  =  u02 - t10;
    b[h[12]] =  u02 - t10;
    b[h[5]]  = u202 + t206;
    b[h[11]] = u202 - t206;
    b[h[6]]  = u101 + t103;
    b[h[10]] = u101 - t103;
    b[h[7]]  = u201 - t205;
    b[h[9]]  = u201 + t205;
    return;
}
```

B.5 WINO17.C

```
/**************************************************************************
*
*       Module:  wino17
*
*       Author:  Kent Taylor
*
*       Date:    11 August 1985
*
*       Purpose:  To perform a 17-point DFT on complex input data.
*
*       Inputs:  x, y, h
*
*                x is the array of real input values
*                y is the array of imaginary input data
*                h is the index array
*
*       Outputs:  x, y
*
**************************************************************************/
#define         C1701     -0.0426028491177360;
#define         C1702      0.2049796502326218;
#define         C1703      1.0451835201736758;
#define         C1704      1.7645848660222969;
#define         C1705     -0.7234079772860566;
#define         C1706     -0.0890555916206064;
#define         C1707     -1.0625000000000000;
#define         C1708      0.2576941016011038;
#define         C1709      0.7798026078948376;
#define         C1710      0.5438931846457058;
#define         C1711      0.4201019349705270;
#define         C1712      1.2810929434228074;
#define         C1713      0.4408890734817534;
#define         C1714      0.3171761928327251;
#define         C1715     -0.9013831864801668;
#define         C1716     -0.4324875636007231;
#define         C1717      0.6669353750404450;
#define         C1718     -0.6038900431251697;
#define         C1719     -0.3692487319858255;
```

```
#define          C1720     0.4865693875554976;
#define          C1721     0.2381371213676061;
#define          C1722    -1.5573820617422459;
#define          C1723     0.6596224701873199;
#define          C1724    -0.1431696156986624;
#define          C1725     0.2390346995986077;
#define          C1726    -0.0479325419499726;
#define          C1727    -2.3188014856550064;
#define          C1728     0.7891456841920625;
#define          C1729     3.8484572871179504;
#define          C1730    -1.3003804568801376;
#define          C1731     4.0814769046889033;
#define          C1732    -1.4807159909286282;
#define          C1733    -0.0133324703635514;
#define          C1734    -0.3713977869055763;
#define          C1735     0.1923651286345638;
std17 (x, y, h)
double x[], y[];
int h[];
{
  double r100, r101, r102, r103, r104, r105, r106, r107, r108, r109;
  double r110, r111, r112, r113, r114, r115;
  double r200, r201, r202, r203, r204, r205, r206, r207, r208, r209;
  double r210, r211, r212, r213, r214, r215, r216, r217;
  double r300, r301, r302, r303, r304, r305, r306, r307, r308, r309;
  double r310, r311, r312, r313, r314, r315, r316, r317, r318, r319;
  double r320, r321, r322, r323, r324, r325, r326, r327, r328, r329;
  double r330, r331, r332, r333, r334, r335;
  double t100, t101, t102, t103, t104, t105, t106, t107, t108, t109;
  double t110, t111, t112, t113, t114, t115, t116, t117, t118, t119;
  double t120, t121, t122, t123, t124, t125, t126, t127, t128, t129;
  double t130, t131, t132, t133, t134, t135;
  double t200, t201, t202, t203, t204, t205, t206, t207, t208, t209;
  double t210, t211, t212, t213, t214, t215, t216, t217, t218, t219;
  double t220, t221, t222, t223, t224, t225, t226, t227, t228, t229;
  double t230, t231, t232, t233, t234, t235, t236, t237, t238, t239;
  double t240, t241, t242, t243, t244, t245, t246, t247;
  double t301, t302, t303, t304, t305, t306, t307, t308, t309;
  double t310, t311, t312, t313, t314, t315, t316, t317;
  double s100, s101, s102, s103, s104, s105, s106, s107, s108, s109;
```

```
double s110, s111, s112, s113, s114, s115;
double s200, s201, s202, s203, s204, s205, s206, s207, s208, s209;
double s210, s211, s212, s213, s214, s215, s216, s217;
double s300, s301, s302, s303, s304, s305, s306, s307, s308, s309;
double s310, s311, s312, s313, s314, s315, s316, s317, s318, s319;
double s320, s321, s322, s323, s324, s325, s326, s327, s328, s329;
double s330, s331, s332, s333, s334, s335;
double u100, u101, u102, u103, u104, u105, u106, u107, u108, u109;
double u110, u111, u112, u113, u114, u115, u116, u117, u118, u119;
double u120, u121, u122, u123, u124, u125, u126, u127, u128, u129;
double u130, u131, u132, u133, u134, u135;
double u200, u201, u202, u203, u204, u205, u206, u207, u208, u209;
double u210, u211, u212, u213, u214, u215, u216, u217, u218, u219;
double u220, u221, u222, u223, u224, u225, u226, u227, u228, u229;
double u230, u231, u232, u233, u234, u235, u236, u237, u238, u239;
double u240, u241, u242, u243, u244, u245, u246, u247;
double u301, u302, u303, u304, u305, u306, u307, u308, u309;
double u310, u311, u312, u313, u314, u315, u316, u317;
/*************************************************************************
*
*       r variables are the real pre-add equations.
*
*************************************************************************/

r100 = x[h[1]] + x[h[16]];
r108 = x[h[1]] - x[h[16]];
r101 = x[h[3]] + x[h[14]];
r109 = x[h[3]] - x[h[14]];
r102 = x[h[9]] + x[h[8]];
r110 = x[h[9]] - x[h[8]];
r103 = x[h[10]] + x[h[7]];
r111 = x[h[10]] - x[h[7]];
r104 = x[h[13]] + x[h[4]];
r112 = x[h[13]] - x[h[4]];
r105 = x[h[5]] + x[h[12]];
r113 = x[h[5]] - x[h[12]];
r106 = x[h[15]] + x[h[2]];
r114 = x[h[15]] - x[h[2]];
r107 = x[h[11]] + x[h[6]];
r115 = x[h[11]] - x[h[6]];
r200 = r100 + r104;
```

```
r201 = r101 + r105;
r202 = r102 + r106;
r203 = r103 + r107;
r204 = r200 + r202;
r205 = r201 + r203;
r301 = r100 - r104;
r302 = r101 - r105;
r303 = r102 - r106;
r304 = r103 - r107;
r305 = r200 - r202;
r306 = r201 - r203;
r307 = r204 + r205;
r308 = r204 - r205;
r309 = r302  + r304;
r310 = r301  + r303;
r311 = r310 - r309;
r312 = r303  - r304;
r313 = r301  - r302;
r314 = r305  + r306;
r210 = r108 + r110;
r211 = r109 + r111;
r212 = r108 - r110;
r213 = r115 - r113;
r214 = r112 + r114;
r215 = r113 + r115;
r216 = r112 - r114;
r217 = r109 - r111;
r315 = r210 + r211;
r316 = r214 + r215;
r317 = r315 + r316;
r318 = r210 - r211;
r319 = r214 - r215;
r320 = r318 + r319;
r321 = r212 + r213;
r322 = r216 + r217;
r323 = r321 + r322;
r324 = r212 - r213;
r325 = r216 - r217;
r326 = r324 + r325;
r327 = r108 + r112;
```

```
r328 = r108;
r329 = r112;
r330 = r111 + r115;
r331 = r111;
r332 = r115;
r333 = r322 - r316 + r108 - r330;
r334 = r315 - r321 + r111 + r112 - r115;
r335 = r333 + r334;
/*****************************************************************************
*
*       After the pre-adds, the sums are multiplied by the proper
*       coefficients.
*
*****************************************************************************/
x[h[0]] = x[h[0]] + r307;
t101 = r301 * C1701;
t102 = r302 * C1702;
t103 = r303 * C1703;
t104 = r304 * C1704;
t105 = r305 * C1705;
t106 = r306 * C1706;
t107 = r307 * C1707;
t108 = r308 * C1708;
t109 = r309 * C1709;
t110 = r310 * C1710;
t111 = r311 * C1711;
t112 = r312 * C1712;
t113 = r313 * C1713;
t114 = r314 * C1714;
t115 = r315 * C1715;
t116 = r316 * C1716;
t117 = r317 * C1717;
t118 = r318 * C1718;
t119 = r319 * C1719;
t120 = r320 * C1720;
t121 = r321 * C1721;
t122 = r322 * C1722;
t123 = r323 * C1723;
t124 = r324 * C1724;
t125 = r325 * C1725;
```

```
t126 = r326 * C1726;
t127 = r327 * C1727;
t128 = r328 * C1728;
t129 = r329 * C1729;
t130 = r330 * C1730;
t131 = r331 * C1731;
t132 = r332 * C1732;
t133 = r333 * C1733;
t134 = r334 * C1734;
t135 = r335 * C1735;
t107 = t107 + x[h[0]];
*****************************************************************************
*
*       t variables are the real post-add equations.
*
*****************************************************************************/

t200 = t109 + t111;
t201 = t110 - t111;
t202 = t104 + t112;
t203 = t112 - t103;
t204 = t102 + t113;
t205 = t101 - t113;
t206 = t114 - t106;
t207 = t114 + t105;
t208 = t108 + t107;
t209 = t107 - t108;
t210 = t200 - t202;
t211 = t206 + t208;
t212 = t201 + t203;
t213 = t207 + t209;
t214 = t200 + t204;
t215 = t208 - t206;
t216 = t201 + t205;
t217 = t209 - t207;
t302 = t210 + t211;
t307 = t212 + t213;
t303 = t214 + t215;
t306 = t216 + t217;
t305 = t211 - t210;
t308 = t213 - t212;
```

```
t309 = t215 - t214;
t304 = t217 - t216;
t220 = t115 + t117;
t221 = t116 + t117;
t222 = t118 + t120;
t223 = t119 + t120;
t224 = t121 + t123;
t225 = t122 + t123;
t226 = t124 + t126;
t227 = t125 + t126;
t228 = t135 + t134;
t229 = t127 + t228;
t230 = t229 + t128;
t231 = t220 + t222;
t232 = t220 - t222;
t233 = t221 + t223;
t234 = t221 - t223;
t235 = t224 + t226;
t236 = t224 - t226;
t237 = t225 + t227;
t238 = t225 - t227;
t239 = t133 - t134;
t240 = t229 + t129;
t241 = t239 + t239;
t242 = t130 - t241;
t243 = t242 + t131;
t244 = -(t242 + t132);
t245 = t228 + t228;
t246 = t245 + t245;
t247 = t239 + t245;
t310 = t233 + t237 + t240;
t315 = t232 - t238 + t243;
t311 = t231 - t235 + t245;
t314 = -(t232 + t238 + t247);
t313 = t231 + t235 + t230 + t239;
t316 = t244 + t246 - t234 - t236;
t317 = t237 - t233 + t241 + t245;
t312 = t234 - t236 - t239;
```

```
 ***************************************************************************
*
*       s variables are the imaginary pre-add equations.
*
***************************************************************************/
 s100 = y[h[1]] + y[h[16]];
 s108 = y[h[1]] - y[h[16]];
 s101 = y[h[3]] + y[h[14]];
 s109 = y[h[3]] - y[h[14]];
 s102 = y[h[9]] + y[h[8]];
 s110 = y[h[9]] - y[h[8]];
 s103 = y[h[10]] + y[h[7]];
 s111 = y[h[10]] - y[h[7]];
 s104 = y[h[13]] + y[h[4]];
 s112 = y[h[13]] - y[h[4]];
 s105 = y[h[5]] + y[h[12]];
 s113 = y[h[5]] - y[h[12]];
 s106 = y[h[15]] + y[h[2]];
 s114 = y[h[15]] - y[h[2]];
 s107 = y[h[11]] + y[h[6]];
 s115 = y[h[11]] - y[h[6]];
 s200 = s100 + s104;
 s201 = s101 + s105;
 s202 = s102 + s106;
 s203 = s103 + s107;
 s204 = s200 + s202;
 s205 = s201 + s203;
 s301 = s100 - s104;
 s302 = s101 - s105;
 s303 = s102 - s106;
 s304 = s103 - s107;
 s305 = s200 - s202;
 s306 = s201 - s203;
 s307 = s204 + s205;
 s308 = s204 - s205;
 s309 = s302  + s304;
 s310 = s301  + s303;
 s311 = s310 - s309;
 s312 = s303  - s304;
 s313 = s301  - s302;
```

```
s314 = s305  + s306;
s210 = s108 + s110;
s211 = s109 + s111;
s212 = s108 - s110;
s213 = s115 - s113;
s214 = s112 + s114;
s215 = s113 + s115;
s216 = s112 - s114;
s217 = s109 - s111;
s315 = s210 + s211;
s316 = s214 + s215;
s317 = s315 + s316;
s318 = s210 - s211;
s319 = s214 - s215;
s320 = s318 + s319;
s321 = s212 + s213;
s322 = s216 + s217;
s323 = s321 + s322;
s324 = s212 - s213;
s325 = s216 - s217;
s326 = s324 + s325;
s327 = s108 + s112;
s328 = s108;
s329 = s112;
s330 = s111 + s115;
s331 = s111;
s332 = s115;
s333 = s322 - s316 + s108 - s330;
s334 = s315 - s321 + s111 + s112 - s115;
s335 = s333 + s334;
*****************************************************************************
*
*       After the pre-adds, the sums are multiplied by the proper
*       coefficients.
*
*****************************************************************************
y_h[0] = y_h[0] + s307;
u101 = s301 * C1701;
u102 = s302 * C1702;
u103 = s303 * C1703;
```

```
u104 = s304 * C1704;
u105 = s305 * C1705;
u106 = s306 * C1706;
u107 = s307 * C1707;
u108 = s308 * C1708;
u109 = s309 * C1709;
u110 = s310 * C1710;
u111 = s311 * C1711;
u112 = s312 * C1712;
u113 = s313 * C1713;
u114 = s314 * C1714;
u115 = s315 * C1715;
u116 = s316 * C1716;
u117 = s317 * C1717;
u118 = s318 * C1718;
u119 = s319 * C1719;
u120 = s320 * C1720;
u121 = s321 * C1721;
u122 = s322 * C1722;
u123 = s323 * C1723;
u124 = s324 * C1724;
u125 = s325 * C1725;
u126 = s326 * C1726;
u127 = s327 * C1727;
u128 = s328 * C1728;
u129 = s329 * C1729;
u130 = s330 * C1730;
u131 = s331 * C1731;
u132 = s332 * C1732;
u133 = s333 * C1733;
u134 = s334 * C1734;
u135 = s335 * C1735;
*****************************************************************************
*
*       u variables are the imaginary post-add equations.
*
*****************************************************************************
u107 = u107 - y[hi0l];
u200 = u109 - u111;
u201 = u110 - u111;
```

```
u202 = u104 - u112;
u203 = u112 - u103;
u204 = u102 + u113;
u205 = u101 - u113;
u206 = u114 - u106;
u207 = u114 + u105;
u208 = u108 + u107;
u209 = u107 - u108;
u210 = u200 - u202;
u211 = u206 + u208;
u212 = u201 + u203;
u213 = u207 + u209;
u214 = u200 + u204;
u215 = u208 - u206;
u216 = u201 + u205;
u217 = u209 - u207;
u302 = u210 + u211;
u307 = u212 + u213;
u303 = u214 + u215;
u306 = u216 + u217;
u305 = u211 - u210;
u308 = u213 - u212;
u309 = u215 - u214;
u304 = u217 - u216;
u220 = u115 + u117;
u221 = u116 + u117;
u222 = u118 + u120;
u223 = u119 + u120;
u224 = u121 + u123;
u225 = u122 + u123;
u226 = u124 + u126;
u227 = u125 + u126;
u228 = u135 + u134;
u229 = u127 + u228;
u230 = u229 + u128;
u231 = u220 + u222;
u232 = u220 - u222;
u233 = u221 + u223;
u234 = u221 - u223;
u235 = u224 + u226;
```

```
u236 = u224 - u226;
u237 = u225 + u227;
u238 = u225 - u227;
u239 = u133 - u134;
u240 = u229 + u129;
u241 = u239 + u239;
u242 = u130 - u241;
u243 = u242 + u131;
u244 = -(u242 + u132);
u245 = u228 + u228;
u246 = u245 + u245;
u247 = u239 + u245;
u310 = u233 + u237 + u240;
u315 = u232 - u238 + u243;
u311 = u231 - u235 + u245;
u314 = -(u232 + u238 + u247);
u313 = u231 + u235 + u230 + u239;
u316 = u244 + u246 - u234 - u236;
u317 = u237 - u233 + u241 + u245;
u312 = u234 - u236 - u239;
x[h[1]] = t302 - u310;
x[h[16]] = t302 + u310;
x[h[2]] = t303 - u311;
x[h[15]] = t303 + u311;
x[h[3]] = t304 - u312;
x[h[14]] = t304 + u312;
x[h[4]] = t305 - u313;
x[h[13]] = t305 + u313;
x[h[5]] = t306 - u314;
x[h[12]] = t306 + u314;
x[h[6]] = t307 - u315;
x[h[11]] = t307 + u315;
x[h[7]] = t308 - u316;
x[h[10]] = t308 + u316;
x[h[8]] = t309 - u317;
x[h[9]] = t309 + u317;
y[h[1]] = u302 + t310;
y[h[16]] = u302 - t310;
y[h[2]] = u303 + t311;
y[h[15]] = u303 - t311;
```

B.6 DIFF15.C

```
*************************************************************************
*
*       Program:  diff15.c
*
*       Author:   Kent Taylor
*
*       Date:     20 November 1985
*
*       Purpose:  To compare the outputs of simluation and standard
*                 implementations of Winograd's 15-point DFT algo-
*                 rithm.  The simulation routine uses integer arith-
*                 metic, while the standard uses double precision
*                 real arithmetic.  The comparison is done using
*                 signal-to-noise ratio (SNR) as the measure.  The
*                 SNR is computed by summing the magnitude of all
*                 signal components (standard outputs) and dividing
*                 by the sum of all the noise components (standard
*                 minus simulation).  The SNR is expressed in dB
*                 (10 log ratio); the magnitude of noise is also
*                 stored for comparison.
*
*       Inputs:   files of random numbers
*
*       Outputs:  files of magnitude differences
*                 SNR in dB
*
*************************************************************************
#include        "stdio.h"
#include        "math.h"
#include        "sim15.c"
#include        "mult.c"
#include        "wino15.c"
#define         SIZE    14
#define         INMASK          037777777
#define         RNDMASK         040
#define         PI      3.14159265358979
double          log10 ();
long            mult ();
```

```
main ()
{
 double dbx, dby, simrsig, simisig, snrsr, snrsi, temp1, temp2;
 double diffx, diffy, snrx, snry, a[SIZE+1], b[SIZE+1];
 double realsig, imagsig, realnoise, imagnoise, tempx, tempy;
 long x[SIZE+1], y[SIZE+1];
 int j, k, n, h[SIZE+1], rxbit, rybit, signx, signy;
 char *outfname;
 double snrx1, snrx2, snry1, snry2;
 FILE *gp, *hp, *fopen ();
 hp = fopen ("snr15", "w");              /*  open file for output SNRs  */
 srand (1);
 for (n = 0; n <= 99; n++)
   {
 ************************************************************************
*
*       Assign a unique filename to each output file.
*
************************************************************************/
    switch (n)
       {
         case 0: outfname = "../output15/result00_15"; break;

         case 1: outfname = "../output15/result01_15"; break;

         case 2: outfname = "../output15/result02_15"; break;

         case 3: outfname = "../output15/result03_15"; break;

         case 4: outfname = "../output15/result04_15"; break;

         case 5: outfname = "../output15/result05_15"; break;

         case 6: outfname = "../output15/result06_15"; break;

         case 7: outfname = "../output15/result07_15"; break;

         case 8: outfname = "../output15/result08_15"; break;

         case 9: outfname = "../output15/result09_15"; break;
```

```
case 10: outfname = "../output15/result10_15"; break;
case 11: outfname = "../output15/result11_15"; break;
case 12: outfname = "../output15/result12_15"; break;
case 13: outfname = "../output15/result13_15"; break;
case 14: outfname = "../output15/result14_15"; break;
case 15: outfname = "../output15/result15_15"; break;
case 16: outfname = "../output15/result16_15"; break;
case 17: outfname = "../output15/result17_15"; break;
case 18: outfname = "../output15/result18_15"; break;
case 19: outfname = "../output15/result19_15"; break;
case 20: outfname = "../output15/result20_15"; break;
case 21: outfname = "../output15/result21_15"; break;
case 22: outfname = "../output15/result22_15"; break;
case 23: outfname = "../output15/result23_15"; break;
case 24: outfname = "../output15/result24_15"; break;
case 25: outfname = "../output15/result25_15"; break;
case 26: outfname = "../output15/result26_15"; break;
case 27: outfname = "../output15/result27_15"; break;
case 28: outfname = "../output15/result28_15"; break;
case 29: outfname = "../output15/result29_15"; break;
```

```
case 30: outfname = "../output15/result30_15"; break;

case 31: outfname = "../output15/result31_15"; break;

case 32: outfname = "../output15/result32_15"; break;

case 33: outfname = "../output15/result33_15"; break;

case 34: outfname = "../output15/result34_15"; break;

case 35: outfname = "../output15/result35_15"; break;

case 36: outfname = "../output15/result36_15"; break;

case 37: outfname = "../output15/result37_15"; break;

case 38: outfname = "../output15/result38_15"; break;

case 39: outfname = "../output15/result39_15"; break;

case 40: outfname = "../output15/result40_15"; break;

case 41: outfname = "../output15/result41_15"; break;

case 42: outfname = "../output15/result42_15"; break;

case 43: outfname = "../output15/result43_15"; break;

case 44: outfname = "../output15/result44_15"; break;

case 45: outfname = "../output15/result45_15"; break;

case 46: outfname = "../output15/result46_15"; break;

case 47: outfname = "../output15/result47_15"; break;

case 48: outfname = "../output15/result48_15"; break;

case 49: outfname = "../output15/result49_15"; break;
```

```
case 50: outfname = "../output15/result50_15"; break;
case 51: outfname = "../output15/result51_15"; break;
case 52: outfname = "../output15/result52_15"; break;
case 53: outfname = "../output15/result53_15"; break;
case 54: outfname = "../output15/result54_15"; break;
case 55: outfname = "../output15/result55_15"; break;
case 56: outfname = "../output15/result56_15"; break;
case 57: outfname = "../output15/result57_15"; break;
case 58: outfname = "../output15/result58_15"; break;
case 59: outfname = "../output15/result59_15"; break;
case 60: outfname = "../output15/result60_15"; break;
case 61: outfname = "../output15/result61_15"; break;
case 62: outfname = "../output15/result62_15"; break;
case 63: outfname = "../output15/result63_15"; break;
case 64: outfname = "../output15/result64_15"; break;
case 65: outfname = "../output15/result65_15"; break;
case 66: outfname = "../output15/result66_15"; break;
case 67: outfname = "../output15/result67_15"; break;
case 68: outfname = "../output15/result68_15"; break;
case 69: outfname = "../output15/result69_15"; break;
```

```
case 70: outfname = "../output15/result70_15"; break;

case 71: outfname = "../output15/result71_15"; break;

case 72: outfname = "../output15/result72_15"; break;

case 73: outfname = "../output15/result73_15"; break;

case 74: outfname = "../output15/result74_15"; break;

case 75: outfname = "../output15/result75_15"; break;

case 76: outfname = "../output15/result76_15"; break;

case 77: outfname = "../output15/result77_15"; break;

case 78: outfname = "../output15/result78_15"; break;

case 79: outfname = "../output15/result79_15"; break;

case 80: outfname = "../output15/result80_15"; break;

case 81: outfname = "../output15/result81_15"; break;

case 82: outfname = "../output15/result82_15"; break;

case 83: outfname = "../output15/result83_15"; break;

case 84: outfname = "../output15/result84_15"; break;

case 85: outfname = "../output15/result85_15"; break;

case 86: outfname = "../output15/result86_15"; break;

case 87: outfname = "../output15/result87_15"; break;

case 88: outfname = "../output15/result88_15"; break;

case 89: outfname = "../output15/result89_15"; break;
```

```
            case 90: outfname = "../output15/result90_15"; break;

            case 91: outfname = "../output15/result91_15"; break;

            case 92: outfname = "../output15/result92_15"; break;

            case 93: outfname = "../output15/result93_15"; break;

            case 94: outfname = "../output15/result94_15"; break;

            case 95: outfname = "../output15/result95_15"; break;

            case 96: outfname = "../output15/result96_15"; break;

            case 97: outfname = "../output15/result97_15"; break;

            case 98: outfname = "../output15/result98_15"; break;

            case 99: outfname = "../output15/result99_15"; break;

        }       /* end switch */
 ******************************************************************************
*
*       Fill the input array with random numbers.
*
******************************************************************************
   for (j = 0; j <= SIZE; j++)
     {
        x[j] = (rand () & INMASK) - 4194304;
        y[j] = (rand () & INMASK) - 4194304;
        a[j] = x[j];
        b[j] = y[j];
      h[j] = j;              * initialize index array *
     }
 ******************************************************************************
*
*       Scale the simulation input data and initialize the index
*       array.
*
******************************************************************************
```

```
  for (j = 0; j <= SIZE; j++)
    {
     x[j] = x[j] << 2;    /* zero fill and sign extend */
     y[j] = y[j] << 2;
    }
  sim15 (x, y, h);
  std15 (a, b, h);
/*****************************************************************************
*
*       Take the 23 most significant bits of the simulation result.
*
*****************************************************************************/
  for (j = 0; j <= SIZE; j++)
    {
       signx = 1;
       signy = 1;
       if (x[j] < 0)
         {
           x[j] = -(x[j]);
           signx = -1;
         }
       if (y[j] < 0)
         {
           y[j] = -(y[j]);
           signy = -1;
         }
      rxbit = x[j] & RNDMASK;
      rybit = y[j] & RNDMASK;
      x[j] = x[j] >> 6;
      y[j] = y[j] >> 6;
      if (rxbit != 0) x[j]++;
      if (rybit != 0) y[j]++;
        x[j] = x[j] * signx;
        y[j] = y[j] * signy;
    }
 *****************************************************************************
*
*       Compute the differences between the standard and simluation
*       results (noise components: realnoise and imagnoise).  Compute
*       the SNR for real and imaginary results by dividing the sum of
```

```
*       the standard results (signal components; realsig and imagsig)
*       by the sum of the noise components.  Send the real and imagi-
*       nary SNRs to a file containing SNRs for all inputs; send the
*       differences to a file for storage.
*
*************************************************************************/
   gp = fopen (outfname, "w");
   realsig = 0;
   imagsig = 0;
   realnoise = 0;
   imagnoise = 0;
   simrsig = simisig = 0.0;
   for (j = 0; j <= SIZE; j++)
    {
        tempx = x[j] << 1;
        tempy = y[j] << 1;
        simrsig = simrsig + (tempx * tempx);
        simisig = simisig + (tempy * tempy);
        temp1 = a[j] / 8.0;       /* scale standard outputs down     */
        temp2 = b[j] / 8.0;       /* by a factor of 8 to account     */
                          /* for the difference in input     */
                                  /* scaling (2), output rounding    */
                                  /* (6), and multiplying (1) of     */
                                  /* simulation; 6-1-2 = 3; 2**3 = 8 */
        diffx = temp1 - x[j];
        diffy = temp2 - y[j];
        realsig = realsig + 4.0 * (temp1 * temp1);
        imagsig = imagsig + 4.0 * (temp2 * temp2);
        realnoise = realnoise + (diffx * diffx);
        imagnoise = imagnoise + (diffy * diffy);
        dbx = 138.0;
        dby = 138.0;
        if ((diffx != 0.0) && (a[j] != 0.0))
          dbx = 10.0 * (log10 (temp1 * temp1) - log10 (diffx * diffx));
        if ((diffy != 0.0) && (b[j] != 0.0))
          dby = 10.0 * (log10 (temp2 * temp2) - log10 (diffy * diffy));
      fprintf (gp,"%d    %20.10f%20.10f0, j, dbx, dby);
     }
    printf ("  Finished transferring output to %s0, outfname);
    fclose (gp);
```

```
    snrx1 = snry1 = 138.0;
    snrx2 = snry2 = 0.0;
    snrsr = snrsi = 0.0;
    snrsr = 10.0 * log10 (simrsig);
    snrsi = 10.0 * log10 (simisig);
    snrx1 = 10.0 * log10 (realsig);
    snrx2 = 10.0 * log10 (realnoise);
    snry1 = 10.0 * log10 (imagsig);
    snry2 = 10.0 * log10 (imagnoise);
    snrx = snrx1 - snrx2;
    snry = snry1 - snry2;
    printf ("simrsig = %20.10f   simisig = %20.10f0, snrsr, snrsi);
    printf ("realsig = %20.10f   imagsig = %20.10f0, snrx1, snry1);
    printf ("realnoise = %20.10f   imagnoise = %20.10f0, snrx2, snry2);
    fprintf (hp,"%d       %20.10f   %20.10f0, n, snrx, snry);
  }       /*  end n loop  */
}         /*  end main    */
```

B.7 DIFF16.C

```
/*************************************************************************
*
*       Program:  diff16.c
*
*       Author:   Kent Taylor
*
*       Date:     20 November 1985
*
*       Purpose:  To compare the outputs of simluation and standard
*                 implementations of Winograd's 16-point DFT algo-
*                 rithm.  The simulation routine uses integer arith-
*                 metic, while the standard uses double precision
*                 real arithmetic.  The comparison is done using
*                 signal-to-noise ratio (SNR) as the measure.  The
*                 SNR is computed by summing the magnitude of all
*                 signal components (standard outputs) and dividing
*                 by the sum of all the noise components (standard
*                 minus simulation).  The SNR is expressed in dB
*                 (10 log ratio); the magnitude of noise is also
*                 stored for comparison.
*
*       Inputs:   files of random numbers
*
*       Outputs:  files of magnitude differences
*                 SNR in dB
*
*************************************************************************/
#include      "stdio.h"
#include      "math.h"
#include      "sim16.c"
#include      "mult.c"
#include      "wino16.c"
#define       SIZE    15
#define       MASK          077777777
#define       INMASK              037777777
#define       NUMBER              8388608
#define       OUTMASK             037777777000
#define       RNDMASK             0400
```

```
long            mult ();
double          log10 ();
main ()
{
 double simrsig, simisig, snrsr, snrsi, temp1, temp2, tempx, tempy;
 double dbx, dby, snrx1, snrx2, snry1, snry2;
 double diffx, diffy, snrx, snry, a[SIZE+1], b[SIZE+1];
 double realsig, imagsig, realnoise, imagnoise;
 long x[SIZE+1], y[SIZE+1];
 int j, k, n, nnn, m, h[SIZE+1], rxbit, rybit, signx, signy;
 char *outfname;
 FILE *gp, *hp, *fopen ();
 hp = fopen ("snr16", "w");          /*  open file for output SNRs  */
 for (n = 0; n <= 99; n++)
  {
/*****************************************************************************
*
*       Assign a unique filename to each output file.
*
*****************************************************************************/
    switch (n)
       {
         case 0: outfname = "../output16/result00_16"; break;

         case 1: outfname = "../output16/result01_16"; break;

         case 2: outfname = "../output16/result02_16"; break;

         case 3: outfname = "../output16/result03_16"; break;

         case 4: outfname = "../output16/result04_16"; break;

         case 5: outfname = "../output16/result05_16"; break;

         case 6: outfname = "../output16/result06_16"; break;

         case 7: outfname = "../output16/result07_16"; break;

         case 8: outfname = "../output16/result08_16"; break;
```

```
case 9: outfname = "../output16/result09_16"; break;

case 10: outfname = "../output16/result10_16"; break;

case 11: outfname = "../output16/result11_16"; break;

case 12: outfname = "../output16/result12_16"; break;

case 13: outfname = "../output16/result13_16"; break;

case 14: outfname = "../output16/result14_16"; break;

case 15: outfname = "../output16/result15_16"; break;

case 16: outfname = "../output16/result16_16"; break;

case 17: outfname = "../output16/result17_16"; break;

case 18: outfname = "../output16/result18_16"; break;

case 19: outfname = "../output16/result19_16"; break;

case 20: outfname = "../output16/result20_16"; break;

case 21: outfname = "../output16/result21_16"; break;

case 22: outfname = "../output16/result22_16"; break;

case 23: outfname = "../output16/result23_16"; break;

case 24: outfname = "../output16/result24_16"; break;

case 25: outfname = "../output16/result25_16"; break;

case 26: outfname = "../output16/result26_16"; break;

case 27: outfname = "../output16/result27_16"; break;

case 28: outfname = "../output16/result28_16"; break;
```

```
case 29: outfname = "../output16/result29_16"; break;

case 30: outfname = "../output16/result30_16"; break;

case 31: outfname = "../output16/result31_16"; break;

case 32: outfname = "../output16/result32_16"; break;

case 33: outfname = "../output16/result33_16"; break;

case 34: outfname = "../output16/result34_16"; break;

case 35: outfname = "../output16/result35_16"; break;

case 36: outfname = "../output16/result36_16"; break;

case 37: outfname = "../output16/result37_16"; break;

case 38: outfname = "../output16/result38_16"; break;

case 39: outfname = "../output16/result39_16"; break;

case 40: outfname = "../output16/result40_16"; break;

case 41: outfname = "../output16/result41_16"; break;

case 42: outfname = "../output16/result42_16"; break;

case 43: outfname = "../output16/result43_16"; break;

case 44: outfname = "../output16/result44_16"; break;

case 45: outfname = "../output16/result45_16"; break;

case 46: outfname = "../output16/result46_16"; break;

case 47: outfname = "../output16/result47_16"; break;

case 48: outfname = "../output16/result48_16"; break;
```

```
case 49: outfname = "../output16/result49_16"; break;
case 50: outfname = "../output16/result50_16"; break;

case 51: outfname = "../output16/result51_16"; break;

case 52: outfname = "../output16/result52_16"; break;

case 53: outfname = "../output16/result53_16"; break;

case 54: outfname = "../output16/result54_16"; break;

case 55: outfname = "../output16/result55_16"; break;

case 56: outfname = "../output16/result56_16"; break;

case 57: outfname = "../output16/result57_16"; break;

case 58: outfname = "../output16/result58_16"; break;

case 59: outfname = "../output16/result59_16"; break;

case 60: outfname = "../output16/result60_16"; break;

case 61: outfname = "../output16/result61_16"; break;

case 62: outfname = "../output16/result62_16"; break;

case 63: outfname = "../output16/result63_16"; break;

case 64: outfname = "../output16/result64_16"; break;

case 65: outfname = "../output16/result65_16"; break;

case 66: outfname = "../output16/result66_16"; break;

case 67: outfname = "../output16/result67_16"; break;

case 68: outfname = "../output16/result68_16"; break;

case 69: outfname = "../output16/result69_16"; break;
```

```
case 70: outfname = "../output16/result70_16"; break;

case 71: outfname = "../output16/result71_16"; break;

case 72: outfname = "../output16/result72_16"; break;

case 73: outfname = "../output16/result73_16"; break;

case 74: outfname = "../output16/result74_16"; break;

case 75: outfname = "../output16/result75_16"; break;

case 76: outfname = "../output16/result76_16"; break;

case 77: outfname = "../output16/result77_16"; break;

case 78: outfname = "../output16/result78_16"; break;

case 79: outfname = "../output16/result79_16"; break;

case 80: outfname = "../output16/result80_16"; break;

case 81: outfname = "../output16/result81_16"; break;

case 82: outfname = "../output16/result82_16"; break;

case 83: outfname = "../output16/result83_16"; break;

case 84: outfname = "../output16/result84_16"; break;

case 85: outfname = "../output16/result85_16"; break;

case 86: outfname = "../output16/result86_16"; break;

case 87: outfname = "../output16/result87_16"; break;

case 88: outfname = "../output16/result88_16"; break;

case 89: outfname = "../output16/result89_16"; break;
```

```
            case 90: outfname = "../output16/result90_16"; break;
            case 91: outfname = "../output16/result91_16"; break;
            case 92: outfname = "../output16/result92_16"; break;
            case 93: outfname = "../output16/result93_16"; break;
            case 94: outfname = "../output16/result94_16"; break;
            case 95: outfname = "../output16/result95_16"; break;
            case 96: outfname = "../output16/result96_16"; break;
            case 97: outfname = "../output16/result97_16"; break;
            case 98: outfname = "../output16/result98_16"; break;
            case 99: outfname = "../output16/result99_16"; break;
        }       /* end switch */
/*************************************************************************
*
*       Fill the input array with random numbers.
*
*************************************************************************/
   for (j = 0; j <= SIZE; j++)
     {
        x[j] = (rand () & INMASK) - 4194304;
        y[j] = (rand () & INMASK) - 4194304;
        a[j] = x[j];
        b[j] = y[j];
     }
 *************************************************************************
*
*       Scale the simulation input data and initialize the index
*       array.
*                                                                       *
*************************************************************************
```

```
    realsig = imagsig = 0.0;
    for (j = 0; j <= SIZE; j++)
     {
      x[j] = x[j] << 5;          /*  zero fill and sign extend  */
      y[j] = y[j] << 5;
      h[j] = j;                  /*  initialize index array     */
     }
    sim16 (x, y, h);
    std16 (a, b, h);
/*****************************************************************************
*
*       Take the 23 most significant bits of the simulation result.
*
*****************************************************************************/
    for (j = 0; j <= SIZE; j++)
     {
        signx = 1;
        signy = 1;
        if (x[j] < 0)
          {
            x[j] = -(x[j]);
            signx = -1;
        }
        if (y[j] < 0)
          {
            y[j] = -(y[j]);
            signy = -1;
        }
       rxbit = x[j] & RNDMASK;
         rybit = y[j] & RNDMASK;
       x[j] = x[j] >> 9;
       y[j] = y[j] >> 9;
         if (rxbit != 0) x[j] ++;
         if (rybit != 0) y[j] ++;
         x[j] = x[j] * signx;
         y[j] = y[j] * signy;
     }
 *****************************************************************************
*
*       Compute the differences between the standard and simluation
```

```
*       results (noise components; realnoise and imagnoise).  Compute
*       the SNR for real and imaginary results by dividing the sum of
*       the standard results (signal components; realsig and imagsig)
*       by the sum of the noise components.  Send the real and imagi-
*       nary SNRs to a file containing SNRs for all inputs; send the
*       differences to a file for storage.
*
*************************************************************************/
   gp = fopen (outfname, "w");
   realsig = 0;
   imagsig = 0;
   realnoise = 0;
   imagnoise = 0;
   simrsig = simisig = 0;
   for (j = 0; j <= SIZE; j++)
    {
        tempx = x[j];
        tempy = y[j];
        simrsig = simrsig + (tempx * tempx);
        simisig = simisig + (tempy * tempy);
        temp1 = a[j] / 8.0;       /*  scale standard outputs down by  */
        temp2 = b[j] / 8.0;       /*  a factor of 8 to account for    */
                                  /*  input scaling (5), output       */
                                  /*  rounding (9), and multiplying   */
                                  /*  (1) for simulation; 9-5-1 = 3;  */
                                  /*  2**3 = 8                        */
        diffx = temp1 - x[j];
        diffy = temp2 - y[j];
        realsig = realsig + (temp1 * temp1);
        imagsig = imagsig + (temp2 * temp2);
        realnoise = realnoise + (diffx * diffx);
        imagnoise = imagnoise + (diffy * diffy);
        dbx = 138.0;
        dby = 138.0;
        if (diffx != 0.0)
          dbx = -10.0 * log10 ((diffx * diffx) / (a[j] * a[j]));
        if (diffy != 0.0)
          dby = -10.0 * log10 ((diffy * diffy) / (b[j] * b[j]));
      fprintf (gp,"%d    %20.10f%20.10f0, j, dbx, dby);
      }
```

```
    printf ("  Finished transferring output to %s0. outfname);
    fclose (gp);
    snrsr = 10.0 * log10 (simrsig);
    snrsi = 10.0 * log10 (simisig);
    snrx1 = 10.0 * log10 (realsig);
    snry1 = 10.0 * log10 (imagsig);
    snrx2 = 10.0 * log10 (realnoise);
    snry2 = 10.0 * log10 (imagnoise);
    snrx = snrx1 - snrx2;
    snry = snry1 - snry2;
    printf ("simrsig = %20.10f   simisig = %20.10f0, snrsr, snrsi);
    printf ("realsig = %20.10f   imagsig = %20.10f0, snrx1, snry1);
    printf ("realnoise = %20.10f imagnoise = %20.10f0, snrx2, snry2);
    fprintf (hp,"%d       %20.10f%20.10f0, n, snrx, snry);
  }      /*  end n loop  */
}      /*  end main    */
```

B.8 DIFF17.C

```
*************************************************************************
*
*       Program:  diff17.c
*
*       Author:   Kent Taylor
*
*       Date:     20 November 1985
*
*       Purpose:  To compare the outputs of simluation and standard
*                 implementations of Winograd's 17-point DFT algo-
*                 rithm.  The simulation routine uses integer arith-
*                 metic, while the standard uses double precision
*                 real arithmetic.  The comparison is done using
*                 signal-to-noise ratio (SNR) as the measure.  The
*                 SNR is computed by summing the magnitude of all
*                 signal components (standard outputs) and dividing
*                 by the sum of all the noise components (standard
*                 minus simulation).  The SNR is expressed in dB
*                 (10 log ratio); the magnitude of noise is also
*                 stored for comparison.
*
*       Inputs:   files of random numbers
*
*       Outputs:  files of magnitude differences
*                 SNR in dB
*
*************************************************************************
#include      "stdio.h"
#include      "math.h"
#include      "sim17.c"
#include      "mult.c"
#include      "wino17.c"
#define       SIZE    16
#define       INMASK              037777777
#define       RNDMASK             0200
#define       PI      3.14179265358979
double        log10 ();
long          mult ();
```

```
main ()
{
 double dbx, dby, simrsig, simisig, snrsr, snrsi, temp1, temp2;
 double diffx, diffy, snrx, snry, a[SIZE+1], b[SIZE+1];
 double realsig, imagsig, realnoise, imagnoise, tempx, tempy;
 long x[SIZE+1], y[SIZE+1];
 int j, k, n, h[SIZE+1], rxbit, rybit, signx, signy;
 char *outfname;
 double snrx1, snrx2, snry1, snry2;
 FILE *gp, *hp, *fopen ();
 hp = fopen ("snr17", "w");              /*  open file for output SNRs  */
 srand (1);
 for (n = 0; n <= 99; n++)
   {
 ************************************************************************
 *
 *       Assign a unique filename to each output file.
 *
 ************************************************************************/
     switch (n)
         {
          case 0: outfname = "../output17/result00_17"; break;

          case 1: outfname = "../output17/result01_17"; break;

          case 2: outfname = "../output17/result02_17"; break;

          case 3: outfname = "../output17/result03_17"; break;

          case 4: outfname = "../output17/result04_17"; break;

          case 5: outfname = "../output17/result05_17"; break;

          case 6: outfname = "../output17/result06_17"; break;

          case 7: outfname = "../output17/result07_17"; break;

          case 8: outfname = "../output17/result08_17"; break;

          case 9: outfname = "../output17/result09_17"; break;
```

```
case 10: outfname = "../output17/result10_17"; break;
case 11: outfname = "../output17/result11_17"; break;
case 12: outfname = "../output17/result12_17"; break;
case 13: outfname = "../output17/result13_17"; break;
case 14: outfname = "../output17/result14_17"; break;
case 15: outfname = "../output17/result15_17"; break;
case 16: outfname = "../output17/result16_17"; break;
case 17: outfname = "../output17/result17_17"; break;
case 18: outfname = "../output17/result18_17"; break;
case 19: outfname = "../output17/result19_17"; break;
case 20: outfname = "../output17/result20_17"; break;
case 21: outfname = "../output17/result21_17"; break;
case 22: outfname = "../output17/result22_17"; break;
case 23: outfname = "../output17/result23_17"; break;
case 24: outfname = "../output17/result24_17"; break;
case 25: outfname = "../output17/result25_17"; break;
case 26: outfname = "../output17/result26_17"; break;
case 27: outfname = "../output17/result27_17"; break;
case 28: outfname = "../output17/result28_17"; break;
case 29: outfname = "../output17/result29_17"; break;
```

```
case 30: outfname = "../output17/result30_17"; break;
case 31: outfname = "../output17/result31_17"; break;
case 32: outfname = "../output17/result32_17"; break;
case 33: outfname = "../output17/result33_17"; break;
case 34: outfname = "../output17/result34_17"; break;
case 35: outfname = "../output17/result35_17"; break;
case 36: outfname = "../output17/result36_17"; break;
case 37: outfname = "../output17/result37_17"; break;
case 38: outfname = "../output17/result38_17"; break;
case 39: outfname = "../output17/result39_17"; break;
case 40: outfname = "../output17/result40_17"; break;
case 41: outfname = "../output17/result41_17"; break;
case 42: outfname = "../output17/result42_17"; break;
case 43: outfname = "../output17/result43_17"; break;
case 44: outfname = "../output17/result44_17"; break;
case 45: outfname = "../output17/result45_17"; break;
case 46: outfname = "../output17/result46_17"; break;
case 47: outfname = "../output17/result47_17"; break;
case 48: outfname = "../output17/result48_17"; break;
case 49: outfname = "../output17/result49_17"; break;
```

```
case 50: outfname = "../output17/result50_17"; break;

case 51: outfname = "../output17/result51_17"; break;

case 52: outfname = "../output17/result52_17"; break;

case 53: outfname = "../output17/result53_17"; break;

case 54: outfname = "../output17/result54_17"; break;

case 55: outfname = "../output17/result55_17"; break;

case 56: outfname = "../output17/result56_17"; break;

case 57: outfname = "../output17/result57_17"; break;

case 58: outfname = "../output17/result58_17"; break;

case 59: outfname = "../output17/result59_17"; break;

case 60: outfname = "../output17/result60_17"; break;

case 61: outfname = "../output17/result61_17"; break;

case 62: outfname = "../output17/result62_17"; break;

case 63: outfname = "../output17/result63_17"; break;

case 64: outfname = "../output17/result64_17"; break;

case 65: outfname = "../output17/result65_17"; break;

case 66: outfname = "../output17/result66_17"; break;

case 67: outfname = "../output17/result67_17"; break;

case 68: outfname = "../output17/result68_17"; break;

case 69: outfname = "../output17/result69_17"; break;
```

```
case 70: outfname = "../output17/result70_17"; break;
case 71: outfname = "../output17/result71_17"; break;
case 72: outfname = "../output17/result72_17"; break;
case 73: outfname = "../output17/result73_17"; break;
case 74: outfname = "../output17/result74_17"; break;
case 75: outfname = "../output17/result75_17"; break;
case 76: outfname = "../output17/result76_17"; break;
case 77: outfname = "../output17/result77_17"; break;
case 78: outfname = "../output17/result78_17"; break;
case 79: outfname = "../output17/result79_17"; break;
case 80: outfname = "../output17/result80_17"; break;
case 81: outfname = "../output17/result81_17"; break;
case 82: outfname = "../output17/result82_17"; break;
case 83: outfname = "../output17/result83_17"; break;
case 84: outfname = "../output17/result84_17"; break;
case 85: outfname = "../output17/result85_17"; break;
case 86: outfname = "../output17/result86_17"; break;
case 87: outfname = "../output17/result87_17"; break;
case 88: outfname = "../output17/result88_17"; break;
case 89: outfname = "../output17/result89_17"; break;
```

```
            case 90: outfname = "../output17/result90_17"; break;
            case 91: outfname = "../output17/result91_17"; break;
            case 92: outfname = "../output17/result92_17"; break;
            case 93: outfname = "../output17/result93_17"; break;
            case 94: outfname = "../output17/result94_17"; break;
            case 95: outfname = "../output17/result95_17"; break;
            case 96: outfname = "../output17/result96_17"; break;
            case 97: outfname = "../output17/result97_17"; break;
            case 98: outfname = "../output17/result98_17"; break;
            case 99: outfname = "../output17/result99_17"; break;
        }       /* end switch */
/*****************************************************************************
*
*       Fill the input array with random numbers.
*
*****************************************************************************/
   for (j = 0; j <= SIZE; j++)
     {
        x[j] = (rand () & INMASK) - 4194304;
        y[j] = (rand () & INMASK) - 4194304;
        a[j] = x[j];
        b[j] = y[j];
     }
 *****************************************************************************
*
*       Scale the simulation input data and initialize the index
*       array.
*
*****************************************************************************/
   for (j = 0; j <= SIZE; j++)
```

```
    {
     x[j] = x[j] << 3;    /* zero fill and sign extend */
     y[j] = y[j] << 3;
     h[j] = j;            /* initialize index array */
    }
  sim17 (x, y, h);
  std17 (a, b, h);
/*************************************************************************
*
*       Take the 23 most significant bits of the simulation result.
*
*************************************************************************/
  for (j = 0; j <= SIZE; j++)
   {
      signx = 1;
      signy = 1;
      if (x[j] < 0)
        {
          x[j] = -(x[j]);
          signx = -1;
        }
      if (y[j] < 0)
        {
          y[j] = -(y[j]);
          signy = -1;
        }
     rxbit = x[j] & RNDMASK;
     rybit = y[j] & RNDMASK;
     x[j] = x[j] >> 8;
     y[j] = y[j] >> 8;
     if (rxbit != 0) x[j]++;
     if (rybit != 0) y[j]++;
       x[j] = x[j] * signx;
       y[j] = y[j] * signy;
   }
 *************************************************************************
*
*       Compute the differences between the standard and simluation
*       results (noise components; realnoise and imagnoise). Compute
*       the SNR for real and imaginary results by dividing the sum of
```

```
*       the standard results (signal components; realsig and imagsig)
*       by the sum of the noise components.  Send the real and imagi-
*       nary SNRs to a file containing SNRs for all inputs; send the
*       differences to a file for storage.
*
*************************************************************************/
   gp = fopen (outfname, "w");
   realsig = 0;
   imagsig = 0;
   realnoise = 0;
   imagnoise = 0;
   simrsig = simisig = 0.0;
   for (j = 0; j <= SIZE; j++)
    {
        tempx = x[j] << 2;
        tempy = y[j] << 2;
        simrsig = simrsig + (tempx * tempx);
        simisig = simisig + (tempy * tempy);
        temp1 = a[j] / 16.0;     /* scale standard outputs down      */
        temp2 = b[j] / 16.0;     /* by a factor of 16 to account     */
                        /* for the difference in input      */
                                 /* scaling (3), output rounding     */
                                 /* (8), and multiplying (1) of      */
                                 /* simulation; 8-1-3 = 4; 2**4 = 16 */
        diffx = temp1 - x[j];
        diffy = temp2 - y[j];
        realsig = realsig + 16.0 * (temp1 * temp1);
        imagsig = imagsig + 16.0 * (temp2 * temp2);
        realnoise = realnoise + (diffx * diffx);
        imagnoise = imagnoise + (diffy * diffy);
        dbx = 138.0;
        dby = 138.0;
        if ((diffx != 0.0) && (a[j] != 0.0))
          dbx = 10.0 * (log10 (temp1 * temp1) - log10 (diffx * diffx));
        if ((diffy != 0.0) && (b[j] != 0.0))
          dby = 10.0 * (log10 (temp2 * temp2) - log10 (diffy * diffy));
      fprintf (gp,"%d    %20.10f%20.10f0, j, dbx, dby);
     }
    printf ("  Finished transferring output to %s0, outfname);
    fclose (gp);
```

```
    snrx1 = snry1 = 138.0;
    snrx2 = snry2 = 0.0;
    snrsr = snrsi = 0.0;
    snrsr = 10.0 * log10 (simrsig);
    snrsi = 10.0 * log10 (simisig);
    snrx1 = 10.0 * log10 (realsig);
    snrx2 = 10.0 * log10 (realnoise);
    snry1 = 10.0 * log10 (imagsig);
    snry2 = 10.0 * log10 (imagnoise);
    snrx = snrx1 - snrx2;
    snry = snry1 - snry2;
    printf ("simrsig = %20.10f   simisig = %20.10f0, snrsr, snrsi);
    printf ("realsig = %20.10f   imagsig = %20.10f0, snrx1, snry1);
    printf ("realnoise = %20.10f   imagnoise = %20.10f0, snrx2, snry2);
    fprintf (hp,"%d       %20.10f   %20.10f0, n, snrx, snry);
  }       /*  end n loop  */
}         /*  end main    */
```

B.9 STDDIFF_16.C

```
/*************************************************************************
*
*       Program:  stddiff_16.c
*
*       Author:   Kent Taylor
*
*       Date:     31 October 1985
*
*       Purpose:  To compare the outputs of direct DFT and standard
*                 implementation of Winograd's 16-point DFT algo-
*                 rithm. Both routines use double precision arith-
*                 metic The comparison is done using signal-to-noise
*                 ratio (SNR) as the measure.  The SNR is computed
*                 by summing the magnitude of all signal components
*                 (DFT outputs) and dividing by the sum of all the
*                 noise components (DFT minus Winograd).  The SNR
*                 is expressed in dB (10 log ratio); the individual
*                 SNRs are also stored for comparison.
*
*       Inputs:   files of random numbers
*
*       Outputs:  files of individual SNRs
*                 SNR in dB
*
*************************************************************************/
#include        "stdio.h"
#include        "math.h"
#include        "wino16.c"
#define         SIZE    15
#define         INMASK                  037777777
#define         PI      3.14159265358979
double          sin ();
double          cos ();
double          hypot ();
double          log10 ();
main ()
{
  double x[SIZE-1], y[SIZE-1], x1, xx;
```

```
double xa[SIZE+1], yb[SIZE+1];
double dbx, dby;
double diffx, diffy, snrx, snry, a[SIZE+1], b[SIZE+1];
double realsig, imagsig, realnoise, imagnoise;
double dbxmin, dbymin;
int j, k, n, h[SIZE+1];
char *outfname;
FILE *gp, *hp, *fopen ();
hp = fopen ("dsnr16", "w");          /*  open file for output SNRs  */
srand (1);
for (n = 0; n <= 99; n++)
  {
/*************************************************************************
*
*       Assign a unique filename to each output file.
*
*************************************************************************/
    switch (n)
        {
         case 0: outfname = "../doutput16/result00_16"; break;

         case 1: outfname = "../doutput16/result01_16"; break;

         case 2: outfname = "../doutput16/result02_16"; break;

         case 3: outfname = "../doutput16/result03_16"; break;

         case 4: outfname = "../doutput16/result04_16"; break;

         case 5: outfname = "../doutput16/result05_16"; break;

         case 6: outfname = "../doutput16/result06_16"; break;

         case 7: outfname = "../doutput16/result07_16"; break;

         case 8: outfname = "../doutput16/result08_16"; break;

         case 9: outfname = "../doutput16/result09_16"; break;

         case 10: outfname = "../doutput16/result10_16"; break;
```

```
case 11: outfname = "../doutput16/result11_16"; break;

case 12: outfname = "../doutput16/result12_16"; break;

case 13: outfname = "../doutput16/result13_16"; break;

case 14: outfname = "../doutput16/result14_16"; break;

case 15: outfname = "../doutput16/result15_16"; break;

case 16: outfname = "../doutput16/result16_16"; break;

case 17: outfname = "../doutput16/result17_16"; break;

case 18: outfname = "../doutput16/result18_16"; break;

case 19: outfname = "../doutput16/result19_16"; break;

case 20: outfname = "../doutput16/result20_16"; break;

case 21: outfname = "../doutput16/result21_16"; break;

case 22: outfname = "../doutput16/result22_16"; break;

case 23: outfname = "../doutput16/result23_16"; break;

case 24: outfname = "../doutput16/result24_16"; break;

case 25: outfname = "../doutput16/result25_16"; break;

case 26: outfname = "../doutput16/result26_16"; break;

case 27: outfname = "../doutput16/result27_16"; break;

case 28: outfname = "../doutput16/result28_16"; break;

case 29: outfname = "../doutput16/result29_16"; break;

case 30: outfname = "../doutput16/result30_16"; break;
```

```
case 31: outfname = "../doutput16/result31_16"; break;

case 32: outfname = "../doutput16/result32_16"; break;

case 33: outfname = "../doutput16/result33_16"; break;

case 34: outfname = "../doutput16/result34_16"; break;

case 35: outfname = "../doutput16/result35_16"; break;

case 36: outfname = "../doutput16/result36_16"; break;

case 37: outfname = "../doutput16/result37_16"; break;

case 38: outfname = "../doutput16/result38_16"; break;

case 39: outfname = "../doutput16/result39_16"; break;

case 40: outfname = "../doutput16/result40_16"; break;

case 41: outfname = "../doutput16/result41_16"; break;

case 42: outfname = "../doutput16/result42_16"; break;

case 43: outfname = "../doutput16/result43_16"; break;

case 44: outfname = "../doutput16/result44_16"; break;

case 45: outfname = "../doutput16/result45_16"; break;

case 46: outfname = "../doutput16/result46_16"; break;

case 47: outfname = "../doutput16/result47_16"; break;

case 48: outfname = "../doutput16/result48_16"; break;

case 49: outfname = "../doutput16/result49_16"; break;
case 50: outfname = "../doutput16/result50_16"; break;
```

```
case 51: outfname = "../doutput16/result51_16"; break;

case 52: outfname = "../doutput16/result52_16"; break;

case 53: outfname = "../doutput16/result53_16"; break;

case 54: outfname = "../doutput16/result54_16"; break;

case 55: outfname = "../doutput16/result55_16"; break;

case 56: outfname = "../doutput16/result56_16"; break;

case 57: outfname = "../doutput16/result57_16"; break;

case 58: outfname = "../doutput16/result58_16"; break;

case 59: outfname = "../doutput16/result59_16"; break;

case 60: outfname = "../doutput16/result60_16"; break;

case 61: outfname = "../doutput16/result61_16"; break;

case 62: outfname = "../doutput16/result62_16"; break;

case 63: outfname = "../doutput16/result63_16"; break;

case 64: outfname = "../doutput16/result64_16"; break;

case 65: outfname = "../doutput16/result65_16"; break;

case 66: outfname = "../doutput16/result66_16"; break;

case 67: outfname = "../doutput16/result67_16"; break;

case 68: outfname = "../doutput16/result68_16"; break;

case 69: outfname = "../doutput16/result69_16"; break;

case 70: outfname = "../doutput16/result70_16"; break;
```

```
case 71: outfname = "../doutput16/result71_16"; break;

case 72: outfname = "../doutput16/result72_16"; break;

case 73: outfname = "../doutput16/result73_16"; break;

case 74: outfname = "../doutput16/result74_16"; break;

case 75: outfname = "../doutput16/result75_16"; break;

case 76: outfname = "../doutput16/result76_16"; break;

case 77: outfname = "../doutput16/result77_16"; break;

case 78: outfname = "../doutput16/result78_16"; break;

case 79: outfname = "../doutput16/result79_16"; break;

case 80: outfname = "../doutput16/result80_16"; break;

case 81: outfname = "../doutput16/result81_16"; break;

case 82: outfname = "../doutput16/result82_16"; break;

case 83: outfname = "../doutput16/result83_16"; break;

case 84: outfname = "../doutput16/result84_16"; break;

case 85: outfname = "../doutput16/result85_16"; break;

case 86: outfname = "../doutput16/result86_16"; break;

case 87: outfname = "../doutput16/result87_16"; break;

case 88: outfname = "../doutput16/result88_16"; break;

case 89: outfname = "../doutput16/result89_16"; break;

case 90: outfname = "../doutput16/result90_16"; break;
```

```
        case 91: outfname = "../doutput16/result91_16"; break;

        case 92: outfname = "../doutput16/result92_16"; break;

        case 93: outfname = "../doutput16/result93_16"; break;

        case 94: outfname = "../doutput16/result94_16"; break;

        case 95: outfname = "../doutput16/result95_16"; break;

        case 96: outfname = "../doutput16/result96_16"; break;

        case 97: outfname = "../doutput16/result97_16"; break;

        case 98: outfname = "../doutput16/result98_16"; break;

        case 99: outfname = "../doutput16/result99_16"; break;

      }       /* end switch */
/*************************************************************************
*
*       Fill the input array with random numbers.
*
*************************************************************************/
  for (j = 0; j <= SIZE; j++)
   {
      x[j] = (rand () & INMASK) - 4194304;
      y[j] = (rand () & INMASK) - 4194304;
      a[j] = x[j];
      b[j] = y[j];
      h[j] = j;
   }
  std16 (a, b, h);
*       Compute the DFT directly       */
 for (j = 0; j <= SIZE; j++)
  {
   xa[j] = yb[j] = 0;
   for (k = 0; k <= SIZE; k++)
      {
       x1 = (2.0 * PI) / (SIZE+1);
```

```
        xx = k * j;
        xa[j] = xa[j] + (x[k] * cos (xx * x1));
        yb[j] = yb[j] - (x[k] * sin (xx * x1));
        yb[j] = yb[j] + (y[k] * cos (xx * x1));
        xa[j] = xa[j] + (y[k] * sin (xx * x1));
      }
  }
/****************************************************************************
*
*       Compute the differences between the standard and direct DFT
*       results (noise components; realnoise and imagnoise). Compute
*       the SNR for real and imaginary results by dividing the sum of
*       the standard results (signal components; realsig and imagsig)
*       by the sum of the noise components. Send the real and imagi-
*       nary SNRs to a file containing SNRs for all inputs; send the
*       differences to a file for storage.
*
****************************************************************************/
  gp = fopen (outfname, "w");
  realsig = 0;
  imagsig = 0;
  realnoise = 0;
  imagnoise = 0;
  for (j = 0; j <= SIZE; j++)
    {
      diffx = xa[j] - a[j];
      diffy = yb[j] - b[j];
      realsig = realsig + (xa[j] * xa[j]);
      imagsig = imagsig + (yb[j] * yb[j]);
      realnoise = realnoise + (diffx * diffx);
      imagnoise = imagnoise + (diffy * diffy);
      dbx = 138.0;
      dby = 138.0;
      if ((diffx != 0.0) && (xa[j] != 0.0))
        dbx = -10.0 * log10 ((diffx * diffx) / (xa[j] * xa[j]));
      if ((diffy != 0.0) && (yb[j] != 0.0))
        dby = -10.0 * log10 ((diffy * diffy) / (yb[j] * yb[j]));
     fprintf (gp,"%d    %20.10f%20.10f0, j, dbx, dby);
    }
  printf ("  Finished transferring output to %s0, outfname);
```

```
    printf ("realsig = %20.10f    imagsig = %20.10f0, realsig, imagsig);
    printf ("realnoise = %20.10f", realnoise);
    printf ("       imagnoise = %20.10f0, imagnoise);
    fclose (gp);
    snrx = 10.0 * log10 (realsig/realnoise);
    snry = 10.0 * log10 (imagsig/imagnoise);
    fprintf (hp,"%d       %20.10f%20.10f0, n, snrx, snry);
  }       /*  end n loop  */
}         /*  end main    */
```

B.10 STDDIFF_240.C

```
/*************************************************************************
*
*       Program:  stddiff_240.c
*
*       Author:   Kent Taylor
*
*       Date:     4 November 1985
*
*       Purpose:  To compare the outputs of simluation and standard
*                 implementations of Winograd's 16-point DFT algo-
*                 rithm.  The simulation routine uses integer arith-
*                 metic, while the standard uses double precision
*                 real arithmetic.  The comparison is done using
*                 signal-to-noise ratio (SNR) as the measure.  The
*                 SNR is computed by summing the magnitude of all
*                 signal components (standard outputs) and dividing
*                 by the sum of all the noise components (standard
*                 minus simulation).  The SNR is expressed in dB
*                 (10 log ratio); the magnitude of noise is also
*                 stored for comparison.
*
*       Inputs:   files of random numbers
*
*       Outputs:  files of magnitude differences
*                 SNR in dB
*
*************************************************************************/
#include      "stdio.h"
#include      "math.h"
#include      "wino15.c"
#include      "wino16.c"
#define       INMASK      037777777
#define       PI      3.14159265357989
#define       SIZE    239
#define       LIMIT   20
#define       NUMBER      4194303
long          rand ();
double        sin ();
```

```
double          cos ();
double          log10 ();
main ()
{
 double winorsig, winoisig, snrsr, snrsi;
 double snrx1, snrx2, snry1, snry2;
 double diffx, diffy, snrx, snry, dbx, dby;
 double a[SIZE+1], a1[SIZE+1], b[SIZE+1], b1[SIZE+1];
 double realsig, imagsig, realnoise, imagnoise;
 double x1, y1;
 double x2[SIZE+1], y2[SIZE+1];
 double x[SIZE+1], y[SIZE+1];
 long temp1, temp2;
 int j, k, l, n, nn, n1, n2, n3, h[LIMIT];
 int ni[4], unsc;
 char *outfname;
 FILE *gp, *hp, *fopen ();
 ni[0] = 15;
 ni[1] = 16;
 unsc = 31;
 hp = fopen ("snr240", "w");
 for (n = 0; n <= 99; n++)
  {
 ************************************************************************
*
*       Assign a unique filename to each output file.
*
************************************************************************
    switch (n)
        {
          case 0: outfname = "../output240/result00_240"; break;

          case 1: outfname = "../output240/result01_240"; break;

          case 2: outfname = "../output240/result02_240"; break;

          case 3: outfname = "../output240/result03_240"; break;

          case 4: outfname = "../output240/result04_240"; break;
```

```
case 5: outfname = "../output240/result05_240"; break;

case 6: outfname = "../output240/result06_240"; break;

case 7: outfname = "../output240/result07_240"; break;

case 8: outfname = "../output240/result08_240"; break;

case 9: outfname = "../output240/result09_240"; break;

case 10: outfname = "../output240/result10_240"; break;

case 11: outfname = "../output240/result11_240"; break;

case 12: outfname = "../output240/result12_240"; break;

case 13: outfname = "../output240/result13_240"; break;

case 14: outfname = "../output240/result14_240"; break;

case 15: outfname = "../output240/result15_240"; break;

case 16: outfname = "../output240/result16_240"; break;

case 17: outfname = "../output240/result17_240"; break;

case 18: outfname = "../output240/result18_240"; break;

case 19: outfname = "../output240/result19_240"; break;

case 20: outfname = "../output240/result20_240"; break;

case 21: outfname = "../output240/result21_240"; break;

case 22: outfname = "../output240/result22_240"; break;

case 23: outfname = "../output240/result23_240"; break;

case 24: outfname = "../output240/result24_240"; break;
```

```
case 25: outfname = "../output240/result25_240"; break;

case 26: outfname = "../output240/result26_240"; break;

case 27: outfname = "../output240/result27_240"; break;

case 28: outfname = "../output240/result28_240"; break;

case 29: outfname = "../output240/result29_240"; break;

case 30: outfname = "../output240/result30_240"; break;

case 31: outfname = "../output240/result31_240"; break;

case 32: outfname = "../output240/result32_240"; break;

case 33: outfname = "../output240/result33_240"; break;

case 34: outfname = "../output240/result34_240"; break;

case 35: outfname = "../output240/result35_240"; break;

case 36: outfname = "../output240/result36_240"; break;

case 37: outfname = "../output240/result37_240"; break;

case 38: outfname = "../output240/result38_240"; break;

case 39: outfname = "../output240/result39_240"; break;

case 40: outfname = "../output240/result40_240"; break;

case 41: outfname = "../output240/result41_240"; break;

case 42: outfname = "../output240/result42_240"; break;

case 43: outfname = "../output240/result43_240"; break;

case 44: outfname = "../output240/result44_240"; break;
```

```
case 45: outfname = "../output240/result45_240"; break;

case 46: outfname = "../output240/result46_240"; break;

case 47: outfname = "../output240/result47_240"; break;

case 48: outfname = "../output240/result48_240"; break;

case 49: outfname = "../output240/result49_240"; break;
case 50: outfname = "../output240/result50_240"; break;

case 51: outfname = "../output240/result51_240"; break;

case 52: outfname = "../output240/result52_240"; break;

case 53: outfname = "../output240/result53_240"; break;

case 54: outfname = "../output240/result54_240"; break;

case 55: outfname = "../output240/result55_240"; break;

case 56: outfname = "../output240/result56_240"; break;

case 57: outfname = "../output240/result57_240"; break;

case 58: outfname = "../output240/result58_240"; break;

case 59: outfname = "../output240/result59_240"; break;

case 60: outfname = "../output240/result60_240"; break;

case 61: outfname = "../output240/result61_240"; break;

case 62: outfname = "../output240/result62_240"; break;

case 63: outfname = "../output240/result63_240"; break;

case 64: outfname = "../output240/result64_240"; break;

case 65: outfname = "../output240/result65_240"; break;
```

```
case 66: outfname = "../output240/result66_240"; break;
case 67: outfname = "../output240/result67_240"; break;
case 68: outfname = "../output240/result68_240"; break;
case 69: outfname = "../output240/result69_240"; break;
case 70: outfname = "../output240/result70_240"; break;
case 71: outfname = "../output240/result71_240"; break;
case 72: outfname = "../output240/result72_240"; break;
case 73: outfname = "../output240/result73_240"; break;
case 74: outfname = "../output240/result74_240"; break;
case 75: outfname = "../output240/result75_240"; break;
case 76: outfname = "../output240/result76_240"; break;
case 77: outfname = "../output240/result77_240"; break;
case 78: outfname = "../output240/result78_240"; break;
case 79: outfname = "../output240/result79_240"; break;
case 80: outfname = "../output240/result80_240"; break;
case 81: outfname = "../output240/result81_240"; break;
case 82: outfname = "../output240/result82_240"; break;
case 83: outfname = "../output240/result83_240"; break;
case 84: outfname = "../output240/result84_240"; break;
case 85: outfname = "../output240/result85_240"; break;
```

```
            case 86: outfname = "../output240/result86_240"; break;

            case 87: outfname = "../output240/result87_240"; break;

            case 88: outfname = "../output240/result88_240"; break;

            case 89: outfname = "../output240/result89_240"; break;

            case 90: outfname = "../output240/result90_240"; break;

            case 91: outfname = "../output240/result91_240"; break;

            case 92: outfname = "../output240/result92_240"; break;

            case 93: outfname = "../output240/result93_240"; break;

            case 94: outfname = "../output240/result94_240"; break;

            case 95: outfname = "../output240/result95_240"; break;

            case 96: outfname = "../output240/result96_240"; break;

            case 97: outfname = "../output240/result97_240"; break;

            case 98: outfname = "../output240/result98_240"; break;

            case 99: outfname = "../output240/result99_240"; break;

        }       /* end switch */
    for (j = 0; j <= SIZE; j++)
      {
        temp1 = (rand () & INMASK) - NUMBER;
        temp2 = (rand () & INMASK) - NUMBER;
        x[j] = temp1;
        y[j] = temp2;
        a[j] = temp1;
        b[j] = temp2;
      }
    for (j = 0; j <= 1; j++)                    * pfa outer loop *
```

```
    {
     n1 = ni[j];
     n2 = 240 / n1;
     for (k = 0; k <= SIZE; k = k + n1)        /*  pfa inner loop  */
         {
          h[0] = k;
          n3 = k;
          for (l = 1; l <= n1-1; l++)
           {
            n3 = n3 + n2;
            if (n3 >= 240)
                n3 = n3 - 240;
            h[l] = n3;
           }
/*****************************************************************************
*
*       Compute the DFT using Winograd's Small DFT algorithm for
*       either the 15-point or the 16-point DFT.
*
*****************************************************************************/
          switch (n1)
           {
            case 15:
                {
                 std15 (a, b, h);
                 break;
                }
            case 16:
            {
                std16 (a, b, h);
                 break;
                }
           }
         }        /*  end pfa inner loop  */
    }     *  end pfa outer loop  */
/*       Unscramble the PFA results       */
  k = 0;
  for (j = 0; j <= SIZE; j++)
   {
    a1[j] = a[k];
```

```
    b1[j] = b[k];
    k = k + unsc;
    if (k >= 240)
       k = k - 240;
  }
/*      Compute the DFT directly       */
 for (j = 0; j <= SIZE; j++)
  {
   x2[j] = y2[j] = 0;
   for (k = 0; k <= SIZE; k++)
       {
        x1 = (2.0 * PI) / 240;
        y1 = k * j;
        x2[j] = x2[j] + (x[k] * cos (y1 * x1));
        y2[j] = y2[j] - (x[k] * sin (y1 * x1));
        y2[j] = y2[j] + (y[k] * cos (y1 * x1));
        x2[j] = x2[j] + (y[k] * sin (y1 * x1));
       }
  }
/*************************************************************************
*
*       Compute the differences between the standard and direct DFT
*       results (noise components; realnoise and imagnoise). Compute
*       the SNR for real and imaginary results by dividing the sum of
*       the standard results (signal components; realsig and imagsig)
*       by the sum of the noise components. Send the real and imagi-
*       nary SNRs to a file containing SNRs for all inputs; send the
*       differences to a file for storage.
*
*************************************************************************/
   gp = fopen (outfname, "w");
   realsig = 0;
   imagsig = 0;
   realnoise = 0;
   imagnoise = 0;
   winorsig = winoisig = 0;
   for (j = 0; j <= SIZE; j++)
     {
       diffx = x2[j] - a1[j];
       diffy = y2[j] - b1[j];
```

```
        winorsig = winorsig + (a1[j] * a1[j]);
        winoisig = winoisig + (b1[j] * b1[j]);
        realsig = realsig + (x2[j] * x2[j]);
        imagsig = imagsig + (y2[j] * y2[j]);
        realnoise = realnoise + (diffx * diffx);
        imagnoise = imagnoise + (diffy * diffy);
        dbx = 138.0;
        dby = 138.0;
        if (diffx != 0.0)
          dbx = -10.0 * log10 ((diffx * diffx) / (x2[j] * x2[j]));
        if (diffy != 0.0)
          dby = -10.0 * log10 ((diffy * diffy) / (y2[j] * y2[j]));
       fprintf (gp,"%d    %20.10f%20.10f0, j, dbx, dby);
      }
    printf ("  Finished transferring output to %s0, outfname);
    fclose (gp);
    snrsr = 10.0 * log10 (winorsig);
    snrsi = 10.0 * log10 (winoisig);
    snrx1 = 10.0 * log10 (realsig);
    snrx2 = 10.0 * log10 (realnoise);
    snry1 = 10.0 * log10 (imagsig);
    snry2 = 10.0 * log10 (imagnoise);
    snrx = snrx1 - snrx2;
    snry = snry1 - snry2;
    printf ("snrsr = %20.10f      snrsi = %20.10f0, snrsr, snrsi);
    printf ("realsig = %20.10f    imagsig = %20.10f0, snrx1, snry1);
    printf ("realnoise = %20.10f", snrx2);
    printf ("       imagnoise = %20.10f0, snry2);
    fprintf (hp,"%d       %20.10f%20.10f0, n, snrx, snry);
  }       /*  end n loop  */
}         /*  end main    */
```

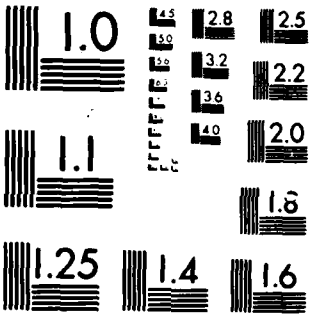
Appendix C

Simulation Result Listings

The following listings are the signal-to-noise ratios (SNRs) computed by the programs given in Appendix B. The first set of listings are for the standard Winograd module compared with the direct DFT. The results are for blocklengths of 15, 16, 17, 240, 255, and 272. The second set of listings are for the standard Winograd module compared with the simulation. The results are for the blocklengths of 15, 16, and 17.

AD-A163 963 ARCHITECTURE AND NUMERICAL ACCURACY OF HIGH-SPEED DFT (DISCRETE FOURIER T.. (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K TAYLOR

UNCLASSIFIED DEC 85 AFIT/GE/ENG/85D-47 F/G 9/1 NL



x[0] =	270.3267211914	y[0] =	271.8259582520
x[1] =	270.1993713379	y[1] =	270.9765930176
x[2] =	270.8795166016	y[2] =	274.7530517578
x[3] =	266.9262084961	y[3] =	270.4941711426
x[4] =	272.4660034180	y[4] =	268.6256713867
x[5] =	270.1086120605	y[5] =	269.2436523438
x[6] =	270.3933715820	y[6] =	273.6777343750
x[7] =	268.2878112793	y[7] =	272.1769409180
x[8] =	269.2735595703	y[8] =	267.9232482910
x[9] =	272.6429443359	y[9] =	268.7232666016
x[10] =	274.5322570801	y[10] =	266.1694641113
x[11] =	265.2796325684	y[11] =	273.1672058105
x[12] =	266.3085327148	y[12] =	272.1916503906
x[13] =	270.5764160156	y[13] =	268.4227294922
x[14] =	267.3804626465	y[14] =	267.9915771484
x[15] =	272.3644104004	y[15] =	268.3309936523
x[16] =	270.6399230957	y[16] =	271.3390502930
x[17] =	272.6228027344	y[17] =	268.4317626953
x[18] =	269.4873046875	y[18] =	271.5711669922
x[19] =	271.9138793945	y[19] =	270.1387939453
x[20] =	268.1859130859	y[20] =	273.8852233887
x[21] =	267.5024414063	y[21] =	272.5805969238
x[22] =	269.6394958496	y[22] =	268.3592529297
x[23] =	269.9055786133	y[23] =	264.6501464844
x[24] =	265.3494873047	y[24] =	270.0167541504
x[25] =	267.4769592285	y[25] =	273.1098937988
x[26] =	269.3949279785	y[26] =	273.7298278809
x[27] =	271.2135009766	y[27] =	269.0087585449
x[28] =	271.2580871582	y[28] =	266.7854003906
x[29] =	273.2968750000	y[29] =	269.7937011719
x[30] =	269.2456359863	y[30] =	271.5105590820
x[31] =	269.8008728027	y[31] =	268.7413940430
x[32] =	268.5186462402	y[32] =	268.1859436035
x[33] =	270.1049804688	y[33] =	270.4666748047
x[34] =	271.1744079590	y[34] =	272.6804504395
x[35] =	269.0790100098	y[35] =	273.7297058105
x[36] =	265.5129394531	y[36] =	273.4932250977
x[37] =	269.5553283691	y[37] =	266.9835510254
x[38] =	270.5327758789	y[38] =	267.7444152832
x[39] =	270.8727416992	y[39] =	269.7133483887

x[40] =	273.2836303711	y[40] =	271.7077026367
x[41] =	271.2531738281	y[41] =	268.3661193848
x[42] =	268.0216979980	y[42] =	270.5642089844
x[43] =	266.8464050293	y[43] =	268.9776000977
x[44] =	270.1110839844	y[44] =	268.9873352051
x[45] =	269.3290710449	y[45] =	269.8417358398
x[46] =	269.3952026367	y[46] =	272.2424621582
x[47] =	270.2401123047	y[47] =	266.4663391113
x[48] =	268.9219970703	y[48] =	270.0077209473
x[49] =	267.0398864746	y[49] =	269.5416564941
x[50] =	271.3050537109	y[50] =	267.8937683105
x[51] =	269.7127380371	y[51] =	270.4773559570
x[52] =	271.4542846680	y[52] =	271.5123901367
x[53] =	270.1429748535	y[53] =	271.8498535156
x[54] =	270.7604370117	y[54] =	266.8149108887
x[55] =	270.9530029297	y[55] =	272.4279479980
x[56] =	266.4183654785	y[56] =	274.4346618652
x[57] =	271.4386291504	y[57] =	266.8522949219
x[58] =	272.0279235840	y[58] =	270.6676635742
x[59] =	270.7712097168	y[59] =	268.7197875977
x[60] =	273.1189880371	y[60] =	262.9579467773
x[61] =	271.7225646973	y[61] =	265.9170227051
x[62] =	270.9621887207	y[62] =	270.9728698730
x[63] =	267.4638671875	y[63] =	269.7576599121
x[64] =	271.7090454102	y[64] =	266.3128662109
x[65] =	270.7944641113	y[65] =	266.5230712891
x[66] =	270.7575988770	y[66] =	271.5975341797
x[67] =	267.7533264160	y[67] =	271.5601806641
x[68] =	271.1271667480	y[68] =	268.4807739258
x[69] =	268.4465637207	y[69] =	273.1932067871
x[70] =	273.8880615234	y[70] =	269.1509399414
x[71] =	268.6777954102	y[71] =	267.7762451172
x[72] =	271.8719787598	y[72] =	267.9895019531
x[73] =	272.1277770996	y[73] =	267.4620971680
x[74] =	273.5524902344	y[74] =	268.9857177734
x[75] =	268.7622680664	y[75] =	267.8697509766
x[76] =	270.1966552734	y[76] =	270.9008178711
x[77] =	270.6634216309	y[77] =	270.4273071289
x[78] =	269.5248413086	y[78] =	271.5830078125
x[79] =	265.8132934570	y[79] =	271.5101623535

x[80] =	273.3381042480	y[80] =	267.4616394043
x[81] =	267.3371276855	y[81] =	272.3781738281
x[82] =	266.9151000977	y[82] =	271.9751586914
x[83] =	271.6332702637	y[83] =	270.5668640137
x[84] =	268.2593078613	y[84] =	268.3897399902
x[85] =	268.4362487793	y[85] =	269.7589111328
x[86] =	267.9328002930	y[86] =	271.5138549805
x[87] =	271.2909545898	y[87] =	272.1745605469
x[88] =	269.0054321289	y[88] =	272.1026611328
x[89] =	270.1526794434	y[89] =	268.9882812500
x[90] =	263.8572082520	y[90] =	274.7151794434
x[91] =	265.2599182129	y[91] =	271.5104980469
x[92] =	271.0217285156	y[92] =	266.5699462891
x[93] =	267.6752014160	y[93] =	274.0068969727
x[94] =	265.1853637695	y[94] =	271.4517822266
x[95] =	269.4737548828	y[95] =	268.9801635742
x[96] =	270.2192687988	y[96] =	266.2698364258
x[97] =	268.3923339844	y[97] =	268.8155822754
x[98] =	273.4592895508	y[98] =	269.0731201172
x[99] =	267.4300231934	y[99] =	271.1868896484

	Real	Imag
Mean	269.7486669922	269.9851544189
Std. Dev.	2.2155283971	2.3963705533
Minimum	263.8572082520	262.9579467773
Maximum	274.5322570801	274.7530517578

Standard vs. Direct DFT, 16-Point

x[0] =	272.7552795410	y[0] =	268.9796142578
x[1] =	272.1343078613	y[1] =	270.4200439453
x[2] =	272.9894104004	y[2] =	267.0741882324
x[3] =	269.5061035156	y[3] =	269.7828369141
x[4] =	271.7872009277	y[4] =	269.5117492676
x[5] =	268.9995422363	y[5] =	267.9895629883
x[6] =	269.0440979004	y[6] =	271.7550048828
x[7] =	272.5062866211	y[7] =	270.9828491211
x[8] =	271.4386291504	y[8] =	265.9161682129
x[9] =	268.2858276367	y[9] =	268.1496582031
x[10] =	269.6635437012	y[10] =	269.8590087891
x[11] =	269.8771972656	y[11] =	269.0490112305
x[12] =	272.2474060059	y[12] =	265.7119445801
x[13] =	269.8631286621	y[13] =	268.6048583984
x[14] =	273.3859863281	y[14] =	270.4118957520
x[15] =	269.6500244141	y[15] =	270.3233337402
x[16] =	269.4855346680	y[16] =	268.4541625977
x[17] =	266.8340759277	y[17] =	269.9486694336
x[18] =	268.6825256348	y[18] =	272.4471130371
x[19] =	265.6889953613	y[19] =	268.9368591309
x[20] =	268.6466369629	y[20] =	271.1831054688
x[21] =	271.3880004883	y[21] =	267.9326171875
x[22] =	269.5439453125	y[22] =	270.4999389648
x[23] =	264.3802795410	y[23] =	273.8593750000
x[24] =	271.4773559570	y[24] =	271.7287597656
x[25] =	268.1225280762	y[25] =	270.9976501465
x[26] =	265.7618713379	y[26] =	273.3869323730
x[27] =	268.4483642578	y[27] =	273.7132568359
x[28] =	269.3742065430	y[28] =	273.4106140137
x[29] =	269.9822692871	y[29] =	270.0431823730
x[30] =	267.9257202148	y[30] =	268.2225036621
x[31] =	266.8103637695	y[31] =	270.1790771484
x[32] =	266.4388122559	y[32] =	269.7553710938
x[33] =	266.2378540039	y[33] =	274.9422912598
x[34] =	271.0850219727	y[34] =	270.1690673828
x[35] =	272.6909790039	y[35] =	270.7511901855
x[36] =	268.4923706055	y[36] =	270.8391723633
x[37] =	271.4857482910	y[37] =	271.5530395508
x[38] =	272.9253845215	y[38] =	270.3072204590
x[39] =	268.4839477539	y[39] =	269.5484313965

x[40] =	268.3330383301	y[40] =	269.7852783203
x[41] =	269.1254272461	y[41] =	269.3233947754
x[42] =	268.8452148438	y[42] =	268.4582214355
x[43] =	273.3160705566	y[43] =	270.5591125488
x[44] =	268.9049377441	y[44] =	266.7042541504
x[45] =	271.9574890137	y[45] =	268.7884826660
x[46] =	260.9253234863	y[46] =	277.2808227539
x[47] =	269.6184692383	y[47] =	273.6904296875
x[48] =	270.7478637695	y[48] =	274.3979187012
x[49] =	271.6245727539	y[49] =	271.3009643555
x[50] =	269.8124389648	y[50] =	271.8434143066
x[51] =	272.3414001465	y[51] =	273.0943298340
x[52] =	272.1879882813	y[52] =	269.4304199219
x[53] =	265.5771789551	y[53] =	272.2980651855
x[54] =	268.9840393066	y[54] =	271.2325439453
x[55] =	271.3327636719	y[55] =	269.8065490723
x[56] =	268.9931335449	y[56] =	270.9717407227
x[57] =	269.2901000977	y[57] =	268.7641601563
x[58] =	269.7671203613	y[58] =	271.5994873047
x[59] =	267.4548950195	y[59] =	271.3090209961
x[60] =	270.0095214844	y[60] =	268.9137268066
x[61] =	275.0334472656	y[61] =	266.4119567871
x[62] =	273.6075744629	y[62] =	270.5734863281
x[63] =	270.5801696777	y[63] =	268.4373779297
x[64] =	264.6386108398	y[64] =	273.6230773926
x[65] =	273.3971557617	y[65] =	268.9710693359
x[66] =	274.4778137207	y[66] =	266.6929626465
x[67] =	270.4531860352	y[67] =	272.0224609375
x[68] =	268.2864990234	y[68] =	271.1570739746
x[69] =	271.3478088379	y[69] =	270.3151550293
x[70] =	269.1685485840	y[70] =	268.4387512207
x[71] =	266.7461242676	y[71] =	273.5340270996
x[72] =	272.3862915039	y[72] =	269.6380920410
x[73] =	273.0481262207	y[73] =	265.4359741211
x[74] =	268.3798828125	y[74] =	270.9328918457
x[75] =	269.9814453125	y[75] =	268.4600219727
x[76] =	271.0657043457	y[76] =	269.3445739746
x[77] =	269.5472106934	y[77] =	269.7560119629
x[78] =	272.6813964844	y[78] =	269.8146057129
x[79] =	269.6702575684	y[79] =	271.4771423340

x[80] =	275.4037780762	y[80] =	266.9851684570
x[81] =	271.6023559570	y[81] =	268.7348327637
x[82] =	270.4073486328	y[82] =	268.1242065430
x[83] =	271.3636474609	y[83] =	266.9815673828
x[84] =	271.1292419434	y[84] =	269.1577758789
x[85] =	271.0298156738	y[85] =	267.4655456543
x[86] =	271.7021179199	y[86] =	269.4682617188
x[87] =	271.2106933594	y[87] =	269.5105590820
x[88] =	271.1216430664	y[88] =	268.2117004395
x[89] =	270.6502685547	y[89] =	269.2516174316
x[90] =	268.8835449219	y[90] =	267.9707946777
x[91] =	271.6461181641	y[91] =	260.9583740234
x[92] =	268.6901855469	y[92] =	273.1594543457
x[93] =	269.4382324219	y[93] =	268.0898742676
x[94] =	270.8758544922	y[94] =	266.6210632324
x[95] =	272.0344543457	y[95] =	266.7901000977
x[96] =	270.6707763672	y[96] =	269.9903564453
x[97] =	267.9556884766	y[97] =	268.6535034180
x[98] =	269.1480712891	y[98] =	271.4492492676
x[99] =	267.2027587891	y[99] =	269.9615783691

	Real	Imag
Mean	270.0033959961	269.8945596313
Std. Dev.	2.3937631598	2.3372463719
Minimum	260.9253234863	260.9583740234
Maximum	275.4037780762	277.2808227539

Standard vs. Direct DFT, 17-Point

x[0] =	269.8819885254	y[0] =	270.7600402832
x[1] =	269.4014282227	y[1] =	272.1129150391
x[2] =	272.0111999512	y[2] =	272.0293884277
x[3] =	268.9029846191	y[3] =	271.9183654785
x[4] =	268.4132995605	y[4] =	268.0768432617
x[5] =	269.8236389160	y[5] =	268.0231933594
x[6] =	273.5191955566	y[6] =	266.7305603027
x[7] =	269.6670532227	y[7] =	269.5509643555
x[8] =	270.8864440918	y[8] =	269.4957885742
x[9] =	270.0984497070	y[9] =	269.8625183105
x[10] =	270.8583374023	y[10] =	270.9628601074
x[11] =	270.8510742188	y[11] =	268.5837097168
x[12] =	270.6906127930	y[12] =	268.6798706055
x[13] =	272.8601684570	y[13] =	273.6946105957
x[14] =	273.4621276855	y[14] =	266.7900085449
x[15] =	268.5950317383	y[15] =	269.5240173340
x[16] =	271.8672485352	y[16] =	264.9834899902
x[17] =	271.0851135254	y[17] =	269.1212768555
x[18] =	266.5239257813	y[18] =	274.0615844727
x[19] =	271.5162048340	y[19] =	270.4394531250
x[20] =	268.3395690918	y[20] =	268.6579895020
x[21] =	267.7672424316	y[21] =	269.1622924805
x[22] =	268.6538391113	y[22] =	270.5011596680
x[23] =	269.3810424805	y[23] =	271.2490844727
x[24] =	270.8980407715	y[24] =	267.3539123535
x[25] =	270.3724365234	y[25] =	268.2624511719
x[26] =	269.9035949707	y[26] =	269.6186218262
x[27] =	269.1344909668	y[27] =	267.0343017578
x[28] =	265.9377746582	y[28] =	270.1463012695
x[29] =	271.7009887695	y[29] =	267.6523132324
x[30] =	270.4732971191	y[30] =	267.3661193848
x[31] =	272.6469726563	y[31] =	272.1127929688
x[32] =	268.8228149414	y[32] =	269.0408935547
x[33] =	273.7104492188	y[33] =	273.3024291992
x[34] =	269.1943054199	y[34] =	269.5977478027
x[35] =	271.5632019043	y[35] =	273.6456604004
x[36] =	269.4310913086	y[36] =	269.6079101563
x[37] =	268.1188354492	y[37] =	267.0449218750
x[38] =	269.1134338379	y[38] =	267.6660461426
x[39] =	268.6231079102	y[39] =	272.7576904297

x[40] =	271.0748596191	y[40] =	264.4887695313
x[41] =	265.2482604980	y[41] =	270.3030700684
x[42] =	265.4088439941	y[42] =	270.5873718262
x[43] =	269.1730346680	y[43] =	264.2818603516
x[44] =	272.2053222656	y[44] =	267.3887329102
x[45] =	267.9991455078	y[45] =	274.9462280273
x[46] =	270.1383361816	y[46] =	274.0894775391
x[47] =	271.5760498047	y[47] =	270.5509338379
x[48] =	270.5822143555	y[48] =	271.3222656250
x[49] =	273.2935485840	y[49] =	268.9748535156
x[50] =	269.3102416992	y[50] =	268.2997741699
x[51] =	270.2906494141	y[51] =	271.1800842285
x[52] =	268.3794555664	y[52] =	269.4108886719
x[53] =	267.1316528320	y[53] =	269.9113769531
x[54] =	270.3419189453	y[54] =	270.9147949219
x[55] =	273.1160583496	y[55] =	265.6920471191
x[56] =	271.4096984863	y[56] =	271.8542175293
x[57] =	268.4126892090	y[57] =	273.2379760742
x[58] =	265.3532409668	y[58] =	274.1091918945
x[59] =	269.4324645996	y[59] =	270.6475219727
x[60] =	269.1028747559	y[60] =	269.0531616211
x[61] =	272.4880065918	y[61] =	269.4778442383
x[62] =	271.4067993164	y[62] =	272.0108032227
x[63] =	271.1753845215	y[63] =	271.3197631836
x[64] =	270.3529968262	y[64] =	269.5959472656
x[65] =	272.8681335449	y[65] =	267.4597167969
x[66] =	269.1588439941	y[66] =	265.5984497070
x[67] =	271.3894653320	y[67] =	267.6537170410
x[68] =	270.2159729004	y[68] =	267.0043334961
x[69] =	266.4914855957	y[69] =	271.4144897461
x[70] =	267.1986083984	y[70] =	270.6966247559
x[71] =	266.6242675781	y[71] =	268.6539916992
x[72] =	271.1471557617	y[72] =	268.8063354492
x[73] =	268.0986633301	y[73] =	272.5935058594
x[74] =	268.4238891602	y[74] =	268.6405029297
x[75] =	269.7712707520	y[75] =	269.3260803223
x[76] =	267.1796264648	y[76] =	266.6194763184
x[77] =	271.7876892090	y[77] =	265.1469116211
x[78] =	271.5567626953	y[78] =	269.6063537598
x[79] =	270.0137939453	y[79] =	269.1286315918

x[80] =	265.9388122559	y[80] =	270.9612121582
x[81] =	268.4763488770	y[81] =	273.0566406250
x[82] =	268.9838867188	y[82] =	274.3491821289
x[83] =	269.2582397461	y[83] =	269.8005065918
x[84] =	269.4443664551	y[84] =	273.0825805664
x[85] =	270.5592346191	y[85] =	261.4719238281
x[86] =	268.5687866211	y[86] =	267.5593872070
x[87] =	272.0215148926	y[87] =	270.6485290527
x[88] =	272.9148864746	y[88] =	262.5549926758
x[89] =	267.7056884766	y[89] =	269.8771667480
x[90] =	272.4746093750	y[90] =	267.7489624023
x[91] =	273.3935546875	y[91] =	273.5779113770
x[92] =	266.5085144043	y[92] =	273.7836914063
x[93] =	264.3068847656	y[93] =	272.3791809082
x[94] =	273.5789794922	y[94] =	267.7886047363
x[95] =	271.1185302734	y[95] =	268.3768310547
x[96] =	263.3368225098	y[96] =	271.6583862305
x[97] =	270.5970153809	y[97] =	269.5770874023
x[98] =	266.7500610352	y[98] =	274.2363281250
x[99] =	270.9354248047	y[99] =	270.1430358887

	Real	Imag
Mean	269.7790359497	269.7181231689
Std. Dev.	2.2153467436	2.6401404206
Minimum	263.3368225098	261.4719238281
Maximum	273.7104492188	274.9462280273

x[0] =	176.9281005859	y[0] =	176.0798339844
x[1] =	175.8159942627	y[1] =	176.5415496826
x[2] =	175.2160034180	y[2] =	175.1606445313
x[3] =	175.4710540771	y[3] =	176.7188262939
x[4] =	176.0864715576	y[4] =	175.0678100586
x[5] =	175.6123352051	y[5] =	175.8395233154
x[6] =	175.0600128174	y[6] =	176.5708312988
x[7] =	176.3147125244	y[7] =	176.7072753906
x[8] =	175.8805694580	y[8] =	175.4317932129
x[9] =	175.9056701660	y[9] =	175.6567382813
x[10] =	175.5078125000	y[10] =	176.0266265869
x[11] =	176.0314483643	y[11] =	176.3258056641
x[12] =	176.7476043701	y[12] =	176.8772735596
x[13] =	175.7143096924	y[13] =	175.5060729980
x[14] =	175.2527618408	y[14] =	175.3540496826
x[15] =	175.5321197510	y[15] =	175.8341979980
x[16] =	175.7600402832	y[16] =	176.0635375977
x[17] =	176.2900390625	y[17] =	176.5039672852
x[18] =	175.4552764893	y[18] =	177.0667724609
x[19] =	176.2388153076	y[19] =	176.0251159668
x[20] =	175.3146209717	y[20] =	177.0319824219
x[21] =	175.7579803467	y[21] =	175.4255218506
x[22] =	175.3465728760	y[22] =	176.9929962158
x[23] =	177.0458984375	y[23] =	176.3945007324
x[24] =	176.5691680908	y[24] =	174.9557189941
x[25] =	175.6906127930	y[25] =	176.5028533936
x[26] =	174.4704895020	y[26] =	176.0759582520
x[27] =	176.4285430908	y[27] =	175.3894348145
x[28] =	176.7745819092	y[28] =	175.2816925049
x[29] =	175.0116577148	y[29] =	175.5447692871
x[30] =	176.6018676758	y[30] =	175.9234008789
x[31] =	175.1762542725	y[31] =	175.9215393066
x[32] =	176.3634948730	y[32] =	175.5064239502
x[33] =	175.9099731445	y[33] =	176.2240753174
x[34] =	176.0561370850	y[34] =	175.6609039307
x[35] =	176.2507324219	y[35] =	175.1939849854
x[36] =	176.3163452148	y[36] =	175.3961791992
x[37] =	175.6886749268	y[37] =	176.1816101074
x[38] =	175.1196289063	y[38] =	176.7167358398
x[39] =	176.7915344238	y[39] =	177.3120269775

x[40] =	175.6382751465	y[40] =	176.2698364258
x[41] =	176.2331542969	y[41] =	176.1804504395
x[42] =	176.0434112549	y[42] =	175.7587432861
x[43] =	175.8974609375	y[43] =	176.3019104004
x[44] =	174.9062652588	y[44] =	176.3957519531
x[45] =	175.9091339111	y[45] =	175.9953918457
x[46] =	176.9732666016	y[46] =	175.7868957520
x[47] =	176.2299804688	y[47] =	176.1300811768
x[48] =	176.7227020264	y[48] =	174.9942169189
x[49] =	175.7549285889	y[49] =	174.9685363770

	Real	Imag
Mean	175.9162899780	175.9954473877
Std. Dev.	0.5917510921	0.6040244160
Minimum	174.4704895020	174.9557189941
Maximum	177.0458984375	177.3120269775

x[0] =	176.9676361084	y[0] =	175.2451019287
x[1] =	174.5153198242	y[1] =	176.2055664063
x[2] =	174.5861968994	y[2] =	174.6942901611
x[3] =	175.7495880127	y[3] =	175.2100830078
x[4] =	175.2971343994	y[4] =	175.2332305908
x[5] =	174.5655364990	y[5] =	174.9668121338
x[6] =	176.4736938477	y[6] =	175.4824066162
x[7] =	175.0034332275	y[7] =	175.5998382568
x[8] =	176.3172302246	y[8] =	174.4532775879
x[9] =	175.8294982910	y[9] =	175.4020233154
x[10] =	175.3798675537	y[10] =	175.9258575439
x[11] =	175.6436309814	y[11] =	176.3606414795
x[12] =	175.7827758789	y[12] =	175.9241943359
x[13] =	175.2949218750	y[13] =	175.1103210449
x[14] =	174.2272186279	y[14] =	175.7443389893
x[15] =	175.2727966309	y[15] =	175.5301971436
x[16] =	175.9711914063	y[16] =	175.7611999512
x[17] =	174.3645629883	y[17] =	177.0066986084
x[18] =	175.1951751709	y[18] =	176.0203552246
x[19] =	175.7872314453	y[19] =	176.1752777100
x[20] =	174.9986877441	y[20] =	176.0463256836
x[21] =	174.9064483643	y[21] =	176.6525115967
x[22] =	175.4567718506	y[22] =	174.6132049561
x[23] =	175.2766571045	y[23] =	176.2723083496
x[24] =	175.6004486084	y[24] =	174.9115295410
x[25] =	175.7740478516	y[25] =	176.3608856201
x[26] =	176.3112030029	y[26] =	174.2060699463
x[27] =	174.8421936035	y[27] =	175.5248870850
x[28] =	176.3071899414	y[28] =	175.9502410889
x[29] =	174.1696319580	y[29] =	176.0451202393
x[30] =	175.4562683105	y[30] =	175.7846221924
x[31] =	175.9324798584	y[31] =	175.1319427490
x[32] =	175.7597045898	y[32] =	175.0276641846
x[33] =	176.0273742676	y[33] =	175.1878051758
x[34] =	174.3921966553	y[34] =	176.3043670654
x[35] =	175.7931365967	y[35] =	175.4776916504
x[36] =	175.6822814941	y[36] =	176.1872253418
x[37] =	175.6284790039	y[37] =	175.6081848145
x[38] =	175.1247863770	y[38] =	175.9377441406
x[39] =	176.3980560303	y[39] =	174.8244781494

x[40] =	175.2262725830	y[40] =	174.9261322021
x[41] =	175.0415039063	y[41] =	176.4279937744
x[42] =	176.1199493408	y[42] =	175.8008728027
x[43] =	175.5458526611	y[43] =	175.7980651855
x[44] =	175.7533569336	y[44] =	176.4787597656
x[45] =	175.9052429199	y[45] =	175.2508544922
x[46] =	175.4692535400	y[46] =	174.8632507324
x[47] =	175.8170623779	y[47] =	175.9199829102
x[48] =	175.5901489258	y[48] =	176.1115264893
x[49] =	175.8327026367	y[49] =	174.9668426514

	Real	Imag
Mean	175.4872805786	175.6130160522
Std. Dev.	0.6166290744	0.6181964662
Minimum	174.1696319580	174.2060699463
Maximum	176.9676361084	177.0066986084

x[0] =	175.3172760010	y[0] =	175.6668548584
x[1] =	175.2134399414	y[1] =	175.1342163086
x[2] =	174.6766662598	y[2] =	175.0067749023
x[3] =	175.8343658447	y[3] =	174.6544647217
x[4] =	174.6794281006	y[4] =	175.0544738770
x[5] =	174.7781066895	y[5] =	174.3569335938
x[6] =	175.7173919678	y[6] =	175.3242187500
x[7] =	174.7194671631	y[7] =	174.3413238525
x[8] =	174.6242370605	y[8] =	174.6977691650
x[9] =	174.6361236572	y[9] =	174.6449890137
x[10] =	174.8958282471	y[10] =	176.0199432373
x[11] =	175.3851013184	y[11] =	175.5915069580
x[12] =	175.1615142822	y[12] =	174.6331939697
x[13] =	173.5044860840	y[13] =	175.2667846680
x[14] =	175.7932128906	y[14] =	173.9664916992
x[15] =	175.0761718750	y[15] =	175.5431213379
x[16] =	174.8210754395	y[16] =	174.4515075684
x[17] =	175.1059112549	y[17] =	175.4931030273
x[18] =	175.3726654053	y[18] =	174.6825256348
x[19] =	175.6221313477	y[19] =	174.6659088135
x[20] =	176.3201446533	y[20] =	174.7219696045
x[21] =	174.5650024414	y[21] =	174.1995239258
x[22] =	174.6980743408	y[22] =	175.3710479736
x[23] =	173.9389953613	y[23] =	174.4743041992
x[24] =	174.4600982666	y[24] =	175.2648773193
x[25] =	175.1066436768	y[25] =	174.6895141602
x[26] =	174.3740081787	y[26] =	176.3616333008
x[27] =	174.1670227051	y[27] =	176.0292053223
x[28] =	175.9723510742	y[28] =	174.9096527100
x[29] =	175.1481628418	y[29] =	174.7274627686
x[30] =	174.8263854980	y[30] =	174.9681091309
x[31] =	175.6766510010	y[31] =	175.3684082031
x[32] =	174.6771087646	y[32] =	175.4441986084
x[33] =	175.5671997070	y[33] =	174.1645965576
x[34] =	176.4279022217	y[34] =	175.6058502197
x[35] =	175.4553070068	y[35] =	175.6769714355
x[36] =	175.3783569336	y[36] =	174.9795837402
x[37] =	175.1912384033	y[37] =	174.8423309326
x[38] =	175.2342376709	y[38] =	174.4134216309
x[39] =	174.3762969971	y[39] =	175.0219268799

x[40] =	175.2068023682	y[40] =	174.7622070313
x[41] =	175.3159332275	y[41] =	175.4618225098
x[42] =	175.1643066406	y[42] =	174.8082733154
x[43] =	175.0156707764	y[43] =	174.7454071045
x[44] =	175.5962066650	y[44] =	175.1154479980
x[45] =	175.4506225586	y[45] =	175.8812713623
x[46] =	174.8144683838	y[46] =	174.0914154053
x[47] =	176.2023773193	y[47] =	174.7069396973
x[48] =	175.1609954834	y[48] =	174.3604431152
x[49] =	175.0452728271	y[49] =	174.8707122803

	Real	Imag
Mean	175.1093688965	174.9846926880
Std. Dev.	0.5806204977	0.5435851097
Minimum	173.5044860840	173.9664916992
Maximum	176.4279022217	176.3616333008

x[0] =	126.9658889771	y[0] =	128.6595001221
x[1] =	127.1139373779	y[1] =	125.0317230225
x[2] =	124.3147811890	y[2] =	130.9378356934
x[3] =	126.6682891846	y[3] =	123.7885589600
x[4] =	128.8446350098	y[4] =	127.2956924438
x[5] =	125.8834075928	y[5] =	124.4208068848
x[6] =	126.0367584229	y[6] =	127.0050735474
x[7] =	125.7816925049	y[7] =	126.9526977539
x[8] =	124.3878250122	y[8] =	125.0147018433
x[9] =	123.5895309448	y[9] =	125.9368896484
x[10] =	127.3346557617	y[10] =	125.1703872681
x[11] =	123.9044952393	y[11] =	130.2939453125
x[12] =	124.7136917114	y[12] =	127.7754821777
x[13] =	126.8959732056	y[13] =	123.9404067993
x[14] =	125.4262542725	y[14] =	125.1703872681
x[15] =	126.2954330444	y[15] =	124.6974182129
x[16] =	126.7282409668	y[16] =	125.8010559082
x[17] =	126.1213912964	y[17] =	123.6008453369
x[18] =	125.6085433960	y[18] =	126.4449844360
x[19] =	125.6101455688	y[19] =	127.8805313110
x[20] =	122.3669967651	y[20] =	126.0266342163
x[21] =	125.6510925293	y[21] =	129.1865692139
x[22] =	128.3963470459	y[22] =	125.2900390625
x[23] =	126.4536743164	y[23] =	124.4100952148
x[24] =	126.0329818726	y[24] =	125.9576416016
x[25] =	124.4752349854	y[25] =	127.6567687988
x[26] =	123.1487426758	y[26] =	126.0815734863
x[27] =	124.8630447388	y[27] =	125.0927429199
x[28] =	128.4418182373	y[28] =	125.2314453125
x[29] =	127.6404342651	y[29] =	127.5886688232
x[30] =	124.8626632690	y[30] =	128.0912170410
x[31] =	125.2672424316	y[31] =	124.3894500732
x[32] =	126.1647338867	y[32] =	129.5290985107
x[33] =	124.9801940918	y[33] =	126.7948989868
x[34] =	126.1622467041	y[34] =	129.0379638672
x[35] =	125.2439117432	y[35] =	127.1847610474
x[36] =	122.9684448242	y[36] =	129.4932861328
x[37] =	126.7451705933	y[37] =	125.9872894287
x[38] =	127.6056671143	y[38] =	125.6809234619
x[39] =	128.3607940674	y[39] =	127.5285491943

x[40] =	127.6127700806	y[40] =	123.8710327148
x[41] =	128.1640472412	y[41] =	124.5534667969
x[42] =	126.5611190796	y[42] =	128.2700653076
x[43] =	126.7883224487	y[43] =	128.5454101563
x[44] =	127.5197525024	y[44] =	128.5200958252
x[45] =	124.3297882080	y[45] =	126.2699432373
x[46] =	126.3443374634	y[46] =	128.9873199463
x[47] =	127.4374389648	y[47] =	125.6204376221
x[48] =	127.3305206299	y[48] =	128.5934906006
x[49] =	124.4188232422	y[49] =	126.6717681885
x[50] =	126.6728057861	y[50] =	126.5397720337
x[51] =	125.3947372437	y[51] =	126.8625488281
x[52] =	125.8040542603	y[52] =	127.7437286377
x[53] =	126.8680572510	y[53] =	128.9866333008
x[54] =	125.6404037476	y[54] =	125.2381591797
x[55] =	124.9454727173	y[55] =	126.1306076050
x[56] =	124.2371215820	y[56] =	126.6063919067
x[57] =	125.3613662720	y[57] =	124.4652938843
x[58] =	126.9989395142	y[58] =	127.6016235352
x[59] =	126.6637268066	y[59] =	127.5234756470
x[60] =	127.3334579468	y[60] =	121.6755905151
x[61] =	127.7255859375	y[61] =	126.1257476807
x[62] =	125.7728652954	y[62] =	123.1785507202
x[63] =	124.9451065063	y[63] =	127.4509277344
x[64] =	127.8579711914	y[64] =	120.5933532715
x[65] =	126.3562469482	y[65] =	124.7063980103
x[66] =	124.9039688110	y[66] =	129.1341094971
x[67] =	125.9150085449	y[67] =	125.2517395020
x[68] =	124.8955612183	y[68] =	124.9134750366
x[69] =	125.8209228516	y[69] =	126.7678756714
x[70] =	128.2415161133	y[70] =	126.4128189087
x[71] =	126.5546951294	y[71] =	127.0057525635
x[72] =	127.0476074219	y[72] =	124.6814117432
x[73] =	127.4869155884	y[73] =	125.4552459717
x[74] =	126.9031829834	y[74] =	125.3369293213
x[75] =	126.5438232422	y[75] =	130.2485656738
x[76] =	125.9665832520	y[76] =	126.6366653442
x[77] =	128.6787719727	y[77] =	127.0724945068
x[78] =	125.3847656250	y[78] =	127.0151290894
x[79] =	121.6201248169	y[79] =	127.9532546997

x[80] =	125.8070831299	y[80] =	123.7141952515
x[81] =	126.6072845459	y[81] =	127.1912384033
x[82] =	125.6616973877	y[82] =	127.1672515869
x[83] =	127.5524520874	y[83] =	125.1179733276
x[84] =	124.9084472656	y[84] =	125.3907165527
x[85] =	124.3799057007	y[85] =	124.6561660767
x[86] =	124.3708190918	y[86] =	124.7729339600
x[87] =	125.6674728394	y[87] =	127.2672195435
x[88] =	126.4710006714	y[88] =	126.4706115723
x[89] =	124.8405914307	y[89] =	125.7980117798
x[90] =	125.8368682861	y[90] =	127.0739288330
x[91] =	123.6946792603	y[91] =	126.4280700684
x[92] =	125.9281616211	y[92] =	123.9990463257
x[93] =	124.6118316650	y[93] =	128.0190582275
x[94] =	124.6166687012	y[94] =	127.2861557007
x[95] =	126.3000869751	y[95] =	126.4723205566
x[96] =	125.5646057129	y[96] =	125.7940750122
x[97] =	125.2095794678	y[97] =	126.0042343140
x[98] =	125.4065628052	y[98] =	127.6092834473
x[99] =	125.3776702881	y[99] =	126.1830444336

	Real	Imag
Mean	125.9391876221	126.3969137573
Std. Dev.	1.3873679241	1.8051794246
Minimum	121.6201248169	120.5933532715
Maximum	128.8446350098	130.9378356934

x[0] =	131.6860504150	y[0] =	128.5588684082
x[1] =	126.2768936157	y[1] =	126.5050506592
x[2] =	129.6031494141	y[2] =	125.9955062866
x[3] =	127.3920211792	y[3] =	125.4859466553
x[4] =	130.0837402344	y[4] =	128.9797515869
x[5] =	129.1377258301	y[5] =	128.2185974121
x[6] =	127.2726821899	y[6] =	127.4755172729
x[7] =	126.6516876221	y[7] =	127.4683837891
x[8] =	128.4108276367	y[8] =	125.9150314331
x[9] =	128.5280303955	y[9] =	127.2252960205
x[10] =	125.5002517700	y[10] =	133.6931915283
x[11] =	127.8747558594	y[11] =	127.8566436768
x[12] =	130.7615966797	y[12] =	128.2600708008
x[13] =	124.7099914551	y[13] =	126.8356246948
x[14] =	128.4129943848	y[14] =	126.2015075684
x[15] =	126.9171371460	y[15] =	128.9027557373
x[16] =	131.8934173584	y[16] =	125.0395584106
x[17] =	127.7030563354	y[17] =	128.3233032227
x[18] =	128.4555358887	y[18] =	127.7306365967
x[19] =	125.1961898804	y[19] =	128.1782684326
x[20] =	127.3693542480	y[20] =	131.4153137207
x[21] =	126.4642181396	y[21] =	126.6883087158
x[22] =	129.0205841064	y[22] =	126.4337387085
x[23] =	124.9503555298	y[23] =	132.9751434326
x[24] =	125.8017578125	y[24] =	127.3063812256
x[25] =	126.2557220459	y[25] =	130.3205108643
x[26] =	124.4999465942	y[26] =	127.4365386963
x[27] =	125.2276916504	y[27] =	129.3834533691
x[28] =	126.0257110596	y[28] =	130.0014343262
x[29] =	125.9543762207	y[29] =	127.5649948120
x[30] =	126.1363525391	y[30] =	128.7181854248
x[31] =	123.9893798828	y[31] =	129.1547698975
x[32] =	127.2222442627	y[32] =	127.8531951904
x[33] =	123.7135543823	y[33] =	130.2106018066
x[34] =	126.2118682861	y[34] =	128.7924041748
x[35] =	128.4245300293	y[35] =	129.1282501221
x[36] =	125.4156341553	y[36] =	127.9233169556
x[37] =	128.2499237061	y[37] =	127.1410751343
x[38] =	128.5164947510	y[38] =	125.4722213745
x[39] =	127.4740142822	y[39] =	128.3810577393

x[40] =	128.7833862305	y[40] =	127.5596847534
x[41] =	127.2169799805	y[41] =	129.6949310303
x[42] =	126.2682266235	y[42] =	128.2455596924
x[43] =	126.7479629517	y[43] =	127.9707107544
x[44] =	129.6607360840	y[44] =	127.3498916626
x[45] =	130.4431762695	y[45] =	127.9753112793
x[46] =	119.8516998291	y[46] =	129.3677520752
x[47] =	128.3192596436	y[47] =	128.5295715332
x[48] =	125.9970321655	y[48] =	128.0255126953
x[49] =	127.9612731934	y[49] =	128.1005249023
x[50] =	128.5718841553	y[50] =	129.4289245605
x[51] =	127.5533065796	y[51] =	128.1962890625
x[52] =	127.1122436523	y[52] =	125.2886352539
x[53] =	124.5508193970	y[53] =	128.9994506836
x[54] =	124.2272644043	y[54] =	128.7111511230
x[55] =	125.7153244019	y[55] =	126.2460174561
x[56] =	126.5410690308	y[56] =	130.8160705566
x[57] =	127.0897369385	y[57] =	126.3504333496
x[58] =	125.0207595825	y[58] =	128.0140686035
x[59] =	129.4002075195	y[59] =	128.8536682129
x[60] =	125.3560104370	y[60] =	127.4634628296
x[61] =	131.5325012207	y[61] =	126.8283767700
x[62] =	128.2161865234	y[62] =	127.7228469849
x[63] =	128.2512817383	y[63] =	123.9412612915
x[64] =	124.1138458252	y[64] =	132.8595428467
x[65] =	128.7828826904	y[65] =	128.0684661865
x[66] =	132.4055175781	y[66] =	124.2985458374
x[67] =	128.9963684082	y[67] =	127.7581024170
x[68] =	127.6422271729	y[68] =	130.9707946777
x[69] =	127.7262344360	y[69] =	127.8050079346
x[70] =	127.9575729370	y[70] =	126.2689437866
x[71] =	127.3208770752	y[71] =	127.8360519409
x[72] =	128.9340209961	y[72] =	126.6705780029
x[73] =	127.7839202881	y[73] =	125.0569152832
x[74] =	126.7753448486	y[74] =	126.6033554077
x[75] =	127.6877136230	y[75] =	126.3947296143
x[76] =	126.5776748657	y[76] =	127.2083358765
x[77] =	128.0478057861	y[77] =	126.9995956421
x[78] =	130.2034301758	y[78] =	128.5143127441
x[79] =	126.2883758545	y[79] =	125.8590240479

x[80] =	129.0808410645	y[80] =	124.5049972534
x[81] =	131.4768829346	y[81] =	125.0545959473
x[82] =	128.4235992432	y[82] =	126.4251098633
x[83] =	129.9603576660	y[83] =	125.8778915405
x[84] =	129.2503814697	y[84] =	130.3973236084
x[85] =	126.9571685791	y[85] =	125.9371643066
x[86] =	126.4190673828	y[86] =	128.4761199951
x[87] =	127.9010543823	y[87] =	124.4585342407
x[88] =	128.4527130127	y[88] =	127.4746475220
x[89] =	126.0189590454	y[89] =	129.0239257813
x[90] =	127.2768096924	y[90] =	125.2287292480
x[91] =	130.0948028564	y[91] =	119.5501251221
x[92] =	125.2066574097	y[92] =	128.3637237549
x[93] =	132.5957794189	y[93] =	127.0037918091
x[94] =	127.9331283569	y[94] =	126.1179885864
x[95] =	130.7555236816	y[95] =	126.5633239746
x[96] =	128.2622222900	y[96] =	128.1271057129
x[97] =	127.0912322998	y[97] =	125.1954727173
x[98] =	126.6822128296	y[98] =	128.4990844727
x[99] =	129.8872528076	y[99] =	126.6480636597

	Real	Imag
Mean	127.5675023651	127.6293053436
Std. Dev.	2.0868934883	1.9825243937
Minimum	119.8516998291	119.5501251221
Maximum	132.5957794189	133.6931915283

x[0] =	121.4185028076	y[0] =	123.7153015137
x[1] =	120.8895874023	y[1] =	120.1140975952
x[2] =	122.3241653442	y[2] =	118.8880996704
x[3] =	122.3515472412	y[3] =	123.6122055054
x[4] =	121.8156280518	y[4] =	120.1766967773
x[5] =	120.1210479736	y[5] =	119.1418151855
x[6] =	122.2692947388	y[6] =	119.3881454468
x[7] =	119.8981628418	y[7] =	120.5820312500
x[8] =	120.7996978760	y[8] =	120.2281799316
x[9] =	119.6428146362	y[9] =	121.8146896362
x[10] =	121.3981552124	y[10] =	120.2237091064
x[11] =	121.5536193848	y[11] =	120.7965164185
x[12] =	119.6506652832	y[12] =	119.2724609375
x[13] =	119.6428375244	y[13] =	120.6252441406
x[14] =	124.6930618286	y[14] =	118.6544799805
x[15] =	120.1872253418	y[15] =	120.4623184204
x[16] =	123.8273925781	y[16] =	115.8238830566
x[17] =	122.0793380737	y[17] =	120.7008056641
x[18] =	117.1603088379	y[18] =	122.6836547852
x[19] =	119.4157943726	y[19] =	120.7734298706
x[20] =	119.6562957764	y[20] =	121.8309707642
x[21] =	119.9624023438	y[21] =	119.7293319702
x[22] =	119.4375228882	y[22] =	122.8455047607
x[23] =	120.8168640137	y[23] =	121.1245498657
x[24] =	121.7880325317	y[24] =	120.8520278931
x[25] =	121.4904479980	y[25] =	121.2198104858
x[26] =	120.1587295532	y[26] =	122.2774047852
x[27] =	119.0045394897	y[27] =	119.2919616699
x[28] =	117.8995742798	y[28] =	122.7669525146
x[29] =	120.6625213623	y[29] =	120.0256042480
x[30] =	120.9794235229	y[30] =	120.6208343506
x[31] =	120.2607574463	y[31] =	120.6785888672
x[32] =	122.3692016602	y[32] =	118.1977386475
x[33] =	122.3941574097	y[33] =	120.3936767578
x[34] =	121.3694000244	y[34] =	121.8156204224
x[35] =	121.0974197388	y[35] =	121.8064270020
x[36] =	122.0919265747	y[36] =	120.8452835083
x[37] =	123.8083877563	y[37] =	122.1518173218
x[38] =	123.2192001343	y[38] =	121.4457931519
x[39] =	118.2859954834	y[39] =	122.6652297974

x[40] =	123.1239242554	y[40] =	117.5355682373
x[41] =	120.5650024414	y[41] =	121.1293640137
x[42] =	120.1896514893	y[42] =	124.0279541016
x[43] =	123.9672164917	y[43] =	120.1972045898
x[44] =	121.1884918213	y[44] =	119.4192123413
x[45] =	121.6450805664	y[45] =	121.3957824707
x[46] =	121.0041503906	y[46] =	121.1322631836
x[47] =	120.9823379517	y[47] =	120.6914672852
x[48] =	121.4914703369	y[48] =	121.1651840210
x[49] =	119.3569335938	y[49] =	118.9468841553
x[50] =	120.3198394775	y[50] =	118.2699966431
x[51] =	118.8627700806	y[51] =	121.3817672729
x[52] =	122.6042404175	y[52] =	120.2716064453
x[53] =	121.0762939453	y[53] =	123.8433456421
x[54] =	119.2531814575	y[54] =	122.4837036133
x[55] =	122.8533477783	y[55] =	120.4808044434
x[56] =	116.8894882202	y[56] =	118.2416763306
x[57] =	119.6834869385	y[57] =	122.6411972046
x[58] =	118.9494781494	y[58] =	124.0523681641
x[59] =	120.2644271851	y[59] =	118.7395095825
x[60] =	119.9867706299	y[60] =	120.0951919556
x[61] =	120.6990814209	y[61] =	120.8575286865
x[62] =	119.1787338257	y[62] =	119.3044891357
x[63] =	121.7292327881	y[63] =	122.4385299683
x[64] =	121.7205047607	y[64] =	120.2123947144
x[65] =	123.3698272705	y[65] =	119.6864852905
x[66] =	123.7213287354	y[66] =	120.7648468018
x[67] =	123.2931976318	y[67] =	119.0507125854
x[68] =	121.8409805298	y[68] =	118.7207031250
x[69] =	117.1487274170	y[69] =	121.1708145142
x[70] =	117.6536636353	y[70] =	125.0488510132
x[71] =	120.4309310913	y[71] =	120.7217025757
x[72] =	120.3559417725	y[72] =	123.0536117554
x[73] =	118.4559555054	y[73] =	122.7774734497
x[74] =	120.2510299683	y[74] =	121.5375595093
x[75] =	119.4515686035	y[75] =	122.0602645874
x[76] =	121.1188812256	y[76] =	118.9356307983
x[77] =	122.2433090210	y[77] =	121.0288772583
x[78] =	121.0973434448	y[78] =	122.0551071167
x[79] =	122.3748474121	y[79] =	120.4308776855

x[80] =	117.8436126709	y[80] =	120.8064880371
x[81] =	119.2608795166	y[81] =	121.7909698486
x[82] =	125.2556686401	y[82] =	122.3520889282
x[83] =	120.3285446167	y[83] =	121.0553054810
x[84] =	118.6132125854	y[84] =	121.0168457031
x[85] =	121.4505996704	y[85] =	113.7971115112
x[86] =	121.8219985962	y[86] =	121.8599853516
x[87] =	122.8372497559	y[87] =	117.4337539673
x[88] =	122.3030853271	y[88] =	114.8612136841
x[89] =	121.3166961670	y[89] =	120.8127822876
x[90] =	123.1911087036	y[90] =	119.7116546631
x[91] =	121.0095520020	y[91] =	118.8702926636
x[92] =	117.8431091309	y[92] =	121.0273895264
x[93] =	117.5745162964	y[93] =	121.1828002930
x[94] =	122.9829483032	y[94] =	118.9766159058
x[95] =	121.1741943359	y[95] =	121.1646194458
x[96] =	115.4362182617	y[96] =	122.9944000244
x[97] =	122.2885818481	y[97] =	121.9745864868
x[98] =	117.1472167969	y[98] =	123.5706710815
x[99] =	120.5247573853	y[99] =	119.1383819580

	Real	Imag
Mean	120.7848806763	120.7065936279
Std. Dev.	1.8456750456	1.8482199671
Minimum	115.4362182617	113.7971115112
Maximum	125.2556686401	125.0488510132

Vita

Captain Kent Taylor was born on 5 September 1957 in St. Johns, Arizona. He graduated from Blue Ridge High School, Lakeside, Arizona, in May 1975 and attended the University of Arizona, Tucson, Arizona, attaining the degree of Bachelor of Science in Electrical Engineering in December 1979. He received his commission from the USAF through the ROTC program on 21 December 1979 and was called to active duty in January 1980. After attending the Communications-Electronics Officer's School at Keesler AFB, Mississippi, Captain Taylor served as a radar evaluation officer at the 1954th Radar Evaluation Squadron (RADES), Hill AFB, Utah, from July 1980 to May 1984. During his tour with the RADES, he evaluated U.S. and allied air defense radar systems, surveyed prospective sites for new radar equipment, and assisted in writing specifications for future air traffic control radar systems. Captain Taylor entered the School of Engineering, Air Force Institute of Technology, in May 1984. He is a member of Eta Kappa Nu and the Institute of Electrical and Electronic Engineers.

Permanent Address: P.O. Box 728

Show Low, Arizona 85901

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GE/ENG/85D-47	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code): Bolling AFB, Washington D.C. 20332	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification): see block 19	
12. PERSONAL AUTHOR(S): Kent Taylor, B.S.E.E., Captain, USAF	
13a. TYPE OF REPORT: MS Thesis; 13b. TIME COVERED: FROM _____ TO _____; 14. DATE OF REPORT (Yr., Mo., Day): 1985 December; 15. PAGE COUNT: 218	
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Algorithms, Computerized Simulation, Discrete Fourier Transform, Fault Tolerant Computing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: ARCHITECTURE AND NUMERICAL ACCURACY OF HIGH-SPEED DFT PROCESSING SYSTEMS

Thesis Chairman: Richard W. Linderman, Captain, USAF
Assistant Professor of Electrical Engineering

Approved for Public release IAW AFR 190-1
LYNN E. WOLAVER 15 JAN 86
Dean for Research and Professional Development
Air Force Institute of Technology (AU)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Richard W. Linderman	22b. TELEPHONE NUMBER (Include Area Code): 513-255-6913; 22c. OFFICE SYMBOL: AFIT/ENG

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

Block 18 (continued): Integrated Circuits, VLSI

Block 19 (continued):

ABSTRACT

This research examines a very large-scale integrated (VLSI) circuit implementation of the Winograd and Good-Thomas algorithms for computing discrete Fourier Transforms (DFTs) with composite blocklengths. The theoretical background for calculating DFTs in general is developed, before the algorithms of interest are presented in detail. Once the validity of the algorithms is established, a VLSI architecture, which exploits the parallelism and pipelining inherent in the algorithms, is discussed. Winograd processors use either the small or the large Winograd DFT algorithm to compute DFTs with blocklengths of 15, 16, and 17. Longer blocklength DFTs (240, 255, 272, and 4080) are computed using a pipeline of Winograd processors, dual-port memories, and an interface processor; the pipeline uses the Good-Thomas Prime Factor Algorithm (PFA). Fault tolerance was included in the initial design of the VLSI architecture. Watchdog processors check both data and addresses of active Winograd processors, while parity checking circuits incorporated in the Winograd processors augment data memory error-correction coding (ECC).

The numerical accuracy of the VLSI circuit was determined using a software simulation. The signal-to-noise ratio(SNR) was used as the accuracy metric. The signal was the output of a standard module, which used double-precision arithmetic, while the noise was the difference between the standard and simulation modules. The simulation module used integer arithmetic to exactly mimic operation of the VLSI circuit. The outputs of the standard module were also compared with a direct evaluation of the DFT to verify that the standard module did compute a DFT. Results of the comparison between the standard and simulation modules for single-factor DFTs (i.e., 15, 16, and 17) indicate the VLSI circuit can produce results accurate enough for synthetic aperture radar and other demanding applications.

END

FILMED

3-86

DTIC