1981

PROCESSOR ARCHITECTURES FOR FAST COMPUTATION OF MULTI-DIMENSIONAL UNITARY TRANSFORMS.

HARI K. NAGPAL
University of Windsor

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

PROCESSOR ARCHITECTURES FOR FAST COMPUTATION OF MULTI-DIMENSIONAL UNITARY TRANSFORMS

by

© Hari K. Nagpal

A Dissertation
Submitted to the Faculty of Graduate Studies
through the Department of Electrical Engineering
in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy at the
University of Windsor

Windsor, Ontario, Canada

1981

ABSTRACT

This work presents the development of new algorithms and special purpose sequential processor architectures for the computation of a class of one-, two- and multi-dimensional unitary transforms.

In particular, a technique is presented to factorize the transformation matrices of a class of multi-dimensional unitary transforms, having separable kernels, into products of sparse matrices. These sparse matrices consist of Kronecker products of factors of the one-dimensional transformation matrix. Such factorizations result in fast algorithms for the computation of a variety of multi-dimensional unitary transforms including Fourier, Walsh-Hadamard and generalized Walsh transforms. It is shown that the u-dimensional Fourier and generalized Walsh transforms can be implemented with a u-dimensional radix-r butterfly operation requiring considerably fewer complex multiplications than the conventional implementation using a one-dimensional radix-r butterfly operation.

Residue number principles and techniques are applied to develop novel special purpose sequential processor architectures for the computation of one-dimensional discrete Fourier and Walsh-Hadamard transforms and convolutions in real-time. The residue number system (RNS) based implementations yield a significant improvement in processing speed over the conventional realizations

using the binary number system.

As an illustration of the factorization techniques developed in this work, novel sequential architectures of RNS-based fast Fourier, Walsh-Hadamard and generalized Walsh transform processors for real-time processing of two-dimensional signals are presented. These sequential processor architectures are capable of processing large bandwidth (> 5 M.Hz) input sequences.

The application of the proposed FFT processors for the real-time computation of two-dimensional convolutions is also investigated. A special memory structure to support two-dimensional convolution operations is presented and it is shown that the two-dimensional FFT processor architecture proposed in this work requires less hardware than the conventional implementations. The FFT algorithms and processor architectures are verified by computer simulation.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisors, Dr. J.J. Soltis and Dr. G.A. Jullien, for their invaluable advice, help and constant encouragement throughout the progress of this research. The valuable advice of Dr. W.C. Miller and other faculty members is gratefully acknowledged. In addition, the help of many of the graduate students and Mr. J. Novasad is sincerely appreciated.

To my parents and brothers, I extend my sincerest thanks and gratitude. Without their help and inspiration, this work would not have started.

Thanks are also due to Mrs. Marion Campeau for her diligence in typing this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

1-D	One-dimensional
2-D	Two-dimensional
u-D	u-dimensional
D_k	Quasi-diagonal Matrix of size (kxk) specifying multiplication by the twiddle factors
f	Vector containing the input of a Unitary Transformation
f_i	Vector containing the input to the ith stage of a Unitary Transform
F	Vector containing the output of the DFT
I_k	Identity Matrix of size (kxk)
p	Vector containing the transformed sequence
$p_i^{(r)}$	Base-r permutation operator of the ith stage
P	Integer conversion factor
P_N	Base-2 perfect shuffle operator of size (NxN)
$P_M^{(r)}$	Base-r perfect shuffle operator of size (MxM)
r	Radix of the FFT algorithm
r_i	ith residue of a number modulo m_i
$R_m^{(r)}$	Multi-dimensional r-point transform operator for the mth stage
$s^{(r)}, s^{(r)}$	Radix-r transform operator
$s_m^{(r)}$	Radix-r transform operator for the mth stage
T	Multi-dimensional unitary transform matrix
T_M	Unitary transformation matrix of size (MxM)

T_{M_i}	One-dimensional unitary transformation matrix of a multi-dimensional transform in the ith dimension
u	Dimension of an multi-dimensional array
W_M	$e^{-j2\pi/M}$
$\mu_m^{(r)}$	Base-r twiddle factor operator for the mth stage
β_i	ith factor of a unitary transformation matrix
$\psi_m^{(r)}$	Multi-dimensional base-r twiddle factor operator for the mth stage
T	Access-time of a buffer/ROM
G*	Complex conjugate of the matrix G
$[\cdot]^T$	Transpose of the matrix enclosed
$[\cdot]_{\ell k}$	ℓ, kth element of the matrix enclosed
$[\cdot]_R$	The closest integer to the term enclosed
$[\cdot]^{\otimes n}$	Kronecker nth power of the matrix enclosed
$A \otimes B$	Kronecker product of the matrices A and B
$\lvert x \rvert_{m_i}$	The residue of x modulo m_i
$\left\lvert \frac{1}{\hat{m}_i} \right\rvert_{m_i}$	Multiplicative Inverse of $\hat{m}_i$ modulo m_i
$\{a_i\}$	Set containing the elements a_i
Re(x)	Real-part of the complex number x
Im(x)	Imaginary-part of the complex number x
BUF, BUFF	Memory buffer
$(H_N)_h$	Natural-order WHT transformation matrix of size (NxN)
$(H_N)_d$	Dyadic-ordered WHT transformation matrix of size (NxN)
MUX	Multiplexer unit

RNS	Residue Number System
ROM	Read-Only Memory
SUB	Memory Sub-buffer
SR	Shift Register

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

In recent years there has been a growing interest in the study of unitary transforms in the area of one- and two- dimensional digital signal processing. Major applications of such transforms include image processing, speech processing, feature selection in pattern recognition, data compression and generalized spectral analysis. For example, the high frequency terms of the Fourier transform of an image give an indication of the amplitude and orientation of the edges within an image. Other applications are in transform coding in which a bandwidth reduction is achieved by discarding or grossly quantizing low-magnitude transform coefficients.

A discrete unitary transform is characterized by a unitary matrix G such that $G \cdot [G^*]^T = I$ where * denotes a complex conjugate, $[\cdot]^T$ denotes a transpose and I is the identify matrix of the same order as G, say N. The computation of the transform of an input vector, f, of dimension N can be written as a matrix vector multiplication $G \cdot f$, which usually requires N^2 operations where an operation is defined as a multiplication and an addition. In 1958 Good [1] proposed a technique to factorize the matrix, G, into a set of sparce matrices, each expressing a stage of computation. In 1965 Cooley and Tukey [2] used the approach followed

by Good and developed the Fast Fourier Transform (FFT) algorithm to compute the discrete Fourier Transform (DFT). In general, the FFT algorithm resulted in the computation of the DFT in $(N \cdot r \cdot \log_r N)$ operations as compared to N^2 operations required for the direct implementation. Recently, the matrix factorization approach described by Good has also been applied to develop fast algorithms for the computation of Hadamard [3], Walsh [4], Haar [5], generalized Walsh [6], generalized Haar [6] and a more general class of orthogonal transforms [7]. These algorithms also achieve a computational saving which is of the same order as that of the DFT. Two new algorithms (prime factor FFT and Winograd Fourier transform algorithms [9],[10]) to compute the DFT have also been proposed recently, which achieve further savings in the computation by computing the DFT via convolution using number theoretic concepts.

Many of the fast algorithms, reported in the literature, deal with one-dimensional transforms. Multi-dimensional unitary transforms are generally computed by taking one-dimensional transforms, utilizing fast one-dimensional unitary transform algorithms, along all dimensions, successively. The processing speed of the multi-dimensional unitary transform algorithms depends on the efficiency of the one-dimensional unitary transform algorithm employed. In [1] Good also proposed techniques to factorize the multi-dimensional transformation matrix into a set of sparse matrices containing elements of the one-dimensional transformation matrix. There is no evidence that this technique of

factorization of a multi-dimensional transformation matrix has been used to develop fast algorithms for the computation of multi-dimensional unitary transforms. Recently, Arambepola and Rayner [34] described a FFT algorithm for the computation of multi-dimensional DFT's which achieves a savings of more than 25% in the number of multiplications over the conventional FFT algorithms. The fast algorithms for the computation of other multi-dimensional unitary transforms are not well developed in the literature.

The principal objective of this research work is to develop two- and multi-dimensional unitary transform algorithms which are well adapted for implementation by a special purpose processor for real-time* processing of two and multi-dimensional signals. In this work, we extend the idea of matrix factorization, proposed by Good [1] to explore further the redundancy available in the transformation matrices of many unitary transforms and present a unified approach to the development of fast algorithms for a class of unitary transforms. Based on this approach we present the development of fast algorithms for a number of unitary transforms for real-time digital signal and image processing applications. Before presenting the approach followed for the implementation of the proposed algorithms and other objectives of this work we give a brief survey of the various special purpose processor architectures, proposed in the literature, for the implementation of unitary transform algorithms.

---

* By real-time we mean that the processor accepts the input data and generates the output data at the same sampling rate, continuously. The processing delay between an input sequence and the corresponding output is ignored.

1.1 UNITARY TRANSFORM IMPLEMENTATIONS

Since the DFT is an important transform in most digital signal processing applications, the FFT algorithm has been explored extensively. A detailed derivation of the FFT algorithms can be found in [7] and [23]. In general, there are two basic versions of the FFT algorithm, namely, the decimation-in-time (DIT) and the decimation-in-frequency (DIF) algorithm [23]. Both of the two versions have various structures which require ordered or digit reversed input and generate digit reversed or ordered output. An FFT algorithm computes the DFT of an N-point sequence for $N = r^n$, in $\log_r N$ stages and there are $\frac{N}{r}$ radix-r butterfly operations in each stage. A radix-r butterfly operation consists of an r-point DFT operation on a set of r data points and multiplication of the r input data points or the result of r-point DFT operation by a set of complex coefficient known as twiddle factors [23]. A radix-2 implementation has the simplest structure but the efficiency of computation is improved by employing a higher radix FFT algorithm [26].

A general purpose processor is normally provided with a real adder, a real multiplier and memory having a word length of one data point. The software implementation of the FFT algorithm on such a processor requires 28 memory cycles [23] to perform a radix-2 butterfly operation. Since there are $N/2 \log_2 N$ butterfly operations, the total time required to compute an N-point DFT is given by $14 \cdot T \cdot N \cdot \log_2 N$ where T is the memory access time. This computation time does not

include the software overhead required to compute the data and twiddle factor addresses. Also, additional time is required to digit reverse the input or output sequence.

To achieve higher processing speeds most of the research effort on the hardware implementation of special purpose FFT processors has been concentrated on the development of

(1) faster and new adder and multiplier structures [19],[27],[28];

(2) Butterfly units with overlapped memory and arithmetic cycles and pipelined arithmetic operations [20],[23],[29];

(3) memory structures to access multiple words simultaneously to reduce memory access time and to simplify the generation of data and twiddle factor addresses [11]-[27].

The structure of the FFT algorithm has also been exploited to perform multiple butterfly operations simultaneously [16]-[18],[30].

Based on the number of Butterfly units employed by a FFT processor, Bergland [30] divided the hardware implementation of the FFT algorithm into four categories, namely, the sequential, cascade and parallel iterative processors and the array analyzer. Table 1.1 shows some of the features of the four different realizations. The sequential realization is simplest to implement but the slowest of the four implementations. The cost of the third and fourth implementation is very high, which limits their applications. For high speed applications many forms of cascade processors have been proposed in the literature [11],[16]-[18],[20],[24]. Pease [16] and Gold [20] proposed the use

TABLE 1.1

COMPARISON OF FFT REALIZATIONS

	Realization	Butterfly units	Butterfly Operations Performed In		Execution Time T
			Parallel	Sequential	
1	Sequential	1	1	$\frac{N}{r} \cdot m$	$T^* \cdot \frac{N}{r} \cdot m$
2	Cascade	m	m	$\frac{N}{r}$	$T \cdot \frac{N}{r}$
3	Parallel Iterative	$\frac{N}{r}$	$\frac{N}{r}$	m	$T \cdot m$
4	Array Analyzer	$m \cdot \frac{N}{r}$	$m \cdot \frac{N}{r}$	1	T

N = number of point in input sequence

$m = \log_r N$ (number of stages)

*T is the time required to perform a Butterfly operation and includes the time required to generate data and twiddle factor addresses.

of serial memories to eliminate the calculation of memory addresses at each stage of the FFT algorithm. Whelchel [11] also proposed the use of serial memories for a high-radix cascade processor. Since the cost of a cascade processor is proportional to the number of Butterfly units employed, Corinthios [12]-[15] proposed the use of serial memories for the implementation of a sequential processor for moderate cost and medium speed applications. Also the proposed memory organization yielded the output Fourier coefficients in a properly ascending order without the need for pre- or post-ordering of the data.

The redundancy in the transformation matrix of Hadamard, Walsh, Haar, generalized Walsh, Cosine and many other unitary transforms has also been exploited to develop the corresponding fast algorithms. Similar to the development of the FFT processor proposed in [13], Corinthios [22] proposed the organization of a Walsh-Hadamard transform processor. The only difference between the two processors is that they specify a different permutation of the imtermediate results and there are no multiplications by the twiddle factors in the Walsh-Hadamard transform processor. In [5] Ahmed has proposed a Cooley-Tukey-type algorithm to compute the Haar transform and thus the Haar transform can be computed in a modified form of the FFT processor. The fast algorithms to compute generalized Walsh [6], Cosine [31] and many other unitary transforms [5],[31],[32] have structures which are similar to the FFT algorithm.

Multi-dimensional unitary transforms are generally computed by

taking one-dimensional unitary transform along all the dimensions, successively [31]-[34]. In applications, such as image processing and seismic data processing, the volume of data handling is quite large and some form of bulk storage must be used to store the input and output data. In earlier computers the size of available fast access memory was limited to only a few columns (or rows) of the two-dimensional data to be transformed. Singleton [35] and Hunt [36] have discussed techniques to transform a large matrix of data stored in an external bulk store with limited amount of high speed memory. Since these methods required transposition of the matrix after the column (or row) transforms, Anderson [37] proposed a modification of the Cooley-Tukey algorithm to avoid matrix transposition and to reduce the associated input/output operations.

Software implementation of the multi-dimensional transforms on a general purpose processor with auxiliary storage results in very slow processing speeds. Recent advances in semi-conductor memory have resulted in the development of high density and low cost memory packages and it is now possible to provide large amounts of fast access storage at a reasonable cost. A large improvement in the computation time of multi-dimensional transforms has also been obtained by the use of special purpose one-dimensional transform processors or general purpose array processors [38]. For very high speed applications, such as real-time transmission of pictures at video rates and real-time filtering of images and robotics, special purpose processors must be developed to cope with the large volume of data.

Joshi, McDonald et al. [21] have proposed a high speed multiprocessor architecture suitable for realizing a 512x512 point FFT in under 1/30 of a second. This processor has been realized with 16 one-dimensional Butterfly units and a number of memory units.

In the above discussion on the implementation of the unitary transform it was implied that all the compututations are performed using the binary number system (fixed point or floating point). Recently, the residue number system (RNS) has received considerable attention for the implementation of digital signal processing elements [39]-[44]. The use of RNS techniques has been proposed for the implementation of recursive and non-recursive filters [39],[42],[43] DFT [40], [41] and Number theoretic transforms [25]. The unique advantage of the RNS, as compared to the binary number system, is that the binary operations of addition, subtraction and multiplication can be performed using a number of independent and parallel paths. Addition and subtraction have no inter-digit carries or borrows and multiplication does not need the generation of partial products, thus resulting in fast operating speed. Also these operations may be performed at very high speed by the use of ROMs to provide parallel arrays of look-up tables [39].

From this discussion on the implementation of one- and multi-dimensional unitary transforms the conclusion that emerges is that a general purpose processor is incapable of satisfying the very high throughput rate requirement of many real-time applications and thus a

special purpose processor must be realized. In order to design a special purpose processor one must consider various options and examine the tradeoffs that must be made. The architecture of a special purpose processor is usually dictated by the performance and cost requirements. Due to the inherent structural properties of the various unitary transform algorithms, a special purpose processor can be designed with increasing degrees of parallelism, performance and cost. Out of the four catagories of the FFT processors, discussed earlier, a cascade architecture satisfies the processing speed requirement of most of the real-time applications and a number of different cascade processors have been proposed in the literature. Sequential architectures have normally been considered for medium speed applications because of the limited computational power. A significant improvement in the processing speed of a sequential processor has been obtained by employing various pipelining techniques in the design of the arithmetic unit and special memory structures. The major factor which limits the processing speed of a sequential processor is the speed limitations associated with the binary multiplications. Although various hardware units are commercially available, higher speed can only be obtained at higher cost and complexity. Fortunately, the residue number system can be used to overcome some of the problems associated with the binary number system.

Another objective of this work is to explore the application of residue number techniques to the implementation of unitary

transforms by a special purpose sequential processor for real-time digital signal and image processing. Although we have already given a preview of the objective of this work, we restate it again for clarity.

1.2 OBJECTIVE AND OUTLINE OF THE RESEARCH

The principal objective of this research is to explore further the redundancy available in the transformation matrix of one- and multi-dimensional unitary transforms and to propose special purpose processor architectures for real-time processing of one- and multi-dimensional signals. Because of the recent advances in memory technology and the development of high speed, high density and inexpensive memory IC's, it is becoming more and more attractive to trade logic gates for memory packages, we explore the use of RNS techniques to the implementation of high speed processors. Since the ROM oriented implementation of a RNS-based Butterfly unit allows pipelining of the basic arithmetic operations within the Butterfly unit, it is believed that a special memory architecture, combined with a single Butterfly unit, will alleviate the processing speed limitation of a sequential processor and thus the throughput rate requirement (>5 M. samples/sec.) of many real-time applications can be satisfied with a sequential processor.

Based on the above philosophy, this work concentrates on the development of multi-dimensional unitary transform algorithms that are well adapted for implementation by the parallel architectures of an

RNS-based special purpose sequential processor, and also result in a simple memory architecture to allow full utilization of the Butterfly unit.

As a starting point, we explore the nature of one-dimensional unitary transforms for which a number of fast algorithms have been proposed in the literature. In particular, we consider the memory structures for the implementation of the RNS-based one-dimensional FFT and fast Walsh-Hadamard transform processors. We show that a significant improvement in processing speed can be obtained with the use of RNS techniques. We also propose the FFT processor architectures for the implementation of one-dimensional convolution and to compute the DFT of multiple sequences in real-time. In the case of Walsh-Hadamard transforms (WHT) we show that the proposed FFT processor architecture can also be used to compute the WHT with a significant improvement in processing speed.

For the implementation of multi-dimensional unitary transforms, we present a technique to factorize the transformation matrices of a class of multi-dimensional unitary transforms into products of sparse matrices. Such factorizations result in fast unitary transform algorithms and yield processor structures which are similar to the one-dimensional processor organizations. As an illustration of the above technique, the architectures of the FFT and WHT processors for real-time processing of two-dimensional signals is presented. Similar to the one-dimensional algorithms, multi-dimensional FFT algorithms compute a DFT in $\log_r M$ stages, where $M = r^n$, but employ a multi-

dimensional radix-r butterfly operation and achieve a considerable saving in the number of multiplications over the conventional implementations.

Since the FFT algorithm is extensively used to perform convolutions, we also show the development of a special memory architecture required to support the implementation of two-dimensional convolutions. These convolver organizations are shown to be suitable for the computation of convolutions using Number theoretic transforms defined over complex residue rings.

1.3 THESIS ORGANIZATION

In Chapter 2, a review of the basic structure of the one-dimensional unitary transformation matrices and their associated fast algorithms is provided. Special emphasis is placed on the factorization techniques used to develop one-dimensional FFT and WHT algorithms as these are used to develop fast algorithms for the multi-dimensional unitary transforms. The RNS and its basic properties are also reviewed.

Chapter 3 covers the development of the RNS-based one-dimensional FFT and WHT processors and convolvers. An algorithm and the resulting processor architecture for processing data from a number of independent channels are also developed.

In Chapter 4, fast algorithms based on the Kronecker product factorization of the multi-dimensional unitary transformation matrix are developed. These algorithms are applied to develop RNS-based processor architectures for the implementation of two-dimensional DFT,

WHT and generalized Walsh transforms. Processor architectures for real-time computation of the two-dimensional DFT, WHT and generalized Walsh transform are also presented.

Chapter 5 covers the implementation of two-dimensional convolution using the one- and two-dimensional FFT processors developed in Chapter 3 and 4. Special memory organizations for the implementation of two-dimensional convolutions are also proposed. On the basis of cost to performance ratio, a comparison of the two convolver architectures realized using one- and two-dimensional FFT processors, respectively, is presented. A comparison of the convolver organizations realized via FFT and NTT defined over complex residue rings is also presented.

In Chapter 6, an approximate number of commerically available IC packages required to implement one- and two-dimensional RNS-based FFT processors and convolvers is given. The results obtained via simulation of the proposed FFT processors and convolvers are compared with the results obtained via the implementation of the standard FFT and convolution algorithms.

Chapter 7 summarizes the conclusions resulting from this research.

CHAPTER 2

ONE-DIMENSIONAL UNITARY TRANSFORM ALGORITHMS

2.1 INTRODUCTION

In the past, the computation of a unitary transform of an N-point sequence required N^2 operations and proved very time consuming and uneconomical. Since the development of the fast Fourier transform algorithm by Cooley and Tukey [2] in 1965, a significant amount of research effort has been directed towards finding efficient algorithms for the implementation of unitary transforms [1]-[10]. The underlying principle for efficient implementation of unitary transforms is the utilization of the high degree of redundancy in the transform matrix description (transformation matrix). If the redundancy in a transformation matrix can be eliminated by matrix factorization, then a more efficient way of implementation is available. The technique of factorization of a unitary transformation matrix into products of sparse matrices was first described by Good [1] in 1958 and resulted in the development of fast Fourier [2], Hadamard [3], Walsh [4], Haar [5], generalized Walsh [6], generalized Haar [6] and a more general class of othogonal transform [7] algorithms.

As indicated in the last Chapter, software implementation of the unitary transforms on a general purpose processor result in relatively slow processing speed. For a large class of applications which have an inherent large bandwidth real-time constraint or have a large amount of

data, speed becomes very important and several special purpose unitary transform processors have been proposed in the literature [11]-[25]. The requirement of high processing speed necessitates the search for available algorithms or the development of new algorithms that are well adapted for a parallel processor architecture. The problem of designing a special purpose processor is thus analogous to the problem in computer architecture in which we seek a proper match between the implemented algorithm and the various resources of a processor. In this work we follow both of the above approaches; that is we search for algorithms already proposed in the literature and we also propose new algorithms that are well adapted for implementation by a special purpose processor. In this Chapter we review the basic concepts of unitary transforms and the techniques used to develop fast unitary transforms algorithms. We also describe some of the algorithms and processor architectures, proposed in the literature, which are suitable for implementation by a special purpose processor. In the following chapters we propose extensions of the published algorithms and processor architectures and also present development of new algorithms and the resulting processor architectures for real-time processing of two-dimensional signals.

2.2 ONE-DIMENSIONAL UNITARY TRANSFORMS

The one-dimensional forward unitary transform of an N-point sequence $f(n)$, $n=0,1,..,N-1$ is defined as

$$p(m) = \sum_{n=0}^{N-1} f(n)\, g(n,m); \quad m=0,1,...,N-1 \tag{2.1}$$

where p(m) is the transformed sequence and g(n,m) is the forward transformation kernel. Similarly, the inverse unitary transform of the sequence p(m) is given by the relation

$$f(n) = \sum_{m=0}^{N-1} p(m)\, h(n,m); \quad n=0,1,\ldots,N-1 \tag{2.2}$$

where h(n,m) denotes the inverse transformation kernel. The transform pair defined in equation (2.1) and (2.2) is unitary if the transformation kernels g(n,m) and h(n,m) satisfy the following orthonormality conditions:

$$\sum_{m=0}^{N-1} g(n,m) \cdot g^*(j,m) = \delta(n-j) \tag{2.3a}$$

$$\sum_{m=0}^{N-1} h(n,m) \cdot h^*(j.m) = \delta(n-j) \tag{2.3b}$$

$$\sum_{n=0}^{N-1} g(n,m) \cdot g^*(n,k) = \delta(m-k) \tag{2.3c}$$

$$\sum_{n=0}^{N-1} h(n,m) \cdot h^*(n,k) = \delta(m-k) \tag{2.3d}$$

where $g^*(x)$ denotes the complex conjugate of g(x) and $\delta(x)$ is the Kronecker delta. Based on the sets of orthogonal functions such as exponential, Walsh and Haar, a number of discrete unitary transforms, namely, discrete Fourier transform (DFT), Walsh-Hadamard transform (WHT), Haar (HT), discrete Cosine transform (DCT), generalized transform (GT) and generalized Walsh transform (GWT), have been defined in the literature [6], [7], [31], [32].

In the development of fast algorithms for the computation of a unitary transform, it is useful to express equations (2.1) and (2.2)

in vector notation. Let p and f be vector representations of the sequences p(n), f(n), n=0,1,..,N-1, i.e.

$$p = [p(0),p(1),\ldots,p(N-1)]^T; \quad f = [f(0),f(1),\ldots,f(N-1)]^T$$

where $[x]^T$ denotes the transpose of [x]. Then in the vector representation equations (2.1) and (2.2) can be written as

$$p = Gf \tag{2.4}$$

and

$$f = Hp \tag{2.5}$$

where G and H are the forward and inverse transformation matrices of size (NxN), respectively, and the elements of G and H are given by $G_{n,m} = g(m,n)$ and $H_{n,m} = h(n,m)$. Since equations (2.4) and (2.5) define a transform pair $H = G^{-1}$, and for a unitary transform $G^{-1} = [G^*]^T = H$, hence the transformation matrices G and H are unitary. For some unitary transforms, the transformation matrix G is real and the unitary transforms are known as orthogonal transforms. For such transforms $H = G^{-1} = G^T$.

In the vector representation of a unitary transform, the rows of its transformation matrix may be obtained by sampling the set of orthogonal functions defining the unitary transform. Also, in terms of the vector respresentation, the transformation matrices of a unitary transform satisfy the following orthonormality conditions which are equivalent to the conditions given by (2.3):

$$[G^*]^T \cdot G = G \cdot [G^*]^T = I_N \tag{2.6a}$$

$$[H^*]^T \cdot H = H \cdot [H^*]^T = I_N \tag{2.6b}$$

where I_N denotes the identity matrix of size (NxN). Examples of the transformation matrices defining the DFT, WHT and HT are shown in Fig. 2.1, for N = 8, where $W = e^{-j2\Pi/N}$. These matrices may be obtained by sampling the sets of complex exponential, Walsh and Haar functions at N equidistant points. A detailed procedure for the generation of transformation matrices for a large class of unitary transforms may be found in the references [7], [31] and [32].

A direct computation of equation (2.4) or (2.5) requires N^2 operations. As mentioned earlier, an efficient way of computing a unitary transform may be obtained if the transformation matrix defining the transform can be factored into a product of a set of sparse matrices. For a number of unitary transforms such a factorization, and the resulting fast algorithms, has been proposed in the literature [1]-[27], [30]-[38]. Since the discrete Fourier and Walsh-Hadamard transforms are most often used in digital signal and image processing, we will concentrate on the factorization of DFT and WHT transformation matrices and the processor architectures implementing the corresponding fast algorithms. The factorization techniques discussed here can be extended easily to other unitary transforms.

2.3 ONE-DIMENSIONAL DISCRETE FOURIER TRANSFORM IMPLEMENTATION

In 1965, Cooley and Tukey [2] applied the matrix factorization technique proposed by Good [1] to compute the DFT of an N-point sequence for $N = 2^{\ell}$, which resulted in a fast algorithm to compute the DFT in $N \log_2 N$ operations instead of the originally required N^2 operations.

$$\frac{1}{\sqrt{8}}\begin{vmatrix} w^0 & w^0 & w^0 & w^0 & w^0 & w^0 & w^0 & w^0 \\ w^0 & w^1 & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ w^0 & w^2 & w^4 & w^6 & w^0 & w^2 & w^4 & w^6 \\ w^0 & w^3 & w^6 & w^1 & w^4 & w^7 & w^2 & w^5 \\ w^0 & w^4 & w^0 & w^4 & w^0 & w^4 & w^0 & w^4 \\ w^0 & w^5 & w^2 & w^7 & w^4 & w^1 & w^6 & w^3 \\ w^0 & w^6 & w^4 & w^2 & w^0 & w^6 & w^4 & w^2 \\ w^0 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w^1 \end{vmatrix}$$

(a)

$$\frac{1}{2\sqrt{2}}\begin{vmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{vmatrix}$$

(b)

$$\frac{1}{\sqrt{8}}\begin{vmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{vmatrix}$$

(c)

Fig. 2.1 8x8 TRANSFORMATION MATRICES OF (a) DFT (b) WHT (c) HT

Since then, many variations of this basic algorithm have been proposed; all variations being termed Fast Fourier transforms (FFT) algorithms [23]. Although all of the FFT algorithms available in literature can be implemented on a general purpose processor as software or firmware routines, the throughput rate obtainable from this type of implementation is sufficient only for non real-time applications. A moderate increase in processing speed has been obtained by the use of a combination of a general purpose computer and an array processor [38] where the array processor performs the arithmetic operations, specified by the FFT algorithm, in its high speed arithmetic unit. Due to the constraints imposed by the internal architecture of most of the commercially available array processors, these processors are unable to perform arithmetic operations on multiple data items simultaneously and thus cannot satisfy the throughput rate requirement of many real-time applications. Hence, for high processing speed, we must investigate various special purpose processor architectures which are more suitable for parallel processing. Since the objective of this work is the development of algorithms that are well adaptable for implementation by the parallel machine architecture of a special purpose processor, we will discuss some of the algorithms, proposed in the literature, which are better suited for parallel processing.

In 1968, Pease [16] discribed a variation of the Cooley-Tukey FFT algorithm in which he first partitioned the DFT transformation matrix and then factored it in terms of arithmetic and permutation operators. In a manner similar to the Cooley-Tukey algorithm, Pease's factorization

specified an ℓ-stage algorithm for $N = 2^{\ell}$, but in Pease's factorization each stage consisted of a constant permutation of the input data elements and the arithmetic operator specified operations of addition and subtraction on pairs of successive data elements. Pease also suggested the implementation of this algorithm on a cascade/parallel iterative type of processor where the permutation operation is performed by a hardwired shift network. In 1969, Pease [17] suggested another architecture where he emphasized the utilitization of relatively slow memories and a number of arithmetic units operating in parallel. Whelchel and Guinn [11] used the matrix factorization techniques suggested by Pease and proposed the implementation of a radix-4 cascade processor utilizing serial shift registers for high-speed digital filtering. Similar cascade/pipeline architectures using serial memories were also proposed by Groginsky and Works [18] and Gold and Bailly [20]. These architectures used less amounts of memory than required by the organization proposed in [11], but were more suitable for simultaneous processing of data from a number of input channels.

All the above mentioned implementations of the FFT algorithm require a number of arithmetic units and generate the Fourier coefficients in a digit-reversed order. For applications which call for an ordered output, a pre-shuffling or post-ordering of the data is required which results in a reduced throughput rate. Corinthios [12]-[15] used the matrix factorization techniques suggested by Pease [16] and proposed a number of algorithms that are well suited for implementation by a serial sequential processor. Also, these algorithms absorb the post-

ordering operations within the permutation operations of each stage and the Fourier coefficients are generated in the natural ascending order without any loss in processing speed. A processor architecture implementing these algorithms and using serial memories is described in [15].

Although the cascade/pipeline FFT processor architectures proposed by Pease, Groginsky and Works, and Gold and Bailly satisfy the processing speed requirement of most of the real-time signal processing applications, the cost of these processors is proportional to the number of arithmetic units they employ. In this work we will show that, with the use of high density, high speed and low cost commercially available memory packages, the throughput rate obtainable from a serial sequential FFT processor architecture, that is well matched to the implemented algorithm, is comparable to the throughput rate obtainable from a cascade/pipeline architecture. We will also show that the architecture of a sequential processor is also adaptable to process two-dimensional signals in real-time.

Since, the FFT algorithms proposed by Corinthios [13], [14] result in a serial sequential hardwired processors having simplified control functions for the generation of data and twiddle factor addresses, we choose the Ordered-input Ordered-output (OIOO) FFT algorithm as the basis of our further investigation. Another advantage of this algorithm is that it generates the Fourier coefficients in a properly ascending order without pre-ordering of the input data. Since OIOO FFT algorithm

also leads to the development of a high speed two-dimensional FFT processor, we describe this algorithm in detail.

2.3.1 One-Dimensional (1-D) OIOO FFT Algorithm and its Implementation

The DFT of an M-point sequence f(n) is defined over the field of complex numbers as

$$F(k) = \sum_{\ell=0}^{M-1} f(n)\, W^{\ell k}; \quad \ell, k = 0,1,\ldots,M-1 \tag{2.7}$$

where $W = e^{-j2\pi/M}$ is the Mth root of unity over the field. This can also be written in the matrix form of equation (2.4) as

$$F = T_M\, f \tag{2.8}$$

where T_M is the MxM DFT transformation matrix and the elements of T_M are given by

$$[T_M]_{\ell,k} = W^{\ell k} \tag{2.9}$$

To simplify notation, we preserve only the exponent of W, that is we write k in place of W^k. Then T_M can be written as

$$T_M = \begin{vmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 2 & 3 & \cdots & (M-1) \\ 0 & 2 & 4 & 6 & \cdots & 2(M-1) \\ 0 & 3 & 6 & 9 & \cdots & 3(M-1) \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & (M-1) & 2(M-1) & 3(M-1) & \cdots & (M-1)^2 \end{vmatrix} \tag{2.10}$$

The matrix T_M can be expressed in terms of a permutation matrix $P_M^{(r)}$ and T_M' as

$$T_M = P_M^{(r)} \, T_M' \tag{2.11}$$

or

$$T_M' = [P_M^{(r)}]^{-1} \, T_M \tag{2.12}$$

where $P_M^{(r)}$ is the ideal shuffle base-r permutation operator which is defined by its operation on a vector of dimension M, where $M/r = q$ is an integer, by the relation

$$P_M^{(r)} \cdot [x_0, x_1, x_2, \ldots, x_{M-1}]^T$$

$$= [x_0, x_q, x_{2q}, \ldots, x_1, x_{q+1}, x_{2q+1} \ldots, x_2, x_{q+2}, x_{2q+2} \ldots, x_{M-1}]^T \tag{2.13}$$

When M is a composite number, i.e. $M = r^n$, Corinthios [14] has shown that the matrix T_M' cam be partitioned into (r x r) square sub-matrices, each of dimension M/r, and factored into a product of matrices including a matrix $T_{M/r}$. $T_{M/r}$ iteself can then be partitioned and factored in terms of T_{M/r^2}. If this process is iterated n times we obtain a complete factorization of T_M. The result of the ith iteration can be written as

$$T_{M/k} = P_{M/k}^{(r)} \, (T_{M/rk} \otimes I_r) \, D_{M/k}^{(r)} \, (I_{M/rk} \otimes T_r) \tag{2.14}$$

where $k = r^i$, the symbol $\otimes$ stands for the Kronecker product of matrices [16] and

$$D_{M/k} = \text{quasi-diag}\,(I_{M/rk},\ L_k,\ L_{2k},\ldots,L_{(r-1)k})$$

$$L_m = \text{diag}\,(0,\ m,\ 2m,\ \ldots,\ (M/rk-1)m) \tag{2.15}$$

$$T_r = \begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & M/r & 2M/r & 3M/r & \cdots & (r-1)M/r \\ 0 & 2M/r & 4M/r & 6M/r & \cdots & 2(r-1)M/r \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ 0 & (r-1)M/r & 2(r-1)M/r & 3(r-1)M/r. & \cdots & (r-1)^2M/r \end{bmatrix} \tag{2.16}$$

and I_k is the identity matrix of dimension k. Using (2.14) and replacing in turn every matrix $T_{M/k}$ by its factors in terms of $T_{M/rk}$ a complete factorization of T_M is obtained and can be written as

$$T_M = \prod_{m=1}^{n} \mu_m^{(r)}\ s_m^{(r)} \tag{2.17}$$

where $\mu_m^{(r)}$ is a twiddle operator specifying multiplications by the twiddle factors and is given by

$$\mu_i^{(r)} = (I_{r^{n-i}} \otimes D_{r^i}^{(r)});\quad i=2,3,..,n \tag{2.18}$$

$s_m^{(r)}$ is an r-point transform operator given by

$$s_{m-1}^{(r)} = s^{(r)}\ p_m^{(r)}\ ;\quad m=2,3,\ldots,n \tag{2.19a}$$

with

$$s_n^{(r)} = s^{(r)} = (I_{M/r} \otimes T_r) \tag{2.19b}$$

where $p_i^{(r)}$ is a permutation operator defined by

$$p_i^{(r)} = (I_{r^{n-i}} \otimes P_{ri}^{(r)}) \tag{2.20}$$

and

$$p_1 = \mu_1 = I_N \tag{2.21}$$

Also

$$s_{m-1}^{(r)} = p_m^{(r)} s^{(r)'}; \quad m=2,3,\ldots,n \tag{2.22}$$

where

$$s^{(r)'} = (I_{M/r^2} \otimes T_r \otimes I_r) \tag{2.23}$$

Substituting equation (2.17) into (2.8) we get

$$F = \{\prod_{m=1}^{n} \mu_m^{(r)} s_m^{(r)}\} f \tag{2.24}$$

or

$$F = \{\mu_1^{(r)} s_1^{(r)} \mu_2^{(r)} s_2^{(r)} \mu_3^{(r)} s_3^{(r)} \ldots \mu_n^{(r)} s_n^{(r)}\} f$$

Equation (2.24) simply states that the computation of F may be divided into n stages where each stage performs the computation specified by the operators $\mu_m^{(r)} s_m^{(r)}$. The operators of any stage operate on the output of the previous stage and operators $\mu_n^{(r)} s_n^{(r)}$ of the first stage operate on the input vector f.

For the intermediate stages the computation specified by the operators $\mu_i^{(r)} s_i^{(r)}$ can be written as

$$\mu_i^{(r)}\, s_i^{(r)} = \mu_i^{(r)}\, p_{i+1}^{(r)}\, s^{(r)'} = (I_{r^{n-i}} \otimes D_{r^i})(I_{r^{n-i-1}} \otimes P_{r^{i+1}}) \qquad (2.25)$$
$$(I_{M/r^2} \otimes T_r \otimes I_r)$$

If we look at the expanded form of the operator $s^{(r)'}$ we find that this operator always operates on data points which are M/r^2 words apart. An address separation of M/r^2 between the data points can be easily obtained if the memory storing the M points is split into r^2 serial sub-memories where each sub-memory stores M/r^2 data words. Thus, when the data from the sub-memories is accessed sequentially, the difference in their address is always M/r^2. A sequential memory organization, which allows both the addressing structure specified by $s^{(r)'}$ and permutation specified by $p_{i+1}^{(r)}$, is shown in Fig. 2.2 for $r = 4$.

The memory structure of Fig. 2.2 consists of two memory buffers BUF1 and BUF2, each storing M data words. Each buffer is divided into r sub-buffers (SUB) where each SUB consists of r shift registers (SR) with a storage capacity of M/r^2 data words. The SRs in each SUB can be connected in series to form a SUB of size M/r, and also the SUBs within BUF1 or BUF2 may be connected in series to form a shift register of size M. During the FFT computation, BUF1 and BUF2 are used to access data for the butterfly operation and to store the results alternatively, i.e. when BUF1 stores the data for input to the Butterfly unit, BUF2 stores the output of the Butterfly unit. The multiplexer (MUX) is used to select between BUF1 and BUF2 for data input to the

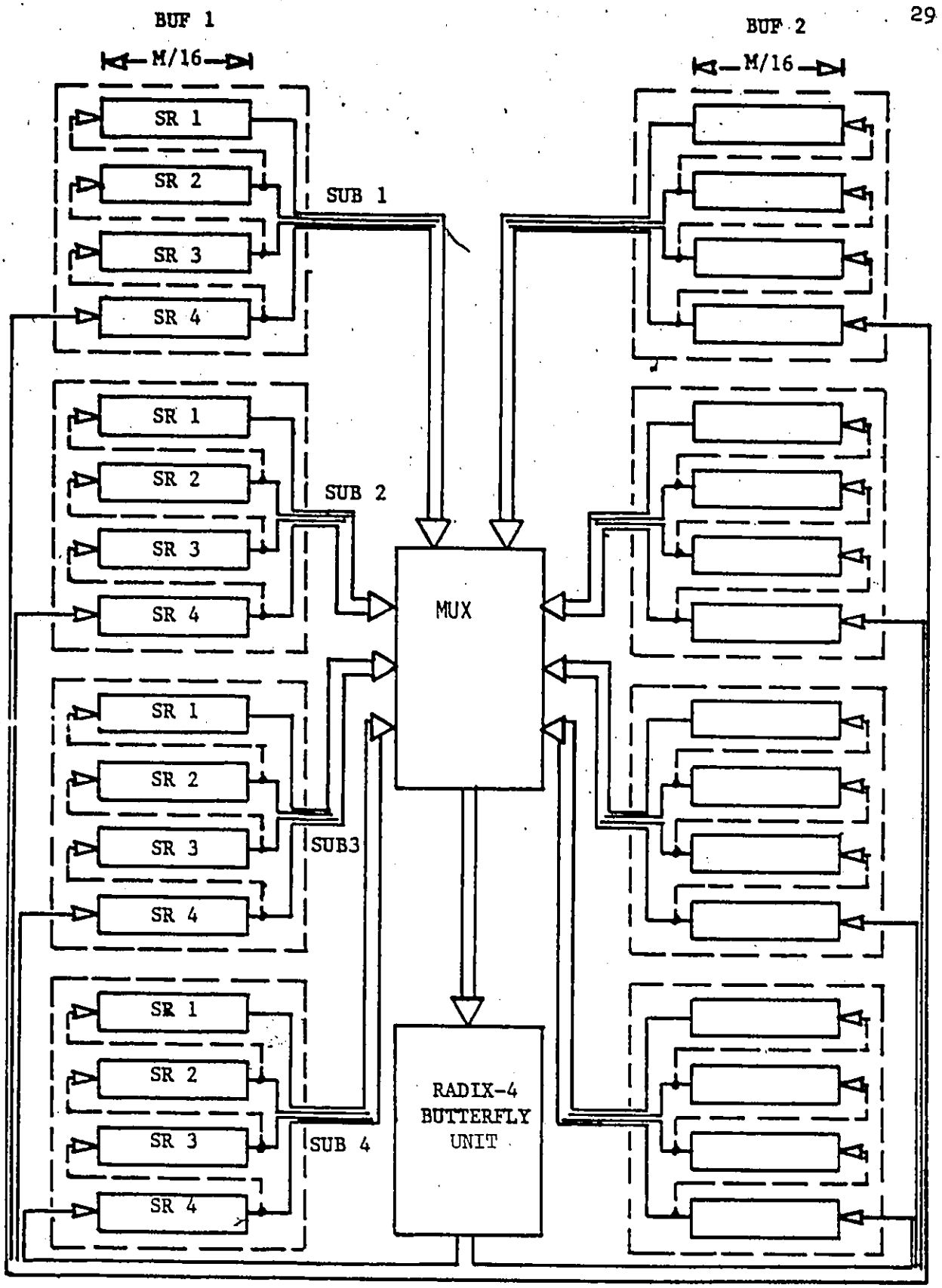


Fig. 2.2 Memory Organisation of the 1-D FFT Processor

Butterfly unit.

To implement the operations specified by $s^{(r)}$, we simply select a SUB and the data at top of the SRs of the selected SUB forms the input to the Butterfly unit. After the data is accepted by the Butterfly unit, each SR in the selected SUB is shifted right by one position so that the next data set becomes available at the output of the SRs. A SUB within a buffer is selected on a rotating basis, starting with SUB1, i.e. the selection of the SUBs always follows the order SUB1 - SUB2 - SUB3 - SUB4 - SUB1 ..... The number of butterflies for which a SUB is selected depends upon the operators $s^{(r)}$, and $p_{i+1}^{(r)}$ and hence the FFT stage. For the jth FFT stage the permutation is specified by the operator $p_{n-(j-2)}^{(r)}$ (the indexing of the FFT stage is opposite to the ordering of the stage operators) and we perform r^{j-2} sequential accesses from one SUB before selecting the next SUB. An additional base-r perfect shuffle operation is required to implement the operations specified by the operators $p_{i+1}^{(r)}\ s^{(r)}$, which can easily be achieved while storing the output of the Butterfly unit into the buffer used for storing the output. We cannect the SRs in each SUB of the output buffer in series and the r output points from the Butterfly unit are shifted into the SUBs, one point in each SUB. At the end of a stage, the output buffer will store the input data for the next stage.

The operations specified by $\mu_i^{(r)}$ are performed within the Butterfly unit. Since $\mu_i^{(r)} = (I_{r^{n-i}} \otimes D_{r^i})$ is a diagonal operator, it simply

specifies multiplication of the output of r-point transform with the twiddle factors. We can store all the required twiddle factors in ROMs and the ROM addresses can be generated by evaluating equation (2.15).

From equation (2.24) we see that the operations for the first FFT stage are specified by the operator $s_n^{(r)} = s^{(r)}$ (not by $s^{(r)}$,) and specifies that the r-point transform operator be applied to data points separated by M/r points. This can be achieved when the SRs in each SUB of the buffer containing the input vector f are connected in series. The r points taken from the top SR of each SUB then forms the input to the Butterfly unit. Throughout the first FFT stage, all the SUBs of the input buffer supply data to the Butterfly unit. The output buffer configuration remains the same as discussed earlier.

It may be noted that $\mu_1^{(r)} = I_N$ and thus for the last FFT stage the twiddle factors are unity. At the end of the last stage the output buffer will store the vector F in a natural sequential order.

In [15], Corinthios et al. proposed a radix-4 processor architecture, implementing the above OIOO algorithm, for digital spectral analysis. This processor incorporated the basic organization of Fig. 2.2 as the central part and an input and an output buffer for real-time processing of input data and the computation of average power spectra. The Butterfly unit of this processor used a very complex three-dimensional multiplier architecture with a dynamic range of 21 bits. This organization of the FFT processor resulted in a real-time processing speed of 1.6

million samples/second. In the next chapter we will present an extension of the above processor architecture to obtain throughput rates of over 10 million samples/second.

2.4 ONE-DIMENSIONAL WALSH-HADAMARD TRANSFORM AND ITS IMPLEMENTATION

The development of the FFT algorithm, to compute the DFT, by Cooley and Tukey [2] led to the development of many other unitary transforms having applications in the areas of digital signal processing, image processing and pattern recognition. In 1968, Whelchel and Guinn [3] introduced the Hadamard transform for signal representation and classifications which is based on the Hadamard matrix. Two variations of the Hadamard transform based on different ordering of the Walsh functions have also been proposed [7]. These different transforms are commonly known as Walsh-Hadamard transforms (WHT). This class of transforms has also been extended to a much larger class of orthogonal transforms known as generalized Walsh transforms [6]. In this section we will give a brief description of the techniques used to factorize the Walsh-Hadamard transformation matrix resulting in fast algorithms for its implementation. In the next two chapters we will show that the one- and two-dimensional FFT processor architecture are also suitable to compute the WHT and generalized Walsh transforms in real-time.

Fino and Algazi [4] and Geadah and Corinthios [22] have described techniques to factorize the transformation matrices of various ordering of the WHT. In [4] Fino showed that the various ordering of the WHT can be expressed as a product of perfect shuffle operators and the

WHT transformation matrix in natural ordering. Geadah [22] used the factorization techniques similar to those used in the development of OIOO FFT algorithm of section 2.3.1 and suggested a wired-in serial processor architecture for its implementation where a permutation stage is performed after a stage of additions and subtractions. In this section we describe briefly the factorization of the WHT transformation matrix proposed by Geadah [22], and in the next chapter we will show that the permutation and computational stages of the dyadic-ordered WHT can be merged into a single stage which results in a higher processing speed than that obtainable from the processor architecture described in [22].

2.4.1 Dyadic-Ordered Fast WHT Algorithm and its Implementation

The algorithms for the computation of various ordering of WHT are based on the Hadamard matrix. A Hadamard matrix of order N can be obtained by the following recursive relation:

$$(H_N)_h = (H_{N/2})_h \otimes H_2 \tag{2.26}$$

where $(H_N)_h$ is a Hadamard or natural-order WHT transformation matrix and H_2 is the core matrix of order 2 which is the lowest order Hadamard matrix:

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.27}$$

By using the relation given by (2.26) recursively we can write

$$(H_N)_h = [H_2]^{\otimes \log_2 N} \tag{2.28}$$

where $[\cdot]^{\otimes n}$ indicates n successive Kronecker products. In [22] Geadah has shown that the Dyadic-ordered WHT transformation matrix $(H_N)_d$ can be written in terms of $(H_N)_h$ as

$$(H_N)_d = \{ \prod_{i=1}^{n} P_2(n-i+1) \otimes I_{2^{i-1}} \} (H_N)_h \tag{2.29}$$

where $n = \log_2 N$. The transformation matrix $(H_N)_h$ can be further factorized into a product of sparse matrices resulting in the following factorization of $(H_N)_d$ [22]:

$$(H_N)_d = \prod_{i=1}^{n} (I_{2^{(n-i)}} \otimes P_{2^i}) \cdot C \tag{2.30}$$

where the operator C specifies addition and subtraction operations on the input data and is given by

$$C = (I_{N/2} \otimes H_2) \tag{2.31}$$

P_{2^i} is a base-2 perfect shuffle operator of dimension $(2^i \times 2^i)$ and I_k is the identity matrix of dimension k.

From equation (2.30) we see that the factorization of $(H_N)_d$ is of the same form as that of DFT transformation matrix T_M of equation (2.17). Comparing equation (2.17) and (2.30) we see that

$$C = s^{(2)} = (I_{N/2} \otimes T_2) \tag{2.31}$$

and

$$p_i^{(2)} = (I_{2^{n-i}} \otimes P_{2^i}) \tag{2.32}$$

where

$$T_2 = H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.33}$$

Thus equation (2.30) can be written as

$$(H_N)_d = \prod_{i=1}^{n} p_i^{(2)} s^{(2)} \tag{2.34}$$

For the implementation of equation (2.34), Geadah [22] suggested a sequential processor where the addition/subtraction and permutation operations specified by the operators $s^{(2)}$ and $p_i^{(2)}$, respectively, are performed sequentially. Fig. 2.3 shows a block diagram of the WHT processor where the input buffers A and B and output buffers C and D consist of serial-access memory, each of size N/2. The addition/subtraction operations specified by $s^{(2)}$ are performed by accessing the data from the buffers A and B in sequential order and the result of these addition/subtraction operations is shifted into the buffers C and D serially. This process is repeated for every pair of data points in the buffers A and B. After a stage of addition/subtraction, the contents of the buffers C and D are fed back into the buffers A and B as specified by the operator $p_i^{(2)}$. Thus a stage consists of the operations of addition/subtraction followed by a permutation phase. The algorithm specified by (2.34) is implemented by repeating the above process n time.

From the above discussion we note that, for each stage of the WHT algorithm, the feedback and computational phases require (N.T)ns and

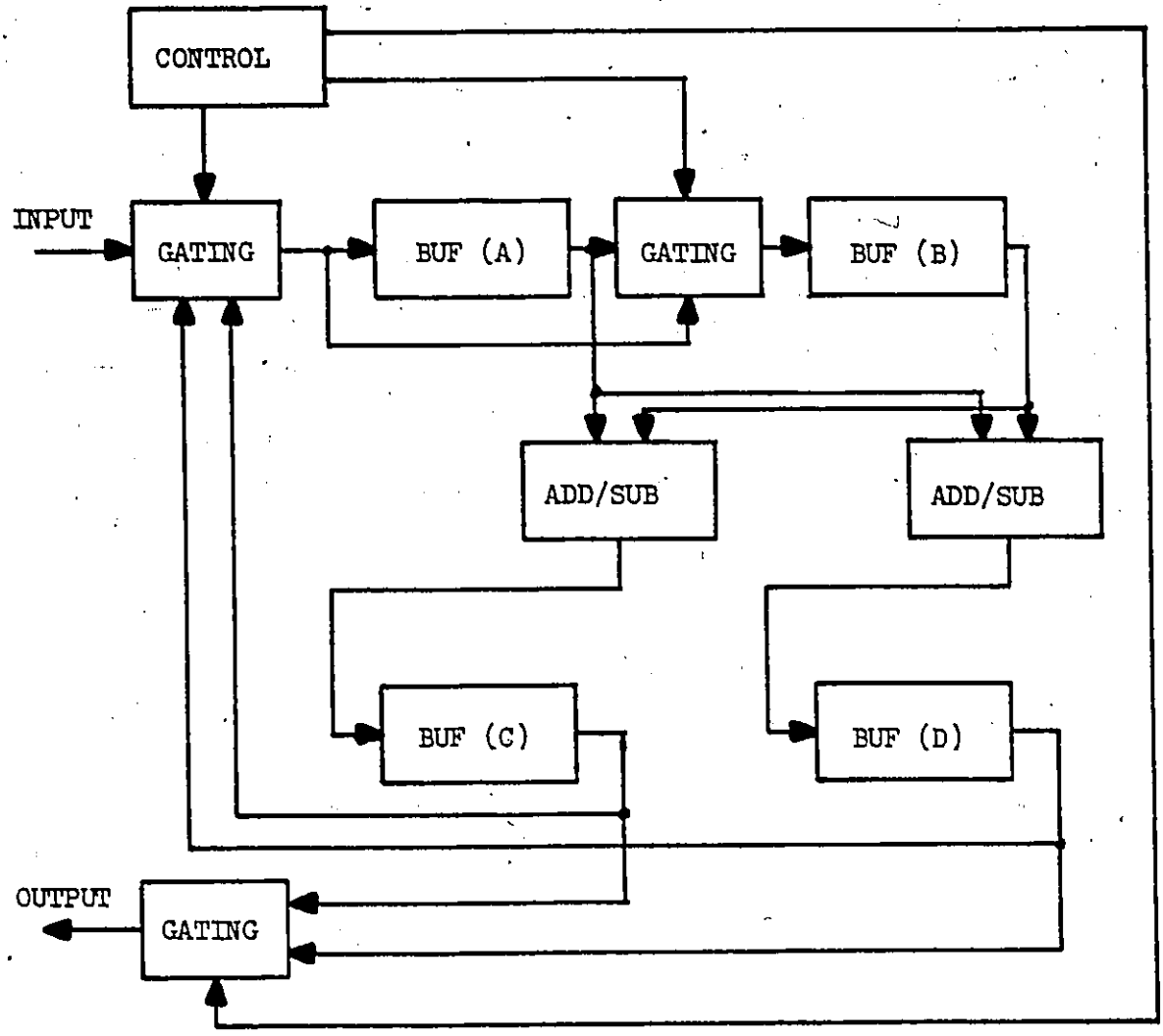


Fig. 2.3 A SEQUENTIAL WHT PROCESSOR

$(\frac{N}{2} \cdot T)$ns, respectively, where T is the data access time of the buffers A, B, C and D. For example, for N=1024 and T=70 ns, the permutation and computational phase requires 71.68 μs and 35.84 μs, respectively. Thus the time required for the permutation phase is twice the time required for the computational phase. In the next chapter we will show that the permutation and computational phases can be merged together resulting in an increase in the throughput rate by a factor of three.

2.5 RESIDUE NUMBER SYSTEM

In the past, most of the research work concentrated on the use of the binary number system (fixed point and floating point) in the hardware realization of signal processing algorithms. In most of the digital signal processing algorithms, the multiplication operation is the most time consuming and many forms of binary multiplier structures have been proposed to reduce the multiplication time [14], [23], and [28]. Also the cost of the high speed binary multipliers tends to be very high compared to the cost of adders and subtractors of equivalent speed.

Recently, the residue number system (RNS) has received considerable attention for the implementation of digital signal processing elements. The main advantage of the RNS, compared to the binary number system, lies in the fact that the operations of addition, subtraction and multiplication, within a given dynamic range, can be performed using N independent and parallel paths and also the dynamic range may be changed by varying the number of parallel paths. With the recent advances in

high density memories, the operations of addition, subtraction and multiplication may be performed at very high speeds by the use of ROMs to provide parallel arrays of look-up tables [39].

To take advantage of the potentially high processing speeds offered by the RNS, Jenkin and Leon [42] investigated RNS techniques for the implementation of non-recursive filters, while Soderstrand [43] and Jullien [39] considered different realizations of recursive filters. A ROM oriented implementation of the DFT and Number theoretic transforms have been described by Tseng, Jullien and Miller [40], Huang and Taylor [41] and Baraniecka and Jullien [25].

From the above mentioned works and the works of various other investigators in the field, the conclusion that emerges is that RNS is becoming an attractive and useful tool for the implementation of digital signal processors. In this work also, we propose to use RNS principles and techniques to obtain very high speed unitary transform processor architectures. In the following chapters we show that, with the use of RNS techniques, the throughput rates obtainable from the architecture of one- and two-dimensional FFT processors are comparable to that of cascade/pipeline and multi-processor architectures. In this section a brief discussion of RNS concepts is given.

2.5.1 Residue Number System Concepts

A number in the RNS is represented by the L-tuple

$$X = (x_0, x_1, x_2, \ldots, x_{L-1}) \tag{2.35}$$

with respect to the set of moduli $(m_0, m_1, \ldots, m_{L-1})$, where $x_i = X$

modulo m_i is the ith residue and is formally written as $x_i = |X|_{m_i}$. When all the moduli in the set are relatively prime, the range of numbers, M, that can be uniquely represented in the RNS is given by the product of the moduli, i.e.

$$M = \prod_{i=0}^{L-1} m_i \tag{2.36}$$

In the RNS, the binary operations of addition and multiplication between two variables, X and Y, can be performed by independent operations on their respective residues, i.e.

$$Z = |X \circ Y|_M \text{ implies } z_i = |x_i \circ y_i|_{m_i} \tag{2.37}$$

where $\circ$ denotes addition or multiplication. This property of the RNS allows the implementation of addition and multiplication operations in L parallel paths. In contrast to this, a weighted number system requires the propogation of interdigit carrys for addition and summation of partial products developed during the multiplication operations. The computing time in a weighted number system increases with the increase in the number range, whereas in the RNS, the number range can be increased or decreased by adding or deleting a parallel path without affecting the computing time.

For the representation of signed numbers in the RNS, a system analogous to the 2's complement binary number system is used. In the number range, M, the range 0 to $\frac{M}{2}-1$ for M even or 0 to $\frac{M-1}{2}$ for M odd is assigned to the positive numbers, and the range M/2 to M-1 for M even

or $\frac{M+1}{2}$ to M - 1 for M odd is assigned to the negative numbers. In such a representation the additive incerse of a number X is given by $\overline{X} = M - X$ and $\overline{x_i} = m_i - x_i$ for each residue. Also this representation preserves the rules of signed arithmetic and the explicit knowledge of the sign of a number is not required to perform the operations defined by (2.37). Since, in the RNS, sign determination is a difficult operation, it is essential to use such a number representation for signed numbers.

General division is a complicated process in RNS [44]. Scaling is easier than a general division but in scaling the diviser is limited to a predetermined constant. For efficient implementation of scaling, the constant scale factor must be a product of some of the moduli [39], [44] or a power of 2 [41].

For the implementation of arithmetic operations in hardware, two basic approaches have been suggested [44]. The first approach involves the logical implementation of the Boolen functions specified by the operation and the second approach requires storing of all the possible outcomes of the operation in look up tables. The second approach is becoming more and more attractive with the recent advances in memory technology and reduction in cost, also it offers the best solution for high speed realizations. A memory size of $(2^{2B} \times B)$ bits is required to store the look up table for a modulus of B bits. For a large modulus this requirement tends to be very high but if the modulus is limited to six bits, currently available single chip memory packages can be used to store a look up table. For example, binary operations for moduli

$m_i \leq 64$, 32 and 16 can be implemented using (4kx8), (1kx8) and (256x4) bit commercially available ROM packages, respectively.

Another advantage of the look up table approach is that it allows easy pipelining of the arithmetic operations. For example, Fig. 2.4 shows the implementation of a function $\left| |a \cdot b|_{31} + |c \cdot d|_{31} \right|_{31}$. For every latch pulse, the output of each ROM is stored and becomes part of the address for the next ROM. A new input is thus presented and a result is generated for each ROM in one ROM access time plus the settling time of the latch. The only control signal required to clock the pipeline is a latch pulse. The look up table implementation can also provide a saving in hardware when some of the operands are known constants. For example, the ROM structure of Fig. 2.4 can also compute the function $\left| |a \cdot b|_{31} \cdot A + |c \cdot d|_{31} \cdot B \right|_{31} \cdot C + D \Big|_{31}$ without additional hardware when A, B, C and D have constant values.

In order to interface conventional binary structures, it is necessary to convert into and out of the RNS representation. A binary to residue conversion (B/R) can be easily implemented using ROM arrays [39]. For example, in an L-moduli RNS, the ith residue of a number X is given by

$$x_i = \left| \sum_{j=0}^{B-1} b_j 2^j \right|_{m_i} \tag{2.38}$$

where b_j is the jth bit of the binary representation of X. For $B \leq 12$ bits and $m_i \leq 256$, equation (2.38) can be implemented with a ROM package of 4kx8 bit. When $B > 12$, the sum in (2.38) can be split into

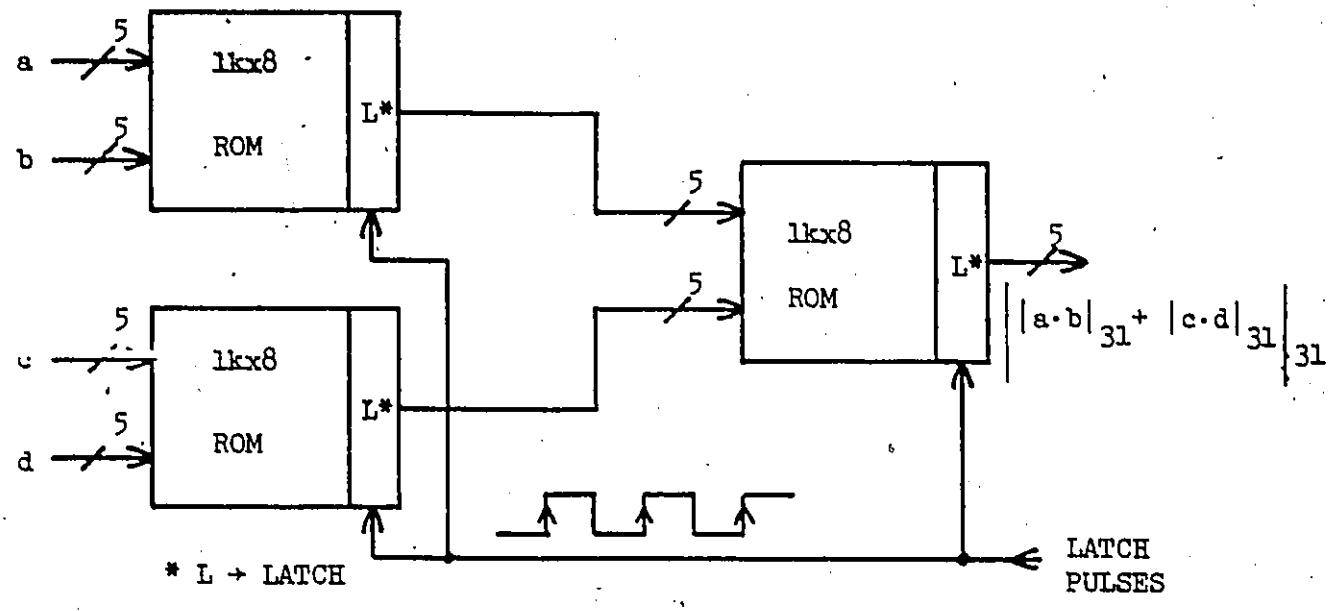


Fig. 2.4 A PIPELINE ROM ARRAY

several sections and the sum in each section modulo m_i can be computed with a single ROM, i.e., equation (2.38) can be written as

$$x_i = \left| \left| \sum_{j=0}^{\frac{B}{2}-1} b_j 2^j \right|_{m_i} + \left| \sum_{j=B/2}^{B-1} b_j 2^j \right|_{m_i} \right|_{m_i} \tag{2.39}$$

The result of the individual summation terms may be combined by summing in residue code. For $m_i \leq 64$, and $B \leq 24$, x_i can be computed using 3 4kx8 bit ROMs.

The conversion from residue to binary (R/B) is more difficult than B/R. Two general techniques based on the Chinese Remainder Theorem and mixed-radix representation [44] are normally used for the R/B conversion. Using the Chinese Remainder Theorem, a number X in the range 0 to M-1 is given by

$$X = \left| \sum_{i=0}^{L-1} \hat{m}_i \left| \frac{x_i}{\hat{m}_i} \right|_{m_i} \right|_M \tag{2.40}$$

where $\hat{m}_i = \frac{M}{m_i}$, $M = \prod_{i=0}^{L-1} m_i$ and $\left|\frac{1}{\hat{m}_i}\right|_{m_i}$ is the multiplicative inverse of $|\hat{m}_i|_{m_i}$. The hardware implementation of (2.40) results in a relatively slow serial implementation [42] and requires a modulo M adder which for large values of M is difficult to implement. For high speed implementation, the mixed-radix technique is normally used for the parallel computation of X. In mixed-radix form, a number in the range 0 to M-1 has the representation

$$X = \sum_{i=0}^{L-1} a_i \, p_i \tag{2.41}$$

where $p_0 = 1$, $p_i = \prod_{k=0}^{i-1} m_k$ and the $\{a_i\}$ are mixed-radix digits in the range $0 \leq a_i < m_i$. The $\{a_i\}$ can be generated using look up tables [39] and equation (2.41) can be implemented by forming the partial products $\{a_i \, p_i\}$ using ROMs. This scheme does not require a Modulo M adder since the summation can be performed with a binary adder.

2.6 SUMMARY

This chapter has covered the basic concepts of unitary transforms and techniques used to develop fast one-dimensional unitary transform algorithms. For the implementation of the discrete Fourier and Walsh-Hadamard transforms we have shown that the transformation matrices of these transforms can be factorized into a product of a set of sparse matrices. Based on these factorizations high speed memory and processor architectures for the implementation of these algorithms were discussed.

A brief introduction to Residue Number System concepts was presented and the hardware implementation of RNS arithmetic operations using high speed ROM arrays were also discussed.

CHAPTER 3

RESIDUE-NUMBER SYSTEM BASED ONE-DIMENSIONAL PROCESSOR ARCHITECTURES

3.1 INTRODUCTION

In the previous Chapters we discovered that the Residue-Number System (RNS) has a unique advantage over the Binary-Number System in the implementation of the binary operation of addition, subtraction and multiplication. In the RNS, these operations can be performed using N independent and parallel paths, where N is the number of moduli, resulting in high operating speeds. Another advantage of the RNS is its adaptability to look-up table implementation and, as discussed in the last Chapter, the look-up table approach allows easy pipelining of the arithmetic operation which also results in high execution speeds. Also, due to recent advances in high-density memory technology and reduction in cost, the use of RNS has become increasingly attractive for the implementation of digital signal processing algorithms.

With these considerations we present a novel special purpose FFT processor architecture. The chapter format is as follows: we first present the architecture of the Butterfly unit of the FFT processor, described in the last chapter, using the RNS principles and techniques and show that the pipeline ROM array architecture of the RNS-based Butterfly unit ideally lends itself to the memory architecture of the

FFT processor of Fig. 2.2. Based on this RNS-based Butterfly unit we then propose a modification of the processor architecture described by Corinthios et al. [15] and show that the RNS-based FFT processor is capable of processing complex input signals at a sampling rate of over 10 MHz in real-time as compared to the 1.6 MHz sampling rate obtainable from the processor proposed in [15]. We also propose architectural extensions of the basic RNS-based FFT processor to process data from a number of input channels and to perform filtering of a real-valued input signal. At the end of this chapter we present modification of the dyadic-ordered WHT algorithm of section (2.4.1) and show that this algorithm can also be implemented on a FFT processor.

3.2 AN RNS-BASED BUTTERFLY UNIT'S ARCHITECTURE

For the implementation of an RNS-based Butterfly unit, the dynamic range requirement of the FFT processor must be determined first. A viable RNS dynamic range, M, can be obtained by choosing a set of L relatively prime moduli. The arithmetic operations, within the dynamic range M, can be computed by providing L independent Butterfly units, one for each modulus. In [39] it is shown that a dynamic range of 28 bits may be obtained by implementing an RNS with 6 moduli, which is sufficient for most of the signal processing applications. By choosing the set of moduli $\{m_i\}$ as 32, 31, 29, 27, 25 and 23, a high speed ROM oriented implementation of the Butterfly unit can be obtained.

The arithmetic operations performed by the Butterfly unit of a radix-r FFT processor are given by the operator T_r of equation (2.16).

For a radix-4 implementation T_4 is given by

$$T_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \tag{3.1}$$

where the elements of T_4 are the actual values and not the exponent of W as represented in (2.16). Using equation (2.17), T_4 can be factored in terms of the radix-2 operator, T_2, as

$$T_4 = P_2^{(2)} \, s^{(2)}, \, \mu_2^{(2)} \, s^{(2)} \tag{3.2}$$

where

$$T_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.3}$$

In the matrix form, equation (3.2) can be written as

$$T_4 = \begin{bmatrix} 1 & 1 & & \\ & & 1 & 1 \\ 1 & -1 & & \\ & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & -j \end{bmatrix} \begin{bmatrix} 1 & & 1 & \\ & 1 & & 1 \\ 1 & & -1 & \\ & 1 & & -1 \end{bmatrix} \tag{3.4}$$

$$P_2^{(2)} \, s^{(2)}, \qquad \mu_2^{(2)} \qquad s^{(2)}$$

which shows that T_4 can be implemented using 4 complex adders and 4 complex subtractors, because the multiplication of j just changes the real part of a number to imaginary part and vice versa.

Since the RNS is an integer number system, all non-integer coefficients in the FFT must be converted to integer values. This can be done by premultiplying the non-integer twiddle factors with a predetermined integer conversion factor, P. A simplified block diagram of an RNS-based Butterfly unit, for the modulus m_i, implementing operations specified by T_4 and the multiplications by the twiddle factors is shown in Fig. 3.1 where $\{x_i\}$ and $\{y_i\}$ are the residues of the four inputs and outputs, respectively, P is the integer conversion factor and $[\cdot]_R$ denotes the rounding operation. To generate $|y_1|_{m_i}$, the multiplier for the multiplication of $|P|_{m_i}$ is not required. Since P is a constant, it can be premultiplied with $|v_1|_{m_i}$ and we generate $|y_1|_{m_i}$ instead of $|v_1|_{m_i}$. A ROM implementation of the radix-4 Butterfly unit is shown in Fig. 3.2. For simplicity we have shown a variable $|a_i|_{m_i}$ as a_i. W_1', W_2' and W_3' represent the values of $|[W^t \cdot P]_R|_{m_i}$, $|[W^{2t} \cdot P]_R|_{m_i}$ and $|[W^{3t} \cdot P]_R|_{m_i}$ respectively. From Fig. 3.2 it is clear that arithmetic operations within the Butterfly unit can be performed in a pipeline by using the structure of Fig. 2.4. If the structure of Fig. 3.2 is divided into a 4-stage pipeline, an increase in throughput rate by a factor of four may be obtained. Pipelining of this structure becomes even more attractive by the availability of a family of registered PROMs (Monolithic Memories series 63RAxxx) [49] thereby avoiding the need to provide extra storage for the

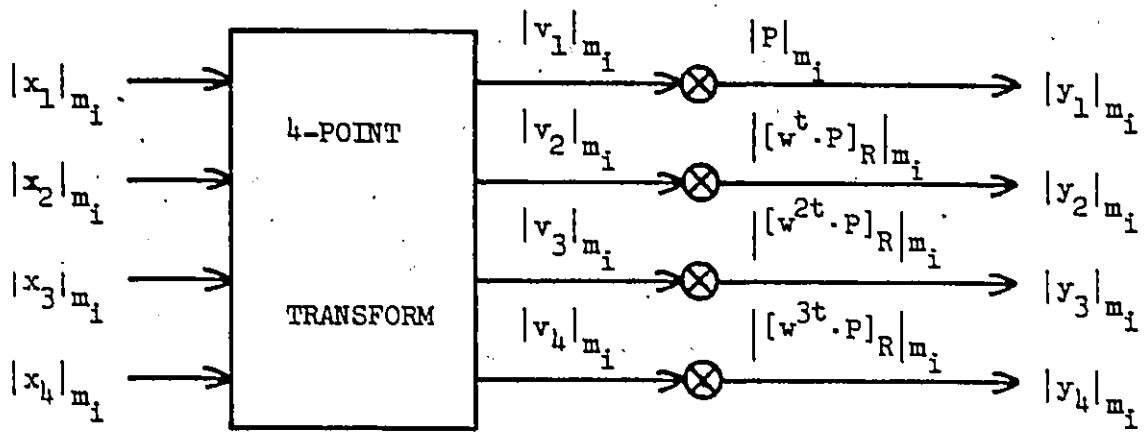


Fig. 3.1 A SIMPLIFIED BLOCK DIAGRAM OF A RNS-BASED BUTTERFLY UNIT FOR THE MODULUS m_i.

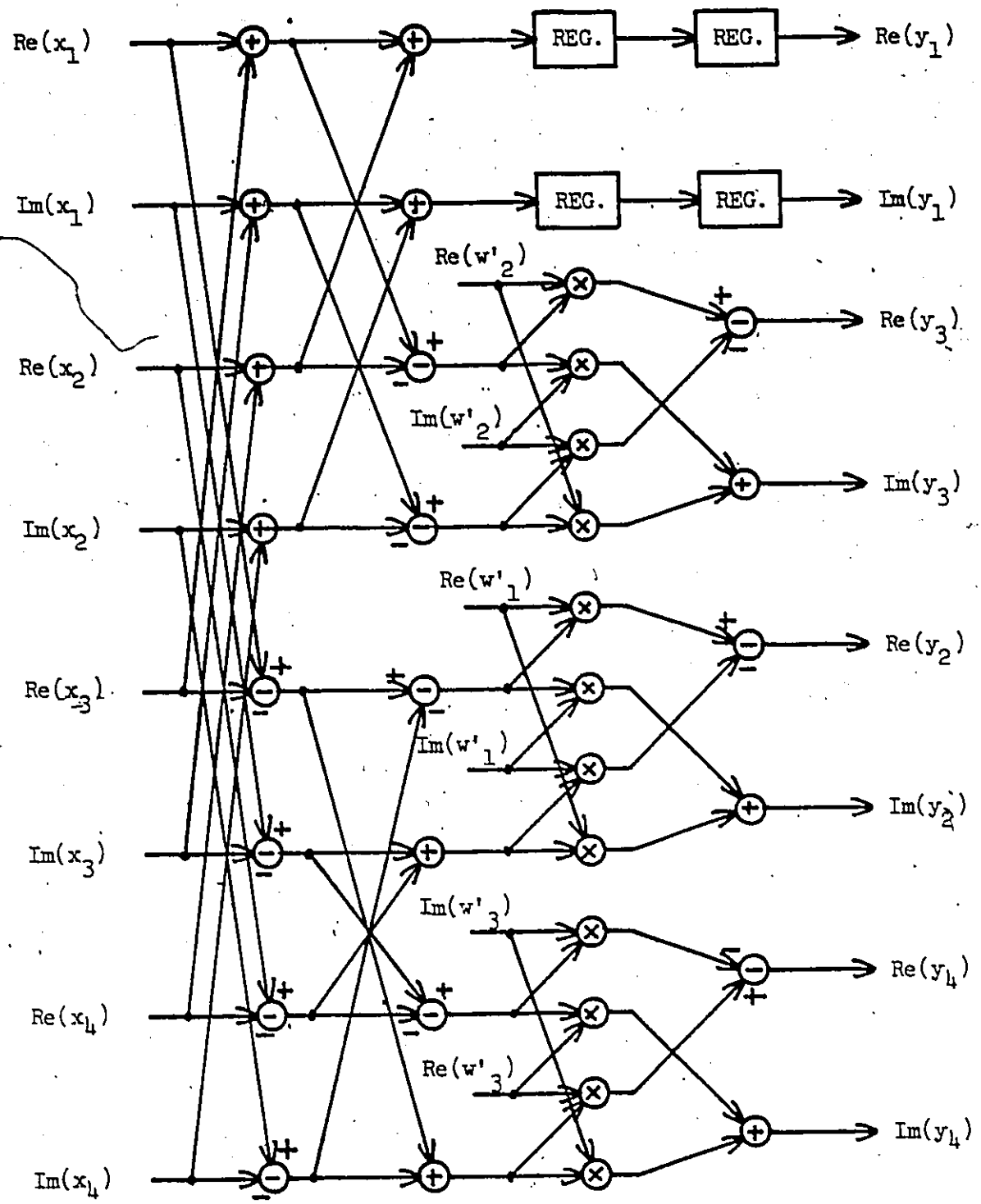


Fig. 3.2 A ROM IMPLEMENTATION OF THE RADIX-4 BUTTERFLY UNIT FOR THE MODULUS m_i.

(*) INDICATES A ROM PERFORMING THE OPERATION *

a, b → c INDICATES $c = |a-b|_{m_i}$

pipeline operations.

In the RNS scaling is important, since we are not interested in computation over the finite ring of integers, Z_M; rather, we wish to obtain an approximation to the calculations performed over the infinite field of real or complex numbers. This may be achieved by scaling down the output of the Butterfly unit by a constant scale factor. In [40] Tseng et al. have developed a design procedure to implement a number of scaling schemes by minimizing the error in the output of the FFT algorithm . To minimize the hardware requirement and to simplify the processor control functions, we choose the scaling scheme that scales down the output of the Butterfly unit at each stage by a pre-determined scale-factor. The selection of the scale factor depends upon the desired error in the FFT output, the dynamic range, M, and the integer conversion factor, P. The scale factor can be determined by the procedure outlined in [40].

Various schemes to scale down a number by a constant scale factor have been proposed [39], [41]. When the scale factor is a product of some of the moduli, Jullien [39] has described a scaling algorithm using ROM arrays. In [41], Huang and Taylor proposed a combination of ROM and adder structures for scaling when the scale factor is a power of 2 and relatively prime to the set of moduli $\{m_i\}$. Although the scheme described in [41] seems to offer greater flexibility in choosing a scale factor, it trades processing speed with flexibility and, in general , has slower processing speed than the scaling scheme described in [39].

Since the scaling scheme described in [39] is easy to pipeline and, as shown later, results in a considerable saving in buffer memory, we choose this scaling scheme to scale down the output of the Butterfly unit.

Fig. 3.3 shows the flow-graph of the Original scaling algorithm for a 6 moduli RNS with 3 scaling moduli (N=6, s=3), where O indicates a ROM storing the result of the arithmetic operations specified by the scaling algorithm. From Fig. 3.3 it is clear that the computation specified by the scaling algorithm may be performed in a N-stage pipeline and when combined with the pipeline of Fig. 3.2, an (N+4)-stage pipeline having a throughput rate of the inverse of one ROM access time is obtained. Also note that the scaling algorithm first computes the (N-s) residues of the scaled output y for m_i: $s \leq i \leq N-1$ and then, using these residues, the other s residues for m_i: $0 \leq i \leq s-1$ are computed using the base-extension techniques [39]. Since the base-extension part of the scaling algorithm requires only the residues of y for m_i: $s \leq i \leq N-1$, the base extension part may be separated from the rest of the pipeline. This separation leads to a significant savings in memory since we do not have to store the residues of y for m_i: $0 \leq i \leq s-1$ and these residues may be generated when needed.

The operator $\mu_i^{(r)}$ of equation (2.18) specifies multiplication of the output of the r-point transform operations by the twiddle factors. These twiddle factors may be pre-computed and stored in ROMs. For example, for a radix-4 implementation, 3 ROMs storing the twiddle factors

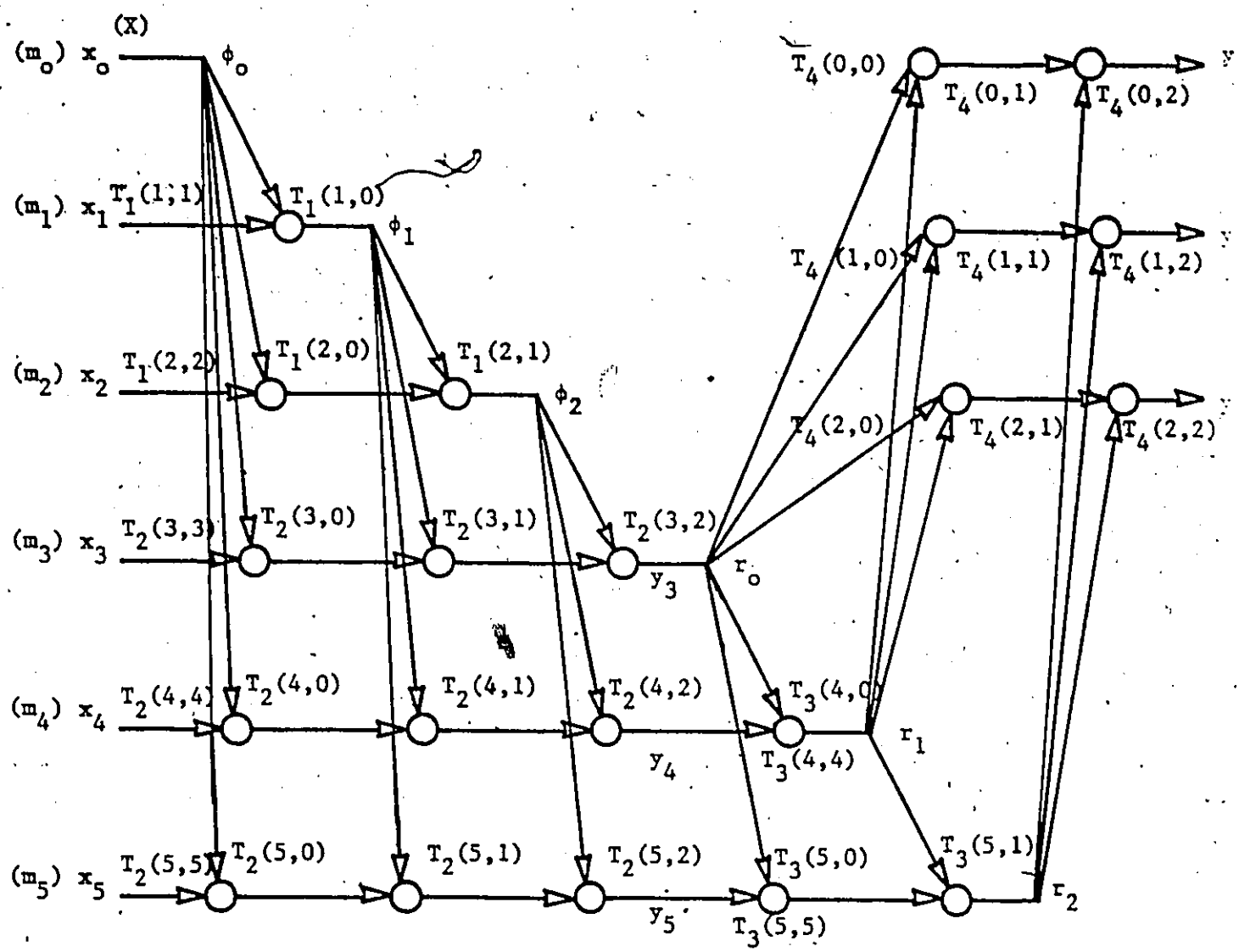


Fig.3.3- Original Scaling Algorithm for N = 6 and S = 3

W_1', W_2' and W_3' of Fig. 3.2 may be used. Each ROM stores the twiddle factors in the sequence shown in Fig. 3.4, starting from address '0'. To access the twiddle factors from the ROMs, we set up a counter i where $i=0,1,2,\ldots, \frac{M}{4}-1$ which is cleared to zero at the beginning of every stage. If i is incremented by one after every butterfly operation then correct twiddle factors for the 1st FFT stage are obtained by addressing the 3 ROMs with i. The twiddle factors for the other FFT stages may be obtained by a masking operation. For the mth stage we set 2(m-1) least significant bits of i to zero and then address the ROMs to access the twiddle factors. Thus the twiddle factors' addresses for all the FFT stages may be generated by a very simple control function.

3.3 A ONE-DIMENSIONAL RNS-BASED FFT PROCESSOR ARCHITECTURE

In the last section we saw that the addition, subtraction, multiplication and scaling operations specified by a radix-4 FFT butterfly can be performed in an (N+4)-stage pipeline with a throughput rate of the inverse of one ROM access time where N is the number of moduli in the RNS. In general it can be shown that by the use of the RNS a general radix-r FFT butterfly can also be implemented in a pipeline structure. Since in the basic architecture of the FFT processor of Fig. 2.2., the butterfly operations in a stage are performed sequentially by accessing the r data points from one buffer and storing the r outputs of the previous butterfly into the other buffer, simultaneously; the ROM-array pipeline architecture of the Butterfly unit fits directly into the memory architecture of the FFT processor. Thus during every clock cycle the r

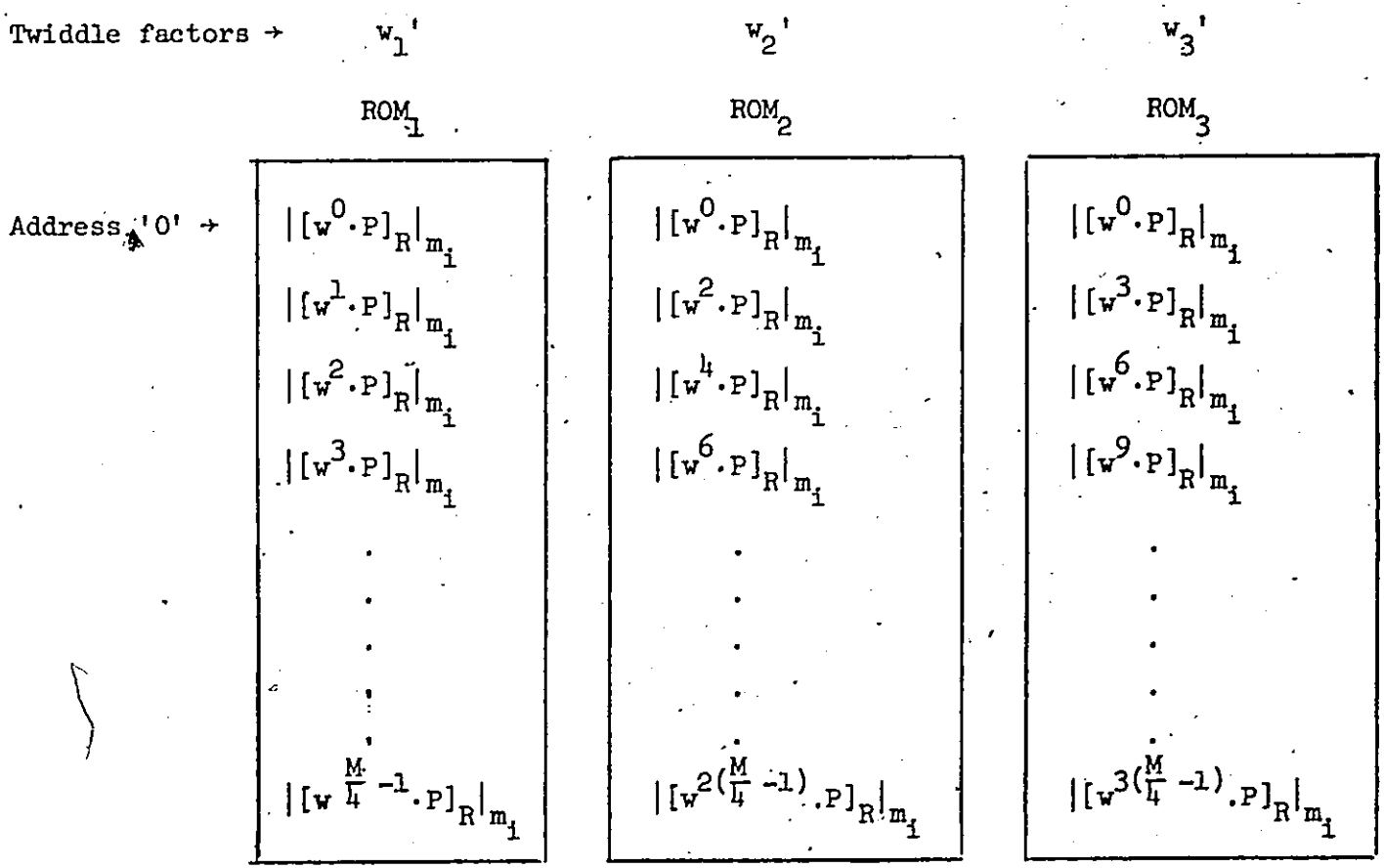


Fig. 3.4 ROMs STORING THE TWIDDLE FACTORS FOR A RADIX-4 FFT PROCESSOR

data points can be shifted into and out of the Butterfly unit's pipeline. Although the pipeline will cause a delay between an input and its corresponding output, an effective throughput rate of the inverse of one ROM access time is obtained. When the data from the memory buffers is accessed at the throughput rate of the Butterfly unit's pipeline, the result of a butterfly operation is obtained in every pipeline clock cycle.

The basic memory architecture of Fig. 2.2 is suitable only for the case of non real-time inputs. For real-time input and output operations Corinthios et al. [15] suggested the use of two extra buffers for real-time input and output operations. In this thesis we show that a three buffer architecture is capable of performing the DFT and convolution operations in real-time since the input and output of the OIOO FFT algorithm are in the natural ascending order. In the next section we will show that the requirement of four buffers arises only when the input data from a number of input channels must be processed.

A three buffer memory architecture of a RNS-based FFT processor is shown in Fig. 3.5. In this organization, two of the three buffers store the input and output of the intermediate FFT stages, the third buffer is used to collect the sampled input and output the previously computed Fourier coefficients simulatneously.

In the beginning of a transform, if BUF1 stores the output of the previous transform, then BUF1 is connected to form a long shift register of size M as shown in Fig. 3.5b. The distributor routes the vector f

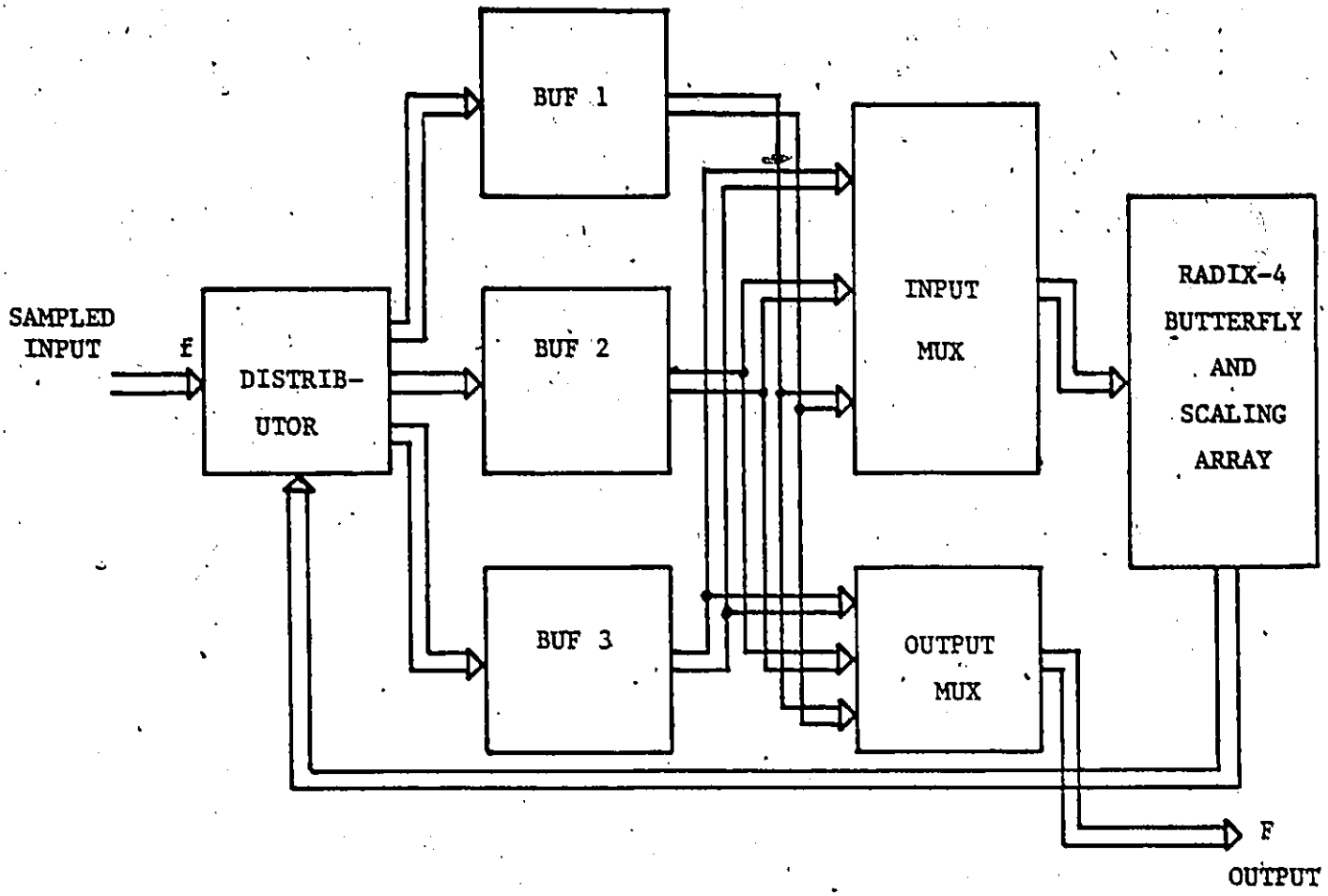


Fig. 3.5a - Organisation of a RNS-based 1-D FFT Processor

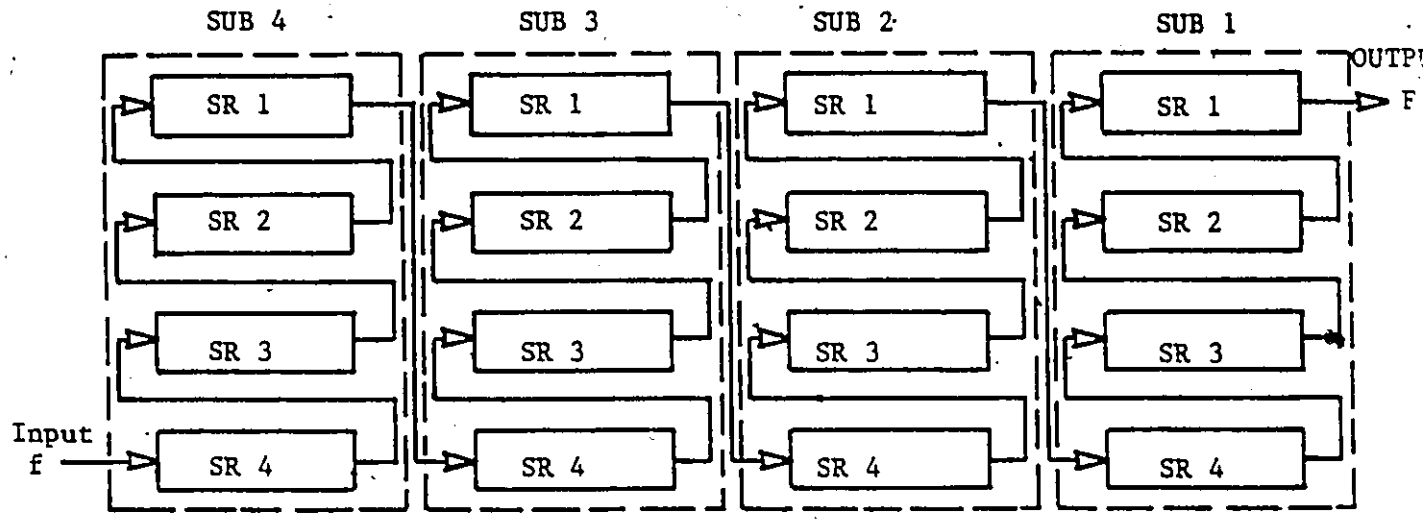


Fig.3.5b - Interconnections of SUBs of BUF1 to form Shift-Register of size N

to the input of BUF1 and the output MUX gates vector F to the output. Thus, during the computation of a transform, the new data for the next transform is shifted in and the result of the previous transform is shifted out simultaneously. At the start of a new transform the buffer selection is changed and the input, output and transform operations are repeated for the next set of input data. Fig. 3.6 shows the buffer allocation for real-time processing of a continuous input.

3.3.1 System Organization of a RNS-Based FFT Processor

In the previous sections we discussed the organization of various memory units, implementation of the RNS-based Butterfly unit and the generation of twiddle factors. Fig. 3.7 shows the interconnection of the various hardware units of the FFT processor for a 6 moduli RNS with 3 scaling moduli. The processor is capable of computing the DFT of a continuous input in blocks where a block contains M-points.

At the start of a new block of data, the input from the A/D is converted into residue code and the residues are stored into the selected memory buffers provided for each residue. When the input vector f has been accumulated, the selected memory buffers contain the residues of M samples ready for processing. At the beginning of a new transform, the buffer allocation is changed as shown in Fig. 3.6, so that the residues for the next transform can be stored in the other unused buffer.

The buffers, storing the residues of the input vector f, are selected and the input MUX gates the residue (r_3, r_4, r_5) corresponding

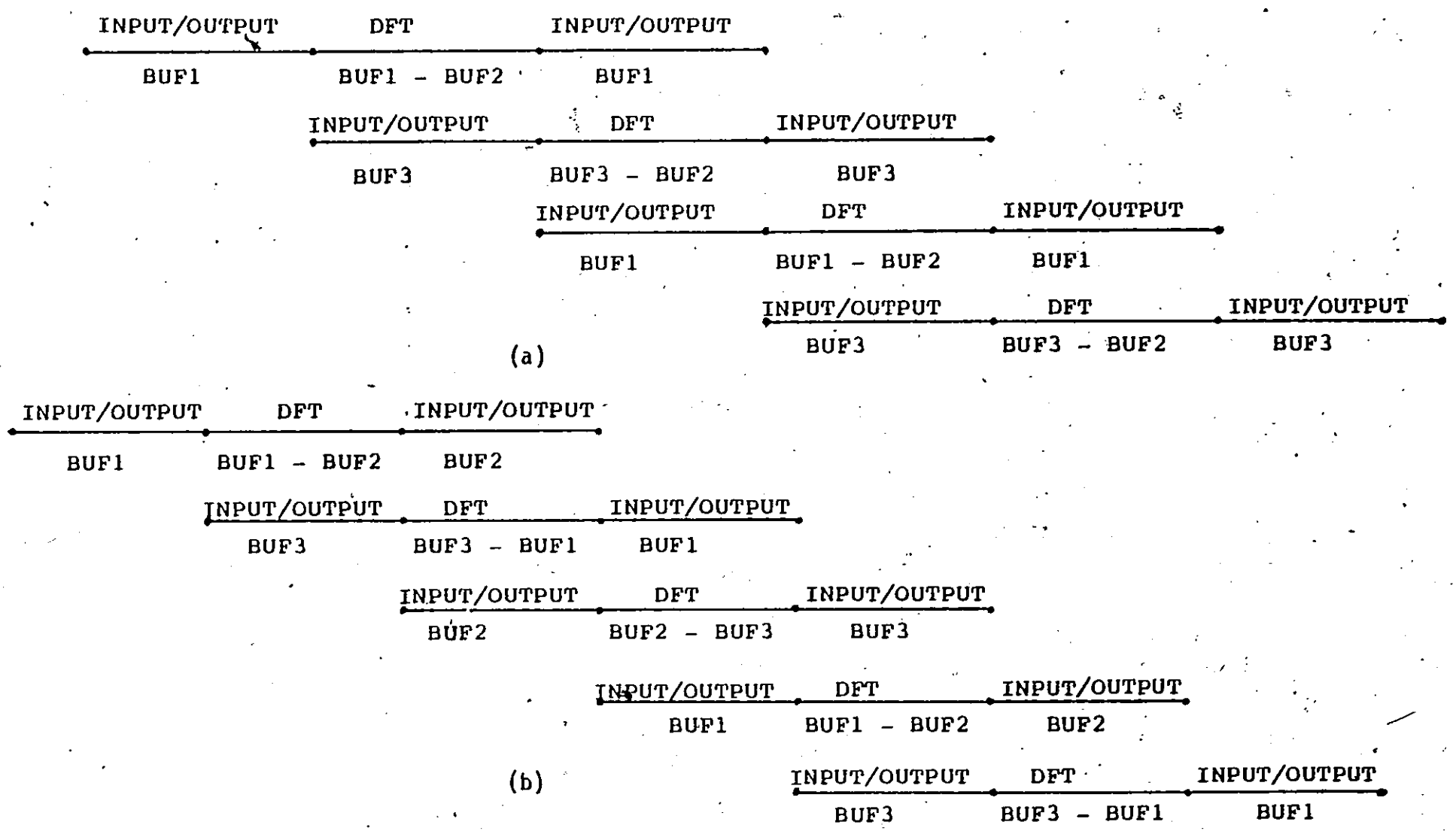


Fig. 3.6 Buffer selection for real-time computation of a transform when the number of stages in the FFT are (a) even (b) odd

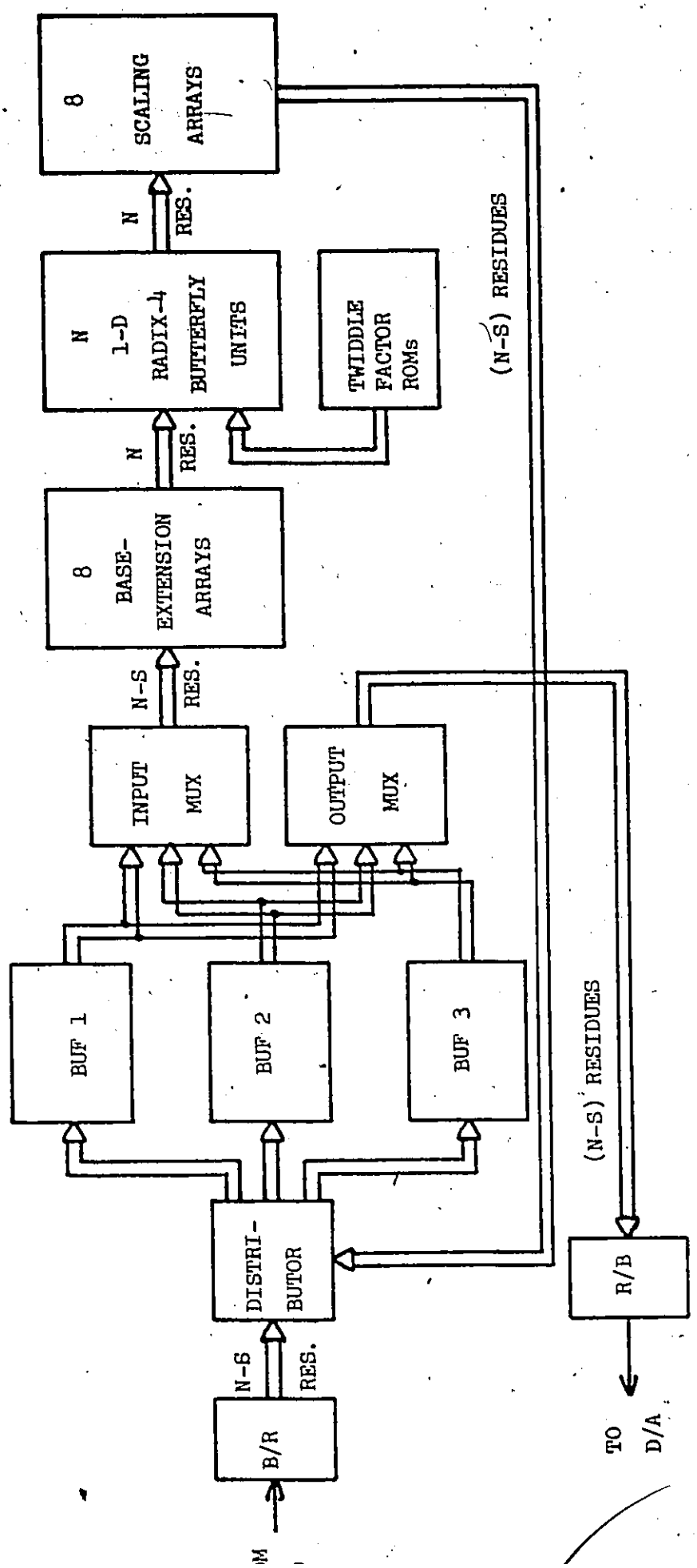


Fig. 3.7 SYSTEM ORGANIZATION OF A RNS-BASED 1-D RADIX-4 FFT PROCESSOR

to the four input points of the next butterfly operation to the base-extension array, which generates the other residues (r_0, r_1, r_2). The Butterfly units perform a 4-point transform and multiplication by the twiddle factors, on each residue of the four input points. The output of the Butterfly units is scaled down by the scaling arrays which generate the residues (r_3, r_4, r_5) of the four outputs. These outputs are stored into the intermediate result buffer for processing in the next stage. This process is repeated for other data points too. At the beginning of a new FFT stage the role of the input and intermediate result buffers is interchanged and the above process is repeated for each stage. At the end of the required number of stages, the residues corresponding to the output Fourier coefficients become available from the selected output buffers. At the beginning of the next transform, the buffer selection is altered in such a way (see Fig. 3.6) that the buffers storing the residues of the Fourier coefficients of the previous transform are connected to the R/B. Thus during the processing of a transform, the Fourier coefficients of the previous transform are generated by the D/A. A time delay, equal to the time required to compute an M-point transform, is generated between an input data block and its Fourier coefficients but that is of little importance for most applications.

Since the butterfly operations, base-extension and scaling are being performed in a pipeline with a throughput rate of 1/T where T is the ROM access time, and also the data from the memory buffers is accessed at the same rate, the time for computing the DFT of a M-point

complex sequence is given by

$$T = \frac{M}{r} \cdot T \cdot \log_r M \qquad (3.5)$$

For example, with T = 70 ns, a radix-4 FFT processor will take about 90 μs to compute the DFT of a 1024-point complex sequence. This gives a sampling rate of about 11 M·samples/sec. When the input data is real-valued, it is possible to process two data sequences simultaneously and hence the throughput rate of the FFT processor may be doubled for real-valued input sequences.

We may note that due to the pipeline architecture of the RNS-based Butterfly unit, a 1024-point radix-4 sequential processor will perform real-time processing of signals sampled at a rate of over 11 M·samples/sec as compared to a sampling rate of 1.6 M·samples/sec obtainable from the same by using a Binary-number system [15]. This throughput rate is also comparable to that obtainable from cascade/pipeline organizations [24], discussed in the literature, which involve a greater degree of parallelism than required in the architecture proposed here.

3.4 PROCESSOR ORGANIZATION TO OBTAIN THE DFT OF SMALLER SEQUENCES

In many applications of a special purpose FFT processor, such as radar and sonar, it is required to process data from a number of independent channels [18]. The memory architecture of the FFT processor discussed earlier, which is designed to process data in blocks of M-points, can be modified to process data from r channels in blocks of M/r-points.

Let $f_1, f_2, \ldots, f_r$ amd $F_1, F_2, \ldots, F_r$ be the data and their DFT vectors, respectively, each of length M/r points. Let f and F be the vectors of length M defined by

$$f = [f_1, f_2, f_3, \ldots, f_r]^T \tag{3.6a}$$

and

$$F = [F_1, F_2, F_3, \ldots, F_r]^T \tag{3.6b}$$

where $[x]^T$ denotes the transponse of [x]. In this representation, the DFT of the vector f can be written as

$$\begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ \cdot \\ \cdot \\ \cdot \\ F_r \end{bmatrix} = \begin{bmatrix} T_N & & & & & \\ & T_N & & & & \\ & & T_N & & & \\ & & & \cdot & & \\ & & & & \cdot & \\ & & & & & T_N \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \cdot \\ \cdot \\ \cdot \\ f_r \end{bmatrix} \tag{3.7}$$

where N=M/r and T_N is the NxN DFT transformation matrix. Equation (3.7) can also be written in Kronecker product form as

$$F = (T_N \otimes I_r)\, f \tag{3.8}$$

Using the transformation [14]

$$P_M^{(r)}\, (T_{M/r} \otimes I_r)\, \{P_M^{(r)}\}^{-1} = I_r \otimes T_{M/r} \tag{3.9}$$

we can write (3.8) as

$$P_M^{(r)}F = P_M^{(r)}(T_{M/r} \otimes I_r)\{P_M^{(r)}\}^{-1}\{P_M^{(r)}\}f \tag{3.10}$$

or

$$P_M^{(r)}F = (I_r \otimes T_{M/r})\, P_M^{(r)}\, f$$

Defining $F' = P_M^{(r)}F$ and $f' = P_M^{(r)}f$ we get

$$F' = (I_r \otimes T_{M/r})f' \tag{3.11}$$

Since $M = r^{\ell}$, $M/r = r^{(\ell-1)}$. Using equation (2.17), the factorization of $T_{M/r}$ can be written as

$$T_{M/r} = \prod_{m=1}^{\ell-1} \mu_m^{(r)}\, s_m^{(r)} \tag{3.12}$$

Substituting (3.12) into (3.11) we get

$$F' = \{I_r \otimes (\prod_{m=1}^{\ell-1} \mu_m^{(r)}\, s_m^{(r)})\, f' \tag{3.13}$$

Using the following relationship between matrix and Kronecker products [16]:

$$(AB) \otimes (CD) = (A \times C)(B \otimes D) \tag{3.14}$$

We can rewrite equation (3.13) as

$$F' = \{\prod_{m=1}^{\ell-1} (I_r \otimes \mu_m^{(r)})(I_r \otimes s_m^{(r)})\}f' \tag{3.15}$$

From equations (2.18) to (2.23) we see that

$$I_r \otimes \mu_i^{(r)} = (I_r{}^{\ell-i} \otimes D_{ri}) = \beta_i \tag{3.16}$$

$$I_r \otimes s_i^{(r)} = I_r \otimes (p_i^{(r)} s^{(r)},)$$

$$= (I_r{}^{\ell-i} \otimes P_{ri}^{(r)})(I_r \otimes I_{N/r^2} \otimes T_r \otimes I_r)$$

$$= (I_r{}^{\ell-i} \otimes P_{ri}^{(r)})(I_{N/r} \otimes T_r \otimes I_r) \tag{3.17}$$

Since N=M/r we can write

$$I_r \otimes s_i^{(r)} = (I_r{}^{\ell-1} \otimes P_{ri}^{(r)})(I_{M/r^2} \otimes T_r \otimes I_r) \tag{3.18}$$

$$= R_i^{(r)}$$

Substituting equations (3.16) and (3.18) into (3.15) we get

$$F' = \{\prod_{m=1}^{\ell-1} \beta_m^{(r)} R_m^{(r)}\} f' \tag{3.19}$$

From equation (2.17) we can also write the factorization of T_M as

$$T_M = \prod_{m=1}^{\ell} \mu_m^{(r)} s_m^{(r)} \tag{3.20}$$

Comparing this factorization of T_M with the factorization of $(I_r \otimes T_{M/r})$ of equation (3.19) we see

$$s_m^{(r)} = R_m^{(r)}; \quad m=1,2,\ldots,\ell-2$$
$$\mu_m = \beta_m ; \quad m=1,2,\ldots,\ell-1 \tag{3.21}$$

Thus we note that if the vectors f and F are permuted as defined by the relations

$$F' = P_M^{(r)} F; \quad f' = P_M^{(r)} f \tag{3.22}$$

we can generate the DFT of r M/r-point sequences with an M-point FFT processor by processing only the last $(\ell-1)$ stages. The operators $R_{\ell-1}^{(r)}$ and $s_{\ell-1}^{(r)}$ can be made equal by using the relation

$$\{P_M^{(r)}\}^{-1} (I_{M/r} \otimes T_r) P_M^{(r)} = (I_{M/r^2} \otimes T_r \otimes I_r) \tag{3.23}$$

Since $R_{\ell-1}^{(r)} = I_{M/r} \otimes T_r$

$$= P_M^{(r)} \cdot \{P_M^{(r)}\}^{-1} (I_{M/r} \otimes T_r) P_M^{(r)} \cdot \{P_M^{(r)}\}^{-1}$$

Using (3.23) we get

$$R_{\ell-1}^{(r)} = P_M^{(r)} (I_{M/r^2} \otimes T_r \otimes I_r) \{P_M^{(r)}\}^{-1}$$
$$= P_M^{(r)} s^{(r)} \{P_M^{(r)}\}^{-1} \tag{3.24}$$

Substituting (3.24) into (3.19) we see that

$$s_m^{(r)} = R_m^{(r)}; \quad m=1,2,\ldots,\ell-1 \tag{3.25}$$

and the $\{P_M^{(r)}\}^{-1}$ cancels with the $P_M^{(r)}$ of f'. Thus we can write equation (3.19) as

$$F' = \{\prod_{m=1}^{\ell-1} \mu_m^{(r)} s_m^{(r)}\} f \qquad (3.26)$$

which can be implemented on an M-point FFT processor by executing the M-point FFT algorithm starting from the 2nd stage.

From equation (3.26) we see that f and F' specify a different permuation of input data and the output coefficients for the r channels resulting in a different organization of the processor's input and output buffers. Thus a three-buffer FFT processor organization is not suitable for processing r M/r-point sequences. One possible structure for the implementation of (3.26) is to use separate buffers for input and output operations. A processor architecture capable of processing r sequences of M/r points each, or r^2 sequences of M/r^2 points each, or one M-point sequence is shown in Fig. 3.8 for r=4.

At any stage of the FFT processor, either BUF1 or BUF2 acts as the input buffer and receives data from the A/D through the input switch and either BUF3 or BUF4 acts as the output buffer and sends data to the D/A through the output multiplexer; the other two buffers are used to store the intermedaite result. The configuration of the buffers, to store the intermediate results and the output Fourier coefficients, is the same as discussed in the previous sections, but the input buffer organization is slightly modified to accommodate multiple sequences of smaller size and is shown in Fig. 3.9. To

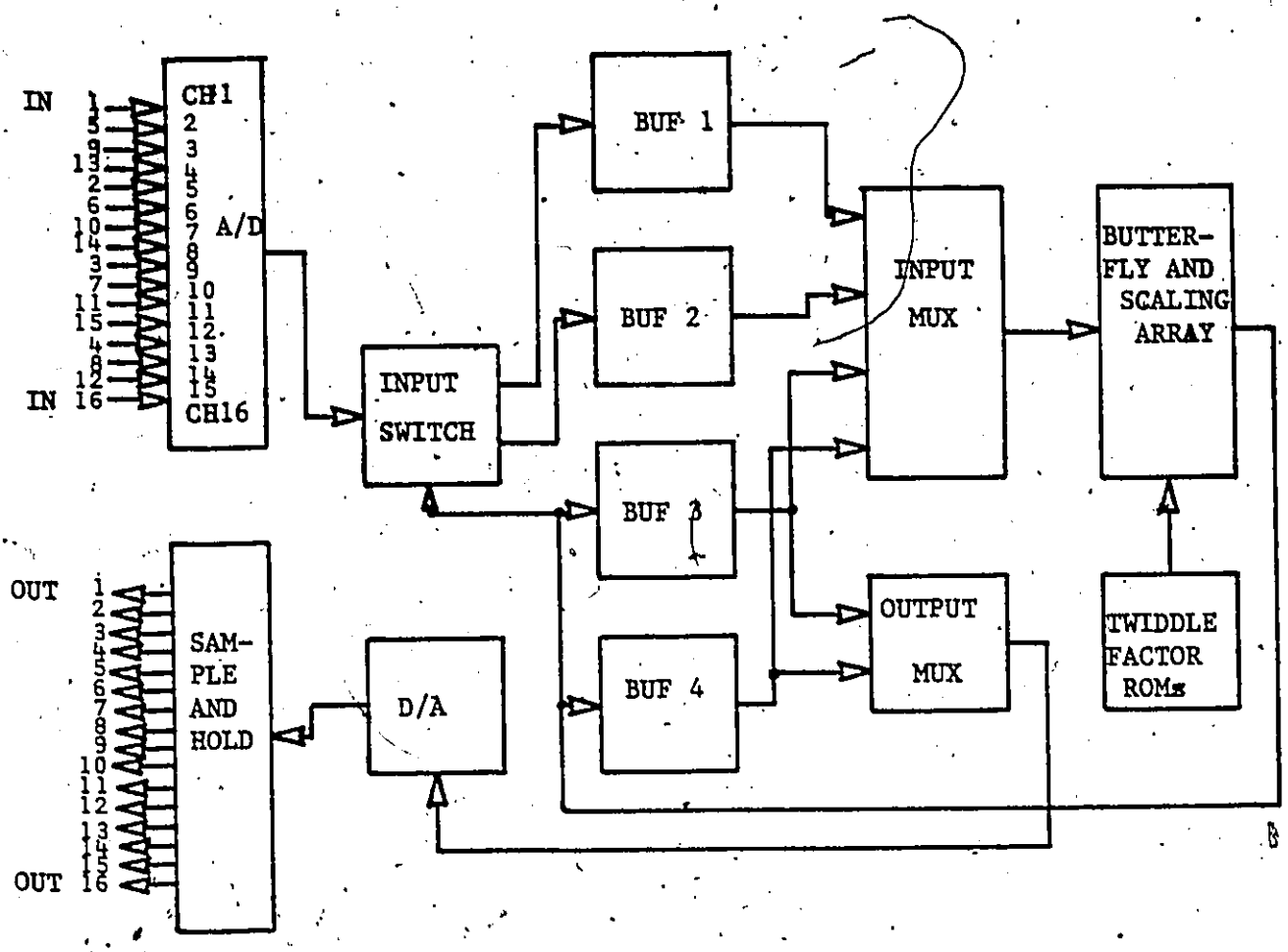
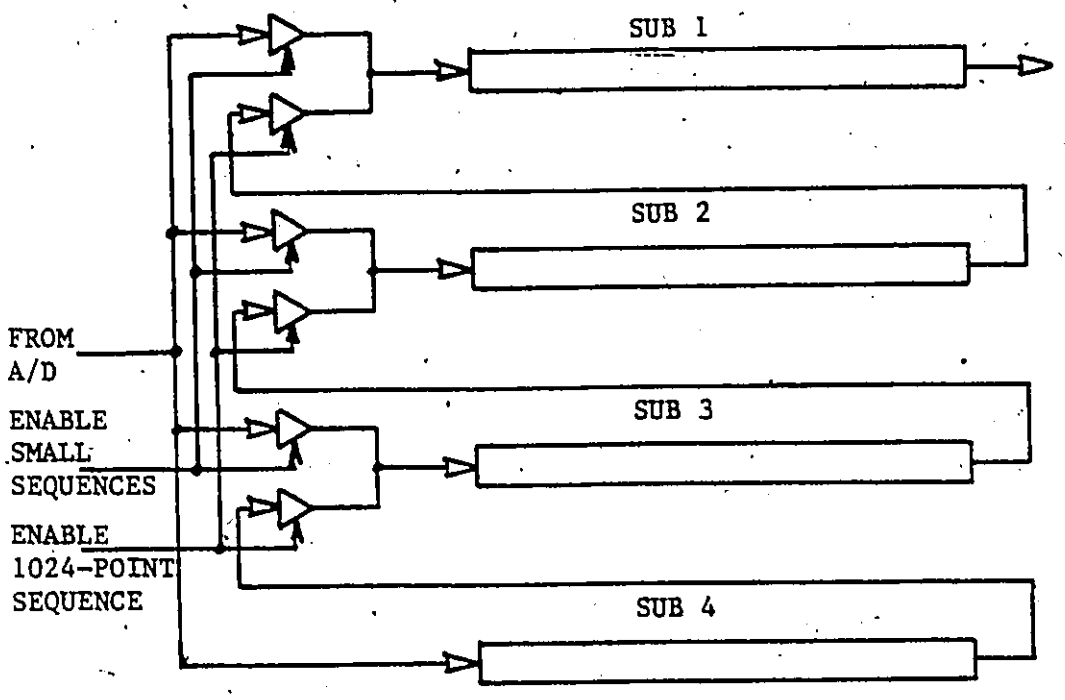


Fig. 3.8 - FFT Processor Configuration for Processing Smaller Sequence



process an M-point sequence the SUBs of the input buffer are connected to form the long shift register of length M, but for r M/r-point sequences, the A/D samples the r input channels and the sampled data is stored into the SUBs in a cyclic order such that SUB1, SUB2,..., SUBr store the data corresponding to input channels 1,2,...,r, respectively. For r^2 M/r^2-point sequences, the sub-buffers still receive the data in a cyclic order, but now from all the r^2 channels. The permutation of the input signals at the input of the A/D allows the output Fourier coefficients of the r^2 sequences, to be in a natural cyclic order. Also in case of r^2 sequences the FFT processor starts processing from the 3rd stage instead of the 1st stage. It may be noted that the D/A performs the demultiplexing operations required to generate F from F', and the Fourier coefficients will appear at the output for each channel in their natural ascending order.

3.5 PROCESSOR ORGANIZATION TO COMPUTE 1-D CONVOLUTIONS OF REAL-VALUED SEQUENCES

The FFT technique has been widely used to perform high speed filtering operations. Since the FFT algorithms offers many forms of parallelism in its implementation, it is often preferred over the direct implementation of digital filters.

The circular convolution of two periodic sequences f and h of period N is given by

$$g(m) = \sum_{n=0}^{N-1} f(n)\, h(|m-n|_N) \qquad (3.27)$$

where $m=0,1,\ldots,N-1$ and $|m-n|_N$ denotes a modulo N operation. If the DFT of the sequences g, f and h is denoted G, F and H, then the DFTs are related by the relationship

$$G(m) = F(m) \cdot H(m); \quad m=0,1,2,\ldots,N-1 \tag{3.28}$$

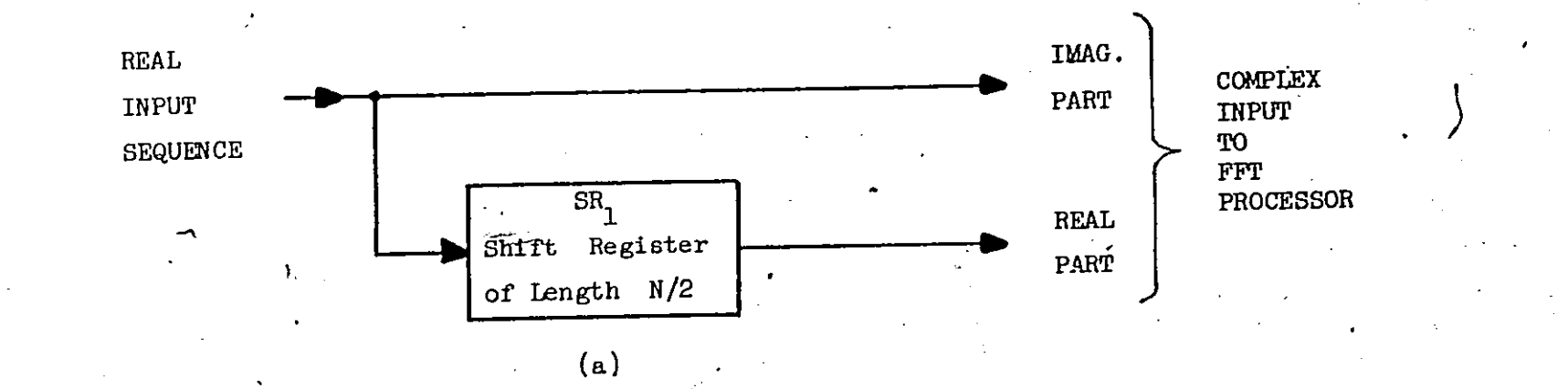
which is a point by point multiplication of the DFTs F and H. The sequence, g, may be computed by taking the inverse DFT of G. In digital filtering application, f and g are the input and the desired output sequences, respectively, and h is the impulse response of a finite impulse response filter. Since the filter impulse response, h, is generally fixed, a filtering operation requires a DFT, an inverse DFT and N complex multiplications.

In many situations we are interested in computing the aperiodic convolution of finite duration sequences rather than circular convolution. The above procedure of computing circular convolution can be applied to compute aperiodic convolution if the finite duration sequences are interpreted as periodic sequences of length (M+L-1), where M and L are the lengths of the f and h, respectively. In situations where M is considerably larger than L, Stockham has described a procedure to convolve the sequences f and h by sectioning the longer sequence f into sections and computing the partial results which can be combined together to form the desired output sequence [45].

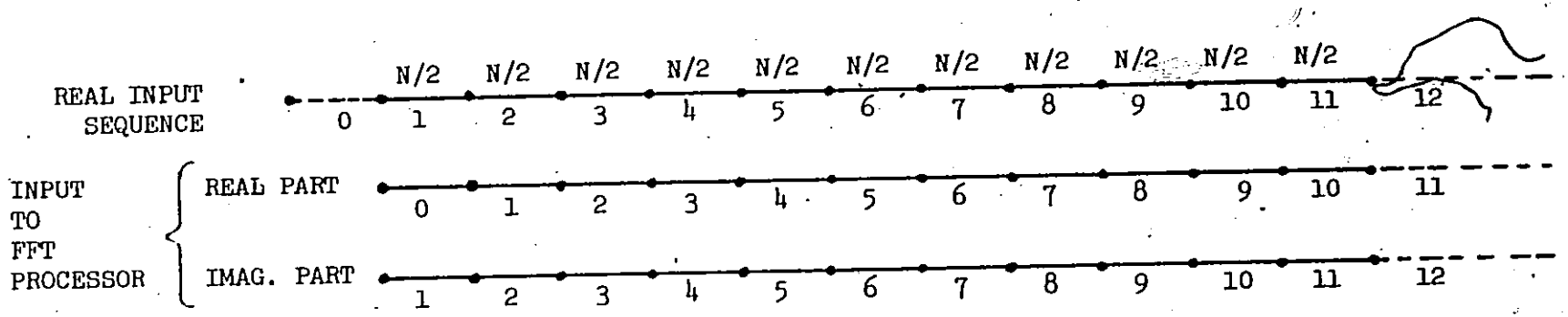
There are two techniques to perform a convolution by the method of sectioned convolutions, namely, Overlap-add and Overlap-save. Both the methods split the sequence f into a large number of sections and

compute the convolution of each section with the sequence h separately. In the overlap-add method we add the results of two consecutive sections to generate the desired result but the Overlap-save method involves overlapping of the input sections and discarding some of the samples of the result of each section. In terms of hardware requirement the overlap-add method requires extra hardware for the addition of the results and seems to have no advantage over the overlap-save method. Thus we choose the overlap-save method and present the implementation of the convolution of real-valued sequences on the FFT processor architecture of Fig. 3.7.

For the implementation of the overlap-save method of sectioned convolution for real-valued input sequences, let the FFT processor be organized to perform the DFT of the N-point sequences and let the length of the impulse response, L, be (N/2 + 1). Fig. 3.10 shows a scheme to generate the complex input to the FFT processor for processing a real-valued input sequence. A shift-register (SR_1) of length N/2 is required to generate a delay of N/2 samples between the real-part and the imaginary part. Initially we clear SR_1 and at the start of the filtering operation the real-part is generated by the output of SR_1 and the imaginary-part is taken directly from the input sequence. The input sequence is also shifted into SR_1 simultaneously. In this way the real-input sequence is divided into sections of length N/2 and a combination of different sections form the input to the FFT processor. For example, a combination of sections 0-1, and 1-2 forms one input, 2-3 and 3-4 forms another input.



(a)



(b)

Fig. 3.10 (a) GENERATION OF COMPLEX INPUT TO FFT PROCESSOR

(b) TIMING DIAGRAM

The processor is organized to perform both the DFT and the inverse DFT (IDFT) of N points and the multiplication by the filter coefficients, pre-stored in ROMs, is done in the last DFT stage. Fig. 3.11 shows the assignment of the processor buffers for input/output, DFT and IDFT operations. To generate the filtered output, a delay of N/2 samples between the real and the imaginary parts of the complex output is required since the 1st half of each output corresponds to the overlapped sections of the input sequence and must be discarded. This is obtained as shown in Fig. 3.12a. As shown in Fig. 3.12b the first half of the complex output of an N-point transform, is discarded. During the period the 2nd half of the complex output is outputted by the processor, the real-part of the complex output is gated to the output bus and the imaginary part is shifted into SR_2. The data in SR_2 is gated to the output bus during the 1st half of the next filtering operation.

For a processor cycle time of T ns a convolution of a N-point real-valued input sequence with a filter impulse-response size of (N/2 + 1) will take $2 \times T \times N/r \times \log_r N$ ns. For example, a 1024-point radix-4 FFT processor with T = 70 ns would take 179.2 μs and a sampling rate of more than 5 MHz is obtainable which is comparable to the pipelined floating-point convolver reported in [24]. The throughput rate may be doubled by using separate radix-4 processors for DFT and IDFT operations.

Note that we may double the processing speed at the expense of an additional inverse FFT processor which amounts to doubling the

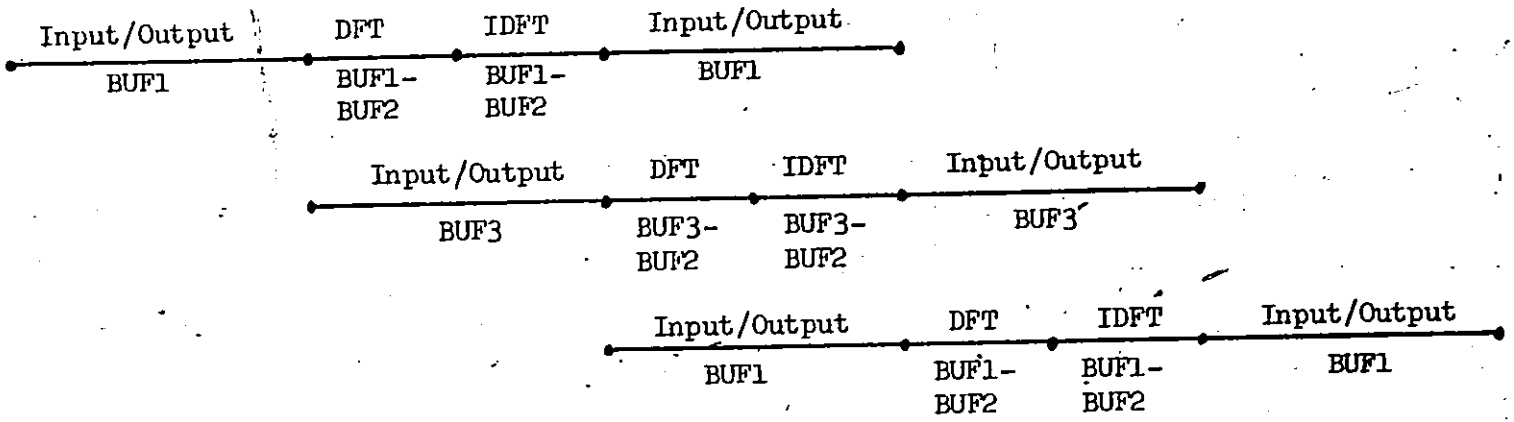


Fig. 3.11 FFT PROCESSOR BUFFER ALLOCATION FOR SECTIONED CONVOLUTION

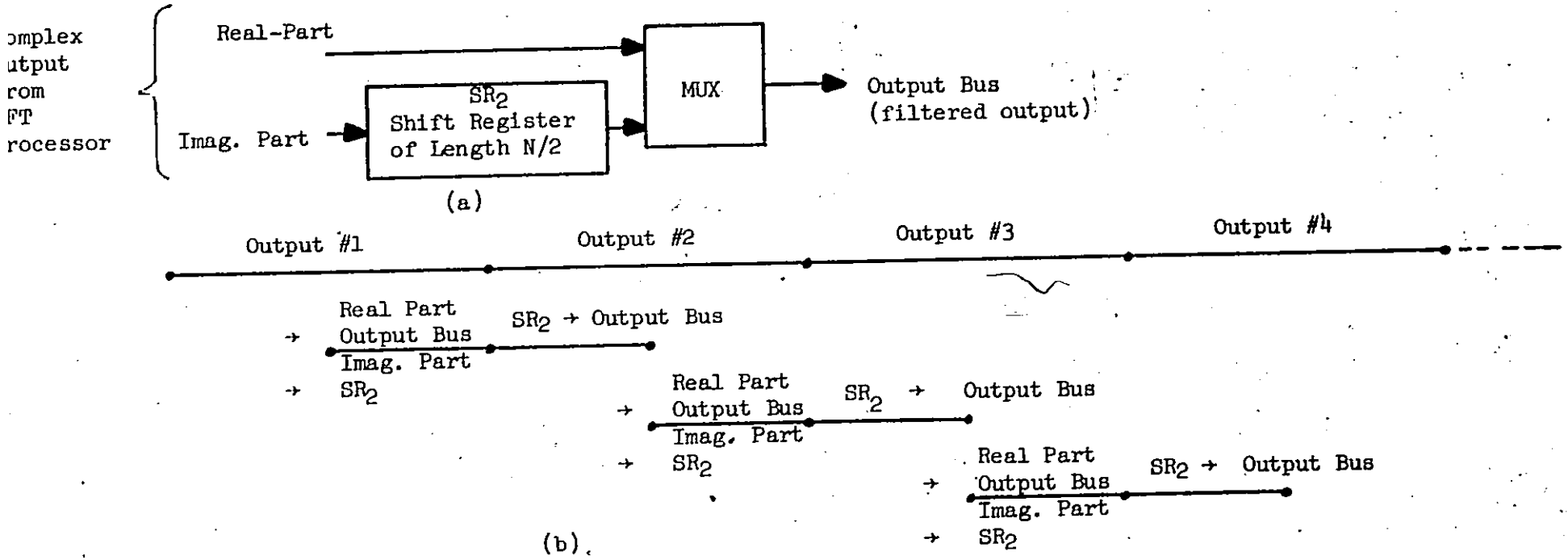


Fig. 3.12 (a) GENERATION OF REAL OUTPUT SEQUENCE FROM THE COMPLEX FFT PROCESSOR OUTPUT

(b) TIMING DIAGRAM

hardware requirement and hence the cost. In many situations the throughput rate of 5 M·samples/sec is sufficient to satisfy the specified processing speed requirement, and thus the requirement of another IDFT processor does not arise. In the case of other processor architectures proposed in the literature [11], [16], [20], [24] we must either use an IDFT processor, or pre- or post-order the data to use a single FFT processor. Since the pre- or post-ordering of data would require extra hardware and time, the cost to performance ratio of these processors would be greater than that of the RNS-based FFT processor proposed here.

3.6 A ONE-DIMENSIONAL WALSH-HADAMARD TRANSFORM PROCESSOR ARCHITECTURE

In section 2.4 we discussed the factorization of the dyadic-ordered WHT transformation matrix and the processor architecture proposed by Geadah et al. [22] in which the WHT is performed in n stages where each stage consists of a computational phase followed by a permutation phase. In this section we show that the permuation and computational phases of a stage of the WHT algorithm can be merged together resulting in an increase in the throughput rate by a factor of three.

In section 2.4 we showed that the factorization of the dyadic-ordered WHT transformation matrix can be written as

$$(H_N)_d = \prod_{i=1}^{n} p_i^{(2)} s^{(2)} \tag{3.29}$$

or

$$(H_N)_d = p_1 s p_2 s p_3 s \ldots\ldots s p_n s \tag{3.30}$$

where the superscript (2) is omitted for clarity and

$$p_1 = (I_{2^{n-1}} \otimes P_2) = I_N \tag{3.31}$$

Defining the operators

$$s_{i-1} = sp_i; \quad i=2,3,4,..,n \tag{3.32}$$

and

$$s_n = s \tag{3.33}$$

equation (3.30) can be written as

$$(H_N)_d = \prod_{i=1}^{n} s_i \tag{3.34}$$

It can be shown that [13]

$$p_i^{-1} (I_{N/2} \otimes T_2)p_i = (I_{N/4} \otimes T_2 \otimes I_2) \tag{3.35}$$

Thus, s_{i-1} can be written as

$$\begin{aligned} s_{i-1} = sp_i &= (I_{N/2} \otimes T_2)p_i \\ &= p_i\, p_i^{-1} (I_{N/2} \otimes T_2)p_i \\ &= p_i (I_{N/4} \otimes T_2 \otimes I_2) \\ &= p_i\, s' \end{aligned} \tag{3.36}$$

where

$$s' = (I_{N/4} \otimes T_2 \otimes I_2) \tag{3.37}$$

The operator s, s' and s_{i-1} are exactly the same as defined in equations (2.19) to (2.23) for r=2. Since the twiddle factor operator μ_i of equation (2.17) is not present in equation (3.34) we can conclude that the OI00 FFT algorithm of section 2.3.1 can be directly applied to compute the dyadic-ordered WHT if the twiddle factor multiplications specified by operator μ_i are omitted.

From the above discussion it is clear that we can compute the dyadic-ordered WHT using FFT processor architecture of section 3.3. Since this organization does not require a feedback phase in which the data is serially moved from the output to the input buffer, the throughput rate obtainable from the FFT processor architecture is greater than that of the architecture proposed in [22]. From the discussion in section 2.4 we note that the FFT processor architecture proposed here will compute the WHT of a N-point sequence in $(\frac{N}{2} \cdot T \cdot \log_2 N)$ ns, whereas the architecture proposed in [22] will compute the same WHT in $(3\frac{N}{2} \cdot T \cdot \log_2 N)$ ns. This gives an increase in throughput rate by a factor of three. Also note that the computation of the WHT does not require multiplication by the twiddle factors, so it is not clear whether the RNS implementation offers a significant advantage over the implementation of a WHT processor using the Binary number system. Depending upon the trade offs involved we may opt for the Binary number system since the basic architecture of the WHT processor proposed here is independent of the number system used.

3.7 SUMMARY

Based on the RNS principles and techniques, a novel special purpose FFT processor architecture for computing the DFT of real-time input sequences has been presented. Using the ROM array architecture of an RNS-based Butterfly unit it has been demonstrated that the butterfly operations required for the computation of the DFT of an M-point sequence can be pipelined, resulting in a throughput rate of the inverse of one ROM access time. It has also been shown that the throughput rate obtainable from the serial sequential processor architecture, presented here, is comparable to that obtainable from the cascade/pipeline organizations which require a number of Butterfly units. When it is required to process data from a number of input channels we also showed that the basic RNS-based FFT processor architecture can be used to process data from r or r^2 input channels with little modification.

For real-time filtering applications it has been shown that the basic RNS-based processor architecture can be used convolve a continuous real-valued input signal with an impulse response of size $(\frac{N}{2} + 1)$, where the FFT processor is organized to perform an N-point DFT and inverse DFT operations. The convolver architecture proposed here does not require pre- or post-ordering of the data and is more cost-effective than the organizations proposed in the literature. It has also been shown that the FFT processor architecture, proposed here, can also be adapted for implementation of the dyadic-ordered WHT.

It was shown that the throughput rate obtainable from the FFT processor architecture proposed here is three times the throughput rate obtainable from the WHT processor proposed in [22].

CHAPTER 4

MULTI-DIMENSIONAL ALGORITHMS AND PROCESSOR ARCHITECTURES FOR COMPUTING A CLASS OF UNITARY TRANSFORMS

4.1 INTRODUCTION

Many of the fast algorithms for computing unitary transforms deal with one-dimensional transforms. Multi-dimensional unitary transforms are generally computed by taking one-dimensional (1-D) transforms, utilizing fast 1-D radix-r unitary transform algorithms, along all dimensions, successively. In the past, due to a large amount of fast access memory required to store the input data and the relatively higher cost of the memory component, it was necessary to store the input/output of a multi-dimensional algorithm in a relatively slow-speed bulk store which resulted in a slow processing speed. To obtain high throughput rate from these implementations, investigations were limited to the development of techniques that reduce the input/output operations between the processor and the bulk store [35]-[37]. Also the throughput rate constraints imposed by the above implementations limited their use to non real-time applications.

With the recent development of high density, high speed and low cost memory packages, it is now possible to provide a large amount of memory at low cost to satisfy the high processing speed requirement imposed

by many real-time applications in the areas of image-processing and robotics. Recently Gilbert et al. [46] described the organization of a real-time image processor using a large-scale computer and Joshi et al. [21] have proposed a multiprocessor architecture for the implementation of a two-dimensional FFT processor capable of processing images at video rates. Both of these architectures use an interleaved memory organization with multiple memory units to support the high bandwidth of the video signals.

Since the implementation of multi-dimensional unitary transform algorithms using a large-scale computer or a multi-processor organization is not very cost effective for specific transforms we will investigate other approaches to the implementation of multi-dimensional unitary transforms, to reduce cost without sacrificing processing speed requirements. In this Chapter we investigate further the structure of multi-dimensional unitary transform algorithms and present special purpose processor architectures capable of processing images or other multi-dimensional signals in real-time. These processor architectures use a single arithmetic processor and multiple memory units and are more viable in terms of cost than implementations using multi-processors or large scale computers.

4.2 MULTI-DIMENSIONAL UNITARY TRANSFORMS

The unitary transform of a u-dimensional array $F(n_1,n_2,\ldots,n_u)$ of size $(M_1 x M_2 x \ldots x M_u)$ is given by

$$P(m_1,m_2,\ldots,m_u) = \sum_{n_1=0}^{M_1-1} \sum_{n_2=0}^{M_2-1} \cdots \sum_{n_u=0}^{M_u-1} F(n_1,n_2,\ldots,n_u)\cdot O(n_1,n_2,\ldots,n_u;$$

$$m_1,m_2,\ldots,m_u)$$

$$m_i = 0,1,2,\ldots,M_{i-1}; \quad i=1,2,3,\ldots,u. \tag{4.1}$$

where $P(m_1,m_2,\ldots,m_u)$ is the transformed array of size $(M_1,M_2,\ldots,M_u)$ and $O(n_1,n_2,\ldots,n_u;\ m_1,m_2,\ldots,m_u)$ is the transformation kernel. For a unitary transform, the transformation given by (4.1) is exactly invertible and the kernel $O(n_1,n_2,\ldots,n_u;\ m_1,m_2,\ldots,m_u)$ satisfies the following orthornormality conditions [31]:

$$\sum_{m_1}\sum_{m_2}\cdots\sum_{m_u} O(n_1,n_2,\ldots,n_u;\ m_1,m_2,\ldots,m_u)\cdot O^*(j_1,j_2,\ldots,j_u;\ m_1,m_2,\ldots,m_u)$$

$$= \delta(n_1-j_1,\ n_2-j_2,\ldots,n_u-j_u) \tag{4.2}$$

and

$$\sum_{n_1}\sum_{n_2}\cdots\sum_{n_u} O(n_1,n_2,\ldots,n_u;\ m_1,m_2,\ldots,m_u)\cdot O^*(n_1,n_2,\ldots,n_u;\ k_1,k_2,\ldots,k_u)$$

$$= \delta(m_1-k_1,\ m_2-k_2,\ldots,m_u-k_u) \tag{4.3}$$

where $O^*(x)$ denotes the complex conjugate of $O(x)$ and $\delta(x)$ is the Kronecker delta.

The kernel is said to be separable if it can be written as

$$O(n_1,n_2,\ldots,n_u;\ m_1,m_2,\ldots,m_u) = O(n_1,m_1)\cdot O_2(n_2,m_2)\ldots O_u(n_u,m_u) \tag{4.4}$$

For a separable kernel, the unitary transform of equation (4.1) is given by

$$P(m_1,m_2,\ldots,m_u) = \sum_{n_1=0}^{M_1-1} \sum_{n_2=0}^{M_2-1} \cdots \sum_{n_u=0}^{M_u-1} F(n_1,n_2,\ldots,n_u)\cdot 0_1(n_1,m_1)\cdot 0_2(n_2,m_2)$$

$$\ldots . 0_u(n_u,m_u) \qquad (4.5)$$

which can be computed in the following u steps:

$$P_1(m_1,n_2,n_3,\ldots,n_u) = \sum_{n_1=0}^{M_1-1} F(n_1,n_2,\ldots,n_u)\cdot 0_1(n_1,m_1); \qquad (4.6)$$

$$P_2(m_1,m_2,n_3,n_4,\ldots,n_u) = \sum_{n_2=0}^{M_2-1} P_1(m_1,n_2,n_3,\ldots,n_u)\cdot 0_2(n_2,m_2);$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$P(m_1,m_2,m_3,\ldots,m_u) = \sum_{n_u=0}^{M_u-1} P_{u-1}(m_1,m_2,\ldots,m_{u-1},n_u)\cdot 0_u(n_u,m_u).$$

Thus a separable u-dimensional (u-D) unitary transform may be computed by taking 1-D transforms along each dimension of the array F, successively.

For purposes of analysis it is convenient to use a 1-D vector representation of the u-D arrays. This may be obtained by scanning the u-D arrays F and P in lexicographical order and then stringing the elements in a long vector. For example, the 1-D vector representation, f, of a 3-D array $F(n_1,n_2,n_3)$ of size (4x3x5) is given by

$$\begin{aligned} f = [&F(0,0,0), F(0,0,1), F(0,0,2), F(0,0,3), F(0,0,4) \\ &F(0,1,0), F(0,1,1), F(0,1,2), F(0,1,3), F(0,1,4) \\ &F(0,2,0), F(0,2,1), F(0,2,2), F(0,2,3), F(0,2,4) \\ &\qquad\qquad \vdots \qquad\qquad\qquad \vdots \\ &F(3,2,0), F(3,2,1), F(3,2,2), F(3,2,3), F(3,2,4)]^T \end{aligned} \qquad (4.7)$$

where $[\cdot]^T$ denotes the transpose operator.

Let the u-D arrays F and P be represented in the lexicographical order and their vector representation be given by f and p respectively. In terms of this vector representation the unitary transform of equation (4.5) can be written as

$$p = Tf \tag{4.8}$$

where T denotes the (NxN) matrix performing a unitary transformation on the (Nxl) input vector f yielding the (Nxl) output vector p, where

$$N = \prod_{i=1}^{u} M_i .$$

In [33], Good has shown that, in terms of the vector representation of the unitary transform of equation (4.8), the transformation matrix T can be written as the Kronecker product of the one-dimensional transformation matrices:

$$T = (T_{M_u}^{(u)} \otimes T_{M_{u-1}}^{(u-1)} \otimes \cdots\cdots \otimes T_{M_1}^{(1)}) \tag{4.9}$$

where $T_{M_i}^{(i)}$ is the one-dimensional transformation matrix of the unitary transform in the ith dimension with elements

$$[T_{M_i}^{(i)}]_{j,k} = O_i(j,k) \tag{4.10}$$

and the Kronecker product is defined as

$$C = A \otimes B = \begin{vmatrix} b_{00}A & b_{01}A & b_{02}A & \cdots \\ b_{01}A & b_{11}A & b_{12}A & \cdots \\ \vdots & \vdots & \vdots & \\ \vdots & \vdots & \vdots & \\ \vdots & \vdots & \vdots & \end{vmatrix} \quad (4.11)$$

where A and B are matrices with elements $b_{jk} = [B]_{jk}$ and $a_{jk} = [A]_{jk}$.

The elements of T may be obtained by the relation

$$[T]_{n,m} = \prod_{i=1}^{u} [T_{M_i}^{(i)}]_{n_i,m_i} \quad (4.12)$$

where the row and column indices n and m represent the lexicographic ordering of $(n_1,n_2,n_3,\ldots,n_u)$ and $(m_1,m_2,m_3,\ldots,m_u)$ respectively. For example, for matrices A and B of size (2x2), the matrix C is given by

$$C = A \otimes B = \begin{array}{c|cccc} m & 0,0 & 0,1 & 1,0 & 1,1 \\ n & & & & \\ \hline 0,0 & a_{00}b_{00} & a_{01}b_{00} & a_{00}b_{01} & a_{01}b_{01} \\ 0,1 & a_{10}b_{00} & a_{11}b_{00} & a_{10}b_{01} & a_{11}b_{01} \\ 1,0 & a_{00}b_{10} & a_{01}b_{10} & a_{00}b_{11} & a_{01}b_{11} \\ 1,1 & a_{10}b_{10} & a_{11}b_{10} & a_{10}b_{11} & a_{11}b_{11} \end{array} \quad (4.15)$$

or $$[C]_{(n_1,n_2),(m_1,m_2)} = b_{n_1m_1} \cdot a_{n_2m_2}$$

For example, $[C]_{(0,1),(1,1)} = b_{01} \cdot a_{11}$.

Since the two definitions of a unitary transform defined by equations

(4.5) and (4.8) are equivalent, equations (4.8) and (4.9) imply that the output array, P, can be computed by taking one-dimensional transforms, with kernel $T_{M_i}^{(i)}$, along each dimension of the array F, sequentially.

From equations (4.9) and (4.10) it is clear that the matrix T is unitary since the 1-D transformation matrices $T_{M_i}^{(i)}$ (i=1,2,...,u) are unitary. Thus, it is possible to specify a different transformation kernel for each of the u dimensions (multi-dimensional unitary transforms generally specify the same transformation along all the u dimensions). Although the above analysis is applicable to the situation where the arrays have different number of elements in each dimension, we restrict the number of elements in each dimension to a constant value, M. With these restrictions, the transformation matrix, T, of equation (4.9) can be written as

$$T = (T_M \otimes T_M \otimes \cdots\cdots \otimes T_M) \qquad (4.16)$$

$$= [T_M]^{\otimes u}$$

where $T_{M_i}^{(i)} = T_M$; i=1,2,...,u (4.17)

and $[\cdot]^{\otimes n}$ denotes the Kronecker nth power. The 1-D transformation matrix T_M specifies the same transformation along all the u dimensions.

4.2.1 Multi-Dimensional Unitary Transform Algorithms

A direct implementation of the transformation specified by (4.16) would require M^{2u} multiplications. Since the transformation matrix T is separable, the implementation of the unitary transform in u steps,

as specified by (4.6), leads to a considerable saving in computation. Since, for many unitary transforms, the 1-D transformation matrix T_M itself can be factorized into a number of sparse matrices, and a further saving in the computation can be obtained. The factorization techniques, described by Good [33], that specify the matrix T in terms of sparse matrices which contain only the elements of the 1-D transformation matrix, T_M, also achieve similar computational efficiency. Since many efficient factorization of the transformation matrix T_M and the 1-D processor architectures implementing these factorizations have been proposed in the literature, we will apply these factorizations of the 1-D transformation matrix T_M to obtain different factorizations of T. It will be shown that the factorizations of T obtained in this way result in high speed processor architectures capable of processing the multi-dimensional input data in real-time.

For a number of unitary transforms, including Fourier [11]-[16], Walsh-Hadamard [3],[4], generalized Walsh [6], generalized transforms [7], Haar [5] and many other transforms [8], the 1-D transformation matrix, T_M, can be expressed as the product of sparse matrices. When M is a highly composite number, i.e. $M = r^n$, T_M can be factored into a product of n matrices of size (MxM) where each factor may consist of Kronecker products of smaller sub-matrices.

Let the 1-D transformation matrix T_M be given by

$$T_M = \prod_{m=1}^{n} \beta_m \tag{4.18}$$

where β_m is a sparse matrix of size (MxM). For example, $\beta_m = \mu_m s_m$ for

the Fourier transform and $\beta_m = s_m$ for Walsh-Hadamard transform as defined in equations (2.17) and (3.34). Factorizations of T_M for other transforms are given in the references [3]-[8].

Substituting the factorization of T_M into (4.16) we get

$$T = \left[\prod_{m=1}^{n} \beta_m \right] \otimes u \tag{4.19}$$

Using the relationship (3.14) between the Kronecker and ordinary matrix product we can write (4.19) as

$$T = \prod_{m=1}^{n} [\beta_m] \otimes u \tag{4.20}$$

Using equations (4.17) and (4.20), the unitary transform defined by (4.8) can be written as

$$p = \left\{ \prod_{m=1}^{n} [\beta_m] \otimes u \right\} f \tag{4.21}$$

which states that the computation of p may be divided into n stages where each stage operates on the output of the preceeding stage and the operator of the first stage $[\beta_n] \otimes u$ operates on the input vector f. The computation of the ith stage is specified by the operator $[\beta_i] \otimes u$ and can be written as

$$p_i = [\beta_i] \otimes u \; f_i \tag{4.22}$$

where f_i and p_i are the input and output of the ith stage respectively. The matrix $[\beta_i] \otimes u$ is of the same form as that of the multi-dimensional

transformation matrix T of equation (4.16). Since T implies that the output array P can be computed by taking 1-D transforms, with kernel T_M, along each dimension of the input array F, $[\beta_i] \otimes^u$ implies that p_i or its multi-dimensional representation P_i may be computed by sequential application of the 1-D operator, β_i, along each dimension of the multi-dimensional representation, F_i, of f_i. For a number of unitary transforms, the matrix β_i itself is a product of permutation and arithmetic operators and when the factors of β_i are substituted into equation (4.20), T can be factorized further into sparser matrices. The factors of T obtained in this way are much more sparse and convenient to implement than the factors obtained by using factorization techniques proposed by Good [33].

To derive a specific algorithm for the computation of a unitary transform, and a special purpose processor architecture for its implementation, it is necessary that the structure of the β_i's be known. Since the structure of β_i is different for different 1-D unitary transforms and a 1-D unitary transform may have a number of factorizations (see, for example [11]-[16] for Fourier transform and [4]-[8], [22] for Walsh-Hadamard, Haar and other unitary transforms), we will discuss, as examples of the above factorization techniques, the implementation of a few algorithms only. Because of the popularity and usefulness of the discrete Fourier and Walsh-Hadamard transforms in the general area of signal processing, we use the above factorization techniques to develop two- and multi-dimensional FFT and WHT algorithms, together with special processor structures for the implementation of these algorithms. This development may be extended

to the implementation of other unitary transforms.

4.2.2 Multi-Dimensional Discrete Fourier Transform

The discrete Fourier transform (DFT) of a u-dimensional array $F(n_1,n_2,\ldots,n_u)$ of size M in all the dimensions, is given by

$$P_{(m_1,m_2,\ldots,m_u)} = \sum_{n_1=0}^{M-1} \sum_{n_2=0}^{M-1} \cdots \sum_{n_u=0}^{M-1} F(n_1,n_2,\ldots,n_u)\, W_M^{\sum_{i=1}^{u} n_i m_i} \qquad (4.23)$$

$$m_1,m_2,\ldots,m_u = 0,1,2,..,M-1$$

where $W_M = e^{-2\Pi j/M}$. The DFT kernel is separable since

$$O_i(n_i,m_i) = W_M^{n_i m_i} \quad ; \; i=1,2,..,u. \qquad (4.24)$$

In terms of the vector representation, the DFT of equation (4.23) can be expressed in the form of equation (4.8) as

$$p = T\, f \qquad (4.25)$$

where

$$T = [T_M] \otimes u \qquad (4.26)$$

and T_M is the (MxM) 1-D DFT transformation matrix with elements $t_{j,k} = W_M^{jk}$.

Based on the Kronecker product factorization of the DFT transformation matrix, T_M, a number of FFT processor architectures have been proposed in the literature [11]-[17]. As discussed in the last Chapter, very high throughput rates may be obtained from the serial sequential type of

processors using state of the art technology; here we use the factorization of the OI00 1-D FFT algorithm of section 2.3.1 and develop a serial sequential type of multi-dimensional FFT processor.

From equations (4.18) and (2.17) we can write the 1-D operator β_i as

$$\beta_m = \mu_m^{(r)} \; s_m^{(r)} \tag{4.27}$$

where the operators $\mu_m^{(r)}$ and $s_m^{(r)}$ are defined in equations (2.18) to (2.23). Substituting the factorization of β_m into equation (4.21) we get

$$p = \{ \prod_{m=1}^{n} [\mu_m^{(r)} s_m^{(r)}] \otimes u \} f \tag{4.28}$$

Using (3.14), equation (4.28) can be written as

$$p = \{ \prod_{m=1}^{n} [\mu_m^{(r)}] \otimes u \cdot [s_m^{(r)}] \otimes u \} f \tag{4.29}$$

From equation (2.19) we have

$$s_{m-1}^{(r)} = p_m^{(r)} \; s_m^{(r)}; \quad m=2,3,\ldots,n \tag{4.30}$$

and

$$s_n^{(r)} = s^{(r)}. \tag{4.31}$$

Thus, $[s_{m-1}^{(r)}] \otimes u$ can be written as

$$[s_{m-1}^{(r)}] \otimes u = [p_m^{(r)} \; s^{(r)}] \otimes u \tag{4.32}$$

Again using (3.14) we can write

$$[s_{m-1}^{(r)}] \otimes u = [p_m^{(r)}] \otimes u \cdot [s^{(r)}] \otimes u \tag{4.33}$$

and

$$[s_n^{(r)}] \otimes u = [s^{(r)}] \otimes u \qquad (4.34)$$

Defining $R_{m-1}^{(r)} = [p_m^{(r)}] \otimes u \; . \; [s^{(r)}] \otimes u; \; m=2,3,\ldots,n. \qquad (4.35)$

$$R_n^{(r)} = [s^{(r)}] \otimes u \qquad (4.36)$$

and

$$\psi_m^{(r)} = [\mu_m^{(r)}] \otimes u \qquad (4.37)$$

we can write equation (4.29) as

$$p = \{ \prod_{m=1}^{n} \psi_m^{(r)} R_m^{(r)} \} f \qquad (4.38)$$

This equation is of the same form as that of (4.21) and hence the computation of p may be divided into n stages where the computation of the ith stage is given by

$$p_i = \{ \psi_i^{(r)} R_i^{(r)} \} f_i \qquad (4.39)$$

Since $\psi_i^{(r)}$ and $R_i^{(r)}$ have been defined as the Kronecker products of the 1-D operators $\mu_i^{(r)}$, $p_i^{(r)}$ and $s_i^{(r)}$, as given by equations (4.35) to (4.37), p_i may be computed by sequential application of the operators $s_i^{(r)}$, $p_i^{(r)}$ and $\mu_i^{(r)}$ along each dimension of the multi-dimensional representation, F_i, of the vector f_i. As the operators $s^{(r)}$ or $s^{(r)}$, $p_i^{(r)}$ and $\mu_i^{(r)}$ specify a base-r DFT, permutation and twiddle factor multiplication operation, respectively, a stage consists of the sequential application of an r-point DFT, permutation and twiddle factor multiplication

operation to the points in each dimension of F_i.

Except for the first and the last stage, the operations specified by the operators $\psi_i^{(r)} R_i^{(r)}$ may be written as

$$\psi_i^{(r)} R_i^{(r)} = [\mu_i^{(r)}] \otimes u \; . \; [p_{i+1}^{(r)}] \otimes u \; . \; [s^{(r)}] \otimes u \qquad (4.40)$$

The operator $R_i^{(r)}$ specifies that the operator $s^{(r)}$ be applied to the points in each dimension of the input array, sequentially, and, as shown in section 2.3.1, the operator $s^{(r)}$ specifies that an r-point DFT operation be performed on the data points which are M/r^2 words apart. Thus the operator $R_i^{(r)}$ operates on the points which are M/r^2 words apart in each dimension of the input array. For example, for a 2-dimensional array of size (8x8) and a radix-2 implementation, the operator $R_i^{(2)}$ specifies a radix-2 DFT operation to be performed on the data points which are 2 words apart in both the row as well as column directions, i.e. on the points (0,0) and (0,2), (0,1) and (0,3), (2,0)

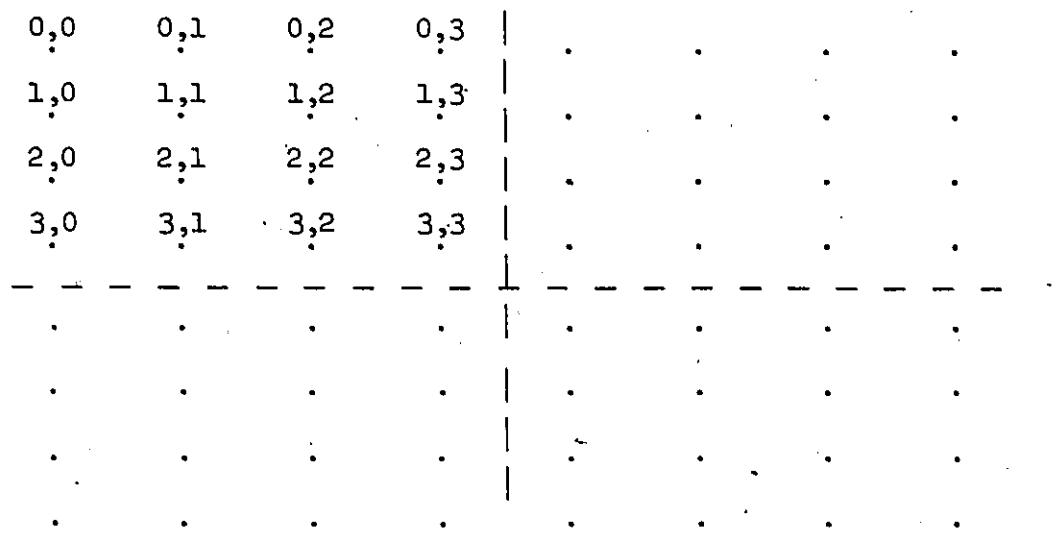


Figure 4.1 A 2-D SEQUENCE.

and (2,2) etc. in the row direction and (0,0) and (2,0), (1,0) and (3,0), (0,2) and (2,2) etc. in column direction (see Fig. 4.1). An address separation of M/r^2 between the data points, in each dimension, may be obtained if the input array is divided into r^u blocks of data where each block of data is further divided into r^u sub-blocks. For example, if the input array is assumed to be a u-dimensional hyper-cube containing M^u data points, the hyper-cube is divided into r^u equal blocks where each block is further sub-divided into r^u equal sub-blocks containing $(M/r^2)^u$ data points. If the data points in each sub-block are arranged in lexicographical order and then accessed sequentially, the data points from the sub-blocks of any block will have an address separation of M/r^2 in each dimension.

Since $R_i^{(r)}$ specifies r^{u-1} r-point DFT operations to be performed, along each of the u dimensions, on the data points which are M/r^2 words apart, sequentially, the sequential ($u\ r^{u-1}$) r-point DFTs may be combined to form a single u-dimensional radix-r DFT operation. For example, for a 2-D radix-2 implementation of the array of Fig. 4.1, the sequential radix-2 DFT operations on the data points (0,0) and (0,2), (2,0) and (2,2) in the row direction and on the data points (0,0) and (2,0), (0,2) and (2,2) in the column direction can be combined to form a single 2-D radix-2 DFT operation, since the radix-2 DFT operations in both the row as well as column directions are specified on the same data points. A u-dimensional radix-r DFT operation requires that the r^u data points from the specified blocks be available simultaneously, which may be achieved by accessing the points from the r^u sub-blocks

of the selected block simultaneously.

Similarly the operator $\psi_i^{(r)}$ specifies that the operator $\mu_i^{(r)}$ be applied to the points along each of the u-dimensions, sequentially, and $\mu_i^{(r)}$ specifies multiplication by the twiddle factors. As before these sequential multiplications may be combined to give composite twiddle factors and the multiplication by the composite twiddle factors may be performed after the u-dimensional radix-r DFT operation.

This separation of the radix-r DFT and twiddle factor multiplication operations in each dimension into u-dimensional radix-r DFT and composite twiddle factor multiplication operations, leads to a considerable saving in the number of multiplications by the twiddle factors. Note that a u-dimensional radix-r DFT operation combines $(u\ r^{u-1})$ 1-D radix-r DFT operations but there are only (r^u-1) composite twiddle factor multiplications. If 1-D radix-r DFT operations were performed along all the u dimensions, sequentially, as specified by the operator $R_i^{(r)}$, the number of multiplications, as specified by $\psi_i^{(r)}$, would be equal to $(u(r^u-r^{u-1}))$. This represents a saving of $(u(r^u-r^{u-1})-(r^u-1))$ complex multiplications. For example, for a 3-D array, a 3-D radix-2 DFT operation requires 7 complex multiplications as compared to 12 complex multiplication required by an equivalent 1-D radix-2 DFT operations, which indicates a saving of 5 complex multiplications.

The above development is a generalization of the multi-dimensional FFT algorithm proposed by Arambepola and Rayner [34]. The algorithm proposed in [34] generates the Fourier coefficients in bit-reversed order and may be developed by using the above factorization techniques

and the post-permutation algorithm proposed by Corinthios [14].

For simplicity and conciseness we restrict our further discussion to the implementation of a two-dimensional (2-D) radix-r FFT processor. The memory architecture of the 2-D processor can be easily extended to higher dimensions.

4.2.3 Two-Dimensional FFT Processor Organization

For the 2-D radix-r FFT algorithm equation (4.40), specifying the operations to be performed during the intermediate FFT stages, can be written as

$$\psi_i^{(r)} R_i^{(r)} = [\mu_i^{(r)}] \otimes 2 \cdot [p_{i+1}^{(r)}] \otimes 2 \cdot [s^{(r)}] \otimes 2 \qquad (4.41)$$

Similar to the u-dimensional case an address separation of M/r^2 in both the row as well as column direction may be obtained if the input matrix is divided into r^2 blocks of data and each block is further sub-divided into r^2 sub-blocks. For example, for M=16 and r=2 the input matrix is divided into 4 blocks where each block consists of 4 sub-blocks as shown in Fig. 4.2. When the data from the sub-blocks is accessed sequentially, as indicated by the arrows, the address separation in the row and column direction is always 4.

For the 2-D implementation $R_i^{(r)}$ specifies 2r r-point DFT operations to be performed on the points in the rows and columns, which are M/r^2 rows and columns apart, respectively. The sequential 2r DFT operations can be combined to form a single 2-D r-point DFT operation. As shown in Fig. 4.2, the r^2 data points required for a 2-D r-point DFT operation

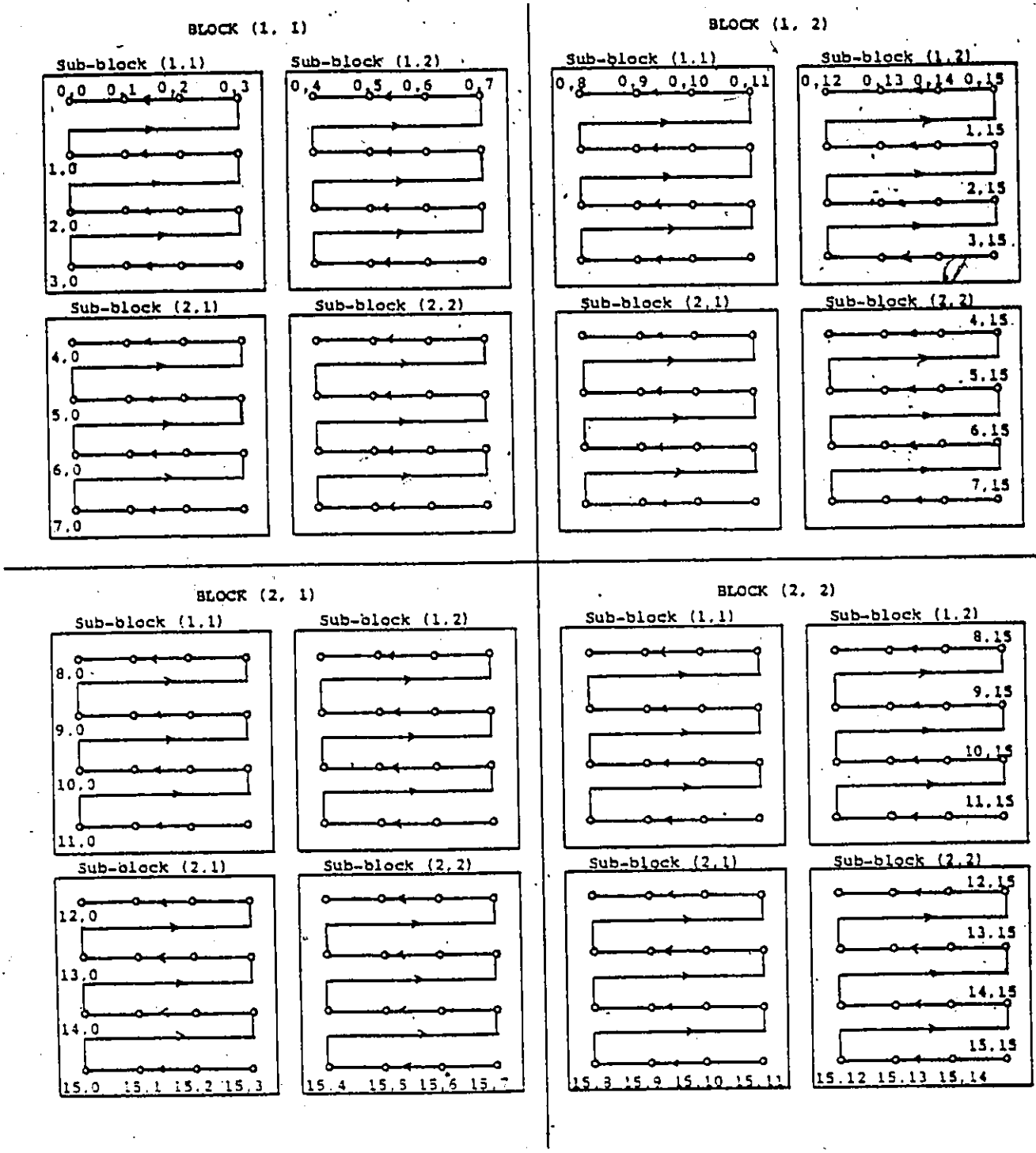


Fig. 4.2 Partitioning of a 16 x 16 array into sub-blocks

$_{o}^{i,j}$ indicate the ijth element of the array

may be obtained, simultaneously, by accessing the points from the r^2 sub-blocks of a selected block.

Similar to the 1-D FFT processor organization of section 2.3.1 Fig. 4.2 suggests a sequential memory structure for the operations specified by the stage operators $\psi_i^{(r)}$ $R_i^{(r)}$. A processor organization utilizing a sequential memory structure is shown in Fig. 4.3 and consists of two memory buffers BUFF1 and BUFF2 and a 2-D radix-r Butterfly unit. Each buffer has a storage capacity of M^2 complex words and consists of r^2 memory blocks where each block is further divided into r^2 sub-blocks (SBs) and each SB stores $(M/r^2)^2$ words. The SB's always store the input data or intermediate output of the FFT stages, sequentially, as indicated by the arrows in Fig. 4.2. BUFF1 and BUFF2 are used to supply data to, and receive output from, the Butterfly unit, alternatively.

Except for the first FFT stage, the permutation and DFT operations are specified by the operator $R_i^{(r)} = (p_{i+1}^{(r)} \otimes p_{i+1}^{(r)})(s^{(r)} \otimes s^{(r)})$.† To implement these operations, the r^2 points at the top of the sub-blocks of a selected block form the input data to the Butterfly unit where the computations for the 2-D radix-r DFT are performed. The data from the SB's of the selected block is always accessed in sequential order and the selection of a block is performed as given in the following steps. The steps are written in FORTRAN like statements where the block indices

---

† The stage index i is specified in the inverse order, i.e. for i=1,2,3,..,n, stage number j=n,n-1,n-2,..,1.

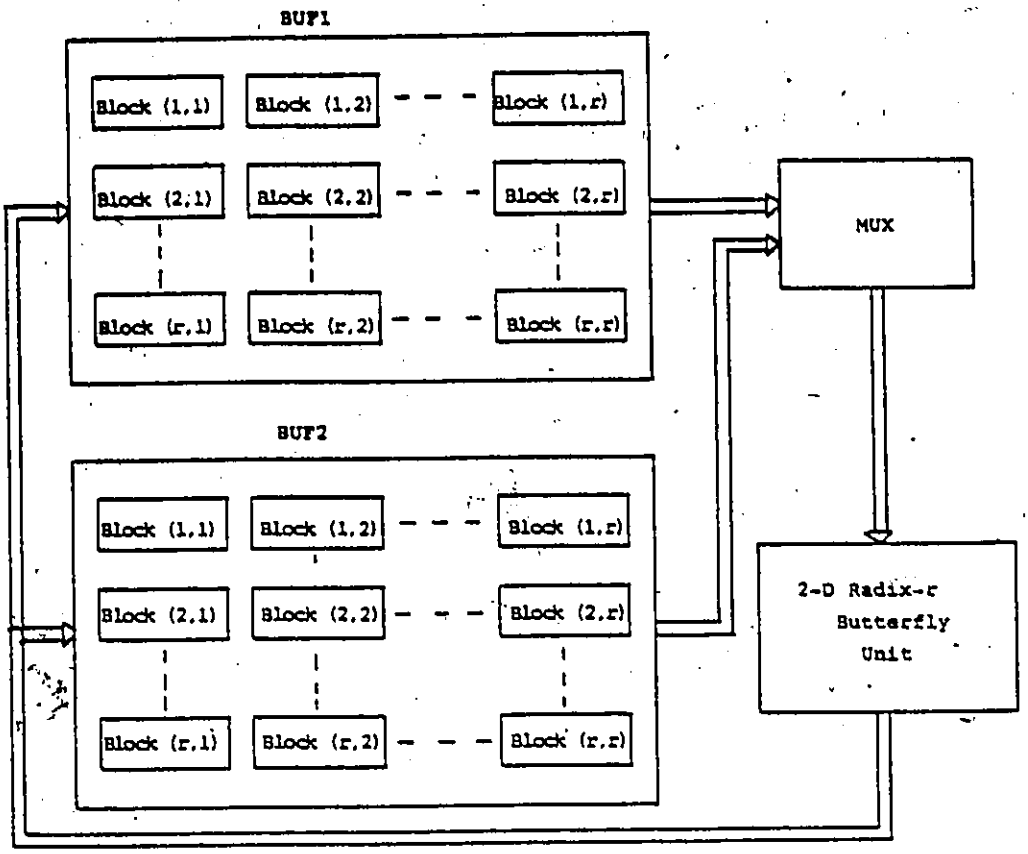


Fig. 4.3a – 2-D radix-r Processor Organization

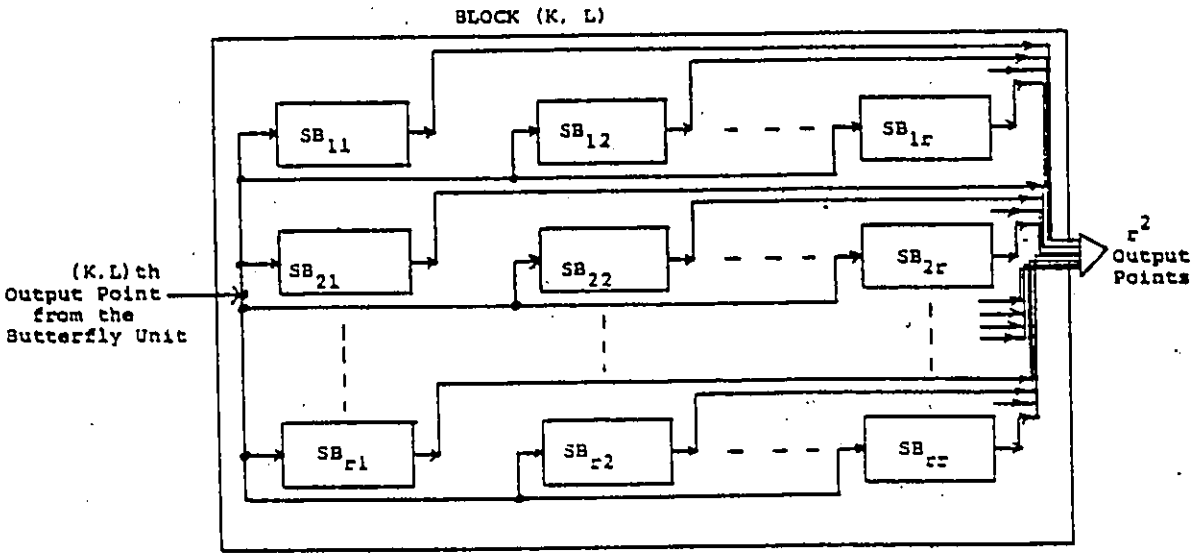


Fig. 4.3b – Sub-division of a block into sub-blocks

refer to the row and column indices of the selected buffer block shown in Fig. 4.3a.

For the jth stage of an (MxM) transform

```
10    DO 100 K = 1,r
      DO 100 I = 1, (r^(j-2)*M/r^2)                    (4.42)
      DO 100 L = 1, r
      {select Block (K,L)}
      DO 100 J = 1, r^(j-2)
      {select the next r^2 points from the r^2 sub-blocks of the
       Block (K,L) in the sequential order and compute a 2-D
       radix-r DFT}
100   CONTINUE
      IF (More data to be processed?) GO TO 10.
```

The above procedure is repeated until the input buffer is empty. Note that for the jth stage we select r^{j-2} points from the selected block consecutively.

The buffer configuration to store the output from the Butterfly unit is fixed and is independent of the FFT stage. The r^2 output points from the Butterfly unit are written into the r^2 blocks of the selected buffer. Each block receives one output point and the sequential output from the Butterfly unit is written into the selected SB sequentially. The procedure to select a SB to store the output point is given in the following FORTRAN like steps, where row and column indices refer to sub-block indices of a block as shown in Fig. 4.3b.

For a transform size of (MxM)

```
DO 100 p = 1,r                                          (4.43)
DO 100 I = 1, M/r^2
DO 100 q = 1, r
DO 100 J = 1,M/r^2
{store the output from the Butterfly Unit in the sub-buffer
  SB_pq of the selected buffer in the sequential order}.
100 CONTINUE
```

At the termination of the above sequence of operations, the selected buffer stores the output of the current FFT stage. This output may now be accessed to process the next FFT stage. When all the FFT stages have been processed, the buffer storing the output of the last FFT stage stores the Fourier coefficients in the natural ascending order.

The operations specified by the operator $R_n^{(r)} = (s^{(r)} \otimes s^{(r)})$ corresponding to the first FFT stage may be implemented by accessing the r^2 points from the selected buffer, in sequential order, as specified by the previous algorithm. Thus the procedure for accessing and storing data for the first FFT stage is the same.

The $(r^2 - 1)$ multiplications by the composite twiddle factors, as specified by the operator $\psi_i^{(r)}$, are performed by the Butterfly unit. These composite twiddle factors may be pre-computed and stored in ROMs. For example, with $r = 2$ and $M = 8$, the composite twiddle factor matrix for the first FFT stage is given by

$$\psi_n^{(2)} = \begin{array}{c} \\ i \downarrow \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \rightarrow j \\ \left| \begin{array}{cccccccc} 1 & 1 & 1 & 1 & w^0 & w^1 & w^2 & w^3 \\ 1 & 1 & 1 & 1 & w^0 & w^1 & w^2 & w^3 \\ 1 & 1 & 1 & 1 & w^0 & w^1 & w^2 & w^3 \\ 1 & 1 & 1 & 1 & w^0 & w^1 & w^2 & w^3 \\ w^0 & w^0 & w^0 & w^0 & w^0 & w^1 & w^2 & w^3 \\ w^1 & w^1 & w^1 & w^1 & w^1 & w^2 & w^3 & w^4 \\ w^2 & w^2 & w^2 & w^2 & w^2 & w^3 & w^4 & w^5 \\ w^3 & w^3 & w^3 & w^3 & w^3 & w^4 & w^5 & w^6 \end{array} \right| \end{array} \qquad (4.44)$$

where the ijth element of this matrix is the composite twiddle factor for the corresponding element of the input matrix. To access these twiddle factors from the ROMs we set up two counters, i and j, where $i,j = 0,1,2,..,\frac{M}{2}-1$. If, after every 2-D radix-2 DFT operation, j is incremented by one and after every M/2 counts of j, i is incremented by one, the composite twiddle factors to be multiplied by the output of the DFT operation are given by 1, w^i, w^j and w^{i+j}. Thus, the required twiddle factors may simply be accessed by addressing the ROMs with i, j and i+j. The twiddle factors required for the other stages may be generated from the twiddle factor matrix of the first stage. For the mth FFT stage we set (m-1) least significant bits of i and j to zero first and then address the twiddle factor ROMs with i, j and i+j. The sequence in which the twiddle factors are multiplied with the output of the 2-D radix-2 DFT operation is shown in Fig. 4.4, where ⊠ indicates a 1-D radix-2 DFT operation and the indices of the input and output

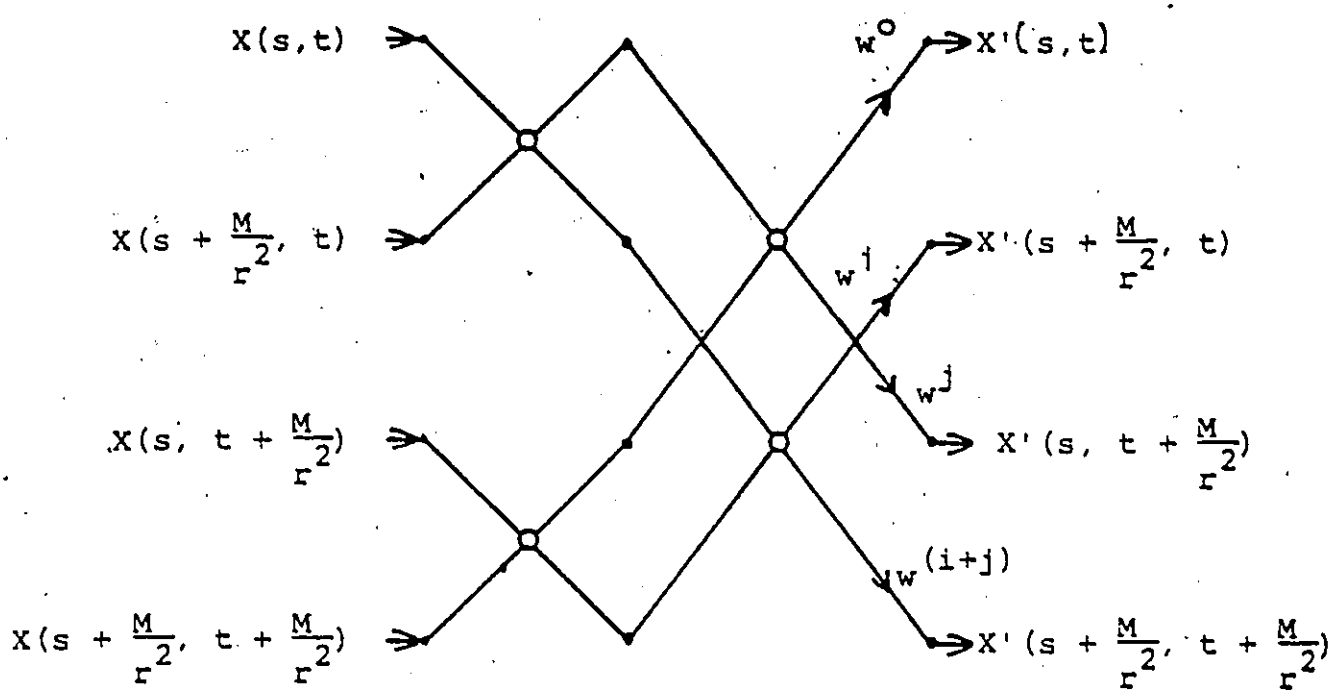


Fig. 4.4 -A 2-D radix-2 Butterfly operation

variables correspond to the row and column indices of the input matrix.

For the last FFT stage the twiddle factors specified by the operator $\psi_1^{(r)}$ are equal to unity and the multiplication operations may be skipped. Fig. 4.5 shows the flow graph of the 2-D radix-2 algorithm for a DFT of size (8x8).

Since a two-buffer organization of Fig. 4.3 is only suitable for the case of non real-time input, we propose a three-buffer memory structure, similar to the 1-D FFT processor organization of Fig. 3.5(a), for real-time inputs as shown in Fig. 4.6(a). Similar to the operation of 1-D FFT processor, two of the buffers of the 2-D FFT processor of Fig. 4.6(a) store the input and output of the intermediate FFT stages. The third buffer is used to collect the sampled input and output the previously computed transform data, simultaneously. All the buffers of Fig. 4.6(a) have the same organization as that of Fig. 4.3(b) except that input data and output Fourier coefficients may now be accessed simultaneously. A buffer organization for real-time input/output operations is shown in Fig. 4.6(b) where M denotes a multiplexer unit. In order to store the input matrix into the selected buffer, each SB must be loaded with the corresponding section of the input matrix. Since the input matrix is generally in the row scanned form, each row of the input matrix can be divided into r^2 sections and each section is stored into the corresponding sub-block, sequentially. When a new input point is to be written into a sub-block, the Fourier coefficient corresponding to the previous FFT operation is recovered first and the new point is

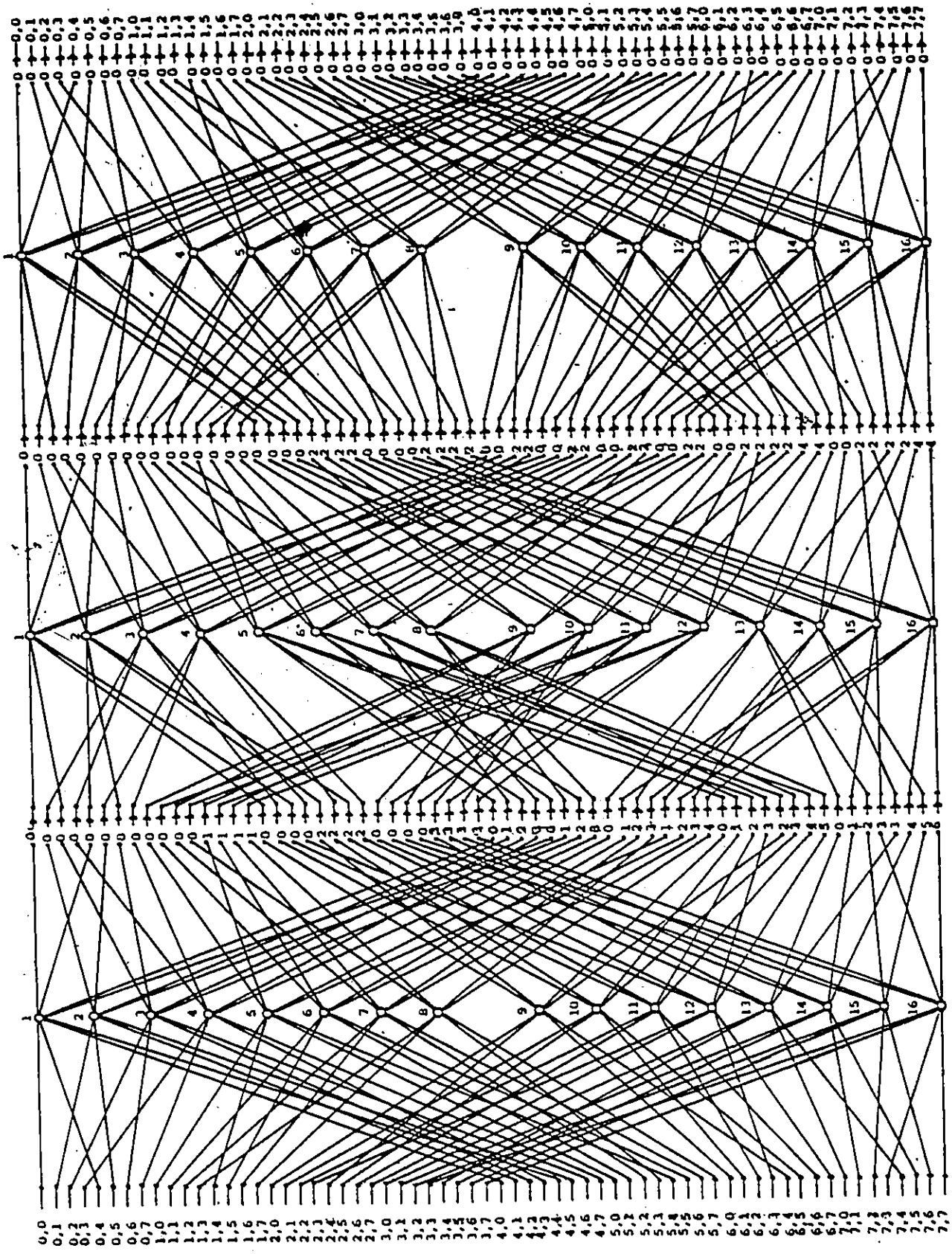


Fig. 4.5 — Flow graph of 2-D radix-2 DIT algorithm for a 8 x 8 array

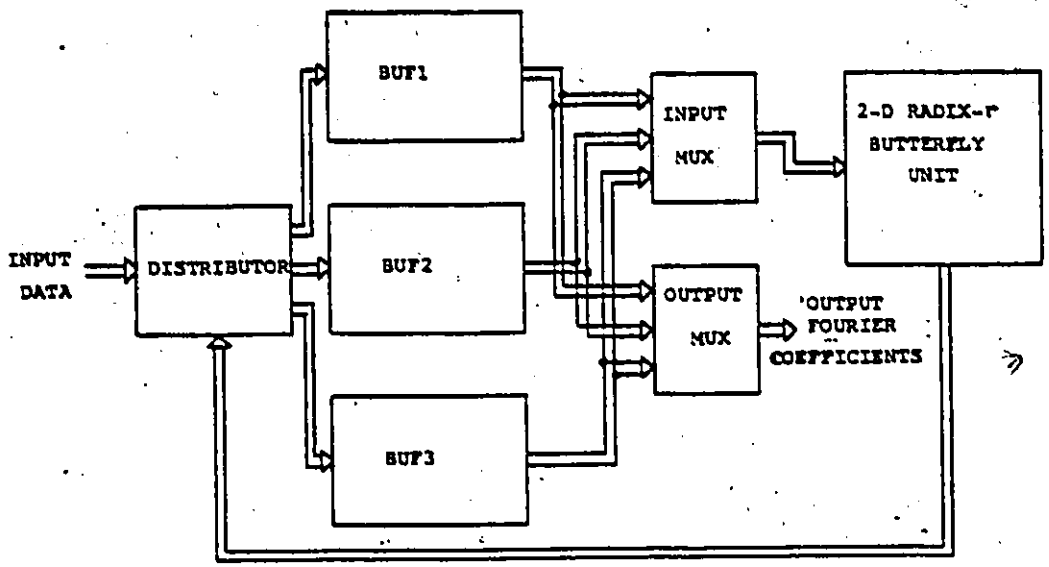


Fig. 4.6a – FFT Processor Organization for Real-Time I/O

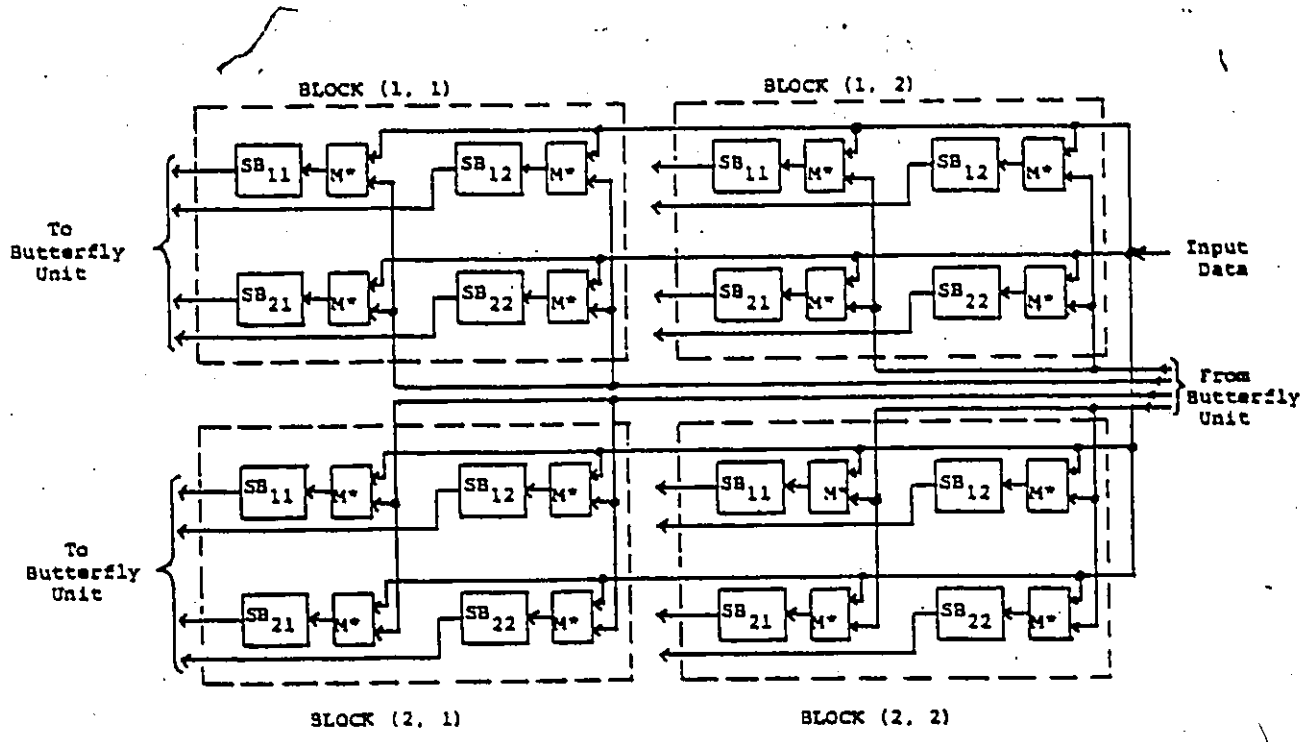


Fig.4.6b – A real-time buffer organization for a 2-D radix-2 FFT processor.
M* - MUX

stored in the same memory location. The Fourier coefficients thus generated are also in the row-scanned naturally ascending order.

Due to the similarity between the 1-D and 2-D FFT processor organizations, the buffer allocation for the real-time operation of the 2-D processor is the same as that of Fig. 3.6 of the 1-D processor. Thus during the computation of a 2-D DFT, the processor organization of Fig. 4.6(a) will perform the input and output operations in real-time.

4.2.4 RNS-Based 2-D FFT Processor Organization

The 2-D FFT processor organization discussed in the previous sections are independent of the architecture of the Butterfly unit. As discussed in Chapter 2, the implementation of the arithmetic operations via RNS offers many advantages over the use of the binary number system. Similar to the architecture of the Butterfly unit of the 1-D FFT processor, we can also organize the 2-D r-point DFT and twiddle factor multiplication operations in a pipelined RNS-based Butterfly unit. This high speed ROM array architecture of the 2-D radix-r Butterfly unit ideally lends itself to the memory organization of the 2-D FFT processor of Fig. 4.6.

The block diagram of the 2-D radix-2 butterfly operation of Fig. 4.4 also represents the arithmetic operations to be performed in a parallel path of the RNS-based Butterfly unit. A ROM array architecture, implementing arithmetic operations for a parallel path, of the 2-D radix-2 Butterfly unit is shown in Fig. 4.7 where O indicates a look-up table (ROM). Similar to the 1-D FFT butterfly structure, the organization

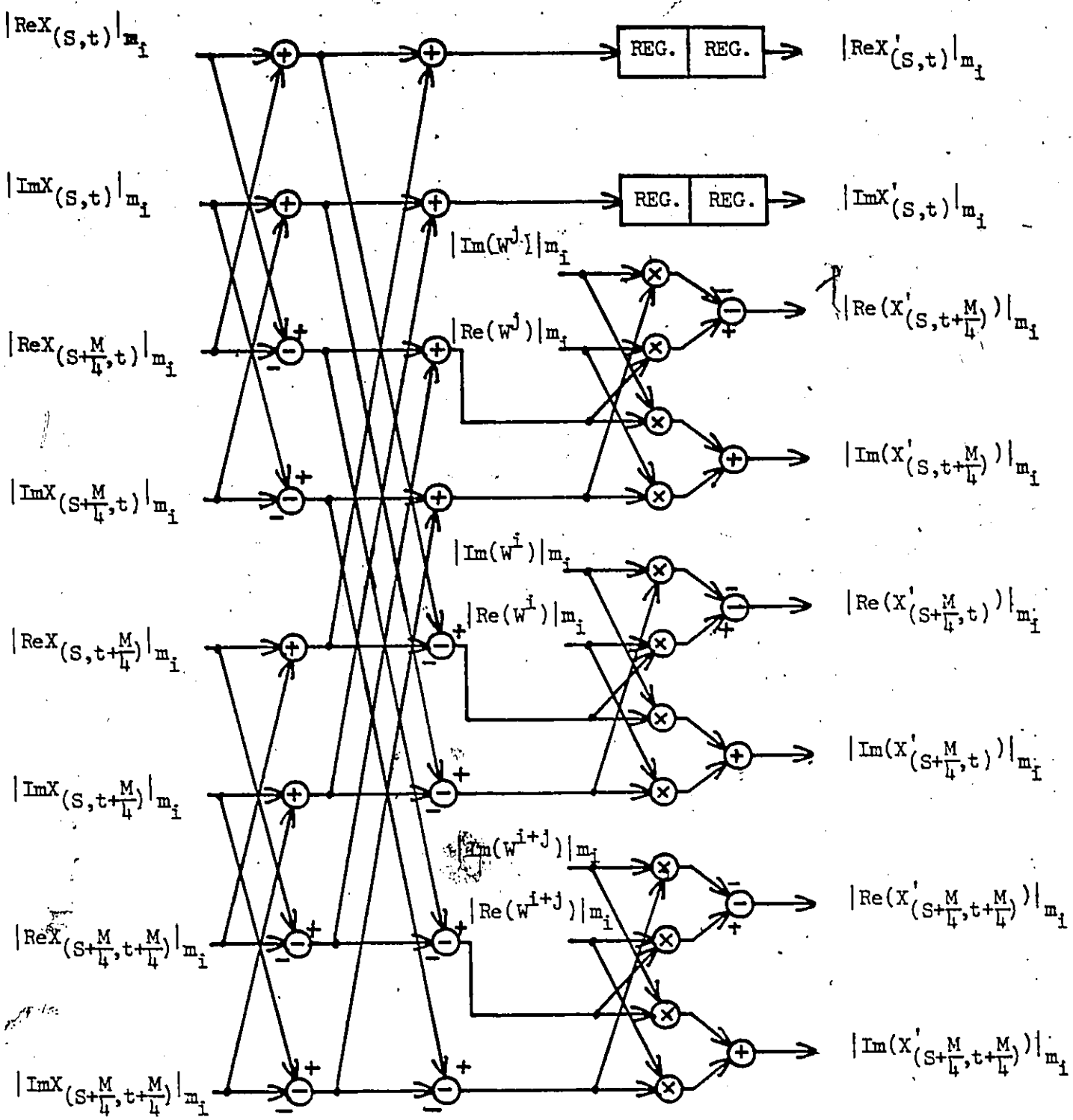


Fig. 4.7 A RNS BASED 2-D RADIX-2 BUTTERFLY STRUCTURE

⊛ INDICATES A ROM PERFORMING THE OPERATION *

a —±⊖→ c INDICATES c = |a-b|

of the 2-D radix-2 Butterfly unit can also be divided into a 4-stage pipeline, and the data through this pipeline may be clocked at a rate of 1/T where T is the ROM access time.

Since, in RNS, scaling of the output of the Butterfly unit is required to avoid any overflow of the RNS dynamic range, the pipelined ROM array implementation of the scaler of Fig. 3.3 for 1-D FFT processor can also be used to scale the output of the 2-D radix-2 Butterfly unit.

When the N-stage pipeline of the scaler of Fig. 3.3 is combined with the 4-stage pipeline of Fig. 4.7, a (N+4)-stage pipeline having a throughput rate of 1/T is obtained where T is the ROM access time.

From the above analysis it is clear that the architecture of the 2-D and 1-D FFT processors are highly similar. Thus if we replace the buffers and Butterfly unit of the RNS-based 1-D FFT processor of Fig. 3.7 with the buffers and Butterfly unit of the 2-D FFT processor, we obtain an RNS-based 2-D FFT processor. Except for some minor differences in the organization of the data paths, the two processors' architectures are functionally similar.

In a 2-D radix-2 (MxM)-point FFT processor, $\frac{M^2}{4}$ 2-D radix-2 butterfly operations are performed in each stage of the FFT algorithm and there are $\log_2 M$ stages. Thus the total time required to compute an (MxM) point DFT is $T \times \frac{M^2}{4} \times \log_2 M$, where T is the inverse of the pipeline throughput rate. For example, for T = 70 ns and M = 256, a (256x256) point DFT can be computed in 9.175 ms which gives a sampling frequency of more than 7 M. samples/sec for the data. Moreover, since

the input matrices are generally real-valued, the throughput can be doubled by processing two sets of input data simultaneously.

4.2.5 Multi-Dimensional Generalized Walsh and Walsh-Hadamard Transform Algorithms

The Walsh-Hadamard transform of section 2.4 has been generalized to a much larger class of unitary transforms. This class of unitary transforms has been termed the Generalized Walsh Transform [6]. In terms of the vector representation, the generalized Walsh transform of a u-dimensional array $F(n_1,n_2,\ldots,n_u)$ of size M in all the dimensions can be expressed in the form of equation (4.8) as

$$p = T\,f \tag{4.45}$$

where f and p are the vector representation of the input and output arrays $F(n_1,n_2,\ldots,n_u)$ and $P(m_1,m_2,\ldots,m_u)$ respectively, and $T=[T_M]^{\otimes u}$, where T_M is the 1-D generalized Walsh transformation matrix. The 1-D generalized Walsh transformation matrix T_M of order q can be generated from the core matrix T_q of order q which is defined by [6]

$$T_q = \begin{vmatrix} w^0 & w^0 & \cdots & w^0 \\ w^0 & w^1 & \cdots & w^{q-1} \\ w^0 & w^2 & \cdots & w^{2(q-1)} \\ \vdots & \vdots & \cdots & \vdots \\ w^0 & w^{q-1} & \cdots & w^{(q-1)^2} \end{vmatrix} \tag{4.46}$$

where $W=e^{2\pi j/q}$ and the elements of T_q are given by $[T_q]_{k\ell}= W^{k\ell}$. Note that the generalized Walsh transform core matrix of order M is same as that of the 1-D DFT transformation matrix of size (MxM) which is defined for the DFT of a M-point sequence. Thus the u-dimensional generalized Walsh transform of order q of an array $f(n_1,n_2,\ldots,n_u)$ of size q in all the u-dimensions can be computed by a u-dimensional FFT algorithm as discussed in the previous sections.

The 1-D generalized Walsh transformation matrix T_M of order q, where $M = q^t$, is defined as [6]

$$T_M = [T_q] \otimes t \tag{4.47}$$

which is a tth order Kronecker product of the core matrix T_q. Since equation (4.47) is of the same form as that of (4.26), an M-point 1-D generalized Walsh transform of order q can be computed as a t-dimensional, $M = q^t$, DFT defined for a t-dimensional array of size (qxqx..xq). For this we must represent the 1-D M-point input sequence as a t-dimensional array which is the inverse of the 1-D vector representation of a t-dimensional array. To represent a 1-D vector f(i),i=0,1,..,M-1, into a t-dimensional array $F(i_1,i_2,..,i_t)$ of size (qxqx..xq) we use the following relation to compute indices i_ℓ, $\ell = 1,2,..,t$ of the array F from the index i of the vector f:

$$i = q^{t-1}.\, i_1 + q^{t-2}.\, i_2 + q^{t-3}.\, i_3 + \ldots + q\,.\, i_{t-1} + i_t \tag{4.48}$$

This is equivalent to finding the t-digit base-q representation of the number i. For example, for t=4, the base-4 representation of the decimal number, 45, is given by $i_1 = 0$, $i_2 = 2$, $i_3 = 3$, $i_4 = 1$. Thus, when

the input vector is represented as a t-dimensional array, we can compute the 1-D generalized Walsh transform by the t-dimensional radix-r FFT algorithm of section 4.2.2 where $q = r^n$.

For the computation of the u-dimensional generalized Walsh transform of order q, equation (4.45) can be written as

$$p = [T_M]^{\otimes u} \cdot f \tag{4.49}$$

Using (4.47), equation (4.49) can be written as

$$p = [\,[T_q]^{\otimes t}\,]^{\otimes u} \cdot f$$

or

$$p = [T_q]^{\otimes u.t} \cdot f \tag{4.50}$$

Since the transformation matrix of equation (4.50) is of the same form as that of (4.26), the generalized Walsh transform given by (4.50) may be computed as (u . t) - dimensional DFT whose 1-D transformation matrix is specified by T_q. For example, a 3-dimensional generalized Walsh transform of order 4 of an array $F(n_1,n_2,n_3)$ of size (16x16x16) is equivalent to a 6-dimensional DFT of an array $F'(n_{11},n_{12};n_{21},n_{22};n_{31},n_{32})$ of size (4x4x4x4x4x4). But, to use the DFT algorithm to compute the generalized Walsh transform we must represent each dimension of the array F as 2-dimensions of array F'.

In section 4.2.2 we saw that, for the computation of a u-dimensional (MxMx..xM)-point DFT, a considerable savings in the number of multiplications may be obtained if the radix-2 butterfly operations in all the u-dimensions are combined to form a u-dimensional butterfly operation. For $q = r^n$,

we can compute the generalized Walsh transform of equation (4.50) by using a (u . t)-dimensional radix-r butterfly operation. This will lead to a saving of $(u \cdot t\ (r^{u.t} - r^{u.t-1}) - (r^{u.t} - 1))$ multiplications over the implementation of equation (4.50) by computing 1-D q-point DFT along all the (u.t) dimensions. For example, for the previous example, u = 3, t = 2, a radix-2 implementation leads to a saving of 67% in the number of multiplications.

The algorithms to compute the multi-dimensional Walsh-Hadamard transform are similar to those for the computation of Generalized Walsh transforms since the core matrix of equation (4.46) reduces to the core matrix of the Hadamard or natural-order Walsh-Hadamard transform for q = 2. Thus from equation (4.46) we have

$$T_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4.51}$$

which is same as the core matrix of equation (2.27). Similar to equation (4.50), the u-dimensional Hadamard transform of an array $F(n_1,n_2,\ldots,n_t)$ of size, $M = 2^t$, in all the u-dimensions can be written as

$$p = [T_2]^{\otimes u.t} \cdot f \tag{4.52}$$

and hence can be computed as a generalized Walsh transform of order 2.

From equation (4.51) and (4.52) it is clear that the computation of (4.52) does not require multiplication by the twiddle factors and can be computed by addition and subtraction of the elements of the u-

dimensional array. Since the number of additions and subtractions are constant and independent of the implementation of equation (4.52), we can also implement (4.52) as

$$p = [T_M] \overset{\otimes u}{\cdot} f \tag{4.53}$$

by using a different factorization of the 1-D transformation matrix T_M. By using the factorization of the natural-order Walsh-Hadamard transformation matrix proposed by Geadah [22] we can develop an algorithm similar to the DFT algorithm of section 4.2.2. Since in Chapter 3 we showed that a 1-D dyadic-ordered Walsh-Hadamard transform can be implemented on a 1-D FFT processor, we show here that a 2-D dyadic-ordered Walsh-Hadamard can also be implemented on the 2-D serial sequential FFT processor of section 4.2.3.

Substituting the factorization of the 1-D dyadic-ordered Walsh-Hadamard transformation matrix T_M given in equation (3.34) into (4.53) we get

$$p = \left[\prod_{i=1}^{n} s_i \right] \overset{\otimes u}{\cdot} f \tag{4.54}$$

where $s_{i-1} = p_i s'$ is defined in equation (3.36). Since s_i of (4.54) and $s_m^{(r)}$ of (4.30) are exactly the same operator for $r = 2$, we can write (4.54) in the form of (4.38) as

$$p = \left\{ \prod_{i=1}^{n} R_i^{(2)} \right\} \cdot f \tag{4.55}$$

where $R_i^{(2)}$ is defined in equations (4.35) and (4.36). Except for the twiddle factor operator $\psi_i^{(r)}$, equations (4.55) and (4.38) specify the

same operations when r = 2. Thus if the twiddle factor multiplications are suppressed, the 2-D radix-2 FFT processor of section 4.2.3 will also compute the dyadic-order Walsh-Hadamard transform of an array of size (MxM).

4.2.6 Other Multi-Dimensional Unitary Transform Algorithms

The factorization of the multi-dimensional transformation matrix T of equation (4.8) is not limited to the DFT, Walsh-Hadamard and generalizated Walsh transforms as discussed in the previous sections. Since it is known that the 1-D transformation matrix T_M of a number of unitary transforms, such as generalized transforms [7], Cosine and Sine [31], Haar and generalized Haar [8], can be specified as a product of sparse matrices, equation (4.21) can be used to compute these multi-dimensional unitary transforms also. Equation (4.21) specifies that a separable unitary transform can be computed in stages where each stage performs the specified arithmetic operations on the data simultaneously along all the dimensions of the multi-dimensional array as opposed to the conventional technique of computing the 1-D unitary transform along all the dimensions, successively. For a multi-dimensional array of size $M = r^n$, in all the dimensions, this leads to a large saving in the number of multiplications for the DFT and generalized Walsh-transforms as shown in the previous sections. To achieve this we used the symmetric property of the factors of 1-D transformation matrices of the DFT and generalized Walsh transforms and combined the sequential butterfly operations in all the dimensions into a multi-dimensional butterfly

operation. Since generalized, Sine and Cosine transforms can be computed using an FFT algorithm, a saving in the number of multiplications can also be achieved in the computation of multi-dimensional generalized, Sine and Cosine transforms. In the case of other unitary transforms, it is not clear whether savings in the computational effort can be achieved or not because the factors of their transformation matrices do not exhibit any symmetry properties [7],[8]. But the factorization techniques discussed in the previous sections may be applied to develop special purpose processor architectures for high speed applications.

4.3 SUMMARY

The development of fast algorithms for computing a class of multi-dimensional unitary transforms, having separable kernels, has been presented. The technique is based on the factorization of the multi-dimensional transformation matrix into Kronecker products of one-dimensional transformation matrices. For a number of unitary transforms, including Fourier, Walsh-Hadamard and generalized Walsh, the one-dimensional transformation matrix can be expressed as a product of sparse matrices. It has been demonstrated that the one-and multi-dimensional factorizations of the transformation matrix can be combined, resulting in fast algorithms for multi-dimensional unitary transforms. These algorithms compute the unitary transform of a u-dimensional array of size M in each dimension, $M = r^n$, in n-stages, where each stage operates on the data in each dimension.

Based on the fast multi-dimensional unitary transform algorithms, the development of multi-dimensional Fourier, Walsh-Hadamard and generalized Walsh transform algorithms have been presented. It is shown that the u-dimensional Fourier and generalized Walsh transforms can be implemented with a u-dimensional radix-r butterfly operation, which requires considerably less complex multiplications that the conventional implementation using one-dimensional radix-r butterfly operation. The savings in complex multiplications is a function of both the radix, r, and the number of dimensions, u, and it is greater for higher radices and larger dimensions.

The nature of the fast algorithms for one-, two- and multi-dimensional Fourier, Walsh-Haramard and generalized Walsh transforms resulted in almost identical processor architectures. Based on the fast algorithms, the development of new two- and multi-dimensional Fourier, Walsh-Hadamard and generalized Walsh transform processor architectures have been presented. It has been demonstrated that these processor architectures are capable of processing large band-width input sequences in real-time. It is also shown that the factorization techniques developed in this work ean be easily extended to the development of special purpose processors for the computation of a number of other unitary transforms.

CHAPTER 5

HIGH-SPEED CONVOLUTION

5.1 INTRODUCTION

Two-dimensional convolution is one of the important operations in image processing, which is widely used in image enchancement, image restoration and feature extraction. For very high speed filtering operations FFT technique is normally used since very high throughput rate may be obtained by the use of special purpose hardwired FFT processors. In the past most of the published work concentrated on the implementation of the convolution operations on a general purpose processor, resulting in a slow processing speed. In this chapter we investigate the use of 2-D FFT processor, discussed in Chapter 4, to perform convolution of a large 2-D input matrix with a filter impulse-response for real-time filtering operations. In Chapter 3 we showed that the 1-D FFT processor can be used to compute convolution by the overlap-save method of sectioned convolutions, by providing extra buffers at the input and output of the FFT processor. In this Chapter we show that with the provision of a special memory organization we can also compute 2-D convolutions by the overlap-save technique of sectional convolutions. Since a 2-D DFT or convolution can also be performed by using a 1-D FFT processor, we show here that the 2-D FFT processor of Chapter 4 is more cost effective than the 1-D FFT processor

of Chapter 3.

5.2 TWO-DIMENSIONAL CONVOLUTION USING A 2-D FFT PROCESSOR

The periodic convolution of two 2-D periodic sequences, f and h, of period M in both the dimensions is defined as

$$g(m,n) = \sum_{k=0}^{M-1} \sum_{\ell=0}^{M-1} f(k,\ell)\, h(|m-k|_M, |n-\ell|_M) \tag{5.1}$$

Similar to the 1-D case, the DFTs of the sequences, g, f and h, are related by

$$G(m,n) = H(m,n) \cdot F(m,n) \; ; \quad m,n = 0,1,2,\ldots,M-1 \tag{5.2}$$

where G, H and F are the DFTs of the sequences g, h and f respectively. The sequence, g, may be computed by taking the inverse DFT of G.

In image processing applications, the 2-D sequences are generally of finite duration and we are interested only in computing the linear rather than the periodic convolution. The above technique of computing periodic convolution can also be used to compute linear convolution by considering the finite duration sequences as periodic sequences of period M in both dimensions, with $M \geq (N+L-1)$, where the finite duration sequences f and h are of size (NxN) and (LxL). A period of an (MxM) sequence, f', may be generated from the finite duration sequence, f, by the relation

$$f'(m,n) = \begin{cases} f(m,n), & m,n = 0,1,\ldots,N-1 \\ 0, & m,n = N, N+1,\ldots,M-1 \end{cases} \tag{5.3}$$

Using (5.3) we can also generate an (MxM) sequence, h', from the (LxL) sequence h. If the (MxM)-point DFTs of the sequences f' and h' are denoted by F' and H' respectively, then, using (5.2) we can compute the sequence G' = F' · H' and the inverse DFT of G' will result in the desired output sequence g.

In many situations the size, (NxN), of the sequence f, is much larger than the size, (LxL), of the sequence h. It becomes inefficient and impractical to compute DFT's of sequences of size (MxM), where M is large, because a large amount of memory is required for its implementation on a general purpose or a special purpose processor. In these circumstances, the overlap-add or overlap-save method of sectioned convolutions may be used to compute the convolution. As discussed in Chapter 3, we preferred the overlap-save method over the overlap-add method to implement 1-D convolutions since the overlap-add method requires an extra operation of addition of the partial results. In this section, we investigate the implementation of 2-D convolutions using the overlap-save method of sectioned convolutions.

For the implementation of a 2-D convolution by the overlap-save method, let f and h be the input data and filter impluse-response sequences, respectively, and N >> L. If the FFT processor can perform a 2-D DFT of size (MxM), the input data matrix, f, is divided into overlapped sections of size ((M-L+1)x(M-L+1)) and the different sections of the input matrix may be convolved individually. Since the input matrix is generally real-valued, two sections of f may be processed

simultaneously. Fig. 5.1 shows the sub-division of the input matrix of size (NxN) where the shaded area is the overlap between the sections. For the simultaneous processing of two data sections, sections 1 and 2, for example, may be combined to form the real and imaginary part of the input to the FFT processor. Similarly sections 3 and 4, (q+1) and (q+2) may be processed simultaneously. Note that when the sections are processed in the row direction i.e. in the sequence 1-2-3-..., a set of (L-1) rows and (L-1) columns overlaps adjacent sections, and hence must be saved. Assuming that the same data memory is used to store the original and the filtered matrix, a separate memory buffer to save part of the input data is required.

The memory containing the input matrix is divided into pq sections where q is an even integer with $q \geq \{\frac{N}{(M-L+1)}\}$, and each section contains (M-L+1) columns. The save memory for the input data consists of three buffers SV_1, SV_2 and SV_3 of size $\{(M-L+1)x(L-1)\cdot q/2\}$, $\{(M-L+1)x(L-1)\cdot q/2\}$ and $(Mx(L-1))$, respectively. Fig. 5.2a shows memory sections containing different sections of the matrix of Fig. 5.1. The save memory buffers SV_1 and SV_2 are divided into q/2 sections, each of size $\{(M-L+1)x(L-1)\}$, and buffer SV_3 consists of two sections CS_1 and CS_2 as shown in Fig. 5.2b.

With the above provision of the data and the save memory buffers, a 2-D convolver organization, using the FFT processor organization of Fig. 4.6a, is given in Fig. 5.3. At the start of the convolution process, the buffers SV_1, SV_2 and SV_3 of the save-memory Ss are

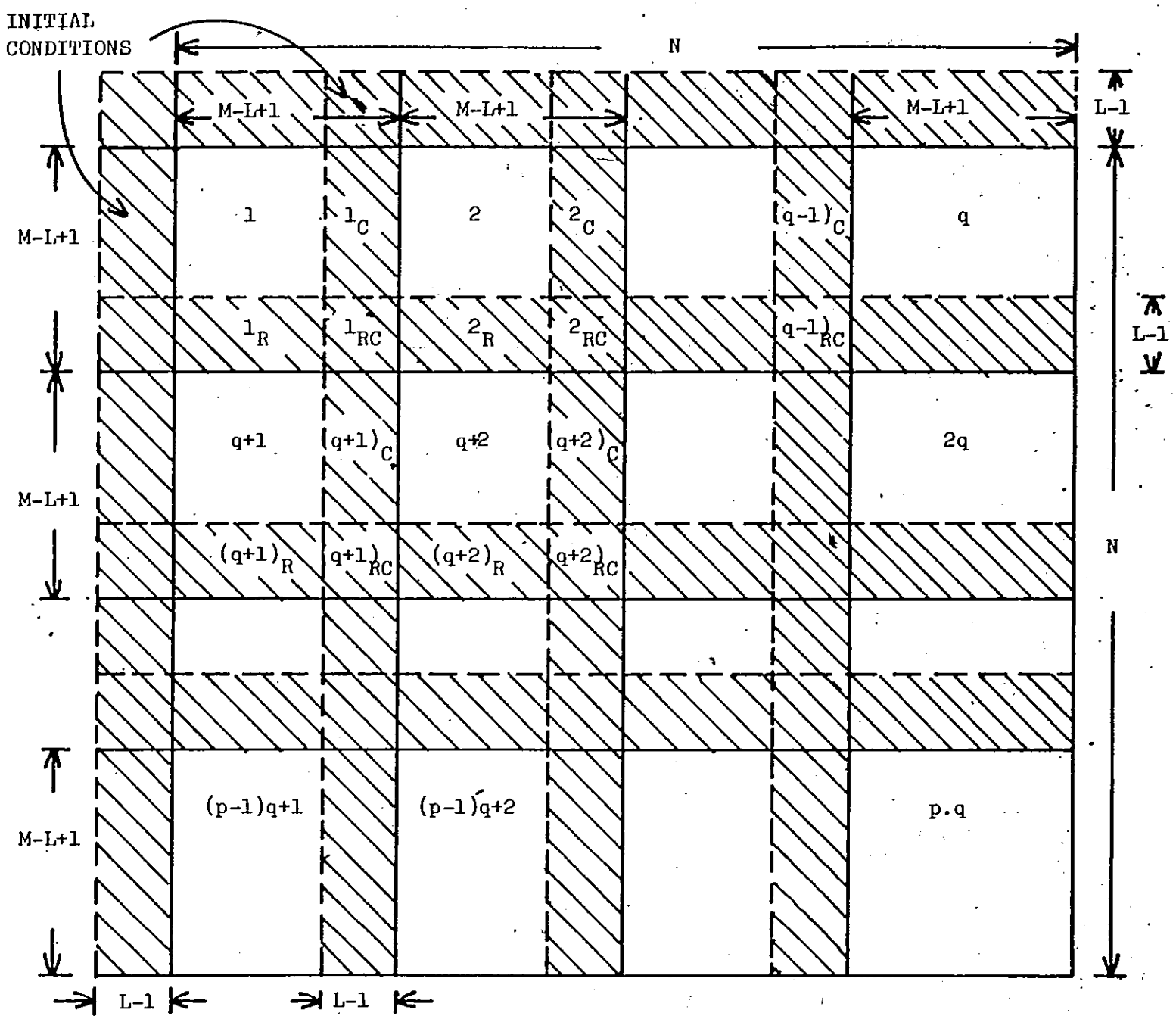


Fig. 5.1 SECTIONS OF AN (NxN) MATRIX.

1	2	3		q
q+1	q+2	q+3		2q
(p-1)q+1	(p-1)q+2	(p-1)q+3		p · q

Fig. 5.2(a) INPUT DATA MEMORY SECTIONS

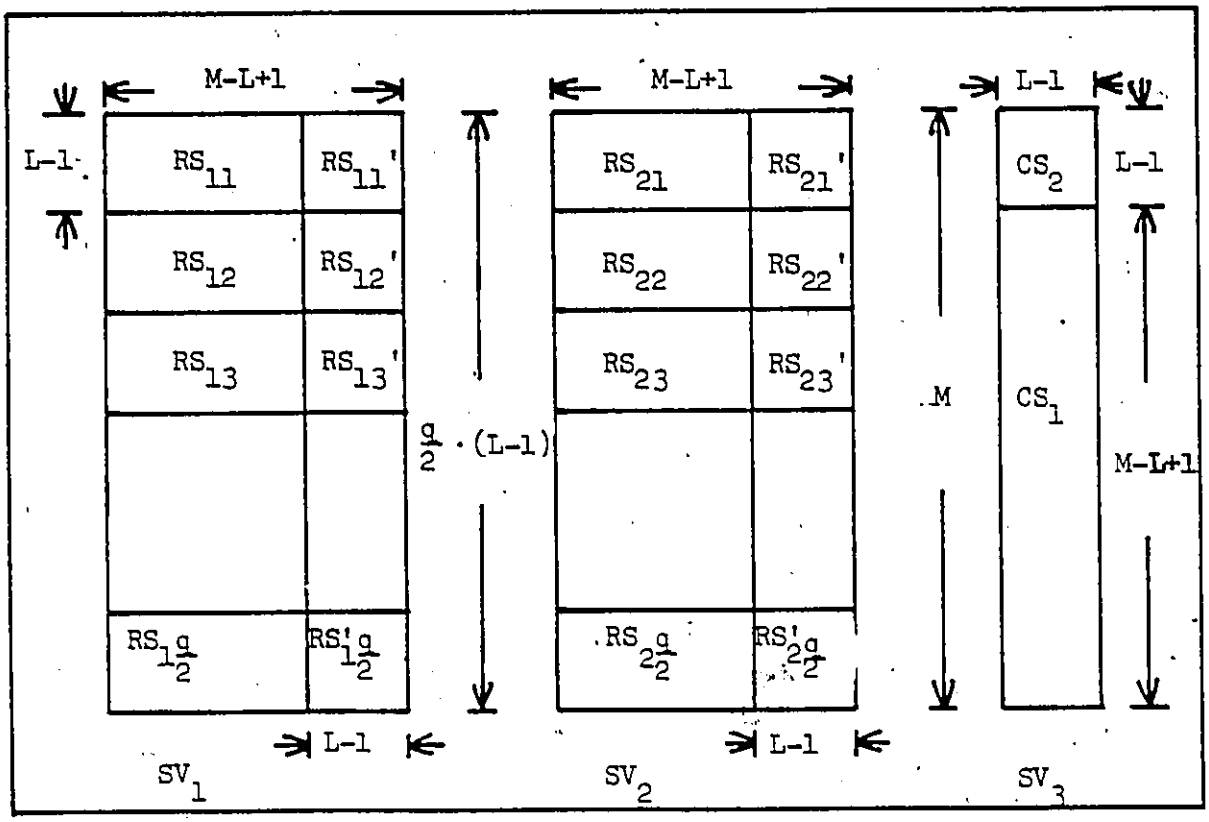


Fig. 5.2(b) SAVE MEMORY SECTIONS

initialized to zero and the process begins by transferring data from save-memory to the FFT processor. The M complex data points to the FFT processor are formed by accessing the first row of CS_2, RS_{11}, RS_{11}', RS_{21}, and RS_{21}' as shown in Fig. 5.4a. The data in RS_{21}' is also saved in CS_2. In a similar fashion the next set of M-points, corresponding to the 2nd row, are accessed from the save-store and transferred to the FFT processor. The above process is repeated (L-1) times which corresponds to transferring the first (L-1) rows of the data sections 1 and 2 (shown as initial conditions in Fig. 5.1).

After the above data transfers have been completed, we start transferring data from the CS_1 section of SV_3 and data section 1 and 2 by forming the real and imaginary parts of the input to the FFT processor as shown in Fig. 5.4b. We also save the last (L-1) points corresponding to the partition 2_C of the data section #2 in CS_1. This process of data transfer is repeated for all the (M-L+1) rows in data sections 1 and 2, but for the last (L-1) rows the data, corresponding to 1_R, 1_{RC}, 2_R and 2_{RC} partitions of Fig. 5.1, is also saved in RS_{11}, RS_{11}', RS_{21} and RS_{21}' sections of the save memory. Fig. 5.5 shows the formation of a block of size MxM for input to the FFT processor and the partitions of this block correspond to the various sections of the data and the save memory.

At the end of the above transfers, the FFT processor starts computing the DFT and the inverse DFT of the (MxM) block just transferred. While the FFT processor is performing the convolution, the next block

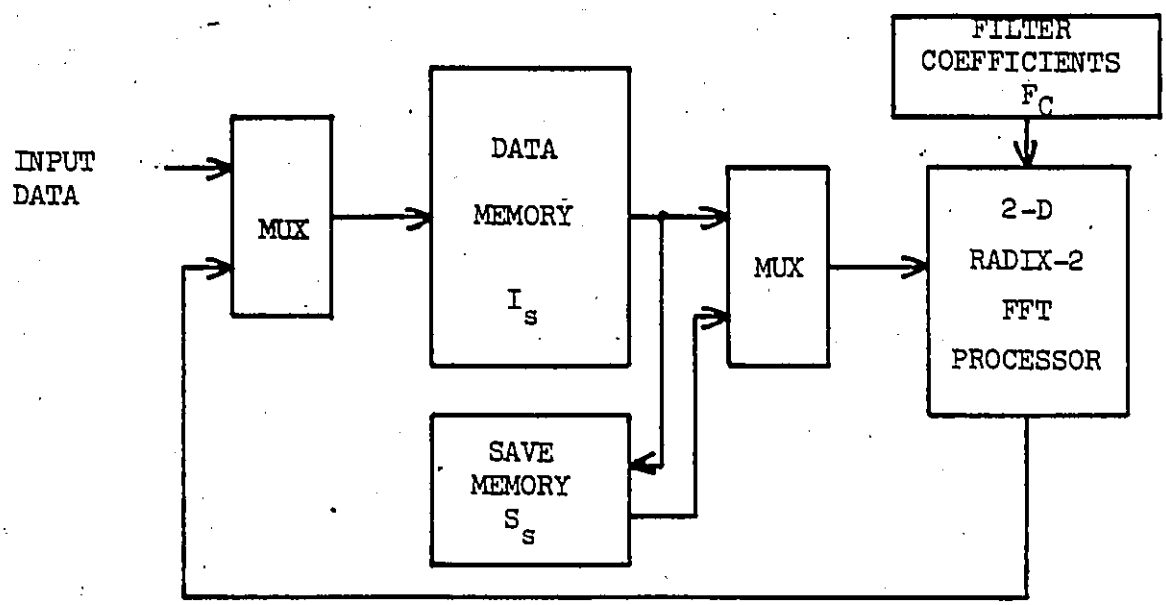
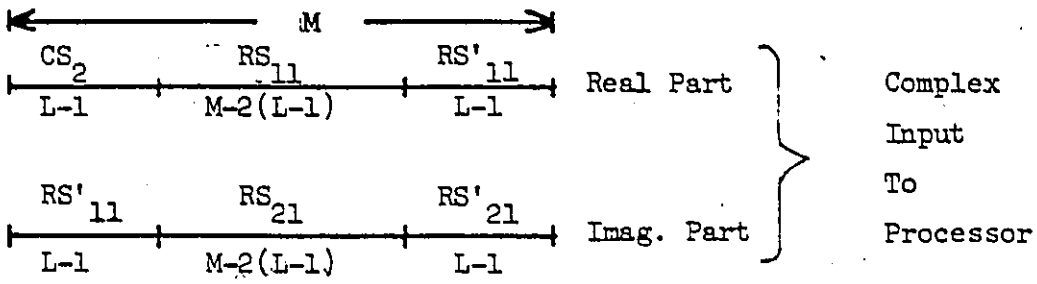
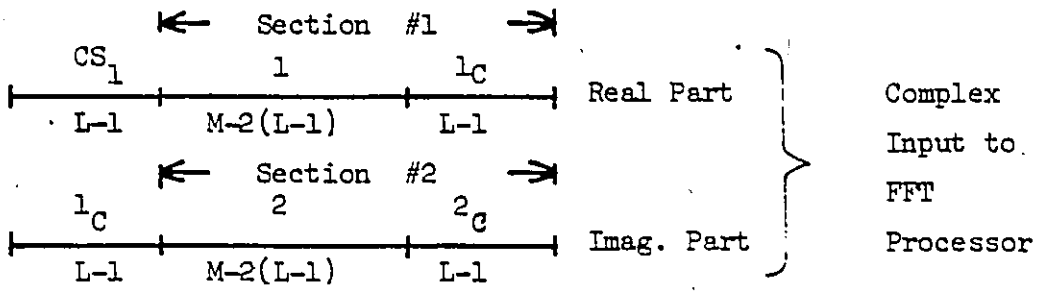


Fig. 5.3 A 2-D RADIX-2 CONVOLVER ORGANIZATION



(a)



(b)

Fig. 5.4 FORMATION OF COMPLEX INPUT TO THE FFT PROCESSOR FOR SIMULTANEOUS PROCESSING OF TWO DATA SECTIONS

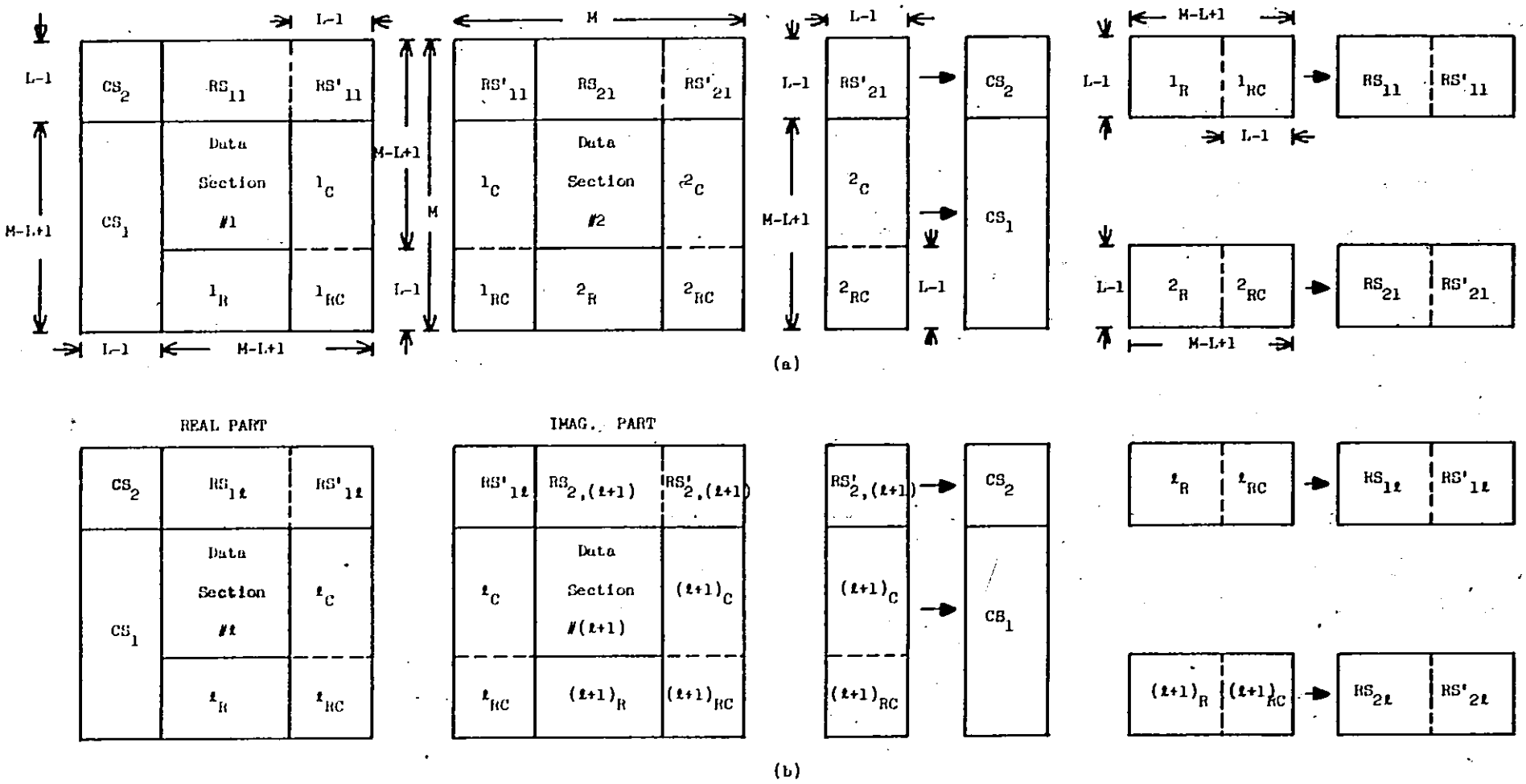


Fig. 5.5 FORMATION OF A (MxM) BLOCK USING DATA ACCESSED FROM (a) THE DATA SECTION 1 AND 2 (b) THE DATA SECTIONS ℓ AND (ℓ+1), AND THE SAVE MEMORY SECTIONS .

of data corresponding to data sections 3 and 4, as shown in Fig. 4.5b is formed. Fig. 5.6 shows the timing diagram and buffer allocation for convolution of successive data sections. As shown in Fig. 5.6, input, output, and filtering operations are performed concurrently. At the completion of a filtering operation, the FFT processor starts processing a new block of data and the filtered data corresponding to the previous block becomes available at the output of the FFT processor. This data is written back into the corresponding sections. Only the output corresponding to the last (M-L+1) rows and columns is stored, since the first (L-1) rows and (L-1) columns of the filtered output correspond to the overlapped portion of the data sections.

The above process of filtering two data sections at a time is repeated q/2 times and the overlapped area between the data sections is saved in the save-memory. Since the saved portion of the data sections in the first row corresponds to the initial conditions for the 2nd row of data sections, the convolution process for the 1st row of q sections is repeated for all the p rows of q sections. The only requirement is that the save-memory sections CS_1 and CS_2 must be re-initialized to zero at the start of a new row of q sections. At the end of the pth row, the filtered data is in the data memory and this process may be repeated for the next input matrix.

From Fig. 5.6 it is clear that, to keep the FFT processor busy all the time, we must be able to read and write two successive sections of the input matrix, simultaneously. This can be achieved if the

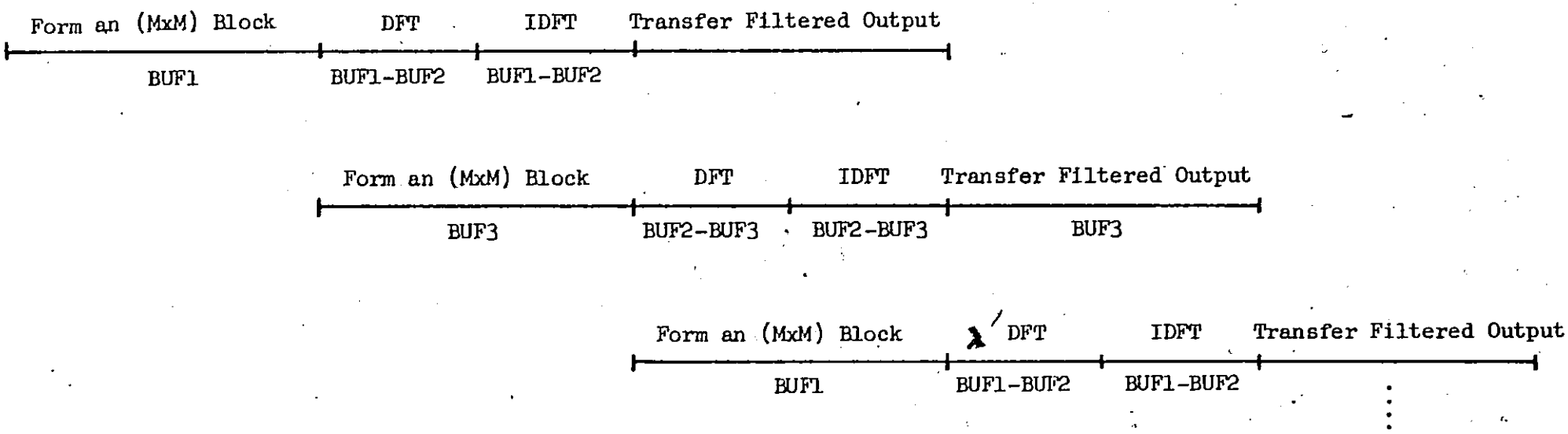


Fig. 5.6 FFT PROCESSOR'S BUFFER ALLOCATION AND TIMING DIAGRAM FOR PROCESSING SUCCESSIVE DATA SECTIONS.

individual sections of Fig. 5.2a are stored in separate memories. For large matrices the number of sections, p·q, may be large, resulting in a large number of separate memories. In many situations the input matrix is stored in a large random access memory with a single read/write port. In this case either the memory access time must be 1/4th of the input/output time of the FFT processor or the memory must have four read/write ports so that four data words may be accessed/stored simultaneously. Since the cost of a four port memory or memory with an access time of 1/4th of the input/out time of the FFT processor may be very high, we will find another way of storing the sections of the input matrix. In the new technique, we store the input matrix into two separate memories where each word of a memory stores two data points of the input matrix. To store the input matrix we pack data points from the two successive sections of the input matrix and store the packed sections into the two memories as follows:

Memory 1	Memory 2
(1,2),(3,4),(9,10),(11,12)	(5,6),(7,8),(13,14),(15,16)
(17,18),(19;20),.....	(21,22),(23,24),.....

For example, sections (1,2),(3,4),(9,10),... are packed and stored in Memory 1 and sections (5,6),(7,8),(13,14),... are stored in Memory 2. From Fig. 5.6 we note that the two memories are never read and written simultaneously. Thus, the provision of two memories with an access time equal to the input/output time of the FFT processor seems to be a cost effective solution.

The convolution process also requires the multiplication of the DFT of an (MxM) block by the DFT of the filter impulse-response stored in F_c. Since there are no multiplications by the twiddle factors in the last FFT stage, the multiplication of the two DFTs may be performed in the last FFT stage. An additional set of ROMs to perform a complex-multiplication is required since there are only (r^2-1) multipliers in the 2-D radix-r Butterfly unit. Since the twiddle factors for the DFT and inverse DFT are never required simultaneously, we can store these twiddle factors in a single set of ROM's. The ROM's storing the twiddle factors and the ROM's storing the filter coefficients, may be multiplexed at the input to the complex multipliers.

From section 4.2.4, we know that for a transform size of (MxM), a 2-D radix-2 FFT processor takes $T \times \frac{M^2}{4} \times \log_2 M$ ns where T is the inverse of the pipeline throughput rate of the FFT processor. Since pq/2 convolution operations are required for an input matrix size of (NxN), where a convolution operation specifies an (MxM)-point DFT and inverse DFT operations, the total time required to filter a matrix is given by

$$T = p \cdot q \cdot \frac{M^2}{4} \cdot T \cdot \log_2 M \text{ ns} \qquad (5.4)$$

For example, for an input matrix size of (256x256) and impulse-response size of (17x17), a (64x64)-point 2-D radix-2 processor, with T = 70 ns, will take about 15.4 ms to filter the input matrix. This gives a throughput rate of 64 matrices/sec. To achieve this processing speed

it is necessary that the FFT processor's pipeline must be kept full and for this the input data and save memory access time should be less than or equal to 210 ns. Also, T = 70 ns assumes that the internal buffers BUFF1, BUFF2 and BUFF3 of the FFT processor also have an access time of 70 ns.

5.3 2-D CONVOLUTION USING 1-D FFT PROCESSOR

In Chapter 4 we showed that a 2-D DFT can also be computed by taking the 1-D DFT of the rows and then the columns of the input matrix. The 1-D DFT of the rows and column can be computed by a special purpose FFT processor such as the one discussed in Chapter 3. Thus, a 2-D convolution can also be implemented by using the above technique and the overlap-save method discussed earlier. In this section we investigate the implementation of a 2-D convolution using the 1-D FFT processor of Chapter 3. The motivation behind this investigation is to determine the cost effectiveness of the previously discussed processor with the more conventional realization. In order to obtain an accurate assessment, the following design is discussed in detail.

The organization of a 2-D convolver using the 1-D FFT processor of Fig. 3.7 and the memories storing the input data and the save memory of the previous section is shown in Fig. 5.7. To store the partial result of the 2-D DFT and the IDFT, another memory, T_s, of capacity (MxM) complex words is also required. Since a 2-D transform of any section is performed by taking the transform of all

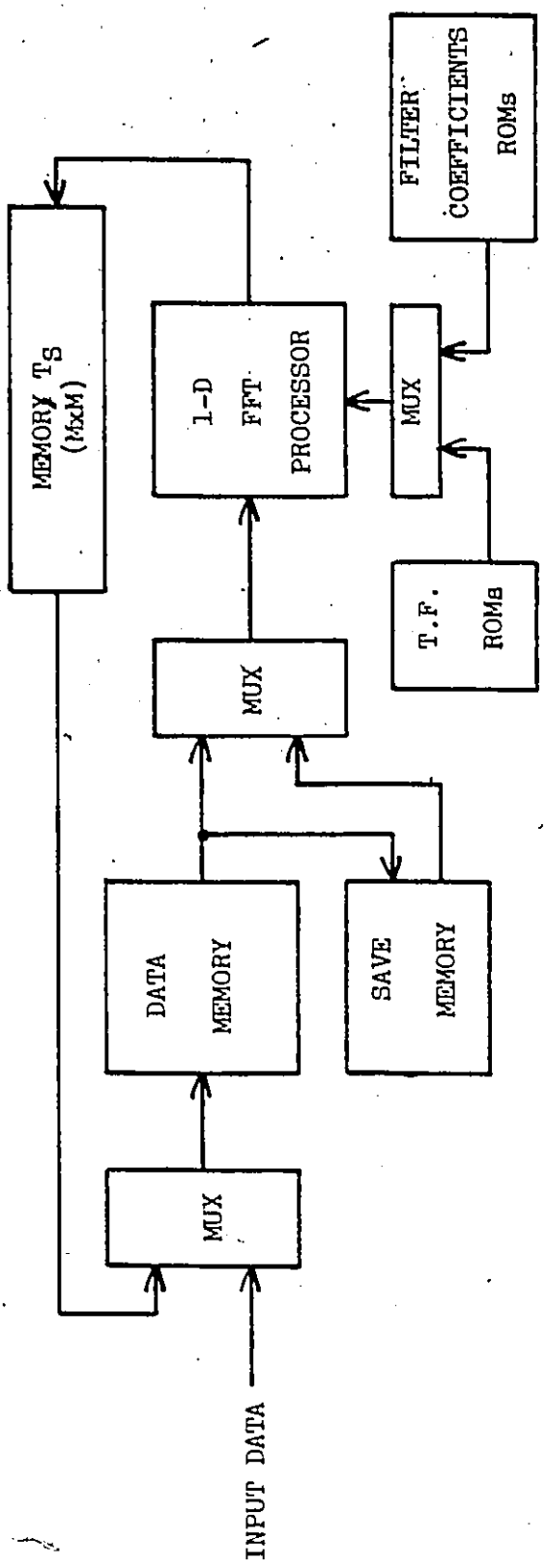


Fig. 5.7 A 2-D CONVOLVER ORGANIZATION USING THE 1-D FFT PROCESSOR

the rows first and then the columns, the result of a row or column transform is always stored in T_s.

At the start of a convolution, data, to the 1-D FFT processor, is formed as complex words, as shown in Fig. 5.4. After a row of the (MxM) block of Fig. 5.5 has been transferred, the FFT processor starts computing the DFT of this row. While the FFT processor is computing the DFT of the 1st row, the 2nd row of the (MxM) block is fed to the FFT processor. At the completion of the 2nd row transfer, the FFT processor completes the DFT of the 1st row and starts computing the DFT of the 2nd row. Also the result of the 1st transform is simultaneously outputted by the processor and is stored in the 1st row of the complex store. Thus the FFT processor can be operated continuously with the input, output and DFT operation being performed concurrently, as shown in Fig. 3.6. The above process is repeated for a complete block of (MxM)-points. The (MxM) blocks are always formed according to Fig. 5.5 and the result of the row transforms is stored in the memory, T_s.

At the end of the row transforms of a block, the processor is ready to perform the column transforms of the result in T_s. The above process for the row transforms is repeated for the column transforms, a column at a time. Note that the data from the next column of T_s is read and the result of the previous column transform is written simultaneously, thus requiring that T_s have either two ports or an access time of one half the input/output time of the 1-D FFT processor.

The convolution process also requires multiplication of the column transforms by the filter coefficients stored in F_C. This multiplication can be done in the last stage of the column transform, similar to the 1-D convolution discussed in Chapter 3. Thus, at the end of the column transforms, the product of the DFTs of a block and the filter impulse-response will be available in T_s. In a similar fashion the inverse DFT of the columns and the rows may be performed. After the convolution of a block is completed we can store the filtered output back into the data memory. Similar to the 2-D convolution of the previous section we repeat the above process for the other sections of the input matrix.

From equation (3.5) we see that a 1-D radix-r FFT processor requires $(T \cdot \frac{M}{r} \cdot \log_r M)$ ns to perform an M-point transform, where T is the inverse of the pipeline throughput rate of the 1-D FFT processor. Hence to compute a 2-D FFT and an inverse FFT of an (MxM) block, it will take $(4 \cdot T \cdot \frac{M^2}{r} \cdot \log_r M)$ ns. Thus, for p.q/2 blocks the total time required is given by

$$T = 2 \cdot pq \cdot \frac{M^2}{r} \cdot T \cdot \log_r M \text{ ns} \qquad (5.5)$$

Since the complexity of a 1-D radix-4 Butterfly unit is same as that of 2-D radix-2 Butterfly unit, we choose a radix-4 1-D FFT processor implementation. Thus for r=4,(5.5) can be written as

$$T = p \cdot q \cdot \frac{M^2}{2} \cdot T \cdot \log_4 M \text{ ns} \qquad (5.6)$$

Comparing equations (5.6) and (5.4) we see that the time required to compute a convolution is the same for an (MxM)-point 2-D radix-2 and

an M-point 1-D radix-4 FFT processor. Thus, taking the example of the previous section, a 64-point 1-D radix-4 FFT processor will take 15.4 ms. To achieve this rate the 1-D processor's pipeline must be kept full, which requires that the access time of data memories storing the input matrix, and the memory, T_s, must be less than or equal to 52.5 ns and T_s must be dual port. Clearly, these memory requirements represent a major cost factor in the processor design and thus place this 1-D approach at a much higher cost than the previously discussed 2-D design.

The constraint of a small memory access time on the data memories and the memory T_s may be relaxed by the prevision of 3 memory units; T_{s1}, T_{s2}, and T_{s3}, as shown in Fig. 5.8. In this organization of the 2-D convolver, each of the memories, T_{s1}, T_{s2} and T_{s3} has a storage capacity of (MxM) complex words. These memories are used to: (a) receive input data blocks of size (MxM) from the data memory D_s storing the input matrix; (b) supply and receive data from the 1-D FFT processor and (c) output the filtered block. Fig. 5.9 shows the allocation of the memories T_{s1}, T_{s2} and T_{s3} for the continuous operation of the 1-D FFT processor. From Fig. 5.9 it is clear that the memories T_{s1}, T_{s2} and T_{s3} be dual port to enable simultaneous read and write operations, but in this case the requirement of data memories access time is relaxed to 210 ns as compared to 52.5 ns for the previous case. The requirement of 52.5 ns access time for the memories T_{s1}, T_{s2} and T_{s3} remain the same. In comparison with the

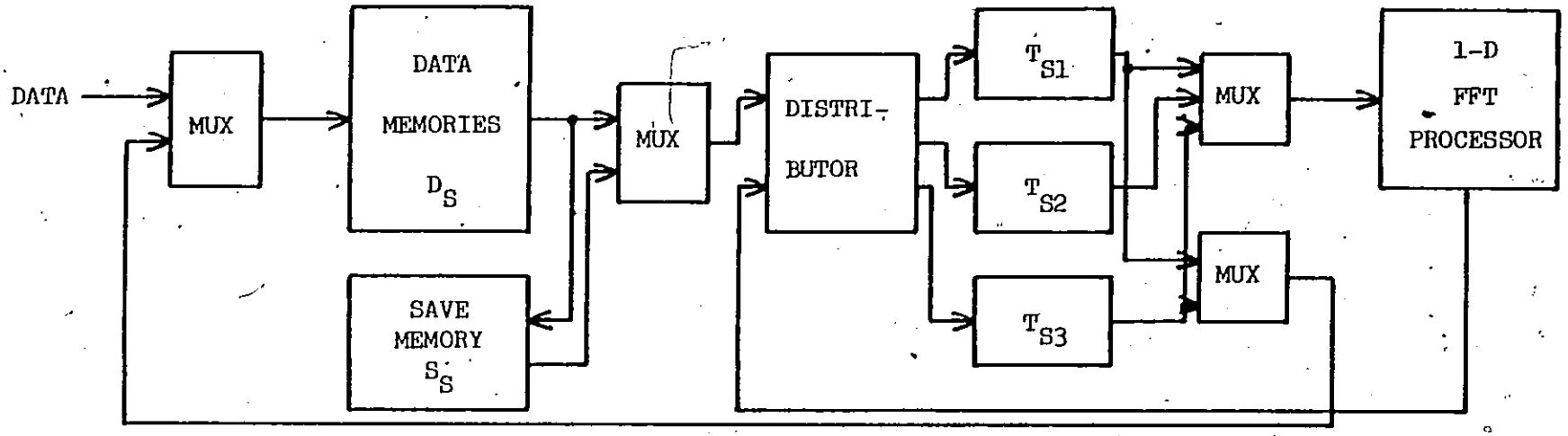


Fig. 5.8 A 2-D CONVOLVER ORGANIZATION USING 1-D FFT PROCESSOR WITH 3 EXTERNAL MEMORIES T_{S1}, T_{S2}, AND T_{S3}.

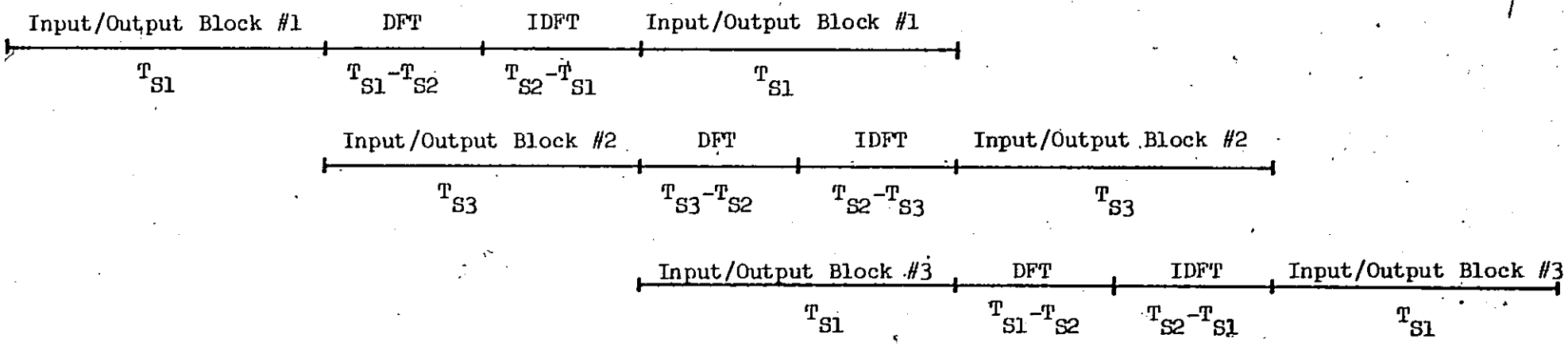


Fig. 5.9 ALLOCATION OF MEMORIES T_{S1}, T_{S2} AND T_{S3} FOR CONTINUOUS OPERATIONS OF A 1-D FFT PROCESSOR.

2-D FFT processor, the access time requirement for the internal buffers of the 2-D processor is 70 ns. In general, the memory access time of the memories T_{s1}, T_{s2}, and T_{s3} depends on the relation $\frac{T}{4} \cdot \log_4 M$. For this time to be equal to T = 70 ns, M must be greater than or equal to 256.

Comparing the two organizations, Fig. 5.3 and Fig. 5.8, of the 2-D convolver we see that for the same cost to performance ratio, the cost of the previously described 2-D radix-2 FFT processor must be equal to the cost of the memories T_{s1}, T_{s2}, T_{s3} and the 1-D FFT processor. From Fig. 3.7 and the discussion in section 4.2.4 we see that, except for the sizes of the sub-blocks and sub-buffers, the complexity of the 1-D radix-4 and 2-D radix-2 FFT processors is the same. Since the size of the memories, T_{s1}, T_{s2}, T_{s3} is the same as that of the internal buffers, BUF1, BUF2, BUF3, we see that the 2-D convolver implementation, using a 2-D radix-2 FFT processor, is more cost effective than the implementation with a 1-D radix-4 FFT processor because the internal buffer storage for the 1-D FFT processor is an extra requirement.

From the above discussion we may conclude that the internal memory architecture of the 2-D radix-2 FFT processor of Fig. 4.6a is equivalent to external memories T_{s1}, T_{s2} and T_{s3} of Fig. 5.8 and the internal buffers of the 1-D FFT processor of Fig. 3.5a. The 2-D FFT processor implementation is even more cost effective for high radices. For example, a 2-D radix-4 Butterfly unit requires 15

multipliers as compared to 23 multipliers required by an equivalent 1-D radix-16 Butterfly unit.

5.4 2-D CONVOLUTION USING NTTs

The 2-D convolver organizations of the last two sections may be implemented using an NTT defined over complex residue rings [25]. In the NTT implementation the scaling hardware is not required, but a set of moduli must be selected to provide enough dynamic range to avoid any overflow during the convolution operation. Also the set of moduli $\{m_i\}$ should be chosen in such a way that the transform length, M, divides each M_i, where M_i depends upon the m_i [25], and $M = r^n$. Because of the exact nature of the computation, NTTs eliminate any round-off error due to integer representation of the twiddle factors and scaling operations, that are inherent in the RNS-based implementation of the DFT. In the computation of the DFT it is known that the error in the output Fourier coefficients can be reduced by increasing the dynamic range. Since, in the NTT implementation, the dynamic range must be large enough to avoid any overflow and it is a function of both the magnitude of the input data and the impulse response, we must scale down the magnitude of the input data and the filter impulse response to get a viable dynamic range. Although scaling of the input data and filter impulse would reduce the dynamic range requirement, it will result in a higher error in the output. Thus, the dynamic range, required for the implementation of both the NTT and the DFT, depends upon the desired error in the output. A thorough investigation of the

dynamic range requirements for DFT and NTT convolver implementation, with similar error properties, is beyond the scope of this thesis.

For an initial comparison, however, we may assume that for the same requirement of the dynamic range for the implementation of both the DFT and the NTT, an acceptable error in the output is obtained. If this is true, we note that the NTT implementation does not require any scaling operations and hence, the residues corresponding to all the moduli must be stored into the buffers BUF1, BUF2 and BUF3 of convolver organization of Fig. 5.3. This doubles the buffer storage requirement for the implementation of the NTT. Also, for a reasonable transform length, M, the moduli required for the implementation of the NTT turn out to be greater than 64, and hence a sub-modular approach using ROM arrays [47] or a combination of adder and ROM structures [48] must be used to implement the Butterfly unit.

In [25], Baraniecka and Jullien have shown that a transform length of 128 and a dynamic range of over 24 bits may be obtained by choosing a set of primes {191, 193, 449} as moduli. A dynamic range of over 24 bits may also be obtained by choosing a set {32, 31, 29, 25, 23} as moduli for the FFT implementation. Assuming that an acceptable error is obtained by choosing a scale factor as the product of two moduli, a preliminary analysis of the hardware requirement shows that a ROM array implementation of both the 2-D radix-2 FFT and NTT Butterfly units require approximately the same number of ROM packages. Since, the package requirement for the buffer storage is greater for the NTT implementation, the FFT implementation of the convolution operation

seems to be more cost effective than the NTT implementation.

The 2-D approach presented in this work offers special advantages in the implementation of NTTs. In situations where the chosen set of moduli $\{m_i\}$ allows a maximum transform length, M, which is a power of 2 but not a power of 4, a 2-D radix-2 implementation offers the cost to performance ratio of a 1-D radix-4 implementation with $M = 2^n$ only. Thus we can conclude that if an NTT implementation is chosen to compute convolution of 2-D sequences, a 2-D NTT processor will be a better choice than a 1-D NTT processor.

5.5 SUMMARY

A novel memory architecture for a special purpose processor to perform 2-D convolutions in real-time has been presented. It has been shown that with the provision of a special memory organization, the 2-D radix-2 FFT processor presented in the last Chapter can be used to perform real-time filtering operations by the overlap-save technique of sectioned convolutions. We also demonstrated that the throughput rate obtainable from the proposed convolver architecture is sufficient to process images of size (256x256) at video rates.

To determine the cost effectiveness of the 2-D convolver using a 2-D radix-2 FFT processor, a more conventional realization of the 2-D convolver using 1-D radix-4 FFT processor, described in Chapter 3, has been presented. It has been demonstrated that, for the same throughput rate requirement, a 2-D convolver implementation using a 2-D radix-2 FFT processor architecture is more cost effective than the implementation

with a 1-D radix-4 FFT processor of equivalent complexity. We showed that the proposed 2-D convolver architecture can also be used to compute convolutions via an NTT defined over complex residue rings. We have made an initial comparison between the hardware requirements for the implementation of a 2-D convolver using the 2-D radix-2 FFT and NTT processors. With the assumption of equal dynamic range requirements for both the FFT and NTT implementation, it is argued that the FFT implementation, is more cost effective than the implementation using a NTT processor. But to establish these results a further investigation of the dynamic range requirements should be done, and is offered as an avenue for future work.

CHAPTER 6

HARDWARE REQUIREMENT AND SIMULATION OF THE FFT PROCESSOR DESIGNS

6.1 INTRODUCTION

In the previous chapters we presented the architectures of RNS-based 1-D and 2-D FFT processors and convolvers. In the discussion of these architectures we found that, for a butterfly operation, the data from the selected buffer is always accessed in sequential order. Also, for real-time processing of the data, the input/output data is stored/accessed in sequential memory locations. Thus the internal buffers of both the 1-D and 2-D FFT processors can be built using serial access memory packages which also leads to simplification of the memory address generation logic and the processor control functions. In many situations, a random-access memory organization may be more suitable because of the commerically available higher density packages with faster access time. The use of higher density random-access instead of serial-access memory will result in smaller hardware requirement and hence may prove to be more cost-effective than the organization with serial-access memories. Thus, depending upon the cost and the performance constraints, one may consider both memory organizations and the trade off involved. In this section we discuss

the use of both the serial- and random-access memories and evaluate the total package requirement for both the 1-D and 2-D FFT processors.

6.2 HARDWARE REQUIREMENTS

Before evaluating the hardware requirements for the RNS-based FFT processors, we must choose a dynamic range, M, which will satisfy the constraints imposed by the application for which the FFT processor is to be used. For a large class of signal processing applications a dynamic range of 28 bits is sufficient. This dynamic range may be obtained by choosing the following moduli set, {32,31,29,27,25,23} where N = 6. Also a set of s = 3 moduli may be chosen as the scaling moduli. The number of commercially available packages needed for the implementation of a 1-D radix-4 1024-point FFT processor/convolver as shown in Fig. 3.7, is given in Table 6.1.

Table 6.1 shows the package requirement for the implementations using both the serial-access (SAM) and the random access memories (RAM). From Table 6.1 we note that, although the use of high density RAM package reduces the number of memory packages it results in an increase in the package requirement for the multiplexers. Thus, the total number of packages required for the implementation using RAM is more than that required by the implementation using serial-access memory. Also, with the use of RAM we are assuming that the memory has two ports so that a simultaneous read and write operation may be performed. The above analysis is valid only for the implementation of a 1024-point FFT processor. If an FFT processor is required to

Table 6.1 PACKAGE REQUIREMENT FOR THE IMPLEMENTATION OF A 1-D RADIX-4 1024-POINT FFT PROCESSOR/CONVOLVER FOR A N = 6 MODULI RNS WITH s = 3 SCALING MODULI

Hardware Unit	Package	Implementation With Serial-access Memory		Implementation With Random-access Memory	
		# of Packages	Size	# of Packages	Size
Butterfly Unit FFT/Convolution	ROM	204/240	1kx8	204/240	1kx8
Scaling/Base Ext. Arrays	ROM	168	1kx8	168	1kx8
Buffers	Serial/Random access memory	288	64x5	192	64x8
Input MUX	Tri-State Octal Buffers	288	8 Bits	336	8 Bits
Output MUX	"	8	8 Bits	128	8 Bits
Distributor	"	96	8 Bits	48	8 Bits
Twiddle factor ROMs FFT/ Convolver	ROM	24	(256/512)x8	24	(256/512)x8
Filter Co-efficients ROMs	ROM	32	128x8	32	128x8
FFT/Convolver	TOTAL	1006/1074		1100/1168	

implement the DFT of a larger array, we must evaluate both the implementations again since high density SAM packages may not be commerically available. This is shown for the implementation of a 2-D FFT processor.

Table 6.2 shows the package requirement for the implementation of a (64x64)-point 2-D radix-2 FFT processor. From Table 6.2 we see that the package requirement for the implementation using SAM is much larger than that of the implementation using RAM due to the non-availability of high-density SAM packages having an access time which is within the constraints imposed by the processor. If the required size of SAM package were commerically available, or if it could have been custom designed, then the total number of packages required through the use of SAM would have been less in comparison with the RAM implementation. Also in the above analysis we have not considered the hardware required for the implementation of the processor control functions, which must be taken into account for both the implementations. Since the SAM implementation results in a simplified control, a further analysis of the cost and performance of the two implementations is required.

In the above analysis, we presented the package requirement for the implementation of the 1-D and 2-D FFT processors/convolvers which is based on the assumption of a dynamic range requirement of 28 bits. Since the dynamic range requirement for a particular application depends upon the desired output error in the Fourier coefficients [40], a lower dynamic range, and hence a lower cost, can be achieved if

Table 6.2 PACKAGE REQUIREMENT FOR THE IMPLEMENTATION OF A 2-D RADIX-2 (64x64)-POINT FFT PROCESSOR/CONVOLVER FOR A N = 6 MODULI RNS WITH s = 3 SCALING MODULI

Hardware Unit	Package	Implementation With Serial-access Memory		Implementation With Random-access Memory	
		# of Packages	Size	# of Packages	Size
Butterfly Unit FFT/Convolver	ROM	204/240	1kx8	204/240	1kx8
Scaling/Base Ext. Arrays	ROM	168	1kx8	168	1kx8
Buffers	Serial/Random access Memory	1152	64x5	192	256x8
Input MUX	Tri-State Octal Buffers	228	8 Bits	336	8 Bits
Output MUX	"	8	8 Bits	128	8 Bits
Distributor	"	192	8 Bits	48	8 Bits
Twiddle factor ROMs	ROM	24	128x8	24	128x8
Filter Co-efficients ROMs	ROM	32	512x8	32	512x8
FFT/Convolver	TOTAL	1976/2044		1100/1168	

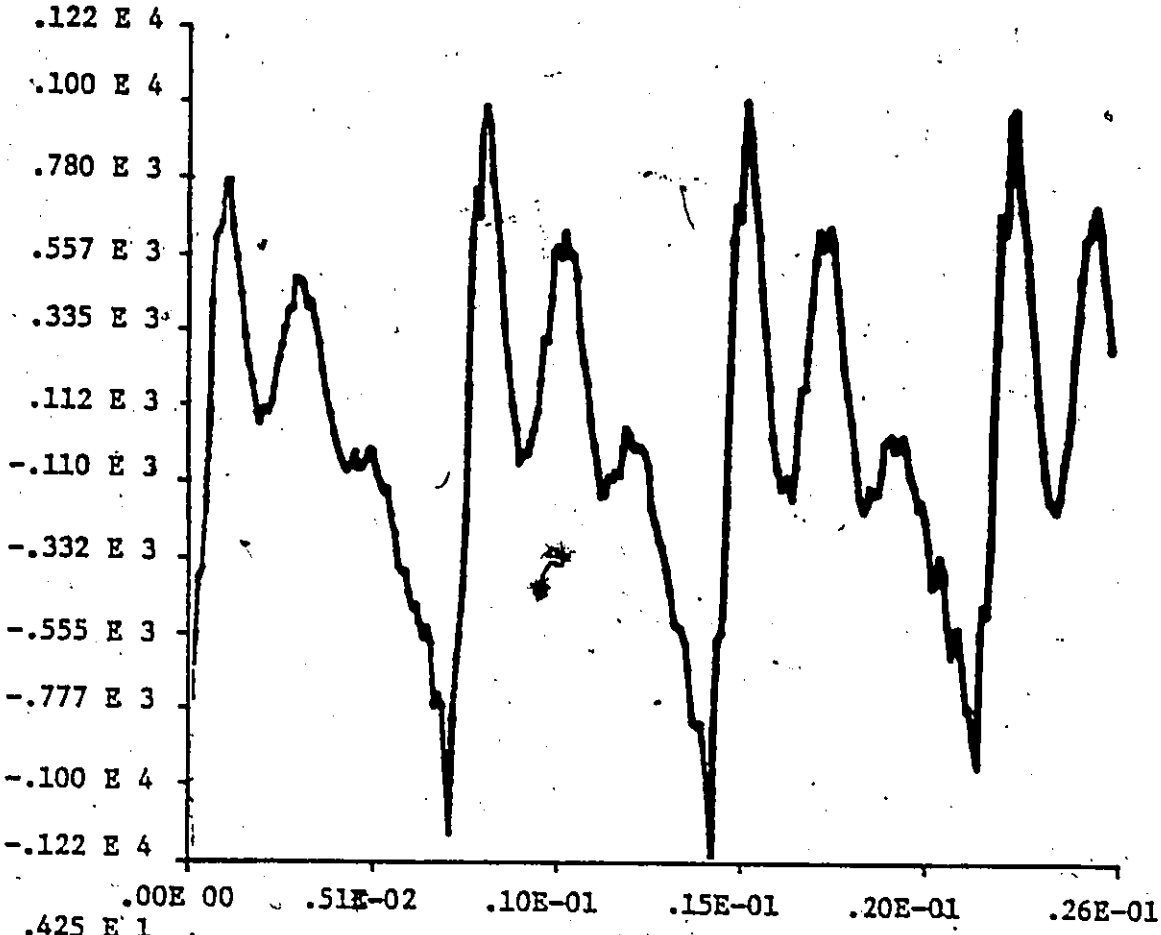
the intended application can tolerate more error in the output.

From the approximate hardware requirement of the 1-D and 2-D FFT processors we can conclude that the RNS-based serial sequential FFT processors are more economically viable than the multiprocessor architecture, organization using a large scale computer and the multi-butterfly FFT processor architectures [20],[21],[46], for real-time processing of 1-D and two-dimensional data. Because of the memory intensive nature of these FFT processors, the cost to performance ratio of these processors is expected to improve with the advancement in memory technology.

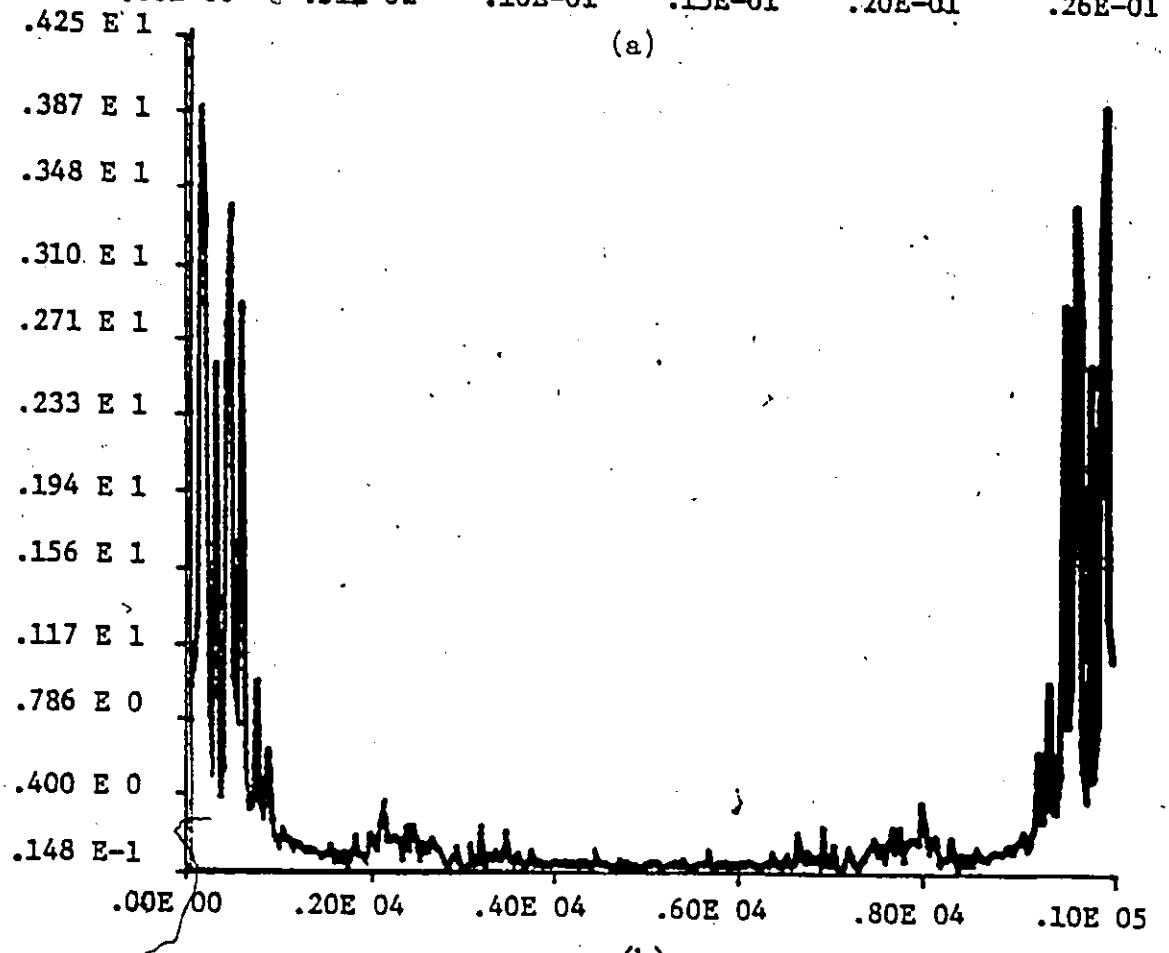
6.3 SIMULATION OF THE FFT PROCESSOR DESIGNS

In order to verify the FFT algorithms and their associated processor architectures proposed in this thesis, the 1-D and 2-D FFT processors/convolvers described in Chapter 3, 4 and 5 have been simulated on a Data General NOVA-840 general purpose computer.

For the verification of the 1-D OI00 FFT algorithm, the memory architecture and the Butterfly unit of the radix-4 FFT processor of Fig. 2.2 has been implemented in software using FORTRAN with single precision floating point arithmetic. The output of the OI00 algorithm was compared with that of the standard decimation-in-time (DIT) FFT algorithm for a number of input sequences and the difference between the two outputs was found to be negligible. For example, Fig. 6.1(a) shows a section of the input speech signal to the two implementations. Figs. 6.1(b), 6.1(c) and 6.1(d) show the outputs



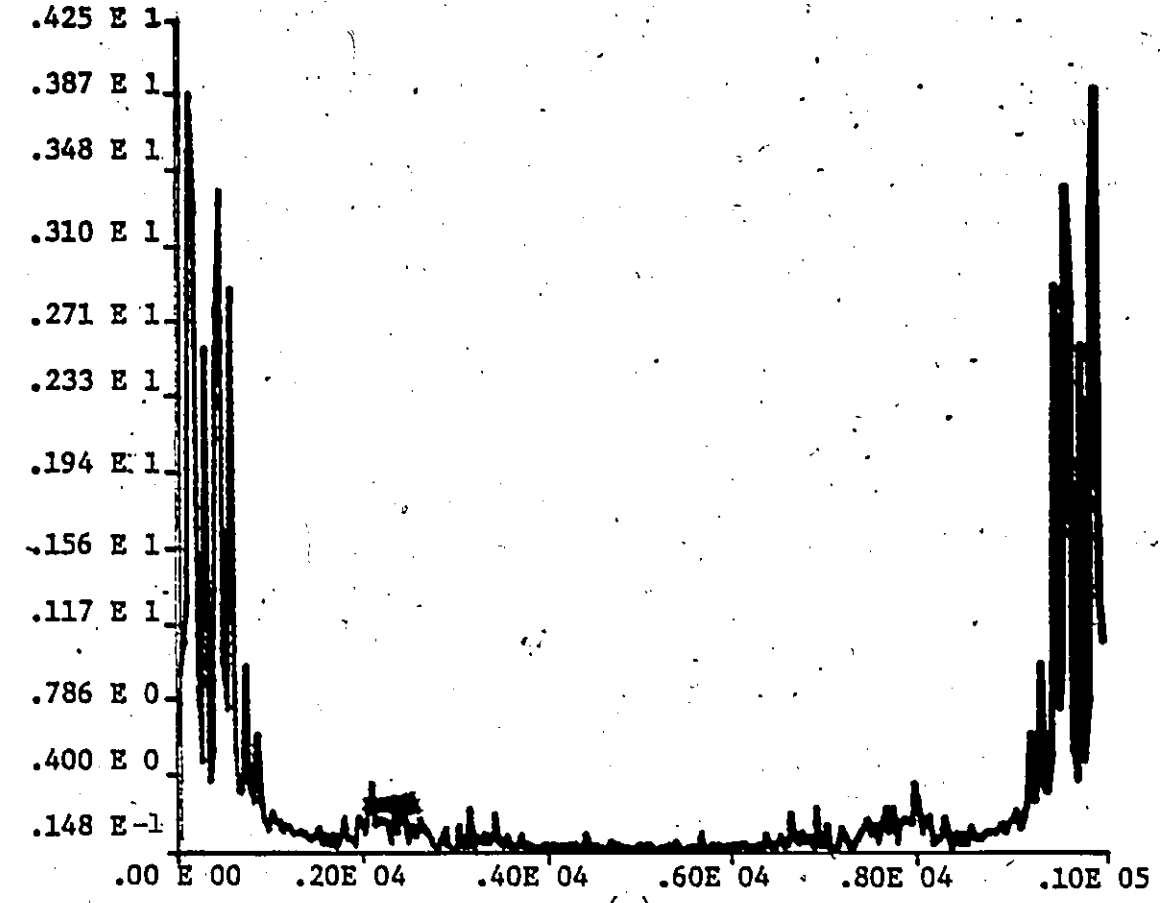
(a)



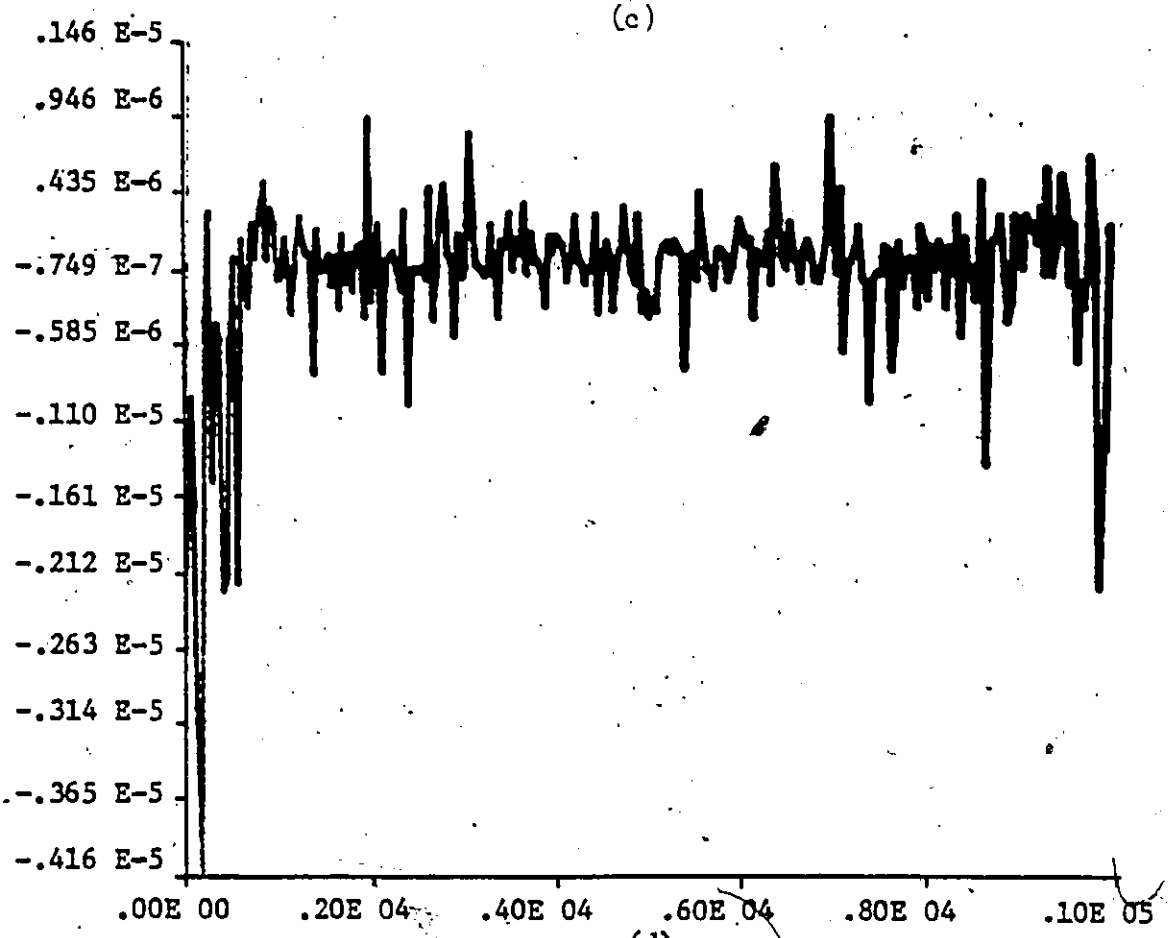
(b)

Fig. 6.1 (a) Input speech signal

(b) Output of the OIOO algorithm



(c)



(d)

Fig. 6.1 (c) Output of the standard DIT FFT algorithm

(d) Error between the outputs generated by the OIOO and DIT FFT algorithms

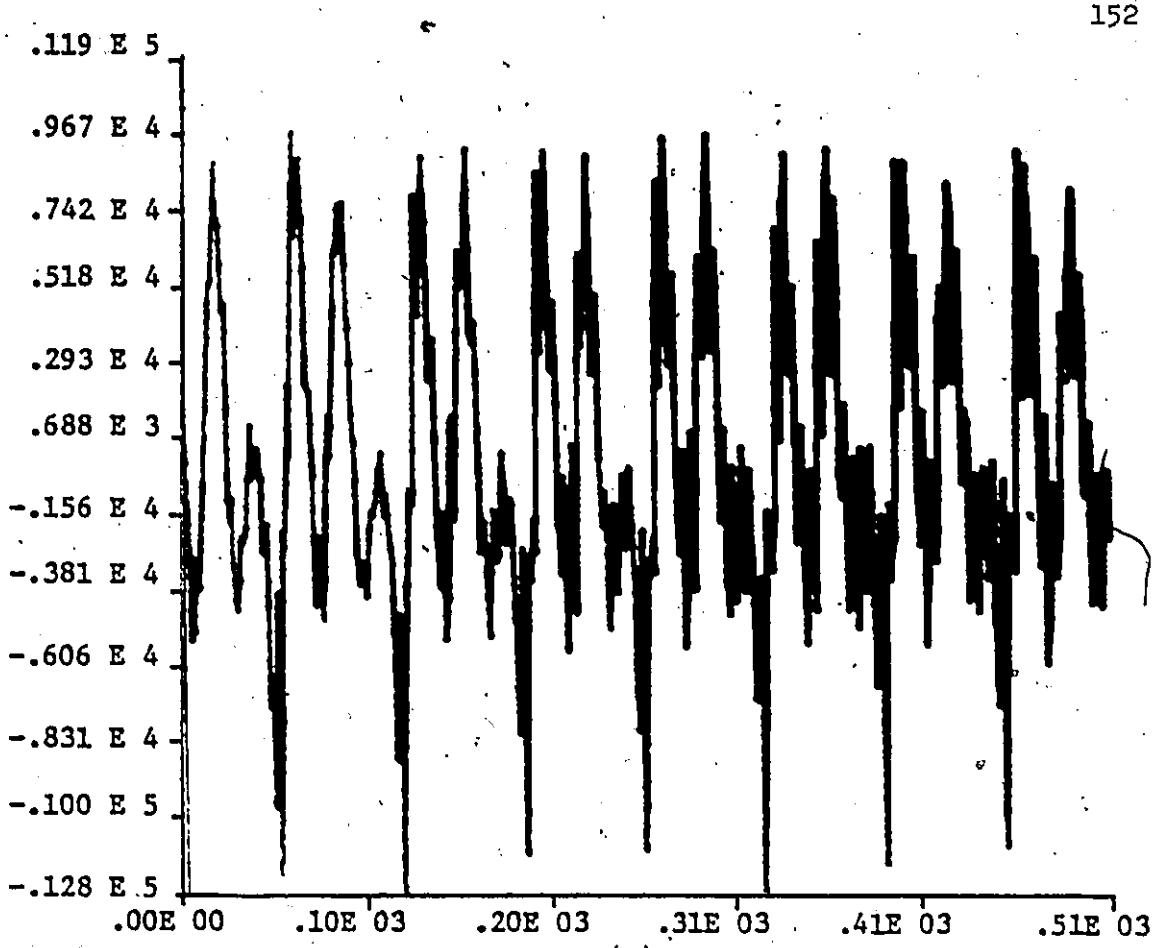
generated by the OIOO and DIT FFT algorithms and the error between them, respectively. From Fig. 6.1(d) we note that the maximum error between the two outputs is less than $\pm 10^{-5}$.

For the simulation of the RNS-based 1-D FFT processor of Fig. 3.7, a simulation program for a 1024-point radix-4 FFT processor has been written using FORTRAN and assembly language. The pipelined processor architecture of Fig. 3.7 was simulated using a 6-moduli RNS with 3 scaling moduli and the values of the integer conversion factor, scale factor and the dynamic range were found from the procedure given in [40]. The look-up tables specified by the butterfly architecture of Fig. 3.2 and scaling algorithm of Fig. 3.3 were generated and stored in a disc file. For the implementations of the processor's pipeline, the input and output of each stage of the pipeline was buffered. The buffers BUF1, BUF2 and BUF3 of the FFT-processor were implemented as shift registers of size 1024 words where each buffer consisted of 16 first-in first-out (FIFO) serial-access memories. The processor control functions required to control the Input and Output MUX's, Distributor, data flow in the FIFO's and the pipeline operations were implemented using integer, logical and bit variables. The input and output of the processor was assumed to reside in the disc files. The simulation program was organized as a big loop where each iteration of the loop performed all the data movement and control functions specified by the FFT processor architecture during one clock period.

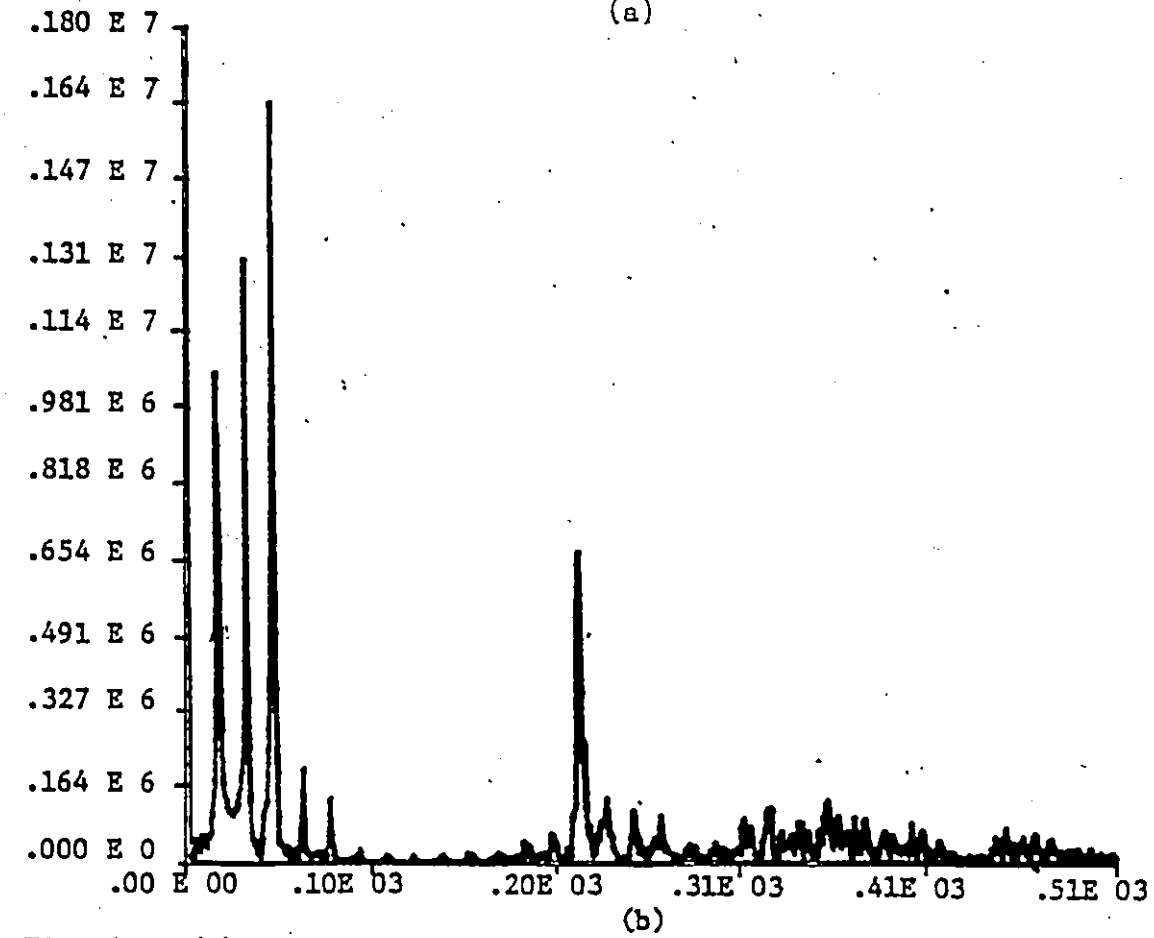
Various types of sinusoidal, random and speech signals were

used as test inputs to the FFT processor simulator. For verification of the result, the simulator output was compared with the output of the 1-D OIOO FFT algorithm. For all the test inputs, the comparison between the two outputs showed that the output of the simulator was within the specified error limits. For example, Fig. 6.2(a) shows a section of the input speech signal to the simulator and OIOO FFT algorithm, and Figs. 6.2(b), 6.2(c), and 6.2(d) show the corresponding output of the simulator, OIOO FFT algorithm and the error between the two outputs, respectively. The RMS error between the two outputs was found to be 0.832% which is less than the specified 1% limit on the output error.

For the verification of the 2-D FFT algorithm of Chapter 4 and the 2-D convolver architecture of Chapter 5, a 2-D radix-2 (32x32)-point FFT algorithm was implemented in FORTRAN. Since the architectures of the 1-D and 2-D FFT processors are almost identical, no effort was made to simulate the real-time architecture of the 2-D FFT processor of Fig. 4.6 using the chosen RNS. Instead, the two-buffer non real-time organization of Fig. 4.3 was implemented using floating-point arithmetic. The memory organization of the convolver of Fig. 5.2 was implemented by using separate areas of the computer memory as Data and Save memory. To verify the convolution algorithm using the overlap-save method of sectioned convolution, as described in Chapter 5, a (17x17) impulse-response of a Laplacian filter was chosen as the test filter. Images of size (128x128) were convolved with the impulse-response of the test filter using the (32x32)-point 2-D radix-2 FFT algorithm and the Data and Save memory organization of Fig. 5.2. The same images were also filtered by using the conventional technique of computing the 1-D DFT along the rows and

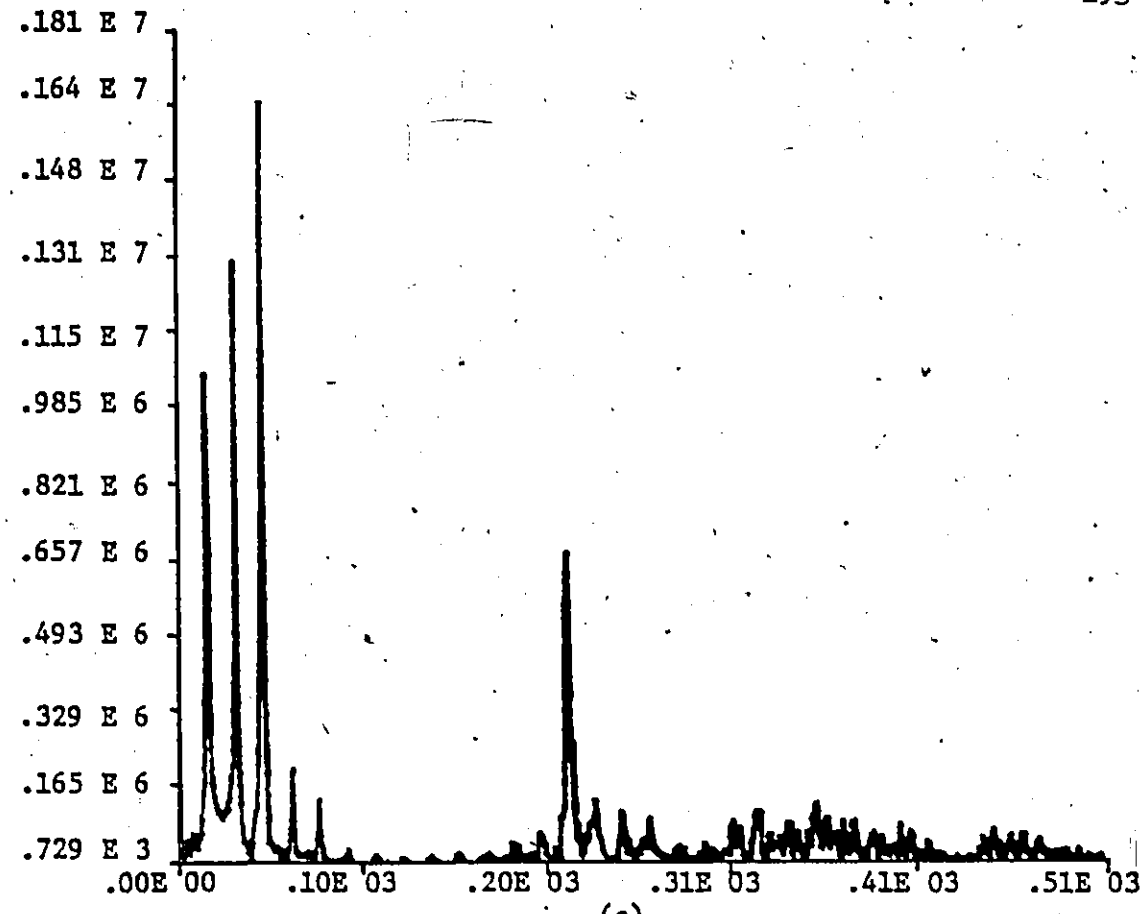


(a)

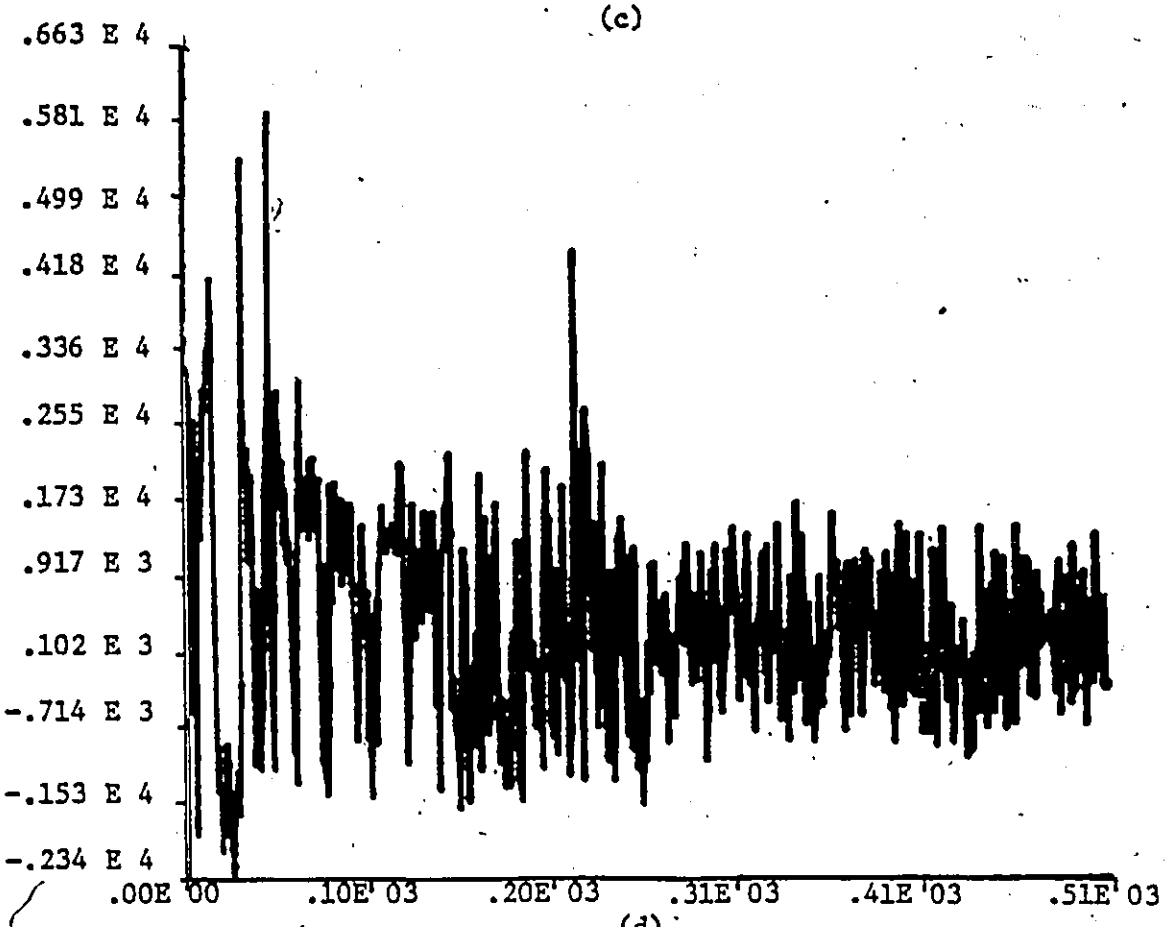


(b)

Fig. 6.2 (a) Input speech signal
(b) Output of the FFT Processor's Simulator



(c)



(d)

Fig. 6.2 (c) Output of the 0100 FFT Algorithm.
(d) Error between the outputs generated by the Simulator and the 0100 FFT Algorithm

then along the columns. The filtered images computed by using the above two techniques were compared and no significant difference between the two outputs was found. Although the 2-D FFT processor and convolver architectures were implemented using floating-point arithmetic, simulation of the RNS-based 1-D FFT processor indicates that the RNS principles can be directly extended to two and multi-dimensions. Fig. 6.3(a) shows an example of the image used to test the 2-D radix-2 convolver and Figs. 6.3(b) and 6.3(c) show the outputs obtained from the 2-D radix-2 convolver and the implementation using the conventional technique, respectively. The output images of Figs. 6.3(b) and 6.3(c) contain 256 intensity levels and the root-mean-square error between the two images was found to be less than 0.006%.

Due to the large length of the simulation programs, a listing of the programs written for the simulation of the 2-D radix-2 FFT processor/convolver only is given in Appendix A. A listing of other programs may be obtained from the Dept. of Electrical Engineering, University of Windsor [50].

6.4 SUMMARY

In this section, the hardware requirements for the implementation of the 1-D and 2-D FFT processor/convolver, proposed in the previous chapters, have been presented. From these hardware requirements it is noted that, for an equivalent level of performance, the complexity of the FFT processors/convolvers proposed in this work is much less than the complexity of the multi-processor architectures and the organizations

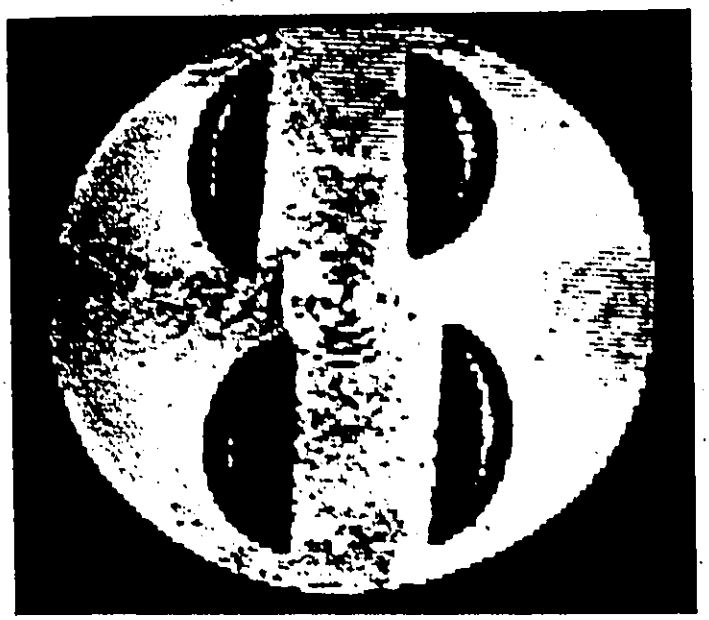


Fig. 6.3a – Input Image

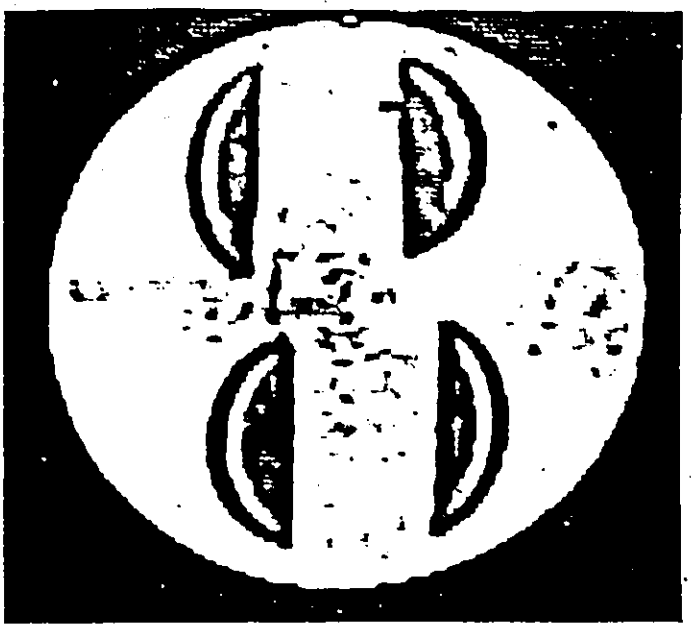


Fig. 6.3b– Filtered image using radix-2 2-D FFT Processor/Convolver

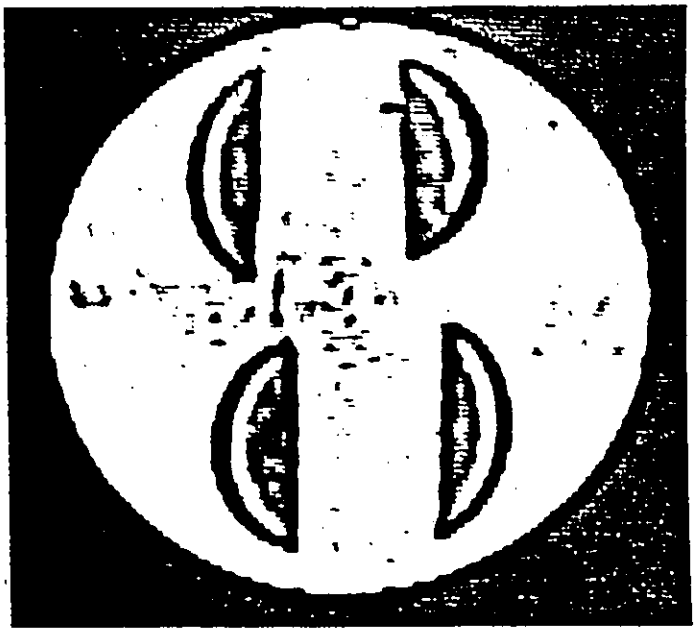


Fig. 6.3c– Filtered image using the conventional technique of computing the 1-D DFTs along the rows and then along the columns.

using general purpose computers proposed in the literature.

The FFT algorithms and their associated processor architectures have been verified via extensive simulation on a general purpose computer. For a number of input sequences, the result obtained via simulation have been presented. It is shown that the results obtained via simulation agree with the results obtained via the implementation of the standard FFT and convolution algorithms, thus proving the validity of the algorithms and processor architectures presented in this thesis.

CHAPTER 7

CONCLUSIONS

The principal objective of the research work described in this thesis was to develop algorithms and special purpose processor architectures for real-time processing of one-, two- and multi-dimensional signals.

To achieve these goals, we presented a unified approach to the development of fast algorithms for computing a class of multi-dimensional unitary transforms having separable kernels. The techniques for the development of fast algorithms are based on the factorization of the multi-dimensional transformation matrix into Kronecker products of one-dimensional transformation matrices. For a number of unitary transforms, including Fourier, Walsh-Hadamard and generalized Walsh transforms, the one-dimensional transformation matrix can be expressed as a product of sparce matrices. A new result, obtained in this work, shows that the factorization of a multi-dimensional transformation matrix can be expressed in terms of the factors of the one-dimensional transformation matrices. The factorization thus obtained results in fast unitary transform algorithms which may be implemented in n-stages where each stage operates on the data in each dimension of the multi-dimensional array. When the number of elements in each dimension of the input array is restricted to M, where $M = r^n$, the arithmetic

operations on the data in each dimension of the input array may be combined to form multi-dimensional butterfly operations for a class of unitary transforms.

Based on the factorization technique developed in this work, the development of new two- and multi-dimensional Fourier, Walsh-Hadamard and generalized Walsh transform algorithms has been presented. The nature of these algorithms for one-, two- and multi-dimensional transforms results in highly similar processor architectures. These processor structures have a high degree of parallelism with simplified control for the generation of data and twiddle factor addresses. It is also shown that the u-dimensional Fourier and generalized Walsh transforms can be implemented with a u-dimensional radix-r butterfly operation, which requires considerably less complex multiplications than the conventional implementation using one-dimensional radix-r butterfly operation. In terms of hardware required for the implementation of special purpose processors this represents a minimum saving of 25% in multiplier hardware for the 2-dimensional radix-2 implementation. A saving of more than 25% is obtained for higher dimensions and higher radix implementations.

The novel implementation of the Butterfly unit of one- and two-dimensional FFT processors via the Residue-number system resulted in a highly structured pipeline organization with only three kinds of IC packages viz ROM, Serial/Random-access memory and tri-state buffers. The pipeline throughput rate is dependent only on the ROM access time and is constant over all the pipeline stages. Using commercially available

IC packages it was shown that the proposed FFT processor architectures are capable of processing one- and two-dimensional sequences sampled at a rate of more than 11 and 7 million samples/second, respectively. Although the proposed processor architectures call for partitioning of the internal memory buffers into several sections, they can be implemented with the use of currently available serial memory packages. From the approximate number of packages required for the implementation of these processors, we can conclude that the FFT processors proposed here are more economically viable than the multi-processor and multi-butterfly architectures proposed in the literature, for the same processing speed requirements.

As applications of the one- and two-dimensional FFT processors, we also showed that these processor organizations can be used to perform convolution of wide-band sequences, in real-time, by using the overlap-save method of sectioned convolutions. In the implementation of one-dimensional convolution it has been shown that by the provision of extra buffers at the input and output of a one-dimensional FFT processor, the architecture can be used to filter a real-valued input signal with little modification. In the case of two-dimensional convolution we presented a new memory organization to support real-time filtering of two-dimensional data. We also showed that a two-dimensional FFT processor requires less hardware than the conventional implementation of the DFT of a two-dimensional sequence (which computes the transform via one-dimensional DFT's of the rows

and then that of the columns). For the computation of the DFT of smaller sequences, we proved that the DFT of r $\frac{M}{r}$ - point one-dimensional sequences can be computed on an M-point one-dimensional FFT processor, and we also suggested some modifications of the M-point FFT processor architecture to support the computation of r $\frac{M}{r}$ - point and r^2 . $\frac{M}{r^2}$ point sequences.

Although the FFT processor architectures developed here are based on the one-dimensional ordered-input ordered-output algorithm, other factorizations of the one-dimensional transformation matrix, reported in literature, may also be used to obtain different realizations of the multi-dimensional FFT processors. The FFT algorithms and processor architectures developed here were shown to be useful for the computation of Number-theoretic transforms defined over complex residue rings.

REFERENCES

[1] I.J. Good, "The interaction algorithm and practical Fourier analysis", Journal of the Royal Statistical Society (London), Vol. B20, pp. 361, 1958.

[2] J.W. Cooley and J.W. Tukey, "An algorithm for the machine calculation of complex Fourier series", Mathematics of Computation, Vol. 19, pp. 297-301, April 1965.

[3] J.E. Whetchel and D.F. Guinn, "The fast Fourier-Hadamard transform and its use in signal representation and classification", Eascon '68 Record, pp. 561-570.

[4] B.J. Fino and V.R. Algazi, "Unified matrix treatment of the fast Walsh-Hadamard transform", IEEE Trans. on Computers, Vol. C-25, No. 11, pp. 1142-1145, Nov. 1976.

[5] N. Ahmed and T. Natarajan, "Cooley-Tukey-type algorithm for Haar-transform", Electronics Letters, Vol. 9, No. 12, pp. 276-278, 14th June 1973.

[6] H.C. Andrews and K.L. Caspari, "A generalized technique for spectral analysis", IEEE Trans. on Computers, Vol. C-19, No. 1, pp. 16-25, Jan. 1970.

[7] N. Ahmed and K.R. Rao, Orthogonal Transforms for Digital Signal Processing, Springer-Verlag, New York, 1975.

[8] V. Vlasenko and K.R. Rao, "Unified matrix treatment of discrete transforms", IEEE Trans. on Computers, Vol. C-28, No. 12, pp. 934-938, Dec. 1979.

[9] D.P. Colba and T.W. Parks, "A prime factor FFT algorithm using high speed convolution", IEEE Trans. on Acoustics, Speech and Signal Processing, Vol. ASSP-25, pp. 281-294, Aug. 1977.

[10] H.F. Silverman, "An introduction to programming the Winograd Fourier transform algorithm", IEEE Trans. on Acoustics, Speech and Signal Processing, Vol. ASSP-25, pp. 152-165, April 1977.

[11] J.E. Whelchel, Jr. and D.F. Guinn, "FFT organizations for high speed digital filtering", IEEE Trans. on Audio and Electroacoustics, Vol. AU-18, No. 2, pp. 159-168, June 1970.

[12] M.J. Corinthios, "A time-series analyzer", MRI Symposia Ser. New York: Polytechnic Press, Vol. 12, pp. 47-61, 1969.

[13] M.J. Corinthios, "A fast Fourier transform for high speed signal processing", IEEE Trans. on Computers, Vol. C-20, No. 8, pp. 843-846, Aug. 1971.

[14] M.J. Corinthios, "The design of a class of fast Fourier transform computers, IEEE Trans. on Computers, Vol. C-20, No. 6, pp. 617-623, June 1971.

[15] M.J. Corinthios et al., "A parallel radix-4 fast Fourier transform computer", IEEE Trans. on Computers, Vol. C-24, pp. 80-92, Jan. 1975.

[16] M.C. Pease, "An adaptation of fast Fourier transform for parallel processing", Journal of Association for Computing Machinary, Vol. 15, pp. 252-264, April 1968.

[17] M.C. Pease, "Organization of large scale Fourier processors", Journal of the Association for Computing Machinery, Vol. 16, No. 3, pp. 474-482, July 1969.

[18] H.L. Groginsky and G.A. Works, "A pipeline fast Fourier transform", IEEE Trans. on Computers, Vol. C-19, pp. 1015-1019, Nov. 1970.

[19] B. Liu and A. Peled, "A new hardware realization of high speed fast Fourier transform", IEEE Trans. on Acoustics, Speech and Signal Processing, Vol. 23, pp. 543-547, Dec. 1975.

[20] B. Gold and T. Bially, "Parallelism in fast Fourier transform hardware", IEEE Trans. on Audio and Electroacoustics, Vol. AU-21, No. 1, pp. 5-16, Feb. 1973.

[21] C.S. Joshi et al., "A video rate two-dimensional FFT processor", IEEE International Conference on Acoustics, Speech and Signal Processing, Colorado, pp. 774-777, April 1980.

[22] Y.A. Geadah and M.J. Corinthios, "Natural, Dyadic and Sequency order algorithms and processors for the Walsh-Hadamard transforms", IEEE Trans. on Computers, Vol. C-26, No. 5, pp. 435-442, May 1977.

[23] L.R. Rabiner and B. Gold, Theory and Applications of Digital Signal Processing, Prentice Hall, Inc., 1975.

[24] L.W. Martinson and R.J. Smith, "Digital matched filtering with pipelined floating-point fast Fourier transforms", IEEE Trans. on Acoustics, Speech and Signal Processing, Vol. ASSP-23, No. 2, pp. 222-234, April 1975.

[25] A. Baraniecka and G.A. Jullien, "Residue number system implementation of number theoretic transforms in complex residue rings", IEEE Trans. on Acoustics, Speech and Signal Processing, Vol. ASSP-28, No. 3, pp. 285-291, June 1980.

[26] G.D. Bergland, "A fast Fourier transform algorithm using base-8 iterations", Mathematics of Computation, Vol. 22, pp. 275-278, April 1968.

[27] R.R. Shively, "A digital processor to generate spectra in real-time", IEEE Trans. on Computers, Vol. C-17, pp. 485-491, May 1968.

[28] S.D. Pezaris, "A 40 ns 17x17 array multiplier", IEEE Trans. on Computers, Vol. C-20, pp. 442-447, April 1971.

[29] T. Hallin and M. Flynn, "Pipelining of arithmetric functions", IEEE Trans. on Computers, Vol. C-21, pp. 880-886, 1972.

[30] G.D. Bergland, "Fast Fourier transform hardware implementation - an overview", IEEE Trans. on Audio and Electroacoustics, Vol. AU-17, pp. 104-108, June 1969.

[31] W.K. Pratt, Digital Image Processing, John Wiley and Sons, 1978.

[32] H.C. Andrews, Computer Techniques in Image Processing, Academic Press, New York, 1970.

[33] I.J. Good, "The relationship between two fast Fourier transforms", IEEE Trans. on Computers, Vol. C-20, No. 3, pp. 310-317, March 1971.

[34] B. Arambepola and P.J.W. Rayner, "Multi-dimensional fast Fourier transform algorithms", Electronics Letters, Vol. 15, No. 3, pp. 382-383, 21st June 1979.

[35] R.C. Singleton, "A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage", IEEE Trans. on Audio and Electroacoustics, Vol. AU-15, pp. 91-97, June 1967.

[36] B.R. Hunt, "Data structures and computational organization in digital image enchancement", Proceedings of the IEEE, Vol. 60, No. 7, pp. 884-887, July 1972.

[37] G.L. Anderson, "A stepwise approach to computing the multi-dimensional fast Fourier transform of large arrays", IEEE Trans. on Acoustic, Speech and Signal Processing, Vol. ASSP-28, No. 3, pp. 280-284, June 1980.

[38] R.E. Twogood, "2-D digital signal processing with an array processor", Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 426, April 1980.

[39] G.A. Jullien, "Residue number scaling and other operations using ROM arrays", IEEE Trans. on Computers, Vol. C-27, No. 4, pp. 325-336, April 1978.

[40] B.D. Tseng, G.A. Jullien and W.C. Miller, "Implementation of FFT structures using the residue number system", IEEE Trans. on Computers, Vol. C-28, No. 11, pp. 831-845, Nov. 1979.

[41] C.H. Huang and F.J. Taylor, "High speed DFT's using Residue Numbers", IEEE International Conference on Acoustics, Speech and Signal Processing, Colorado, pp. 238-242, April 1980.

[42] W.K. Jenkins and B.J. Leon, "The use of residue number systems in the design of finite impulse response digital filters", IEEE Trans. on Circuits and Systems, Vol. CAS-24, pp. 191-201, April 1977.

[43] M.A. Soderstrand, "A high speed low-cost recursive digital filter using residue number arithmetic", Proceedings of IEEE, Vol. 65, pp. 1065-1067, July 1977.

[44] N.S. Szabo and R.I. Tanaka, Residue Arithmetic, and its Applications to Computer Technology, McGraw Hill Inc., New York, 1967.

[45] T.G.Stockham, "High speed convolution and correlation", 1966 Spring Joint Conference, AFIPS Conference Proceedings, Vol. 28, pp. 229-233, 1966.

[46] B.K. Gilbert et al., "A real-time hardware system for digital processing of wide-band video images", IEEE Trans. on Computers, Vol. C-25, No. 11, pp. 1089-1099, November 1976.

[47] G.A. Jullien, "Implementation of Multiplication modulo a prime number, with applications to number theoretic transforms", IEEE Trans. on Computers, Vol. C-29, No. 10, pp. 899-905, Oct. 1980.

[48] Fred J. Taylor, "Large modulo multipliers", IEEE International Conference on Acoustics, Speech and Signal Processing, Colorado, pp. 792-795, April 1980.

[49] Bipolar LSI Data Book, Monolithic Memories, Sunnyvale, CA., 1980.

[50] A FFT Processor's Simulator, Dept. of Electrical Engineering, University of Windsor, Windsor, Canada.

APPENDIX A

Computer Programs for the Verification of the 2-D radix -2 OIOO FFT Algorithm and the 2-D Convolver Architecture

```
C
C
C          ********************** FFTERR **********************
C
C          THIS PROGRAM GENERATES THE DFT OF A (32*32) SECTION OF AN IMAGE
C          USING THE 2-D RADIX-2 OIOO AND 1-D RADIX-2 DIT FFT ALGORITHMS.
C          IT ALSO COMPUTES THE ERROR BETWEEN THE MAGNITUDE OF THE FOURIER
C          COEFFICIENTS GENERATED BY THE TWO ALGORITHMS.  THE SPECIFIED
C          (32*32) IMAGE SECTION, THE FOURIER COEFFICIENTS GENERATED BY THE
C          TWO ALGORITHMS AND THE ERROR BETWEEN THE TWO OUTPUTS ARE STORED
C          IN THE SPECIFIED DISC FILES FOR DISPLAY.
C
C          SUBROUTINES REQUIRED - FFT2D64, R2DIT, SHFT
C
C          XIO   - BUFF1 AND BUFF2 OF THE FFT PROCESSOR
C          DATA  - TEMP. STORAGE AREA
C          R2RDT - REAL PART OF THE INPUT TO THE DIT FFT ALGORITHM
C          R2IDT - IMAG. PART OF THE INPUT TO THE DIT FFT ALGORITHM
C          TFTAB - ARRAY STORING THE TWIDDLE FACTORS
C          INB   - INDEX OF THE INPUT BUFFER
C          IOB   - INDEX OF THE OUTPUT BUFFER
C          NSTG  - # OF STAGES IN THE FFT ALGORITHM
C
      COMPLEX XIO(32,32,2),DATA(32,32)
      REAL R2RDT(32),R2IDT(32)
      DOUBLE PRECISION COMPLEX TFTAB(32,2),TFEX
      DOUBLE PRECISION PI,TFX
      INTEGER IMAG(128),INFIL(5),IMGFIL(5),OUTFIL1(5),OUTFIL2(5),ERRFIL(
      COMMON/FFT1/ INB,IOB,NSTG
      DATA INB/1/,IOB/2/,NSTG/5/
      TYPE"INPUT IMAGE  FILE NAME : "
      READ(11,1)(INFIL(I),I=1,5)
    1 FORMAT(5S2)
      TYPE"FILE NAME TO STORE (32*32) IMAGE SECTION : "
      READ(11,1)(IMGFIL(I),I=1,5)
      TYPE"FILE NAME FOR OIOO OUPUT :"
      READ(11,1)(OUTFIL1(I),I=1,5)
      TYPE"FILE NAME FOR DIT OUTPUT :"
      READ(11,1)(OUTFIL2(I),I=1,5)
      TYPE"FILE NAME FOR ERROR OUTPUT :"
      READ(11,1)(ERRFIL(I),I=1,5)
      ACCEPT"IMAGE BLOCK # ? ROW BLK, COL. BLK : ",IROW,ICOL
      IROW=32*(IROW-1)
      ICOL=32*(ICOL-1)+1
      OPEN 0,INFIL
      OPEN 1,IMGFIL,LEN=64,REC=32
C
C          READ THE SPECIFIED (32*32) SECTION FROM THE FILE 'INFIL'.
C
```

```
      IF(IROW.EQ.0) GO TO 12
      DO 10 I=1,IROW
      READ(0)(IMAG(J),J=1,128)
10    CONTINUE
12    DO 20 I=1,32
      READ(0)(IMAG(J),J=1,128)
      J1=0
      DO 15 J=ICOL,(ICOL+31)
      J1=J1+1
      XIO(I,J1,1)=CMPLX(FLOAT(IMAG(J)),0.)
      DATA(I,J1)=XIO(I,J1,1)
15    CONTINUE
C
C     WRITE THE SELECTED IMAGE SECTION INTO THE FILE 'IMGFIL'.
C
      WRITE(1)(IMAG(J),J=ICOL,(ICOL+31))
20    CONTINUE
      CLOSE 0
      CLOSE 1
C
C     COMPUTE THE REQUIRED TWIDDLE FACTORS AND SAVE THESE IN TFTAB.
C     TFTAB(J,1) AND TFTAB(J,2) STORE THE TWIDDLE FACTORS FOR THE
C     FORWARD AND INVERSE FFT, RESPECTIVELY.
C
      PI=-1.D0*DATAN(1.D0)/4.D0
      INV=1
      DO 40 I=1,2
      DO 30 J=1,32
      TFX=PI*INV*(J-1)
      TFEX=DCMPLX(0.D0,TFX)
      TFTAB(J,I)=DCEXP(TFEX)
30    CONTINUE
      INV=-1
40    CONTINUE
      INV=0
C
C     COMPUTE THE DFT OF THE IMAGE SECTION IN XIO USING THE 2-D RADIX-2
C     OIOO FFT ALGORITHM.
C
      CALL FFT2D64(XIO,TFTAB,INV,32)
      OPEN 0,OUTFIL1,LEN=128,REC=32
      OPEN 1,OUTFIL2,LEN=128,REC=32
      OPEN 2,ERRFIL,LEN=128,REC=32
C
C     SAVE THE MAGNITUDE OF THE FOURIER COEFFICIENTS GENERATED BY THE
C     OIOO FFT ALGORITHM IN THE FILE 'OUTFIL1'.
C
      DO 100 I=1,32
      WRITE(0)(CABS(XIO(I,J,IOB)),J=1,32)
100   CONTINUE
      DO 150 J=1,32
      DO 120 K=1,32
      R2RDT(K)=REAL(DATA(J,K))
      R2IDT(K)=AIMAG(DATA(J,K))
120   CONTINUE
```

```
C
C       COMPUTE THE 1-D 32-POINT DFT OF THE JTH ROW OF THE IMAGE SECTION
C       AND SAVE THE ROW TRANSFORMS IN THE ARRAY 'DATA'.
C
        CALL R2DIT(R2RDT,R2IDT,32,5)
        DO 130 K=1,32
        DATA(J,K)=CMPLX(R2RDT(K),R2IDT(K))
130     CONTINUE
150     CONTINUE
        DO 200 J=1,32
        DO 160 K=1,32
        R2RDT(K)=REAL(DATA(K,J))
        R2IDT(K)=AIMAG(DATA(K,J))
160     CONTINUE
C
C       COMPUTE THE 1-D 32-POINT DFT OF THE JTH COLUMN AND SAVE THE OUTPUT
C       OF THE COLUMN TRANSFORMS IN THE ARRAY 'DATA'.
C
        CALL R2DIT(R2RDT,R2IDT,32,5)
        DO 170 K=1,32
        DATA(K,J)=CMPLX(R2RDT(K),R2IDT(K))
170     CONTINUE
200     CONTINUE
C
C       COMPUTE THE MAGNITUDE OF THE FOURIER COEFFICIENTS GENERATED BY
C       THE 1-D DIT FFT ALGORITHM AND ALSO THE ERROR BETWEEN THE MAGNI-
C       TUDE OF THE FOURIER COEFFICIENTS GENERATED BY THE 1-D DIT AND
C       2-D OIOO ALGORITHMS.
C
        DO 250 I=1,32
        DO 240 J=1,32
        R2RDT(J)=CABS(DATA(I,J))
        R2IDT(J)=CABS(XIO(I,J,IOB))-R2RDT(J)
240     CONTINUE
C
C       SAVE THE MAGNITUDE OF THE FOURIER COEFFICIENTS AND THE ERROR
C       IN THE FILES 'OUTFIL2' AND 'ERRFIL'.
C
        WRITE(1)(R2RDT(J),J=1,32)
        WRITE(2)(R2IDT(J),J=1,32)
        WRITE FREE(12)(R2IDT(J),J=1,32)
250     CONTINUE
        CLOSE 0
        CLOSE 1
        CLOSE 2
        STOP
        END
```

```
      SUBROUTINE FFT2D64(XIO,TFTAB,INV,NPTS)
C
C
C     THIS SUBROUTINE COMPUTES THE DFT/IDFT OF A 2-D ARRAY XIO OF
C     SIZE (NPTS*NPTS) USING THE MEMORY ARCHITECTURE OF THE 2-D
C     RADIX-2 DIOO FFT ALGORITHM.
C     TFTAB - ARRAY STORING THE TWIDDLE FACTORS FOR THE DIRECT AND
C             INVERSE FFT
C     INV   = 0 FOR DIRECT TRANSFORM, =1 FOR INVERSE TRANSFORM
C
C
      COMPLEX XIO(32,32,2)
      DOUBLE PRECISION COMPLEX TFTAB(32,2)
      COMPLEX XBUTT(4)
      INTEGER IOFS(4,2),NBOFS(4,2),IPNT(4)
      COMMON/FFT1/ INB,IOB,NSTG
      PTSSQ=NPTS*NPTS
      NB2=NPTS/2
      NB4=NB2/2
      NBOFS(1,1)=0
      NBOFS(1,2)=0
      NBOFS(2,1)=0
      NBOFS(2,2)=NB2
      NBOFS(3,1)=NB2
      NBOFS(3,2)=0
      NBOFS(4,1)=NB2
      NBOFS(4,2)=NB2
      INVIN=INV+1
C
C     PERFORM NSTG STAGES OF THE FFT ALGORITHM.
C
      DO 500 ISTG=1,NSTG
      TYPE"INV=",INV,"  STAGE=",ISTG
C
C     INITIALISE THE STARTING BLOCK (IBLK),# OF POINTS TO BE PROCESSED,
C     (NPBLK),CURRENT POINT POINTER (IPNT),ROW/COLUMN OFFSET POINTER(IOFS),
C     AND THE MASK FOR THE TWIDDLE FACTOR GENERATOR.
C
      IBLK=1
      NPBLK=2**(ISTG-2)
      ICADD=0
      DO 5 J=1,4
      IPNT(J)=0
      DO 5 K=1,2
      IOFS(J,K)=0
5     CONTINUE
      MASK1=-1
      CALL SHFT(MASK,MASK1,(ISTG-1))
C
C     COMPUTE THE BLOCK, SUB-BLOCK AND THE POSITION OF THE 4 INPUT
C     POINTS FOR THE NEXT BUTTERFLY OPERATION. WE HANDLE THE 1ST FFT
C     STAGE SEPARATELY.
C
```

```
      DO 400 I=1,NB2
      DO 300 J=1,NB2
      IF(ISTG.NE.1) GO TO 10
      I1=I
      J1=J
      I2=I+NB2
      J2=J
      I3=I
      J3=J+NB2
      I4=I2
      J4=J3
      GO TO 15
10    I1=NBOFS(IBLK,1)+IOFS(IBLK,1)+1
      J1=NBOFS(IBLK,2)+IOFS(IBLK,2)+1
      I2=I1+NB4
      J2=J1
      I3=I1
      J3=J1+NB4
      I4=I2
      J4=J3
      IPNT(IBLK)=MOD((IPNT(IBLK)+1),NPBLK)
      IOFS(IBLK,2)=MOD((IOFS(IBLK,2)+1),NB4)
      IF(IPNT(IBLK).NE.0) GO TO 15
      IF(IOFS(IBLK,2).NE.0) GO TO 12
      IOFS(IBLK,1)=IOFS(IBLK,1)+1
      IF(MOD(IOFS(IBLK,1),NPBLK).NE.0) GO TO 12
      IF(IBLK.EQ.2) ICADD=2
      IF(IBLK.EQ.4) ICADD=0
12    IBLK=MOD(IBLK,2)+ICADD+1
15    K1=I
      L1=J
C
C     COMPUTE THE ADDRESSES OF THE TWIDDLE FACTORS REQUIRED FOR THE
C     CURRENT BUTTERFLY OPERATION.
C
      ITF=IAND((K1-1),MASK)+1
      JTF=IAND((L1-1),MASK)+1
      IJTF=ITF+JTF-1
C
C     PERFORM THE NEXT BUTTERFLY OPERATION AND STORE THE OUTPUT INTO
C     THE OUTPUT BUFFER.
C
      XBUTT(1)=XIO(I1,J1,INB)+XIO(I2,J2,INB)
      XBUTT(2)=XIO(I1,J1,INB)-XIO(I2,J2,INB)
      XBUTT(3)=XIO(I3,J3,INB)+XIO(I4,J4,INB)
      XBUTT(4)=XIO(I3,J3,INB)-XIO(I4,J4,INB)
      K2=K1+NB2
      L2=L1+NB2
      XIO(K1,L1,IOB)=XBUTT(1)+XBUTT(3)
      XIO(K2,L1,IOB)=(XBUTT(2)+XBUTT(4))*TFTAB(ITF,INVIN)
      XIO(K1,L2,IOB)=(XBUTT(1)-XBUTT(3))*TFTAB(JTF,INVIN)
      XIO(K2,L2,IOB)=(XBUTT(2)-XBUTT(4))*TFTAB(IJTF,INVIN)
C
```

```
C
C       IF WE ARE COMPUTING THE IFFT, WE DIVIDE THE OUTPUT OF THE LAST
C       FFT STAGE BY NPT**2.
C
        IF(INV.EQ.0) GO TO 300
        IF(ISTG.NE.NSTG) GO TO 300
        XIO(K1,L1,IOB)=XIO(K1,L1,IOB)/PTSSQ
        XIO(K2,L1,IOB)=XIO(K2,L1,IOB)/PTSSQ
        XIO(K1,L2,IOB)=XIO(K1,L2,IOB)/PTSSQ
        XIO(K2,L2,IOB)=XIO(K2,L2,IOB)/PTSSQ
300     CONTINUE
400     CONTINUE
        IOB1=IOB
        IOB=INB
        INB=IOB1
500     CONTINUE
        IOB1=IOB
        IOB=INB
        INB=IOB1
        RETURN
        END
```

```
        ;SUBROUTINE SHFT(IRES,IWORD,NP)
        ;SUBROUTINE TO SHIFT LEFT OR RIGHT THE DATA IN IWORD BY NP PLACES.
        ;A +VE NP SPECIFIES SHIFT LEFT AND -VE SHIFT RIGHT.
        ;THE RESULT IS STORED IN IRES.
        .TITL SHFT
        .ENT SHFT
        .ZREL
SHFT:   .SHFT
        .NREL
.SHFT:  SAVE 0
        LDA 0,ARG0,3
        STA 0,IRES
        LDA 0,@ARG1,3
        LDA 1,@ARG2,3
        MOV 1,1,SNR     ;CHECK FOR NO SHIFT
        JMP @UIT
        MOVZL# 1,1,SNC  ;CHECK SHIFT RIGHT OR LEFT
        JMP SHLF
SHRT:   MOVZR 0,0,SNR   ;SHIFT RIGHT
        JMP QUIT
        INC 1,1,SZR     ;DONE OR NOT
        JMP SHRT        ;NO
        JMP @UIT        ;YES
SHLF:   NEG 1,1         ;GET -VE OF NP
SLOP:   MOVZL 0,0,SNR   ;SHIFT LEFT
        JMP @UIT
        INC 1,1,SZR     ;DONE?
        JMP SLOP        ;NO
@UIT:   STA 0,@IRES     ;STORE THE RESULT
        RTN
IRES:   0
        .END
```

```
C
C
C       ******************** R2DIT ********************
C
C       THIS SUBROUTINE COMPUTES THE DFT OF A SEQUENCE OF LENGTH N
C       BY THE 1-D RADIX-2 DIT FFT ALGORITHM.
C       XT - REAL PART OF THE INPUT SEQUENCE
C       YT - IMAG. PART OF THE INPUT SEQUENCE
C       M  - # OF STAGES (N=2**M)
C       XT AND YT CONTAIN THE REAL & IMAG. PARTS OF THE FOURIER
C       COEFFICIENTS ON RETURN.
C
C
        SUBROUTINE R2DIT(XT,YT,N,M)
        DIMENSION XT(N),YT(N)
        DOUBLE PRECISION PI2,SCL,ARG,C,S
        NV2=N/2
        NM1=N-1
        J=1
        DO 7 I=1,NM1
        IF(I.GE.J) GO TO 5
        TEMP=XT(J)
        XT(J)=XT(I)
        XT(I)=TEMP
        TEMP=YT(J)
        YT(J)=YT(I)
        YT(I)=TEMP
 5      K=NV2
 6      IF(K.GE.J) GO TO 7
        J=J-K
        K=K/2
        GO TO 6
 7      J=J+K
        PI2=2.D0*3.14159265358979 3
        DO 20 L=1,M
        LE=2**L
        LE1=LE/2
        SCL=PI2/LE
        DO 20 J=1,LE1
        ARG=(J-1)*SCL
        C=COS(ARG)
        S=SIN(ARG)
        DO 20 I=J,N,LE
        IP=I+LE1
        T1=XT(IP)*C+YT(IP)*S
        T2=YT(IP)*C-XT(IP)*S
        XT(IP)=XT(I)-T1
        YT(IP)=YT(I)-T2
        XT(I)=XT(I)+T1
        YT(I)=YT(I)+T2
 20     CONTINUE
        RETURN
        END
```

```
C
C       ************************** FILTER2D **************************
C
C       THIS PROGRAM IMPLEMENTS THE MEMORY ARCHITECTURE OF THE 2-D
C       CONVOLVER DESCRIBED IN THE THESIS. THE PROGRAM IS DESIGNED TO
C       CONVOLVE AN IMAGE OF SIZE (128*128) WITH AN FILTER IMPULSE
C       RESPONSE OF SIZE (17*17) BY THE USE OF OVERLAP-SAVE TECHNIQUE
C       OF SECTIONED CONVOLUTIONS.
C       THE ARRAYS IRS1, IRS2 AND ICS IMPLEMENTS THE SAVE BUFFER OF THE
C       CONVOLVER.
C
C       SUBROUTINES REQUIRED - EXRDB, RMAP, FFT2D64, SHFT.
C
C       IWIND - WINDOW AREA FOR ACCESSING THE EXT. MEMORY
C       XIO   - FFT PROCESSOR'S BUFFERS
C       XRES  - ARRAY STORING THE TRANSFORM OF THE FILTER
C               IMPULSE RESPONSE
C       IRS1  - BUFFER SV1 OF THE SAVE MEMORY
C       IRS2  - BUFFER SV2 OF THE SAVE MEMORY
C       ICS   - BUFFER SV3 OF THE SAVE MEMORY
C
        INTEGER IWIND(1024)
        COMPLEX XIO(32,32,2),XRES(32,32),XIN1,XIN2
        DOUBLE PRECISION COMPLEX TFTAB(32,2)
        COMMON/WIND/ IWIND
        COMMON/FFT1/INB,IOB,NSTG
        DATA INB/1/,IOB/2/,NSTG/5/
        INTEGER INFIL(5),OUTFIL(5),RESFIL(5),TFFIL(5)
        INTEGER IRS1(16,16,4),IRS2(16,16,4),ICS(16,16,4)
C
C       GET THE INPUT/OUTPUT FILE NAMES.
C
        TYPE"INPUT IMAGE FILENAME : "
        READ(11,1)(INFIL(I),I=1,5)
1       FORMAT(5S2)
        TYPE"OUPUT IMAGE FILENAME : "
        READ(11,1)(OUTFIL(I),I=1,5)
        TYPE"FILTER RESPONSE FILENAME : "
        READ(11,1)(RESFIL(I),I=1,5)
        TYPE"TWIDDLE FACTOR FILENAME : "
        READ(11,1)(TFFIL(I),I=1,5)
C
C       READ THE FILES CONTAINING THE TRANSFORM OF THE FILTER IMPULSE
C       RESPONSE AND THE TWIDDLE FACTORS.
C
        OPEN 0,RESFIL
        OPEN 1,TFFIL
        DO 20 I=1,32
        READ(0)(XRES(I,J),J=1,32)
20      CONTINUE
        DO 30 I=1,2
        READ(1)(TFTAB(J,I),J=1,32)
30      CONTINUE
        CLOSE 0
        CLOSE 1
```

```
      CALL VMEM(NBLK, IER)
      IF(NBLK. GE. 20) GO TO 40
      TYPE"NOT ENOUGH EX. MEM?? NBLK = ", NBLK
      STOP
C
C     RESERVE 20 BLOCKS OF EX. MEM. & READ THE INPUT IMAGE INTO FIRST 16
C     BLOCKS OF THE EX. MEM. INITIAL DISK BLOCK = 0, INITIAL EX. MEM. BLOCK
C     = 1, TOTAL # OF DISK BLOCKS TO BE READ = 64.
C
40    IDBLK=64
      CALL EXRDB(INFIL, IWIND, 0, 1, 20, IDBLK, IER)
      IF(IER. EQ. 0) GO TO 50
      TYPE"EXRDB ERROR #, PARTIAL DISC BLOCKS READ = ", IER, IDBLK
      STOP
50    OPEN 0, OUTFIL, LEN=256, REC=128
C
C     INITIALIZE THE SAVE BUFFERS TO ZERO.
C
55    DO 60 I=1, 4
      DO 60 J=1, 16
      DO 60 K=1, 16
      IRS1(K, J, I)=0
      IRS2(K, J, I)=0
60    CONTINUE
      DO 70 I=1, 2
      DO 70 J=1, 16
      DO 70 K=1, 16
      ICS(K, J, I)=0
70    CONTINUE
C
C     BEGINING OF THE CONVOLUTION BY THE OVERLAP-SAVE TECHNIQUE.
C     WE SPLIT THE (128*128)-POINT INPUT IMAGE INTO 64 SECTIONS
C     WHERE EACH SECTION CONTAINS (16*16) PIXELS. THE DATA FROM
C     TWO SUCCESSIVE IMAGE SECTIONS AND THE SAVE BUFFERS IS COMBINED,
C     TO GENERATE A BLOCK OF (32*32) COMPLEX POINT.
C
      DO 500 I=1, 8
      DO 400 J=1, 4
      TYPE"PROCESSING SECTION # ", I, "  , ", J
C
C     GET THE DATA FROM THE SAVE BUFFERS AND STORE IT IN THE
C     ARRAY XIO.
C
      DO 100 K1=1, 16
      DO 100 K2=1, 16
      XIN1=CMPLX(FLOAT(ICS(K1, K2, 2)), FLOAT(IRS1(K1, K2, J)))
      XIN2=CMPLX(FLOAT(IRS1(K1, K2, J)), FLOAT(IRS2(K1, K2, J)))
      XIO(K1, K2, INB)=XIN1
      XIO(K1, K2+16, INB)=XIN2
      ICS(K1, K2, 2)=IRS2(K1, K2, J)
100   CONTINUE
```

```
      IWBLK=(I-1)*2+1
      IWBLK1=IWBLK+1
      CALL RMAP(IWBLK, IER)
      IF(IER.EQ.0) GO TO 110
105   TYPE"RMAP ERROR # = ", IER
      STOP
110   INCOL=(J-1)*32
C
C     GET THE DATA FROM THE IMAGE SECTIONS IN THE ROW I AND COLUMNS
C     J AND J+1 AND FORM A (32*32)-POINT BLOCK IN THE ARRAY XIO.
C
      DO 150 J1=1,2
      DO 140 K1=1,8
      DO 140 K2=1,16
      IOFS1=(K1-1)*128+INCOL+K2
      IOFS2=IOFS1+16
      K3=(J1-1)*8+K1
      XIN1=CMPLX(FLOAT(ICS(K3,K2,1)),FLOAT(IWIND(IOFS1)))
      XIN2=CMPLX(FLOAT(IWIND(IOFS1)),FLOAT(IWIND(IOFS2)))
      K4=K3+16
      XIO(K4,K2,INB)=XIN1
      XIO(K4,K2+16,INB)=XIN2
      ICS(K3,K2,1)=IWIND(IOFS2)
      IRS2(K3,K2,J)=IWIND(IOFS2)
      IRS1(K3,K2,J)=IWIND(IOFS1)
140   CONTINUE
      CALL RMAP(IWBLK1, IER)
      IF(IER.NE.0) GO TO 105
150   CONTINUE
C
C     COMPUTE THE DFT OF THE IMAGE SECTIONS IN THE ARRAY XIO BY THE
C     2-D RADIX-2 DIOO FFT ALGORITHM.
C
      INV=0
      CALL FFT2D64(XIO,TFTAB,INV,32)
C
C     MULTIPLY THE DFT OF THE CURRENT IMAGE SECTIONS WITH THE DFT
C     OF THE FILTER IMPULSE RESPONSE.
C
      DO 160 K1=1,32
      DO 160 K2=1,32
      XIO(K1,K2,INB)=XIO(K1,K2,IOB)*XRES(K1,K2)
160   CONTINUE
C
C     COMPUTE THE IDFT OF THE PRODUCT IN THE ARRAY XIO.
C
      INV=1
      CALL FFT2D64(XIO,TFTAB,INV,32)
C
C     RECOVER THE TWO SECTIONS OF THE FILTERED IMAGE FROM THE REAL
C     AND IMAG. PARTS OF THE LAST 16 ROWS AND 16 COLS. OF XIO, AND
C     STORE THE FILTERED IMAGE SECTIONS INTO THE EXT. MEMORY.
C
```

```
      CALL RMAP(IWBLK, IER)
      IF(IER.NE.0) GO TO 105
      DO 200 J1=1,2
      DO 180 K1=1,8
      DO 180 K2=1,16
      IOFS1=(K1-1)*128+INCOL+K2
      IOFS2=IOFS1+16
      K3=(J1-1)*8+K1+16
      K4=K2+16
      IWIND(IOFS1)=REAL(XIO(K3,K4,IOB))
      IWIND(IOFS2)=AIMAG(XIO(K3,K4,IOB))
      IF(ABS(REAL(XIO(K3,K4,IOB))).GT.32767.0) GO TO 170
      IF(ABS(AIMAG(XIO(K3,K4,IOB))).GT.32767.0) GO TO 170
      GO TO 180
170   WRITE FREE(12) K3,K4,XIO(K3,K4,IOB),IOFS1,IWIND(IOFS1),IOFS2,
     1  IWIND(IOFS2)
180   CONTINUE
      CALL RMAP(IWBLK1,IER)
      IF(IER.NE.0) GO TO 105
200   CONTINUE
C
C     REPEAT THE ABOVE PROCESS FOR THE OTHER IMAGE SECTIONS IN THE
C     CURRENT ROW I.
C
400   CONTINUE
C
C     AT THE BEGINING OF A NEW ROW OF SECTIONS WE CLEAR THE SAVE
C     BUFFER ICS.
C
      DO 450 J1=1,2
      DO 450 K1=1,16
      DO 450 K2=1,16
      ICS(K2,K1,J1)=0
450   CONTINUE
500   CONTINUE
C
C     AT THIS POINT ALL THE IMAGE SECTIONS HAS BEEN PROCESSED.  STORE
C     THE FILTERED IMAGE INTO THE FILE 'OUTFIL'.
C
510   DO 550 I=1,16
      CALL RMAP(I,IER)
      IF(IER.NE.0) GO TO 105
      DO 540 K=1,8
      K1=(K-1)*128
      WRITE(0)(IWIND(J+K1),J=1,128)
540   CONTINUE
550   CONTINUE
      STOP
      END
```

```
        ;SUBROUTINE EXRDB(TABFIL,IWIND,DISKBLK,EXMBLK,NBLK,PBLK,IER)
        ;SUBROUTINE TO READ A DISK FILE INTO EXT. MEMORY
        ;AND DEFINE AN EXT. MEMORY MAP
        ;TABFIL: NAME OF THE FILE
        ;IWIND: ADDRESS OF THE WINDOW IN LOGICAL MEMORY
        ;DISKBLK: INITIAL DISK BLOCK # TO BE READ
        ;EXMBLK: INITIAL EXT. MEMORY MAP BLOCK WHICH WILL RECIEVE THE DATA
        ;(THIS MUST BE MULTIPLE OF 1K)
        ;NBLK: TOTAL # OF 1K WORD BLOCKS IN THE MAP
        ;PBLK: SPECIFIES TOTAL # OF DISK BLOCKS TO BE READ AND ALSO STORES
        ;ON RETURN, THE PARTIAL BLOCK COUNT IN CASE OF END OF FILE ERROR.
        ;IER: CONTAINS SYSTEM ERROR CODE .
        .TITL EXRDB
        .ENT EXRDB
        .ZREL
EXRDB:  .EXDB
        .NREL
.EXDB:  SAVE 0
        LDA 0,ARG0,3
        MOVZL 0,0       ;GET BYTE POINTER OF THE FILE NAME.
        STA 0,TABFIL
        LDA 1,ARG1,3
        LDA 2,MASK3
        ANDS 2,1
        MOVZR 1,1
        MOVZR 1,1
        STA 1,WINBLK
        LDA 1,@ARG2,3
        STA 1,DISKBLK
        LDA 1,@ARG3,3
        STA 1,EXMBLK
        LDA 1,@ARG4,3
        STA 1,MAPBLK
        LDA 1,ARG5,3
        STA 1,PBLK
        LDA 1,ARG6,3
        STA 1,IER
        SUB 1,1
        STA 1,@IER
        .SYSTM
        .GCHN
        JMP ERR1
        STA 2,FCHAN
        .SYSTM
        .OPEN 77
        JMP ERR1
        LDA 0,MAPBLK
        LDA 1,WINBLK
        LDA 2,EXMBLK
        .SYSTM
        .MAPDF
        JMP ERR1
        LDA 0,EXMBLK
```

```
        LDA 1,@PBLK     ;GET TOTAL # OF DISK BLOCKS
        MOVS 1,1        ;SWITCH THE # TO LEFT BYTE
        LDA 2,FCHAN
        ADDZ 1,2
        LDA 1,DISKBLK
        .SYSTM
        .ERDB 77
        JMP ERR2
CLFIL:  LDA 2,FCHAN
        .SYSTM
        .CLOSE 77
ERR1:   STA 2,@IER
        RTN
ERR2:   LDA 1,MASK1
        ANDS 2,1
        STA 1,@PBLK
        LDA 1,MASK2
        AND 1,2
        STA 2,@IER
        JMP CLFIL
TABFIL: 0
DISKBLK:        0
EXMBLK: 0
PBLK:   0
IER:    0
FCHAN:  0
MASK1:  177400
MASK2:  377
MASK3:  176000
WINBLK: 0
MAPBLK: 0
        .END

        ;SUBROUTINE RMAP(BLOCK,IER)
        ;THIS SUBROUTINE MAPS WINDOW BLOCK # 0 TO THE RELATIVE BLOCK #
        ;'BLOCK' OF THE EXT. MEMORY MAP.
        .TITL RMAP
        .ENT RMAP
        .EXTN .REMA
        .ZREL
RMAP:   .RMAP
        .NREL
.RMAP:  SAVE 0
        LDA 1,@ARG0,3   ;GET RMAP BLOCK #
        MOVS 1,1        ;MOVE BLOCK # TO THE LEFT BYTE.RIGHT BYTE =0
        LDA 0,ARG1,3
        STA 0,IER
        SUB 0,0
        STA 0,@IER
        LDA 2,C1        ;ONLY BLOCK 0 OF THE WINDOW WILL BE RMAPPED
        .REMA
        JMP ERR
        RTN
ERR:    STA 2,@IER
        RTN
C1:     1
IER:    0
        END
```

```
C
C
C       ********************* ERRIMAG *********************
C
C       THIS PROGRAM COMPUTES THE ERROR BETWEEN THE TWO INPUT IMAGES
C       AND STORES THE ERROR INTO THE SPECIFIED DISC FILE.  IT ALSO
C       PRINTS THE % RMS ERROR  ON THE CONSOLE.
C
C

        INTEGER IMG1(256), IMG2(256), IFILE1(10), IFILE2(10), EFILE(10)
        DOUBLE PRECISION SUM, ERR
        TYPE"FILE NAME:  IMAGE # 1 :"
        READ(11,1)(IFILE1(I), I=1,10)
1       FORMAT(10S2)
        TYPE"FILE NAME:  IMAGE # 2 :"
        READ(11,1)(IFILE2(I), I=1,10)
        TYPE"FILE NAME TO STORE ERROR : "
        READ(11,1)(EFILE(I), I=1,10)
        ACCEPT"# OF ROWS/COLS.  IN THE IMAGES : ", IRL
        IRC=IRL*2
        OPEN 0, IFILE1
        OPEN 1, IFILE2
        OPEN 2, EFILE, LEN=IRC, REC=IRL
        SUM=0.D0
        ERR=0.D0
        DO 50 I=1, IRL
        READ(0)(IMG1(J), J=1, IRL)
        READ(1)(IMG2(J), J=1, IRL)
        DO 40 J=1, IRL
        SUM=SUM+DFLOAT(IMG1(J))**2
        IMG1(J)=IMG1(J)-IMG2(J)
        ERR=ERR+DFLOAT(IMG1(J))**2
40      CONTINUE
        WRITE(2)(IMG1(J), J=1, IRL)
50      CONTINUE
        ERR=(ERR/SUM)*100.D0
        TYPE"% RMS ERROR BETWEEN THE TWO IMAGES = ", ERR
        CLOSE 0
        CLOSE 1
        CLOSE 2
        STOP
        END
```

VITA AUCTORIS

1950	Born on August 12, Abohar, Punjab, India.
1965	Completed high school at Govt. High School, Abohar, Punjab, India.
1967	Completed Pre-Engineering at the University of Panjab, Punjab, India.
1972	Graduated from University of Kurukshetra, Kurukshetra, Haryana, India, with the degree of Bachelor of Science in Electrical Engineering.
1972-74	Worked as Maintenance Engineer at Indo Superfine Corp., Delhi, India.
1976	Graduated from University of Windsor, Windsor, Ontario, with the degree of M.A.Sc. in Electrical Engineering.
1981	Candidate for the degree of Doctor of Philosophy in Electrical Engineering, University of Windsor, Windsor, Ontario, Canada.