Order Number 9108112

The design of discrete Fourier transform and convolution algorithms for RISC architectures

Granata, John A., Ph.D.

City University of New York, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

THE DESIGN OF DISCRETE FOURIER TRANSFORM AND CONVOLUTION ALGORITHMS FOR RISC ARCHITECTURES

by

John A. Granata

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1990

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

9/19/90.
Date

Michael Cohen
Chair of Examining Committee

9/21/90
Date

Gerard G. Gowen
Executive Officer

Dr. Richard Tolimieri

Dr. James Cooley

Dr. Sanghamitra Basu

Dr. Joseph Barba

Supervisory Committee

The City University of New York

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Statement of Problem

The performance of an algorithm is highly dependent on the underlying architectural configuration of the computer that executes it, and therefore, the emergence of the modern RISC architecture presents software engineers with new criteria and challenges for fast algorithm design. The problem of designing efficient discrete Fourier transform (DFT) and convolution algorithms for these architectures is investigated in this thesis. Special attention is given to DFT and convolution because applications of these algorithms are found whenever dealing with linear systems or piece-wise linear approximations of nonlinear systems. Some of these include: spectral analysis, linear predictive coding, digital image restoration and compensation, sonar, radar, seismology, computerized tomography, decoding schemes for error control encoders, etc..

The tensor product is at the center of the techniques and strategies that will be presented. The tensor product is a binary matrix operator and has recently been demonstrated as a powerful tool for modeling DFT algorithms for vector and multi-processor computers. In this work the breadth of the tensor product is extended in two ways: (1) by developing a tensor product decomposition for convolution and several other transforms it is demonstrated that many of the results obtained for DFT on vector and multi-processors are directly applicable to a much larger class of recursive algorithms, (2) by establishing connections between particular tensor product constructs and newly emerging RISC architectures it becomes possible

to approach the problem of designing fast algorithms for vector, multi, and RISC processors in a unified format.

1.2 The RISC Architecture

The RISC, or Reduced Instruction Set Computer, is not a new idea to computing; in fact, the first computers were RISC machines. It was the emergence of high level programming languages and complicated multi-tasking operating systems that fueled the transition from RISC machines to Complex Instruction Set Computers or CISCs. By having hundreds of instructions and many addressing modes CISCs dramatically reduced the programming effort needed to develop these applications. Today, most computers are based on the CISC methodology.

Although CISC is very appealing from a programmer's point of view, there is invariably a performance price to pay for this convenience. In order to support a large instruction set a CISC usually has a micro-coded control unit. For all the benefit of micro-coding it is ultimately a software procedure which makes it inherently slower than a hardware approach. In addition to being slow, a micro-coded control unit also requires its own memory and therefore consumes a large percentage of valuable chip area.

By having only one or two data instructions modern RISC machines eliminate the data control unit and thereby gain two critical advantages.

(1): Because the basic compute cycle is essentially fixed, it can be highly optimized by using pipelining and other hardware techniques.

(2): By eliminating the large micro coded control unit valuable chip area is made available for other features such as: dedicated hardware floating point adders, multipliers, large cache memory, and fixed point addressing units.

The exploitation of these two advantages makes possible the enormous gains in performance achieved by modern RISC machines. These also present the software engineer with a new set of challenges for fast algorithm design.

Most RISC machines have three basic resources: a dedicated fixed point addressing unit, floating point hardware multiplier, and adder. The pipelining introduced into the compute cycle allows these to operate in parallel and also makes the time needed to implement a multiplication the same

as that needed for addition. This feature of modern RISC architectures is very different from computers based on the architectures of the late sixties to seventies where multiplication took much longer to compute than addition. Since many fast algorithms were designed to exploit this feature of the underlying architecture, the implementation of algorithms based on this approach can lead to very disappointing results when used on the modern RISC. This was pointed out as early as 1983 by Temperton [34] as it also pertains to the design of FFT algorithms on large vector processors.

The return to the RISC approach began with the development of RISC digital signal processing (DSP) chips [40-43]. These have been designed exclusively for numeric computation and therefore have very simple addressing units. The great increase in performance attained by these chips has spurred the development of RISC chips with more elaborate addressing units that could be used as a general purpose CPUs [43]. The study of these chips is important not only because they represent an important trend in uni-processor computer design but also because many modern supercomputers are based on multi-processor configurations of high performance RISC chips [56-58].

The problem of designing DFT and convolution algorithms for RISC machines will be studied by introducing two abstract computational models. These are representative of many commercially available RISC machines. Before proceeding to describe the particular aspects of the models that make them different, some features common to both are presented. Note that the description of these models refers to the programmers view for a given RISC and need not reflect the actual underlying physical configuration.

For both models it is assumed that data is represented using a floating point format, while the data addresses are specified using a fixed point representation. Both models have a dedicated floating point hardware multiplier and adder and the time required to implement addition is the same as that required for multiplication. In conjunction with the adder and multiplier there exists a separate addressing unit (AU). The function of the addressing unit is to compute operand addresses for the adder and multiplier. The addressing unit can access operands either in order or by some fixed stride k.

It is further assumed that all data reside in main memory. The existence of cache memories and 'on chip' data registers is not initially considered.

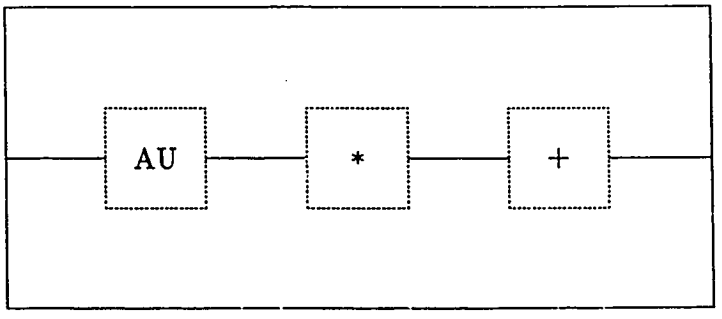

Figure 1.1: Multiply Add RISC

This fixes the time required to access an operand. This assumption is made because this particular aspect of an architecture is highly machine dependent. For example, the TMS 320C30 DSP chip [41] employs a local instruction cache, the ATT DSP32 [40] separates main memory into two banks and requires that memory access be interleaved between banks in order to avoid the insertion of wait states into the instruction cycle, while the Intel 860 [43] employs a complete, on chip, paged memory management system. Rather than trying to incorporate these various arrangements into the algorithmic design process, this level of design should be dealt with at the implementation level.

There are two variations of the basic RISC architecture that will be studied. These two reflect different levels of parallelism between that adder and the multiplier.

Multiply - Add Kernel: For the multiply- add kernel computational model the addressing unit, the data adder, and the data multiplier all operate in parallel, and the programmer has independent control of each of these resources. This configuration will be refered to as a model I RISC.

Multiply - Accumulate Kernel: For the multiply- accumulate computational model it is assumed that the addressing unit, the data adder, and the data multiplier, operate in parallel only when using multiply-accumulate instructions (MAC) of the form $a + (b * c)$. This configuration will be refered to as a model II RISC.

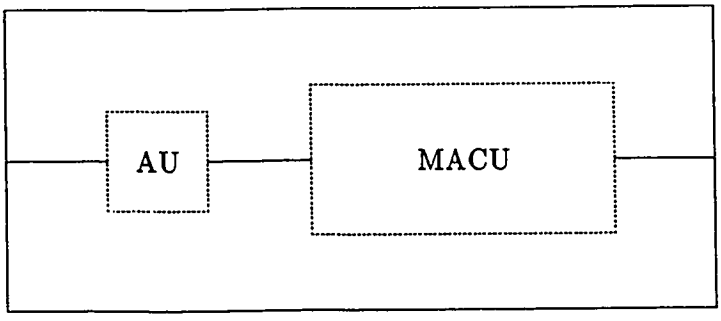

Figure 1.2: Multiply Accumulate RISC

It should be clear that any algorithm designed for a multiply accumulate RISC will automatically be legitimate for a multiply add RISC. The benefit, however, of considering these models separately is derived from the possibility of exploiting the different levels of parallelism to achieve better performance.

1.3 Thesis Outline

The objective of this work has been stated. Chapter two begins with some mathematical preliminaries that will be needed throughout. This is followed by a survey of the most widely known fast algorithms for DFT and convolution. Use of the tensor product is made wherever possible. In chapter three a tensor product decomposition of the linear convolution matrix is presented. It is shown that the use of tensor product for convolution leads to a new family of fast convolution algorithms which are analogous to additive FFT algorithms. Fully vectorized and parallel generalized radix r as well as mixed radix convolution algorithms are derived. In chapter four the breadth of the tensor product is enhanced to deal with new RISC configurations. This is achieved by establishing connections between certain tensor matrices and RISC architectures. Chapter five offers tensor product representations of some other important transforms; these include Walsh transform, Hartley transform, and cosine transform. It is shown that the tensor product allows one to model a more generalized class of recursive algorithms. This greatly widens the ideas presented in chapters two, three, and four because many important algorithms rely on recursive structure.

Some concluding remarks and directions for future research are offered in chapter six.

Chapter 2

A Survey of Fast Algorithms

A survey of fast algorithms for convolution and discrete Fourier transform is presented in this chapter. This material spans three decades of journal articles and technical reports. Blahut [1], provides an excellent starting point for those interested in this field, but who lack the mathematical background needed to understand the many journal articles referenced. More advanced readers will fully appreciate the more recent text by Tolimieri *et al* [2].

2.1 Mathematical Preliminaries

In this section a mathematical foundation of fast algorithm design is developed. For the most part the presentation will not be tutorial in nature, but will instead, represent a collection of important concepts and theorems. This will establish notation for those familiar with this material and also offer a roadmap for those unfamiliar with this material. The exception will be section 2.1.5 where the tensor product is demonstrated to be a mathematical programming language.

This section has been organized into six subsections. The first begins with some basic definitions and results from elementary algebra. Much of the material in this section can be found in any basic text; two are [3,4]. Some important properties of matrices are presented in the second subsection. These and many other properties of matrices can be found in [5]. In subsection three, the discussion turns to some important results from complexity theory as it pertains to the computation of convolution and the

discrete Fourier transform. This material is due to Winograd and can be found in references [6,7,8]. Also see reference [36] for a unified representation of multiplicative complexity theory. Some important definitions and theorems from scheduling theory [46] are offered in subsection four. The connection between certain tensor matrices and computer architectures, as presented in [2,53], is established in subsection five. The section ends by presenting a list of theorems to be referenced in later chapters.

2.1.1 Review of Elementary Algebra

An algebraic system is a set S together with a binary operator $\circ : (S \times S) \mapsto S$. The ordered pair $(S, \circ)$ is called the algebraic system defined by $\circ$. By adding special properties to the operator $\circ$, special types of algebraic systems are obtained. If, for instance, $\circ$ is an associative binary operator the algebraic system $(S, \circ)$ is called a *semigroup* . More importantly, a *group* is defined when

1. $(S, \circ)$ is a semigroup.
2. There exists an identity element (called the zero element and denoted by 0) in S such that $a \circ 0 = a, \forall a \in S$.
3. If for any element a there exists a unique element b such that $a \circ b = 0$.

Note that if $\circ$ satisfies 1, 2 and 3, and is also a commutative operator, the system $(S, \circ)$ is called a commutative or *abelian group* . The study of these structures is called group theory and it plays a central role in the design of fast algorithms for digital signal processing.

Another type of algebraic system of fundamental importance is the *ring*. In this system there are two binary operations on a set S. Following the above convention, the three tuple $(S, \circ, \wedge)$ is called the ring defined by $\circ$ and $\wedge$. The properties of $\circ$ and $\wedge$ are such that

1. $(S, \circ)$ is an abelian group.
2. $(S, \wedge)$ is a semigroup
3. $\wedge$ distributes over $\circ$. That is,

$$a \wedge (b \circ c) = (a \wedge b) \circ (a \wedge c) \quad \forall a, b, c \in S$$

By giving more structure to the multiplicative semigroup of a ring another algebraic system of interest is obtained, this is a *field* . Fields have the following properties on $\wedge$ and $\circ$.

1. $(S, \circ, \wedge)$ is a ring
2. $(S', \wedge)$ is an abelian group, where $S' = S - \{0\}$.

The identity element of the group $(S', \wedge)$ is called the unity element and is denoted by the symbol 1. Finally, a ***vector space*** is defined when two sets of objects are specified. A set of scalers S, and a set of vectors V. The properties of these sets are given below for the arbitrary elements $v, v_1 \in V$ and $s, s_1 \in S$.

1. $(S, \circ, \wedge)$ is a field
2. $(V, \diamond)$ is an abelian group

   There exists a scaler-vector multiplication, denoted by juxtaposition, such that

3. $1v = v$
4. $s(v \diamond v_1) = sv \diamond sv_1$
5. $(s \wedge s_1)v = sv \diamond s_1 v$
6. $s(s_1 v) = (s \wedge s_1)v$

Familiar examples of these abstract algebraic structures are numerous. For example, the set of integers taken with the usual binary operators of addition and multiplication is readily seen to be a ring. This ring is denoted by $(Z, +, *)$, or is usually abbreviated to simply Z. Because the integers form a ring, it follows immediately that $(Z, +)$, or the set of integers taken with the normal addition operator, constitutes an abelian group. Another example of an algebraic structure is the set of real numbers with conventional multiplication and addition. This is readily shown to be a field and is denoted by $(R, +, *)$, or simply R. Similarly the set of complex numbers is the field $(C, +, *)$, which is denoted by C.

In the examples considered, the cardinality of the set S has been infinite. It is also possible that S have a finite number of elements. For instance, the set $Z_n = \{0, 1, ..., n-1\}$ together with multiplication and addition taken *mod* n is the finite ring $(Z_n, +, *)$ or simply Z_n. If n is a prime p, in the definition of Z_n, it is easily verified that Z_p is a field. Finite fields are often referred to as *Galios fields* after Evariste Galios (1811-1832) who first investigated their structure. Galois fields of order n are denoted by $GF(n)$.

The integer ring Z_n, as described above, can be shown to be structurally equivalent (or isomorphic) to the *quotient ring* Z/n. The quotient ring is introduced by considering the ring Z; two elements $a, b \in Z$ are congruent *mod* n if and only if n divides $(a-b)$ where $(-b)$ is the additive inverse of b. Congruence *mod* n imposes a partition on the set of integers Z into n *equivalence classes* which is denoted by $Z^n = \{Z^0, Z^1, ...Z^n - 1\}$. This set taken with the binary operations

$$\circ : (Z^j, Z^k) \mapsto Z^{(j+k) \bmod n}$$

$$\wedge : (Z^j, Z^k) \mapsto Z^{(j*k) \bmod n}$$

is the ring $(Z^n, \circ, \wedge)$ which is denoted by Z/n.

The above discussion of structurally equivalent groups leads very naturally to the concept of isomorphic groups. Two groups $(S, +)$ and $(A, \circ)$ are said to be *isomorphic* if there exists a bijection $b : S \mapsto A$ such that $b(s_1 + s_2) = b(s_1) \circ b(s_2)$ where $s_i \in S$. Stated another way; two groups are isomorphic if and only if there exists a one to one correspondence between the elements of the set S and the set A which preserves their group operations under the map b. If two groups $(S, +)$ and $(A, \circ)$ are isomorphic this is denoted by $S \cong A$. From an algebraic point of view isomorphic groups are equivalent.

The concept of isomorphism carries over to other algebraic structures like rings and fields. With respect to fields, it can be shown that every finite field of order n is isomorphic to every other field of order n. Therefore, to within a relabeling of the elements of the set S, there is precisely one finite field of order n. This result is used extensively in the following chapters.

Consider the group $(S, *)$; let $s_i \in S$. Then, the element $(s_i)^k$ is obtained by multiplying s_i by itself k times. The *order of the group* $(S, *)$ is the cardinality of the set S. The *order of an element* s_i is the smallest integer k such that $(s_i)^k = 1$ where 1 is the identity of $(S, *)$. It can be shown

that the order of any element of a group divides the order of the group. Further, if there exists an element s_i, of the group S, whose order is the cardinality of the set S, the group S is a *cyclic group* and the element s_i is called the *generator* of the group. In a cyclic group any element can be uniquely expressed as a power of the generator.

A *primitive* element α of a field is that element which is a generator of the multiplicative group of the field. If the field is a Galios field then for every $GF(n)$ there exists at least one primitive element for every n. This means that the multiplicative group of any Galois field is a cyclic group, and that every non-zero element of $GF(n)$ can be expressed uniquely as a power of a primitive element.

In order to derive fast algorithms, it will be necessary to decompose large algebraic structures into smaller ones. One way of reducing a large ring to obtain smaller rings has already been presented in the construction of Z/n from Z. In the case where n is prime it was stated that Z/p is a field. This gave a way of constructing smaller fields from a large ring. There are many other ways in which algebraic systems can be combined and decomposed. Two additional techniques will be presented; the first is the method of forming an extension field from a given base field, the second is the direct sum representation.

Consider a field F and let the set $\{x^{n-1}, \cdots, x^1, x^0\}$ be a set of n indeterminates. An extension of the field F to the indeterminates $\{x^i\}$ is the set of forms of the type $\{f_{n-1}x^{n-1} + \cdots + f_1x^1 + f_0x^0\}$ where $f_i \in F$. This form can be thought of as a polynomial of degree $n-1$ over the field F. Let $F[x]$ denote the set of all possible polynomials over the field F, then it can be shown that $(F[x], +, *)$ is a ring where $+$ and $*$ are taken as conventional polynomial addition and multiplication. In this construction the field F is called the *base field* , and ring $F[x]$ is called the *extension ring* .

In the same way that the quotient ring Z/n was formed from Z, it is possible to form the quotient ring $F[x]/q(x)$ from the ring $F[x]$. In this respect, it is readily shown that $F[x]$ taken with mod $q(x)$ polynomial addition and multiplication is a ring, which will be denoted by $F[x]/q(x)$. As for integers, $F[x]/p(x)$ is a field when $p(x)$ is a *prime polynomial* over the field F. A prime polynomial over a field F is a polynomial that is both *monic* and *irreducible* over F. A polynomial over a field F is irreducible if none of its roots are in F. A polynomial is monic if its leading coefficient is unity. In these constructions the field $F[x]/p(x)$ is called an *extension*

field and the underlying field F is called the *base field.* Hence, from a given field F, larger fields $F[x]/p(x)$ can be made as long as it is possible to find prime polynomials $p(x)$ over $F[x]$.

Another way of combining algebraic structures is by using the direct sum operator. Consider two groups $(G,+)$ and $(S,*)$; the *direct sum* of G and S is the set which is the cartesian product of the sets G and S together with the binary operator $\circ : G \times S \mapsto G \times S$ where $\circ$ is defined by $(g_i, s_i) \circ (g_k, s_k) = (g_i + g_k, s_i * s_k)$. The direct sum of the groups $(G,+)$ and $(S,*)$ is denoted by $(G \oplus S, \circ)$. It can be shown that $(G \oplus S, \circ)$ is itself a group. Hence, the direct sum gives a way of combining arbitrary groups in order to make new larger groups, as well as a way of decomposing a large group into its smaller *direct summands* . Note that this concept extends directly to the notion of direct sums of rings.

Having reviewed these basic algebraic notions the next subsection presents some definitions and theorems regarding matrices. The matrix plays a central role in our work because so many fast algorithms for digital signal processing can be thought of as matrix factorizations.

2.1.2 Introductory Matrix Material

It is possible to think of linear algorithms as matrix multiplications. To be more precise, an input vector $\bar{x_n}$ of length n will be given and an output vector $\bar{y_k}$ of length k will be desired. The particular computation being specified by the k by n matrix M which carries $\bar{x_n}$ to $\bar{y_k}$. Thinking in this way allows one to state the problem of finding fast algorithms for a computation $\bar{y_k} = (M)\bar{x_n}$ as finding a factorization of the matrix $(M) = \prod_i m_i$ such that the product of the input vector $\bar{x_n}$ with the factors $(\prod_i m_i)$ requires fewer operations than the product of the input and the original matrix (M). This will be the approach taken in designing algorithms. Because matrices play such a central role, this section is used to state some basic properties of matrices which may not, in general, be familiar to an electrical engineer or computer scientist. These are the notion of permutation matrices, the direct sum of matrices, and the tensor product of matrices.

A *permutation matrix* P_n of order n is an n by n matrix in which each row and each column has precisely a single 1 and zeros elsewhere. A matrix of this form is called a permutation matrix because when acted upon by an

input vector of length n the resulting output vector is a shuffled version of the original input. It is easily verified that there are $n!$ unique permutation matrices of order n. It is also a fact that the set of n by n permutation matrices form a subgroup of the multiplicative group of all n by n matrices.

A particular permutation matrix P^ϕ can be specified by letting ϕ denote the one to one mapping of the set $N = \{0, 1, 2, \cdots, n-1\}$ onto itself, $\phi : N \mapsto N$. Then P^ϕ is the permutation matrix which orders the indexing set of an input vector according to the permutation ϕ. The following properties of P^ϕ are readily established.

$$A' = P^\phi A \tag{2.1}$$

where A' is A with its rows shuffled according to the permutation ϕ.

$$A'' = AP^\phi \tag{2.2}$$

where A'' is A with its columns shuffled according to the permutation ϕ^{-1}.

$$P^{\phi_1} P^{\phi_2} = P^{(\phi_1 \circ \phi_2)} \tag{2.3}$$

where $(\phi_1 \circ \phi_2)$ denotes the permutation obtained by first applying the permutation ϕ_2 and then applying ϕ_1.

$$P^\phi (P^\phi)^t = I \tag{2.4}$$

where t denotes matrix transpose and I is the identity matrix. From this it follows immediately that

$$(P^\phi)^t = (P^\phi)^{-1} = P^{\phi^{-1}} \tag{2.5}$$

Finally, let $\bar{k_n} = (k_0, k_1, k_2, \cdots, k_{n-1})^t$ then

$$(P^\phi) diag(K_n) (P^\phi)^{-1} = diag(P^\phi K_n) \tag{2.6}$$

where

$$diag(\bar{k_n}) = \begin{pmatrix} k_0 & & & & \\ & k_1 & & & \\ & & k_2 & & \\ & & & \ddots & \\ & & & & k_{n-1} \end{pmatrix}$$

Having stated these general properties, some special types of permutation matrices of particular importance are now introduced. Define the matrix π_n as the matrix which shuffles an input vector $\bar{x_n}$ as

$$\begin{bmatrix} x_{n-1} \\ x_0 \\ \vdots \\ x_{n-3} \\ x_{n-2} \end{bmatrix} = (\Pi_n) \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{bmatrix}$$

It is easily verified that

$$(\pi_n)^n = I_n \tag{2.7}$$

In light of this, and the discussion of section 2.1.1, the set of n by n permutation matrices $\{I_n, \pi_n, \pi_n^2, \cdots, \pi_n^{n-1}\}$ is the cyclic group of order n, generated by π_n, with the binary operation being ordinary matrix multiplication.

Another subset of n by n permutation matrices is the set of stride by s permutation matrices which is denoted as $P_{n,s}$. For $n = rs$ the matrix $P_{n,s}$ is defined as the matrix which shuffles $\bar{x}$ as

$$\begin{bmatrix} x_0 \\ x_s \\ \vdots \\ x_{(r-1)s} \\ x_1 \\ x_{s+1} \\ \vdots \\ x_{n-1} \end{bmatrix} = P_{n,s} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_s \\ x_{s+1} \\ x_{s+2} \\ \vdots \\ x_{n-1} \end{bmatrix}$$

For $n = rs$, $n = p^k$, $s = p$, where p is a prime it can be shown that

$$(P_{n,s})^k = I_n$$

and that the set of permutation matrices $\{I_n, P_{n,s}P_{n,s}^2, \cdots, P_{n,s}^{k-1}\}$ is a cyclic group of order k taken with the binary operator of matrix multiplication. The stride permutation plays an important role in the theory of the tensor product of matrices.

Given two arbitrary square matrices A and B

$$A_{n_1,n_2} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n_2-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n_2-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_1-1,0} & a_{n_1-1,1} & \cdots & a_{n_1-1,n_2-1} \end{pmatrix}$$

$$B_{m_1,m_2} = \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,m_2-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,m_2-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m_1-1,0} & b_{m_1-1,1} & \cdots & b_{m_1-1,m_2-1} \end{pmatrix}$$

The *tensor product of* A and B, $C = (A \otimes B)$ is defined as the $n_1 m_1$ by $n_2 m_2$ matrix given by

$$C = (A_{n_1,n_2} \otimes B_{m_1,m_2}) = \begin{pmatrix} a_{0,0}B & a_{0,1}B & \cdots & a_{0,n_2-1}B \\ a_{1,0}B & a_{1,1}B & \cdots & a_{1,n_2-1}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_1-1,0}B & a_{n_1-1,1}B & \cdots & a_{n_1-1,n_2-1}B \end{pmatrix} \quad (2.8)$$

Extensive use of this operator will be made. Listed below are some important properties of tensor products.

$$(\alpha A) \otimes B = A \otimes (\alpha B) = \alpha(A \otimes B) \qquad \alpha \in scaler \quad (2.9)$$

$$(A + B) \otimes C = (A \otimes C) + (B \otimes C) \quad (2.10)$$

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C \quad (2.11)$$

$$(A \otimes B)(C \otimes D) = (AC \otimes BD) \quad (2.12)$$

Property 2.12 is called the distributive property of the tensor product. If A and B are nonsingular, so is $(A \otimes B)$ and

$$(A \otimes B)^{-1} = (A^{-1} \otimes B^{-1}) \quad (2.13)$$

$$(A \otimes B)^t = (A^t \otimes B^t) \quad (2.14)$$

$$(I_s \otimes I_r) = I_{sr} \quad (2.15)$$

where t denotes matrix transpose and I_n is the n by n identity matrix. Property 2.15 is called the identity property of the tensor product. The

final property shows an intimate connection between the tensor product and the stride permutation matrix $P_{n,s}$.

$$(A_{s_1,s_2} \otimes B_{r_1,r_2}) = P_{r_1 s_1, s_1} (B_{r_1,r_2} \otimes A_{s_1,s_2}) P_{r_2 s_2, r_2} \tag{2.16}$$

This formula allows the use of stride permutations in order to commute matrices with respect to tensor products. This is called the commutative property of the tensor product.

The final point of this section introduces another way of combining and decomposing matrices. Let $\{A_k\}$ $k = 0, 1, \cdots, n-1$ be a set of square matrices. Then the *direct sum* of the set $\{A_k\}$ is the block diagonal matrix A defined as

$$A = \begin{pmatrix} A_0 & & & & \\ & A_1 & & & \\ & & A_2 & & \\ & & & \ddots & \\ & & & & A_{n-1} \end{pmatrix}$$

The matrix A is denoted by

$$A = A_0 \oplus A_1 \oplus \cdots A_{n-1} = \sum_{k=0}^{n-1} \oplus A_k$$

As with the tensor product, some important properties of the direct sum are listed. These are

$$(A + B) \oplus (C + D) = (A \oplus B) + (C \oplus D) \tag{2.17}$$

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C \tag{2.18}$$

$$(A \oplus B)(C \oplus D) = (AC \oplus BD) \tag{2.19}$$

If A and B are nonsingular, so is $(A \oplus B)$ and

$$(A \oplus B)^{-1} = (A^{-1} \oplus B^{-1}) \tag{2.20}$$

$$(A \oplus B)^t = (A^t \oplus B^t) \tag{2.21}$$

where t denotes matrix transpose.

2.1.3 Basic Results from Complexity Theory

Some basic ideas from a branch of mathematics known as *complexity theory* are introduced in this section. The major problems in this theory are: (1) What is the minimum number of arithmetic operations required in order to perform a computation? and (2) How can one realize algorithms which obtain this minimum? Despite the ease of which these problems are stated there are very few concrete results toward answering these questions in the general case.

The major results in this area, for our purposes, are due to Winograd and obtained by restricting the class on computations considered to those of bilinear forms, and then only minimizing in terms of multiplications. His work has lead to great success for two reasons. First, many computations in digital signal processing (including convolution and discrete Fourier transform) can be expressed as a system of bilinear forms. Second, on the computer architectures of the early to mid seventies, the time at which his work was published, multiplication was usually much slower than addition. Because of this the optimization of the number of multiplications leads to algorithms for convolution and discrete Fourier transform which, in many cases, executed orders of magnitude faster than algorithms employing a direct implementation approach.

As pointed out in chapter one, the revolution in VLSI technology has changed this primary design criteria for fast algorithms. Nevertheless, the definitions introduced and results obtained by Winograd will serve as a foundation and important starting point for us. The development starts by making more precise the notion of an algorithm and its multiplicative complexity. Some theorems which fix lower bounds on the number of multiplications required to realize a general algorithm are then given. Finally, bilinear systems are introduced and their relevance to the computation of convolution and DFT is presented.

In determining the multiplicative complexity of systems, a distinction will be made between general multiplications and multiplications with an element of a set G called the *ground set*. Say, temporarily, that the set G is the ring of integers; clearly the multiplication $y = g_i * x$ for $g_i \in Z$ could be implemented by adding x to itself g_i times. Because of this, a multiplication of an indeterminate and an integer is not counted as a multiplication. This seems valid when G is the ring of integers. However, in order to facilitate the

mathematical derivation, it is assumed that G is the field of rationals. This assumption may cause a difficulty in some implementations for the obvious reason that the realization of $y = (1/3) * x$, for example, does in fact require multiplication. Nevertheless, these difficulties are dealt with when and if they arise. This avoids any unnecessary mathematical complication at the outset.

The *base set* B is defined as the set of all possible inputs to an algorithm. The *result field* H is defined to be the field of all possible elements computed by repeated application of field operations to the base set B. Further, a *system* Z is defined to be a set $Z = \{z_0, z_1, z_2, ..., z_t\}$ of desired outputs. Based on these, a more precise definition of an algorithm can now be given.

Definition 2.1.1 *Let G, B, and H, be as described above. Then an algorithm A (over the set B) is a finite sequence of elements $\{h_0, h_1, h_2, \cdots, h_n\} \in H$ such that either $h_i \in B$ or $h_i = h_j \circ h_k$ where $j, k < i$, and $\circ$ is any of the field operators $(*, \div, -, +)$.*

An algorithm A *computes* a system Z if $Z = \{z_0, z_1, ...z_t\} \subseteq \{h_0, h_1, ...h_n\}$. The number of multiplications and divisions (m/d steps) will be minimized. These are defined below.

Definition 2.1.2 *Let A be an algorithm over some base set B. An $h_i \in H$ is said to be a non m/d step if either one of the following conditions hold.*

1. $h_i \in B$
2. $h_i = h_j \pm h_k$ *where* $j, k < i$
3. $h_i = g * h_j$ *where* $g \in G, j \leq i$

otherwise h_i is a m/d step.

The number of m/d steps in any algorithm A is called its *m/d complexity* and is denoted by $\mu(A)$.

Definition 2.1.3 *Let G,B, and H, be as described above and let $Z = \{z_0, z_1, ..., z_t\}$ be elements of H. Define the m/d complexity of the system Z as $\min_A \mu(A)$ where A ranges over all algorithms over B which computes the system Z.*

The m/d complexity of a system Z is denoted by $\mu(Z)$.

The *linear span* of a set X is the set of elements of the form $\sum_i g_i x_i$ where $g_i \in G, x_i \in X$. This set is denoted by $L_G(X)$. It should be clear from the above definitions that the m/d complexity of every element of $L_G(B)$ is zero. $L_G(B)$ is a vector space over the field G with respect to the field operator + of H. Since H is also a vector space in this sense, the homomorphism $r : H \rightarrow H'$, where H' is the quotient space $H' = H/L_G(B)$, can be introduced. This construction allows us to state our first theorem of this section.

Theorem 2.1.1 *Let $A = \{h_1, h_2, \cdots, h_n\}$ be an algorithm over B, and let $(h(1), h(2), \cdots, h(s))$ be the m/d steps of A; then $\forall i$ it follows that*

$$r(h_i) \in L_G(r(h(1), r(h(2)), \cdots, r(h(s)).$$

This theorem gives the fundamental result that under the homomorphism r the set of m/d steps spans the space H. This theorem leads to the most important result of this section.

Theorem 2.1.2 *For a given system Z*

$$\mu_B(Z) \geq dim L_G(r(z_1), r(z_2), \cdots, r(z_t)).$$

Theorem 2.1.2 gives a lower bound for the number of m/d steps required for computing a given system Z. If an algorithm A is found which computes the system in $\alpha = dim L_G(r(z_1, (z_2), \cdots, r(z_t))$, then $\mu_B(Z) = \alpha$ and algorithm A is a *minimal algorithm* for the system Z. By restricting the type of algorithms considered to those of bilinear form, the existence of a minimal algorithm for computing a system Z which does not require division can be guaranteed. Below, bilinear systems are introduced.

A system of *bilinear forms* is an expression of the form

$$f_k = \sum_{j=1}^{s} \sum_{i=1}^{r} g_{ijk} x_i y_j \qquad k = 1, 2, \cdots, t$$

where the sets $\{x_i\}$ and $\{y_j\}$ are sets of indeterminates and $g_{ijk} \in G$. This system can be thought of as the matrix product $f_t = (\Phi) y_s$ where f_t is a t point vector, y_s is an s point vector, and the matrix (Φ) is a t by s matrix whose entries ϕ_{jk} are of the form $\phi_{jk} = \sum_{i=1}^{r} g_{ijk} x_i$. Through an

appropriate selection of the coefficients $g_{ijk} \in G$ one can show that linear, cyclic convolution, and finite impulse response filtering are all systems of bilinear forms. This makes the study of bilinear systems very important.

In studying bilinear systems it will be advantageous to introduce the concept of a *non-commutative algorithm* . This is an algorithm A that computes the system Z without using any of the commutative laws of multiplication. Non-commutative algorithms are useful in dealing with non-commutative systems of bilinear forms such as matrix multiplication.

Definition 2.1.4 *A non-commutative algorithm is one that does not depend on the commutative law of multiplication. Denote the minimum number of multiplications required by any non-commutative algorithm to compute the system S of bilinear forms by $\bar{\mu}(S)$.*

The following theorem gives bounds on the non-commutative complexity of a bilinear system S in terms of its commutative complexity.

Theorem 2.1.3 *For every system S of bilinear forms*

$$\mu(S) \leq \bar{\mu}(S) \leq 2\mu(S).$$

Our first major result regarding minimal algorithms for systems of bilinear forms can now be presented.

Theorem 2.1.4 *Let S be a system of bilinear forms. If G has infinitely many elements, then a minimal non-commutative algorithm A computing S exists such that each m/d step of A is of the form $h(i) = M_i(x)N_i(y)$, where $M_i(x)$ and $N_i(y)$ are linear in x and y respectively.*

This theorem implies that a minimal non-commutative algorithm for computing a system of bilinear forms exists which does not require division. In addition, none of the multiplications depend on any other multiplications. This reveals that the multiplication steps can be done in any order.

An important concept which results from restricting the system Z to one of bilinear forms S is the tensor T of the system. The notion is introduced by considering the system of bilinear forms

$$f_k = \sum_{i=1}^{r} \sum_{j=1}^{s} a_{ijk} x_i y_j, k = 1, 2, \cdots, t. \tag{2.22}$$

and let T be the trilinear form or *tensor* of the system f_k defined as

$$T = \sum_{k=1}^{t} f_k z_k = \sum_{k=1}^{t} \sum_{i=1}^{r} \sum_{j=1}^{s} a_{ijk} x_i y_j z_k \tag{2.23}$$

Suppose now that the minimal algorithm for the system of bilinear forms has n m/d steps m_l. By Theorem 2.1.1

$$f_k = \sum_{l=1}^{n} \gamma_{kl} m_l \tag{2.24}$$

Further by Theorem 2.1.4

$$m_l = \left(\sum_{i=1}^{r} \eta_{il} x_i \right) \left(\sum_{j=1}^{s} \beta_{jl} y_j \right) \tag{2.25}$$

Substitution of 2.25 into 2.24 yields

$$f_k = \sum_{l=1}^{n} \gamma_{kl} \left(\sum_{i=1}^{r} \eta_{il} x_i \right) \left(\sum_{j=1}^{s} \beta_{jl} y_j \right) \tag{2.26}$$

Forming the tensor as above and equating expressions 2.26 and 2.23 gives us the identity

$$\sum_{k=1}^{t} \sum_{i=1}^{r} \sum_{j=1}^{s} a_{ijk} x_i y_j z_k = \sum_{l=1}^{n} \left(\sum_{k=1}^{t} \gamma_{kl} z_k \right) \left(\sum_{i=1}^{r} \eta_{il} x_i \right) \left(\sum_{j=1}^{s} \beta_{jl} y_j \right) \tag{2.27}$$

By equating coefficients a variety of different bilinear systems is obtained. Hence, the introduction of the tensor facilitates the construction of several bilinear systems given a single bilinear system. The m/d complexity of the resulting systems are related as shown by the following theorem.

Theorem 2.1.5 *Let S_1 be the system of bilinear forms*

$$S_1 = \sum_{i=1}^{r} \sum_{j=1}^{s} a_{ijk} x_i y_j, \quad k = 1, 2, \cdots, t.$$

Construct from S_1 the following five systems of bilinear forms:

$$S_2 = \sum_{j=1}^{s}\sum_{k=1}^{t} a_{ijk}x_j y_k, \quad i = 1, 2, \cdots, r.$$

$$S_3 = \sum_{j=1}^{r}\sum_{k=1}^{t} a_{ijk}x_i y_k, \quad j = 1, 2, \cdots, t.$$

$$S_4 = \sum_{i=1}^{r}\sum_{j=1}^{s} a_{ijk}x_j y_i, \quad k = 1, 2, \cdots, t.$$

$$S_5 = \sum_{j=1}^{s}\sum_{k=1}^{t} a_{ijk}x_k y_j, \quad i = 1, 2, \cdots, r.$$

$$S_6 = \sum_{j=1}^{r}\sum_{k=1}^{t} a_{ijk}x_k y_i, \quad j = 1, 2, \cdots, s.$$

Then

$$\bar{\mu}(S_1) = \bar{\mu}(S_2) = \bar{\mu}(S_3) = \bar{\mu}(S_4) = \bar{\mu}(S_5) = \bar{\mu}(S_6)$$

Moreover, given a non-commutative algorithm with n m/d steps for one of these systems, it is possible to construct a non-commutative algorithm for any of the other systems with n m/d steps.

Given a system of bilinear forms and a minimal algorithm which computes it, theorem 2.1.5 gives a way of constructing minimal algorithms for other systems of bilinear forms.

A special type of bilinear system of particular importance is the *linear system* . A linear system is obtained from a bilinear one by fixing one of the sets of indeterminates to constant values. This type of system comes up in convolution and FIR filters when one of the sets of indeterminates is fixed and known prior to compute time. This type of system also comes up when modeling the discrete Fourier transform. In dealing with linear systems it is useful to introduce another field $G \subseteq F$. For a set of known constants $\{k_0, k_1, \cdots, k_{n-1}\}$, define F to be an extension field of the set $\{k_i\}$ over the ground field G. Every element of this field is of the form

$$\sum_r \sum_s \cdots \sum_t g_{r,s,\cdots,t} k_0^r k_1^s \cdots k_{n-1}^t$$

where $g_{r,s,\cdots,t} \in G$.

Since the elements of the set $\{k_i\}$ are known prior to the computation, it is possible to think of the elements of the field F as being the field of all possible things that can be pre-computed using G, the set $\{k_i\}$, and the field operations of H. Theorem 2.1.2 can now be redeveloped for systems of linear forms.

Let the system Z be a system of linear forms as described above. Then the system $Z = \{z_0, z_1, z_2, \cdots, z_{t-1}\}$ can be realized from

$$B = F \cup \{y_0, y_1, \cdots, y_{i-1}\}$$

by considering the matrix multiplication $z_k = (\Phi)y_s$ where Φ is a t by i matrix over the field F. Since $G^n \subseteq F^n$ is a vector space, introduce the homomorphism $r : F^n \mapsto F^n/G^n$. Let $\rho_r(\Phi)$ denote the row rank of the matrix (Φ) under the homomorphism r and $\rho_c(\Phi)$ denote the column rank of (Φ) under r. Then, theorem 2.1.2 becomes

Theorem 2.1.6 *Let $Z = (\Phi)y$ be a linear system and let $\rho_r(\Phi)$ be its row rank under r, then $\mu_B(\Phi y) \geq \rho_r(\Phi)$.*

similarly it can be shown that

Theorem 2.1.7 $\mu_B(\Phi y) \geq \rho_c(\Phi)$.

By using these results it will be possible, in subsequent chapters, to place lower bounds on the m/d complexity of convolution, FIR filtering, and DFT. Further, it will be possible to find corresponding minimal algorithms for these computations.

2.1.4 Basic Scheduling Theory

The basic scheduling problem is stated as follows: Let R be a set of resources and f be some desired computation which we call a function. The function f is evaluated by a sequence of subsets of R. Suppose f is evaluated many times, then choose an initiation strategy for the functions that makes optimal use of the resources. A *reservation table* or space-time diagram is an important tool for depicting scheduling problems. In this respect a reservation table is a graph which depicts resource usage versus time. A mark at the intersection of R_1 and time t_1 indicates that resource R_1 is

Resource 1	x	x				
Resource 2			x			
Resource 3				x	x	x
Time	t_1	t_2	t_3	t_4	t_5	t_6

Table 2.1: Forbidden list = {1,2}

active at time t_1. As an example consider the reservation table depicted by table 2.1.

This reservation table indicates that resource one is active for the first two time units, resource two for the third, and resource three for the final three time units. With respect to reservation tables it is possible to introduce the following important definitions.

initiation: The start of one function evaluation.

latency: The number of time units between two successive initiations.

collision: This occurs when two initiations try to access the same stage at the same time.

loading strategy: The sequence of initiations loaded into a pipeline. (Good loading strategies minimize latency while avoiding collisions).

forbidden list: A set of latencies which will result in a collision for a particular reservation table (the forbidden list for reservation table two is given in the caption).

Using these definitions it is possible to state three important results due to Shar [13].

Lemma 2.1.1 *For any statically configured reservation table, the minimal achievable latency (MAL) is always greater or equal to the maximum number of marks in any single row.*

Lemma 2.1.2 *A constant loading strategy (that is loading a new problem every n time units) will not result in a collision if and only if $n * k$, where k is any integer, is not a member of the reservation table's forbidden list.*

Theorem 2.1.8 *For any statically configured reservation table, it is always possible to insert delay elements in order to obtain a constant loading strategy which achieves MAL.*

These definitions and results will be used extensively in chapter four.

2.1.5 The Implementation of Tensor Matrices

In section 2.2.2 the tensor product was defined and some important properties were established. One of the things that makes working with the tensor product so convenient when designing fast algorithms is the fact that there is a strong connection between the structure of a given tensor matrix and the feasibility of efficient implementations of it on a given computer architecture. In this section this idea is made explicit by associating special types of tensor matrices with particular computer organizations.

The first matrix considered is obtained from expression (2.8) by letting A_{n_1,n_2} be the identity matrix of order s and B_{m_1,m_2} be a square matrix of dimension r. The resulting matrix C is block diagonal of dimension $n = rs$, and its structure is given by

$$C_n = (I_s \otimes B_r) = \begin{pmatrix} B_r & & & & \\ & B_r & & & \\ & & B_r & & \\ & & & \ddots & \\ & & & & B_r \end{pmatrix}$$

Suppose that C_n is part of some DSP algorithm and that it is necessary to compute the matrix product $\bar{y_n} = (C_n)\bar{x_n}$ where $\bar{x_n}$ is a vector of input data. Given the very special structure of C_n it should be clear that a direct multiplication of C_n and $\bar{x_n}$ would not be particularly efficient. It would be better to exploit the symmetries of $(I_s \otimes B_r)$ by splitting the vector $\bar{x_n}$ into s sub-vectors $\bar{x_i}$ of length r and performing the matrix product $(B_r)\bar{x_i}$ on each of these.

As a specific example of this let I_3 be the identity matrix of order 3 and B_2 be an arbitrary 2 by 2 matrix, i.e.,

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad B_2 = \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix}$$

then,

$$\bar{y_6} = (C_6)\bar{x_6} = (I_3 \otimes B_2)\bar{x_6}$$

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} & & & & \\ b_{1,0} & b_{1,1} & & & & \\ & & b_{0,0} & b_{0,1} & & \\ & & b_{1,0} & b_{1,1} & & \\ & & & & b_{0,0} & b_{0,1} \\ & & & & b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

This matrix product can be realized by splitting the input vector $\bar{x_6}$ into 3 sub-vectors of length 2 and performing the matrix product $(B_2)\bar{x_2}$ on each of these.

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \qquad \begin{pmatrix} y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

$$\begin{pmatrix} y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}$$

In terms of an actual program the structure of $(I_s \otimes B_r)$ leads quite naturally to the idea of looping. Given a routine to compute the matrix B_r, C_n can be implemented by looping through B_r s times; each time the data pointer is incremented by r. Conventional pseudo-code for realizing this stage on a serial computer might thus look like:

```
for(i=0 to s-1)
  Br(x+ir);
```

Matrices of the form $C_n = (I_s \otimes B_r)$ also lead themselves to efficient implementation on more elaborate architectures. Consider a SIMD multiprocessor for which identical processors obtain instructions from a single instruction stream and perform operations on multiple data streams. If such a computer has s processors there is a very direct mapping between $(I_s \otimes B_r)$ and this architectural configuration. This mapping is depicted by figure 2.1. As is shown, each of the s processors gets instructions to compute B_r from a single instruction stream and operates on a different part

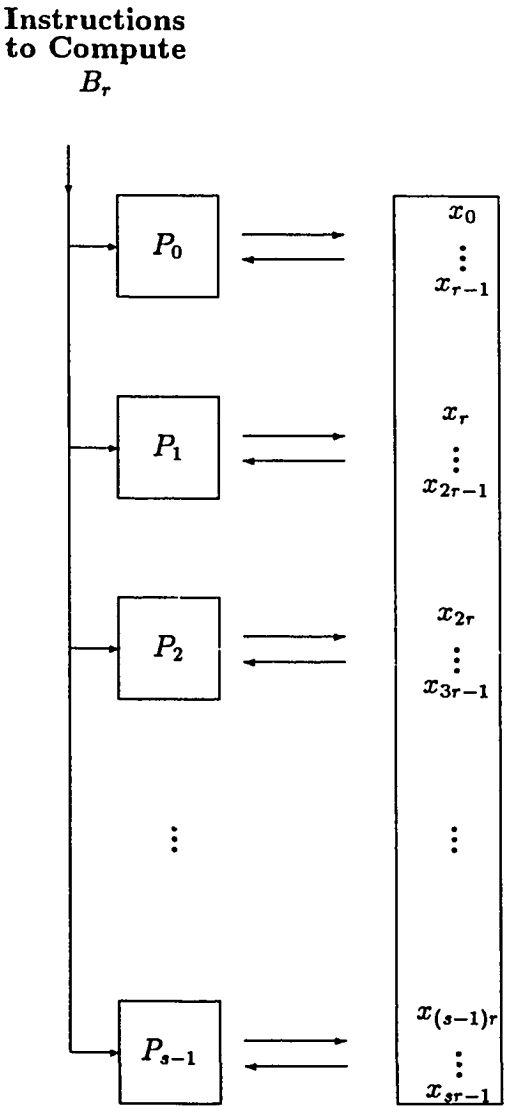

Figure 2.1: The mapping between a tensor matrix of the from $(I_s \otimes B_r)$ and an s processor SIMD computer.

of the input data. Because there is such an explicit connection between this type of multi-processor configuration and the stage $(I_s \otimes B_r)$, tensor matrices of this form are called *parallel stages*.

The second special form that will be considered is obtained by letting B_{m_1,m_2} be the identity matrix of order s and A_{n_1,n_2} be a square matrix of dimension r. The definition of the tensor product reveals that C is now given by

$$C_n = \begin{pmatrix} a_{0,0}I_s & a_{0,1}I_s & \cdots & a_{0,r-1}I_s \\ a_{1,0}I_s & a_{1,1}I_s & \cdots & a_{1,r-1}I_s \\ \vdots & \vdots & \ddots & \vdots \\ a_{r-1,0}I_s & a_{r-1,1}I_s & \cdots & a_{r-1,r-1}I_s \end{pmatrix}$$

As before assume that C_n is part of a DSP algorithm and that it is necessary to realize an efficient implementation of the product $\bar{y_n} = (C_n)\bar{x_n}$. The tensor product again reveals symmetries which are useful. In this case the stage $\bar{y_n} = (A_r \otimes I_s)\bar{x_n}$ can be realized by splitting the vector $\bar{x_n}$ into r sub-vectors of length s and performing a single A_r operation on these entire sub-vectors.

As an example let A_r be an arbitrary 2 by 2 matrix and let I_s be the identity matrix of order 3. Then

$$\bar{y_6} = (C_6)\bar{x_6} = (A_2 \otimes I_3)\bar{x_6}$$

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} a_{0,0} & & & a_{0,1} & & \\ & a_{0,0} & & & a_{0,1} & \\ & & a_{0,0} & & & a_{0,1} \\ a_{1,0} & & & a_{1,1} & & \\ & a_{1,0} & & & a_{1,1} & \\ & & a_{1,0} & & & a_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

This matrix product can be computed by splitting the vector $\bar{x_6}$ into 2 sub-vectors of length 3 and performing a single A_2 on these entire sub-vectors.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = a_{0,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{0,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

$$\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} = a_{1,0} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + a_{1,1} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

This is precisely the type of operation performed by a vector processor. Thus in the same sense that $(I_s \otimes B_r)$ maps to a parallel configuration, $(A_r \otimes I_s)$ maps to a vector configuration. For this reason matrices of the form $(A_r \otimes I_s)$ are called *vector* stages.

By choosing special forms of the tensor product it has been shown that there is a direct connection between the structure of a tensor matrix and the feasibility of mapping it to important architectural configurations. Two basic forms of tensor matrices were identified, the *parallel form* and the *vector form*. A fundamental relationship between these two tensor forms will be established in the next section.

Addressing and the Tensor Product

Many DSP algorithms require some type of data shuffling operation. The cost of these are often excluded from the operations count of a given algorithm in spite of the fact that they can consume a significant proportion of the total run time. One way to greatly reduce, and in some cases eliminate, these effects is to consider the data shuffle an addressing operation. In doing so data is never actually moved; instead the shuffle is effected when operands are retrieved from memory. This is only feasible when the data shuffles are "matched" to the target computer's addressing capabilities. In this section the theory of the tensor product is broadened to provide mechanisms for dealing with these issues. At the heart of this development is the notion of the *permutation* matrix.

Permutation matrices are a very natural way of representing data movement. These are square matrices of zeros and ones where each row and each column has precisely a single 1 entry. When acting on a vector of input data a permutation matrix simply shuffles the original data.

We will be focusing on one particular type of permutation matrix which shuffles data in a very special way. These are called *stride by s* permutation matrices. Let $\bar{x_n}$ be an n point vector of data, $n = rs$, and $P_{n,s}$ denote the stride by s permutation matrix. The shuffling action of $\bar{y_n} = (P_{n,s})\bar{x_n}$ on $\bar{x_n}$ is as follows:

The first r elements of $\bar{y_n}$ are obtained by starting at the element x_0 and selecting each s^{th} element of $\bar{x_n}$, namely $x_0, x_s, x_{2s}, \ldots, x_{(r-1)s}$. The next r elements of $\bar{y_n}$ are obtained by the same procedure but this time the process starts at x_1. The next r elements of $\bar{y_n}$ are thus $x_1, x_{1+s}, x_{1+2s}, \ldots, x_{1+(r-1)s}$. This process is repeated s times or until the vector $\bar{y_n}$ is filled. Thus the last r elements of $\bar{y_n}$ are $x_{s-1}, x_{2s-1}, x_{3s-1}, \ldots, x_{rs-1}$.

Let's take a look at some examples of stride permutations and their action on input data.

$$\bar{y} = (P_{4,2})\bar{x} = \begin{pmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

$$\bar{y} = (P_{6,2})\bar{x} = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

$$\bar{y} = (P_{8,4})\bar{x} = \begin{pmatrix} x_0 \\ x_4 \\ x_1 \\ x_5 \\ x_2 \\ x_6 \\ x_3 \\ x_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}$$

The shuffling operation of stride permutations can be easily implemented as an addressing operation on many computers. In these cases stride permutations can be realized at virtually no cost in compute cycles. Thus, expressing a data shuffle as a stride permutation or a simple combination of stride permutations greatly enhances the run time performance of

a DSP algorithm. (We note here that the infamous bit reversal shuffle of the FFT algorithm can be decomposed in this way. Thus, a tensor product implementation of the FFT algorithm reduces the bit reversal to a data addressing operation which is realized at virtually no cost of compute cycles. In other implementations the bit reversal can consume up to 30% of the entire processing time!)

As powerful as the stride permutation matrix is at representing data movement, recall that it plays an even more critical role in the theory of tensor matrices. Its function is associated with the *commutative property* of the tensor product which was given as formula (2.16). Let A_s and B_r be any square matrices of order s and r, respectively, and let $n = rs$. Then

$$(A_s \otimes B_r) = P_{n,s}\,(B_r \otimes A_s)\,P_{n,r} \tag{2.28}$$

This rather innocent looking result is at the heart of the reason the tensor product is a powerful tool for modeling and designing algorithms. By the commutative theorem a parallel stage can be written as

$$(I_s \otimes A_r) = P_{n,s}\,(A_r \otimes I_s)\,P_{n,r} \tag{2.29}$$

Recall that the term $(A_r \otimes I_s)$ in this expression was identified as a vector stage. The commutative theorem thus provides a mechanism for modifying the structure of a computational stage for optimized implementation on either a parallel processor, vector processor, or even a serial computer. This is done through the introduction of stride permutations which can be realized as addressing operations and hence implemented at virtually no cost.

Design Examples

The tensor product has been defined, the connection between tensor matrices and important computer organizations established, and the commutative theorem introduced as a method to modify the structure of a tensor matrix. In this section the power of the tensor product as a tool for modeling and designing DSP algorithms is now demonstrated.

The multiplication of an arbitrary tensor matrix of the form $C_n = (A_r \otimes B_s)$ and a vector of input data will be carried out on several radically different types of architectures.

Particular features of the underlying architecture will dictate how to use the inherent algebraic structure of the tensor product to decompose C_n efficiently. The relationships already established permit us to consider these matrix factorizations as actual algorithms for carrying out the computation of C_n. In order to make this connection clearer the tensor product algorithms designed will be expressed in terms of more "conventional looking" pseudo-code. Keep in mind, however, that this is done only as an aid to the reader and that the algorithms are fully characterized by their tensor product decomposition.

Problem 1: An important operation on a set of n inputs is the matrix product $\bar{y_n} = (C_n)\bar{x_n}$. It is necessary to realize an efficient implementation of this operation on a serial computer. The matrix C_n can be expressed as a tensor product of two smaller matrices (A_s) and (B_r). There already exist subroutines to compute these sub-matrices. Design the algorithm.

Solution :

$$\bar{y_n} = (C_n)\bar{x_n} = (A_s \otimes B_r)\bar{x_n} \tag{2.30}$$

Note that

$$(A_s \otimes B_r) = (A_s I_s \otimes I_r B_r) \tag{2.31}$$

where I_s and I_r are identity matrices of order s and r respectively. Application of the distributive property yields

$$(A_s I_s \otimes I_r B_r) = (A_s \otimes I_r)(I_s \otimes B_r) \tag{2.32}$$

The implementation of $(I_s \otimes B_r)$ on a serial computer was already discussed. The commutative property may be used to change the structure of the second stage, $(A_s \otimes I_r)$, so it is also of this form. In particular

$$(A_s \otimes I_r) = P_{n,s}(I_r \otimes A_s)P_{n,r} \tag{2.33}$$

The matrices $P_{n,s}$ and $P_{n,r}$ are stride permutations as defined in section 2.1.2 and can be implemented as addressing operations. Putting these two pieces together yields algorithm 1:

$$\bar{y_n} = (C_n)\bar{x_n} = [P_{n,s}(I_r \otimes A_s)P_{n,r}](I_s \otimes B_r)\bar{x_n} \tag{2.34}$$

This tensor product factorization of C_n is an efficient algorithm for computing C_n. In terms of conventional pseudo-code this algorithm can represented as

```
for(i=0 to s-1)
  Br(input, output, i);
for(i=0 to r-1)
  As(input, output, i, r, s);
```

In order to see the efficiency of the approach we compare the performance of algorithm 1 and a direct implementation of C_n. Since the addressing of operands is specified by stride permutations we can assume that they can be computed while performing data operations. Thus, the performance of the algorithm is specified by the operations count. If C_n were computed directly we would require n^2 multiplications and $n(n-1)$ additions for a total operation count of $2n^2 - n$. If algorithm 1 is used to realize the multiplication of C_n with an input vector $\bar{x_n}$ we require $sr^2 + rs^2$ multiplications and $sr(r-1) + rs(s-1)$ additions for a total of $s(2r^2 - r) + r(2s^2 - s)$. If $r = s = 10$ a direct implementation would require 19,990 operations, while algorithm 1 would require 3,800 operations.

Problem 2: Let C_n be the same matrix as described in problem one. This time it is necessary to implement $\bar{y} = (C)\bar{x}$ on a vector processor. Design an algorithm which makes use of the underlying architectural configuration.

Solution :

$$\bar{y_n} = (C_n)\bar{x_n} = (A_s \otimes B_r)\bar{x_n} \tag{2.35}$$

By the distributive property it was shown that

$$C_n = (A_s \otimes I_r)(I_s \otimes B_r) \tag{2.36}$$

$(A_s \otimes I_r)$ is a vector stage and can easily be implemented on a vector processor. The stage $(I_s \otimes E_r)$ can also be vectorized by using the

$$\bar{y_n} = (C_n)\bar{x_n} = [P_{n,s}(I_r \otimes A_s)P_{n,r}](I_s \otimes B_r)\bar{x_n}$$

$$[P_{n,s}(I_r \otimes A_s)P_{n,r}] \Leftrightarrow \left[(P_{n,s}) \begin{pmatrix} A_s & & & & \\ & A_s & & & \\ & & A_s & & \\ & & & \ddots & \\ & & & & A_s \end{pmatrix} (P_{n,r}) \right]$$

$$(I_s \otimes B_r) \Leftrightarrow \begin{bmatrix} B_r & & & & \\ & B_r & & & \\ & & B_r & & \\ & & & \ddots & \\ & & & & B_r \end{bmatrix}$$

$$\Updownarrow$$

```
for(i=0 to s-1)
  Br(input, output, i);
for(i=0 to r-1)
  As(input, output, i, r, s);
```

Figure 2.2: The relationship between algorithm 1, its matrix definition, and pseudo-code for implementation on a serial computer.

commutative theorem. In particular

$$(I_s \otimes B_r) = P_{n,s}(B_r \otimes I_s)P_{n,r} \tag{2.37}$$

The substitution of this into (2.36) yields algorithm 2:

$$\bar{y_n} = (C_n)\bar{x_n} = [(A_s \otimes I_r)P_{n,s}][(B_r \otimes I_s)P_{n,r}]\bar{x_n} \tag{2.38}$$

Conventional pseudo-code for this algorithm might look like

```
VSIZE = S                  /* Set vector size to s          */
LOADSTRIDE X1,VI,R         /* Load each vector register, Vi,
*/
                           /* with a stride of r.           */
START BR                   /* Start Br operating with vectors*/
   .
   .
   .
END BR                     /* End Br                        */

VSIZE = R                  /* Repeat above for As           */
LOADSTRIDE X2,VI,S
START AS
   .
   .
   .
END AS
```

Problem 3: Let C_n be a square matrix of order 2500 which can be expressed as a tensor product of two matrices $C_{2500} = (A_{50} \otimes B_{50})$. Design an algorithm for the efficient implementation of $\bar{y} = (C)\bar{x}$ on a SIMD multi-processor which has 5 processors. Assume that subroutines to compute both A and B exist.

Solution :

As in the previous examples begin by decomposing C_{2500} into a vector and parallel stage.

$$C_{2500} = (A_{50} \otimes I_{50})(I_{50} \otimes B_{50}) \tag{2.39}$$

Next, apply the commutative theorem to get

$$C_{2500} = P_{2500,50}(I_{50} \otimes A_{50})P_{2500,50}(I_{50} \otimes B_{50}) \tag{2.40}$$

Partition the computation to take full advantage of each of the 5 processors. In terms of tensor products this is done by making use of the fact that

$$I_{50} = (I_5 \otimes I_{10}) \tag{2.41}$$

The substitution of this into (2.40) yields algorithm 3:

$$\bar{y} = (C_{2500})\bar{x} = [P_{2500,50}(I_5 \otimes \hat{A})][P_{2500,50}(I_5 \otimes \hat{B})]\bar{x} \tag{2.42}$$

$$\hat{A} = (I_{10} \otimes A_{50}) \tag{2.43}$$

$$\hat{B} = (I_{10} \otimes B_{50}) \tag{2.44}$$

In this algorithm each of the 5 processors operates on a different part of the data and performs exactly the same operations: either $\hat{A}$ or $\hat{B}$. The mapping between (2.44) and our 5 processor SIMD machine was already made explicit. Furthermore, since each processor is a serial machine, $\hat{A}$ and $\hat{B}$ are implemented as illustrated in problem 1. In terms of conventional pseudo-code algorithm 3 is given by

```
ENBL (ALL);          /* Enable all processors              */
SEND (B);            /* Broadcast the code B to each node */
for (i=0 to 9)       /* The implementation of B_hat        */
  GO (B, i);

SEND (A);            /* Broadcast the code A to each node */
for (i=0 to 9)       /* The implementation of A_hat        */
  GO (A, i);
```

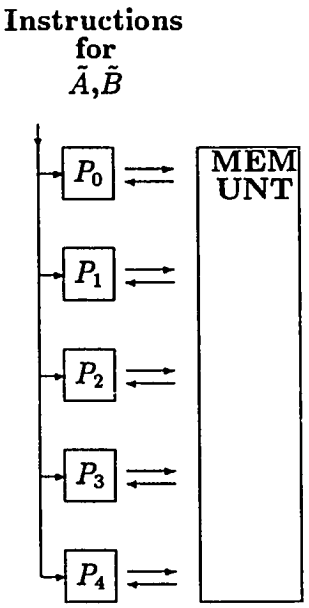

Figure 2.3: The implementation of algorithm 3 on a five processor SIMD machine. Let $t(M)$ denote the time required to multiply a matrix M and an input vector. Then the time required for this algorithm is $t(C_n) = t(\hat{A}) + t(\hat{B}) = 10 * t(A) + 10 * t(B) = 10 * (t(A) + t(B))$.

Problem 4: Five more processors have been added to the architecture of problem 3. Redesign the algorithm to exploit this fact.

Solution :

Proceed exactly as before except instead of (2.41) use

$$I_{50} = (I_{10} \otimes I_5) \tag{2.45}$$

The algorithm then becomes algorithm 4:

$$C_{2500} = [P_{2500,50}(I_{10} \otimes \hat{A})][P_{2500,50}(I_{10} \otimes \hat{B})] \tag{2.46}$$

$$\hat{A} = (I_5 \otimes A_{50}) \tag{2.47}$$

$$\hat{B} = (I_5 \otimes B_{50}) \tag{2.48}$$

In this case the mapping to pseudo-code is the same as for algorithm 3. Note that doubling the number of processors reduced the processing time by one-half (see fig 2.4).

Problem 5: It is necessary to implement the matrix of problem 3 on a digital signal processing chip. An important feature of this chip is its instruction cache. The cache is too small to hold all of the instructions which constitute A_{50} or B_{50}. It is possible, however, to express A_{50} and B_{50} as $A_{50} = (A_5^1 \otimes A_{10}^2)$ and $B_{50} = (B_5^1 \otimes B_{10}^2)$ where these sub-blocks are small enough to fit into the cache. Design an algorithm which exploits the cache memory.

Solution :

Start from expression (2.40) of problem 3. This was

$$C_{2500} = P_{2500,50}(I_{50} \otimes A_{50})P_{2500,50}(I_{50} \otimes B_{50}) \tag{2.49}$$

Next use that fact that

$$A_{50} = (A_5^1 \otimes A_{10}^2) = (I_5 \otimes A_{10}^2)P_{50,5}(I_{10} \otimes A_5^1)P_{50,10} \tag{2.50}$$

$$B_{50} = (B_5^1 \otimes B_{10}^2) = (I_5 \otimes B_{10}^2)P_{50,5}(I_{10} \otimes B_5^1)P_{50,10} \tag{2.51}$$

The substitution of these into expression (2.49) and the use of the associative and distributive property of the tensor product yields algorithm 5:

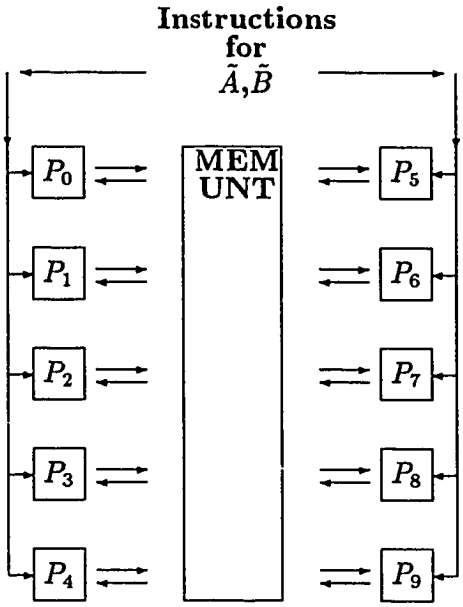

Figure 2.4: The implementation of algorithm 4 on a ten processor SIMD machine. Let $t(M)$ denote the time required to multiply a matrix M and an input vector. Then the time required for this algorithm is $t(C_n) = t(\hat{A}) + t(\hat{B}) = 5 * t(A) + 5 * t(B) = 5 * (t(A) + t(B))$.

$$C_{2500} = [P_{2500,50}(I_{250} \otimes A_{10}^2)][(I_{50} \otimes P_{50,5})] \quad (2.52)$$
$$[(I_{500} \otimes A_5^1)(I_{50} \otimes P_{50,10})][P_{2500,50}(I_{250} \otimes B_{10}^2)(I_{50} \otimes P_{50,5})]$$
$$[(I_{500} \otimes B_5^1)(I_{50} \otimes P_{50,10})]$$

In this algorithm we are looping through blocks of instructions that are small enough to fit into the cache memory. Again the stride permutations are implemented as addressing instructions and thus carried out at no cost. Conventional pseudo-code for this algorithm might look like

```
LDC B_5^1              /* Load the cache with B_5^1      */
FRZC                   /* Freeze cache memory            */
for (i=0 to 499)       /* By looping the contents of the*/
  B_5^1();             /* cache, compute first stage     */

LDC B_10^2             /* Load the cache with B_10^2     */
FRZC                   /* Freeze cache memory            */
for (i=0 to 249)       /* By looping the contents of the*/
  B_10^2();            /* cache, compute second stage    */

LDC A_5^1              /* Load the cache with A_5^1      */
FRZC                   /* Freeze cache memory            */
for (i=0 to 499)       /* By looping the contents of the*/
  A_5^1();             /* cache, compute third stage     */

LDC A_10^2             /* Load the cache with A_10^2     */
FRZC                   /* Freeze cache memory            */
for (i=0 to 249)       /* By looping the contents of the*/
  A_10^2();            /* cache, compute fourth stage    */
```

2.1.6 Some Important Theorems

This subsection is basically a list of important theorems which will be referenced in future chapters. Since they are elementary results, found in any text of algebra, they are listed here without proof.

Theorem 2.1.9 (Division Algorithm) *For every integer c and positive integer d, there is a unique pair of integers Q and s, such that $c = dQ + s$, where $0 \leq s < d$.*

Theorem 2.1.10 *For any integers s and t there exists integers a and b such that the greatest common divisor $GCD(s,t) = as + bt$*

Theorem 2.1.11 (Division Algorithm for Polynomials) *For every polynomial $c(x)$ and nonzero polynomial $d(x)$, there is a unique pair of polynomials $Q(x)$ and $s(x)$, such that $c(x) = d(x)Q(x) + s(x)$, where deg $s(x) <$ deg $d(x)$.*

Theorem 2.1.12 *A polynomial over a field has a unique factorization into a field element times a product of prime polynomials over the field, each polynomial being of degree at least one.*

Theorem 2.1.13 *For any polynomials $s(x)$ and $t(x)$ there exists polynomials $a(x)$ and $b(x)$ such that the greatest common divisor $GCD[s(x),t(x)] = a(x)s(x) + b(x)t(x)$*

Theorem 2.1.14 (Lagrange Interpolation) *Let $B_0, B_1, \cdots, B_n$ be a set of $n+1$ distinct points, and let $p(B_k)$ for $k = 0, \cdots, n$ be given. There is exactly one polynomial $p(x)$ of degree n or less that has value $p(B_k)$ when evaluated at B_k for $k = 0, 1, \cdots, n$. It is given by*

$$p(x) = \sum_{i=0}^{n} p(B_i) \frac{\prod_{j \neq i}(x - B_j)}{\prod_{j \neq i}(B_i - B_j)}$$

Theorem 2.1.15 (Chinese Remainder for Integers) *Let $M = \prod_{r=0}^{k} m_r$ be product of pair-wise relatively prime integers; let $M_i = M/m_i$ for each i, let N_i satisfy $N_i M_i + n_i m_i = 1$. Then the system of congruences*

$$c_i = c \bmod m_i \qquad i = 0, \cdots, k$$

is uniquely solved by

$$c = \sum_{i=0}^{k} c_i N_i M_i \pmod{M}$$

Theorem 2.1.16 (Chinese Remainder for Polynomials) *Let* $M(x) = \prod_{r=0}^{k} m_r(x)$ *be product of relatively prime polynomials; let* $M_i(x) = M(x)/m_i(x)$ *for each* i*, let* $N_i(x)$ *satisfy* $N_i(x)M_i(x)+n_i(x)m_i(x) = 1$*. Then the system of congruences*

$$c_i(x) = c(x) \qquad \bmod m_i(x) \qquad i = 0, \cdots, k$$

is uniquely solved by

$$c(x) = \sum_{i=0}^{k} c_i(x)N_i(x)M_i(x) \qquad (\bmod\ M(x))$$

Theorem 2.1.17 *Let* M_n *denote the group of nonzero integers relatively prime to* n *with group operation multiplication modulo* n*, and let* C_n *denote the cyclic group of order* n*. Then*

$$M_{p^r} \cong C_{(p-1)p^{r-1}}$$

for p *and odd prime, and*

$$M_{2^r} \cong C_2 \times C_{2^{r-2}}$$

Theorem 2.1.18 *Let* C_n *denote the cyclic group of order* n*. Then there exists an element of order* α *for any* α *that divides* n*.*

Theorem 2.1.19 (Convolution Theorem I) *Let* C *be a circulant matrix over some field* F*, that is a matrix of the form*

$$\begin{bmatrix} h_0 & h_{n-1} & h_{n-2} & \cdots & h_1 \\ h_1 & h_0 & h_{n-1} & \cdots & h_2 \\ h_2 & h_1 & h_0 & \cdots & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & h_{n-3} & \cdots & h_0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

and let $\alpha \in F$ *be an element of order* n*. ie.* $\alpha^n = 1$*. Then there exists a factorization of* C *such that*

$$C = W^{-1}DW$$

where D *is a diagonal matrix over* F *given by* $D = diag(Wd)$ *where* d *is the 0-th column of* C*, and the* $\{ij\}^{th}$ *element of* W *is* $w_{ij} = \alpha^{ij} \pmod n$

Theorem 2.1.20 (Convolution Theorem II) *Let C be a skew-circulant matrix over some field F, that is a matrix of the form*

$$\begin{bmatrix} h_0 & h_1 & h_2 & \cdots & h_{n-1} \\ h_1 & h_2 & h_3 & \cdots & h_0 \\ h_2 & h_3 & h_4 & \cdots & h_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_0 & h_1 & \cdots & h_{n-2} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

and let $\alpha \in F$ be an element of order n. ie. $\alpha^n = 1$. Then there exists a factorization of C such that

$$C = WDW$$

where D is a diagonal matrix over F given by $D = diag(W^{-1}d)$ where d is the 0-th column of C, and the $\{ij\}^{th}$ element of W is $w_{ij} = \alpha^{ij} \pmod n$.

2.2 Convolution

The importance of convolution was established in chapter one. In this section, methods of obtaining efficient algorithms for the implementation of both linear and cyclic convolution are presented. These methods will be based primarily on the mathematical material outlined in section 2.1.

This material has been organized into four subsections. In the first, three representations of the convolution operator are given. Having three different ways of expressing convolution is beneficial because it makes it possible to approach the problem of designing fast algorithms in many ways. Using these representations, the multiplicative complexity of both linear and cyclic convolution is determined. This is followed by a presentation of several methods of obtaining fast algorithms for convolution. The presentation ends by introducing number theoretic transforms and discussing algorithms based on these methods.

2.2.1 Three Representations

Given two sequences h_n, of length n, and x_m, of length m, the convolution of h_n and x_m is denoted by the sequence $y_{n+m-1} = h_n * x_m$ whose terms are given by

$$y_z = \sum_k h_{(z-k)} d_k \tag{2.53}$$

This definition of convolution is most familiar to electrical engineers. Although it is very concise, it unfortunately hides a great deal of the inherent symmetry of the computation. Since fast algorithms exploit this underlying symmetry, it stands to reason that the summation form is not one that will be used extensively.

By expanding the summation formula (2.53) the computation can be

written as the system of equations

$$\begin{aligned}
y_0 &= h_0 d_0 \\
y_1 &= h_1 d_0 + h_0 d_1 \\
y_2 &= h_2 d_0 + h_1 d_1 + h_0 d_2 \\
&\vdots \quad \vdots \quad \vdots \\
y_{n-1} &= h_{n-1} d_0 + h_{n-2} d_1 + h_{n-3} d_2 + \cdots h_0 d_{n-1} \\
&\vdots \quad \vdots \quad \vdots \\
y_{n+m-4} &= h_{n-1} d_{m-3} + h_{n-2} d_{m-2} + h_{n-3} d_{m-1} \\
y_{n+m-3} &= h_{n-1} d_{m-2} + h_{n-2} d_{m-1} \\
y_{n+m-2} &= h_{n-1} d_{m-1}
\end{aligned}$$

From these equations, it is easy to see that this system can be expressed as the matrix product

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_{n+m-2} \end{bmatrix} = \begin{bmatrix} h_0 & 0 & \cdots & 0 \\ h_1 & h_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & \cdots & h_0 \\ 0 & h_{n-1} & \cdots & h_1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{n-1} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_{m-1} \end{bmatrix} \tag{2.54}$$

In this representation the symmetry of the computation is much more visible. This makes the matrix representation well suited for use in the design of algorithms.

Another characterization of convolution is obtained by considering the polynomials $h_{n-1}(x) = \sum_{j=0}^{n-1} h_j x^j$ and $d_{m-1}(x) = \sum_{k=0}^{m-1} d_k x^k$ of degrees $n-1$ and $m-1$ respectively. Forming the product of these yields

$$y_{n+m-2}(x) = \sum_{z=0}^{m+n-2} y_z x^z = h_{n-1}(x) d_{m-1}(x) = \sum_{j=0}^{n-1} h_j x^j \sum_{k=0}^{m-1} d_k x^k \tag{2.55}$$

Rearranging the terms, making the substitution $z = j + k$, and equating the coefficients of x;

$$\sum_{z=0}^{m+n-2} y_z x^z = \sum_{k=0}^{m-1} \sum_{j=0}^{n-1} h_j d_k x^{k+j}$$

$$\sum_{z=0}^{m+n-2} y_z x^z = \sum_{z=k}^{n+k-2} \sum_{k=0} h_{z-k} d_k x^z$$

$$y_z = \sum_{k=0} h_{z-k} d_k$$

is obtained, which is the convolution operator as defined in (2.53). Hence, the convolution of two sequences $y_{(m+n-1)} = h_n * d_m$ is the same as the polynomial product $y_{m+n-2}(x) = h_{n-1}(x)d_{m-1}(x)$ where the elements of the sequences are the coefficients of the polynomials.

Cyclic convolution can also be represented as matrix and polynomial products. The cyclic convolution of the sequences h_n and d_n is denoted by $y_n = h_n((*))d_n$ and is defined identically to linear convolution except, in this case, the indices of the sequences are taken *mod* n. The notation $((\ ))_n$ is used to denote arithmetic *mod* n. Using this definition (2.53) becomes

$$y_z = \sum_k h_{((z-k))} d_k \tag{2.56}$$

Proceeding as above, this is expressed as a system of equations

$$\begin{array}{rcccccccccc}
y_0 & = & h_0 d_0 & + & h_{n-1} d_1 & + & h_{n-2} d_2 & + & \cdots & + & h_1 d_{n-1} \\
y_1 & = & h_1 d_0 & + & h_0 d_1 & + & h_{n-1} d_2 & + & \cdots & + & h_2 d_{n-1} \\
y_2 & = & h_2 d_0 & + & h_1 d_1 & + & h_0 d_2 & + & \cdots & + & h_3 d_{n-1} \\
\vdots & & \vdots & & & & \vdots & & & & \\
y_{n-1} & = & h_{n-1} d_0 & + & h_{n-2} d_1 & + & h_{n-3} d_2 & + & \cdots & + & h_0 d_{n-1}
\end{array}$$

which can be represented by the matrix product

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} h_0 & h_{n-1} & h_{n-2} & \cdots & h_1 \\ h_1 & h_0 & h_{n-1} & \cdots & h_2 \\ h_2 & h_1 & h_0 & \cdots & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & h_{n-3} & \cdots & h_0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix} \tag{2.57}$$

In this representation the matrix of expression (2.57) is a circulant matrix as defined by theorem 2.1.19.

Another useful representation is obtained by the change of variables $k = -k$ on (2.56), this yields

$$y_z = \sum_k h_{((z+k))} d_{((-k))} \tag{2.58}$$

In matrix form, this becomes

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} h_0 & h_1 & h_2 & \cdots & h_{n-1} \\ h_1 & h_2 & h_3 & \cdots & h_0 \\ h_2 & h_3 & h_4 & \cdots & h_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_0 & h_1 & \cdots & h_{n-2} \end{bmatrix} \begin{bmatrix} d_0 \\ d_{n-1} \\ d_{n-2} \\ \vdots \\ d_1 \end{bmatrix} \tag{2.59}$$

Now the symmetry of the matrix of expression (2.59) is that of a skew-circulant matrix as defined by theorem 2.1.20. Note, further, that this matrix also has the symmetry of the additive group of Z_n. These observations give other revealing ways of studying the structure of cyclic convolution. In order to derive fast algorithms all these different representations will be used.

2.2.2 Multiplicative Complexity

Linear Convolution

In section 2.1.3 theorems were presented which give lower bounds on the numbers of multiplications required to compute systems of bilinear forms. It was also pointed out that both linear and cyclic convolution are special types of bilinear forms. In this section this association is made explicit by using the matrix representations developed above. Once this is done, the complexity of convolution is determined by deriving algorithms which achieve the lower bounds established.

It has been shown that a system of bilinear forms

$$f_k = \sum_{j=1}^{s} \sum_{i=1}^{r} g_{ijk} x_i y_j \qquad k = 1, 2, \cdots, t$$

could be expressed as the matrix product

$$f_t = (\Phi) y_s \tag{2.60}$$

where each element of (Φ) is of the form $\phi_{ij} = \sum_k g_{ijk} x_k$. In this, $g_{ijk} \in G$ and the sets $\{x_i\}$ and $\{y_j\}$ were sets of indeterminates. Equation (2.59) is a

special case of this form obtained by making $\{h_i\}$ and $\{d_j\}$ indeterminates and by defining g_{ijk} as

$$g_{ijk} = \begin{cases} 1 & i+j=i, j<n \\ 0 & otherwise \end{cases}$$

If one of the sets of indeterminates is known prior to compute time, this association permits the use of theorems 2.1.7 and 2.1.8 which give lower bounds for the multiplicative complexity of (2.54). Since all the columns and rows of (2.54) are linearly independent the use of these theorems yields

$$\mu(Z) \geq \rho_r(Z) \geq \rho_c(Z)$$

$$\mu(Z) \geq n+m-1$$

By finding an algorithm A whose m/d complexity is $\mu(A) = m+n-1$, it is established that the m/d complexity of the system (2.54) is $\mu(Z) = m+n-1$. The algorithm, due to Cook and Toom [9], is based on the Lagrange Interpolation Theorem which is presented as theorem 2.1.14.

Lagrange Interpolation gives a method of uniquely determining a polynomial of degree n or less by knowing it's value at $n+1$ unique points $\{B_i\}$. The relationship between the set $\{B_i\}$ and a polynomial $y(x)$ is given by

$$y(x) = \sum_{i=0}^{n} = y(B_i)y_i(x) \qquad y_i(x) = \frac{\prod_{j\neq i}(x-B_j)}{\prod_{j\neq i}(B_i-B_j)} \qquad (2.61)$$

Based on this formula, the Cook-Toom algorithm for the convolution $y_{m+n-2}(x) = h_{n-1}(x)d_{m-1}(x)$ is now presented.

1. Choose $n+m-1$ unique elements of the ground field G.
2. Evaluate the polynomials $h_{n-1}(x)$ and $d_{m-1}(x)$ at these points. Since $B_i \in G$, the evaluation of $h(B_i)$ and $d(B_i)$ does not require any general multiplication as described in section 2.1.3.
3. Evaluate the polynomial $y(B_i)$ by forming the products $y(B_i) = h(B_i)d(B_i) \quad \forall i$. This step will require $m+n-1$ multiplications.
4. Use the interpolation formula (2.61) in order to determine the coefficients of the polynomial $p(x)$ from $p(B_i)'s$. Again, this step will not require any general multiplications, only additions and multiplications of indeterminates and elements of G.

The algorithm described computes the system $y_{m+n-2}(x) = h_{n-1}(x)d_{m-1}(x)$ $n+m-1$ multiplications. Since $\mu(Z) \geq n+m-1$, the multiplicative complexity of the linear convolution of the sequences h_n and d_m is $n+m-1$. Note that the interpolation formula requires that the set of $\{B_i's\}$ be unique. If the cardinality of the field G is less than $n+m-2$ then it has been shown [5] that $\mu(Z)$ is indeed $> m+n-1$. The algorithm is illustrated below for the product $y_2(x) = h_1(x)d_1(x)$ when G is the field of rationals.

- **Example 2.2.1** Cook-Toom for $y(x) = h(x)d(x)$
  - step 1

$$B_0 = 0 \qquad B_1 = 1 \qquad B_2 = -1$$

  - step 2

$$h(x) = h_1x + h_0 \qquad d(x) = d_1x + d_0$$

$$\begin{array}{lll} h(B_0) = h_0 & & d(B_0) = d_0 \\ h(B_1) = h_1 + h_0 & & d(B_1) = d_1 + d_0 \\ h(B_2) = h_0 - h_1 & & d(B_2) = d_0 - d_1 \end{array}$$

  - step 3

$$\begin{array}{lll} y(B_0) = h(B_0)d(B_0) & = & (h_0)(d_0) \\ y(B_1) = h(B_1)d(B_1) & = & (h_0+h_1)(d_0+d_1) \\ y(B_2) = h(B_2)d(B_2) & = & (h_0-h_1)(d_0-d_1) \end{array}$$

  - step 4

$$\begin{array}{lll} y_0(x) & = & -x^2+1 \\ y_1(x) & = & 1/2(x^2+x) \\ y_2(x) & = & 1/2(x^2-x) \end{array}$$

$$y(x) = y_2x^2 + y_1x + y_0 =$$
$$y(B_0)y_0(x) + y(B_1)y_1(x) + y(B_2)y_2(x)$$

Equating the coefficients of x yields

$$\begin{array}{lll} y_2 & = & 1/2y(B_1) + 1/2y(B_2) - y(B_0) \\ y_1 & = & 1/2y(B_1) - 1/2y(B_2) \\ y_0 & = & y(B_0) \end{array}$$

□

Although the Cook-Toom algorithm is very useful in determining the complexity of linear convolution, it is not a very good choice in terms of implementations. First, the number of additions required grows very quickly, even for a modest size convolution. Second, (for large convolutions) this algorithm is very susceptible to numerical instabilities.

Cyclic Convolution

The determination of the multiplicative complexity of cyclic convolution is not as straight forward as for linear convolution. The reason for this is that although theorem 2.1.2 can be used to determine a lower bound for the m/d complexity, a general algorithm which achieves this bound is not available. Fortunately, Winograd has worked out this problem in detail and has proved two theorems which solve the more general problem of determining the m/d complexity of modulo polynomial multiplication. The proofs are rather lengthy and will not be reproduced, the interested reader is referred to reference [36].

The first theorem determines the m/d complexity of polynomial multiplication mod an irreducible polynomial to some power. The second theorem extends the result by determining the m/d complexity of polynomial multiplication mod an arbitrary polynomial. The m/d complexity of cyclic convolution is then determined as a corollary of the second theorem.

Theorem 2.2.1 *Let $Q(u)$ be a monic irreducible polynomial over the field G and let n be the degree of $P(u) = Q^k(u)$. If G contains at least $2n-2$ distinct elements then $\mu_B(S;G) = \bar{\mu_B}(S;G) = 2n-1$.*

Theorem 2.2.2 *Let $Q_i(u), i = 1,2,\cdots,k$, be distinct monic irreducible polynomials over the field G and let n be the degree of $P(u) = \prod_{i=1}^{k} Q_i^{e_i}(u)$. If G contains at least $2r-2$ distinct elements, $r = max_i(e_i * degQ_i(u))$, then $\mu_B(S;G) = \bar{\mu_B}(S;G) = 2n-k$.*

In light of these theorems the multiplicative complexity of cyclic convolution is determined by letting the polynomial $P(u) = u^n - 1$. The result is that the m/d complexity of n point cyclic convolution is $2n-$ (the number of irreducible factors of $u^n - 1$). This is stated as a corollary.

Corollary 2.2.1 *$\mu_B(u^n - 1;Q) = 2n - \tau(n)$, where $\tau(n)$ is the number of positive divisors of n.*

In the next section, algorithms for convolution will be designed by using these results from complexity theory.

2.2.3 Fast Algorithms

In this section, algorithms for linear and cyclic convolution are derived. In determining the complexity of these operations one approach was already given. This was the Cook-Toom algorithm which, as pointed out above, is based on Lagrange Interpolation. This algorithm is again discussed along with several other well known methods.

The Convolution Theorem

This first fast algorithm considered is for cyclic convolution and is based on the convolution theorem 2.1.19. It has already been shown in section 2.2.1 that the matrix representation of cyclic convolution can be expressed as

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} h_0 & h_{n-1} & h_{n-2} & \cdots & h_1 \\ h_1 & h_0 & h_{n-1} & \cdots & h_2 \\ h_2 & h_1 & h_0 & \cdots & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & h_{n-3} & \cdots & h_0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix} \tag{2.62}$$

where the matrix above is circulant in structure. If the indeterminates in this equation are elements of the field of complex numbers, then the matrix W in the convolution theorem becomes the discrete Fourier transform (DFT) matrix as defined in section 2.3.2. Using this approach, the circulant matrix above is factored as

$$\bar{y_n} = (C_n)\bar{d_n} = (F_n^{-1} D_n F_n)\bar{d_n} \tag{2.63}$$

$$D_n = diag[(F_n)\bar{h_n}]$$

where $\bar{h_n}$ is the first column of the matrix (2.62). This method was presented by Stockham in [11]. The advantage of this factorization comes from the fact that efficient algorithms (FFTs) for the computation of DFT exist. The exact amount of computational savings achieved, by using this approach, is highly dependent on the length of the sequences. If for example $n = 2^\alpha$ the DFT stage is implemented using a Cooley-Tukey type algorithm

Length	Mult.	Adds	Total Ops.
8	48	120	168
16	200	356	556
32	584	932	1516
64	1544	2308	3852
128	3080	5124	8204

Table 2.2: Convolution by Convolution Theorem

as described in 2.3.2. If $n = p$ for p prime, the Rader algorithm or one of Tolimieri-Lu variants, as described in section 2.3.4, could be used. The numbers of operations required assuming complex input data for various sizes of n is shown in table 2.1. The Cooley-Tukey algorithms, of section 2.3.2, were used in determining these arithmetic counts.

The Cook-Toom Algorithm

The Cook-Toom algorithm was already used in section 2.2.2 in order to establish the multiplicative complexity of linear convolution. As was shown, the procedure is based on the Lagrange Interpolation formula. The algorithm was shown to be

1. Choose $n + m - 1$ unique elements of the ground field G
2. Evaluate the polynomials $h_{n-1}(x)$ and $d_{m-1}(x)$ at these points. Since $B_i \in G$, the evaluation of $h(B_i)$ and $d(B_i)$ does not require any general multiplication as described in section 2.2.3.
3. Evaluate the polynomial $y(B_i)$ by forming the products $y(B_i) = h(B_i)d(B_i) \quad \forall i$. This step will require $m + n - 1$ multiplications.
4. Use the interpolation formula in order to determine the coefficients of the polynomial $p(x)$ from $p(B_i)'s$. Again, this step will not require any general multiplications, only additions and multiplications of indeterminates and elements of G.

Since a detailed example of this is given in 2.2.2 one will not be repeated here. Instead the algorithm, as derived, is presented as a matrix factorization. Recall that the algorithm was given by

$$\begin{aligned} y(B_0) &= (h_0)(d_0) \\ y(B_1) &= (h_0 + h_1)(d_0 + d_1) \\ y(B_2) &= (h_0 - h_1)(d_0 - d_1) \end{aligned}$$

$$\begin{aligned} y_2 &= 1/2y(B_1) + 1/2y(B_2) - y(B_0) \\ y_1 &= 1/2y(B_1) - 1/2y(B_2) \\ y_0 &= y(B_0) \end{aligned}$$

This can be expressed as

$$\begin{bmatrix} h_0 & \\ h_1 & h_0 \\ & h_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} k_1 & & \\ & 1/2(k_2) & \\ & & 1/2(k_3) \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.64)$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix}$$

Winograd Algorithm

The Winograd Convolution algorithm is a special case of a more general algorithm due to Winograd for the efficient computation of the product $y(x) = h(x)d(x) \bmod M(x)$. The procedure is based on the Chinese Remainder Theorem (CRT) 2.1.16 for polynomials. The CRT gives a way of determining $y(x)$ from its residues $\{y_i(x)\}$. Instead of forming the product $h(x)d(x)$ directly the images $y_i(x) = h_i(x)d_i(x) \bmod m_i(x)$ are determined. Using the CRT, $y(x)$ is then obtained from the set of $\{y_i(x)\}$. The procedure is summarized below.

1. Reduce the polynomials $h(x)$ and $d(x) \bmod m_i(x)$ for each i. A set of images $\{h_i(x)\}$ and $\{d_i(x)\}$ is obtained.

2. Form the polynomial products $h_i(x)d_i(x) \bmod m_i(x)$ in each of the subrings. This gives the images of $y_i(x) \bmod m_i(x)$.

3. Use the reconstruction formula

$$y(x) = \sum_{i=0}^{k} y_i(x) N_i(x) M_i(x) \qquad \bmod M(x)$$

in order to determine $y(x)$ from the images $y_i(x)$.

By properly choosing $M(x)$ it is possible to derive algorithms for linear and cyclic convolution. In particular, letting $M(x) = x^n - 1$ yields algorithms for cyclic convolutions, while choosing deg $M(x)$ such that deg $M(x) >$ deg $y(x)$, algorithms for linear convolution are obtained. Examples of both of these cases are given below.

- **Example 2.2.2** 3 by 3 Linear Convolution

$$h(x) = h_2x^2 + h_1x + h_0 \qquad d(x) = d_2x^2 + d_1x + d_0$$
$$y(x) = h(x)d(x) \qquad \bmod M(x)$$
$$M(x) = x^5 - x^4 + x^3 - x^2$$
$$M(x) = m_0(x)m_1(x)m_2(x) = x^2(x^2+1)(x-1)$$

- step 1

$$\begin{aligned} h_0(x) &= h_1x + h_0 & \qquad d_0(x) &= d_1x + d_0 \\ h_1(x) &= h_1x + (h_0 - h_2) & \qquad d_1(x) &= d_1x + (d_0 - d_2) \\ h_2(x) &= (h_0 + h_1 + h_2) & \qquad d_2(x) &= (d_0 + d_1 + d_2) \end{aligned}$$

- step 2

$$\begin{aligned} y_0(x) &= (h_1x + h_0)(d_1x + d_0) \pmod{x^2} \\ y_1(x) &= (h_1x + (h_0 - h_2))(d_1x + (d_0 - d_2)) \pmod{x^2+1} \\ y_2(x) &= (h_0 + h_1 + h_2)(d_0 + d_1 + d_2) \pmod{x-1} \end{aligned}$$

$$\begin{aligned} m_1 &= h_0d_1 \\ m_2 &= h_1d_0 \\ m_3 &= h_0d_0 \\ m_4 &= (h_1 - h_0 + h_2)(d_0 - d_2) \\ m_5 &= (h_1 + h_0 - h_2)(d_1) \\ m_6 &= (h_1)(d_1 + d_0 - d_2) \\ m_7 &= (h_2 + h_1 + h_0)(d_2 + d_1 + d_0) \end{aligned}$$

$$\begin{aligned} y_0(x) &= (m_1 + m_2)x + m_3 \\ y_1(x) &= (m_4 + m_6)x + (m_5 - m_6) \\ y_2(x) &= m_7 \end{aligned}$$

- step 3

$$y(x) = \alpha_0(x)y_0(x) + \alpha_1(x)y_1(x) + \alpha_2(x)y_2(x)$$

$$\begin{aligned} \alpha_0(x) &= -x^4 + 1 \\ \alpha_1(x) &= 1/2(x^4 - x^2) \\ \alpha_2(x) &= 1/2(x^4 + x^2) \end{aligned}$$

$$y(x) = y_4x^4 + y_3x^3 + y_2x^2 + y_1x + y_0 =$$

$$\alpha_0(x)y_0(x) + \alpha_1(x)y_1(x) + \alpha_2(x)y_2(x) \pmod{M(x)}$$

Equating coefficients yields

$$\begin{aligned} y_4 &= -(m_1 + m_2) + m_3 + 1/2m_7 + 1/2(m_4 + m_6) + 1/2(m_5 - m_6) \\ y_3 &= (m_1 + m_2) - 3/2(m_4 + m_6) \\ y_2 &= -(m_1 + m_2) + 1/2m_7 + 1/2(m_4 + m_6) - (m_5 - m_6) \\ y_1 &= (m_1 + m_2) \\ y_0 &= m_3 \end{aligned}$$

□

As with the Cook-Toom algorithm, this can be expressed as a matrix factorization. This is

$$\begin{bmatrix} h_0 & & \\ h_1 & h_0 & \\ h_2 & h_1 & h_0 \\ & h_2 & h_1 \\ & & h_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & -2 & -1 & 1 \\ 1 & 1 & 0 & -3 & 0 & -3 & 0 \\ -1 & -1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ 1/2k_4 \\ 1/2k_5 \\ 1/2k_6 \\ 1/2k_7 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ 1/2k_4 \\ 1/2k_5 \\ 1/2k_6 \\ 1/2k_7 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}$$

- **Example 2.2.3** 3 by 3 Cyclic Convolution

$$h(x) = h_2x^2 + h_1x + h_0 \qquad d(x) = d_2x^2 + d_1x + d_0$$

compute

$$y(x) = h(x)d(x) \qquad \text{mod } M(x) = (x^3 - 1)$$

$$M(x) = m_0(x)m_1(x) = (x - 1)(x^2 + x + 1)$$

- step 1

$$\begin{aligned} h_0(x) &= h_2 + h_1 + h_0 \\ h_1(x) &= (h_1 - h_2)x + (h_0 - h_2) \end{aligned} \qquad \begin{aligned} d_0(x) &= d_2 + d_2 + d_0 \\ d_1(x) &= (d_1 - d_2)x + (d_0 - d_2) \end{aligned}$$

- step 2

$$y_0(x) = h_0(x)d_0(x) \text{ mod } x$$

$$m_0 = h_0d_0$$

$$y_1(x) = h_1(x)d_1(x) \text{ mod } x^2 + x + 1 =$$

$$[(h_1 - h_2)x + (h_0 - h_2)]\,[(d_1 - d_2)x + (d_0 - d_2)]$$

Implement this by using the Cook-Toom algorithm. This yields

$$\begin{aligned} m_1 &= (h_0 - h_2)(d_0 - d_2) \\ m_2 &= (h_0 + h_1 - 2h_2)(d_0 + d_1 - 2d_2) \\ m_3 &= (h_0 - h_1)(d_0 - d_1) \end{aligned}$$

$$y_1(x) = (m_1 - m_3)x + (2m_1 - 1/2m_2 - 1/2m_3)$$

- step 3

$$y(x) = \alpha_0(x)y_0(x) + \alpha_1(x)y_1(x)$$

$$\alpha_0(x) = 1/3(x^2 + x + 1)$$
$$\alpha_1(x) = 1/3(2 - x - x^2)$$

$$y(x) = y_2x^2 + y_1x + y_0 = 1/3(x^2 + x + 1)y_0(x) + 1/3(2 - x - x^2)y_1(x)$$

Equating coefficients yields

$$\begin{aligned} y_2 &= 1/3(m_0 - 3m_1 + 1/2m_2 + 3/2m_3) \\ y_1 &= 1/3(m_0 + 1/2m_2 - 3/2m_3) \\ y_0 &= 1/3(m_0 + m_1 - 1/2m_2 + 1/2m_3) \end{aligned}$$

□

As a matrix factorization, this algorithm is represented by

$$\begin{bmatrix} h_0 & h_2 & h_1 \\ h_1 & h_0 & h_2 \\ h_2 & h_1 & h_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 & 1 \\ 1 & 0 & 1 & -3 \\ 1 & -3 & 1 & 3 \end{bmatrix} diag \begin{bmatrix} 1/3k_1 \\ 1/3(k_2) \\ 1/6(k_3) \\ 1/6k_4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} \tag{2.65}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}$$

The Winograd algorithm offers some flexibility as to the factors of $M(x)$. This allows for the possibility of deriving several different Winograd algorithms for a given computation. These different algorithms will have different breakdowns of additions and multiplications. The best algorithm being determined by the particular implementation at hand. A partial listing, reproduced from [1], is given in table 2.3.

Length	Mult.	Adds	Total Ops.
2	2	4	6
3	4	11	15
4	5	15	20
5	8	62	70
7	16	70	86
8	12	72	84
9	19	74	93
16	33	181	214
16	35	155	190

Table 2.3: Small Winograd Convolution

Multidimensional Techniques

Multidimensional approaches decompose a convolution of size $n = n_1 n_2$ into two dimensional convolutions of size (n_1, n_2). There are two important advantages to this. First, it allows for the computation of large convolutions from a library of highly optimized small size convolutions. Second, it allows one to mix different techniques in order to obtain a wide variety of new algorithms with different arithmetic counts. These points will be very important in subsequent chapters. There are at least two ways of expressing a one dimensional convolution as a two dimensional convolution. The first due to Agarwal and Cooley [14], and another which is referred to as the iteration approach.

The Agarwal-Cooley algorithm uses the Chinese Remainder Theorem (CRT) for integers, which is given as theorem 2.1.15, to establish the ring isomorphism

$$Z_n \cong Z_{n0} \times Z_{n1} \tag{2.66}$$

This isomorphism reveals that a one dimensional n point cyclic convolution is equivalent to a two dimensional cyclic convolution (n_0, n_1), if $n = n_0 n_1$ and $GCD(n_0, n_1) = 1$. For $n = n_0 n_1$ theorem 2.1.15 asserts that $c \in Z_n$ is uniquely determined by

$$c = c_0 e_0 + c_1 e_1 \qquad e_i = N_i M / m_i$$

where $c_0 \in Z_{n0}$ and $c_1 \in Z_{n1}$. Every c is expressed uniquely as a linear combination of the idempotents e_i. Consider two arbitrary elements of Z_n, c and c', since

$$c = c_0 e_0 + c_1 e_1 \Longleftrightarrow (c_0, c_1)$$
$$c' = c'_0 e_0 + c'_1 e_1 \Longleftrightarrow (c'_0, c'_1)$$

and using

$$e_0 * e_1 = 0, \qquad e_0^2 = e_0 \qquad e_1^2 = e_1 \pmod{n}$$

the isomorphism is readily established by

$$c + c' \Longleftrightarrow (c_o + c'_0, c_1 + c'_1)$$

$$c * c' \Longleftrightarrow (c_o * c'_0, c_1 * c'_1)$$

Note that this development extends directly the case $n = \prod n_i$.

In terms of the matrix representation (2.57)

$$\bar{y_n} = (C_n)\bar{d}_n = \begin{bmatrix} h_0 & h_{n-1} & h_{n-2} & \cdots & h_1 \\ h_1 & h_0 & h_{n-1} & \cdots & h_2 \\ h_2 & h_1 & h_0 & \cdots & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & h_{n-3} & \cdots & h_0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

The CRT guarantees the existence of n by n permutation matrices P_1 and P_2 such that

$$C'_n = P_1(C_n)P_2$$

where the permutation matrices are given by the index maps

$$j = e_0 j_0 + e_1 j_1 \qquad 0 \le j_0, k_0 < n_0$$

$$k = e_0 k_0 + e_1 k_1 \qquad 0 \le j_1, k_1 < n_1$$

and the matrix C'_n has two dimensional cyclic structure. An example will help to clarify the procedure.

- **Example 2.2.4** 6 Point Cyclic Convolution

By definition this is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} h_0 & h_5 & h_4 & h_3 & h_2 & h_1 \\ h_1 & h_0 & h_5 & h_4 & h_3 & h_2 \\ h_2 & h_1 & h_0 & h_5 & h_4 & h_3 \\ h_3 & h_2 & h_1 & h_0 & h_5 & h_4 \\ h_4 & h_3 & h_2 & h_1 & h_0 & h_5 \\ h_5 & h_4 & h_3 & h_2 & h_1 & h_0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{bmatrix} \tag{2.67}$$

Let $n_0 = 2$, $n_1 = 3$, $e_0 = 3$, $e_1 = 4$, and use the index maps

$$\begin{aligned} j &= 3j_0 + 4j_1 \pmod 6 & 0 \le j_0, k_0 < n_0 \\ k &= 3k_0 + 4k_1 \pmod 6 & 0 \le j_1, k_1 < n_1 \end{aligned}$$

to yield the permutation $\phi = (0, 4, 2, 3, 1, 5)$. This permutation is applied to the rows and columns of 2.42 to get

$$\begin{bmatrix} y_0 \\ y_4 \\ y_2 \\ y_3 \\ y_1 \\ y_5 \end{bmatrix} = \left[\begin{array}{ccc|ccc} h_0 & h_2 & h_4 & h_3 & h_5 & h_1 \\ h_4 & h_0 & h_2 & h_1 & h_3 & h_5 \\ h_2 & h_4 & h_0 & h_5 & h_1 & h_3 \\ \hline h_3 & h_5 & h_1 & h_0 & h_2 & h_4 \\ h_1 & h_3 & h_5 & h_4 & h_0 & h_2 \\ h_5 & h_1 & h_3 & h_2 & h_4 & h_0 \end{array}\right] \begin{bmatrix} d_0 \\ d_4 \\ d_2 \\ d_3 \\ d_1 \\ d_5 \end{bmatrix} \tag{2.68}$$

which does exhibit two by two block circulant structure, each block having three by three circulant structure.

□

Another method uses an iteration technique in order to decompose algorithms. The idea is best illustrated by an example. Below, an efficient algorithm for a 2 point linear convolution is used in order to derive an efficient algorithm for a four point convolution.

- **Example 2.2.5** An iteration technique

  An efficient algorithm for the computation of 2 point liner convolution is presented below. The algorithm is based on the identities

$$\begin{aligned} h_0 d_0 &= h_0 d_0 \\ h_1 d_0 + h_0 d_1 &= (h_0 + h_1)(d_0 + d_1) - h_0 d_0 - h_1 d_1 \\ h_1 d_1 &= h_1 d_1 \end{aligned}$$

Now, consider the polynomial product

$$y(x) = (h_3x^3 + h_2x^2 + h_1x + h_0)(d_3x^3 + d_2x^2 + d_1x + d_0) \quad (2.69)$$

Letting

$$H_1 = (h_3x^2 + h_2x) \qquad H_0 = (h_1x + h_0)$$
$$D_1 = (d_3x^2 + d_2x) \qquad D_0 = (d_1x + d_0)$$

(2.69) becomes

$$y(x) = (H_1x + H_0)(D_1x + D_0)$$

which can be computed using the identities

$$\begin{aligned} H_0D_0 &= H_0D_0 \\ H_1D_0 + H_0D_1 &= (H_0 + H_1)(D_0 + D_1) - H_0D_0 - H_1D_1 \\ H_1D_1 &= H_1D_1 \end{aligned}$$

Each polynomial product H_iD_i, however, is a 2 point linear convolution. Again use the identies in order to compute these. Thus it is possible to obtain algorithms for large convolutions by iterating smaller algorithms.

□

Heuristic Algorithms

In the previous sections many different types of algorithms were presented for the computation of both linear and cyclic convolution. These algorithms were designed by a variety of relatively sophisticated mathematical techniques. It is important to understand, however, that although the mathematics provides an organized framework for approaching fast convolution, it does not exhaust all the possible ways of obtaining fast algorithms. In this section, this fact is illustrated by presenting some algorithms with cannot be derived by any of the techniques presented. In many cases they result from clever factorizations of systems of bilinear equations.

The first example presented is for the computation of three by three linear convolution. Consider the polynomial product

$$y(x) = y_4x^4 + h_3x^3 + h_2x^2 + h_1x + h_0 = (h_2x^2 + h_1x + h_0)(d_2x^2 + d_1x + d_0)$$

Carrying out the multiplication and equating the coefficients yields

$$\begin{aligned} y_0 &= h_0 d_0 \\ y_1 &= h_1 d_0 + h_0 d_1 \\ y_2 &= h_2 d_0 + h_1 d_1 + h_0 d_2 \\ y_3 &= h_1 d_2 + h_2 d_1 \\ y_4 &= h_2 d_2 \end{aligned}$$

Through the use of the factorizations

$$\begin{aligned} h_0 d_0 &= h_0 d_0 \\ h_1 d_0 + h_0 d_1 &= (h_0 + h_1)(d_0 + d_1) - h_0 d_0 - h_1 d_1 \\ h_2 d_0 + h_1 d_1 + h_0 d_2 &= (h_0 + h_2)(d_0 + d_2) - h_0 d_0 + h_1 d_1 - h_2 d_2 \\ h_1 d_2 + h_2 d_1 &= (h_1 + h_2)(d_1 + d_2) - h_1 d_1 - h_2 d_2 \\ h_2 d_2 &= h_2 d_2 \end{aligned}$$

an algorithm requiring 6 m/d steps is obtained. Although this is suboptimal in terms of multiplications, the number of additions required is far less than that required by the optimal Cook-Toom algorithm. This aspect of the arithmetic breakdown makes it much more desireable in many applications. Note that this algorithm cannot be obtained by the Chinese Remainder Theorem or any of the other formal mathematical procedures presented in this section.

As another example consider the implementation of 2 point cyclic convolution.

$$y_2(x) = h_2(x) d_2(d) \bmod x^2 - 1$$

$$(y_1 x + y_0) = (h_1 x + h_0)(d_1 x + d_0)$$

Equating the coefficients yields

$$y_0 = h_0 d_0 + h_1 d_1$$

$$y_1 = h_1 d_0 + h_0 d_1$$

which can be implemented using the identities

$$h_0 d_0 + h_1 d_1 = h_0(d_0 + d_1) + (h_1 - h_0) d_1$$

$$h_1 d_0 + h_0 d_1 = h_0(d_0 + d_1) + (h_1 - h_0) d_0$$

Written as a matrix factorization this is

$$\begin{bmatrix} h_0 & h_1 \\ h_1 & h_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_1 & & \\ & k_2 & \\ & & k_3 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \tag{2.70}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix}$$

There are many other ways of using heuristics in order to design convolution algorithms. One of the most powerful methods combines the structure of the formal mathematical approaches with the flexibility of the purely heuristic approaches presented above. In this approach the first step is to use one of the multidimensional techniques described in this section and decompose a large size convolution into combinations of smaller ones. Once this is done, the mathematical techniques presented are mixed and matched in order to obtain algorithms which could not otherwise be easily derived.

For example, consider the problem of designing an algorithm for the computation of the cyclic convolution of length six. By the Agarwal-Cooley algorithm this is reexpressed as a $(3,2)$ two dimensional cyclic convolution. The designer is now free to choose any approaches in order to implement the different dimensions. For instance, it may be advantageous in some applications to use the Winograd algorithm for the 3 point cyclic convolutions, the heuristic algorithm (2.70) for the 2 point convolution. The resulting algorithm is presented below.

- **Example 2.2.6** Four point linear convolution

  Since algorithm (2.70) is non-commutative (see section 2.1.3) it can be used to factor the block circulant matrix (2.68). Rewriting (2.70) for this matrix yields

$$\begin{bmatrix} H_0 & H_1 \\ H_1 & H_0 \end{bmatrix} = \begin{bmatrix} 0 & I_3 & I_3 \\ I_3 & 0 & I_3 \end{bmatrix} \begin{bmatrix} K_1 & & \\ & K_2 & \\ & & K_3 \end{bmatrix} \begin{bmatrix} I_3 & 0 \\ 0 & I_3 \\ I_3 & I_3 \end{bmatrix} \tag{2.71}$$

$$\begin{bmatrix} K_1 \\ K_2 \\ K_3 \end{bmatrix} = \begin{bmatrix} -I_3 & I_3 \\ -I_3 & I_3 \\ I_3 & 0 \end{bmatrix} \begin{bmatrix} H_0 \\ H_1 \end{bmatrix}$$

$$H_0 = \begin{bmatrix} h_0 & h_2 & h_4 \\ h_4 & h_0 & h_2 \\ h_2 & h_4 & h_0 \end{bmatrix} \qquad H_1 = \begin{bmatrix} h_3 & h_5 & h_1 \\ h_1 & h_3 & h_5 \\ h_5 & h_1 & h_3 \end{bmatrix}$$

and I_n is an n point identity matrix. Note that each K_i is a 3 point circulant matrix. The Winograd algorithm will be used in order to implement these stages. This was expressed as the matrix factorization

$$\begin{bmatrix} h_0 & h_2 & h_1 \\ h_1 & h_0 & h_2 \\ h_2 & h_1 & h_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 & 1 \\ 1 & 0 & 1 & -3 \\ 1 & -3 & 1 & 3 \end{bmatrix} diag \begin{bmatrix} 1/3k_1 \\ 1/3(k_2) \\ 1/6(k_3) \\ 1/6k_4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} \tag{2.72}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}$$

Embedding the algorithm for C_3 into the block algorithm for C_2 results in

$$\begin{bmatrix} H_0 & H_1 \\ H_1 & H_0 \end{bmatrix} = \begin{bmatrix} 0 & A & A \\ A & 0 & A \end{bmatrix} \begin{bmatrix} \Gamma_1 & & \\ & \Gamma_2 & \\ & & \Gamma_3 \end{bmatrix} \begin{bmatrix} B & 0 \\ 0 & B \\ B & B \end{bmatrix} \tag{2.73}$$

where

$$A = \begin{bmatrix} 1 & 1 & -1 & 1 \\ 1 & 0 & 1 & -3 \\ 1 & -3 & 1 & 3 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix}$$

and

$$\Gamma_1 = \Gamma_2 \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} h_5 - h_0 \\ h_1 - h_4 \\ h_5 - h_2 \end{bmatrix}$$

$$\Gamma_3 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -2 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} h_0 \\ h_4 \\ h_2 \end{bmatrix}$$

□

This section has illustrated that although formal mathematical approaches to designing algorithms are theoretically important and significant, they may also be somewhat restrictive if one only considers these. It is very possible that the best algorithm for a particular implementation is obtained by the heuristic approaches outlined above.

2.2.4 Number Theoretic Transforms

Many ways of designing convolution algorithms have been presented. In these, some aspect of the underlying ring or field structure of the indexing set of the indeterminates was exploited. The exact nature of the field of indeterminates was not an issue. In this section a closer look is taken at the precise nature of this field in an attempt to gain further insights.

The fundamental observation on this section is that, on any digital computer, numbers are represented by a finite number of bits. Because of this, no infinite field can be exactly represented by these finite machines. The by-product of assuming that they can is round-off error. This observation allows us to think of any computation on a digital computer as taking place in a suitably large finite ring or field. This interpretation leads to very different types of algorithms.

It is well known that the convolution theorem 2.1.19 is valid over finite fields and in some instances even finite rings (in the case of a finite ring the further requirement being that the length of the transform n have a multiplicative inverse in the ring). The kernel of the transform α can be any element of the ring of order n. There may be some choices of α which make the computation very efficient. If, for example, $\alpha = 2^k$ and numbers are represented using a fixed point binary representation, the convolution theorem can be applied directly without the need for any multiplication whatsoever.

Another advantage of thinking in terms of finite structures is that no round off error is incurred in the computation. As mentioned, round off error results from trying to express an infinite structure on a finite computer. Since our new interpretation is also inherently finite its representation on a computer is direct. This results in error free computation, provided the computer word is of sufficient length.

To a large extent the success of a number theoretic approache lies in the ability to find "good" fields and rings in which to compute. In searching for these, three requirements are juggled. As pointed out by Rice [15], fields/rings are needed that (1) admit transforms of favorable lengths, (2) are large enough to allow for sufficient dynamic range, and (3) have some special property that enables multiplications to be implemented efficiently.

The first approaches embedded the entire computation into a suitably large Galios field. Rader [16] has investigated the use of Mersenne primes, while, Agarwal-Burrus [17] explored the use of Fermat primes, for the determination of Galios fields $GF(p)$ in which to compute. A Mersenne prime is a prime of the form $p = 2^t - 1$; a Fermat prime is of the form $p = 2^t + 1$. If integers are represented by t bit binary words the arithmetic in these fields is relatively straight forward. In both cases it is ordinary binary arithmetic; the difference being how the overflow is handled. In the case of Mersenne fields the overflow bits are subtracted in order to obtain a final answer, and in the case of Fermat fields the overflow bits are added to achieve the modulo effect. This makes these fields very attractive for use in implementations.

As for any field, the multiplicative group of Fermat and Mersenne fields is cyclic. Using this fact, and theorem 2.1.18, it is established that for Mersenne fields, an element of order k exists for every k that divides $2^t - 2$. In the case of Fermat fields, this result guarantees the existence of an element of order k for every k that divides 2^t. Hence, for Fermat fields it is possible to choose a kernel as a power of two. Further, since the algorithm is based on the structure of the indexing set and not dependent on the exact nature of the constants involved, it is also possible to use a Cooley-Tukey type FFT algorithm, as described in section 2.3.2, in order to implement this cyclic convolution. Below, an example is given for the Fermat Field $GF(2^2 + 1)$. Note that Blahut [18] provides an excellent introduction to the ideas presented in this subsection.

- **Example 2.2.7** Four point linear convolution

  The arithmetic of $GF(5)$ is given by

((+))	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

((*))	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

and $\alpha = 2$ is a primitive element. The matrix product

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} h_0 & h_3 & h_2 & h_1 \\ h_1 & h_0 & h_3 & h_2 \\ h_2 & h_1 & h_0 & h_3 \\ h_3 & h_2 & h_1 & h_0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

$$\begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 0 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}$$

over $GF(5)$ can be computed two ways, directly or by the convolution theorem.

- Direct Computation

$$\begin{bmatrix} 4 \\ 3 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 0 & 4 \\ 4 & 3 & 1 & 0 \\ 0 & 4 & 3 & 1 \\ 1 & 0 & 4 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}$$

- By Convolution Theorem

$$y = Cd = (W^{-1}DW)d$$

$$D = diag(Wh)$$

$$W = \begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^1 & \alpha^2 & \alpha^3 \\ \alpha^0 & \alpha^2 & \alpha^0 & \alpha^2 \\ \alpha^0 & \alpha^3 & \alpha^2 & \alpha^1 \end{bmatrix} \qquad W^{-1} = (4^{-1}) \begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^3 & \alpha^2 & \alpha^1 \\ \alpha^0 & \alpha^2 & \alpha^0 & \alpha^2 \\ \alpha^0 & \alpha^1 & \alpha^2 & \alpha^3 \end{bmatrix}$$

By the Cooley-Tukey algorithm of section xx

$$W = \begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^1 & \alpha^2 & \alpha^3 \\ \alpha^0 & \alpha^2 & \alpha^0 & \alpha^2 \\ \alpha^0 & \alpha^3 & \alpha^2 & \alpha^1 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & \alpha^2 & 0 \\ 0 & 1 & 0 & \alpha^2 \end{bmatrix} diag \begin{bmatrix} 1 \\ 1 \\ 1 \\ \alpha \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & \alpha^2 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & \alpha^2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$W^{-1} = \begin{bmatrix} \alpha^0 & \alpha^0 & \alpha^0 & \alpha^0 \\ \alpha^0 & \alpha^3 & \alpha^2 & \alpha^1 \\ \alpha^0 & \alpha^2 & \alpha^0 & \alpha^2 \\ \alpha^0 & \alpha^1 & \alpha^2 & \alpha^3 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & \alpha^2 & 0 \\ 0 & 1 & 0 & \alpha^2 \end{bmatrix} diag \begin{bmatrix} 1 \\ 1 \\ 1 \\ \alpha^3 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & \alpha^2 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & \alpha^2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag(W)h = \begin{bmatrix} 3 & & & \\ & 4 & & \\ & & 3 & \\ & & & 2 \end{bmatrix}$$

$$y = Cd = (W^{-1}DW)d = \begin{bmatrix} 4 \\ 3 \\ 3 \\ 1 \end{bmatrix}$$

□

Although $GF(5)$ is too small of a field for most practical problems the example does illustrate the technique. Unfortunately, for the case of Fermat fields, it seems the largest t for which 2^t+1 is prime is $t = 16$. This imposes a relatively strong restriction on the dynamic range of sequences in addition to the length of the sequences which can be cyclically convolved using this approach.

There have been a number of proposals which address this limitation. Rader, Agarwal, and Burrus, suggested the use a multidimensional technique, such as described in section 2.2.3, to represent a long one dimensional convolution as a two dimensional convolution. Once this is done, the convolution theorem is used to implement each dimension separately. More recently, Lee and Lu [19] have proposed the use of primes of the form $p = k^q \pm (k-1)$ and implementations using multi- valued logic. Still, other approaches advocate the use of finite rings. The ring is deposed via the Chinese Reminder Theorem into a direct sum of smaller subrings. Multiplication is avoided in each subring either by using the convolution theorem with $\alpha = 2^k$ or by using small look-up tables in the cases where no such α exists.

Number theoretic approaches will not play a role in this work because trading multiplications for additions, or even shifts, is not particularly advantageous on the modern RISC computer.

2.3 Discrete Fourier Transform

This discussion of the Discrete Fourier Transform parallels that of section 2.2, which focused on convolution. As for convolution, many fast algorithms (FFT's) for the implementation of the discrete Fourier transform (DFT) will be presented. The tool which unifies the development is the tensor product as described in section 2.1.5.

The section begins by establishing the multiplicative complexity of the DFT operator. Having done this, several techniques of obtaining fast algorithms are presented. The algorithms have been organized into three types: additive algorithms, multiplicative algorithms, and algorithms based on artificial intelligence approaches.

2.3.1 Multiplicative Complexity

The search for efficient algorithms begins by presenting theorems which establish the multiplicative complexity of DFT. The three theorems below appear in [36], which also gives detailed proofs of these results (see also, [24] for a less formidable discussion). The theorems establish the multiplicative complexity of DFT for sequence lengths which are powers of primes. The multiplicative complexity, for n an odd prime, is established first.

Theorem 2.3.1 *For an input vector of p indeterminates, p an odd prime, $\mu_B(DFT(p);Q) = 2p-\phi(p-1)-3$, where Q is the field of rational numbers and $\phi(k)$ is the number of divisors of k.*

The next theorem generalizes the above by extending it to the case where the sequence length n is the power of an odd prime.

Theorem 2.3.2 *For an input vector of p^r indeterminates, p an odd prime, and r a positive integer, $\mu_B(DFT(p^r);Q) = 2p^r - r - 2 - (r^2+r)/2\phi(p-1)$.*

The final theorem treats the case of sequence length n being a power of two.

Theorem 2.3.3 *For an input vector of 2^r indeterminates, r a positive integer, $\mu_B(DFT(2^r);Q) = 2^{r+1} - r - r^2 - 2$.*

These results assume real valued inputs. For complex inputs the complexities, as established above, are doubled. It should be mentioned that reference [36] presents a theorem which determines the multiplicative complexity of DFT for arbitrary length sequences, complete with proof, as well as extensive tables which list the multiplicative complexity for a large variety of different sequence lengths. For our purposes, however, theorems 2.3.1-2.3.3 will be sufficient. As for convolution, these results will serve as guidelines in determining the multiplicative efficiency of the algorithms derived. The optimal multiplicative algorithm for a large length n will seldom be used directly due to number of additions required.

2.3.2 Additive Algorithms

Additive FFT algorithms make use of the additive structure of the indexing set to decompose a one dimensional DFT into a multidimensional DFT. The decomposition of the large one dimensional computation into a set of smaller ones significantly reduces the number of arithmetic operations required. This reduction is possible when the transform size $n = sr$ is composite, and is achieved through use of index maps of the form

$$j = j_1 + j_2 s \qquad 0 \leq j_1, k_1 < s \tag{2.74}$$

$$k = rk_1 + k_2 \qquad 0 \leq j_2, k_2 < r$$

All of the additive algorithms discussed are obtained by choosing special forms of expression (2.74), and by manipulating the addressing requirements of the resulting expressions. The use of these maps is illustrated below.

The definition of the DFT operator is given by

$$Y(k) = \sum_{j=0}^{n-1} \omega_n^{jk} X(j) \qquad \omega_n = e^{-\frac{2\pi i}{n}} \tag{2.75}$$

Using (2.74), this one dimensional transform is made into a two dimensional transform. The substitution (2.74) into (2.75) yields

$$Y(rk_1 + k_2) = \sum_{j_1=0}^{s-1} \sum_{j_2=0}^{r-1} \omega_n^{(j_1+sj_2)(rk_1+k_2)} X(j_1 + sj_2) \tag{2.76}$$

$$Y(rk_1 + k_2) = \sum_{j_1=0}^{s-1} \sum_{j_2=0}^{r-1} \omega_n^{j_1 k_1 r} \omega_n^{j_1 k_2} \omega_n^{s j_2 r k_1} \omega_n^{s j_2 k_2} X(j_1 + s j_2) \qquad (2.77)$$

rearranging the resulting terms, and using the fact that

$$\omega_n^{j_1 k_1 r} = \omega_s^{j_1 k_1}, \quad \omega_n^{s j_2 r k_1} = 1, \quad and \quad \omega_n^{s j_2 k_2} = \omega_r^{j_2 k_2}$$

the following two dimensional expression is obtained.

$$Y(rk_1 + k_2) = \sum_{j_1=0}^{s-1} \omega_s^{j_1 k_1} (\omega_n^{j_1 k_2} (\sum_{j_2=0}^{r-1} \omega_r^{j_2 k_2} X(j_1 + s j_2))) \qquad (2.78)$$

Tensor Product Representation

In section 2.2.1 a matrix representation of convolution was used in order to gain insights into its underlying symmetry. This representation was introduced because the symmetries were hidden by the summation formula. Similarly, for the DFT, the summation representation hides a great deal of computational symmetry. As was done in section 2.2.1, it is possible to introduce a matrix representation of formula (2.75). Algorithm (2.78) can then be rederived using this matrix representation. The tool which facilitates this construction is the tensor product of matrices as introduced in section 2.1.2. Once the tensor product representation of algorithm (2.78) is established, it is used as a unification tool in order to describe the structure of all the additive algorithms presented throughout this section.

Again, begin with the definition of DFT

$$Y(k) = \sum_{j=0}^{n-1} \omega_n^{jk} X(j) \qquad \omega_n = e^{-\frac{2\pi i}{n}}$$

This time, express the computation as a matrix product by carrying out the summation expression. This yields

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{n-2} & \cdots & \omega^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \qquad (2.79)$$

Let

$$F_n = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{n-2} & \cdots & \omega^1 \end{bmatrix}$$

and note that the symmetry of F_n is given by the multiplicative group of Z_n. By using this and other facts about (2.75) it is possible to factor this matrix as

$$F_n = (F_s \otimes I_r)\, T_{n,s}\, (I_s \otimes F_r)\, P_{n,s} \tag{2.80}$$

where $P_{n,s}$ is the stride permutation matrix and $\otimes$ is the tensor product, as defined in section 2.1.2, and $T_{n,s}$ is the diagonal matrix given by

$$T_{n,s} = \sum_{j=0}^{s-1} \oplus D^j_{n,n/s} = \begin{pmatrix} D^0_{n,n/s} & & & & \\ & D^1_{n,n/s} & & & \\ & & D^2_{n,n/s} & & \\ & & & \ddots & \\ & & & & D^{s-1}_{n,n/s} \end{pmatrix}$$

$$D^j_{n,r} = diag[1^j, \omega_n^j, \omega_n^{2j}, \cdots, \omega_n^{(r-1)j}] \qquad \omega_n = e^{-\frac{2\pi i}{n}}, \quad n = r \times s$$

The derivation of formula (2.80) is rather lengthy and will not be reproduced here. The interested reader is referred to Rodriguez [20]. Using this expression and the properties 2.9 to 2.16 of section 2.1.2, the resulting matrix factors can be manipulated to obtain a variety of different algorithms.

Cooley-Tukey Algorithm

The Cooley-Tukey FFT algorithm [21] is probably the most studied and referenced algorithm in digital signal processing. In their original paper Cooley-Tukey assume the transform length is of the form $n = 2^\alpha$. This allows the one dimensional transform to be expressed as an α dimensional transform, and diagonal multiplications (these diagonal multiplications are often referred to as twiddle factors).

Before giving a general formula for the Cooley-Tukey decomposition, the technique is illustrated by example. The Cooley-Tukey algorithm for

$n = 2^3$ is derived from (2.80) by letting $n = 8$, $r = 4$, and $s = 2$. This yields

$$F_8 = (F_2 \otimes I_4)\, T_{8,2}\, (I_2 \otimes F_4)\, P_{8,2} \tag{2.81}$$

This can be decomposed further by noting that for $n = 4$, $r = 2$, and $s = 2$, (2.80) also yields

$$F_4 = (F_2 \otimes I_2)\, T_{4,2}\, (I_2 \otimes F_2)\, P_{4,2} \tag{2.82}$$

Substituting (2.82) into (2.81) and using the distributive property of the tensor product

$$F_8 = (F_2 \otimes I_4) T_{8,2} (I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_{4,2})(I_2 \otimes I_2 \otimes F_2)(I_2 \otimes P_{4,2}) P_{8,2}$$

is obtained. Finally, using the identity property (2.12), this becomes

$$F_8 = (F_2 \otimes I_4) T_{8,2} (I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_{4,2})(I_4 \otimes F_2) Q \tag{2.83}$$

$$Q = (I_2 \otimes P_{4,2}) P_{8,2}$$

This matrix factorization for $n = 2^3$ is readily generalized to an arbitrary power of two $n = 2^\alpha$ by the expression

$$F_n = \left(\prod_{i=1}^{\alpha} (I_{2^{i-1}} \otimes F_2 \otimes I_{2^{\alpha-i}}) \left(I_{2^{i-1}} \otimes T_{2^{\alpha-i+1},2} \right) \right) Q \tag{2.84}$$

$$Q = \prod_{i=\alpha}^{1} \left(I_{2^{i-1}} \otimes P_{2^{\alpha-i+1},2} \right)$$

Algorithms of the form (2.84) are referred to as radix two algorithms because every non-twiddle computational stage is a succession of two point Fourier transforms. The development presented above is readily extended to the case of higher radices $n = r^\alpha$. In these cases the resulting algorithms have r point Fourier transform for the non-twiddle computational stages. Algorithms of this form are called radix r Cooley-Tukey. The general formula for a radix r Cooley- Tukey decomposition is

$$F_n = \left(\prod_{i=1}^{\alpha} (I_{r^{i-1}} \otimes F_r \otimes I_{r^{\alpha-i}}) \left(I_{r^{i-1}} \otimes T_{r^{\alpha-i+1},r} \right) \right) Q \tag{2.85}$$

$$Q = \prod_{i=\alpha}^{1} \left(I_{r^{i-1}} \otimes P_{r^{\alpha-i+1},r} \right)$$

The performance of some radix 2 Cooley-Tukey algorithms is given by table 2.4.

Length	Mult.	Adds	Total Ops.
2	0	4	4
4	0	16	16
8	8	52	60
16	68	162	230
32	196	418	614
64	516	1026	1542
128	1284	2434	3718
256	3076	5634	8710
512	7172	12802	19974
1024	16388	28674	45062

Table 2.4: Cooley-Tukey Performance

Mixed-Radix Algorithm

Mixed radix algorithms [25] remove the requirement that $n = r^\alpha$ and have different size Fourier transforms in each F stage. Again, before giving a general formula, the idea is illustrated with an example. In this, a mixed radix algorithm for $n = 120 = (4)(5)(6)$ is derived. First, let $n = 120$, $r = 4$, and $s = 30$; using (2.80) yields

$$F_{120} = (F_4 \otimes I_{30})\, T_{120,4}\, (I_4 \otimes F_{30})\, P_{120,4}$$

Next, let $n = 30$, $r = 5$, and $s = 6$, (2.80) now yields

$$F_{30} = (F_5 \otimes I_6)\, T_{30,5}\, (I_5 \otimes F_6)\, P_{30,5}$$

Substitution and use of the properties 2.9-2.16 as illustrated for the radix two Cooley-Tukey yields

$$F_{120} = (F_4 \otimes I_{30}) T_{120,4} (I_4 \otimes F_5 \otimes I_6)(I_4 \otimes T_{30,5})(I_{20} \otimes F_6) Q$$

$$Q = (I_4 \otimes P_{30,5}) P_{120,4}$$

Extending this example to the general case $N = \prod_{i=1}^{\alpha} n_i$ yields

$$F_N = \left(\prod_{i=1}^{\alpha} \left(I_{(\prod_{k<i} n_k)} \otimes F_{n_i} \otimes I_{(\prod_{k>i} n_k)} \right) \left(I_{(\prod_{k<i} n_k)} \otimes T_{(\prod_{k \geq i} n_k), n_i} \right) \right) Q \tag{2.86}$$

Length	Mult.	Adds	Total Ops.
64	204	930	1134
128	900	2242	3142
256	2052	4354	6406
512	3524	10978	14502
1024	11780	26370	38150

Table 2.5: Mixed Radix Performance

$$Q = \prod_{i=\alpha}^{1} \left(I_{(\prod_{k<i} n_k)} \otimes P_{(\prod_{k \geq i} n_k), n_i} \right)$$

Note that this expression is a generalization of both (2.84) and (2.85) given above. Table 2.5 summarizes performance.

Pease Algorithm

Pease [26] used the group structure of the stride permutation matrices which arise in the radix two Cooley-Tukey algorithm to obtain an algorithm with the same data flow for each computational stage. As an example, start with the Cooley-Tukey algorithm as derived for the case $n = 2^3$. This was shown to be

$$F_8 = (F_2 \otimes I_4) T_{8,2} (I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_{4,2})(I_4 \otimes F_2) Q \tag{2.87}$$

$$Q = (I_2 \otimes P_{4,2}) P_{8,2}$$

Take the transpose of this algorithm by using $F_n = F_n^t$ and the transposition property (2.14) of the tensor product. This yields

$$F_8 = Q^t (I_4 \otimes F_2)(I_2 \otimes T_{4,2})(I_2 \otimes F_2 \otimes I_2) T_{8,2} (F_2 \otimes I_4) \tag{2.88}$$

$$Q^t = P_{8,4} (I_2 \otimes P_{4,2})$$

Use the commutation theorem 2.16. In particular, the fact that

$$(F_2 \otimes I_4) = P_{8,2} (I_4 \otimes F_2) P_{8,4}$$

$$(I_2 \otimes F_2 \otimes I_2) = P_{8,4} (I_4 \otimes F_2) P_{8,2}$$

The substitution of these into (2.88) yields

$$F_8 = Q^t(I_4 \otimes F_2)(I_2 \otimes T_{4,2})P_{8,4}(I_4 \otimes F_2)P_{8,2}T_{8,2}P_{8,2}(I_4 \otimes F_2)P_{8,4} \quad (2.89)$$

Next, use the group structure of the stride permutation matrices as described in section 2.1.2. This yields

$$P_{8,2} = P_{8,4}P_{8,4} \quad (2.90)$$

The substitution of (2.90) into (2.89) and the introduction of the factors

$$T^2 = (I_2 \otimes T_{4,2})P_{8,4}$$

$$T^1 = P_{8,4}T_{8,2}P_{8,2}$$

yields

$$F_8 = Q^t\left[(I_4 \otimes F_2)\right]\left[T^2(I_4 \otimes F_2)P_{8,4}\right]\left[T^1(I_4 \otimes F_2)P_{8,4}\right] \quad (2.91)$$

where computational stages are partitioned by the square brackets. Note, as expected, that each stage exhibits the same data flow and only the twiddle factors are a function of the stage. This approach is generalized by the following formula

$$F_N = Q^t \prod_{i=\alpha}^{1}\left[P_{2^\alpha,2^{\alpha-1}}(T^i)(I_{2^{\alpha-1}} \otimes F_2)P_{2^\alpha,2^{\alpha-1}}\right] \quad (2.92)$$

$$T^i = P_{2^\alpha,2^{\alpha-i}}\left(I_{2^{i-1}} \otimes T_{2^{\alpha-i+1},2}\right)P_{2^\alpha,2^i}$$

$$Q^t = \prod_{i=1}^{\alpha}\left(I_{2^{i-1}} \otimes P_{2^{\alpha-i+1},2^{\alpha-i}}\right)$$

The arithmetic performance of the Pease algorithms will be the same as for radix two Cooley-Tukey.

Stockham Auto-Sort Algorithm

In the Stockham auto-sort algorithm, which is described in [23], a variation of expression (2.86) is used. The variation distributes the permutation Q

through the algorithm. The approach is illustrated below. Begin with expression (2.80)

$$F_n = (F_s \otimes I_r) T_{n,s} (I_s \otimes F_r) P_{n,s}$$

By using the properties

$$(I_s \otimes F_r) = P_{n,s}(F_r \otimes I_s)P_{n,r}$$

$$P_{n,r}P_{n,s} = I_n$$

This is readily re-expressed as

$$F_n = (F_s \otimes I_r)T_{n,s}P_{n,s}(F_r \otimes I_s) \tag{2.93}$$

By iterating (2.93), as shown above, the permutation Q is distributed throughout the algorithm. The general case formula for the Auto-Sort algorithm $N = \prod_{i=1}^{\alpha} n_i$ is

$$F_N = \prod_{i=1}^{\alpha} \left(F_{n_i} \otimes I_{N/n_i}\right)\left(I_{(\prod_{k<i} n_k)} \otimes T_{(\prod_{k\geq i} n_k),n_i}\right)\left(I_{(\prod_{k<i} n_k)} \otimes P_{(\prod_{k\geq i} n_k),n_i}\right) \tag{2.94}$$

The arithmetic performance of the Stockham algorithms is the same as for mixed radix Cooley-Tukey.

Split-Radix Algorithm

The Split radix algorithm is not a new approach for deriving FFT algorithms , but has recently received considerable attention [27,28,29,59]. For this method, a given stage $(I_r \otimes F_s)$ is not thought of as a succession of r Fourier transforms, but instead the radix of the transform is split within the stage. As an example of this consider

$$(I_2 \otimes F_s) = \begin{bmatrix} F_s & \\ & F_s \end{bmatrix}$$

This can be written as

$$\begin{bmatrix} F_s & \\ & F_s \end{bmatrix} = \begin{bmatrix} F_s & \\ & I_s \end{bmatrix}\begin{bmatrix} I_s & \\ & F_s \end{bmatrix}$$

Using the direct sum representation introduced in section 2.1.2, this can be re-written as

$$(I_2 \otimes F_s) = (F_s \oplus I_s)(I_s \oplus F_s) \tag{2.95}$$

But for s composite (2.80) yields

$$F_s = (F_{s1} \otimes I_{s2})\, T_{s,s1}\, (I_{s1} \otimes F_{s2})\, P_{s,s1} \tag{2.96}$$

Substitution of (2.96) into (2.95) yields

$$(I_2 \otimes F_s) = (F_s \oplus I_s)(I_s \oplus ((F_{s1} \otimes I_{s2})\, T_{s,s1}\, (I_{s1} \otimes F_{s2})\, P_{s,s1})) \tag{2.97}$$

So that the stage $(I_2 \otimes F_s)$ is implemented using F_s, F_{s1}, and F_{s2}.

Using this idea the split radix algorithm for $n = 2^3$ is derived by letting $n = 8$, $r = 4$, $s = 2$, and using (2.80), this yields

$$F_8 = (F_2 \otimes I_4)\, T_{8,2}\, (I_2 \otimes F_4)\, P_{8,2} \tag{2.98}$$

Next use

$$(I_2 \otimes F_4) = (F_4 \oplus I_4)(I_4 \oplus F_4)$$

and

$$F_4 = (F_2 \otimes I_2)\, T_{4,2}\, (I_2 \otimes F_2)\, P_{4,2}$$

and substitute back in (2.98) to yield

$$F_8 = (F_2 \otimes I_4) T_{8,2} (F_4 \oplus I_2) \Gamma P_{8,2} \tag{2.99}$$

$$\Gamma = [(I_2 \oplus (F_2 \otimes I_2))(I_2 \oplus T_{4,2})(I_2 \oplus (I_2 \otimes F_2))(I_2 \oplus P_{4,2})]$$

In order to derive the general case split radix factorization, start with the radix r algorithm

$$F_n = \left(\prod_{i=1}^{\alpha} (I_{r^{i-1}} \otimes F_r \otimes I_{r^{\alpha-i}}) \left(I_{r^{i-1}} \otimes T_{r^{\alpha-i+1},r} \right) \right) Q \tag{2.100}$$

$$Q = \prod_{i=\alpha}^{1} \left(I_{r^{i-1}} \otimes P_{r^{\alpha-i+1},r} \right)$$

and use the fact that

$$(I_{r^{i-1}} \otimes F_r) = \prod_{k=0}^{r^{i-1}-1} (I_{rk} \oplus F_r \oplus I_{r^{i-1}(r-k-1)}) \tag{2.101}$$

The substitution of (2.101) into (2.100) yields

$$F_n = \left(\prod_{i=1}^{\alpha} (\Gamma_i \otimes I_{r^{\alpha-i}}) \left(I_{r^{i-1}} \otimes T_{r^{\alpha-i+1},r}\right)\right) Q \tag{2.102}$$

$$\Gamma_i = \left[\prod_{k=0}^{r^{i-1}-1} (I_{rk} \oplus F_r \oplus I_{r^{i-1}(r-k-1)})\right]$$

$$Q = \prod_{i=\alpha}^{1} \left(I_{r^{i-1}} \otimes P_{r^{\alpha-i+1},r}\right)$$

where each F_r can be factored further, at will.

2.3.3 Multiplicative Algorithms

In the previous subsection various algorithms for FFT, which made use of the additive structure of the indexing set, were presented. In this subsection algorithms which make use of the multiplicative structure of the indexing set are developed. Although multiplicative algorithms usually requires fewer arithmetic operations than additive algorithms, they tend to have more complex addressing requirements. Also, it is worth noting that, the transform lengths are usually more restrictive than for their additive counterparts.

Rader Algorithm

The additive algorithms provide fast algorithms for transform lengths n, composite. Note that if the transform length required n is a prime, these methods do not apply. Rader's algorithm [30] addresses this problem by proposing a FFT algorithm for precisely this case.

Recall that the symmetry of the DFT matrix F_n is given by the multiplicative group of Z_n. Recall further that Z_p, for p prime, is the Galios field $GF(p)$ and its multiplicative group is cyclic. Using these results it is readily seen, for n prime, that the submatrix of F_n obtained by eliminating the first row and column is skew-circulant. Rader was aware of this and proposed that this submatrix be computed using the convolution theorem 2.1.20. Hence, in order to implement F_n for n prime, it is possible to use a transform of length $n-1$ which is composite and applicable to the additive techniques of the previous subsection. The approach is illustrated by

deriving a five point Rader algorithm. Note this development and notation parallels [2].

By definition a five point DFT is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ \omega^0 & \omega^2 & \omega^4 & \omega^1 & \omega^3 \\ \omega^0 & \omega^3 & \omega^1 & \omega^4 & \omega^2 \\ \omega^0 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{2.103}$$

Re-order the row and columns in terms of the primitive element 2. This introduces the permutation

$$\phi : (0 \to 0), (1 \to 1), (2 \to 2), (3 \to 4), (4 \to 3)$$

Applying P_ϕ to the row and columns of (2.103) yields

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_4 \\ y_3 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^4 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^3 & \omega^1 \\ \omega^0 & \omega^4 & \omega^3 & \omega^1 & \omega^2 \\ \omega^0 & \omega^3 & \omega^1 & \omega^2 & \omega^4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_4 \\ x_3 \end{bmatrix} \tag{2.104}$$

In this form the F_5 can now be factored as

$$\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^4 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^3 & \omega^1 \\ \omega^0 & \omega^4 & \omega^3 & \omega^1 & \omega^2 \\ \omega^0 & \omega^3 & \omega^1 & \omega^2 & \omega^4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \omega^1 & \omega^2 & \omega^4 & \omega^3 \\ 0 & \omega^2 & \omega^4 & \omega^3 & \omega^1 \\ 0 & \omega^4 & \omega^3 & \omega^1 & \omega^2 \\ 0 & \omega^3 & \omega^1 & \omega^2 & \omega^4 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Letting

$$C_4 = \begin{bmatrix} \omega^1 & \omega^2 & \omega^4 & \omega^3 \\ \omega^2 & \omega^4 & \omega^3 & \omega^1 \\ \omega^4 & \omega^3 & \omega^1 & \omega^2 \\ \omega^3 & \omega^1 & \omega^2 & \omega^4 \end{bmatrix} \qquad A_5 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

F_5 is factored as

$$F_5 = P(1 \oplus C_4)(A_5)Q \tag{2.105}$$

Length	Mult.	Adds	Total Ops.
5	16	56	72
7	24	108	132
11	104	428	532
13	180	344	524
17	144	392	536

Table 2.6: Rader Performance

Further, since C_4 is skew circulant in structure, use theorem 2.1.20 to factor it as

$$C_4 = F_4 D F_4 \tag{2.106}$$

where

$$D = \left[F_4^{-1}\right] \begin{bmatrix} \omega^1 \\ \omega^2 \\ \omega^4 \\ \omega^3 \end{bmatrix}$$

The substitution of (2.106) into (2.105) and re-distribution of the resulting terms yields the Rader algorithm

$$F_5 = P(1 \oplus F_4)(1 \oplus D)(1 \oplus F_4)A_5 Q$$

Of course, each F_4 can be further decomposed using the additive techniques of the previous subsection. This procedure is readily generalized for arbitrary primes. The general formula is derived in [2] and is given by

$$F_p = P(1 \oplus F_{p-1})(1 \oplus D)(1 \oplus F_{p-1})A_p Q \tag{2.107}$$

where

$$A_p = \begin{bmatrix} 1 & 1_{p-1}^t \\ -1_{p-1} & I_{p-1} \end{bmatrix}$$

and 1_n is an n dimensional vector of unity entries, I_n is the n by n identity matrix. The performance of some Rader algorithms is shown in table 2.6. The arithmetic counts were calculated using small Winograd cores, the Good-Thomas algorithm, and mixed radix algorithms.

When Rader introduced his approach the use of the convolution theorem was the primary method for implementing circulant matrices. Many other techniques have since been introduced. Any of these may be used to obtain algorithms.

Good-Thomas Algorithm

The Good-Thomas Algorithm [31,32] is totally analogous to the Agarwal-Cooley algorithm which expressed a one dimensional convolution as a two dimensional convolution. As was the case for convolution the approach is valid for length $n = n_0 n_1$ with the additional restriction that $GCD(n_0, n_1) = 1$.

The CRT establishes the ring isomorphism

$$Z_n \cong Z_{n0} \times Z_{n1} \tag{2.108}$$

The isomorphism is established by making use of the index maps

$$j = e_0 j_0 + e_1 j_1 \qquad 0 \le j_0, k_0 < n_0$$

$$k = e_0 k_0 + e_1 k_1 \qquad 0 \le j_1, k_1 < n_1$$

The substitution of these into the definition of the DFT yields

$$Y(e_0 k_0 + e_1 k_1) = \sum_{j_0} \sum_{j_1} \omega_n^{(e_0 j_0 + e_1 j_1)(e_0 k_0 + e_1 k_1)} X(e_0 j_0 + e_1 j_1) \tag{2.109}$$

$$Y(e_0 k_0 + e_1 k_1) = \sum_{j_0} \sum_{j_1} \omega_n^{e_0^2 j_0 k_0} \omega_n^{e_0 e_1 j_0 k_1} \omega_n^{e_0 e_1 j_1 k_0} \omega_n^{e_1^2 j_1 k_1} X(e_0 j_0 + e_1 j_1)$$

Using the fact that

$$e_0^2 = e_0 \qquad e_1^2 = e_1 \qquad e_1 e_0 = 0$$

and

$$\omega_n^{e_0 j_0 k_0} = \omega_{n0}^{g_0 j_0 k_0}$$

$$\omega_n^{e_1 j_1 k_1} = \omega_{n1}^{g_1 j_1 k_1}$$

yields

$$Y(e_0 k_0 + e_1 k_1) = \sum_{j_0} (\omega_{n0}^{g_0 j_0 k_0} (\sum_{j_1} \omega_{n1}^{g_1 j_1 k_1} X(e_0 j_0 + e_1 j_1)))$$

Length	Mult.	Adds	Total Ops.
105	932	1192	2924
120	508	2028	2536
240	1256	4656	3284
252	2416	5408	7824
315	3776	7516	11292

Table 2.7: Good-Thomas Performance [34]

To within permutations, this expression is identical to that of the additive FFT (2.78), except now, the twiddle stage is no longer present. This observation permits the use of the tensor product expression (2.80) designed for the additive algorithms in order to describe the structure of the Good-Thomas algorithm. The tensor product factorization is

$$F_n = Q(F_{n0} \otimes I_{n1})(I_{n0} \otimes F_{n1})P \tag{2.110}$$

The diagonal multiplication stage is no longer present and the permutations matrices are now given by (2.108). By using the distributive property 2.12 it is possible to express this factorization as

$$F_n = Q(F_{n0} \otimes F_{n1})P \tag{2.111}$$

which clearly reveals its two dimensional structure. Table 2.7 summarizes the performance of some Good-Thomas algorithms for complex input data.

Winograd Algorithm

The Winograd approach to small FFT algorithms [6] is a generalization of the Rader algorithm. The method uses his cyclic convolution algorithms as derived in section 2.2.3. Recall that this procedure uses the Chinese Remainder Theorem for polynomials in order to decompose a multiplication in the ring $F[x]/(x^n-1)$ into multiplications in the subrings $F[x]/(x^n-1) = \sum_i \oplus F[x]/m_i(x)$. Once small FFT algorithms are determined using this approach large algorithms are built up from these small ones by making use of a nesting procedure also due to Winograd.

There are two key observations which underlie the success of the Winograd's small FFT approach. First, there is the fact that under certain permutations of its rows and columns, the coefficients of the Fourier transform matrix exhibit cyclic structure. This permits the use of cyclic convolution algorithms in order to compute the DFT. This observation is made precise by theorem 2.1.17 of section 2.1.6. Second, by using Winograd convolution algorithms it is guaranteed that, under the homomorphism established by the Chinese Remainder Theorem, the complex coefficients of the DFT matrix become either pure real or pure imaginary numbers. Below, the procedure is illustrated for the case of F_9.

- **Example 2.3.1** Four point linear convolution

  By theorem 2.1.17 it is established that $M_9 \cong Z_6$, hence there exists a 6 by 6 submatrix of F_9 whose symmetry is circulant. This fact is made apparent by applying permutations to the rows and columns of F_9 to obtain

$$\begin{bmatrix} y_0 \\ y_3 \\ y_6 \\ y_1 \\ y_2 \\ y_4 \\ y_8 \\ y_7 \\ y_5 \end{bmatrix} = \left[\begin{array}{ccc|ccc|ccc} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^0 & \omega^0 & \omega^3 & \omega^6 & \omega^3 & \omega^6 & \omega^3 & \omega 6 \\ \omega^0 & \omega^0 & \omega^0 & \omega^6 & \omega^3 & \omega^6 & \omega^3 & \omega^6 & \omega^3 \\ \hline \omega^0 & \omega^3 & \omega^6 & \omega^1 & \omega^5 & \omega^7 & \omega^8 & \omega^4 & \omega^2 \\ \omega^0 & \omega^6 & \omega^3 & \omega^2 & \omega^1 & \omega^5 & \omega^7 & \omega^8 & \omega^4 \\ \omega^0 & \omega^3 & \omega^6 & \omega^4 & \omega^2 & \omega^1 & \omega^5 & \omega^7 & \omega^8 \\ \hline \omega^0 & \omega^6 & \omega^3 & \omega^8 & \omega^4 & \omega^2 & \omega^1 & \omega^5 & \omega^7 \\ \omega^0 & \omega^3 & \omega^6 & \omega^7 & \omega^8 & \omega^4 & \omega^2 & \omega^1 & \omega^5 \\ \omega^0 & \omega^6 & \omega^3 & \omega^5 & \omega^7 & \omega^8 & \omega^4 & \omega^2 & \omega^1 \end{array}\right] \begin{bmatrix} x_0 \\ x_3 \\ x_6 \\ x_1 \\ x_5 \\ x_7 \\ x_8 \\ x_4 \\ x_2 \end{bmatrix}$$

  The 6 by 6 circulant submatrix is computed using a Winograd 6 point convolution algorithm as described in section 2.2.3. The remaining blocks are essentially 3 point Fourier transforms. Again, these are computed using a Winograd 2 point convolution. Applying these techniques a matrix factorization is obtained. This was reproduced from [1] with $\theta = 2\pi/9$.

$$F_9 = CD_9A$$

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & -1 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 & 2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 1 & 1 & 1 & 0 & 0 & -j & -j & -j \\ 1 & 0 & -1 & 1 & 0 & -1 & j & 0 & -j & -j & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & j & 0 & 0 & 0 \\ 1 & 0 & -1 & 1 & -1 & 0 & -j & 0 & j & j & -j \\ 1 & 0 & -1 & 1 & -1 & 0 & -j & 0 & -j & -j & j \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & -j & 0 & 0 & 0 \\ 1 & 0 & -1 & 1 & 0 & -1 & j & 0 & j & j & 0 \\ 1 & 0 & -1 & 1 & 1 & 1 & 0 & 0 & -j & -j & -j \end{bmatrix}$$

$$D_9 = diag \begin{bmatrix} 1 \\ 3/2 \\ -1/2 \\ 1/3(2cos(\theta) - cos(2\theta) - cos(4\theta)) \\ 1/3(cos(\theta) + cos(2\theta) - 2cos(4\theta)) \\ 1/3(cos(\theta) - 2os(2\theta) + cos(4\theta)) \\ sin(3\theta) \\ sin(3\theta) \\ -sin(\theta) \\ -sin(4\theta) \\ -sin(2\theta) \end{bmatrix}$$

□

Table 2.8 summarizes the performance of some small Winograd FFT for complex valued input.

Length	Mult.	Adds	Total Ops.
5	10	34	44
7	16	72	88
8	4	52	56
9	20	88	108
16	20	148	168

Table 2.8: Small Winograd FFT Performance [7]

Once small FFT algorithms are designed by the procedure outlined above, Winograd binds these algorithms together by using a technique which is a generalization of the Good-Thomas approach. His idea is illustrated by starting with the Good-Thomas algorithm. Recall for n_1, n_2 relatively prime this is

$$F_n = Q(F_{n1} \otimes F_{n2})P \tag{2.112}$$

As illustrated above each of F_{n1} and F_{n2} can be factored using the Winograd small FFT approach. This yields

$$F_{n1} = C_{n1}D_{n1}A_{n1} \tag{2.113}$$

$$F_{n2} = C_{n2}D_{n2}A_{n2}$$

The substitution of these equations into (2.112) yields

$$F_n = Q(C_{n1}D_{n1}A_{n1} \otimes C_{n2}D_{n2}A_{n2})P \tag{2.114}$$

Finally, by using the distributive property of the tensor product, this expression is factored as

$$F_n = Q(C_{n1} \otimes C_{n2})(D_{n1} \otimes D_{n2})(A_{n1} \otimes A_{n2})P \tag{2.115}$$

Table 2.9 summarizes the performance of some large Winograd FFT algorithms assuming complex valued inputs.

Length	Mult.	Adds	Total Ops.
105	322	2418	2740
120	276	2076	2376
240	632	5016	5648
252	784	6640	7424
315	1186	10406	115926

Table 2.9: Large Winograd FFT Performance [34]

Kolba-Parks Algorithm

Kolba and Parks [33] made the observation that for machines on which the multiply time is almost the same as the add time, it is advantageous to use the small Winograd algorithms directly with the Good-Thomas approach and not perform the redistribution of factors as called for by the Winograd nesting. In the language of the tensor product their algorithms are obtained by starting with the Good-Thomas algorithm

$$F_n = Q(F_{n1} \otimes F_{n2})P \tag{2.116}$$

$$F_n = Q(F_{n1} \otimes I_{n2})(I_{n1} \otimes F_{n2})P \tag{2.117}$$

and then using the Winograd small factorizations

$$F_{n1} = C_{n1}D_{n1}A_{n1} \tag{2.118}$$

$$F_{n2} = C_{n2}D_{n2}A_{n2}$$

This yields

$$F_n = Q(C_{n1}D_{n1}A_{n1} \otimes I_{n2})(I_{n1} \otimes C_{n2}D_{n2}A_{n2})P \tag{2.119}$$

Table 2.10 summarizes the performance of some of these algorithms assuming complex input data.

Tolimieri-Lu Algorithms

Tolimieri *et al* [2] and Lu [50] use the multiplicative structure of the indexing set, and a variety of clever matrix factorizations, to derive several

Length	Mult.	Adds	Total Ops.
105	590	2214	2804
120	460	2076	2536
240	1100	4812	5912
252	1136	6064	7200
315	2050	8462	10512

Table 2.10: Kolba-Parks Performance [34]

algorithms for the case n prime and $n = pq$, where p and q are distinct primes. Below, some of their results are summarized.

Consider the case n a prime; by the Rader algorithm the matrix F_p is expressible as

$$F_p = P(1 \oplus F_{p-1})(1 \oplus D)(1 \oplus F_{p-1})A_pQ \tag{2.120}$$

where

$$A_p = \begin{bmatrix} 1 & 1_{p-1}^t \\ -1_{p-1} & I_{p-1} \end{bmatrix}$$

and 1_n is an n dimensional vector of unity entries, I_n is the n by n identity matrix. Given this factorization, introduce the matrix

$$B_p = FA_pF^{-1} = \begin{bmatrix} 1 & e_{p-1}^t \\ -(p-1)e_{p-1} & I_{p-1} \end{bmatrix}$$

where e_n is an n dimensional vector with a one in the first position and zeros elsewhere. By substituting this into (2.120) an alternate factorization of F_p is obtained. This is

$$F_p = P(1 \oplus F_{p-1})(1 \oplus D)B_p(1 \oplus F_{p-1})Q \tag{2.121}$$

Tolimieri-Lu refer to expression (1.121) as variant 2 of the Rader algorithm.

Another variant of the Rader algorithm is obtained from

$$F_p = P(1 \oplus C_{p-1})(A_p)Q \tag{2.122}$$

Length	Mult.	Adds	Total Ops.
5	18	44	62
7	26	88	418
11	106	392	498
13	182	268	450
17	146	332	478

Table 2.11: Tolimieri-Lu-Rader Variant 2 Performance

by observing that C_{p-1} is always of the form

$$\begin{bmatrix} X_{(p-1)/2} & X^*_{(p-1)/2} \\ X^*_{(p-1)/2} & X_{(p-1)/2} \end{bmatrix}$$

These general complex terms can be changed into a pure real or a pure imaginary number by introducing the factorization

$$\begin{bmatrix} X_{(p-1)/2} & X^*_{(p-1)/2} \\ X^*_{(p-1)/2} & X_{(p-1)/2} \end{bmatrix} =$$

$$\begin{bmatrix} I_{(p-1)/2} & I_{(p-1)/2} \\ I_{(p-1)/2} & -I_{(p-1)/2} \end{bmatrix} \begin{bmatrix} Re(X_{(p-1)/2}) & \\ & Im(X_{(p-1)/2}) \end{bmatrix} \begin{bmatrix} I_{(p-1)/2} & I_{(p-1)/2} \\ I_{(p-1)/2} & -I_{(p-1)/2} \end{bmatrix}$$

letting

$$Y_{p-1} = \begin{bmatrix} 1/2(X_{(p-1)/2} + X^*_{(p-1)/2}) & \\ & 1/2(X_{(p-1)/2} - X^*_{(p-1)/2} \end{bmatrix}$$

$$H_{p-1} = \begin{bmatrix} I_{(p-1)/2} & I_{(p-1)/2} \\ I_{(p-1)/2} & -I_{(p-1)/2} \end{bmatrix}$$

and substituting into

$$F_p = P(1 \oplus C_{p-1})A_pQ$$

results in the factorization

$$F_p = P(1 \oplus H_{p-1})(1 \oplus Y_{p-1})(1 \oplus H_{p-1})(A_p)Q \qquad (2.123)$$

which Tolimieri-Lu call variant 3.

Length	Mult	Adds	Total Ops.
5	16	40	56
7	36	72	108
11	100	160	260
13	144	216	360
17	256	332	588
19	361	432	793

Table 2.12: Tolimieri-Lu-Rader Variant 3 Performance [2]

Length	Mult.	Adds	Total Ops.
5	16	32	48
7	36	60	96
11	100	140	240
13	144	192	336
17	256	320	576
19	361	396	757

Table 2.13: Tolimieri-Lu-Rader Variant 4 Performance [2]

Finally, by introducing the matrix

$$B' = HA_pH^{-1} = \begin{bmatrix} 1 & f_{p-1}^t \\ -2f_{p-1} & I_{p-1} \end{bmatrix}$$

where f_n is the n dimensional vector whose first $n/2$ terms are one, and zeros elsewhere, the variant 4 algorithms is obtained. This is

$$F_p = P(1 \oplus H_{p-1})(1 \oplus Y_{p-1})(B'_p)(1 \oplus H_{p-1})Q \qquad (2.124)$$

An analogous procedure is carried out for the case $n = pq$, p and q distinct primes. Again the results are summarized below. For a complete derivation and examples see [2].

- Fundamental Factorization

$$F_p = PCAQ \tag{2.125}$$

where

$$C = 1 \oplus C_{p-1} \oplus C_{q-1} \oplus (C_{p-1} \otimes C_{q-1})$$

$$A = \begin{bmatrix} A_p & (A_p \otimes 1_{p-1}^t) \\ (A_p \otimes -1_{p-1}) & (A_p \otimes I_{p-1}) \end{bmatrix}$$

and C_{p-1}, A_p, 1_n are as defined for (2.120).

- Variant One

$$F_\pi = FDFA \tag{2.126}$$

where

$$D = 1 \oplus D_{p-1} \oplus D_{q-1} \oplus (D_{p-1} \otimes D_{q-1})$$

$$D_p = F_{p-1}^{-1} C_p F_{p-1}^{-1}$$

and

$$F = 1 \oplus F_{p-1} \oplus F_{q-1} \oplus (F_{p-1} \otimes F_{q-1})$$

- Variant Two

$$F_\pi = FDBF \tag{2.127}$$

where

$$B = FAF^{-1} = \begin{bmatrix} B_p & (B_p \otimes e_{p-1}^t) \\ -(p-1)(B_p \otimes e_{p-1}) & (B_p \otimes I_{p-1}) \end{bmatrix}$$

and e_n, B_p are as defined for (2.121).

- Variant Three

$$F_\pi = HYHA \tag{2.128}$$

where

$$H = 1 \oplus \left(F_2 \otimes I_{(p-1)/2}\right) \oplus \left(F_2 \otimes I_{(q-1)/2}\right) \oplus \left[\left(F_2 \otimes I_{(p-1)/2}\right) \otimes \left(F_2 \otimes I_{(q-1)/2}\right)\right]$$

$$Y = 1 \oplus Y_{p-1} \oplus Y_{q-1} \oplus (Y_{p-1} \otimes Y_{q-1})$$

and

$$Y_p = \frac{1}{2}\left[(X_p + X_p^*) \otimes (X_p - X_p^*)\right]$$

- Variant Four

$$F_\pi = HYB'H \tag{2.129}$$

where

$$B' = HAH^{-1} = \begin{bmatrix} B'_p & (B'_p \otimes f^t_{q-1}) \\ -2(B'_p \otimes f_{q-1}) & (B'_p \otimes I_{q-1}) \end{bmatrix}$$

and B'_p, f_q is as for (2.124).

It should also be mentioned that in [2] these results are extended to the cases; $n = mr$, m and r relatively prime and r prime and $n = 4s$, s a product of distinct primes. Also, in a recent paper, Tolimieri *et al* [35] introduce algorithms for $n = p^k$, for p a prime.

2.3.4 Artificial Intelligence Approaches

The additive and multiplicative approaches provide a great variety of design techniques for FFT algorithms. When using either of these methods the number of algorithms attainable grows rapidly as the transform length increases. In fact, for even modest length transforms, a direct listing of all possible algorithms would be almost impossible. In the artificial intelligence approach to FFT design, algorithms are automatically generated by a computer program and chosen on the basis of some "target" machine dependent design criteria.

Johnson-Burrus

The Johnson-Burrus [38] method uses the Good-Thomas algorithm, the Winograd small FFT approach, and the distributive property of the tensor product, to generate a large class of FFT algorithms. The class of algorithms derived using this approach include the Good-Thomas and the large Winograd algorithms as extreme cases. As an example of the approach, consider the Good-Thomas algorithm

$$F_n = Q(F_{n0} \otimes F_{n1})P \tag{2.130}$$

By the Winograd small FFT method, these can be factored as

$$F_{n0} = A_{n0}D_{n0}C_{n0} \tag{2.131}$$

$$F_{n1} = A_{n1} D_{n1} C_{n1}$$

By substituting (2.131) into (2.130) and repeatedly using the distributive property it is possible to generate an entire class of algorithms. A partial listing of some possible algorithms is given below.

- From Good-Thomas

$$F_n = Q(F_{no} \otimes I_{n1})(I_{n0} \otimes F_{n1})P$$

- Algorithm 1

$$F_n = Q(A_{n0} D_{n0} C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1} D_{n1} C_{n1})P$$

- Algorithm 2

$$F_n = Q(A_{n0} \otimes I_{n1})(D_{n0} C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1} D_{n1} C_{n1})P$$

- Algorithm 3

$$F_n = Q(A_{n0} D_{n0} \otimes I_{n1})(C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1} D_{n1} C_{n1})P$$

- Algorithm 4

$$F_n = Q(A_{n0} D_{n0} C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1})(I_{n0} \otimes D_{n1} C_{n1})P$$

- Algorithm 5

$$F_n = Q(A_{n0} D_{n0} C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1} D_{n1})(I_{n0} \otimes C_{n1})P$$

- Algorithm 6

$$F_n = Q(A_{n0} \otimes I_{n1})(D_{n0} \otimes I_{n1})(C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1} D_{n1} C_{n1})P$$

- Algorithm 7

$$F_n = Q(A_{n0} \otimes I_{n1})(D_{n0} C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1})(I_{n0} \otimes D_{n1} C_{n1})P$$

- Algorithm 8

$$F_n = Q(A_{n0} \otimes I_{n1})(D_{n0}C_{n0} \otimes I_{n1})(I_{n0} \otimes A_{n1}D_{n1})(I_{n0} \otimes C_{n1})P$$

$$\vdots$$

As pointed out in [37], the number of possible algorithms attainable by using this approach grows extremely fast. In the general case $N = \prod_{i=1}^{\alpha} n_i$ where each F_{n_i} is factored into k_i matrices, there are

$$\frac{[\sum_{i=1}^{\alpha} k_i]!}{\sum_{i=1}^{\alpha}(k_i)!}$$

possible algorithms. This makes a direct search of all possible algorithms impractical for large sequence lengths. As a solution Johnson-Burrus [37,38] use an artificial intelligence algorithm generator. Optimization parameters are set for a variety of add/multiply ratios, addressing requirements, or even for particular memory read/write cycles.

Chapter 3

The Tensor Product and Convolution

Having illustrated the use of the tensor product in section 2.2.5 and presented tensor product representations of many well known DFT algorithms in section 2.3, it seems natural to investigate if the tensor product is also of value when modeling convolution.

As is demonstrated in this chapter, this is indeed the case. In the following sections various tensor product factorizations of the linear convolution matrix are presented. These algorithms are somewhat analogous to the additive FFT algorithms of section 2.3.2. In section 3.1 a fundamental factorization is derived. Based on this, in section 3.2, a radix r linear convolution algorithm is presented. In section 3.3 several variants of the radix r algorithm are derived including: fully parallel radix r, and mixed radix in addition to fully vectorized radix r and mixed radix. In section 3.4 the performance of the algorithms derived is discussed.

3.1 The Fundamental Factorization

Let

$$h_{n-1}(x) = \sum_{j=0}^{n-1} h_j x^j \qquad d_{n-1}(x) = \sum_{k=0}^{n-1} d_k x^k$$

The linear convolution of the sequences of coefficients $y = (h_{n-1} * d_{n-1})$ is then given by the coefficients of the polynomial product

$$y_{2n-2}(x) = h_{n-1}(x)d_{n-1}(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} h_j d_k x^{j+k} \tag{3.1}$$

Assume that n is composite and $n = n_1 n_2$. In this case it is possible to make use of the additive structure of the indexing set by defining the index maps

$$\begin{aligned} j &= n_1 j_1 + j_2 \qquad & 0 \le k_1, j_1 < n_2 \\ k &= n_1 k_1 + k_2 \qquad & 0 \le k_2, j_2 < n_1 \end{aligned} \tag{3.2}$$

These maps impose a one to one correspondence between the indices $(j),(k)$ and the pairs $(j_1, j_2),(k_1, k_2)$. Using (3.2) in place of (j) and (k) yields

$$h_{n-1}(x) = \sum_{j1=0}^{n_2-1}\sum_{j2=0}^{n_1-1} h_{(n_1 j1+j2)} x^{(n_1 j1+j2)}$$

$$d_{n-1}(x) = \sum_{k1=0}^{n_2-1}\sum_{k2=0}^{n_1-1} d_{(n_1 k1+k2)} x^{(n_1 k1+k2)}$$

The polynomial product (3.1) then becomes

$$y_{2n-2}(x) = \sum_{k1=0}^{n_2-1}\sum_{j1=0}^{n_2-1}\sum_{k2=0}^{n_1-1}\sum_{j2=0}^{n_1-1} h_{(n_1 k1+k2)} d_{(n_1 k1+k2)} x^{n_1(j1+k1)+(j_2+k_2)} \tag{3.3}$$

Define

$$H_{j1} = \sum_{j2=0}^{n_1-1} h_{(n_1 j1+j2)} x^{j2}$$

$$D_{k1} = \sum_{k2=0}^{n_1-1} d_{(n_1 k1+k2)} x^{k2}$$

and re-write (3.3) as

$$y_{2n-2}(x) = \sum_{j1=0}^{n_2-1}\sum_{k1=0}^{n_2-1} H_{j1} D_{k1} x^{n_1(j1+k1)} \tag{3.4}$$

$$H_{j1} D_{k1} = \sum_{j2=0}^{n_1-1}\sum_{k2=0}^{n_1-1} h_{(n_1 j1+j2)} d_{(n_1 k1+k2)} x^{(k2+j2)}$$

In this expression each product $H_i D_i$ is a product of polynomials of degree $(n_1 - 1)$. Thus, expression (3.4) reveals that the convolution of two sequences of length $n = n_1 n_2$ is, in essence, a n_2 point convolution of polynomials of degree $(n_1 - 1)$. This simple observation makes it possible to use (3.4) as a core of fast algorithms.

In order to introduce the notion of the tensor product, express the convolution operator as a matrix multiplication. Note that the i^{th} coefficient of the polynomial $y_{2n-2}(x)$ is given by

$$y_i = \sum_k h_k d_{i-k}$$

and that the coefficients y_i and d_i are related by the matrix multiplication $\bar{y}_{2n-1} = (C_n)\bar{d}_n$ or

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_{2n-2} \end{bmatrix} = \begin{bmatrix} h_0 & 0 & \cdots & 0 \\ h_1 & h_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & \cdots & h_0 \\ 0 & h_{n-1} & \cdots & h_1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{n-1} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_{n-1} \end{bmatrix} \tag{3.5}$$

The matrix (C_n) is referred to as an n point linear convolution matrix.

Let C_{n_2} denote the matrix which corresponds to n_2 point linear convolution, and suppose there exists a matrix factorization (ex., Winograd convolution) of the form

$$C_{n_2} = (B)(D_\gamma)(A) \tag{3.6}$$

$$D_\gamma = \left[(G)\bar{h}_{n_2}\right]$$

where B and A are post and pre addition matrices, and D_γ is a diagonal matrix of degree γ whose elements are given by $(G)\bar{h}_{n_2}$ where $\bar{h}_{n_2}$ is the vector of nonzero elements of the first column of C_{n_2}. Expression (3.4) makes it possible to use (3.6) in order to obtain a factorization of $C_{n=n_1 n_2}$.

By temporarily ignoring the factor n_1 in (3.4), the tensor product construct makes it possible to "step-up" factorization (3.6) in order to factor $C_{n=n_1 n_2}$. Observe, first, that the pre-addition matrix A now operates on

polynomials of degree $(n_1 - 1)$. Using the tensor product operator, (A) becomes $(A \otimes I_{n_1})$. The second stage of algorithm (3.6) called for a γ multiplications. The multiplication of indeterminates $h_i d_i$ of (3.4) now become polynomial multiplications, each being of degree $(n_1 - 1)$. Thus, the D_γ stage becomes a direct sum of n_1 point linear convolutions. This is denoted by

$$\Gamma = \sum_{j=0}^{\gamma-1} \oplus C_{n_1,j}$$

Each of the γ n_1 point convolution matrices, $C_{n_1,j}$, is completely specified by the nonzero elements of each first column. These are given by

$$\bar{h}_{n_1,i} = (G \otimes I_{n_1})\bar{h_n}$$

where $\bar{h_n}$ is the first column of the C_n matrix. After the Γ stage the polynomials are of degree $2n_1 - 2$. Again using a tensor product, the (B) stage becomes $(B \otimes I_{2n_1-1})$.

In order to correct for the fact that n_1 in (3.4) was ignored, a 'reduction' stage of computation is needed. This is denoted by the matrix R_{n_1,n_2}. R_{n_1,n_2} is a $2n-1$ by $(2n_1-1)(2n_2-1)$ matrix of zeros and ones. The [row,column] coordinates of the ones are R_{n_1,n_2} is given by letting i and j range over the integers

$$0 \leq i < 2n_2 - 1$$

$$0 \leq j < 2n_1 - 1$$

in the expression $[(in_1 + j),(ik_1 + j)]$, where $k_1 = 2n_1 - 1$; all other entries of R_{n_1,n_2} are zero.

Thus, in order to obtain a tensor product factorization of the $C_{n=n_1 n_2}$ point linear convolution matrix, start with a core factorization of n_2 of the form

$$C_{n_2} = (B)(D_\gamma)(A)$$

$$D_\gamma = diag\left[(G)\bar{h}_{n_2}\right]$$

where B and A are addition matrices, D_γ is a diagonal matrix of degree γ, and $\bar{h}_{n_2}$ is the vector of nonzero elements of the first column of C_{n_2}. On the basis of (3.4), there exists a factorization of C_n of the form

$$C_n = R_{n_1,n_2}(B \otimes I_{2n_1-1})\Gamma(A \otimes I_{n_1}) \tag{3.7}$$

$$\Gamma = \sum_{j=0}^{\gamma-1} \oplus C_{n_1,j}$$

where R_{n_1,n_2} is a reduction matrix as defined above, and $C_{n_1,j}$ is an n_1 point linear convolution matrix; the nonzero elements of their first columns are given by

$$\bar{h}_{n_1,i} = (G \otimes I_{n_1})\bar{h}_n$$

where $\bar{h}_n$ is the vector of nonzero elements of C_n.

This expression will be called the fundamental factorization. The tensor product algorithms of the following sections are obtained by choosing special cases of this. In order to make the development a little easier to follow we will not explicitly state what the coefficients $C_{n_1,j}$ are until the final formula is given. As is many times the case for the core factorizations assume that $G = A$ in (3.6).

3.2 The Radix r Factorization

As a notational convenience introduce the operator $\hat{\otimes}$ and define it such that

$$\Gamma = \sum_{j=0}^{\gamma-1} \oplus C_{n_1,j} = (I_\gamma \hat{\otimes} C_{n_1,j})$$

Using this the fundamental factorization becomes

$$C_n = R_{n_1,n_2}(B \otimes I_{2n_1-1})(I_\gamma \hat{\otimes} C_{n_1,j})(A \otimes I_{n_1}) \tag{3.8}$$

Let

$$n = r^\alpha \qquad n_1 = r^{\alpha-1} \qquad n_2 = r$$

and substitute into (3.7), then

$$C_{r^\alpha} = R_{r^{\alpha-1},r}\Big(B \otimes I_{2(r^{\alpha-1})-1}\Big)(I_\gamma \hat{\otimes} C_{r^{\alpha-1},j})(A \otimes I_{r^{\alpha-1}}) \tag{3.9}$$

Similarly $n = r^{\alpha-1}$ $\quad n_1 = r^{\alpha-2}$ $\quad n_2 = r$ yields,

$$C_{r^{\alpha-1}} = R_{r^{\alpha-2},r}\Big(B \otimes I_{2(r^{\alpha-2})-1}\Big)(I_\gamma \hat{\otimes} C_{r^{\alpha-2},j})(A \otimes I_{r^{\alpha-2}}) \tag{3.10}$$

Substitute (3.10) into (3.9) and the use of the distributive property to get

$$C_{r^\alpha} = R_{r^{\alpha-1},r}\Big(B \otimes I_{2(r^{\alpha-1})-1}\Big)(I_\gamma \otimes R_{r^{\alpha-2},r})(I_\gamma \otimes B \otimes I_{2(r^{\alpha-2})-1}) \tag{3.11}$$

$$(I_{\gamma^2} \hat{\otimes} C_{r^{\alpha-2},j})(I_\gamma \otimes A \otimes I_{r^{\alpha-2}})(A \otimes I_{r^{\alpha-1}})$$

If this process is repeated α times it gives rise to the following matrix factorization:

Radix r algorithm

Assume $C_r = BD_\gamma A$, $D_\gamma = diag[(A)\bar{h}_r]$, then

$$C_{r^\alpha} = \tilde{B}D\tilde{A} \tag{3.12}$$

$$D = diag[(\tilde{A})\bar{h}_{r^\alpha}]$$

where

$$\tilde{A} = \prod_{k=0}^{\alpha-1} (I_{\gamma^{\alpha-1-k}} \otimes A \otimes I_{r^k})$$

$$\tilde{B} = \prod_{k=1}^{\alpha} \left(I_{\gamma^{k-1}} \otimes R_{r^{\alpha-k},r} \left(B \otimes I_{2(r^{\alpha-k})-1}\right)\right)$$

and $\bar{h}_{r^\alpha}$ is the vector of nonzero elements of the first column of C_{r^α}.

Algorithm (3.12) will be called a radix r linear convolution algorithm. It is analogous to a general radix r Cooley-Tukey FFT algorithm [21]. In the case of the FFT, r point Fourier Transforms and diagonal matrices are used to decompose an r^α point Fourier Transform matrix. Similarly for expression (3.12), r point linear convolutions and reduction matrices are used to decompose an r^α linear convolution matrix.

3.3 The Mixed Radix Factorization

By using the fundamental factorization and the radix r factorization of the previous section it is possible to derive a mixed radix decomposition of the linear convolution matrix. Like the radix r algorithm, this is analogous to the mixed radix [25] Cooley-Tukey decomposition.

By the fundamental factorization

$$C_n = R_{n_1,n_2}(B \otimes I_{2n_1-1})(I_\gamma \hat{\otimes} C_{n_1,j})(A \otimes I_{n_1}) \tag{3.13}$$

Assume that $n_1 = r_1^{\alpha_1}$ and $n_2 = r_2^{\alpha_2}$. By the radix r algorithm C_{n_1} and C_{n_2} can be decomposed as

$$C_{n_1} = \tilde{B_{n_1}} D_{\gamma_1^{\alpha_1}} \tilde{A_{n_1}} \tag{3.14}$$

and

$$C_{n_2} = \tilde{B_{n_2}} D_{\gamma_2^{\alpha_2}} \tilde{A_{n_2}} \tag{3.15}$$

The substitution of (3.15) into (3.13) yields

$$C_n = R_{n_1,n_2}\left(\tilde{B_{n_2}} \otimes I_{2n_1-1}\right)(I_{\gamma_2\alpha_2} \hat{\otimes} C_{n_1,j})\left(\tilde{A_{n_2}} \otimes I_{n_1}\right) \tag{3.16}$$

and the substitution of (3.14) into (3.16) combined with the use of the distributive property of the tensor product yields

$$C_n = R_{n_1,n_2}\left(\tilde{B_{n_2}} \otimes I_{2n_1-1}\right)(I_{\gamma_2\alpha_2} \otimes \tilde{B_{n_1}}) \tag{3.17}$$

$$(I_{\gamma_2\alpha_2} \hat{\otimes} D_{\gamma_1\alpha_1})(I_{\gamma_2\alpha_2} \otimes \tilde{A_{n_1}})\left(\tilde{A_{n_2}} \otimes I_{n_1}\right)$$

By multiplying out $\left(\tilde{B_{n_2}} \otimes I_{2n_1-1}\right)(I_{\gamma_2\alpha_2} \otimes \tilde{B_{n_1}})$ and $(I_{\gamma_2\alpha_2} \otimes \tilde{A_{n_1}})\left(\tilde{A_{n_2}} \otimes I_{n_1}\right)$ we get a mixed radix convolution algorithm.

Mixed Radix Algorithm

$$C_n = \tilde{\tilde{B}} D \tilde{\tilde{A}} \tag{3.18}$$

$$D = diag\left[(\tilde{\tilde{A}})\bar{h_n}\right]$$

where

$$\tilde{\tilde{A}} = (\tilde{A_{n_2}} \otimes \tilde{A_{n_1}})$$

$$\tilde{\tilde{B}} = R_{n_1,n_2}(\tilde{B_{n_2}} \otimes \tilde{B_{n_1}})$$

3.4 Variants of the Basic Algorithm

By exploiting the underlying algebraic structure of the tensor product representation, it is possible to modify the basic radix r factorization for efficient implementation on important parallel processing computers. The general radix r factorization was shown to be

$$C_{r^\alpha} = \tilde{B} D \tilde{A} \tag{3.19}$$

$$D = diag\left[(\tilde{A})\bar{h}_{r^\alpha}\right]$$

$$\tilde{A} = \prod_{k=0}^{\alpha-1}(I_{\gamma^{\alpha-1-k}} \otimes A \otimes I_{r^k})$$

$$\tilde{B} = \prod_{k=1}^{\alpha}\left(I_{\gamma^{k-1}} \otimes R_{r^{\alpha-k},r}\left(B \otimes I_{2(r^{\alpha-k})-1}\right)\right)$$

By making direct use of the commutation theorem of section 2.1.2 it can be shown that the structure of the pre and post addition matrices can be modified to

$$\tilde{A} = \prod_{k=0}^{\alpha-1} P_{\gamma_1 r^k, s_1}(A \otimes I_{s_1 r^k})P_{s_1 r^{k+1}, r^{k+1}}$$

$$\tilde{B} = \prod_{k=1}^{\alpha}\left(P_{t_3 t_1, t_1}\left(R_{r^{\alpha-k},r}(B \otimes I_{t_2}) \otimes I_{t_1}\right)P_{\gamma^k t_2, \gamma t_2}\right)$$

where $s_1 = \gamma^{\alpha-k-1}$, $t_1 = \gamma^{k-1}$, $t_2 = 2(r^{\alpha-k}) - 1$, and $t_3 = 2(r^{\alpha-k+1}) - 1$.

Since every tensor matrix of this algorithm is a vector stage as described in (2.1.5), these modified pre and post addition matrices can be easily implemented on a vector processor. Thus, one obtains a fully vectorized variant of the basic radix r factorization. Note that many of the techniques established in [47,48,49,2,53] for DFT can be used to implement this algorithm.

Vectorized Radix r Algorithm

$$C_n = \tilde{B}D\tilde{A} \tag{3.20}$$

$$D = diag\left[(\tilde{A})\bar{h}_{r^\alpha}\right]$$

$$\tilde{A} = \prod_{k=0}^{\alpha-1} P_{\gamma_1 r^k, s_1}(A \otimes I_{s_1 r^k})P_{s_1 r^{k+1}, r^{k+1}}$$

$$\tilde{B} = \prod_{k=1}^{\alpha}\left(P_{t_3 t_1, t_1}\left(R_{r^{\alpha-k},r}(B \otimes I_{t_2}) \otimes I_{t_1}\right)P_{\gamma^k t_2, \gamma t_2}\right)$$

where $s_1 = \gamma^{\alpha-k-1}$, $t_1 = \gamma^{k-1}$, $t_2 = 2(r^{\alpha-k})-1$, and $t_3 = 2(r^{\alpha-k+1})-1$.

In a similar way one can derive a vectorized mixed radix algorithm, a parallel radix r algorithm, as well as a parallel mixed radix algorithm. These are given below.

Parallel Radix r Algorithm

$$C_n = \tilde{B} D \tilde{A} \tag{3.21}$$

$$D = diag\left[(\tilde{A})\bar{h}_{r^\alpha}\right]$$

$$\tilde{A} = \prod_{k=0}^{\alpha-1} P_{\gamma s_1 r^k, \gamma s_1}(I_{s_1 r^k} \otimes A) P_{s_1 r^{k+1}, r^k}$$

$$\tilde{B} = \prod_{k=1}^{\alpha} \left(I_{\gamma^{k-1}} \otimes R_{r^{\alpha-k}, r}\left(P_{t_2 t_1, t_2}(I_{t_1} \otimes B) P_{t_1 \gamma, t_1}\right)\right)$$

where $s_1 = \gamma^{\alpha-1-k}$, $t_2 = (2r-1)$, **and** $t_1 = (2r^{\alpha-k}-1)$.

Vectorized Mixed Radix Algorithm

$$C_n = \tilde{\tilde{B}} D \tilde{\tilde{A}} \tag{3.22}$$

$$D = diag\left[(\tilde{\tilde{A}})\bar{h}_n\right]$$

$$\tilde{\tilde{A}} = P_{s_1 s_2, s_2}(\tilde{A_{n_1}} \otimes I_{s_2}) P_{n_1 s_2, n_1}(\tilde{A_{n_2}} \otimes I_{n_1})$$

$$\tilde{\tilde{B}} = R_{n_1, n_2}(\tilde{B_{n_2}} \otimes I_{t_1}) P_{s_2 t_1, s_2}(\tilde{B_{n_1}} \otimes I_{s_2}) P_{s_1 s_2, s_1}$$

Where $s_1 = \gamma_1^{\alpha_1}$, $s_2 = \gamma_2^{\alpha_2}$, $t_1 = (2n_1 - 1)$, **and** $\tilde{A}$ **and** $\tilde{B}$ **are vectorized radix** r **algorithms.**

Parallel Mixed Radix Algorithm

$$C_n = \tilde{\tilde{B}} D \tilde{\tilde{A}} \tag{3.23}$$

$$D = diag\left[(\tilde{\tilde{A}})\bar{h}_n\right]$$

$$\tilde{\tilde{A}} = (I_{s_2} \otimes \tilde{A_{n_1}}) P_{n_1 s_2, s_2}(I_{n_1} \otimes \tilde{A_{n_2}}) P_{n_2 n_1, n_1}$$

SIZE	DIRECT	CNV. TH.	TEN. PR.
4	25	160	25
8	113	468	99
16	481	1148	349
32	1985	2764	1155
64	8065	6796	3685
128	32513	15372	11499
256	130561	34572	35389
512	523265	79372	107955
1024	2095105	173068	327445

Table 3.1: Radix 2 Convolution vs DFT Approach

$$\tilde{\tilde{B}} = R_{n_1,n_2} P_{t_1 t_2, t_2}(I_{t_1} \otimes \tilde{B_{n_2}}) P_{s_2 t_1, t_1}(I_{s_2} \otimes \tilde{B_{n_1}})$$

Where $s_2 = \gamma_2^{\alpha_2}$, $t_1 = (2n_1 - 1)$, $t_2 = (2n_2 - 1)$, and $\tilde{A}$ and $\tilde{B}$ are parallel radix r algorithms.

3.5 Performance of Algorithms

Table (3.1) gives the total number of arithmetic operations for linear convolution by direct implementation, the convolution theorem, and a "slightly modified" radix two tensor product algorithm. These numbers are based on the assumption that one of the sequences is known and available for pre-computation prior to compute time. As is shown, the tensor product algorithm offers a considerable improvement over both of the other methods for small and intermediate size convolutions. For a 256 point linear convolution the convolution theorem and the tensor product approach break about even, after which the convolution theorem becomes advantageous. The algorithms of table one are given in appendix A. It is interesting to note that these particular radix two algorithms are slightly modified tensor product representations of Toom's [9] algorithm.

The derivation of a general radix r algorithm starts with a core factor-

ization of a r point convolution matrix. This should be of the form

$$C_r = BD_\gamma A$$

where A,B are pre and post addition matrices and D_γ is a diagonal matrix of order γ. The core gives rise to a factorization of the form

$$C_{r^\alpha} = \tilde{B} D_{\gamma^\alpha} \tilde{A}$$

where $\tilde{B}$ and $\tilde{A}$ can be expressed in terms of B and A.

Let the arithmetic complexity of a computational stage be designated by the cartesian pair (number of multiplications, number of additions) and let the arithmetic complexity of vector matrix multiplication with the matrices A,B and C_r be denoted by $A(m,a)$, $B(m,a)$, and $C_r(m,a)$, respectively. The arithmetic complexity of vector matrix multiplication with the matrix C_r is then given by

$$C_r(m,a) = B(m,a) + A(m,a) + (\gamma, 0)$$

On the basis of the formulas given in section (3.2), the arithmetic complexity of the algorithm C_{r^α} can be shown to be

$$\begin{aligned} C_{r^\alpha}(m,a) = {} & k_1 \left[A(m,a) + 2B(m,a) + (0, 2r-2)\right] \\ & -k_2 \left[B(m,a) + (0, 2r-2)\right] + (\alpha^\gamma, 0) \end{aligned}$$

where

$$k_1 = \left[\frac{\gamma^\alpha - r^\alpha}{\gamma - r}\right] \qquad k_2 = \left[\frac{\gamma^\alpha - 1}{\gamma - 1}\right]$$

An examination of this formula reveals that an important factor in controlling the performance of a radix r family is the constant γ of the underlying core. Thus, the Winograd convolution approach can be extremely useful in designing the core factorizations.

The asymptotic performance of a radix r algorithm is always best when γ is minimized. This approach, however, can lead to disappointing results when α is small. The problem is that $A(m,a)$ and $B(m,a)$ usually become quite large when γ is minimized. For intermediate size α this can eliminate the benefit of reducing γ. The solution to the problem thus lies in striking a balance between γ, $A(m,a)$ and $B(m,a)$.

Although the tensor product algorithms perform well in terms of operation count, this alone is not a sufficient measure of performance. There are many other points to consider. For instance:

- The manner in which operands are addressed can greatly affect performance. In fact, a reduction in the operations count achieved by introducing irregular permutations can sometimes significantly degrade performance. In these cases data arithmetic is traded for address computation. With this in mind note that the only permutations used by the tensor product algorithms are those characterized by stride permutations. Because these are highly regular they can be implemented easily and, in many cases, at no cost of compute cycles.

- The feasibility of realizing an efficient parallel implementation of the algorithm cannot be overlooked. Current trends in parallel computation center around the SIMD multi-processor and the vector processor configurations. Note that it is possible to easily modify the structure of a tensor product algorithm for efficient implementation on either of these configurations.

- The numerical stability of the algorithm [10] is another important issue. Extraneous constants introduced by an algorithm can often be the source of numerical errors in the final answer. In the FFT approach these constants are the roots of unity; in the Winograd approach these are rational constants introduced by the Chinese remainder theorem. In both cases these constants become ill-behaved as the transform length, n, becomes very large.

  The only constants used by the tensor product approach are introduced in the core factorizations. Since the cores are for small n, these constant are usually not a source of considerable error. Once established a core can be extended to compute arbitrarily large convolutions without the introduction of any new constants.

In chapter two important connections were established between certain tensor product constructs and their implementations on vector and multi processor computers. By developing tensor product representations of DFT it was demonstrated that these lead to a unified treatment of many algorithms. In this chapter it was shown that the structure of the linear convolution matrix also permits a tensor product decomposition. By

virtue of this representation many of the results and techniques developed for DFT become directly applicable to convolution. In the next chapter special forms of the algorithms derived here are shown to lead to highly efficient implementations on the constraints imposed by modern RISC architectures.

Chapter 4

The Tensor Product and RISC

In chapter one, important features of the modern RISC architecture were identified. In chapters two and three, a mathematical foundation of fast algorithm design for both linear convolution and DFT was developed and presented in a unified format. In this chapter, the mathematical tools of chapters two and three will be used to design algorithms which exploit the attributes pertinent to the modern RISC architecture.

The material developed in this chapter is organized into four sections. In section one the most important features of the RISC architecture, as outlined in chapter one, are briefly restated. Design criteria for fast algorithms for model I (multiply - add) and model II (multiply - accumulate) RISC architectures is established. In section two new tensor product techniques are developed and shown to be applicable to the problem of satisfying the design goals outlined for RISCs. In section three highly efficient linear convolution algorithms are designed for both model I and model II RISC machines. Analogous algorithms are developed in section four for the DFT.

4.1 Design Goals for RISC

Computational models for two modern RISC architectures were already introduced in chapter one. Particular features of these models will be exploited in order to design efficient algorithms. In this section these features are reviewed, and design goals based on these are established.

It is important to keep in mind that the attributes identified, and therefore the algorithms which exploit them, are only indirectly related to the

modern RISC architecture. These features are relevant to modern RISC machines because the elimination of the large complex control unit, a feature unique to the RISC approach, makes chip space available for floating point hardware multipliers, adders, separate fixed point addressing units, etc.. Note that the assumptions made here may very well be relevant to future CISC computers.

The assumptions regarding the underlying computer architecture for the algorithms designed in this chapter are as follows:

1. The time required to implement a floating point addition is the same as that required for a floating point multiplication.

2. Floating point multiplication and addition can be implemented in parallel. Two levels of parallelism will be considered:

   - For the multiply - add kernel (model I) there is no restriction regarding the arrangement of multiplication and addition in order to obtain parallel operation.
   - For the multiply - accumulate kernel (model II) parallelism is prevalent only when performing multiply accumulate instructions of the form $y = (a * b) + c$.

3. Fixed point operand addressing occurs in parallel with the operations performed by the floating point data unit. It will be assumed that accessing the memory by a fixed stride requires the same amount of time as accessing the memory in continuous order.

A survey of fast algorithms for convolution and DFT was presented in chapter two. A review of this work will reveal that assumptions 1,2, and 3, are quite different than those traditionally made when designing fast algorithms. For example, note that many of the Winograd convolution algorithms of section 2.2.3 and DFT algorithms of section 2.3.2 eliminate multiplications by increasing the number of additions required for a given computation. On the basis of assumption (1), however, this would not be a beneficial approach for the modern RISC architecture.

There are many other examples of fast algorithms which reduce the number of multiplications needed by increasing the number of additions required for a given computation. In fact, almost all of the work presented

in Blahut [1] is highly focused on reducing the number of multiplications required for many types of digital signal processing algorithms. Clearly, in the case of RISCs, new techniques are needed.

Since assumption (1) makes the number of additions just as important as the number of multiplications, the total number of operations becomes a more relevant measure of performance for RISC machines. Assumption (2) also effects design criteria for fast algorithms. By finding ways to properly schedule additions and multiplications a programmer can exploit the parallelism between the floating point adder and multiplier which results in enhanced performance.

For computational model I the data adder and the data multiplier operate in parallel and the programmer has independent control of both of these resources. In order to exploit this parallelism it would be beneficial if the additions and multiplications required by an algorithm were also independent. Then, fast algorithms on this model would minimize $max(m, a)$, where the cartesian pair (number of multiplications, number of additions) denotes the arithmetic complexity of an algorithm and $max(m, a)$ denotes the greater of the two integers m and a.

For computational model II the data adder and multiplier operate in parallel only when using multiply-accumulate instructions (MAC) of the form $r+(s*t)$. In order to exploit this parallelism it is necessary to establish dependency between addition and multiplications in the form of MAC instructions. Then, fast algorithms on this model would minimize $(m) + (a)+(MAC)$, where (a), (m), and (MAC) denote the number of additions, multiplications, and multiply-accumulate instructions respectively.

Another very important issue not usually taken up by algorithm designers is the fact that operand addressing has a profound effect on performance. These issues were explored in Tolimieri *et al* [2] and Johnson *et al* [53] as they pertain to the design of DFT algorithms for large vector and multiprocessor computers. Through this work the tensor product was shown to be a powerful tool for modeling and designing DFT algorithms. In chapter three analogous tensor product representations were derived for linear convolution. Through these formulations many of the design techniques developed in [2,53] become applicable to the important linear convolution operator.

By assumption three the addressing unit operates in parallel with the data unit. If the memory is accessed by a fixed stride no time is wasted

- Design Goals for Model I
  1. Make the additions and multiplications as independent as possible.
  2. Minimize $\max(m, a)$ where (m, a) denotes the arithmetic complexity of an algorithm.
  3. Devise a means of scheduling the multiplications and additions such that they can be carried out concurrently, hence exploiting the parallelism of the model.
  4. Arrange the algorithm such that operands are accessed by some fixed stride so no time is wasted doing addressing or data shuffles.
- Design Goals for Model II
  1. Establish dependency between the additions and multiplications in the form of MAC instructions.
  2. Minimize $(m) + (a) + (MAC)$.
  3. Arrange the algorithm such that operands are accessed by some fixed stride.

Figure 4.1: Design Goals for RISC

for address computation. Since address computation can consume significant amounts of processing time, it would very beneficial to organize the operands so they are indeed retrieved from memory with a fixed stride. Doing so would avoid the overhead associated with address computation. This design goal, as well as the others established in this section, are summarized in figure 4.1.

4.2 Design Strategies for RISC

Design strategies, based on the goals identified in section 4.1 are developed in this section.

4.2.1 Scheduling for Model I

Let F be a matrix and suppose there exists an algorithm to implement the product $\bar{y} = (F)\bar{x}$, where $\bar{x}$ is a vector whose arithmetic complexity is (m, a); m is the number of multiplications required and a is the number of additions. Suppose further that this algorithm is to be implemented on a model I RISC where the adder and multiplier operate in parallel. How many compute cycles are required to implement the product $\bar{y} = (F)\bar{x}$?

In order to answer this question two pieces of information are needed.

1. The magnitudes of m and a.
2. The dependence relationships between m and a.

On the basis of the dependency established by (2), the number of compute cycles required can be as small as $max(m, a)$ or as large as $m + a$. These correspond to the case of totally independent multiplications and additions and totally dependent multiplications and additions, respectively. These extreme cases are illustrated by examples 4.2.1 and 4.2.2.

- **Example 4.2.1** Fully independent arithmetic

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & g_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & g_2 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{bmatrix} \tag{4.1}$$

In this example the complexity of F is (3,3) and the number of operations required to implement $\bar{y} = (F)\bar{x}$ is $max(m, a) = max(3, 3) = 3$.

□

- **Example 4.2.2** Fully dependent arithmetic

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} g_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} \tag{4.2}$$

In this example the complexity of F is (1,3) and the number of operations required to implement $\bar{y} = (F)\bar{x}$ is $(m + a) = (1 + 3) = 4$.

□

These examples illustrate the importance of having a suitable scheduling strategy for addition and multiplication on model I RISCs. In section 2.1.4 some fundamental definitions and theorems from scheduling theory were introduced. The results established in this section provide the theoretical foundation needed in order to develop methods of properly scheduling addition and multiplication on the modern RISC.

On the RISC architecture there are several important resources that operate in parallel. Using the results established in section 2.1.4, a function f is evaluated by a sequence of subsets of the set of resources $\{R\}$. Suppose f must be computed many times, then theorem 2.1.8 asserts the existence of a scheduling strategy which achieves the minimal average latency (MAL) for the particular reservation table under consideration.

This important result can be used here by considering the floating point adder and multiplier to be two resources. Let the matrix F be the basic function to be evaluated, and consider the implementation of a tensor matrix of the form $(I_n \otimes F)$. Let the arithmetic complexity of the matrix F be given by (m, a), then an optimal scheduling strategy allows the implementation of $(I_n \otimes F)$ in $n * max(m, a)$ compute cycles. Theorem 2.1.4 guarantees near optimal strategies for this important class of tensor matrix. An example will help to clarify the technique.

- **Example 4.2.3** Scheduling $(I_n \otimes F)$ on RISC

  Let

$$F = \begin{bmatrix} 0 & p_1 & p_2 & 0 \\ p_3 & 1 & p_4 & 0 \\ p_5 & 0 & 0 & 1 \\ 0 & 1 & p_6 & 0 \end{bmatrix}$$

  Then the multiplication of F with a vector $\bar{d}$ is given by

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & p_1 & p_2 & 0 \\ p_3 & 1 & p_4 & 0 \\ p_5 & 0 & 0 & 1 \\ 0 & 1 & p_6 & 0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

Multiplier	m_1	m_2		m_3	m_4	m_5	m_6
Adder			a_1	a_2	a_3	a_4	a_5

Table 4.1: Forbidden list = {1,2,3,4,5,6}

This can be realized using the instructions:

$$\begin{array}{llll} m_1 = d_1 * p_1 & & a_1 = m_1 + m_2 \\ m_2 = d_2 * p_2 & & a_2 = m_3 + d_1 \\ m_3 = d_0 * p_3 & & a_3 = a_2 + m_4 \\ m_4 = d_2 * p_4 & & a_4 = m_5 + d_3 \\ m_5 = d_0 * p_5 & & a_5 = m_6 + d_1 \\ m_6 = d_2 * p_6 & & \end{array}$$

In this case, the arithmetic complexity of F is $(6, 5)$, the arithmetic complexity of $(I_n \otimes F)$ is $(n * 6, n * 5)$, and the optimal scheduling strategy allows the realization of $(I_n \otimes F)$ in $max(n * 6, n * 5) = 6n$ compute cycles. The MAL for this computation is given by the matrix F and is seen to be 6. The appropriate choice of reservation table will allow this minimal value to be realized.

There are many possible reservation tables for the evaluation of F. Two of these are given by tables 4.1 and 4.2 where the forbidden lists for these are also given. By lemmas 2.1.1 and 2.1.2 it is seen that table 4.2 will be optimal because $k * MAL$, $k \in I$, is not a member of the forbidden list for this choice of scheduling strategy. The reservation table 4.2 can be effectively realized using the pseudo code given in figure 4.2. Note that the number of compute cycles needed to realize $(I_n \otimes F)$ using this scheduling approach is $6n + 1$, which closely approximates the optimal $6n$ for perfectly scheduled operations.

□

Example 4.2.3 illustrates that a matrix of the form $(I_n \otimes F)$, where F is a matrix of arithmetic complexity (m, a), can be efficiently scheduled on

Multiplier	m_1	m_2	m_3	m_4	m_5	m_6	
Adder			a_1	a_2	a_3	a_4	a_5

Table 4.2: Forbidden list = {1,2,3,4,5}

Adder	Multiplier
	$m_1 = d_1 * p_1$
	$m_2 = d_2 * p_2$
$a_1 = m_1 + m_2$	$m_3 = d_0 * p_3$
$a_2 = m_3 + d_1$	$m_4 = d_2 * p_4$
$a_3 = a_2 + m_4$	$m_5 = d_0 * p_5$
$a_4 = m_5 + d_3$	$m_6 = d_2 * p_6$
Do $n-1$ times	
$a_5 = m_6 + d_1$	$m_1 = d_1 * p_1$
	$m_2 = d_2 * p_2$
$a_1 = m_1 + m_2$	$m_3 = d_0 * p_3$
$a_2 = m_3 + d_1$	$m_4 = d_2 * p_4$
$a_3 = a_2 + m_4$	$m_5 = d_0 * p_5$
$a_4 = m_5 + d_3$	$m_6 = d_2 * p_6$
end Do	
$a_5 = m_6 + d_1$	

Figure 4.2: Pseudo code for table 4.2

a model I RISC computer. The greater n is, the closer the technique approximates the $n*max(m,a)$ instructions of perfectly scheduled operations. Thus, the design criteria, minimize $max(m,a)$, is valid when n is large.

Suppose now that F represents a small convolution or DFT algorithm. A fast algorithm for F would be warranted only under the assumption that many implementations of F are needed, and therefore for small algorithms, the design criteria minimize $max(m,a)$ is very reasonable.

When implementing large DFT or convolution algorithms, the criteria $max(m,a)$ is no longer appropriate because the assumption that many will be performed is less valid. In this case it would be beneficial to decompose a large algorithm F into stages of the form

$$F = \prod_i (I_{n_i} \otimes f_i)$$

and then estimate the performance of a large algorithm by

$$\sum_i n_i * max(f_i(m,a))$$

. Both of these performance measures are used in the next section.

The use of the techniques presented in this section requires the ability to store the partial results of the matrix F. These results must be stored either in main memory or on chip. If there are enough on chip data registers available then all the partial results can be stored in these. If there are not enough on chip data registers, then some of the partial results must be stored in main memory. Using main memory to store partial results may be convenient but it also creates a greater demand for memory bandwidth. This brings us up against a fundamental problem associated with RISCs and with high performance computing in general:

> **A highly optimized compute cycle and special purpose hardware will greatly enhance arithmetic throughput but it becomes increasing difficult to increase memory bandwidth in order to sustain the high rate of computation realized by these techniques.**

The availability of on chip data registers and memory bandwidth are extremely machine dependent, and therefore it does not make sense to incorporate these issues into the theoretical development. These limitations,

however, can be very important when implementing the algorithms developed here.

As in chapter two, where particular techniques were presented for implementing tensor matrices on vector and multi-processor computers, in this section very basic results from scheduling theory are used in order to connect these ideas to features prevalent to the RISC architecture. Again the tensor product serves as the common platform for this. Recall that the commutation theorem provided the well defined method of modifying the structure of a vector stage for parallel implementation (and vice versa) by introducing very particular stride permutations. The commutation theorem plays precisely the same role here. Techniques for implementing the addressing introduced by the stride permutations are presented in the next section.

4.2.2 Addressing Strategies

In order to achieve efficient run time performance on any computer the addressing needs of an algorithm must be matched with the addressing capabilities of its addressing unit. Two important assumptions regarding the capabilities of the addressing unit of a RISC machine were made in section 4.1. First, the addressing unit is capable of parallel operation with the floating point data arithmetic unit. Second, the addressing unit is capable of accessing the memory with a stride without incurring a loss in performance.

Some important preliminary issues regarding operand addressing and the tensor product were taken up in section 2.1.5 The stride permutation matrix was at the heart of this discussion. Through the several examples presented, the stride permutation was shown to offer a method for modify the structure of tensor matrices to match the addressing capabilities of many different types of computers. In working out these examples considerable detail was omitted for the sake of clarity. Some of these details, as they pertain to the RISC addressing unit, are now presented.

As mentioned above, it is assumed that the addressing unit can access the memory with a stride at the same rate as accessing the memory in stride one order. In many cases, this implies that the RISC makes use of some type of register-direct indexed addressing mode. Thus, accessing the memory with a fixed stride s is efficient only if there is an available index register

to hold the stride value s. Although most RISC machines support this type of addressing, the availability of a suitable number of index registers can be a serious problem in actually implementing the addressing of stride permutations.

As an example of this, consider the addressing requirements for implementing the tensor stage

$$\bar{y_{4n}} = (I_n \otimes F)(P_{4n,n})\bar{x_{4n}} \tag{4.3}$$

where F is given by

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

If there are three index registers available then the addressing of the stride permutation $(P_{4n,n})$ can be realized in a highly efficient manner using the following addressing scheme.

> The address of the vector $\bar{x}$ is input and the address of the vector $\bar{y}$ is output. The registers dp1, dp2, and dp3 are data pointer registers; *dp1, *dp2, and *dp3 refer to the operands contained in these addresses. There are three index registers: indx1, indx2, and indx3. Register direct auto indexed addressing is realized by instructions of the form *dpi++indxi. Pseudo code for the efficient realization of (4.3) is then given in example 4.2.4.

- **Example 4.2.4** *Addressing of (4.3)*

```
dp1 = input
dp2 = input + 2n
dp3 = output
indx1 = n
indx2 = -n
indx3 = -n+1
for i = 1 to (n)
{
```

```
    *dp3++ = *dp1++indx1 + *dp2++indx1
    *dp3++ = *dp1++indx2 + *dp2++indx2
    *dp3++ = *dp1++indx1 - *dp2++indx1
    *dp3++ = *dp1++indx3 - *dp2++indx3
}
```

□

If n is large most of the computation for this stage occurs inside the loop. Note that for this implementation no time is wasted for address computation while looping.

Suppose now that the index register indx3 was not available for the implementation of this stage. With only two available index registers this stage can still be realized but with a considerable loss in efficiency. Pseudo code for this implementation is given in example 4.2.5.

- **Example 4.2.5** *Addressing of (4.3)*

```
dp1 = input
dp2 = input + 2n
dp3 = output
indx1 = n
indx2 = -n
for i = 1 to (n)
{
    *dp3++ = *dp1++indx1 + *dp2++indx1
    *dp3++ = *dp1++indx2 + *dp2++indx2
    *dp3++ = *dp1++indx1 - *dp2++indx1
    *dp3++ = *dp1++indx3 - *dp2++indx3
    dp1 = input + i
    dp2 = input + i + 2n
}
```

□

Note that in this implementation, the lack of another index register results in a significant increase in the number of operations required to

realize (4.3). This is because valuable compute cycles must now be used to refresh the data pointer registers dp1 and dp2 while looping. This example clearly indicates that the number of available index registers can be a very important parameter in efficiently implementing the addressing of stride permutations on RISC machines.

Another important parameter is the number of available data registers. As an illustration of this suppose that there are no index registers available but that there are several data pointer registers at hand. This can also lead to efficient implementation of $P_{4n,n}$. In particular, this stage can now be realized using the pseudo code of example 4.2.6.

- **Example 4.2.6** *Addressing of (4.9)*

```
dp1 = input
dp2 = input + n
dp3 = input + 2n
dp4 = input + 3n
dp5 = output
for i = 1 to (n)
{
    *dp5++ = *dp1 + *dp3
    *dp5++ = *dp2 + *dp4
    *dp5++ = *dp1++ - *dp3++
    *dp5++ = *dp2++ - *dp4++
}
```

□

Thus, the number of data pointer registers is also an important parameter when implementing efficient addressing schemes on RISC architectures. In fact, the results of this example indicate that there can be a trade off between the required number of increment registers and the required number of data pointer registers.

4.3 Convolution Algorithms

The major theoretical work in the area of fast algorithm design is based on multiplicative complexity theory and was developed to a large degree by

Winograd in [6,7,8]. The Winograd convolution algorithms of section 2.2.3 are a result of this approach. These algorithms are based on the Chinese remainder theorem (CRT) which was given in chapter two as theorem 2.1.16 The Winograd convolution algorithms have been shown to be generalizations of many well know algorithms including the Cook-Toom algorithm and the well known convolution theorem both of which are discussed in section 2.2.3.

The great advantage of the Winograd approach lies in the fact that it is possible to use the CRT in order to decompose the convolution operator in many different ways. A particular decomposition, and hence a new algorithm, is specified by the choice of quotient rings used to decompose the polynomial ring $F[x]/M(x)$. Particular choices of these lead to different algorithms with unique characteristics. Because this method gives a way of describing many different types of convolution algorithms, it seems reasonable to investigate if it also yields algorithms suitable for implementation on RISC architectures. The feasibility of this approach is now explored.

As an illustration of a Winograd convolution algorithm, recall example 2.2.2 for three point linear convolution. This algorithm was based on decomposing the quotient ring $F[x]/M(x)$ into the subrings $F[x]/m_0(x)$, $F[x]/m_1(x)$, and $F[x]/m_2(x)$, where

$$M(x) = x^5 - x^4 + x^3 - x^2$$

and

$$M(x) = m_0(x)m_1(x)m_2(x) = x^2(x^2+1)(x-1)$$

As pointed out in chapter two this decomposition can be characterized by the matrix factorization

$$\begin{bmatrix} h_0 & & \\ h_1 & h_0 & \\ h_2 & h_1 & h_0 \\ & h_2 & h_1 \\ & & h_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & -2 & -1 & 1 \\ 1 & 1 & 0 & -3 & 0 & -3 & 0 \\ -1 & -1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} [D] \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \tag{4.4}$$

$$[D] = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ 1/2k_4 \\ 1/2k_5 \\ 1/2k_6 \\ 1/2k_7 \end{bmatrix}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ 1/2k_4 \\ 1/2k_5 \\ 1/2k_6 \\ 1/2k_7 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}$$

The arithmetic complexity of this algorithm is $(7, 20)$ where 20 denotes the number of additions and 7 denotes the number of multiplications. Note that the multiplicative complexity of this algorithm is 7, as opposed to 9 that would be needed for a direct implementation. This reduction would be advantageous on many conventional von Neumann computer architectures in spite of the increase in the total number of operations. This is due to the fact that on these architectures floating point multiplication takes much longer than floating point addition.

Now consider the implementation of algorithm (4.4) on a modern RISC machine. In terms of the design criteria outlined in the previous section for the model I RISC, assume that many 3 point convolutions are required; then it is necessary to reduce $max(m, a)$. Note that $max(7, 20) = 20$ would be greater for algorithm (4.4) than the $max(9, 4) = 9$ which would be required for a direct implementation. Thus, this algorithm would be a very poor choice for implementation on model I.

Perhaps a different choice of quotient rings would have led to a better suited algorithm for model I. Lets repeat this example with another choice of residue polynomials. In order to use the CRT, express the three point linear convolution

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} h_0 & & \\ h_1 & h_0 & \\ h_2 & h_1 & h_0 \\ & h_2 & h_1 \\ & & h_2 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \tag{4.5}$$

as a polynomial multiplication. In particular let

$$h(x) = h_2x^2 + h_1x + h_0 \tag{4.6}$$

$$d(x) = d_2x^2 + d_1x + d_0 \tag{4.7}$$

Then (4.5) is given by the coefficients of $y(x)$ in

$$y(x) = h(x)d(x) \tag{4.8}$$

If the degree of $M(x)$ is greater than four, the linear convolution of the coefficients of (4.3) and (4.4) is given by the product

$$y(x) = h(x)d(x) \qquad \text{mod } M(x) \tag{4.9}$$

Note also that if the degree of $M(x)$ is equal to four, then linear convolution of the these coefficients is given by

$$y(x) = \{h(x)d(x) \qquad \text{mod } M(x)\} + h_2d_2\{M(x)\} \tag{4.10}$$

A polynomial $M(x)$ must be chosen; in this case let $M(x)$ be

$$M(x) = x^4 - x^2$$

and use (4.10).

$M(x)$ can be factored into a product of relatively prime polynomials $\prod m_i(x)$ as

$$M(x) = m_0(x)m_1(x) = x^2(x^2 - 1)$$

Following the procedure outlined for Winograd convolution in chapter two, this choice of factors implies that the homomorphic images of $h(x)$ and $d(x)$ are given by

$$\begin{aligned} h_0(x) &= h_1x + h_0 & \qquad d_0(x) &= d_1x + d_0 \\ h_1(x) &= h_1x + (h_0 + h_2) & \qquad d_1(x) &= d_1x + (d_0 + d_2) \end{aligned}$$

The polynomial products $y_i(x)d_i(x)$ are now carried out in the residue rings $F[x]/m_0(x)$ and $F[x]/m_1(x)$. In particular the desired products are

$$\begin{aligned} y_0(x) &= (h_1x + h_0)(d_1x + d_0) \pmod{x^2} \\ y_1(x) &= (h_1x + (h_0 + h_2))(d_1x + (d_0 + d_2)) \pmod{x^2 - 1} \end{aligned}$$

which can be implemented using the following operations:

$$\begin{aligned} m_1 &= h_0d_1 \\ m_2 &= h_1d_0 \\ m_3 &= h_0d_0 \\ m_4 &= (-(h_0 + h_2) + h_1)(d_0 + d_2) \\ m_5 &= (-(h_0 + h_2) + h_1)(d_1) \\ m_6 &= (h_0 + h_2)(d_1 + d_0 + d_2) \end{aligned}$$

$$\begin{aligned} y_0(x) &= (m_1 + m_2)x + m_3 \\ y_1(x) &= (m_4 + m_6)x + (m_5 + m_6) \end{aligned}$$

Finally, the residue polynomials must be combined in order to construct $y(x)$ mod $M(x)$ from the set $y_i(x)$ using the reconstruction formula

$$y(x) \bmod M(x) = \alpha_0(x)y_0(x) + \alpha_1(x)y_1(x)$$

The set of idempotents $\{\alpha_i(x)\}$ for the subrings $F[x]/m_0(x)$ and $F[x]/m_1(x)$ are given by

$$\begin{aligned} \alpha_0(x) &= -x^2 + 1 \\ \alpha_1(x) &= x^2 \end{aligned}$$

The polynomial $y(x)$ mod $M(x)$ is given by

$$y(x) = y_3x^3 + y_2x^2 + y_1x + y_0 = \tag{4.11}$$

$$\alpha_0(x)y_0(x) + \alpha_1(x)y_1(x) \pmod{M(x)}$$

Equating coefficients in expression (4.11) yields

$$\begin{aligned} y_3 &= -(m_1 + m_2) + (m_4 + m_6) \\ y_2 &= -m_3 + (m_5 + m_6) \\ y_1 &= (m_1 + m_2) \\ y_0 &= m_3 \end{aligned}$$

The original polynomial $y(x)$ can now be computed using (4.10),

$$y(x) = h(x)d(x) \qquad \text{mod } M(x) + h_2 d_2 M(x)$$

Let $m_7 = h_2 d_2$, then

$$y(x) = h(x)d(x) \qquad \text{mod } M(x) + m_7(x^4 - x^2)$$

and the coefficients of $y(x)$ are

$$\begin{aligned} y_4 &= m_7 \\ y_3 &= -(m_1 + m_2) + (m_4 + m_6) \\ y_2 &= -(m_3 + m_7) + (m_5 + m_6) \\ y_1 &= (m_1 + m_2) \\ y_0 &= m_3 \end{aligned}$$

This procedure can be represented by a matrix factorization; in particular

$$\begin{bmatrix} h_0 & & \\ h_1 & h_0 & \\ h_2 & h_1 & h_0 \\ & h_2 & h_1 \\ & & h_2 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = [B][D] \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \tag{4.12}$$

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 1 & -1 \\ -1 & -1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad [D] = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \end{bmatrix}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ -1 & 1 & -1 \\ -1 & 1 & -1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}$$

If the coefficients $\{h_i\}$ are known prior to compute time the diagonal constants of the matrix D can be pre-computed. This implies that the matrix product $\bar{y} = (C_3)\bar{d}$ can be computed using the following operations:

- Pre-compute:

$$\begin{aligned} t_1 &= d_0 + d_2 \\ t_2 &= d_1 - t_1 \end{aligned}$$

- Pre-addition Stage:

$$\begin{aligned} t_3 &= h_0 + h_2 \\ t_4 &= t_3 + h_1 \end{aligned}$$

- Multiplication Stage:

$$\begin{aligned} m_1 &= d_0 * h_1 \\ m_2 &= d_1 * h_0 \\ y_0 = m_3 &= d_0 * h_0 \\ m_4 &= t_2 * t_3 \\ m_5 &= t_2 * h_1 \\ m_6 &= t_1 * t_4 \\ y_4 = m_7 &= d_2 * h_2 \end{aligned}$$

- Post-addition Stage:

$$\begin{aligned} y_1 &= m_1 + m_2 \\ y_2 &= m_5 + m_6 - m_7 - m_3 \\ y_3 &= m_4 + m_6 - y_1 \end{aligned}$$

This suggests that seven multiplications and eight additions are required to implement (4.12); this is represented this by the cartesian pair $(7, 8)$. For

Example	B(m,a)	D(m,a)	A(m,a)	$max \sum(m,a)$
B.1.1	(5,10)	(5,0)	(2,6)	(12,16)=16
B.1.2	(0,9)	(7,0)	(0,4)	(7,13)=13
B.1.3	(0,6)	(6,0)	(0,3)	(6,9)=9
B.1.4	(5,10)	(5,0)	(2,5)	(12,15)=15
B.1.5	(0,9)	(6,0)	(0,5)	(6,14)=14

Table 4.3: 3 Point Linear by CRT Approach

this choice of residue polynomials, $max(7,8)$ is reduced from 9, in the case of direct implementation, to 8. This represents a reduction of about 13 % in the number of compute cycles needed to realize multiple three point convolution on model I.

It seems from the result of this example that the CRT approach may indeed be of value when designing convolution algorithms for RISC architectures. Maybe another choice of quotient rings $F[x]/m_i(x)$ would have led to an even further reduction in $max(m,a)$? This seems like a very reasonable question and merits further investigation.

In appendix B, several other examples have been worked out; the results are summarized in table 4.3. These indicate that in all of the examples presented, the use of the CRT leads to an increase in $max(m,a)$.

Suppose now that four point convolutions are required. This computation is given by $\bar{y} = (C_4)\bar{x}$ or

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} h_0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 \\ h_2 & h_1 & h_0 & 0 \\ h_3 & h_2 & h_1 & h_0 \\ 0 & h_3 & h_2 & h_1 \\ 0 & 0 & h_3 & h_2 \\ 0 & 0 & 0 & h_3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

which by direct implementation requires $(16,9)$ operations. The same procedure used for three point convolution was exploited to derive four point

Example	B(m,a)	D(m,a)	A(m,a)	$max \sum(m,a)$
B.2.1	(0,12)	(10,0)	(0,6)	(10,18)=18
B.2.2	(19,26)	(7,0)	(6,11)	(32,37)=37
B.2.3	(17,21)	(7,0)	(6,18)	(30,39)=39
B.2.4	(9,15)	(8,0)	(3,12)	(20,27)=27

Table 4.4: 4 Point Linear by CRT Approach

convolution algorithms. The resulting algorithms are given in appendix B. The arithmetic performance of these algorithms is summarized in table 4.4.

In assessing the results of these examples take notice of the following points:

1. There are many more possible residue polynomials to try in the case of four point convolution.

2. The rational coefficients introduced via the CRT can become ill behaved and can result in considerable roundoff errors when realizing large convolutions.

3. The process of designing algorithms and writing actual code based on the algorithms is very difficult. This makes using the technique very time consuming.

4. The CRT approach does not promote the use of looping. This results in large code. This may be a significant disadvantage when dealing with RISCs that make use of small instruction caches to increase throughput because the larger code may not fit. This will result in lost compute cycles when reloading the cache becomes necessary.

Others have tried to deal with the drawbacks of the CRT approach by developing a software package [22] that automates the design process. The *Scratchpad* package implements the CRT, gives arithmetic breakdowns, and even writes Fortran code that realizes the resulting pre and post addition matrices. Although *Scratchpad* can be a valuable time saving piece of software, when available, it only makes the problem of designing Winograd

algorithms easier to deal with. There is still the inherent problem that large convolution algorithms designed using the Winograd approach suffer serious implementation difficulties.

This work reveals that the CRT can be very useful for designing very particular types of small convolution algorithms, but it becomes extremely difficult to work with and implement when larger convolution algorithms are needed. This is true even with the aid of very sophisticated software tools like *Scratchpad* . What is needed in order to design convolution algorithms for modern architectures is another algebraic structure for tying together small convolutions in order to a realize large convolution. The tensor product representation developed in chapter three is a very natural and effective choice for this.

By the fundamental factorization of the linear convolution matrix derived in section 3.1, it was shown that C_n, $(n = n_1 n_2)$ can be decomposed as

$$C_n = R_{n_1,n_2}(B \otimes I_{2n_1-1})\Gamma(A \otimes I_{n_1}) \tag{4.13}$$

$$\Gamma = \sum_{j=0}^{\gamma-1} \oplus C_{n_1,j} = (I_\gamma \hat{\otimes} C_{n_1,j})$$

A core factorization of the form $C_{n_2} = B(D_\gamma)A$, where A and B are pre and post addition matrices respectively and D is a diagonal matrix of order γ, is needed in order to use the fundamental factorization (4.13) to design convolution algorithms. For the present discussion let $n_2 = 2$ and define the core C_2 as

$$\begin{bmatrix} h_0 & 0 \\ h_1 & h_0 \\ 0 & h_1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_1 & 0 & 0 \\ 0 & k_2 & 0 \\ 0 & 0 & k_3 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \end{bmatrix} \tag{4.14}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix}$$

Note that this core implies that two point linear convolution can be realized in 3 multiplications and 3 additions using the instructions:

Multiplier		y_0	m_2	y_2		
Adder	a_1				a_2	y_1

Table 4.5: Forbidden list = {1,2,4,5}

- Pre-compute:

$$t_1 = h_0 + h_1$$

- Compute-time:

$$\begin{array}{llll} y_0 = & m_1 = d_0 * h_0 & & a_1 = d_0 + d_1 \\ & m_2 = a_1 * t_1 & & a_2 = m_2 - m_1 \\ y_2 = & m_3 = h_1 * d_1 & y_1 = & a_3 = a_2 - m_3 \end{array}$$

Factorization (4.14) implies that n independent 2 point convolutions of the form

$$(I_n \hat{\otimes} C_{2,j})$$

can be scheduled on a model I RISC computer in approximately $3n$ instructions by using the reservation table 4.5.

As is now shown, the core (4.14) and the fundamental factorization (4.13) can be used to derive a family of convolution algorithms with the property that the number of multiplications is the same as the number of additions, and that $max(m, a)$ for these is significantly reduced as compared to a direct implementation.

Before presenting the general decomposition, it will be useful to examine some examples. Let $n = 4$, $n_1 = n_2 = 2$, and make direct use of (4.13) to yield the decomposition

$$C_4 = R_{2,2}(B \otimes I_3)(I_3 \hat{\otimes} C_{2,j})(A \otimes I_2) \tag{4.15}$$

where

$$R_{2,2} = \begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & 1 & \\ & & & & & & & 1 \\ & & & & & & & & 1 \end{bmatrix}$$

$$(B \otimes I_3) = \begin{bmatrix} I_3 & 0 & 0 \\ -I_3 & I_3 & -I_3 \\ 0 & 0 & I_3 \end{bmatrix} \qquad (A \otimes I_2) = \begin{bmatrix} I_2 & 0 \\ I_2 & I_2 \\ 0 & I_2 \end{bmatrix}$$

and

$$(I_3 \hat{\otimes} C_{2,j}) = \begin{bmatrix} C_1 & 0 & 0 \\ 0 & C_2 & 0 \\ 0 & 0 & C_3 \end{bmatrix}$$

$$C_1 = \begin{bmatrix} G_1 & \\ G_2 & G_1 \\ & G_2 \end{bmatrix} \quad C_2 = \begin{bmatrix} G_3 & \\ G_4 & G_3 \\ & G_4 \end{bmatrix} \quad C_3 = \begin{bmatrix} G_5 & \\ G_6 & G_5 \\ & G_6 \end{bmatrix}$$

$$\begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \\ G_5 \\ G_6 \end{bmatrix} = \begin{bmatrix} I_2 & 0 \\ I_2 & I_2 \\ 0 & I_2 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

In general the arithmetic complexity of the stage R_{n_1,n_2} is

$$(0, 2(n_1 - 1)(n_2 - 1))$$

In this particular case the combined action of the stages

$$R_{n_1,2}(B \otimes I_{2n_1-1})$$

where B is as defined in (4.14), makes it possible to save half of the additions of $R_{n_1,2}$. Thus, in this case, the complexity of $R_{n_1,2}$ is $(0, n_1 - 1)$ and the matrix product $\bar{y} = (C_4)\bar{x}$ can be realized by using the following operations:

- Pre-compute:

$$\begin{aligned} G_1 &= h_0 \\ G_2 &= h_1 \\ G_3 &= h_0 + h_2 \\ G_4 &= h_1 + h_3 \\ G_5 &= h_2 \\ G_6 &= h_3 \end{aligned}$$

- Compute-time:

$$\begin{aligned} T_1 &= G_1 * d_0 \\ T_2 &= (G_2 * d_0) + (G_1 * d_0) \\ T_3 &= (G_2 * d_1) \\ T_4 &= G_2 * (d_0 + d_2) \\ T_5 &= G_4 * (d_0 + d_2) + G_3 * (d_1 + d_2) \end{aligned}$$

$$\begin{aligned} T_6 &= G_4 * (d_1 + d_2) \\ T_7 &= G_5 * d_2 \\ T_8 &= (G_6 * d_2) + (G_5 * d_3) \\ T_9 &= G_6 * d_3 \end{aligned}$$

$$\begin{aligned} y_0 &= T_1 \\ y_1 &= T_2 \\ y_2 &= (T_3 - T_7) + (T_4 - T_1) \\ y_3 &= T_5 - T_2 - T_8 \\ y_4 &= -(T_3 - T_7) + (T_6 - T_9) \\ y_5 &= T_8 \\ y_6 &= T_9 \end{aligned}$$

These imply that the $(12, 12)$ operations are required to realize (4.15) and that $max(12, 12) = 12$. Recall that $max(16, 9) = 16$ for a direct implementation of $\bar{y} = (C_4)\bar{x}$. Algorithm (4.15) thus offers a reduction in $max(m, a)$ of 25 %. Also compare the performance of the tensor product algorithm with the algorithms derived based on the CRT approach as summarized in table 4.3. Again the tensor product algorithm offers considerable improvements over these. In addition, note the tensor product algorithm was derived by a straightforward substitution into the fundamental factorization (4.13). This makes the procedure very easy to understand and to use.

In order to exploit the fact that $max(m,a)$ was reduced from 16 to 12 it is assumed that many independent four point convolutions are to be performed. This operation is expressed by

$$\Gamma = \sum_{j=0}^{n-1} \oplus C_{4,j} = (I_n \hat{\otimes} C_{4,j}) \tag{4.16}$$

In order to efficiently realize (4.15) it is necessary to devise a scheduling scheme that exploits the parallelism of the underlying architectural model. The operations for achieving this are given below; a scheduling scheme for these is given by the reservation table 4.6.

$$\begin{array}{llll}
t_5 = d_0 + d_2 & & t_{12} = G_5 * d_3 & \\
t_6 = d_1 + d_3 & & y_6 = t_9 = G_6 * d_3 & \\
y_0 = t_1 = G_1 + d_0 & & y_1 = t_2 = t_2 + t_{10} & \\
t_2 = G_0 * d_0 & & t_5 = t_5 + t_{11} & \\
t_{10} = G_1 * d_1 & & y_5 = t_8 = t_8 + t_{12} & \\
t_3 = G_2 * d_1 & & t_1 = t_4 - t_1 & \\
t_4 = G_3 * t_5 & & t_2 = t_3 - t_7 & \\
t_5 = G_4 * t_5 & & y_2 = t_1 = t_1 + t_2 & \\
t_{11} = G_3 * t_6 & & t_1 = t_5 - t_2 & \\
t_6 = G_4 * t_6 & & y_3 = t_1 = t_1 - t_8 & \\
t_7 = G_5 * d_2 & & t_1 = t_6 - t_9 & \\
t_8 = G_6 * d_2 & & y_4 = t_1 = t_1 - t_2 &
\end{array}$$

Next, consider the implementation of six point linear convolution. Let $n_1 = 3$,$n_2 = 2$, again use (4.14) as the core factorization. Direct use of (4.13) yields the decomposition

$$C_6 = R_{3,2}(B \otimes I_5)(I_3 \hat{\otimes} C_{3,j})(A \otimes I_3) \tag{4.17}$$

By using a procedure analogous to that described above for four point linear convolution, it can be shown that the number of operations required for this algorithm is

$$C_6(m,a) = (0,2) + 5(0,2) + 3(9,4) + 3(0,1) = max(27,27) = 27$$

Note that for the direct implementation of C_6, $max(36,25) = 36$. In the case of algorithm (4.17) $max(27,27) = 27$ which again represents a savings

*	t_5	t_6										
+			y_0	t_3	t_{10}	t_3	t_4	t_5	t_{11}	t_6	t_7	t_8

		y_1	t_5	y_5	t_1	t_2	y_2	t_1	y_3	t_1	y_5
t_{12}	y_6										

Table 4.6: Forbidden list = { Z/24 - {0,12 } }

of 25%. An appropriate scheduling strategy for this algorithm can readily be found.

As a final example, consider the implementation of eight point linear convolution. In this case $n_1 = 4$, $n_2 = 2$ and direct use of (4.13) yields

$$C_8 = R_{4,2}(B \otimes I_7)(I_3 \hat{\otimes} C_{4,j})(A \otimes I_4) \tag{4.18}$$

The arithmetic complexity of this algorithm is readily shown to be

$$C_8(m,a) = (0,3) + 7(0,2) + 3(16,9) + 4(0,1) = max(48,48) = 48$$

The direct implementation of C_8 requires $max(64,49) = 64$ compute cycles. The use of (4.18) reduces this by 25% to $max(48,48) = 48$.

The generalization of these examples leads to an arithmetically balanced variant of the fundamental factorization (4.13).

Balanced Convolution Variant

Let $n_2 = 2$, and $C_2 = B(D_\gamma)A$ be given by

$$\begin{bmatrix} h_0 & 0 \\ h_1 & h_0 \\ 0 & h_1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_1 & 0 & 0 \\ 0 & k_2 & 0 \\ 0 & 0 & k_3 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \end{bmatrix} \tag{4.19}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix}$$

SIZE	DIRECT	TEN. PR.
2	4	3
4	16	12
6	36	27
8	64	48
10	100	75
12	144	108
14	196	147
16	256	192

Table 4.7: Performance of Small Balanced Convolution

then C_n can be decomposed as

$$C_n = R_{n_1,2}(B \otimes I_{2n_1-1})(I_3 \hat{\otimes} C_{n_1,j})(A \otimes I_{n_1}) \tag{4.20}$$

The arithmetic complexity of the balanced variant algorithm is given by

$$C_n(m,a) = (0, n_1 - 1) + 2n_1 - 1(0,2) + 3(n_1^2, (n_1-1)^2) + n_1(0,1)$$

or

$$C_n(m,a) = max(3n_1^2, 3n_1^2) = 3n_1^2 \tag{4.21}$$

A direct implementation of C_n has a complexity of $4n_1^2$. The family of algorithms of factorization (4.20) has a complexity of $3n_1^2$. This makes the balanced family of algorithms very well suited for implementation on a model I RISC when an application calls for many independent small convolutions to be performed. Table 4.7 compares $max(m,a)$ for these algorithms with $max(m,a)$ based on direct implementation of C_n.

Matrices of the form

$$(I_\gamma \hat{\otimes} C_{n_1,j})$$

are important not only because it is often necessary to realize many independent convolutions, but also because they are an important stage of

computation for large convolution algorithms. This is made evident from the fundamental factorization (4.13).

In assessing the performance of a large convolution algorithm the performance measure $max(m, a)$ is no longer valid because it may not be correct to assume that many large convolutions are preformed. However, if a large convolution is decomposed into several stages of the form

$$(I_\gamma \hat{\otimes} K)$$

were K is any matrix of complexity $K(m, a)$, it is valid to measure the performance of each of these individual stages by $\gamma * maxK(m, a)$. The performance of a large algorithm is then obtained by added up the number of compute cycles needed for the individual stages. Expression (4.13) facilitates the use of this procedure for convolution.

As an example of this, suppose it is necessary to realize a single 128 point linear convolution. If this where directly implemented on a model I RISC it would require $(128)^2 = 16,384$ compute cycles. Since this represents a considerable amount of computation it would be beneficial to use a fast algorithm if possible. Consider the use of the convolution theorem 2.1.19 in this case.

For linear convolution, the two sequences must be zero padded to length 255 before 2.1.19 can be used. The convolution theorem then asserts that the convolutions can be carried out using two 255 point Fourier transforms and 255 complex multiplications. The performance of this approach is enhanced if the sequences are zero padded to length 256 so that 256 point Fourier transforms can be used. This is the approach taken here.

The number of operations required to realize C_{128} by the convolution theorem is given by

$$C_{128}(m, a) = 2F_{256}(m, a) + 256(4, 2)$$

The 256 point Fourier transform that will be used is given in appendix A as

$$F_{256} = (F_{64} \otimes I_4)T_{256,4}(I_{64} \otimes F_4)P_{128,4}$$

The performance of this algorithm must be reevaluated for a model I implementation. In appendix C this is shown to be

$$F_{256} = 4(1028) + 189(4) + 64(16) = 5892$$

Using this, the number of compute cycles needed to realize the convolution theorem is given by

$$C_{128} = 2(5892) + 256(4) = 12,808$$

In this case the use of the convolution theorem leads to a considerable reduction in the number of compute cycles needed.

Consider now the use of the fundamental factorization. Let $n_1 = 32$, $n_2 = 4$, and let the core C_4 be given by the balanced varient (4.20). Then, (4.13) leads to the decomposition

$$C_{128} = R_{32,4}(B \otimes I_{63})(I_{12}\hat{\otimes}C_{32,j})(A \otimes I_{32}) \tag{4.22}$$

The arithmetic complexity of B and A is $(0, 10)$ and $(0, 2)$, respectively. In order to realize the convolutions $C_{32,j}$, again use the small balanced variant of expression (4.20). On the basis of these factors, the number of compute cycles needed to realize C_{128} is

$$C_{128} = (186) + 63(10) + 12(768) + 32(2) = 10,096$$

Thus, the straight forward application of the fundamental factorization leads to a significant improvement over the results obtained by the convolution theorem.

Although this is a significant improvement, it certainly is not the best that can be achieved using this approach. Consider what happens to the operations count as some of the other cores derived in appendix B are used with (4.22).

- Core of example $B.2.1$

  In example B.2.1 the complexity of a Winograd core for $C_4 = BDA$ was derived as

  $$C_4(m, a) = (0, 12) + (10, 0) + (0, 6)$$

  Direct use of this with expression (4.22) yields an operations count of

  $$C_{128} = (186) + 63(12) + 10(768) + 32(6) = 8,814$$

- Core of example $B.2.4$

  In example B.2.4 the complexity of a Winograd core for $C_4 = BDA$ was derived as

$$C_4(m, a) = (9, 15) + (8, 0) + (3, 12)$$

  Direct use of this with expression (4.22) yields an operations count of

$$C_{128} = (186) + 63(15) + 8(768) + 32(12) = 7,659$$

- Core of example $B.2.3$

  In example B.2.3 the complexity of a Winograd core for $C_4 = BDA$ was derived as

$$C_4(m, a) = (17, 21) + (7, 0) + (6, 18)$$

  Direct use of this with expression (4.22) yields an operations count of

$$C_{128} = (186) + 63(21) + 7(768) + 32(18) = 7,461$$

Although these are dramatic reductions in the number of compute cycles needed, even greater improvements are possible. Consider what happens to the operations counts if instead of a small balanced convolution for C_{32}, an alternate decomposition is used. In particular let $n_1 = 8$, $n_2 = 4$, use the core of example B.2.4, and realize the resulting $C_{8,j}$ by the small convolution approach of expression (4.20). Then C_{32} can be decomposed as

$$C_{32} = R_{8,4}(B \otimes I_{15})(I_8 \hat{\otimes} C_{8,j})(A \otimes I_8) \tag{4.23}$$

The arithmetic complexity of this algorithm is given by

$$C_{32} = (42) + 15(15) + 8(48) + 8(12) = 747$$

If this is used in expression (4.22) the resulting algorithm would have a complexity of

$$C_{128} = (186) + 63(21) + 7(747) + 32(18) = 6,747$$

SIZE	DIRECT	CNV. TH.	TEN. PR.
32	1985	2312	747
64	8065	5640	2283
128	32513	12808	6747
256	130561	28680	20178
512	523265	65544	55650
1024	2095105	143368	152739

Table 4.8: Large Convolution on Model I

Compare this with the 16,384 needed by direct implementation and the 12,808 needed by the convolution theorem.

Applying this approach to convolution design leads to the results summarized in table 4.8. The algorithms that lead to these results are presented in appendix C. Note that a similar line of reasoning will also lead to highly efficient algorithms for model II RISC architectures.

It is important to understand that the tensor product approach does not favor sequences of length 2^α in any way. These were chosen above only because it is convenient to compare convolutions of this size to algorithms designed using the convolution theorem.

In order to illustrate of this point, suppose that an 80 point convolution is required. If a direct implementation is used, 6400 compute cycles are needed. If the convolution theorem is used, it would be necessary to design an 160 point Fourier transform algorithm or to zero pad the sequences to length 256.

By the tensor product approach, let $n_1 = 4$, $n_2 = 20$, and use the core of factorization B.2.3 to get

$$C_{80} = R_{20,4}(B \otimes I_7)(I_7 \hat{\otimes} C_{20,j})(A \otimes I_4) \tag{4.24}$$

The resulting $C_{20,j}$'s in this expression can be realized by letting $n_1 = 4$, $n_2 = 5$ and using the core of example B.2.4. This yields the decomposition

$$C_{20} = R_{5,4}(B \otimes I_7)(I_8 \hat{\otimes} C_{5,j})(A \otimes I_4) \tag{4.25}$$

If the resulting $C_{5,j}$'s in expression (4.25) are realized by direct implementation, this implies that the complexity of algorithm (4.25) is given by

$$C_{20}(m,a) = (24) + 7(15) + 8(25) + 4(12) = 377$$

The substitution of (4.25) into (4.24) results in an algorithm for C_{80} whose complexity is

$$C_{80}(m,a) = (114) + 7(21) + 7(377) + 4(18) = 2,972$$

4.4 Discrete Fourier Transform

4.4.1 Bit Reversal Addressing

In section 2.1.5 an addressing scheme for realizing the shuffling action of stride permutations was presented. By using this scheme it is possible to realize highly efficient implementations of tensor matrices of the form

$$\bar{y} = P_1(I \otimes F)P_2\bar{x} \tag{4.26}$$

where P_1 and P_2 are stride permutations.

Now, consider the permutations which arise in a mixed radix additive FFT algorithm. In section 2.3.2 this algorithm for $N = \prod_{i=1}^{\alpha} n_i$ was shown to be given by the matrix product.

$$F_N = \left(\prod_{i=1}^{\alpha}\left(I_{(\prod_{k<i} n_k)} \otimes F_{n_i} \otimes I_{(\prod_{k>i} n_k)}\right)\left(I_{(\prod_{k<i} n_k)} \otimes T_{(\prod_{k\geq i} n_k),n_i}\right)\right) Q$$

$$Q = \prod_{i=\alpha}^{1}\left(I_{(\prod_{k<i} n_k)} \otimes P_{(\prod_{k\geq i} n_k),n_i}\right)$$

In this expression there are two types of compute stages; the Fourier transform stages

$$\left(I_{(\prod_{k<i} n_k)} \otimes F_{n_i} \otimes I_{(\prod_{k>i} n_k)}\right) \tag{4.27}$$

and the twiddle multiplication stages

$$\left(I_{(\prod_{k<i} n_k)} \otimes T_{(\prod_{k\geq i} n_k),n_i}\right) \tag{4.28}$$

Expression (4.28) is a special case of expression (4.26) and therefore amenable to the techniques presented in section 4.2.2. Expression (4.27) can be put in the form of expression (4.26) by making use of the commutative property (2.16) of the tensor product. Direct use of this yields

$$\left(I_{(\prod_{k<i} n_k)} \otimes F_{n_i} \otimes I_{(\prod_{k>i} n_k)}\right) = P_{N,(\prod_{k\leq i} n_k)}\left(I_{N/n_i} \otimes F_{n_i}\right)P_{N,(\prod_{k>i} n_k)} \tag{4.29}$$

which can now be implemented efficiently using the methods already discussed. The techniques presented above take care of the addressing for every stage except the permutation matrix Q which is now considered.

When $N = 2^\alpha$, and F_N fully decomposed into radix two Fourier transforms, the matrix Q represents the well known *bit reversal* addressing scheme. In order to deal with bit reversal addressing some RISC machines (most notably the DSP chips [40,41,42]) have a special addressing mode for FFT. Indeed, if all one requires is efficient implementations of radix 2 Cooley-Tukey algorithms, the use of this addressing mode can save the time required for the bit reversal permutation. Although Papamichalis *et al* [12] show that this addressing mode can also be used for radix $r = 2^n$ algorithms, it is of no help when implementing a general mixed radix FFT algorithm or many of the other algorithms designed in this dissertation. In this treatment, the Q permutation will be approached in a much more general way. In doing so it becomes possible to use techniques already developed in order to realize it efficiently.

Although the addressing specified by Q is not that of the stride permutation, its addressing can be expressed as a tensor product of stride permutations. This provides the key to extending the results already established. Consider the complexity of Q for $N = (n_1n_2)(n_3n_4)$. Direct use of the formula yields the factorization

$$F_N = (F_{n_1n_2} \otimes I_{n3n_4})T_{N,n_1n_2}(I_{n_1n_2} \otimes F_{n_3n_4})Q$$
$$Q = P_{N,n_1n_2}$$

In this case, Q is a stride permutation matrix. As such it can be implemented using the techniques already outlined in examples 4.2.4 and 4.2.6.

Note that the addressing specified by Q can also be expressed in terms of an index map. Let $\bar{x_N}$ be an input vector of complex data and let $x(j)$

represent the j^{th} element of this array. Then the addressing of Q is specified by the map

$$j = j_1 + (n_1 n_2) j_2 \tag{4.30}$$

where

$$0 \leq j_1 < (n_1 n_2)$$
$$0 \leq j_2 < (n_3 n_4)$$

Let $r1$ represent a typical data pointer register and let the vector $\bar{x}$ be at location $DATA$. Then it is possible to realize $(I_{n_1 n_2} \otimes F_{n_3 n_4})Q\bar{x}$ using the technique illustrated in example 4.2.4. Pseudo-code for implementing this is given by example 4.4.1.

- **Example 4.4.1** Pseudo-code for $(I_{n_1 n_2} \otimes F_{n_3 n_4})Q\bar{x}$

```
r1 = DATA
for i = 1 to (n1n2)
{
    Fn3n4 :
    On each reference to r1,
    r1 = r1+(n1n2)
    ⋮
    r1 = DATA +i
}
```

□

As an alternative, suppose that F_N were fully decomposed into its factors. In this case $N = n_1 n_2 n_3 n_4$ and the use of (2.86) now yields the factorization

$$F_N = (F_{n_1} \otimes I_{n_2 n_3 n_4}) T_{N,n_1} (I_{n_1} \otimes F_{n_2} \otimes I_{n_3 n_4})(I_{n_1} \otimes T_{n_2 n_3 n_4, n_2})$$
$$(I_{n_1 n_2} \otimes F_{n_3} \otimes I_{n_4})(I_{n_1 n_2} \otimes T_{n_3 n_4, n_3})(I_{n_1 n_2 n_3} \otimes F_{n_4})Q$$
$$Q = (I_{n_1 n_2} \otimes P_{n_3 n_4, n_3})(I_{n_1} \otimes P_{n_2 n_3 n_4, n_2}) P_{N,n1}$$

The addressing of Q is no longer that of simple stride addressing. Because it is specified in terms of tensor products of stride permutations, however, it is possible to extend the results of example 4.4.1 to realize Q. Since the addressing of Q is now specified by the index map

$$j = j_1 + n_1 j_2 + n_1 n_2 j_3 + n_1 n_2 n_3 j_4 \tag{4.31}$$

where

$$0 \le j_1 < n_1 \qquad 0 \le j_2 < n_2$$
$$0 \le j_3 < n_3 \qquad 0 \le j_4 < n_4$$

the extension of example 4.4.1 results in the pseudo-code of example 4.4.2.

- **Example 4.4.2** Pseudo-code for $(I_{n_1 n_2 n_3} \otimes F_{n_4})Q\bar{x}$

```
r1 = DATA
for k = 1 to (n1)
    for j = 1 to (n2)
        for i = 1 to (n3)
        {
            Fn4 :
            On each reference to r1,
            r1 = r1+(n1n2n3)
            ⋮
            r1 = DATA+i*(n1n2)
        }
    r1 = DATA+j*(n1)
r1 = DATA +k
```

□

In comparing the pseudo-code of these examples, two facts should become apparent:

1. The more an F_N is decomposed the more complicated the addressing of the matrix Q becomes and the harder it is to realize it efficiently.

	Conventional FFT	Factors	Optimized FFT
F-32	.434 ms	(4)(8)	.143 ms
F-64	.940 ms	(8)(8)	.331 ms
F-128	2.01 ms	(2)(8)(8)	.886 ms
F-256	4.28 ms	(4)(8)(8)	1.93 ms
F-512	9.09 ms	(8)(8)(8)	4.22 ms
F-1024	19.2 ms	(8)(8)(8)(4)	9.91 ms

Table 4.9: Timings for various FT algorithms

2. The less an F_N is decomposed the more difficult it becomes to realize an efficient implementation of the stages $\left(I \otimes F_{n_j}\right)$

Thus, in order to make Q as simple as possible it would be beneficial to decompose N as little as possible. On the other hand, if N is not decomposed enough the resulting Fourier transform stages become computationally complex.

In the case of a Cooley-Tukey algorithm, a given F_{2^k} is fully decomposed into radix two butterflies and the permutation Q becomes the complicated bit reversal permutation. This represents a worst case situation for the efficient implementation of Q and a best case situation for stages $(I \otimes F_2)$. Better implementations of this algorithm can be achieved by choosing the factors of N more carefully.

This approach, combined with some observations regarding symmetries of twiddle constants, have led to the results reported in table 4.9. The actual tensor product decompositions for these are given in appendix A. These algorithms were developed in [39] and implemented on an AT&T DSP 32. This chip is a model II RISC machine. The resulting algorithms were compared with the FFT algorithms included as part of the DSP32 applications library.

4.4.2 Re-Distributing the DFT

The general mixed radix additive DFT decomposition was derived in section 2.3.2 as formula (2.86). This was given by

$$F_N = \left(\prod_{i=1}^{\alpha} \left(I_{(\prod_{k<i} n_k)} \otimes F_{n_i} \otimes I_{(\prod_{k>i} n_k)}\right)\left(I_{(\prod_{k<i} n_k)} \otimes T_{(\prod_{k\geq i} n_k),n_i}\right)\right) Q \tag{4.32}$$

$$Q = \prod_{i=\alpha}^{1} \left(I_{(\prod_{k<i} n_k)} \otimes P_{(\prod_{k\geq i} n_k),n_i}\right)$$

Disregarding the Q permutation stage, the direct use of this expression results in matrix factors of the form

$$F_n = [F_1][T_1][F_2][T_2][F_3][T_3]\cdots[F_{\alpha-1}][T_{\alpha-1}][F_\alpha]$$

where $[F_i]$ are stages involving small Fourier transforms and $[T_i]$ are diagonal complex matrices (twiddle multiplication stages). An examination of the computational requirements of these stages reveals that, in many cases, the Fourier transforms stages are dominated by additions while the twiddle stages are dominated by multiplications. Thus, if this algorithm were directly implemented on a modern RISC computer, the multiplier would be idle during an $[F]$ stage while the adder would be idle during a $[T]$ stage.

By using the underlying algebraic structure of the tensor product representation of the DFT, it is possible to alter (4.32) and thereby avoid this. The resulting algorithm has computational stages of the form [FT]. This algorithm will make better use of the parallelism of the RISC models. For the sake of notational clarity the algorithm will be derived using the general radix r factorization expression (2.85) and then extended to the general mixed radix case.

In section 2.3.2 the tensor product radix r DFT decomposition was given as

$$F_n = \left(\prod_{i=1}^{\alpha} (I_{r^{i-1}} \otimes F_r \otimes I_{r^{\alpha-i}})\left(I_{r^{i-1}} \otimes T_{r^{\alpha-i+1},r}\right)\right) Q$$

Use the distributive property of the tensor product (2.12) to yield

$$F_n = \left(\prod_{i=1}^{\alpha} (I_{r^{i-1}} \otimes [F_r \otimes I_{r^{\alpha-i}}]\, T_{r^{\alpha-i+1},r})\right) Q \tag{4.33}$$

By the commutative property (2.16) note that

$$(F_r \otimes I_{r^{\alpha-i}}) = P_{r^{\alpha-i+1},r}(I_{r^{\alpha-i}} \otimes F_r)P_{r^{\alpha-i+1},r^{\alpha-i}} \tag{4.34}$$

and that the substitution of (4.34) into (4.33) yields

$$F_n = \left(\prod_{i=1}^{\alpha}(I_{r^{i-1}} \otimes \left[P_{r^{\alpha-i+1},r}(I_{r^{\alpha-i}} \otimes F_r)P_{r^{\alpha-i+1},r^{\alpha-i}}\right] T_{r^{\alpha-i+1},r}\right) Q \tag{4.35}$$

Next,

$$(P_{r^{\alpha-i+1},r^{\alpha-i}})(T_{r^{\alpha-i+1},r}) = (P_{r^{\alpha-i+1},r^{\alpha-i}})(T_{r^{\alpha-i+1},r})(I_{r^{\alpha-i+1}})$$

$$(I_{r^{\alpha-i+1}}) = (P_{r^{\alpha-i+1},r^{\alpha-i}})^{-1}(P_{r^{\alpha-i+1},r^{\alpha-i}})$$

and

$$(P_{r^{\alpha-i+1},r^{\alpha-i}})(T_{r^{\alpha-i+1},r}) = \left[(P_{r^{\alpha-i+1},r^{\alpha-i}})(T_{r^{\alpha-i+1},r})(P_{r^{\alpha-i+1},r^{\alpha-i}})^{-1}\right](P_{r^{\alpha-i+1},r^{\alpha-i}})$$

Using property (2.6) of section 2.1.2 and appealing to the definition of $(T_{r^{\alpha-i+1},r})$, one can show that

$$\left[(P_{r^{\alpha-i+1},r^{\alpha-i}})(T_{r^{\alpha-i+1},r})(P_{r^{\alpha-i+1},r^{\alpha-i}})^{-1}\right] = (T_{r^{\alpha-i+1},r^{\alpha-i}}) \tag{4.36}$$

or

$$(P_{r^{\alpha-i+1},r^{\alpha-i}})(T_{r^{\alpha-i+1},r}) = (T_{r^{\alpha-i+1},r^{\alpha-i}})(P_{r^{\alpha-i+1},r^{\alpha-i}}) \tag{4.37}$$

The substitution of (4.37) into (4.35) yields

$$F_n = \left(\prod_{i=1}^{\alpha}(I_{r^{i-1}} \otimes \left[P_{r^{\alpha-i+1},r}(I_{r^{\alpha-i}} \otimes F_r)(T_{r^{\alpha-i+1},r^{\alpha-i}})(P_{r^{\alpha-i+1},r^{\alpha-i}})\right]\right) Q \tag{4.38}$$

By appealing to its matrix definition

$$(T_{r^{\alpha-i+1},r^{\alpha-i}}) = \sum_{j=0}^{r^{\alpha-i}-1} \oplus D^j_{r^{\alpha-i+1},r}$$

Introduce the pseudo tensor operator $\hat{\otimes}$ introduced in chapter three for linear convolution. This yields

$$\sum_{j=0}^{r^{\alpha-i}-1} \oplus D^j_{r^{\alpha-i+1},r} = (I_{r^{\alpha-i}} \hat{\otimes} D^j_{r^{\alpha-i+1},r})$$

Using this,

$$(I_{r^{\alpha-i}} \otimes F_r)(T_{r^{\alpha-i+1},r^{\alpha-i}}) =$$

$$(I_{r^{\alpha-i}} \otimes F_r)(I_{r^{\alpha-i}} \hat{\otimes} D^j_{r^{\alpha-i+1},r}) =$$

$$(I_{r^{\alpha-i}} \hat{\otimes} F_r D^j_{r^{\alpha-i+1},r}) \tag{4.39}$$

Finally, the substitution of (4.39) into (4.37) yields the algorithm

$$F_n = \left(\prod_{i=1}^{\alpha} \left(I_{r^{i-1}} \otimes \left[P_{r^{\alpha-i+1},r} (I_{r^{\alpha-i}} \hat{\otimes} F_r D^j_{r^{\alpha-i+1},r}) (P_{r^{\alpha-i+1},r^{\alpha-i}}) \right] \right) \right) Q \tag{4.40}$$

This formulation readily extends to a generalized mixed radix FFT. This is given by

$$F_N = \left(\prod_{i=1}^{\alpha} I_{(\prod_{k<i} n_k)} \otimes \Gamma_i \right) Q \tag{4.41}$$

$$\Gamma_i = \left[P_{(\prod_{k \geq i} n_k), n_i} (I_{(\prod_{k>i} n_k)} \hat{\otimes} F_{n_i} D^j_{(\prod_{k \geq i} n_k), n_i} (P_{(\prod_{k \geq i} n_k), (\prod_{k>i} n_k)}) \right]$$

For algorithm (4.41) all computational stages are combinations of small Fourier transforms and complex multiplications. This does not change the number of additions and multiplications required, but instead arranges them to better exploit the parallelism of the adder and multiplier.

As an example of the benefit obtained by using this approach consider a standard 1K Cooley-Tukey algorithm. This can be obtained by letting $r = 2$ and $\alpha = 10$. Direct use of (4.32) yields the algorithm

$$\begin{aligned} F_{1024} = {} & (F_2 \otimes I_{512}) T_{1024,2} (I_2 \otimes F_2 \otimes I_{256}) (I_2 \otimes T_{512,2}) \\ & (I_4 \otimes F_2 \otimes I_{128}) (I_4 \otimes T_{256,2}) (I_8 \otimes F_2 \otimes I_{64}) (I_8 \otimes T_{128,2}) \\ & (I_{16} \otimes F_2 \otimes I_{32}) (I_{16} \otimes T_{64,2}) (I_{32} \otimes F_2 \otimes I_{16}) (I_{32} \otimes T_{32,2}) \\ & (I_{64} \otimes F_2 \otimes I_8) (I_{64} \otimes T_{16,2}) (I_{128} \otimes F_2 \otimes I_4) (I_{128} \otimes T_{8,2}) \\ & (I_{256} \otimes F_2 \otimes I_2) (I_{256} \otimes T_{4,2}) (I_{512} \otimes F_2) Q \end{aligned} \tag{4.42}$$

This algorithm requires 5120 F_2's and 4097 complex multiplications. The number of compute cycles needed to implement this algorithm using the scheduling scheme developed in section two of this chapter for a model I RISC is

$$F_{1K} \leftrightarrow 5120 * max(0,4) + 4097 * max(4,2) = 36,868$$

Compare this with the number of compute cycles needed to implement a 1K point transform using algorithm (4.40). Direct use of this yields

$$\begin{aligned} F_{1024} = &\left[P_{1024,2}(I_{512}\hat{\otimes}F_2D^j_{1024,2})P_{1024,512}\right] \\ &\left[I_2 \otimes P_{512,2}(I_{256}\hat{\otimes}F_2D^j_{512,2})P_{512,256}\right] \\ &\left[I_4 \otimes P_{256,2}(I_{128}\hat{\otimes}F_2D^j_{256,2})P_{256,128}\right]\left[I_8 \otimes P_{128,2}(I_{64}\hat{\otimes}F_2D^j_{128,2})P_{128,64}\right] \\ &\left[I_{16} \otimes P_{64,2}(I_{32}\hat{\otimes}F_2D^j_{64,2})P_{64,32}\right]\left[I_{32} \otimes P_{32,2}(I_{16}\hat{\otimes}F_2D^j_{32,2})P_{32,16}\right] \\ &\left[I_{64} \otimes P_{16,2}(I_8\hat{\otimes}F_2D^j_{16,2})P_{16,8}\right]\left[I_{128} \otimes P_{8,2}(I_4\hat{\otimes}F_2D^j_{8,2})P_{8,4}\right] \\ &\left[I_{256} \otimes P_{4,2}(I_2\hat{\otimes}F_2D^j_{4,2})P_{4,2}\right]\left[(I_{512} \otimes F_2)\right]Q \end{aligned} \tag{4.43}$$

This algorithm requires 1023 (F_2)'s and 4097 $(F_2D^j_2)'s$. Thus, the number of compute cycles needed for this algorithm when implemented on model I is approximately

$$F_{1K} \leftrightarrow 1023 * max(0,4) + 4097 * max(4,6) = 28,674$$

Although the total number of operations for the algorithms are the same, algorithm (4.43) reduces the compute time of a 1K FFT by approximately 22% on a model I RISC. The scheduling needed for this is illustrated in example 4.4.3 where the scheduling for the stage $I_n\hat{\otimes}F_2D^j_2$ is given.

- **Example 4.4.3** Scheduling $I_n\hat{\otimes}F_2D^j_2$

  The term $F_2D^j_2$ is given by the factors

$$(F_2D^j_2)\bar{x} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & g_r & -g_i \\ 0 & 0 & g_i & g_r \end{pmatrix}\begin{bmatrix} x_{1r} \\ x_{1i} \\ x_{2r} \\ x_{2i} \end{bmatrix}$$

  There are many reservation tables which can be used in order to realize $F_2D^j_2$. An optimal one is given by table 4.10. Let the marks of table 4.10 correspond to the following operations:

Multiplier	x_1	x_2	x_3	x_4						
Adder					x_5	x_6	x_7	x_8	x_9	x_{10}

Table 4.10: Forbidden list = {1,2,3,4,5}

$$
\begin{aligned}
x_1 &\leftrightarrow t_1 = x2_r * g_r \\
x_2 &\leftrightarrow t_3 = x2_r * g_i \\
x_3 &\leftrightarrow t_2 = x2_i * -(g_i) \\
x_4 &\leftrightarrow t_4 = x2_i * g_i \\
x_5 &\leftrightarrow t_5 = t_1 - t_2 \\
x_6 &\leftrightarrow t_6 = t_3 + t_4 \\
x_7 &\leftrightarrow Re_1 = x1_r + t_5 \\
x_8 &\leftrightarrow Re_2 = x1_r - t_5 \\
x_9 &\leftrightarrow Im_1 = x1_i + t_6 \\
x_{10} &\leftrightarrow Im_2 = x1_i - t_6
\end{aligned}
$$

Then the reservation table 4.10 can be implemented using the pseudo code shown in figure 4.3. In this example, the input data is stored as real-imaginary and P_1 and P_2 point to $x1_r$ and $x2_r$ respectively. The constants are stored consecutively as $g_r, g_i, \cdots$ and P_3 initially points to g_r. The number of compute cycles needed to realize $I_n \hat{\otimes} F_2 D_2^j$ using this scheduling approach is $6n + 4$, which closely approximates the optimal $6n$ for perfectly scheduled operations.

□

Algorithm (4.40) also leads to an efficient implementation on the model II RISC architecture. Recall that for this model the parallelism of the adder and multiplier is restricted to multiply-accumulate instructions of the form $y = (a * b) + c$. If the standard Cooley-Tukey algorithm (4.42) were implemented on a model II RISC it would still be necessary to compute 5120 F_2's and 4097 complex multiplications. Complex multiplication can be realized in 4 multiply accumulate (MAC) instructions; an F_2 can be realized in 4 additions. Thus, in this case, the number of MAC instructions needed to implement algorithm (4.42) is the same as for model I. This is

Adder	Multiplier
	$t_1 = P_2 * P_3 + +$
	$t_3 = P_2 + + * P_3$
	$t_2 = -P_2 * P_3 - -$
	$t_4 = P_2 + +3 * P_3 + +2$
$t_5 = t_1 - t_2$	
$t_6 = t_3 + t_4$	
Do $n-1$ times	
$Re1 = P_1 + t_5$	$t_1 = P_2 * P_3 + +$
$Re2 = P_1 + + - t_5$	$t_3 = P_2 + + * P_3$
$Im1 = P_1 + t_6$	$t_2 = -P_2 * P_3 - -$
$Im2 = P_1 + 3 - t_6$	$t_4 = P_2 + +3 * P_3 + +2$
$t_5 = t_1 - t_2$	
$t_6 = t_3 + t_4$	
end Do	
$Re1 = P_1 + t_5$	
$Re2 = P_1 + + - t_5$	
$Im1 = P_1 + t_6$	
$Im2 = P_1 + 3 - t_6$	

Figure 4.3: Pseudo code for table 4.10

$$F_{1K} \leftrightarrow 4 * 5120 + 4 * 4097 = 36,868$$

Similarly, if algorithm (4.43) were implemented on a model II RISC computer it would still be necessary to compute 1023 (F_2)'s and 4097 $(F_2 D_2^j)'s$. The structure of $(F_2 D_2^j)$, however, can now be exploited to yield a more efficient implementation. In this case note that $(F_2 D_2^j)\bar{x}$ can be written as

$$(F_2 D_2^j)\bar{x} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & g_r & -g_i \\ 0 & 0 & g_i & g_r \end{pmatrix} \begin{bmatrix} x_{1r} \\ x_{1i} \\ x_{2r} \\ x_{2i} \end{bmatrix}$$

$$(F_2 D_2^j)\bar{x} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{pmatrix} 1 & 0 & g_r & -g_i \\ 0 & 1 & g_i & g_r \\ 1 & 0 & -g_r & g_i \\ 0 & 1 & -g_i & -g_r \end{pmatrix} \begin{bmatrix} x_{1r} \\ x_{1i} \\ x_{2r} \\ x_{2i} \end{bmatrix} \tag{4.44}$$

Expression (4.44) can now be implemented in six MAC instructions using the operations given below.

$$\begin{aligned} t_1 &= (x_{2r} * g_r) + x_{1r} \\ y_0 &= -(x_{2i} * g_i) + t_1 \\ t_2 &= (x_{2r} * g_i) + x_{1i} \\ y_1 &= (x_{2i} * g_r) + t_2 \\ y_2 &= (2 * x_{1r}) - y_0 \\ y_3 &= (2 * x_{1i}) - y_1 \end{aligned}$$

Thus, as in the case of a model I implementation, algorithm (4.43) can be realized in

$$F_{1K} \leftrightarrow 4 * 1023 + 6 * 4097 = 28,674$$

instructions.

4.4.3 Twiddle Multiplication

In accessing the performance of all of the DFT algorithms presented in this section, it was assumed throughout that the twiddle constants were precomputed and available in external memory. Although this assumption leads to a reduction in the required number of compute cycles, it also creates two undesirable side effects:

1. The number of twiddle constants is proportional to the size of the transform being implemented. Thus, for large transforms, a considerable amount of memory is needed to store these values.

2. Because the constants reside in external memory each one must be retrieved before it is used. This arrangement creates more demand for memory bandwidth which, as discussed in section 4.2, may not be available.

Because of these undesirable features, many practical implementations of FFT algorithms avoid the use of pre-computed twiddle constants and instead compute these constants at run time. Methods of efficiently realizing this on model I and model II RISC architectures are developed in this section.

In order to avoid the negative side effects mentioned above, it would be beneficial to be able to compute the entire twiddle stage from a few *seed* values. The seed values would be loaded into local registers and the entire twiddle stage would be computed from these. The structure of the twiddle stages, as revealed through the tensor product representation, makes this a relatively straight forward task.

In section 2.3.2 it was shown that the general twiddle stage $T_{n,s}$ was given by

$$T_{n,s} = \sum_{j=0}^{s-1} \oplus D^j_{n,n/s} = \begin{pmatrix} D^0_{n,n/s} & & & & \\ & D^1_{n,n/s} & & & \\ & & D^2_{n,n/s} & & \\ & & & \ddots & \\ & & & & D^{s-1}_{n,n/s} \end{pmatrix}$$

$$D^j_{n,r} = diag[1, \omega^j_n, \omega^{2j}_n, \cdots, \omega^{(r-1)j}_n] \qquad \omega_n = e^{-\frac{2\pi i}{n}}, \quad n = r \times s$$

Thus, $T_{n,s}$ is a diagonal matrix of complex numbers of the form ω^{jk}_n. Using the property

$$\omega^{j(0)}_n = 1$$
$$\omega^{jk}_n = \omega^j_n * \omega^{j(k-1)}_n$$

it is possible to compute every element of the block $D^j_{n,r}$ from ω^j_n. The element ω^j_n becomes the seed value for the block $D^j_{n,r}$; this is loaded into registers or local memory and iteritively multiplied to itself in order to generate the elements of the matrix $D^j_{n,r}$. This can be realized on model I by using the following instructions:

$seed_{re} = real(\omega^j_n)$	
$seed_{im} = imag(\omega^j_n)$	
$tw_{re} = 1$	
$tw_{im} = 0$	
Adder	Multiplier
	$t_1 = seed_{re} * tw_{re}$
	$t_2 = seed_{im} * tw_{im}$
$tw_{re} = t_1 + t_2$	$t_3 = seed_{re} * tw_{im}$
	$t_4 = seed_{im} * tw_{re}$
Do $r-1$ times	
$tw_{im} = t_3 + t_4$	$t_1 = seed_{re} * tw_{re}$
	$t_2 = seed_{im} * tw_{im}$
$tw_{re} = t_1 + t_2$	$t_3 = seed_{re} * tw_{im}$
	$t_4 = seed_{im} * tw_{re}$
end Do	
$tw_{im} = t_3 + t_4$	

In this implementation, each of the non-trivial twiddle terms requires four compute cycles in order to be determined. It is possible, however, to use some well known results from multiplicative complexity theory in order to realize a more efficient implementation of this on a model I RISC.

Let

$$k_1 = r_1 + j(I_1)$$
$$k_2 = r_2 + j(I_2)$$

represent two complex numbers where r is the magnitude of the real part and I is the magnitude of the imaginary. Then, the multiplication $k_1 * k_2$ can be realized using the identities

$$re = r_1 r_2 - I_1 I_2 = r_1(r_2 - I_2) + I_2(r_1 - I_1) \tag{4.45}$$
$$im = r_1 I_2 + r_2 I_1 = -r_1(r_2 - I_2) + r_2(r_1 + I_1)$$

Multiplier		m_1	m_2	m_3		
Adder	a_1				a_2	a_3

Table 4.11: Forbidden list = {1,2,4,5}

Assume that k_1 is fixed and known prior to the compute time. Then identities (4.45) imply that the complex multiplication can be carried out in (3,3) operations. These are given by

- Pre-compute

$$t_1 = r_1 + I_1$$
$$t_2 = r_1 - I_1$$

- Compute

$$\begin{array}{lllllll} m_1 & = & r_2 * t_1 & & a_1 & = & r_2 - I_2 \\ m_2 & = & r_1 * a_1 & & tw_{re} & = & m_1 - m_2 \\ m_3 & = & I_2 * t_2 & & tw_{im} & = & m_2 + m_3 \end{array}$$

In order to realize an efficient implementation on model I these operations must be properly scheduled to exploit the parallelism of the adder and the multiplier. An appropriate reservation table to achieving this is given in table 4.11. Pseudo code for this table is given in figure 4.4.

The scheme proposed above will work for general complex multiplication on model I RISCs provided that one of the sequences is known prior to computation. Is there any way to improve to performance of complex multiplication on a model II RISC ? In general, it is not possible to realize complex multiplication in three multiply-accumulate instructions. It is possible, however, to use some special properties of the twiddle factors to realize the twiddle factor generation in three multiply-accumulate instructions. This is developed below:

Let two complex numbers be denoted by

$$k_1 = r_1 + j(I_1)$$
$$k_2 = r_2 + j(I_2)$$

load t_1	
load t_2	
$t_3 = real(\omega_n^j)$	
$tw_{re} = 1$	
$tw_{im} = 0$	
Adder	Multiplier
$a_1 = tw_{re} - tw_{im}$	
	$m_1 = tw_{re} * t_1$
	$m_2 = t_3 * a_1$
Do $r-1$ times	
$a_1 = tw_{re} - tw_{im}$	$m_3 = tw_{im} * t_2$
$tw_{re} = m_1 - m_2$	$m_1 = tw_{re} * t_1$
$tw_{im} = m_2 + m_3$	$m_2 = t_3 * a_1$
end Do	
	$m_3 = tw_{im} * t_2$
$tw_{re} = m_1 - m_2$	
$tw_{im} = m_2 + m_3$	

Figure 4.4: Pseudo-code for table 4.11

then the product $(k_1 * k_2) = re + j(im)$ is given by

$$re = r_1 r_2 - I_1 I_2$$
$$im = r_1 I_2 + r_2 I_1$$

Introduce multiply accumulate instructions which are of the form $a+(b*c)$. This is done by noting that

$$re = [r_1 r_2 - I_1 I_2 - I_2] + I_2$$
$$re = r_1 \left[r_2 + \frac{I_2(I_1 + 1)}{-r_1} \right] + I_2$$

The computation of re thus consumes two multiply accumulate instructions, leaving one for the computation of im. In order to realize this it is necessary to determine a function of $f(r_1, I_1)$ such that

$$im = -\left[r_2 + \frac{I_2(I_1 + 1)}{-r_1} \right] + f(r_1, I_1)[r_1 r_2 - I_1 I_2] \tag{4.46}$$

This is established by using the fact that

$$im = r_1 I_2 + r_2 I_1 \tag{4.47}$$

and equating the coefficients of r_2 and I_2 of (4.46) and (4.47). By equating the coefficients of r_2 it is determined that

$$I_1 = -1 + f(r_1, I_1) r_1$$

which implies

$$f(r_1, I_1) = \frac{(I_1 + 1)}{r_1} \tag{4.48}$$

Similarly, equating the coefficients of I_2 yields

$$r_1 = \frac{(I_1 + 1)}{r_1} - f(r_1, I_1) I_1$$

which implies

$$f(r_1, I_1) = \frac{\frac{(I_1+1)}{-r_1} - r_1}{I_1} \tag{4.49}$$

In order to realize the computation in three MAC instruction, (4.48) must equal (4.49); this yields conditions on k_1. In particular,

$$r_1\left(\frac{(I_1+1)}{r_1}-r_1\right)=I_1(i_1+1)$$

$$I_1+1-r_1^2=I_1^2+I_1$$

or

$$I_1^2+r_1^2=1 \tag{4.50}$$

If condition (4.50) is satisfied then, the multiplication $(k_1 * k_2)$ can be carried out in three multiply accumulate instructions. This is indeed the case for the twiddle terms of the matrix $T_{n,s}$. Instructions for realizing this on model II are given below.

- Pre-compute

$$f_1=\frac{I_1+1}{r_1}$$

- Compute

$$\begin{aligned} t &= r_2-I_2*f_1 \\ re &= I_2+r_1*t \\ im &= -t+f_1*re \end{aligned}$$

Although the use of this technique leads to a 25% reduction in the number of compute cycles needed to realize twiddle generation on model II, there is an obvious problem with the procedure: The division required in order to pre-compute the constant f_1 introduces numerical instability into the procedure. The error due to this can be quite large especially in the neighborhood of $r_1 = 0$. In fact, if $r_1 = 0$, $f_1 = \infty$. This prohibits the use of this as a general procedure. In the special case of twiddle generation the approach is useful because the seed values for a stage $T_{n,s}$ are not usually in the vicinity of $-j$.

This approach was used in order to generate twiddle constants for the stage $T_{32,2}$. The resulting constants were compared to those generated by using standard complex multiplication. The results of this comparison show that the error of the resulting complex numbers does not exceed 0.000000000000001. These constants and the resulting errors are given in appendix D.

Chapter 5

Other Recursive Transforms

5.1 Introduction

In chapter two the tensor product was shown to be a very useful tool for modeling many very different types of DFT algorithms. In chapter three the breadth of the tensor product approach was extended to include the very important linear convolution operator. In particular, it was shown that expressing the linear convolution matrix in terms of tensor products makes it possible to use many of the techniques originally developed for DFT in order to implement convolution. In chapter four the tensor product was also shown to be an important tool for designing new algorithms which exploit many of the features of modern RISC architectures.

In this, the final chapter, the work presented thus far is extended by demonstrating that the tensor product is not limited to modeling and designing DFT and convolution algorithms. In fact, through several examples, it is shown that the tensor product technique can be exploited when recursion is prevalent in an algorithm. This greatly widens the scope of the ideas presented as many important algorithms rely on recursive structure.

Although tensor product factorizations of several different computations will be presented, the method in which the tensor product is introduced and exploited is the same in each case. This general methodology is as follows:

1. Express the desired computation as a matrix multiplication $\bar{y} = (M_n)\bar{x_n}$, where M_n is a matrix specifying the computation, $\bar{x_n}$ is a given input vector of order n, and $\bar{y}$ a desired output vector.

2. Use some property of the matrix M_n in order to introduce recursion. Because the computation is modeled as matrix multiplication, the recursion introduced results in a factorization of M_n in terms of M_{n/n_1}.

3. Iteratively use the recursion introduced in step 2 to fully decompose a large M_n in terms of smaller M_{n/n_i}'s. The tensor product is introduced at this stage in order to "hold together" the matrix structure of M_n.

4. Use the underlying algebraic structure of the tensor product representation to modify the structure of the original algorithm for efficient hardware or software implementation.

In this chapter we will be focusing on steps 1,2 and 3. As was shown throughout this work, the details of step 4 are highly machine dependent and will not be considered in detail. It is possible, however, to draw analogies from the work already presented in chapters 2, 3, and 4. A knowledge of these and the basic factorizations and methodology presented in this chapter represent a formidable set of tools for designing and implementing many different types of efficient digital signal processing algorithms.

5.2 Walsh-Hadamard Transform

Perhaps one of the most straightforward example of using the tensor product to model an algorithm is the case of the Walsh- Hadamard Transform (WHT). This is because recursion is inherent in the definition of the transform and therefore, there is no need to perform step 2 as outlined in the introduction of this chapter. The WHT is defined as follows; let $n = 2^\alpha$ and

$$W_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

then W_n is given by

$$W_n = \begin{pmatrix} W_{n/2} & W_{n/2} \\ W_{n/2} & -W_{n/2} \end{pmatrix} \tag{5.1}$$

As is presented in [51] W_{2^α} can be expressed as a tensor product of W_2 and $W_{2^{\alpha-1}}$, in particular

$$W_{2^\alpha} = (W_2 \otimes W_{2^{\alpha-1}}) \tag{5.2}$$

Similarly,

$$W_{2^{\alpha-1}} = (W_2 \otimes W_{2^{\alpha-2}}) \tag{5.3}$$

The substitution of (5.3) into (5.2) yields

$$W_{2^\alpha} = (W_2 \otimes (W_2 \otimes W_{2^{\alpha-2}})) \tag{5.4}$$

and by the associative law of the tensor product this is

$$W_{2^\alpha} = (W_2 \otimes W_2 \otimes W_{2^{\alpha-2}}) \tag{5.5}$$

Since

$$W_{2^{\alpha-2}} = (W_2 \otimes W_{2^{\alpha-3}}) \tag{5.6}$$

this process can be repeated until W_{2^α} is fully decomposed, ex.

$$W_{2^\alpha} = (W_2 \otimes W_2 \otimes W_2 \otimes \cdots \otimes W_2) \tag{5.7}$$

As mentioned above this expression is given in reference [51]. An alternate form of this will now be derived. This alternate expression will be more suitable for implementation and designing variants.

By the associative law

$$W_{2^\alpha} = (W_2 \otimes (W_2 \otimes W_2 \otimes \cdots \otimes W_2)) \tag{5.8}$$

Next, note that

$$W_{2^\alpha} = (W_2 I_2 \otimes I_{2^{\alpha-1}} (W_2 \otimes W_2 \otimes \cdots \otimes W_2)) \tag{5.9}$$

where I_n is the n point identity matrix. Using the distributive property of the tensor product this can be factored as

$$W_{2^\alpha} = (W_2 \otimes I_{2^{\alpha-1}}) (I_2 \otimes W_2 \otimes W_2 \otimes \cdots \otimes W_2) \tag{5.10}$$

Similarly,

$$(I_2 \otimes W_2 \otimes W_2 \otimes \cdots \otimes W_2) = (I_2 \otimes W_2 I_2 \otimes I_{2^{\alpha-2}} (W_2 \otimes \cdots \otimes W_2)) \tag{5.11}$$

$$= (I_2 \otimes (W_2 \otimes I_{2^{\alpha-2}})) (I_2 \otimes I_2 \otimes W_2 \otimes \cdots \otimes W_2) \tag{5.12}$$

$$= (I_2 \otimes W_2 \otimes I_{2^{\alpha-2}})(I_4 \otimes W_2 \otimes \cdots \otimes W_2) \tag{5.13}$$

and therefore,

$$W_{2^\alpha} = (W_2 \otimes I_{2^{\alpha-1}})(I_2 \otimes W_2 \otimes I_{2^{\alpha-2}})(I_4 \otimes W_2 \otimes \cdots \otimes W_2) \tag{5.14}$$

Continuing in this way W_{2^α} is decomposed as

$$W_{2^\alpha} = (W_2 \otimes I_{2^{\alpha-1}})(I_2 \otimes W_2 \otimes I_{2^{\alpha-2}})(I_4 \otimes W_2 \otimes I_{2^{\alpha-3}}) \cdots \tag{5.15}$$

$$\cdots (I_{2^{\alpha-2}} \otimes W_2 \otimes I_2)(I_{2^{\alpha-1}} \otimes W_2)$$

which is called the fundamental factorization of the WHT and is written more concisely as

Alorithm 5.2.1 *Fundamental WHT*

$$W_{2^\alpha} = \prod_{i=0}^{\alpha-1} (I_{2^i} \otimes W_2 \otimes I_{2^{\alpha-i-1}})$$

Once in this representation the fundamental factorization can be easily manipulated to obtain many interesting variants. As an illustration of this note that the commutative theorem yields

$$(I_{2^i} \otimes [W_2 \otimes I_{2^{\alpha-i-1}}]) = P_{2^\alpha,2^{\alpha-i}}(W_2 \otimes I_{2^{\alpha-1}})P_{2^\alpha,2^i}$$

This gives a vectorized WHT algorithm as discussed in [2,53].

Alorithm 5.2.2 *Vectorized WHT*

$$W_{2^\alpha} = \prod_{i=0}^{\alpha-1} P_{2^\alpha,2^{\alpha-i}}(W_2 \otimes I_{2^{\alpha-1}})P_{2^\alpha,2^i}$$

Also note that

$$([I_{2^i} \otimes W_2] \otimes I_{2^{\alpha-i-1}}) = P_{2^\alpha,2^{i+1}}(I_{2^{\alpha-1}} \otimes W_2)P_{2^\alpha,2^{\alpha-i-1}}$$

gives a parallel variant of algorithm 5.2.1

Alorithm 5.2.3 *Parallel WHT*

$$W_{2^\alpha} = \prod_{i=0}^{\alpha-1} P_{2^\alpha,2^{i+1}}(I_{2^{\alpha-1}} \otimes W_2)P_{2^\alpha,2^{\alpha-i-1}}$$

Many other variants can be easily derived. In particular note that the tensor product representation makes it possible to use some of the techniques developed in chapter 4 for RISC architectures.

The same procedure, outlined here for WHT, is used for all the other transforms considered. In the other cases, however, recursion is not inherent in the definition of the transform but instead has to be introduced through some "artificial means". This can be done either by exploiting some symmetry possessed by the transformation, as is the case for DFT, DCT, and DHT, or by using multiplicative complexity theory, as introduced in chapter 2, to obtain multiplicatively efficient cores and decomposing the transform in terms of these, as is done for linear convolution and Strassen matrix multiplication.

5.3 Discrete Hartley Transform

Let $\bar{x_n}$ and $\bar{y_n}$ be vectors of length n, then the discrete Hartley transform (DHT) of $\bar{x_n}$ is given by

$$y(k) = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} H_n(k,j)x(j) \qquad H_n(k,j) = cos(\frac{2\pi jk}{n}) + sin(\frac{2\pi jk}{n}) \quad (5.16)$$

As pointed out in Hou [52], this definition of H_n imposes useful symmetry. In particular,

- $H_n(k,2j) = H_{n/2}(k,j)$
- $H_n(k+n/2,2j) = H_{n/2}(k,j)$
- $H_n(k,2j+1) = -H_n(k+n/2,2j+1)$
- $H_n(k,2j+1) = D_k H_{n/2}(k,j)$

These imply the matrix H_n is of the form

$$(H_n)(P_{n,2})^{-1} = \begin{pmatrix} H_{n/2} & D_{n/2}H_{n/2} \\ H_{n/2} & -D_{n/2}H_{n/2} \end{pmatrix}$$

where

$$D_{n/2} = diag\left(cos(\frac{2\pi k}{n})\right) + diag\left(sin(\frac{2\pi k}{n})\right)(1 \oplus J_{n/2-1}) \qquad 0 \leq k < n/2$$

and J_n is the exchange matrix of order n, specified by

$$J_n = \begin{pmatrix} & & 1 \\ & 1 & \\ & \vdots & \\ 1 & & \end{pmatrix} \tag{5.17}$$

Thus, as in the case of the radix two DFT of chapter 2, H_n can be factored as

$$H_n = \begin{pmatrix} I_{n/2} & I_{n/2} \\ I_{n/2} & -I_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & \\ & D_{n/2} \end{pmatrix} \begin{pmatrix} H_{n/2} & \\ & H_{n/2} \end{pmatrix} P_{n,2}$$

or

$$H_n = \big(H_2 \otimes I_{n/2}\big) T_n \big(I_2 \otimes H_{n/2}\big) P_{n,2} \tag{5.18}$$

$$T_n = (I_{n/2} \oplus D_{n/2})$$

Assume $n = 2^\alpha$, then (5.18) can be used to fully decompose H_{2^α}, ex.

$$H_{2^\alpha} = (H_2 \otimes I_{2^{\alpha-1}}) T_{2^\alpha} (I_2 \otimes H_{2^{\alpha-1}}) P_{2^{\alpha-1},2} \tag{5.19}$$

$$H_{2^{\alpha-1}} = (H_2 \otimes I_{2^{\alpha-2}}) T_{2^{\alpha-1}} (I_2 \otimes H_{2^{\alpha-2}}) P_{2^{\alpha-2},2} \tag{5.20}$$

$$H_{2^{\alpha-2}} = (H_2 \otimes I_{2^{\alpha-3}}) T_{2^{\alpha-2}} (I_2 \otimes H_{2^{\alpha-3}}) P_{2^{\alpha-3},2} \tag{5.21}$$

$$\vdots$$

$$H_4 = (H_2 \otimes I_2) T_4 (I_2 \otimes H_2) P_{4,2} \tag{5.22}$$

The substitution of (5.20) into (5.19) yields

$$H_{2^\alpha} = (H_2 \otimes I_{2^{\alpha-1}}) T_{2^\alpha} (I_2 \otimes H_2 \otimes I_{2^{\alpha-2}}) (I_2 \otimes T_{2^{\alpha-1}}) (I_4 \otimes H_{2^{\alpha-2}}) \tag{5.23}$$

$$(I_2 \otimes P_{2^{\alpha-1}}) P_{2^{\alpha-1},2}$$

Similarly, the substitution of (5.21) into (5.23) yields

$$H_{2^\alpha} = (H_2 \otimes I_{2^{\alpha-1}}) T_{2^\alpha} (I_2 \otimes H_2 \otimes I_{2^{\alpha-2}}) (I_2 \otimes T_{2^{\alpha-1}}) \tag{5.24}$$

$$(I_4 \otimes H_2 \otimes I_{2^{\alpha-3}})(I_4 \otimes T_{2^{\alpha-2}})(I_8 \otimes H_{2^{\alpha-3}})$$

$$(I_4 \otimes P_{2^{\alpha-3},2})(I_2 \otimes P_{2^{\alpha-2},2})P_{2^{\alpha-1},2}$$

Continuing this yields the factorization

$$H_{2^\alpha} = (H_2 \otimes I_{2^{\alpha-1}})T_{2^\alpha}(I_2 \otimes H_2 \otimes I_{2^{\alpha-2}})(I_2 \otimes T_{2^{\alpha-1}}) \tag{5.25}$$

$$(I_4 \otimes H_2 \otimes I_{2^{\alpha-3}})(I_4 \otimes T_{2^{\alpha-2}})(I_8 \otimes H_{2^{\alpha-3}})\cdots$$

$$\cdots(I_{2^{\alpha-2}} \otimes H_2 \otimes I_2)(I_{2^{\alpha-2}} \otimes T_4)(I_{2^{\alpha-1}} \otimes H_2)$$

$$(I_{2^{\alpha-2}} \otimes P_{4,2})(I_{2^{\alpha-3}} \otimes P_{8,2})(I_{2^{\alpha-4}} \otimes P_{16,2})\cdots P_{2^{\alpha-1},2}$$

which can be written more concisely as

Alorithm 5.3.1 *Fundamental DHT*

$$H_{2^\alpha} = \left(\prod_{i=0}^{\alpha-1}(I_{2^i} \otimes H_2 \otimes I_{2^{\alpha-i-1}})(I_{2^i} \otimes T_{2^{\alpha-i}})\right) Q$$

$$Q = \prod_{i=0}^{\alpha-2}(I_{2^{\alpha-2-i}} \otimes P_{2^{i+2},2})$$

Note the striking resemblance between the fundamental DFT radix two factorization of chapter two and algorithm 5.3.1 . Once in the tensor product representation it becomes possible to use the techniques introduced in [2,53], as well as the methods introduced in chapter 4, to derive variants of the fundamental DHT algorithm optimized for particular architectures.

5.4 Discrete Cosine Transform

Let $\bar{x_n}$ and $\bar{y_n}$ be vectors of length n, then the discrete cosine transform (DCT) of $\bar{x_n}$ is given by

$$y(k) = \frac{2\epsilon_k}{n} \sum_{j=0}^{n-1} C_n(k,j)x(j) \qquad C_n(k,j) = cos\left(\frac{\pi k(2j+1)}{2n}\right) \tag{5.26}$$

$$\epsilon_k = \begin{cases} 1/\sqrt{2} & k = 0 \\ 1 & \text{otherwise} \end{cases}$$

Define

$$\tilde{C}_n(k,j) = C_n(k,j)Q_n^{-1}$$
$$Q_n = (I_{n/2} \oplus J_{n/2})P_{n,2}$$

where I_n is the identity matrix of order n, J_n is the exchange matrix (5.17) of order n, and $P_{n,2}$ is the stride permutation. As is shown in [54],

$$\tilde{C}_n(k,j) = cos\left(\frac{\pi k(4j+1)}{2n}\right)$$

By Hou [55], the symmetry conditions

- $\tilde{C}_n(2k,j) = \tilde{C_{n/2}}(k,j)$
- $\tilde{C}_n(2k,j+n/2) = \tilde{C_{n/2}}(k,j)$
- $\tilde{C}_n(2k+1,j) = -\tilde{C}_n(2k+1,j+n/2)$
- $\tilde{C}_n(2k+1,j) = 2\tilde{C_{n/2}}(k,j)D_j - \tilde{C}_n(2k-1,j)$

imply $\tilde{C}_n$ is of the form

$$P_{n,2}\tilde{C}_n = \begin{pmatrix} \tilde{C_{n/2}} & \tilde{C_{n/2}} \\ \tilde{L_{n/2}}\tilde{C_{n/2}}\tilde{D_{n/2}} & -\tilde{L_{n/2}}\tilde{C_{n/2}}\tilde{D_{n/2}} \end{pmatrix}$$

where

$$\tilde{D}_n = diag\left(cos\left(\frac{\pi(j+1/4)}{n}\right)\right) \qquad 0 \le j < n$$

$$\tilde{L}_n(k,j) = \begin{cases} 0 & j > k \\ (-1)^k & j = 0 \\ 2(-1)^{k-j} & j \le k \end{cases} \qquad 0 \le j,k < n$$

$P_{n,2}\tilde{C}_n$ can now be factored as

$$P_{n,2}\tilde{C}_n = \begin{pmatrix} I_{n/2} & \\ & \tilde{L}_{n/2} \end{pmatrix} \begin{pmatrix} \tilde{C}_{n/2} & \\ & \tilde{C}_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & \\ & \tilde{D}_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & I_{n/2} \\ I_{n/2} & -I_{n/2} \end{pmatrix}$$

Which can be written as

$$\tilde{C}_n = P_{n,n/2}L_n\big(I_2 \otimes \tilde{C}_{n/2}\big)D_n\big(F_2 \otimes I_{n/2}\big) \tag{5.27}$$

$$L_n = (I_{n/2} \oplus \tilde{L}_{n/2})$$
$$D_n = (I_{n/2} \oplus \tilde{D}_{n/2})$$

Using $\tilde{C}_n(k,j) = C_n(k,j)Q^{-1}$ yields

$$C_n = P_{n,n/2}L_n\big(I_2 \otimes C_{n/2}\big)T_n\big(F_2 \otimes I_{n/2}\big)Q_n \tag{5.28}$$
$$T_n = (I_2 \otimes Q_{n/2}^{-1})D_n$$

The iteritive use of (5.28) gives rise to a fundamental factorization of C_n for $n = 2^\alpha$. Let $n = 2^\alpha$, then (5.28) yields

$$C_{2^\alpha} = P_{2^\alpha,2^{\alpha-1}}L_{2^\alpha}(I_2 \otimes C_{2^{\alpha-1}})T_{2^\alpha}(F_2 \otimes I_{2^\alpha})Q_{2^\alpha} \tag{5.29}$$

$$C_{2^{\alpha-1}} = P_{2^{\alpha-1},2^{\alpha-2}}L_{2^{\alpha-1}}(I_2 \otimes C_{2^{\alpha-2}})T_{2^{\alpha-1}}(F_2 \otimes I_{2^{\alpha-1}})Q_{2^{\alpha-1}} \tag{5.30}$$

$$C_{2^{\alpha-2}} = P_{2^{\alpha-2},2^{\alpha-3}}L_{2^{\alpha-2}}(I_2 \otimes C_{2^{\alpha-3}})T_{2^{\alpha-2}}(F_2 \otimes I_{2^{\alpha-2}})Q_{2^{\alpha-2}} \tag{5.31}$$

$$\vdots$$

$$C_4 = P_{4,2}L_4(I_2 \otimes C_2)T_4(F_2 \otimes I_2)Q_4 \tag{5.32}$$

The substitution of (5.30) into (5.29) yields

$$C_{2^\alpha} = P_{2^\alpha,2^{\alpha-1}}L_{2^\alpha}\,(I_2 \otimes P_{2^{\alpha-1},2^{\alpha-2}})\,(I_2 \otimes L_{2^{\alpha-1}}) \tag{5.33}$$

$$(I_4 \otimes C_{2^{\alpha-2}})(I_2 \otimes T_{2^{\alpha-1}})(I_2 \otimes F_2 \otimes I_{2^{\alpha-1}})(I_2 \otimes Q_{2^{\alpha-1}})$$
$$T_{2^\alpha}(F_2 \otimes I_{2^\alpha})Q_{2^\alpha}$$

This process can be repeated until C_{2^α} is fully decomposed to yield:

Alorithm 5.4.1 *Fundamental DCT*

$$C_{2^\alpha} = \tilde{B}(I_{2^{\alpha-1}} \otimes C_2)\tilde{A}$$

$$\tilde{B} = \prod_{i=0}^{\alpha-2} \left(I_{2^i} \otimes P_{2^{\alpha-i},2^{\alpha-i-1}}\right)(I_{2^i} \otimes L_{2^{\alpha-i}})$$

$$\tilde{A} = \prod_{i=0}^{\alpha-2} (I_{2^{\alpha-2-i}} \otimes T_{2^{i+2}})(I_{2^{\alpha-i-2}} \otimes F_2 \otimes I_{2^{i+1}})(I_{2^{\alpha-2-i}} \otimes Q_{2^{i+2}})$$

Again, once in this form, the tensor product representation allows easy derivation of many variants of the basic DCT algorithm.

5.5 Strassen Matrix Multiplication

In [45] Strassen gives a recursive algorithm for computing the product for two arbitrary square matrices of order $n = 2^\alpha$. Recently, Huang *et al* [44] show that the Strassen algorithm can be expressed in terms of a tensor product decomposition. As for Toom's convolution, the core of Strassen's method is based on a clever factorization of 2 by 2 matrix multiplication. In particular, let

$$\begin{pmatrix} y_0 & y_2 \\ y_1 & y_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_2 \\ a_1 & a_3 \end{pmatrix} \begin{pmatrix} x_0 & x_2 \\ x_1 & x_3 \end{pmatrix}$$

or

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_2 & & \\ a_1 & a_3 & & \\ & & a_0 & a_2 \\ & & a_1 & a_3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Strassen showed that this matrix product can be computed using 7 multiplications. His idea can be expressed as the matrix factorization

$$\begin{bmatrix} a_0 & a_2 & & \\ a_1 & a_3 & & \\ & & a_0 & a_2 \\ & & a_1 & a_3 \end{bmatrix} = BDA \tag{5.34}$$

$$D = diag\left(\Gamma \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}\right)$$

$$B = \begin{bmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 1 & 0 & -1 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\Gamma = \begin{bmatrix} 0 & 0 & 1 & -1 \\ 1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

By iterating on this core factorization Huang [44] *et al* get a tensor product representation of Strassen's algorithm.

Alorithm 5.5.1 *Fundamental Strassen Multiplication*

$$(I_{2^\alpha} \otimes M_{2^\alpha}) = \tilde{B}D\tilde{A}$$

$$\tilde{B} = \prod_{i=1}^{\alpha} (I_{7^{i-1}} \otimes B \otimes I_{2^{2(\alpha-i)}})$$

$$\tilde{A} = \prod_{i=0}^{\alpha-1} (I_{7^{\alpha-i-1}} \otimes A \otimes I_{2^{2i}})$$

$$D = diag\left(\prod_{i=0}^{\alpha-1}(I_{7^{\alpha-i-1}} \otimes \Gamma \otimes I_{2^{2i}})\, a\right)$$

Some variants of this algorithm are given in [44]. It is also possible to draw analogy from the linear convolution algorithms developed in chapter 4 in order to derive varients which are optimized for RISC architectues.

Chapter 6

Concluding Discussion

6.1 Role of the Tensor Product

The use of fast algorithms for digital signal processing has revolutionized the way electrical engineers design digital systems. As a by-product of significantly reducing the computational effort required, digital systems employing fast algorithms need much less hardware than would otherwise be required. This leads to an increase in the reliability and flexibility of the entire system, and at the same time, reduces the cost and power consumption. These advantages can be critical in many applications.

Although the use of fast algorithms offers many benefits, there are also difficulties associated with their use. The design and implementation of fast algorithms is difficult and requires a very broad range of skills in areas as diverse as number theory to assemble language programming. The lack of unifying language or format to represent the many different types of fast algorithms that are available has also been a consistent problem. In many cases, these facts have deterred electrical and software engineers from using fast algorithms in their designs in spite of the apparent potential gains. The tensor product offers a vehicle around these obstacles.

Because there is a very definite connection between certain tensor product constructs and important computer architectures, an algorithm can be represented in terms of tensor products and modified for efficient implementation on many types of radically different computers. This is made possible through the linguistic power of the tensor product. The ability to easily and accurately modify a fast algorithm makes it infinitely more

appealing for practical use.

The unifying role of the tensor product is illustrated in figure 6.2. The left column lists several important types of algorithms that have been developed for DFT over the past three decades. The tensor product provides a structured way of representing all of these algorithms. The right column lists several important transforms. By virtue of the decompositions developed, the tensor product facilitates the use any of the algorithms given in the left column in order to realize any of the transforms given in the right column.

In this dissertation, the use of the tensor product approach to algorithm design was promoted in several ways. In chapter three, a tensor product decomposition of the linear convolution matrix was presented. The algorithms derived using the tensor product approach were shown to offer a reduction in the amount of computation needed to realize linear convolution. It was also demonstrated that the underlying algebraic structure of the tensor product representation leads to variants of the basic algorithm which are suitable for implementation on vector and multi processor computers.

In chapter four, the breadth of the tensor product was enhanced to include the important RISC architecture. By introducing computational models, representative of commercially available RISC machines, it was shown that the tensor product again leads to algorithms with enhanced performance.

In chapter five, tensor product decompositions of several other important transforms are developed. A general procedure for using the tensor product to model an entire class of recursive algorithms was presented.

The ideas presented in this dissertation offer many avenues for further research. A few of these are mentioned below.

- Although the tensor product was shown to offer enhanced performance of many different types of algorithms, no claims as to the optimality of these algorithms was made. It would be interesting to establish lower bounds on the arithmetic complexity of the computations considered here and see if these optimal algorithms are attainable through tensor product techniques.

- Although algorithms were developed for computational models, actual implementation of these on commercially available RISC chips

will offer new challenges.

- Tensor product representations of several important algorithms were presented in chapter five. This opens the door to designing many new algorithms, optimized for particular computer organizations, for these transforms.
- Further enhance the scope of the tensor product approach to include more algorithms and architectures.
- Develop a tensor product compiler. Such a compiler would use well defined algebraic rules when optimizing an algorithm for implementation and would thus represent a significant advancement.

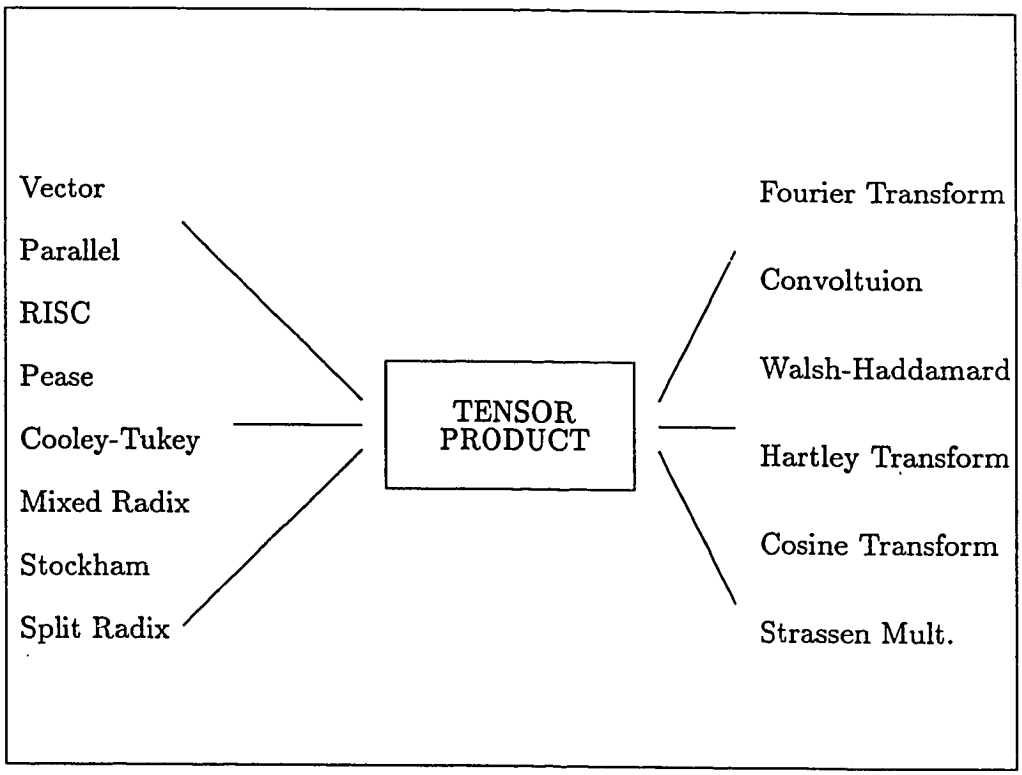

Figure 6.1: The Unifying Role of the Tensor Product

Appendix A

In appendix A, the operations count reported in table 3.1 are justified by presenting the actual algorithms.

A.1 Direct Implementation

The first column of table 3.1 reflects the number of arithmetic operations needed when directly multiplying the n point convolution matrix and input data. For a n point convolution it can be verified that this requires n^2 multiplications and $(n-1)^2$ additions. Thus, the total number of operations for a direct implementation is

$$operations = n^2 + (n-1)^2$$

A.2 By Convolution Theorem

The second column reflects the number of operations for implementation of the convolution theorem. In this approach the n point sequences are zero padded to length $2n$, two $2n$ point Fourier transforms are used to carry the sequences into the frequency domain where they are multiplied together point by point, and an inverse $2n$ point Fourier transform is used in order to the carry the answer back into the time domain.

Let the arithmetic complexity of an arbitrary stage M be given by the cartesian pair $M(m, a)$ where m is the number of multiplications and a is the number of additions. Furthermore, assume one of the sequences is available for pre-computation. The arithmetic complexity of linear convolution

by convolution theorem is then

$$C_n(m,a) = 2F_{2n}(m,a) + 2n(4,2)$$

where $F_{2n}(m,a)$ is the arithmetic complexity of performing an $2n$ point FFT and $(4,2)$ is the arithmetic complexity of complex multiplication.

The FFT algorithms used here were not straight forward implementations of the Cooley-Tukey algorithm. Better performance is obtainable by using various 2, 4 and 8 mixed-radix algorithms. In order to further boost performance, a Winograd F_8 was used as a core. These algorithms were developed in order to improve the performance of FFT implementation on the AT&T DSP32/DSP32C family of digital signal processors. These algorithms will run more than twice as fast as the standard Cooley-Tukey algorithms included as part of the DSP32 applications library.

Let $n = n_1 n_2$ then, as is shown by (2.80), the n point Fourier transform matrix can be factored as

$$F_n = (F_{n_1} \otimes I_{n_2}) T_{n,n_2} (I_{n_1} \otimes F_{n_2}) P_{n,n_2} \tag{A.1}$$

This expression will be used in order to establish $F_{2n}(m,a)$ for the various convolutions considered. Note that the arithmetic complexity of F_2, F_4, and F_8 is $F_2(0,4)$ $F_4(0,16)$ and $F_8(4,52)$, respectively.

- SIZE = 4,

$$C_4(m,a) = 2F_8(4,52) + 8(4,2) = 160$$

- SIZE = 8, $n_1 = 2$, $n_2 = 8$.

$$C_8(m,a) = 2F_{16}(m,a) + 16(4,2)$$

$$F_{16} = (F_2 \otimes I_8) T_{16,8} (I_2 \otimes F_8) P_{16,8}$$

$$F_{16}(m,a) = 8F_2(0,4) + 7(4,2) + 2F_8(4,52) = 186$$

$$C_8(m,a) = 2(186) + 16(4,2) = 468$$

- SIZE = 16, $n_1 = 4$, $n_2 = 8$.

$$C_{16}(m,a) = 2F_{32}(m,a) + 32(4,2)$$

$$F_{32} = (F_4 \otimes I_8)T_{32,8}(I_4 \otimes F_8)P_{32,8}$$

$$F_{32}(m,a) = 8F_4(0,16) + 21(4,2) + 4F_8(4,52) = 478$$

$$C_{16}(m,a) = 2(478) + 32(4,2) = 1148$$

- SIZE = 32, $n_1 = 8$, $n_2 = 8$.

$$C_{32}(m,a) = 2F_{64}(m,a) + 64(4,2)$$

$$F_{64} = (F_8 \otimes I_8)T_{64,8}(I_8 \otimes F_8)P_{64,8}$$

$$F_{64}(m,a) = 16F_8(4,52) + 49(4,2) = 1190$$

$$C_{32}(m,a) = 2(1190) + 64(4,2) = 2764$$

- SIZE = 64, $n_1 = 64$, $n_2 = 2$.

$$C_{64}(m,a) = 2F_{128}(m,a) + 128(4,2)$$

$$F_{128} = (F_{64} \otimes I_2)T_{128,2}(I_{64} \otimes F_2)P_{128,2}$$

$$F_{128}(m,a) = 2F_{64}(1190) + 63(4,2) + 64F_2(0,4) = 3014$$

$$C_{64}(m,a) = 2(3014) + 128(4,2) = 6796$$

- SIZE = 128, $n_1 = 64$, $n_2 = 4$.

$$C_{128}(m,a) = 2F_{256}(m,a) + 256(4,2)$$

$$F_{256} = (F_{64} \otimes I_4)T_{256,4}(I_{64} \otimes F_4)P_{128,4}$$

$$F_{256}(m,a) = 4F_{64}(1190) + 189(4,2) + 64(F_4(0,16) = 6918$$

$$C_{128}(m,a) = 2F_{256}(6918) + 256(4,2) = 15372$$

- SIZE = 256, $n_1 = 64$, $n_2 = 8$.

$$C_{256}(m,a) = 2F_{512}(m,a) + 512(4,2)$$

$$F_{512} = (F_{64} \otimes I_8)T_{512,8}(I_{64} \otimes F_8)P_{512,8}$$

$$F_{512}(m,a) = 8F_{64}(1190) + 441(4,2) + 64F_8(4,52) = 15750$$

$$C_{256}(m,a) = 2F_{512}(15750) + 512(4,2) = 34572$$

- SIZE = 512, $n_1 = 512$, $n_2 = 2$.

$$C_{512}(m,a) = 2F_{1024}(m,a) + 1024(4,2)$$

$$F_{1024} = (F_{512} \otimes I_2)T_{1024,2}(I_{512} \otimes F_2)P_{1024,2}$$

$$F_{1024}(m,a) = 2F_{512}(15750) + 511(4,2)512F_2(0,4) = 36614$$

$$C_{512}(m,a) = 2F_{1024}(36614) + 1024(4,2) = 79372$$

- SIZE $= 1024$, $n_1 = 512$, $n_2 = 4$.

$$C_{1024}(m,a) = 2F_{2048}(m,a) + 2048(4,2)$$

$$F_{2048} = (F_{512} \otimes I_4)T_{2048,4}(I_{512} \otimes F_4)P_{2048,4}$$

$$F_{2048}(m,a) = 4F_{512}(15750) + 1533(4,2) + 512F_4(0,16) = 80390$$

$$C_{1024}(m,a) = 2F_{2048}(80390) + 2048(4,2) = 173068$$

A.3 By Tensor Product

The final column of table 3.1 reflects the performance of a radix two family of convolution algorithms. First, define a radix two core factorization of the two point convolution matrix. This will be

$$\begin{bmatrix} h_0 & 0 \\ h_1 & h_0 \\ 0 & h_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_1 & 0 & 0 \\ 0 & k_2 & 0 \\ 0 & 0 & k_3 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{A.2}$$

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix}$$

Thus, in this case

$$B = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \qquad A = G = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$

and $\gamma = 3$. Using this core and

$$C_n = R_{n_1,n_2}(B \otimes I_{2n_1-1})(I_\gamma \hat{\otimes} C_{n_1,j})(A \otimes I_{n_1})$$

leads us to the following results.

- SIZE $= 4$, $n_1 = 2$, $n_2 = 2$.

$$C_4 = R_{2,2}(B \otimes I_3)(I_3 \hat{\otimes} C_{2,j})(A \otimes I_2)$$

$$C_4(m,a) = (0,2) + 3(0,2) + 3(4,1) + 2(0,1) = (12,13) = 25$$

- SIZE $= 8$, $n_1 = 4$, $n_2 = 2$.

$$C_8 = R_{4,2}(B \otimes I_7)(I_3 \hat{\otimes} C_{4,j})(A \otimes I_4)$$

$$C_8(m,a) = (0,6) + 7(0,2) + 3(16,9) + 4(0,1) = (48,51) = 99$$

- SIZE $= 16$, $n_1 = 8$, $n_2 = 2$.

$$C_{16} = R_{8,2}(B \otimes I_{15})(I_3 \hat{\otimes} C_{8,j})(A \otimes I_8)$$

$$C_{16}(m,a) = (0,14) + 15(0,2) + 3(48,51) + 8(0,1) = (144,205) = 349$$

- SIZE $= 32$, $n_1 = 16$, $n_2 = 2$.

$$C_{32} = R_{16,2}(B \otimes I_{31})(I_3 \hat{\otimes} C_{16,j})(A \otimes I_{16})$$

$$C_{32}(m,a) = (0,30) + 31(0,2) + 3(144,205) + 16(0,1) = (432,723) = 1155$$

- SIZE $= 64$, $n_1 = 32$, $n_2 = 2$.

$$C_{64} = R_{32,2}(B \otimes I_{63})(I_3 \hat{\otimes} C_{32,j})(A \otimes I_{32})$$

$$C_{64}(m,a) = (0,62) + 63(0,2) + 3(432,723) + 32(0,1) = (1296,2389) = 3685$$

- SIZE = 128, $n_1 = 64$, $n_2 = 2$.

$$C_{128} = R_{64,2}(B \otimes I_{127})(I_3 \hat{\otimes} C_{64,j})(A \otimes I_{64})$$

$$C_{128}(m,a) = (0,126) + 127(0,2) + 3(1296,2389) + 64(0,1) =$$
$$(3888,7611) = 11499$$

- SIZE = 256, $n_1 = 128$, $n_2 = 2$.

$$C_{256} = R_{128,2}(B \otimes I_{255})(I_3 \hat{\otimes} C_{128,j})(A \otimes I_{128})$$

$$C_{256}(m,a) = (0,254) + 255(0,2) + 3(3888,7611) + 128(0,1) =$$
$$(11664,23725) = 35389$$

- SIZE = 512, $n_1 = 512$, $n_2 = 2$.

$$C_{512} = R_{256,2}(B \otimes I_{511})(I_3 \hat{\otimes} C_{256,j})(A \otimes I_{256})$$

$$C_{512}(m,a) = (0,510) + 511(0,2) + 3(11664,23725) + 256(0,1) =$$
$$(34992,72963) = 107955$$

- SIZE = 1024, $n_1 = 512$, $n_2 = 2$.

$$C_{1024} = R_{512,2}(B \otimes I_{1023})(I_3 \hat{\otimes} C_{512,j})(A \otimes I_{512})$$

$$C_{1024}(m,a) = (0,1022) + 1023(0,2) + 3(34992,72963) + 512(0,1) =$$
$$(104976,222469) = 327445$$

Appendix B

In appendix B the algorithms summarized in tables 4.3 and 4.4 are presented. These algorithms are based on the Chinese Remainder Theorem.

B.1 Three Point Convolution

- **Example B.1.1** Let

$$M(x) = x^4 - 5x^2 + 4$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{aligned} m_0(x) &= (x+1) \\ m_1(x) &= (x-1) \\ m_2(x) &= (x+2) \\ m_3(x) &= (x-2) \end{aligned}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{aligned} \alpha_0(x) &= 1/6(x^3 - x^2 - 4x + 4) \\ \alpha_1(x) &= -1/6(x^3 + x^2 - 4x - 4) \\ \alpha_2(x) &= -1/12(x^3 - 2x^2 - x + 2) \\ \alpha_3(x) &= 1/12(x^3 + 2x^2 - x - 2) \end{aligned}$$

This gives rise the matrix factorization $C_3 = B(D_5)A$ where

$$A = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & -2 & 4 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & -4 & 2 & -2 & 4 \\ -4 & -4 & -1 & -1 & 0 \\ -1 & 1 & -2 & 2 & -5 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{bmatrix} = \left(\begin{bmatrix} 1/6 \\ -1/6 \\ -1/12 \\ 1/12 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & -2 & 4 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_3)\bar{x}$ can be realized in 12 multiplications and 16 additions by using the operations below.

- Stage A: $(m, a) = (2, 6)$
  Let $\bar{y_5} = (A)\bar{x_3}$ and use:

$$\begin{array}{lll} t_1 = x_0 + x_2 & & t_3 = 2 * x_1 \\ y_0 = -x_1 + t_1 & & y_2 = t_2 - t_3 \\ y_1 = x_1 + t_2 & & y_3 = t_2 + t_3 \\ t_2 = 4 * x_2 + x_0 & & y_4 = x_2 \end{array}$$

- Stage B: $(m, a) = (5, 10)$
  Let $\bar{y_5} = (B)\bar{x_5}$ and use:

$$\begin{array}{lcllcl} t_1 & = & x_0 + x_1 & t_9 & = & t_2 - t_5 \\ t_2 & = & -4 * t_1 & t_{10} & = & t_3 - t_7 \\ t_3 & = & x_0 - x_1 & t_{11} & = & t_1 + t_5 \\ t_4 & = & -4 * t_3 & y_0 & = & 4 * x_4 + t_8 \\ t_5 & = & x_2 + x_3 & y_1 & = & t_9 \\ t_6 & = & x_2 - x_3 & y_2 & = & -5 * x_4 + t_{10} \\ t_7 & = & 2 * t_6 & y_3 & = & t_{11} \\ t_8 & = & t_4 + t_7 & y_4 & = & x_4 \end{array}$$

- **Example B.1.2** Let

$$M(x) = x^4 - 1$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{array}{lcl} m_0(x) & = & (x^2 + 1) \\ m_1(x) & = & (x^2 - 1) \end{array}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{array}{lcl} \alpha_0(x) & = & -1/2(x^2 - 1) \\ \alpha_1(x) & = & 1/2(x^2 + 1) \end{array}$$

This gives rise the matrix factorization $C_3 = B(D_7)A$ where

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & -1 & 1 & 0 & 1 & 1 & -1 \\ -1 & 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag\begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \end{bmatrix} = \left(\begin{bmatrix} -1/2 \\ -1/2 \\ -1/2 \\ 1/2 \\ 1/2 \\ 1/2 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \\ -1 & 1 & -1 \\ -1 & 1 & -1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_3)\bar{x}$ can be realized in 7 multiplications and 13 additions by using the operations below.

- Stage A: $(m, a) = (0, 4)$
  Let $\bar{y_7} = (A)\bar{x_3}$ and use:

$$\begin{array}{lcllcl} t_1 & = & x_0 - x_2 & y_3 & = & t_2 \\ t_2 & = & x_0 + x_2 & y_4 & = & x_1 \\ y_0 & = & x_1 & y_5 & = & t_2 + x_1 \\ y_1 & = & t_1 & y_6 & = & x_2 \\ y_2 & = & t_1 + x_1 & & & \end{array}$$

- Stage B: $(m, a) = (0, 9)$
  Let $\bar{y_5} = (B)\bar{x_7}$ and use:

$$\begin{array}{lcllcl} t_1 & = & x_1 - x_2 & y_0 & = & t_5 - x_6 \\ t_2 & = & x_0 + x_2 & y_1 & = & t_4 - t_2 \\ t_3 & = & x_4 + x_5 & y_2 & = & t_3 + t_1 \\ t_4 & = & x_3 + x_5 & y_3 & = & t_2 + t_4 \\ t_5 & = & t_2 - t_1 & y_4 & = & x_6 \end{array}$$

- **Example B.1.3** Let

$$M(x) = x^4 - x^2$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{array}{lcl} m_0(x) & = & x^2 \\ m_1(x) & = & (x + 1) \\ m_2(x) & = & (x - 1) \end{array}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{aligned} \alpha_0(x) &= -1(x^2-1) \\ \alpha_1(x) &= -1/2(x^3-x^2) \\ \alpha_2(x) &= 1/2(x^3+x^2) \end{aligned}$$

This gives rise the matrix factorization $C_3 = B(D_6)A$, where

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 1 & -1 \\ -1 & -1 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \end{bmatrix} = \left(\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1/2 \\ 1/2 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_3)\bar{x}$ can be realized in 6 multiplications and 9 additions by using the operations below.

- Stage A: $(m, a) = (0, 3)$
  Let $\bar{y_6} = (A)\bar{x_3}$ and use:

$$\begin{aligned} t_1 &= x_0 + x_2 & y_2 = y_1 &= x_0 \\ t_2 &= x_1 + t_1 & y_3 &= t_3 \\ t_3 &= -x_1 + t_1 & y_4 &= t_2 \\ y_0 &= x_1 & y_5 &= x_2 \end{aligned}$$

– Stage B: $(m, a) = (0, 6)$
Let $\bar{y_5} = (B)\bar{x_6}$ and use:

$$\begin{array}{lll} t_1 = x_0 + x_1 & & t_4 = x_2 + x_5 \\ y_0 = x_2 & & y_2 = t_2 - t_5 \\ y_1 = t_1 & & y_3 = t_3 - t_1 \\ t_2 = x_3 + x_4 & & y_4 = x_5 \\ t_3 = -x_3 + x_4 & & \end{array}$$

- **Example B.1.4** Let

$$M(x) = x^4 + 2x^3 - x^2 - 2x$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{array}{lll} m_0(x) & = & x \\ m_1(x) & = & (x+1) \\ m_2(x) & = & (x-1) \\ m_3(x) & = & (x+2) \end{array}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{array}{lll} \alpha_0(x) & = & -1/2(x^3 + 2x^2 - x - 2) \\ \alpha_1(x) & = & 1/2(x^3 + x^2 - 2x) \\ \alpha_2(x) & = & 1/6(x^3 + 3x^2 + 2x) \\ \alpha_3(x) & = & -1/6(x^3 - x) \end{array}$$

This gives rise the matrix factorization $C_3 = B(D_5)A$, where

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} -2 & 0 & 0 & 0 & 0 \\ -1 & -2 & 2 & -1 & -2 \\ 2 & 1 & 3 & 0 & -1 \\ 1 & 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{bmatrix} = \left(\begin{bmatrix} -1/2 \\ 1/2 \\ 1/6 \\ -1/6 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_3)\bar{x}$ can be realized in 12 multiplications and 15 additions by using the operations below.

- Stage A: $(m, a) = (2, 5)$
  Let $\bar{y_5} = (A)\bar{x_3}$ and use:

$$\begin{array}{lcllcl} t_1 & = & x_0 + x_2 & y_1 & = & t_1 - x_1 \\ t_2 & = & -2 * x_1 & y_2 & = & t_1 + x_1 \\ t_3 & = & 4 * x_2 & y_3 & = & x_0 + t_4 \\ t_4 & = & t_2 + t_3 & y_4 & = & x_2 \\ y_0 & = & x_0 & & & \end{array}$$

- Stage B: $(m, a) = (5, 10)$
  Let $\bar{y_5} = (A)\bar{x_5}$ and use:

$$\begin{array}{lcllcl} t_1 & = & -2 * x_0 & t_{10} & = & t_9 + t_4 \\ t_2 & = & -2 * x_1 & t_{11} & = & x_1 + x_2 \\ t_3 & = & 2 * x_2 & t_{12} & = & t_{11} - t_5 \\ t_4 & = & 3 * x_2 & y_0 & = & t_1 \\ t_5 & = & -2 * x_4 & y_1 & = & t_7 - t_8 \\ t_6 & = & t_2 + t_3 & y_2 & = & t_{10} - x_4 \\ t_7 & = & t_6 + t_5 & y_3 & = & t_{12} + t_8 \\ t_8 & = & x_0 + x_3 & y_4 & = & x_4 \\ t_9 & = & -t_1 + x_1 & & & \end{array}$$

- **Example B.1.5** Let

$$M(x) = x^5 - x$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{aligned} m_0(x) &= (x^2+1) \\ m_1(x) &= (x-1) \\ m_2(x) &= (x+1) \\ m_3(x) &= x \end{aligned}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{aligned} \alpha_0(x) &= 1/2(x^4 - x^2) \\ \alpha_1(x) &= 1/4(x^4 + x^3 + x^2 + x) \\ \alpha_2(x) &= 1/4(x^4 - x^3 + x^2 - x) \\ \alpha_3(x) &= -1(x^4 - 1) \end{aligned}$$

This gives rise the matrix factorization $C_3 = B(D_6)A$, where

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & -1 & 0 \\ 0 & -1 & 1 & 1 & 1 & 0 \\ -1 & 0 & -1 & 1 & -1 & 0 \\ 0 & 1 & -1 & 1 & 1 & 1 \end{bmatrix}$$

$$D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \end{bmatrix} = \left(\begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/4 \\ 1/4 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_3)\bar{x}$ can be realized in 12 multiplications and 15 additions by using the operations below.

- Stage A: $(m, a) = (0, 5)$
  Let $\bar{y_6} = (A)\bar{x_3}$ and use:

$$\begin{array}{lll} t_1 = x_0 + x_2 & & y_1 = t_2 \\ t_2 = x_0 - x_2 & & y_2 = t_2 + x_1 \\ t_3 = t_1 + x_1 & & y_3 = t_3 \\ t_4 = t_1 - x_1 & & y_4 = t_4 \\ y_0 = x_1 & & y_5 = x_0 \end{array}$$

- Stage B: $(m, a) = (0, 9)$
  Let $\bar{y_5} = (B)\bar{x_6}$ and use:

$$\begin{array}{lll} t_1 = x_1 + x_2 & & y_0 = x_5 \\ t_2 = x_0 + x_2 & & y_1 = t_3 + t_2 \\ t_3 = x_3 - x_4 & & y_2 = t_4 - t_1 \\ t_4 = x_3 + x_4 & & y_3 = t_3 - t_2 \\ t_5 = t_1 - x_5 & & y_4 = t_4 + t_5 \end{array}$$

B.2 Four Point Convolution

- **Example B.2.1** Let

$$M(x) = x^6 - x^2$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{array}{lll} m_0(x) = x^2 \\ m_1(x) = (x^2 + 1) \\ m_2(x) = (x^2 - 1) \end{array}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{array}{lll} \alpha_0(x) = -1(x^4 - 1) \\ \alpha_1(x) = 1/2(x^4 - x^2) \\ \alpha_2(x) = 1/2(x^4 + x^2) \end{array}$$

This gives rise the matrix factorization $C_4 = BD_{10}A$, where

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 1 & 1 & -1 & -1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & -1 & 0 & 1 & 1 & 0 \\ -1 & -1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \\ k_8 \\ k_9 \\ k_{10} \end{bmatrix} = \left(\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_4)\bar{x}$ can be realized in 10 multiplications and 18 additions by using the operations below.

– Stage A: $(m, a) = (0, 6)$

Let $\bar{y_{10}} = (A)\bar{x_4}$ and use:

$$\begin{array}{lcllcl} t_1 & = & x_0 + x_2 & y_4 & = & t_2 \\ t_2 & = & x_0 - x_2 & y_5 & = & t_2 + t_4 \\ t_3 & = & x_1 + x_3 & y_6 & = & t_1 \\ t_4 & = & x_1 - x_3 & y_7 & = & t_3 \\ y_0 & = & x_1 & y_8 & = & t_1 + t_3 \\ y_1 = y_2 & = & x_0 & y_9 & = & x_3 \\ y_3 & = & t_4 & & & \end{array}$$

– Stage B: $(m, a) = (0, 12)$
Let $\bar{y_7} = (A)\bar{x_{10}}$ and use:

$$\begin{array}{lcllcl} t_1 & = & x_0 + x_1 & t_9 & = & t_5 + t_3 \\ t_2 & = & x_4 - x_5 & y_0 & = & x_2 \\ t_3 & = & x_3 + x_5 & y_1 & = & t_1 \\ t_4 & = & x_7 + x_8 & y_2 & = & t_7 - x_9 \\ t_5 & = & x_6 + x_8 & y_3 & = & t_8 \\ t_6 & = & t_4 + t_2 & y_4 & = & t_6 - x_2 \\ t_7 & = & t_4 - t_2 & y_5 & = & t_9 - t_1 \\ t_8 & = & t_5 - t_3 & y_6 & = & x_9 \end{array}$$

- **Example B.2.2** Let

$$M(x) = x^6 + 4x^5 - 5x^4 - 20x^3 + 4x^2 + 16$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{array}{lcl} m_0(x) & = & x \\ m_1(x) & = & (x+1) \\ m_2(x) & = & (x-1) \\ m_3(x) & = & (x+2) \\ m_4(x) & = & (x-2) \\ m_5(x) & = & (x+4) \end{array}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$
\begin{aligned}
\alpha_0(x) &= 1/16(x^5 + 4x^4 - 5x^3 - 20x^2 + 4x + 16) \\
\alpha_1(x) &= -1/18(x^5 + 3x^4 - 8x^3 - 12x^2 + 16x) \\
\alpha_2(x) &= -1/30(x^5 + 5x^4 - 20x^2 - 16x) \\
\alpha_3(x) &= -1/48(x^5 + 2x^4 - 9x^3 - 2x^2 + 8x) \\
\alpha_4(x) &= 1/144(x^5 + 6x^4 + 7x^3 - 6x^2 - 8x) \\
\alpha_5(x) &= -1/720(x^5 - 5x^3 + 4x)
\end{aligned}
$$

This gives rise the matrix factorization $C_4 = BD_{10}A$, where

$$
A = \begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 \\
1 & 1 & 1 & 1 \\
1 & -2 & 4 & -8 \\
1 & 2 & 4 & 8 \\
1 & -4 & 16 & -64 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

$$
B = \begin{bmatrix}
16 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 16 & -16 & 8 & -8 & 4 & 16 \\
-20 & -12 & -20 & -2 & -6 & 0 & 4 \\
-5 & -8 & 0 & -9 & 7 & -5 & -20 \\
4 & 3 & 5 & 2 & 6 & 0 & -5 \\
1 & 1 & 1 & 1 & 1 & 1 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

$$
D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \end{bmatrix} = \left(\begin{bmatrix} 1/16 \\ -1/18 \\ -1/30 \\ 1/48 \\ 1/144 \\ -1/720 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 \\ 1 & 2 & 4 & 8 \\ 1 & -4 & 16 & -64 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} \right)
$$

On the basis of these, the product $\bar{y} = (C_4)\bar{x}$ can be realized in 32 multiplications and 36 additions by using the operations below.

– Stage A: $(m, a) = (6, 11)$
Let $\bar{y_7} = (A)\bar{x_4}$ and use:

$$
\begin{aligned}
t_1 &= x_0 + x_2 & t_{10} &= -4 * x_1 \\
t_2 &= x_1 + x_3 & y_1 &= t_1 - t_2 \\
t_3 &= 4 * x_2 & y_2 &= t_1 + t_2 \\
t_4 &= 2 * x_1 & y_3 &= t_6 - t_7 \\
t_5 &= 8 * x_3 & y_4 &= t_6 + t_7 \\
t_6 &= x_0 + t_3 & y_5 &= x_0 + t_8 + t_9 + t_{10} \\
t_7 &= t_4 + t_5 & y_6 &= x_3 \\
t_8 &= -64 * x_3 & y_0 &= x_0 \\
t_9 &= 16 * x_3
\end{aligned}
$$

– Stage B: $(m, a) = (19, 26)$
Let $\bar{y_7} = (A)\bar{x_7}$ and use:

$$
\begin{aligned}
t_1 &= 16 * x_0 & t_{21} &= 7 * x_4 \\
t_2 &= x_0 + x_5 & t_{22} &= -5 * t_5 \\
t_3 &= x_1 + x_2 & t_{23} &= t_{22} + t_{21} + t_{20} \\
t_4 &= x_3 + x_4 & t_{24} &= x_2 - x_6 \\
t_5 &= 4 * x_6 & t_{25} &= 5 * t_{24} \\
t_6 &= 4 * t_2 & t_{26} &= 4 * x_0 \\
t_7 &= x_1 - x_2 + x_6 & t_{27} &= 3 * x_1 \\
t_8 &= x_3 - x_4 & t_{28} &= 2 * x_3 \\
t_9 &= 8 * t_8 & t_{29} &= 6 * x_4 \\
t_{10} &= 16 * t_7 & t_{30} &= t_{27} + t_{28} + t_{29} \\
t_{11} &= x_0 + x_2 & y_0 &= t_1 \\
t_{12} &= -20 * t_{11} & y_1 &= t_6 + t_9 + t_{10} \\
t_{13} &= -12 * x_1 & y_2 &= t_{12} + t_{16} + t_5 \\
t_{14} &= -2 * x_3 & y_3 &= t_{23} + t_{18} + t_{19} \\
t_{15} &= -6 * x_4 & y_4 &= t_{30} + t_{26} + t_{25} \\
t_{16} &= t_{13} + t_{14} + t_{15} & y_5 &= t_2 + t_3 + t_4 + t_5 \\
t_{17} &= x_0 + x_5 & y_6 &= x_6 \\
t_{18} &= -5 * t_{17} \\
t_{19} &= -8 * x_1 \\
t_{20} &= -9 * x_3
\end{aligned}
$$

- **Example B.2.3** Let

$$M(x) = x^6 - 21x^4 + 84x^2 - 64$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$\begin{aligned} m_0(x) &= (x+1) \\ m_1(x) &= (x-1) \\ m_2(x) &= (x+2) \\ m_3(x) &= (x-2) \\ m_4(x) &= (x+4) \\ m_5(x) &= (x-4) \end{aligned}$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{aligned} \alpha_0(x) &= -1/90(x^5 - x^4 - 20x^3 + 20x^2 + 64x - 64) \\ \alpha_1(x) &= 1/90(x^5 + x^4 - 20x^3 - 20x^2 + 64x + 64) \\ \alpha_2(x) &= 1/144(x^5 - 2x^4 - 17x^3 + 34x^2 + 16x - 32) \\ \alpha_3(x) &= -1/144(x^5 - 4x^4 - 5x^3 + 20x^2 + 4x - 16) \\ \alpha_4(x) &= -1/1440(x^5 - 4x^4 - 5x^3 + 20x^2 + 4x - 16) \\ \alpha_5(x) &= 1/1440(x^5 + 4x^4 - 5x^3 - 20x^2 + 4x + 16) \end{aligned}$$

This gives rise the matrix factorization $C_4 = BD_7A$, where

$$A = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 \\ 1 & 2 & 4 & 8 \\ 1 & -4 & 16 & -64 \\ 1 & 4 & 16 & 64 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} -64 & 64 & -32 & 32 & -16 & 16 & -64 \\ 64 & 64 & 16 & 16 & 4 & 4 & 0 \\ 20 & -20 & 34 & -34 & 20 & -20 & 84 \\ -20 & -20 & -17 & -17 & -5 & -5 & 0 \\ -1 & 1 & -2 & 2 & -4 & 4 & -21 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \end{bmatrix} = \left(\begin{bmatrix} -1/90 \\ 1/90 \\ 1/144 \\ -1/144 \\ -1/1440 \\ 1/1440 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 \\ 1 & 2 & 4 & 8 \\ 1 & -4 & 16 & -64 \\ 1 & 4 & 16 & 64 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_4)\bar{x}$ can be realized in 30 multiplications and 39 additions by using the operations below.

- Stage A: $(m, a) = (6, 18)$
  Let $\bar{y_7} = (A)\bar{x_4}$ and use:

$$\begin{array}{lcllcl} t_1 & = & 2 * x_1 & t_{13} & = & x_2 - x_3 \\ t_2 & = & 2 * t_1 & t_{14} & = & x_2 + x_3 \\ t_3 & = & 4 * x_2 & t_{15} & = & t_3 - t_5 \\ t_4 & = & 4 * t_3 & t_{16} & = & t_3 + t_5 \\ t_5 & = & 8 * x_3 & t_{17} & = & t_4 - t_6 \\ t_6 & = & 8 * t_5 & t_{18} & = & t_4 + t_6 \\ t_7 & = & x_0 - x_1 & y_0 & = & t_7 + t_{13} \\ t_8 & = & x_0 + x_1 & y_1 & = & t_8 + t_{14} \\ t_9 & = & x_0 - t_1 & y_2 & = & t_9 + t_{15} \\ t_{10} & = & x_0 + t_1 & y_3 & = & t_{10} + t_{16} \\ t_{11} & = & x_0 - t_2 & y_4 & = & t_{11} + t_{17} \\ t_{12} & = & x_0 + t_2 & y_5 & = & t_{12} + t_{18} \\ & & & y_6 & = & x_3 \end{array}$$

- Stage B: $(m, a) = (17, 21)$
  Let $\bar{y_7} = (B)\bar{x_7}$ and use:

$$
\begin{aligned}
t_1 &= x_1 + x_0 & t_{20} &= t_1 + t_7 + t_{14} \\
t_2 &= x_1 - x_0 & t_{21} &= t_2 + t_8 + t_{15} \\
t_3 &= -20 * t_1 & t_{22} &= t_3 + t_9 + t_{16} \\
t_4 &= -20 * t_2 & t_{23} &= t_4 + t_{10} + t_{17} \\
t_5 &= 64 * t_1 & t_{24} &= t_5 + t_{11} + t_{18} \\
t_6 &= 64 * t_2 & t_{25} &= t_6 + t_{12} + t_{19} \\
t'_6 &= x_2 - x_3 & y_0 &= -64 * x_6 + t_{25} \\
t_7 &= x_2 + x_3 & y_1 &= t_{24} \\
t_8 &= -2 * t'_6 & y_2 &= 84 * x_6 + t_{23} \\
t_9 &= -17 * t_7 & y_3 &= t_{22} \\
t_{10} &= 34 * t'_6 & y_4 &= -21 * x_6 + t_{21} \\
t_{11} &= 16 * t_7 & y_5 &= t_{20} \\
t_{12} &= -32 * t'_6 & y_6 &= x_6 \\
t_{13} &= x_4 - x_5 & & \\
t_{14} &= x_4 + x_5 & & \\
t_{15} &= -4 * t_{13} & & \\
t_{16} &= -5 * t_{14} & & \\
t_{17} &= 20 * t_{13} & & \\
t_{18} &= 4 * t_{14} & & \\
t_{19} &= -16 * t_{13} & &
\end{aligned}
$$

- **Example B.2.4** Let

$$M(x) = x^6 - 5x^4 + 4x^2$$

This can be factored into a product of relatively prime polynomials $M(x) = \prod_i m_i(x)$ where

$$
\begin{aligned}
m_0(x) &= x^2 \\
m_1(x) &= (x+1) \\
m_2(x) &= (x-1) \\
m_3(x) &= (x+2) \\
m_4(x) &= (x-2)
\end{aligned}
$$

The idempotents $\alpha_i(x)$ for the subrings $F[x]/m_i(x)$ are given by

$$\begin{aligned}
\alpha_0(x) &= 1/4(x^4 - 5x^2 + 4) \\
\alpha_1(x) &= 1/6(x^5 - x^4 - 4x^3 + 4x^2) \\
\alpha_2(x) &= -1/6(x^5 + x^4 - 4x^3 - 4x^2) \\
\alpha_3(x) &= -1/48(x^5 - 2x^4 - x^3 + 2x^2) \\
\alpha_4(x) &= 1/48(x^5 + 2x^4 - x^3 - 2x^2)
\end{aligned}$$

This gives rise the matrix factorization $C_4 = BD_8A$, where

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 \\ 1 & 2 & 4 & 8 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -5 & 4 & -4 & 2 & -2 & 4 \\ -5 & -5 & 0 & -4 & -4 & -1 & -1 & 0 \\ 0 & 0 & 1 & -1 & 1 & -2 & 2 & -5 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = diag \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \\ k_8 \end{bmatrix} = \left(\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/6 \\ -1/6 \\ -1/48 \\ 1/48 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 \\ 1 & 2 & 4 & 8 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} \right)$$

On the basis of these, the product $\bar{y} = (C_4)\bar{x}$ can be realized in 20 multiplications and 27 additions by using the operations below.

– Stage A: $(m, a) = (3, 12)$
Let $\bar{y_8} = (A)\bar{x_4}$ and use:

$$
\begin{array}{lcllcl}
t_1 & = & x_0 + x_1 & t_{10} & = & t_6 + t_7 \\
t_2 & = & x_0 - x_1 & t_{11} & = & t_6 - t_7 \\
t_3 & = & x_2 + x_3 & y_0 & = & x_1 \\
t_4 & = & x_2 - x_3 & y_1 = y_2 & = & x_0 \\
t_5 & = & 2 * x_1 & y_3 & = & t_2 + t_4 \\
t_6 & = & 4 * x_2 & y_4 & = & t_1 + t_3 \\
t_7 & = & 8 * x_3 & y_5 & = & t_9 + t_{11} \\
t_8 & = & x_0 + t_5 & y_6 & = & t_8 + t_{10} \\
t_9 & = & x_0 - t_5 & y_7 & = & x_3
\end{array}
$$

– Stage B: $(m, a) = (9, 15)$
Let $\bar{y_7} = (B)\bar{x_8}$ and use:

$$
\begin{array}{lcllcl}
t_1 & = & x_1 + x_0 & t_{12} & = & t_2 + t_6 \\
t_2 & = & x_4 + x_3 & t_{13} & = & 4 * x_7 + t_9 \\
t_3 & = & x_4 - x_3 & t_{14} & = & -5 * x_7 + t_{11} \\
t_4 & = & -4 * t_2 & y_0 & = & 4 * x_2 \\
t_5 & = & -4 * t_3 & y_1 & = & 4 * t_1 \\
t_6 & = & x_5 + x_6 & y_2 & = & -5 * x_2 + t_{13} \\
t_7 & = & x_5 - x_6 & y_3 & = & -5 * t_1 + t_{10} \\
t_8 & = & -2 * t_7 & y_4 & = & x_2 + t_{14} \\
t_9 & = & t_5 - t_8 & y_5 & = & t_1 + t_{12} \\
t_{10} & = & t_4 - t_6 & y_6 & = & x_7 \\
t_{11} & = & t_3 + t_8 & & &
\end{array}
$$

Appendix C

In appendix C the algorithms that led to the results of table 4.8 are presented.

C.1 By Convolution Thoerem

- SIZE $= 32$, $n_1 = 8$, $n_2 = 8$.

$$C_{32}(m,a) = 2F_{64}(m,a) + 64(4,2)$$

$$F_{64} = (F_8 \otimes I_8)T_{64,8}(I_8 \otimes F_8)P_{64,8}$$

$$F_{64}(m,a) = 16(52) + 49(4) = 1028$$

$$C_{32}(m,a) = 2(1028) + 64(4) = 2312$$

- SIZE $= 64$, $n_1 = 64$, $n_2 = 2$.

$$C_{64}(m,a) = 2F_{128}(m,a) + 128(4,2)$$

$$F_{128} = (F_{64} \otimes I_2)T_{128,2}(I_{64} \otimes F_2)P_{128,2}$$

$$F_{128}(m,a) = 2(1028) + 63(4) + 64(4) = 2564$$

$$C_{64}(m,a) = 2(2564) + 128(4) = 5640$$

- SIZE = 128, $n_1 = 64$, $n_2 = 4$.

$$C_{128}(m,a) = 2F_{256}(m,a) + 256(4,2)$$

$$F_{256} = (F_{64} \otimes I_4)T_{256,4}(I_{64} \otimes F_4)P_{128,4}$$

$$F_{256}(m,a) = 4(1028) + 189(4) + 64(16) = 5892$$

$$C_{128}(m,a) = 2(5892) + 256(4) = 12808$$

- SIZE = 256, $n_1 = 64$, $n_2 = 8$.

$$C_{256}(m,a) = 2F_{512}(m,a) + 512(4,2)$$

$$F_{512} = (F_{64} \otimes I_8)T_{512,8}(I_{64} \otimes F_8)P_{512,8}$$

$$F_{512}(m,a) = 8(1028) + 441(4) + 64(52) = 13316$$

$$C_{256}(m,a) = 2(13316) + 512(4) = 28680$$

- SIZE = 512, $n_1 = 512$, $n_2 = 2$.

$$C_{512}(m,a) = 2F_{1024}(m,a) + 1024(4,2)$$

$$F_{1024} = (F_{512} \otimes I_2)T_{1024,2}(I_{512} \otimes F_2)P_{1024,2}$$

$$F_{1024}(m,a) = 2(13316) + 511(4) + 512(4) = 30724$$

$$C_{512}(m,a) = 2(30724) + 1024(4) = 65544$$

- SIZE $= 1024$, $n_1 = 512$, $n_2 = 4$.

$$C_{1024}(m,a) = 2F_{2048}(m,a) + 2048(4,2)$$

$$F_{2048} = (F_{512} \otimes I_4)T_{2048,4}(I_{512} \otimes F_4)P_{2048,4}$$

$$F_{2048}(m,a) = 4(13316) + 1533(4) + 512(16) = 67588$$

$$C_{1024}(m,a) = 2F(67588) + 2048(4) = 143368$$

C.2 By Tensor Product

- SIZE $= 32$, $n_1 = 8$, $n_2 = 4$, and use the core of example B.2.4. In this case the complexity of B, D, and A, is $(9,15)$, $(8,0)$, and $(3,12)$ repectively. Then,

$$C_{32} = R_{8,4}(B \otimes I_{15})(I_8 \hat{\otimes} C_{8,j})(A \otimes I_8)$$

Use the balanced algorithm from (4.20) for $C_{8,j}$ to yield

$$C_{32}(m,a) = (42) + 15(15) + 8(48) + 8(12) = 747$$

- SIZE $= 64$, $n_1 = 4$, $n_2 = 16$, and use the core of example B.2.4. In this case the complexity of B, D, and A, is $(9,15)$, $(8,0)$, and $(3,12)$ repectively. Then,

$$C_{64} = R_{16,4}(B \otimes I_{31})(I_8 \hat{\otimes} C_{16,j})(A \otimes I_{16})$$

Use the balanced algorithm from (4.20) for $C_{16,j}$ to yield

$$C_{64}(m,a) = (90) + 31(15) + 8(192) + 16(12) = 2283$$

- SIZE = 128, $n_1 = 32$, $n_2 = 4$, and use the core of example B.2.3. In this case the complexity of B, D, and A, is $(17,21)$, $(7,0)$, and $(6,18)$ repectively. Then,

$$C_{128} = R_{32,4}(B \otimes I_{63})(I_7 \hat{\otimes} C_{32,j})(A \otimes I_{32})$$

Let $n_1 = 8$, $n_2 = 4$, and use the core of example B.2.4. In this case the complexity of B, D, and A, is $(9,15)$, $(8,0)$, and $(3,12)$ repectively. Then,

$$C_{32} = R_{8,4}(B \otimes I_{15})(I_8 \hat{\otimes} C_{8,j})(A \otimes I_8)$$

and

$$C_{32}(m,a) = (42) + 15(15) + 8(48) + 8(12) = 747$$

Which implies

$$C_{128}(m,a) = (186) + 63(21) + 7(747) + 32(18) = 6747$$

- SIZE = 256, $n_1 = 64$, $n_2 = 4$, and use the core of example B.2.3. In this case the complexity of B, D, and A, is $(17,21)$, $(7,0)$, and $(6,18)$ repectively. Then,

$$C_{256} = R_{64,4}(B \otimes I_{127})(I_7 \hat{\otimes} C_{64,j})(A \otimes I_{64})$$

and

$$C_{64}(m,a) = (90) + 31(15) + 8(192) + 16(12) = 2283$$

which implies

$$C_{256}(m,a) = (378) + 127(21) + 7(2283) + 64(18) = 20178$$

- SIZE = 512, $n_1 = 128$, $n_2 = 4$, and use the core of example B.2.3. In this case the complexity of B, D, and A, is $(17,21)$, $(7,0)$, and $(6,18)$ repectively. Then,

$$C_{512} = R_{128,4}(B \otimes I_{255})(I_7 \hat{\otimes} C_{128,j})(A \otimes I_{128})$$

and

$$C_{128}(m,a) = (186) + 63(21) + 7(747) + 32(18) = 6747$$

which imples

$$C_{512}(m,a) = (762) + 255(21) + 7(6747) + 128(18) = 55650$$

- SIZE = 1024, $n_1 = 256$, $n_2 = 4$, and use the core of example B.2.3. In this case the complexity of B, D, and A, is $(17,21)$, $(7,0)$, and $(6,18)$ repectively. Then,

$$C_{1024} = R_{256,4}(B \otimes I_{255})(I_7 \hat{\otimes} C_{256,j})(A \otimes I_{256})$$

and

$$C_{256}(m,a) = (378) + 127(21) + 7(2283) + 64(18) = 20178$$

which implies

$$C_{1024}(m,a) = (1530) + 255(21) + 7(20178) + 256(18) = 152739$$

Appendix D

In appendix D the twiddle constants generated by using the technique of section 4.4.3 are compared to the twiddle constants as generated by stright forward complex multiplication. Table D.3 gives the error between the two methods.

	REAL	IMAGINARY
ω_{16}^{0}	1.000000000000000000000000	0.000000000000000000000000
ω_{16}^{1}	0.980785280403230400000000	-0.195090322016128200000000
ω_{16}^{2}	0.923879532511286700000000	-0.382683432365089700000000
ω_{16}^{3}	0.831469612302545200000000	-0.555570233019602200000000
ω_{16}^{4}	0.707106781186547600000000	-0.707106781186547500000000
ω_{16}^{5}	0.555570233019602300000000	-0.831469612302545100000000
ω_{16}^{6}	0.382683432365089800000000	-0.923879532511286600000000
ω_{16}^{7}	0.195090322016128400000000	-0.980785280403230300000000
ω_{16}^{8}	0.000000000000001305379416	-0.999999999999999900000000
ω_{16}^{9}	-0.195090322016128100000000	-0.980785280403230300000000
ω_{16}^{10}	-0.382683432365089600000000	-0.923879532511286600000000
ω_{16}^{11}	-0.555570233019602000000000	-0.831469612302545100000000
ω_{16}^{12}	-0.707106781186547200000000	-0.707106781186547500000000
ω_{16}^{13}	-0.831469612302544900000000	-0.555570233019602200000000
ω_{16}^{14}	-0.923879532511286400000000	-0.382683432365089800000000
ω_{16}^{15}	-0.980785280403230100000000	-0.195090322016128400000000

Table D.1: Twiddles in 4 *MACS*

	REAL	IMAGINARY
ω_{16}^{0}	1.000000000000000000000000	0.000000000000000000000000
ω_{16}^{1}	0.980785280403230400000000	-0.195090322016128200000000
ω_{16}^{2}	0.923879532511286700000000	-0.382683432365089700000000
ω_{16}^{3}	0.831469612302545200000000	-0.555570233019602300000000
ω_{16}^{4}	0.707106781186547500000000	-0.707106781186547500000000
ω_{16}^{5}	0.555570233019602200000000	-0.831469612302545200000000
ω_{16}^{6}	0.382683432365089700000000	-0.923879532511286800000000
ω_{16}^{7}	0.195090322016128200000000	-0.980785280403230700000000
ω_{16}^{8}	-0.000000000000000001099923104	-1.000000000000000000000000
ω_{16}^{9}	-0.195090322016128400000000	-0.980785280403230700000000
ω_{16}^{10}	-0.382683432365089900000000	-0.923879532511287000000000
ω_{16}^{11}	-0.555570233019602400000000	-0.831469612302545500000000
ω_{16}^{12}	-0.707106781186547700000000	-0.707106781186547700000000
ω_{16}^{13}	-0.831469612302545500000000	-0.555570233019602400000000
ω_{16}^{14}	-0.923879532511287000000000	-0.382683432365089900000000
ω_{16}^{15}	-0.980785280403230700000000	-0.195090322016128300000000

Table D.2: Twiddles in 3 *MACS*

	REAL ERROR	IMAGINARY ERROR
ω_{16}^{0}	0.00000000000000005551115123	-0.00000000000000001314730659
ω_{16}^{1}	0.00000000000000005551115123	-0.00000000000000002775557562
ω_{16}^{2}	0.00000000000000005551115123	-0.00000000000000003330669074
ω_{16}^{3}	0.00000000000000005551115123	-0.00000000000000004440892099
ω_{16}^{4}	0.00000000000000004440892099	-0.00000000000000005551115123
ω_{16}^{5}	0.00000000000000003330669074	-0.00000000000000006661338148
ω_{16}^{6}	0.00000000000000001665334537	-0.00000000000000006661338148
ω_{16}^{7}	0.00000000000000000000000000	-0.00000000000000006661338148
ω_{16}^{8}	-0.00000000000000001422066674	-0.00000000000000007771561172
ω_{16}^{9}	-0.00000000000000003053113318	-0.00000000000000007771561172
ω_{16}^{10}	-0.00000000000000004996003611	-0.00000000000000006661338148
ω_{16}^{11}	-0.00000000000000006661338148	-0.00000000000000005551115123
ω_{16}^{12}	-0.00000000000000007771561172	-0.00000000000000004440892099
ω_{16}^{13}	-0.00000000000000008881784197	-0.00000000000000003330669074
ω_{16}^{14}	-0.00000000000000008881784197	-0.00000000000000001665334537
ω_{16}^{15}	-0.00000000000000008881784197	-0.00000000000000000277555756

Table D.3: Error Between Methods

Bibliography

[1] R. Blahut, "Fast Algorithms for Digital Signal Processing," *Addison-Wesley Publishing Company,* Massachusetts 1985.

[2] R. Tolimieri, M. An, and C. Lu " Algorithms for Discrete Fourier Transform and Convolution," *Springer-Verlag Publishing Company,* New York 1989.

[3] G. Birkoff, and S. Maclane," A Survey of Modern Algebra," *Macmillam Publishing Company,* New York 1953.

[4] N. Jacobson , "Lectures in Abstract Algebra," *Van Nostrand Publishing Company,* New Jersey 1955.

[5] P. Davis, "Circulant Matrices," *John Wiley & Sons,* New York 1979.

[6] S. Winograd, "Some Bilinear Forms Whose Multiplicative Complexity Depends on the Field of Constants," *Math Syst. Theor.,* vol.10, pp 169-180, 1978.

[7] S. Winograd, "On Computing the Discrete Fourier Transform," *Math Comp,* vol.32, num 141, pp 175-199, January 1978.

[8] S. Winograd, " Arithmetic Complexity of Computations," *CBMS-NSF Conference Series in Applied Mathematics SIAM* 1980.

[9] A. Toom, "The Complexity of a Scheme of Functional Elements Realizing the Multiplications of Integers," *Soviet Math. Dokl.,*4, pp. 714-716, 1963.

[10] L. Auslander, J. Cooley, and A. Silberger "Numerical Stability of Fast Convolution Algorithms for Digital Filtering, " *IEEE Workshop on VLSI,* pp. 172-213, November 1984.

[11] T. Stockham, "High Speed Convolution and Correlation," *Spring Joint Comput. Conf., AFIPS Conf. Proc.,*28, pp. 229-233, 1966.

[12] P. Papamichalis and C. Burrus, "Conversion of Digit-Reversed to Bit Reversed Order in FFT Algorithms," *IEEE ICASSP 89* , pp 984-987, 1989.

[13] L. E. Shar, "Design and Scheduling of Statically Configured Pipelines," *Digital Systems Lab Report,* SU-SEL-72-042, Standford University, September 1972.

[14] R. Agarwal and J. Cooley, "New Algorithms for Digital Convolution," *IEEE ASSP* , vol. 25, num. 5, pp 392-410, October 1977.

[15] B. Rice,"Some Good Fields and Rings for Computing Number Theoretic Transforms," *IEEE ASSP* , vol. 27, num. 4, pp. 432-433, August 1977.

[16] C. Rader,"Discrete Convolution via Mersenne Transforms ," *IEEE Trans. on Comp.* , vol. 21, num. 12, pp. 1269-1273, December 1972.

[17] R. Agarwal and C. Burrus,"Number Theoretic Transforms to Implement Fast Digital Convolution," *IEEE ASSP* , vol. 63, num. 4, pp. 550-560, April 1975.

[18] R. Blahut,"Algebraic Fields, Signal Processing, and Error Control ," *IEEE Proceedings* , vol. 73, num. 5, pp. 874-893, May 1985.

[19] S. Lee and H. Lu,"Fast Convolution Using Generalized Fermat/Mersenne Number Transforms," *IEEE ICASSP 88* , pp. 1910-1913, 1988.

[20] D. Rodriguez, *On Tensor Product Formulations of Additive Fast Fourier Transform Algorithms and their Implementations.* PhD thesis, E.E. Department. The City College of New York, of the City University of New York, 1987.

[21] J. Cooley and J. Tukey, "An Algorithm for the Machine Calculation of the Complex Fourier Series," *Math. Comp.*, vol. 19, pp. 297-301, April 1965.

[22] L. Auslander, and A. Silberger " On the Use of Scratchpad in the Construction of Convolution Algorithms," *Research Report RC 10554, IBM Watson Research Center,* May 1984.

[23] W.T. Cochran, et al, "What is the Fast Fourier Transform?"*IEEE Trans. Aud. and Elec.,* vol. 15, num. 2, pp. 45-55, June 1967.

[24] M. Heideman and S. Burrus, "On the Number of Multiplications Necessary to Compute a Length-2^n DFT," *IEEE ASSP* ., num. 34, vol. 1, pp. 91-95, February 1986.

[25] R. Singleton,"An Algorithm for Computing the Mixed Radix Fast Fourier Transform ," *IEEE Tran. Aud. and Elec.* , AU -17, pp. 93-103, 1969.

[26] M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," *JACM* , vol. 15, pp 252-264, April 1968.

[27] H. Sorensen, M. Heideman, and S. Burrus," On Computing the Split Radix FFT," *IEEE ASSP* , num. 34, vol. 1, pp 152-156, February 1986.

[28] M. Richards," On the Efficient Implementation of the Split Radix FFT," *IEEE ICASSP 87* , pp 1801-1804, 1987.

[29] M. Vetterli and P. Duhamel," Split Radix Algorithms for Length p^m DFT's," *IEEE ICASSP 88,* pp 1415-1418, 1988.

[30] C. Rader, "Discrete Fourier Transforms When the Number of Data Samples Is Prime," *IEEE Proceedings* , vol. 56, pp 1107-1108, June 1968.

[31] I. J. Good, "The Interaction Algorithm and Practical Fourier Analysis," *J. Royal Statist. Soc. Ser. B 20* , addendum 22 pp 372-375, 1960.

[32] L. H. Thomas, "Using a Computer to Solve Problems in Physics," *App. of Digital Computers* , Gin and Co. 1963.

[33] D. Kolba and T. Parks," A Prime Factor FFT Algorithm Using High Speed Convolution," *IEEE ASSP* , vol. 25, num. 4, pp 281-294, August 1977.

[34] C. Temperton," A Note on Prime Factor FFT Algorithms," *J. of Comp. Phy.* , vol. 52, pp 198-204, October 1983.

[35] R. Tolimieri, C. Lu, and, R. Johnson" Modified Winograd FFT Algorithm and Its Variants for Transform Size $N = p^k$ and Their Implementations," *Advances in Applied Mathematics* , vol. 10, pp 228-251, 1989.

[36] M. Heideman, *Applications of Multiplicative Complexity Theory to Convolution and Discrete Fourier Transform,* PhD thesis, Department of Electrical and Computer Engineering, Rice University, Houston TX, 1986.

[37] H. Johnson, *The Design of DFT Algorithms,* PhD thesis, Department of Electrical and Computer Engineering, Rice University, Houston TX, 1982.

[38] H. Johnson and S. Burrus," The Design of Optimal DFT Algorithms Using Dynamic Programming," *IEEE ASSP* , vol. 31, num. 2, pp 378-387, April 1983.

[39] J. Granata, M. Rofheart, and M. Conner "The Tensor Product as a Tool for Designing Efficient Fourier Transform Algorithms for Digital Signal Processing," In revision.

[40] *WE ATT DSP32/DSP32C Digital Signal Processor-Information Manual.* AT&T Technologies, Inc., 1988.

[41] *TMS320C30 The Third Generation of the TMS320 Family of Digital Signal Processors* . Texas Instruments Inc. , 1988.

[42] *Technical Summary- 96 Bit General-Purpose Floating Point DSP* Motorola Inc. 1988.

[43] *The Intel 860 Information Manual.* Intel. 1989.

[44] C. Huang, J. Johnson, and R. Johnson," A Tensor Product Formulation of Strassen's Matrix Multiplication Algorithm," In review.

[45] V. Strassen, "Gaussian Elimination is Not Optimal," *Numer. Math* ,vol. 13 , pp. 354-356, 1969.

[46] P. M. Kogge, "The Architecture of Pipelined Computers," *Hemishere Publishing Corporation,* New York 1981.

[47] D. Korn and J. Lambiotte," Computing the Fast Fourier Transform on a Vector Computer," *Math. Comput.* , vol. 33, pp 977-992, July 1979.

[48] P. Swarztrauber, "Vectorizing the FFTs in Parallel Computations," *Academic Press Publishing Company,* New York 1982.

[49] R. Agarwal and J. Cooley," Vectorized Mixed Radix Discrete Fourier Transform Algorithms," *Proc. of the IEEE* , vol. 75, num 9, pp 1283-1291, September 1987.

[50] C. Lu, *Fast Fourier Transform Algorithms for Special N's and the Implementations on Vax.*PhD thesis, E.E. Department. The City College of New York, of the City University of New York, 1988.

[51] H. Andrews " Computer Techniques in Image Processing," *Academic Press Publishing Company,* New York 1970.

[52] H. Hou, "The Fast Hartly Transform Algorithm," *IEEE Trans on Comp.* vol. C-36, no. 2, pp 147-156, February 1987.

[53] J. Johnson, R, Johnson, D. Rodriguez, R. Tolimieri." A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. " *IEEE Trans. on Circuits and Systems,* . In press.

[54] M. Narashimha and M. Peterson," On Computing the Discrete Cosine Transform," *IEEE Trans. Commun.* , vol. 26, pp 934-936, June 1978.

[55] H. Hou, "A Fast Recursive Algorithm for Computing the Discrete Cosine Transform," *IEEE ASSP* vol. 35, no. 10, pp 1455-1485, October 1987.

[56] S. Bogoch, I. Bason, J. Williams, and M. Russell, "Supercomputers Get Personal," *BYTE Magazine* pp 231-237, May 1990.

[57] *AT&T DSP Parallel Processor BT-100 User Manual* . AT&T 1988.

[58] *iPSC/860 Supercomputer Advanced Information Fact Sheet* . Intel 1990.

[59] P. Duhamel," Implementation of Split Radix FFT Algorithms for Complex, Real, and Real Symmetric Data," *IEEE ASSP* , vol. 34, num. 2, April 1986.