1993

Squared Law Algorithms: Theory and Applications.

Poornachandra Bellamkonda Rao
Louisiana State University and Agricultural & Mechanical College

Order Number 9405417

Squared law algorithms: Theory and applications

Rao, Poornachandra Bellamkonda, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1993

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

SQUARED LAW ALGORITHMS: THEORY AND APPLICATIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Poornachandra B. Rao
B. E., Osmania University, 1984
M.S. in E.E., Louisiana State University, 1989
August 1993

Acknowledgments

I would like to thank my major professor, Dr. Alexander Skavantzos, for his advice and guidance throughout this research effort.

I would also like to thank Drs. Ahmed El-Amawy, Bush Jones, Subhash Kak, Charles Monlezun, and Suresh Rai for willing to serve as members of my doctoral committee. In particular, I would like to thank Professor Subhash Kak who through many conversations not only provided encouragement but also helped me see the broader picture.

Most of all, I would like to thank my parents, sister, and brother whose patience, faith, and moral support made this dissertation possible.

Table of Contents

List of Tables

List of Figures

Abstract

This dissertation focuses on a new approach for a hardware implementation of the cyclic convolution operation. The cyclic convolution operation is the core of several functions used in applications related to digital signal processing and error control. Since the operation is multiplication intensive and the cost of a multiplication operation is very high, most of the present research effort attempts to reduce the number of multiplications.

Our approach, however, aims at obtaining an efficient implementation by relying on the properties of the special case of multiplication, namely, the squaring operation. Due to the properties exhibited by the squaring operation the hardware cost and time delay of a squarer unit is both cheaper and faster than that of a multiplication unit. This is true for both memory and non-memory based implementations.

In this dissertation we have developed all the necessary theory required to express the cyclic convolution of two n-point sequences, where n is a power of 2, in terms of the elementary arithmetic operations add, square, and subtract. Our algorithms require fewer squaring operations than multiplication operations required by a traditional implementation of the cyclic convolution operation, do not introduce any round-off errors, place no restriction on word length, and are valid when the number of points to be convolved is a power of two. We then clearly demonstrate that our algorithms are also more hardware efficient for both memory and non-memory based implementations. Further, schemes to multiply two numbers based on the cyclic convolution operation are presented. Finally, efficient ways of computing the squaring operation when arithmetic is performed in modular rings are developed.

Chapter 1

Introduction

Applications in the fields of digital signal processing (DSP) and error control are a few of the many interests of a hardware design engineer. Hardware design for these applications are challenging because of their high computational complexity. The main computational tasks in these applications are convolutions, Fourier transforms, and the inversion of Toeplitz systems of equations for spectral estimation [1]. A wealth of literature already exists in these areas [2]-[6], to name a few. Apart from these books there are several journals dedicated to research and development in these areas. Much of the material in these fields is centered around the discrete Fourier transform (DFT). The DFT has many powerful algebraic properties that are valid in numerous number systems. Researchers have exploited these properties by exploring several different alternatives for the field of operations [6]-[11]. An appropriate selection provides the designer with a number of tricks that can speed up algorithms and simplify hardware implementations, for instance, selecting a Galois field of the form $GF(2^n - 1)$ or $GF(2^n + 1)$ simplifies significantly the arithmetic processing. We observe that arithmetic performed modulo $(2^n - 1)$ is similar to one's complement arithmetic.

Much of the field of digital signal processing and error control coding is devoted to the task of removing noise by passing a known signal through a suitable filter [1]. The main computational problem involved in this is the convolution operation. The convolution operation is used in implementations of finite impulse response filters [12], infinite impulse response filters [13], auto and cross correlations [14], and polynomial multiplication and multiplication of very large integers [15],[16]. The large sized

problems in filtering are broken into smaller linear convolutions or cyclic convolutions using well known overlap techniques [17],[18]. This dissertation focuses on computing the convolution operation using a new approach that does not rely on any transforms. Instead, we focus on this operation from the computer arithmetic point of view by examining the applicability of other elementary functions in evaluating the convolution operation.

The rest of the introductory chapter is organized into two sections. The first is a brief overview of existing approaches for computing convolutions and the second is an introduction to our approach. The intent of the first section is to impress upon the reader, some of the difficulties and complexities associated with existing methods and further, to motivate the need for approaching the problem from a fundamentally different angle.

1.1 Overview of existing methods

Signals are typically generated whenever things vibrate, pump, pulse, or in any other way change with time [19]. While such signals or waveforms in real life are continuous in nature they can, for pragmatic purposes, only be represented with a finite amount of precision. Further, while the data may be a real or complex number, it can be temporarily rescaled by shifting the decimal point to the right and treating the number as an integer. This practice of treating data as integer sequences is common [1], [8] and does not in any way detract from the quality of the analysis. Based on the application the designer may choose an appropriate word length to prevent overflow after data manipulation. We assume without loss of generality that data, also referred to as points of input and output sequences, are integers. The linear and cyclic convolution operations are defined as computations on two sets of integers that yields a third set of integers. More precisely, the linear convolution is defined as

$$c_i = \sum_{k=0}^{n-1} a_{i-k} b_k \qquad \text{for } i = 0, 1, ..., n-1 \qquad (1.1)$$

and the cyclic convolution as

$$c_i = \sum_{k=0}^{n-1} a_{<i-k>_n} b_k \qquad \text{for } i = 0, 1, ..., n-1 \qquad (1.2)$$

where the a_i and b_i are the input sets of data and the c_i are the data of the convolved sequence. The notation $<x>_m$ denotes the operation x modulo m. The number of points in the sequence, 'n', is also known as the block length. We note that the above computation requires n^2 multiplications. Performing the arithmetic in the above two equations, (1.1) and (1.2), modulo p, where p is prime, changes the entire picture. This is because, now the computations are being carried out in the Galois field GF(p). This is very attractive due to the fact that the properties of the convolution theorem can be used [20],[21]. Before we discuss the usefulness of the convolution theorem one must note that if the choice of p in the above is such that the input data and the computed results are smaller than p then the modulo p operation is redundant.

The convolution theorem [1] enables the computation of the cyclic convolution of two vectors A and B by first computing the Fourier transforms of the vectors, then obtaining a new vector by performing a point by point multiplication of the transformed vectors, and then finally applying the inverse Fourier transform on the new vector, i.e. on the vector obtained in the transform domain. If we assume that there are n elements in each of the input vectors then it is easy to see from the above that only n multiplications are needed, the multiplications being in the transformed domain. Clearly the convolution theorem is useful if and only if their exist efficient ways of computing the Fourier and inverse Fourier transforms. The existence of such transforms is discussed in [20],[21].

The discrete Fourier transform can be applied on a discrete set of points. The DFT maps a discrete time domain waveform x(n) into a frequency domain X(k) and an inverse discrete time Fourier transform (IDFT) maps it back into the time domain. The transforms are symbolically represented as

$$x(t) \xrightarrow{F_{dt}} X(k) \qquad \text{and} \qquad X(k) \xrightarrow{F_{dt}^{-1}} x(t)$$

and defined by

$$X(k) = \sum_{t=0}^{n-1} x(t) e^{\frac{-j2\pi}{n} tk} \qquad k = 0, 1, ..., n-1 \qquad (1.3)$$

and

$$x(t) = \frac{1}{n} \sum_{k=0}^{n-1} X(k) e^{\frac{j2\pi}{n} tk} \qquad t = 0, 1, ..., n-1 \qquad (1.4)$$

From equations (1.3) and (1.4) it can be seen that to obtain c(t) which is the cyclic convolution of a(t) and b(t), a(t) and b(t) are mapped into the frequency domain, represented by parameters A(k) and B(k), using n multiplications for each point for a total of $2n^2$ multiplications and $2n^2$ additions. Multiplication of A(k) and B(k) requires another n multiplications while the inverse mapping requires another n^2 multiplications and n^2 additions. We thus have a total of $(3n^2 + n)$ multiplications and $3n^2$ additions. Thus, when the DFTs are computed directly this approach is not of much practical value as the direct computation of the cyclic convolution itself requires only n^2 multiplications.

However, Cooley and Tukey [6] introduced the fast Fourier transform (FFT) which is an efficient algorithm to compute the DFT. Other efficient FFT algorithms can be found in [22]-[24]. While these algorithms require only $n \log_2 n$ multiplications to map an input sequence of length n into the frequency domain, they have two primary disadvantages. One is that they produce significant round off errors [25],[26] and the other is that they are not very well suited for VLSI implementations [27]. Although the

FFT reduces the number of multiplications required for evaluating the DFT, the count of multiplications in itself does not determine the computational efficiency of the algorithm. With the widespread use of VLSI to build application specific integrated circuits (ASIC) the architectural details of implementation have gained significant importance. Reference [28] discusses Fourier transforms in VLSI, [29] discusses architectural issues in DSP applications, and [30] discusses multiplier policies in DSP applications. Many researchers have also explored the applicability of systolic array architectures to compute the DFT [31]-[33]. Some of the other irritants are block lengths and wordlengths [1]. While none of these are insurmountable they do require certain awkward design choices. The DFT can also be evaluated using number theoretic transforms (NTT) which are defined over finite fields and rings of integers with all the arithmetic performed modulo an integer. An evaluation of the various NTT algorithms as applied to digital filtering applications can be found in [34],[35].

The main focus of the various FFT and NTT algorithms [36] has been on reducing the number of multiplications. However, in real time applications large amounts of data have to be processed in relatively small periods of time and thus apart from reducing the number of computations, it has also become necessary to parallelize the computations. With the cost of hardware reducing and the acceptance of application specific integrated circuits (ASICs) increasing, it has become possible to build dedicated systems in an economical fashion. The inherent properties of the residue number system (RNS) lends itself as a viable candidate for parallel computations [37]. Implementations using the RNS and the quadratic residue number system (QRNS) can be found in [38]-[41]. More recently, the polynomial residue number system (PRNS) has been developed [42],[43] which combines both features, i.e. reducing the number of computations while simultaneously increasing the level of parallelism.

We have so far described the importance of the cyclic convolution operation in digital signal processing applications. In our research we propose to develop new algorithms for computing the cyclic convolution of two n-point sequences by performing all required operations in a single domain, i.e. we will not map the points into a frequency domain or for that matter into any other domain. To do this successfully we will have to use fewer than n^2 multiplications. To achieve this reduction in the multiplication count all existing research has focused on mapping strategies that result in reduced number of multiplications in the mapped domain. However, our research effort focuses on using squaring operations instead of multiplication operations. The next section provides an introduction to our approach.

1.2 Our approach

The vast amount of literature in the DSP area focuses on the convolution operation by exploiting the properties of the DFT and the algebraic field in which it is applied. Most of the algorithms therefore have specific properties and perform well in the environments that they were designed to function in. We have looked at the problem from a more broader perspective and our approach therefore does not rely on the DFT at all. Instead we have focused on the definition of cyclic convolution and have attempted to develop efficient algorithms centered around the elementary arithmetic function, the *squaring* operation, and hence the title of the dissertation "Squared Law Algorithms: Theory and Applications." Our motivation is based on the general underlying theme of the DFT and FFT algorithms, which has been primarily, to reduce the amount of hardware required to perform the convolution operation. Since the operation is multiplication intensive, the emphasis was first to reduce the count on the number of multiplications and then with the development of integrated circuit technology the emphasis was to improve the implementation architecture. Similarly, we attempt to

reduce the amount of hardware by zeroing in on the fact that the squaring operation requires lesser amount of hardware than the multiplication operation. One must also keep in mind that this reduction in hardware is not at the expense of speed, contrary to this it is also faster to evaluate the squaring as opposed to the multiplication operation.

Table look-up techniques for performing the multiplication operation using ROMs have been researched in [44]-[50]. Of these [46] is based on the index calculus technique which can only be used with prime moduli and [47]-[50] are based on the quarter squared algorithm technique. These designs offer attractive speed-complexity trade offs for small word lengths while for large word lengths the ROM size increases to the point where it becomes unrealistic.

Consider a simple ROM based direct implementation of the multiplication operation. The two input operands of length, say L bits each, serve as the address to the ROM. The data stored at this location is the result of the multiplication. Such a ROM implementation of the multiplication operation would require a ROM of size $2^{2L} \times 2L$. On the other hand the squaring operation would have only one input operand and a ROM implementing this would be of size $2^{L} \times 2L$. The immediate savings in ROM bits is apparent. The motivation is now clear. The more important question is now therefore: how does one replace all the multiplication operations in a given application with the squaring operations? We researched this problem with the convolution operation as our application and have developed algorithms to compute the convolution operation using squaring operations as opposed to multiplication operations. While we reduce the number of squaring operations compared to the number of multiplication operations we do increase the number of additions. Initial results of our research were published in [51]. The next natural question is: the squaring operation is a special case of the multiplication operation which in turn is repetitive additions, thus how does the increase

in the number of additions compare with the decrease in the number of the squarings. These two issues are addressed in detail in this dissertation.

The rest of the dissertation is organized as follows. Chapter 2 lays the mathematical foundations for the algorithms to compute the cyclic convolution using squaring operations. It also provides formulae for the count on the number of squares and the number of two-operand additions. Chapter 3 discusses various implementation issues including both non-ROM based and ROM based implementations. In this chapter the addition-squaring trade-off is also analyzed in detail. Some initial results on this were published in [52]. Since the focus of the research was on the usefulness of the squaring operation, the behavior with respect to hardware costs of the computation of this operation in modular rings was also studied. Initial results on this were published in brief in [53]. Details of these results are presented in chapter 4. Finally, chapter 5 concludes the research effort of this dissertation with some suggestions for future research.

Chapter 2

Mathematical Foundations

In this chapter we present the mathematical basis to prove the validity of our algorithms. The material in this chapter has a natural flow in the sense that it is presented in the order in which it was developed. This chapter also defines the extensive notations that are used throughout this dissertation.

2.1 Background

The starting point for this research has been reference [50], which described a novel approach for implementing convolutions with small tables. The algorithm developed in that paper, titled the one over eight squared algorithm applies the idea of the quarter squared algorithm to compute a two-point cyclic convolution. The method is briefly described.

The quarter squared algorithm technique [47]-[50] is based on the fact that the product of two n-bit numbers x and y can be given as

$$xy = 1/4\{(x + y)^2 - (x - y)^2\} \qquad (2.1)$$

Here look up tables can be used to compute the values of $(x + y)^2$ and $(x - y)^2$. If the result of the operation xy is computed directly by using a ROM then the size of the ROM required would be $2^{2n} \times 2n$, however, if (2.1) is used then two ROMs each of size $2^{n+1} \times 2(n+1)$ would be required. Thus the use of (2.1) yields a total ROM bit requirement of $2^{n+2} \times 2(n+1)$ bits. Clearly for $n > 2$ the use of (2.1) requires fewer ROM bits, however, there is an overhead in terms of adders. In general, it can be said that the use of the quarter squared technique reduces the ROM bits from the order of 2^{2n} to 2^n.

Now consider the problem of obtaining the cyclic convolution of two two-point sequences. The cyclic convolution of two sequences $A = \{a_0, a_1\}$ and $B = \{b_0, b_1\}$ is by definition given as $C = \{c_0, c_1\}$ where $c_0 = a_0b_0 + a_1b_1$ and $c_1 = a_0b_1 + a_1b_0$. Define[50]

$$u = a_0 + a_1 + b_0 + b_1 \tag{2.2}$$

$$v = -a_0 + a_1 - b_0 + b_1 \tag{2.3}$$

$$w = -a_0 - a_1 + b_0 + b_1 \tag{2.4}$$

$$x = -a_0 + a_1 + b_0 - b_1 \tag{2.5}$$

Then the two points of the cyclic convolution can be given as

$$c_0 = 1/8(u^2 + v^2 - w^2 - x^2) \tag{2.6}$$

$$c_1 = 1/8(u^2 + x^2 - v^2 - w^2) \tag{2.7}$$

Equations (2.6) and (2.7) constitute the one-over eight squared algorithm of [50] and they clearly demonstrate that the cyclic convolution of two two-point sequences can be obtained solely by the use of additions, subtractions, and squaring operations. A subtraction can be simply thought of as an addition as the hardware units that perform subtraction and addition are approximately equal in cost. Thus hereinafter the number of additions will include the number of subtractions. Also (2.6) and (2.7) show that the term w^2 always appears in the negative and hence the ROM that generates w^2 can be designed to directly generate $-w^2$.

The one over eight squared algorithm can also be applied in modular rings, provided the multiplicative inverse of 8 exists in the chosen ring. In the case when the chosen modulus m is odd $<8^{-1}>_m$ always exists, where $<x>_m$ is read as x modulo m. This can easily be shown as follows: when m is odd, m + 1 is even which implies that (m + 1)/2 is an integer. Therefore the multiplicative inverse of 2 modulus m can be given as $<2^{-1}>_m = (m + 1)/2$ as $2(m + 1)/2 = m + 1 = <1>_m$ [50]. Thus since $<2^{-1}>_m$

always exists, $<8^{-1}>_m$ also always exists as $<8^{-1}>_m = <(2^{-1})^3>_m$. Similarly, the multiplicative inverses of all numbers that are powers of 2 exist when m is odd. However, when m is even $<2^{-1}>_m$ does not exist. To see this let us assume that it did exist and its value is k. We then have $<2k>_m = 1$, which implies that $2k = mx + 1$ (x is some integer). But this is impossible as 2k and mx are even (m is even) and the difference of two even numbers can never be equal to 1. We thus have a problem and [50] provides some theorems to account for the round-off errors caused by this non-existence of $<2^{-1}>_m$ (m even).

2.2 Algorithm for convolution using an exponential number of squares

The first effort in generalizing the one over eight squared algorithm resulted in an algorithm for doing convolution using an exponential number of squares. Since the algorithm used an exponential number of squares, it is impractical from the view point of the cost of its hardware implementation. However, the insight gained from this algorithm was that it might be impossible to obtain in an efficient manner each point of the cyclic convolution directly as a function of a summation of squares. We next present the algorithm along with an example.

Algorithm 2.1

Input: The points of two n-point sequences $\{a_0, a_1, ..., a_{n-1}\}$ and $\{b_0, b_1, ..., b_{n-1}\}$.

Output: The cyclic convolution $\{c_0, c_1, ..., c_{n-1}\}$ of the two given input sequences.

Method: The procedure uses only addition and squaring operations.

Procedure: Each term of the cyclic convolution is given by

$$c_p = \frac{1}{2^{n+1}} \sum_{k=0}^{2^n-1} z_{pk}^2 (-1)^k, \qquad p = 0, 1, ..., n-1$$

where the z_{pk}'s are terms of the matrix Z_p. Matrix Z_p is of size $2^n \times 1$ and is formed as follows:

1) $Z_p = X_p \times Y$ where X is a $2^n \times 2n$ matrix whose terms are +1 or -1 and Y is a $2n \times 1$ transpose matrix of $\{a_0, a_1, ..., a_{n-1}, b_0, b_1, ..., b_{n-1}\}$.

2) The rows and columns of matrix X_p are represented by subscripts i and j respectively. Subscript i is in the range 0 to 2^n -1 and subscript j is in the range 1 to 2n.

3) The terms of matrix X_0 are defined by the following set of rules.

   a) $x_{ij} = 1$ if either i = 0 or j = 1.

   b) For $i > 0$ and $1 < j <= n+1$, let $r_j = 2^n/2^{j-1}$

   Then

$$x_{ij} = \begin{cases} 1 & \text{if } i<r_j \\ -x_{i-1\,j} & \text{if } \left[\sum_{i=i-r_j}^{i-1} x_{ij}\right] = \pm r_j \\ x_{i-1\,j} & \text{otherwise} \end{cases}$$

   c) For $i > 0$ and $n+1 < j <= 2n$, let $r_j = 2^{j-n-1}$

   Then

$$x_{ij} = \begin{cases} -x_{i-1\,j} & \text{if } i<r_j \\ x_{i-1\,j} & \text{if } i \text{ is an integer multiple of } r_j \\ -x_{i-1\,j} & \text{otherwise} \end{cases}$$

4) The matrices X_p for p = 1, 2, ..., n-1 are obtained from matrix X_0 by retaining its columns 1 through n as it is and by rotating right the columns n+1 through 2n by (p) positions.

Example: Suppose we wish to compute the cyclic convolution of two 3-point sequences $\{a_0, a_1, a_2\}$ and $\{b_0, b_1, b_2\}$.

From the above algorithm matrix Y is the transpose of $[a_0, a_1, a_2, b_0, b_1, b_2]$.

From step 3 we have

$$X_0 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & 1 & -1 \\ 1 & -1 & 1 & 1 & 1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 & -1 \\ 1 & -1 & -1 & -1 & 1 & 1 \end{bmatrix}$$

Multiplying the above matrix X_0 with matrix Y results in Z_0 which is a 8×1 matrix with terms $z_{00}, z_{01}, \ldots, z_{07}$.

$$Z_0 = \begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \\ z_{05} \\ z_{06} \\ z_{07} \end{bmatrix} = \begin{bmatrix} a_0 + a_1 + a_2 + b_0 + b_1 + b_2 \\ a_0 + a_1 + a_2 - b_0 - b_1 - b_2 \\ a_0 + a_1 - a_2 + b_0 - b_1 + b_2 \\ a_0 + a_1 - a_2 - b_0 + b_1 - b_2 \\ a_0 - a_1 + a_2 + b_0 + b_1 - b_2 \\ a_0 - a_1 + a_2 - b_0 - b_1 + b_2 \\ a_0 - a_1 - a_2 + b_0 - b_1 - b_2 \\ a_0 - a_1 - a_2 - b_0 + b_1 + b_2 \end{bmatrix}$$

Thus

$$c_0 = \frac{1}{16}\left[z_{00}^2 - z_{01}^2 + z_{02}^2 - z_{03}^2 + z_{04}^2 - z_{05}^2 + z_{06}^2 - z_{07}^2\right]$$
$$= a_0 b_0 + a_2 b_1 + a_1 b_2$$

Similarly we have,

$$X_1=\begin{bmatrix}1 & 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & -1 & -1 & -1\\ 1 & 1 & -1 & 1 & 1 & -1\\ 1 & 1 & -1 & -1 & -1 & 1\\ 1 & -1 & 1 & -1 & 1 & 1\\ 1 & -1 & 1 & 1 & -1 & -1\\ 1 & -1 & -1 & -1 & 1 & -1\\ 1 & -1 & -1 & 1 & -1 & 1\end{bmatrix}$$

Multiplying matrix $Z_1$with matrix B gives us Z_1 which is a 8×1 matrix whose terms are $z_{10}, z_{11}, ..., z_{17}$.

$$Z_1 = \begin{bmatrix}z_{10}\\ z_{11}\\ z_{12}\\ z_{13}\\ z_{14}\\ z_{15}\\ z_{16}\\ z_{17}\end{bmatrix} = \begin{bmatrix}a_0+a_1+a_2+b_0+b_1+b_2\\ a_0+a_1+a_2-b_0-b_1-b_2\\ a_0+a_1-a_2+b_0+b_1-b_2\\ a_0+a_1-a_2-b_0-b_1+b_2\\ a_0-a_1+a_2-b_0+b_1+b_2\\ a_0-a_1+a_2+b_0-b_1-b_2\\ a_0-a_1-a_2-b_0+b_1-b_2\\ a_0-a_1-a_2+b_0-b_1+b_2\end{bmatrix}$$

Thus giving

$$\begin{aligned}c_1 &= \frac{1}{16}\left[z_{10}^2 - z_{11}^2 + z_{12}^2 - z_{13}^2 + z_{14}^2 - z_{15}^2 + z_{16}^2 - z_{17}^2\right]\\ &= a_1b_0 + a_0b_1 + a_2b_2\end{aligned}$$

and finally,

$$X_2=\begin{bmatrix}1 & 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & -1 & -1 & -1\\ 1 & 1 & -1 & -1 & 1 & 1\\ 1 & 1 & -1 & 1 & -1 & -1\\ 1 & -1 & 1 & 1 & -1 & 1\\ 1 & -1 & 1 & -1 & 1 & -1\\ 1 & -1 & -1 & -1 & -1 & 1\\ 1 & -1 & -1 & 1 & 1 & -1\end{bmatrix}$$

Multiplying matrix X_2 with matrix Y gives us Z_2 which is a 8×1 matrix whose terms are $z_{20}, z_{21}, \ldots, z_{27}$.

$$Z_2 = \begin{bmatrix} z_{20} \\ z_{21} \\ z_{22} \\ z_{23} \\ z_{24} \\ z_{25} \\ z_{26} \\ z_{27} \end{bmatrix} = \begin{bmatrix} a_0 + a_1 + a_2 + b_0 + b_1 + b_2 \\ a_0 + a_1 + a_2 - b_0 - b_1 - b_2 \\ a_0 + a_1 - a_2 - b_0 + b_1 + b_2 \\ a_0 + a_1 - a_2 + b_0 - b_1 - b_2 \\ a_0 - a_1 + a_2 + b_0 - b_1 + b_2 \\ a_0 - a_1 + a_2 - b_0 + b_1 - b_2 \\ a_0 - a_1 - a_2 - b_0 - b_1 + b_2 \\ a_0 - a_1 - a_2 + b_0 + b_1 - b_2 \end{bmatrix}$$

Thus giving

$$\begin{aligned} c_2 &= \frac{1}{16}\left[z_{20}^2 - z_{21}^2 + z_{22}^2 - z_{23}^2 + z_{24}^2 - z_{25}^2 + z_{26}^2 - z_{27}^2\right] \\ &= a_2 b_0 + a_1 b_1 + a_0 b_2 \end{aligned}$$

It appears that we are multiplying two matrices for each point. However, this is not the case as the matrix notation is only a convenient form to represent the several equations that are developed for each point. Also there is no actual division involved as the last four bits are zero and by simply ignoring them we achieve the division by 16.

Although this algorithm is well structured from the implementation point of view, it relies on squaring operations in the order of $n2^n$, plus additions and subtractions, and thus the cost is prohibitive. Clearly this is far greater than even the n^2 multiplications required by the definition of the problem and thus no further work was done on this algorithm. However this motivated us to look in other directions and our results are presented in the next section.

2.3 Direct extension of the one over eight squared algorithm

We now try to extend the one over eight squared algorithm to obtain the cyclic convolution of two 4-point sequences. We first re-write the equations of section 2.1 in a more nicer form as follows. Equations (2.2) - (2.5) can be re-written as

$$u = a_0 + a_1 + b_0 + b_1 \quad (2.8)$$

$$v = a_0 + a_1 - b_0 - b_1 \quad (2.9)$$

$$w = a_0 - a_1 + b_0 - b_1 \quad (2.10)$$

$$x = a_0 - a_1 - b_0 + b_1 \quad (2.11)$$

Then the two points of the cyclic convolution can be given as

$$c_0 = 1/8(u^2 - v^2 + w^2 - x^2) \quad (2.12)$$

$$c_1 = 1/8(u^2 - v^2 - w^2 + x^2) \quad (2.13)$$

Now, our objective is to extend this method to obtain the cyclic convolution of two 4-point sequences. Let the sequences be A = $\{a_0, a_1, a_2, a_3\}$ and B = $\{b_0, b_1, b_2, b_3\}$ and by definition, the cyclic convolution of the two sequences is given as C = $\{c_0, c_1, c_2, c_3\}$ where

$$c_0 = a_0b_0 + a_3b_1 + a_2b_2 + a_1b_3 \quad (2.14)$$

$$c_1 = a_1b_0 + a_0b_1 + a_3b_2 + a_2b_3 \quad (2.15)$$

$$c_2 = a_2b_0 + a_1b_1 + a_0b_2 + a_3b_3 \quad (2.16)$$

$$c_3 = a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \quad (2.17)$$

Now, re-defining equations u through x based on the pattern of terms and signs in equations (2.8) - (2.11), we have

$$u = a_0 + a_1 + a_2 + a_3 + b_0 + b_1 + b_2 + b_3 \quad (2.18)$$

$$v = a_0 + a_1 + a_2 + a_3 - b_0 - b_1 - b_2 - b_3 \quad (2.19)$$

$$w = a_0 - a_1 + a_2 - a_3 + b_0 - b_1 + b_2 - b_3 \quad (2.20)$$

$$x = a_0 - a_1 + a_2 - a_3 - b_0 + b_1 - b_2 + b_3 \quad (2.21)$$

Then we find that $1/8(u^2 - v^2 + w^2 - x^2) = c_0 + c_2$ and $1/8(u^2 - v^2 - w^2 + x^2) = c_1 + c_3$. This gives us the notion that for any two n-point sequences if equations are built on lines similar to that of equations (2.8) - (2.11) and then plugged into equations of type (2.12) and (2.13) we can obtain $8 \sum c_{2i}$ and $8 \sum c_{2i+1}$. In the next section we reinforce this notion by providing four theorems on the summation of the even c_i and odd c_i taken separately. Going back to the cyclic convolution of the two 4-point sequences we observe that if we have the quantities $c_0 - c_2$ and $c_1 - c_3$ then they can be added, subtracted with $c_0 + c_2$ and $c_1 + c_3$ to obtain the individual points of the cyclic convolution. This gives us the indication that we have to have formulae like equations (2.8) - (2.11) that will generate the difference of c_0 and c_2 , and c_1 and c_3. In general, we would need formulae to generate the sum of the even c_i with alternating terms having negative signs and similarly the sum of the odd c_i with the alternating terms having negative signs. In the next section we provide two theorems that can be applied in general to n-point sequences.

It is easy to see that if this method is extended further, say for two 8-point sequences then the above promised extensions will yield $\sum c_{2i}$ and $\sum c_{2i}(-1)^i$, however, this would be inadequate. This is because the sum of $\sum c_{2i}$ and $\sum c_{2i}(-1)^i$ would give us $c_0 + c_4$ and the difference $c_2 + c_6$. (Similarly, for the odd points we would obtain $c_1 + c_5$ and $c_3 + c_7$.) Thus we would need to obtain $c_0 - c_4$, $c_2 - c_6$, $c_1 - c_5$, and $c_3 - c_7$. In other words we would need to obtain the alternating sum and difference of c_i where the difference of the indices of two consecutive c_i differ by 2, 4, ..., n/2. In later discussions we let j denote this difference between consecutive c_i. Also, we observe that

such a methodology of adding and subtracting different combinations of c_i will be valid only when n is a power of 2.

We divide the methodology of computing the cyclic convolution of two n-point sequences based on squaring and addition operations into two parts: part 1 comprises the computation of Σc_{2i} and Σc_{2i+1} while part 2 that of $\Sigma c_{i+kj} (-1)^k$ for all $0 <= i, j < n/2, k = 0, 1, 2, ..., (n/j -1)$. (We note that j is a power of 2, >= 2). In section 2.4 we present eight theorems and proofs that are required for our complete methodology of computing the cyclic convolution of two n-point sequences. Following that, in section 2.5 we compare and contrast three methods by which we can compute part 1 of the methodology. In section 2.6 we present a formula for the number of squares required for any given n while in section 2.7 we present a formula for the total number of additions required by our methodology. We conclude the chapter by presenting a full blown example of computing the cyclic convolution of two 16-point sequences.

2.4 Squared law theorems for cyclic convolutions

In section 2.2 we have shown that trying to obtain each point of the cyclic convolution directly as a function of a summation of squares required a total of $n2^n$ squaring operations and thus was not of any practical use. In this section we present eight theorems that we have developed for computing the sum and difference of the even points taken together and the odd points taken together. Theorems 2.1 and 2.2 are taken from our paper titled "New Multipliers Modulo 2^N -1"[51]. They are presented here for both, the sake of completeness and consistent notation.

Consider two sequences each of length n-points given as $A = \{a_0, a_1, ... , a_{n-1}\}$ and $B = \{b_0, b_1, ... , b_{n-1}\}$. Then the cyclic convolution between these two sequences can be given as a n-point sequence $C = \{c_0, c_1, ... , c_{n-1}\}$ with each c_i defined by

$$c_i = \sum_{k=0}^{n-1} a_{<i-k>_n} b_k \qquad \text{for } i = 0, 1, ..., n-1 \qquad (2.22)$$

In the above and in the rest of the dissertation $<x>_m$ denotes the operation x modulo m between integers. With reference to the above computation the following theorems apply.

Theorem 2.1: Assume that n is even and define

$$w_{n1} = a_0 + a_1 + ... + a_{n-1} + b_0 + b_1 + ... + b_{n-1} \qquad (2.23)$$

$$w_{n2} = a_0 + a_1 + ... + a_{n-1} - b_0 - b_1 - ... - b_{n-1} \qquad (2.24)$$

$$w_{n3} = a_0 - a_1 + ... - a_{n-1} + b_0 - b_1 + ... - b_{n-1} \qquad (2.25)$$

$$w_{n4} = a_0 - a_1 + ... - a_{n-1} - b_0 + b_1 - ... + b_{n-1} \qquad (2.26)$$

Then

$$w_{n1}^2 - w_{n2}^2 + w_{n3}^2 - w_{n4}^2 = 8\sum_{i=0}^{\frac{n}{2}-1} c_{2i} = 8(c_0 + c_2 + ... + c_{n-2}) \qquad (2.27)$$

Proof:

$$w_{n1}^2 - w_{n2}^2 + w_{n3}^2 - w_{n4}^2 = (w_{n1} + w_{n2})(w_{n1} - w_{n2}) + (w_{n3} + w_{n4})(w_{n3} - w_{n4})$$

$$= 4\sum_{i=0}^{n-1} a_{2i} \sum_{i=0}^{n-1} b_{2i} + 4(\sum_{i=0}^{\frac{n}{2}-1} a_{2i} - \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1}) \times (\sum_{i=0}^{\frac{n}{2}-1} b_{2i} - \sum_{i=0}^{\frac{n}{2}-1} b_{2i+1})$$

$$= 4[(\sum_{i=0}^{\frac{n}{2}-1} a_{2i} + \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1})(\sum_{i=0}^{\frac{n}{2}-1} b_{2i} + \sum_{i=0}^{\frac{n}{2}-1} b_{2i+1}) + (\sum_{i=0}^{\frac{n}{2}-1} a_{2i} - \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1})(\sum_{i=0}^{\frac{n}{2}-1} b_{2i} - \sum_{i=0}^{\frac{n}{2}-1} b_{2i+1})]$$

$$= 8\sum_{i=0}^{\frac{n}{2}-1} a_{2i} \sum_{i=0}^{\frac{n}{2}-1} b_{2i} + 8\sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} \sum_{i=0}^{\frac{n}{2}-1} b_{2i+1}$$

$$= 8[\sum_{\text{all possible products}} a_{even} b_{even} + \sum_{\text{all possible products}} a_{odd} b_{odd}]$$

$$= 8 \sum_{i=0}^{\frac{n}{2}-1} c_{2i}$$

and the proof of (2.27) is completed.

Note from equation (2.22) that $c_{2i} = \sum a_v b_w$ such that $<v+w>_n = 2i$ or $v+w = nx + 2i$. But since n=even it implies that v+w = even and therefore v and w are either both even or both odd. This justifies the last step of the proof.

Theorem 2.2: Assume that n is even and define w_{n1}, w_{n2}, w_{n3}, and w_{n4} as given in theorem 2.1.

Then

$$w_{n1}^2 - w_{n2}^2 - w_{n3}^2 + w_{n4}^2 = 8 \sum_{i=0}^{\frac{n}{2}-1} c_{2i+1} = 8(c_1 + c_3 + ... + c_{n-1}) \qquad (2.28)$$

Proof: The proof is similar to that of (2.27) and is thus omitted.

Theorem 2.3: Assume that n is even and define

$$x_{n1} = a_0 + a_2 + ... + a_{n-2} + b_0 + b_2 + ... + b_{n-2} \qquad (2.29)$$

$$x_{n2} = a_0 + a_2 + ... + a_{n-2} - b_0 - b_2 - ... - b_{n-2} \qquad (2.30)$$

$$x_{n3} = a_1 + a_3 + ... + a_{n-1} + b_1 + b_3 + ... + b_{n-1} \qquad (2.31)$$

$$x_{n4} = a_1 + a_3 + ... + a_{n-1} - b_1 - b_3 - ... - b_{n-1} \qquad (2.32)$$

Then

$$x_{n1}^2 - x_{n2}^2 + x_{n3}^2 - x_{n4}^2 = 4 \sum_{i=0}^{\frac{n}{2}-1} c_{2i} = 4(c_0 + c_2 + ... + c_{n-2}) \qquad (2.33)$$

Proof:

$$x_{n1}^2 - x_{n2}^2 + x_{n3}^2 - x_{n4}^2 = (x_{n1} + x_{n2})(x_{n1} - x_{n2}) + (x_{n3} + x_{n4})(x_{n3} - x_{n4})$$

$$= 4\sum_{i=0}^{\frac{n}{2}-1} a_{2i} \sum_{i=0}^{\frac{n}{2}-1} b_{2i} + 4\sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} \sum_{i=0}^{\frac{n}{2}-1} b_{2i+1}$$

$$= 4[\sum_{\text{all possible products}} a_{even} b_{even} + \sum_{\text{all possible products}} a_{odd} b_{odd}]$$

$$= 4\sum_{i=0}^{\frac{n}{2}-1} c_{2i}$$

and the proof of (2.33) is completed.

Note again from the definition of cyclic convolution that $c_{2i} = \sum a_v b_w$ such that $<v+w>_n = 2i$ or $v+w = nx + 2i$. But since n=even it implies that v+w = even and therefore v and w are either both even or both odd. This justifies the last step of the proof.

Theorem 2.4: Assume that n is even and define

$$y_{n1} = a_0 + a_2 + \ldots + a_{n-2} + b_1 + b_3 + \ldots + b_{n-1} \quad (2.34)$$

$$y_{n2} = a_0 + a_2 + \ldots + a_{n-2} - b_1 - b_3 - \ldots - b_{n-1} \quad (2.35)$$

$$y_{n3} = a_1 + a_3 + \ldots + a_{n-1} + b_0 + b_2 + \ldots + b_{n-2} \quad (2.36)$$

$$y_{n4} = a_1 + a_3 + \ldots + a_{n-1} - b_0 - b_2 - \ldots - b_{n-2} \quad (2.37)$$

Then

$$y_{n1}^2 - y_{n2}^2 + y_{n3}^2 - y_{n4}^2 = 4\sum_{i=0}^{\frac{n}{2}-1} c_{2i+1} = 4(c_1 + c_3 + \ldots + c_{n-1}) \quad (2.38)$$

Proof: The proof is similar to that of (2.33) and is thus omitted.

Theorem 2.1 and 2.3 both compute c_{2i}, however theorem 2.1 is more useful for computing the cyclic convolution in modular rings while theorem 2.3 is more hardware

efficient for computing the full precision cyclic convolution. The same can be said of theorems 2.2 and 2.4. The advantages are described in detail in section 2.5.

Theorem 2.5: Assume that $n > 2$, $n = 2^p$ (=> n=4m), and define

$$z_{n1} = a_0 - a_2 + a_4 - a_6 + \ldots - a_{n-2} + b_0 - b_2 + b_4 - b_6 + \ldots - b_{n-2} \quad (2.39)$$

$$z_{n2} = a_0 - a_2 + a_4 - a_6 + \ldots - a_{n-2} - b_0 + b_2 - b_4 + b_6 - \ldots + b_{n-2} \quad (2.40)$$

$$z_{n3} = a_1 - a_3 + a_5 - a_7 + \ldots - a_{n-1} + b_1 - b_3 + b_5 - b_7 + \ldots - b_{n-1} \quad (2.41)$$

$$z_{n4} = a_1 - a_3 + a_5 - a_7 + \ldots - a_{n-1} - b_1 + b_3 - b_5 + b_7 - \ldots + b_{n-1} \quad (2.42)$$

Then

$$z_{n1}^2 - z_{n2}^2 - z_{n3}^2 + z_{n4}^2 = 4(c_0 - c_2 + c_4 - c_6 + \ldots - c_{n-2}) \quad (2.43)$$

Proof:

$$\begin{aligned}
z_{n1}^2 - z_{n2}^2 - z_{n3}^2 + z_{n4}^2 &= (z_{n1} + z_{n2})(z_{n1} - z_{n2}) - (z_{n3} + z_{n4})(z_{n3} - z_{n4}) \\
&= 4[(a_0 - \ldots - a_{n-2})(b_0 - \ldots - b_{n-2}) - (a_1 - \ldots - a_{n-1})(b_1 - \ldots - b_{n-1})] \\
&= 4[(a_0 + a_4 + a_8 + \ldots + a_{n-4}) - (a_2 + a_6 + a_{10} + \ldots + a_{n-2})]x \\
&\quad [(b_0 + b_4 + b_8 + \ldots + b_{n-4}) - (b_2 + b_6 + b_{10} + \ldots + b_{n-2})] \\
&\quad -4[(a_1 + a_5 + a_9 + \ldots + a_{n-3}) - (a_3 + a_7 + a_{11} + \ldots + a_{n-1})]x \\
&\quad [(b_1 + b_5 + b_9 + \ldots + b_{n-3}) - (b_3 + b_7 + b_{11} + \ldots + b_{n-1})] \\
&= 4[(\sum_{k=0}^{m-1} a_{4k})(\sum_{\lambda=0}^{m-1} b_{4\lambda}) + (\sum_{k=0}^{m-1} a_{4k+2})(\sum_{\lambda=0}^{m-1} b_{4\lambda+2})] \\
&\quad +4[(\sum_{k=0}^{m-1} a_{4k+1})(\sum_{\lambda=0}^{m-1} b_{4\lambda+3}) + (\sum_{k=0}^{m-1} a_{4k+3})(\sum_{\lambda=0}^{m-1} b_{4\lambda+1})] \\
&\quad -4[(\sum_{k=0}^{m-1} a_{4k})(\sum_{\lambda=0}^{m-1} b_{4\lambda+2}) + (\sum_{k=0}^{m-1} a_{4k+2})(\sum_{\lambda=0}^{m-1} b_{4\lambda})] \\
&\quad -4[(\sum_{k=0}^{m-1} a_{4k+1})(\sum_{\lambda=0}^{m-1} b_{4\lambda+1}) + (\sum_{k=0}^{m-1} a_{4k+3})(\sum_{\lambda=0}^{m-1} b_{4\lambda+3})] \\
&= 4(c_0 - c_2 + c_4 - c_6 + \ldots - c_{n-2})
\end{aligned}$$

and the proof of (2.43) is completed.

Note that for the positive terms in the above expression the sum of the indices modulo n of the a terms and b terms is always a multiple of four, thus the sum of all positive terms will be $4(c_0 + c_4 + c_8 +...+ c_{n-4})$. Similarly for the negative terms, the sum of the indices is a ((multiple of 4) + 2), thus the sum of all negative terms will be $4(c_2 + c_6 + c_{10} +...+ c_{n-2})$. This justifies the last step of the proof. Also observe that the theorem holds good for any n that is a multiple of 4, i.e. not necessarily a power of 2. However, since our main algorithm that describes our methodology requires n to be a power of 2, we focus only on such cases.

Theorem 2.6: Assume that $n > 2$, $n = 2^p$ (=> n=4m), and define

$$z_{n5} = a_0 - a_2 + a_4 - a_6 + ... - a_{n-2} + b_1 - b_3 + b_5 - b_7 + ... - b_{n-1} \quad (2.44)$$

$$z_{n6} = a_0 - a_2 + a_4 - a_6 + ... - a_{n-2} - b_1 + b_3 - b_5 + b_7 - ... + b_{n-1} \quad (2.45)$$

$$z_{n7} = a_1 - a_3 + a_5 - a_7 + ... - a_{n-1} + b_0 - b_2 + b_4 - b_6 + ... - b_{n-2} \quad (2.46)$$

$$z_{n8} = a_1 - a_3 + a_5 - a_7 + ... - a_{n-1} - b_0 + b_2 - b_4 + b_6 - ... + b_{n-2} \quad (2.47)$$

Then

$$z_{n5}^2 - z_{n6}^2 + z_{n7}^2 - z_{n8}^2 = 4(c_1 - c_3 + c_5 - c_7 + ... - c_{n-1}) \quad (2.48)$$

Proof: The proof is similar to that of (2.43) and is thus omitted.

Theorem 2.7: Assume that n is even and define

$$w_{n5} = a_0 - a_1 + a_2 - a_3 + ... - a_{n-1} + b_0 - b_1 + b_2 - b_3 + ... + b_{n-1} \quad (2.49)$$

$$w_{n6} = -a_0 + a_1 - a_2 + a_3 + ... + a_{n-1} + b_0 - b_1 + b_2 - b_3 + ... - b_{n-1} \quad (2.50)$$

Then

$$w_{n5}^2 - w_{n6}^2 = 4\sum_{i=0}^{n-1} c_i(-1)^i \quad (2.51)$$

Proof:

$$
\begin{aligned}
w_{n5}^2 - w_{n6}^2 &= (w_{n5} + w_{n6})(w_{n5} - w_{n6}) \\
&= 2(\sum_{i=0}^{n-1} b_{2i} - \sum_{i=0}^{n-1} b_{2i+1}) 2(\sum_{i=0}^{n-1} a_{2i} - \sum_{i=0}^{n-1} a_{2i+1}) \\
&= 4[(\sum_{i=0}^{n-1} a_{2i} \sum_{i=0}^{n-1} b_{2i} + \sum_{i=0}^{n-1} a_{2i+1} \sum_{i=0}^{n-1} b_{2i+1}) \\
&\quad -(\sum_{i=0}^{n-1} a_{2i} \sum_{i=0}^{n-1} b_{2i+1} + \sum_{i=0}^{n-1} a_{2i+1} \sum_{i=0}^{n-1} b_{2i})] \\
&= 4[(\sum_{\text{all possible products}} a_{even} b_{even} + \sum_{\text{all possible products}} a_{odd} b_{odd}) \\
&\quad -(\sum_{\text{all possible products}} a_{even} b_{odd} + \sum_{\text{all possible products}} a_{odd} b_{even})] \\
&= 4\{\sum_{i=0}^{n-1} c_{2i} - \sum_{i=0}^{n-1} c_{2i+1}\} \\
&= 4\sum_{i=0}^{n-1} c_i (-1)^i
\end{aligned}
$$

and the proof of (2.51) is completed.

Theorem 2.5 generates the sum (or difference) of the even points of the cyclic convolution with the alternating points having negative signs. We need to generalize this theorem so that it is possible for us to automate the generation of the sum (or difference) of the even points where the indices of the even points differ by powers of two. i.e. we need a generalized theorem to generate (c_0 - c_2 + c_4 - ... c_{n-2}), (c_0 - c_4 + c_8 - ... c_{n-4}) and (c_2 - c_6 + c_{10} - ... c_{n-2}), (c_0 - c_8 + c_{16} - ... c_{n-8}), (c_2 - c_{10} + c_{18} - ... c_{n-6}), (c_4 - c_{12} + c_{20} - ... c_{n-4}), and (c_6 - c_{14} + c_{22} - ... c_{n-2}), and so on. We would also need a generalized theorem corresponding to theorem 2.6, i.e. for the automatic generation of the odd points. We first present an algorithm to generate the equations that the generalized theorem will use. We then present the generalized theorem and its associated proof. We show that the same theorem can also be used for odd points.

Algorithm 2.2

Input: The points of two n-point sequences $\{a_0, a_1, ..., a_{n-1}\}$ and $\{b_0, b_1, ..., b_{n-1}\}$ and j, where j is the difference of the indices of two consecutive c_i with $j = 2*2^k$; $k = 0, 1, ... (\log_2 (n/2) - 1)$.

Output: $4 \sum_{k=0}^{n/j\ -1} c_{i+kj}\ (-1)^k$

Procedure:

Step 1: Initialization step.

$k = 1, q = 0, p = i.$

(Obtain equations of type V^i_{nk})

Step 2: $V^i_{nk} = a_p - a_{<p + j>_n} + a_{<p + 2j>_n} \cdots - a_{<p - j>_n} +$

$b_q - b_{<q + j>_n} + b_{<p + 2j>_n} \cdots - b_{<q - j>_n}$;

Obtain $V^i_{n(k+1)}$ by reversing signs of b terms in V^i_{nk};

$k \leftarrow k + 2;$

$p \leftarrow <p - 1>_n$;

$q \leftarrow q + 1;$

If $k <= 2j - 1$, go to step 2.

(Note that $<p + q>_n = i$)

Step 3: $\sum_{k=1}^{2j} (V^i_{nk})^2\ (-1)^{k+1} = 4 \sum_{k=0}^{n/j\ -1} c_{i+kj}\ (-1)^k$

Step 4: $i \leftarrow i + 2;$

If $i <= j - 1$, then $k = 1; q = 0; p = i;$ and go to step 2

else STOP.

Theorem 2.8: Assume that n is greater than 2, a power of 2, and for a given value of i, j, and n define V^i_{nk} based on steps 1 and 2 of algorithm 2.2.

$$\text{Then} \quad \sum_{k=1}^{2j} (V^i_{nk})^2 \ (-1)^{k+1} = 4 \sum_{k=0}^{n/j\,-1} c_{i+kj} \ (-1)^k \tag{2.52}$$

Proof:

$$\sum_{k=1}^{2j} (V^i_{nk})^2 \ (-1)^{k+1}$$

$$= \sum (V^i_{nk} + V^i_{n(k+1)})(V^i_{nk} - V^i_{n(k+1)}) \qquad \forall \qquad k \text{ odd and} < 2j$$

$$= 4(a_{<p>_n} - a_{<p+j>_n} \cdots - a_{<p-j>_n})(b_{<q>_n} - b_{<q+j>_n} \cdots - b_{<q-j>_n}) +$$

$$4(a_{<p-1>_n} - a_{<p-1+j>_n} \cdots - a_{<p-1-j>_n})$$

$$(b_{<q+1>_n} - b_{<q+1+j>_n} \cdots - b_{<q+1-j>_n}) +$$

$$\ldots +$$

$$\ldots +$$

$$4(a_{<p-j+1>_n} - a_{<p+1>_n} + \cdots - a_{<p+1-2j>_n})$$

$$(b_{<q+j-1>_n} - b_{<q+2j-1>_n} + \cdots - b_{<q-1>_n})$$

$$= 4\Big(\sum_{l=0}^{n/2j\,-1} a_{p+2lj} - \sum_{l=0}^{n/2j\,-1} a_{p+(2l+1)j}\Big)\Big(\sum_{l=0}^{n/2j\,-1} b_{q+2lj} - \sum_{l=0}^{n/2j\,-1} b_{q+(2l+1)j}\Big) +$$

$$\ldots +$$

$$\ldots +$$

$$4\Big(\sum_{l=0}^{n/2j\,-1} a_{p-j+1+2lj} - \sum_{l=0}^{n/2j\,-1} a_{p-j+1+(2l+1)j}\Big)\Big(\sum_{l=0}^{n/2j\,-1} b_{q+j-1+2lj} - \sum_{l=0}^{n/2j\,-1} b_{q+j-1+(2l+1)j}\Big)$$

$$= 4\Big(\sum_{l=0}^{n/2j-1} a_{p+2lj} \sum_{l=0}^{n/2j-1} b_{q+2lj} + \sum_{l=0}^{n/2j-1} a_{p+(2l+1)j} \sum_{l=0}^{n/2j-1} b_{q+(2l+1)j}\Big)$$

$$+ \cdots$$

$$+ \cdots$$

$$+ 4\Big(\sum_{l=0}^{n/2j-1} a_{p-j+1+2lj} \sum_{l=0}^{n/2j-1} b_{q+j-1+2lj} + \sum_{l=0}^{n/2j-1} a_{p-j+1+(2l+1)j} \sum_{l=0}^{n/2j-1} b_{q+j-1+(2l+1)j}\Big)$$

$$- 4\Big(\sum_{l=0}^{n/2j-1} a_{p+(2l+1)j} \sum_{l=0}^{n/2j-1} b_{q+2lj} + \sum_{l=0}^{n/2j-1} a_{p+2lj} \sum_{l=0}^{n/2j-1} b_{q+(2l+1)j}\Big)$$

$$- \cdots$$

$$- \cdots$$

$$- 4\Big(\sum_{l=0}^{n/2j-1} a_{p-j+1+(2l+1)j} \sum_{l=0}^{n/2j-1} b_{q+j-1+2lj} + \sum_{l=0}^{n/2j-1} a_{p-j+1+2lj} \sum_{l=0}^{n/2j-1} b_{q+j-1+(2l+1)j}\Big)$$

$$= 4\Big(\sum_{l=0}^{n/2j-1} c_{i+2lj} - \sum_{l=0}^{n/2j-1} c_{i+(2l+1)j}\Big)$$

$$= 4 \sum_{k=0}^{n/j-1} c_{i+kj} (-1)^k$$

and the proof of (2.52) is completed.

Notice, from the last but third step, that in all the positive terms the sum of the indices of a and b is of the form p+q+2lj and since $\langle p+q \rangle_n = i$ (from step 2 of algorithm 2.2), the sum of all positive terms reduces to $4\Big(\sum_{l=0}^{n/2j-1} c_{i+2lj}\Big)$. A similar argument can be made for the negative terms. Also, note that in the initialization step of algorithm 2.2, i takes a value passed by theorem 2.8 as opposed to being equal to zero. Other parts of the initialization step remain unchanged. Thus the theorem can be used for odd and even points alike.

Algorithm 2.3

Input: The points of two n-point sequences $\{a_0, a_1, ..., a_{n-1}\}$ and $\{b_0, b_1, ..., b_{n-1}\}$.

Output: The cyclic convolution $\{c_0, c_1, ..., c_{n-1}\}$ of the two given input sequences.

Method: The procedure uses only addition and squaring operations.

Procedure:

Step 1: $r = 1, i = 0$.

Do theorem 2.3. Set result to X^i.

We thus obtain $X^i = 4 \sum_{k=0}^{n/2-1} c_{2k}$

Step 2: $j = 2^r$.

Do theorem 2.8. Set result to Z.

We thus obtain $Z = 4 \sum_{k=0}^{n/j-1} c_{i+kj} (-1)^k$

Step 3: $X^i \leftarrow\!\!- X^i + Z\,2^{r-1}$; $X^{i+j} \leftarrow\!\!- X^i - Z\,2^{r-1}$

If $j = n/2$, then set $c_i = X^i / (4*2^r)$, $c_{i+j} = X^{i+j} / (4*2^r)$, and go to step 5.

Step 4: If $j <= n/2$, then $r \leftarrow\!\!- r + 1$ and go to step 2.

Step 5: $i \leftarrow\!\!- i + 2$; If $i <= (n/2 - 1)$, then $r = \lfloor \log_2 i \rfloor + 1$ and go to step 2

else STOP. (All c_{2i} have been computed).

In order to compute c_{2i+1} algorithm 2.3 with a few changes can be employed. The changes are in step 1. Step 1 will read,

Step 1(odd c_i): $r = 1, i = 1$.

Do theorem 2.4. Set result to X^i.

We thus obtain $X^i = \sum_{k=0}^{n/2-1} c_{2k+1}$

Steps 2 through 5 will remain unchanged with the obvious exception in step 5 wherein when the algorithm stops, all c_{2i+1} would have been computed. An implementation of the above algorithm in Mathematica for n = 32, along with the results can be found in the appendix.

2.5 Analysis of part I

As explained in section 2.3, part I of the overall methodology for determining the cyclic convolution of the two n-point sequences, is comprised of the computations: Σc_{2i} and Σc_{2i+1}. These can be obtained by using one of the following three methods. The advantages and disadvantages of the three methods are compared and contrasted below.

Method 1:

We can use theorem 2.1 to obtain the sum of the cyclic convolution of all the even points and theorem 2.2 to obtain the sum of the cyclic convolution of all the odd points. Let us assume that each of the n points has k bits. Looking at equations (2.23)-(2.26) we notice that each of the equations is a function of all 2n points. In equation (2.23) all terms have positive signs and therefore the length of the result is $k + \log_2 2n$. In the other three equations, namely equations (2.24)-(2.26), half have positive signs and the other half have negative signs. In order to determine the length of the result, the worst case assumption would be when all the points with positive signs have a value of zero and all the points with negative signs have a value of $2^k - 1$, i.e. a maximum value. Such a situation will yield the smallest negative number of size $k + \log_2 n$. But since this number is negative, in an actual hardware implementation it will be represented in its two's complement form and will therefore require an additional bit. Thus the number of bits required for each of equations (2.24)-(2.26) will be $k + 1 + \log_2 n$ which is equal to

$k + \log_2 2n$. Thus we see that the number of bits required to represent a number is the same in all of the equations (2.23)-(2.26) and in general we can say that the number of bits required for any equation will be the same if either all terms have positive signs or half have positive and the other half have negative signs. Thus the ROMs required to generate the squares of each of these equations will be of the same size.

Each of the squares required by equation (2.27) can be generated by a ROM of size $2^{k + \log_2 2n} \times 2(k + \log_2 2n)$. Since there are four squares the total number of ROM bits required is given by

$$\text{ROM bits - 4 squares} = 8 \times 2^{k + \log_2 2n} \times (k + \log_2 2n) \qquad (2.53)$$

The number of additions required for adding 2n terms is equal to 2n -1. In order to get w_{n1} we would need 2n -1 additions. w_{n2} can be written as $(a_0 + a_1 + ... + a_{n-1}) - (b_0 + b_1 + ... + b_{n-1})$ thus needing only one more addition. (We assume that the cost of an adder is the same as that of a negator). w_{n3} can be written as $(a_0 + a_2 + ...+ a_{n-2}) - (a_1 + a_3 +... + a_{n-1}) + (b_0 + b_2 + ... + b_{n-2}) - (b_1 + b_3 + ... + b_{n-1})$ thus requiring only three additional additions while w_{n4} can be written as $(a_0 + a_2 + ...+ a_{n-2}) - (a_1 + a_3 +... + a_{n-1}) - ((b_0 + b_2 + ... + b_{n-2}) - (b_1 + b_3 + ... + b_{n-1}))$ thus requiring only one more addition as the rest is generated while generating w_{n3}. To do the additions and subtractions required by equations (2.27) and (2.28) we need a total of four more additions. Thus the total number of additions can be given by

$$\text{Additions - 4 squares} = 2n + 8 \qquad (2.54)$$

The time delay for performing all the additions required by equations (2.23) - (2.26) can be given by $\log_2 2n\ \Delta$ as each equation has 2n terms. Equations (2.27) and (2.28) will require an additional 2 Δ. Here we assume that $\Delta$is the time required for

adding two l bit numbers where l is in the range $k < l < 2(k + \log_2 2n)$. Thus the total time delay can be given as

$$\text{Time delay - 4 squares} = (3 + \log_2 n)\,\Delta \tag{2.55}$$

Method 2:

In method 1 we used theorems 2.1 and 2.2 while the same results can be obtained by using theorems 2.3 and 2.4. Here since each of equations (2.29)-(2.32) and (2.34)-(2.37) have only half the number of total points, the length of the result of each of these equations will be $k + \log_2 n$. Thus the square of each these equations can be implemented by using a ROM of size $2^{k + \log_2 n} \times 2(k + \log_2 n)$ and for 8 squares the total ROM bits can be given as

$$\text{ROM bits - 8 squares} = 16 \times 2^{k + \log_2 n} \times (k + \log_2 n) \tag{2.56}$$

The number of additions required for adding n terms is equal to n -1. In order to obtain x_{n1} and x_{n3} as given by equations (2.29) and (2.31) we would need 2(n-1) additions. x_{n2} and x_{n4} as given by equations (2.30) and (2.32) can then be obtained by just performing two more additions while y_{n1} - y_{n4} as given by equations (2.34) - (2.37) can be obtained by performing four more additions. Thus the total number of additions including those required by equations (2.33) and (2.38) can be given by

$$\text{Additions - 8 squares} = 2n + 10 \tag{2.57}$$

The time delay for performing all the additions required by equations (2.29) - (2.32) and (2.34) - (2.37) can be given by $\log_2 n\,\Delta$ as each equation has n terms. Further, equations (2.33) and (2.38) will require another 2Δ thus giving the total time delay as

$$\text{Time delay - 8 squares} = (2 + \log_2 n)\Delta \tag{2.58}$$

Method 3:

Another way of obtaining the sum of the cyclic convolution of all the even points and all the odd points separately would be by using theorems 2.3 and 2.7. Theorem 2.3 gives us the sum of the even points of the cyclic convolution while theorem 2.7 gives us the difference of the sum of the even points and the sum of the odd points. Therefore the sum of the odd points of the cyclic convolution can be obtained by taking the difference of the results of theorems 2.3 and 2.7.

In method 2 we have shown that the length of the result of each of the equations required by theorem 2.3 is equal to $k + \log_2 n$ while their squares can be implemented by ROMs of size $2^{k + \log_2 n} \times 2(k + \log_2 n)$. We note that there are four such ROMs. In theorem 2.7, both equations are a function of all 2n points and in method 1 we have shown that the result of such an equation is of length $k + \log_2 2n$. Each of the two squares can be implemented by using a ROM of size $2^{k + \log_2 2n} \times 2(k + \log_2 2n)$. Thus the total number of ROM bits required can be given as

$$\begin{aligned} \text{ROM bits - 6 squares} &= [8 \times 2^{k + \log_2 n} \times (k + \log_2 n)] + \\ &\quad [4 \times 2^{k + \log_2 2n} \times (k + \log_2 2n)] \\ &= 2^{k + 3 + \log_2 n} \times [2(k + \log_2 n) + 1] \end{aligned} \tag{2.59}$$

We have already shown in method 2 that the number of additions required by equations (2.29)-(2.32) is equal to (2(n-1) + 2). Equation (2.49) that generates w_{n5}, can also be written as $x_{n1} - x_{n3}$ thus requiring only one more addition. Equation (2.50) that generates w_{n6}, can also be written as $x_{n4} - x_{n2}$ thus again requiring only one more addition. Equations (2.33) and (2.51) require four additions while to subtract equation (2.51) from (2.33) we would need one more addition. Thus the total number of additions can be given by

$$\text{Additions - 6 squares} = 2n + 7 \tag{2.60}$$

To compute the time delay for performing all the additions required by this method it is easy for one to see that since equations (2.49) and (2.50) are the longest it will be sufficient to determine the time delay to compute these equations. It was shown in the analysis for method 1 that this is equal to $\log_2 2n\ \Delta$. To obtain equation (2.51) we will need one additional time unit or the total time can be given as $(\log_2 n + 2)\ \Delta$. The computation of equation (2.33) will also take $(\log_2 n + 2)\ \Delta$ while to obtain the difference of equation (2.33) and (2.51) we will need one more time unit, thus giving the total time delay as

$$\text{Time delay - 6 squares} = (3 + \log_2 n\ )\ \Delta \qquad (2.61)$$

2.5.1 Comparison of methods 1 - 3

Comparing equations (2.53), (2.56), and (2.59) we can see that the number of ROM bits required by method 2 is the least. With respect to time delay for performing the additions, comparing equations (2.55), (2.58), and (2.61) we once again find that method 2 is the best although it is only marginally faster than the other two methods. With respect to the number of additions, comparing equations (2.54), (2.57), and (2.60) we find that method 3 is the best although again by only a marginal amount.

However, we should note that if computations are being performed in some modular ring then method 1 would be the best as the size of the equations does not grow and therefore the number of squares is of prime importance. Clearly, method 1 would be the best as it requires only four squares.

For the sake of clarity we have in the remainder of the chapter assumed that part 1 is computed based on method 2 while keeping in mind that method 1 is to be used in the event arithmetic is done in some modular ring.

2.6 Number of squares

In section 2.5 we have shown that part 1 would need eight squares if method 2 was used. Irrespective of the methods used for computing part 1 the methodology for part 2 remains the same. In this section we present a formula to compute the number of squares required to compute part 2.

There are two ways of computing the number of squares required for part 2 of the overall methodology. One is by examining algorithm 2.3 and determining how many times step 2 is executed. This is because in step 2, theorem 2.8 is evaluated and theorem 2.8 in turn requires 2j squares to be computed. Thus by knowing the number of times step 2 is executed for different values of j, one can estimate the number of squares required. On the other hand, we can estimate the number of squares in a more intuitive fashion by looking more closely at the methodology of part 2. We recall that in algorithm 2.2 the following notations are used. The number of points being convolved is represented by the variable n while variable i is used to represent the index of the points of the cyclic convolution. The variable j is used to denote the difference between two consecutive indices of the points of the cyclic convolution while k is a local variable used to indicate the relationship between j and n. We now make the following observations:

1) Part 2 of the methodology treats the even points separately from the odd points, however either computation uses exactly the same concept. Therefore if we compute the number of squares required by the even points all we need to do is double that number to get the total number of squares.

2) For any given n the values of j are of the form 2^k with k = 1, 2, ..., $\log_2(n/2)$. The maximum value of j is equal to n/2 and we can say there are $\log_2(n/2)$ stages.

3) For any given value of n, i lies in the range 0 through (n/2 -1). Since we are computing the even points and odd points separately i is incremented by 2 every time. Thus the number of distinct i for a given n at every stage is equal to j/2.

4) The number of squares is a recursive formula, i.e., the number of squares required for any value of n is equal to the number of squares required when j = n/2 plus all the squares required for half the number of points i.e. when n := n/2. Thus at every stage we only need to determine the number of squares required for the maximum allowable value of j.

5) For every i there are 2j squares. Thus at every stage the number of squares is equal to the number of distinct i times 2j, thus giving $j/2 \times 2j = j^2$.

Let l_k represent the number of squares required at stage k. Then from the above observations it follows that $l_k = j^2 = 4^k$, k = 1, 2, 3, ..., $\log_2 (n/2)$. Thus the total number of squares required is given by

$$\text{number of squares} = \sum_{k=1}^{\log_2 n/2} 4^k$$

$$= 4/3 \times (4^{\log_2 n/2} - 1) \qquad (2.62)$$

Thus the total number of squares, for computing both odd and even points, can be given as

$$\text{Total number of squares} = 2 \times 4/3 \times (4^{\log_2 n/2} - 1)$$

$$= 2/3 \times (n^2 - 4) \qquad (2.63)$$

Table 2.1 compares the number of multiplication versus squaring operations required for cyclically convolving two n-point sequences. This table assumes that part 1 of our

Table 2.1: Comparison of the number of multiplication versus squaring operations

n	Number of multiplications n^2	Number of squares $2/3 \times (n^2 + 8)$	% savings
4	16	16	0
8	64	48	25
16	256	176	31.25
32	1024	688	32.81
64	4096	2736	33.20

overall methodology is evaluated based on method 2. Thus the number of squares in this table is computed based on equation (2.63) + 8 = 2/3 × (n^2 + 8).

2.7 Number of additions

We have already estimated the number of additions required for part 1 of the methodology. In this section we obtain a formula for determining the number of additions required for part 2 of the methodology. A brief description of the meaning of each of the variables can be found in the previous section.

For a given n, j has a maximum value of n/2. Also $j = 2 \times 2^k$; k = 0, 1, 2, ... , $\log_2 n/4$. For every value of j we have 2j equations consisting of a_i and b_i. From theorem 2.8 we see that each equation contains (n/j - 1) a_i terms and (n/j - 1) b_i terms. Also from algorithm 2.2, which shows how to construct these equations, we find that the a_i and b_i terms appear only j distinct times. The a_i and b_i terms appear in the same combinations for both odd and even points. The difference, however, is that the a_i and b_i terms are combined differently for even and odd points. Thus the number of additions for computing these equations can be given as

$$\text{Term 1 - additions} = 2\sum_{\forall j} j(n/j - 1) + \sum_{\forall j} 2j \times j \qquad (2.64)$$

or

$$\text{Term 1 - additions} = 2n \log_2 (n/2) - 2\sum_{\forall j} j + 2\sum_{\forall j} j^2 \qquad (2.65)$$

We note that the first term of equation (2.64) represents the number of additions required for obtaining the sum of the a_i terms and b_i terms separately. For every j there are n/j a_i terms and therefore we have (n/j -1) additions. Also, for every j there are j sets of such equations. This explains the product under the summation. The 2 outside the

summation accounts for the additions required by the b_i terms. The second term represents the number of additions required for combining the a_i and b_i terms. For every j there are 2j equations and j sets of such equations. This explains the product under the second summation in equation (2.64). (Note that this includes both the odd and even points).

Since for every value of j there are 2j squares to add-subtract these squares we would need 2j - 1 additions. Thus, we have

$$\text{Term 2 - additions} = \sum_{\forall j} j(2j-1) \tag{2.66}$$

The addition/subtraction of squares, which are accounted for by equation (2.66), generate for every j, $\sum_{k=0}^{n/j-1} c_{i+kj}\,(-1)^k$. These are in turn added with X^i as shown in step 3 of algorithm 2.3. Initially, X^i is the sum of all even c_i, or in other words it consists of n/2 points. An individual c_i is obtained by adding to this quantity the alternating difference of c_i with varying distances between consecutive c_i. This process thus requires (n/2 - 1) additions and subtractions, before individual c_i are computed. Thus the number of additions is 2(n/2 - 1) or (n - 2) additions and a similar amount would be needed for the odd c_i. Thus we have the total number of additions for this process as

$$\text{Term 3 - additions} = 2(n-2) \tag{2.67}$$

Therefore the total number of additions for part 2 of the methodology can be given by the sum of equations (2.65) - (2.67), thus giving

$$\text{Part 2 - additions} = 2n \log_2 (n/2) - 2\sum_{\forall j} j + 2\sum_{\forall j} j^2 + \sum_{\forall j} j(2j-1) + 2(n-2)$$

$$= 2(n + n \log_2 (n/2) - 2) + 4\sum_{\forall j} j^2 - 3\sum_{\forall j} j$$

$$= 2(n + n \log_2 (n/2) - 2) + 16 \sum_{k=0}^{\log(n/4)} 4^k - 6 \sum_{k=0}^{\log(n/4)} 2^k$$

$$= (1/3)[4n^2 + 6n \log_2 (n/2) - 3n - 10] \qquad (2.68)$$

Assuming that we use method 2 for part 1, we have

Total number of additions = $(1/3)[4n^2 + 6n \log_2 (n/2) + 3n + 20]$ (2.69)

2.8 Example

In this section we present in detail all the necessary computations required to compute the cyclic convolution of two 16-point sequences. The purpose of this example is to illustrate the methodology, theorems, algorithms, and notations developed earlier in this chapter. We consider two sequences A and B whose points are given as

$A = \{a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}\}$ and

$B = \{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}, b_{15}\}$ while their cyclic convolution is given by

$C = \{c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}\}$ where each point is defined by equation (2.22) or

$$c_i = \sum_{k=0}^{15} a_{<i-k>_{16}} b_k \qquad \text{for } i = 0, 1, 2, ..., 15$$

In order to use our methodology, we have to run through the steps of algorithm 2.3.

Procedure:

Step 1: $r = 1, i = 0$

Do theorem 2.3

This will, based on equations (2.29) - (2.32), yield:

$x_{n1} = a_0+a_2+a_4+a_6+a_8+a_{10}+a_{12}+a_{14}+b_0+b_2+b_4+b_6+b_8+b_{10}+b_{12}+b_{14}$

$x_{n2} = a_0+a_2+a_4+a_6+a_8+a_{10}+a_{12}+a_{14}-b_0-b_2-b_4-b_6-b_8-b_{10}-b_{12}-b_{14}$

$x_{n3} = a_1+a_3+a_5+a_7+a_9+a_{11}+a_{13}+a_{15}+b_1+b_3+b_5+b_7+b_9+b_{11}+b_{13}+b_{15}$

$x_{n4} = a_1+a_3+a_5+a_7+a_9+a_{11}+a_{13}+a_{15}-b_1-b_3-b_5-b_7-b_9-b_{11}-b_{13}-b_{15}$

and from equation (2.33) we get

$$x_{n1}^2 - x_{n2}^2 + x_{n3}^2 - x_{n4}^2 = 4\sum_{i=0}^{7} c_{2i} = 4(c_0 + c_2+...+c_{14})$$

Set result to X^i

Thus,

$$X^0 = 4(c_0+c_2+c_4+c_6+c_8+c_{10}+c_{12}+c_{14})$$

Step 2: $j = 2^r = 2$

Do theorem 2.8

This will call step 2 of algorithm 2.2 with i = 0, j = 2, k = 1, q = 0, p = i, and n = 16, thus yielding:

$V_{161}^0 = a_0-a_2+a_4-a_6+a_8-a_{10}+a_{12}-a_{14}+b_0-b_2+b_4-b_6+b_8-b_{10}+b_{12}-b_{14}$

$V_{162}^0 = a_0-a_2+a_4-a_6+a_8-a_{10}+a_{12}-a_{14}-b_0+b_2-b_4+b_6-b_8+b_{10}-b_{12}-b_{14}$

$V_{163}^0 = -a_1+a_3-a_5+a_7-a_9+a_{11}-a_{13}+a_{15}+b_1-b_3+b_5-b_7+b_9-b_{11}+b_{13}-b_{15}$

$V_{164}^0 = -a_1+a_3-a_5+a_7-a_9+a_{11}-a_{13}+a_{15}-b_1+b_3-b_5+b_7-b_9+b_{11}-b_{13}+b_{15}$

and based on equation (2.52) we get

$$\sum_{k=1}^{4}(V_{16k}^0)^2(-1)^{k+1} = 4\sum_{k=0}^{7} c_{2k}(-1)^k = 4(c_0-c_2+c_4-c_6+c_8-c_{10}+c_{12}-c_{14})$$

Set result to Z

$$Z = 4(c_0-c_2+c_4-c_6+c_8-c_{10}+c_{12}-c_{14})$$

Step 3:

$X^0 \leftarrow\!\!- X^0 + Z$

$= 8(c_0+c_4+c_8+c_{12})$

and $X^2 \leftarrow\!\!- X^2 - Z$

$= 8(c_2+c_6+c_{10}+c_{14})$

Also, $j \neq 8$, and therefore we got to step 4.

Step 4: $j <= 8$ is true and therefore $r \leftarrow\!\!- r + 1 = 2$ and we go to step 2.

Step 2: $j = 2^r = 4$

Do theorem 2.8

This will call step 2 of algorithm 2.2 with $i = 0$, $j = 4$, $k = 1$, $q = 0$, $p = i$, and $n = 16$, thus yielding:

$V^0_{161} = a_0 - a_4 + a_8 - a_{12} + b_0 - b_4 + b_8 - b_{12}$

$V^0_{162} = a_0 - a_4 + a_8 - a_{12} - b_0 + b_4 - b_8 + b_{12}$

$V^0_{163} = -a_3 + a_7 - a_{11} + a_{15} + b_1 - b_5 + b_9 - b_{13}$

$V^0_{164} = -a_3 + a_7 - a_{11} + a_{15} - b_1 + b_5 - b_9 + b_{13}$

$V^0_{165} = -a_2 + a_6 - a_{10} + a_{14} + b_2 - b_6 + b_{10} - b_{14}$

$V^0_{166} = -a_2 + a_6 - a_{10} + a_{14} - b_2 + b_6 - b_{10} + b_{14}$

$V^0_{167} = -a1 + a5 - a9 + a13 + b3 - b7 + b11 - b15$

$V^0_{168} = -a1 + a5 - a9 + a13 - b3 + b7 - b11 + b15$

and based on equation (2.52) we get

$$\sum_{k=1}^{8}(V^0_{16k})^2(-1)^{k+1} = 4\sum_{k=0}^{3}c_{4k}(-1)^k = 4(c_0 - c_4 + c_8 - c_{12})$$

Set result to Z

$Z = 4(c_0 - c_4 + c_8 - c_{12})$

Step 3:

$$X^0 \text{<--- } X^0 + Z \times 2$$
$$= 16(c_0 + c_8)$$

and $$X^4 \text{<--- } X^2 - Z \times 2$$
$$= 16(c_4 + c_{12})$$

Also, $j \neq 8$, and therefore we got to step 4.

Step 4: $j <= 8$ is true and therefore r <--- $r + 1 = 3$ and we go to step 2.

Step 2: $j = 2^r = 8$

Do theorem 2.8

This will call step 2 of algorithm 2.2 with $i = 0$, $j = 8$, $k = 1$, $q = 0$, $p = i$, and $n = 16$, thus yielding:

$$V^0_{161} = a_0 - a_8 + b_0 - b_8$$
$$V^0_{162} = a_0 - a_8 - b_0 + b_8$$
$$V^0_{163} = -a_7 + a_{15} + b_1 - b_9$$
$$V^0_{164} = -a_7 + a_{15} - b_1 + b_9$$
$$V^0_{165} = -a_6 + a_{14} + b_2 - b_{10}$$
$$V^0_{166} = -a_6 + a_{14} - b_2 + b_{10}$$
$$V^0_{167} = -a_5 + a_{13} + b_3 - b_{11}$$
$$V^0_{168} = -a_5 + a_{13} - b_3 + b_{11}$$
$$V^0_{169} = -a_4 + a_{12} + b_4 - b_{12}$$
$$V^0_{1610} = -a_4 + a_{12} - b_4 + b_{12}$$
$$V^0_{1611} = -a_3 + a_{11} + b_5 - b_{13}$$
$$V^0_{1612} = -a_3 + a_{11} - b_5 + b_{13}$$
$$V^0_{1613} = -a_2 + a_{10} + b_6 - b_{14}$$
$$V^0_{1614} = -a_2 + a_{10} - b_6 + b_{14}$$
$$V^0_{1615} = -a_1 + a_9 + b_7 - b_{15}$$
$$V^0_{1616} = -a_1 + a_9 - b_7 + b_{15}$$

and based on equation (2.52) we get

$$\sum_{k=1}^{16}(V_{16k}^{0})^2(-1)^{k+1} = 4\sum_{k=0}^{1}c_{8k}(-1)^{k} = 4(c_0 - c_8)$$

Set result to Z

$Z = 4(c_0 - c_8)$

Step 3:

$X^0 <\text{---} X^0 + Z \times 4$

$= 32c_0$

and $X^8 <\text{---} X^0 - Z \times 4$

$= 32c_8$

Also, j = 8, and we note that c_0 and c_8 are indeed equal to $X^0/32$ and $X^8/32$, and we go to step 5.

Step 5: i <--- i + 2 = 2 and i <= 7 is true, therefore r = 2 and we go to step 2.

Step 2: $j = 2^r = 4$

Do theorem 2.8

This will call step 2 of algorithm 2.2 with i = 2, j = 4, k = 1, q = 0, p = i, and n = 16, thus yielding:

$V_{161}^2 = a_2 - a_6 + a_{10} - a_{14} + b_0 - b_4 + b_8 - b_{12}$

$V_{162}^2 = a_2 - a_6 + a_{10} - a_{14} - b_0 + b_4 - b_8 + b_{12}$

$V_{163}^2 = a_1 - a_5 + a_9 - a_{13} + b_1 - b_5 + b_9 - b_{13}$

$V_{164}^2 = a_1 - a_5 + a_9 - a_{13} - b_1 + b_5 - b_9 + b_{13}$

$V_{165}^2 = a_0 - a_4 + a_8 - a_{12} + b_2 - b_6 + b_{10} - b_{14}$

$V^2_{166} = a_0 - a_4 + a_8 - a_{12} - b_2 + b_6 - b_{10} + b_{14}$

$V^2_{167} = -a_3 + a_7 - a_{11} + a_{15} + b_3 - b_7 + b_{11} - b_{15}$

$V^2_{168} = -a_3 + a_7 - a_{11} + a_{15} - b_3 + b_7 - b_{11} + b_{15}$

and based on equation (2.52) we get

$$\sum_{k=1}^{8}(V^2_{16k})^2(-1)^{k+1} = 4\sum_{k=0}^{3}c_{2+4k}(-1)^k = 4(c_2 - c_6 + c_{10} - c_{14})$$

Set result to Z

$$Z = 4(c_2 - c_6 + c_{10} - c_{14})$$

Step 3:

$X^2 <\text{---} X^2 + Z \times 2$

$= 16(c_2 + c_{10})$

and $X^6 <\text{---} X^2 - Z \times 2$

$= 16(c_6 + c_{14})$

Also, $j \neq 8$, and therefore we got to step 4.

Step 4: $j <= 8$ is true and therefore $r <\text{---} r + 1 = 3$ and we go to step 2.

Step 2: $j = 2^r = 8$

Do theorem 2.8

This will call step 2 of algorithm 2.2 with i = 2, j = 8, k = 1, q = 0, p = i, and n = 16, thus yielding:

$V^2_{161} = a_2 - a_{10} + b_0 - b_8$

$V^2_{162} = a_2 - a_{10} - b_0 + b_8$

$V^2_{163} = a_1 - a_9 + b_1 - b_9$

$V^2_{164} = a_1 - a_9 - b_1 + b_9$

$V^2_{165} = a_0 - a_8 + b_2 - b_{10}$

$V^2_{166} = a_0 - a_8 - b_2 + b_{10}$

$V^2_{167} = -a_7 + a_{15} + b_3 - b_{11}$

$V^2_{168} = -a_7 + a_{15} - b_3 + b_{11}$

$V^2_{169} = -a_6 + a_{14} + b_4 - b_{12}$

$V^2_{1610} = -a_6 + a_{14} - b_4 + b_{12}$

$V^2_{1611} = -a_5 + a_{13} + b_5 - b_{13}$

$V^2_{1612} = -a_5 + a_{13} - b_5 + b_{13}$

$V^2_{1613} = -a_4 + a_{12} + b_6 - b_{14}$

$V^2_{1614} = -a_4 + a_{12} - b_6 + b_{14}$

$V^2_{1615} = -a_3 + a_{11} + b_7 - b_{15}$

$V^2_{1616} = -a_3 + a_{11} - b_7 + b_{15}$

and based on equation (2.52) we get

$$\sum_{k=1}^{16}(V^2_{16k})^2(-1)^{k+1} = 4\sum_{k=0}^{1}c_{2+8k}(-1)^k = 4(c_2 - c_{10})$$

Set result to Z

$Z = 4(c_2 - c_{10})$

Step 3:

$X^2 \text{ <--- } X^2 + Z \times 4$

$= 32c_2$

and $X^{10} \text{ <--- } X^2 - Z \times 4$

$= 32c_{10}$

Also, j = 8, and we note that c_2 and c_{10} are indeed equal to $X^2/32$ and $X^{10}/32$, and we go to step 5.

Step 5: i <--- i + 2 = 4 and i <= 7 is true, therefore r = 3 and go to step 2.

Step 2: $j = 2^r = 8$

Do theorem 2.8

This will call step 2 of algorithm 2.2 with i = 4, j = 8, k = 1, q = 0, p = i, and n = 16, thus yielding:

$V^4_{161} = a_4 - a_{12} + b_0 - b_8$

$V^4_{162} = a_4 - a_{12} - b_0 + b_8$

$V^4_{163} = a_3 - a_{11} + b_1 - b_9$

$V^4_{164} = a_3 - a_{11} - b_1 + b_9$

$V^4_{165} = a_2 - a_{10} + b_2 - b_{10}$

$V^4_{166} = a_2 - a_{10} - b_2 + b_{10}$

$V^4_{167} = a_1 - a_9 + b_3 - b_{11}$

$V^4_{168} = a_1 - a_9 - b_3 + b_{11}$

$V^4_{169} = a_0 - a_8 + b_4 - b_{12}$

$V^4_{1610} = a_0 - a_8 - b_4 + b_{12}$

$V^4_{1611} = -a_7 + a_{15} + b_5 - b_{13}$

$V^4_{1612} = -a_7 + a_{15} - b_5 + b_{13}$

$V^4_{1613} = -a_6 + a_{14} + b_6 - b_{14}$

$V^4_{1614} = -a_6 + a_{14} - b_6 + b_{14}$

$V^4_{1615} = -a_5 + a_{13} + b_7 - b_{15}$

$V^4_{1616} = -a_5 + a_{13} - b_7 + b_{15}$

and based on equation (2.52) we get

$$\sum_{k=1}^{16} (V^4_{16k})^2 (-1)^{k+1} = 4 \sum_{k=0}^{1} c_{4+8k} (-1)^k = 4(c_4 - c_{12})$$

Set result to Z

$Z = 4(c_4 - c_{12})$

Step 3:

$$X^4 \text{ <--- } X^4 + Z \times 4$$
$$= 32c_4$$

and $$X^{12} \text{ <--- } X^4 - Z \times 4$$
$$= 32c_{12}$$

Also, j = 8, and we note that c_4 and c_{12} are indeed equal to $X^4/32$ and $X^{12}/32$, and we go to step 5.

Step 5: i <--- i + 2 = 6 and i <= 7 is true, therefore r = 3 and go to step 2.

Step 2: $j = 2^r = 8$

Do theorem 2.8

This will call step 2 of algorithm 2.2 with i = 6, j = 8, k = 1, q = 0, p = i, and n = 16, thus yielding:

$V^6_{161} = a_6 - a_{14} + b_0 - b_8$

$V^6_{162} = a_6 - a_{14} - b_0 + b_8$

$V^6_{163} = a_5 - a_{13} + b_1 - b_9$

$V^6_{164} = a_5 - a_{13} - b_1 + b_9$

$V^6_{165} = a_4 - a_{12} + b_2 - b_{10}$

$V^6_{166} = a_4 - a_{12} - b_2 + b_{10}$

$V^6_{167} = a_3 - a_{11} + b_3 - b_{11}$

$V^6_{168} = a_3 - a_{11} - b_3 + b_{11}$

$V^6_{169} = a_2 - a_{10} + b_4 - b_{12}$

$V^6_{1610} = a_2 - a_{10} - b_4 + b_{12}$

$V^6_{1611} = a_1 - a_9 + b_5 - b_{13}$

$V^6_{1612} = a_1 - a_9 - b_5 + b_{13}$

$V^6_{1613} = a_0 - a_8 + b_6 - b_{14}$

$V^6_{1614} = a_0 - a_8 - b_6 + b_{14}$

$V^6_{1615} = -a_7 + a_{15} + b_7 - b_{15}$

$V^6_{1616} = -a_7 + a_{15} - b_7 + b_{15}$

and based on equation (2.52) we get

$$\sum_{k=1}^{16}(V^6_{16k})^2(-1)^{k+1} = 4\sum_{k=0}^{1}c_{6+8k}(-1)^k = 4(c_6 - c_{14})$$

Set result to Z

$$Z = 4(c_6 - c_{14})$$

Step 3:

$X^6 <\text{---} X^6 + Z \times 4$

$= 32c_6$

and $X^{14} <\text{---} X^6 - Z \times 4$

$= 32c_{14}$

Also, j = 8, and we note that c_6 and c_{14} are indeed equal to $X^6/32$ and $X^{14}/32$, and we go to step 5.

Step 5: i <--- i + 2 = 8 and i <= 7 is false and so we stop. We observe that all c_{2i} have been computed.

The c_{2i+1} can be computed in a similar fashion and therefore it is not presented here.

2.9 Summary

All the necessary theory required to compute the cyclic convolution of two n-point sequences where n is a power of 2 has been developed in this chapter. We have presented a new methodology for a hardware based implementation of the cyclic convolution operation. Eight theorems that were developed as a part of this dissertation form the mathematical basis for our methodology. Our methodology consists of two parts, part 1 and part 2. By selective utilization of the theorems, part 1 can be evaluated in three different ways, referred to as methods 1, 2, and 3. A comparative analysis of the three methods was provided in section 2.5. Part 2 can only be evaluated in a unique manner and the required equations are provided by algorithm 2.2. The overall methodology was described by algorithm 2.3. The algorithms did not use approximations of any kind and are therefore inherently free of any round-off errors, thus eliminating the need for error correcting hardware. To complete the theory we have also derived non-recursive formulae to obtain the number of squares and additions required by our methodology.

We have shown that while our algorithms require fewer squares than multiplications required by a traditional computation, we require more two-operand additions. Further, the number of squares has been approximately reduced by one-third while the number of additions have been increased by about a third. However, we observe that this is not a zero gain, for we have decreased the number of expensive operations, namely, the multiplication operations, at the cost of increasing the inexpensive operations, namely, the addition operations. One must note that this is a fact independent of the technology of implementation. Also, we note that the formula on the number of additions is not an accurate reflection of the increase in hardware cost. This is because our implementations rely primarily on multi-operand additions and by selecting

a suitable implementation, one can not only reduce the amount of hardware but can also decrease the time delay associated with the computation. In the next chapter implementation issues of the convolution operation based on the definition and our method are discussed in detail.

Chapter 3

Implementation Issues

In the previous chapter we have developed algorithms for performing the cyclic convolution of two n-point sequences. In place of the multiplication operation, we used the squaring operation. While our algorithms require fewer squaring operations when compared to the number of multiplication operations required by the traditional technique, we require more addition operations. However, since our equations require multi-operand adders the count of two-operand addition operations may be somewhat misleading. In this chapter we present carry-save adder (CSA) and read only memory (ROM) based implementations and discuss hardware cost and speed trade-offs. The purpose of these implementations are not to provide the DSP engineer with an off the shelf design but more for the purpose of precisely analyzing the effect of the increase in the number of additions caused by our methodology. We then also show how the convolution operation can be applied to the problem of multiplying two numbers.

3.1 CSA implementation of the multiplication operation

Since our goal is to compare hardware requirements of algorithms based on the multiplication operation with that of algorithms based on the squaring operation, we first consider the implementation of the elementary functions. We illustrate this comparison by calculating the hardware required for a CSA implementation of an eight-by-eight multiplier and in the following section we calculate the hardware required for a CSA implementation of an eight bit squarer. These implementations are based on schemes for parallel multipliers offered by Dadda [54]. Dadda's method is based on successively

adding the corresponding significant columns of the partial products until only two numbers are left. The sum of these two numbers yields the product of the original two numbers. In [54], Dadda has shown that there exists a sequence of numbers which should be used to determine the appropriate height of the partial product matrix at each level. For the case when a full adder is used the sequence of numbers is: 2, 3, 4, 6, 9, 13, 19, 28, e.t.c. Each term in the sequence is obtained by multiplying the preceding term by 3/2 and taking the integral part. The use of such a sequence of numbers generally yields the fastest implementation with the least amount of hardware required.

Figure 3.1 shows the multiplier scheme for obtaining the product of two eight bit numbers. The multiplication of two eight bit numbers requires the summation of eight partial products each of length eight bits. Each 'x' in the figure represents a single bit of the partial product, with the least significant bit on the right most side.

The CSA tree implementation requires four levels, labeled 1 through 4 on the left side of figure 3.1, to obtain two numbers whose sum yields the product. Each of the level requires a delay equal to the delay through one full adder, in this context also known as a carry-save adder, and is denoted as $1 \times D_{CSA}$ while the final two numbers can be added using a fast adder such as the carry look ahead adder. For the purpose of providing a fair comparison between all methods we assume a simple ripple carry propagate adder(CPA). We denote the delay of such an adder as $1 \times D_{CPA}$. The number of full adders and half adders required in each of the levels is indicated on the right side of figure 3.1. (The notation of indicating the level numbers and the number of full and half adders will be used in all figures depicting CSA tree implementations.) The maximum height of a column in level 1 is 8. From the sequence given earlier, we find that the largest number less than 8 is 6. Therefore, the objective at this level is to ensure that the height of every column is not greater than 6 at the next level. Also, this is to be

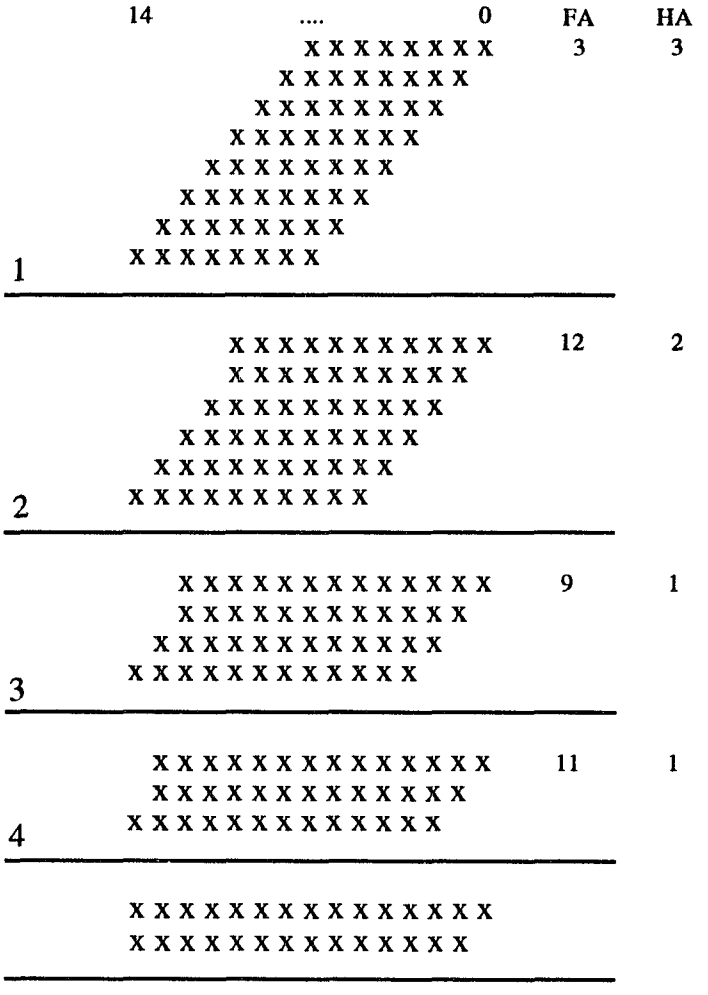


Figure 3.1: CSA implementation of an 8 x 8 multiplier

achieved by using a minimum number of half and full adders. Note that the use of a full adder reduces the height of the column by 2 while a half adder reduces the height of the column by 1. For instance, at level 1, columns 0 through 5 need no manipulations, for they all have a height less than or equal to 6. Column 6 uses one half adder to reduce its height from 7 to 6, while column 7 uses one half adder and one full adder to reduce its height from 8 to 6. The use of one half adder and one full adder actually reduces the height of column 7 by three but since there is a carry-in from column 6, the final height of column 7 in level 2 is 6. In a similar fashion the others columns are reduced. Eventually after level 4 we are left with two numbers that are added using a CPA to produce the final result. From figure 3.1 we see that the hardware required by an 8 x 8 multiplier requires 35 full adders, 7 half adders, and one 15 bit CPA. If the CPA is based on a simple ripple carry adder then the CPA would require 14 full adders and 1 half adder. Thus the total number of full adders required is 49 and the total number of half adders required is 8. The time delay for the entire computation can be given as $4\times D_{CSA} + 1\times D_{CPA}$.

In order to mechanize the computation of the amount of hardware required for a CSA based implementation of multi-operand addition, the above procedure is written as an algorithm and then coded in Mathematica. The Mathematica version can be found in the appendix.

Algorithm 3.1

Input: A list, L, whose elements are the heights of the columns of the array of numbers to be added, with the elements listed in the order of least to most significant. Let the elements of this set be $L_1, L_2, ..., L_j$.

Output: The amount of hardware required to add the array of numbers in terms of full adders, half adders, and two-input gates and the time delay required to perform this computation in terms of number of CSA and CPA levels.

Procedure:

Step 1: Denote the largest element in L as max[L]. Obtain a set, T, of the numbers of the sequence based on [54], with the last element of T being the largest number smaller than max[L]. Let the elements of this set be T_1, T_2, ..., T_i. Use variables fa and ha to keep track of the number of full adders and half adders respectively. Initially these variables are set to zero.

Step 2: The cardinality of set T is the number of CSA levels required to add the array of numbers. The number of CPA levels is always one.

Step 3: Set j = 1.

Step 4: If $L_j > T_i$, then fa-temp = $\lfloor (L_j - T_i)/2 \rfloor$;
ha-temp = $\lceil (L_j - T_i)/2 \rceil$ - fa-temp;

Step 5: Set $L_j = T_i$, and increase the height of the next column by the carries generated by the current column. i.e. the height will be increased by an amount equal to the sum of fa-temp and ha-temp.

Step 6: fa = fa + fa-temp and ha = ha + ha-temp.

Step 7: j = j + 1; repeat steps 4, 5, 6. (i.e these steps are performed on all columns of set L.)

Step 8: When there are no more columns to update, repeat steps 3 through 6 for the next smaller element in T, i.e., T_{i-1}.

Step 9: When there are no more elements in set T, the number of full and half adders required by the CSA tree has been computed.

Step 10: Add to the number of full and half adders the size of the carry propagate adder in terms of full and half adders.

Step 11: Compute the number of two-input gates by treating a full adder as 5 two-input gates and a half adder as 2 two-input gates.

The justification for step 4 is given as follows: Each full adder reduces the column size by 2 bits. Therefore, if the number of bits to be reduced is odd, say e = 2k + 1, then k full adders and 1 half adder will be required to reduce the height by e bits. Since 1 half adder reduces the height by 1 bit, one can also say that 1/2 a full adder reduces the column size by 1 bit. Thus each non-zero fractional part of the computation of the number of full adders contributes a single half adder.

3.2 CSA implementation of the squaring operation

Now we consider the squaring of an eight bit number. Let the number be represented as $A = a_7a_6a_5a_4a_3a_2a_1a_0$. The square of A can be given as

$$\begin{aligned} A^2 = {} & a_7\,2^{14} + a_7a_6\,2^{14} + a_7a_5\,2^{13} + a_6\,2^{12} + a_7a_4\,2^{12} + a_6a_5\,2^{12} + a_7a_3\,2^{11} + a_6a_4\,2^{11} \\ & + a_5\,2^{10} + a_7a_2\,2^{10} + a_6a_3\,2^{10} + a_5a_4\,2^{10} + a_7a_1\,2^9 + a_6a_2\,2^9 + a_5a_3\,2^9 + a_4\,2^8 \\ & + a_7a_0\,2^8 + a_6a_1\,2^8 + a_5a_2\,2^8 + a_4a_3\,2^8 + a_6a_0\,2^7 + a_5a_1\,2^7 + a_4a_2\,2^7 + a_3\,2^6 \\ & + a_5a_0\,2^6 + a_4a_1\,2^6 + a_3a_2\,2^6 + a_4a_0\,2^5 + a_3a_1\,2^5 + a_2\,2^4 + a_3a_0\,2^4 + a_2a_1\,2^4 \\ & + a_2a_0\,2^3 + a_1\,2^2 + a_1a_0\,2^2 + a_0 \end{aligned} \qquad (3.1)$$

Each of the product terms a_ia_j, also called summands, can be obtained by a 2-input AND gate since a_i and a_j are each one bit long. The addition of the various terms of equation 3.1 can be obtained by re-arranging them as an array of summands as shown in figure 3.2. Here the terms of each column have the same weight. In a sense, we can say that we are adding five 15-bit numbers many of whose individual bits are zero. These zero bits are not shown in the figure.

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a_7	a_7a_5	a_6	a_6a_4	a_5	a_5a_3	a_4	a_4a_2	a_3	a_3a_1	a_2	a_2a_0	a_1		a_0
a_7a_6		a_6a_5	a_7a_3	a_5a_4	a_6a_2	a_4a_3	a_5a_1	a_3a_2	a_4a_0	a_2a_1		a_1a_0		
		a_7a_4		a_6a_3	a_7a_1	a_5a_2	a_6a_0	a_4a_1		a_3a_0				
				a_7a_2		a_6a_1		a_5a_0						
						a_7a_0								

Figure 3.2: Array of summands for an 8 bit squarer

Figure 3.3 shows the CSA implementation of the eight bit squarer. Here each term of figure 3.2 is represented by a x. The figure relies again on the sequence of numbers given in section 3.1, namely 2, 3, 4. Since the height of the tallest column is 5, in level 1 the objective is to group the x's such that the height of no column is greater than 4. In level 2 the objective is to limit the height to 3 and in level 3 to 2.

From the figure we see that the amount of hardware required is 10 full adders, 5 half adders, and one 15 bit CPA and the time delay is $3\times D_{CSA}$. Comparing this with the hardware required by an 8 x 8 multiplier we find that the squarer requires around a third of that required by a multiplier and is faster by $1\times D_{CSA}$. Clearly, there is an advantage to designing algorithms around the squaring operation. While designing CSA based implementations, one is generally guided by [54]-[56]. However, a closer look at the array of summands to be added for the squaring operations yields the configuration shown in figure 3.4. Here although the total amount of hardware is the same as that of figure 3.3, we find that there are fewer levels , i.e. it is faster by $1\times D_{CSA}$. We observe that by treating the cost of a full adder as 5 two-input gates and the cost of a half adder as 2 two-input gates the total cost including the cost of $a_i a_j$ terms is 88 two-input gates plus one CPA.

In summary, it appears that the number of levels required to add a set of summands in a parallel fashion is not only a function of the height of the tallest column but is also a function of the heights of the other columns and their relative placements. One must note that such a situation does not arise in the multiplication of two distinct numbers as the height of the columns then monotonically increases, reaches a maximum value, and then monotonically decreases.

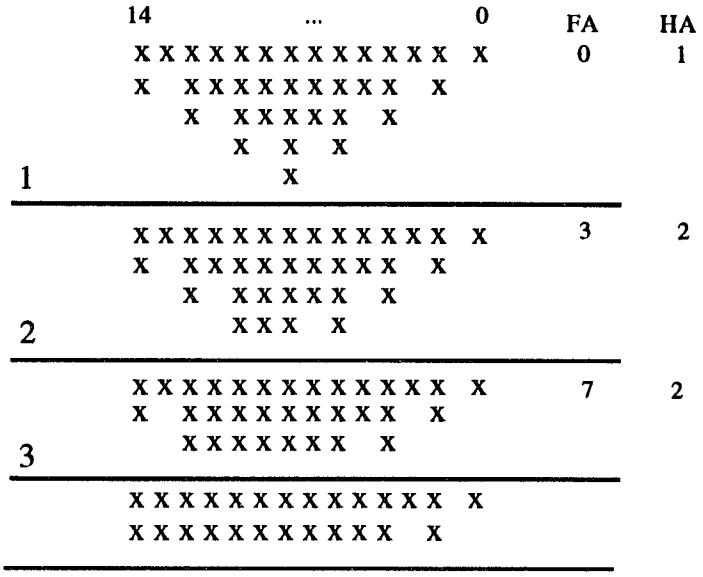


Figure 3.3: CSA implementation of an 8 bit squarer.

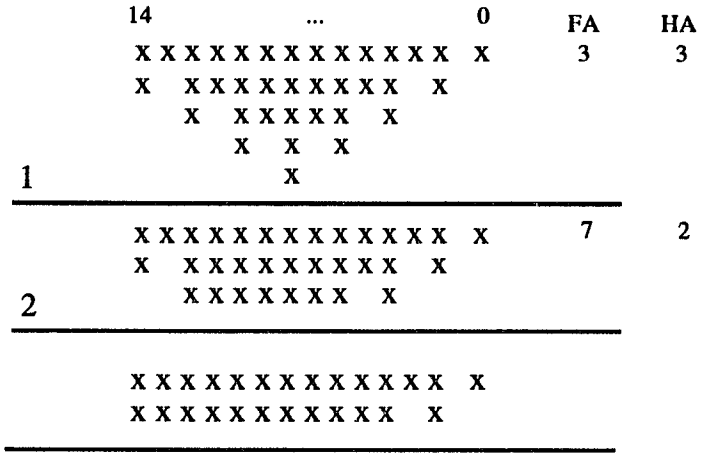


Figure 3.4: Intuitive CSA implementation of an 8 bit squarer.

3.3 Alternate CSA implementation of the squaring operation

Other parallel implementations of the squaring operation can be found in [57], [58] while serial implementations can be found in [59]. Jayashree and Basu in [58] show that their method is both faster and cheaper than that of [57]. In this section we propose yet another parallel implementation that not only compares very well with [58] with respect to both cost and speed but is also regular, more modular, and easier to design.

We first present our alternate method and then compare it with reference [58]. Looking at figure 3.2 we notice that in each of the columns 2, 4, 6, 8, 10, 12, and 14 there exist terms of the nature a_i and $a_i a_{i-1}$. For instance, in column 6 we have the terms a_3 and $a_3 a_2$. We can therefore substitute in place of these two terms their sum, $a_i \overline{a_{i-1}}$, and carry, $a_i a_{i-1}$. In the case of column 6 the sum $a_3 \overline{a_2}$ replaces the terms a_3 and $a_3 a_2$ and the carry $a_3 a_2$ is placed in column 7. Thus we have reduced the height of column 6 by one and at the same time increased the height of column 7 by one. Performing this simple manipulation on every such pair of terms yields the array as shown in figure 3.5.

We observe from this figure:

i) the height of the tallest column is less than that of figure 3.2 by one, and

ii) the array of summands to be added now exhibits a very regular structure.

Method of [54] is then applied to this reduced regular array of summands to yield the final result. Figure 3.6 details the implementation and from this we see that the amount of hardware required is 9 full adders, 5 half adders, and one 15 bit CPA and the time delay is $2 \times D_{CSA} + 1 \times D_{CPA}$. Thus, compared with the implementation in section 3.2, with no loss in speed we have reduced the number of full adders by 1 and have

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a_7a_6	$a_7\bar{a}_6$	a_7a_5	a_7a_4	a_7a_3	a_7a_2	a_7a_1	a_7a_0	a_6a_0	a_5a_0	a_4a_0	a_3a_0	a_2a_0	$a_1\bar{a}_0$		a_0
		a_6a_5	$a_6\bar{a}_5$	a_6a_4	a_6a_3	a_6a_2	a_6a_1	a_5a_1	a_4a_1	a_2a_1	$a_2\bar{a}_1$	a_1a_0			
				a_5a_4	$a_5\bar{a}_4$	a_5a_3	a_5a_2	a_3a_2	$a_3\bar{a}_2$	a_3a_1					
						a_4a_3	$a_4\bar{a}_3$	a_4a_2							

Figure 3.5: Reduced and regular array of summands for an 8 bit squarer

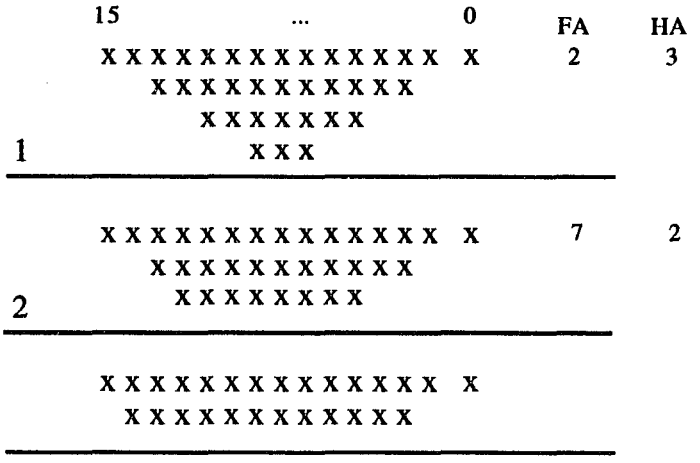


Figure 3.6: CSA implementation of reduced 8 bit squarer.

also produced a regular structure. However, our manipulations will require 7 additional two-input gates which compares with the cost of a full adder. Treating the cost of a full adder as 5 two-input gates and the cost of the half adder as 2 two-input gates the total cost including the cost of $a_i a_j$ terms is 90 two-input gates plus one CPA. In summary our manipulations have not resulted in either hardware or speed improvement but has achieved regularity.

Our method and that of [58] rely on the same basic principle, i.e. first reduce the squaring matrix and then apply Dadda's scheme to obtain the final result. In the process of using Dadda's scheme both methods rely on a CPA to compute the final sum. Since this step requires the maximum amount of time the CPA is generally implemented using some fast carry lookahead adder. However, since both methods require this CPA, the cost and delay of this unit can be ignored without affecting the quality of the analysis. One might argue that the CPA in [58] is smaller than ours by 2 bits, but needless to say, this is marginal. Thus the comparison process reduces to estimating the hardware and time delay required for the data stream to reach the CPA. In [58] the authors rely on the properties of the squaring matrix which has the shape of a parallelogram while we rely on equation 3.1 and some simple manipulations.

We now estimate the amount of hardware and time delay required by [58] and for the sake of clarity we use their notation. In [58], in order to reduce the height of the columns, the authors define equations L_1 through L_{16}. The authors of [58] state that the generation of these equations requires no full adders. While this is true, the hardware and time delay required by these equations is the same as that of a full adder. The cost of hardware required by the terms L_i where i = 1 through 16 is estimated as follows. Terms L_1 and L_2 require no hardware. Terms L_i, where i = 5, 7, 9, 11, 13, are of the form $a_{(i+1)/2}\bar{a}_{(i-1)/2}(\bar{a}_{(i-3)/2} + \bar{a}_{(i+3)/2}) + a_{(i-3)/2}(a_{(i+1)/2} \oplus a_{(i+3)/2})$. Clearly this is a 3-

level circuit with 6 two-input gates. A full adder can also be realized with 3 levels while requiring only 5 two-input gates. Terms L_i, where i = 6, 8, 10, 12, 14, are of the form $a_{(i-2)/2}(a_{i/2} \oplus a_{(i+2)/2}) + a_{(i-4)/2}a_{i/2}a_{(i+2)/2}$. Each of these terms require 5 two-input gates and 3 levels. Each of terms L_3 and L_{16} require one gate while each of the terms L_4 and L_{15} require two gates. From the above analysis it is clear that the generation of all L_i terms requires the same delay as that of a full adder, i.e., $1 \times D_{CSA}$ while the total hardware is 61 gates. The remaining a_ia_j terms require 10 two-input gates and to reduce the L_i and a_ia_j terms to two numbers that can serve as inputs to the CPA requires 2 full adders and 3 half adders. These adders contribute another $1 \times D_{CSA}$. Thus the hardware required is 87 two-input gates and the time delay is $2 \times D_{CSA}$. Once again, there is no improvement in speed while the hardware cost is only marginally better. However, this method requires four additional types of hardware units, over and above the full adder and half adder. Two distinct types of units are needed to realize L_4 and L_{15} while two more distinct types of units are required to realize the other odd and even L_i terms.

In summary, we have presented an alternate parallel implementation for the squaring operation and have shown that while its performance and hardware cost is approximately the same as that of [58], our implementation is not only regular but also simpler to design and more modular in the sense of requiring fewer types of hardware units. Further analysis shows that the technique of [58] for higher word lengths produces hardware savings, but is slower. It appears that for small word lengths different designs yield similar hardware cost and speed functions. Thus for VLSI implementations it may be more important to focus on designs that are both regular and modular.

3.4 CSA implementation of the cyclic convolution

In the previous section we have discussed CSA implementations of the squaring operation in detail. We had clearly demonstrated that the computation of the squaring operation is both faster and cheaper than that of a multiplication operation. We observe that this is primarily due to the fact that the multiplication operation has n^2 summands while the squaring operation has $\binom{n}{2} + n$ summands. While both operations require summands in the order of n^2, the squaring operation in terms of absolute values contains approximately half the summands. Thus, if a particular function can be evaluated by using either squaring or multiplication operations and the number of operations in either case are of the same order, then it is reasonable to expect to have hardware savings in the magnitude of a factor as opposed to an order.

Before we discuss implementations of the cyclic convolution operation, we would once again like to emphasize that the purpose of these implementations are not to provide the DSP engineer with an off the shelf design but more for the purpose of analyzing the effect of the increase in the number of additions by our methodology. Referring to equations (2.63) and (2.69) we find that we have reduced the number of squaring operations by one-third and at the same time we have increased the number of addition operations by a third. At this point we hypothesize that this is not a zero gain. In the ensuing sections we demonstrate the validity of this hypothesis by deriving the cost and speed functions of the cyclic convolution of 4, 8, and 16 points. We present three CSA based implementations, the first two are based on the definition and the third is based on our methodology. We call the first implementation, traditional, the second, modular, and the third, squares. We conclude our section by presenting a detailed discussion that analyzes all the results obtained.

3.4.1 4-point cyclic convolution-traditional

The cyclic convolution of two four point sequences is considered. Let the two sequences be A and B with $A = \{a_3, a_2, a_1, a_0\}$ and $B = \{b_3, b_2, b_1, b_0\}$. Let each of these points be of length 8 bits and represented in two's complement form. An implementation by definition would require the computation of 16 products, i.e. every point of a sequence is multiplied with every point of the other sequence. The cyclic convolution C is given by $C = \{c_3, c_2, c_1, c_0\}$ with c_0, c_1, c_2, and c_3 defined as

$$c_0 = a_0b_0 + a_3b_1 + a_2b_2 + a_1b_3 \tag{3.2}$$

$$c_1 = a_1b_0 + a_0b_1 + a_3b_2 + a_2b_3 \tag{3.3}$$

$$c_2 = a_2b_0 + a_1b_1 + a_0b_2 + a_3b_3 \tag{3.4}$$

$$c_3 = a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \tag{3.5}$$

A pictorial representation of the cyclic convolution operation of two 4-point sequences, each point consisting of 8 bits is shown in figure 3.7. In this figure each point is represented by +, the individual bits of each point by x, the product of two points by $\otimes$, and each point of the cyclic convolution by Ø. Note that the product of two x's gives another x and this operation is achieved by a two-input AND gate. Now, instead of computing each product, i.e. evaluating $\otimes$, and then adding the four products to obtain a point of the cyclic convolution, we can line up all the partial products of the four points and then add them simultaneously using a CSA tree implementation. This way, we would need only one CPA for each point of the cyclic convolution. We refer to such a method of computation as traditional. The CSA tree is reduced based on the rules given in [54]. The list L for $\otimes$, a 8 by 8 multiplication is given as {1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1}. Since we are trying to add the partial products of four multiplications, each element of this list has to be multiplied by 4. Applying this list to algorithm 3.1, the following results are obtained:

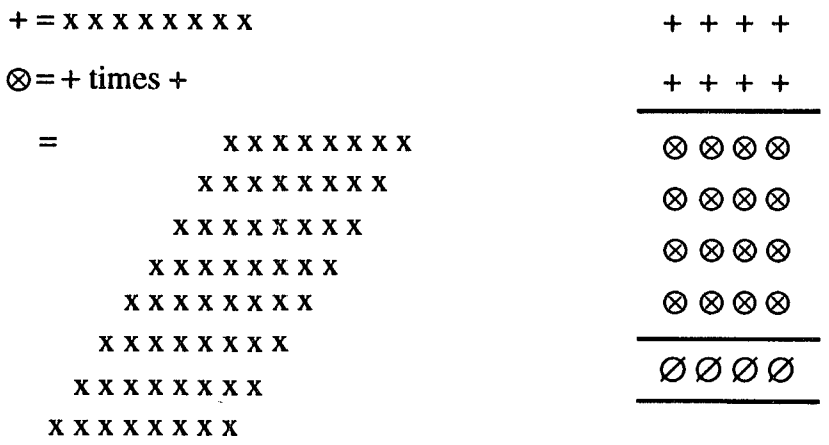


Figure 3.7: Pictorial representation of a 4-point cyclic convolution

L = {4, 8, 12, 16, 20, 24, 28, 32, 28, 24, 20, 16, 12, 8, 4}

# of Full Adders = 222

# of Half Adders = 13

# of Full Adders including CPA = 238

# of Half Adders including CPA = 14

# of CSA Levels = 8

# of CPA Levels = 1

Size of CPA = 17

Number of 2-input gates including CPA = 1218

Since there are four points the total number of two input gates is given as 4872 while the time delay remains as $8\times D_{CSA} + 1\times D_{CPA}$. This is because we are performing the computation of all four points in parallel. Also, each bit of the partial products requires a two input gate. Since each multiplication operation consists of 8 partial products each with 8 bits, the number of two input gates required to obtain these bits is equal to 64. Since there are 16 multiplications the total number of two input gates required to compute these bits is equal to 1024. The time delay to compute these bits is the delay of one two input AND gate, however, this delay is ignored as no matter which method is used it always exists. The results are summarized in the following two equations:

$$\text{Hardware, 4T8} = 5896 \tag{3.6}$$

$$\text{Time Delay, 4T8} = 8\times D_{CSA} + 1\times D_{CPA} \tag{3.7}$$

3.4.2 4-point cyclic convolution-modular

As the problem size becomes larger, i.e. both the number of points and the size of each point increases, it may not be possible to add the partial products of all the multiplication operations simultaneously as done in the previous section. Therefore in

equations (3.2)-(3.5) each of the 16 products are first computed or in other words, referring to figure 3.7 each of the ⊗'s is evaluated. Then each point of the cyclic convolution is obtained by adding its four associated ⊗'s. We refer to computation based on such a method as modular. This addition is again done using a CSA tree implementation. Thus, clearly there are two CPA delays, one for evaluating ⊗ and the other for evaluating Ø. Also there are some CSA delays that are associated with the computation of ⊗ and Ø. To compute hardware and delays associated with the computation of each of the 16 multiplication operations, ⊗, the list L = {1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1} is applied to algorithm 3.1. The following results are obtained:

L = {1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1}

# of Full Adders = 35

# of Half Adders = 7

# of Full Adders including CPA = 49

# of Half Adders including CPA = 8

# of CSA Levels = 4

# of CPA Levels = 1

Size of CPA = 15

Number of 2-input gates including CPA = 261

Since there are 16 such multiplications, the number of gates is 4176. Each one of these multiplication operations produces a result that can be at most 16 bits long. Four such results are added to obtain one point of the cyclic convolution. The hardware and delay associated with such a computation can be obtained by applying the list L = {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4} to algorithm 3.1. The following results are obtained:

L = {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}

# of Full Adders = 30

# of Half Adders = 2

# of Full Adders including CPA = 46

# of Half Adders including CPA = 3

# of CSA Levels = 2

# of CPA Levels = 1

Size of CPA = 17

Number of 2-input gates including CPA = 236

Since there are four points the total number of two input gates is 944. Also, as explained before we need an additional number of two input gates to generate the bits of the partial products. Thus the total hardware and speed delays associated with the modular approach can be summarized as

$$\text{Hardware, 4M8} = 6144 \tag{3.8}$$

$$\text{Time Delay, 4M8} = 6\times D_{CSA} + 2\times D_{CPA} \tag{3.9}$$

3.4.3 4-point cyclic convolution-squares

In this section we estimate the hardware cost and speed required based on our method. Since the points are expressed in the two's complement form the CSA tree implementation can be designed with minor modifications [60]. These modifications will require no additional cost and thus the negative sign in the equations can be treated as a positive sign and the design is carried out as usual. Our method outlined in chapter 2, is by nature modular, i.e. the design is broken into several small parts. Essentially the following computations have to be performed: $c_0 + c_2$, $c_1 + c_3$, $c_0 - c_2$, $c_1 - c_3$. As explained in chapter 2, the first two computations constitute part 1 of our methodology while the later two constitute part 2. Part 1 can be evaluated in three different ways while there is only a singular way for part 2. In chapter 2 we compared the three methods of part 1 based on ROM implementations. However, here we are interested in

CSA based implementations. For now assuming we use method 2 for part 1, then based on theorems (2.3)-(2.6) developed in Chapter 2, the following equations are defined.

$$x_{41} = a_0 + a_2 + b_0 + b_2 \quad (3.10)$$

$$x_{42} = a_0 + a_2 - b_0 - b_2 \quad (3.11)$$

$$x_{43} = a_1 + a_3 + b_1 + b_3 \quad (3.12)$$

$$x_{44} = a_1 + a_3 - b_1 - b_3 \quad (3.13)$$

$$y_{41} = a_0 + a_2 + b_1 + b_3 \quad (3.14)$$

$$y_{42} = a_0 + a_2 - b_1 - b_3 \quad (3.15)$$

$$y_{43} = a_1 + a_3 + b_0 + b_2 \quad (3.16)$$

$$y_{44} = a_1 + a_3 - b_0 - b_2 \quad (3.17)$$

$$z_{41} = a_0 - a_2 + b_0 - b_2 \quad (3.18)$$

$$z_{42} = a_0 - a_2 - b_0 + b_2 \quad (3.19)$$

$$z_{43} = a_1 - a_3 + b_1 - b_3 \quad (3.20)$$

$$z_{44} = a_1 - a_3 - b_1 + b_3 \quad (3.21)$$

$$z_{45} = a_0 - a_2 + b_1 - b_3 \quad (3.22)$$

$$z_{46} = a_0 - a_2 - b_1 + b_3 \quad (3.23)$$

$$z_{47} = a_1 - a_3 + b_0 - b_2 \quad (3.24)$$

$$z_{48} = a_1 - a_3 - b_0 + b_2 \quad (3.25)$$

The amount of hardware required for each of these equations can be obtained by applying the list L = {4, 4, 4, 4, 4, 4, 4, 4} to algorithm 3.1. Note that we are adding four operands each of length eight bits. Thus the height of each column is equal to 4. The following results are obtained:

L = {4, 4, 4, 4, 4, 4, 4, 4}

# of Full Adders = 14

# of Half Adders = 2

# of Full Adders including CPA = 22

# of Half Adders including CPA = 3

# of CSA Levels = 2

# of CPA Levels = 1

Size of CPA = 9

Number of 2-input gates including CPA = 116

Since there are 16 such equations, we have a total of 1856 two-input gates. Now suppose we were to use theorems 2.1 and 2.2 for evaluating part 1. Then instead of the 8 equations given by (3.10)-(3.17) we would have four equations defined by (2.23)-(2.26). The amount of hardware required for each of these equations can be obtained by applying the list L = {8, 8, 8, 8, 8, 8, 8, 8} to algorithm 3.1, which gives the total number of 2-input gates including the CPA as 275. Thus for four equations we would need 1100 2-input gates. Since 1100 (required by method 1) is greater than 928 (required by method 2), we can conclude that method 2 is better. A similar argument can be constructed for method 3. We must also note that the terms of methods 1 and 3 contain more bits than method 2 which in turn implies that computation of their squares will also require more hardware. Therefore from now on we will confine ourselves to evaluating part 1 of our methodology based on method 2.

The points of the cyclic convolution are given, based on theorems (2.3)-(2.6) by

$$c_0 = \frac{1}{8}(x_{41}^2 - x_{42}^2 + x_{43}^2 - x_{44}^2 + z_{41}^2 - z_{42}^2 - z_{43}^2 + z_{44}^2) \tag{3.26}$$

$$c_1 = \frac{1}{8}(y_{41}^2 - y_{42}^2 + y_{43}^2 - y_{44}^2 + z_{45}^2 - z_{46}^2 + z_{47}^2 - z_{48}^2) \tag{3.27}$$

$$c_2 = \frac{1}{8}(x_{41}^2 - x_{42}^2 + x_{43}^2 - x_{44}^2 - z_{41}^2 + z_{42}^2 + z_{43}^2 - z_{44}^2) \tag{3.28}$$

$$c_3 = \frac{1}{8}(y_{41}^2 - y_{42}^2 + y_{43}^2 - y_{44}^2 - z_{45}^2 + z_{46}^2 - z_{47}^2 + z_{48}^2) \tag{3.29}$$

Each of the terms to be squared in equations (3.26)-(3.29) is of length 10 bits. Thus we first compute the cost and delay of a 10 bit squarer. Let A be a 10 bit number with A = $a_9a_8 \ldots a_0$. Then the square of A is given as

$$\begin{aligned} A^2 = {} & a_9 2^{18} + a_9a_8 2^{18} + a_9a_7 2^{17} + a_8 2^{16} + a_9a_6 2^{16} + a_8a_7 2^{16} + a_9a_5 2^{15} + a_8a_6 2^{15} \\ & + a_7 2^{14} + a_9a_4 2^{14} + a_8a_5 2^{14} + a_7a_6 2^{14} + a_9a_3 2^{13} + a_8a_4 2^{13} + a_7a_5 2^{13} + a_6 2^{12} \\ & + a_9a_2 2^{12} + a_8a_3 2^{12} + a_7a_4 2^{12} + a_6a_5 2^{12} + a_9a_1 2^{11} + a_8a_2 2^{11} + a_7a_3 2^{11} \\ & + a_6a_4 2^{11} + a_5 2^{10} + a_9a_0 2^{10} + a_8a_1 2^{10} + a_7a_2 2^{10} + a_6a_3 2^{10} + a_5a_4 2^{10} + a_8a_0 2^9 \\ & + a_7a_1 2^9 + a_6a_2 2^9 + a_5a_3 2^9 + a_4 2^8 + a_7a_0 2^8 + a_6a_1 2^8 + a_5a_2 2^8 + a_4a_3 2^8 \\ & + a_6a_0 2^7 + a_5a_1 2^7 + a_4a_2 2^7 + a_3 2^6 + a_5a_0 2^6 + a_4a_1 2^6 + a_3a_2 2^6 + a_4a_0 2^5 \\ & + a_3a_1 2^5 + a_2 2^4 + a_3a_0 2^4 + a_2a_1 2^4 + a_2a_0 2^3 + a_1 2^2 + a_1a_0 2^2 + a_0 \end{aligned} \tag{3.30}$$

Each of the product terms a_ia_j can be obtained using a two-input AND gate. Since there are 10 bits, the number of gates required is $\binom{10}{2}$ or 45. We apply the manipulation outlined in section 3.3 before we square the number. Such a manipulation does not yield hardware savings, however, it results in a compact array of summands thus allowing the application of the rules of [54] more effectively. Also, this manipulation adds a small cost by increasing the number of summands, in this case by 9. Thus each term to be squared requires 54 two-input gates and the total for all 16 squares is 864. The cost and delay associated with the computation of such a square can be obtained by applying the list L = {1, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1} to algorithm 3.1. The results are:

L = {1, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1}

# of Full Adders = 20

# of Half Adders = 7

# of Full Adders including CPA = 39

# of Half Adders including CPA = 8

# of CSA Levels = 3

# of CPA Levels = 1

Size of CPA = 20

Number of 2-input gates including CPA = 211

Since there are 16 such squares, the total number of 2-input gates can be given as 3376. Finally, equations (3.26)-(3.29) have to be evaluated. From these equations we find that there are basically only four terms that have to be computed. These terms are

$$P = x_{41}^2 - x_{42}^2 + x_{43}^2 - x_{44}^2 \quad (3.31)$$

$$Q = y_{41}^2 - y_{42}^2 + y_{43}^2 - y_{44}^2 \quad (3.32)$$

$$R = z_{41}^2 - z_{42}^2 - z_{43}^2 + z_{44}^2 \quad (3.33)$$

$$S = z_{45}^2 - z_{46}^2 + z_{47}^2 - z_{48}^2 \quad (3.34)$$

Then equations (3.26) - (3.29) can be rewritten as

$$c_0 = 1/8(P + R) \quad (3.35)$$

$$c_1 = 1/8(Q + S) \quad (3.36)$$

$$c_2 = 1/8(P - R) \quad (3.37)$$

$$c_3 = 1/8(Q - S) \quad (3.38)$$

The cost and delay required to compute each of equations (3.31)-(3.34) can be obtained by applying the list L = {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4} to algorithm 3.1. Note that although the size of the CPA required in the previous computation was 20 bits, we know that there will be no carry-out as the square of a 10 bit number can be no more than 20 bits. Thus we are adding four 20-bit numbers and therefore the height of each column is 4. The results are:

L = {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}

# of Full Adders = 38

# of Half Adders = 2

# of Full Adders including CPA = 58

# of Half Adders including CPA = 3

# of CSA Levels = 2

# of CPA Levels = 1

Size of CPA = 21

Number of 2-input gates including CPA = 296

Since there are four such equations, the total number of gates required is 1184. Finally equations (3.35)-(3.38) have to be evaluated. Each one of these equations requires a CPA of size 22 bits, or 21 full adders and 1 half adder, giving a total of 107 gates. Four CPAs would therefore require 428 two-input gates. In summary

$$\text{Hardware, 4S8} = 7708 \tag{3.39}$$

$$\text{Time Delay, 4S8} = 7\times D_{CSA} + 4\times D_{CPA} \tag{3.40}$$

To summarize briefly, we observe that our method based on squaring operations, in the case of 4-point cyclic convolution neither achieves hardware savings nor gain in speed of computation. However, we note that there was no savings in the number of operations to begin with.

3.4.4 8-point cyclic convolution-traditional

The cyclic convolution of two eight point sequences is now considered. Let the two sequences be A and B with A = $\{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\}$ and B = $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. Let each of these points be of length 8 bits and represented in two's complement form. A traditional implementation would require the computation of 64 products, i.e. every point of a sequence is multiplied with every point of the other

sequence. The cyclic convolution C is given by C = $\{c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0\}$ with $c_0, ..., c_7$ defined as

$$c_0 = a_0b_0 + a_7b_1 + a_6b_2 + a_5b_3 + a_4b_4 + a_3b_5 + a_2b_6 + a_1b_7 \quad (3.41)$$

$$c_1 = a_1b_0 + a_0b_1 + a_7b_2 + a_6b_3 + a_5b_4 + a_4b_5 + a_3b_6 + a_2b_7 \quad (3.42)$$

$$c_2 = a_2b_0 + a_1b_1 + a_0b_2 + a_7b_3 + a_6b_4 + a_5b_5 + a_4b_6 + a_3b_7 \quad (3.43)$$

$$c_3 = a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 + a_7b_4 + a_6b_5 + a_5b_6 + a_4b_7 \quad (3.44)$$

$$c_4 = a_4b_0 + a_3b_1 + a_2b_2 + a_1b_3 + a_0b_4 + a_7b_5 + a_6b_6 + a_5b_7 \quad (3.45)$$

$$c_5 = a_5b_0 + a_4b_1 + a_3b_2 + a_2b_3 + a_1b_4 + a_0b_5 + a_7b_6 + a_6b_7 \quad (3.46)$$

$$c_6 = a_6b_0 + a_5b_1 + a_4b_2 + a_3b_3 + a_2b_4 + a_1b_5 + a_0b_6 + a_7b_7 \quad (3.47)$$

$$c_7 = a_7b_0 + a_6b_1 + a_5b_2 + a_4b_3 + a_3b_4 + a_2b_5 + a_1b_6 + a_0b_7 \quad (3.48)$$

Now, as outlined in section 3.4.1, instead of computing each product and then adding the eight products to obtain a point of the cyclic convolution, we can line up all the partial products of the eight points and then add them simultaneously using a CSA tree implementation. This way, we would need only one CPA for each point of the cyclic convolution. The list L for a 8 by 8 multiplication is given as {1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1}. Since we are trying to add the partial products of eight multiplications, each element of this list has to be multiplied by 8. Applying this list to algorithm 3.1, the following results are obtained:

L = {8, 16, 24, 32, 40, 48, 56, 64, 56, 48, 40, 32, 24, 16, 8}

# of Full Adders = 476

# of Half Adders = 18

# of Full Adders including CPA = 493

# of Half Adders including CPA = 19

# of CSA Levels = 10

# of CPA Levels = 1

Size of CPA = 18

Number of 2-input gates including CPA = 2503

Since there are eight points the total number of two input gates is given as 20,024 while the time delay remains as $10\times D_{CSA} + 1\times D_{CPA}$. This is because we are performing the computation of all eight points in parallel. Also, each bit of the partial products requires a two input gate. Since each multiplication operation consists of 8 partial products each with 8 bits, the number of two input gates required to obtain these bits is equal to 64. Since there are 64 multiplications the total number of two input gates required to represent these bits is equal to 4096. The time delay to compute these bits is the delay of one two-input AND gate, however, again this delay is ignored as no matter which method is used it always exists. The results are summarized in the following two equations:

$$\text{Hardware, 8T8} = 24120 \tag{3.49}$$

$$\text{Time Delay, 8T8} = 10\times D_{CSA} + 1\times D_{CPA} \tag{3.50}$$

3.4.5 8-point cyclic convolution-modular

As the problem size becomes larger, i.e. both the number of points and the size of each point increases, it may not be possible to add the partial products of all the multiplication operations simultaneously. Therefore in equations (3.41)-(3.48) each of the 64 products are first computed. Then each point is evaluated by adding its eight associated operands. This addition is again done using a CSA tree implementation. Thus, clearly there are two CPA delays and some CSA delays which are determined as follows. Since each of the 64 multiplications are of the same size as that in section 3.4.2, hardware and delays associated with the computation are the same as that estimated earlier. This cost is therefore 261 two-input gates for each multiplication for a total of 16704 and a time delay of $4\times D_{CSA} + 1\times D_{CPA}$ with a CPA size of 15. Each

one of these multiplication operations produces a result that can be at most 16 bits long. Eight such results are added to obtain one point of the cyclic convolution. The hardware and delay associated with such a computation can be obtained by applying the list L = {8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8} to algorithm 3.1. The following results are obtained:

L = {8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8}

# of Full Adders = 92

# of Half Adders = 4

# of Full Adders including CPA = 109

# of Half Adders including CPA = 5

# of CSA Levels = 4

# of CPA Levels = 1

Size of CPA = 18

Number of 2-input gates including CPA = 555

Since there are eight points the number of two-input gates is 4440. Also, as explained before we need an additional 4096 two-input gates to generate the bits of the partial products. Thus the total hardware and speed delays associated with the modular approach can be summarized as

$$\text{Hardware, 8M8} = 25240 \tag{3.51}$$

$$\text{Time Delay, 8M8} = 8 \times D_{CSA} + 2 \times D_{CPA} \tag{3.52}$$

We must note that such a modular approach has, in all the steps limited the height of the tallest column in any CSA tree to a maximum of 8. Thus in order to ensure a fair comparison we must make sure that our new proposed methods do not involve steps that require CSA trees whose columns are much taller.

3.4.6 8-point cyclic convolution-squares

In this section we estimate the hardware cost and speed required based on our method, using a total of 48 squares. Our method can be divided into three distinct modules as shown in figures 3.8, 3.9, and 3.10. We do not develop all the equations as the intent in this section is primarily to estimate the hardware cost and speed. However, we list all the steps in each module and their associated costs and delays.

Module 1: *Computes* $(c_0 + c_4)$, $(c_2 + c_6)$, $(c_1 + c_5)$, $(c_3 + c_7)$

Step 1:

a) Use theorem 2.3 to generate terms x_{81} through x_{84}.

b) Use theorem 2.4 to generate terms y_{81} through y_{84}.

c) Use theorem 2.5 to generate terms z_{81} through z_{84}.

d) Use theorem 2.6 to generate terms z_{85} through z_{88}.

Each of the above 16 terms are of the same size and also have an identical structure, i.e. they are formed by adding/subtracting 8 points of the input sequences. The hardware and delay associated with the computation of these terms can be estimated by applying the list L = {8, 8, 8, 8, 8, 8, 8, 8} to algorithm 3.1. Note that we are adding eight 8-bit numbers. Thus the height of each column is equal to 8. The results are:

L = {8, 8, 8, 8, 8, 8, 8, 8}

# of Full Adders = 44

# of Half Adders = 4

# of Full Adders including CPA = 53

# of Half Adders including CPA = 5

# of CSA Levels = 4

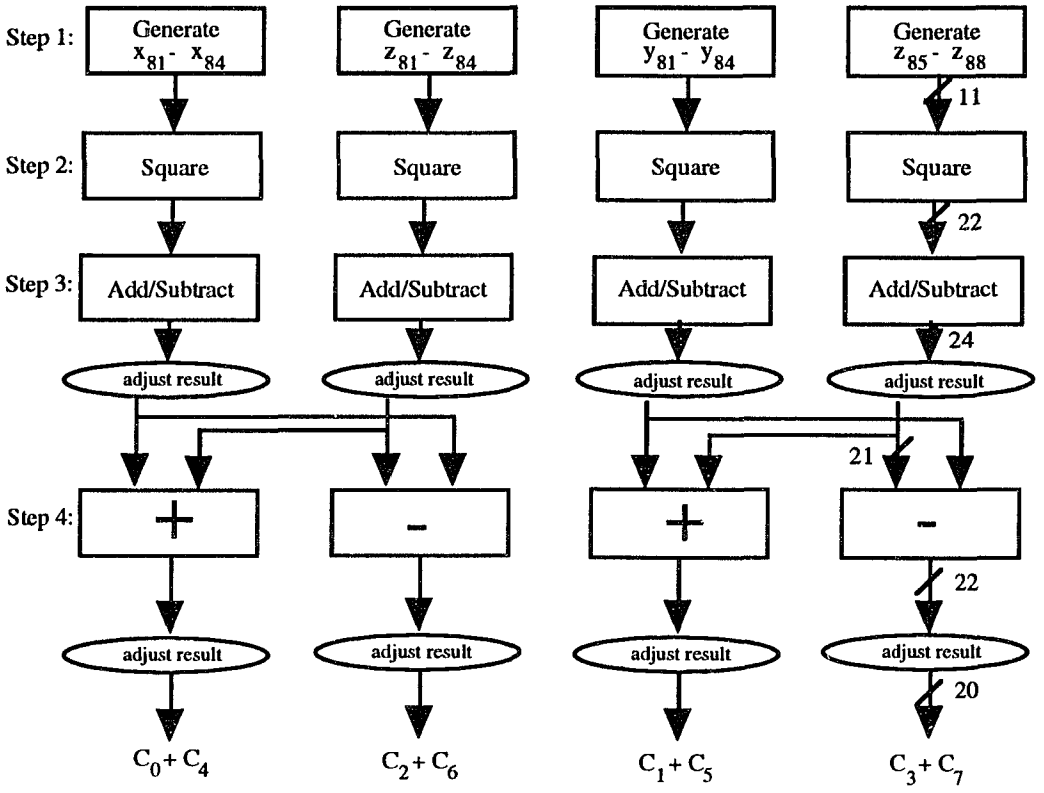


Figure 3.8: 8-point cyclic convolution-module 1

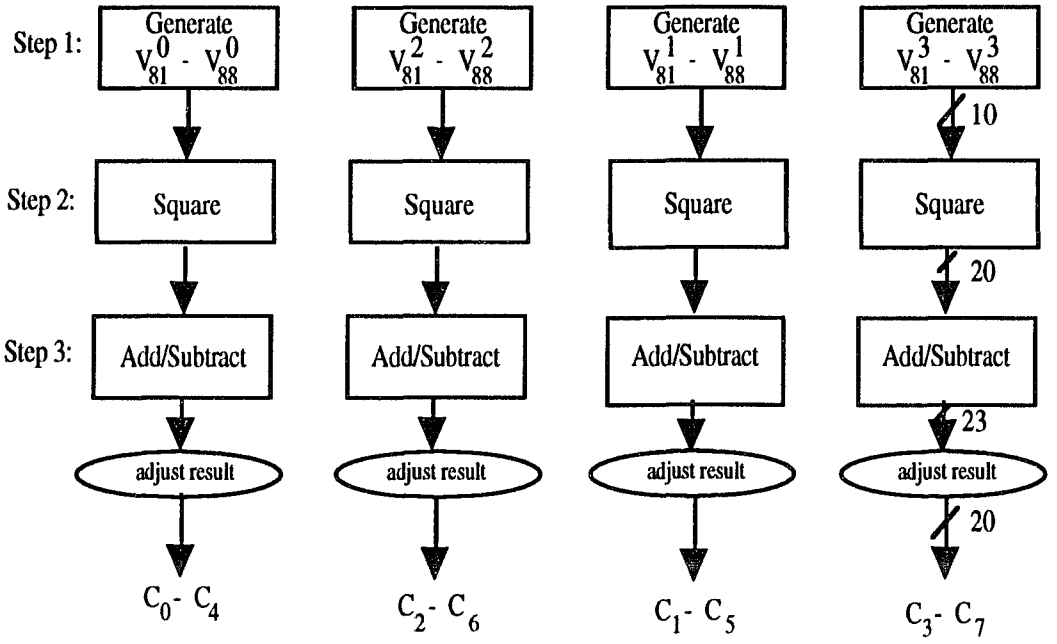


Figure 3.9: 8-point cyclic convolution-module 2

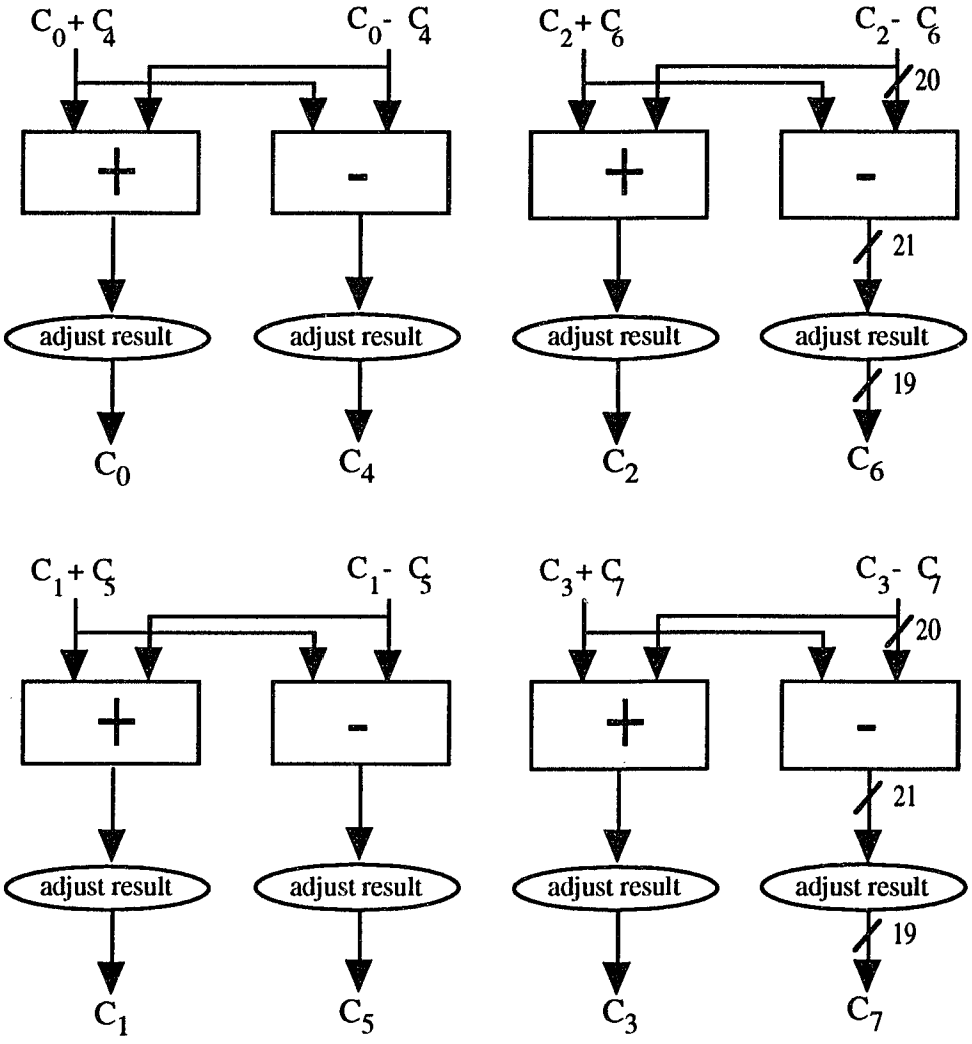


Figure 3.10: 8-point cyclic convolution-module 3

# of CPA Levels = 1

Size of CPA = 10

Number of 2-input gates including CPA = 275

Since there are 16 such equations, we have a total of 4400 two-input gates. Results of step 1 can be summarized as

$$\text{Step 1, hardware} = 4400 \tag{3.52}$$

$$\text{Step 1, time delay} = 4 \times D_{CSA} + 1 \times D_{CPA} \tag{3.53}$$

We note that the height of the tallest column in this step is 8.

Step 2:

Each of the terms obtained in step 1 is of length 11 bits and needs to be squared, resulting in a term that can have at most 22 bits. The cost and delay associated with the computation of the square can be obtained by applying the list L = {1, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1}, to algorithm 3.1. The results are:

{1, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1}

# of Full Adders = 27

# of Half Adders = 9

# of Full Adders including CPA = 48

# of Half Adders including CPA = 10

# of CSA Levels = 3

# of CPA Levels = 1

Size of CPA = 22

Number of 2-input gates including CPA = 260

Since there are 16 such squares, the total number of 2-input gates can be given as 4160. Also, each of the 16 terms that need to be squared have $\binom{11}{2} + 10 = 65$ input bits of the

form $a_i a_j$, thus requiring a total of 1040 two-input AND gates. Results of step 2 can be summarized as

$$\text{Step 2, hardware} = 5200 \tag{3.54}$$

$$\text{Step 2, time delay} = 3 \times D_{CSA} + 1 \times D_{CPA} \tag{3.55}$$

We note that the height of the tallest column in this step is 6.

Step 3:

The results of step 2 are used to generate $4\sum c_{2i}$, $4\sum c_{2i+1}$, $4\sum c_{2i}(-1)^i$, $4\sum c_{2i+1}(-1)^i$ based on equations (2.33), (2.38), (2.43), and (2.48). This is achieved by grouping the 16 squares into sets of four and adding/subtracting the terms. The hardware and delay associated with this computation is obtained by applying the list L = {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}. Note that we are adding four 22-bit numbers. The results are:

L = {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}

# of Full Adders = 42

# of Half Adders = 2

# of Full Adders including CPA = 64

# of Half Adders including CPA = 3

# of CSA Levels = 2

# of CPA Levels = 1

Size of CPA = 23

Number of 2-input gates including CPA = 326

Since there are four such groups, the total is 1304. Results of step 3 can be summarized as

$$\text{Step 3, hardware} = 1304 \tag{3.56}$$

$$\text{Step 3, time delay} = 2 \times D_{CSA} + 1 \times D_{CPA} \tag{3.57}$$

We note that the height of the tallest column in this step is 4. The output of the CPAs at this step have a length of 24 bits. However, since we know that the result is four times the desired value, the last two bits of this computation are zero and can hence be ignored to divide by four. Also, since each of $\sum c_{2i}$, $\sum c_{2i+1}$, $\sum c_{2i}(-1)^i$, $\sum c_{2i+1}(-1)^i$ is the sum of four points of the cyclic convolution, their length cannot be greater than $16 + \log_2 32 = 21$. Thus the most significant bit (MSB) is also stripped.

Step 4:

The following computations now need to be performed:

$$\sum c_{2i} + \sum c_{2i}(-1)^i = 2(c_0 + c_4)$$

$$\sum c_{2i} - \sum c_{2i}(-1)^i = 2(c_2 + c_6)$$

$$\sum c_{2i+1} + \sum c_{2i+1}(-1)^i = 2(c_1 + c_5)$$

$$\sum c_{2i+1} - \sum c_{2i+1}(-1)^i = 2(c_3 + c_7)$$

Each of these computations requires a CPA of size 21 bits. The cost of such a CPA is equal to 102 two-input gates. Since we have four CPAs the total cost is 408 gates. Results of step 4 can be summarized as

$$\text{Step 4, hardware} = 408 \tag{3.58}$$

$$\text{Step 4, time delay} = 1 \times D_{CPA} \tag{3.59}$$

Again, since we know that the result is twice the desired value, the last bit of this computation is zero and can hence be ignored to divide by two. Also, since each of $\sum c_{4i}$, $\sum c_{4i+1}$, $\sum c_{4i+2}$, $\sum c_{4i+3}$ is the sum of two points of the cyclic convolution, their length cannot be greater than $16 + \log_2 16 = 20$. Therefore the MSB is also stripped. Thus these results are of length 20 bits.

Module 2: *Computes* $(c_0 - c_4)$, $(c_2 - c_6)$, $(c_1 - c_5)$, $(c_3 - c_7)$

Step 1:

a) Use theorem 2.8 to generate terms V_{81}^0 through V_{88}^0.

b) Use theorem 2.8 to generate terms V_{81}^1 through V_{88}^1.

c) Use theorem 2.8 to generate terms V_{81}^2 through V_{88}^2.

d) Use theorem 2.8 to generate terms V_{81}^3 through V_{88}^3.

Each of the above 32 terms are of the same size and also have an identical structure, i.e. they are formed by adding/subtracting 4 points of the input sequences. The hardware and delay associated with the computation of these terms can be estimated by applying the list L = {4, 4, 4, 4, 4, 4, 4, 4} to algorithm 3.1. Note that we are adding four 8-bit numbers. Thus the height of each column is equal to 4. Previously, the hardware cost for such a computation was calculated as 116 two-input gates and the time delay as $2\times D_{CSA} + 1\times D_{CPA}$ with a CPA of size 9 bits. Since there are 32 such terms, we have a total of 3712 two-input gates. Results of step 1 can be summarized as

$$\text{Step 1, hardware} = 3712 \tag{3.60}$$

$$\text{Step 1, time delay} = 2\times D_{CSA} + 1\times D_{CPA} \tag{3.61}$$

We note that the height of the tallest column in this step is 4.

Step 2:

Each of the terms obtained in step 1 is of length 10 bits and needs to be squared, resulting in a term that can have at most 20 bits. The cost and delay associated with the computation of a 10 bit square was previously estimated at 211 two-input gates and a time delay of $3\times D_{CSA} + 1\times D_{CPA}$ with a 20-bit CPA. Since there are 32 such squares, the total number of 2-input gates can be given as 6752. Also, each of the 32 terms that

need to be squared have 54 input bits of the form $a_i a_j$, thus requiring a total of 1728 two-input AND gates. Results of step 2 can be summarized as

$$\text{Step 2, hardware} = 8480 \tag{3.62}$$

$$\text{Step 2, time delay} = 3 \times D_{CSA} + 1 \times D_{CPA} \tag{3.63}$$

We note that the height of the tallest column in this step is 6.

Step 3:

The results of step 2 are used to generate $4\sum c_{4i}$, $4\sum c_{4i+1}$, $4\sum c_{4i} (-1)^i$, $4\sum c_{4i+1} (-1)^i$ based on equation (2.38). This is achieved by grouping the 32 squares into sets of four and adding/subtracting the 8 terms within each set. The hardware and delay associated with this computation is obtained by applying the list L ={8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8} to algorithm 3.1. Note that we are adding eight 20-bit numbers. The results are:

L = {8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8}

# of Full Adders = 116

# of Half Adders = 4

# of Full Adders including CPA = 137

# of Half Adders including CPA = 5

# of CSA Levels = 4

# of CPA Levels = 1

Size of CPA = 22

Number of 2-input gates including CPA = 695

Since there are four such groups, the total is 2780. Results of step 3 can be summarized as

$$\text{Step 3, hardware} = 2780 \tag{3.64}$$

$$\text{Step 3, time delay} = 4 \times D_{CSA} + 1 \times D_{CPA} \tag{3.65}$$

We note that the height of the tallest column in this step is 8. Also since we know that the result is four times the desired value, the last two bits of this computation are zero and can hence be ignored to divide by four. However, since each of $\sum c_{4i} (-1)^i$, $\sum c_{4i+1} (-1)^i$, $\sum c_{4i+2} (-1)^i$, $\sum c_{4i+3} (-1)^i$, is the sum of two points of the cyclic convolution, their length cannot be greater than $16 + \log_2 16 = 20$. Therefore the MSB is also stripped. Thus the length of each of these results is 20 bits.

Module 3: *Computes* $c_0, c_4, c_2, c_6, c_1, c_5, c_3, c_7$

This module adds/subtracts the results of modules 1 and 2 to obtain the points of the cyclic convolution. All the computations are done in a single step with the help of 8 CPAs of size 20 bits. The cost of such a CPA is 97 gates thereby giving a total of 776 gates. Results of this module can be summarized as

$$\text{Step 1, hardware} = 776 \tag{3.66}$$

$$\text{Step 1, time delay} = 1 \times D_{CPA} \tag{3.67}$$

Again, since we know that the result is twice the desired value, the last bit of this computation is zero and can hence be ignored to divide by two. Also, since each point of the cyclic convolution cannot be of length greater than 19 bits, the MSB is also stripped.

In summary, adding the results of equations (3.52), (3.54), (3.56), (3.58), (3.60), (3.62), (3.64), and (3.66) we obtain the hardware cost of all modules as

$$\text{Hardware, 8S8} = 27060 \tag{3.68}$$

Looking at figures (3.8) and (3.9) we observe that these two modules operate in parallel and therefore the module with the higher delay and the delay of module 3 account for the total delay. Thus adding the results of equations (3.53), (3.55), (3.57), (3.59), and (3.67) we obtain the total delay for the computation as

$$\text{Time Delay, 8S8} = 9 \times D_{CSA} + 5 \times D_{CPA} \tag{3.69}$$

3.4.7 16-point cyclic convolution-traditional

Since we have already presented the cyclic convolution of 4 and 8 points in detail, we keep the presentation over here very brief. Each point of the cyclic convolution is obtained by adding simultaneously the summands of 16, 8x8 products. The cost of such a computation is 5066 gates for a total of 81056 gates. The number of gates required for the summands is 256 times 64, or 16384. In summary,

$$\text{Hardware, 16T8} \quad = 97440 \tag{3.70}$$

$$\text{Time Delay, 16T8} \quad = 11\times D_{CSA} + 1\times D_{CPA} \tag{3.71}$$

We note that the height of the tallest column is 128.

3.4.8 16-point cyclic convolution-modular

Each of the 16 products that constitute a single point of the cyclic convolution are first evaluated and then the 16 results are added. The cost of an 8x8 product is 261 gates and since in all there are 256 such products we have a total of 66816 gates. The cost of adding 16 products is 1198 gates and for 16 such computations we have a total of 19168 gates. The number of gates required by the summands is as before, 16384. In summary,

$$\text{Hardware, 16M8} \quad = 102368 \tag{3.72}$$

$$\text{Time Delay, 16M8} \quad = 10\times D_{CSA} + 2\times D_{CPA} \tag{3.73}$$

We note that the height of the tallest column is 16.

3.4.9 16-point cyclic convolution-squares

Module 1 computes $(c_0 + c_4 + c_8 + c_{12})$, $(c_2 + c_6 + c_{10} + c_{14})$, $(c_1 + c_5 + c_9 + c_{13})$, and $(c_3 + c_7 + c_{11} + c_{15})$. The cost and time delay associated with such a computation can be summarized as

Hardware, module 1, 16S8 $= 17552$ (3.74)

Time Delay, module 1, 16S8 $= 11\times D_{CSA} + 4\times D_{CPA}$ (3.75)

Module 2 computes $(c_0 - c_4 + c_8 - c_{12})$, $(c_2 - c_6 + c_{10} - c_{14})$, $(c_1 - c_5 + c_9 - c_{13})$, and $(c_3 - c_7 + c_{11} - c_{15})$. The cost and time delay associated with such a computation can be summarized as

Hardware, 16S8, module 2 $= 23116$ (3.76)

Time Delay, 16S8, module 2 $= 11\times D_{CSA} + 4\times D_{CPA}$ (3.77)

Module 3 computes $(c_0 - c_8)$, $(c_2 - c_{10})$, $(c_4 - c_{12})$, $(c_6 - c_{14})$, $(c_1 - c_9)$, $(c_3 - c_{11})$, $(c_5 - c_{13})$, $(c_7 - c_{15})$. The cost and time delay associated with such a computation can be summarized as

Hardware, 16S8, module 3 $= 61352$ (3.78)

Time Delay, 16S8, module 3 $= 11\times D_{CSA} + 3\times D_{CPA}$ (3.79)

Module 4 computes c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, and c_{15}. The cost and time delay associated with such a computation can be summarized as

Hardware, 16S8, module 4 $= 1632$ (3.80)

Time Delay, 16S8, module 4 $= 1\times D_{CPA}$ (3.81)

In summary,

Hardware, 16S8 $= 103652$ (3.82)

Time Delay, 16S8 $= 11\times D_{CSA} + 6\times D_{CPA}$ (3.83)

3.4.10 Discussion

Table 3.1 summarizes the hardware cost in 2-input gates for computing the cyclic convolution of 4, 8, and 16 points based on all three methods. At first glance it

Table 3.1: Hardware cost in 2-input gates for cyclic convolution of 4, 8, and 16 points

Method	4 points	8 points	16 points
Traditional	5896	24120	97440
Modular	6144	25240	102368
Squares	7708	27060	103652

appears that our method is at best competitive. But this is not the case for several reasons.

1) In the traditional implementation there are basically no modules. Therefore the silicon area of the chip is directly a function of the height of the tallest column in the CSA tree. The height of the tallest column is the product of the number of points in the sequences to be convolved and the word length of each point. For instance, with a word length of 8 bits per point, in the case of four points the height is 4 times 8 or 32, for 8 points it is 8 times 8 or 64, and for 16 points it is 16 times 8 or 128. Clearly as the number of points increases the height increases. If in a given technology, this height can be managed then the traditional method is the best method of implementation. On the other hand if the problem has to be broken into smaller components then one must have a systematic way of doing so. The modular approach is one method and our approach based on squares is the other. Thus in the event the traditional implementation is not feasible one might consider implementations based on these methods and thus the comparison is restricted to these two methods, modular and squares.

2) Looking at table 3.1 again, we find that in the case of 4 points our method based on squares is worse than the modular by 25%, in the case of 8 points by 7%, and in the case of 16 points by 1%. However, from table 2.1, we see that the savings in squaring operations in the three cases is 0%, 25%, and 31.25%. Thus it is only reasonable for us to speculate that with an increase in the number of points being convolved the hardware savings will increase.

3) Our main purpose of all the preceding analysis was to show that in spite of the increase in the number of additions caused by the use of our methodology, the decrease in squaring operations will be more beneficial. Now, here is the surprise. All the

preceding implementations were 100% parallel. In other words they used many more addition operations than that given by equation (2.69). This is because, in the construction of our equations, that need to be squared, there are several common terms. But for a parallel implementation these common terms are evaluated as many times as they are needed, thus increasing the hardware. By the same token, for the modular approach, there are no common terms in the first place and therefore there is no question of redundant computation. Thus our implementation model is very kind to the modular approach, in the sense that it is ideally suited to that approach. Thus it is only fair for us to conclude that if an implementation structure that is capable of exploiting the properties of our method is selected, then the savings in squaring operations will pay off.

4) One might argue as to why another model that would be more suitable to our approach was not selected. However, if one has to be fair to all methods, then the selection of such a model would be difficult if not impossible. Therefore we chose the worst case scenario for our method which at the same time is best case scenario for the modular method and have shown that in spite of being against all odds, we are at the least competitive.

5) With regard to the speed of computation, for the 100% parallel implementation, referring to table 3.2 we can see that our method is the slowest. However, our approach has a lot of other properties that can be exploited by a clever architecture. To illustrate, looking at figure 3.8 one can see that the four CPAs used in step 3 can also be used for step 4 with minor modifications. These minor modifications require negligible additional hardware and at the same time do not slow down the operations. Similarly we observe that the hardware required by module 3 is redundant as the same can be achieved by CPAs in step 3 of modules 1 and 2. Thus by simple modifications to the architecture we can reduce the hardware costs. Our method thus also provides for alternate

Table 3.2: Time delay of cyclic convolution of 4, 8, and 16 points

Method	4 points	8 points	16 points
Traditional	$8D_{CSA} + 1D_{CPA}$	$10D_{CSA} + 1D_{CPA}$	$11D_{CSA} + 1D_{CPA}$
Modular	$6D_{CSA} + 2D_{CPA}$	$8D_{CSA} + 2D_{CPA}$	$10D_{CSA} + 2D_{CPA}$
Squares	$7D_{CSA} + 4D_{CPA}$	$9D_{CSA} + 5D_{CPA}$	$11D_{CSA} + 6D_{CPA}$

implementations while such is not the case with either the traditional or modular approach. Further, the fine granularity of our approach makes it an ideal candidate for incorporating several sophisticated methods, for example, pipe-lining and systolic arrays, to improve the speed of the computation. However, while employing these and other techniques one must also account for the interconnection delays. Analysis in this area is highly application dependent. For instance since the convolution operation is quite often required to be performed in real time, several different sequences have to be processed one after the other. Thus, the throughput of the model should not only include the number of points processed per cycle but also include the number of sequences processed per cycle.

In summary, while we have not provided an actual factor for the amount of savings in CSA based implementations, we have, however, convincingly demonstrated with the help of the preceding sections that in spite of an increase in the number of two-operand additions our method produces efficient designs. In the next section we consider hybrid implementations, i.e. we implement the squaring and multiplication operations using ROMs and show that in such a case our method yields phenomenal hardware savings in spite of using an unkind model.

3.5 Hybrid implementation of cyclic convolution

In a hybrid implementation of the cyclic convolution operation, we substitute the CSA implementation of the squaring or multiplication operation by a ROM implementation. The rest of the implementation is left unchanged. We first present a ROM model and derive its cost and speed functions. We then use these functions to estimate the cost and speed of the hybrid implementations of the cyclic convolution

operation. Since the traditional approach involves no multiplication operation (the partial products were simply added) there exists no hybrid implementation for that approach.

ROM Hardware Cost: The ROM can be designed based on the model given in [55]. In such a model the address lines are split into two halves called the X and Y sections. The address lines in X and Y can then be decoded simultaneously. Let the ROM be of size $2^L \times n$, where L is the number of address lines and n is the number of outputs. If there are L address lines then each half has L/2 address lines with $2^{L/2}$ minterms. In each of these decoders every minterm requires $(2^0 + 2^1 + ... + 2^{(\log_2 L/2)} {}^{-1})$ 2-input gates. The outputs of the diode matrix are logically ANDed with the outputs of the Y decoder before being multiplexed to form the final output. The number of gates required for this purpose is $(2^0 + 2^1 + ... + 2^{(\log_2 2^{**}(L/2)})$ 2-input gates. Adding and simplifying we have,

$$\text{ROM, Hardware} = 2^{L/2}(2n + L - 2) - n \tag{3.84}$$

In the above we note that each of the $2^{L/2}$ minterms are realized independently. However, we assume that each of these minterms is realized only once for every output of the ROM. Also we ignore the cost and delay of the diode matrix.

ROM Time delay: The delay of the such a model is given by [55] in gate delays as

$$\text{ROM, Delay} = 2 + \log_2 (L/2) + \log_2 2^{(L/2)} \tag{3.85}$$

The above computation assumes that the X and Y decoders operate in parallel and the input line from the diode-matrix is multiplexed at the outputs of the Y decoder. We note that this model does not account for several unique characteristics of ROM implementations [55]. However, the purpose of using this model is to provide a more meaningful hardware cost in terms of gates as opposed to ROM bits and also to estimate the time delay of the overall implementation.

3.5.1 8-point cyclic convolution-hybrid, modular

In section 3.4.2, the results of the 64 multiplications were obtained by using CSA tree implementations. Instead if we were to use ROMs for the multiplication operations then we would need 64 ROMs, each of size $2^{16}\times 16$ or a total of 67,108,864 ROM bits. In terms of gates, from equation (3.84) we get the number of 2-input gates as 752640. The time delay of this ROM is computed from equation (3.85) as 13 gate delays. Over and above this we would need 4440 two-input gates for adding the results of the ROMs with a time delay of $4\times D_{CSA} + 1\times D_{CPA}$. In summary,

$$\text{Hardware, 8HM8} \quad = 757080 \tag{3.86}$$

$$\text{Time Delay, 8HM8} \quad = 13 + 4\times D_{CSA} + 1\times D_{CPA} \tag{3.87}$$

3.5.2 8-point cyclic convolution-hybrid, squares

In section 3.4.6, the results of the 48 squares were obtained by using CSA tree implementations. Instead if we were to use ROMs for the squaring operations then we would need 16 ROMs, each of size $2^{11} \times 22$ and 32 ROMs, each of size $2^{10} \times 20$ or a total of 1,376,256 ROM bits. In terms of gates the count is 86544 2-input gates with a time delay of 10 gates. Over and above this we would need 13380 two-input gates for adding the results of the ROMs with a time delay of $4\times D_{CSA} + 3\times D_{CPA}$. In summary,

$$\text{Hardware, 8HS8} \quad = 99924 \tag{3.88}$$

$$\text{Time Delay, 8HS8} \quad = 10 + 6\times D_{CSA} + 4\times D_{CPA} \tag{3.89}$$

Clearly, from equations (3.86)-(3.89), one can see that our method is very attractive. We require approximately one-eighth the gate count of the hybrid-modular method while being slower by about only $3 \times D_{CPA}$.

3.6 Applications to computer arithmetic

The multiplication operation is one of the four basic operations and is used extensively, both in general purpose and special purpose computing. As such there exists a vast amount of literature on the variety of multiplication algorithms, references [47]-[49],[56] to name a few. Recently new multipliers modulo $(2^N -1)$, [51] and modulo $(2^N +1)$, [61] have been developed. Apart from the requirements of computer arithmetic, the fields of digital signal processing and cryptography have several algorithms that perform arithmetic in modular rings [17],[62]. Multipliers designed using look-up tables [46],[47],[51] offer attractive speed-complexity trade offs [61], however their main draw back has been excessive ROM sizes and thus the inability to integrate the entire design on a single chip.

3.6.1 Modulo 2^N -1 multiplication

Consider the multiplication of two N-bit binary numbers A and B. Let each of the numbers be decomposed into four parts given as $\{a_3, a_2, a_1, a_0\}$ and $\{b_3, b_2, b_1, b_0\}$. The number A is then given as $a_3 2^{3N/4} + a_2 2^{2N/4} + a_1 2^{N/4} + a_0$, and number B can be evaluated in a similar fashion. Then their product modulo 2^N -1 can be given as

$$<A \times B>_{2^N -1} = < c_0 + c_1 2^{N/4} + c_2 2^{N/2} + c_3 2^{3N/4} >_{2^N -1} \qquad (3.90)$$

with c_0, c_1, c_2, and c_3 given by equations (3.2)-(3.5). Note that the c_i in equations (3.2)-(3.5) are the terms of cyclic convolution of two four point sequences with the points being a_i and b_i. Thus we can apply the theorems developed in chapter 2 to obtain c_0, c_1, c_2, and c_3. However, over here since our objective is to minimize the total number of ROM bits and not the total number of squaring operations we use theorems (2.3)-(2.6). We then define equations x_{ij}, y_{ij}, and z_{ij} as given by equations (3.10)-

(3.25). Then theorems (2.3)-(2.6) give us $4(c_0 + c_2)$, $4(c_1 + c_3)$, $4(c_0 - c_2)$, $4(c_1 - c_3)$. Finally equations (3.26)-(3.29) give the values of c_0, c_1, c_2, and c_3.

Each square required by equations (3.26)-(3.29) is realized using a ROM. The advantages of such techniques are detailed in references [47]-[51]. Although the number of squares required is more than that of [51] the total number of ROM bits required is less than that required by [51]. Hardware in terms of adders and subtracters is comparable with that required by [51]. Section 3.6.4 offers a detailed comparative analysis.

3.6.2 Extending the modulo 2^N -1 multiplier

Continuing with the same notation as before, the modulo $2^N + 1$ product of two numbers A and B can be given as

$$<A \times B>_{2^N+1} = < d_0 + d_1 2^{N/4} + d_2 2^{N/2} + d_3 2^{3N/4} >_{2^N+1} \qquad (3.91)$$

with d_0, d_1, d_2, and d_3 defined as

$$d_0 = a_0b_0 - a_3b_1 - a_2b_2 - a_1b_3 \qquad (3.92)$$

$$d_1 = a_1b_0 + a_0b_1 - a_3b_2 - a_2b_3 \qquad (3.93)$$

$$d_2 = a_2b_0 + a_1b_1 + a_0b_2 - a_3b_3 \qquad (3.94)$$

$$d_3 = a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \qquad (3.95)$$

Note that term d_3 of equation (3.95) is the same as term c_3 of equation (3.5) and so no extra ROM bits are required for computing d_3. To compute d_0, d_1, and d_2 define

$$g_0 = a_3b_1 + a_2b_2 + a_1b_3 \qquad (3.96)$$

$$g_1 = a_3b_2 + a_2b_3 \qquad (3.97)$$

$$g_2 = a_3b_3 \qquad (3.98)$$

Then

$$d_0 = c_0 - 2g_0 \qquad (3.99)$$

$$d_1 = c_1 - 2g_1 \tag{3.100}$$

$$d_2 = c_2 - 2g_2 \tag{3.101}$$

The terms g_0, g_1, and g_2 can be computed by directly applying the quarter squared algorithm [47]-[51] while these equations are not presented here. Doubling these terms can be obtained by simply shifting the numbers to the left by one position and thus this needs no extra ROM bits.

The product of two numbers A and B modulo 2^N can be given as

$$<A \times B>_{2^N} = < e_0 + e_1 2^{N/4} + e_2 2^{N/2} + e_3 2^{3N/4} >_{2^N} \tag{3.102}$$

with e_0, e_1, e_2, and e_3 defined as

$$e_0 = a_0 b_0 \tag{3.103}$$

$$e_1 = a_1 b_0 + a_0 b_1 \tag{3.104}$$

$$e_2 = a_2 b_0 + a_1 b_1 + a_0 b_2 \tag{3.105}$$

$$e_3 = a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 \tag{3.106}$$

Again, term e_3 of equation (3.106) is the same as term c_3 of equation (3.5) and so no extra ROM bits are needed for this computation. The other terms can also be obtained without the expense of any more ROM bits by using the following equations:

$$e_0 = (c_0 + d_0) \times 1/2 \tag{3.107}$$

$$e_1 = (c_1 + d_1) \times 1/2 \tag{3.108}$$

$$e_2 = (c_2 + d_2) \times 1/2 \tag{3.109}$$

The full precision product of two integer numbers A and B can be given as

$$A \times B = f_0 + f_1 2^{N/4} + f_2 2^{N/2} + f_3 2^{3N/4} + f_4 2^N + f_5 2^{5N/4} + f_6 2^{6N/4} \tag{3.110}$$

with

$$f_0 = a_0 b_0 = e_0 \tag{3.111}$$

$$f_1 = a_1 b_0 + a_0 b_1 = e_1 \tag{3.112}$$

$$f_2 = a_2 b_0 + a_1 b_1 + a_0 b_2 = e_2 \tag{3.113}$$

$$f_3 = a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 = e_3 \quad (3.114)$$

$$f_4 = a_3b_1 + a_2b_2 + a_1b_3 = g_0 \quad (3.115)$$

$$f_5 = a_3b_2 + a_2b_3 = g_1 \quad (3.116)$$

$$f_6 = a_3b_3 = g_2 \quad (3.117)$$

Again, all of these computations need no extra ROM bits.

3.6.3 Example

In this section we present a numerical example to illustrate the various techniques described earlier in this section. Consider two 16-bit numbers A and B with A = 54682 = $(1101010110011010)_2$ and B = 57811 = $(1110000111010011)_2$. We decompose each number into four parts, each with four bits. Thus we have A = {a_3, a_2, a_1, a_0} and B = {b_3, b_2, b_1, b_0} with $a_3 = (1101)_2 = 13$, $a_2 = (0101)_2 = 5$, $a_1 = (1001)_2 = 9$, $a_0 = (1010)_2 = 10$ and $b_3 = (1110)_2 = 14$, $b_2 = (0001)_2 = 1$, $b_1 = (1101)_2 = 13$, $b_0 = (0011)_2 = 3$. Here N = 16 and n = 4.

3.6.3.1 Modulo 2^N -1 product

Equations (3.10)-(3.25) give:

$x_{41} = 19$, $x_{42} = 11$, $x_{43} = 49$, $x_{44} = -5$, $y_{41} = 42$, $y_{42} = -12$, $y_{43} = 26$, $y_{44} = 18$,

$z_{41} = 7$, $z_{42} = 3$, $z_{43} = -5$, $z_{44} = -3$, $z_{45} = 4$, $z_{46} = 6$, $z_{47} = -2$, and $z_{48} = -6$.

Evaluating equations (3.26)-(3.29) we get $c_0 = 330$, $c_1 = 240$, $c_2 = 324$, $c_3 = 253$.

Evaluating equation (3.90) we have $\langle A \times B \rangle_{2^N-1} = 9307$ and the result checks correct.

3.6.3.2 Modulo 2^N + 1 product

Equations (3.96)-(3.98) give $g_0 = 300$, $g_1 = 83$, and $g_2 = 182$.

Evaluating equations (3.99)-(3.101) we get $d_0 = -270$, $d_1 = 74$, and $d_2 = -40$.

Evaluating equation(3.91) we have $\langle A \times B \rangle_{2^N+1} = 43907$ and the result checks correct.

3.6.3.3 Modulo 2^N product

Equations (3.107)-(3.109) give $e_0 = 30$, $e_1 = 157$, and $e_2 = 142$.
Evaluating equation (3.102) we have $<A \times B>_{2^N} = 26606$ and the result checks correct.

3.6.3.4 Full precision product

Equations (3.111)-(3.117) give $f_0 = 30$, $f_1 = 157$, $f_2 = 142$, $f_3 = 253$, $f_4 = 300$, $f_5 = 83$, and $f_6 = 182$.
Evaluating equation(3.110) we have A x B = 3161221102 and the result checks correct.

3.6.4 Hardware and speed analysis

All the analysis in this section is provided for the case when the product of two numbers is obtained by decomposing each number into four equal parts, say each with k bits. Four methods are compared:

i) traditional techniques,

ii) quarter squared algorithm,

iii) new multipliers modulo $2^N - 1$ [51],

iv) techniques of this chapter.

The traditional way of computing c_0, c_1, c_2, and c_3 would be by using equations (3.2)-(3.5). Here each product term can be realized by a ROM of size $2^{2k} \times 2k$. Since sixteen product terms have to be realized, we would need a total of $k \times 2^{2k+5}$ ROM bits.

Direct application of the quarter squared algorithm to each term of equations (3.2)-(3.5) would require for each term two ROMs, each of size $2^{k+1} \times (2k + 2)$. Thus sixteen product terms would require a total of $(k + 1) \times 2^{k+7}$ ROM bits.

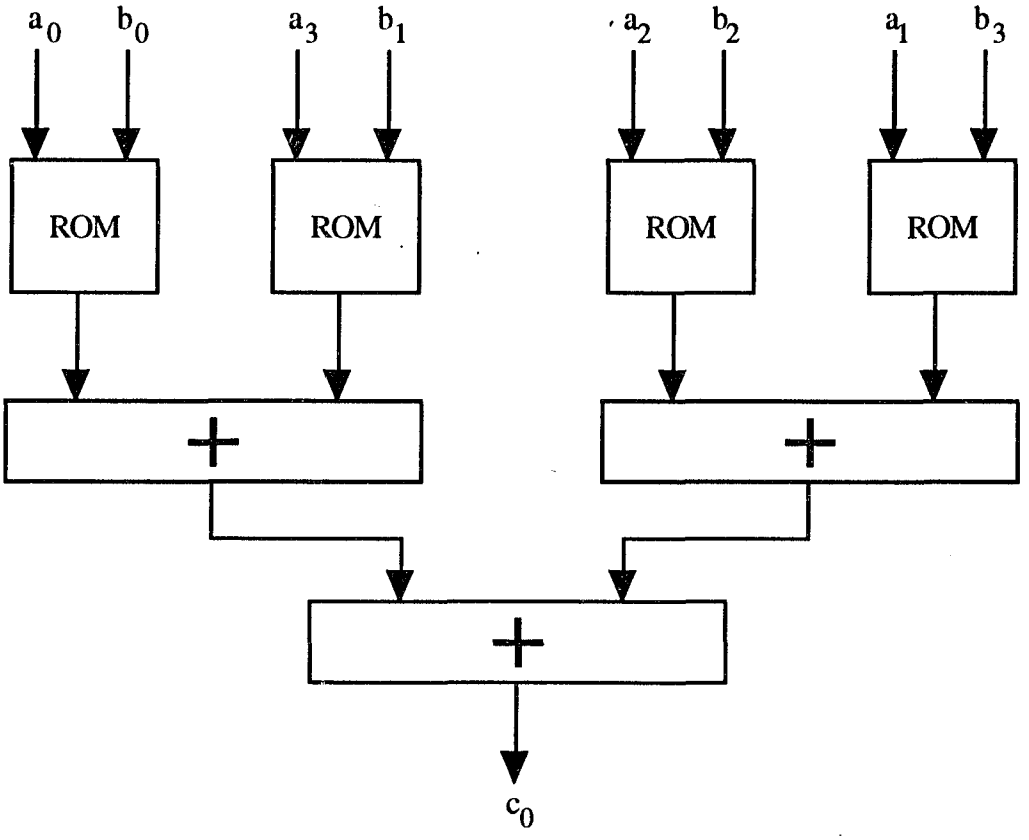
New multipliers modulo 2^N -1, [51] requires a total of $(2k + 5) \times 2^{k+6}$ ROM bits.

Based on our techniques, equations (3.26)-(3.29) dictate that 16 ROMs, each of size $2^{k+2} \times (2k + 4)$ would be required, thus giving a total of $(k + 2) \times 2^{k+7}$ ROM bits.

Table 3.3 summarizes these results and also presents data on the number of adders required by each method. Here we have assumed that the operation (a - b) requires only one adder. This is a reasonable assumption because our numbers are integers, and therefore the operation (a - b) can be realized by (a + (-b)), where (-b) is the two's complement representation of b. The hardware module for this adder can be suitably wired to obtain this function. Figures 3.11, 3.12, and 3.13 show the hardware structure needed to compute c_0 for all methods except that of (iii) which can be found in [51]. These hardware structures can be replicated appropriately for the other terms. The number of levels through which the data has to flow is indicated on each figure, however this might be irrelevant if data is being processed continuously. In such a case the limiting factor will be the speed at which the ROM can deliver. Method (i) requires a ROM whose size is in the order of $O(2^{2k})$ while all the other three methods require ROMs with sizes of order $O(2^k)$. Since method (i) requires the largest sized ROMs, it will be the slowest. Also since the total number of ROM bits is very high, it will not be possible to integrate the entire design on a single chip. With respect to speed and number of ROM bits, methods (ii) through (iv) are comparable. With respect to the total number of ROM bits required, method (ii), i.e. the direct application of the quarter squared algorithm, appears to be the best but considering the fact that it requires 60 adders it will be the most complex one to build. Our techniques in this chapter yield the best trade off for speed and hardware; while they require more ROM bits than the

Table 3.3: Hardware and speed comparison of various look-up table techniques

Method	Total number of ROM bits	Number of two operand adders	Size of largest ROM	Speed as a function of ROM size	Integration on a single chip
Traditional Techniques	$k \times 2^{2k+5}$	12	$2^{2k} \times 2k$	$O(2^{2k})$	NO, for N>= 16
Quarter Squared	$(k+1) \times 2^{k+7}$	60	$2^{k+1} \times (2k+2)$	$O(2^k)$	NO, for N>= 16
Reference[51]	$(2k+5) \times 2^{k+6}$	38	$2^{k+3} \times (2k+6)$	$O(2^k)$	Possible
This section	$(k+2) \times 2^{k+7}$	40	$2^{k+2} \times (2k+4)$	$O(2^k)$	Possible



1) Number of levels = 3 (2 adders + 1 ROM)

2) Size of each ROM = 2^{2k} x 2k

Fig. 3.11. Hardware architecture to implement equation (3.2) using traditional techniques

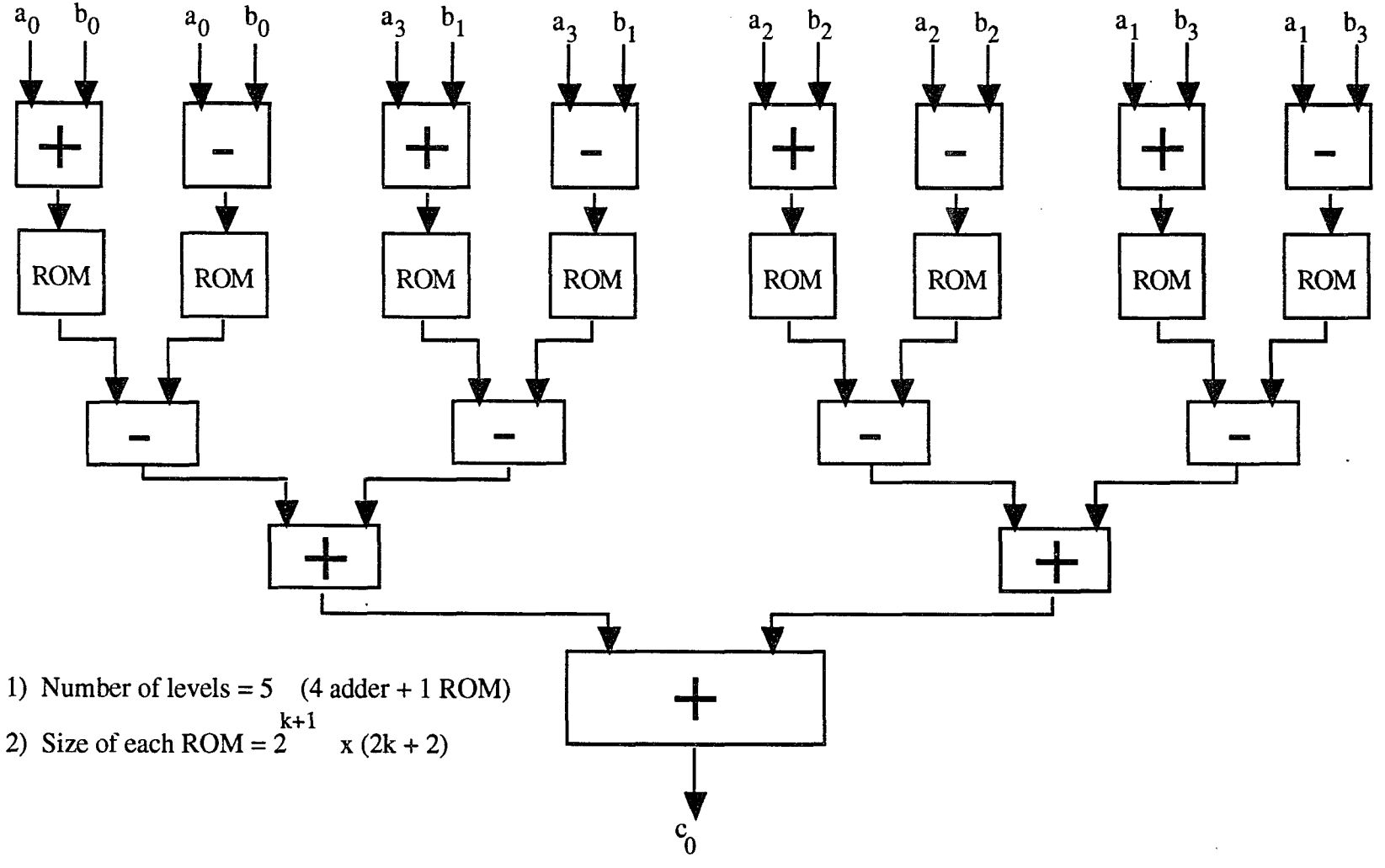


Fig. 3.12: Hardware architecture for implementing equation (3.2) using the quarter squared algorithm.

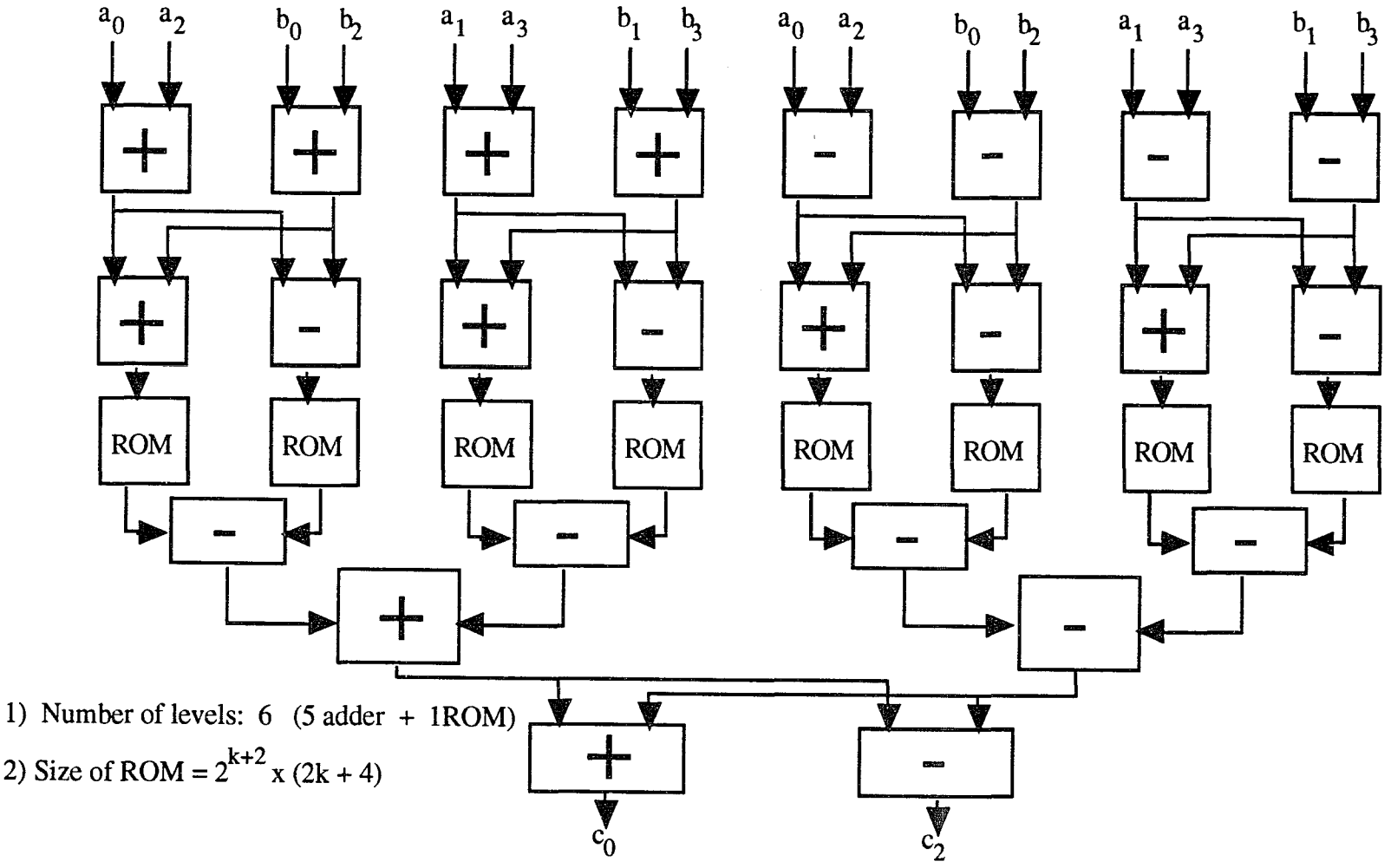


Fig. 3.13: Hardware architecture to realize equations (3.26) and (3.28)

quarter squared algorithm, they require far fewer adders. Techniques of this chapter out perform those of [51] in many respects viz. smaller maximum ROM size, a total of fewer ROM bits, and higher speed. Also based on the techniques of this chapter, all the ROMs are of the same size and hence identical. This again is a big advantage for VLSI designs. The regularity of the hardware architecture is clearly seen in figure 3.13.

Table 3.4 summarizes the ROM requirements for different wordlengths(N) for the case when the numbers are each decomposed into four parts. Table 3.5 summarizes the overhead ROM requirements required for computing the product modulo 2^N+1, modulo 2^N, and the full precision product. Again, each product is computed by decomposing each of the numbers into four equal parts. Overhead is defined as the number of ROM bits needed over and above those required for the computation of the modulo 2^N-1 product. The values are based on equations (3.96)-(3.98), are obtained in a straightforward manner and are hence not detailed.

3.7 Summary

In this chapter we have discussed several implementation issues of the squaring and convolution operations. In section 3.2 we presented an intuitive CSA based implementation for the squaring operation that was faster than the schemes suggested by [54]. We showed, by counter example, that the number of levels required to add a set of summands in a parallel fashion is not only a function of the height of the tallest column but is also a function of the heights of the other columns and their relative placements. In section 3.3 we presented an alternate implementation for the squaring operation and compared its performance with existing schemes. We found that for VLSI implementations of small wordlength squarers, the prime factors in the selection of a design would be regularity and modularity. This was because different schemes for

Table 3.4: Cost Comparison in ROM bits of the various techniques for computing $\langle A \times B \rangle_{2^N-1}$

Word Length N	Decomposed Part Length k	Traditional Techniques cost	Quarter Squared Alg. cost	Reference [51] cost	This section cost	% savings (vs. trad. tech.)
16	4	$2^5 \times 2^{10}$	10×2^{10}	13×2^{10}	12×2^{10}	62.50
32	8	$2^{10} \times 2^{14}$	18×2^{14}	21×2^{14}	20×2^{14}	98.04
64	16	$2^{19} \times 2^{22}$	34×2^{22}	37×2^{22}	36×2^{22}	99.99

Table 3.5: Cost in ROM bits for integrated multiplier, based on techniques of this section

Word Length N	Decomp-osed Part Length k	Mod. $(2^N - 1)$ product cost $(k+2) \times 2^{k+7}$	Mod. $(2^N + 1)$ product overhead cost $6(k+1) \times 2^{k+3}$	Mod. (2^N) product overhead cost	Full precis-ion product overhead cost	% over-head
16	4	6×2^{11}	30×2^7	None	None	31.25
32	8	10×2^{15}	54×2^{11}	None	None	33.75
64	16	18×2^{23}	102×2^{19}	None	None	35.41

small wordlength squarers had similar hardware costs. In section 3.4 we presented in detail CSA based implementations for 4, 8, and 16 point cyclic convolutions. The analysis showed that the increase in the number of addition operations does not significantly diminish the hardware savings obtained by the reduction in the number of the squaring operations. We again emphasize that the purpose of the implementations was solely to argue the case in point and not for the purpose of field implementations. We also clearly demonstrated that our approach is an excellent candidate for smart architectures. In section 3.5 hybrid implementations of the convolution operation are presented. Here, we've shown that if the multiplication and squaring operation are implemented by ROMs then our method while being a little slower, yields phenomenal hardware savings. Finally, in section 3.6 we presented an application of the convolution operation in the field of computer arithmetic, namely, the problem of integer multiplication. We present the case of a modulo 2^N -1 multiplier and show how our techniques can be extended to multiplication in other rings, namely, modulo 2^N +1 and modulo 2^N. We also present the case of full precision multiplication. We show that in all cases our methods produce significant ROM bit savings when compared with traditional implementations.

Chapter 4

ROM Based Methods for Computing the Squaring Operation in Modular Rings

In the previous chapters we developed algorithms for modular multiplication and cyclic convolution that relied primarily on squaring operations. The focal point of those algorithms was to how best reduce the number of squaring operations to perform the desired computation. Also, these algorithms were discussed in the context of full precision computation. However, signal processing applications often rely on the properties of the residue number system (RNS) [38] to perform efficient computations. In such an environment computations are performed over modular rings, the popular choices being 2^n, 2^{n-1}, and 2^{n+1} [38], [47]. Therefore, in this chapter we focus on hardware efficient compression schemes for computing the square of a number modulo 2^n, modulo $2^n - 1$, and modulo $2^n + 1$, using ROM look-up tables. In this process we present several schemes and compare their relative merits and de-merits.

In section 4.1 we attempt to motivate the reader by showing how a few simple arithmetic manipulations can reduce the size of the ROM required for the squaring operation. These schemes were presented in brief in [63]. In section 4.2 we present our newly proposed optimized schemes which were also presented very briefly in [53]. In this chapter these schemes are presented in detail, for both the sake of completeness and for comparison with the newly proposed schemes.

4.1 Memory compression schemes for arithmetic in modulo 2^n

Our objective here is to find efficient ways to compute the square of a number. In this chapter we consider ROM based methods to perform this computation. Let us consider a number A belonging to the modular ring $Z_{2^n} = \{0, 1, ..., 2^n - 1\}$. Then A has a n-bit binary representation as in $A = a_{n-1}a_{n-2} \ldots a_1a_0$; $a_i \in \{0,1\}$. Our task is to compute $<A^2>_{2^n}$, where $<x>_m$ denotes the operation x modulo m. Our method basically consists of decomposing the number A into two words, a high word, say A_{Hi}, and a low word, say A_{Li}, and then performing certain arithmetic manipulations to yield significant savings in ROM bits. We then show that by varying the lengths of A_{Hi} and A_{Li} we can obtain more savings in ROM bits at the expense of an overhead consisting of a few gates and multiplexers. We present the analysis for three different decompositions of the number A.

4.1.1 Analysis when the high word is one bit long

Let A_{H1} be a 1 bit word and A_{L1} be a n-1 bit word with $A_{H1} = a_{n-1}$ and $A_{L1} = a_{n-2} \ldots a_1a_0$. Then we have

$$A = A_{H1}2^{n-1} + A_{L1} \tag{4.1}$$

and

$$A^2 = A_{H1}^2\, 2^{2n-2} + A_{H1}A_{L1}\, 2^n + A_{L1}^2 \tag{4.2}$$

If $n > 2$, then $2n - 2 > n$, $<2^{2n-2}>_{2^n} = 0$, and we get

$$<A^2>_{2^n} = <A_{L1}^2>_{2^n} \tag{4.3}$$

For a table look-up approach, as shown in figure 4.1, the square of the number A can be computed by simply using a ROM of size $2^n \times n$. We shall refer to this as the direct or traditional implementation as it involves no arithmetic manipulations. However,

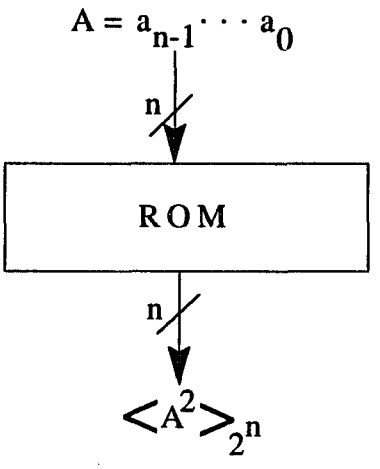


Figure 4.1: The direct computation of $\langle A^2 \rangle_{2^n}$, ROM size = $2^n \times n$

equation (4.3) shows us that the same task can be accomplished by using a ROM of size $2^{n-1} \times n$ as shown in figure 4.2. Thus we have a savings of 50% in ROM bits with no additional overhead. In the next section, by applying the same techniques, we analyze the effects of increasing the length of A_{H1} and reducing the length of A_{L1} on the savings in ROM bits.

4.1.2 Analysis when the high word is two bits long

Let A_{H2} be a 2 bit word and A_{L2} be a n-2 bit word with $A_{H2} = a_{n-1}a_{n-2}$ and $A_{L2} = a_{n-3} \ldots a_1a_0$. Then we have

$$A = A_{H2}2^{n-2} + A_{L2} \tag{4.4}$$

and

$$A^2 = A_{H2}^2\, 2^{2n-4} + A_{H2}A_{L2}\, 2^{n-1} + A_{L2}^2 \tag{4.5}$$

If $n > 4$, then $2n - 4 > n$, $<2^{2n-4}>_{2^n} = 0$, and we get

$$<A^2>_{2^n} = <A_{H2}A_{L2}\, 2^{n-1} + A_{L2}^2>_{2^n} \tag{4.6}$$

The possible values of $<A_{H2}A_{L2}\, 2^{n-1}>_{2^n}$ are shown in Table 4.1. Further,

$$<A_{L2}\, 2^{n-1}>_{2^n} = <(a_{n-3} \ldots a_1a_0)2^{n-1}>_{2^n} = <(a_{n-3} \ldots a_1)2^n + a_02^{n-1}>_{2^n} = a_02^{n-1} \tag{4.7}$$

Combining equation (4.7) and Table 4.1 we can write

$$<A^2>_{2^n} = \begin{cases} <A_{L2}^2>_{2^n} & \text{if } a_{n-2}=0 \\ <a_02^{n-1}+A_{L2}^2>_{2^n} & \text{if } a_{n-2}=1 \end{cases} \tag{4.8}$$

If $<A_{L2}^2>_{2^n}$ is represented in binary as $b_{n-1}b_{n-2}\ldots b_1b_0$, and letting $c_{n-1} = a_0 \oplus b_{n-1}$ where $\oplus$ denotes the exclusive-or operation, equation (4.8) can then be rewritten as

$$<A^2>_{2^n} = \begin{cases} b_{n-1}b_{n-2}\ldots b_1b_0 & \text{if } a_{n-2}=0 \\ c_{n-1}b_{n-2}\ldots b_1b_0 & \text{if } a_{n-2}=1 \end{cases} \tag{4.9}$$

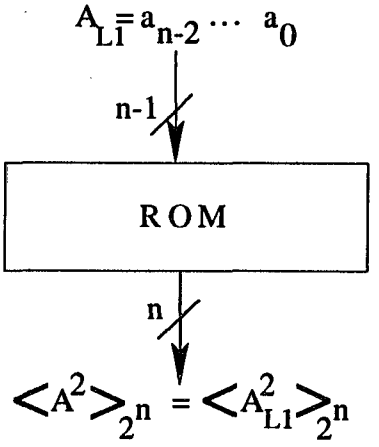


Figure 4.2: The computation of $\langle A^2 \rangle_{2^n}$ based on equation (4.3), ROM size = $2^{n-1} \times n$

Table 4.1: Values of $\langle A_{H2}A_{L2}\, 2^{n-1}\rangle_{2^n}$

A_{H2}		$\langle A_{H2}A_{L2}2^{n-1}\rangle_{2^n}$
a_{n-1}	a_{n-2}	
0	0	0
0	1	$\langle A_{L2}2^{n-1}\rangle_{2^n}$
1	0	0
1	1	$\langle A_{L2}2^{n-1}\rangle_{2^n}$

We note from Table 4.1 that the value of bit a_{n-1} is irrelevant and thus does not appear in equation 4.9.

To obtain $<A_{L2}^2>_{2^n}$ a ROM of size $2^{n-2} \times n$ would be sufficient as the length of A_{L2} is only n-2 bits. Thus we have a savings of 75% in ROM bits when compared with the direct implementation. However, to obtain $<A^2>_{2^n}$ we need to realize equation (4.9) and for this in addition to the ROM we would need an exclusive-or gate and a single 2×1 multiplexer(mux) as shown in figure 4.3. Thus while equation (4.9) produces more savings in ROM bits than equation (4.3) it also has a small overhead.

In the next section we increase further the length of the high word by one more bit and simultaneously reduce the length of the low word by one bit. The objective of the analysis here is to show that while more savings in ROM bits are obtained the overhead increases in a disproportionate fashion.

4.1.3 Analysis when the high word is three bits long

Let A_{H3} be a 3-bit word and A_{L3} be a (n-3)-bit word with $A_{H3} = a_{n-1}a_{n-2}a_{n-3}$ and

$A_{L3} = a_{n-4} \ldots a_1a_0$. Then we have

$$A = A_{H3}2^{n-3} + A_{L3} \tag{4.10}$$

and

$$A^2 = A_{H3}^2 2^{2n-6} + A_{H3}A_{L3}\, 2^{n-2} + A_{L3}^2 \tag{4.11}$$

If $n > 6$, then $2n - 6 > n$, $<2^{2n-6}>_{2^n} = 0$, and we get

$$<A^2>_{2^n} = <A_{H3}A_{L3}\, 2^{n-2} + A_{L3}^2>_{2^n} \tag{4.12}$$

The possible values of $<A_{H3}A_{L3}\, 2^{n-2}>_{2^n}$ are shown in Table 4.2. Further, remembering that $A_{L3} = a_{n-4} \ldots a_1a_0$, we have

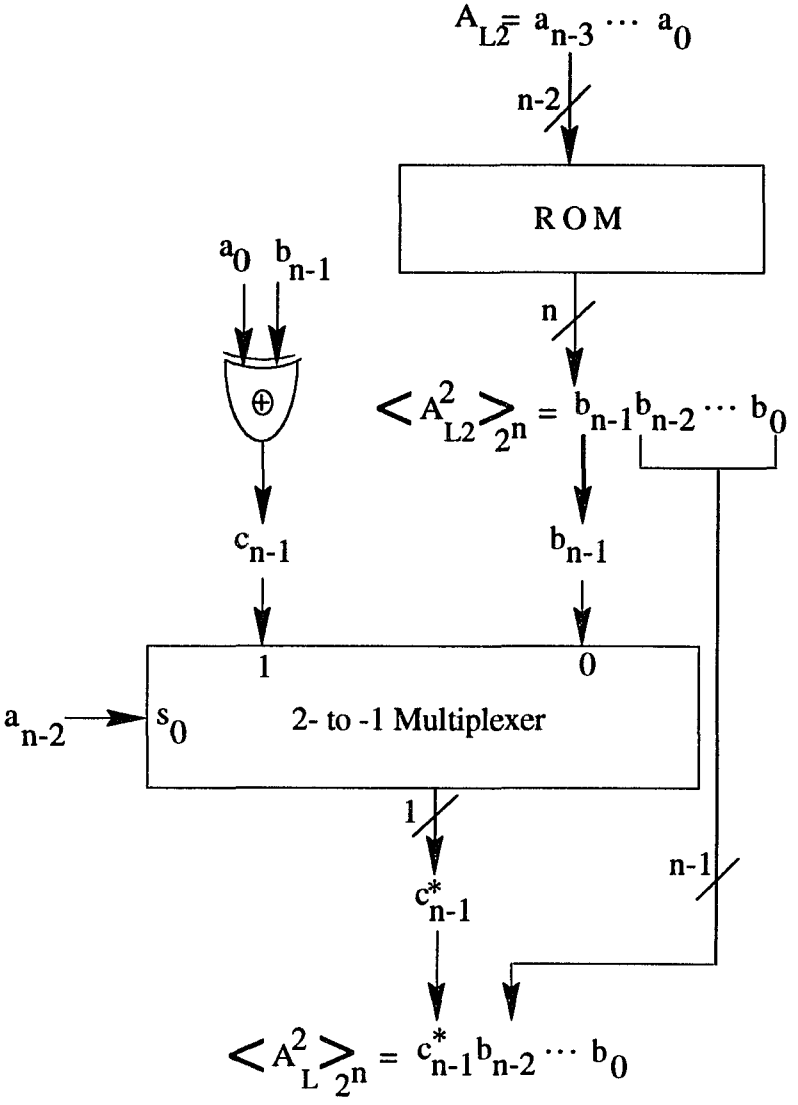


Figure 4.3: The computation of $\langle A^2 \rangle_{2^n}$ based on equation (4.9), ROM size = $2^{n-2} \times n$

Table 4.2: Values of $\langle A_{H3}A_{L3}\, 2^{n-2}\rangle_{2^n}$

A_{H3}			$\langle A_{H3}A_{L3}2^{n-2}\rangle_{2^n}$
a_{n-1}	a_{n-2}	a_{n-3}	
0	0	0	0
0	0	1	$\langle A_{L3}2^{n-2}\rangle_{2^n}$
0	1	0	$\langle A_{L3}2^{n-1}\rangle_{2^n}$
0	1	1	$\langle 3A_{L3}2^{n-2}\rangle_{2^n}$
1	0	0	0
1	0	1	$\langle A_{L3}2^{n-2}\rangle_{2^n}$
1	1	0	$\langle A_{L3}2^{n-1}\rangle_{2^n}$
1	1	1	$\langle 3A_{L3}2^{n-2}\rangle_{2^n}$

$$\begin{aligned} <A_{L3}\, 2^{n-2}>_{2^n} &= <(a_{n-4} \ldots a_1 a_0)2^{n-2}>_{2^n} \\ &= <(a_{n-4} \ldots a_2)2^n + a_1 a_0 2^{n-2}>_{2^n} \\ &= a_1 a_0 2^{n-2} \end{aligned} \tag{4.13}$$

and similarly

$$<A_{L3}\, 2^{n-1}>_{2^n} = <(a_{n-4} \ldots a_1 a_0)2^{n-1}>_{2^n} = <(a_{n-4} \ldots a_1)2^n + a_0 2^{n-1}>_{2^n} = a_0 2^{n-1} \tag{4.14}$$

Adding (4.13) and (4.14) we get

$$<3A_{L3}\, 2^{n-2}>_{2^n} = <a_1 a_0 2^{n-2} + a_0 2^{n-1}>_{2^n} = (a_1 \oplus a_0) a_0 2^{n-2} \tag{4.15}$$

If $<A_{L3}^2>_{2^n}$ is represented in binary as $d_{n-1}d_{n-2}\ldots d_1 d_0$, and letting $h_{n-1} = a_1 \oplus a_0$, equations (4.12) - (4.15) combined with table 4.2 can then be rewritten as

$$<A^2>_{2^n} = \begin{cases} d_{n-1}d_{n-2}d_{n-3}\ldots d_1 d_0 & \text{if } a_{n-2}a_{n-3} = 00 \\ e_{n-1}e_{n-2}d_{n-3}\ldots d_1 d_0 & \text{if } a_{n-2}a_{n-3} = 01 \\ f_{n-1}d_{n-2}d_{n-3}\ldots d_1 d_0 & \text{if } a_{n-2}a_{n-3} = 10 \\ g_{n-1}e_{n-2}d_{n-3}\ldots d_1 d_0 & \text{if } a_{n-2}a_{n-3} = 11 \end{cases} \tag{4.16}$$

where the bits e_{n-1}, e_{n-2}, f_{n-1}, and g_{n-1} are given by

$$e_{n-1} = a_1 \oplus d_{n-1} \oplus (a_0 \wedge d_{n-2}) \tag{4.17}$$

$$e_{n-2} = a_0 \oplus d_{n-2} \tag{4.18}$$

$$f_{n-1} = a_0 \oplus d_{n-1} \tag{4.19}$$

$$g_{n-1} = h_{n-1} \oplus d_{n-1} \oplus (a_0 \wedge d_{n-2}) \tag{4.20}$$

with $\wedge$ denoting the AND operation. We once again see from Table 4.2 that the value of $<A_{H3}A_{L3}\, 2^{n-2}>_{2^n}$ is not a function of bit a_{n-1} and this is accordingly reflected by equation (4.16).

Now, to obtain $<A_{L3}^2>_{2^n}$ a ROM of size $2^{n-3} \times n$ would be adequate as the length of A_{L3} is only n-3 bits. Thus we have a savings of 87.5% in ROM bits when compared with the direct implementation. However, to compute $<A^2>_{2^n}$ we need to

implement equation (4.16) and for this in addition to the above ROM we would need 6 gates and a 4×1 multiplexer of word length two. This implementation is shown in figure 4.4.

The preceding analysis shows that a direct extension of the above method, i.e. increasing the length of the high word while simultaneously reducing the length of the lower word results in reducing the number of ROM bits but at the same time increases the overhead both in terms of gate count and complexity of the design. In the next section we show the optimal size for the high and low words to obtain not only the maximum savings in ROM bits but also an overhead that is less than that required by the above method and is also streamlined with respect to implementation. In a later section we show that this overhead is also streamlined with respect to arithmetic modulo 2^n -1 and modulo 2^n +1.

4.2 Optimized memory compression schemes for arithmetic in modulo 2^n

The following proposed schemes were published very briefly in [53]. In this section the schemes are presented in detail followed by a comparative analysis. We use the same notation as before and our task remains the same, i.e. we wish to compute $\langle A^2 \rangle_{2^n}$, where once again $\langle x \rangle_m$ denotes the operation x modulo m.

Recognizing that the first bit that might produce an overflow when an n-bit (n even) number is squared, is located at the $2^{n/2}$ position we decompose the number into two parts each of length n/2 bits. The technique is explained in detail for the case when the arithmetic is done modulo 2^n and n even. All other cases are summarized in Tables 4.3 and 4.4.

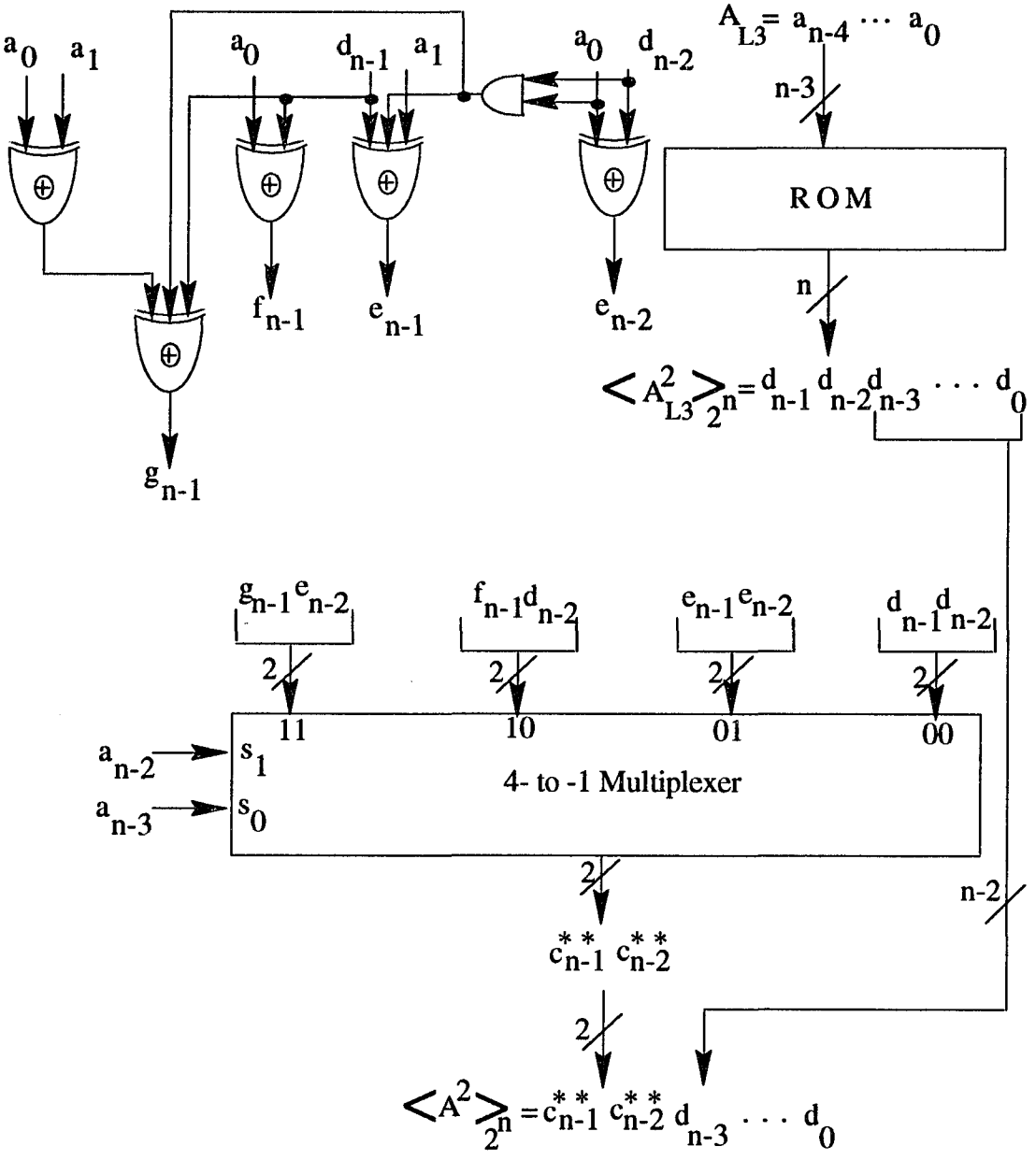


Figure 4.4: The computation of $\langle A^2 \rangle_{2^n}$ based on equation (4.16), ROM size = $2^{n-3} \times n$

Table 4.3: Results when n is even. $A_H = a_{n-1}a_{n-2} \cdots a_{n/2}$, $A_L = a_{(n/2)-1}a_{(n/2)-2} \cdots a_1a_0$, and $QS = 2^{n/2-1}\{(A_H + A_L)^2 - (A_H - A_L)^2\}$

Operation	Formula	Cost in ROM bits	Overhead	%savings of ROM bits	
				n = 16	n = 32
$<A^2>_{2^n}$	$<A_L^2 + QS>_{2^n}$	$5 \times 2^{n/2} \times n$	4 adders	98.04	99.99
$<A^2>_{2^n-1}$	$<A_H^2 + A_L^2 + QS>_{2^n-1}$	$6 \times 2^{n/2} \times n$	5 adders	97.65	99.99
$<A^2>_{2^n+1}$	$<-A_H^2 + A_L^2 + QS>_{2^n+1}$	$6 \times 2^{n/2} \times n$	5 adders	97.65	99.99

Table 4.4: Results when n is odd. $A_H = a_{n-1}a_{n-2} \cdots a_{(n+1)/2}$, $A_L = a_{(n-1)/2} \cdots a_1a_0$, and $QS = 2^{(n-1)/2}\{(A_H + A_L)^2 - (A_H - A_L)^2\}$

Operation	Formula	Cost in ROM bits	Over-head	%savings of ROM bits	
				n = 15	n = 31
$<A^2>_{2^n}$	$<A_L^2 + QS>_{2^n}$	$5 \times 2^{(n+1)/2} \times n$	4 adders	96.09	99.99
$<A^2>_{2^n-1}$	$<2A_H^2 + A_L^2 + QS>_{2^n-1}$	$11 \times 2^{(n-1)/2} \times n$	5 adders	95.70	99.99
$<A^2>_{2^n+1}$	$<-2A_H^2 + A_L^2 + QS>_{2^n+1}$	$11 \times 2^{(n-1)/2} \times n$	5 adders	95.70	99.99

4.2.1 Analysis when n is even

Consider a number A belonging in the modular ring Z_{2^n} with a n-bit representation as given before. Let $A_H = a_{n-1}a_{n-2} \cdots a_{n/2}$ and $A_L = a_{(n/2)-1}a_{(n/2)-2} \cdots a_1a_0$. Then we have

$$A = A_H 2^{n/2} + A_L \tag{4.21}$$

and

$$A^2 = A_H^2\, 2^n + A_H A_L\, 2^{n/2+1} + A_L^2 \tag{4.22}$$

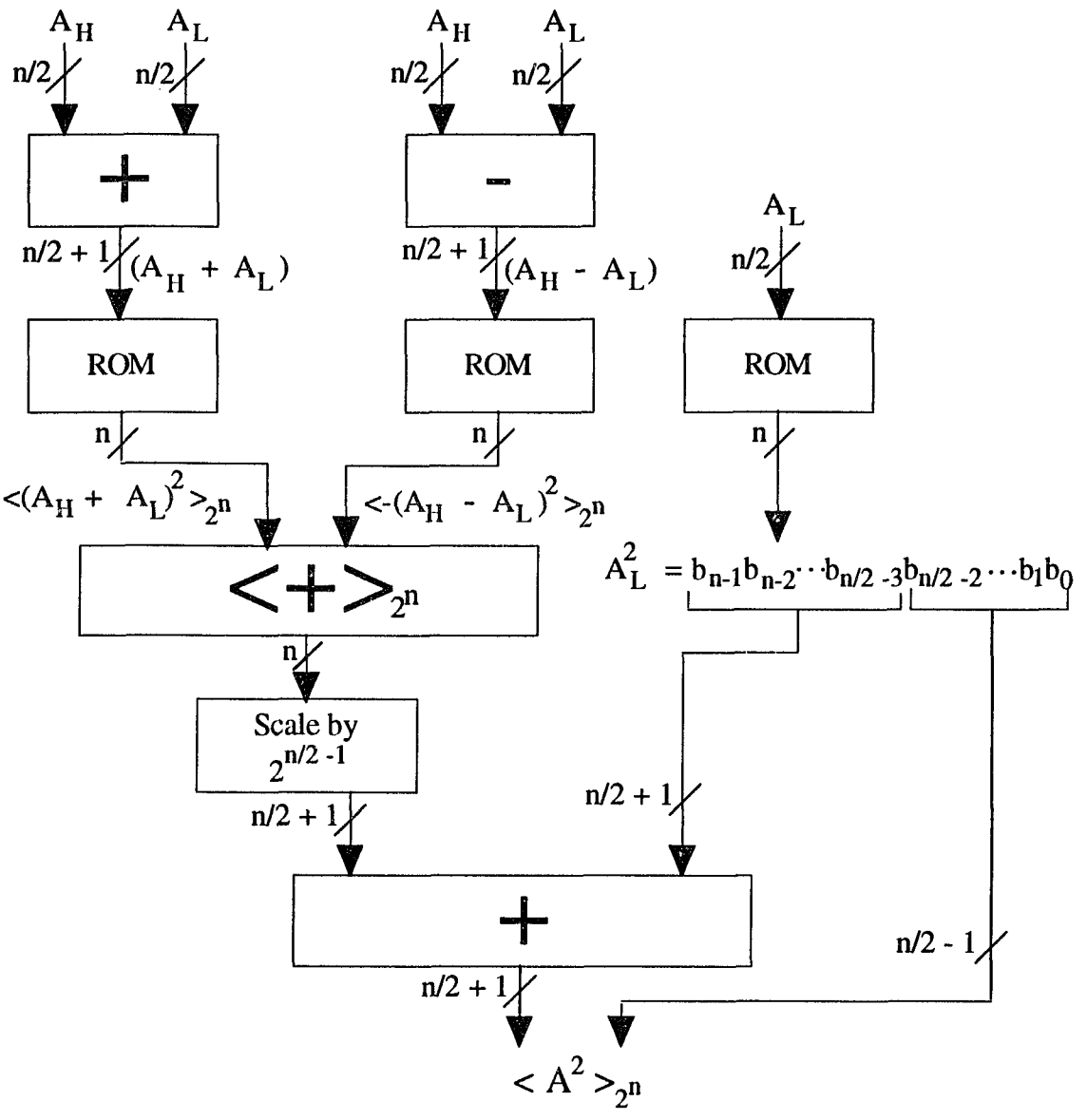
while

$$\langle A^2 \rangle_{2^n} = \langle A_L^2 + A_H A_L\, 2^{n/2+1} \rangle_{2^n} \tag{4.23}$$

The term A_L^2 can be computed using a ROM of size $2^{n/2} \times n$. The product $A_H A_L$ can be realized by using the quarter squared algorithm [47], thus giving

$$\langle 2^{n/2+1}\, A_H A_L \rangle_{2^n} = \langle 2^{n/2-1} \{(A_H + A_L)^2 - (A_H - A_L)^2\} \rangle_{2^n} \tag{4.24}$$

Each of the square terms in equation (4.24) can be realized using a ROM of size $2^{n/2+1} \times n$. Thus the total number of ROM bits required to compute $\langle A^2 \rangle_{2^n}$ is $5 \times 2^{n/2} \times n$. For the case when n=16, we have obtained a savings of 98% in ROM bits while the overhead is two (n/2)-bit adders and two modulo 2^n adders of size n. Note that while n increases the savings in ROM bits increases whereas the overhead remains the same with respect to the count of adders, i.e. the number of adders is not a function of n. Also, when it is required to compute terms like $-(A^2)$, a negator is not needed as the ROM used for this purpose can be designed to directly generate the negative result. The implementation of this technique is shown in figure 4.5.



Note: 1) The scaler unit simply shifts it's input to the left by $2^{n/2}-1$ positions. Thus the lower (n/2) -1 bits are zeros and the upper (n/2) -1 bits are = 0 modulo 2^n and are hence ignored. The remaining bits can be simply hardwired at the appropriate locations in the next unit and thus the scaler unit requires no additional hardware.

2) The modulo 2^n adders are regular adders with only the lower n significant bits, i.e. carry's are ignored.

Figure 4.5: The computation of $<A^2>_{2^n}$ based on equations (4.23)-(4.24), total ROM bits = $5 \times 2^{n/2} \times n$

4.3 Numerical example

We now present a numerical example to illustrate the techniques presented in sections 4.1 and 4.2. Our task is to compute $<A^2>_{2^n}$ with n = 10 and A = $a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$ = 1111011101 = $(989)_{10}$. We expect to obtain $<989^2>_{1024}$ = 201.

4.3.1 Illustrating techniques of section 4.1.1

Decomposing A into a 1-bit high word and a 9-bit low word, we get $A_{H1} = a_9$ = 1 and $A_{L1} = a_8a_7a_6a_5a_4a_3a_2a_1a_0$ = 111011101 = $(477)_{10}$. Equation (4.3) gives $<A^2>_{1024} = <A_{L1}^2>_{1024} = <477^2>_{1024}$ = 201 and this agrees with the expected result.

4.3.2 Illustrating techniques of section 4.1.2

Decomposing A into a 2-bit high word and a 8-bit low word, we get $A_{H2} = a_9a_8$ = 11 = $(3)_{10}$ and $A_{L2} = a_7a_6a_5a_4a_3a_2a_1a_0$ = 11011101 = $(221)_{10}$. Thus $<A_{L2}^2>_{2^n} = <221^2>_{1024}$ = $(713)_{10}$ = $b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ = 1011001001. Since $a_{n-2} = a_8$ = 1, equation (4.9) gives $<A^2>_{1024} = c_9b_8 \ldots b_1b_0$ with $c_9 = a_0 \oplus b_9$ = 0. Plugging the values we have $<A^2>_{1024}$ = 0011001001 = $(201)_{10}$ and this agrees with the expected result.

4.3.3 Illustrating techniques of section 4.1.3

Decomposing A into a 3-bit high word and a 7-bit low word, we get $A_{H3} = a_9a_8a_7$ = 111 = $(7)_{10}$ and $A_{L3} = a_6a_5a_4a_3a_2a_1a_0$ = 1011101 = $(93)_{10}$. Thus $<A_{L3}^2>_{2^n} = <93^2>_{1024}$ = $(457)_{10}$ = $d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0$ = 0111001001. Since $a_{n-2}a_{n-3} = a_8a_7$ = 11, equation (4.16) gives $<A^2>_{1024} = g_9e_8d_7 \ldots d_1d_0$ with equations (4.18) and (4.20)

giving $g_9 = a_1 \oplus a_0 \oplus d_9 \oplus (a_0 \wedge d_8)$ and $e_8 = a_0 \oplus d_8$. Plugging the values we have $<A^2>_{1024} = 0011001001 = (201)_{10}$ and this agrees with the expected result.

4.3.4 Illustrating techniques of section 4.2.1

Decomposing A into a 5-bit high word and a 5-bit low word, we get $A_{H5} = a_9a_8a_7a_6a_5 = 11110 = (30)_{10}$ and $A_{L3} = a_4a_3a_2a_1a_0 = 11101 = (29)_{10}$. Thus $<A_L^2>_{2^n} = <29^2>_{1024} = (841)_{10}$. Equation (4.24) gives $<2^6A_HA_L>_{2^n} = <2^4\{(A_H + A_L)^2 - (A_H - A_L)^2\}>_{2^n} = <2^4\{59^2 - 1^2\}>_{1024} = 384$. Plugging the values into equation (4.23) we have $<A^2>_{1024} = <384 + 841>_{1024} = (201)_{10}$ and this agrees with the expected result.

4.4 Comparing techniques of section 4.1 with 4.2

In order to make a fair comparison of the techniques presented in section 4.1 with those of section 4.2 we decompose the number as given in section 4.2 and then apply the techniques of section 4.1. We compare the two techniques with respect to hardware cost and speed. The hardware cost is expressed as a function of 2-input gates while the speed as a function of gate delays. Since both methods are implementing equation (4.23) the cost for implementing the term A_L^2 does not need to be taken into account as both methods implement this term in exactly the same fashion, viz. using a ROM of size $2^{n/2} \times n$. The difference in cost and speed arises based on the manner in which the other term, namely $A_HA_L\ 2^{n/2+1}$ is implemented and added to A_L^2.

4.4.1 Cost and speed analysis for section 4.1

In order to implement the term $A_HA_L\ 2^{n/2+1}$ we would need (n/2) -1 multiplexers of size $2^{(n/2)-1} \times 1$. We arrive at this figure based on the following:

Recall that in this method we make use of the fact that the lower (n/2) + 1 bits of the end result is the same as the lower (n/2) + 1 bits of the term A_L^2. This is simply

because the term $A_H A_L \, 2^{n/2+1}$ is the quantity $A_H A_L$ shifted to the left by (n/2) + 1 positions with zeroes filled in. Thus the remaining (n/2) - 1 bits of the end result are determined by the summation of $A_H A_L$ with the upper (n/2) -1 bits of the term A_L^2. This accounts for the number of multiplexers. This scheme is pictorially shown in figure 4.6. Let $A_H A_L$ be represented in binary as $r_{h-1} r_{h-2} \ldots r_0$ and the upper (n/2) -1 bits of the term A_L^2 as $s_{h-1} s_{h-2} \ldots s_0$ where the subscript h also represents the number of bits in the high word.

Since A_H is n/2 bits long its value lies in the range 0 to $2^{(n/2)} - 1$. However, the most significant bit of A_H i.e a_{n-1} has a weight of $2^{n/2-1}$ and when multiplied by $2^{n/2+1}$ it gives $a_{n-1} \times 2^n$ which modulo 2^n is equal to 0. Therefore the only bits of A_H that are of interest to us are $a_{n-2} a_{n-3} \ldots a_{(n/2)}$. This gives us $2^{(n/2)-1}$ different terms to be multiplied with $A_L \, 2^{n/2+1}$ thus giving us the size of the multiplexer as $2^{(n/2)-1} \times 1$.

Let A_L^2 be represented in binary as $b_{n-1} b_{n-2} \ldots b_1 b_0$. Note that A_L^2 is inherently a n-bit number. The inputs to the multiplexer are terms, each one of which is the sum of $b_{n-1} b_{n-2} \ldots b_{(n/2)+1}$ and one of the $A_H A_L$ terms. (There are $2^{(n/2)-1}$ such terms).

The following assumptions are made for calculating the amount of hardware:

1) We assume that the design is based on 2-input gates. We do not count the cost of inverters. We allow all types of 2-input gates including exclusive-or gates.

2) All gates have a fanout of 1. This assumption is necessary as the technique employed here is essentially bit manipulation and we are trying to give a general formula for any size n. While this estimate gives a conservative estimate on the number of gates it is a fair assumption as the same criteria is applied to the techniques of section 4.2. Also most units of section 4.2 have fanouts that are not a function of n and so to allow an arbitrary fanout will not be fair as the size of a gate is also a function of the fanout.

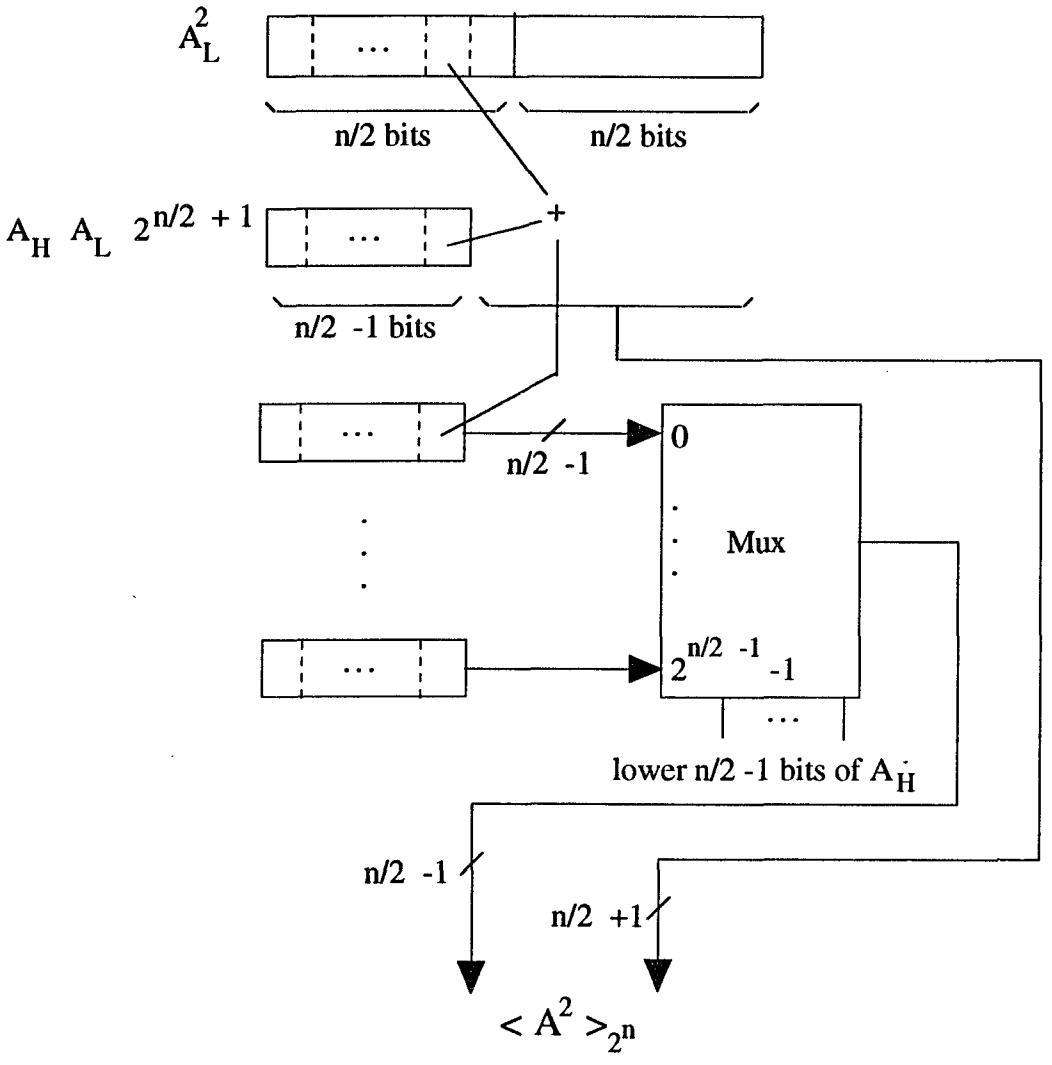


Figure 4.6: Basic scheme for techniques of section 4.1

The size of the overall multiplexer is determined as follows:

Let p be the number of select lines and q be the number of bits in each input data word. The number of 2-input gates for q multiplexers each of size $2^p \times 1$ is calculated as follows:

Since there are p select lines, there are 2^p minterms each containing p-bits. Each minterm will have $\log_2 p$ stages, thus giving the number of gates as $(2^0 + 2^1 + ... + 2^{\log_2 p - 1})$. Normally the select lines of all the multiplexers would be tied together implying that each minterm needs to be realized only once. However, since we are assuming a fanout of 1 we cannot use this fact and so these terms have to be realized q times. In each multiplexer, each minterm is combined with one bit of the input word. Since there are 2^p minterms the number of stages required to transfer one bit of the q-bit input to the output is equal to p, thus giving the number of gates as $(2^0 + 2^1 + ... + 2^p)$. Thus the total number of 2-input gates for the multiplexer is given by,

$$\text{mux hardware} = q2^p(2^0 + 2^1 + ... + 2^{\log_2 p - 1}) + q(2^0 + 2^1 + ... + 2^p) \qquad (4.25)$$

In our case $p = q = (n/2) - 1$. Plugging this into the above equation and simplifying we get

$$\text{Mux h/w} = (n/2 - 1)[2^{n/2 - 1}(2^{\log_2(n/2 - 1)} - 1) + (2^{n/2} - 1)] \qquad (4.26)$$

The computation for the number of gates required for obtaining each of the input words is based on a recursive formula and is thus not as straight forward as the above analysis. It is thus presented in detail. Let G_h denote the number of gates required for computing the input words to the multiplexer when the high word has h bits and let g_i denote the number of gates required for the modulo 2^n addition of two i-bit words.

For the case when the high word is one bit long there is no additional hardware. Thus $G_1 = 0$.

For the case when the high word is two bits long we require a 2×1 mux and one gate. Using the notation introduced in this section this gate performs the addition of r_0 and s_0. Thus $G_2 = 1$ and $g_1 = 1$.

For the case when the high word is three bits long we require a 4×1 mux and 10 gates. We explain below how the figure of 10 is obtained. The words of interest now are r_1r_0 and s_1s_0 while the following summations need to be performed.

s_1 s_0	r_1 r_0	s_1 s_0	s_1 s_0
$+ r_1$ r_0	$+ r_0$		$+ r_0$
	result ---------->	result	
Gates ⊕ ⊕	⊕	⊕ ⊕	⊕
req. ⊕ ∧		⊕ ∧	
4 gates	1 gate	4 gates	1 gate

Note that the summation of the last column is given by G_2. Thus the number of gates can be given as $G_3 = 4 + 1 + 4 + G_2 = 10$. This checks with equations (4.17) - (4.20). Here the first term is g_2 and is equal to 4.

For the case when the high word is four bits long we require an 8×1 mux and 60 gates. We explain below how the figure of 60 is obtained. The words of interest now are $r_2r_1r_0$ and $s_2s_1s_0$ while the following summations need to be performed.

1)

s_2	s_1	s_0
$+ r_2$	r_1	r_0
⊕	6	⊕
⊕		∧
10 gates		

The number 6 appears because to add s_1 and r_1 we need a full adder. (For a fanout of one a full adder needs 6 gates.)

2)

	r_2	r_1	r_0		s_2	s_1	s_0	
	r_1	r_0						
	result	---------->			result			
Gates req.	⊕	⊕			⊕	6	⊕	
	⊕	∧			⊕		∧	
	4 gates				10 gates			

3)

	r_2	r_1	r_0		s_2	s_1	s_0	
	r_0							
	result	---------->			result			
Gates req.	⊕				⊕	6	⊕	
					⊕		∧	
	1 gate				10 gates			

4)

	r_2	r_1	r_0		r_0		s_2	s_1	s_0	
	r_1	r_0								
	result	---------->			result	---------->	result			
Gates req.	⊕	⊕			⊕		⊕	6	⊕	
	⊕	∧					⊕		∧	
	4 gates				1 gate		10 gates			

5) In addition to the above we would need all the gates required for the case when the high word had three bits.

Thus the number of gates can be given as $G_4 = 10 + 4 + 10 + 1 + 10 + 4 + 1 + 10 + G_3 = 60$. Here the first term is g_3 and is equal to 10.

From the above discussion we see that for each decomposition the term g_i also represents the number of gates required for summing the two words of interest viz. $r_{h-1}r_{h-2}\ldots r_0$ and $s_{h-1}s_{h-2}\ldots s_0$. We let set $G = \{g_1, g_2, \ldots, g_{i-1}\}$ where $i = h-1$. Then the general formula for the number of gates required for all the input words of the multiplexer can be given for all $h > 2$, by

$$G_h = g_i + \binom{|G|}{1}g_i + \binom{|G|}{2}g_i + \cdots + \binom{|G|}{|G|}g_i + \sum_{k=1}^{i-1} g_k + \sum(g_k + g_l)\ \forall\{k,l \in 1, 2, \ldots, i-1\} + \ldots + \sum(g_1, g_2, \ldots, g_{i-1}) + G_{h-1} \qquad (4.27)$$

where $G_1 = 0$, $G_2 = 1$, $g_1 = 1$, $g_2 = 4$, and $g_i = 4 + 6(i-2)$ for $i > 2$ and $|\ G\ |$ is the cardinality of set G.

Thus the total hardware is given by

Total Hardware -Section (4.1) = Equation (4.26) + Equation (4.27) (4.28)

Table 4.5 lists the costs of the hardware for various values of n and also gives a cost comparison of the two sections. Here the hardware cost is based on equation (2.26) and not on equation (4.28) merely to illustrate the fact that in spite of ignoring the cost of equation (4.27), section 4.2 is far more cost efficient. Also, because of this we use the term minimum % savings as opposed to simply % savings.

We now present the analysis to compute the time delay associated with the computation of the squaring operation. The delay of the multiplexer is given by $1 + p + \log_2 p$ while the delay to compute the input terms of the mux is given by the time to obtain the mod 2^n sum of $r_{h-1}r_{h-2}\ldots r_0$ and $s_{h-1}s_{h-2}\ldots s_0$. The worst case delay arises when $r_{h-1}r_{h-2}\ldots r_0$ assumes its maximum value of $2^h - 1$. In such a situation h-1 summations have to be performed in a sequential fashion while the delay of each

Table 4.5: Cost comparison in 2-input gates of techniques of section 4.1 with 4.2

Word Length n	H/w cost of section 4.1 based on eqn (4.26)	H/w cost of section 4.2 based on eqn. (4.34)	Minimum % savings.
16	7161	1411	80.30
32	7864305	40339	99.48

summation is given by the time to compute the corresponding g_i's. The time delay for g_i is denoted by t_{gi}, while $t_{g1} = 1$, $t_{g2} = 2$, $t_{gi} = 2(i-1)$ for $i > 2$. Thus the total worst case delay is given by

$$T_h = 3 + \sum_{i=3}^{h-1} t_{gi} = (h-3)[(h-3)(h+2) -2] + 3 \quad \text{for } h > 2 \qquad (4.29)$$

We have assumed that the full adders are connected in a ripple fashion. Thus the total delay in gate delays is given by

$$1 + p + \log_2 p + (h-3)[(h-3)(h+2) -2] + 3 \qquad (4.30)$$

For the decomposition considered $p = q = (n/2) -1$ and $h = n/2$ and plugging these into the above equation we get

$$\text{Time Delay -Section (4.1)} = n/2 + (1/8)(n-6)[(n-6)(n+4)-8] + \log_2(n/2 -1) + 3 \qquad (4.31)$$

Table 4.6 lists the time delays for various values of n and also gives a delay comparison of the two sections. Once again, it is seen that techniques of section 4.2 are better than those of section 4.1.

4.4.2 Cost and speed analysis for section 4.2

The hardware cost of a modulo 2^n adder for $n > 2$ is given by $4 + 6(n-2)$. Referring to figure 4.5 there are 2 adders of size $n/2 + 1$, one adder of size n, and one adder of size n/2. (Note that the $A_H - A_L$ unit produces the result in 2's complement form.) Thus giving a total adder cost of $15n -13$. The cost of a ROM with L address lines can be given based on [55] as

$$\text{ROM Cost} = 2 \times (2^0 + 2^1 + \ldots + 2^{(\log_2 L/2) -1}) + (2^0 + 2^1 + \ldots + 2^{\log_2 2^{**}(L/2)}) \qquad (4.32)$$

Table 4.6: Speed comparison in 2-input gate delays of techniques of section 4.1 with 4.2

Word Length n	Time delay of section 4.1 based on eqn. (4.31)	Time delay of section 4.2 based on eqn. (4.35)	Ratio of time delays of sections 4.1 and 4.2
16	254	62	4.10
32	3039	126	24.12

However, since we assume that each minterm is realized independently the first term is to be multiplied by $2^{L/2}$. The second term has to be multiplied by n as the output of our ROM has n-bits. Here L = n/2, thus giving

$$\text{ROM Cost} = 2^{n/4\ -1} \times (5n - 4) - n \qquad (4.33)$$

Thus the total hardware cost can be given by

$$\text{Total Hardware -Section (4.2)} = 15n\text{-}13 + 2 \times \text{Equation (4.33)} \qquad (4.34)$$

Referring to figure 4.5 again the delay can be given by three levels of adders plus one level of ROMs. The total delay of the adders in gate delays is given by 4n-2 while the delay of the ROM in gate delays [55] is given by $2 + \log_2 (L/2) + \log_2 2^{(L/2)}$. However, referring to figure 4.6 we note that the same ROM delay is also associated with techniques of section 4.1. Therefore for comparison purposes we do not need to include the delay of the ROM unit. Thus we have

$$\text{Time Delay -Section(4.2)} = 4n - 2 \qquad (4.35)$$

Tables 4.5 and 4.6 summarize the costs and delays for various values of n and also compare them with the techniques of section 4.1.

We make the following observations:

1) From table 4.5 we see that techniques of section 4.2 result in considerable savings in hardware, of up to 99.48% , when compared with those of section 4.1. Note that in this table for section 4.1 we have only taken into account the cost of the multiplexer. From table 4.6 we see that techniques of section 4.2 also yield a much faster hardware, of up to approximately 20 times for a 32-bit word.

2) The bulk of the delay in equation (4.35) is due to the adder circuit. By using better adders such as the carry-look ahead adder the timing can be drastically improved for section (4.2) while it will make little difference for techniques of section (4.1) as, referring to equation (4.31), the adder delay here is $O(n^3)$. Also in section (4.1) the number of summands is a function of n while for section (4.2) it is a constant.

3) The ROM delay models used for section (4.2) are very conservative [55] as they do not take into account the density, regular implementation structure, e.t.c. while the model used for section (4.1) is very generous as it does not take into effect the delays of interconnection wiring.

4) A big advantage of section (4.2) is that it is very modular. Thus in a practical implementation a design change from n = 16 to say n = 32 will require much lesser design turn around time as only the blocks have to be changed while for techniques of section (4.1) a complete new set of schematics will have to be created.

4.5 Memory compression schemes for arithmetic in modulo 2^n -1

In this section we present the arithmetic manipulations required to compute the square of a number modulo 2^n -1. Our objective here again is to find ROM based efficient methods to compute the square of a number modulo 2^n -1. Let us consider a number A belonging to the modular ring $Z_{2^n-1} = \{0, 1, ..., 2^n -2\}$. Then A has a n-bit binary representation as in $A = a_{n-1}a_{n-2} \ldots a_1a_0$; $a_i \in \{0,1\}$. Our task is to compute $\langle A^2 \rangle_{2^n-1}$, where as usual $\langle x \rangle_m$ denotes the operation x modulo m. Our method is essentially the same as that outlined in section 4.1. Here we present the analysis for two different decompositions of the number A.

4.5.1 Analysis when the high word is one bit long

Let A_{H1} be a 1 bit word and A_{L1} be a n-1 bit word with $A_{H1} = a_{n-1}$ and $A_{L1} = a_{n-2} \ldots a_1 a_0$. Then the value of A is given by equation (4.1) and its square by equation (4.2). Evaluating equation (4.2) modulo 2^n -1 we get

$$\langle A^2 \rangle_{2^n-1} = \begin{cases} \langle A_{L1}^2 \rangle_{2^n-1} & \text{if } a_{n-1} = 0 \\ \langle 2^{n-2} + A_{L1} + A_{L1}^2 \rangle_{2^n-1} & \text{if } a_{n-1} = 1 \end{cases} \qquad (4.36)$$

In the computation of equation (4.36) we use the following: $<2^n>_{2^n-1} = 1$, $<2^{2n-2}>_{2^n-1} = 2^{n-2}$, and $A_{H1} = a_{n-1} \in \{0,1\}$. Further, the summation of 2^{n-2} and A_{L1}, can be given in binary as $a_{n-2}\overline{a_{n-2}}\, a_{n-3} \ldots a_1 a_0$, is less than 2^n-1 as both a_{n-2} and its complement $\overline{a_{n-2}}$ cannot be one at the same time.

We realize the term A_{L1}^2 using a ROM thus needing a ROM of size $2^{n-1} \times n$. Therefore we have a savings of 50% in ROM bits, however the overhead is one modulo 2^n -1 adder and a single 2×1 multiplexer of word length n. As seen from equation (4.36) the select line of the multiplexer is a_{n-1}. In the next section, by applying the same techniques, we analyze the effects of increasing the length of A_{H1} and reducing the length of A_{L1} on the savings in ROM bits.

4.5.2 Analysis when the high word is two bits long

Let A_{H2} be a 2 bit word and A_{L2} be a n-2 bit word with $A_{H2} = a_{n-1}a_{n-2}$ and $A_{L2} = a_{n-3} \ldots a_1 a_0$. Then the value of A is given by equation (4.4) and its square by equation (4.5). Evaluating equation (4.5) modulo 2^n -1 we get

$$<A^2>_{2^n-1} = <2^{n-4} A_{H2}^2 + A_{H2} A_{L2}\, 2^{n-1} + A_{L2}^2>_{2^n} \qquad (4.37)$$

In the above we have used the simplification $<2^{2n-4}>_{2^n-1} = 2^{n-4}$. The possible values of $<2^{n-4}A_{H2}^2 + A_{H2}A_{L2}\,2^{n-1}>_{2^n-1}$ are shown in Table 4.7 while the following can be observed:

Case 1: $a_{n-1}a_{n-2} = 01.$

In this case, according to Table 4.7, $<2^{n-4} + A_{L2}2^{n-1}>_{2^n-1}$ needs to be computed. Since $A_{L2} = a_{n-3} \ldots a_0$, $A_{L2}2^{n-1} = (a_{n-3} \ldots a_1a_0)2^{n-1} = (a_{n-3} \ldots a_1)2^n + a_0 2^{n-1}$ and $<A_{L2}2^{n-1}>_{2^n-1} = <(000a_{n-3} \ldots a_1) + (a_000 \ldots 00)>_{2^n-1} = a_000a_{n-3} \ldots a_1$. One step further we observe that

$$<2^{n-4} + A_{L2}2^{n-1}>_{2^n-1} = <(00010 \ldots 0) + (a_000a_{n-3} \ldots a_1)>_{2^n-1}$$

$$= a_00a_{n-3}\overline{a_{n-3}}\,a_{n-4} \ldots a_1 \qquad (4.38)$$

Case 2: $a_{n-1}a_{n-2} = 10.$

In this case the desired $<2^{n-2} + A_{L2}>_{2^n-1}$ is given by

$$<2^{n-2} + A_{L2}>_{2^n-1} = <(010 \ldots 0) + (00a_{n-3} \ldots a_0)>_{2^n-1}$$

$$= 01a_{n-3} \ldots a_0 \qquad (4.39)$$

Case 3: $a_{n-1}a_{n-2} = 11.$

Here we need to compute $<2^{n-1} + 2^{n-4} + A_{L2} + A_{L2}2^{n-1}>_{2^n-1}$. It is easy to see that summation of equations (4.38), (4.39), and the quantity 2^{n-2} yields the desired result. We thus have

$$<2^{n-1} + 2^{n-4} + A_{L2} + A_{L2}2^{n-1}>_{2^n-1} = <(a_00a_{n-3}\overline{a_{n-3}}\,a_{n-4} \ldots a_1) +$$

$$(10a_{n-3} \ldots a_1a_0)>_{2^n-1} \qquad (4.40)$$

Table 4.7: Values of $<2^{n-4}A_{H2}^2 + A_{H2}A_{L2}\,2^{n-1}>_{2^n-1}$

A_{H2}		$<2^{n-4}A_{H2}^2 + A_{H2}A_{L2}\,2^{n-1}>_{2^n-1}$
a_{n-1}	a_{n-2}	
0	0	0
0	1	$<2^{n-4} + A_{L2}2^{n-1}>_{2^n-1}$
1	0	$<2^{n-2} + A_{L2}>_{2^n-1}$
1	1	$<2^{n-1} + 2^{n-4} + A_{L2} + A_{L2}2^{n-1}>_{2^n-1}$

Finally, combining Table 4.7 and equations (4.38) - (4.40) we get

$$\langle A^2\rangle_{2^n-1} = \begin{cases} \langle A_{L2}^2\rangle_{2^n-1} & \text{if } a_{n-1}a_{n-2} = 00 \\ \langle (a_0 0 a_{n-3}\overline{a_{n-3}}a_{n-4}\cdots a_1) + A_{L2}^2\rangle_{2^n-1} & \text{if } a_{n-1}a_{n-2} = 01 \\ \langle (01 a_{n-3}a_{n-4}\cdots a_1) + A_{L2}^2\rangle_{2^n-1} & \text{if } a_{n-1}a_{n-2} = 10 \\ \langle (a_0 0 a_{n-3}\overline{a_{n-3}}a_{n-4}\cdots a_1) + \\ (10 a_{n-3}\cdots a_1 a_0) + A_{L2}^2\rangle_{2^n-1} & \text{if } a_{n-1}a_{n-2} = 11 \end{cases} \quad (4.41)$$

We realize the term A_{L2}^2 using a ROM thus needing a ROM of size $2^{n-2} \times n$. Therefore we have an increase in savings to 75% in ROM bits, however the overhead is two modulo 2^n -1 adders and a single 4 × 1 multiplexer of word length n. As seen from equation (4.41) the select lines of the multiplexer are $a_{n-1}a_{n-2}$. It is easy to see that if this method is increased further the savings in ROM bits will increase but at the same time the number of adders and the size of the multiplexer will also increase.

4.6 Optimized memory compression schemes for arithmetic in modulo 2^n -1

We present the analysis when n is even. Once again, we consider a number A belonging to the modular ring $Z_{2^n-1} = \{0, 1, ..., 2^n - 2\}$. Then A has a n-bit binary representation as in $A = a_{n-1}a_{n-2} \ldots a_1a_0$; $a_i \in \{0,1\}$. Let $A_H = a_{n-1}a_{n-2} \ldots a_{n/2}$ and $A_L = a_{(n/2)-1}a_{(n/2)-2} \ldots a_1a_0$. Then the value of A is given by equation (4.21) and its square by equation (4.22). Evaluating equation (4.21) modulo 2^n -1 we get

$$\langle A^2\rangle_{2^n-1} = \langle A_H^2 + A_L^2 + A_H A_L\, 2^{n/2+1}\rangle_{2^n-1} \quad (4.42)$$

Comparing this with equation (4.23) we find that it is very similar except for the fact that this has the additional term A_H^2 which in turn can be realized using a ROM of size $2^{n/2} \times n$. The other terms can be realized as outlined in section 4.2.1. Also note that

the ROM for A_H^2 is identical to the ROM for A_L^2. The similarities between equation (4.23) and (4.42) yields an overhead that is very streamlined with respect to implementation and is thus suited for VLSI implementation. We should note that arithmetic in this ring makes use of the following: $<2^n>_{2^n-1} = 1$, $<2^{n+1}>_{2^n-1} = 2$. Tables 4.3 and 4.4 summarize the results for the cases when n is even and n is odd.

4.7 Memory compression schemes for arithmetic in modulo 2^n+1

The number A that needs to be squared now belongs to the ring $Z_{2^n+1} = \{0, 1, ..., 2^n\}$. If $A = 2^n$, then A has an (n+1)-bit representation as in $A = 100 ...00 = <-1>_{2^n+1}$ and $<A^2>_{2^n+1} = 1$. For all the other cases A assumes an n-bit binary representation as in $A = a_{n-1}a_{n-2} ... a_1a_0$; $a_i \in \{0,1\}$. Considering the decomposition of A into $A_{H1} = a_{n-1}$ and $A_{L1} = a_{n-2} ...a_0$, the following equation can be derived on lines similar to that of equation (4.36).

$$\langle A^2 \rangle_{2^n+1} = \begin{cases} \langle A_{L1}^2 \rangle_{2^n+1} & \text{if } a_{n-1} = 0 \\ \langle -(a_{n-2}\overline{a_{n-2}}a_{n-3}\cdots a_0) + A_{L1}^2 \rangle_{2^n+1} & \text{if } a_{n-1} = 1 \end{cases} \quad (4.43)$$

4.8 Optimized memory compression schemes for arithmetic in modulo 2^n+1

We once again present the analysis when n is even. We consider a number A belonging to the modular ring $Z_{2^n+1} = \{0, 1, ..., 2^n\}$. Again, if $A = 2^n$, then A has an (n+1)-bit representation as in $A = 100 ...00 = <-1>_{2^n+1}$ and $<A^2>_{2^n+1} = 1$. For all the other cases A assumes an n-bit binary representation as in $A = a_{n-1}a_{n-2} ... a_1a_0$; $a_i \in \{0,1\}$.Then A has a n-bit binary representation as in $A = a_{n-1}a_{n-2} ... a_1a_0$; $a_i \in \{0,1\}$. Let $A_H = a_{n-1}a_{n-2} ... a_{n/2}$ and $A_L = a_{(n/2)-1}a_{(n/2)-2} ... a_1a_0$. Then the value of A is

given by equation (4.21) and its square by equation (4.22). Evaluating equation (4.21) modulo 2^n+1 we get

$$<A^2>_{2^n+1} = <-A_H^2 + A_L^2 + A_H A_L\, 2^{n/2+1}>_{2^n+1} \qquad (4.44)$$

Comparing this with equation (4.42) we find that it is very similar except for the fact that the term A_H^2 is negative. But this does not need any extra hardware as the ROM used to realize this term can directly generate the negative result. Therefore, the amount of hardware required for realizing this equation is the same as that required for realizing equation (4.42) plus a 2×1 multiplexer. The select line of this multiplexer is bit a_n and if this bit is equal to 1 then the output is set to one as explained before while if it is zero the output is the result of equation (4.44). The similarities between equations (4.23), (4.42), and (4.44) again suggests that the overhead very streamlined with respect to implementation and is thus well suited for VLSI implementation. We should note that arithmetic in this ring makes use of the following: $<2^n>_{2^n+1} = -1$, and $<2^{n+1}>_{2^n+1} = -2$. Tables 4.3 and 4.4 summarize the results for the cases when n is even and n is odd. From these tables it is clearly seen that these techniques are also ideally suited for building an integrated squarer, i.e. a unit that can compute either one of three operations viz. $<A^2>_{2^n}$, $<A^2>_{2^n-1}$, or $<A^2>_{2^n+1}$. Note that A_L^2 is inherently a n-bit representation and thus the same unit can be used in all three computations.

4.9 Conclusions

In this chapter we have presented in detail two ROM based methods that compute the squaring operation in modular rings. When compared with traditional techniques, both techniques reduce the number of ROM bits significantly. However, for a fair comparison of the two techniques the cost of the overhead must be included and in the ensuing analysis we show that techniques of section 4.2 are very optimal in all

respects viz. cost, speed, and regularity of the hardware structure. The techniques of section 4.2 are very systematic and result in a modular design, i.e.,

i) a modulo 2^n squarer unit can be easily extended to a modulo $2^n - 1$ or modulo $2^n + 1$ squarer and

ii) design changes for different values of n are minimal.

While we have not presented the comparative analysis for arithmetic in modulo $2^n - 1$ and modulo $2^n + 1$, one can see from equations (4.36)-(4.41) and (4.43) that the techniques of section 4.2 will yield optimal results as the techniques of section 4.1 require the use of modulo adders and multiplexers whose input words have a length of n bits. Also, the number of adders required is a function of the decomposition length. We also note that the cost of computing the overhead in these rings is far simpler than when the arithmetic is performed modulo 2^n. This is because the size of the multiplexer data words is always the same, i.e. it is not a function of the length of decomposition.

Chapter 5

Conclusions

In this chapter we first summarize the results of this dissertation and then discuss avenues for further research initiated by this effort.

5.1 Summary

In this dissertation we have developed algorithms for obtaining the cyclic convolution of two n-point sequences where n is a power of two, with no restriction on the size of each point. These algorithms rely only on square, add, and subtract operations. All the necessary theory for computing the cyclic convolution operation is developed in chapter 2. The correctness of these algorithms is based on eight theorems also developed in chapter 2. We have also derived non-recursive formulae for the count on additions and squaring operations. These formulae show that while we decrease the number of squaring operations we increase the number of addition operations. Issues relating to CSA and ROM based implementations were discussed in detail in chapter 3. The main purpose of this exercise was to demonstrate that the increase in the number of addition operations does not negate the decrease in the number of squaring operations. Results of the chapter prove convincingly the usefulness of *squared law algorithms*. Further, we have shown that our methods are far more superior than traditional methods when ROMs are used. Our methods also result in modular implementations and exhibit properties that can be exploited by clever architectural designs to obtain elegant and efficient implementations. Our methods also do not introduce any round-off errors and thereby eliminate the need for

error correction hardware. Some interesting observations were found in CSA based implementations of squarers and these along with schemes for multiplying two numbers based on the cyclic convolution operation were also presented in chapter 3. In chapter 4, the behavior of the squaring operation when computed in modular rings was examined. Two methods of this computation were presented and we have clearly shown that one is far better than the other, both in terms of speed and cost.

5.2 Future research emphasis

Since we have shown the usefulness of *squared law algorithms* in applications of digital signal processing and error control coding, further work can be classified under research and development.

Research: The research emphasis can be on finding other multiplication intensive environments and deriving similar algorithms. For instance, some of the other useful operations are linear convolutions, skew-cyclic convolutions [35], and higher order correlations [64]. Since skew-cyclic convolutions are less symmetric than cyclic convolutions, algorithms developed on lines similar to that of this dissertation are likely to be less efficient. However, no such hypothesis can be made for triple correlations. While triple-order correlations contain more information on the signal they also require more computation. Thus, it might be useful to explore the applicability of our methods in these computations.

Development: For some specific needs hardware units and software programs can be developed around our algorithms and their performance can be compared with existing products that have the same goals. Existing processors and routines do not exploit the properties of the squaring operation and multi-operand additions. Thus, for a fair comparison new units and routines have to be created.

References

[1] R. E. Blahut, "Algebraic fields, signal processing, and error control," Proceedings of the IEEE, vol. 73, no. 5, pp. 874-893, May 1985.

[2] A.V. Oppenheim and R.W. Schafer, Digital Signal Processing. Englewood Cliffs, NJ: Prentice Hall, 1975.

[3] E. R. Berlekamp, Algebraic Coding Theory. New York: McGraw-Hill, 1968.

[4] J. H. McClellan and C. M. Rader, Number Theory in Digital Signal Processing. Englewood Cliffs, NJ: Prentice Hall, 1979.

[5] M.A. Soderstrand, W.K. Jenkins, G.A. Jullien and F.J. Taylor, Eds., Residue Number System Arithmetic: Modern Applications in Digital Signal Processing. New York: IEEE Press, 1986.

[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine computation of complex Fourier series," Mathematics of Computation, vol. 19, pp. 297-301, 1965.

[7] R. C. Agarwal and C. S. Burrus, "Fast convolution using Fermat number transforms with applications to digital filtering," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-22, pp. 87-97, April 1974.

[8] R. C. Agarwal and C. S. Burrus, "Number theoretic transforms to implement fast digital convolution," Proceedings of the IEEE, vol. 63, no. 4, pp. 550-560, April 1975.

[9] I. S. Reed and T. K. Truong, "The use of finite fields to compute convolutions," IEEE Transactions on Information Theory, vol. IT-21, pp. 208-213, 1975.

[10] H. J. Nussbaumer, "Digital filtering using polynomial transforms," Electronics Letters, vol. 13, pp. 386-387, 1977.

[11] B. Rice, "Some good fields and rings for computing number theoretic transforms," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-27, no. 4, pp. 432-433, August 1979.

[12] B. Gold and C. M. Rader, Digital Signal Processing of Signals. New York: McGraw-Hill: 1969.

[13] C. S. Burrus, "Block realization of digital filters," IEEE Transactions on Audio Electroacoustics, vol. AU-20, pp. 230-235, October 1972.

[14] T. G. Stockham, "High speed convolution and correlation," Proceedings of the AFIPS Conference, 1966 Joint Computer Conference, vol. 28, pp. 229-233.

[15] D. H. Lehmer, "Large-scale digital calculating machinery," Proceedings of the 2nd Symposium, Cambridge, MA: Harvard University Press, 1951, pp. 141-146.

[16] D. E. Knuth, The Art of Computer Programming, vol. 2, Semi-numerical Algorithms. Reading, MA: Addison-Wesley, 1969.

[17] A. V. Oppenheim and R. W. Schafer, Discrete-Time Signal Processing. Englewood Cliffs, NJ: Prentice Hall, 1989.

[18] L. R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing. Englewood Cliffs, NJ: Prentice Hall, 1975.

[19] R.W. Ramirez, The FFT Fundamentals and Concepts. Englewood Cliffs, NJ: Prentice Hall, 1985.

[20] R. E. Blahut, Fast Algorithms for Digital Signal Processing. Reading, MA: Addison-Wesley, 1985.

[21] J. M. Pollard, "The fast Fourier transform in a finite field," Mathematics of Computation, vol. 25, pp. 365-374, April 1971.

[22] I. J. Good, "The interaction algorithm and practical Fourier analysis,"Journal of the Royal Statistics Society, Section B-20, 361-375, 1958 and B-22, 372-375, 1960.

[23] L. H. Thomas, "Using a computer to solve problems in physics," in Application of Digital Computers. Boston, MA: Ginn and Co., 1963.

[24] S. Winograd, "On computing the discrete Fourier transform," Proceedings of the National Academy of Sciences, USA, vol. 73, pp. 1005-1006, 1976.

[25] A. V. Oppenheim and C. Weinstein, "Effects of finite register length in digital filtering and the fast Fourier transform," Proceedings of the IEEE, vol. 60, pp. 957-976, August 1972.

[26] A. Despain, "Very fast Fourier transform algorithms hardware for implementation," IEEE Transactions on Computers, vol. C-28, no. 5, May 1979.

[27] J. Guo, C. Liu, and C. Jen, "The efficient memory-based VLSI array designs for DFT and DCT," IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing, vol. 39, no. 10, pp. 723-733, October 1992.

[28] C. D. Thompson, "Fourier transforms in VLSI," IEEE Transactions on Computers, vol. C-32, no. 11, pp. 1047-1057, November 1983.

[29] J. Allen, "Computer architecture for signal processing," Proceedings of the IEEE, vol. 63, no. 4, pp. 624-632, April 1975.

[30] G. Ma and F. J. Taylor, "Multiplier policies for digital signal processing," IEEE ASSP Magazine, pp. 6-20, January 1990.

[31] J. A. Beraldin, T. Aboulnasr, and W. Steenaart," Efficient one-dimensional systolic array realization of discret Fourier transform," IEEE Transactions on Circuits and Systems, vol. 36, pp. 95-100, January 1989.

[32] L. W. Chang and M. Y. Chen, "A new systolic array for discrete Fourier transform, "IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-36, pp. 1665-1667, October 1988.

[33] C. M. Liu and C. W. Jen, "A new systolic array algorithm for discrete Fourier transform," Proceedings of ISCAS, pp. 2212-2215, 1991.

[34] H.J. Nussbaumer, "Relative evaluation of various number theoretic transforms for digital filtering applications," IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-26, pp. 88-93, February 1978.

[35] H.J. Nussbaumer, Fast Fourier Transform and Convolution Algorithms. Berlin, FRG: Springer Verlag1982.

[36] D.F. Elliot and K.R. Rao, Fast Transforms, Algorithms, Analyses, Applications. New York: Academic Press, 1982.

[37] N.S. Szabo and R.I. Tanaka, Residue Arithmetic and its Applications to Computer Technology. McGraw-Hill, NY: 1967.

[38] F.J. Taylor, "Residue arithmetic: A tutorial with examples," IEEE Computer, vol. 17, no. 5, pp. 50-62, May 1984.

[39] S.H. Leung, "Application of residue number systems to complex digital filters," Proceedings of the Fifteenth Asilomar Conference on Circuits, Systems and Computers, Pacific Grove, CA, November 1981, pp. 70-74.

[40] J.V. Krogmeier and W.K. Jenkins, "Error detection and correction in quadratic residue number systems," Proceedings of the 26th Midwest Symposium on Circuits and Systems, Puebla, MX, August 1983, pp. 408-411.

[41] F.J. Taylor, G. Papadourakis, A. Skavantzos and A. Stouraitis, "A radix-4 FFT using complex RNS arithmetic," IEEE Transactions on Computers, vol. C-34, no. 6, pp. 573-576, June 1985.

[42] A. Skavantzos and F.J. Taylor, "On the polynomial residue number system," IEEE Transactions on Signal Processing, vol. 39, no. 2, pp. 376-382, February 1991.

[43] A. Skavantzos and N. Mitash, "Implementation issues of 2-dimensional polynomial multipliers for signal processing using residue arithmetic," IEE Proceedings-E, vol. 140, no. 1, pp. 45-53, January 1993.

[44] G.A. Jullien : "Residue number scaling and other operations using ROM arrays," IEEE Transacations on Computers, pp. 325-336, April 1978.

[45] C.H. Huang and F.J. Taylor, "A memory compression scheme for modular arithmetic," IEEE Transactions on Acoustics Speech Signal Processing, ASSP-27, vol. 6, pp. 608-611, December 1979.

[46] G.A. Jullien : "Implementation of multiplication, modulo a prime number, with applications to number theoretic transforms," IEEE Transacations on Computers, vol. C-29, pp. 899-905, October 1980.

[47] F.J. Taylor, "Large moduli multipliers for signal processing," IEEE Transactions on Circuits and Systems, vol. CAS-28, no. 7, pp. 731-736, July 1981.

[48] M.A. Soderstrand and E.L. Fields, "Multipliers for residue number arithmetic digital filters," Electronics Letters, vol. 13, no. 6, pp. 164-166, March 1977.

[49] M.A. Soderstrand and C. Vernia, "A high-speed low-cost modulo P_i multiplier with RNS arithmetic applications," Proceedings of the IEEE, vol. 68, no. 4, pp. 529-532, April 1980.

[50] A. Skavantzos, "Novel approach for implementing convolutions with small tables," IEE Proceedings-E, vol. 138, no. 4, pp. 255-259, July 1991.

[51] A. Skavantzos and P. B. Rao, "New multipliers modulo 2^N -1," IEEE Transactions on Computers, vol. 41, no. 8, pp. 957-961, August 1992.

[52] P. B. Rao and A. Skavantzos, "New multiplier designs based on squared law algorithms and table look-ups," Proceedings of the 26th Annual Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, October 1992, pp. 686-690.

[53] P. B. Rao and A. Skavantzos, "Efficient computation of squaring operation in modular rings," Electronics Letters, vol. 28, no. 17, pp. 1628-1630, August 1992.

[54] L. Dadda, "Some schemes for parallel multipliers," Alta Frequenza, vol. 34, pp. 349-356, 1965.

[55] S. Waser and M. J. Flynn, Introduction to Arithmetic for Digital Systems Designers. Orlando, FL: Holt, Rinehart and Winston, 1982.

[56] K. Hwang, Computer Arithmetic. New York: John Wiley and Sons, 1979.

[57] T. C. Chen, "A binary multiplication scheme based on squaring," IEEE Transactions on Computers, vol. C-20, pp. 678-680, June 1971.

[58] T. Jayashree and D. Basu, "On binary multiplication using the quarter squared algorithm," IEEE Transactions on Computers, pp. 957-960, September 1976.

[59] L. Dadda, "Squarers for binary numbers in serial form," Proceedings of the 7th Symposium on Computer Arithmetic, Urbana, IL, June 1985, pp. 173-180.

[60] H. Kobayashi, "A multioperand two's complement addition algorithm," Proceedings of the 7th Symposium on Computer Arithmetic, Urbana, IL, June 1985, pp. 16-17.

[61] A.V. Curiger, H. Bonnenberg, and H. Kaeslin, "Regular VLSI architectures for multiplication modulo (2^n +1)," IEEE Journal of Solid-State Circuits, vol. 26, no. 7, pp. 990-994, July 1991.

[62] X. Lai and J.L. Massey, "A proposal for a new block encryption standard," presented at EUROCRYPT '90, Aarhus, Denmark, May 1990.

[63] A. Skavantzos, "ROM table reduction techniques for computing the squaring operation using modular arithmetic," Proceedings of the 25th Asilomar Conference on Circuits, Systems, and Computers, Pacific Grove, CA, October 1991, pp. 413-417.

[64] A. W. Lohmann and B. Wirnitzer, "Triple corrleations," Proceedings of the IEEE, vol. 72, no. 7, pp. 889-901, July 1984.

Appendix

Implementations in Mathematica

Algorithm 2.3:

```
Clear[x,z,q,j];
n:=32;
i:=0;
r:=0;
done = False;
x[i_,r_] := x[i,r] := 4 * Sum[c[2k],{k,0,(n/2)-1}];
x[0,0];
Print[x[0,0]];
r:=1;
j[r_] := 2^r;
z[i_,r_] := z[i,r] := 4 * Sum[(c[i+ k j[r]] * (-1)^k),{k,0,(n/j[r])-1}];
x[i_,r_] := x[i,r] := Simplify[x[i,r-1] + z[i,r]*2^(r-1)];
q[i_,r_] := x[i+j[r],r] := Simplify[x[i,r-1] - z[i,r]*2^(r-1)];

While[!done,
        Print[i,r];
        z[i,r];
        Print[z[i,r]];
        x[i,r];
        Print[x[i,r]];
```

```
q[i,r];
Print[x[i+j[r],r]];

        If[(j[r]==(n/2)),
            cy[i] = x[i,r]/(4*2^r);
            cy[i+j[r]] = x[(i+j[r]),r]/(4*2^r);
            Print[cy[i]];
            Print[cy[i+j[r]]];
            i = i + 2;
            If[i <= (n/2 -1),
                r = Floor[N[Log[2, i]]] + 1;
                ,
                done = True;
             ]
            ,
            If[j[r] <= (n/2),
                r = r + 1;
                ,
             ]
         ]
]
```

Output when the program is run for n =32:

4 (c[0] + c[2] + c[4] + c[6] + c[8] + c[10] + c[12] + c[14] + c[16] +
c[18] + c[20] + c[22] + c[24] + c[26] + c[28] + c[30])

01

4 (c[0] - c[2] + c[4] - c[6] + c[8] - c[10] + c[12] - c[14] + c[16] -
c[18] + c[20] - c[22] + c[24] - c[26] + c[28] - c[30])

8 (c[0] + c[4] + c[8] + c[12] + c[16] + c[20] + c[24] + c[28])

8 (c[2] + c[6] + c[10] + c[14] + c[18] + c[22] + c[26] + c[30])

02

4 (c[0] - c[4] + c[8] - c[12] + c[16] - c[20] + c[24] - c[28])

16 (c[0] + c[8] + c[16] + c[24])

16 (c[4] + c[12] + c[20] + c[28])

03

4 (c[0] - c[8] + c[16] - c[24])

32 (c[0] + c[16])

32 (c[8] + c[24])

04

4 (c[0] - c[16])

64 c[0]

64 c[16]

c[0]

c[16]

22

4 (c[2] - c[6] + c[10] - c[14] + c[18] - c[22] + c[26] - c[30])

16 (c[2] + c[10] + c[18] + c[26])

16 (c[6] + c[14] + c[22] + c[30])

23

4 (c[2] - c[10] + c[18] - c[26])

32 (c[2] + c[18])

32 (c[10] + c[26])

24

4 (c[2] - c[18])

64 c[2]

64 c[18]

c[2]

c[18]

43

4 (c[4] - c[12] + c[20] - c[28])

32 (c[4] + c[20])

32 (c[12] + c[28])

44

4 (c[4] - c[20])

64 c[4]

64 c[20]

c[4]

c[20]

63

4 (c[6] - c[14] + c[22] - c[30])

32 (c[6] + c[22])

32 (c[14] + c[30])

64

4 (c[6] - c[22])

64 c[6]

64 c[22]

c[6]

c[22]

84

4 (c[8] - c[24])

64 c[8]

64 c[24]

c[8]

c[24]

104

4 (c[10] - c[26])

64 c[10]

64 c[26]

c[10]

c[26]

124

4 (c[12] - c[28])

64 c[12]

64 c[28]

c[12]

c[28]

144

4 (c[14] - c[30])

64 c[14]

64 c[30]

c[14]

c[30]

?cy

Global`cy

cy[0] = c[0]

cy[2] = c[2]

cy[4] = c[4]

cy[6] = c[6]

cy[8] = c[8]

cy[10] = c[10]

cy[12] = c[12]

cy[14] = c[14]

cy[16] = c[16]

cy[18] = c[18]

cy[20] = c[20]

cy[22] = c[22]

cy[24] = c[24]

cy[26] = c[26]

cy[28] = c[28]

cy[30] = c[30]

Algorithm 3.1:

```
Clear[c,t];
c= {1,2,3,4,5,6,7,8,7,6,5,4,3,2,1};
Print[c];
temp = Length[c];
ha = 0;
fa = 0;
ha1 = 0;
fa1 = 0;
ex = Ceiling[Log[2, Max[c]]];
Do[c=AppendTo[c,0], {ex}];
jm = Length[c];
i = 1;
t[1] = 2;
While[t[i] < Max[c],
i = i+1;
t[i_] := t[i] = Floor[3/2 * t[i-1]]
      ]
i = i-1;
le = i;
j = 1;
While[i > 0,
          While[j < jm,
          If[(c[[j]] > t[i]),
               fa1 = Floor[(c[[j]] - t[i])/2];
```

```
            ha1 = Ceiling[(c[[j]] - t[i])/2] - fa1;
            c[[j]] = t[i];
            c[[j+1]] = c[[j+1]] + fa1 + ha1;
            fa = fa + fa1;
            ha = ha + ha1;
            ,
          ];
        j = j + 1;
        ];
      j = 1;
      i = i - 1
      ]
zero = Take[c, {temp + 1, Length[c]}];
cpa = Length[c] - Count[zero,0];
facpa = cpa -1;

Print["# of Full Adders = ",fa];
Print["# of Half Adders = ",ha];
fa = fa + facpa;
ha = ha + 1;
Print["# of Full Adders including CPA = ",fa];
Print["# of Half Adders including CPA = ",ha];
Print["# of CSA Levels  = ",le];
Print["# of CPA Levels  = 1"];
Print["Size of CPA = ",cpa];
gates = 5*fa + 2*ha;
Print["Number of 2-input gates including CPA = ",gates]
```

Output when program is run:

{1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1}

# of Full Adders = 35

# of Half Adders = 7

# of Full Adders including CPA = 49

# of Half Adders including CPA = 8

# of CSA Levels = 4

# of CPA Levels = 1

Size of CPA = 15

Number of 2-input gates including CPA = 261

Note: In this example the input numbers to be added are the partial products obtained when two eight bit numbers are multiplied. The results check with a conventional calculation, as shown in figure 3.1.

Vita

Poornachandra B. Rao received the B.E. degree from Osmania University, India, in 1984, and the M.S. degree from Louisiana State University in 1989, both in Electrical Engineering. From 1984-1987, he worked with Larsen & Toubro Ltd., India as an Electrical Systems Design Engineer. He has also held summer research positions at Rühr University, Germany, in 1989 and Circuit Technology Group, Hewlett-Packard, in 1991. Currently, he is a candidate for the doctoral degree in the Department of Electrical and Computer Engineering at Louisiana State University. His research interests include application specific integrated circuit design, computer arithmetic, and parallel processing. He is a member of IEEE Computer Society and Eta Kappa Nu.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Poornachandra B. Rao

Major Field: Electrical Engineering

Title of Dissertation: Squared Law Algorithms: Theory and Applications

Approved:

Alex Stavrakoudis
Major Professor and Chairman

Daniel Fogel
Dean of the Graduate School

EXAMINING COMMITTEE:

J Bares

A. Elamawy

Subhash Kak

Sri Ch k

Charles Thorpe

Date of Examination:

June 9, 1993