DESIGN AND MODELING OF HIGH SPEED MODULO MULTIPLIERS FOR CRYPTOSYSTEMS

BY

Muhammad Yahya Imam Mahmoud

A THESIS PRESENTED TO THE

DEANSHIP OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

KING FAHD UNIVERSITY
OF PETROLEUM & MINERALS

May 2004

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by Muhammad Yahya Imam Mahmoud under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN COMPUTER ENGINEERING.

Thesis Committee

Dr. Alaaeldin Amin (Advisor)

Prof. Mostafa Abd-El-Barr (Member)

Dr. Muhammad F. Khan (Member)

Prof. Sadiq M. Sait
(Department Chairman)

Dr. Mohammad Al-Ohali
(Dean of Graduate Studies)

May 2004

ACKNOWLEDGMENT

Praise be to ALLAH the Lord of the universe who has created mankind and made them into tribes and nations, that they may know each other. Peace be upon the Prophet Muhammad, his family, his companions, and all those who followed him until the day of judgment.

ALLAH said (9-105):

> “And say: Work (righteousness): Surly will ALLAH observe your work, so will his messenger, and the believers.”

I would like to express my sincere and deepest gratitude to my advisor Dr. Alaaeldin Amin for his constant help, personal attention, inspiring guidance, suggestions, and encouragement throughout the period of this research. I also would like to express my sincere appreciation to Dr. Mostafa Abd-El-Barr and Dr. Muhammad F. Khan who have given me invaluable help and support. I also wish to thank my colleagues at KFUPM for their encouragement and good will wishes.

TABLE OF CONTENTS

LIST OF TABLES

TABLE Page

LIST OF FIGURES

Figure Page

LIST OF ALGORITHMS

THESIS ABSTRACT

Name: Muhammad Yahya Imam Mahmoud

Title: Design and Modeling of High Speed Modulo Multipliers for Cryptosystems

Major Field: Computer Engineering

Date of Degree: May 2004

Advances in networking and data processing speeds have led to the need for high-speed *cryptosystems*. The speed of a *cryptosystem* is function of its complexity and the technology used to implement it. This work investigates the techniques of designing *fast modulo multipliers* since *modulo-multiplication* is a basic essential operation in *public-key cryptography*. Two types of *modulo multipliers* have been designed and modeled using VHDL and MatLab. While the first multiplier is based on *asynchronous* adder design, the other multiplier is based on *four-to-two compressor* design. In addition, a **B**uilt **I**n **S**elf **T**est (BIST) methodology has been developed for the *Compressor* based multiplier design. The two multiplier designs have been evaluated and compared based on their *area-delay* cost.

THESIS ABSTRACT (ARABIC)

ملخص الرسالة

الاســـــــــم: محمد يحيى إمام محمود

عنوان الرسالة: تصميم ومحاكاة ضاربة باقي القسمة عالية السرعة لأنظمة التشفير.

التخصـــــــص: هندسة الحاسب الآلي

تاريخ التخـرج: ربيع ثاني 1425 هـ

أوجد التطور المتزايد في عالم الشبكات ومعالجة البيانات الحاجة لتطوير أنظمة تشفير عالية السرعة، بينما تعتمد سرعة أي نظام للتشفير على مدى صعوبة وتعقيد النظام المستخدم وعلى التقنية المستخدمة في تصنيعه. ترتكز تقنيات التشفير المعتمدة على المفتاح المعلن على عملية الضرب وباقي القسمة. تستكشف هذه الأطروحة بعض الأساليب المتبعة في تصميم ضاربة باقي القسمة عالية السرعة وتتضمن تصميم ومحاكاة نوعين منها باستخدام اللغة الوصفية لبرمجة مكونات الحاسب الآلي (VHDL) ولغة (MatLab). بينما يعتمد تصميم الضاربة الأولى على المجمِّع غير المتزامن، يعتمد التصميم الآخر على المجمع الموفِّر للانتشـار. تضمنت هذه الأطروحة نظام فحص ذاتي للضاربة الأخيرة كما تم تقييم التصميمين بناءً على تكلفة الزمن في المساحة.

CHAPTER 1

INTRODUCTION

In the information age, the age of public electronic connectivity, as computer systems and their internetworking grow in complexity, the dependence on secure data storage and exchange has become critical. Data security and integrity is threatened by the increased activity of hackers, electronic fraud, and eavesdropping. This has led to a need for protecting and authenticating access to data and their resources. There has been no age where data security and integrity have received as much attention as this age. Military applications, financial transactions, and multimedia communications, are examples that require authentication and data protection algorithms [53].

Public-key cryptosystems, which are based on one way mathematical functions, are becoming very popular because they do not need complex key distribution mechanisms. Based on modulo operations, the RSA [43] and Elgamal [48] encryption algorithms are examples of public-key crypto-algorithms. The speed of a crypto-algorithm and its hardware cost are important performance measures particularly for mobile systems. They are direct function of the algorithm complexity, and the technology used to implement it. Thus, efficient implementation of modular multipliers is essential for the design of efficient high-speed crypto-processors [53].

In this work, two types of modular multiplication algorithms are evaluated and the corresponding hardware is designed and modeled. The first is a self-timed asynchronous modular multiplier, while the other is a synchronous multiplier that is based on a new fast architecture utilizing a four-to-two compressor. We have used area-delay cost as basis for comparison between the two designs. Furthermore, we have devised a BIST structure for the synchronous design based on a realistic sequential fault model for iterative logic arrays.

The rest of this thesis is organized as follows. In CHAPTER 1, a brief review of essential arithmetic operations is provided, some modular multiplication algorithms are outlined and a brief overview of clocked and event-driven systems is given. CHAPTER 2 describes the design of the proposed asynchronous modulo multiplier. CHAPTER 3 presents a complete design solution for Montgomery modular multiplication. It starts with a review of the algorithm's notation then a hardware implementation of the algorithm is illustrated. In CHAPTER 4, a brief background of hardware testing is given then a Built-In-Self-Test methodology for the compressor modulo multiplier is presented. Finally, the thesis results and conclusion are given in CHAPTER 5.

1.1. BACKGROUND

1.1.1. Cryptographic Systems

1.1.1.1. Symmetric Cryptosystems (Secret Key)

Conventional cryptosystems, also referred to as symmetric or single-key cryptosystems, are based on the use of a common single key and a common algorithm. For the same plaintext message, the algorithm produces different ciphertexts for different keys, see Figure 1.1.

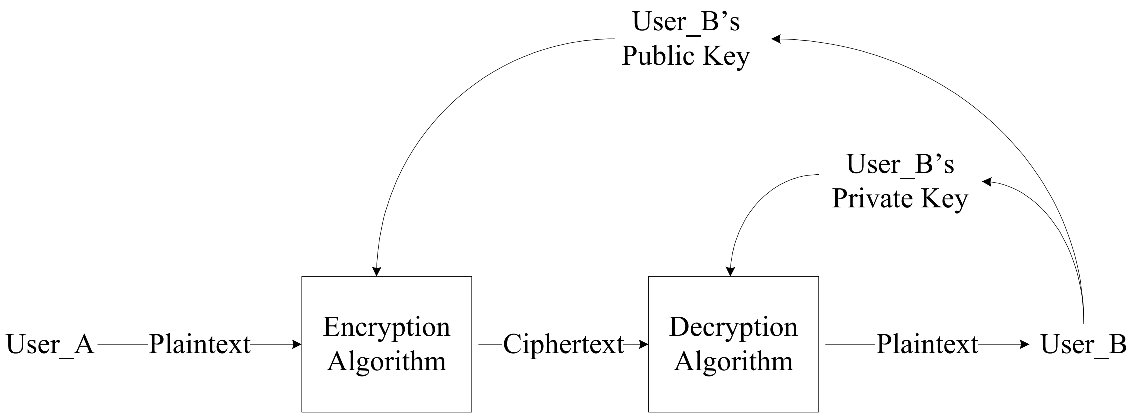

Figure 1.1: Symmetric Cryptosystem.

In conventional cryptosystems, the algorithm should not depend on the input message and should not to be kept secret. Only the key needs to be kept secret and it should be computationally impractical to decrypt the ciphertext knowing only the encryption/decryption algorithm together with samples of plaintext and their corresponding ciphertext [53].

Since, in this method, both the sender and the receiver have the same key, which must be kept secret, There should be a secure key distribution mechanism. Good key distribution mechanisms are not trivial and are not without disadvantages [53].

1.1.1.2. Public-key Cryptosystems

Unlike symmetric cryptosystems, public-key cryptosystems do not use the same key to encrypt and decrypt messages. Instead, each of the two parties has two different but related keys, a public key (KU) and a private key (KR) and they consider a message as consisting of a number of blocks where every message block M has a binary value that is less than some value N (known to both ends). Encryption and decryption algorithms used for public-key cryptosystems are mainly based on modulo operations.

For USER_A to send an encrypted message to USER_B, he must use USER_B's public-key. This message cannot be decrypted without USER_B's private key that is known only to USER_B. Figure 1.2 shows an illustration of the public-key cryptography principle.

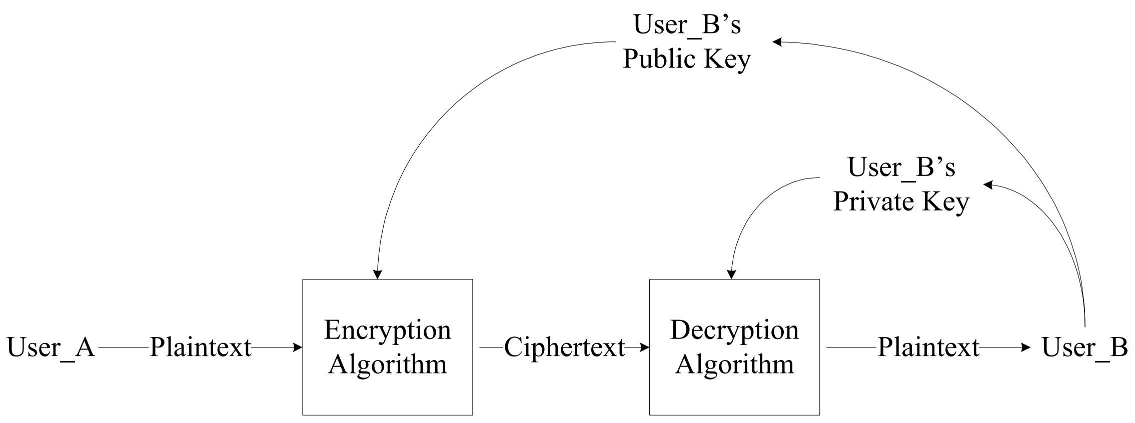

Figure 1.2: Public-Key Encryption.

For any public-key algorithm, the following equations must hold whenever User_A needs to send an encrypted message to User_B:

$$Ciphered = E_{KU\_B}\ (Message)$$

$$Message = D_{KR\_B}\ (Ciphered)$$

$$= D_{KR\_B}\ [E_{KU\_B}\ (Message)]$$

$$= E_{KU\_B}\ [D_{KR\_B}\ (Message)]$$

Where *E* and *D* are the encryption and decryption algorithms respectively.

It should be computationally infeasible to infer the decryption key or the original message given only the algorithm, the encryption key and samples of ciphertexts [29] [53].

Encryption algorithms can be implemented using software or hardware. Whereas software implementations are less expensive, easier to modify, and slow, hardware implementations are more expensive, difficult to modify but are quite faster.

Hardware implementations are evaluated based on their running time (speed), VLSI area, and power dissipation. A practical complexity measure for fast mobile cryptosystems is the area-delay product (AT) [49].

1.1.1.2.1. **RSA Algorithm.** One of the most commonly used public-key cryptosystems is the RSA algorithm. The RSA algorithm was devised by Rivest, Shamir, and Adleman [43]. If M is the message to be encrypted and C is the ciphered message, the RSA algorithm is based on the following three requirements:

- Finding integers *e*, *d*, and *N* such that $M = M^{ed} \bmod N$.
- It should be easy to compute M^e and C^d.
- It should be almost impossible to find *d* knowing only *e* and *N*.

Usually N is a large difficult to factor integer and the message block *M* is such that $0 \leq M < N$. The ciphertext *C* is computed as follows:

$$C = M^e \ mod \ N$$

The plaintext message can be retrieved back using the decryption key *d* as follows:

$$M = C^d \ mod \ N = (M^e)^d \ mod \ N = M^{ed} \ mod \ N$$

Both the sender and the receiver know *N*, and *e*, while only the receiver knows *d* and the keys are represented as:

$$KU=\{e, N\}, KR=\{d, N\}$$

To satisfy the algorithm requirements, the modulus *N* is defined as the product of two prime numbers *p*, *q* (*N=pq*). Therefore $\Phi(pq) = (p-1)(q-1)$ where $\Phi(x)$ is the number of positive integers which are smaller than *x* and are relatively prime to *x*. The decryption key *d* is computed as: [43] [53].

$$gcd(\Phi(N), d)=1 \text{ and } 1<d< \Phi(N), \text{ and } e \equiv d^{-1} \bmod \Phi(N)$$

1.1.1.2.2. **The Elgamal Algorithm.** In this algorithm [48], the system has two public keys; N and g, where N is a large prime and N-1 has at least one large prime factor, and g is a primitive element mod N. Each party has its own private key KR_x (1 < KR_x < N-1) and its public key KU_x that can be computed from the private key as follows:

$$KU\_x = g^{KR\_x} \bmod N$$

For USER_A to send a message *M* ($0 \leq M < N$) to USER_B, he should first choose some random number *U* ($0< U < N$), then a transaction key *K* is computed using USER_B’s Public key (KU_b).

$$K= KU\_b^{U} \bmod N$$

The ciphered message is then computed as a pair $C= (c_1, c_2)$ where

$$c_1 = g^U \mod N \qquad \& \qquad c_2 = KM \mod N$$

Note that the size of the encrypted message is double the size of the original message.

USER_B can decrypt the ciphered message *C* by first retrieving the transaction key *K*. This should be easy for USER_B since

$$K \equiv KU\_b^U \equiv (g^{KR\_b})^U \equiv (g^U)^{KR\_b} \equiv c_1^{KR\_b} \mod N$$

Then the original message *M* will be easily retrieved by dividing c_2 by *K*.

$$M = c_2 /K$$

For increased security the transaction key *K* should not be used for more than one message block. Otherwise, the knowledge of one block allows attackers to know all other blocks [15] [21] [29] [33] [48].

1.1.2. Arithmetic Operations

In this section, after reviewing several multiplication schemes, some division algorithms are also discussed. In addition, the exponentiation operation is described. Finally several modulo multiplication algorithms are outlined.

1.1.2.1. Multiplication Algorithms

Three types of multiplication algorithms are reviewed in this section. First, the sequential multiplication for two's complement singed numbers is described. Then, Booth multiplication algorithm which is based on multiplier recoding is presented. The third algorithm is a high-radix version of Booth multiplication algorithm.

1.1.2.1.1. **Sequential Multiplication.** If X and A are two k-digit numbers, their product P will be $2k$-digits long. Let the multiplier X and the multiplicand A be represented as:

$X= x_{k-1}\, x_{k-2} \ldots x_1\, x_0$

$A= a_{k-1}\, a_{k-2} \ldots a_1\, a_0$

Where x_i and a_i are digits in a number system of radix β.

For unsigned numbers, the product P requires k steps to obtain. In step i, the product Ax_i is shifted and cumulatively added to the partial product P.

$$X = \sum_{i=0}^{k-1} x_i\beta^i.$$

$$P = X\cdot A$$

$$=A * \sum_{i=0}^{k-1} x_i\beta^i.$$

$$= Ax_0\beta^0 + Ax_1\beta^1 + Ax_2\beta^2 + \ldots + Ax_{k-1}\beta^{k-1}$$

This multiplication is illustrated in Figure 1.3 using dot notation for two 4-digit numbers [4].

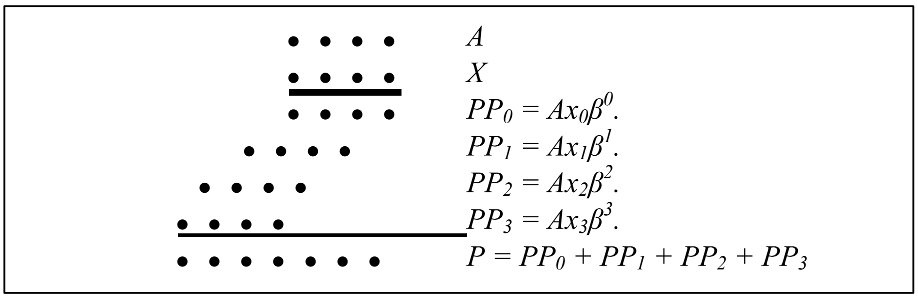

Figure 1.3: Dot Notation for Two 4-word Numbers Multiplication.

In the case of signed numbers, x_{k-1}, a_{k-1} are the sign bits. The product P requires k steps to obtain. In step i, the product Ax_i is shifted and added to the partial product P. The multiplication algorithm can be expressed using the following recursion:

$$X=-x_{k-1}\cdot\beta^{k-1}+\sum_{i=0}^{k-2} x_i\cdot\beta^i.$$

$$P = X\cdot A$$

$$P = -x_{k-1} \cdot \beta^{k-1} \cdot A + \sum_{i=0}^{k-2} x_i \beta^i \cdot A.$$

The following example (see TABLE 1.1) illustrates the multiplication algorithm for binary signed numbers in two's complement representation.

TABLE 1.1: Two's Complement Signed Numbers Multiplication Example.

A		0 0 1 0 1		5
X		1 1 0 1 1		-5
$P^{(0)}=0$		0 0 0 0 0		
$X_0=1$ ➔ add A	+	<u>0 0 1 0 1</u>		
		0 0 1 0 1		
Shift ➔ $P^{(1)}=$		0 0 0 1 0	1	
$X_1=1$ ➔ add A	+	<u>0 0 1 0 1</u>		
		0 0 1 1 1	1	
Shift ➔ $P^{(2)}=$		0 0 0 1 1	1 1	
$X_2=0$ ➔ Shift ➔ $P^{(3)}=$		0 0 0 0 1	1 1 1	
$X_3=1$ ➔ add A	+	<u>0 0 1 0 1</u>		
		0 0 1 1 0	1 1 1	
Shift ➔ $P^{(4)}=$		0 0 0 1 1	0 1 1 1	
$X_{sign}=1$ ➔ add -A	+	<u>1 1 0 1 1</u>		
P=		1 1 1 1 0	0 1 1 1	-25

To speed up the multiplication process we can do one of the following:

- Use faster adders using:
    - Faster Architecture.
    - Faster technology.
- Reduce the number of partial products
    - Using high radix multipliers (scanning more than one multiplier bit at a time)
    - Using Multiplier Recoding techniques.

1.1.2.1.2. **Booth Multiplication.** In Booth multiplication, the number of partial products is reduced using multiplier recoding technique. The multiplication process consists of add and shift operations with addition requiring much more time than the shift operation. The objective is to recode the multiplier bits such that it has less number of ones and more zeros, which reduces the required number of add operations. This can be achieved by skipping chains of zeros and recoding chains of ones [4]. Booth multiplier recoding is illustrated in TABLE 1.2.

Example

Both recoding of (1 1 0 0 1 1 1 0) is (1 0 $\bar{1}$ 0 1 0 0 $\bar{1}$ 0)

TABLE 1.2: Booth Recoding.

Current Bit X_i	Previous Bit X_{i-1}	Recoded Bit Y_i	Note
0	0	0	No string of ones in sight
0	1	1	End of string of ones
1	0	$\overline{1}$	Beginning of string of ones
1	1	0	Middle of string of ones

1.1.2.1.3. **High Radix Booth Multiplication.** Booth algorithm has two disadvantages. The first one is that the number of shifts is not constant. Therefore, the algorithm can not be useful for synchronous systems. The second disadvantage is that the algorithm will give worst results for multipliers that have many isolated ones. For example, if we try to recode 001010101(0), which has four ones (add operations), we will get $1\ \overline{1}\ 1\ \overline{1}\ 1\ \overline{1}\ 1\ \overline{1}$, which has eight ones. To overcome the latter problem, higher radix Booth recoding can be used.

TABLE 1.3 illustrates radix-four Booth recoding in which two bits are recoded at a time.

TABLE 1.3: Radix-4 Booth Recoding Algorithm.

Current Bits	Previous Bit	Recoded Bits	Note
X_{i+1} X_i	X_{i-1}	Y_{i+1} Y_i	
0 0	0	0 0	No string of ones in sight
0 0	1	0 1	End of string of ones
0 1	0	0 1	One between Zeros
0 1	1	1 0	End of string of ones
1 0	0	$\overline{1}$ 0	Beginning of string of ones
1 0	1	0 $\overline{1}$	End of string of ones and starting of another.
1 1	0	0 $\overline{1}$	Beginning of string of ones
1 1	1	0 0	Middle of string of ones

For more on binary numbers recoding see APPENDIX B.

Fast multiplication can also be achieved in a number of other ways, such as tree multiplication and array multiplication [4].

1.1.2.2. Division Algorithms

Division is the most complex of the four basic arithmetic operations. Unlike the other three arithmetic operations, the result of division consists of two components; a quotient Q and a remainder R. Therefore, the result of dividing some dividend X by a divisor D consists of a quotient Q and a remainder R such that $X=Q\cdot D+R$ where $|R|<|D|$.

1.1.2.2.1. **SRT Division.** In non-restoring binary division the divisor D is a normalized fraction. The quotient is computed digit by digit starting with the most significant digit. In is in a form of binary singed digits q_i i.e., $q_i \in \{-1, 0, 1\}$. the remainder is computed using the following recurrence:

$$r_i = 2r_{i-1} - q_iD$$

Where:

$$r_0 = X$$

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq 0 \\ \bar{1} & \text{if } 2r_{i-1} < 0 \end{cases}$$

The selection criteria can be modified to perform the comparison with D as:

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq D \\ 0 & \text{if } -D \leq 2r_{i-1} < D \\ \bar{1} & \text{if } 2r_{i-1} < -D \end{cases}$$

However, this selection criterion requires full precision comparison of r_{i-1} and D. We can overcome this costly comparison by restricting D to be normalized fraction, and the comparison needed will be with ±1/2 instead of ±D. Therefore, all numbers will be presented as fractions and the new quotient selection mechanism is given by

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq 1/2 \\ 0 & \text{if } -1/2 \leq 2r_{i-1} < 1/2 \\ \bar{1} & \text{if } 2r_{i-1} < -1/2 \end{cases}$$

This is known as SRT algorithm after its three authors D. W. **S**weeney, J. E. **R**obertson, and K. D. **T**ocher [37].

Example: The example in TABLE 1.4 illustrates SRT division.

TABLE 1.4: SRT Division Example.

r_0= X $2r_0$ Add -D	 +	0 .0 1 0 1 0 .1 0 1 0 1 .0 1 0 0	>1/2 => q_1=1
r_1 $2r_1$=r_2 $2r_2$= r_3 $2r_3$ Add D	 +	1 .1 1 1 0 1 .1 1 0 0 1 .1 0 0 0 1 .0 0 0 0 0 .1 1 0 0	>-1/2 => q_2=0 >-1/2 => q_3=0 <-1/2 =>q_4=-1
r_4 Add D	 +	1 .1 1 0 0 0 .1 1 0 0	-ve remainder & +ve X correction
r_4		0 .1 0 0 0	Final remainder

1.1.2.2.2. **High Radix SRT Division.** One can reduce the number of shift/subtract operations by using higher radices. Therefore, instead of using the radix-2 SRT, we use higher radix β, where $\beta = 2^m$, the number of shift/subtract steps will reduce from n to $\lceil n/m \rceil$, and m quotient bits are produced per step. The remainder recurrence relation will be:

$$r_i = 2^m r_{i-1} - q_i D$$

Where $q_i \in \{\overline{\beta - 1}, \ldots, \bar{2}, \bar{1}, 0, 1, 2, \ldots, \beta\text{-}1\}$.

This recurrence relation gives the maximum redundancy for radix-*β*, which might be too costly. To find the lower bound, let us assume that $q \in D_a = \{\bar{a}, \ldots, \bar{2}, \bar{1}, 0, 1, 2, \ldots, a\}$ meaning that it can be any of these *2a+1* digits. However, we need at least β digits to represent a number in radix-*β* therefore the following must hold [4]:

$$2a+1 \geq \beta$$

$$\Rightarrow \beta > a \geq (\beta-1)/2$$

1.1.2.2.3. **Vitit's Division Algorithm.** This algorithm [51] is simpler than radix-4 SRT and is based on radix-2 SRT. It has simpler iteration hardware and the number of iterations - at worst - is equal to those of radix-4 SRT [4] and it does not require the use of a lookup table.

Recall that, the SRT algorithm is based on the following recurrence:

$$r_i = 2r_{i-1} - q_iD$$

Initially assume that both $2r_{i-1}$ and D are positive. Therefore, q_i can be selected as follows:

$$q_i = \begin{cases} 0 & \text{if} \quad 2r_{i-1} < 1/2 \\ 1 & \text{if} \quad 2r_{i-1} \geq 1/2 \text{ and } q_{i+1} = \begin{cases} \bar{1} \text{ if } 2r_{i-1} < D \text{ (3-bit comparison)} \\ 0 \text{ if } 2r_{i-1} = D \text{ (3-bit comparison)} \\ 1 \text{ if } 2r_{i-1} > D \text{ (3-bit comparison)} \end{cases} \end{cases}$$

The 3-bit comparison is done on the most significant fractional bits.

The algorithm can be easily modified for signed numbers as follows:

$$q_i = \begin{cases} 0 \quad \text{if } |2r_{i-1}| < 1/2 & \text{and } r_i = 2r_{i-1} \\ \text{SS} \quad \text{if } |2r_{i-1}| \geq 1/2 \quad \text{if this is last q digit else} & \\ \quad q_iq_{i+1} = \begin{cases} 0 \ \text{SS if } |2r_{i-1}| < |D| \text{ (3-bit comparison)} \\ \text{SS } 0 \text{ if } |2r_{i-1}| = |D| \text{ (3-bit comparison)} \\ \text{SS SS if } |2r_{i-1}| > |D| \text{ (3-bit comparison)} \end{cases} & \text{and } r_i = 2r_{i-1} - SSD, \text{ and } r_{i+1} = 2r_i - SSD \end{cases}$$

Where:

SS is Sign(r_{i-1})*Sign(D).

We can use only 2-bit comparison instead of 3-bit comparison if $2r_{i-1}$ and D are positive as follows:

$$q_i = \begin{cases} 0 \quad \text{if} \quad 2\mathrm{r}_{\text{i-1}} < 1/2 & \text{and } \mathrm{r}_{\text{i}} = 2\mathrm{r}_{\text{i-1}} \\ 1 \quad \text{if} \quad 2\mathrm{r}_{\text{i-1}} \geq 1/2 \qquad \text{if this is last q digit else} & \\ \quad q_i q_{i+1} = \begin{cases} 10 \text{ if } 2\mathrm{r}_{\text{i-1}} \leq \mathrm{D} \text{ (2-bit comparison)} & \text{and } \mathrm{r}_{\text{i+1}} = 4\mathrm{r}_{\text{i-1}} - 2D \\ 11 \text{ if } 2\mathrm{r}_{\text{i-1}} > \mathrm{D} \text{ (2-bit comparison)} & \text{and } \mathrm{r}_{\text{i+1}} = 4\mathrm{r}_{\text{i-1}} - 3D \end{cases} & \end{cases}$$

Up to four dividend bits may be retired instead of 3 per iteration as follows:

$$q_i = \begin{cases} 0 \quad \text{if} \quad 2\mathrm{r}_{\text{i-1}} < 1/2 & \text{and } \mathrm{r}_{\text{i}} = 2\mathrm{r}_{\text{i-1}} \\ 1 \quad \text{if} \quad 2\mathrm{r}_{\text{i-1}} \geq 1/2 \qquad \text{if this is last q digit else} & \\ \quad q_i q_{i+1} q_{i+2} = \begin{cases} 011 \text{ if } 2\mathrm{r}_{\text{i-1}} \leq \mathrm{D} \text{ (2-bit comparison)} & \text{and } \mathrm{r}_{\text{i+2}} = 8\mathrm{r}_{\text{i-1}} - 3D \\ 100 \text{ if } 2\mathrm{r}_{\text{i-1}} > \mathrm{D} \text{ (2-bit comparison)} & \text{and } \mathrm{r}_{\text{i+2}} = 8\mathrm{r}_{\text{i-1}} - 4D \end{cases} & \end{cases}$$

It is obvious that we need to compute 3D and -3D in order to run this algorithm improvement[51].

The proof of this algorithm is given in APPENDIX A.

1.1.2.3. Exponentiation

Exponentiation is performed as a number of squaring and multiplication operations depending on the length of the exponent. The algorithm is shown in Algorithm 1.1.

Algorithm 1.1: Exponentiation.

```
Objective:
Compute $X=Y^E$

Algorithm:
$X=1$.
For $i=0$ to $n-1$
    If $e_i = 1$ Then $X= X.Y$
    $Y=Y^2$
End
```

Where:

n: number of bits in the exponent *E*.

$E= e_{n-1}\, e_{n-2} \ldots e_2\, e_1\, e_0$.

e_i: the *i'th* bit of *E*.

The algorithm can be easily modified for modular exponentiation by replacing the multiplication step in Algorithm 1.2 with a modular multiplication [16].

Algorithm 1.2: Modular Exponentiation.

```
Objective:
Compute $X=Y^E$ Mod N

Algorithm:
$X=1$.
For $i=0$ to $n-1$
    If $e_i = 1$ Then $X= (X.Y)$ Mod N
    $Y=(Y.Y)$ Mod N
End
```

1.1.2.4. Modular Multiplication

1.1.2.4.1. **Triangle Addition Algorithm.** Modular multiplication algorithms -for n-word numbers- are mostly classified into the following two categories:

1. Division-after-multiplication: Here, an n-word by n-word multiplication is performed first, then a 2n-word by n-word division is carried out. This method requires 2n-word memory space to store intermediate results, but it does not need many subtraction steps.
2. Division-during-multiplication: In this category, division residue subtraction steps are interleaved with the multiplication addition steps and only (n+1)-word memory space is needed. On the other hand, it requires n-word subtractions and (n+1)-word by n-word division per residue calculation step.

The modular multiplication with triangle addition algorithm is a new algorithm that does not belong to any of the above categories [34]. It combines the advantages of the other two approaches by having the same memory space requirement as division-during-multiplication, and the same number of steps needed by the division-after-multiplication category. It is a completely new algorithm in which the upper half triangle of the all partial products is added up and its residue is calculated. Then the sum of the lower half triangle of all partial products is added to the pre-calculated residue. Finally, the final residue of the total result is calculated.

Assumptions and Notations

For (A × B mod N)

- A, B and N are n-word numbers satisfying $0 \le A, B < N$.
- $n \le \beta$. Where β is the radix.
- $\delta = \sum_{i=0}^{n-1} \delta_i . \beta^i$ Where δ can be A, B or N.

The Algorithm

The algorithm (shown in Algorithm 1.3) is based on the following formula [34]:

$$\begin{aligned} A \times B \bmod N &= \sum_{i,j} A_i . B_j . \beta^{i+j} \bmod N \\ &= \left(\left(\sum_{i+j \ge n-1} A_i . B_j . \beta^{i+j} \right) \bmod N + \sum_{i+j < n-1} A_i . B_j . \beta^{i+j} \right) \bmod N \end{aligned}$$

Algorithm 1.3: Triangle Addition Modular Multiplication.

```
$P \Leftarrow \sum_{i+j \ge n-1} A_i . B_j . \beta^{i+j-n+1}$
$P \Leftarrow \left( P . \beta^{n-1} \right) \bmod N$
$P \Leftarrow P + \sum_{i+j < n-1} A_i . B_j . \beta^{i+j}$
$P \Leftarrow P \bmod N$
```

1.1.2.4.2. **Holger and Peter's Interleaving Algorithm.** In [16], modular multiplication is done by interleaving the multiplication with the division. In Algorithm 1.4, P plays the role of partial remainder in SRT division and partial product in multiplication where the algorithm adds partial product, subtract divisor multiple, and shift left the result P.

Algorithm 1.4: Interleaving Modular Multiplication.

```
Objective:
Compute P=(A.B)mod N

Algorithm:
P=0
For i=n-1 downto 0
    q = estimate(P/N)
    $P=2^k P + a_i B - 2^k qN$
End
P correction

Where:
    n: number of radix $2^k$ words.
    $a_i$: the i'th digit of the multiplier
```

The P correction step is required because an estimated value of q is used. The estimate of q should be good enough to keep P from diverging during the calculations.

In order to save time, carry save redundant representation of P is used – in which the value of P is represented in two registers usually called SUM and CARRY.

To estimate q, a *parallel exhaustive search* is used, in which the following expression is performed in parallel for all values of q:

$$P -q\, 2^k N$$

Only few bits of P and $-q\, 2^k N$ are used in this estimation, twelve bits for radix 32 as an example. According to [16], this method generally is very expensive in terms of hardware, but may be acceptable for operands longer than 500 bits.

1.1.2.4.3. **Takagi's Radix-4 Algorithm.** Takagi [35] has presented a fast Radix-4, division-during-multiplication, modular multiplication algorithm. In this algorithm, operands and partial products are represented in redundant formats and the intermediate results are stored in more redundant format to reduce the number of additions/subtractions required. There is only one time-consuming, carry propagation step at the end of the algorithm. The algorithm calculates P=A.B (mod N), and uses the following recurrence:

$$P_j := 4.P_{j+1} + \widehat{b}_j.A - 4c_j.N$$

Where:

- N: n-bit binary number, $2^{n-1} \leq N < 2^n$.
- A: (n+1)-digit redundant binary number, $-N < A < N$.
- B: (n+1)- digit redundant binary number, $-N < B < N$.
- P: (n+1)- digit redundant binary number, $-N < P < N$.
- $\hat{b}_j$: The *i'th* digit of the recoded B (multiplier). It depends only on five digits of B (b_{2j+1} down to b_{2j-3}).
- c_j : used for residual calculation, $c_j \in \{\overline{2}, \overline{1}, 0, 1, 2\}$.

The recoded multiplier $\widehat{B}$ is a $(\left\lfloor \frac{n}{2} \right\rfloor + 1)$ – digit radix-4 signed digit number that can be obtained using TABLE 1.5 where u_j, t_j are intermediate temporary values. A recoding example is shown in Figure 1.4.

TABLE 1.5: Multiplier Recoding Rule.

u_j, t_j

b_{2j+1} \ b_{2j}	$\bar{1}$	0	1
$\bar{1}$	$\bar{1}$,1	[1]0, $\bar{2}$ / $\bar{1}$,2	0, $\bar{1}$
0	0, $\bar{1}$	0,0	0,1
1	0,1	[1]1, $\bar{2}$ /0,2	1, $\bar{1}$

Stage 1

$\widehat{b}_j$

t_j \ u_{j-1}	$\bar{1}$	0	1
$\bar{2}$	×	$\bar{2}$	$\bar{1}$
$\bar{1}$	$\bar{2}$	$\bar{1}$	0
0	$\bar{1}$	0	1
1	0	1	2
2	1	2	×

Stage 2

$$\begin{array}{cccccccccccc}
0 & \bar{1} & 0 & 1 & 1 & 0 & B & 0 & 0 & 1 & & B \\
\multicolumn{2}{c}{0} & \multicolumn{2}{c}{1} & \multicolumn{2}{c}{0} & \multicolumn{2}{c}{0} & \multicolumn{2}{c}{0} & (0) & u \\
\multicolumn{2}{c}{\bar{1}} & \multicolumn{2}{c}{1} & \multicolumn{2}{c}{\bar{2}} & \multicolumn{2}{c}{0} & \multicolumn{2}{c}{1} & & t \\
\hline
\multicolumn{2}{c}{\bar{1}} & \multicolumn{2}{c}{2} & \multicolumn{2}{c}{\bar{2}} & \multicolumn{2}{c}{0} & \multicolumn{2}{c}{1} & & \widehat{B}
\end{array}$$

Figure 1.4: Multiplier recoding example.

[1] b2j-1 is nonnegative/otherwise

Choice of c_j based on the following:

$$c_j := \begin{cases} \overline{2} & \text{if} \quad \text{top}(R_j) < -\text{top}(6.N) \\ \overline{1} & \text{if} \quad -\text{top}(6.N) \le \text{top}(R_j) < -\text{top}(2.N) \\ 0 & \text{if} \quad -\text{top}(2.N) \le \text{top}(R_j) < \text{top}(2.N) \\ 1 & \text{if} \quad \text{top}(2.N) \le \text{top}(R_j) < \text{top}(6.N) \\ 2 & \text{if} \quad \text{top}(R_j) \ge \text{top}(6.N) \end{cases}$$

Top means the most significant 4 digits of N, top 5 digits of (2.N), the top 6 digits of (6.N), or 8 digits of R [32] [35] [36]. In other words, left pad N, 2N, and 6N with zeros to make them (n+4)-digits numbers. Then the comparison is carried out on the most significant eight bits of them as illustrated in Figure 1.5. The algorithm steps are shown in Algorithm 1.5.

n+4	n+3	n+2	n+1	N	n-1	n-2	n-3	… 2	1	
				☒	☒	☒	☒	☒	☒	N
			☒	☒	☒	☒	☒	☒	☒	2N
		☒	☒	☒	☒	☒	☒	☒	☒	6N
☒	☒	☒	☒	☒	☒	☒	☒	☒	☒	R_j

Figure 1.5: Illustration of the compared bits.

Algorithm 1.5: Takagi's Radix-4Modular Multiplication.

```
Step 1: $P_{\lfloor n/2 \rfloor+1} := 0$
Step 2: for $j=\lfloor n/2 \rfloor$ down to -1
    $R_j = 4.P_{j+1} + \hat{b}_j.A$
    $P_j = R_j - 4.c_j.N$
Step 3: $P=P_{-1}/4$
```

Where:

R_j is (n+4)-digit RBN (Residue Binary Number) and $R_j := 4.P_{j+1} + \hat{b}_j.A$

1.1.2.4.4. **Montgomery's Algorithm.** Montgomery [39] came up with an elegant way to calculate the modular multiplication. The idea is to transfer the problem to another domain which will be referred to as the Montgomery domain. The modulo multiplication in the Montgomery domain is made easier and faster.

It is required to compute $A{\cdot}B \bmod N$, where A, B, and N are n-bit numbers with $0< A,\ B < N,$ and N being an odd number. First the operands A and B are mapped into the Montgomery domain where A is mapped into $\overline{A} = AR \bmod N$ and B is mapped into $\overline{B} = BR \bmod N$ where $R=2^n$. The two mapped numbers $\overline{A}$ and $\overline{B}$ are presented to the Montgomery product procedure $Mon\_\Pr o(\overline{A}, \overline{B})$. The algorithm requires the calculation of R^{-1} and $\acute{N}$ where $R{\cdot}R^{-1} \bmod N = 1$ and $R{\cdot}R^{-1}\text{-}N{\cdot}\acute{N} = 1$. The calculation of $\acute{N}$ and the transformation to and form the Montgomery domain are time consuming steps. However, this cost is tolerable for modulo exponentiation ($X^E \bmod N$) where modulo multiplication is performed repeatedly. Hence, transformation to Montgomery domain is performed once at the beginning, and then the result of the modulo exponentiation operation is transformed back form the Montgomery domain at the end [6].

Modifications to Montgomery algorithm were made by V.Bunimov et al [49], where the original operands are fed directly to the algorithm as $Mon\_\Pr o(A, B) = \overline{P}$ while the second pass will be for $Mon\_Pro(\overline{P},\Re) = P$ where $\Re = R^2 \bmod N$. The steps are shown in Algorithm 1.6.

Algorithm 1.6: Montgomery Modular Multiplication Algorithm.

```
Objective:
Compute Mon_Pro(A,B)

Algorithm:
T=0
For i=0 to n-1
    $T = T + ai \times B + t0 \times N$
    $T = T / 2$
If $T \geq N$ then $T = T - N$

Where:
    $a_i$ is the i'th bit of A and $t_0$ is the LSB of T.
```

Keeping in mind that adding multiple of the modulus (N) does not affect the final result and since N is an odd number, the result of the first line of the ***for*** loop is always even. Therefore, the division in the second line of the ***for*** loop will have no remainder. By going through the first pass, the algorithm would have performed division by $R = 2^n$, i.e., $T= (A \times B/R) \bmod N$. This is why we need to run the algorithm for a second time with T and R^2 as operands. The output of the second pass will be

$$P= (R^2 \times T/R) \bmod N = [R\,(A \times B/R)] \bmod N = (A \times B) \bmod N.$$

The only time consuming step in this version is the computation of $\Re = R^2 \bmod N$ [6] [39] [49]. A more detailed description on versions of Montgomery algorithm is given in page 57.

1.1.2.5. Synchronous and Asynchronous Circuits

1.1.2.5.1. **Clocked synchronous circuits.** Synchronous circuits generally use a common global clock. It is quite simple to design circuits using synchronous logic, because it is commonly understood and used. Moreover, clocked-logic parts are widely available in the market and there is no timing hazards associated with it.

Synchronous systems performance follows the worst-case behavior and suffers from clock skew problems that limit the clock speed. Replacing any system module will require complex and costly timing analysis. While asynchronous circuit module are activated and consume power only upon request, synchronous circuit module, however, dissipate power even if not active since they are regularly clocked (charged and discharged) [8] [23] [48].

1.1.2.5.2. **Event-driven asynchronous circuits.** These circuits use a request-Acknowledge handshaking protocol rather than a global clock signal. For such systems, various modulus act in an independent manner based on local events and the system overall speed performance follows the average-case behavior. Overall system speed is improved by replacing any module with a faster one without any need for timing analysis since they do not use a global clock. Using CMOS technology for implementation, power dissipation of asynchronous systems is less since only active modules will consume power.

On the other hand, it is more difficult to design asynchronous modules since they are subject to timing hazards and signal races. Asynchronous circuit modules generally require more silicon area than their synchronous components [8] [23] [48].

For controllers, event-driven transition signaling is based on signal transitions (events). All signal transitions have the same meaning and there is no distinguishing between rising or falling transactions, which might double the speed over clocked logic. The following components are typically employed in event-driven based controllers:

1. **C-Element**: C-Element performs ANDING of events where an event is generated at the output only if events are detected on all of the input ports. It is assumed that no simultaneous events may occur at the inputs of a C-Element [29] [30].
2. **Merge Element**: Merge element performs ORING of events in which an event on the output is generated if an event occurs on any of its two input ports. It is assumed that no simultaneous events may occur at the inputs of a Merge-Element [29] [30].

□

CHAPTER 2

DESIGN AND MODELING OF ASYNCHRONOUS MODULO MULTIPLIER

This work investigates the use of asynchronous techniques for the design of efficient modulo multipliers. With the large size operands commonly used in cryptosystems, using array or parallel multipliers would require prohibitively large areas. Instead, sequential multipliers are employed in this work. Since sequential multipliers use repeated add and shift operations, an asynchronous implementation can significantly improve the speed at a modest increase in area. For k-bit adders, the speed of an asynchronous adder is O (Log k) on the average [14] compared to the O (k) speed of carry-propagate adders. Asynchronous event logic based on transition signaling is used, where signal transitions are used as control events [21].

The multiplication process consists of a number of add and shift operations with addition requiring much more time than the shift operation. In addition to using an asynchronous adder with O (Log k) average speed [1], a number of other measures were adopted to further improve the overall speed of the system. For one, the developed algorithm uses radix-4 system, which retires two bits per iteration instead of one. For another, multiplier recoding as a signed-digit number [4] is used to allow skipping over chains of zeros as well as chains of ones which results in a considerable reduction in the number of required add operations, and hence a significant speed improvement.

2.1. DESCRIPTION OF THE ASYNCHRONOUS ALGORITHM

It is required to compute $P = X \times Y \ mod\ N$, where the modulus N, the multiplicand X and the multiplier Y are k-bit numbers. Typically, N is a very large odd number, i.e., generally $N_{k-1} = N_0 = 1$. The developed algorithm uses radix-4 system, but may be extended to higher radixes as well. In addition, a Booth-like recoding of the multiplier (Y) into an equivalent signed digit representation is used. Such recoding increases the number of 0s and reduces the number of *1s* and *-1s*. This reduces the number of required add/subtract operations thus improving the overall speed. The overall procedure is given in Algorithm 2.1 and consists of four major steps:

a. Initialization
b. Recoding and adding
c. Scaling
d. Correction

Algorithm 2.1: Asynchronous Modulo Multiplication.

```
a.  Initialization:
    P(k+4 bits) ← 0 where k is the number of bits in N.
    Left pad Y by two bits, i.e., $Y_{k+1}= Y_k=0$.
    Compute (N-X), 3N and 5N. i=k+1

b.  Recoding and Adding:
    WHILE i >0
    {   P ←4P;
        CASE $P_{k+2}\ Y_i\ Y_{i-1}\ Y_{i-2}$ IS
        {   X000, X111      : skip
            0001, 0010      : P ← P-(N-X)
            0011            : P ← P-2(N-X)
            0100            : P ← P-2X
            0110, 0101      : P ← P-X

            1110, 1101      : P ← P+(N-X)
            1100            : P ← P+2(N-X)
            1011            : P ← P+2X
            1001, 1010      : P ← P+X
        }

c.  Scaling:
        CASE $P_{k+2}\ P_{k+1}\ P_k\ P_{k-1}\ N_{k-2}$ IS
        {   000XX, 111XX    : skip
            001XX           : P ← P-2N
            010X1           : P ← P-3N
            010X0, 011X1    : P ← P-4N
            01100           : P ← P-5N
            01110           : P ← P-6N

            110XX           : P ← P+2N
            101X1           : P ← P+3N
            101X0, 100X1    : P ← P+4N
            10010           : P ← P+5N
            10000           : P ← P+6N
        }
        i=i-2
    }

d.  Correction:
    WHILE P>0
    {   P ← P-N
    }
    P ← P+N
```

2.1.1. Initialization Step

In this step, the partial product register is cleared and the values required throughout the algorithm are computed. These values are (N-X) which is used in the recoding step, 3N and 5N that are used in the scaling step. Then the partial product accumulator -register P- is cleared again.

2.1.2. The Recoding And Adding Step

The algorithm scans one multiplier digit (2bits) plus one look-ahead bit from left-to-right- every iteration. Digit recoding is based on TABLE 2.1.

To compute $P = X \times Y \bmod N$, the product register P is initially left padded with a total of four bits. Three bits to accommodate the sign and the left shift operation by one digit (2bits) and the fourth is needed because the multiple $6N$ is needed in the scaling step. For proper recoding, the multiplier is also left padded with two Zeros. After the initialization step, the proper multiple of X is added or subtracted from P based on the value of the recoded multiplier digit and the sign of P, i.e., P_{k+4}. It should be noted that instead of subtracting/adding X, (N-X) may be equivalently added/subtracted. To reduce the chance of overflow, the performed operation, i.e., adding X or equivalently subtracting (N-X), is chosen to oppose the current sign of P. For example, according to TABLE 2.1, if $y_i\ y_{i-1}\ y_{i-2}$=001 then X should be added to P. In this case, if P is negative, we add X to P, but if P is positive we subtract (N-X) from P. This requires pre-computation and storage of (N-X).

TABLE 2.1: Left-to-Right Recoding.

Scanned Multiplier Digit $y_i\ y_{i-1}$	Look Ahead Bit y_{i-2}	Action
00	0	Shift 2-bits
00	1	+1 X; Shift 2-bits
01	0	+1 X; Shift 2-bits
01	1	+2 X; Shift 2-bits
10	0	-2 X; Shift 2-bits
10	1	-1 X; Shift 2-bits
11	0	-1 X; Shift 2-bits
11	1	Shift 2-bits

2.1.3. <u>Scaling Step</u>

The scaling step subtracts/adds a proper multiple of N such that the three most significant bits of P are guaranteed to have the same value either all 0s for positive values or all 1s for negative ones. This step is necessary to prevent overflow after the left shift operation (i.e., $P = 4P$). The selected multiple (jN) should satisfy the following inequality for all possible values of P and N.

$$-2^k \leq P-jN < 2^k$$

Extreme values of j are determined by the two corner points:

1. *Maximum (P)-j Minimum (N).*
2. *Minimum (P)-j Maximum (N).*

To determine the proper multiple (j) of N that should be subtracted/added from/to P, the four most significant bits of P ($P_{k+2}\, P_{k+1}\, P_k\, P_{k-1}$) and the two most significant bits of N ($N_{k-1}\, N_{k-2}$) are considered. Since N_{k-1} is assumed to be always 1, only N_{k-2} need to be considered. Based on the values of these bits, the algorithm determines the proper N multiple to be subtracted/added from/to P such that P can be expressed in only k bits.

TABLE 2.2 shows the detailed analysis of this problem for positive values of P. For example, in the fifth row where $P_{k+2}\, P_{k+1}\, P_k\, P_{k-1}$ = 1100 and N_{k-2} =0, five is the only multiple of N that keeps P in k bits after performing $P\text{-}5N$ for all possible values of P and N. Similar analysis is applied for negative values of P.

TABLE 2.2: Extreme Cases with the Proper Multiple on N.

P_{k+2}	P_{k+1}	P_k	P_{k-1}	N_{k-2}	Maximum P-j Minimum N	Minimum P-j Maximum N	j
0	0	1	×	×	$2^{k+1}-1-2\times(2^{k-1}+1)$ $=2^k-3$	$2^k-2\times(2^k-1)$ $=-2^k+2$	2
0	1	0	×	0	$2^{k+2}-2^{k-1}-4\times(2^{k-1}+1)$ $=2^k-5$	$2^{k+1}-4\times(2^k-2^{k-2}-1)$ $=-2^k+4$	4
0	1	0	×	1	$2^{k+2}-2^{k-1}-3\times(2^{k-1}+2^{k-2}+1)$ $=3\times2^{k-2}-4$	$2^{k+1}-3\times(2^k-1)$ $=-2^k+3$	3
0	1	1	×	1	$2^{k+2}-1-4\times(2^{k-1}+2^{k-2}+1)$ $=2^k-5$	$2^{k+1}2^k-4\times(2^k-1)$ $=-2^k+4$	4
0	1	1	0	0	$2^{k+2}-2^{k-1}-1-5\times(2^{k-1}+1)$ $=2^k-6$	$2^{k+1}2^k-5\times(2^k-2^{k-2}-1)$ $=-3\times2^{k-2}+5$	5
0	1	1	1	0	$2^{k+2}-1-6\times(2^{k-1}+1)$ $=2^k-7$	$2^{k+1}+2^k+2^{k-1}-6\times(2^k-2^{k-2}-1)$ $=-2^k+6$	6

2.1.4. Correction Step

After all k/2 iterations are performed, the resulting value of P may need correction. The correction step is guaranteed to require no more than one addition or two subtractions, since the final P value coming out of the scaling step is a k-bit number and N is a k-bit number. The general data flow of the algorithm is as illustrated in Figure 2.1.

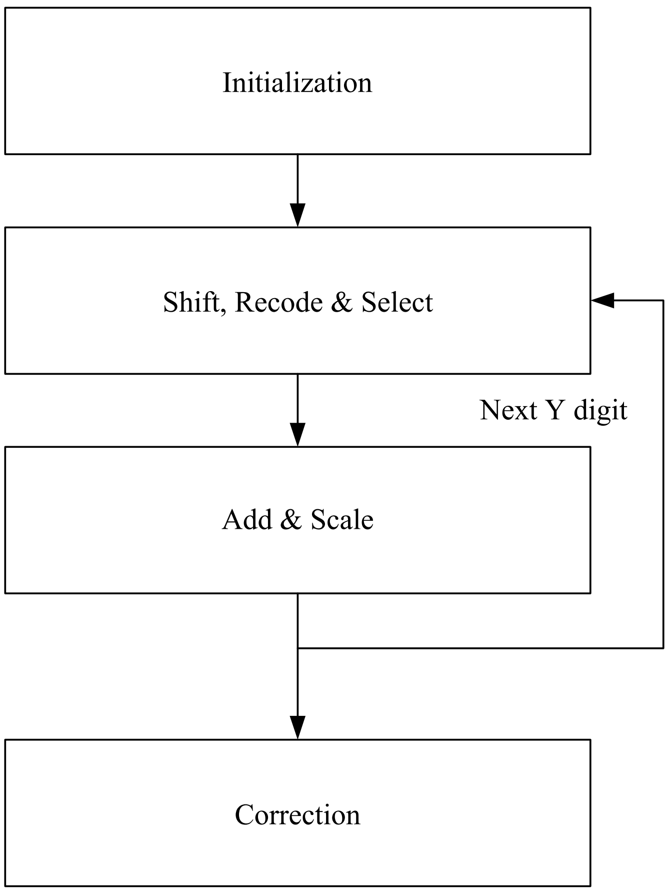

Figure 2.1: Modular Multiplication Data Flowchart.

2.1.5. <u>Illustrative Example</u>

Compute *(9×11) Mod 13*.

initialization	X=1001, Y=001011, N=1101, i=5,(N-X)=0100,
	3N=100111, 5N=10000001, and P= 0000 0000.
Recoding	P=P-(N-X)
	P=1111 1100
Scaling	Skip
	i=3
Shifting	P=1111 0000
Recoding	P=P+(N-X)
	P=1111 0100
Scaling	Skip
	i=1
Shifting	P=1101 0000
Recoding	P=P+(N-X)
	P=1101 0100
Scaling	P=P+3N
	P=1111 1011
	False
Correct	P=P+N
	P=0000 1000 =8

2.2. DESIGN ISSUES

The adopted asynchronous system implementation of the above algorithm is based on event control logic [19]. This implementation was modeled using VHDL where the select module was used to implement decisions (IF statements) and Loops were implemented using a merge element with the loop condition checked through a select module [19] as shown in Figure 2.2.

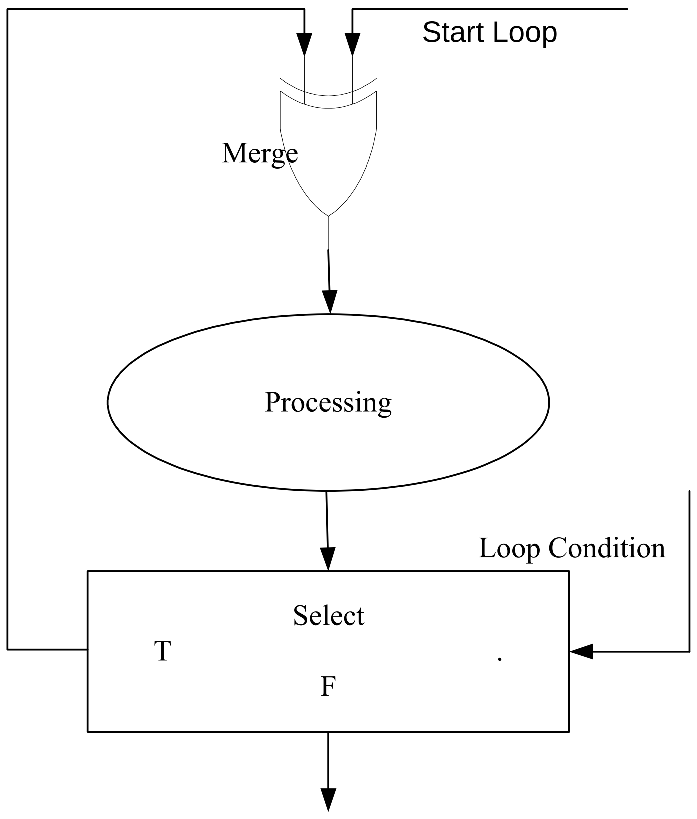

Figure 2.2: Loop Implementation.

For area efficiency, counters and registers were implemented as clocked synchronous elements, i.e., it is a globally asynchronous locally synchronous design. The local clock input of a register (counter) receives a single clock pulse whenever a signal event is received at the input request line. This is achieved using an edge detector and a one shot circuitry as shown in Figure 2.3.

Although all intermediate results are (*k*+3) bits or less including the sign, *6N* and *5N* might need (*k*+4) bits. Therefore, the width of registers and adder is chosen to be (k+4) bits, but input interfaces need only to be *k*-bit wide.

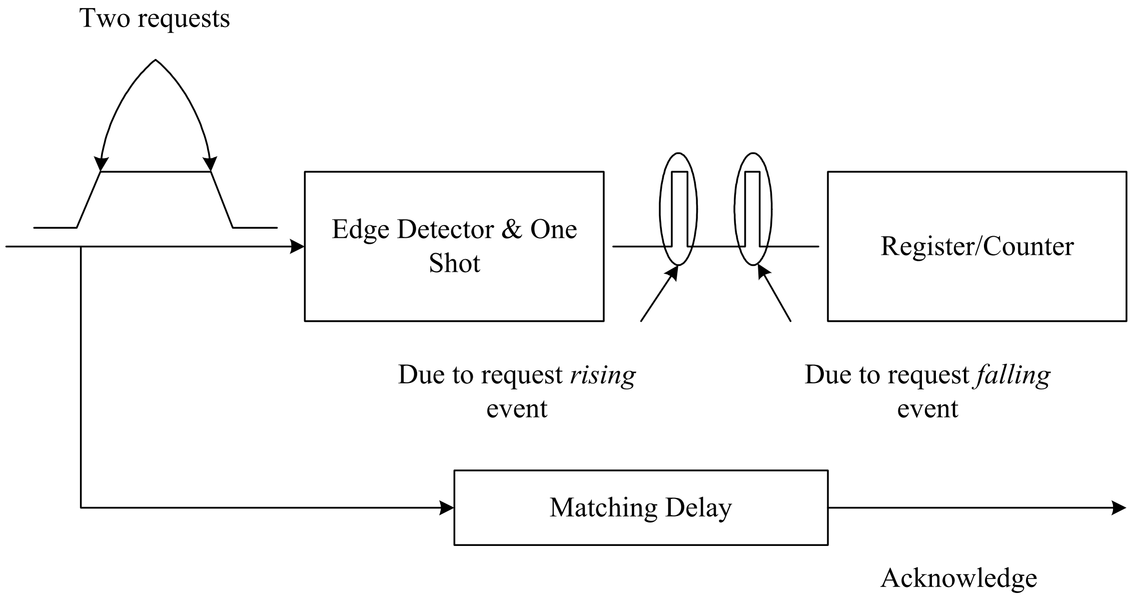

Figure 2.3: Clock Pulse Generation for Registers/Counter.

The design assumes that the most significant (k^{th}) bit of N is one. Therefore, in case of smaller values of N, N should be shifted left to be a k-bit number and the multiplicand X should also be shifted left by the same number of bits. In this case, the final result should to be corrected by shifting it right by the same number of bits. This can be seen from the following equation:

$$P = (X\times Y) \bmod N \Leftrightarrow P\times 2^m = (X\times Y\times 2^m) \bmod (N\times 2^m)$$

For example, $(3\times5) \bmod 7 = 1$ implies that $(3\times5\times8) \bmod (7\times8) = 8$ and visa versa.

2.2.1. Design Data Flow

The data path in Figure 2.4 consists of seven (k+4)-bit registers, one (*Log k*)-bit counter and one ($k+4$)-bit adder. The adder is capable of performing ($A+B$), ($B-A$) and ($A-B$) operations. The operations to be performed are $P\pm X$, $P\pm 2X$, $P\pm(N-X)$, $P\pm 2(N-X)$, $P+N$, $P\pm 2N$, $P\pm 3N$, $P\pm 4N$, $P\pm 5N$, $P\pm 6N$ and $N+2N$. With these, we need multiplexers at both inputs of the adder where one of the multiplexers needs only to select between P and N. For symmetry and delay reasons, the values are distributed between the two multiplexers.

The registers *3N*, *5N* and *N-X* (later designated as *NmX*) are loaded only once during the initialization phase, therefore they are not connected directly to the adder's output. This is to speed up loading register *P*, which occurs in all states of the design.

Since we are using synchronous modules, all control signals (Load/shift, Clear, Add/Sub) are level, not event, signals.

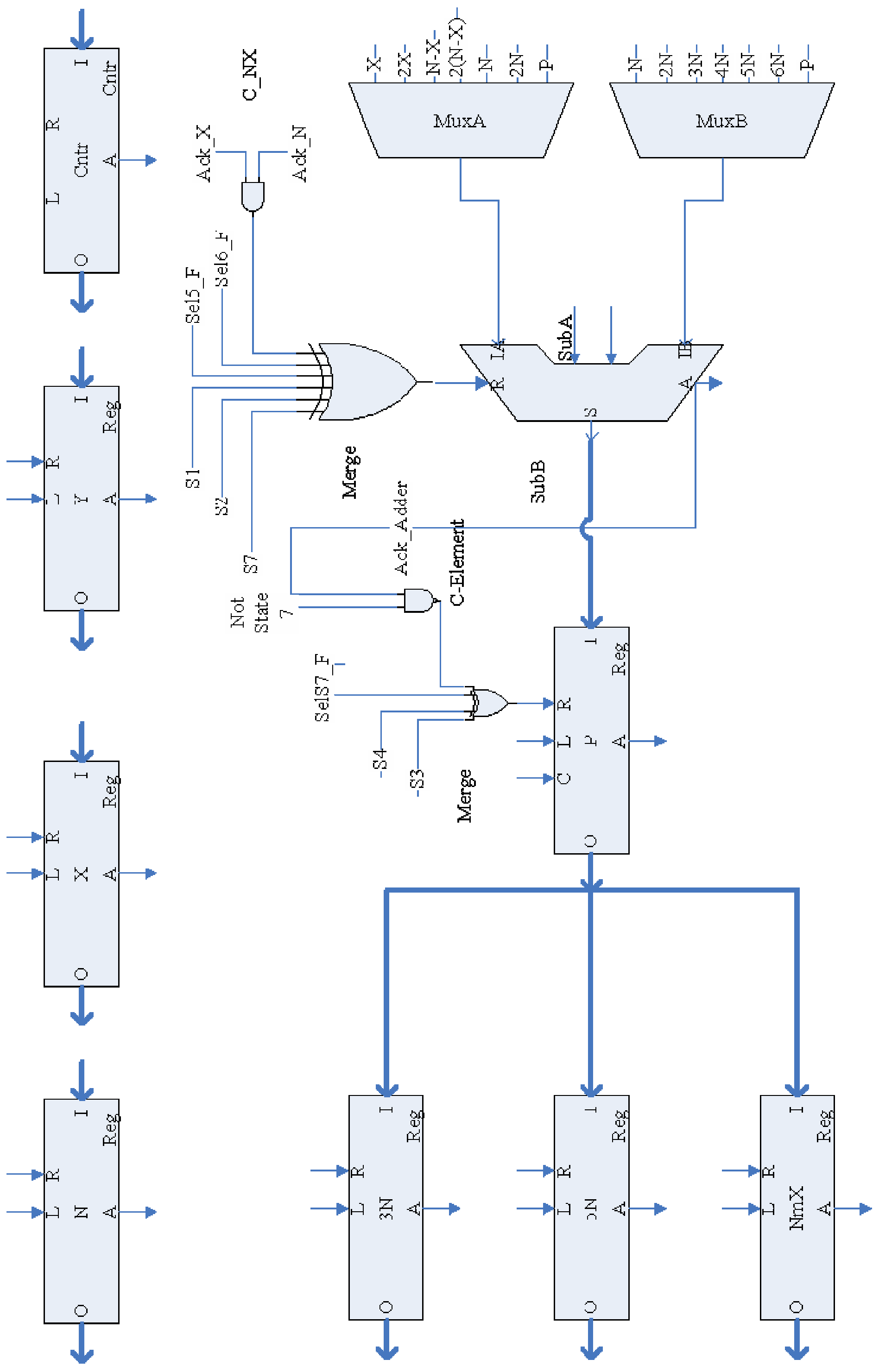

Figure 2.4: Asynchronous Modulo Multiplier Hardware Data Flow.

2.2.2. Controller Design

The design contains eight states, the first four states are initialization states, then the algorithm loops through three states; namely the shift, add and scale. The eighth state is for correction. Figure 2.5 shows the state diagram of the design.

The controller states are implemented using three synchronous D-Flip-Flops with local clock generation. These three Flip-Flops will be referred to as the “state register”. The start signal causes state register to be cleared while the signals S1, S2, S3, S4, S5, S6, S7 and Loop activate state transitions between various states. Data stored in the state register represents the state variable designating the current state.

Load/shift control inputs of the registers and load/decrement input of the counter are controlled based on the value of current state variables. Request events, which initiate a particular operation, are generated when certain conditions or events are satisfied in a given state. A description of what is performed in each state is detailed below.

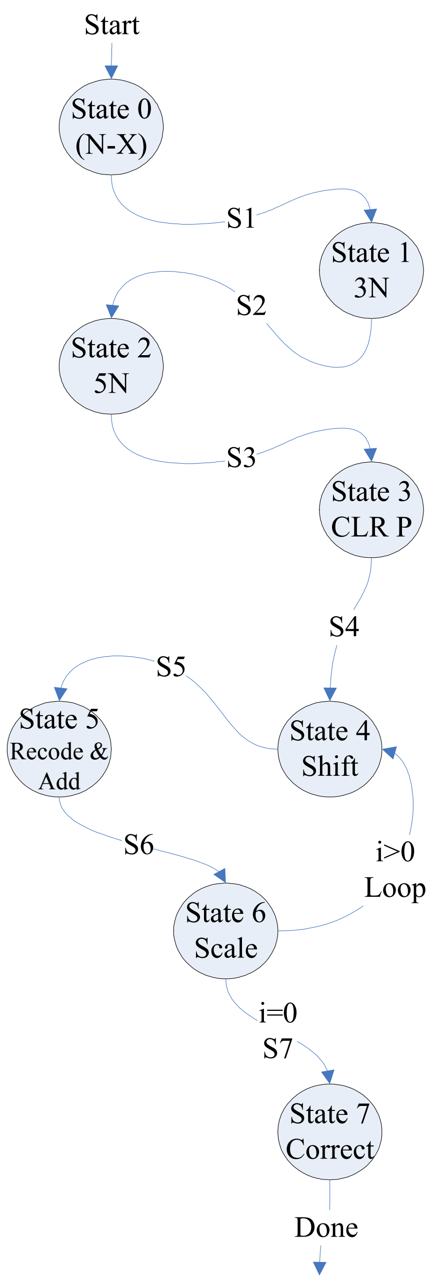

Figure 2.5: Asynchronous Modulo Multiplier State Diagram.

2.2.2.1. Computing N-X (State 0)

A start signal clears the state register to 0 which generates requests to load initial values in registers *X*, *Y*, *N* and the counter *i* (referred to later as *Cntr*). Then;

- Acknowledgments from registers *X* and *N* generate a request for the adder to compute (*N-X*)
- The adder's acknowledgment signal is used as a request for *P* to load the computed result (*N-X*).
- Acknowledgment from register *P* requests the register *NmX* to load the value of register *P*.
- The acknowledgments from registers *Y*, *NmX* and *Cntr* cause the state register value to change to 1.

Figure 2.6 shows the data flow in State 0.

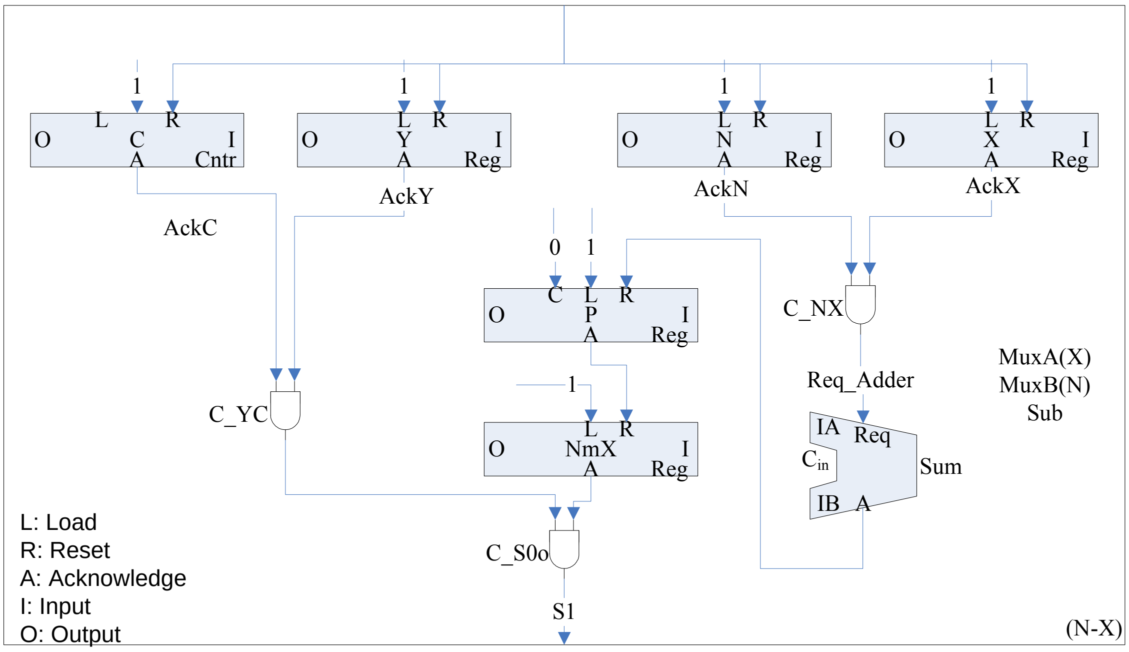

Figure 2.6: State 0, Computing N-X.

2.2.2.2. Computing 3N (State 1)

Figure 2.7 shows the date flow of State 1 as follows:

- If the present State is 1, a request is generated for the adder to compute *N+2N*.
- Adder's acknowledgment requests register P to load the result (*3N*).
- Acknowledgment from register *P* requests register *3N* to copy register *P*'s value.
- Acknowledgment from register *3N* causes a transaction to State 2.

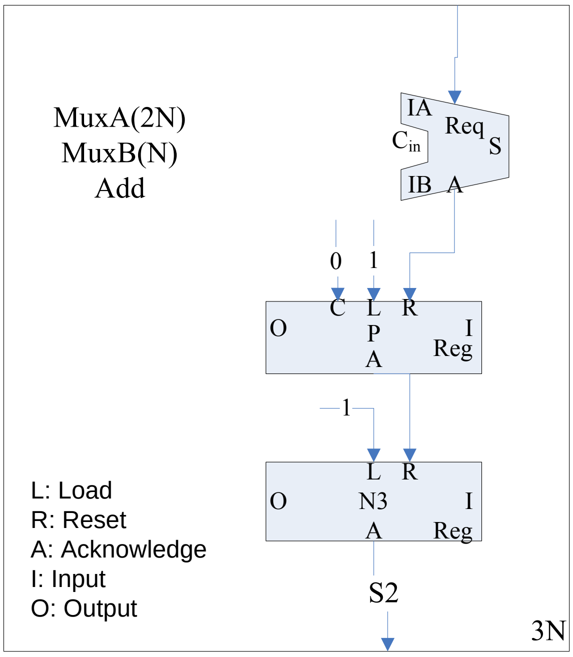

Figure 2.7: State 1, Computing 3N.

2.2.2.3. Computing 5N (State 2)

The data flow of this state is shown in Figure 2.8 and can be described as follows:

- Once the state register value changes to two, the adder is requested to compute *P+2N* –where register *P* contains the previous value, which is *3N*.
- Acknowledgment from the adder requests the register P to load the result (5N).
- Acknowledgment from register P's requests register 5N to copy register P's value.
- Acknowledgment from register 5N changes the state register value to three.

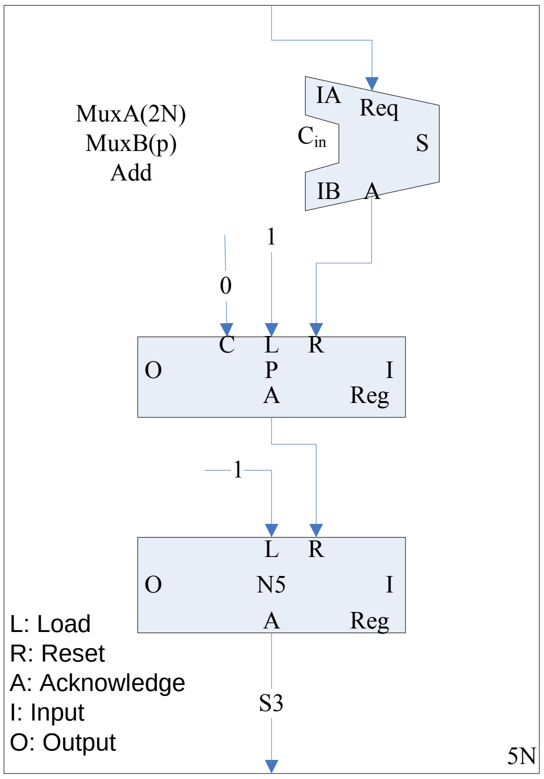

Figure 2.8: State2, Computing 5N.

2.2.2.4. Clear Register P (State 3)

This is the last state in the initialization phase. As Figure 2.9 shows, In State 3, register P is cleared. The acknowledgment from register P changes the state register value to four.

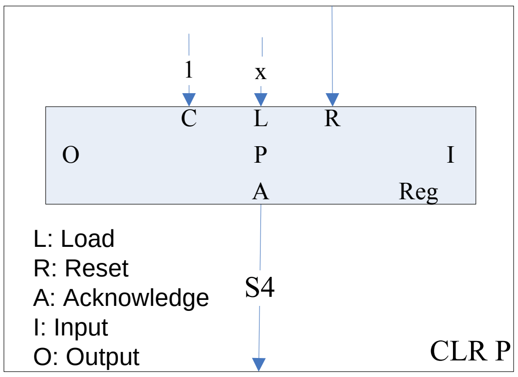

Figure 2.9: State3, Clearing P.

2.2.2.5. Shift Left P and Y (State 4)

The loop starts with state four. A change on state register value to four, requests the registers P and Y to be shifted left by two bits. Acknowledgments from register P and Y changes the state register value to five. Figure 2.10 shows the data flow digram of State 4.

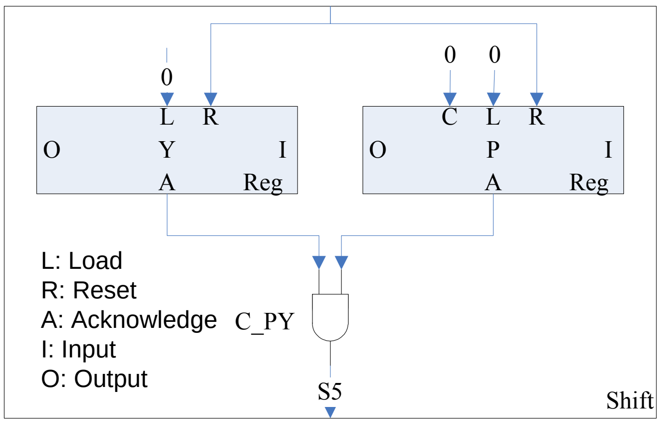

Figure 2.10: State 4, Shifting P and Y.

2.2.2.6. <u>Computing Partial Product (State 5)</u>

In State 5, a select module is requested to check if the three most significant bits of Y are identical. If they are, State 5 is exit, otherwise the adder is requested to perform any of P±X, P±2X, P±(N-X) or P±2(N-X) operations and its acknowledgment requests the register P to load the result. After register P’s acknowledgment, State 6 is triggered. This is outlined in Figure 2.11.

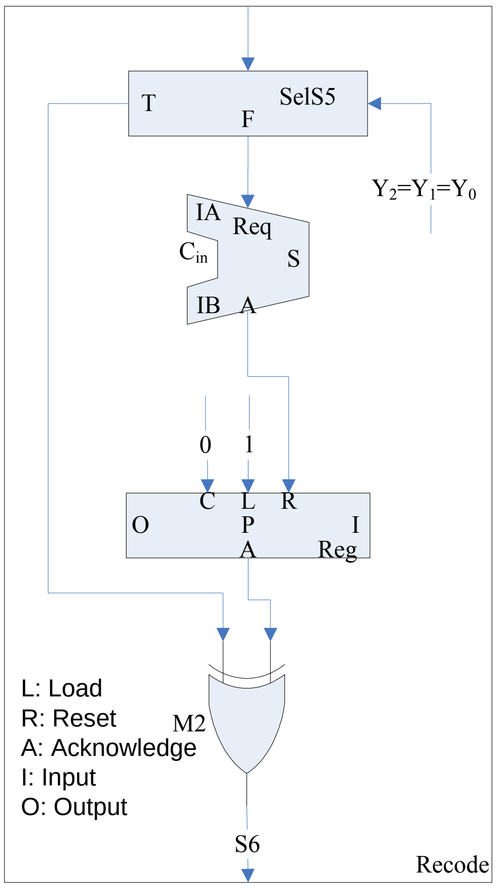

Figure 2.11: State 5, Computing Partial Products.

2.2.2.7. Scaling Partial Product (State 6)

In State 6, a select module is requested to check if the three most significant bits of P are identical. If they are, the loop select module is requested, otherwise the adder is requested to perform any of P±2N, P±3N, P±4N, P±5N or P±6N operations and its acknowledgment requests the register P to load the result. After that, register P's acknowledgment requests the loop select module. The loop select module requests Cntr to decrement if Cntr's value is greater than zero and then Cntr's acknowledgment triggers State 4. Otherwise, if Cntr's value is zero, State 7 is triggered. The date flow of this state is shown in Figure 2.12.

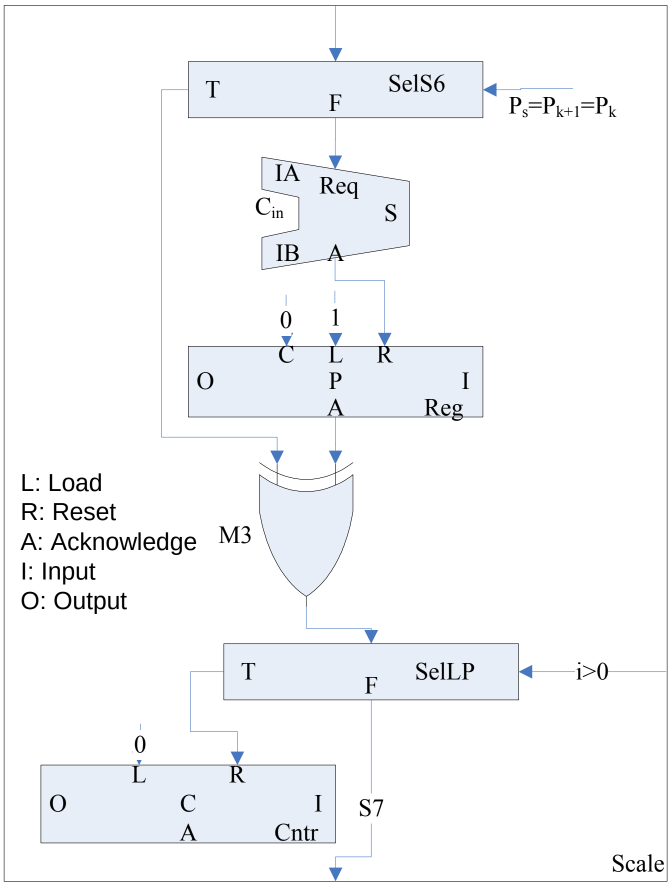

Figure 2.12: State 6, Scaling & Loop Condition Checking.

2.2.2.8. Correction (State 7)

In State 7, the adder is requested to perform P±N. Then, and its acknowledgment requests a select module to check if the sign of the result is negative and the sign of register P is positive. If true, the done signal is brought high. Otherwise, register P is requested to load the result and its acknowledgment requests the adder again as far as the done signal is low. If the sign bit of register P is zero at the time of register P's acknowledgment the done signal is brought high. The maximum number of adder requests in this state is two as mentioned in section 2.1.4. The data flow of this state is shown in Figure 2.13.

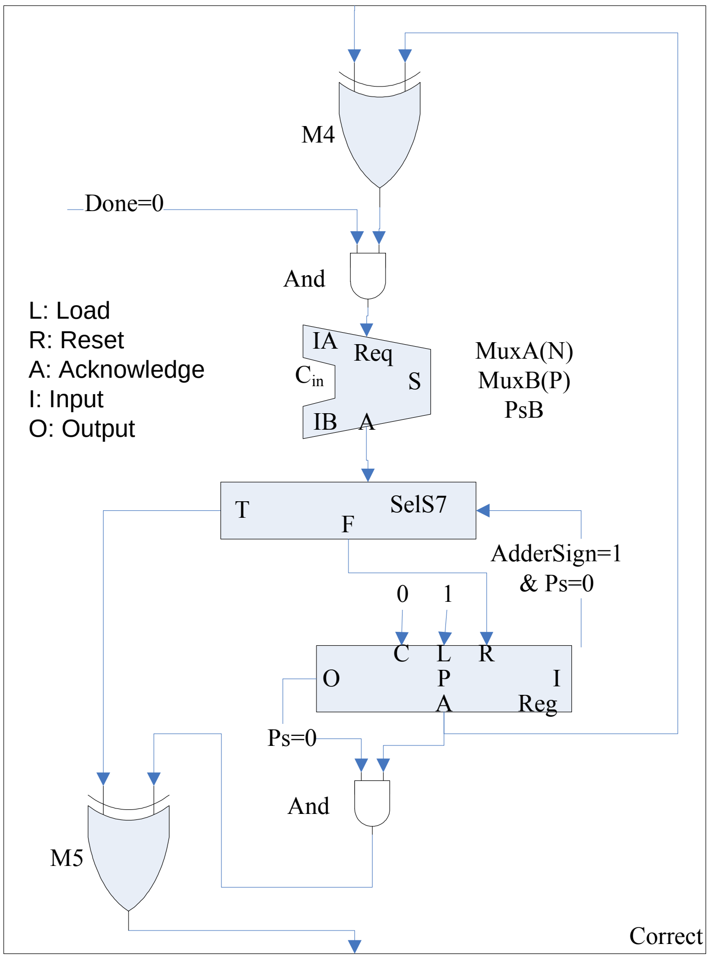

Figure 2.13: State 7, Correction.

2.3. ALGORITHM COMPLEXITY

2.3.1. Hardware Complexity

To reduce the hardware area, clocked asynchronous elements are used. The scaling step reduces the register width by a factor of two at the expense of an extra addition step per iteration. A one bit register is used as a hardware complexity reference and its hardware cost is assumed to be λ. Accordingly, the overall hardware (area) complexity consists of:

- Area cost of eight k-bit registers is $8k\,\lambda$
- Area cost of a k-bit asynchronous adder is $3\,k\,\lambda$

2.3.2. Time Complexity

Reducing the overall time and area complexity of modulo multiplication operation is one of the main purposes of the algorithm. The use of radix-4 reduces the number of iteration by factor of two, but we have two additions per iteration each is performed 3/4 of the time on average.

On the other hand, the use of an asynchronous adder reduces the average rippling to $O(log_2(k))$. The cost of $O(log\ k)$ can be expressed [12] [40]as $\alpha\ log\ k$, where $½ < \alpha < 2$.

Assuming v to be the delay of a two input gate, the average case delay is given by:

$$\tfrac{3}{4} \times \alpha\, k\, \log_2(k)\, v$$

Accordingly, the total area-delay Cost will approximately be:

$$33/4\ \alpha\, v\, \lambda\, k^2 \log_2(k) \approx 8 \times \alpha\, v\, \lambda\, k^2 \log_2(k)$$

□

CHAPTER 3

MONTGOMERY MULTIPLIER

(A COMPLETE SOLUTION)

This work investigates the use of a four-to-two compressor for the design of efficient modulo multipliers. With the large size operands commonly used in cryptosystems, using array or parallel multipliers would require prohibitively large area. Instead, sequential multipliers are employed in this work. Since sequential multipliers use repeated add and shift operations, a Compressor implementation can significantly improve the speed at a modest increase in area. For compressors, the speed of the addition is fixed regardless of the length of the operands. This comes at the cost of having the result in a redundant format.

The four-to-two compressor used in this design is different from the conventional four-to-two compressor. It is constructed from two full-adders and one half-adder as shown in Figure 3.1. It has a total of four inputs and two outputs with two carry-ins and two carry-outs. This particular design of the four-to-two compressor allows us to control the output of the least two bits and make them zero after every addition step as detailed in section 3.3.2.

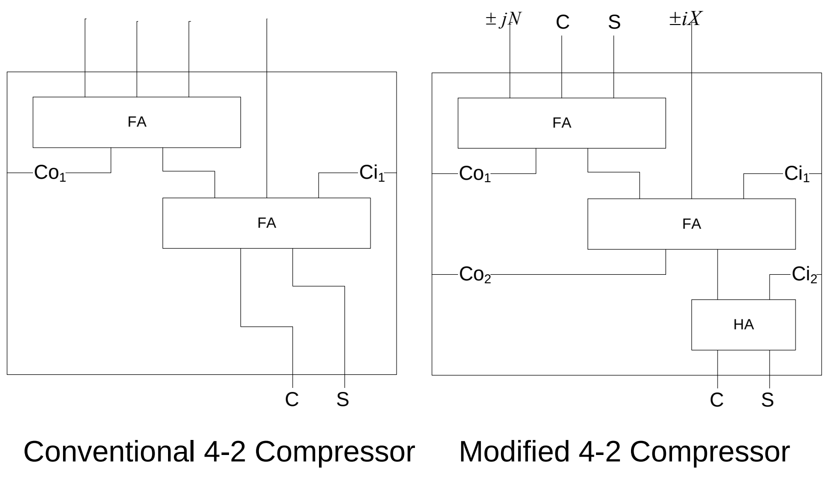

Figure 3.1: Four-to-Two Compressor Structure.

Some measures were adopted to further improve the overall system speed. For one, the developed algorithm uses radix-4 -which retires two bits per iteration instead of one and multiplier recoding into singed-digit format [4] is used to avoid computing multiple of operands that are not multiple of two.

3.1. MONTGOMERY’S MODULAR MULTIPLICATION

P. L. Montgomery introduces a new method of computing modular multiplication without the need for quotient determination [39]. In comparison to conventional SRT division method, Montgomery’s method needs considerable pre and post processing. However, this added processing can be neglected in case of repeated modular multiplication as the case of modular exponentiation.

3.1.1. Algorithm Parameters and Notations

For better understanding of the algorithm, we start by reviewing the notations used in the algorithm [6] [45].

- Modulus N is a k-bit difficult to factor odd integer.
- $R=2^k$.
- R^{-1} is the multiplicative inverse of R mod N (i.e., $(R^{-1}\times R) mod\ N=1$).
- $(N' = (R\times R^{-1}-1)/N) \equiv (R\times R^{-1}- N\times N'=1) \equiv (N'=-N^{1}\ mod\ R)$
- The N-residue of A is defined as $\overline{A}=(A\times R) mod\ N$
- The N-residue of B is defined as $\overline{B}=(B\times R) mod\ N$
- The Montgomery product of $\overline{A}$ and $\overline{B}$ is defined as follows:

$$\overline{C} = (\overline{A}\times\overline{B}\times R^{-1})\ mod\ N.$$

$$= (A\times R\times B\times R\times R^{-1})\ mod\ N.$$

$= (C \times R) \ mod \ N$

= the N-residue of C.

Where: $C=(A \times B)$

The new N-residue domain $\{(A \times R) \ mod \ N \mid 0 \leq A < N\}$ (referred to later as Montgomery's domain) contains all the values between 0 and (N-1) [6]. Therefore, it is one-to-one mapping between its elements and integers between 0 and (N-1). TABLE 3.1 shows the mapping between integers from zero to 12 and the 13-residue system where:

- N=13.
- R=2^4=16.

TABLE 3.1: Mapping Integers to 13-Residue Class.

A	$\overline{A}$ =(A×16)mod13
0	0
1	3
2	6
3	9
4	12
5	2
6	5
7	8
8	11
9	1
10	4
11	7
12	10

3.1.2. Algorithm Features.

Montgomery algorithm (Algorithm 3.1) requires the computation of *N'* and R^{-1}. These are time consuming operations and can be completed using Euclidean algorithm [3]. The other two time consuming tasks are the conversion to and from the Montgomery domain [6].

On the other hand, computations in the Montgomery domain are much faster since division is by *R* and the multiplication is modulo *R*, where *R* is a power of two number.

The result of Montgomery's multiplication is represented in Montgomery's domain. To convert it back to the ordinary domain, the result is fed back to the algorithm with one as the second operand.

Algorithm 3.1: Montgomry's Multiplication.

```
Objective:
Compute $\overline{C}$ =MonPro($\overline{A}$ , $\overline{B}$ )

Algorithm:
tmp1= $\overline{A} \times \overline{B}$
tmp2=(tmp1×N' mod R)×N
tmp3=(tmp1+tmp2)/R
IF tmp3≥N THEN
    {tmp3=tmp3-N
    }
$\overline{C}$ =tmp3.
```

3.1.2.1. Illustrative Example

Let: N=13, k=4, $R=2^k=16$, A=5 and B=3.

Then: $R^{-1}=9$, N'=11, $A^{bar}=2$, and $B^{bar}=9$.

MonPro(2,9)

1. tmp1=9×2=18
2. tmp2=(18×11 mod 16) ×13=6×13=78
3. tmp3=[18+78]/16=6
4. return(6)

Notice that as shown in TABLE 3.1, 6 in Montgomery domain corresponds to 2 in normal domain which is the correct result. Using the same algorithm again with the result (6) and 1 as operands, the result is converted back from the Montgomery domain.

One can notice that *N'* is needed only to make *tmp1+tmp2* divisible by *R*. Therefore, in case of binary interleaved modulo multiplication algorithm, the algorithm can be modified to function without the need for computing *N'* as shown in Algorithm 3.2.

The algorithm simply performs every binary multiplication step (add operation) then checks the result, if found to be odd the modulus is added/subtracted to make it even. Hence, the result is divisible by two, which is simply a shift right operation.

Algorithm 3.2: Modified Montgomery Multiplication.

```
Objective:
Compute $\overline{C}$ =MonPro2( $\overline{A}$ , $\overline{B}$ )

Algorithm:
tmp=0
FOR i=0 to k-1 DO
{   tmp=tmp+$a_i \times B^{bar}$
    tmp=tmp+$tmp_0 \times N$
    tmp=tmp/2
}
IF tmp≥N THEN
    {tmp=tmp-N
    }

$\overline{C}$ =tmp.

Where:
$a_i$ is the ith bit of $\overline{A}$ and $tmp_0$ is the LSB of tmp
```

Since the algorithm introduces factor of $2^{k^{-1}}$ or in general $r^{n^{-1}}$ where r is the radix and n is number of digits in the radix-r number, we can avoid the conversion from the ordinary domain to Montgomery's domain. To obtain the correct result, however, the result of the multiplication should be passed to the algorithm again with 2^{2k} (or r^{2n} in general) as the second operand [45].

Although there are many hardware implementations for Montgomery modular multiplication[6] [13] [38] [45] [51] [54], none of them have implemented the pre and post processing steps. In the following, we are going to show a complete design of Montgomery modular multiplier that implements the main algorithm together with the pre/post processing steps hence.

First, we will start by showing the design of radix-4 Montgomery multiplier. Then, a modification of the design will be made to enable it to perform the pre/post full modular multiplication operation.

3.2. DESCRIPTION OF THE ALGORITHM

In Montgomery modulo multiplication, a multiple of the multiplicand is added to the accumulator, and then a multiple of the modulus is subtracted to make the least significant bit(s) zero. In this chapter, the use of a four-to-two compressor allows simultaneous execution of these two operations. This comes at the expense of representing the result in a redundant format in two registers SUM and CARRY as shown in Figure 3.2.

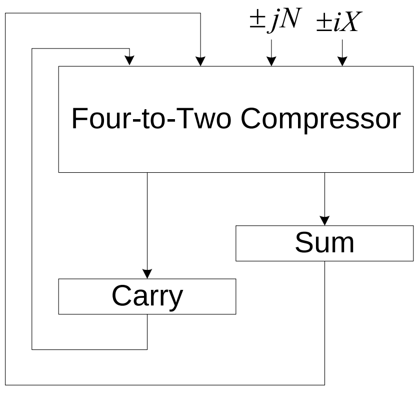

Figure 3.2: Simultaneous Use of Modulus and Multiplicand.

The algorithm uses an optimal technique to recode the multiplier to radix-4 signed-digit format. The use of radix-4 reduces the number of iterations by a factor of two and avoids the need to pre-compute multiples of the multiplicand since the recoded digits are 1, 2, and 4 –which are obtained by simple shift operation.

3.2.1. Approach

The design uses a right shift approach. Therefore, to keep the adder and register sizes at k-bit limit, modulus multiple must be chosen to make the least two bits of the accumulator – which is represented by the SAM and CARRY registers – zero. The multiplier is scanned from right-to-left and optimally recoded on-the-fly to allow proper selection of the multiplicand multiple and the modulus multiple using the recoding algorithm shown in TABLE 3.2. The least two significant bits of the SUM and CARRY outputs are guaranteed to be zero hence a right shift operation can be safely performed. After *k/2* iterations, the Montgomery modular multiplication is over. However, since the result could be any number between *-2N* and *2N*, the correction step is only one subtraction if the result is positive or a maximum of two additions if it is negative. Since the result is stored in a redundant format, a carry propagate adder is required to add the *SUM* and *CARRY* and identify the sign of the result. If instead of using a carry propagate adder, the carry save adder is used repeatedly till *CARRY* become zero, a maximum of *k/3* cycles (see APPENDIX C), and an average of $log_2(k)$ cycles to complete. The Montgomery modular multiplication algorithm is shown in Algorithm 3.3.

3.2.2. Illustrative Example

This example illustrate the algorithm behavior in Montgomery multiplication phase

Example: Compute $(3\times 8)\ Mod\ 13$.

Let A=3 and B=8, from TABLE 3.1, we find the 13-residues of A and B to be:

$\overline{A}$ =9

$\overline{B}$ =11

Compute *(9×11) MonPro 13*.

a. Initialization:

X=0000 1001, Y=0000 1011, N=0000 1101, i=3

C=0000 0000, S=0000 0000.

b. Right Shift Loop:

Right-Shift [*C*(0000 0000),*S*(0000 0000)]➔[*C*(0000 0000),*S*(0000 0000)]

Add [*N*(0000 1101),*C*(0000 0000),*S*(0000 0000),-*X*(1111 0111)]➔[*C*(0001 0100),*S*(1111 0000)]

i=2

Right-Shift [*C*(0001 0100),*S*(1111 0000)]➔[*C*(0000 0101),*S*(1111 1100)]

Add [(0000 0000),*C*(0000 0101),*S*(1111 1100),-*X*(1111 0111)]➔[*C*(0000 0100),*S*(1111 0100)]

i=1

Right-Shift [*C*(0000 0100),*S*(1111 0100)]➔[*C*(0000 0001),*S*(1111 1101)]

Add [*N*(0000 1101),*C*(0000 0001),*S*(1111 1101),*X*(0000 1001)]➔[*C*(0100 0100),*S*(1101 0000)]

i=0

d. Correction:

Add [*N*(0000 1101), *C*(0100 0100),*S*(1101 0000),(0000 0000)]➔[*C*(0010 0000),*S*(0000 0001)]

Add [-2*N*(1110 0110),*C*(0010 0000),*S*(0000 0001),(0000 0000)]➔[*C*(0000 0000),*S*(0000 0111)]

Done

From TABLE 3.1, we find that the result 7 maps to 11 in the integer domain, which is the correct answer.

Algorithm 3.3: Montgory Modular Multiplication.

```
a. Initialization:
    Counter = k/2+1
    Clear the Sum and Carry registers
    Multiplier= Multiplier*4
b. Right shift Loop:
    While Counter >0
    {   Right Shift (Multiplier, Sum, Carry).
        Add (Multiple of N, Carry, Sum, Multiple of X)
        Decrement Counter
    }
c. Correction:
    If (estimated Accumulator sign is positive)
    {   Add(Multiple of N, Carry, Sum, 0)
        While Carry $\neq$ 0
        {   Add(0, Sum, Carry, 0)
        }
        If (Accumulator sign is positive)
        {   Return(Sum)
        }
    }
    While (estimated Accumulator sign is negative)
    {   Add(Multiple of N, Carry, Sum, 0)
        While Carry $\neq$ 0
        {   Add(0, Sum, Carry, 0)
        }
    }
    Return(Sum)
```

3.3. DESIGN ISSUES

3.3.1. An Optimal Radix-4 Right-To-Left Recoding Algorithm

In this algorithm, we use a tag bit (Tg) which is set initially to 0. In addition, the number to be recoded is left padded with one bit (0) if it has even number of bits or two bits if it has an odd number of bits. The number is then scanned from LSB to MSB with one look ahead (left) bit. TABLE 3.2 illustrates the recoding mechanism.

TABLE 3.2: Radix 4 Optimal Recoding Algorithm.

Case No.	OUT			IN				Note
	Y_{i+1}	Y_i	Tg	X_{i+2}	X_{i+1}	X_i	Tg	
1	0	0	0	×	0	0	0	In the middle of 0 series.
2	0	1	0	×	0	0	1	End of 1 series and starting 0 series.
3	0	1	0	×	0	1	0	1 between 0's.
4	1	0	0	0	0	1	1	End of 1 series and starting 0 series.
5	$\bar{1}$	0	1	1	0	1	1	0 between 1 series and a single 1, so we can't make it 1. Instead we make it -1 to be a beginning of 1 series.
6	1	0	0	0	1	0	0	1 between 0's.
7	$\bar{1}$	0	1	1	1	0	0	End of 0 series and beginning of 1 series.
8	0	$\bar{1}$	1	×	1	0	1	0 between 1 series and a single 1, so we can't make it 1. Instead we make it -1 to be a beginning of 1 series.
9	0	$\bar{1}$	1	×	1	1	0	End of 0 series and s beginning of 1 series.
10	0	0	1	×	1	1	1	In the middle of 1 series.

3.3.2. <u>N-Multiple Selection For Shift Right Loop</u>

A proper selection of N multiple to be subtracted/added to ensure that the least significant two significant bits of the result are zero. This is essential for the design to only use k-bit processing, e.g. k-bit addition. Yet another condition is needed for the modulus selection criterion, that is the result in the accumulator after adding the selected multiple of modulus and multiplicand should be contained in $k + 2$ bits. As a result, the accumulator will have a k-bit number after right shifting by two bits. TABLE 3.3 shows how the modulus multiple is selected. Whenever we have the choice of positive or negative multiple of the modulus, the opposite sign of accumulator is selected. As an example, assume the following:

N=1001, X=1100

Let's assume that at some stage we have *SUM*=00111001 and *CARRY*=01001001.

This makes the summation of the least significant two bits of *SUM* and *CARRY* equals two (10). Since the least significant two bits of X equals zero (00), from the third entry of TABLE 3.3 we have the choice of selecting $\pm 2N$. Since the estimated sign of the accumulator (*SUM*+*CARRY*) is positive $-2N$ is selected.

TABLE 3.3: Modulus Selection for Shift Right Loop.

No	$X_{1:0}$	$C_{1:0}+S_{1:0}$	$N_{1:0}$	
			01	11
1	0	0	0	0
2	0	1	-1	1
3	0	2	±2	±2
4	0	3	1	-1
5	0	4	0	0
6	0	5	-1	1
7	0	6	±2	±2
8	1	0	-1	1
9	1	1	±2	±2
10	1	2	1	-1
11	1	3	0	0
12	1	4	-1	1
13	1	5	±2	±2
14	1	6	1	-1
15	2	0	±2	±2
16	2	1	1	-1
17	2	2	0	0
18	2	3	-1	1
19	2	4	±2	±2
20	2	5	1	-1
21	2	6	0	0
22	3	0	1	-1
23	3	1	0	0
24	3	2	-1	1
25	3	3	±2	±2
26	3	4	1	-1
27	3	5	0	0
28	3	6	-1	1

3.3.3. Data Path Design

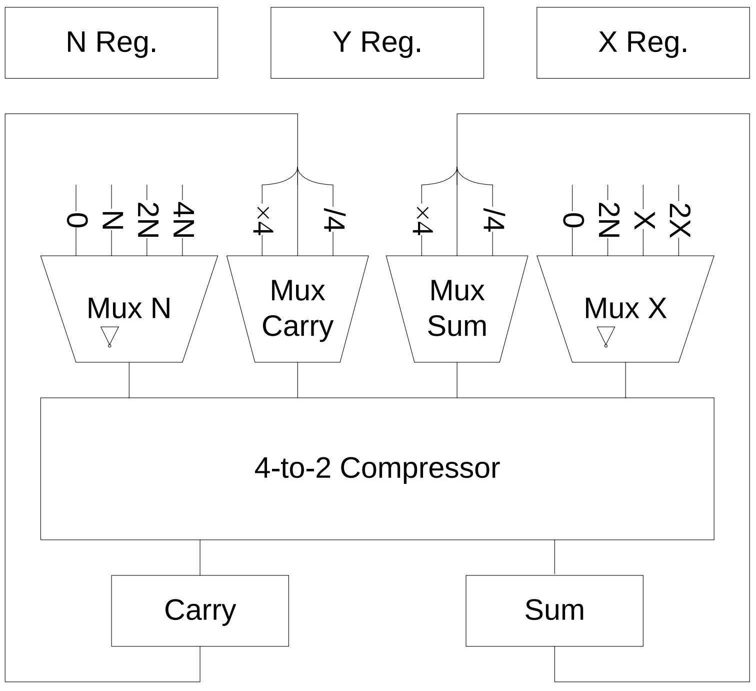

Figure 3.3: Compressor Multiplier Data Path.

The data path of this design is shown in Figure 3.3 and the main components of the data path are described as follows:

- **Adder:** A specially designed four-to-two compressor shown in Figure 3.4 is used in this design. Proper selection of the modulus multiple is guarantees that the two least significant bits of the result (*SUM* + *CARRY*) will be zero. This allows shifting the result to right without needing another register to keep the lower half of the result.

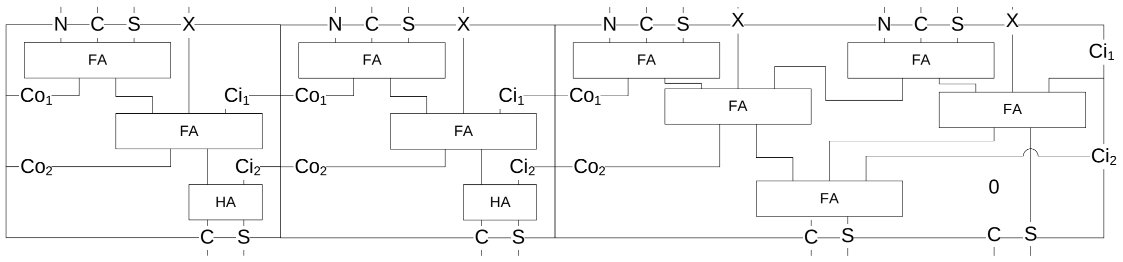

Figure 3.4: Four-to-Two Compressor.

- **Multiplexers:** The design has four multiplexers, one to select a proper multiple of the modulus N (*Multiplexer-N*), and another is to select the proper multiple of the multiplicand X (*Multiplexer-X*). The other two multiplexers are used to generate a shifted value for the *SUM* and *CARRY* registers. Possible multiples of N are N, 2N, 4N or 6N where 6N multiple is needed only during the shift left loop –in which no multiple of X is needed-. Therefore, it can be obtained by selecting 4N from *multiplexer-N* and 2N form *multiplexer-X* to avoid computing a multiple of N that is not power of two. This implies that *multiplexer-X* should get N and X as inputs, and gives 2N, X, 2X or zeros as output. *Multiplexer-N* gets N as an input and gives N, 2N, 4N or zeros as output. Both multiplexers have the capability of providing these values or their one's complement simply by XORing them with one. To get the two's complement value of either of them, one is fed to the first Carry-in of the adder. In case that the two's complement is needed on both multiplexers; one is fed to the second Carry-in of the adder.
- **Registers:** This design needs only five registers; three for the inputs multiplicand (X), multiplier (Y) and modulus (N) and two for the accumulator represented in a redundant format CARRY and SUM. While registers X and N are simple

registers, register Y needs the shift right by two bits capability. Yet the accumulator registers (CARRY and SUM) are more complicated since they need the capability of shifting two bits to the right or to the left. Furthermore, the Carry register should have the capability of detecting Zero value in it. This is essential to know that the carry rippling phase is over.

- **Counter:** The counter is used to control the number of loop iterations.

3.4. A Complete Hardware Implementation for Montgomery Multiplication

The mechanism of making the least two bits zero and shifting the result right introduces the factor $2^{k^{-1}}$ to the result which is required by Montgomery multiplication algorithm. However, in the pre/post phase, it is required to eliminate this factor to compute the regular modulo multiplication result. Therefore, another loop is needed to eliminate this factor by shifting the accumulator to the left and subtracting a proper multiple of the modulus *N* to prevent overflow.

To have the capability of performing the Montgomery multiplication together with its pre and post calculation on the same hardware, the controller need one bit variable to know if the current phase is a Montgomery multiplication or pre/post phase. In case of pre/post phase, the shift-left loop stage of the algorithm will be executed. On the other hand, during the Montgomery multiplication the algorithm will skip the shift-left phase as shown in Figure 3.5. A complete description is shown in Algorithm 3.4.

3.4.1. Modulus Selection for Shift Left Loop

The idea of this step is to keep the result in a k-bit boundary. The two most significant bits of both the accumulator and the modulus are inspected to determine the proper multiple of the modulus to be added and the selected modulus multiple is always the opposite sign of the accumulator's. As shown in TABLE 3.4, there are cases where 6×N is needed. This value is obtained by selecting 4×N and 2×N to two inputs of the compressor.

TABLE 3.4: Modulus Selection for Shift Left Loop.

Sign	$P_{k+2:k+1}$	$N_{k-1:k-2}$	Selected N
+	0	2	-1
+	0	3	-1
+	1	2	-2
+	1	3	-2
+	2	2	-4
+	2	3	-2
+	3	2	-6
+	3	3	-4
-	0	2	1
-	0	3	1
-	1	2	2
-	1	3	2
-	2	2	4
-	2	3	2
-	3	2	6
-	3	3	4

3.4.2. Illustrative Example

This example illustrate the algorithm behavior in the pre/post phase

Compute *(9×11) Mod 13*.

a. Initialization:

X=0000 1001, Y=0000 1011, N=0000 1101, i=3

C=0000 0000, S=0000 0000.

b. Right Shift Loop:

Right-Shift [*C*(0000 0000),*S*(0000 0000)]➔[*C*(0000 0000),*S*(0000 0000)]

Add [*N*(0000 1101),*C*(0000 0000),*S*(0000 0000),-*X*(1111 0111)]➔[*C*(0001 0100),*S*(1111 0000)]

i=2

Right-Shift [*C*(0001 0100),*S*(1111 0000)]➔[*C*(0000 0101),*S*(1111 1100)]

Add [(0000 0000),*C*(0000 0101),*S*(1111 1100),-*X*(1111 0111)]➔[*C*(0000 0100),*S*(1111 0100)]

i=1

Right-Shift [*C*(0000 0100),*S*(1111 0100)]➔[*C*(0000 0001),*S*(1111 1101)]

Add [*N*(0000 1101),*C*(0000 0001),*S*(1111 1101),*X*(0000 1001)]➔[*C*(0100 0100),*S*(1101 0000)]

i=0

c. Left-Shift Loop:

Add [(0000 0000),*C*(0100 0100),*S*(1101 0000),(0000 0000)]➔[*C*(0000 0000),*S*(0001 0100)]

i=3

Add [-*N*(1111 0011), *C*(0000 0000),*S*(0001 0100),(0000 0000)]➔[*C*(1000 0000),*S*(1000 0111)]

i=2

Left-Shift [*C*(1000 0000),*S*(1000 0111)]➔[*C*(0000 0000),*S*(0001 1100)]

Add [-*N*(1111 0011),*C*(0000 0000),*S*(0001 1100),(0000 0000)]➔[*C*(1000 0000),*S*(1000 1111)]

i=1

Left-Shift [*C*(1000 0000),*S*(1000 1111)]➔[*C*(0000 0000),*S*(0011 1100)]

Add [-2*N*(1110 0110),*C*(0000 0000),*S*(0011 1100),(0000 0000)]➔[*C*(0010 0000),*S*(0000 0010)]

i=0

d. Correction:

Add [-*N*(1111 0011), *C*(0010 0000),*S*(0000 0010),(0000 0000)]➔[*C*(0000 0000),*S*(0000 1000)]

Done

Algorithm 3.4: A Complete Montgory Modular Multiplication.

```
a. Initialization:
    Counter = k/2+1
    Clear the Sum and Carry registers
        Multiplier= Multiplier*4
b. Right shift Loop:
    While Counter >0
    {   Right Shift (Multiplier, Sum, Carry).
        Add (Multiple of N, Carry, Sum, Multiple of X)
        Decrement Counter
    }
c. Left Shift Loop:
    If MonPro=0       -- (i.e., Pre/Post processing phase)
    {   Counter = K/2+1
        While Counter >0
        {   Add (Multiple of N, Carry, Sum, Multiple of N)
            Decrement Counter
            If Counter >0
            {   Left Shift (Sum, Carry)
            }
        }
    }
d. Correction:
    If (estimated Accumulator sign is positive)
    {   Add(Multiple of N, Carry, Sum, 0)
        While Carry ≠ 0
        {   Add(0, Sum, Carry, 0)
        }
        If (Accumulator sign is positive)
        {   Return(Sum)
        }
    }
    While (estimated Accumulator sign is negative)
    {   Add(Multiple of N, Carry, Sum, 0)
        While Carry ≠ 0
        {   Add(0, Sum, Carry, 0)
        }
    }
    Return(Sum)
```

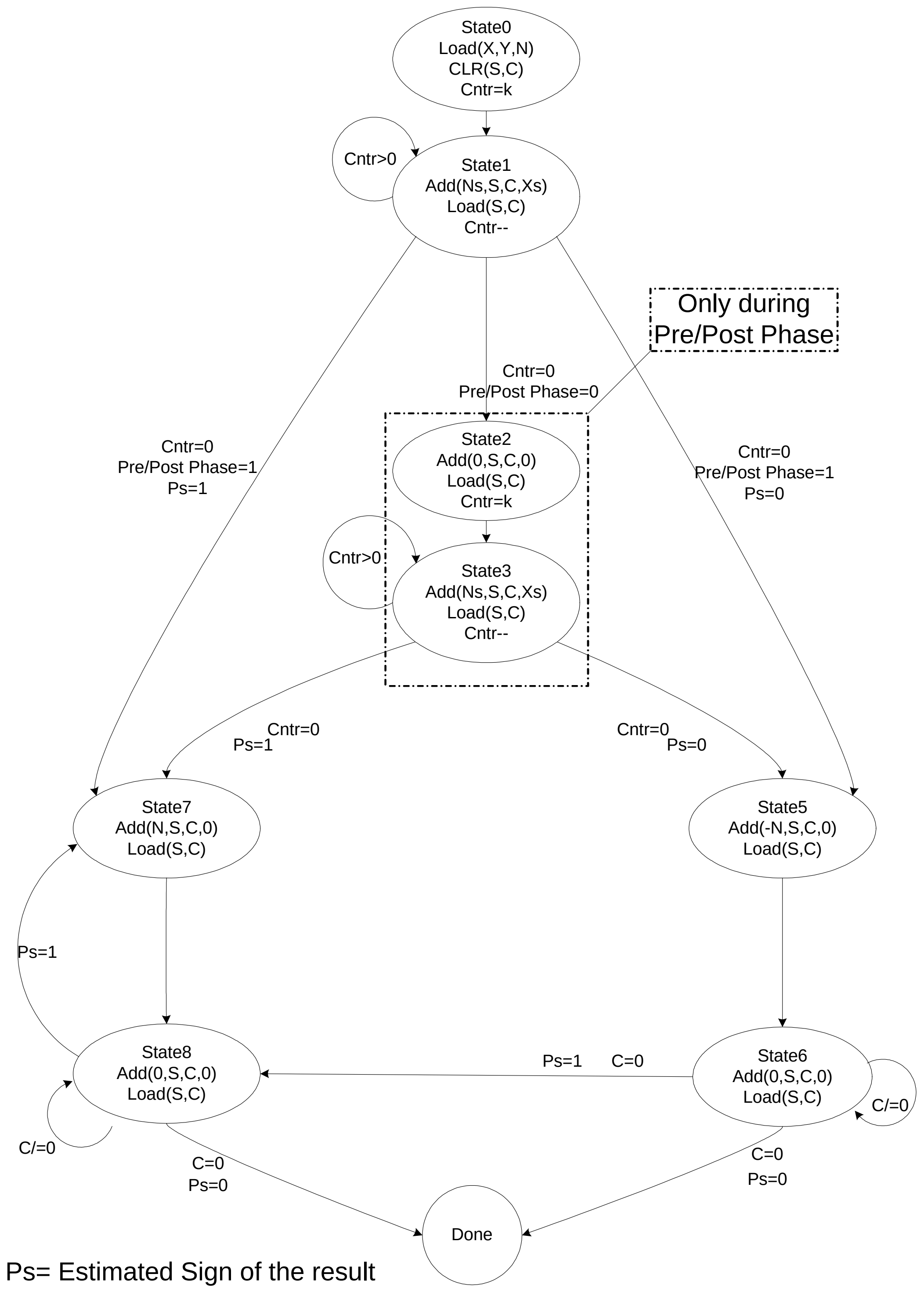

Figure 3.5: Montgomery Modulo Multiplier State Diagram.

3.5. ALGORITHM COMPLEXITY

3.5.1. Hardware Complexity

The hardware complexity of the algorithm is as follows:

- Area cost for five registers is *5k* λ
- Area cost for the capability of some registers to both shift right and left and detecting zero is *2k* λ
- Area cost of the k-bit compressor is *5 k* λ

3.5.2. Time Complexity

In the case of repeated modulo multiplication the cost of states 2, and 3 (Figure 3.5) becomes negligible. States zero and one need (k/2+1) clock cycles and after state one the design branches. In this section, the best, average, and worst case number of clock cycles is presented.

The best case occurs when the result is positive and no rippling is required, one clock cycle is needed for each of state five and state six. Therefore, the number of cycles in the best case is:

$$((k/2+1)+2)$$

In the worst case, we have negative result and it needs two correction steps. Therefore, the negative branch is passed twice where state eight needs one clock cycle and state nine needs maximum rippling time, which is k/3 cycles (see APPENDIX C). The number of cycles in the worst case is:

$$((k/2+1)+2(k/3+1))$$

On the average, we need 1½ correction steps, while rippling takes $O(\log_2(k))$ cycles. As in [12], the maximum value of α for rippling is only one. Therefore, the number of cycles on the average is:

$$((k/2+1)+ 1\tfrac{1}{2} (\log_2(k)+1))$$

The clock period is the worst case delay of the four-to-two compressor plus the register loading time. For the four-to-two compressor used in this work, the worst compressor delay is the delay of five XOR gates that is $10\times v$ delay (see APPENDIX C), where v is a 2-input gate delay.

Thus the average area-delay cost will be:

$$(12k)\ \lambda \times 10\times v\ ((k/2+1)+ 1\tfrac{1}{2} (\log(k)+1)) \cong (60k^2+180\ k \log_2(k)+300k)\ \lambda\ v$$

□

CHAPTER 4

TESTING OF THE FOUR-TO-TWO COMPRESSOR ARRAY

An **I**terative **L**ogic **A**rray (ILA) is a logic array that is composed of combinational modules (cells), connected in a regular manner (array). With large number of cells, ILAs typically have large number of inputs. This makes the task of exhaustive testing of these arrays quite prohibitive. The increased use of synthesis tools has caused internal implementations of the ILAs to be quite abstract to the designer. Accordingly, test methodologies for such synthesized hardware typically use functional fault models. However, it is quite impractical to exhaustively test ILAs using conventional fault models because of their huge number of inputs.

Different fault models are used to test ILAs using C-testability [1] [10] [20] [28] [42]. Using the C-testability concept, the whole ILA can be tested with a fixed number of test patterns regardless of size of the array.

In this chapter, we will consider two fault models for ILAs. Then, a **B**uilt **I**n **S**elf **T**est (BIST) methodology for the used four-to-two compressor ILA will be outlined. After that, a practical example will be used to illustrate the used of C-testability on the four-to-two compressor ILAs, and BIST for this ILA will be provided.

C-TESTABILITY FOR ITERATIVE LOGIC ARRAY

Iterative Logic Array (ILA) can be one or two dimensional. A one-dimensional ILA has all of its cells connected in one row (or column). If the data flow in the ILA goes in one direction, the system is known as *unilateral* ILA, otherwise, it is called *bilateral* ILA. In two-dimensional ILAs, the cells are connected in rows and columns where cell C_{ij} is the cell in the *ith* row and *jth* column. ILAs are said to be testable if it is possible to detect any faulty cell in the array [1].

4.1.1. Preliminaries

- *Single Input Change* (SIC): a given test sequence is called SIC if the Hamming distance between consecutive test vectors is one. Any circuit can be designed to be fully robustly testable with respect to stuck-open faults using SIC pairs only [11].
- *Variable Testability measure* (VTM): "*is the coefficient assigned to each bit of the input and output variables of a given functional primitive, and is a separate measure for each bit*". VTM for a bit is the minimum number of test vectors needed to test this particular bit. It allows prediction of the number of test vectors that are needed for a given primitive. Thus, VTM deals with the functional level of abstraction, and can be considered in testing of data paths [28].
- An ILA is said to be *C-testable* if it can be tested with a constant number of test vectors regardless of the array size [10] [20].

- An ILA is said to be *O-testable* (Optimal-testable) or *M-testable* (Minimal-testable) if it can be tested with a minimum test set that is equivalent to the minimum test set needed to test one cell. M-Testability can also be used for ILA with non-identical cells and also for data paths (at the functional level) [28] [42].

4.1.2. <u>Circuit Hazards</u>

Hazards can be classified into two categories; *function* and *logic* hazards.

- A circuit F is said to have a *function hazard* for input transition from A to C if the following conditions are satisfied:
  1. The circuit has to pass by the input state B in its transition from A to C.
  2. $F(A) = F(C) \neq F(B)$.
- A circuit f is said to have a *logic hazard* for input transition from A to B if the following conditions are satisfied:
  1. $F(A) = F(B)$.
  2. There is no function hazard for the transition from A to B.
  3. Due to timing skews and delays inside the circuit, glitch(es) appear at the circuit output.

Logic hazards are dynamic property of the circuit and they depend on the propagation delays of various signals. Logic hazards can be avoided by using careful design strategies. However, one can never avoid function hazards [11].

4.1.3. <u>Test Invalidation</u>

In sequential fault testing, a fault is detected by a sequence of two test vectors. An initialization test vector and an excitation test vector. The initialization test vector prepares the circuit nodes for fault excitation by the second test vector. Non-robust test patterns, however, maybe invalidated due to time or delay skews. Test invalidation will cause the target fault to escape detection. Therefore, robustness must be considered in sequential fault test patterns [11] [41] [50].

4.1.4. <u>Robustness</u>

By reducing the Hamming distance between test-vectors, the probability of test invalidation decreases. Therefore, if we use Single input Change (SIC), we will achieve the highest robustness. SIC means that pairs for test pattern $<V_i, V_j>$ are different only by one bit [11] [41]. There are two levels of robustness:

1. **<u>Cell Level Robustness:</u>** The test pattern pair applied on a cell can not be invalidated due to function hazards in the cell given that there are no glitches at the cell inputs [11] [41].
2. **<u>Array Level Robustness:</u>** The inputs of a tested cell must not have any glitches and the changes on the inputs of other fault free cells do not affect the propagation of a fault to a primary output. Array level robustness can be guaranteed if the following conditions are satisfied:

a) The inputs of the tested cell receive SIC without glitches through test application cells (cells that affect the input of the tested cell).
b) The fault is propagated to primary output(s) through fault propagation cells (cells that propagate the fault to primary output(s)) robustly [11] [41].

4.1.5. <u>Fault Models For ILA</u>

4.1.5.1. <u>Cell Fault Model</u>

In cell fault model (CFM), we assume that at most one cell is at fault, and that the fault does not convert the cell to a sequential one. The fault is also assumed to be permanent and can affect the cell's output in any manner [10] [27] [41].

To test an ILA using CFM, all cells are exhaustively tested and the output of the faulty cell is propagated to a primary output. The CFM does not require any knowledge of the cell's internal structure [10] [27].

Using CFM, the lower limit of the number of test vectors to test an ILA is ($2^m.(2^n-1)$) where m and n are the numbers of cell inputs and outputs, respectively.

If sequential faults are considered, upon input change, every output might either keep its previous value (due to the fault), or change its value. Therefore, the lower limit on the number of test vectors to exhaustively test an ILA with sequential faults is ($2^m.(2^m-1).(2^n-1)$). This makes CFM unrealistic for testing sequential faults [11] [42].

4.1.5.2. Realistic Sequential Cell Fault Model

For CMOS technology, the assumption of only combinational faults of the CFM is unrealistic, and sequential faults should be considered. These faults may be as transistor stuck-open faults, gate delay faults, or path delay faults. Unlike combinational faults, each sequential fault requires a pair of test vectors (an initialization vector, and a test vector) to be detected. A fault model that considers these fault types is called a sequential Fault Model (FM) [11] [42]. For a sequential fault model to be efficient it should satisfy the following three conditions:

1. The FM should be comprehensive including CMOS stuck-open faults, but with reasonable test set size.
2. Because of the increasing use of synthesis tools, the FM should be independent of internal circuit implementation.
3. The FM should avoid test invalidation (robustness) [41].

In realistic sequential cell fault model (RS-CFM) [11], it is assumed that at most one cell is at fault, and that *test application* should contain all possible single input change (SIC) pairs. In RS-CFM a fault is detected when the output of the faulty cell does not change while it should in a fault-free cell. In addition, the faulty cell output(s) should be *propagated* to primary output(s) [11].

By using SIC in RS-CFM the first and third conditions are satisfied. The total number of SIC pairs in m-input cell is ($m.2^m.(2^{n+1}-2-n)/n$) which is reasonably reduced compared to the number of exhaustive test pattern pairs of ($2^m.(2^m-1).(2^n-1)$). RS-CFM requires no knowledge of the cell's internal implementation. However, an important assumption is that the cell design is free of logic hazards [11].

4.2. A BIST METHODOLOGY FOR ILA.

A **B**uilt-**In** **S**elf-**T**est (BIST) structure allows a circuit to test itself. A BIST structure consists of a **T**est **P**attern **G**enerator (TPG) and an **O**utput **R**esponse **A**nalyzer (ORA) [25].

In pseudo-exhaustive testing, the **C**ircuit **U**nder **T**est (CUT) is partitioned into several modules, each is tested exhaustively. Thus, it is guaranteed to detect all detectable faults in each module.

In this work, a modification is done on the pseudo-exhaustive test method. Instead of guaranteeing that all detectable faults in a module are detected, we guarantee that all detectable faults that can occur in normal operation are detected.

Each cell of the ILA has vertical inputs, vertical outputs, horizontal (lateral) inputs, and horizontal outputs. The assumption is that all possible combinations of the cell's vertical inputs are of acceptable size, and some set (say S_A) of all possible combinations of the cell's horizontal inputs are of acceptable size [23].

The algorithm runs as follows:

- Apply all possible input patterns (vertical and horizontal) to the leftmost module so as to know all possible output patterns on the horizontal outputs.

- On the next module, apply only the output patterns of the first module as horizontal inputs (and all possible vertical input patterns) and observe horizontal output patterns.
- Repeat for all succeeding modules until the output horizontal patterns are equal to the input patterns. This pattern is called S_A [9].

4.3. TESTABILITY OF THE 4-2 COMPRESSOR

In this section, we show how to apply C-testability on a one dimensional ILA with a basic cell composed of 4-2 compressor. The structure of the 4-2 compressor ILA is shown in Figure 4.1.

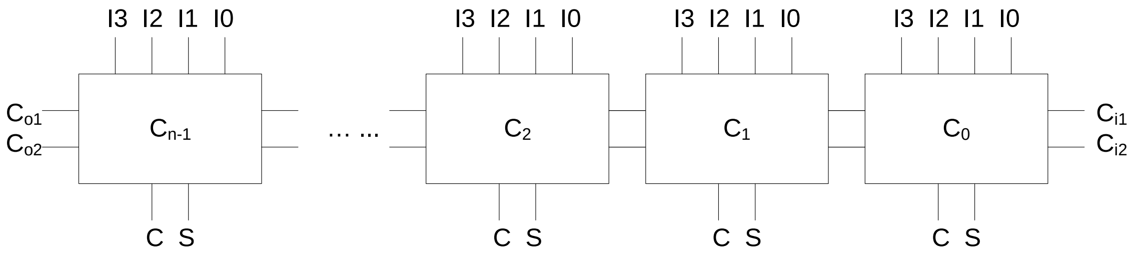

Figure 4.1: One Dimensional 4-2 Adder ILA.

Theorem: For a one dimensional ILA defined by a basic cell C in which all single cell functional faults are propagated by all inputs, the ILA is O-testable if the state diagram of C can be partitioned into disjoint subsets of branches where each subset defines a closed cycle [1].

Figure 4.2 shows the state diagram of the basic (4-2 compressor) cell where the states 00, 01, 10, and 11 correspond to the value of the carry-out of the 4-2 compressor (C_{out2} C_{out1}). The carry-out of a compressor cell is applied as carry-in for its succeeding cell. Figure 4.3 shows one possible partitioning of the state diagram to follow the O-testability theorem. The labels on the arrows represent the value of the vertical inputs while the source state shows the value of the carry-in and the target state shows the value of the carry-out. For example, if the source state is 00 and the target state is 11 the arrow label will be F, which means if the carry-in is 00 and we apply a value F on the vertical inputs of the CUT, the carry-out will be 11.

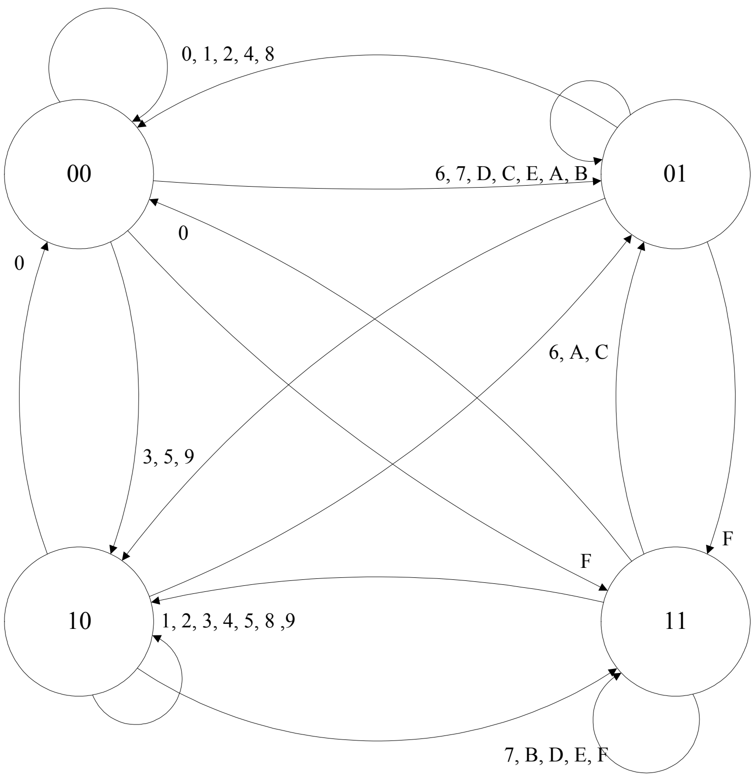

Figure 4.2: 4-2 Adder's Carry-Out State Diagram.

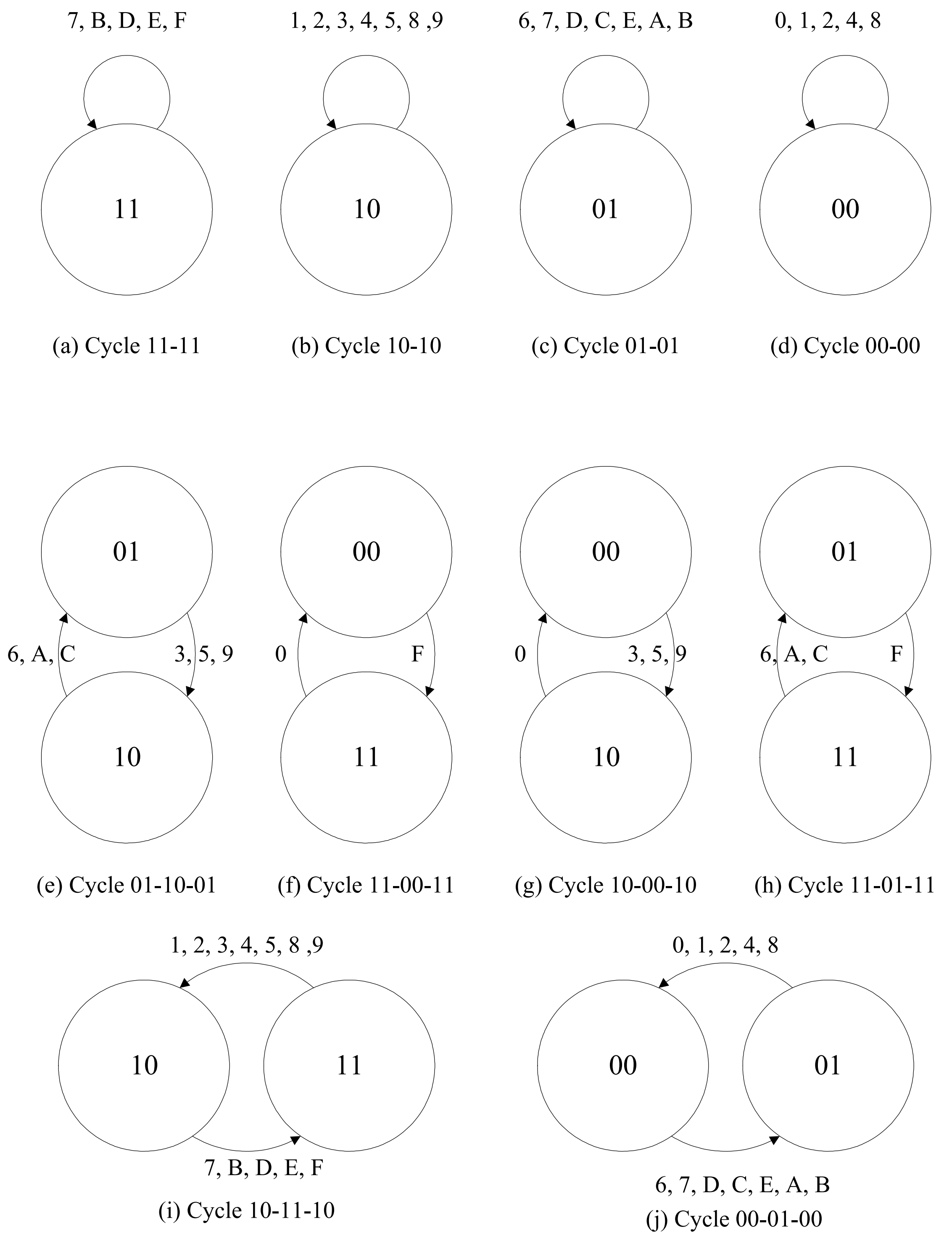

Figure 4.3: Possible Partitioning of the State Diagram 4-2 Adder.

In the following, the cost of a test set is chosen to be the summation of the Hamming distance between all the consecutive test-vectors. The objective is to make the circuit O-testable using the RS-CFM fault model as much as possible. Since the basic compressor cell has six inputs, we need to test the system use only 64 (2^6) test vectors so as to keep it O-testable. However, for the RS-CFM each test vector needs to have an initialization vector and the Hamming distance between consecutive test-vectors must be one. In other words, the best case scenario will be a test set of 64 test vectors with a total cost of 64 only. Another essential condition is that the vertical outputs of every consecutive pair of test vectors must be different to enable detection of faults.

The task of ordering 64 vectors to satisfy the above conditions mentioned is non trivial if at all possible. Therefore, a genetic evolutionary algorithm was developed to conduct the ordering problem (see APPENDIX D). For more background on evolutionary algorithms see [47].

TABLE 4.1 shows a test set with 64 test vectors and a total cost of 84. The carry in inputs of the test vectors 1, 37, 42, 47, 53, 56, and 59 are not the same as the carry out of their predecessors. Therefore, we have manually inserted some vectors in between to maintain the lateral output flow. For example, if we take vector 1 in TABLE 4.1, we find that its carry-out is 11, however the carry-in of vector 2 is 00. in order to maintain the lateral output-input flow, an extra test vector that has 11 as carry-in and 00 as carry-out is inserted as shown in TABLE 4.2. The new test set contains 74 test vectors and has total Hamming distance cost of 99.

TABLE 4.1: Genetic Algorithm's Output Test Set.

	Outputs				Inputs					
No	Co_2	Co_1	C	S	I_3	I_2	I_1	I_0	Ci_2	Ci_1
0	1	0	0	1	1	0	0	0	1	1
1	0	0	1	0	0	0	0	0	1	1
2	0	0	0	1	1	0	0	0	0	0
3	1	0	0	0	1	0	0	1	0	0
4	1	0	0	1	1	0	0	1	1	0
5	0	1	1	0	1	0	1	1	1	0
6	0	1	0	1	1	0	1	0	0	1
7	1	0	0	0	0	0	1	0	0	1
8	1	0	0	1	0	0	1	1	1	0
9	0	0	1	0	0	0	1	0	1	0
10	0	0	0	1	0	0	1	0	0	0
11	0	1	0	0	1	0	1	0	0	0
12	1	0	0	1	1	0	0	1	0	1
13	0	0	1	0	1	0	0	0	1	0
14	0	0	0	1	0	0	0	1	0	0
15	1	0	0	0	0	1	0	1	0	0
16	0	1	0	1	1	1	0	0	1	0
17	1	0	0	0	1	0	0	0	0	1
18	0	0	0	1	0	0	0	0	1	0
19	1	1	0	0	1	1	1	1	0	0
20	1	1	0	1	1	1	1	0	1	1
21	0	1	1	0	1	0	1	0	1	1
22	1	0	0	1	0	0	1	1	0	1
23	0	1	1	0	0	1	1	1	1	0
24	1	1	0	1	1	1	1	1	0	1
25	1	1	1	0	1	1	1	1	1	1
26	1	1	0	1	1	1	0	1	1	1
27	0	1	1	0	1	1	0	0	1	1
28	1	0	0	1	0	1	0	1	0	1
29	0	1	1	0	1	1	0	1	1	0
30	0	1	0	1	1	1	0	0	0	1
31	1	1	0	0	1	1	1	0	0	1
32	1	1	0	1	1	0	1	1	1	1
33	1	0	1	0	0	0	1	1	1	1
34	0	1	0	1	1	0	1	0	1	0
35	1	1	0	0	1	0	1	1	0	1
36	1	0	1	0	1	0	0	1	1	1
37	0	1	0	1	1	0	1	1	0	0
38	1	0	0	0	0	0	0	1	0	1
39	1	0	0	1	0	1	0	1	1	0
40	0	0	1	0	0	1	0	0	1	0
41	0	1	0	1	1	1	1	0	0	0
42	0	1	0	0	1	1	0	0	0	0
43	0	0	0	1	0	0	0	0	0	1
44	1	0	0	0	0	0	1	1	0	0
45	1	1	0	1	1	1	1	1	1	0
46	1	0	1	0	0	1	0	1	1	1
47	1	0	0	1	0	0	0	1	1	1
48	0	0	1	0	0	0	0	1	1	0
49	0	1	0	1	1	1	0	1	0	0
50	1	1	0	0	1	1	0	1	0	1
51	1	1	0	1	0	1	1	1	1	1
52	0	1	1	0	0	1	1	0	1	1
53	0	1	0	1	0	1	1	1	0	0
54	1	1	0	0	0	1	1	1	0	1
55	1	0	0	1	0	1	0	0	1	1
56	0	0	0	0	0	0	0	0	0	0
57	0	0	0	1	0	1	0	0	0	0
58	0	1	0	0	0	1	1	0	0	0
59	1	0	0	1	0	0	1	0	1	1
60	0	1	1	0	1	1	1	0	1	0
61	0	1	0	1	0	1	1	0	0	1
62	1	0	0	0	0	1	0	0	0	1
63	0	1	0	1	0	1	1	0	1	0

TABLE 4.2: Modified Test Set.

	Outputs				Inputs					
No	Co_2	Co_1	C	S	I_3	I_2	I_1	I_0	Ci_2	Ci_1
0	1	0	0	1	1	0	0	0	1	1
X	1	1	0	0	1	1	1	0	1	0
1	0	0	1	0	0	0	0	0	1	1
2	0	0	0	1	1	0	0	0	0	0
3	1	0	0	0	1	0	0	1	0	0
4	1	0	0	1	1	0	0	1	1	0
5	0	1	1	0	1	0	1	1	1	0
6	0	1	0	1	1	0	1	0	0	1
7	1	0	0	0	0	0	1	0	0	1
8	1	0	0	1	0	0	1	1	1	0
9	0	0	1	0	0	0	1	0	1	0
10	0	0	0	1	0	0	1	0	0	0
11	0	1	0	0	1	0	1	0	0	0
12	1	0	0	1	1	0	0	1	0	1
13	0	0	1	0	1	0	0	0	1	0
14	0	0	0	1	0	0	0	1	0	0
15	1	0	0	0	0	1	0	1	0	0
16	0	1	0	1	1	1	0	0	1	0
17	1	0	0	0	1	0	0	0	0	1
18	0	0	0	1	0	0	0	0	1	0
19	1	1	0	0	1	1	1	1	0	0
20	1	1	0	1	1	1	1	0	1	1
21	0	1	1	0	1	0	1	0	1	1
22	1	0	0	1	0	0	1	1	0	1
23	0	1	1	0	0	1	1	1	1	0
24	1	1	0	1	1	1	1	1	0	1
25	1	1	1	0	1	1	1	1	1	1
26	1	1	0	1	1	1	0	1	1	1
27	0	1	1	0	1	1	0	0	1	1
28	1	0	0	1	0	1	0	1	0	1
29	0	1	1	0	1	1	0	1	1	0
30	0	1	0	1	1	1	0	0	0	1
31	1	1	0	0	1	1	1	0	0	1
32	1	1	0	1	1	0	1	1	1	1
33	1	0	1	0	0	0	1	1	1	1
34	0	1	0	1	1	0	1	0	1	0
35	1	1	0	0	1	0	1	1	0	1
36	1	0	1	0	1	0	0	1	1	1
X	1	0	0	0	0	0	0	1	1	0
X	0	0	1	0	0	0	0	0	1	0
37	0	1	0	1	1	0	1	1	0	0
38	1	0	0	0	0	0	0	1	0	1
39	1	0	0	1	0	1	0	1	1	0
40	0	0	1	0	0	1	0	0	1	0
41	0	1	0	1	1	1	1	0	0	0
X	0	0	1	0	0	0	0	0	0	1
42	0	1	0	0	1	1	0	0	0	0
43	0	0	0	1	0	0	0	0	0	1
44	1	0	0	0	0	0	1	1	0	0
45	1	1	0	1	1	1	1	1	1	0
46	1	0	1	0	0	1	0	1	1	1
X	1	1	0	0	0	1	1	1	1	0
47	1	0	0	1	0	0	0	1	1	1
48	0	0	1	0	0	0	0	1	1	0
49	0	1	0	1	1	1	0	1	0	0
50	1	1	0	0	1	1	0	1	0	1
51	1	1	0	1	0	1	1	1	1	1
52	0	1	1	0	0	1	1	0	1	1
X	0	0	0	1	0	0	1	0	0	1
X	0	0	1	0	0	0	1	0	0	0
53	0	1	0	1	0	1	1	1	0	0
54	1	1	0	0	0	1	1	1	0	1
X	1	0	0	1	0	1	0	0	1	1
55	0	0	1	0	0	0	0	0	1	0
56	0	0	0	0	0	0	0	0	0	0
57	0	0	0	1	0	1	0	0	0	0
X	0	1	0	0	0	1	1	0	0	0
58	1	1	1	0	1	1	1	1	0	1
59	1	0	0	1	0	0	1	0	1	1
60	0	1	1	0	1	1	1	0	1	0
61	0	1	0	1	0	1	1	0	0	1
62	1	0	0	0	0	1	0	0	0	1
63	0	1	0	1	0	1	1	0	1	0
X	1	1	1	0	1	1	1	1	0	1

The TPG consist of two sequence generators, one for the vertical inputs (I_3-I_0) and the other for the carry-in inputs of the first cell (C_{i2} C_{i1}). The vertical input pattern is repeated every 74 cells while the needed lateral input for intermediate cells is the lateral output of their predecessors.

There are three ways of detecting a faulty cell. First, the fault-free output of the 74 test vector sequence is stored and the ORA compares the output of every 74 with it to detect any fault. The second approach of detecting faults is by using a 74-word comparator to compare the outputs of every 74 cells with the outputs of the 74 successor/predecessor cells and a fault is detected in any case of inequality. Finally, since the output of every to consecutive cells is different, only 2-word comparator can be used to compare the output of every two consecutive cells and a fault is detected in case of any equality. The last ORA option is used because of its lower hardware cost however it is not efficient as the other two approaches.

□

CHAPTER 5

RESULTS AND CONCLUSION

In the age of public electronic connectivity, as computer systems and their inter-networking grow in complexity, the dependence on secure data storage and transfer is becoming increasingly critical. The danger of hackers, electronic fraud, and eavesdropping has become a serious threat to reliable data communication and storage. This has led to the need for protecting and authenticating access to data and other digital information. Military applications, business and financial transactions, and multimedia communications, are examples that use authentication and data protection algorithms [53].

Public-key cryptosystems are popular because they do not need complex key distribution mechanisms and are mainly based on mathematical functions. The RSA [43] and Elgamal [48] encryption algorithms are examples of public-key crypto-algorithms which are based on modulo operations. The speed of a cryptosystem is an important performance measure. It is a direct function of the algorithm complexity, and the technology used to implement it. Efficient modular multipliers are essential for the design of high-speed crypto-processors [53].

In this work, two types of modulo multipliers were modeled and evaluated. The first is an asynchronous modulo multiplier which is based on a self-timed adder design where the average delay for a k-bit adder is $O(log_2(k))$. The second modulo multiplier is a complete implementation of Montgomery modulo multiplier utilizing a four-to-two compressor architecture that has a fixed addition delay regardless of the size of operands.

Based on the developed VHDL models, the area delay cost of the two multiplier designs were compared and the results are shown in TABLE 5.1 ignoring the time cost of the pre/post operations of Montgomery modulo multiplier.

Figures 5.1, 5.2, 5.3, 5.4 and 5.5 show the area delay cost comparison for α values of *½*, *1*, *1½*, *2*, and *2½* respectively. The value of α has a great impact on the overall area delay cost. For example, where α equals *½* Figure 5.1 shows that the asynchronous multiplier has a lower cost compared to the Montgomery multiplier even for large values of k. On the other hand, as α increases, the Montgomery multiplier exhibits better cost as the size of operands increases. Figure 5.2 shows that for $\alpha = 1$, the asynchronous implementation has a lower cost for operand sizes less than *270*-bits. For α value of*1½*, however, the asynchronous implementation has lower cost only for operand sizes less than *85*-bits, *50*-bit as α increases to *2*, and *32*-bits as α increases to *2½*, as illustrated in the figures 5.3, 5.4, and 5.5 respectively.

This shows that the asynchronous modulo multiplier design will be better suited for **R**esidue **N**umber **S**ystems (RNS) even for large α values. However, for elliptic curve crypto-systems with key sizes less than 270 bits, an efficient asynchronous adder with $\alpha \leq 1$ must be used to give cost results better than the Montgomery multiplier.

Promising high-speed low-cost transistor level implementation of self timed adders [40] need to be developed and further investigated.

TABLE 5.1: AT Cost Comparison for the Multiplier Designs.

Design / Cost	Asynchronous	Montgomery
Number of iterations	*¾ k*	*k/2+1*
Average time per iteration (v)	*(α $\log_2 k$)*	*10*
Average number of correction iterations	*1*	*1½ ($\log_2(k)$+1)*
Total time (v)	*(α $\log_2 k$)(¾k+1)*	*5 k+ 15 $\log_2 k$+25*
Hardware cost (λ)	*11 k*	*12 k*
AT cost ($v\ \lambda$)	*33/4 α k^2 $\log_2(k)$ +11 k $\mathrm{Log}_2(k)$*	*$60k^2$+180 k $\log_2(k)$ +300k*

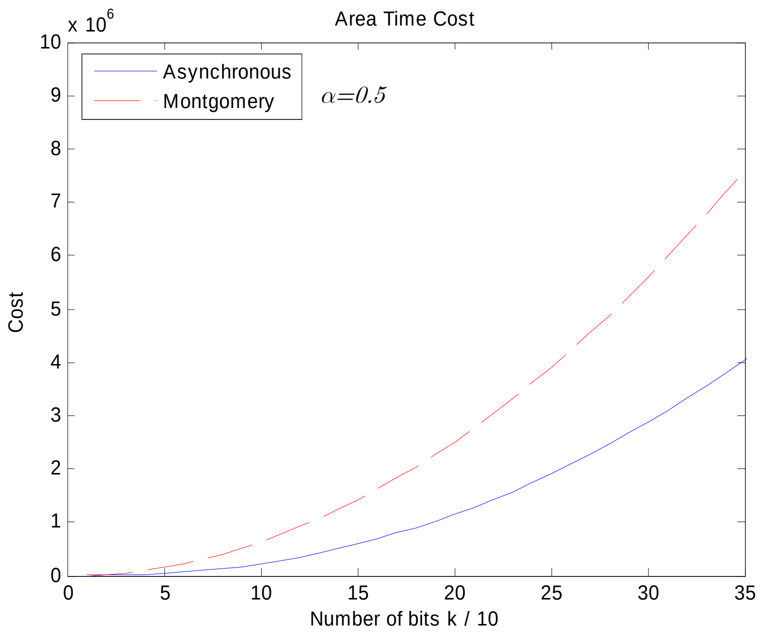

Figure 5.1: Area Delay Cost Comparison for α = ½.

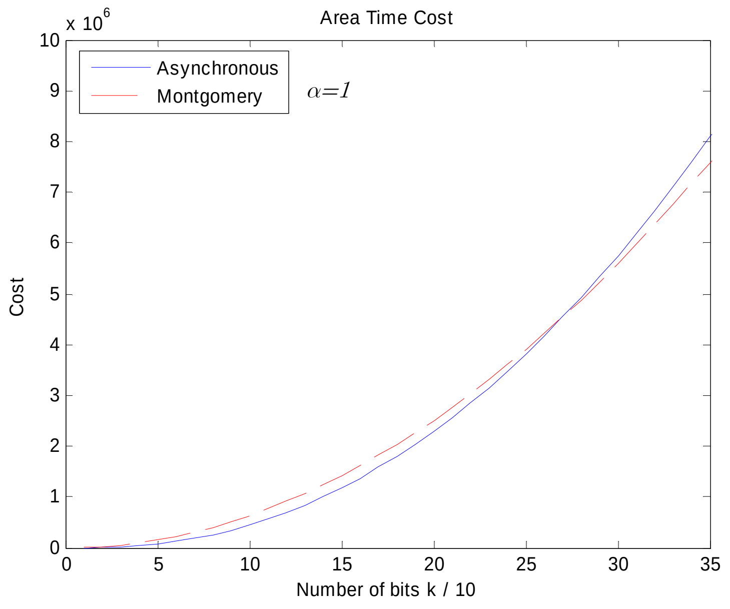

Figure 5.2: Area Delay Cost Comparison for *α* = 1.

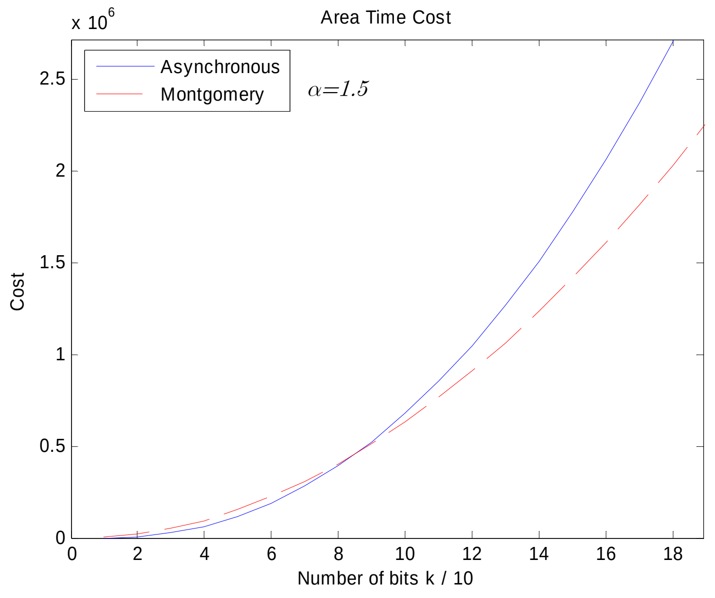

Figure 5.3: Area Delay Cost Comparison for *α* = 1½.

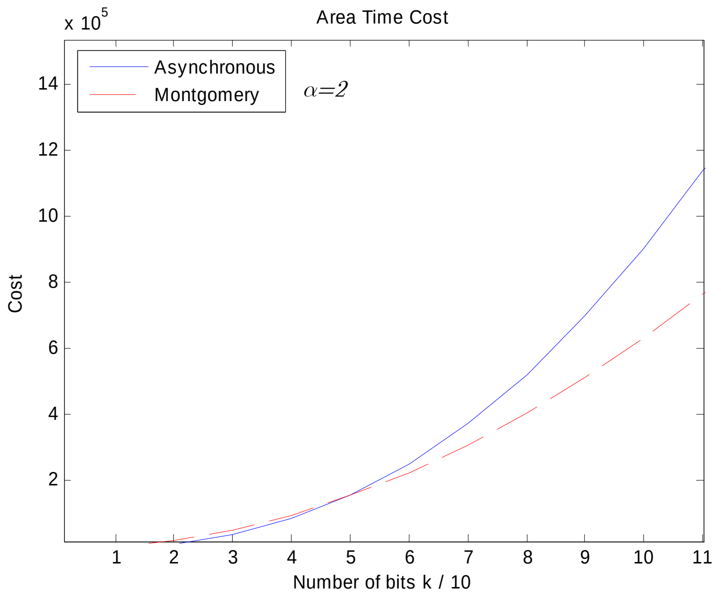

Figure 5.4: Area Delay Cost Comparison for α = 2.

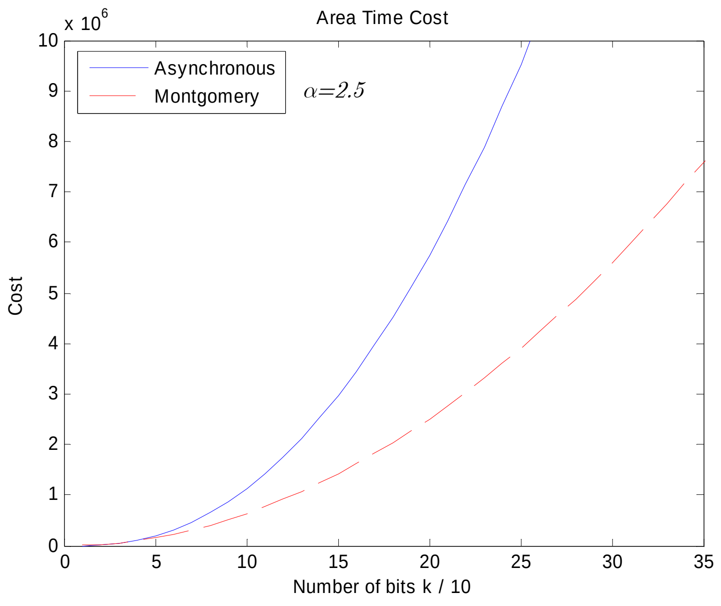

Figure 5.5: Area Delay Cost Comparison for α = 2½.

APPENDICES

APPENDIX A : ANALYSIS ON VITIT'S MULTIPLICATION ALGORITHM

A.1. Assumptions

- Dividend X, and divisor D, are fractions such that $|X| \leq |D|$ and D is a nonzero normalized number.
- q_i: is chosen based on the SRT algorithm, so it satisfies the convergence condition.

A.2. Algorithm's Proof

For q_{i+1} we have three cases when $q_i = 1$:

I. $2r_{i-1} < D$:

Since $D < 1 \Rightarrow 2r_{i-1} < 1$

$r_i = 2r_{i-1} - D < 0$

$\Rightarrow -1/2 < r_i < 0$ (Because $2r_{i-1} < D$, and both D and $2r_{i-1}$ are in $[1/2,1[$)

For convergence condition, we need $|r_{i+1}| \leq |D|$ so pick $q_{i+1} = -1$. To show that the condition is satisfied:

$r_{i+1} = 2r_i \in]-1,0] + D \in [1/2,1[$

=> r_{i+1} >-1/2

$\therefore \; |r_{i+1}| \leq |D|$

II. $2r_{i-1} = D$:

Since q_i =1

=> $r_i = 2r_{i-1} - D \approx 0$

Or $|r_i| \leq 0.000111111... < 0.001$

=> $|2r_i| < 0.01$

So pick $q_{i+1} = 0$

=> $ri + 1 = 2ri \in \{-0.01, +0.01\} << D$

$\therefore \; |r_{i+1}| \leq |D|$

III. $2r_{i-1} > D$:

Since q_i =1

=> $r_i = 2r_{i-1} - D > 0$

So pick $q_{i+1} = 1$

=> $r_{i+1} = 2r_i - D$

Given that $2r_i > 0$ and $D > 0$ and $|r_i| \le |D|$ (Because q_i was chosen using STR).

$\therefore \; |r_{i+1}| \le |D|$

□

APPENDIX B : BINARY NUMBERS RECODING ANALYSIS

To find out the reduction we gain by recoding, the total number of 1s and -1s for all possible combinations needs to be computed. A k-bit binary number needs k+1 bit after it gets recoded. We will look at the number bit by bit from bit 1 to bit k+1.

Bit # C_i	Number of 1s and -1s	
C_1	2^{k-1}	$=2^{k-1}$
C_2	C_1+2^{k-2}	$=C_1+2^k\,(2^{-2})$
C_3	$C_2+2^{k-3}\,(2+1)$	$=C_2+2^k(2^{-2}+2^{-3})$
C_4	$C_3+2^{k-4}\,(2^2+2-1)$	$=C_3+2^k(2^{-2}+2^{-3}-2^{-4})$
C_5	$C_4+2^{k-5}\,(2^3+2^2-2+1)$	$=C_4+2^k(2^{-2}+2^{-3}-2^{-4}+2^{-5})$
C_6 ⋮	$C_5+2^{k-6}\,(2^4+2^3-2^2+2-1)$ ⋮	$=C_5+2^k(2^{-2}+2^{-3}-2^{-4}+2^{-5}-2^{-6})$
C_i ⋮	$C_{i-1}+2^{k-i}\,(2^{i-2}+2^{i-3}-2^{i-4}+2^{i-5}-\ldots \pm 2^{i-i})$ ⋮	$=C_{i-1}+2^k(2^{-2}+2^{-3}-2^{-4}+2^{-5}-2^{-6}+\ldots\pm 2^{-i})$
C_{k+1}	$\mathbf{C_k+2^{k+1-2}\,(2^{k+1-2}+2^{k+1-3}-2^{k+1-4}+2^{k+1-5}\ldots 2^{k+1-k-1})}$	$\mathbf{=C_k+2^k(2^{-2}+2^{-3}-2^{-4}+2^{-5}-2^{-6}+\ldots\pm 2^{-(k+1)})}$

The total summation will be:

$$2^{k-1}+k\times 2^{k-2}+2^{k}\left(\sum_{3}^{k+1}\left(-1\right)^{i-1}\times\left(k+2-i\right)\times 2^{-i}\right)$$

To find out the total number of add operations in radix-4 system for all possible combinations the following analysis was carried out.

Bit #C_i	Number of Additions
C_1	1
C_2	C_1+2
C_3	$C_2+2^2+(2^2+2^0)$
C_4	$C_3+2^3+(2^3+2^1)$
C_5	$C_4+2^4+(2^4+2^2)+(2^4+2^2)$
C_6 ⋮	$C_5+2^5+(2^5+2^3)+(2^5+2^3)$ ⋮
C_i ⋮	$C_{i-1}+\lceil i/2 \rceil \times 2^{i-1}-(\lceil i/2 \rceil -1)\times 2^{i-3}$ ⋮
C_k	$C_{k-1}+\lceil k/2 \rceil \times 2^{i-1}-(\lceil k/2 \rceil -1)\times 2^{k-3}$

The total summation will be:

$$\sum_{1}^{k}\left\lceil \frac{i}{2} \right\rceil \times 2^{i-1} - \left((\left\lceil \frac{i}{2} \right\rceil - 1) \times 2^{i-3} \right)$$

Here is a comparison of the total number of adder operations for all possible combinations. It shows that the gain (reduction of add operations) of using recoded radix-4 over radix-2 is about 0.66 for k=9. However, it is as low as 0.89 for recoded radix-4 over readix-4.

K	Total number of adder operations in case of		
	Radix-2	Radix-4	Readix-4 Recoded
2	4	3	4
4	32	24	28
8	1024	768	796
16	524288	393216	378652
32	6.8719×10^{10}	5.154×10^{10}	4.7722×10^{10}
64	5.903×10^{20}	4.4272×10^{20}	4.0173×10^{20}
128	2.1778×10^{40}	1.6334×10^{40}	1.467×10^{40}
256	1.4821×10^{79}	1.1116×10^{79}	0.9932×10^{79}
512	3.4324×10^{156}	2.5743×10^{156}	2.2942×10^{156}

□

APPENDIX C : COMPRESSOR'S WORST-CASS DELAY

The worst case carry rippling is when one is added to a value that is all ones (i.e., 1+111111). This is similar to adding one to a number that has all nines in decimal (i.e., 1+999999).

As Figure 5.6 shows, every adder bit is made of two full-adders and one half-adder. Therefore, S and C can never both be one. Also, due to the fact that each adder bit has two carry-ins and two carry-outs, the carry is rippled through three bits at a time and this rippling delay is equivalent to one adder bit delay (i.e., two full-adders and one half-adder delays). Therefore, the worst case rippling for a k-bit number is k/3. On the other hand, for the average case delay, it is equivalent to the average delay for the asynchronous adder (see [14] for more details). This is because of the capability of the CARRY register to detect zero value (i.e., rippling is over).

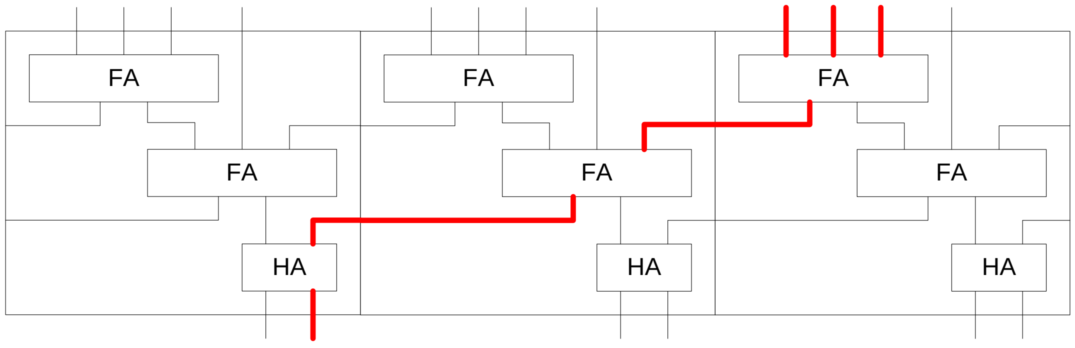

Figure 5.6: Worst Case Rippling.

□

APPENDIX D : APPLYING GENETIC EVOLUTION ALGORITHM TO OBTAIN A GOOD TEST SET.

The algorithm starts by generating an initial population of 64 different test sets. After that, a new 128 generation of test sets are obtained by *crossover* of the test sets of the initial population with a mutation probability of 5%. Finally, a new population of 64 sets is collected by selecting the best 32 sets and randomly picking another 32 sets of the current generation. The algorithm ran for more than 100,000 generations. The goodness of a test set is based on its Hamming distance cost where the least cost set has the best goodness.

Two versions of the algorithm were built. One to find the least costly test set maintaining some vertical output pattern, whereas the second algorithm finds the least costly test set regardless of the pattern of the vertical output.

The **cost function** for both algorithms is based mainly on the Hamming distance between the inputs of consecutive test vectors with a high added high penalty cost if the carry in of a test vector is not the same as the carry out of its predecessor. For the second algorithm, another high penalty is added if the outputs of two consecutive vectors are the same.

To **crossover** to test sets (Set1 and Set2), both sets are split at a random location, the upper half of Set1 and the lower half of Set2 are combined to produce the new set then, the lower half of the new set is scanned and compared to the upper half to find repeated vectors. The repeated vectors in the lower half are replaced with other vectors to make all vectors of a test set different.

The advantage of the first algorithm is the simplicity of its fault detection hardware since it has output pattern that is repeated every four vectors, but unfortunately there are seven vectors with carry in is different that the carry out of their predecessors. To resolve this problem, we can either enforce external inputs during the testing phase – which is very expensive-, or add intermediate vectors to maintain the carry out - carry in flow. Insertion of test vectors to overcome this problem disturbs the output sequence therefore groups of at least four vectors should be inserted instead, which means that at least 28 vectors will be inserted. This is about 30% increase of the number of test vectors.

The test set cost is the summation of the Hamming distance between the inputs of every two consecutive test vectors. If the carry-in of a test vector is different than the carry-out of its predecessor, the cost is increased by 10 to penalize this undesirable vector. The same action is taken in the output of a test vector is identical to the output of its predecessor.

□

REFERENCES

[1] A. D. Friedman, “A functional approach to efficient fault detection in iterative logic arrays”, *Computers, IEEE Transactions on*, Volume 43, Pages1365-1375, December 1994.

[2] Alaaeldin Amin and Feras Maadi, “Double-rail encoded self-timed adder with matched delays”, proceedings of the 10th *IEEE International Conference on Electronics, Circuits and Systems*, December 2003, (ICECS-2003).

[3] Arjen K. Lenstra, “Computational Methods in Public Key Cryptology”, *IMS Lecture Notes Series*, Coding Theory and Cryptology, Pages 175-238, 2002.

[4] Behrooz Parhami, “COMPUTER ARITHMETIC ALGORITHMS AND HARDWARE DESIGN”, Oxford, Oxford University Press, 2000.

[5] C. D. Walter, “Still faster modular multiplication”, *Electronic Letter*, Volume 31, Pages 263-264, February 1995.

[6] C. K. Koc, T. Acar and B. S. KaliskiJr, “Analyzing and comparing montgomery multiplication algorithms”, *IEEE Micro Chip, Systems, Software and Applications*, Pages 26-33, June 1996.

[7] C.-C. Yang, T.-S. Chang and C.-W. Jen, “A new RSA cryptosystem hardware design based on montgomery's algorithm”, *IEEE Transactions, Circuits Systems II*, Volume 45, Pages 908-913, July 1998.

[8] C.-W. Lu, et al, “Designing self-testable cellular arrays”, *Proceedings, Computer Design: VLSI in Computers and Processors, IEEE International Conference on*, Pages 110-113, 14-16 October 1991.

[9] Chauchin Su, et al, C.R. “A BIST methodology for iterative logic arrays”, *Circuits and Systems, ISCAS Proceedings, IEEE International Symposium on*, Volume 1, Pages 411-414, May 1992.

[10] D. Gizopoulos, D. Nikolos and A. Paschalis, “Testing combinational iterative logic arrays for realistic faults”, *Proceedings, VLSI Test Symposium, 13th IEEE*, Pages 35-40, April-May 1995.

[11] D. Gizopoulos, M. Psarakis and A. Paschalis, “Robust sequential fault testing of iterative logic arrays”, *VLSI Test Symposium, 15th IEEE*, Pages 238-244, April-May 1997.

[12] D. J. Kinnement, “An evaluation of asynchronous addition”, *IEEE Transactions VLSI Systems*, Volume 4, Pages 137-140, March 1996.

[13] G. Hachez and J. Quisquater, “Montgomery exponentiation with no final subtraction: Improved results”, In C. K. Koc and Paar [cKKP00], Pages 293-301.

[14] G. W. Reitwiesner, “The Determination of Carry Propagation Length of Binary Addition”, *IRE Transactions on Electronic Computers*, Pages 35-38, 1960.

[15] H. J. Tiersma, “Enhancing the security of El Gamal's signature scheme”, *IEE Proceedings, Computers and Digital Techniques*, Volume 144 No. 1, Pages 47-48, January 1997.

[16] H. Orup and P. Kornerup, “A high-radix hardware algorithm for calculating the exponential M/sup E/ modulo N”, *Proceedings, IEEE 10th symposiums Computer Arithmetic*, Pages 51-56, June 1991.

[17] H. Orup, “Simplifying quotient determination in high-radix modular multiplication”, *Proceedings, 12th symposiums on Computer Arithmetic*, Pages 193-199, July 1995.

[18] H. M. Sun and T. Hwang, “An efficient probabilistic public-key block encryption and signature scheme based on El-Gamal's scheme”, *IEEE International Carnahan Conference on Security Technology*, 1991. Proceedings. 25th Annual, Pages 145 - 148, October 1991.

[19] I. E. Sutherland, “Micropipelines”, *Communications of the ACM*, Volume 32 No. 6, Pages 720-738, June 1989.

[20] J. H. Kim and H. Sung, “An enhanced one-step C-testable design of two-dimensional iterative logic arrays”, *Proceedings, Wafer Scale Integration, [4th] International Conference on*, Pages 331-340, January 1992.

[21] J. He and T. Kiesler, “Enhancing the security of El Gamal's signature scheme”, *IEE Proceedings, Computers and Digital Techniques*, Volume 141 No. 4, Pages 249-252, July 1994.

[22] K. Eshraghian, “Efficient design of gallium arsenide Muller-C element”, *IEEE Electronics Letters*, Volume 33 No 9, Pages 757-759, April 1997.

[23] K. Y. Yun, “Automatic synthesis of extended burst-mode circuits using generalized C-elements”, *IEEE Design Automation Conference, with EURO-VHDL and*

Exhibition, Proceedings EURO-DAC '96, European, Pages 290-295, September 1996.

[24] M. A. Gharaybeh, M. L. Busnell and V. D. Agra-wal, “Classification and Test Generation for Path Delay Faults Using Single Stuck-Fault Tests”, *Proceedings, IEEE ITC*, Pages 139-148, 1995.

[25] M. Abramovici, M. A. Breuer, and A. D. Friedman, “DIGITAL SYSTEMS TESTING AND TESTABLE DESIGN”, Computer Science Press, 1990.

[26] M. D. Ercegovac and T. Lang, “DIVISION AND SQUARE ROOT: DIGIT-RECURRENCE ALGORITHMS AND IMPLEMENTATIONS”, Boston: Kluwer Academic, 1994.

[27] M. Gala, et al, “Built-in self test for C-testable ILA's”, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Volume 14, Pages 1388-1398, November 1995.

[28] M. Jamoussi, B. Kaminska, “A functional-level testability evaluation using a new M-testability approach”, *Circuits and Systems, ISCAS, IEEE International Symposium on*, Volume 3, Pages 1611-1614, May 1993.

[29] M. Shams, et al, “A comparison of CMOS implementations of an asynchronous circuits primitive: the C-element”, *IEEE International Symposium on Low Power Electronics and Design*, Pages 93-96, August 1996.

[30] M. Shams, et al, “Modeling and comparing CMOS implementations of the C-element”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 6 No. 4, Pages 563-567, December 1998.

[31] M. Shams, et al, "Optimizing CMOS implementations of the C-element", *IEEE International Conference Proceedings on Computer Design: VLSI in Computers and Processors*, Pages 700-705, October 1997.

[32] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography", *Proceedings, 11th Symposium on Computer Arithmetic*, Pages 252-259, 1993.

[33] N. Takagi and S. Yajima, "Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem", *IEEE Transactions on Computers*, Volume 41 Issue 7, Pages 887 -891, July 1992.

[34] N. Takagi, "A modular multiplication algorithm with triangle additions", *Proceedings, 11th Symposium on Computer Arithmetic*, Pages 272-276, June-July 1993.

[35] N. Takagi, "A radix-4 modular multiplication hardware algorithm efficient for iterative modular multiplications", *Proceedings, 10th IEEE Symposium on Computer Arithmetic*, Pages 35 -42, June 1991.

[36] N. Takagi, "A radix-4 modular multiplication hardware algorithm for modular exponentiation", *IEEE Transactions on Computers*, Volume 41 No. 8, Pages 949-956, August 1992.

[37] Norman Scott, "COMPUTER NUMBER SYSTEM & ARITHMETIC", New Jersey, Prentice-Hall, 1985.

[38] O. Nibouche, A. Bouridane and M. Nibouche, "Architectures for Montgomery's multiplication", *Computers and Digital Techniques, IEE Proceedings*, Pages 361-368, November 2003.

[39] P. L. Montgomery, “Modular Multiplication without Trial Division”, *Mathematics of Computation*, Volume 44, No. 170, 1985, Pages 519-521.

[40] Private Communication- Dr. Amin.

[41] Psarakis, et al, “Robustly testable array multipliers under realistic sequential cell fault model”, *Proceedings, VLSI Test Symposium, 16th IEEE*, 152-157, 26-30 April 1998

[42] Psarakis, et al, “Sequential fault modeling and test pattern generation for CMOS iterative logic arrays”, *Computers, IEEE Transactions on*, Volume 49, Pages 1083-1099, October 2000.

[43] R. L. Rivest, A. Shamir and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems”, *Communications of the ACM*, Volume 21, No. 2, Pages 120-126, February 1978.

[44] R. L. Rivest, A. Shamir and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems”, *Communications of the ACM*, Volume 26, No. 1, Pages 96-99, January 1983.

[45] R. M. Davies and J. V. Woods, “Timing verification for asynchronous design”, *IEEE Design Automation Conference, with EURO-VHDL and Exhibition, Proceedings EURO-DAC '96, European*, Pages 78 -83, September 1996.

[46] S. E.Eldridge and C. D.Walter, “Hardware implementation of Montgomery's modular multiplication algorithm”, *IEEE Trans. Computer*, Volume 42, Pages 693-699, June 1993.

[47] S. Sadiq and Y. Habib, “VLSI PHYSICAL DESIGN AUTOMATION: THEORY AND PRACTICE”, McGraw-Hill Book Co., Europe, December 1994.

[48] T. Elgamal, “A public key cryptosystem and a signature scheme based on discrete logarithms”, *IEEE Transactions on Information Theory*, Volume 31 No. 4, Pages 469-472, July 1985.

[49] V. Bunimov, et al, “A Complexity-Effective Version of Montgomery's Algorithm”, Workshop on Complexity Effective Designs (WCED02), May 2002. www.ece.rochester.edu/~albonesi/wced02/papers/bunimov.pdf

[50] V. Hert and A. J. van de Goor, “Test generation for C-testable one-dimensional CMOS ILA's without observable vertical outputs”, *Proceedings, Design Automation, [3rd] European Conference on*, Pages 421-427, March 1992

[51] V. Kantabutra, “A new algorithm for division in hardware”, *IEEE International Conference on Computer Design*, Pages 551-556, October 1996.

[52] W. Diffie and M. Hellman, “New directions in cryptography”, *IEEE Transactions on Information Theory*, Volume 22 No. 6, Pages 644-654, November 1976.

[53] W. Stallings, “NETWORK AND INTERNETWORK SECURITY”, Prentice-Hall, NJ: IEEE Press, 1995

[54] Young Sae Kim; Woo Seok Kang and Jun Rim Choi, “Asynchronous implementation of 1024-bit modular processor for RSA cryptosystem”, *ASICs, 2000. AP-ASIC 2000. Proceedings of the Second IEEE Asia Pacific Conference on*, Pages 187-190, August 2000.

VITAE

- Muhammad Yahya Imam Mahmoud.
- Born in Jeddah, Saudi Arabia.
- Married and father of three daughters.
- Received the B.Sc. degree in Computer Engineering from KFUPM, Saudi Arabia in May 2000.
- Completed the M.Sc. degree requirement at KFUPM, Saudi Arabia in May 2004