Fast finite field multipliers for public key cryptosystems

Satzoda Ravi Kumar

2007

a52998 7

Fast Finite Field Multipliers for Public Key Cryptosystems

Satzoda Ravi Kumar

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2007

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my heartfelt gratitude to my supervisor, **Associate Professor Chang Chip-Hong** for his expert guidance, constant support and unceasing encouragement that were vital for research and development of this thesis. The numerable discussions that we have had regarding various related research problems have proved to be invaluable in making the whole experience very fulfilling. His constructive criticism and high expectations have driven me to bring about quality and credibility to the work. I am greatly indebted to him for also providing a conducive research atmosphere and all the latest facilities at Center for High Performance Embedded Systems (CHiPES).

My deep gratitude to **Professor Srikanthan Thambipillai**, Director, CHiPES who gave me an opportunity to work in the research environment during the course of the Masters' project.

Special thanks to my wife, **Suchitra**, who has been with me throughout the project. She has given enormous support, confidence and encouragement at every step. I thank my parents who have been a constant source of confidence and encouragement. It would be incomplete without thanking **Swamiji**, who has been a tremendous source of inspiration, without whom it would not have been possible for me to complete this thesis.

Thanks also to my fellow researchers and colleagues, who have made my stay at CHiPES memorable and enjoyable. I extend my thanks to the lab technicians **Ms. Boh-Nah Kiat Joo** and **Mr. Jeremiah Chua Ngee Tat**, who have supported me in one way or the other towards completion of this thesis.

I would also like to thank my friends Phalgun Maddali and Aditya Naidu who have helped me in innumerable ways during the course of this research work.

Finally, I would like to acknowledge School of Electrical and Electronic Engineering, Nanyang Technological University for providing me with this opportunity for pursuing this Masters degree.

Abstract

The security strength of Public Key Cryptosystems (PKCs) is attributed to the complex computations that are employed in the encryption and decryption algorithms. These algorithms are executed by cryptoprocessors which are used as embedded co-processors in devices like smartcards, smart cameras etc. Speed of operation, circuit area and security strength variability are vital design considerations in such applications. These criteria call for efficient hardware implementation of the underlying arithmetic operations of the algorithms. Multiplication in finite fields, widely known as finite field multiplication or modular multiplication, forms the core computational engine of all algorithms involved in PKCs. The research work in this thesis aims at developing efficient architectures for finite field multiplications in the prime field $GF(N)$ and the extended binary field $GF(2^m)$. The research focuses particularly on high speed hardware implementations of finite field multiplication algorithms giving due consideration to area utilization. Three different kinds of algorithms, called LSB-first and MSB-first, and Montgomery modular multiplications, are studied in this thesis.

A modified Montgomery modular multiplication is proposed for Rivest-Shamir-Adleman (RSA) cryptosystems. Two fast array-based architectures, a one-dimensional serial-in parallel-out array and a two-dimensional pipelined parallel array, are derived from the proposed algorithmic modification. These architectures are evaluated for speed and area utilization against recently reported implementations using both FPGA and semi-custom design flows.

Though the proposed Montgomery modular multipliers operate at nearly twice the clock rate as the existing architectures, they do not address scalability issues. Moreover, they are designed to operate in $GF(N)$, which is only suitable for RSA cryptosystems. We have further proposed a novel pipelined and scalable unified Montgomery modular multiplication processing unit that can operate in $GF(N)$ or $GF(2^m)$ for both RSA and Elliptic Curve Cryptosystems (ECCs). The proposed architectures eliminate the redundant logic in existing dual field architectures. Implementation using FPGA design flow shows a twofold speedup in $GF(2^m)$ operation with no timing degradation in $GF(N)$ operation. The operation of the pipeline is derived from reservation tables and dependency graphs. A comprehensive analysis of the pipelined architecture is performed to study the relationship between latency cycles, total computational time and number of processing units in the pipeline.

Novel serial-in parallel-out LSB-first and MSB-first modular multipliers are also proposed. The proposed multipliers can operate in arbitrary field $GF(2^m)$ with the field order $m \leq M$ where M is the maximum permissible field order. In contrast to existing parallel generalized LSB-first/MSB-first multiplier, the dependency of the critical path on the size of the multiplier is eliminated. To the best of the author's knowledge, these are the first reported programmable serial-in parallel-out LSB-first and MSB-first architectures for any field order m reported in the literature. The proposed architectures are synthesized using the TSMC 0.18μm standard cell libraries. The prelayout simulation results show an area savings of about 96% and a speedup of about 17 times compared to the existing parallel generalized multipliers.

Chapter 1

Introduction

1.1 Background

In the present era of network communications, information security is a major concern. Applications like automatic teller machines, internet transactions, satellite communication etc., involve large amounts of sensitive data which requires encryption to prevent eavesdropping. Several cryptosystems have already been developed and the strengths of most of these systems have been tested with extensive cryptanalysis [50]. All these cryptosystems have been implemented on software and hardware and a common rule that goes with all cryptosystems is that hardware implementations are faster and less vulnerable to willful tampering than their software counterparts [50].

Cryptosystems can be broadly classified as symmetric key and public key cryptosystems [50]. Symmetric key cryptography requires both parties - sender and receiver, to have the same key that encrypts and decrypts the message. Moreover, the encryption/decryption key should be kept private. In contrast to symmetric key cryptography, public key cryptography involves a 'public' encryption key and a 'private' decryption key. Thus a public database holds one's encryption key and based on this encryption key, a legitimate user can derive the decryption key which is private to that user.

In contrast to the symmetric key cryptosystems where key management plays a vital role in maintaining the secrecy of information, public key cryptosystems are less vulnerable to eavesdropping because the encryption and decryption keys are different and there is no transmission of private keys over secure channels. The transmission channel also needs to be highly secure in a symmetric key cryptosystem to prevent an eavesdropper from accessing the key which is also being transmitted [50]. Moreover, public key cryptosystems can be used to encrypt the secret keys for symmetric key information exchange sessions over insecure channels. Public key cryptosystems have superceded symmetric key cryptosystems for sensitive information protection, limited only by their greater computational complexity.

The strength of public key cryptosystems (PKCs) like Rivest Shamir Adleman (RSA) algorithm, Elliptic Curve Cryptography (ECC) algorithm, Diffie-Hellman key exchange etc., lies in the mathematical irreversibility (in computational complexity sense) of their encryption in the absence of the key. For example, the strength of RSA is dependent on the intractable NP-complete problem of factoring a product of two large relatively-prime numbers. Similarly ECC and Diffie-Hellman key exchange are logarithm-based operations. The security strength of RSA has been proven for large prime number domain which are of the order 2^{10} bits [50]. For the same security strength, ECC operates at shorter key lengths, making it preferable to RSA [3].

1.2 Motivation

The two commonly used PKC algorithms - RSA and ECC are formulated using abstract algebra. It involves algebra in two kinds of finite fields (also known as Galois Fields) - prime field $GF(N)$ and extended binary field $GF(2^m)$. RSA cryptosystem generally operates in $GF(N)$ whereas ECC operates in both $GF(N)$ and $GF(2^m)$. The main problem that is common to both cryptosystems is the complex mathematical operations. The algorithms for both cryptosystems are founded on the compute intensive finite field multiplications. These modulo computations are the performance bottleneck not only in software but also in hardware implementations.

Thus finite field arithmetic, more specifically finite field multiplication, for both ECC and RSA is in the spotlight of the network security research community. Hardware implementation of the operations in finite field arithmetic scores over the software implementation mainly because of its high speed. Besides, the area and power efficiency in application-specific integrated circuit (ASIC) make it a prerogative choice for secured information exchange via RFIDs and smartcards. It is evident from literature [5, 6, 10, 11, 21, 22, 32, 34, 35, 35, 36, 39], [8, 13–20, 28, 40, 51, 52], [23], [35] that finite field multiplication has been a dominant research area. Due to several new developments from a detailed literature survey, and in the context of hardware of public key cryptography, there is plenty of scope, new challenges and new perspectives for the design and development of hardware efficient and improved performance finite field multipliers. There is both need and potential to exploit the inherent properties of the modular multiplication algorithms for efficient hardware architectures. Larger benefits could be derived from architectural translation if the optimization is considered at the algorithmic level itself.

1.3 Objectives

Considering the ever increasing need for secure cryptoprocessors, this thesis aims to develop efficient hardware architectures for finite field multiplication which is instrumental to the advancement of Elliptic Curve Cryptosystems and RSA cryptosystems. This thesis addresses the need for efficient cryptoprocessors by proposing novel algorithms for finite field multiplication and translating them into high performance architectures. Existing finite field multiplication algorithms will be studied in detail and the hardware implementation aspects of these algorithms will be explored. Scalability and parallelism are important factors that improve the performance of hardware implementations. These issues will also be addressed at the algorithmic level. In addition to algorithm development and architectural translation, it is also aimed to evaluate the architectures comprehensively for VLSI performance metrics. Standard hardware development platforms - ASIC and FPGA, will be used to evaluate the performance of the proposed architectures against the existing architectures. Due to the large number of circuits involved and the man-hour required for the backend process of ASIC design, all designs are coded in structural VHDL at Register Transfer Level (RTL) and the VLSI metrics are reported based on prelayout synthesis results using the TSMC 0.18μm CMOS standard cell libraries.

1.4 Thesis Contributions

This research work was embarked on with a very clear focus and emphasis. The development of novel algorithms and their translation into efficient hardware architectures were the prime objectives, making the following important contributions possible during the course of research.

Firstly, a detailed survey of fast and efficient hardware implementations of systolic and semisystolic finite field multipliers in $GF(2^m)$ with two algorithmic schemes - LSB-first and MSB-first, is conducted. These algorithms have been mapped to seven variants of recently proposed array-type finite-field multiplier implementations with different input-output configurations. The relative VLSI performance merits of these ASIC prototypes are evaluated and compared under uniform constraints and in properly defined simulation runs on a Synopsys environment using the TSMC 0.18μm CMOS standard cell library. The results of the simulation provide an insight into the behavior of various configurations of array-type finite-field multiplier so that a system architect can use them to determine the most appropriate finite field multiplier topology for the required design features.

A modified Montgomery modular multiplication (MMM) algorithm is proposed in this research. It can be translated into high speed carry-save systolic architectures due to the elimination of data-dependent control signal that dominates the critical path. Two new systolic Montgomery multipliers are designed based on the proposed algorithm - two-dimensional pipelined parallel systolic architecture and serial-in parallel-out one-dimensional area-efficient implementation. Both architectures are evaluated against recently reported Montgomery multipliers in terms of resource utilization of FPGA and clock frequency. The one-dimensional serial architecture is further assessed for ASIC performance metrics in TSMC 0.18 μm against the fastest reported one-dimensional implementation. The proposed architectures show improved performance and low area-time product as compared to the existing architectures.

The next major contribution is a novel fast kernel comprising a new processing element and dual field adder for a scalable and pipelined unified MMM in $GF(N)$ and $GF(2^m)$. The proposed architecture has successfully reduced the slack of the MMM in $GF(2^m)$ without jeopardizing the timing of its operation in $GF(N)$. Acceleration of multiplication in $GF(2^m)$ for all ranges of modulus and in $GF(N)$ for higher precision modulus is made possible through a new kernel of dual field adder and processing unit. The proposed dual field adder has been optimized to operate in an existing architecture that has been retimed to overcome the conflicts for speeding up the pipelined architecture. The total latency has been analytically expressed in terms of the input wordlength, modulus precision and number of pipeline stages. The processing unit has been implemented on FPGA.

The experimental results show evidence of reduction in total computation time and faster operating clock rates over existing dual field processing unit.

Lastly, novel generalized LSB-first and MSB-first modular multipliers in $GF(2^m)$ have been proposed. The proposed multipliers can operate in any arbitrary field with field order m up to a maximum field order, M. The proposed 128-bit multipliers speed up the maximum clock frequency of the existing generalized LSB-first/MSB-first multipliers by nearly 16.7 times. By using our proposed generalized field multiplier for the point multiplication operation in an ECC with parametric $m = 113$, the total computation time is reduced by ten fold and the area cost is reduced by 96% as compared to that implemented with existing parametric finite field multiplier. This savings is attributed to serializing of the whole multiplication in addition to making it programmable.

The above contributions have resulted in several international journal and conference papers which are listed in the List of Author's Publications towards the end of the thesis.

1.5 Organization of Thesis

The thesis is organized as follows.

Chapter 1 presents the background and motivation for the research work. The major contributions of the thesis are also highlighted.

In Chapter 2, the foundational mathematical structures on finite fields, $GF(N)$ and $GF(2^m)$ in particular, are presented as a preamble to cryptosystems. Two major public key cryptosystems, RSA and elliptic curve cryptography are introduced followed by the principles and protocols of their encryption and decryption. The fundamentals of elliptic curves and their applicability to public key cryptosystem are further elaborated. Finite field multiplication algorithms in $GF(N)$ and $GF(2^m)$, being the workhorse of these cryptosystems, are surveyed. Three algorithms, Montgomery modular multiplication, LSB-first and MSB-first multiplication algorithms are described. In addition, hardware implementations of recently reported LSB-first and MSB-first multipliers are discussed in greater detail. The VLSI metrics of these architectures using TSMC $0.18\mu m$ standard cell libraries are also evaluated.

Chapter 3 presents two new systolic architectures for Montgomery modular multiplication in $GF(N)$ for RSA cryptosystems. First, some existing implementations of Montgomery modular multiplication are discussed at algorithmic level. The issues associated with the architectural translation of these algorithms are discussed. A modified multiplexer based method is then proposed which leads to two multiplier designs - one-dimensional serial multiplier and two-dimensional parallel multiplier. These multipliers are then evaluated against some of the recent most Montgomery multipliers on FPGA and ASIC platforms.

In Chapter 4, a new scalable and pipelined unified kernel for Montgomery multiplications in $GF(N)$ and $GF(2^m)$ is proposed. MMM algorithms in $GF(N)$ and $GF(2^m)$ are first revisited to study the similarities between them. After studying some existing unified architectures, the impediment to the speed of even one of the fastest available unified architecture is identified. A kernel that employs a new processing unit is then devised to reduce the total time of computation of the Montgomery product. The logic of the proposed processing unit is described and new dependency graphs are derived from a detailed description of the dataflow. They are used to explain the design of a new pipelined architecture using the proposed processing unit. For different use scenarios, the reservation tables corresponding to the new pipeline are illustrated. The latency and pipeline stalls are derived analytically. Detailed qualitative and quantitative analysis and evaluation of latency, critical path delay, total computation time and area using FPGA platform are conducted.

Chapter 5 presents new serial-in parallel-out generalized LSB-first and MSB-first modular multipliers in $GF(2^m)$, for any field order m. Some preliminaries of fixed order LSB-first/MSB-first algorithms are provided and the challenges to make them adaptable to varying field orders are discussed. This is followed by a scrutiny of the limitations of existing generalized LSB-first/MSB-first multipliers. The proposed methods based on field selectable switches are then presented. Signal flow graphs are used to illustrate the data flow in the proposed methods. Architectural innovations for efficient implementation using the standard cell libraries are discussed. A comprehensive qualitative evaluation of the proposed multipliers against existing multiplier, in terms of the latency and gate count, is carried out. The synthesis results based on standard cell implementation are also analyzed and discussed.

Important conclusions from the research work are drawn in Chapter 6. It summarizes the key results of this research. Relevant future work is recommended.

Chapter 2

Background and Literature Review

2.1 Introduction

This chapter surveys the RSA and ECC public key cryptosystems (PKCs) and their essential operational blocks. A brief introduction of the algebraic structures, that are necessary to understand the encryption and decryption algorithms in PKCs, is provided. Elliptic curves and the associated arithmetic operations are further discussed because of their relative importance in this research. The finite field multiplication is identified as a core operation of the PKCs. A detailed review and performance evaluation of some existing algorithms and architectures of finite field multiplier are performed. The survey is by no means complete, but it does provide a good insight into the complexity and trade-offs of the problems considered in this thesis.

2.2 Algebraic Fundamentals

Vectors are used extensively to describe generic signals and variables of the algorithms and architectures. The following conventions are adopted to maintain consistency in the notations of vector quantities in this chapter. A k-bit vector $(x_{k-1}x_{k-2}\ldots x_1x_0)$ is represented by italicized capital letter X. The j-th bit of vector X is given by x_j.

2.2.1 Fundamentals of Groups and Finite Fields

A group is an algebraic system defined on a set of elements, G, with a binary operation $\circ$ satisfying the following four axioms [4, 12, 27]:

- closure: $\forall x, y \in G, x \circ y \in G$
- associativity: $x \circ (y \circ z) = (x \circ y) \circ z$
- identity: $\exists e \in G, \forall x \in G : x \circ e = e \circ x = x$ where e is the identity element
- inverse: $\forall x \in G, \exists y \in G : x \circ y = y \circ x = e$.

In addition to the above properties if a group satisfies commutative property, defined as $\forall x, y \in G, x \circ y = y \circ x$, the group is called an Abelian group.

A field is a special type of group. It is an algebraic system defined on a set of elements, F with two binary operations $+$ and $\times$. It satisfies the following axioms [4]:

- $(F, +)$ is an Abelian group
- $(F/\{0\}, \times)$ is an Abelian group, where $\{0\}$ is the identity of addition and the zero of multiplication
- distributivity: $\forall x, y, z \in F : x \times (y + z) = x \times y + x \times z; (x + y) \times z = x \times z + y \times z$

A field with a finite number of elements is called a Galois Field (GF). Operations that are performed in a finite field are called finite field operations. Two types of finite fields are important to understand the core properties and implementation considerations of PKCs. A prime field, denoted $GF(N)$ is a Galois field of N elements, where N is a prime number [37]. An m-th degree extension of the prime field is denoted by $GF(p^m)$. The m-degree extension of a binary field, which is denoted by $GF(2^m)$ [37], is the most important in terms of hardware realization of finite field operations.

Prime Field $GF(N)$ or F_N

A prime field $GF(N)$ is defined on a set of numbers $F = \{0, 1, \dots N-1\}$, where N is prime. All the elements, that are operated on in this field, lie in F. The results of all finite field operations in $GF(N)$ are reduced by a modulus N. Finite field addition and finite field multiplication in $GF(N)$ are defined as follows.

Addition: Integer addition of a and b followed by a reduction operation.

$$c = (a + b) \;\; mod\, N \tag{2.1}$$

Multiplication: Integer multiplication of a and b followed by a reduction operation.

$$c = (a \times b) \;\; mod\, N \tag{2.2}$$

These operations are the conventional integer addition and multiplication methods which involve carry propagation. The reduction operation at the end of the addition and multiplication ensures that the result c belongs to F.

Extended Binary Field $GF(2^m)$ or F_{2^m}

The elements in $GF(2^m)$ for ECC are commonly represented in two different ways [37] but in this thesis, only the polynomial basis representation is used. In this representation, every element in $GF(2^m)$ is expressed as a polynomial, $a(x)$, in an indeterminate x with degree less than m.

$$a(x) = \sum_{i=0}^{m-1} a_i x^i = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0 \tag{2.3}$$

where the coefficients, $a_i \in \{0, 1\}$ of the polynomial $a(x)$, are the elements in $GF(2)$. The finite field addition and multiplication operations in $GF(2^m)$ are defined as shown below.

Addition: Two elements $a(x)$ and $b(x)$ in $GF(2^m)$ are added to produce $c(x)$ using a bitwise XOR operation.

$$\begin{aligned} c(x) &= a(x) + b(x) \\ \sum_{i=0}^{m-1} c_i x^i &= \sum_{i=0}^{m-1} a_i x^i + \sum_{i=0}^{m-1} b_i x^i \\ where \quad & c_i = (a_i + b_i) \ mod\, 2 = a_i \oplus b_i \end{aligned} \tag{2.4}$$

The modulo operation in finite field addition operation in $GF(2^m)$ makes it free from carry propagation.

Multiplication: Multiplication of elements $a(x)$ and $b(x)$ in $GF(2^m)$ is more complicated than addition. It involves an irreducible polynomial of degree m, $f(x) = x^m + \sum_{i=0}^{m-1} f_i x^i, f_i \in \{0, 1\}$, that is used for the reduction of the product $a(x) \times b(x)$ as shown below.

$$\begin{aligned} c(x) &= (a(x) \times b(x)) \ mod\, f(x) \\ \sum_{i=0}^{m-1} c_i x^i &= \left(\sum_{i=0}^{m-1} a_i x^i \times \sum_{i=0}^{m-1} b_i x^i \right) \ mod\, f(x) \end{aligned} \tag{2.5}$$

The product, $c(x)$, is a result of modular multiplication with $f(x)$ as the reduction polynomial. The product, $c(x)$, is also a polynomial of degree less than m.

Note: All finite field operations, either in $GF(N)$ or $GF(2^m)$ involve a modulus or reduction operation. This implies that the linear congruences of all finite field operations are realized by modular operations. Henceforth, modular multiplication and finite field multiplication are used interchangeably in the thesis. These preliminary mathematical structures are vital in understanding some important concepts to be reviewed in the subsequent sections. More number theoretic fundamentals will be introduced as and when they are needed.

2.3 Public Key Cryptosystems

Public key cryptography was first introduced by Whitfield Diffie and Martin Hellman in 1976 [50]. Two different keys, called private key and public key, are used for the encryption and decryption of data in public key cryptosystems (PKCs). The two keys are used in PKC protocol [50] in the following way to generate cipher text.

1. Alice and Bob agree on a public-key cryptosystem.
2. Alice gets Bob's public key from a database.
3. Alice encrypts her message using Bob's public key and sends the resulting cipher text to Bob.
4. Bob then decrypts the cipher text to retrieve Alice's message using his private key.

A public key is available to all users of the PKC. It is used for encryption by a sender to generate the cipher text. The receiver decrypts the cipher text using his own private key. The public and private keys are mathematically related. A public key can have only one corresponding private key and it is computationally intractable to derive the private key from the public key.

Ease of key management and non-repudiation are the two major advantages in PKCs over symmetric key cryptosystems [50]. In the latter, both the sender and the receiver use the same key for encryption and decryption. Sharing a common key for a communication session poses the same threat as the secret data to be protected by the cryptosystem itself over the public channel. In PKC, the encryption keys of users are known to everyone. So any sender can encrypt and send a message to a desired receiver using the receiver's public key. An eavesdropper cannot decrypt it because he does not have the private key needed for decryption. Only a valid receiver can decrypt the message using the private key. Therefore, the more secure PKC is often used to exchange a secret key used for a more efficient symmetric key communication session. A trustworthy governing body is usually engaged to generate and distribute the public keys for information exchange.

In what follows, we will review the algorithms of two important public key cryptosystems.

2.4 RSA Cryptosystem

RSA algorithm is named after Rivest, Shamir and Adleman who proposed it in 1978 [41]. Its security is theoretically guaranteed by the difficulty in factoring large numbers [50]. The private and public keys are a pair of large prime numbers. The mathematical details for generating these keys are described in [50]. The encryption and decryption of a message is done by modular exponentiation of the message. A plain text, m_i is encrypted to produce the cipher text c_i using the following equation.

$$c_i = m_i^e \ mod \ N \tag{2.6}$$

where e is the public key and N is a large prime number. This cipher text is decrypted using the private key, d, by the following equation.

$$m_i = c_i^d \ mod \ N \tag{2.7}$$

The validity of (2.7) to decrypt the cipher text back to give the original message is proved in [50]. Since all numbers involved in the algorithm lie in the set $\{0, 1, \ldots, N-1\}$, RSA operates in the prime field $GF(N)$.

The security of RSA cryptosystem is founded on a NP-hard problem commonly known as the Discrete Logarithmic Problem (DLP) [50]. DLP deals with computing k in $a^k \equiv b$ $mod\, N$ when b is known or finding a when k is known. From (2.6) and (2.7), it can be seen that finding message m_i, when c_i and e are known, is a DLP problem. Hence, to break RSA, DLP is encountered to determine the message m_i given the cipher text c_i. RSA deals with large integers that typically have more than 100 digits. Begin NP-hard, solving DLP is computationally intractable when k and N are large [50].

2.4.1 Modular Exponentiation and Multiplication in RSA

From (2.6) and (2.7), it can be inferred that modular exponentiation is the key computational block in RSA cryptosystem. Modular exponentiation is implemented as repeated modular multiplication [36]. The following algorithm from [36] illustrates the implementation of modular exponentiation using modular multiplication. m, e, c and N in Algorithm

Algorithm 1 Binary modular exponentiation

```
1: Input: $m, e, N$
2: Output : $c = m^e \ mod \ N$
3: $c = m$
4: for $i = k - 2$ downto 0 do
5:     $c = c^2 \ mod \ N$
6:     if $e_i = 1$ then
7:         $c = (c \times m) \ mod \ N$
8:     end if
9: end for
10: Return $c_i$
```

1 represent the message, public key, cipher text and the field order respectively. In this algorithm, c is the output of the modular exponentiation operation. Input m is first assigned to c. In a loop that runs through all the bits of the exponent e, a squaring operation is done on c and the squared result is multiplied to m if and only if the bit in the exponent is equal to '1'. All the multiplications and squaring operations done are reduced by modulus operation to keep the result less than N.

The modular exponentiation algorithm shows that modular multiplication (MM) in `Line 7` is the most important computational block. The numbers involved in cryptographic operations are huge. For example, in RSA cryptosystem, the operand lengths are at least 128 bits to give a reasonable security strength. Hence, standard division and modular multiplication algorithms are not commonly used for cryptographic operations. In the above algorithm, modular multiplication is the key computation step and modulo operation is a costly operation in terms of hardware metrics. Circumventing the division problem in modulo operation by using fast algorithms and their corresponding efficient hardware implementations of modular multiplication (MM) in $GF(N)$ is a widely researched topic in computer arithmetic community. From the available MM algorithms, Montgomery modular multiplication (MMM) in $GF(N)$ proposed by P. L. Montgomery in [32] is the most commonly used algorithm for efficient hardware implementation. It converts trial divisions in the modulo reduction to shifts and adds.

The MMM algorithm is briefly described as follows. Given m bit integers A, B and N, an m bit integer $C = AB2^{-m} \, mod \, N$ is determined using MMM Algorithm 2. The result from the MMM algorithm is $AB2^{-m} \, mod \, N$ and not the actual modular product $AB \, mod \, N$. To obtain the actual modular product, the inputs are converted to Montgomery domain [48] by precomputing the following modular products.

$$\begin{aligned} \overline{A} &= MMM(A, 2^{2m}) &= A2^m \ mod \ N \\ \overline{B} &= MMM(B, 2^{2m}) &= B2^m \ mod \ N \end{aligned} \tag{2.8}$$

When $\overline{A}$ and $\overline{B}$ are used as inputs to the MMM algorithm, it produces $\overline{C}$ in Montgomery domain, which is converted to the correct result in the original field using the following equations.

$$\begin{aligned} \overline{C} &= MMM(\overline{A}, \overline{B}) &= AB2^m \ mod \ N \\ C &= MMM(\overline{C}, 1) &= AB \ mod \ N \end{aligned} \tag{2.9}$$

As shown in Algorithm 2, the modulo operation is determined by simple vector additions and shifts. The additions in `Lines 5 & 6` are integer additions with carry propagation.

A number of implementations of this algorithm are found in the literature [5, 6, 10, 11, 21, 22, 32, 34, 35, 35, 36, 39]. [21] surveys different codings of this algorithm in software. [10,11,34–36,48] provide its hardware realizations using carry-save representation for faster accumulation of partial products.

Algorithm 2 Montgomery Modular Multiplication in $GF(N)$

```
1: Input: $A = \sum_i^{m-1} a_i 2^i, B = \sum_i^{m-1} b_i 2^i, N = \sum_i^{m-1} n_i 2^i$
2: Output : $C = MMM(A, B) = AB2^{-m}(mod\ N)$
3: $S \leftarrow 0$
4: for $i = 0$ to $m - 1$ do
5:     $S \leftarrow S + a_i B$
6:     $S \leftarrow S + s_0 N$
7:     $S \leftarrow S/2$
8: end for
9: if $S \geq N$ then $S \leftarrow S - N$
10: Return $S = AB2^{-m}(mod\ N)$
```

2.5 Elliptic Curve Cryptosystems

Elliptic curves have been studied for many years but their application to the cryptosystems is relatively recent [31, 33]. RSA cryptosystems (discussed in the previous section) involve integers that are 100 to 200 decimal digits long. As a result, the processing time of RSA cryptosystems is long. ECCs, on the other hand, operate on smaller numbers. The security strength of an ECC with keylength of 572 bits is equivalent to an RSA with keylength of 15360 bits [3]. In the recent years, there has been a wave of research in the field of Elliptic curve cryptosystems. A detailed description of the mathematical structures governing the elliptic curves and their applicability to cryptosystems can be found in [29, 33]. In this section, we review some mathematical properties of elliptic curves that make them suitable for secure cryptosystems.

2.5.1 Elliptic Curves

The mathematical expressions and framework presented in this section are taken from [4,9]. An elliptic curve, E, over real numbers is defined as the set of points (x, y) which satisfy the following equation

$$E : y^2 = x^3 + ax + b \tag{2.10}$$

where x, y, a and b are real numbers. Different values of a and b result in different elliptic curves. If $4a^3 + 27b^2 \neq 0$ then $x^3 + ax + b$ does not have repeated factors. In such a case, the elliptic curve $E : y^2 = x^3 + ax + b$ can be used to form a group.

The curve E defined on real numbers forms a large set of points. Also, dealing with real numbers incurs precision errors. When all points on E are defined in a set $F = \{0, 1, \ldots, N-1\}$, where N is prime, E becomes a field defined on F_N or $GF(N)$. E is now defined as

$$E(GF(N)) : y^2 \ mod\, N = (x^3 + ax + b) \ mod\, N \tag{2.11}$$

All the points on $E(GF(N))$ are now defined in $GF(N)$. In a similar fashion, an elliptic curve E can also be defined in the field $GF(2^m)$ as shown below.

$$E(GF(2^m)) : (y^2 + xy) \ mod\, f(x) = (x^3 + ax + b) \ mod\, f(x) \tag{2.12}$$

where x and y are defined in $GF(2^m)$ as polynomial basis (Section 2.2) and $f(x)$ is an irreducible polynomial defined over $GF(2^m)$.

The number of points on $E(GF(N))$ and $E(GF(2^m))$ are finite. Operations defined on elliptic curves are called point operations. Three point operations are crucial in ECCs. They are point addition (PA), point doubling (PD) and point multiplication (PM).

Point Operations

Two points $P(x_1, y_1)$ and $Q(x_2, y_2)$ on an elliptic curve E can be added graphically as shown in Fig. 2.1 to give $R(x_3, y_3)$. Mathematically $R(x_3, y_3)$ is computed using the following equations.

$$\begin{aligned}
R(x_3,y_3) &= P(x_1,y_1) + Q(x_2,y_2) && \text{where } P \neq Q \text{ and } x_1 \neq x_2 \\
&= (\lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1) && \text{where } \lambda = \tfrac{y_2 - y_1}{x_2 - x_1} \\
& && \text{and } P, Q \in E(GF(N)) \\
or & \quad (\lambda^2 + \lambda + x_1 + x_2 + a, \lambda(x_1 + x_3) + x_3 + y_1) && \text{where } \lambda = \tfrac{y_2 + y_1}{x_2 + x_1} \\
& && \text{and } P, Q \in E(GF(2^m))
\end{aligned} \tag{2.13}$$

Point doubling (PD) of a point P on an elliptic curve E is defined as

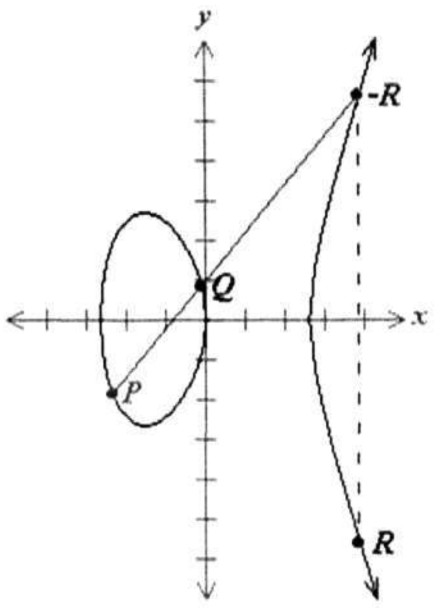


Figure 2.1: Point addition (PA) on elliptic curves

$$R = 2P \quad \text{where } R, P \in E \tag{2.14}$$

R can be graphically calculated as shown in Fig. 2.2. Mathematically, $R(x_3, y_3)$ is computed using the following equations.

$$\begin{aligned} R(x_3,y_3) &= P(x_1,y_1)+P(x_1,y_1) && \text{where } y_1 \neq 0 \\ &= (\lambda^2 - 2x_1, \lambda(x_1 - x_3) - y_1) && \text{where } \lambda = \tfrac{3x_1^2+a}{2y_1} \\ & && \text{and } P \in E(GF(N)) \\ \mathit{or} \;\; & (\lambda^2+\lambda+a, \lambda(x_1+x_3)+x_3+y_1) && \text{where } \lambda = x_1 + \tfrac{x_1}{y_1} \\ & && \text{and } P \in E(GF(2^m)) \end{aligned} \tag{2.15}$$

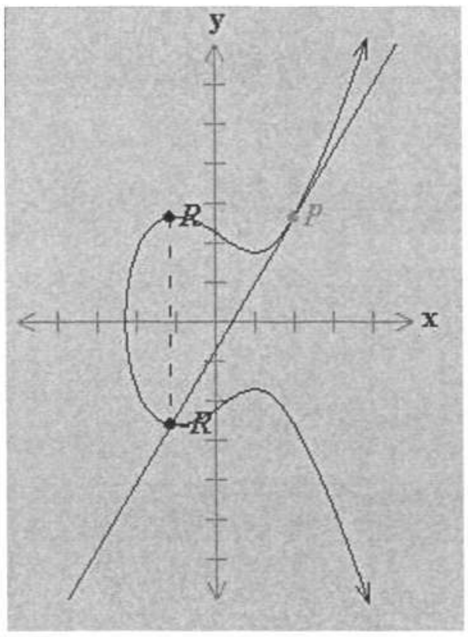


Figure 2.2: Point doubling (PD) on elliptic curves

Point multiplication (PM) or scalar multiplication is defined as

$$Q = kP = \overbrace{P + P + \cdots + P}^{k} \tag{2.16}$$

where k is an integer and Q is called the scalar product. Bit-wise implementation of PM is given in [26] as shown in Algorithm 3. The PM algorithm employs point addition (PA)

Algorithm 3 Elliptic curve point multiplication

```
 1: Input: Point $P(x, y)$ on elliptic curve $E$,
    large integer $k = k_{l-1}k_{l-2}\ldots k_0$ where $k_{l-1} = 1$
 2: Output: $Q = kP$
 3: $Q \leftarrow P$
 4: for $i = l - 2$ downto 0 do
 5:   $Q \leftarrow 2Q$
 6:   if $k_i = 1$ then
 7:     $Q \leftarrow Q + P$
 8:   end if
 9: end for
10: Return $Q$
```

and point doubling (PD) operations repeatedly in `Lines 5` and `7`, respectively. [25,30,42] list different algorithms for PA and PD. According to [25], the number of finite field multiplications F_{MULT}, additions F_{ADD}, inverses F_{INV} and squarings F_{SQR} involved in one PM operation are listed below.

$$\begin{aligned} F_{INV} &= 2\lfloor log_2 k \rfloor + 1 \\ F_{ADD} &= 4\lfloor log_2 k \rfloor + 6 \\ F_{MULT} &= 2\lfloor log_2 k \rfloor + 4 \\ F_{SQR} &= 2\lfloor log_2 k \rfloor + 2 \end{aligned} \qquad (2.17)$$

It can be seen from the above equations that modular or finite field multiplication is the most frequently used operation that comprises a point multiplication operation. It will be illustrated later that modular multiplication algorithms have repeated addition operations. Though addition operation is simple, the numbers that are involved in PKCs are of the order of 128 bits, at the very least. Thus, the area-time complexity increases due to the wordlengths of the operands that are involved in these operations. Hence modular multiplication becomes the bottleneck of encryption/decryption algorithms in PKCs. The hardware complexity of this operation in the context of cryptosystems is an actively researched topic and it is also the focus of this thesis. Before surveying some commonly used modular multiplication techniques in ECCs, let us study how it is applied in elliptic curve cryptosystems by looking into the ECC protocol.

2.5.2 Protocol in ECC

In ECC, an elliptic curve needs to be identified by the users first. An elliptic curve is defined using a septuple shown below [4, 25]

$$T = (q, FR, a, b, G, n, h) \tag{2.18}$$

In this septuple, q is either a prime number N or a power-of-two integer, 2^m. It defines the field representation FR in which the elliptic curve operates. FR is either $GF(N)$ or $GF(2^m)$ depending on q. a and b are the coefficients in the curve. Their values are dependent on the security requirement of the ECC. G is called the base point of the elliptic curve. It is used for the generation of keys. n is the order of G and the bit length of n is equal to the key length of ECC and h is called the cofactor of the elliptic curve [4].

The basic principle of ECC [4] is explained by the following scenario.

1. Alice and Bob decide on using elliptic curve cryptosystem.

2. Bob selects a large random integer $d_B \in [1, n-1]$ as private key and publishes the scalar product $Q_B = d_B G$ as public key.

3. Alice selects a random integer k and generates the ciphertext using the public key Q_B by employing $(kG, (kQ_B)_x + m)$ where the subscript x signifies the x-coordinate on the elliptic curve and m is the message. This ciphertext is sent to Bob.

4. Bob decrypts the cipher text by using his private key d_B using the following equation.

$$[m + (kQ_B)_x] - d_B [kG_x] = m + (kd_BG)_x - (d_BkG)_x = m \tag{2.19}$$

Point multiplication is an integral part of ECC, starting from the key generation in Step 2 of the above procedure to encryption and decryption in Steps 3 and 4, respectively. This operation forms the basis for the security offered by ECC which is termed as Elliptic Curve Discrete Logarithmic Problem (ECDLP). Given two points $P, Q \in E(GF(p))$, ECDLP determines an integer k that satisfies $Q = kP$, where $0 \leq k \leq n-1$ and n is the order of P [4]. This equation is the point multiplication operation, that is used in ECC. When k is large, determining k from this equation is a difficult problem. To date, the most efficient general algorithm to solve the ECDLP is Pollard ρ algorithm [4, 25], which has a run time of $\frac{\sqrt{\pi n}}{2r}$, where r is the number of parallel processors.

2.5.3 Modular Multiplication in $GF(2^m)$

The mathematical formulations of an ECC reveal that modular multiplication or finite field multiplication is the propeller to all operations, i.e. point multiplication, addition and doubling. Thus modular multiplication warrants a detailed study to improve the existing algorithms.

In ECC, modular multiplication can be performed in two different fields - $GF(N)$ and $GF(2^m)$. The choice of the field depends on the characteristics of the elliptic curve. RSA cryptosystem that was discussed in Section 2.4 employs modular multiplication in $GF(N)$. We have also discussed Montgomery modular multiplication that computes modular product in $GF(N)$. In the subsequent sections, modular multiplication in $GF(2^m)$ will be reviewed.

Important techniques for software and hardware implementations of finite field multiplication in $GF(2^m)$ have been reported in the literature [8,13–20,28,40,51,52], [23], [35], [54–57]. Several software implementations of finite field multiplication are listed in [15] whereas efficient hardware architectures are described separately in [8, 13, 14, 16–20, 28, 40, 51, 52], [23], [35], [54–57]. There are generally three different categories into which reported multipliers can be categorized. [8] illustrates a multiplier of the first category where reduction is given higher priority than multiplication. Multiplication is performed by simple *AND-XOR* network. However, reduction is more complicated and efficient implementations of reduction are designed. In [14,16,28,35,40,52], a variety of implementations of Mastrovito, Karatsuba and Massey-Omura multipliers have been reported, which form the second category. The third category comprises systolic and semisystolic multipliers which perform multiplication and reduction simultaneously. This includes Montgomery modular multiplication [22,48,49], LSB-first and MSB-first modular multiplication techniques [13,17–20,51].

The work reported in this thesis focuses on the third category. Novel Montgomery, LSB-first and MSB-first modular multipliers will be designed and developed. Hence, an appraisal of existing methods in this category will be done in detail.

2.5.4 Montgomery Modular Multiplication in $GF(2^m)$

In this section, MMM in $GF(2^m)$ will be studied. Cetin Koc developed MMM for $GF(2^m)$ in [22]. The algorithm is straightforward and is very similar to MMM in $GF(N)$ (Algorithm 2). MMM in $GF(2^m)$ can be formulated as shown in Algorithm 4.
The inputs to this algorithm, $A(x)$, $B(x)$ and $f(x)$, and its output, $S(x)$, are in polynomial basis representation. The main difference between MMM in $GF(2^m)$ and $GF(N)$ is the addition operations that take place in `Lines 5 and 6`. These addition operations are performed in $GF(2^m)$. As a result, all vector additions are performed modulo 2, making them simple bitwise XOR operations. These additions do not involve any carry propagation. In

Algorithm 4 Montgomery Modular Multiplication in $GF(2^m)$

```
1: Input: $A(x) = \sum_i^{m-1} a_i x^i, B(x) = \sum_i^{m-1} b_i x^i, f(x) = \sum_i^{m-1} f_i x^i$
2: Output : $S(x) = A(x)B(x)x^{-m}(mod\ f(x))$
3: $S \leftarrow 0$
4: for $i = 0$ to $m-1$ do
5:     $S \leftarrow S \oplus a_i B$
6:     $S \leftarrow S \oplus s_0 f_i$
7:     $S \leftarrow S/x$
8: end for
9: Return $S(x) = A(x)B(x)x^{-m}(mod\ f(x))$
```

addition, Line 7 of the two algorithms are different but they are mathematically equivalent in their respective fields. A division by x in $GF(2^m)$ is equivalent to a right shift by one bit, which is the same as division by 2 in $GF(N)$.

Considering the similarities of MMM in the two fields, [48, 49] illustrate unified Montgomery multipliers that can operate in either field using an additional control signal called the field select signal. Specialized adder cells, called dual field adders, are employed. In these dual field adders, the field select signal forces the carry output to zero while operating in $GF(2^m)$ mode. A comprehensive survey of the Montgomery modular multiplication can be found in [21].

2.5.5 LSB-first and MSB-first Multipliers

In this section, a detailed survey of existing LSB-first and MSB-first modular multipliers is carried out. The survey presented in this section has been published by the author in [45]. Different input-output configurations of these multipliers will be studied in detail. These multipliers were synthesized and mapped to standard cell libraries using Synopsys Design Compiler. The prelayout simulation results on power, area and delay of these multipliers are analyzed.

The LSB-first and MSB-first modular multiplication in $GF(2^m)$, for multiplying the polynomials - $a(x)$ and $b(x)$, in a field defined by the reduction polynomial $f(x)$ are formulated as shown below.

LSB-first modular multiplication:

$$\begin{aligned} c(x) &= a(x)b(x) \; mod \; f(x) \\ &= b_0 a(x) + b_1[a(x)x \; mod \; f(x)] + b_2[a(x)x^2 \; mod \; f(x)] + \cdots \\ &\quad + b_{m-1}[a(x)x^{m-1} \; mod \; f(x)] \end{aligned} \tag{2.20}$$

MSB-first modular multiplication:

$$\begin{aligned} c(x) &= a(x)b(x) \; mod \; f(x) \\ &= \{\cdots[a(x)b_{m-1}x \; mod \; f(x) + a(x)b_{m-2}]x \; mod \; f(x) + \cdots \\ &\quad + a(x)b_1\}x \; mod \; f(x) + a(x)b_0 \end{aligned} \tag{2.21}$$

As the names suggest, one of the input operands is read from the least significant bit (lsb) in the LSB-first multiplication (2.20) and from the most significant bit (msb) in the MSB-first multiplication (2.21). The algorithms for their bit-level implementation are listed in Algorithms 5 and 6. In these algorithms, the j-th bit of vector X in the i-th iteration is represented as $x_j^{(i)}$.

Algorithm 5 LSB-first bit-level algorithm for fixed field order

```
1: Input: $a(x)$, $b(x)$, $f(x)$
2: Output : $c(x) = a(x)b(x) \; mod \; f(x)$
3: $t_j^{(0)} = 0$ for $0 \leq j \leq m-1$
4: $a_{-1}^{(i)} = 0$ for $0 \leq i \leq m-1$
5: $a_j^{(0)} = 0$ for $0 \leq j \leq m-1$
6: for $i = 1$ to $m$ do
7:     for $j = 0$ to $m-1$ do
8:         $a_j^{(i)} = a_{j-1}^{(i-1)} + a_{m-1}^{(i-1)} f_j$
9:         $t_j^{(i)} = a_j^{(i-1)} b_{i-1} + t_j^{(i-1)}$
10:    end for
11: end for
12: $c(x) = t^{(m)}(x)$
```

Algorithm 6 MSB-first bit-level algorithm for fixed field order

```
1: Input: $a(x)$, $b(x)$, $f(x)$
2: Output : $c(x) = a(x)b(x)\ mod\ f(x)$
3: $t_j^{(0)} = 0$ for $0 \leq j \leq m-1$
4: $t_0^{(i)} = 0$ for $1 \leq j \leq m$
5: $t_{-1}^{(i)} = 0$ for $0 \leq i \leq m-1$
6: for $i = 1$ to $m$ do
7:     for $j = m-1$ to $0$ do
8:         $t_j^{(i)} = t_{m-1}^{(i-1)} f_j + b_{m-i} a_j + t_{j-1}^{(i-1)}$
9:     end for
10: end for
11: $c(x) = t^{(m)}(x)$
```

Both algorithms have been realized with bit-serial and bit-parallel architectures in [13, 17–20, 51] [23] [35]. Bit-parallel multipliers [18, 51] provide higher throughput at a cost of increased hardware. Hardware complexity is an important criterion for public key cryptography on smart cards in view of the need to realize finite field operations with large word length on a small footprint. Therefore, bit-serial multipliers [51] are preferred over bit-parallel counterparts in those applications. However, latency becomes an issue in bit-serial multipliers. Digit-serial multipliers [13, 19, 20] have also been reported to reduce latency but digitizing the design increases hardware complexity and combinational delay as compared to bit-serial multipliers. Digit-serial multipliers however compute the modular product in lesser number of cycles.

2.5.6 Architectures of LSB-first/MSB-first Multipliers

In this section, some systolic and semisystolic architectures of the LSB-first and MSB-first multipliers will be evaluated to study their implications on hardware implementation. A quantitative analysis in terms of gate count and latency is presented here. Algorithm to architecture mapping is described briefly for each of the multipliers to be evaluated.

Bit-serial/parallel Array Multipliers

In [18], a bit-level pipelined parallel-in parallel-out LSB-first semisystolic multiplier is proposed based on Algorithm 5. The basic cell, which computes $a_j^{(i)}$ and $t_j^{(i)}$ at `Line 'i'` is shown in Fig. 2.3(a) with the architecture of an m-bit multiplier. The dotted lines indicate cutsets where registers are placed to pipeline it. This implementation comprises m^2 basic cells and $2m^2$ l-bit latches. The latency of this implementation is $m+1$ clock cycles. From Fig. 2.3(a), we see that the critical path consists of one two-input AND gate followed by one two-input XOR gate, i.e., $\Delta_{2-AND} + \Delta_{2-XOR}$.

Similarly, a bit-level pipelined MSB-first semisystolic multiplier is implemented in [18, 23] as shown in Fig. 2.3(b). The basic cell in Fig. 2.3(b) computes $t_j(i)$ at `Line 'i'`. The critical path is limited by the delay in the basic cell and is given by $\Delta_{2-AND} + 2\Delta_{2-XOR}$ which is longer than the LSB-first multiplier by Δ_{2-XOR}. The number of cycles, however, to complete the operation remains the same, i.e., $m + 1$ clock cycles. In addition, the number of 1-bit registers is reduced to m^2 in the MSB-first implementation because only $t_j^{(i)}$ for all $0 \leq i, j \leq (m-1)$ have to be latched in each basic cell.

In [51], a serial-in serial-out multiplier is described based on the MSB-first algorithm. The basic cell for this architecture is shown in Fig. 2.4(a). In addition to the logic of an MSB-first basic cell, it comprises one 2-to-1 multiplexer MUX, one 2-input AND gate and a register. The multiplexer and register are added to propagate data in a serial-in serial-out fashion. Also, an m-bit control sequence $111\cdots110$ is sent in serially into the array for it to execute in a serial-in serial-out mode.

Fig. 2.4(b) shows the serial-in serial-out architecture of an m-bit multiplier in $GF(2^m)$. It comprises m basic cells and has a latency of $3m$ clock cycles with a throughput rate of $1/m$. The combinational delay is equivalent to $\Delta_{2-AND} + 2\Delta_{2-XOR}$ (a 3-input XOR gate

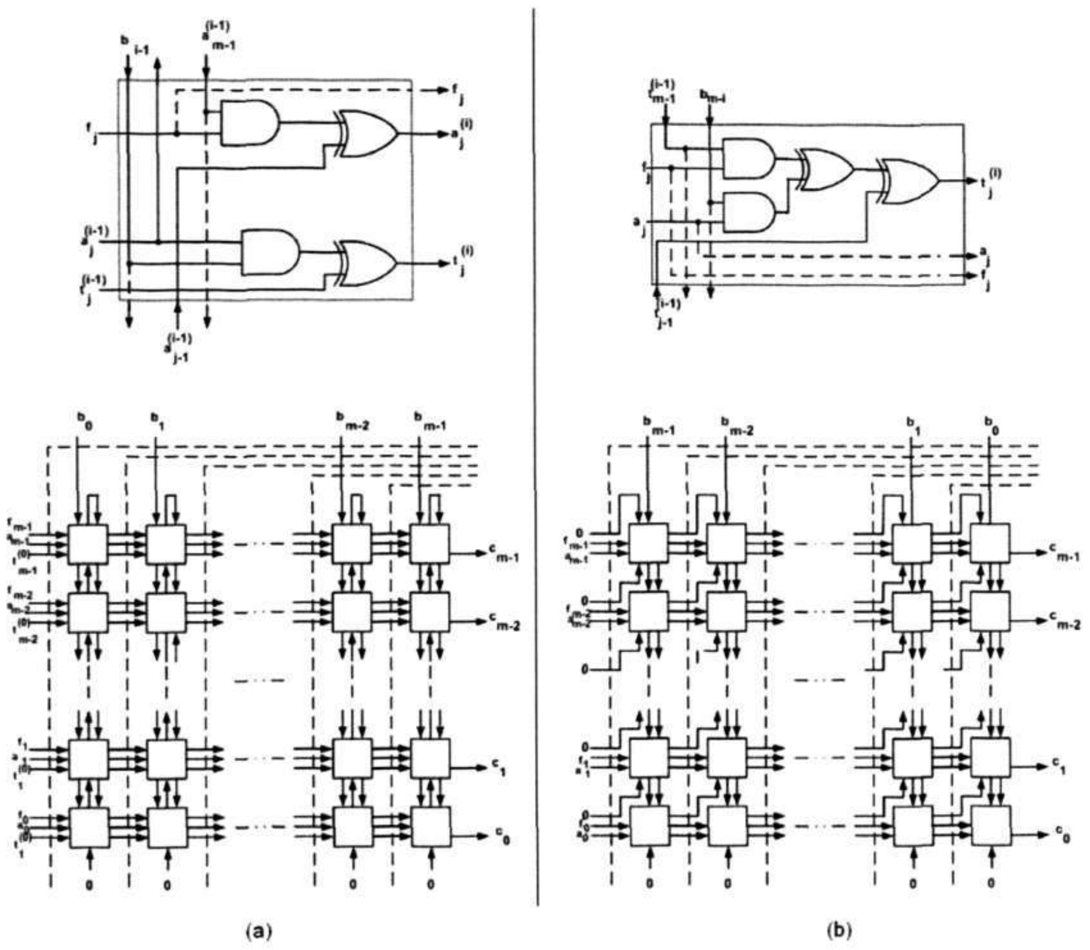


Figure 2.3: (a) Basic cell and architecture for LSB-first bit-parallel finite field multiplier [18](b) Basic cell and architecture for MSB-first bit-parallel finite field multiplier [18]

is considered as equivalent to two 2-input XOR gates). The logic complexity of the basic cell is given by three 2-input AND gates, one 3-input XOR gate, nine 1-bit latches and one switch (multiplexer).

Digit Serial Systolic Multipliers

Two digit-serial systolic multipliers are described in [13] and [20]. Both of them are based on the MSB-first algorithm. The dependency graph (DG) shown in Fig. 2.5(a) is modified in [13] and [20] to create a digit-serial multipliers. Each block in Fig. 2.5 refers to the

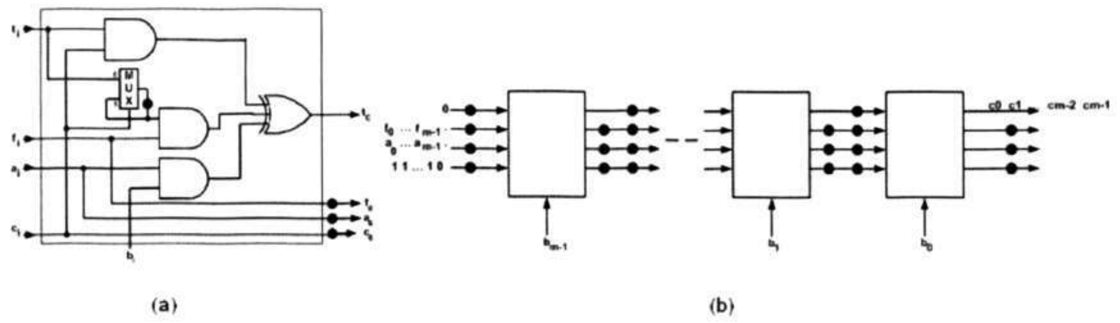


Figure 2.4: (a) Basic cell and (b) Bit-serial MSB-first finite field multiplier [51]

basic MSB-first combinational logic shown in Fig. 2.3(b).

The problem encountered in both architectures to make this two-dimensional DG to a one-dimensional signal flow graph is - how to project the DG in the east direction to obtain a one-dimensional signal flow. In [13], for a digit length 'L', a basic module comprising L^2 cells is selected. These cells form a square grid of L cells in the **x** and **y** directions. Extra circuitry consisting of 4-input XOR gates is used in the basic module to overcome the bidirectional signal flow in the DG of Fig. 2.5(a). In an attempt to digitize the architecture, nine MSB-first logic blocks shown previously in Fig. 2.3(b) are grouped into one block. The flow of data is controlled by additional multiplexers external to this circuitry. More details on the complete derivation can be found in [13]. For a digit size of L, each basic cell has a logic complexity of $(L-1)$ 2-input XOR gates, $(2L^2+L)$ 2-input AND gates, $(L-1)$ 4-input XOR gates, $10L$ 1-bit latches and $2L$ switches (2-to-1 multiplexers) [13].

In [20], the DG is modified to prevent the use of 4-input XOR gates. The authors transform indices of the DG to form a new DG where each row in DG (see Fig. 2.5(a)) is shifted towards the right by one basic cell, i.e., Cell $(2, m-1)$ is placed under Cell $(1, m-2)$ instead of Cell $(1, m-1)$. The new DG is then divided into m/L parts horizontally where each part comprises L rows. Fig. 2.5(b) shows the DG partitioning for a 4-bit multiplier

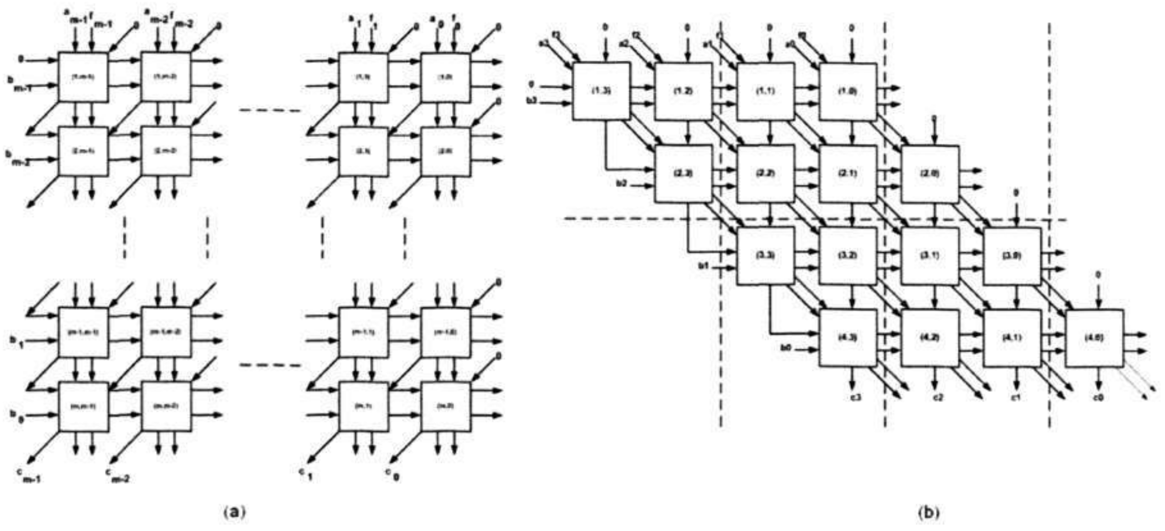


Figure 2.5: (a) DG as in [13] (b) DG as in [20]

where $m = 4$ and $L = 2$. Each part is further divided into $m/L + 1$ regions vertically. Due to the transformation of indices, the regions have different number of cells. Each basic module as shown in [20] has L^2 cells. In contrast, the cutsets in Fig. 2.5(b) do not form equal regions. Latches and multiplexers are introduced between cells in the basic module to accommodate the cutsets. More information pertaining to the basic module and cutsets can be obtained from [20]. The complexity of the basic module is given by $2L^2$ 2-input AND gates, L^2 3-input XOR gates, $(8L + 2)$ 1-bit latches and $2L$ switches.

Generalized Cellular-array Multipliers

In [18], two generalized cellular-array multipliers are proposed based on the LSB-first and MSB-first algorithms presented earlier. The prefix 'generalized' refers to the applicability of the structure to varying field order, i.e., the multiplier is designed for multiplication over $GF(2^M)$ but the primitive polynomial $f(x)$ is of order $'m'$ such that $m \leq M$. The higher order bits of all inputs are padded with zeros to make them M bits long.

The authors of [18] first determine the order of the primitive polynomial by the bit-locator cells - LCELL. The H vector from LCELLs is an M-bit binary string with only one non-zero bit. This *1* bit corresponds to the field order. For example, if $p(x) = x^5 + x + 1$ and $M = 9$, then $H = 00010000$. This is followed by modifying both algorithms to incorporate programmability of field order m. The array is now made up of MCELLs (multiplier cells), which are based on the modified algorithms in [18]. Based on the architectures presented in [18], the complexity of the basic cell in the generalized LSB-first array multiplier is increased by four gates compared to the conventional LSB-first multiplier described previously. Similarly the complexity is increased by 3 gates in the MSB-first programmable multiplier compared to its fixed order counterpart. LCELLs form extra circuitries that are needed for the precomputation of H vector for the generalized architectures. One disadvantage of generalized architectures is that the critical path is longer. It runs vertically in the array due to signals propagated vertically in the array. As a result, the clock frequency has to be reduced. The critical path is equivalent to $(m-1)\Delta_{2-OR}$ in both cases.

These generalized architectures will be studied in greater detail and new architectures with prominent performance improvement over the architectures of [18] will be proposed in Chapter 5.

2.5.7 Evaluation of LSB-first/MSB-first Multipliers

In this section, the architectures discussed in Section 2.5.6 are evaluated using ASIC implementation. These prelayout simulation results provide a reasonably good estimate of the relative VLSI performance in terms of metrics such as silicon area, critical path delay and dynamic power dissipation. Structural VHDL codes were generated automatically using C programs to facilitate more rigorous simulations of different architectures under varying field size. The designs were synthesized, optimized and simulated using Synopsys Design Compiler version 2004.06. The finite field multipliers were optimized with consistent constraints set in Design Compiler. The input and output loads were set to 0.8pF and 0.9pF, respectively. Cross boundary optimization was enabled to harness further area reduction. Power simulations were performed using Synopsys Power Compiler with a set of thousand random input vectors at a clock frequency of 20MHz. The area results are also obtained at this clock frequency.

Nomenclature

Nine different derivatives of finite field multipliers from Section 2.5.6 were simulated. The reason for choosing the different multipliers is to survey a wide variety of bit-serial, bit-parallel, digit-serial and generalized multipliers for a better understanding of LSB-first/MSB-first multipliers.

For the sake of convenience, a nomenclature is used. The names for the different multipliers are discussed here.

- **MUL1:** It is a parallel-in parallel-out LSB-first multiplier from [18].
- **MUL2:** It is a parallel-in parallel-out MSB-first multiplier from [18].
- **MUL3:** It is a serial-in serial-out MSB-first multiplier from [51].
- **MUL4_BS:** It is a digit serial multiplier from [13] but it is converted to a bit serial multiplier by setting the digit length to 1 bit.
- **MUL5_BS:** It is a digit serial multiplier from [20] but it is converted to a bit serial multiplier by setting the digit length to 1 bit.
- **MUL4_DS:** It is a digit serial multiplier from [13] with the digit size set to 8 bits.
- **MUL5_DS:** It is a digit serial multiplier from [20] with the digit size set to 8 bits.
- **MUL6:** It is a generalized parallel-in parallel-out LSB-first multiplier from [18]
- **MUL7:** It is a generalized parallel-in parallel-out MSB-first multiplier from [18]

In the above list, the digit serial multipliers in [13] and [20] are implemented in two different ways. The digit serial multipliers with a digit size of 1 bit become bit serial multipliers. MUL4_BS and MUL5_BS are the names for these multipliers. MUL4_DS and MUL5_DS represent the same digit serial multipliers from [13] and [20] respectively, but with a digit size of 8 bits. MUL4_BS and MUL5_BS are generated to study the affect of changing a digit serial multiplier into a bit serial multiplier.

MUL1, MUL2, MUL3, MUL4_BS, MUL5_BS, MUL4_DS and MUL5_DS are implemented for six different field orders - 57, 97, 113, 131, 163 and 193. These field orders are taken from the ECC standards in [3,37].

The last two multipliers - MUL6 and MUL7, are generalized multipliers, i.e. they should be able to operate for any field order less than a maximum field order. So, MUL6 and MUL7 are implemented for a field order of 193 bits only. A keylength of 193 bits is considered 'highly secure' in ECC as it is equivalent to a 1536-bit key length in RSA [3].

Table 2.1 summarizes the properties of the different multipliers. The input/output configuration, algorithm used, cell complexity, number of cells and the latency for each of the multipliers are tabulated.

Bit-level parallel-in parallel-out implementations [18]

MUL1 and MUL2 fall in this category. Fig. 2.6(a) and (b) compare the silicon area and dynamic power dissipation respectively. Fig. 2.6(a) shows that MUL1 (LSB-first) occupies higher silicon area than MUL2 (MSB-first). The difference is attributed mainly to the extra latches in MUL2. It was shown previously in Section 2.5.6 that MUL2 employs m^2 latches whereas MUL1 has $2m^2$ latches. However, there is a marked increase in the difference between the two designs as m increases. As m increases towards 193, the area of MUL2

Table 2.1: Summary of different multipliers

Multiplier	Algorithm Configuration	Input/Output	Cell complexity	No. of cells	Latency cycles
MUL1 [18]	LSB-first	Parallel-in parallel-out	2 AND_2 2 XOR_2 2 FFs	m^2	$m+1$
MUL2 [18]	MSB-first	Parallel-in parallel-out	2 AND_2 2 XOR_2 1 FFs	m^2	$m+1$
MUL3 [51]	MSB-first	Bit-serial	3 AND_2 2 XOR_2 1 2to1 MUX 7 FFs	m	$3m$
MUL4_BS [13]	MSB-first	Digit-serial converted to bit-serial, $L=1$	$(2L^2+L)$ AND_2 $(L-1)$ XOR_2 $(L-1)$ XOR_4 $2L$ 2to1 MUX $10L$ FFs	$\frac{m}{L}$	$3m$
MUL5_BS [20]	MSB-first	Digit-serial converted to bit-serial, $L=1$	$2L^2$ AND_2 L^2 XOR_3 $2L$ 2to1 MUX $(8L+2)$ FFs	$\frac{m}{L}$	$3m$
MUL4_DS [13]	MSB-first	Digit-serial $L=8$	$(2L^2+L)$ AND_2 $(L-1)$ XOR_2 $(L-1)$ XOR_4 $2L$ 2to1 MUX $10L$ FFs	$\frac{m}{L}$	$\frac{m}{L}$
MUL5_DS [20]	MSB-first	Digit-serial $L=8$	$2L^2$ AND_2 L^2 XOR_3 $2L$ 2to1 MUX $(8L+2)$ FFs	$\frac{m}{L}$	$\frac{m}{L}$
MUL6 [18]	Generalized LSB-first	Parallel-in parallel-out	5 AND_2 2 XOR_2 2 FFs	M^2	$M+1$
MUL7 [18]	Generalized MSB-first	Parallel-in parallel-out	5 AND_2 2 XOR_2 1 FFs	M^2	$M+1$
AND_2: 2-input AND gate, XOR_2: 2-input XOR gate, XOR_3: 3-input XOR gate, XOR_4: 4-input XOR gate					

approaches that of MUL1 with a lower field order. Fig. 2.6(b) shows that both MUL1 and MUL2 dissipate around the same amount of dynamic power with MUL2 having a slight edge over MUL1. This shows the ascendancy of MSB-first algorithm in terms of silicon area and dynamic power dissipation, particularly for higher field orders. The critical path delays of both designs are comparable and is approximately 1.36 ns.

Serial-in serial-out implementations [13, 20, 51]

MUL3 in [51] is a bit serial multiplier whereas those reported in [13] and [20] are digit serial multipliers. Thus based on the architectures of [13] and [20], four variants, bit-serial multipliers - MUL4_BS, MUL5_BS and digit-serial multipliers - MUL4_DS and MUL5_DS,

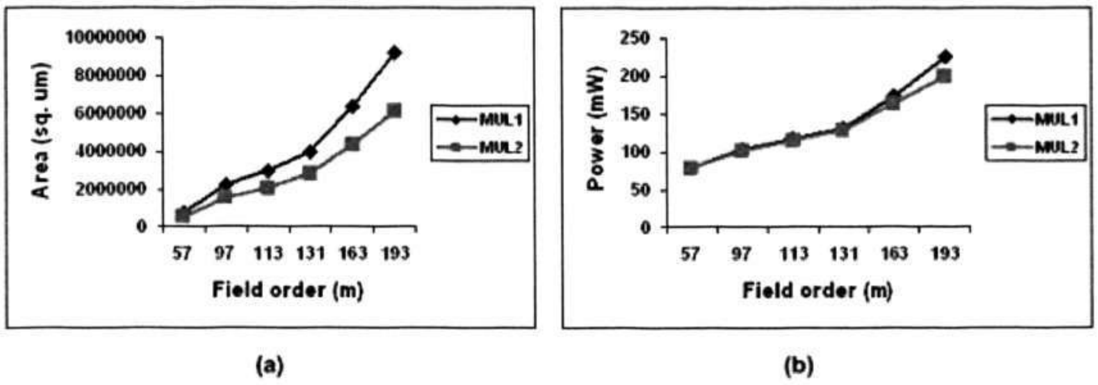
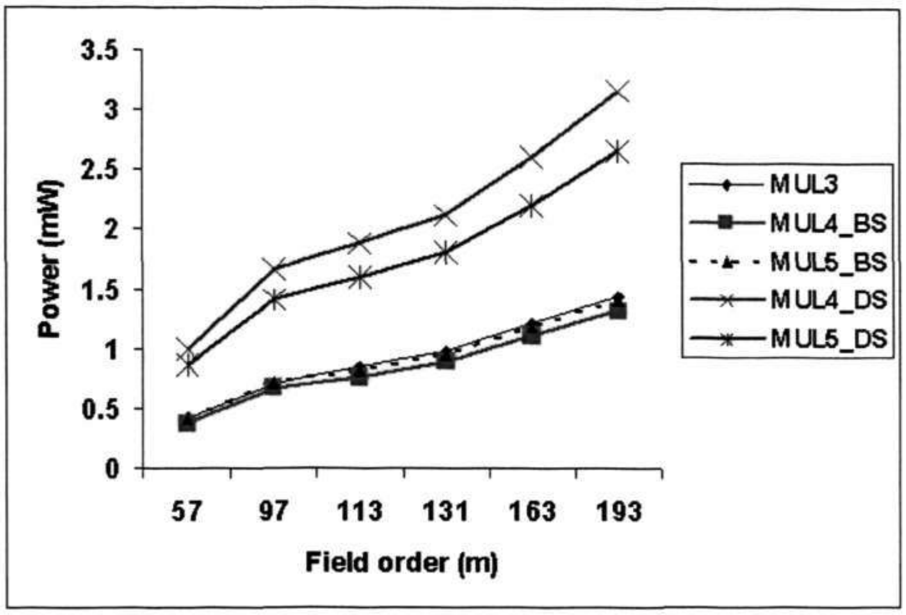


Figure 2.6: (a) Area results and (b) Power results of MUL1 and MUL2

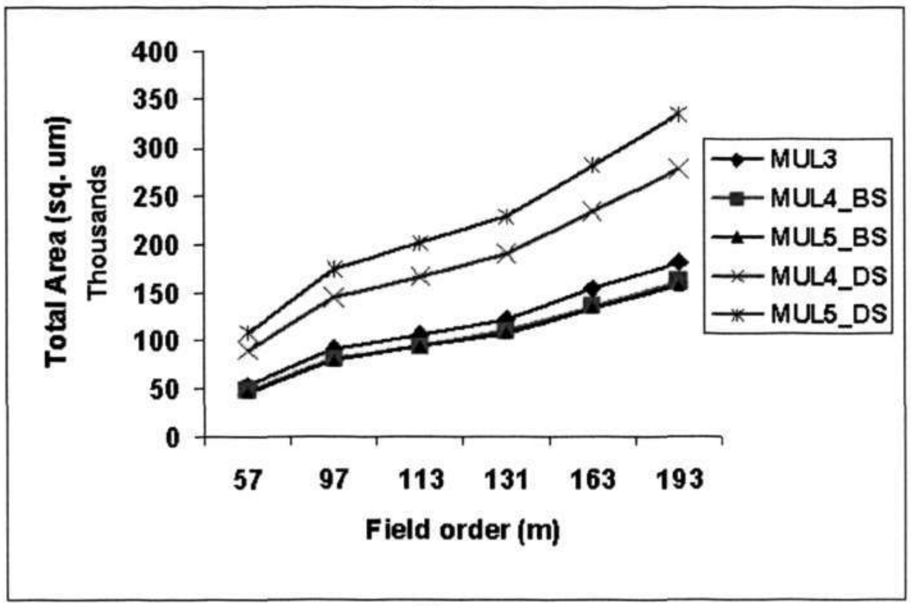
are created. The delay in all the bit-serial implementations is equal to 1.36 ns because the critical path is $\Delta_{2-AND} + 2\Delta_{2-XOR}$ in all the designs. In contrast to the bit-serial implementations, the delays of digit-serial multipliers MUL4_DS and MUL5_DS are 4.16 ns and 9.18 ns, respectively for a digit length of 8. The critical path delay in the digit serial multipliers depends on the digit length L. As L increases, the delay also increases.

The serial multipliers are compared for power and area in Fig. 2.7(a) and Fig. 2.7(b), respectively. MUL3, MUL4_BS and MUL5_BS show almost equal power results for all field orders and MUL3 occupies more area than MUL4_BS and MUL5_BS. It is interesting to note that the bit-serial multipliers, MUL4_BS and MUL5_BS, which are the bit-serial derivatives of the digit-serial implementations, outperform MUL3 in terms of area which was designed with an intention to operate as a bit-serial multiplier. Since the complexity of digit-serial implementations increases as the digit length increases, they show a poor performance when compared to their bit-serial counterparts in terms of both power and area.

Fig. 2.8 and 2.9 compare area and power of the two digit serial implementations, MUL4_DS and MUL5_DS. The two multipliers were compared at gate level in [20]. The ASIC imple-



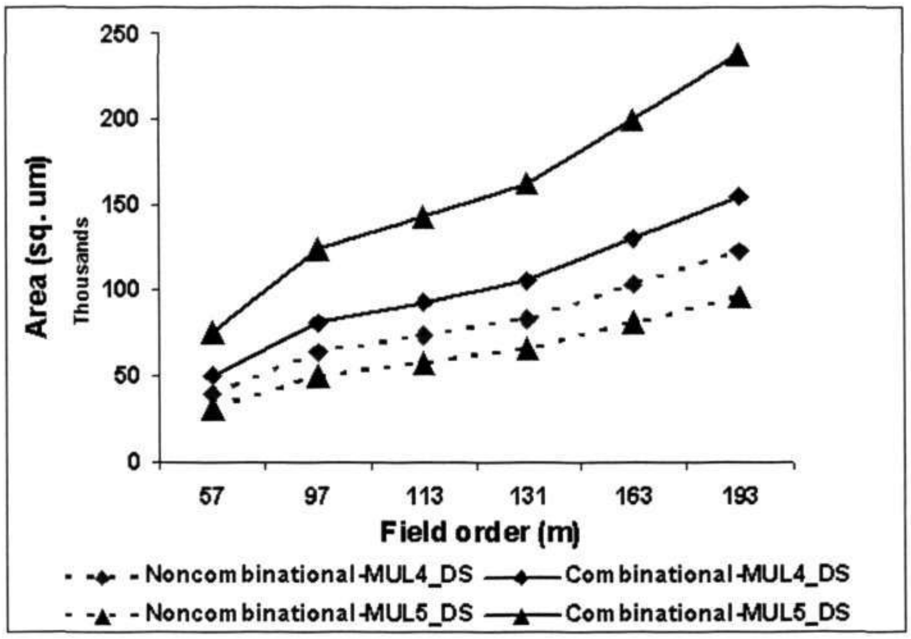
(a)



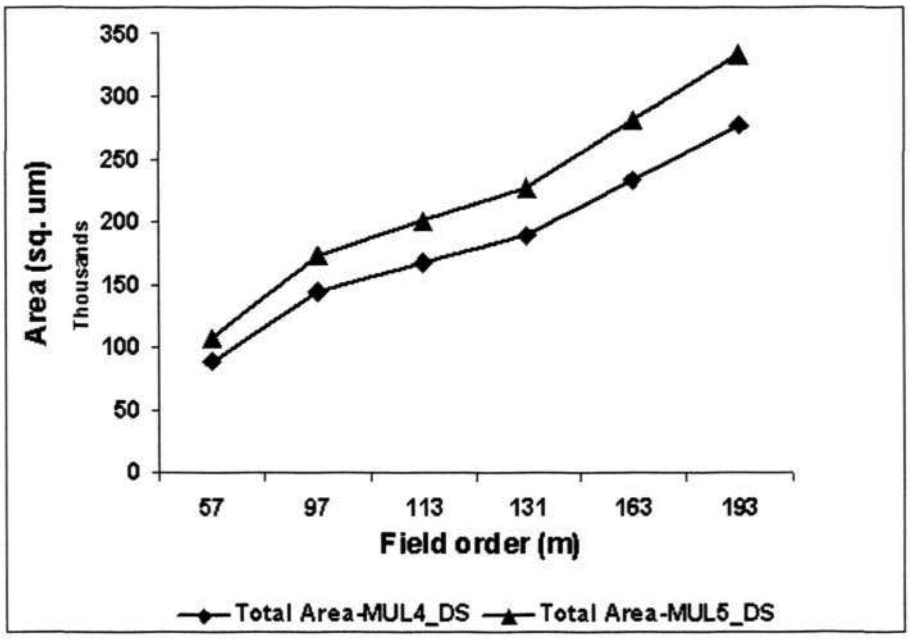
(b)

Figure 2.7: (a) Power and (b) Area results of serial multipliers

mentation of the two designs shows that MUL4_DS is better than MUL5_DS in terms of area but worse off with respect to dynamic power dissipation. In Fig. 2.8 (a), combinational and noncombinational areas of MUL4_DS and MUL5_DS are compared. Although



(a)



(b)

Figure 2.8: (a) Combinational and non-combinational areas (b) Total areas of digit-serial multipliers

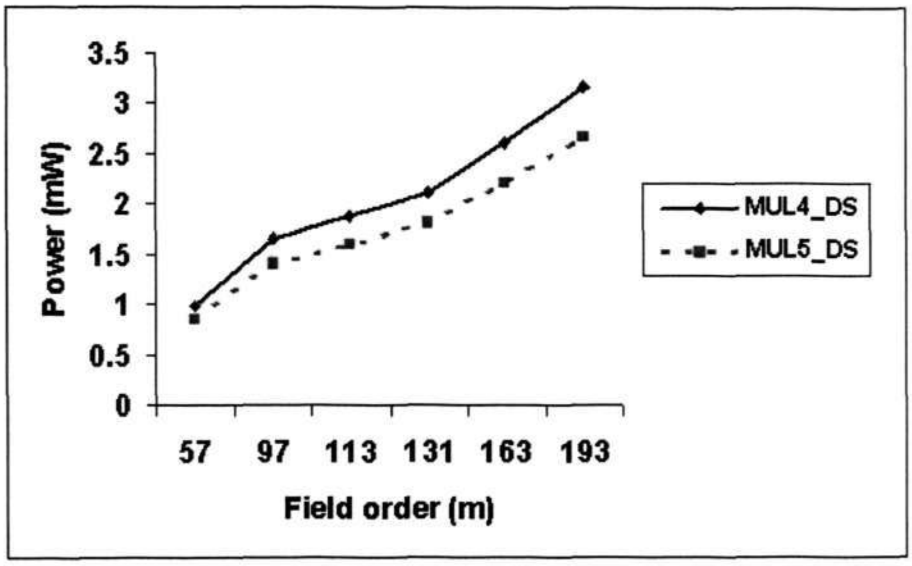


Figure 2.9: Power dissipation of digit-serial multipliers

MUL5_DS has lower non-combinational area, its outweighing combinational area makes it less area efficient than MUL4_DS as shown by the total area plot in Fig. 2.8 (b). This somewhat unexpected synthesis result also explains the higher critical path delay of MUL5_DS (9.18 ns as opposed to 4.16 ns of MUL4_DS).

Generalized bit-level parallel implementations [18]

MUL6 and MUL7 are the generalized bit-parallel implementations of the LSB-first and MSB-first algorithms, respectively, reported in [18] and are implemented for the highest order $M = 193$. Table 2.2 shows the results of the two designs. The results show no significant deviation in all VLSI metrics between the two implementations. If these results are compared with the fixed order bit-parallel architectures of MUL1 and MUL2, the critical path delay is higher. Though MUL6 and MUL7 are configurable in terms of field order, they are an order of magnitude (21 times) slower than their fixed order counterparts due to the longitudinal critical path discussed previously. The area of MUL6 and MUL7 is also higher than MUL1 and MUL2 (1.77 and 2.72 times higher, respectively) due to the

Table 2.2: Results for MUL6 and MUL7

Design	Critical path delay (ns)	Silicon area (μm^2)	Dynamic power (mW)
MUL6	29.34	16358291	341.2509
MUL7	29.55	16678856	342.4379

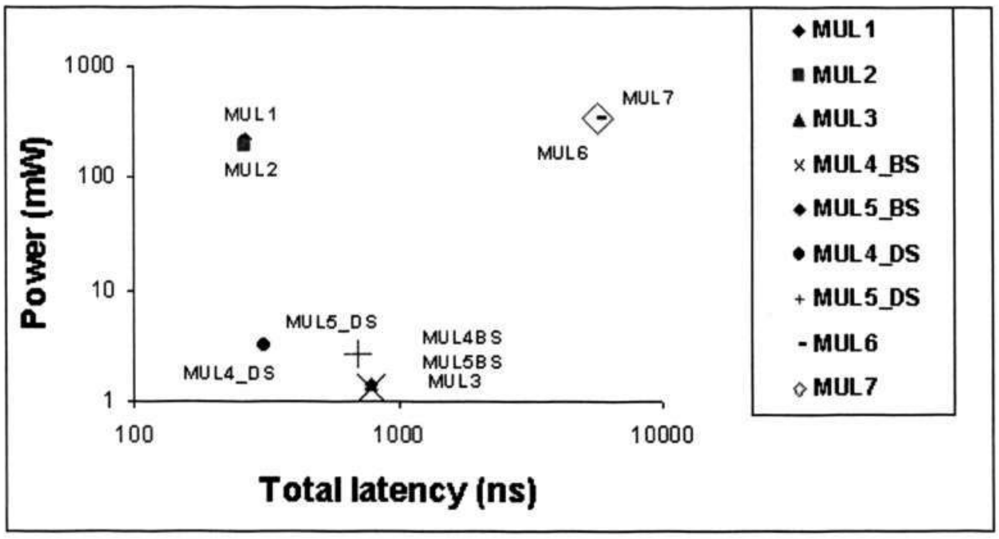


Figure 2.10: Consolidated Results: Power vs. Total Latency

extra circuitry inside the processing elements needed to generalize the architectures for arbitrary field orders.

Consolidated results

Power vs. total latency:Fig. 2.10 shows a scatter plot of power dissipation against latency of all finite field multipliers discussed in this chapter for a field order of 193. The axes are scaled logarithmically for clarity. This plot provides the system architect with a choice of finite field multipliers in a large design space to trade off between power and performance. For fair proposition that involves both bit serial and digit serial multipliers, total latency instead of critical path delay is used. It refers to the product of the critical path delay and number of cycles to complete a single multiplication. Serial implementations outperform parallel multipliers in terms of power dissipation. Despite having similar critical path delays (1.36 ns) due to the use of similar basic cells, the bit-serial multipliers show thrice as much total latency as the bit-parallel multipliers. This is attributed to the difference in the number of clock cycles ($3m$ clock cycles for bit-serial multipliers and $m+1$ clock cycles for bit-parallel multipliers) needed to complete one finite field multiplication operation. MUL4_BS has twice the total latency of its digit-serial counterpart, MUL4_DS. However, the total latency of MUL5_BS is around 1.13 times that of MUL5_DS. This contrast between digit-serial multipliers is because of the critical path delays. Generalized bit-parallel multipliers - MUL6 and MUL7, have a total latency of 21 times their fixed bit-parallel counterparts - MUL1 and MUL3. This is due to the chain of LCELLs and the circuitry used in the array to determine the actual field order. In terms of power dissipation, parallel implementations consume on average, around 78 times higher power than the serial implementations.

Area vs. total latency: Silicon area of each multiplier is plotted against total latency

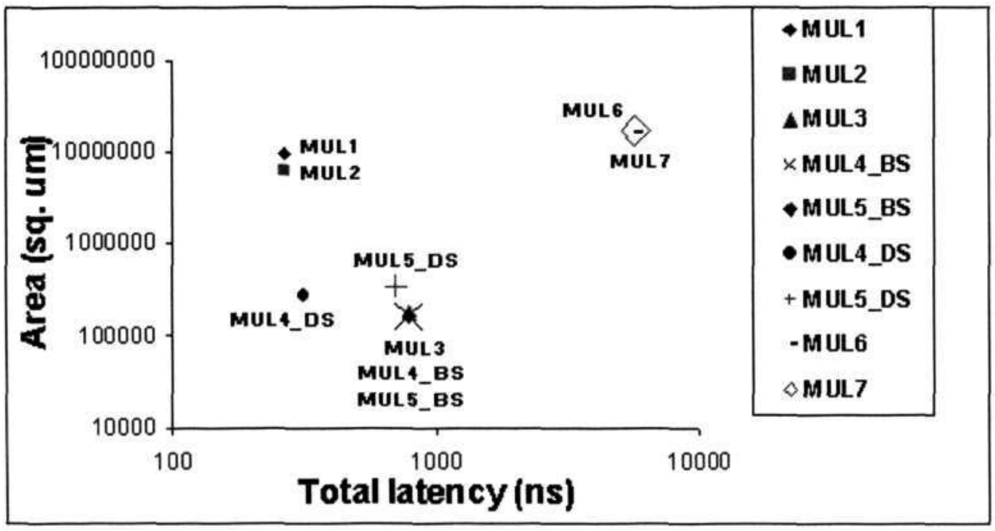


Figure 2.11: Consolidated Results: Area vs. Total Latency

in Fig. 2.11. Serial multipliers again occupy the lower region of the graph with digit-serial multipliers occupying higher area than bit-serial implementations. Bit-serial multipliers, MUL3, MUL4_BS and MUL5_BS, on an average use only 1/58 the area of bit-parallel architectures, MUL1 and MUL2. Digit serial multipliers, MUL4_DS and MUL5_DS, occupy about twice the area of their bit-serial designs, MUL4_BS and MUL5_BS. Generalized bit-parallel designs occupy more area than fixed bit-parallel multipliers. The area of MUL6 (LSB-first architecture) is double that of MUL1, and the area of MUL7 (MSB-first architecture) is around 3 times that of MUL2.

Power vs. area: Fig. 2.12 shows the relationship between the average power dissipa-

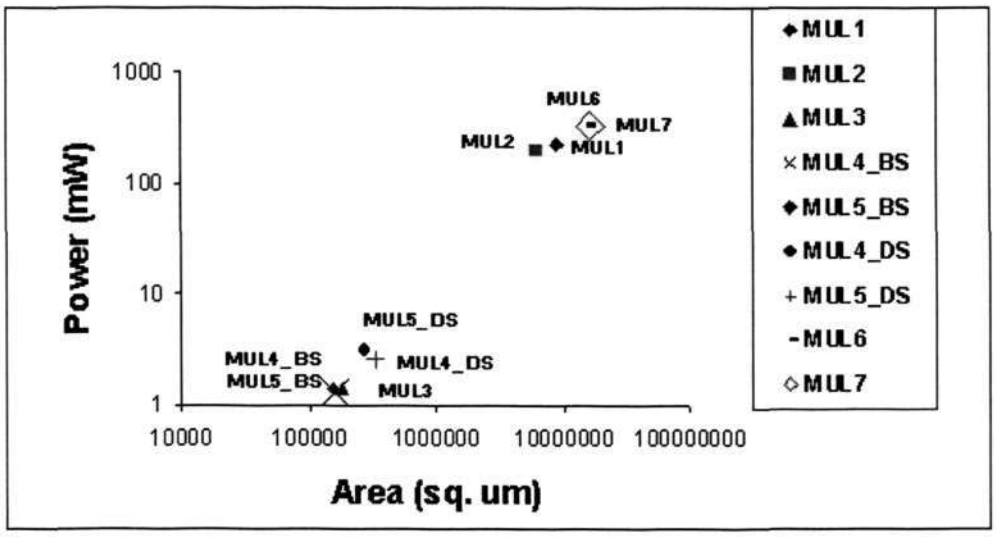


Figure 2.12: Consolidated Results: Power vs. Area

tion and area of each implementation. All serial multipliers are cluttered in the low power, lower area region whereas the parallel implementations occupy the other extreme.

Power per gate vs. field order: Fig. 2.13 shows the power consumed per gate against field order for all multipliers except the generalized parallel implementations. Power per gate is calculated by dividing the average dynamic power by the number of gates used for the implementation. The total number of gates is determined from the total area divided by the area occupied by one two-input $NAND$ gate in TSMC 0.18μm technology

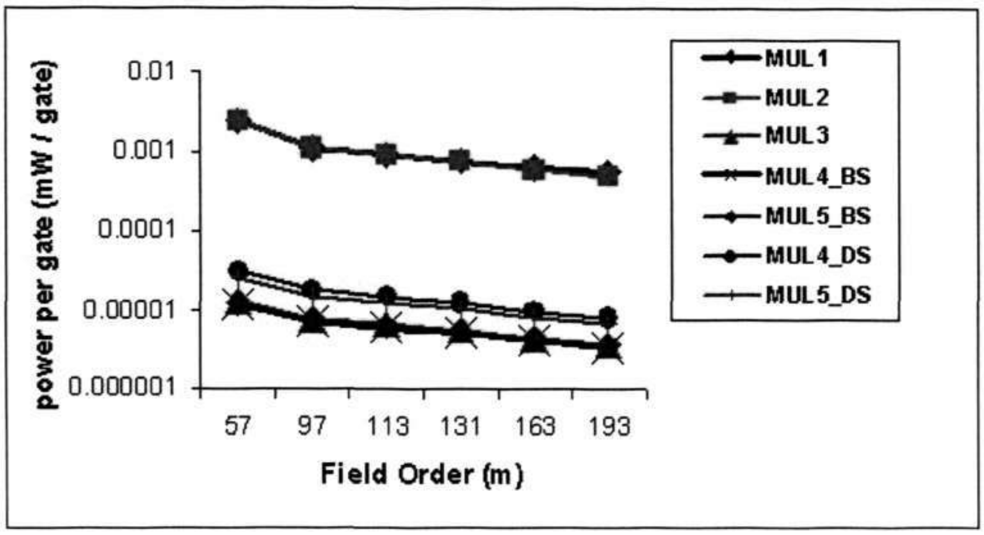


Figure 2.13: Consolidated Results: Power per gate vs. Field order

library. Though the total power increases with the field order, power consumed by each gate is lower as the field order increases as evinced by the consistent trends of all designs in Fig. 2.13. The bit parallel multipliers exhibit higher power consumption per gate than the bit serial or digit serial multipliers. This implies the increased probability of spurious computations in some gates in architectures with more parallelism.

2.6 Inferences and Summary

In this chapter, a detailed literature survey was presented. The mathematical structures of groups and finite fields, $GF(p)$ and $GF(2^m)$ in particular, and their arithmetic operations are described. A brief description of public key cryptosystems (PKCs), with emphasis on RSA and ECC were presented. The elliptic curve theory was also introduced to identify the most time critical and complex computational blocks in PKCs.

We have identified modular multiplication or finite field multiplication as the backbone of modern PKCs through this review. The timing, area and power dissipation of the entire cryptosystem is largely dependent on this operation as it is used repeatedly to perform various higher level operations during encryption and decryption. In order to better understand the advantages and disadvantages of the various modular multiplication techniques, three dominant classes of modular multiplication algorithms were rigorously studied. Montgomery modular multiplication in $GF(N)$ and $GF(2^m)$ was discussed in detail followed by the LSB-first and MSB-first algorithms in $GF(2^m)$. An extensive survey of the available LSB-first and MSB-first algorithms is also presented along with their standard cell implementations followed by an evaluation of VLSI metrics.

Through this comprehensive survey, we envision the room for performance enhancement in the present modular multiplication methods. In the subsequent chapters of this thesis, incremental refinements for improvements, especially in terms of timing, will be proposed for the Montgomery and the LSB-first/MSB-first modular multiplication algorithms. The issues and limitations of existing methods will be detailed in the forthcoming chapters and solutions will be proposed to address them.

In the next chapter, we will look into one of the most commonly used modular multiplier - Montgomery modular multiplier. Modified algorithms that operate in $GF(N)$ for prime N are proposed and translated into efficient array architectures. The proposed architectures will be evaluated against existing Montgomery multipliers on FPGA and ASIC platforms.

Chapter 3

New Systolic Architectures for Montgomery Modular Multiplication

3.1 Introduction

In the last chapter, public key cryptosytems (PKCs) and the mathematics involved in different PKCs like RSA, ECC etc. were discussed. Finite field multiplication was identified as the most critical computation engine due to its area cost and computation time. This operation involves modular multiplication in the $GF(N)$ and $GF(2^m)$ fields and is repeatedly used in exponentiation and point multiplication algorithms in PKCs.

Among the different kinds of multipliers surveyed in Chapter 2, Montgomery modular multiplier is the most frequently used algorithm in many cryptosystems. The use of Montgomery modular multiplication (MMM) in the two fields - $GF(N)$ and $GF(2^m)$ was also presented. In general, RSA cryptosystems employ MMM in only $GF(N)$ and ECCs employ MMM in both $GF(N)$ and $GF(2^m)$ fields.

In this chapter, two new modified systolic arrays to implement Montgomery modular multiplication (MMM) in $GF(N)$ are presented. A large portion of this chapter has been published in *WSEAS Transaction on Circuits and Systems* [47] and presented in *Proc. 5th WSEAS Intl. Conf. on Instrumentation, Measurement, Circuits and Systems (IMCAS 2006)* [46]. To make this chapter self-contained, Section 3.2 revisits MMM in $GF(N)$. Different kinds of bit-level MMM algorithms are studied and analyzed. The problem statement is framed in Section 3.3. The proposed MMM algorithms are derived and listed in Section 3.4 followed by their architectural translations in Section 3.5. The proposed architectures are then evaluated for VLSI metrics against some recently reported Montgomery multipliers in Section 3.6.

3.2 Montgomery Modular Multiplication Revisited

MMM algorithm in its primitive form is listed in [5,6,10,11,21,32,34,35,39]. Firstly some notations are presented, that are used in this chapter. For consistency, a binary variable name is written in lower case letters. A vector of binary variables is represented with a variable name with the first letter in upper case. A binary variable, x at the i-th row and the j-th column in a systolic array is denoted by $x_j^{(i)}$. Vector addition in carry-save representation is commonly used in this chapter. A tuple (C, S) that is used to represent the carry and sum vectors, resulting from vector addition, implies the summation $2C + S$.

The input to output mapping of MMM is given by

$$MMM(A, B, N) = ABR^{-1} \ mod \ N \tag{3.1}$$

where R and N are relatively prime. If N is odd, as is the case in RSA, R can be the even number 2^k, k being the length of the cryptographic key. Let

$$A = \sum_{i=0}^{k-1} a_i 2^i \tag{3.2}$$

$$B = \sum_{i=0}^{k-1} b_i 2^i \tag{3.3}$$

$$N = \sum_{i=0}^{k-1} n_i 2^i \tag{3.4}$$

where $a_i, b_i, n_i \in \{0,1\}$. Then MMM in its primitive form is described by Algorithm 7 [39]. The original MMM algorithm by P. L. Montgomery in [32] involves a subtraction at the

Algorithm 7 Montgomery Modular Multiplication - MMM_orig

```
1: Input: $A = \sum_{i=0}^{k-1} a_i 2^i, B = \sum_{i=0}^{k-1} b_i 2^i, N = \sum_{i=0}^{k-1} n_i 2^i$
2: Output : $S = \sum_{i=0}^{k-1} s_i 2^i$
3: $S \leftarrow 0$
4: for $i = 0$ to $k-1$ do
5:     $q \leftarrow (s_0 + a_i b_0)$ $mod$ $2$
6:     $S \leftarrow (S + a_i B + qN)$
7:     $S \leftarrow S/2$
8: end for
9: Return $S$
```

end but it can be eliminated as shown in Algorithm 7 [10,11,39]. In Line 5 of Algorithm 7 - MMM_orig, the *mod* operation on the sum $s_0 + a_i b_0$ results in a bitwise XOR operation without any carry propagation. However, in Line 6, a long carry chain is formed from the addition of k-bit vectors - S, $a_i B$ and qN. Thus, a straight forward implementation of Algorithm 7 induces a long critical path [6,39].

In order to remove the long carry chains in Algorithm 7, Montgomery modular multiplication in carry-save representation (MMM_CS) is proposed in [10]. This is listed in Algorithm 8.

In each iteration of Algorithm 8 - MMM_CS, besides the sum signal - S, two carry signals - $C1$ and $C2$, are generated to compute the vector additions in carry-save representation. Also, an additional stage of adders is required to add $C2$, $C1$ and S in MMM_CS. With the carry-save format, the carry chain in Line 6 of Algorithm 7 is saved for the next

Algorithm 8 Carry Save Montgomery Modular Multiplication - MMM_CS

```
1: Input: $A = \sum_{i=0}^{k-1} a_i 2^i, B = \sum_{i=0}^{k-1} b_i 2^i, N = \sum_{i=0}^{k-1} n_i 2^i$
2: Output : $C2 = \sum_{i=0}^{k-1} c2_i 2^i, C1 = \sum_{i=0}^{k-1} c1_i 2^i, S = \sum_{i=0}^{k-1} s_i 2^i$
3: $C1 = 0, C2 = 0, S = 0$
4: for $i = 0$ to $k - 1$ do
5:    $q \leftarrow (s_0 + c1_0 + c2_0 + a_i b_0)\ mod\ 2$
6:    $(C2, C1, S) \leftarrow C2 + C1 + S + a_i B + qN$
7:    $C2 \leftarrow C2/2, C1 \leftarrow C1/2, S \leftarrow S/2$
8: end for
9: Return $C2, C1, S$
```

stage instead of being propagated in the current stage in Algorithm 8.

Algorithm 9 Montgomery Modular Multiplication with Precomputation - MMM_OP

```
1: Input: $A = \sum_{i=0}^{k-1} a_i 2^i, B = \sum_{i=0}^{k-1} b_i 2^i, N = \sum_{i=0}^{k-1} n_i 2^i$
2: Output : $C = \sum_{i=0}^{k-1} c_i 2^i, S = \sum_{i=0}^{k-1} s_i 2^i$
3: $C = 0, S = 0$
4: for $i = 0$ to $k - 1$ do
5:    $q \leftarrow (s_0 + c1_0 + a_i b_0)\ mod\ 2$
6:    if $a_i = 0, q = 0$ then
7:       $I \leftarrow 0$
8:    else if $a_i = 0, q = 1$ then
9:       $I \leftarrow N$
10:   else if $a_i = 1, q = 0$ then
11:      $I \leftarrow B$
12:   else if $a_i = 1, q = 1$ then
13:      $I \leftarrow B + N$
14:   end if
15:   $(C, S) \leftarrow C + S + I$
16:   $C \leftarrow C/2, S \leftarrow S/2$
17: end for
18: Return $C, S$
```

A further modification to MMM_CS is proposed in [11, 34], where the sum - $B + N$ (in Line 6 of Algorithm 8) is precomputed. In this algorithm, a_i and q are used to select either B or N or $B + N$ or 0. So, $B + N$ should be precomputed and stored in a buffer. Algorithm 9 from [11, 34] illustrates this idea. The selective assignment clauses in Lines 6 to 14 of Algorithm 9 are implemented using a 4-to-1 multiplexer with select signals - a_i and q. The output I from the multiplexer is then added to the sum and carry signals from the previous iteration. The precomputation of $B + N$ reduces the number of carry vectors to one as opposed to Algorithm 8. This method however has an initial latency due to the precomputation.

3.3 Problem Statement

On studying the three algorithms, following observations can be made. Algorithm 7, MMM_orig, suffers from long carry chains. In Algorithm 8, MMM_CS, carry-save representation is used to remove the long critical paths due to the carry propagation in MMM_orig. There are however, three vectors at the end, two carrys and one sum vectors which need to be added using another set of adders. The number of carry vectors are reduced in Algorithm 9, MMM_OP, by precomputing $B + N$. Moreover, in MMM_OP, one addition operation is eliminated by the precomputation but an additional 4-to-1 multiplexer is introduced in every processing element.

However, in all the algorithms, the main limiting factor in terms of timing is their dependency on the intermediate signal q. In all the three algorithms, in each iteration, q is computed using the least significant bit (lsb) of sum, carry and input B (Line 5 in MMM_CS and MMM_OP). In MMM_CS, q is then used to determine the carry and sum signals in Line 6. Similarly, in MMM_OP, q is used to select I in Lines 6 to 14, which is later used for the computation of carry and sum in Line 15.

The critical path of the primitive MMM_orig depends on the carry chain. For the rest of the chapter, only MMM_CS and MMM_OP are studied and evaluated because the critical path of the multiplier does not depend on the carry chain.

MMM is generally implemented as systolic arrays [5, 6, 10, 11, 32, 34, 35, 39]. In a two-dimensional systolic array, a row of basic computing cells implements one iteration in the algorithms. A basic cell in the j-th column and the i-th row computes the sum and carry, $(c_j^{(i)}, s_j^{(i)})$, in each iteration. In the implementation of MMM_CS [10,39] and MMM_OP [11], the basic cell at the least significant position computes q. This signal is then propagated to the remaining basic cells in each row. The maximum operating frequency of the architecture is limited by the critical path which results from the dependency on q. As a result the critical path that passes through the basic cell in the lsb position comprises 3 two-input XOR gates in MMM_CS and 2 two-input XOR gates and 1 4-to-1 multiplexer in MMM_OP. In addition, MMM_OP requires the buffering of the precomputed result, $B+N$. This step will take up additional clock cycles between successive modular multiplication operations and hence the throughput rate is affected when the multiplier is used to perform exponentiation.

In the next section, a modified Montgomery modular multiplication algorithm - MMM_MX, is proposed. The dependency on q is resolved by introducing an additional clock cycle. Breaking the dependency not only reduce the critical path but facilitates hardware reuse also. Carry-save representation is used to avoid long carry chains. Also, no precomputation is necessary in the proposed architecture. Therefore, the extra clock cycles needed between successive multiplications by using MMM_OP have been eliminated. In addition, the number of carry vectors will be reduced from two (in MMM_CS) to one in the proposed algorithm.

3.4 Proposed Modified MMM

In this section, the modified Montgomery modular multiplication algorithm - MMM_MX is derived. First, Lines 5 to 7 of Algorithm 7 - MMM_orig are listed below for analysis

```
Listing 1:
Line 5:  $q \leftarrow (s_0 + a_i b_0)\ mod\ 2$
Line 6:  $S \leftarrow (S + a_i B + qN)$
Line 7:  $S \leftarrow S/2$
```

In `Listing 1`, q is computed in Line 5 by the modulo 2 summation of the lsb of S and $a_i b_0$. The value is then used in Line 6 to compute S. These equations can be rewritten as follows:

```
Listing 2:
Line 5a:  $(Ct, St) \leftarrow C + S + a_i B$
Line 5b:  $q \leftarrow st_0$
Line 6:   $(C, S) \leftarrow Ct + St + qN$
Line 7:   $C \leftarrow C/2, S \leftarrow S/2$
```

The above steps are written in carry-save representation. The tuple (C, S) in the above listing refers to the carry-save representation $(C, S) = 2C + S$. In `Listing 2`, the addition in Line 6 of `Listing 1` is split into two parts. In Line 5a, C, S and $a_i B$ are added to produce temporary carry, Ct and sum St. In `Listing 1`, the sum of s_0 and $a_i b_0$ is assigned to q. So the lsb of sum, st_0 obtained from Line 5a of `Listing 2`, can be assigned to q in Line 5b. With the value of q in hand, Ct and St are then added to qN in Line 6 to give C and S. The resulting carry and sum are then right shifted in Line 7.

In `Listing 2`, the following points are observed.

- If a register is introduced between Lines 5 and 6, q can be simply obtained from the lsb of the intermediate signal, St. This q can then be used for the calculation of the sum, S and carry, C in the next cycle.
- Moreover, Lines 5a and 6 perform the same operation 'addition', but with different

operands. Thus the adder that computes Ct and St in Line 5a can be reused to compute C and S in Line 6. Thus a simple 2-to-1 multiplexing of inputs to the adder will enable the same adder to be used in both cycles - Lines 5a and 6.

The above observations result in a different Montgomery modular multiplication algorithm - MMM_MX as shown in Algorithm 10. The proposed algorithm has two for loops (with loop counters, i and j) for the bit-level scanning of inputs. These loops are present in MMM_CS and MMM_OP also. MMM_MX has one extra loop (with loop counter $ctrl$). This additional loop implements the control signal to multiplex the correct inputs to the adder that is reused in Lines 5a and 6 in `Listing 2`. In Algorithm 10, the inputs to the multiplexer - $in1$, $in2$, $in3$ and $in4$, are selected by $ctrl$ in Lines 5 to 7. These inputs are then added using a gated full adder (GFA) in Line 8 to produce the logical sum and carry of the expression $in1.in2 + in3 + in4$ according to the inputs selected by $ctrl$. When $ctrl$ = '0', the intermediate carry and sum, Ct and St, are computed from the inputs, a_i and B, the sum, S and the carry, C from the previous iteration. At the end of the first clock cycle, $q = st_0$ is obtained. In the next clock cycle, when $ctrl$ = '1' the full adder is used to compute C and S for the current iteration by using q, N and the intermediate sum and carry signals, St and Ct. The control signal then returns to 0 and the process continues until all bits of the input operand A have been exhausted.

3.5 Architectures

The systolic architecture corresponding to the conventional carry-save implementation, described by Algorithm 8, MMM_CS, is shown in [10] and those based on the precomputation of $B + N$ in MMM_OP, are implemented in [10, 11, 34]. This section describes the algorithm to architecture translation of the proposed MMM_MX algorithm. MMM_MX is implemented as two systolic architectures. The first one is a pipelined two-dimensional parallel array architecture. The second one is a serial-in parallel-out one-dimensional array

Algorithm 10 Montgomery Modular Multiplication with Multiplexing - MMM_MX

```
1: Input: $A = \sum_{i=0}^{k-1} a_i 2^i, B = \sum_{i=0}^{k-1} b_i 2^i, N = \sum_{i=0}^{k-1} n_i 2^i$
2: Output : $C = \sum_{i=0}^{k-1} c_i 2^i, S = \sum_{i=0}^{k-1} s_i 2^i$
3: $C \leftarrow 0, S \leftarrow 0, St \leftarrow 0, Ct \leftarrow 0, q \leftarrow 0$
4: for $i = 0$ to $k-1$ do
5:     for $ctrl = 0$ to $1$ do
6:         for $i = 0$ to $k-1$ do
7:             if $ctrl = 0$ then
8:                 $in1 \leftarrow a_i$; $in2 \leftarrow b_j$; $in3 \leftarrow s_j$; $in4 \leftarrow c_j$
9:             else
10:                $in1 \leftarrow q$; $in2 \leftarrow n_j$; $in3 \leftarrow st_j$; $in4 \leftarrow ct_j$
11:            end if
12:            $carry + sum \leftarrow GFA(in1, in2, in3, in4)$
13:        end for
14:        $Ct \leftarrow Carry$
15:        $St \leftarrow Sum$
16:        $q \leftarrow st_0$
17:    end for
18:    $C \leftarrow Ct/2$
19:    $S \leftarrow St/2$
20: end for
21: Return $C, S$
```

architecture.

Basic Computing Cell

The basic cell or computing element that is used in the systolic array is shown in Fig. 3.1. It has four single bit 2-to-1 multiplexers. As shown in Algorithm 10, the control signal *ctrl*

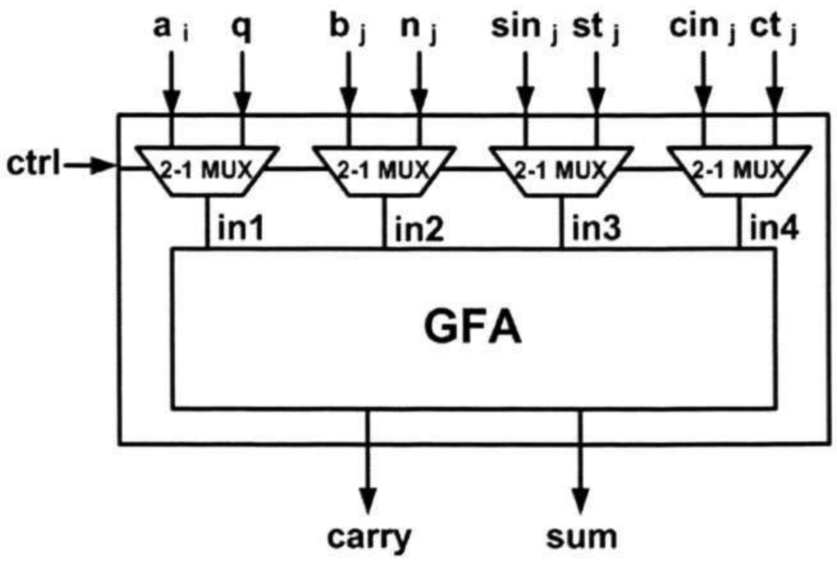


Figure 3.1: Basic cell (BC) or computation unit

selects between the two groups of four input signals - a_i, b_j, s_j, c_j or q, n_j, st_j, ct_j. The four signals $in1$, $in2$, $in3$ and $in4$ are then fed to the gated full adder (GFA). In GFA, the following summation is performed.

$$(carry, sum) \leftarrow in1 \cdot in2 + in3 + in4 \tag{3.5}$$

Two-dimensional Pipelined Parallel Architecture

The two-dimensional parallel systolic implementation 'Design 1' of MMM_MX is shown in Fig. 3.2. It comprises k^2 replicas of the basic cell in a 2-D array. The outputs of all basic cells are registered using flip-flops denoted by • in Fig. 3.2.

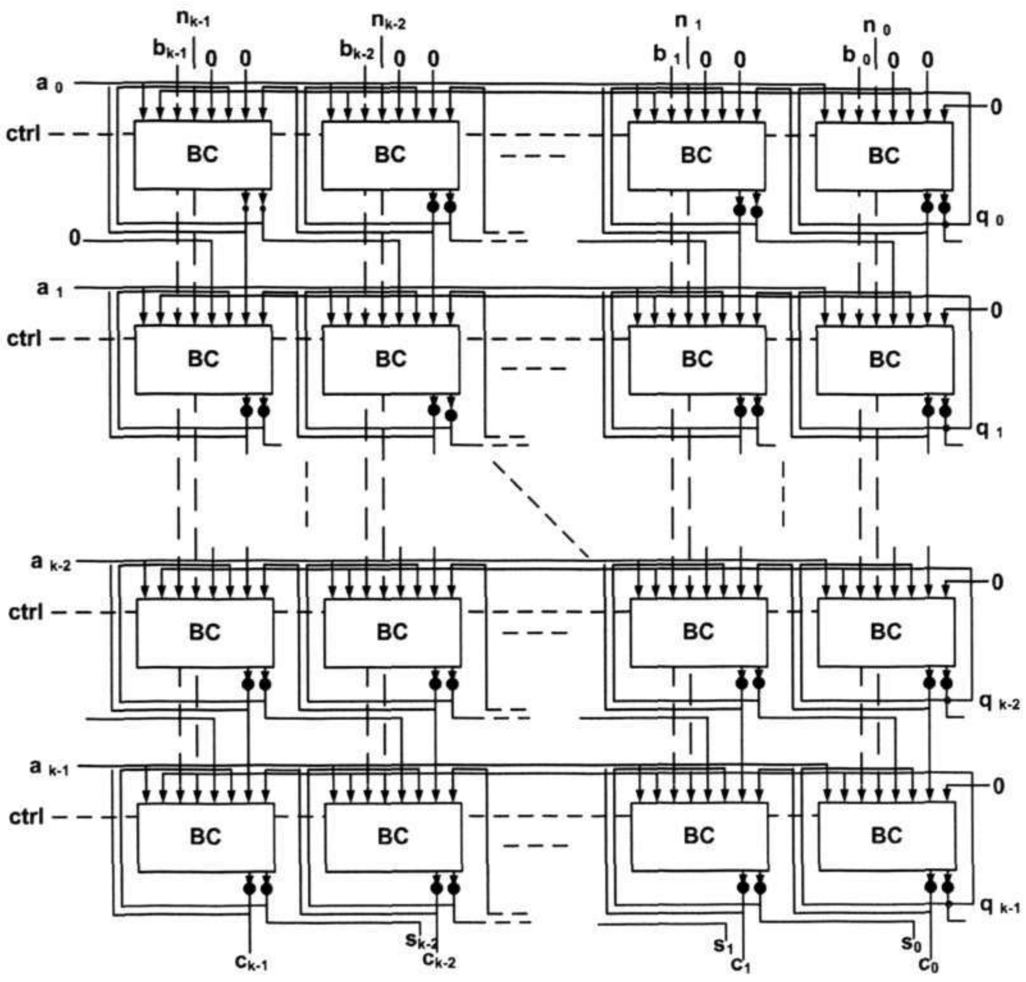


Figure 3.2: Two dimensional pipelined systolic implementation - Design 1

The assignment of intermediate vectors S, C, St and Ct in each row of the array needs some explanation. Let Sum and $Carry$ denote the k-bit sum and carry outputs resulting from the basic cells. The sum output of the basic cell in the i-th row and the j-th column, $sum_j^{(i)}$, is connected to $st_j^{(i)}$ and $s_{j-1}^{(i+1)}$. Similarly the carry output of the basic cell, $carry_j^{(i)}$ is connected to $ct_{j+1}^{(i)}$ and $cin_j^{(i+1)}$. Fig. 3.3(a) illustrates this assignment of Sum and $Carry$ signals to the GFA at the i-th row. In the m-th clock cycle, when $ctrl = 0$, the right shifted Sum and $Carry$ signals from the $(i-1)$-th row, i.e., $sin_{j+1}^{(i-1)}$ and $cin_j^{(i-1)}$ are multiplexed into the GFA (Line 8 of Algorithm 10). According to Line 10 of Algorithm 10, in the next clock cycle, the inputs of GFA should be assigned to the sum and carry signals that were generated by the basic cells in the i-th row. Therefore, in the $(m+1)$-th cycle, *Sum* and *Carry* signals, $st_j^{(i)}$ and $ct_{j+1}^{(i)}$ of the basic cells in the same row are selected by the control signal that has just been toggled. Fig. 3.3(a) shows the *Sum* and *Carry* signals in the m-th and $(m+1)$-th clock cycles. The *Sum* and *Carry* that are generated by the $(i-1)$-th row in the $(m-1)$-th cycle are right shifted before it is assigned to the i-th row in the m-th clock cycle. The subscripts of *Sum* and *Carry* signals generated in the i-th row of basic cells remain in the $(m+1)$-th cycle indicating that they have not been right shifted at this juncture.

k rows of basic cells are connected as illustrated above to form a two-dimensional pipelined systolic array. The reservation table of the pipeline is shown in Fig. 3.3(b). Although the total time taken to generate the first S and C outputs is $2k$ clock cycles, the latency after filling up the pipeline is merely two clock cycles.

k-1	k-2		j		2	1	0	Clock Cycle
0 $C_{k-1}^{(i-1)}$	$S_{k-1}^{(i-1)}$ $C_{k-2}^{(i-1)}$	....	$S_{j+1}^{(i-1)}$ $C_j^{(i-1)}$	....	$S_3^{(i-1)}$ $C_2^{(i-1)}$	$S_2^{(i-1)}$ $C_1^{(i-1)}$	$S_1^{(i-1)}$ $C_0^{(i-1)}$	m
$S_{k-1}^{(i)}$ $C_{k-2}^{(i)}$	$S_{k-2}^{(i)}$ $C_{k-3}^{(i)}$	....	$S_j^{(i)}$ $C_{j+1}^{(i)}$	....	$S_2^{(i)}$ $C_1^{(i)}$	$S_1^{(i)}$ $C_0^{(i)}$	$S_0^{(i)}$ 0	m+1

(a)

	Clock Cycle												
Stage	0	1	2	3	4	5	6	7	....	2k-5	2k-4	2k-3	2k-2
0	0	0	1	1	2	2	3	3	....				
1			0	0	1	1	2	2	....				
2					0	0	1	1	....				
									....				
k-2									....	0	0	1	1
k-1									....			0	0

(b)

Figure 3.3: a) Indices of Sum S and Carry C of the i-th row of basic cells in the m-th and $(m+1)$-th clock cycles (b) Reservation table of the pipeline of the 2-D architecture

One-dimensional Variant

The two-dimensional systolic array is fast in terms of latency but it involves large area overhead. To reduce the area cost, a bit-serial variant of the two-dimensional systolic architecture - 'Design 2' is proposed. The architecture is shown in Fig. 3.4. This is a serial-in parallel-out multiplier. It involves only one row of basic cells. Input a_i is sent serially in alternate clock cycles into this row of basic cells. Inputs B and N are applied in a parallel fashion. Since each a_i takes 2 clock cycles, Design 2 would require $2k$ clock cycles to compute one Montgomery multiplication result. The chip area however, is reduced substantially at the expense of a lower throughput rate and higher latency.

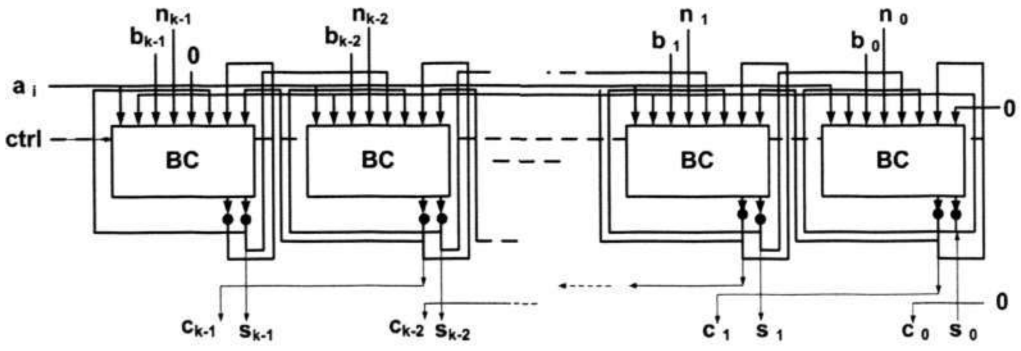


Figure 3.4: One-dimensional systolic implementation - Design 2

3.6 Hardware Implementation Results

MMM_MX 'Design 1' and 'Design 2' are evaluated against recently reported Montgomery modular multipliers in [6,10,11,39] for maximum clocking frequency, latency, throughput and area. Design 1 and Design 2 are compared against different kinds of architectures, with and without precomputation of $B+N$, carry-save/normal representations and FPGA specific implementations, reported in [6,10,11,39]. The MMM architectures in [6,11,39] are one-dimensional arrays whereas those in [10] are two-dimensional systolic architectures. [10] and [11] employ carry-save representation but [6] and [39] are based on carry-propagate adders. Unlike the 'Conventional' carry-save architecture in [10], the 'Optimized' architecture in [10] is a two-dimensional systolic array that is based on the precomputation of $B + N$. The more recent architecture in [11] is a one-dimensional variant of 'Optimized' architecture in [10]. Fournaris et al. [11] also compared some the latest reported Montgomery modular multiplier architectures with their proposed one-dimensional precomputation based architecture. We compare the implementation of our architectures against the results reported in [11].

The comparisons are based on both FPGA and ASIC implementation. Since most implementations were originally implemented on FPGA platform, the proposed designs are first implemented on Xilinx Virtex 2 chip and synthesized using Xilinx Synthesis Tool (XST). They are then evaluated against the fastest implementation - 'Optimized' [11], in terms of ASIC performance metrics - silicon area, critical path delay and area-time product. Both the architectures were implemented using Synopsys Design Compiler with TSMC 0.18μm standard cell library. The clock period constraint was set to a typical value of 50 ns.

The designs are first qualitatively analyzed followed by a quantitative evaluation of their hardware implementation using FPGA and ASIC design flows.

3.6.1 Qualitative Results

Table 3.1 lists the time complexity of different multipliers. The critical path delay, t_D is expressed in terms of the delay of primitive logic modules, such as full/half adders (Δ_{FA}/Δ_{HA}), 2-to-1/4-to-1 multiplexers ($\Delta_{2MUX}/\Delta_{4MUX}$) and 2-input XOR/AND gates ($\Delta_{XOR}/\Delta_{AND}$). The critical path delay t_D determines the maximum clock frequency of the architecture. The total number of clock cycles required to complete one multiplication operation (t_{total}) and the latency cycles (L) of each design are expressed as a factor of the input wordlength, k.

Table 3.1: Qualitative Timing Analysis

Two-dimensional Arrays			
Architecture	t_D (ns)	t_{total} (cycles)	L (cycles)
Design 1	$1\Delta_{2MUX} + 1\Delta_{FA}$	$2k + k_1$	$2 + k_1'$
Optimized [10]	$1\Delta_{4MUX} + 1\Delta_{FA} + 2\Delta_{XOR}$	$k + m + k_1$	$1 + m' + k_1'$
Conventional [10]	$2\Delta_{FA} + 1\Delta_{HA}$	$k + k_1$	$1 + 2k_1'$
One-dimensional Arrays			
Architecture	t_D (ns)	t_{total} (cycles)	L (cycles)
Design 2	$1\Delta_{2MUX} + 1\Delta_{FA}$	$2k + k_1$	$2k + k_1'$
Optimized [11]	$1\Delta_{4MUX} + 2\Delta_{FA} + 2\Delta_{XOR}$	$k + m + k_1$	$k + m' + k_1'$
Daly [6]	FPGA specific carry chain	$k + 3 + j - 1$	$k + 3 + j - 1$
Ors [39]	$2\Delta_{FA} + 1\Delta_{HA} + 1\Delta_{AND}$	$3k + 4$	$2k + 1$

In Table 3.1, the results of Designs 1 and 2, and the designs 'Optimized' and 'Conventional' from [10] and [11] are produced in redundant carry-save format. k_1 is the number of additional clock cycles needed for the final CPA to convert them into normal binary form. The architecture 'Optimized' in both [10] and [11] involves the precomputation of $B + N$ and hence another m clock cycles are incurred. The latency of k_1 and m are each equal to k full adder delays if a pipelined 1-bit ripple carry adder is used for these additions. The architecture described in [6] uses FPGA specific carry chain architecture. The parameter j is defined as $j = k/p$, where p is the maximum length of the carry chain for the FPGA device. k_1' and m' in the latency column correspond to the additional clocks cycles required for the final CPA and precomputation, respectively.

The expressions in Table 3.1 show that Designs 1 and 2 outperform all other designs in terms of critical path delay. The critical path delay is reduced as a result of breaking up the computation of q from the calculation of the sum and carry signals. Lower critical path delay enables the circuit to operate at higher clock speed. The advantage of the pipelined 2-D Design 1 is seen in its low latency cycles. Latency of Design 2 is $2k + k'_1$ clock cycles but it consumes a smaller area than Design 1 as shown in Table 3.2.

Table 3.2: Qualitative Area Analysis

Two-dimensional Arrays		
Architecture	Basic Cell	Number of cells
Design 1	PE: 4 2-to-1 MUXs, 1 FA, 1 AND	k^2
Optimized [10]	PE: 1 4to1 MUX, 1 FA qPE: 1 4to1 MUX, 1 FA, 2 XOR, 1 AND Precomputation CPA	$(k-1)^2$ k 1
Conventional [10]	PE: 2 FA, 1 HA	k^2
One-dimensional Arrays		
Architecture	Basic Cell	Number of cells
Design 2	PE: 4 2-to-1 MUXs, 1FA, 1 AND	k
Optimzed [11]	PE: 1 4to1 MUX, 1 FA qPE: 1 4to1 MUX, 1 FA 2 XOR, 1 AND Precomputation CPA	k 1
Daly [6]	FPGA specific CPA	-
Ors [39]	PE: 2 FA, 1HA	$k-2$

3.6.2 FPGA Implementation Results

The FPGA implementation results of the proposed one-dimensional architecture - Design 2, are compared against the one-dimensional arrays of [6, 11, 39] in Table 3.3. Xilinx

Synthesis tools are used for synthesizing the proposed architectures and for architectures in [10] and [11] on Xilinx Virtex 2 chip. The results for the architectures of [39] and [6] are taken directly from the papers wherein the architectures are implemented on Virtex family FPGAs - Virtex E and Virtex chips, respectively.

All the different architectures are implemented on Virtex family of Xilinx FPGAs. In the Virtex family [53], the FPGA is divided into configurable logic blocks (CLBs). Each CLB is further divided into slices. Each CLB in Virtex 2 and Virtex E contain 4 slices. In Virtex chip, each CLB comprises 2 slices. So the basic logic block is a slice. Each slice is composed on arithmetic logic units, memory elements (ROM and RAM), shift registers and multiplexers.

The proposed architectures are implemented on the same FPGA platform as [10] and [11]. However, the architectures in [39] and [6] are implemented on different FPGA chips. Though these architectures from [39] and [6] were implemented on different FPGAs, the FPGA chips belong to the same family and have similar structural composition as Virtex 2 chips. Timing is generally more susceptible to different FPGA technologies. However the architectures of [6] and [39] employ long carry propagation chains which almost always have poorer timing than the carry-save architectures of [10,11] and our proposed architectures.

Table 3.3 compares the proposed 1024-bit one-dimensional Design 2 multiplier against 1024-bit one-dimensional array architectures - 'Optimized' in [11] and carry-propagation adder based designs in [39] and [6]. The implementation results of the proposed architecture are coherent with the qualitative results in Table 3.1 and 3.2. When a one-bit basic cell of the proposed architecture was implemented on FPGA, it was found to cover 3 slices. From Table 3.2, the area of a k-bit multiplier is proportional to k. Hence a 1024-bit multiplier should cover about $3 \times 1024 = 3072$ slices. From Table 3.3, it can be seen that the area result from the implementation of 1024-bit multiplier is 2947 slices which is close to the theoretical estimate. Moreover, the delay does not depend on k because the critical path passes through one stage of basic cells.

From Table 3.3, it is observed that the proposed Design 2 is three times faster than the recently reported 'Optimized' [11]. The throughput of the proposed architecture is 1.5 times that of 'Optimized' [11]. The speed up is because of the shorter critical path in the proposed architecture as compared to 'Optimized' in [11]. The critical path passes through one 2-to-1 multiplexer and a gated full adder in the proposed architecture as against a 4-to-1 multiplexer followed by a full adder in the 'Optimized' [11]. Thus, even though the proposed architecture takes one extra clock cycle in every iteration to compute the Montgomery product, due to its shorter critical path, it is still faster than all the one-dimensional architectures shown in Table 3.3. As a result of this, the total computation time is reduced and the reduction in the total computation time is prominent when it is used for modular exponentiation, where the operands are multiplied repetitively.

In addition to timing, the proposed architectures show an area savings of about 18% over 'Optimized' in [11]. This is attributed to lower cell complexity of the proposed architecture which does not employ complex 4-to-1 multiplexers. The architectures in [6] and [39] are computationally more complex than proposed architecture because they employ carry-propagation adders at every stage of the algorithm.

The two-dimensional systolic architectures were implemented for a bit-length of 128. The FPGA implementation results of the proposed 2-D arrays - Design 1, are compared against the two-dimensional 'Optimized' and 'Conventional' architectures (from [10]) in Table 3.4.

Table 3.3: FPGA implementation results of 1024-bit one-dimensional Montgomery modular multipliers

Architecture	Chip Area (slices)	Clock Frequency (MHz) (MHz)	Throughput (bit/sec)
Design 2	2947	386.84	193.42 M
Optimized [11]	3611	129.10	129.00 M
Daly [6]	5458	54.61	54.40 M
Ors [39]	5706	95.62	31.83 M

Table 3.4: FPGA implementation results of 128-bit two-dimensional Montgomery modular multipliers

Architecture	Chip Area (slices)	Clock Frequency (MHz) (MHz)
Design 1	44330	358.17
Optimized [10]	48767	168.70
Conventional [10]	65473	156.30

In the 2-D arrays, the proposed multiplier also outperforms its counterparts in both aspects - the area and the maximum clock frequency. This can be validated using the qualitative results in Tables 3.1 and 3.2. From Table 3.1, the critical path of the proposed multiplier comprises one 2-to-1 multiplexer and one gated full adder where as 'Optimized' [10] has a minimum delay of one 4-to-1 multiplexer followed by 1 full adder and 2 XOR gates. The 'Conventional' architecture in [10] has delay of 2 full adders and one half adder. Hence the gain in operating frequency of the proposed architecture can be explained by the shorter critical path.

In terms of area, one-bit cell of the proposed architecture occupies 3 slices. From the qualitative results in Table 3.2, it can be seen that the proposed k-bit two-dimensional architecture would occupy an area equivalent to k^2 cells. So for a 128-bit multiplier, total area is equal to $128^2 \times 3 = 49152$ slices. This theoretical value is close to the actual implementation value of 44330 slices. The lower value could be accounted for by the optimization done by the tool. Moreover, the proposed two-dimensional architecture outperforms the other designs in Table 3.4 because of the simpler cell complexity of the proposed design as shown in Table 3.2.

3.6.3 ASIC Implementation Results

Since Design 1 is a two-dimensional architecture, the one-dimensional Design 2 would show a clear advantage over Design 1 in terms of area. The critical path delay depends on the basic cell complexity which is the same for both. Hence only Design 2 is used to illustrate the advantage of the proposed solution on an ASIC platform. Design 2 is evaluated against 'Optimized' [11] for post synthesis ASIC performance in Table 3.5. Synopsys Design Compiler v2004.12-SP2 was used to synthesize all the circuits. SAGE-X TSMC $0.18\mu m$ standard cell library, which uses 1.8V supply, was employed for synthesis. All designs were optimized for a constraint set on clock period for 50 ns. The implementation of 'Optimized' does not comprise the precomputation carry-propagation adder. Design 2 clearly outperforms 'Optimized' by 2.5 times in terms of the minimum clock period necessary to register the inputs and outputs. If the precomputation CPA in 'Optimized' is excluded, Design 2 shows 27% higher area but the 1024-bit CPA in 'Optimized' is a crucial component that provides one of the inputs to the computation core. This CPA, if implemented with a simple one-bit ripple-carry adder, incurs minimal area overhead but the latency is extended by 1024 clock cycles On the other hand, using a CLA for this purpose would increase the hardware complexity. If the area-time (AT) product is considered, the ascendancy of Design 2 over 'Optimized' [11] is evident. The area-time product of Design 2 is only half that of 'Optimized'.

Table 3.5: ASIC implementation results of 1024-bit one-dimensional Montgomery modular multipliers

Architecture	Silicon Area (μm^2)	Minimum Clock Period (ns)	Area $\times$ Time) ($\mu m^2 \times ns$)
Design 2	954184.50	1.84	1755699.48
Optimized* [11]	751151.06	4.76	3575479.06

* Does not include precomputation CPA

3.7 Summary

In this chapter, a modified Montgomery modular multiplication algorithm was proposed which reduces the critical path of the operation by removing its dependency on an intermediate signal. The algorithm does not require any precomputation and it operates in carry-save format. The proposed algorithm - MMM_MX was mapped to a systolic array architecture. A one-dimensional variant of the 2-D systolic architecture was also proposed to trade throughput for area. The two architectures were qualitatively and quantitatively evaluated against several recently reported Montgomery modular implementations. The proposed algorithm reduces the minimum clock period by three times when compared to one of the fastest algorithms when it is mapped onto FPGA and ASIC platforms. When prototyped on FPGA, the area-time analysis shows that the proposed designs perform well in terms of both critical path delay and resource utilization. ASIC implementation of the one-dimensional architecture shows 50% improvement in area-time product as compared to the fastest one-dimensional architecture available.

The proposed architectures are fast and area-efficient. However, the proposed architectures are designed to operate in $GF(N)$ only. It is worthwhile to study the flexibility to scale a Montgomery multiplier architecture for a parameterizable moduli N. In the following chapter, a new kernel for unifying Montgomery modular multiplications in both

$GF(N)$ and $GF(2^m)$ is proposed. It will also address issues that impede architectural scalability.

Chapter 4

Unified Montgomery Modular Multiplication

4.1 Introduction

The wordlength of the modulus in RSA and the order of the reduction polynomial in an elliptic curve cryptosystem (ECC) determine the strength of these public key cryptosystems (PKCs). The security strength of applications varies according to the needs and values of the secrets to be protected. For example, the cryptoprocessors used in smart cards need high security strength but a login card in a low secure area may not need that tight security. So the flexibility to vary field orders of the cryptoprocessor is a necessary. Additionally, some applications may run different cryptosystems at different times. It is also desirable to adapt a cryptosystem to the constraints. For example an ECC with 571-bit keylength gives a security strength equivalent to RSA cryptosystem with a 15360-bit keylength [3].

In many situations, changing the hardware accelerators for the cryptosystems to address the above scenarios can be costly and it is not always feasible. Hence, there is a need to develop scalable and flexible architectures that possess the following features:

- Varying fields: RSA cryptosystem operates in the prime field $GF(N)$ and ECC operates in both $GF(N)$ and the extended binary field $GF(2^m)$. Thus a cryptoprocessor that has the agility to switch among fields can be used versatilely in different cryptosystems.

- Varying field orders or moduli: Changing the field order (in ECC) or moduli (in ECC and RSA cryptosystems) varies the security strength. Lower field orders or moduli permit faster processing with lower power consumption at the cost of security strength.

In Chapter 3, modified Montgomery modular multipliers for RSA cryptosystem were discussed. The algorithm and architectures, in Chapter 3, are designed for a fixed modulus N. As a result they are inflexible to variable moduli. Moreover they cannot cater to ECCs that operate in $GF(2^m)$. In this chapter, a new and fast processing unit is proposed that can be used to unify Montgomery modular multiplication (MMM) in two finite fields - the prime field, $GF(N)$, for prime N, and the extended binary field, $GF(2^m)$. A new pipelined and scalable processing kernel is generated using the new processing unit.

A part of this chapter has been presented in the *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006)*. This chapter is organized as follows. In Section 4.2 Montgomery modular multiplication in the two fields is revisited. In Section 4.3 the problem statement is formulated after studying the issues with the existing unified architecture in [48]. The proposed architecture for unified Montgomery multiplier is discussed in detail in Section 4.4. The new processing unit and dual field adder are derived in this section. The dependency graphs and reservation tables of the pipeline are illustrated followed by a block diagram of the multiplier architecture. In Section 4.5, the proposed processing kernel is first evaluated qualitatively for latency and critical path followed by an FPGA implementation to determine the area and total computation time. The chapter is summarized in Section 4.6.

4.2 Preliminaries

4.2.1 $GF(N)$ and $GF(2^m)$

The finite fields $GF(N)$ and $GF(2^m)$ have been presented in Chapter 2. Their differences are discussed briefly here.

The two fields differ primarily in the representation of their elements, which affects the arithmetic operations performed in the respective field. In $GF(N)$, N is a prime number and the elements are integers in $\{0, 1, \ldots, N-1\}$. Addition and multiplication in $GF(N)$ are the conventional integer addition and multiplication operations followed by a reduction using the modulus N. The reduction is done to ensure that the result lies in $\{0, 1, \ldots, N-1\}$.

In $GF(2^m)$, the elements are represented as polynomials of order m, i.e. $a(x) = \sum_{i=0}^{m-1} a_i x^i$. Addition is a simple bitwise-XOR operation and multiplication involves reduction using an irreducible reduction polynomial $p(x) = \sum_{i=0}^{m} p_i x^i$.

It was shown in Chapter 2 that ECCs can operate in both $GF(N)$ and $GF(2^m)$. The standard elliptic curves and their corresponding reduction polynomials are defined for the two fields by NIST [37] and SEC [3]. RSA cryptosystems, however, operate in $GF(N)$ only.

4.2.2 Montgomery multiplication in $GF(N)$ and $GF(2^m)$

Montgomery modular multiplication (MMM) was first introduced for $GF(N)$ in [32]. A number of algorithmic innovations have been done to implement MMM for $GF(N)$ in the most efficient way [6, 10, 11, 34, 35, 39]. Let us now look into MMM algorithms in $GF(N)$ and $GF(2^m)$ and then describe how they are amalgamated to obtain a unified architecture.

The Montgomery modular multiplication in $GF(N)$ for the product of two m-bit binary inputs, $A = \sum_{i=0}^{m-1} a_i 2^i$ and $B = \sum_{i=0}^{m-1} b_i 2^i$ is defined as $C = ABR^{-1}(mod\ N)$, where N is an odd modulus and the integer R, generally taken as 2^m, is relatively prime to N. The bit-level Montgomery multiplication algorithm is listed in Algorithm 11 [48].

Algorithm 11 Montgomery Modular Multiplication in $GF(N)$

```
1: Input: $A = \sum_{i=0}^{m-1} a_i 2^i, B = \sum_{i=0}^{m-1} b_i 2^i, N = \sum_{i=0}^{m-1} n_i 2^i$
2: Output : $C = AB2^{-m}(mod\ N)$
3: $S \leftarrow 0$
4: for $i = 0$ to $m - 1$ do
5:     $S \leftarrow S + a_i B$; $q \leftarrow s_0$
6:     $S \leftarrow S + qN$
7:     $S \leftarrow S/2$
8: end for
9: if $S \geq N$ then $S \leftarrow S - N$
10: Return $S = AB2^{-m}(mod\ N)$
```

In $GF(2^m)$, the inputs are represented as the polynomial bases. If $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$ and $p(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$, where $a_i, b_i, p_i \in \{0, 1\}$, the bit level MMM for $GF(2^m)$ can be formulated as in Algorithm 12 [22].

Algorithm 12 Montgomery Modular Multiplication in $GF(N)$

```
1: Input: $A(x) = \sum_{i}^{m-1} a_i x^i, B(x) = \sum_{i}^{m-1} b_i x^i, p(x) = \sum_{i}^{m-1} p_i x^i$
2: Output : $S(x) = A(x)B(x)x^{-m}(mod\ p(x))$
3: $S \leftarrow 0$
4: for $i = 0$ to $m - 1$ do
5:     $S \leftarrow S \oplus a_i B$; $q \leftarrow s_0$
6:     $S \leftarrow S \oplus qp_i$
7:     $S \leftarrow S/x$
8: end for
9: Return $S(x) = A(x)B(x)x^{-m}(mod\ p(x))$
```

Both algorithms look similar. The most important difference is the addition in `Lines 5 & 6`. In $GF(2^m)$ this addition is a simple bit-wise XOR operation but it involves carry propagation in $GF(N)$. Another difference is the division in `Line 7`. The division by 2 in Algorithm 11 is equivalent to a right shift. This is the same as division by x in Algorithm

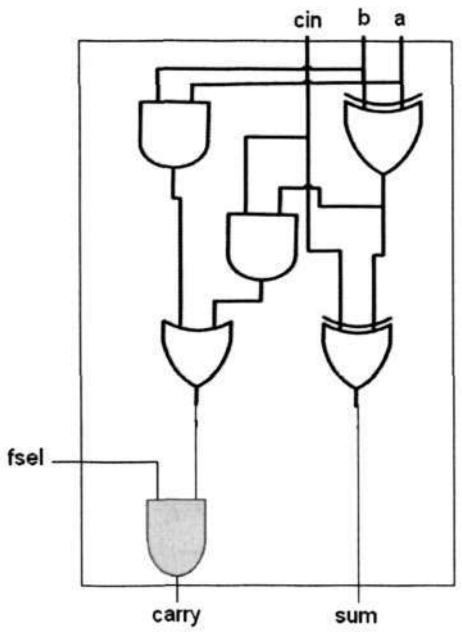


Figure 4.1: Dual field adder in [48]

12. Thus `Line 7` performs the same operation in both the algorithms.

4.2.3 Review of Existing Unified MMMs

The 'kernel' that implements `Lines 5 & 6` needs to be designed efficiently. This kernel comprises identical processing units that form a pipeline. Considering the similarities in Algorithms 11 and 12, by forcing the carry signal in `Lines 5 & 6` of Algorithm 11 to logic zero, Algorithm 12 can be implemented.

The Montgomery multiplications implemented in the unified architectures of [2,43,48,49] introduce dual field adders, where a control signal, called a field select signal, is used to transmit the carry signal generated from the adders. The carry signal is masked out when the multiplication in $GF(2^m)$ is required, by logically *AND*ing the carry and the field select input. Fig. 4.1 shows a one-bit slice of the dual field adder (DFA) as illustrated in [48]. The single bit DFA has an additional AND gate connected to the carry output of a full adder. When $fsel = 0$, the carry signal is forced to '0'. When operating in $GF(2^m)$ mode, since all additions are carry-free bitwise XOR operations, $fsel$ is set to '0'.

Similarly, in [43], a scalable dual-field elliptic curve processor is illustrated. A dual-field r-bit $\times$ r-bit Montgomery multiplier based on Wallace tree carry save architecture is described. [48] describes a scalable unified architecture which employs a processing element with dual field adders. In a more recent paper by Savas et. al. [49], a unified architecture based on Montgomery Multiplication with precomputation is illustrated. This again uses dual field adders. A unified radix-4 architecture is described in [2] which uses a novel (4:2) adder. This architecture however, involves the digit coding of inputs to minimize delay due to a globally-broadcast field select signal

In this chapter, a novel unified processing unit based on the architectures in [48] is presented. [48] is chosen as this is the simplest and also one of the fastest available unified multipliers.

4.3 Problem Statement

The Montgomery multiplication algorithm in $GF(N)$ that is implemented in carry-save format (as listed in [48]) requires two gated full adders to process every partial product bit in each iteration. Its $GF(2^m)$ counterpart can use the same two gated full adders for each partial product bit but the carry signals are forced to logic zero. To illustrate the dual field adder issue that is being addressed in this chapter, the communications between two w-bit DFAs, DFA1 and DFA2, within each processing unit (PU) of the Montogmery multiplier in [48], are shown in Fig. 4.2. A w-bit DFA consists of w one-bit DFA cells for word-level processing. $DFA1_j$ denotes the j-th single bit DFA cell of w-bit DFA1.

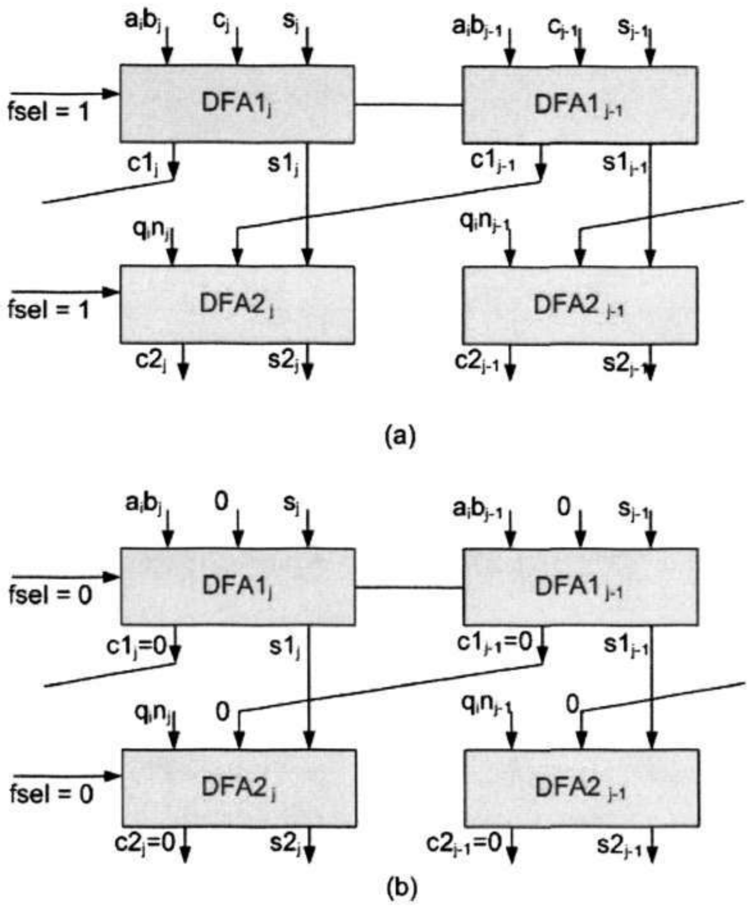


Figure 4.2: Input/output of adders (a) operation in $GF(N)$ when $fsel = 1$ (b) operation in $GF(2^m)$ when $fsel = 0$

When the multiplier is to be operated in $GF(N)$, $fsel$ is set to '1' (see Fig. 4.2(a)). $DFA1_j$ of the first level DFA of the PU receives three inputs - sum (s_j), carry (c_j) and a_ib_j and outputs the sum ($s1_j$) and carry ($c1_j$) signals to the next level. When $fsel$ = '1', $c1_j$ is not forced to 0. These sum and carry bits are then propagated to the corresponding bit slice of DFA2. The third input feeding into the j-th cell of DFA2 is q_in_j. Consequently, the logic circuits of the two DFAs can hardly be simplified when $fsel$ = '1' as all the inputs are being fed with values.

However, when $GF(2^m)$ multiplication is selected by setting $fsel = 0$, the carry input c_j of $DFA1_j$ is '0', as shown in Fig. 4.2(b). This is because the $fsel$ of the previous level of adders forces the carry, $cout_j$ to '0'. Similarly, $fsel$ also sets the carry output of $DFA1_j$, $c1_j$, to '0'. Thus, the logic functions performed in the j-th cell of DFA1 are $s1_j = s_j \oplus a_i b_j$ and $c1_j = 0$. This reduces the logic functions of $DFA2_j$ to $sout_j = s1_j \oplus q_i n_j$ and $cout_j = 0$. One input and one output of each dual field adder cell are redundant in $GF(2^m)$ multiplication.

It is important to note that MMM in $GF(2^m)$ requires only two 2-input XOR operations (Algorithm 12). These two XOR gates are available in *one* full adder. In [48], as a result of unification, two full adders (or 4 XOR gates) are being used to perform an operation that requires only two XOR operations, as illustrated in Fig. 4.2(b). The logic function of one XOR gate in each adder is redundant. By using two full adders for the computation in both $GF(N)$ and $GF(2^m)$, the speed of computation in $GF(2^m)$ is penalized.

This redundancy will be removed in our proposed scalable unified architecture which is derived based on the architecture presented in [48]. In the next section, a novel processing unit with modified dual field adders is derived. The proposed architecture uses one full adder for the multiplication in $GF(2^m)$ but the designated two full adders are employed in $GF(N)$ mode, thus improving the speed of multiplication in $GF(2^m)$ without compromising the total time required in $GF(N)$. The dependency graphs resulting from these architectural transformations will be used for a qualitative timing analysis.

4.4 Proposed Unified Montgomery Multiplier

4.4.1 Notations

For the rest of the chapter, A, B, N in $GF(N)$ are equivalent to $A(x)$, $B(x)$ and $p(x)$ in $GF(2^m)$ respectively. All the inputs are as wide as the modulus (in $GF(N)$) or the reduction polynomial (in $GF(2^m)$), which is m bits wide. Input A is read bit-wise. B and

p are read as w-bit words giving a total of $e = \left\lceil \frac{m}{w} \right\rceil$ words. i is used as a bit counter for A and j is used as a word counter for B and p. The sum and carry signals will also be produced at word level in each iteration. Each PU of the unified Montgomery multiplier comprises two levels of DFA cells. The w-bit sum and carry words produced from the first level of DFA cells (DFA1) are represented as a tuple $(C1, S1)$. Its equivalent normal binary number is $2C1 + S1$. The sum and carry words generated by the second level of DFA cells (DFA2) are represented as TS and TC. These two outputs are shifted right before feeding into the corresponding cells of DFA1 of the next PU. The generalized notation for the k-th bit of the j-th partial sum and carry words generated in the 'i'th iteration are represented as $ts_{j,k}^{(i)}$ and $tc_{j,k}^{(i)}$, respectively.

4.4.2 Derivation of the Dual-field Logic

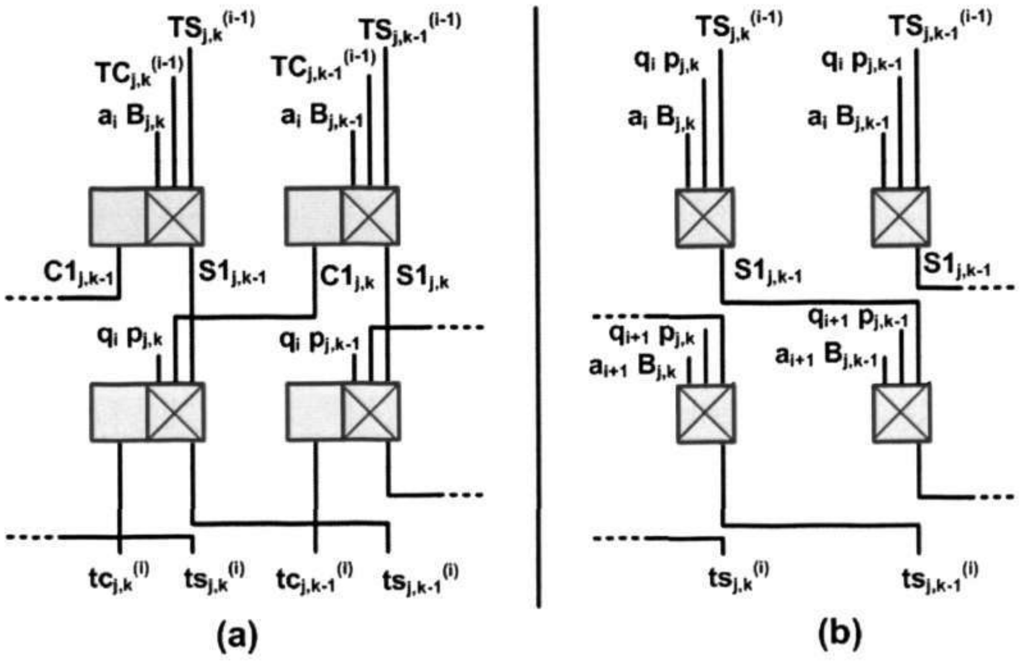


Figure 4.3: Sum and carry logics in (a) $GF(N)$ (b) $GF(2^m)$

With the above notations, the logic constituting the dual field adder, that is employed in MMM algorithms, can be scrutinized. A DFA uses gated full adder to produce the intermediate output in `sum logic` (represented by crossed box in Fig. 4.3) and `carry logic` (represented by empty box in Fig. 4.3). During the operation in $GF(N)$, two `sum` and `carry logics` are needed to compute one `sum` and one `carry` word ($TS_j^{(i)}$ and $TC_j^{(i)}$). The first `sum logic`, `S1_logic`, is produced from three inputs -

1. right shifted `sum` word from the previous iteration, $TS_j^{(i-1)}$,
2. right shifted `carry` word from the previous iteration, $TC_j^{(i-1)}$, and
3. $a_i{\cdot}B_j$.

The second `sum logic`, `S2_logic`, is generated from the following three inputs -

1. sum and carry, $(C1_j^{(i)}, S1_j^{(i)})$, from `S1_logic` and
2. $q{\cdot}p_j$ where $q = s1_{0,0}^{(i)}$.

The corresponding `carry logics`, `C1_logic and C2_logic` are generated based on the same sets of input signals accordingly.

On the other hand, operations in $GF(2^m)$ (Fig. 4.3(b)) need no `carry logics` since there is no carry propagation. Therefore, the value of $TC_j^{(i)}$ is immaterial. Instead of forcing $TC_j^{(i)}$ to 0 and using it as an input to the successive DFA cell, that DFA cell's input can be better utilized to receive other critical non-cosntant input. If q_i can be precomputed in some way (to be shown later), `Lines 5 and 6` in Algorithm 12 can be merged into one single 3-bit addition operation as $TS_j^{(i)} \leftarrow TS_j^{(i-1)} \oplus a_iB_j \oplus q_iP_j$. Hence, to compute one ring sum, $TS_j^{(i)}$ in $GF(2^m)$, only one `sum logic` is sufficient. It takes the following inputs:

1. right shifted sum from the previous iteration, $TS_j^{(i-1)}$,
2. $a_i{\cdot}B_j$ and,
3. $q \cdot p_j$

where $q = a_i \cdot b_{0,0} \oplus ts_{0,0}^{(i-1)}$. For the time being, let us assume we have the intermediate signal q, the generation of which will be discussed later.

In order to compute the j-th sum and carry words in the i-th iteration in $GF(N)$, two **sum logics** are needed but a single **sum logic** can calculate the sum in $GF(2^m)$. Thus in a processing unit (PU), we need two **sum** and **carry logics** to calculate the necessary signals in $GF(N)$ mode. While operating in $GF(2^m)$ mode, the second **sum logic** can be used to compute the sum for the next iteration, i.e. the $(i+1)$-th iteration to produce $TS_j^{(i+1)}$.

Although the above proposition is viable in principle, from Fig. 4.3, the following contentions are observed in the inputs to the logic blocks.

1. The first input signal to $DFA1_j$ and $DFA2_j$ are $a_i B_j$ and $q_i p_j$, respectively in $GF(N)$ mode but they become $a_i B_j$ and $a_{i+1} B_j$, respectively in $GF(2^m)$ mode.

2. In $GF(N)$ mode, the sum logics require the carry signal ($TC_j^{(i-1)}$ in $DFA1_j$ and $C1_j^{(i)}$ in $DFA2_j$) as their second input but this input is $q_i p_j$ and $q_{i+1} p_j$ in $GF(2^m)$ mode.

3. The sum signals to **sum logic - S2_logic** are different for the two fields. While the sum output $S1$ of DFA1 feeds directly into the corresponding sum circuit of DFA2 (without any shift) in $GF(N)$, the same sum signal needs to be shifted right (according to `Line 7` of Algorithm 12) in the case of $GF(2^m)$.

An external control logic can handle the first contention by multiplexing (using the field select signal) the inputs from serial shift registers. In order to resolve the next two contentions, we need multiplexers in the processing unit. The second contention can be resolved by selecting either the carrys that are generated in each level or the input, i.e. select $TC_j^{(i-1)}$ or $q_i p_j$ in $DFA1_j$ and $C1_j^{(i)}$ or $q_{i+1} p_j$ in $DFA2_j$, with a multiplexer and the field select signal. The third contention needs further elaboration. This contention arises because of the `sum` signals that are generated by the `sum logic` in the first level of DFAs, i.e. $S1_j^{(i)}$ of the PU. This signal is branched into two signals - S_{GFP} and S_{GF2}. S_{GFP} is obtained by passing $S1_j^{(i)}$ directly whereas S_{GF2} is obtained by right shifting $S1_j^{(i)}$ by one bit. Depending on the field select signal, one of these two signals is multiplexed into one free input of the DFAs in the second level, i.e. `S2_logic` in DFA2 of the PU. This allows the `sum logics` to carry out the operations of the next iteration when it is operating in $GF(2^m)$ mode. A right shift of the sum is required according to `Line 7` of Algorithm 12. On the other hand, when the PU is operating in $GF(N)$ mode, the sum generated from `S1_logic` in DFA1 should not be right shifted as its second level of DFAs has to perform the operation in `Line 6` of Algorithm 11. The right shifting in $GF(N)$ is deferred until DFA2 has completed its operation for the current iteration. By generating two signals TS_{GFP} and TS_{GF2} from the sum of `S1_logic`, and then using a multiplexer to select one of these signals into the free input of `S2_logic` of the same PU, the third contention is resolved.

4.4.3 Derivation of the Dependency Graphs

A processing unit (PU) embracing the above mentioned DFAs forms the kernel of the proposed unified Montgomery modular multiplier. In this new PU, two levels of DFA cells are required for one iteration of MMM in $GF(N)$ (see Lines 5 to 7 in Algorithm 11). However, in $GF(2^m)$ mode, the second level of DFAs will be used to process the next iteration of MMM. While two cycles are required to complete one iteration of MMM in $GF(N)$, only one cycle is required to complete one iteration of MMM in $GF(2^m)$. To accommodate these changes due to the use of the new dual field adders, new dependency graphs (DGs) need to be derived.

Table 4.1 shows the desired input data flow while operating in $GF(N)$ and $GF(2^m)$. This table is constructed by assuming that the inputs to the PUs are 3-bit words. Each row represents a clock cycle and each column represents a level of DFA cells in a PU. The PUs are uniquely identified by PU_0, PU_1, etc. Since each PU has two DFA levels, the first DFA level of the first PU is labeled as PU_0S1 and the second level as PU_0S2 and so on for the other PUs. Each column of the table shows only the inputs - $a(x), b(x)$ and $p(x)$ to the adders. The sum and carry output signals are not shown. The j-th sum and carry words from *level 1* of an i-th PU are denoted as $C1_j^{(i)}$ and $S1_j^{(i)}$, respectively. These two numerals are often paired and expressed in a convenient tuple notation, $(C1_j^{(i)}, S1_j^{(i)})$, which is equivalent to the decimal sum of $2C1_j^{(i)} + S1_j^{(i)}$. Similarly, the j-th sum and carry words from *level 2* of an i-th PU can also be represented as $(C2_j^{(i)}, S2_j^{(i)}) \rightarrow 2C2_j^{(i)} + S2_j^{(i)}$. The j-th right shifted carry and sum words for i-th iteration are represented as $TC_j^{(i)}$ and $TS_j^{(i)}$, respectively with the prefix T. Also, $x_{j,k}^{(i)}$ represents the k-th bit of the j-th word of X in the i-th iteration.

Let us first consider the multiplication in $GF(N)$ and examine the process step-by-step.

- *Clock cycle 0:*
  - PU_0S1: Computes the first level sum and carry - $(C1_0^{(0)}, S1_0^{(0)}) = (c1_{0,2}^{(0)}c1_{0,1}^{(0)}c1_{0,0}^{(0)}$ $, s1_{0,2}^{(0)}s1_{0,1}^{(0)}s1_{0,0}^{(0)})$ using the inputs a_0 and the 3-bit word $B_0 = b_2b_1b_0$, and the sum and carry from the previous level - $TS_0^{(-1)}$ and $TC_0^{(-1)}$ which are initialized as 0.

- *Clock cycle 1:*
  - PU_0S1: Processes the next word $a_0 \cdot B_1 = a_0 \cdot (b_5b_4b_3)$ to compute $(C1_1^{(0)}, S1_1^{(0)}) = (c1_{1,5}^{(0)}c1_{1,4}^{(0)}c1_{1,3}^{(0)}, s1_{1,5}^{(0)}s1_{1,4}^{(0)}s1_{1,3}^{(0)})$.
  - PU_0S2: The outputs $(C1_0^{(0)}, S1_0^{(0)})$ are added to $q_0 \cdot N_0$ where q_0 is buffered from $s1_{0,0}^{(0)}$. The result is $(C2_0^{(0)}, S2_0^{(0)}) = (c2_{0,2}^{(0)}c2_{0,1}^{(0)}c2_{0,0}^{(0)}, s2_{0,2}^{(0)}s2_{0,1}^{(0)}s2_{0,0}^{(0)})$. This result needs to be right shifted (according to the MMM algorithm) to produce the 3-bit words $TC_0^{(0)}$ and $TS_0^{(0)}$.

Table 4.1: Data flow of inputs

			$GF(N)$			
Clock	Adder levels in PUs					
cycle	PU_0S1	PU_0S2	PU_1S1	PU_1S2	PU_2S1	PU_2S2
Cycle 0	$a_0(b_2b_1b_0)$					
Cycle 1	$a_0(b_5b_4b_3)$	$q_0(n_2n_1n_0)$				
Cycle 2	$a_0(b_8b_7b_6)$	$q_0(n_5n_4n_3)$				
Cycle 3	$a_0(b_{11}b_{10}b_9)$	$q_0(n_8n_7n_6)$	$a_1(b_2b_1b_0)$			
Cycle 4		$q_0(n_{11}n_{10}n_9)$	$a_1(b_5b_4b_3)$	$q_1(n_2n_1n_0)$		
Cycle 5			$a_1(b_8b_7b_6)$	$q_1(n_5n_4n_3)$		
Cycle 6			$a_1(b_{11}b_{10}b_9)$	$q_1(n_8n_7n_6)$	$a_2(b_2b_1b_0)$	
Cycle 7				$q_1(n_{11}n_{10}n_9)$	$a_2(b_5b_4b_3)$	$q_2(n_2n_1n_0)$
Cycle 8					$a_2(b_8b_7b_6)$	$q_2(n_5n_4n_3)$
Cycle 9					$a_2(b_{11}b_{10}b_9)$	$q_2(n_8n_7n_6)$
Cycle 10						$q_2(n_{11}n_{10}n_9)$

			$GF(2^m)$			
Clock	Adder levels in PUs					
cycle	PU_0S1	PU_0S2	PU_1S1	PU_1S2	PU_2S1	PU_2S2
Cycle 0	$a_0(b_2b_1b_0)$ $q_0(n_2n_1n_0)$					
Cycle 1	$a_0(b_5b_4b_3)$ $q_0(n_5n_4n_3)$					
Cycle 2	$a_0(b_8b_7b_6)$ $q_0(n_8n_7n_6)$	$a_1(b_2b_1b_0)$ $q_1(n_2n_1n_0)$				
Cycle 3	$a_0(b_{11}b_{10}b_9)$ $q_0(n_{11}n_{10}n_9)$	$a_1(b_5b_4b_3)$ $q_1(n_5n_4n_3)$				
Cycle 4		$a_1(b_8b_7b_6)$ $q_1(n_8n_7n_6)$	$a_2(b_2b_1b_0)$ $q_2(n_2n_1n_0)$			
Cycle 5		$a_1(b_{11}b_{10}b_9)$ $q_1(n_{11}n_{10}n_9)$	$a_2(b_5b_4b_3)$ $q_2(n_5n_4n_3)$			
Cycle 6			$a_2(b_8b_7b_6)$ $q_2(n_8n_7n_6)$	$a_3(b_2b_1b_0)$ $q_3(n_2n_1n_0)$		
Cycle 7			$a_2(b_{11}b_{10}b_9)$ $q_2(n_{11}n_{10}n_9)$	$a_3(b_5b_4b_3)$ $q_3(n_5n_4n_3)$		
Cycle 8				$a_3(b_8b_7b_6)$ $q_3(n_8n_7n_6)$	$a_4(b_2b_1b_0)$ $q_4(n_2n_1n_0)$	
Cycle 9				$a_3(b_{11}b_{10}b_9)$ $q_3(n_{11}n_{10}n_9)$	$a_4(b_5b_4b_3)$ $q_4(n_5n_4n_3)$	
Cycle 10					$a_4(b_8b_7b_6)$ $q_4(n_8n_7n_6)$	$a_5(b_2b_1b_0)$ $q_5(n_2n_1n_0)$

- *Clock cycle 2:*
  - PU_0S1: Processes the next word $a_0 \cdot B_2 = a_0 \cdot (b_8b_7b_6)$ to compute $(C1_2^{(0)}, S1_2^{(0)})$.
  - PU_0S2: The outputs $(C1_1^{(0)}, S1_1^{(0)})$ from level 1 and Cycle 1 are added to $q_0 \cdot N_1$ to produce $(C2_1^{(0)}, S2_1^{(0)}) = (c2_{1,5}^{(0)}c2_{1,4}^{(0)}c2_{1,3}^{(0)}, s2_{1,5}^{(0)}s2_{1,4}^{(0)}s2_{1,3}^{(0)})$

    Note: It is important to note here that S1 of the next processing unit PU_1 does not start processing the results from the level 2 DFA of PU_0 obtained in Cycle 1. This is because the sum and carry obtained from PU_0S2, i.e. $(C2_0^{(0)}, S2_0^{(0)})$ need to be right shifted first. On right shifting $(C2_0^{(0)}, S2_0^{(0)})$, the 3-bit words $TC_0^{(0)}$ and $TS_0^{(0)}$ would be $TC_0^{(0)} = c2_{0,2}^{(0)}c2_{0,1}^{(0)}c2_{0,0}^{(0)}$ and $TS_0^{(0)} = s2_{0,3}^{(0)}s2_{0,2}^{(0)}s2_{0,1}^{(0)}$. The most significant bit of $TS_0^{(0)}$, which is $s2_{0,3}^{(0)}$ will only be available at the end of Cycle 2. Hence the results obtained from PU_0S2 obtained in Cycle 1 have to be buffered at this point.

- *Clock cycle 3:*
  - PU_0S1: Processes the next word $a_0 \cdot B_3 = a_0 \cdot (b_{11}b_{10}b_9)$ to compute $(C1_3^{(0)}, S1_3^{(0)})$.
  - PU_0S2: The outputs, $(C1_2^{(0)}, S1_2^{(0)})$ from level 1 DFA and Cycle 1 are added to $q_0 \cdot N_2$ to produce $(C2_2^{(0)}, S2_2^{(0)})$.
  - PU_1S1: Now, it processes the next iteration with the inputs $a_1 \cdot B_0$ and the right-shifted vectors, $TS_0^{(0)}$ and $TC_0^{(0)}$ obtained the from PU_0S2. The result is $(C1_0^{(1)}, S1_0^{(1)})$.

- This process continues as shown in Table 4.1.

The data flow for the MMM in $GF(2^m)$ is also listed in Table 4.1. This process is more straightforward since each DFA level of a PU completes one iteration. In every column of Table 4.1, both a_iB_j and q_iN_j are listed. This is because there are only three inputs that need to be XORed - a_iB_j, q_iN_j and the right shifted sum from the previous iteration $TS_j^{(i-1)}$. For the time being, let us assume the availability of q_i. In Cycle 0, Algorithm 12 commences the computation of $S1_0^{(0)} = s1_{0,2}^{(0)}s1_{0,1}^{(0)}s1_{0,0}^{(0)}$ with PU_0S1. In Cycle 1, PU_0S1 computes the next word in the same iteration (Iteration 0) to produce $S1_0^{(0)} = s1_{0,5}^{(0)}s1_{0,4}^{(0)}s1_{0,3}^{(0)}$. In Cycle 2, PU_0S1 carries on with the computation of the next word while the second DFA level of PU_0 can start the next iteration, i.e., Iteration 1 of Algorithm 12. It follows similar logical steps explained previously for $GF(N)$. The exception is that the right shifted sum output from the even iteration number is needed in the successive odd iteration number. So the input required by DFA2 of PU_1 to start Iteration 1 is $TS_0^{(0)} = s1_{0,3}^{(0)}s1_{0,2}^{(0)}s1_{0,1}^{(0)}$. In this vector, $s1_{0,3}^{(0)}$ is available only at the end of Cycle 1. As a result, the computation of Iteration 1 in the level 2 adders of PU_0 can only start in Cycle 2. This implies that the result from PU_0S1 needs to be buffered.

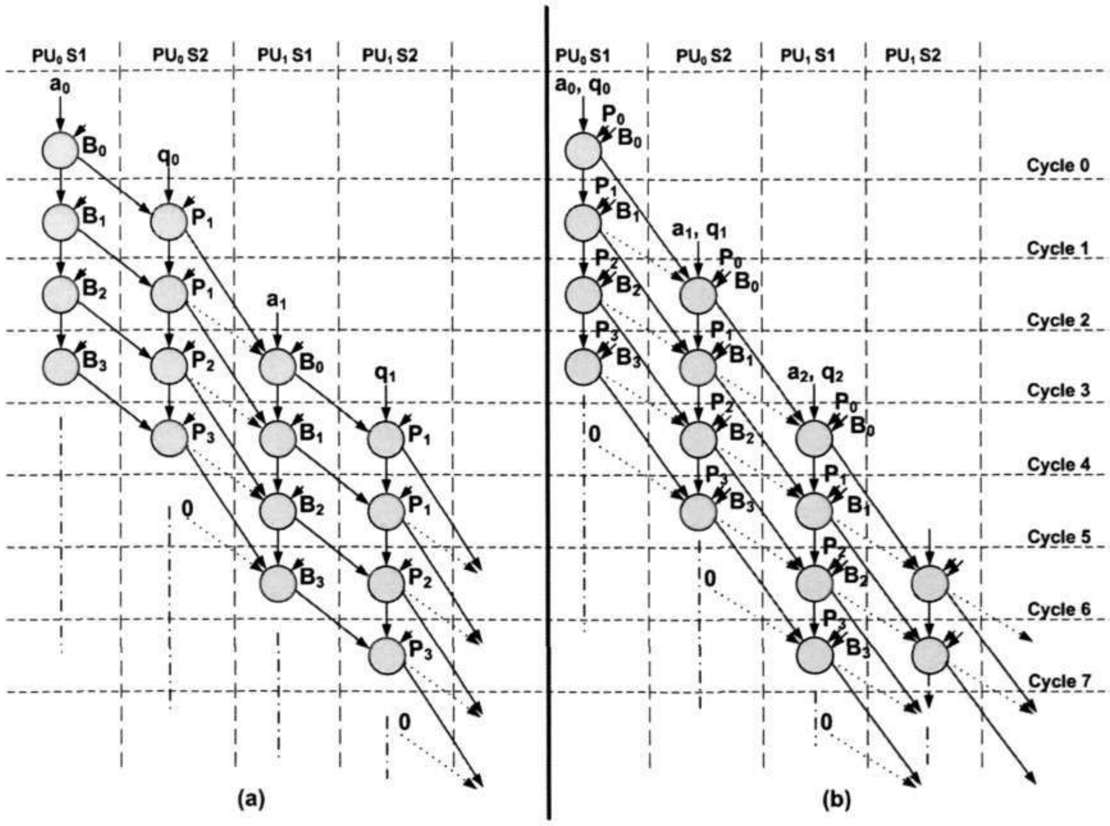


Figure 4.4: Dependency graphs for (a) $GF(N)$ (b) $GF(2^m)$

Based on the data flow discussed above, the dependency graphs (DGs) for the kernel to operate in the two modes can be derived. These are shown in Fig. 4.4. Each bubble represents a DFA level of a PU. The PU and its DFA level are indicated at the top of each column of the DGs. The vertical dotted lines between the columns of bubbles are cutsets which will be implemented as flip-flops in the architecture. The clock cycle is enumerated on the right hand side of the DGs. Each solid line arrow represents two w-bit vectors, $TS_j^{(i)}$ and $TC_j^{(i)}$, for MMM in $GF(N)$ but it represents only $TS_j^{(i)}$ for MMM in $GF(2^m)$. Additionally, each dotted line arrow in the DG for $GF(N)$ represents a one bit signal that is generated after the sum vector from the second DFA level has been right shifted. In the case of $GF(2^m)$, each dotted line represents a one bit signal from the right shifted sum vector for every DFA level.

The DGs show the temporal relationship of the operators graphically. Two levels of DFAs are used to compute the sum and carry for one iteration of MMM in $GF(N)$, whereas in $GF(2^m)$ mode, each level of DFAs computes one iteration. Furthermore, the DGs lucidly illustrate the shifting and buffering of data according to the data flow in Table 4.1. In Fig. 4.4(a), PU_1S1 waits until Cycle 3 to receive $TS_0^{(0)}$ (which is $s2_{0,3}^{(0)}s2_{0,2}^{(0)}s2_{0,1}^{(0)}$) from PU_0S2. In $TS_0^{(0)}$, $s2_{0,2}^{(0)}$ and $s2_{0,1}^{(0)}$ are generated at the end of Cycle 1 as shown by the solid line arrow pointing to the first bubble in the third column of Fig. 4.4. $s2_{0,3}^{(0)}$ is available at the end of Cycle 2, as indicated by the dotted line arrow pointing towards PU_1S1. In a similar way, the data flow for MMM in $GF(2^m)$ can also be interpreted from Fig. 4.4(b).

4.4.4 Processing Unit for Unified Multiplier

Based on the dual-field logic and dependency graphs proposed in Sections 4.4.2 and 4.4.3, a new dual field adder and processing unit for the proposed unified multiplication scheme are devised as shown in Fig. 4.5. The '/' between two signal names indicates that these signals are to be multiplexed using an external control logic. The small filled rectangules are registers. The PU consists of two levels of DFA cells separated by registers. One PU computes two iterations of Algorithm 12 in $GF(2^m)$, but one iteration of Algorithm 11

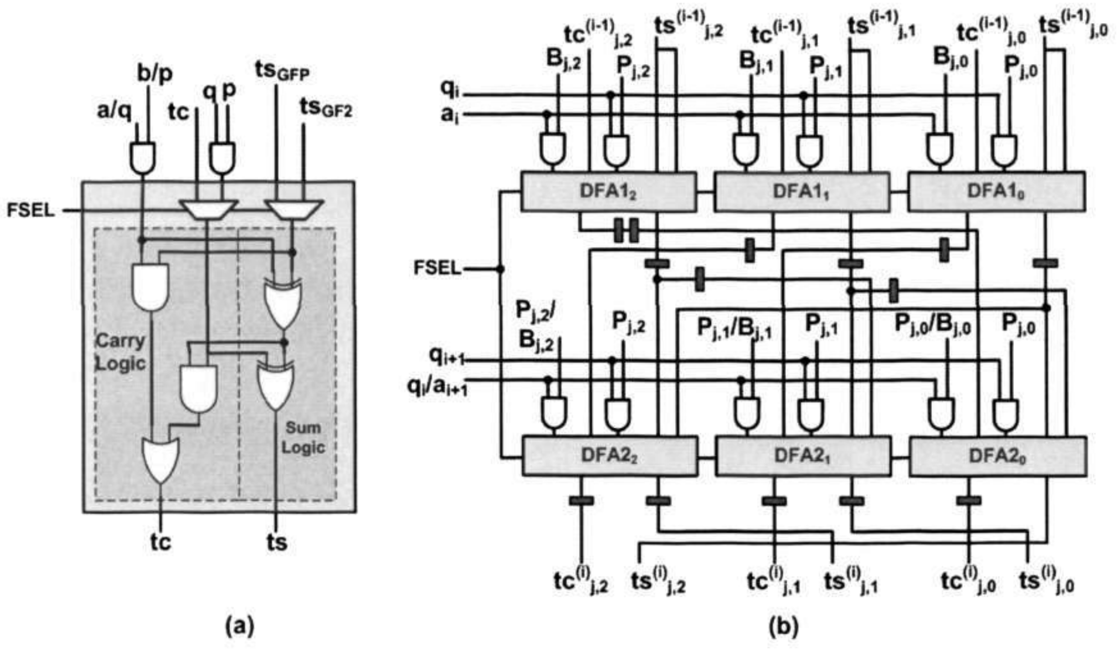


Figure 4.5: (a) Dual Field Adder for $w = 1$ (b) Processing unit for wordlength $w = 3$

$(GF(N))$ in two clock cycles. The registers are inserted based on the dependency graphs and the data flow tables. The multiplier is made scalable to field orders or moduli by forming a pipelined processing kernel with a number of these processing units. The area-time trade off will be studied to determine the optimal number of PUs to be employed in the pipeline.

4.4.5 Computation of q

For ease of exposition, the issue pertaining to the computation of the intermediate signal q was ignored in earlier presentation . In Algorithms 11 and 12, q is the least significant bit, s_0 of S in `Line 5`. There is actually a data dependency on q in `Lines 5 and 6` of these algorithms. The addition in `Line 6` can only be executed after computing q in `Line 5`. This dependency was seen in Chapter 3 also which happened to lie in the critical path of the systolic array implementations.

The dependency on q remains an issue in unified architectures. In the architecture proposed in [48], the least significant bit of the sum generated from the first row of DFAs (that computes $a_iB_0 + TC_0^{(i)} + TS_0^{(i)}$) is q and this value is buffered till the end of the current iteration. In the proposed architectures, the computation of q is tackled differently. Referring to Table 4.1, when the multiplier is operating in $GF(N)$ mode, q_0 is required in Cycle 1 for the first iteration involving a_0. So, in Cycle 0, q_0 is computed as shown below.

$$q_0 = a_0b_0 + ts_{0(0)}^{(-1)} + tc_{0(0)}^{(-1)} \tag{4.1}$$

where $ts_{0(0)}^{(-1)}$ and $tc_{0(0)}^{(-1)}$ are initialization values set to 0. For the next iteration, q_1 is required in Cycle 4. In Cycle 3, the required inputs, a_1b_0, $ts_{0,0}^{(0)}$ (please refer to notations in Section 4.4.1) and $tc_{0,0}^{(0)}$, are available for the computation of q_1. So q_i is computed in the cycle as soon as the first byte of input B in each iteration has been received.

The computation of q_i in $GF(2^m)$ is more challenging. In $GF(N)$, the operations in `Lines 5 & 6` of the MMM algorithm are split into two cycles. In the first cycle, q_i is computed along with S and in the next cycle, q_i is used. In the case of $GF(2^m)$, the very idea of the proposed logic is to execute these two lines with one single operation, i.e., it is aimed to add $a_iB_0 \oplus q_iN_0 \oplus TS_0^{(i-1)}$ in the same cycle. In other words, q_i needs to be made available before this operation takes place. With reference to Table 4.1, in $GF(2^m)$ mode, q_0 is required in Cycle 0. Here precomputation is necessary. It is important to note that this precomputation is required only for the first iteration, i.e., for $i = 0$. After the first iteration, the pipeline provides the latency to compute q_i. For $i = 0$, q_0 is equal to $a_0 \cdot b_0$ (because the initial value of the sum is '0'). The next iteration ($i = 1$) starts in Cycle 2. The inputs required to compute q_1 are $a_1 \cdot b_0$ and $ts_{0,1}^{(0)}$. The partial product bit $a_1 \cdot b_0$ is available from the inputs. The second input is the lsb of the right-shifted sum from the previous iteration. This is available in Cycle 1. Hence, q_1 can be computed in Cycle 1 and

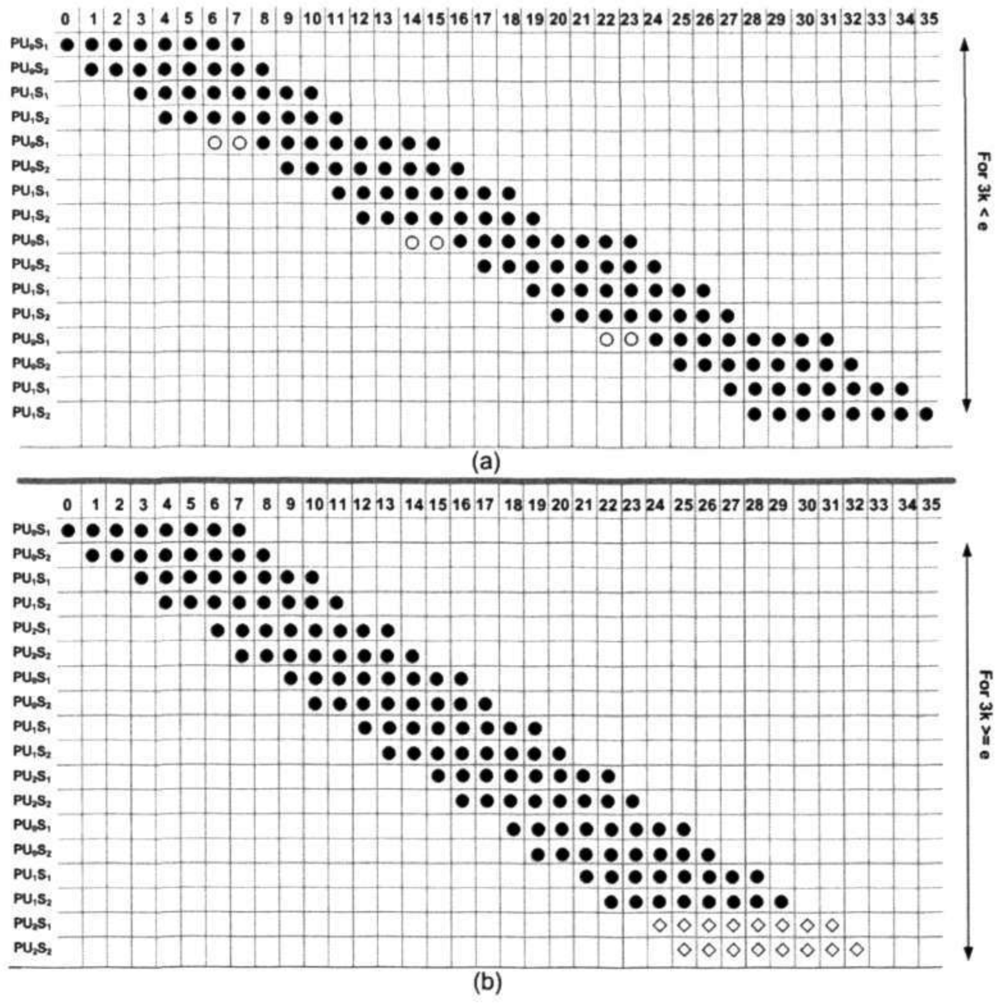


Figure 4.6: Reservation table for operation in $GF(N)$ (a) $m = 8$, $w = 1$ and $k = 2$ (b)$m = 8$, $w = 1$ and $k = 3$

then be used in Cycle 2 to start iteration 1.

4.4.6 Reservation Table of the Pipeline

The reservation tables for the pipelined operation of the multiplier in $GF(N)$ and $GF(2^m)$ are illustrated in Fig. 4.6 and 4.7, respectively.

Fig. 4.6(a) shows the reservation table for the pipeline in $GF(N)$ mode of operation for $m = 8$, $w = 1$ and $k = 2$. As discussed previously, PU_0 starts processing Iteration

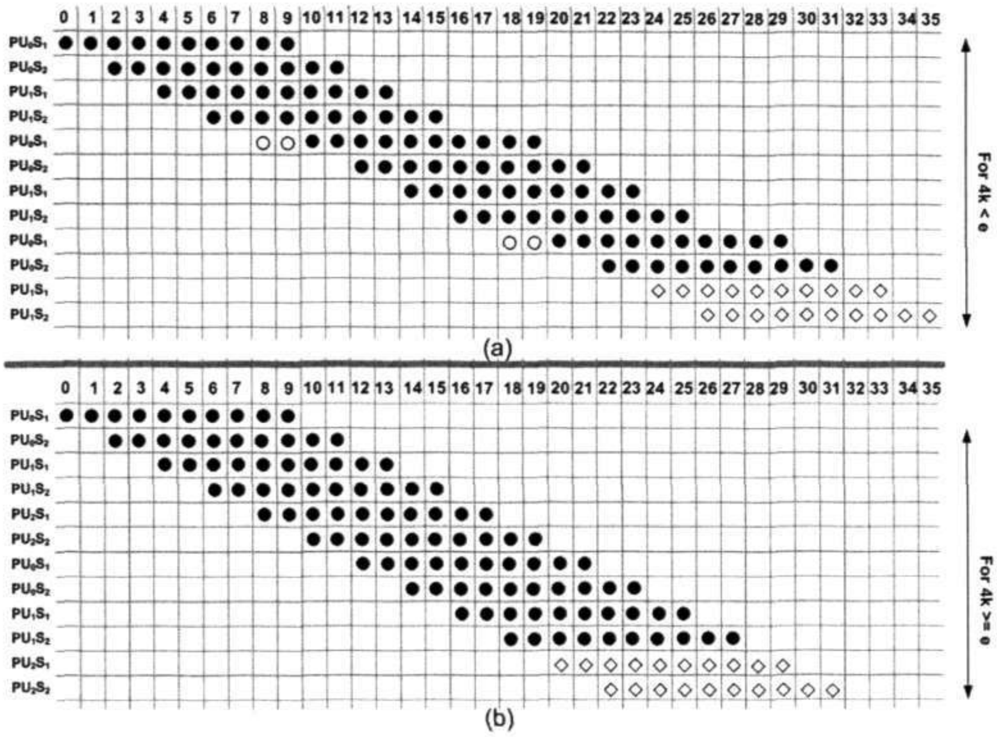


Figure 4.7: Reservation table for operation in $GF(2^m)$ (a) $m = 10$, $w = 1$ and $k = 2$ (b) $m = 10$, $w = 1$ and $k = 3$

0 of MMM algorithm from Cycle 0 (shown as solid dot) and PU_1 starts Iteration 1 from Cycle 3. The outputs from PU_1 are ready in Cycle 6 to start Iteration 2 but the pipeline is not free (PU_0 is still processing). Therefore the pipeline is stalled (marked as ◯) for two cycles before PU_0 can begin Iteration 2 in Cycle 8. In Fig. 4.6(b), the reservation table for $m = 8$, $w = 1$ and $k = 3$ is shown. Here there is no pipeline stall due to the additional PU (PU_2). By the time the first output of PU_2 is available, PU_0 is free and ready to process the next iteration. In this case, however, extra computations are needed at the end to flush the pipeline (represented by ◇). This is because the input A is 8 bits long which is not a factor of the number of PUs in the pipeline. Analogous reservation table for the operation of the pipeline in $GF(2^m)$ mode can also be derived. This is illustrated in Fig. 4.7.

The conditions for the pipeline stall can be formulated as follows.

$$\begin{aligned} 3k &\leq e \quad \text{for pipeline stall in } GF(N) \\ 4k &\leq e \quad \text{for pipeline stall in } GF(2^m) \end{aligned} \tag{4.2}$$

e is the number of words, $\left\lceil \frac{m}{w} \right\rceil$ as defined previously. These conditions are important in evaluating the latency of the unified MMM later.

4.4.7 Multiplier Architecture

The block diagram of the multiplier architecture is illustrated in Fig. 4.8. The control architecture can be derived from the DGs in Fig. 4.4 and the data flow in Table 4.1. The inputs, A, B and N are stored in three sets of shift registers - SR-A, SR-B and SR-N, respectively. The shift register, SR-A is an m-bit linear shift register. An m-bit linear shift register is a cascade connection of m single bit registers. SR-B and SR-N are two banks of linear shift registers. Each bank consists of w parallel $(e+1)$-bit long linear shift registers to allow a w-bit word to be shifted in parallel. The processing kernel has two PUs in the pipeline. Each PU is associated with a q-computation block to compute q using the lsb generated from its first level of DFAs. q_i and q_{i+1} are computed from each q-computation block. The multiplexers are allocated to resolve the signal contention problems 1 and 2 mentioned in Section 4.4.2. Two banks of w parallel shift registers are used to store the sum and the carry from PU_1 (labeled as SR-TC and SR-TS). (4.3) gives the length of shift registers, SR-TC and SR-TS, in terms of the number of words per input, e, and the number of PUs, k. These shift registers are required to store the results when there is a pipeline stall. The control architecture does not fall in the critical path of the multiplier as it involves only simple 2-to-1 multiplexers.

$$\begin{aligned} \text{For } GF(N) \quad l_{SR} &= e - 3k \quad \text{for } 3k \geq e \\ l_{SR} &= 0 \quad \text{otherwise} \\ \text{For } GF(2^m) \quad l_{SR} &= e - 4k \quad \text{for } 4k \geq e \\ l_{SR} &= 0 \quad \text{otherwise} \end{aligned} \tag{4.3}$$

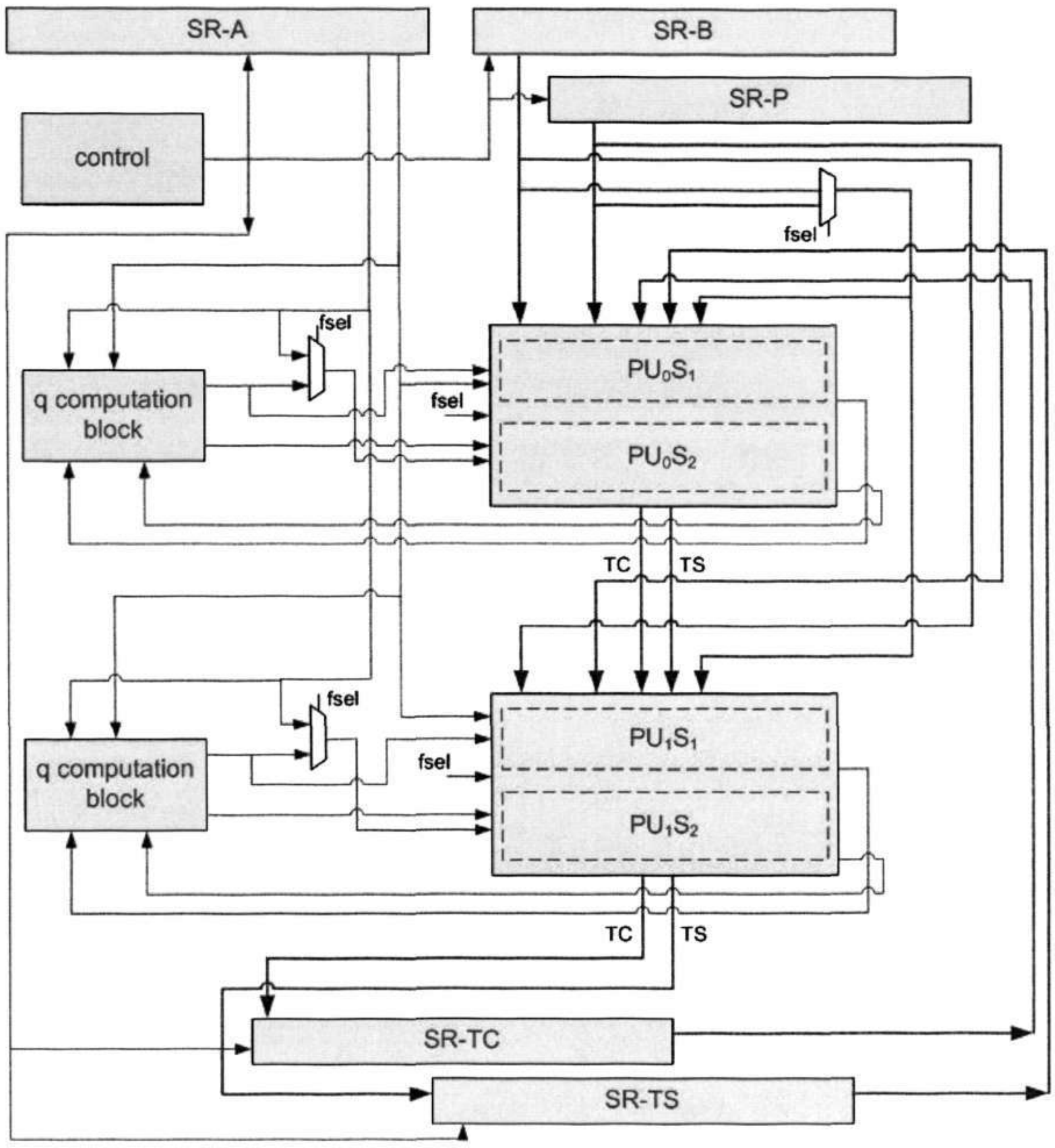


Figure 4.8: Block diagram of the multiplier

4.5 Results

In this section, the hardware implications of the proposed PU, when implemented on FPGA platform, are studied. It is evaluated against [48] for the latency (L)in terms of the number of cycles, the critical path delay (t_{min}), the area cost in terms of the number of FPGA slice used, and the total computation time (t_{total}) for different wordlengths w, number of PUs, k and modulus precision, m.

4.5.1 Critical Path and Latency

The maximum operating frequency of the circuit is determined by the critical path delay t_{min} expressed in ns, which is defined as the maximum combinational delay between two registers. The t_{min} of the proposed PU and the one in [48] are given by

$$\begin{aligned} \text{In [48]} \quad t_{min} &= 11\Delta_{2inp} + 8\Delta_{inv} \\ \text{Proposed} \quad t_{min} &= 6\Delta_{2inp} + 5\Delta_{inv} \end{aligned} \tag{4.4}$$

where Δ_{2inp} is the delay of a 2-input AND/OR gate and Δ_{inv} is the delay of an inverter. The delays of the XOR gates and multiplexers are estimated and expressed in terms of the 2-input gates and inverters. From the above expressions, it is seen that t_{min} of the proposed PU is almost half that of [48]. The reduction in t_{min} is because the critical path in [48] passes through two levels of adders but in the proposed PU, it passes through only one level of adders.

The latency, L_{GFN}, in terms of the total number of clock cycles required for the multiplication in $GF(N)$ is given by

$$L_{GFN} = \begin{cases} e \times \lceil \frac{m}{k} \rceil + 3 \times (k-1) + 1 & \text{for } 3k < e \\ 3k \times \lceil \frac{m}{k} \rceil + e - 2 & \text{for } 3k \geq e \end{cases} \tag{4.5}$$

where $m = log_2(N)$ is the bit width of modulus 'N', k is the number of processing units used in the kernel, $e = \lceil (m/w) \rceil$ where w is the wordlength of B and p. Similarly, the

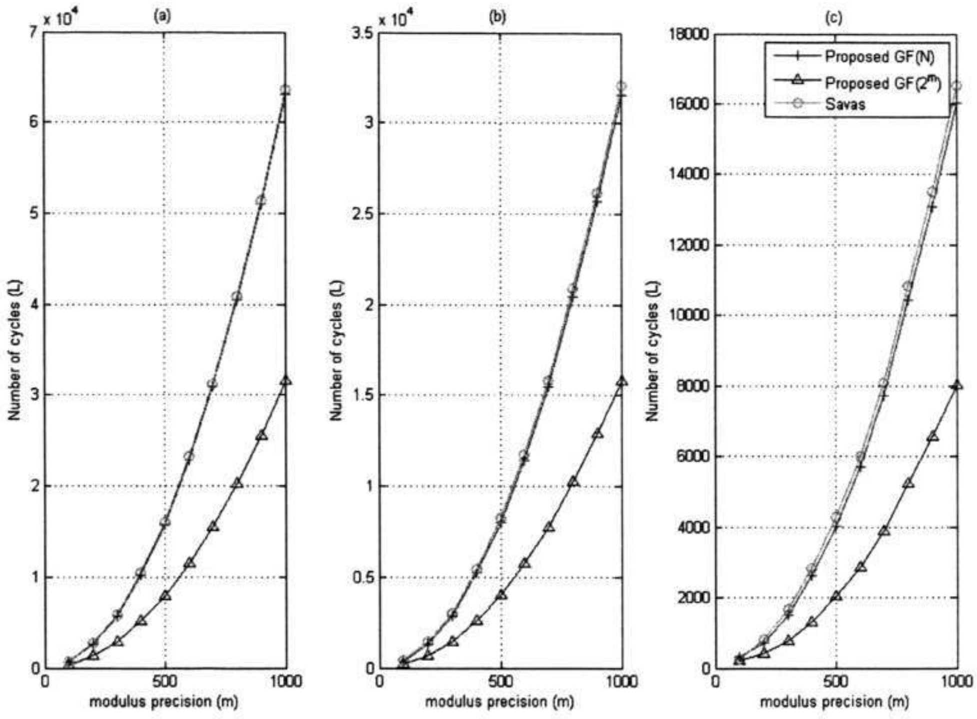


Figure 4.9: Total clock cycles (L) vs. modulus precision (m) for (a) $w = 8$ (b) $w = 16$ (c) $w = 32$

following expressions give the latency of the PU when it is operating in $GF(2^m)$ mode.

$$L_{GF2} = \begin{cases} e\times\lceil\frac{m}{2k}\rceil + 2\times(2k-1) & \text{for } 4k < e \\ 4k\times\lceil\frac{m}{2k}\rceil + e - 2 & \text{for } 4k \geq e \end{cases} \tag{4.6}$$

The latency cycles for wordlengths, w = 8, 16 and 32, of the proposed PU are plotted against the modulus precision, m, in Fig. 4.9 and the results are compared against those of [48]. The results are obtained for a pipelined kernel comprising two PUs. For each w, three curves are plotted. Two of these curves labeled as 'Proposed $GF(N)$' and 'Proposed $GF(2^m)$' are the results of the proposed kernel operating in $GF(N)$ and $GF(2^m)$ modes, respectively. The curve labeled 'Savas' gives the results of the kernel of [48] in either $GF(N)$ or $GF(2^m)$ mode since the latency for this kernel is the same in both modes of operation. The shape of the curves can be explained by the square relationship between

L and m in the expressions stated above.

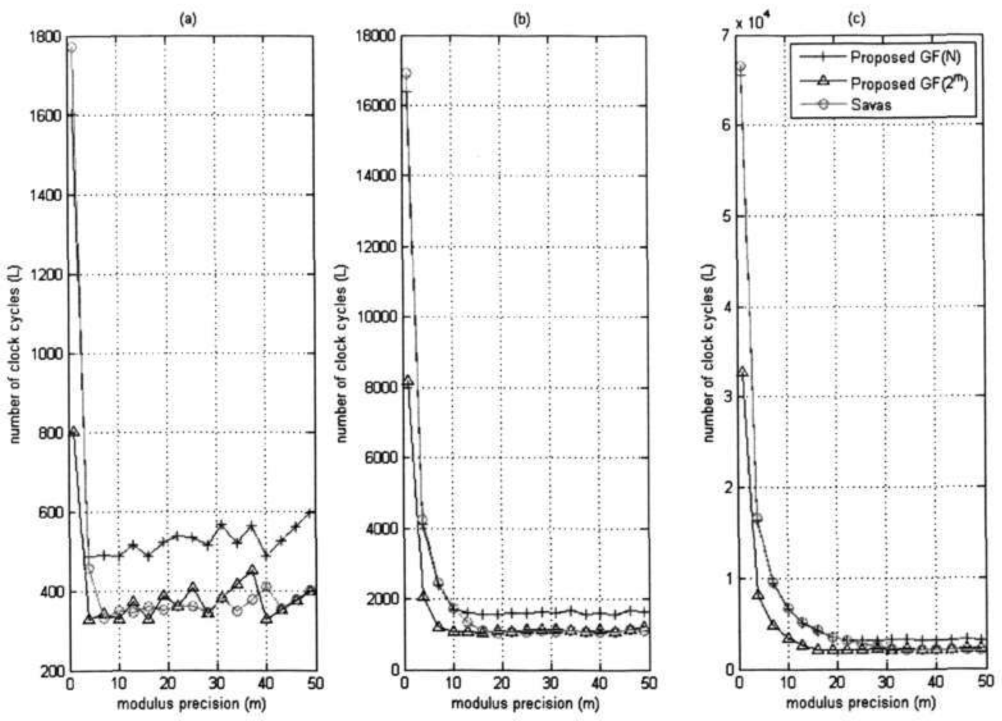


Figure 4.10: Total clock cycles (L) vs. number of PUs (k) for $w = 16$ and (a) $m = 160$ (b) $m = 512$ (c) $m = 1024$

The proposed kernel outperforms that of [48] in the case of $GF(2^m)$. It takes merely half the latency of that of [48] for nearly all values of m. In $GF(N)$, the proposed kernel performs either equally well or slightly better than that of [48]. These results corroborate the claim made in the problem statement of Section 4.3. In [48] two levels of DFAs are used for one iteration in both $GF(N)$ and $GF(2^m)$, but one level of DFAs is sufficient for one iteration in $GF(2^m)$. In the proposed PU, only one of the two levels of adders designated for MMM in $GF(N)$ is used in $GF(2^m)$ operation. This advantage of the proposed PU in $GF(2^m)$ operation is clearly shown in Fig. 4.9. Although there is no clear saving in the number of latency cycles consumed in $GF(N)$ over the kernel of [48], the reduction in total computation time of the proposed architecture will be exemplified later in this section.

The number of latency cycles (L) are plotted against the number of 16-bit PUs (k) in the pipeline for three different modulus precisions, $m = 160$, 512 and 1024 in Fig. 4.10. In this figure, it can be seen that as k increases, L decreases for both architectures. However increasing k would increase the area cost. When $m = 160$, the proposed architecture has lower L as compared to [48] for $k \leq 4$. When m increases to 512 and 1024, the range of k where the proposed architectures perform better extends to 10 and 20, respectively. The proposed PU shows higher savings in latency as compared to [48] when k is small (typically less than 10). The shape of the curve can be explained by the inverse relationship on k but a direct relation on m. As m increases, the factor $\frac{m}{k}$ becomes larger and the effect of m dominates that of k. This results in a steady value of L as m increases in Fig. 4.10.

4.5.2 FPGA Implementation Results

The proposed PU and the PU in [48] are implemented on FPGA platform using Xilinx Synthesis Tool - ISE (version 8.1). The PUs are synthesized on three different Virtex chips (by Xilinx) - VirtexE XVC100E, Virtex2 XC2V1000 and Virtex4 XC4VLX100. All the three FPGAs belong to the same family of Xilinx FPGAs - Virtex. Each configurable logic block (CLB) of these FPGAs comprise the same structure. Each CLB has four slices, where a slice is the basic area unit of an FPGA. Each slice comprises look up tables, multiplexers, carry chain logic and registers. The main difference between the three FPGAs is the process technology and supply voltage requirements. Virtex4 is the most recent of the three, thus having a sub-micron technology. The process technologies of VirtexE, Virtex2 and Virtex4 are 180nm, 150nm and 90nm respectively. Also the supply voltages of the three chips are 1.8V, 1.5V and 1.2V respectively.

Table 4.2 lists the mapping results of the proposed PU and the PU in [48] for three different wordlengths, w =8, 16 and 32. The metrics - critical path delay (t_{min}) in ns and logic utilization in terms of the number of slices (A) are listed. The savings of each metric in % are

Table 4.2: FPGA implementation results of processing unit

8-bit PU						
	$t_{min}(ns)$			A (slices)		
Chip	Prop.	[48]	Savings %	Prop.	[48]	Savings %
Virtex 2	2.73	3.229	15.45	67	69	2.89
Virtex 4	1.911	2.592	26.27	67	45	-48.88
Virtex E	4.724	7.76	39.12	67	45	-48.88
16-bit PU						
Chip	Prop.	[48]	Savings %	Prop.	[48]	Savings %
Virtex 2	2.758	3.11	11.31	130	110	-18.18
Virtex 4	1.929	2.592	25.57	133	90	-47.77
Virtex E	5.324	7.76	31.39	134	90	-48.88
32-bit PU						
Chip	Prop.	[48]	Savings %	Prop.	[48]	Savings %
Virtex 2	3.083	3.139	1.78	264	224	-17.85
Virtex 4	1.911	2.466	22.50	271	184	-47.28
Virtex E	4.884	7.01	30.32	272	184	-47.82

also listed alongside each metric. The savings is calculated as $\left\{\frac{metric_{prop}-metric_{[48]}}{metric_{[48]}} \times 100\right\}$. A positive percentage indicates an improvement of the proposed architecture over that of [48] and vice versa.

t_{min} in Table 4.2 shows that the critical path of the proposed PU is smaller than the PU in [48] for all values of w and all the three Virtex chips. Critical path delay determines the minimum clock period (or the maximum frequency) for the circuit to function. The percentage savings are higher in 8-bit and 16-bit PUs as compared to 32-bit PUs. The critical path is reduced in the proposed PU because it passes through just one level of DFAs as opposed to two levels in [48] (see Section 4.5.1). Also, it can be seen that the proposed architecture performs faster on Virtex4 as compared to Virtex2 and VirtexE. This is attributed to the advanced process technology used in Virtex4 chips.

In terms of logic slice utilization, the proposed PU shows a higher area cost as compared to that of [48]. The increase in area is mainly due to the extra registers that are used to split the two levels of DFAs in the proposed PU. The area cost is less inferior when it is implemented in Virtex2.

4.5.3 Total Computation Time

The actual time taken for one complete multiplication is given by the total computation time. Expressed in ns, it is defined as the minimum clock period (t_{min}) multiplied by the total number of latency cycles (L) as follows:

$$t_{total} = t_{min} \times L \tag{4.7}$$

In Fig. 4.9 of Section 4.5.1, the variation of the total number of cycles (L) with respect to the modulus precision (m) was shown. Similarly, in Fig. 4.10, the variation of L with

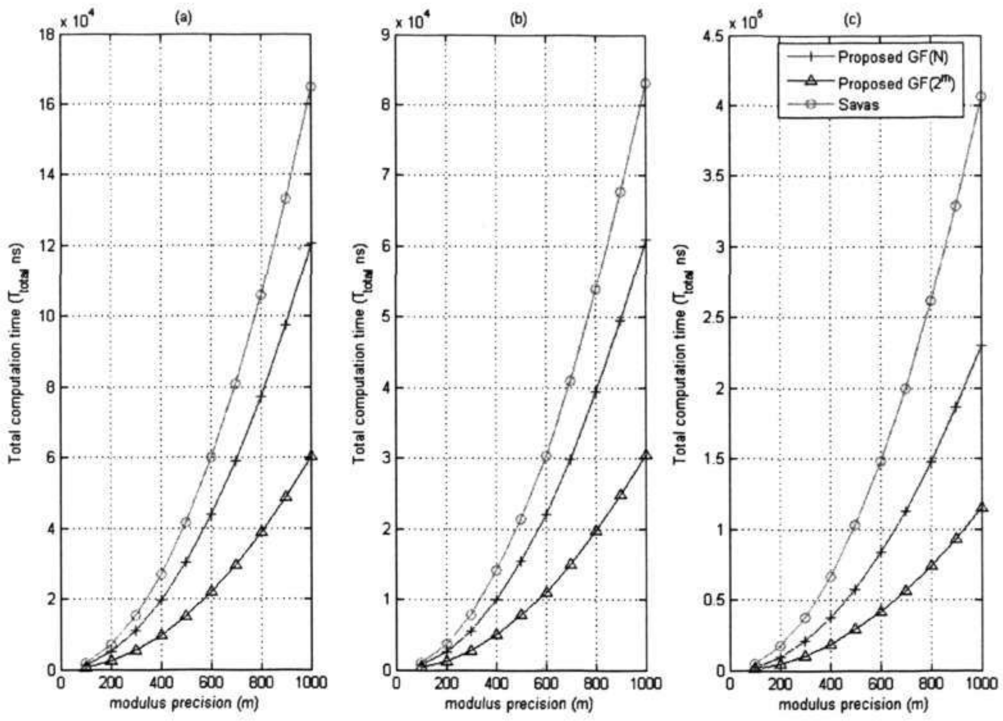


Figure 4.11: Total computation time (t_{total}) vs. modulus precision (m) for (a) $w = 8$ (b) $w = 16$ (c) $w = 32$

respect to the number of PUs, k in the pipeline was shown. The advantage of the proposed PU in terms of L is evident from these graphs. Now, in Fig. 4.11 and 4.12, the combined effect of t_{min} and L on the total computation time is shown. In these graphs, t_{total} is plotted against m and k. The advantage of the proposed PU is further exemplified with the consideration of t_{min}. This shows the timing ascendancy of the proposed architecture. It is now amplified as well in $GF(N)$, especially when the modulus precision is high.

4.5.4 Area Time Product

Area time product is defined as

$$\begin{aligned} AT &= Area \times t_{total} \qquad (4.8) \\ &= Area \times t_{min} \times L \end{aligned}$$

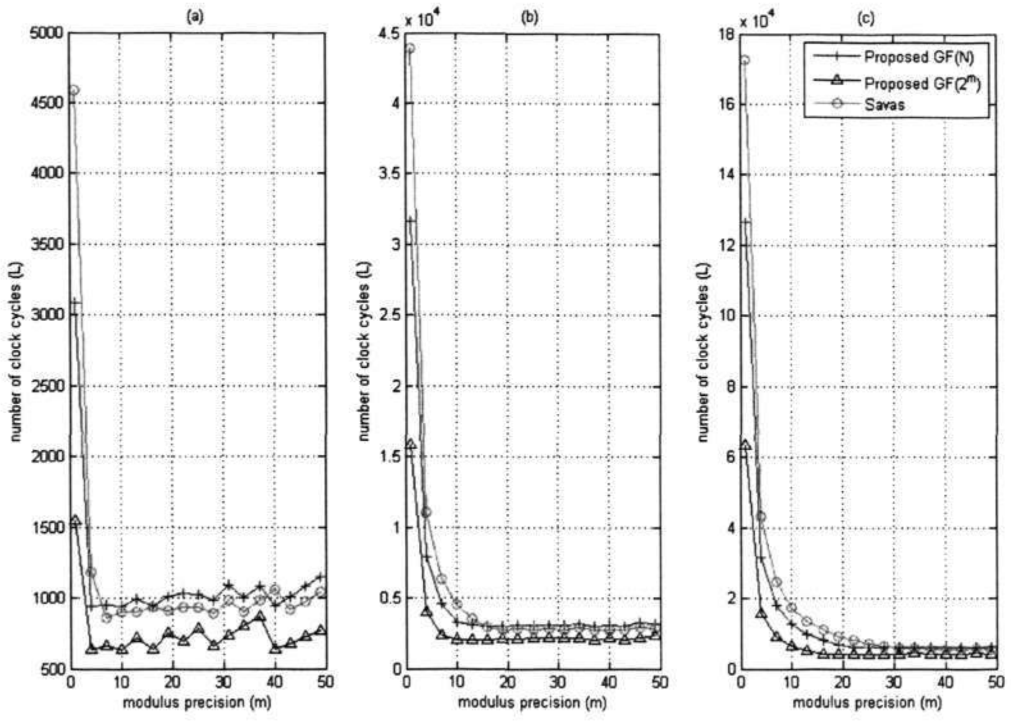


Figure 4.12: Total computation time (t_{total}) vs. number of PUs (k) for (a) $w = 8$ (b) $w = 16$ (c) $w = 32$

Table 4.3: Comparison of Area Time Product ($A \times t_{total}$)

	Proposed $GF(N)$	Proposed $GF(2^m)$	Savas [48]	Savings in $GF(N)$	Savings in $GF(2^m)$
8-bit PU	7617835.68	3836720.16	9393290.16	18.90%	59.15%
16-bit PU	3825013.92	1940309.28	4773288.624	19.86%	59.35%
32-bit PU	2551960.32	1709111.04	2463287.856	-3.59%	30.61%

For the sake of comparison, the area and t_{min} values are taken from Virtex-2 based implementations given in Table 4.2 previously. The latency values of the proposed architecture are calculated using the expressions for latency in Section 4.5.1 and those of [48] are computed using the expressions given in [48]. The AT results are computed for modulus/field order of $m = 572$ and 8 PUs in the pipeline. Table 4.3 lists the AT values for PUs of three different PUs - 8, 16 and 32 bit. AT values for the proposed architecture are different for the two fields $GF(N)$ and $GF(2^m)$ because of the different latencies in the two fields. The multipliers in [48], however, give the same AT values for both the fields.

The advantage of the proposed architectures can be clearly seen in the savings' columns. In both $GF(N)$ and $GF(2^m)$, the proposed architectures show tremendous savings for 8-bit and 16-bit PUs. In the case of 32-bit PUs, the savings is negative in the case of $GF(N)$ whereas in $GF(2^m)$ the proposed architectures still show 30% savings. The drop in savings can be attributed to higher cell complexity of the proposed multiplier, especially in terms of registers.

4.5.5 Discussion

These results show that the proposed PU is faster than the PU of [48]. However there is premium on area cost due to the addition of registers between two levels of DFAs in the PU. When the proposed PU operates in $GF(N)$, it takes equal amount of time to compute the modular product as compared to PU in [48]. However, the advantage of the proposed PU is exemplified when it operates in $GF(2^m)$. It outperforms [48] for all values of field orders by nearly two times. Thus the proposed architectures are well suited for fast generalized Montgomery modular multiplication where area is not a constraint.

4.6 Summary

In this chapter, a new scalable and pipelined architecture is described which uses the proposed novel processing unit and dual field adder for unified Montgomery Modular multiplication. The novelty stems from the efficient use of the XOR gates in the full adder for $GF(2^m)$ operation. This helps to reduce the minimum clock period and hence the total time of computation for $GF(2^m)$ without slowing down the speed of computation in $GF(N)$. The analytical relationship between the latency of computation in both fields and the modulus precision, input wordlength as well as the number of pipeline stages (or processing units) is derived. The processing unit was implemented on FPGA and the tradeoffs for higher speed to area utilization were analyzed against existing processing unit. In comparison with the reported architecture, the speedup ratio in $GF(2^m)$ is nearly double. It was also observed that in $GF(N)$ mode, the proposed architecture is faster particularly for high modulus precision.

Montgomery modular multiplication has been discussed in detail. In the next chapter, the LSB-first/MSB-first modular multiplication algorithms will be studied for generic elliptic curves. Novel serial modular multiplication methods will be explored to improve the timing and area of the existing generalized multipliers.

Chapter 5

LSB-first/MSB-first Multipliers for Generic Curves

5.1 Introduction

In Chapter 3 and 4, Montgomery modular multiplication was discussed for different operating fields, like $GF(2^m)$ and $GF(N)$, and different cryptosystems, like RSA and ECC. A scalable and unified architecture was discussed in Chapter 4 that can accommodate different fields and varying modulus precisions or field orders. In this chapter, LSB-first and MSB-first modular multiplication algorithms are investigated for ECC operating in $GF(2^m)$.

In Chapter 2, a detailed literature survey shows that LSB-first and MSB-first modular multiplication algorithms in $GF(2^m)$ are generally implemented using systolic/semisystolic arrays [13, 17–20, 51]. The systolic arrays are either bit/digit-serial [13, 19, 20, 51] or parallel [13,17,18,51]. All these architectures are designed for a fixed field order m which leads to inflexible applicability of the architecture. In [18], a pipelined parallel LSB-first/MSB-first modular multiplier is designed for a large field order M and the architecture is generalized so that it can be programmed to operate in any field order $m \leq M$. The programming is done by generating vectors that run through a long chain of M processing elements for each iteration of the algorithm, resulting in a long critical path.

In this chapter, two new fast serial-in parallel-out finite field LSB-first and MSB-first modular multipliers are presented that can operate for any field order $m \leq M$, M being the maximum field order. The limitations of the existing generalized LSB-first/MSB-first multipliers in [18] are studied and simple switch based methods to overcome these limitations are proposed. The proposed methods reduce the long critical path delays t_c in [18] and reduce the area tremendously.

The rest of the chapter is organized as follows. The fundamental LSB-first and MSB-first algorithms for a fixed field order m and their data dependencies are first discussed in Section 5.2. In this section, existing LSB-first and MSB-first multipliers in [18] for any generic field order are reviewed. The issues with the architectures in [18] are discussed in Section 5.3 and the problem statement is formulated. The proposed method and architectures are presented in Section 5.4. The hardware complexity of the proposed architectures is evaluated against [18] in Section 5.5. The chapter is concluded in Section 5.6.

5.2 Preliminaries

In Chapter 2, the LSB-first and MSB-first algorithms were explained in detail. Let us briefly revisit the two algorithms for a fixed field order m and discuss the algorithm presented in [18] that generalizes the algorithms to any field order $m \leq M$.

5.2.1 Fixed Order Algorithms

The inputs to the LSB-first and the MSB-first modular multiplications in $GF(2^m)$ are represented in polynomial basis as shown below.

$$\begin{aligned} a(x) &= \textstyle\sum_{i=0}^{m-1} a_i x^i \quad \text{for } a_i \in \{0,1\} \\ b(x) &= \textstyle\sum_{i=0}^{m-1} b_i x^i \quad \text{for } b_i \in \{0,1\} \\ f(x) &= \textstyle\sum_{i=0}^{m} f_i x^i \quad \text{for } f_i \in \{0,1\} \end{aligned} \tag{5.1}$$

$a(x)$ and $b(x)$ are the multiplicand and the multiplier and $f(x)$ is an irreducible polynomial that defines the finite field $GF(2^m)$ of the arithmetic operation. It is used in the reduction operation during modular multiplication.

The LSB-first and MSB-first modular multiplication for the product of two polynomials - $a(x)$ and $b(x)$ in a field defined by the reduction polynomial $f(x)$ are formulated as shown below.

LSB-first modular multiplication:

$$\begin{aligned} c(x) &= a(x)b(x) \, mod \, f(x) \\ &= b_0 a(x) + b_1[a(x)x \, mod \, f(x)] + b_2[a(x)x^2 \, mod \, f(x)] + \cdots \\ &\quad + b_{m-1}[a(x)x^{m-1} \, mod \, f(x)] \end{aligned} \tag{5.2}$$

MSB-first modular multiplication:

$$\begin{aligned} c(x) &= a(x)b(x) \, mod \, f(x) \\ &= \{\cdots[a(x)b_{m-1}x \, mod \, f(x) + a(x)b_{m-2}]x \, mod \, f(x) + \cdots \\ &\quad + a(x)b_1\}x \, mod \, f(x) + a(x)b_0 \end{aligned} \tag{5.3}$$

(5.2) and (5.3) are defined for the polynomial operands of a fixed field order, m. As the names suggest, one of the input operand is read from the least significant bit (lsb) in the LSB-first algorithm and from the most significant bit (msb) in the MSB-first algorithm.

These fixed order algorithms are implemented as either two-dimensional or one-dimensional systolic arrays in [13, 17–20, 51]. The bit-level algorithms to implement them are listed in Algorithms 13 and 14. If a parallel two-dimensional systolic array is considered for the implementation, each iteration can be represented by a column of m processing elements. The iteration number, i can be used to refer to a column of parallel processing elements that compute the results for the i-th iteration. The index, j of a processing element identifies its position in a column of the array. Thus, $t_j^{(i)}$ refers to the j-th bit of an intermediate vector T in the i-th iteration. In the systolic array, it refers to the intermediate value t from the j-th processing element located in the i-th column. Henceforth, unless otherwise specified, t_j is akin to the j-th bit of the word T expressed in binary notation, where the least significant bit, designated as the 0-th bit, is t_0.

Algorithm 13 LSB-first bit-level algorithm for fixed field order

```
1: Input: $a(x)$, $b(x)$, $f(x)$
2: Output : $c(x) = a(x)b(x) \ mod \ f(x)$
3: $t_j(0) = 0$ for $0 \leq j \leq m-1$
4: $a_{-1}^{(i)} = 0$ for $0 \leq i \leq m-1$
5: $a_j^{(0)} = 0$ for $0 \leq j \leq m-1$
6: for $i = 1$ to $m$ do
7:     for $j = 0$ to $m-1$ do
8:         $a_j^{(i)} = a_{j-1}^{(i-1)} + a_{m-1}^{(i-1)} f_j$
9:         $t_j^{(i)} = a_j^{(i-1)} b_{i-1} + t_j^{(i-1)}$
10:    end for
11: end for
12: $c(x) = t^{(m)}(x)$
```

5.2.2 Problem of Field Order Generalization

Both algorithms 13 and 14 are applicable for a fixed field order, m. There is a critical data dependency that limits them from being adaptable to any arbitrary field. In the LSB-first algorithm, each iteration computes two m-bit words - $A^{(i)}$ in `Line 8` and $T^{(i)}$ in `Line 9`. In `Line 8`, the msb of A from the $(i-1)$-th iteration, i.e. $a_{m-1}^{(i-1)}$-th bit, is used to compute the vector, A in the i-th iteration. Similarly, in the MSB-first algorithm, the computation

Algorithm 14 MSB-first bit-level algorithm for fixed field order

```
1: Input: $a(x)$, $b(x)$, $f(x)$
2: Output : $c(x) = a(x)b(x)\ mod\ f(x)$
3: $t_j^{(0)} = 0$ for $0 \leq j \leq m-1$
4: $t_0^{(i)} = 0$ for $1 \leq j \leq m$
5: $t_{-1}^{(i)} = 0$ for $0 \leq i \leq m-1$
6: for $i = 1$ to $m$ do
7:    for $j = m-1$ to $0$ do
8:       $t_j^{(i)} = t_{m-1}^{(i-1)} f_j + b_{m-i} a_j + t_{j-1}^{(i-1)}$
9:    end for
10: end for
11: $c(x) = t^{(m)}(x)$
```

of $T^{(i)}$ in `Line 7` contains a dependency on the msb of $T^{(i-1)}$.

It can be inferred that the field order plays a vital role in the computation of intermediate iterations in both algorithms. The result from the current iteration is dependent on the *most significant bit* of the m-bit words, $A^{(i-1)}$ in the LSB-first implementation or $T^{(i-1)}$ in the MSB-first implementation. The most significant bit is the $(m-1)$-th bit of these words where m is the field order. In a multiplier designed for a fixed field order m, it is straight forward to propagate the msb to the next row of computing elements.

Let us now define a generalized multiplier. It is a multiplier that is designed for a large field order M which can be programmed to operate in any field order $m \leq M$ with low VLSI area and timing overheads. Unlike the fixed order multipliers, all the intermediate results in a generalized multiplier are M bits long. The $(m-1)$-th bit of these intermediate M-bit results needs to be selected correctly in every iteration. In the generalized multipliers, the msb of any intermediate vector is not the $(m-1)$-th bit required in the next iteration. The $(m-1)$-th bit detection and selection from the M-bit intermediate vectors in every iteration poses difficulty in the design of fast and low complexity generalized multipliers for the LSB-first and MSB-first algorithms.

5.2.3 Existing Generalized LSB-first/MSB-first Multipliers

The existing generalized LSB-first and MSB-first modular multipliers in [18] were simulated and the results were analyzed in the literature survey chapter of this thesis (see Chapter 2). In [18], Algorithms 13 and 14 are generalized to cater for any arbitrary field order $m \leq M$ where M is the maximum field order that can be programmed. Each of the m-bit inputs, $a(x)$, $b(x)$ and $f(x)$, are padded with zeros in the most significant positions to make them M-bit words, A, B and F, respectively. An M-bit word H, is precomputed as shown below using F. All the bits of H are set to '0' except the m-th bit, h_m, which is set to '1' by detecting the first occurrence of '1' in F. This determines the actual field order, m. This vector is then used in the generalized multiplication algorithms.

$$\begin{aligned} h_j &= \overline{p_{j-1}} \cdot f_j, \quad 0 \leq j \leq M-1 \ \textit{and} \ s_{-1} = 0 \\ p_j &= p_{j-1} \oplus f_j, \quad 0 \leq j \leq M-1 \ \textit{and} \ p_{-1} = 0 \end{aligned} \tag{5.4}$$

Let us review the generalized MSB-first multiplication algorithm of [18] to study its limitations. In `Iteration` i, the following computations take place.

$$\begin{aligned} y_j &= t_{j-1}^{(i-1)} \cdot h_j + y_{j-1} \\ t_j^{(i)} &= (t_{j-1}^{(i-1)} \oplus y_j \cdot f_i \oplus a_i.b_{M-i}).\overline{h_i} \end{aligned} \tag{5.5}$$

where $0 \leq i \leq M-1$, $0 \leq j \leq M-1$, $T^{(-1)} = 0$, $y_{-1} = 0$ and $c(x) = T^{(M-1)}$. From (5.5), the computation of the intermediate variable $t_j^{(i)}$ is dependent on y_j. y_j and h_j are used in (5.5) to select $t_{m-1}^{(i-1)}$ from the intermediate vector T. More importantly, y_j is computed using y_{j-1}. In [18], (5.5) is implemented as a parallel pipelined architecture. Each column of processing units computes just one iteration. Due to the dependency between y_j and y_{j-1}, a long chain of $M-1$ OR gates are connected end to end from one processing unit to the next in each column. Similar issue is also observed in the implementation of the generalized LSB-first algorithm of [18].

5.3 Problem Statement

The end to end connection of OR gates due to the Y computation in each iteration in the existing generalized algorithms [18], results in a long critical path delay, $t_c = (M-1)\Delta_{OR}$. The critical path delay is defined as the longest combinational delay between two registers in the architecture. It determines the maximum operating frequency of the generalized multiplier. Thus the size of the generalized multiplier matters as t_c now depends on M regardless of the field order. Parallel architectures are adopted in [18] to take advantage of pipelining. Their parallel implementations incur an impractically high area overhead. Generally, pipelining reduces the latency for multiple multiplications after the pipeline is filled by the first computation. However, the architectures in [18] are designed such that the pipeline is effective only if one of the input operands remains the same in subsequent field multiplication operations (FM).

Consider the pipelined parallel generalized MSB-first multiplier of [18]. It comprises M pipeline stages. Each stage computes a M-bit vector T using two M-bit vectors, A and P and an one-bit input b_i where $0 \leq i \leq (M-1)$. Since only one bit of the input operand, B is computed at a time in any stage, even if the new values of A and P input vectors are ready, the next input operand B cannot be fed until all M bits of the operand B for the current FM operation have entered the pipeline. In this case, the pipeline has to wait for $M+1$ cycles for the current operation to complete before it can commence the next FM operation. Only under the special condition that B remains constant for all consecutive FM operations to be computed, A and P can enter the pipeline continuously to produce one modular product per clock cycle after the first product is generated. Unfortunately, the inputs to successive FM operations for point multiplication (PM) operation during encryption/decryption are always different [25]. Consequently, the latency of the architectures in [18] is always $M+1$ cycles.

The investment on parallel architecture to pipeline the operations might not benefit as much because of the limitation posed by FM. In the subsequent sections, a method to overcome the above mentioned issues with these architectures will be discussed. Novel serial-in parallel-out one-dimensional array architectures will be developed that show tremendous cost savings in area. Besides, it can operate at higher clock speeds. To the best of our knowledge, this is the first time programmable serial-in parallel-out generalized LSB-first/MSB-first multipliers being reported.

5.4 Proposed Method and Architectures

The main issue with the generalized architectures in [18] is that in each iteration a vector Y is computed to select the $(m-1)$-th bits of the intermediate vectors. The critical paths are mainly due to the propagation of the value of y_j in (5.5) from the first processing element in each column to the last. This data dependency has to be eliminated in order to make the critical path independent of M.

In the proposed methods, the dependency on Y is eliminated by using the H vector generated in (5.4). As mentioned earlier, all the bits in H except h_m are set to '1'. If H is shifted right by 1 bit, we obtain the vector S where only $s_{m-1} = 1$. The computation of S is a one time operation only that is done at the beginning of the point multiplication operation as the field order m remains the same for the entire point multiplication operation. During field multiplication, S that is stored can be used to select the $(m-1)$-th bit of the intermediate vector as shown below.

$$a_{sw} \quad = \quad a_j^{(i-1)} \; if \; s_j = 1 \tag{5.6}$$

where j runs from 0 to $M-1$. a_{sw} is computed in every iteration using the vector A from the $(i-1)$-th iteration. The LSB-first multiplication for any generic field $m \leq M$ can be accomplished by using a_{sw} as shown below. In `Iteration` i, the following operations take place.

$$t_j^{(i)} = a_j^{(i)} \cdot b_i \oplus t_j^{(i-1)} \tag{5.7}$$

$$a_j^{(i)} = a_{j-1} \oplus a_{sw} \cdot f_j \tag{5.8}$$

where $0 \leq i \leq m-1$, $0 \leq j \leq M-1$ and $T^{(-1)} = 0$. At the end of the iteration, $A^{(i)}$ is left shifted by one bit.

$$A^{(i)} \ << \ 1 \tag{5.9}$$

Equations (5.7) and (5.8) can be implemented in a serial-in parallel out fashion using a one-dimensional array of processing elements. In this array, i is the iteration index for the bit-serial input, $b(x)$. The subscript j is used to locate the j-th processing element. In (5.6), since only $s_{m-1} = 1$, the $(m-1)$-th bit of $A^{(i-1)}$ is assigned to a_{sw} which is used in (5.8) to compute the vector $A^{(i)}$.

The MSB-first multiplication can also be similarly formulated using S as shown below. S is used to select the $(m-1)$-th bit of T in this case.

$$t_{sw} = t_j^{(i-1)} \ if \ s_j = 1 \tag{5.10}$$

t_{sw} is then used in `Iteration` i to compute $T^{(i)}$ as shown below.

$$t_j^{(i)} = t_{sw} f_j + b_{m-i} a_j + t_j^{(i-1)} \tag{5.11}$$

After the vector $T^{(i)}$ has been computed, it is left shifted by one bit, i.e., $T^{(i)} << 1$.

Equations (5.6) and (5.10) can be implemented using M parallel switches that are tied to one output signal a_{sw} or t_{sw}. Using this switch array, the LSB-first/MSB-first multiplications can be generalized for any field order up to a maximum field order of M without the need to propagate the control vector, Y all the way through M processing elements as in [18]. However, there are still intriguing architectural issues pertaining to the latency, critical path delay and the technology mapping to standard cell libraries in its implementation. These issues will be addressed to arrive at an efficient parametric architecture for the generalized field order modular multiplication. For ease of exposition, the generalized modular multiplication architecture is divided into two parts - the multiplier block and the switch array abbreviated as the S-array.

5.4.1 Multiplier Block

For the discussion of the multiplier block architecture, it is assumed that two vectors, S and P are available. The generation of these vectors by the S-array block will be discussed later.

Signal Flow Graphs

The signal flow graphs (SFGs) of the proposed algorithms are shown in Fig. 5.1. For the LSB-first algorithm in Fig. 5.1(a), each circle represents a computation element that executes (5.7) and (5.8) in each iteration. The input operand, $b(x)$ is read into the processing elements serially along with the M-bit parallel inputs, A, F and T. In every iteration, the array produces two M-bit intermediate vectors, A and T. A bank of M switches are parallelly controlled by the vector S. In any iteration, only one switch is activated by $s_{m-1} = 1$ to produce $a_{sw} = a_{m-1}$. The value of a_{sw} is then simultaneously broadcast to the entire linear array. The current outputs of A and T are also registered and fed back to the processing elements via registers or flip flops (FFs) represented by the symbol $\bullet$ in Fig. 5.1 for the next iteration. The SFG of the MSB-first architecture can be similarly explained and it is shown in Fig. 5.1(b). In MSB-first architecture, the signal being selected in the switch array is t_{m-1}. In addition, only one intermediate vector, T needs to be registered and fedback for each iteration.

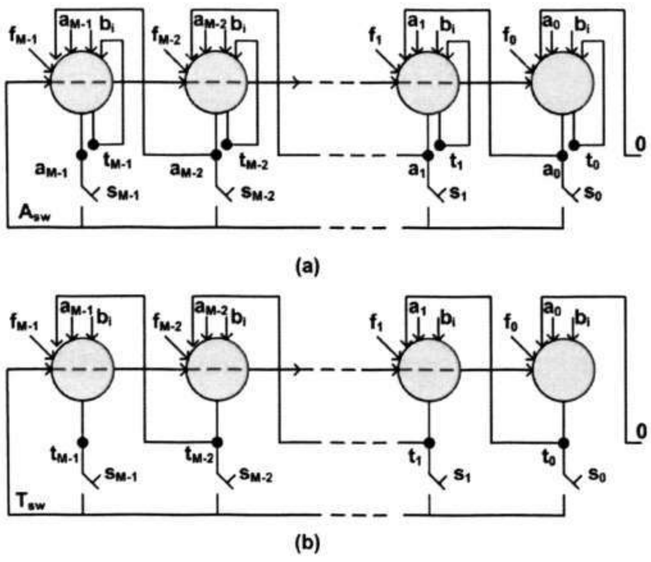


Figure 5.1: Signal flow graphs (a) LSB-first multiplier (b) MSB-first multiplier for $GF(2^M)$

Implementation of SFGs

The SFGs shown in Fig. 5.1 are a direct translation of Equations (5.6)-(5.11). The switches in these SFGs are mapped to tristate buffers from the standard cell libraries by the silicon compiler in an application-specific integrated circuit (ASIC) design flow. The direct implementation of the proposed SFGs for $M = 256$ using Synopsys Design Compiler (DC) failed to optimize the design on Sun Solaris 8 system, even after 86 hours of synthesis time using TSMC 0.18 μm standard cell library for 1.8V power supply.

On detailed analysis by different experimentation, it was found that a fan-in problem occurs in the implementation. DC is able to synthesize the same architectures with $M = 128$.

The difference between the two cases, i.e. when $M = 256$ and $M = 128$, is the fan-in to the common bus, a_{sw} in LSB-first or t_{sw} in MSB-first. If LSB-first multiplier is considered, with $M = 256$, the 1-bit signal bus, a_{sw}, is driven by 256 tristate buffers. When $M = 128$, the fan-in to a_{sw} is reduced to just 128. The library that was employed does not seem to allow such high fan-in values. Besides TSMC 0.18 μm library, CMP 90 nm digital library was are tried. Synopsys DC failed to synthesize the circuit using this library. This, however, was not a problem when a Synopsys custom library called class.db was used for the synthesis because a multiplier with $M = 572$ could be synthesized effectively.

The synthesis problem is resolved by a multiplexer based bus architecture. This is explained using the LSB-first multiplier but it is equally applicable to the MSB-first multiplier. The PE array is apportioned into p parallel processing blocks (PBs) of M/p PEs. Each PB generates a ternary output, $a_{swk} \in \{0, 1, Z\}$ for $k = 0, 1, .., p-1$ where Z stands for a high impedance state in tri-state buffer. Only one of the p sections will generate a valid binary (0 or 1) a_{sw} signal and the remaining outputs will be in high impedance state. The a_{sw} is fedback to the PE array through the p-input multiplexor using the control vector P. This vector determines which of the p PBs contains the $(m-1)$-th bit. This control vector is generated along with S.

Since the maximum field order for an ECC in $GF(2^m)$ listed in NIST is 572 [37], with $M = 572$, $p = 4$, DC is able to synthesize and map the design to TSMC 0.18 μm standard cell library without problem using a minimal 4-input multiplexor and a two-bit control vector P. P selects the valid a_{swk} from the four PB and assigns it to a_{sw} thus generating a_{m-1}. Fig. 5.2 shows the architecture of the LSB-first algorithm for a maximum field order of 572. Fig. 5.3(a) shows a single processing element (PE). There are 143 such PEs in each of the four PBs of Fig. 5.2. Each PB generates 143-bit wide vectors $t_{571:429}, \ldots, t_{143:0}$ and $a^{(i-1)}_{571:429}, \ldots, a^{(i-1)}_{143:0}$. The output vector T is registered in a bank of registers and fed back to the PB for the next iteration. According to Equations (5.6)-(5.8), the vectors, A and S are fed to the field select blocks (FSBs). Each PB is connected to a FSB of 143 field selectors (FSs). Fig. 5.3(b) shows the architecture of a single FS cell. The selector logic in each FS cell of the LSB-first multiplier is explained as follows. In the first iteration, $i = 0$, according to (5.6), the input to the PE array is the operand, $a(x)$ whereas in the subsequent iterations, i.e., for $i > 0$, the vector A from (5.8) of the $(i-1)$-th iteration is fed into the array. Therefore, an OR gate is used in each FS cell to propagate either a_j or $a_j^{(i-1)}$. $a_j^{(i-1)}$ is obtained from the computation of (5.8) in the processing array whereas a_j is an one time input from $a(x)$. A simple external logic circuit is used to set a_j to the j-th bit of $a(x)$ only for the first cycle of computation. After the first cycle, a_j is reset to 0 and $a_j^{(i-1)}$ is assigned to a_{sj} in the FS. The signal a_{sj} is sent to the corresponding PE for the next iteration. It is also tristate buffered to a_{swk} at the same time. The four FSBs generate four single-bit signals - $a_{sw0}, \ldots, a_{sw3}$. For instance, if $M = 572$, when $m = 471$, $a_{sw3} = a_{470}$ and a_{sw0}, a_{sw1} and a_{sw2} are pulled to a high impedance 'Z' state. a_{sw3} is multiplexed into a_{sw} by $P = 11$.

It is however, important to note that three out of the four inputs to the multiplexer are high impedance states in this design abstraction. In the actual ASIC implementation, the high impedance states are resolved and the multiplexer is mapped to standard logic structure. $P(1:0)$ is first converted into four enabling signals and then it is used to select one of a_{sw0}, a_{sw1}, a_{sw2} and a_{sw3} to a_{sw}.

Fig. 5.4 shows the hardware architecture of the proposed MSB-first algorithms for $M = 572$. Fig. 5.5 shows the corresponding processing element and field selector cell. The architecture is analogous to the LSB-first multiplier discussed above. The field selection block is simpler because only the T vector is selected in each iteration.

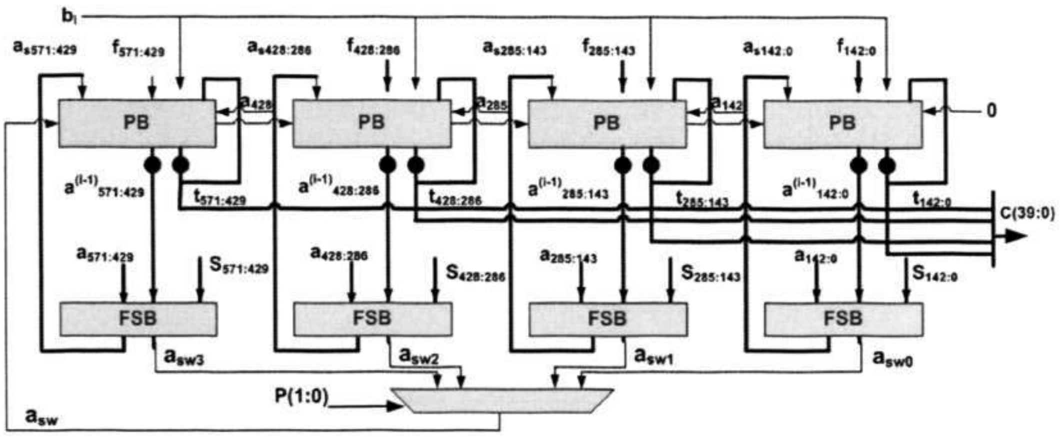


Figure 5.2: Proposed LSB-first multiplier for arbitrary $m \leq 572$

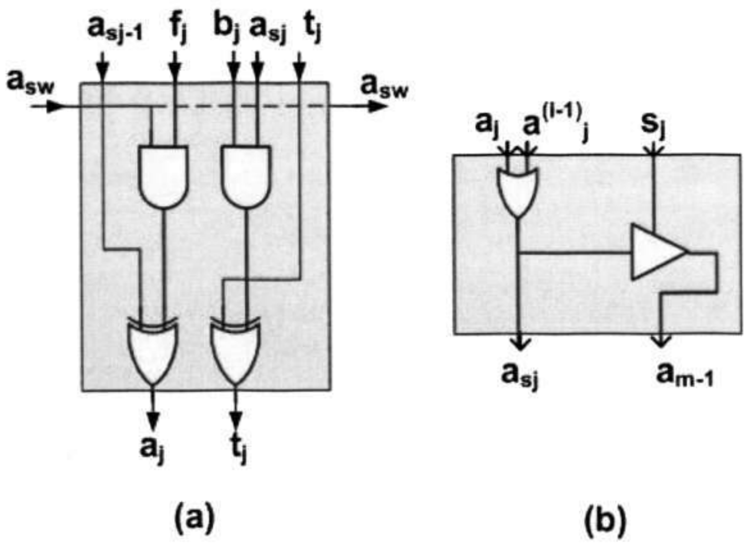


Figure 5.3: (a) Processing element (b) Field selector for the proposed LSB-first mulitplier

5.4.2 The S-Array

The vectors S and P are generated in the S-array. When the H vector from (5.4) is shifted right by 1 bit, it results in S. This logic expression can be implemented using a simple array of AND and OR gates. It is similar to that shown in [18] but the direct implementation as illustrated in [18] creates a long critical path that passes through M 2-input OR gates. The resultant critical path delay is $t_c = (M-1)\Delta_{OR}$. This worst case delay is dependent on the value of M. If the S-array is implemented this way in the proposed architectures, it

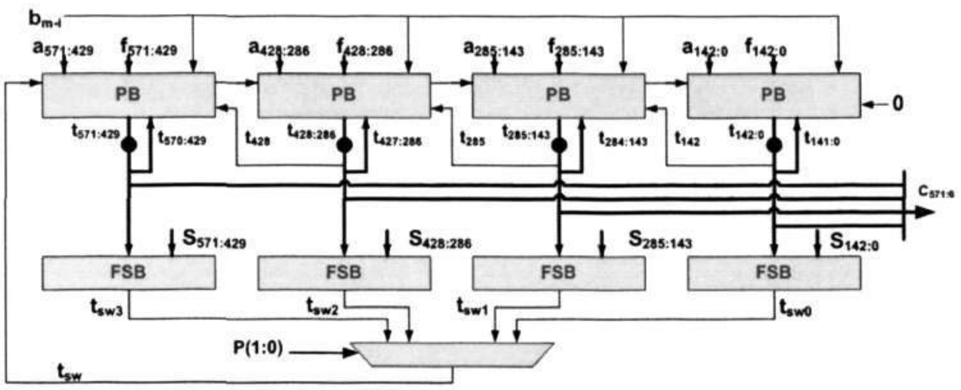


Figure 5.4: Proposed MSB-first multiplier for arbitrary $m \leq 571$

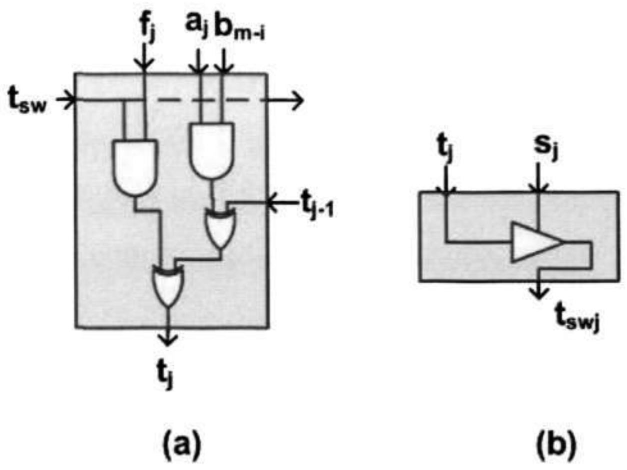


Figure 5.5: (a) Processing element (b) Field selector for the proposed MSB-first mulitplier

turns out to be the timing stumbling block since the multiplier block passes through just one layer of processing elements and is independent of M.

The operating frequency of the entire multiplier is penalized if this long array of OR gates is used to generate S. Introducing registers after every OR gate results in a high latency of M cycles. To reduce t_c of the S-array and make it independent of M, the array of AND and OR gates is optimized such that the critical path of the S-array is just equal to the critical path of the multiplier block. In this way, the latency is also reduced.

The array of AND gates is divided into p equal length blocks. Each block is called the `S-subarray`. For example if $M = 572, p = 4$, every 143 bits are processed in one `S-subarray`. Each `S-subarray` is further divided into smaller units called the `S-cells` with no more than 8 OR gates in each cell. This is because the critical path of the multiplier block is approximately the delay of eight 2-input AND gates. So, it can have a maximum of eight 2-input OR gates connected end to end. In this particular example, the 143-bit `S-subarray` is divided into 17 8-bit and one 7-bit `S-cells` connected serially via registers. In addition to the AND gates that generate S, each `S-subarray` generates a `P-propagate` signal (pp) and a `block select` signal (bs). When m is detected in a `S-subarray`, bs of that `subarray` is set to '1'. Its pp output is also set to 1 to reset the values of bs in the succeeding `S-subarrays`. Once a pp of 1 is received by a `S-subarray`, it resets its bs ouput to zero and propagates the pp signal to the `S-subarray` to its right. When all the bits of F have been read, one of the four bs signals is set to '1'. These signals are used to generate the 2-bit vector P as shown in Fig. 5.6(c).

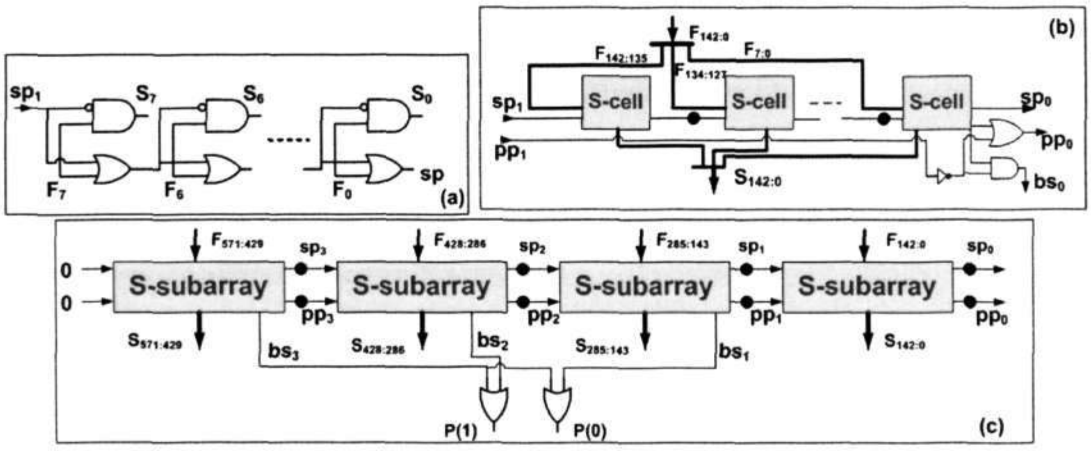


Figure 5.6: (a) Logic in S-cell (b) Logic in S-subarray (c) **S** and **P** vectors generation circuit for $M = 572$

5.5 Results and Discussion

The proposed serial-in parallel-out architectures are compared with the generalized pipelined architectures for the LSB-first and MSB-first modular multiplications discussed in [18]. There are other generic implementations in [8] but they are based on partial reduction methodology wherein the reduction is performed after the multiplication. Our work is the first reported *programmable* LSB-first/MSB-first serial-in parallel-out architectures for generic elliptic curves.

5.5.1 Gate Count

Table 5.1 evaluates the primitive computation elements required for the complexity analysis of the proposed architectures against the existing generalized cellular LSB first/MSB-first architecture of [18]. In Table 5.1, the 'multiplier block' executes the basic operations in each iteration of the algorithms whereas the 'S-array' performs the precomputation to determine the control vectors, S and P for our proposed architectures and it is equivalent to the H computation in [18]. Each basic cell (BC) in the proposed architectures comprises one processing element (PE) followed by one field selector logic (FS). In [18], a BC implements the logical functions of each iteration like those shown in (5.5) for the MSB-first multiplication. The total gate count for the combinational logic is expressed in terms of the number of equivalent 2-input gates (A_2) and inverters (A_1). From TSMC 0.18 library, a 2-input XOR is assumed to have 3 2-input gates and 2 inverters. A 4-to-1 multiplexor is equivalent to 11 2-input gates and 4 inverters. A tristate buffer is assumed to be a 2-to-1 multiplexor comprising 3 2-input gates and 1 inverter. The results in Table 5.1 show that for the highest possible standard field order of 572 bits [37], a tremendous area savings of 99% can be obtained over those reported in [18].

Table 5.1: Complexity analysis

		Proposed		[18]	
		LSB-first	MSB-first	LSB-first	MSB-first
Multiplier core	Number of basic cells (BCs)	M BCs, 1 4-to-1 MUX	M BCs, 1 4-to-1 MUX	M^2 BCs	M^2 BCs
	Basic cell (BC)	2 XOR, 2 AND, 1 OR 1 Tristate buffer, 2 FFs	2 XOR, 2 AND 1 Tristate buffer, 2 FFs	2 XOR, 4 AND, 1 OR, 2 FFs	2 XOR 4 AND, 1 OR, 1 FF
S-array block	Number of cells	$M/8$ S-cells, 2 OR		M cells	
	Cell	9 AND, 1 OR, 2 inverters, 2 FFs		1 AND, 1 OR, 1 inverter	
Total gate count (Combinational)		$\left(\frac{53}{4}M+13\right)A_2+$ $\left(\frac{21M}{4}+4\right)A_1$	$\left(\frac{49}{4}M+13\right)A_2+$ $\left(\frac{21M}{4}+4\right)A_1$	$\left(11M^2+2M\right)A_2+$ $\left(2M^2+1M\right)A_1$	
Latency of multiplier		m cycles		$M+1$ cycles	
Minimum clock period		$9\Delta_{2inp}+1\Delta_{tristate}+\Delta_{interconnect}$		$(M-1)\Delta_{OR}$	

5.5.2 Latency and critical path delay

Latency is defined as the number of clock cycles required to compute a valid output for a given input. The proposed algorithm has an initial latency of L_s clock cycles when it computes S and P. L_s depends on the number of `S cells` that are employed in the `S-subarray`. These vectors are computed only once in the entire encryption/decryption process. In addition to L_s, the multiplier itself possesses a latency of $L_{MM} = m+1$ cycles.

The proposed architectures generate parallel outputs in fewer number of clock cycles than those of [18]. It was shown in Section 5.2.3 that the latency of the pipelined architectures of [18] is always equal to $M+1$ cycles. The pipelining does not improve the latency due to the limitations of the algorithms and architectures itself. In the proposed architectures, one of the inputs $b(x)$, which is an m-bit vector, is received serially. Instead of having to wait for the entire array to be processed for M cycles in [18], the product of the proposed multiplier is available after $m+1$ clock cycles. The initial latency of L_s cycles to setup the S-array is a one-time investment. Upon amortizing over the entire encryption/decryption process, its contribution to the total computation time is negligible. This is because L_s which is a one-time initialization setup time is very small compared to the total computational time required for the entire encryption and decryption operations.

Critical path is the longest combinational delay between two registers. The critical path delay t_c is independent of M and is equal to $\Delta_{tristate} + 9\Delta_{2inp} + \Delta_{interconnect}$, considering $\Delta_{4-to-1-MUX} = 5\Delta_{2inp}$ and $\Delta_{XOR} = 2\Delta_{2inp}$ where Δ is the critical path of a logic gate. One of the main issues with the multipliers of [18], as discussed in Section 5.2.3, is the long t_c of $(M-1)\Delta_{OR}$. In the proposed algorithms, the critical path runs through one processing element and one field select block followed by a 4-to-1 multiplexer. This critical delay path remains the same for all values of M. In addition, the interconnect will add some delay and this would depend on the wire load model of the library. However, the component from interconnect delay would be far less compared to $M-1$ OR gate delay in architectures in [18].

The generalized MSB-first multipliers based on the proposed algorithm and [18] were implemented for $M = 572$ (the largest field order in NIST [37] and SEC [3]). The designs were synthesized using Synopsys Design Compiler v2004.12-SP2 and the TSMC 0.18 μm standard cell library which uses 1.8 V supply. Input and output loads of 0.8 pF and 0.9 pF were applied to both designs and the clock period was set to 50 ns. The synthesis was performed on Sun solaris 8 dual-processor systems having a RAM of 4 GB. Synopsys Design Compiler could not synthesize the parallel architecture of [18] due to the complexity of the design requiring a memory (RAM) of over 4.4 GB. The proposed multiplier was synthesizable and the synthesized circuit has $t_c = 2.81ns$. This shows the complexity and practicality of realizing the fully parallel architectures of [18] for large M.

For the sake of comparison, the two generalized MSB-first multipliers were implemented again for $M = 128$ on the same platform and constraints as stated above. The minimum clock period reported by our architecture and that of [18], on optimizing under similar operating conditions, are $1.98ns$ and $18.7ns$, respectively. That is, our proposed architecture can run at 16.7 times faster clock rate than the architecture of [18]. In order to validate the delay of the proposed architecture with respect to the qualitative results in Table 5.1, a smaller multiplier for $M = 16$ was synthesized. The delay was obtained to be $1.75ns$. This shows that even if M increases to 128 from 16, i.e. 8 times, the delay increases by just 1.13 times. Thus the delay is almost independent of M.

In terms of area, the proposed 128-bit multiplier occupied 64026 μm^2 of silicon area whereas the architecture in [18] covered 3555525.75 μm^2 which is 98% more than the proposed architecture. To validate the qualitative area results in Table 5.1, it was found that the proposed 16 bit multiplier covers 8558 μm^2. On linear approximation of this result, it can be seen that the area of 128 bit multiplier is equal to $\frac{8558}{16} \times 128$ which gives 68464 μm^2. This is close to the actual implementation result, i.e. 64026 μm^2 reported by Design Compiler.

For point multiplication (PM) that executes finite field multiplication (FM) repeatedly, the proposed architectures outperform those of [18] in terms of the total computation time T_{total}. According to [25], a PM involves at least $6\lfloor log_2 k \rfloor + 10$ FMs where the size of k is of the order of m. Thus, T_{total} required in one PM is

$$T_{total} = ((6\lfloor log_2 k \rfloor + 10) \times L_{MM} + L_s) \times t_c \tag{5.12}$$

where L_s are the latency cycles incurred by the S-array only once at the beginning of the PM. If we consider an ECC with $m = 113$ from [3], a multiplier that supports a maximum field order of $M = 128$ would suffice. For this case, $T_{total-proposed-MSB} = 11.4\mu s$ and $T_{total-[18]} = 121.88\mu s$. A speed up by nearly ten times is achieved. The speedup factor increases for higher values of M. In addition to this speed up, the proposed architecture saves logic area by over 96% as compared to its MSB-first counterpart in [18].

5.6 Summary

In this chapter, we discussed novel finite field LSB-first/MSB-first multiplication algorithms for arbitrary field order m for $m \leq M$, M being the highest possible field order. Efficient multiplier architectures were derived for the algorithms by considering the feasibility of implementation of the multipliers using semi-custom design flow with standard cell libraries. On implementing a 128-bit multiplier using TSMC 0.18 μm libraries, a speed up of 16.7 times in operating frequency and an area savings of over 96% was achieved. In addition, superior performance of the proposed multipliers is also seen in the point multiplication operation. Our analysis infers that for an ECC operating in the $GF(2^{113})$ field, the proposed architectures can operate atleast ten times faster than those cited in [18] with an area savings of over 96%.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The work presented in this thesis is aimed at developing improved algorithms and efficient architectures for finite field multiplication in $GF(N)$ and $GF(2^m)$ for encryption/decryption in RSA and elliptic curve cryptosystems. The main focus of the study is on three most commonly used modular multipliers called Montgomery modular multiplication, LSB-first and MSB-first modular multiplication algorithms. New architectures were designed and developed by exploiting the properties of the algorithms. Both FPGA and ASIC platforms were used to evaluate the proposed architectures against existing architectures.

The outcomes of this research work are summarized as follows.

A detailed survey of several important LSB-first and MSB-first modular multipliers for $GF(2^m)$ was conducted. Multipliers with different input-output topologies, like bit-parallel, bit-serial and digit-serial, were analyzed and evaluated qualitatively and quantitatively against each other by implementing them using TSMC $0.18\mu m$ standard cell library. Synopsys Design Compiler and Power Compiler were used to report the three VLSI performance metrics - silicon area, delay and dynamic power dissipation. These metrics were computed for multipliers operating in different fields that are defined by elliptic curve standards' institutions. The different multipliers were analyzed in various perspectives like area vs. delay, area vs. power dissipation, power dissipation vs. delay etc. and the results were consolidated as a designer's chart.

Next, a modified Montgomery modular multiplication algorithm for RSA cryptosystem was proposed. It improves the clock rate by about two times as compared to existing fast Montgomery multipliers by eliminating its data dependency on intermediate control signal. The algorithm was translated into two systolic array architectures - two-dimensional pipelined parallel array and one-dimensional serial-in parallel-out array. The two dimensional array provided Montgomery product every two clock cycles once the pipeline is filled. The one-dimensional serial array was more area efficient than its parallel counterpart but it incurred more latency cycles. The proposed architectures were implemented on FPGA and ASIC platforms and compared against some of the fastest available recently cited Montgomery multipliers. The proposed architectures show improved clock rates and occupy lower logic areas.

Scalability and adaptability to varying field orders and modulus precisions was also considered in this research, leading to the development of pipelined Montgomery modular multipliers that can operate in different fields and cryptosystems. A novel fast processing unit, that can operate in both $GF(N)$ and $GF(2^m)$ to compute Montgomery modular multiplication, was proposed. A modified dual field adder was proposed to reduce the critical path delay of existing unified architecture by removing the logic redundancy in $GF(2^m)$ mode of operation. To accommodate the proposed dual field adders, new dependency graphs were developed. Using the revised dependency graphs and reservations tables, a new processing unit for the pipelined kernel was developed. The proposed architectures reduced the minimum clock period and hence the total time required for computation in $GF(2^m)$ mode without slowing down the speed of computation in $GF(N)$ mode. There is however a tradeoff in area. The latency is analytically formulated in terms of the modulus precision, the number of processing units in the pipeline and the wordlength of the inputs. The proposed processing unit accelerates the $GF(2^m)$ Montgomery multiplication by two times over the existing unified architecture while keeping the same speed of operation as existing unified architecture for $GF(N)$ multiplication.

Last but not the least, our research on the generalized LSB-first and MSB-first modular multipliers in $GF(2^m)$ for arbitrary field orders also matured into novel serial-in parallel-out generalized multipliers that can be adapted to any generic field order m up to and including the maximum field order, M. Switch arrays were designed to select the actual field order to reduce the critical path. Unlike the existing generalized parallel multipliers which have long critical paths dependent on M, the proposed architectures have a short constant critical path delay. There was some library mapping problems, possibly caused by the large circuit fan-in, that prevented the proposed architectures to be directly synthesized using Synopsys Design Compiler with TSMC $0.18\mu m$ standard cell library. The problem was resolved by a smart architectural tweak. The final implementation of the proposed 128-bit multipliers showed a speed up of about 16.7 times over existing architectures with an area savings of nearly 96%.

6.2 Recommendations for Future Work

The research work presented in this thesis has some rooms for further improvement and the scope of study can be extended to advanced research topics. The following potential ideas are recommended for future work.

1. In this research work, the algorithm and architectural refinements are primarily derived and driven from the area and timing perspectives. Power dissipation and redundancy for counter cyptanalysis are also important criteria worth investigating for the proposed architectures. Furthermore, recently reported power based side channel attacks on cryptosystems can be studied to design and develop power efficient and secure arithmetic circuits for PKCs.

2. The architectures that were proposed in this thesis were implemented either on FPGA platform or using semi-custom design flow. Standard cell libraries were used for gate level synthesis. Several interesting optimizations are possible when the modular multiplication algorithms are studied at transistor level. Full-custom design and development of core computational cells based on a rich combination of different logic styles for modular multiplication is certainly a prospective research area.

3. Dynamic reconfiguration in FPGA platforms is another area that cryptoprocessors can avail. Cryptoprocessors can be made reconfigurable by dynamically reconfiguring parts of FPGA with computational blocks that satisfy the user constraints. Based on the area-time-power constraint of the applications, an intelligent mapper can be developed to select the most suitable instances from a pool of architectures to configure an amorphic cryptoprocessor architecture.

List of Author's Publications

1. Ravi Kumar Satzoda and Chip Hong Chang, "Programmable LSB-first and MSB-first Modular Multipliers for ECC in $GF(2^m)$," *IEEE International Symposium on Circuits and Systems 2008*, **(Submitted)**.

2. Ravi Kumar Satzoda, Huy Nguyen Quang and Chip-Hong Chang, "Programmable Montgomery Modular Multiplier for Trinomial Reduction Polynomials in $GF(2^m)$," *Proc. of IEEE International Symposium on Integrated Circuits*, pp. 248-251, Sept. 2007.

3. Ravi Kumar Satzoda, Chip Hong Chang and C. C. Jong, "Throughput and Low Complexity Bit Parallel and Bit Serial Systolic Architectures for Montgomery Modular Multiplication," *WSEAS Transaction on Circuits and Systems*, vol. 5, no. 5, pp. 734-741, May 2006.

4. Ravi Kumar Satzoda and Chip Hong Chang, "VLSI Performance Evaluation of Systolic and Semisystolic Finite Field Multipliers," in *Advances in Computer Systems Architecture, Lecture Notes in Computer Science LNCS 3740*, T. Srikanthan, J. Xue and C. H. Chang, Ed., pp. 693-704, Springer-Verlag, Berlin, Oct. 2005 (Book Chapter)

5. Ravi Kumar Satzoda, Chip Hong Chang and Thambipillai Srikanthan, "Monte Carlo Statistical Analysis for Power Simulation in Synopsys Design Compiler," in *Synopsys Users' Group Conference*, Singapore, Online publication, http://www.snug-universal

.org/ papers/papers.htm, Jun. 2006

6. Ravi Kumar Satzoda and Chip Hong Chang, "High Speed Systolic Montgomery Modular Multipliers for RSA Cryptosystems," in *Proc. 5th WSEAS International Conference on Instrumentation, Measurement, Circuits and Systems (IMCAS 2006)*, pp. 240-245, Apr. 2006.

7. Ravi Kumar Satzoda, and Chip Hong Chang, "A Fast Kernel for Unifying $GF(p)$ and $GF(2^m)$ Montgomery Multiplications in a Scalable Pipelined Architecture," in *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS-2006)*, pp. 3378-3381, May 2006.

8. Ravi Kumar Satzoda,, K. H. Quek and T. Srikanthan, "Analysis of Design Complexity of Datapaths," in *Synopsys Users' Group Conference*, Singapore, Online publication, http://www.snug-universal.org/papers/papers.htm, Jun. 2005

Bibliography

[1] A. A-A. Gutub and M. K. Ibrahim, "High Radix Parallel Architectures for GF(p) Elliptic Curve Crypto Processor," *Proc. Intl. Conf. on Acoustic Signal and Speech Processing*, pp. 625-628, 2003.

[2] L.-S. Au and N. Burgess, "Unified Radix-4 Multiplier for GF(p) and GF(2^n)," in *Proc. of IEEE Intl. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP'03)*, pp. 226-236, 24-26 June 2003.

[3] Certicom Research, "SEC 2: Recommended elliptic curve domain parameters," *Standards for Efficient Cryptography*, Technical document, Sept. 20, 2000.

[4] Z. Cheng, "Simple Tutorial on Elliptic Curve Cryptography," *Technical report*, Jun. 2003.

[5] Y. Ching-Chao, C. Tian-Sheuan and J. Chein-Wei, "A New RSA Cryptosystem Hardware Design Based on Montgomery's Algorithm," *IEEE Trans. on Circuits and Systems – II: Analog and Digital Singal Processing*, vol. 45, no. 7, pp. 908 – 913, July 1998.

[6] A. Daly and W. Marnane, "Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic," in *Proc. of the 2002 ACM/SIGDA 10th Intl. Symp. on Field-programmable Gate Arrays*, pp. 40-49, 2002.

[7] W. Diffie and M. Hellman, "New Directions in Cryptography", *IEEE Trans. on Information Theory*, pp. 644-654, 1976.

[8] H. Eberle, N. Gura, S. Chang-Shantz, A cryptographic processor for arbitrary elliptic curves over $GF(2^m)$, in *Proc. IEEE Intl. Conf. on Application-Specific Systems, Architectures, and Processors*, Hague, Netherlands, pp. 444-454, June, 2003.

[9] Elliptic Curve Cryptosystems: an online tutorial, Certicom research.

[10] A. P. Fournaris and O. Koufopavlou, "Montgomery Modular Multiplier Architecture and Hardware Implementations for an RSA Cryptosystem," in *Proc. of 46th IEEE Intl. Midwest Symp. on Circuits and Systems (MWSCAS 2003)*, vol. 2, pp. 778-781, 27-30 Dec. 2003.

[11] A. P. Fournaris and O. Koufopavlou, "A new RSA encryption architecture and harware implementation based on optimized Montgomery multiplication," in *Proc. of IEEE Symp. on Circuits and Systems (ISCAS 2005)*, pp. 4645-4648, May 2005.

[12] J. M. Gawron, "Groups, Modular Arithmetic, and Cryptography," *Technical Report*, San Diego State University, July 2004.

[13] J.-H. Guo and C.-L. Wang, "Digit-serial systolic multiplier for finite fields $GF(2^m)$," *IEE Proc. Comput. Digit. Tech.*, vol. 145, no. 2, pp. 143-148, Mar. 1998.

[14] A. Halbutogullari, C. K. Koc, "Mastrovito Multiplier for general irreducible polynomials," *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Lecture Notes in Computer Science Series No. 1719*, pp. 198-507, Springer-Verlag, Berlin, 1999.

[15] D. Hankerson, J. L. Hernandez, A. Menezes, "Software Impelementation of elliptic curve cryptography over binary fields," *Cryptographic Hardware and Embedded Systems*, pp. 1-24, Springer-Verlag, Heidelberg, 2000.

[16] M.A. Hasan, M.Z.Wang, V.K. Bhargava, "A modified Massey-Omura parallel multiplier for a class of finite fields," *IEEE Trans. on Computers*, vol. 42, no. 10, pp. 1278-1280, Oct. 1993.

[17] S. K. Jain, K. K. Parhi, "Low latency standard basis $GF(2^m)$ multiplier and squarer architectures," in *Proc. IEEE Intl. Conf. on Acoustic, Speech and Signal Processing (ICASSP-1995)*, Detroit, Michigan, USA, pp. 2747-2750, May 1995.

[18] S. K. Jain, L. Song, K. K. Parhi, "Efficient semisystolic architectures for finite-field arithmetic," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 1, pp. 101-113, Mar. 1998.

[19] C. H. Kim, S. D. Han, C. P. Hong, "An efficient digit-serial systolic multiplier for finite fields $GF(2^m)$," in *Proc. of 14th Annual IEEE Intl. ASIC/SOC Conference*, pp. 361-165, Sept. 2001.

[20] K.-W. Kim, K.-J. Lee, K.-Y. Yoo, "A new digit-serial multiplier for finite fields $GF(2^m)$," in *Proc. of Intl. Conf. on Info-tech and Info-net (ICII 2001)*, Beijing, China, vol. 5, pp. 128-133, Oct. 2001.

[21] C. K. Koc, T. Acar and B. S. Kaliski, Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, vol. 16, pp. 26-33, June 1996.

[22] Ç. K. Koç and T. Acar, "Montgomery Multiplication in GF(2^k)," *Design, Codes and Cryptography*, no. 14(1), pp. 57-69, Kluwer Academic Publishers, Boston, April 1998.

[23] B. A. Laws, C. K. Rushforth, "A cellular-array multiplier for $GF(2^m)$," *IEEE Trans. on Computers*, vol. C-20, pp. 869-874, Nov. 1982.

[24] R. Lidl, H. Niederreiter, *Introduction to finite fields and their applications*, Revised Edition, Cambridge University Press, 1994.

[25] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation," *Lecture Notes In Computer Science, Proc. of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)*, vol. 1717, pp. 316-327, 1999.

[26] J. Lopez and R. Dahab, "An overview of elliptic curve cryptography," *Technical Report*, Institute of Computing, State Uniersity of Campinas, May 2000.

[27] D. S. Malik, John N. Mordeson and M. K. Sen, *Fundamentals of Abstract Algebra*. McGraw-Hill International, 1997.

[28] E.D. Mastrovito, "VLSI designs for multiplication over finite fields $GF(2^m)$," in *Proc. Sixth Intl. Conf. Applied Algebra, Algebraic Algorithms, and ErrorCorrecting Codes (AAECC-1988)*, Rome, Italy, pp. 297-309, July 1988.

[29] A. Menezes, *Elliptic Curve Public Key Cryptography*. Kluwer Academic Publishers, 1993.

[30] N. Mentens, S. B. $\ddot{O}rs$ and B. Preneel, "An FPGA Implementation o f an Elliptic Curve PRocessor over $GF(2^m)$," Proc. of GLSVLSI 2004, pp. 454-457, Boston, Apr. 2004.

[31] V. Miller, "Uses of Elliptic Curves in Cryptography," *Advances in Cryptology - CRYPTO'85*, Lecture Notes in Computer Science, vol. 218, Springer-Verlag, pp.417-426, 1986.

[32] P. L. Montgomery, "Modular Multiplication Without Trial Division," *Math. Comput.*, vol. 44, pp. 519-521, Apr. 1985.

[33] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203-209, 1987.

[34] N. Nedjah and L. de Macedo Mourelle, "Two Hardware Implementations for the Montgomery Modular Multiplication: Sequential versus Parallel," in *Proc. of the 15th Symp. on Integrated Circuits and System Design (SBCCI'02)*, pp. 3-8, Sept. 2002.

[35] N. Nedjah, L. de Macedo Mourelle, "A reconfigurable recursive and efficient hardware for Karatsuba-Ofman's multiplication algorithm," in *Proc. IEEE Intl. Conf. on Control Applications (ICCA 2003)*, Istanbul, Turkey, vol. 2, pp. 1076-1081, June 2003.

[36] N. Nedjah and Ld. M. Mourelle, "Three hardware architectures for the binary modular exponentiation: sequential, parallel, and systolic," *IEEE Trans. on Circuits and Systems - I: Regular Papers*, vol. 53, no. 3, pp. 627-633, Mar. 2006.

[37] National Institute of Standards and Technology, United States of America, http://csrc.nist.gov/csrc/fedstandards.html

[38] G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for $GF(2^m)$," *Proc. of Cryptographic Hardware and Embedded Systems 2000 (CHES 2000), Lecture Notes in Computer Science*, vol. 1965, pp. 41-56, 2000.

[39] S. B. Ors, L. Batina, B. Preneel and J. Vandewalle, "Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array," in *Proc. of Intl. Parallel and Distributed Processing Symp. (IPDPS'03)"*, 8. pp, Apr. 2003.

[40] A. Reyhani-Masoleh, M. A. Hasan, "Low complexity bit parallel architectures for polynomial basis multiplication over $GF(2^m)$," *IEEE Trans. on Computers*, vol. 53, no. 8, pp. 945-958, Aug. 2004.

[41] R.L. Rivest, A. Shamir, and L.M. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120-126, Feb 1978.

[42] N. A. Saqib, F. Rodriguez-Henriquez, A. Diaz-Perez, "A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication over $GF(2^m)$," *Proc of 18th Intl. Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 3*, pp. 144-151, 2004.

[43] A. Satoh and K. Takano, "A Scalable Dual-Field Elliptic Cryptographic Processor," in *IEEE Trans. on Computers*, vo. 52, no. 4, April 2003.

[44] R. K. Satzoda, K. H. Quek and T. Srikanthan, "Analysis of Design Complexity of Datapaths," Synopsys Users' Group Conference, Singapore, June, 2005.

[45] R. K. Satzoda and C.-H. Chang, "VLSI Performance Evaluation and Analysis of Systolic and Semisystolic Finite Field Multipliers," in *Asia-Pacific Computer Systems Architecture Conference*, pp. 693-706, Oct. 2005.

[46] R. K. Satzoda and C.-H. Chang, "High Speed Systolic Montgomery Modular Multipliers for RSA Cryptosystems," in *Proc. 5th WSEAS Intl. Conf. on Instrumentation, Measurement, Circuits and Systems (IMCAS 2006)*, pp. 240-245, Apr. 2006.

[47] R. K. Satzoda, C.-H. Chang and C. C. Jong, "Throughput and Low Complexity Bit Parallel and Bit Serial Systolic Architectures for Montgomery Modular Multiplication," *WSEAS Trans. on Circuits and Systems*, vol. 5, no. 5, pp. 734-741, May 2006.

[48] E. Savas, A. F. Tenca, and Ç. K. Koç, "A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$," *Cryptographic Hardware and Embedded Systems - CHES 2000*, Ç. K. Koç and C. Paar, editors, Lecture Notes in Computer Science, no. 1965, pp. 281-296, Springer Verlag, Berlin, Germany, 2000.

[49] E. Savas, A. F. Tenca, M. E. Çiftçibasi and Ç. K. Koç, "Multiplier Architectures for $GF(p)$ and $GF(2^n)$," *IEE Proc. -Comput. Digit. Tech.*, vol. 151, no. 2, pp. 147-160, Mar. 2004.

[50] B. Schneier, *Applied Cryptography.* 2nd Edition, Wiley 1996.

[51] C.-L. Wang and J.-L. Lin, "Systolic array implementation of multipliers for finite fields $GF(2^m)$," *IEEE Trans. on Circuits and Systems-I*, vol. 38, no. 7, pp. 796-800, July 1991.

[52] T. Zhang, K.K. Parhi, "Systematic design of original and modified mastrovito multipliers for general irreducible polynomials," *IEEE Trans. on Computers*, vol. 50, no. 7, pp. 734-749, July 2001.

[53] Virtex II Datasheets, http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-19283&sGlobalNavPick=&sSecondaryNavPick=

[54] M.A. Hasan and V.K. Bhargava, "Bit-serial systolic divider and multiplier for finite fields $GF(2^m)$," *IEEE Trans. on Computers*, vol. 41, no. 8, pp. 972-980, Aug. 1992.

[55] M.A. Hasan and V.K. Bhargava, "Division and bit-serial multiplication over $GF(q^m)$," *IEE Proc. of Computers and Digital Techniques*, vol. 139, no. 3, pp. 230-236, May 1992.

[56] C.K. Koc and B. Sunar, "Low-complexity bit-parallel canonical and normal basis multipliers for a class of finite fields," *IEEE Trans. on Computers*, vol. 47, no. 3, pp. 353-356, Mar. 1998.

[57] P. Scott, S. Tavares and L. Peppard, "A Fast VLSI Multiplier for $GF(2^m)$," *IEEE Journal on Selected Areas in Communications*, vol. 4, no. 1, pp. 62-66, Jan. 1986.