Copyright

by

Moboluwaji Olusegun Sanu

2005

The Dissertation Committee for Moboluwaji Olusegun Sanu certifies that this is the approved version of the following dissertation:

Parallel Multipliers for Modular Arithmetic

Committee:

____
Earl E. Swartzlander, Jr., Supervisor

____
Jacob A. Abraham

____
Mircea D. Driga

____
Jose F. Voloch

____
Baxter F. Womack

Parallel Multipliers for Modular Arithmetic

by

Moboluwaji Olusegun Sanu, B.S.; M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2005

This dissertation is dedicated to the Almighty God.

Acknowledgments

"*But as for me, this secret is not revealed to me for any wisdom that I have more than any living*" (Daniel 2:30a) – all the glory be to God for His infinite wisdom and knowledge.

I would like to thank my supervisor, Prof. Earl E. Swartzlander, Jr., for his constant support. He guided me with keen insight and kind understanding during the course of my research. His unassuming attitude in spite of his immense stature in his field of expertise has left an indelible impression on me. I would also like to thank Prof. Jacob Abraham, Mircea Driga, Jose Voloch, and Baxter Womack for finding time to serve on my dissertation committee and providing valuable suggestions to improve this dissertation. I also acknowledge all the support and guidance of Prof. Craig Chase during the course of my graduate study.

Appreciation also goes to the staff of the Graduate Office – Melanie Gulick and Michelle Belisle for all their help. I'm grateful to my family - parents, sister, both of my brothers and the bride of my youth - for all their encouragement and support.

Parallel Multipliers for Modular Arithmetic

Publication No. ______________________

Moboluwaji Olusegun Sanu, Ph.D.
The University of Texas at Austin, 2005
Supervisor: Earl E. Swartzlander, Jr.

Modular multiplication is a core operation in virtually all public-key cryptosystems in use today. In this research, parallel, high-speed designs for modular multiplication are presented. These high speed designs take advantage of the transistor bounty provided by Moore's law and the continuously diminishing average cost of a transistor. In addition, advances in Computer-Aided Design (CAD) synthesis are leveraged to explore designs that are otherwise difficult to manually layout. Novel parallel algorithms and high-speed multipliers for prime and extension Galois fields are presented in this work. A tool is developed that automatically generates Hardware Description Language (HDL) code for the various designs. Simulation is used to evaluate the area and delay complexities of all the designs.

Table of Contents

Acknowledgments v

Abstract vi

List of Tables x

List of Figures xi

Chapter 1. Introduction 1

1.1 Motivation ... 1

1.2 Mathematical background and notation ... 6

1.3 Contribution and dissertation overview ... 8

Chapter 2. Conventional Multiplication Algorithms and Architectures 10

2.1 Bit-serial multipliers ... 11

2.2 Serial-parallel multipliers ... 11

2.3 Parallel multipliers ... 12

Chapter 3. Modular Multiplication Algorithms 13

3.1 Related work ... 13

3.1.1 Classical modular multiplication algorithm ... 14

3.1.2 Montgomery algorithm ... 15

3.1.3 Barrett's algorithm ... 17

3.1.4 Algorithms for special moduli ... 17

3.2 Contributions ... 18

3.2.1 Generalization of Montgomery multiplication ... 18

3.2.2 Parallel Montgomery multiplication algorithm ... 19

Chapter 4. Modular Multiplier Architectures – GF(p) 21

4.1 Related work ... 21

4.1.1 Bit-serial architectures ... 21

4.1.2 High-radix architectures ... 21

4.2 Contributions ... 22

4.2.1 Parallel Montgomery multipliers ... 22

Chapter 5. Polynomial Modular Multiplication Algorithms 34

5.1 GF(p^m) polynomial arithmetic ... 34

5.1.1 Related work ... 36

5.1.1.1 Element-serial algorithms ... 36

5.1.1.2 Element-serial/parallel algorithms ... 37

5.1.2 Contributions ... 37

5.1.2.1 Merged-arithmetic GF(p^m) multiplication algorithm ... 37

5.2 GF(2^m) polynomial arithmetic ... 40

5.2.1 Related work ... 40

5.2.1.1 Bit-serial algorithms ... 40

5.2.1.2 Bit-parallel algorithms ... 41

5.2.2 Contributions ... 42

5.2.2.1 Parallel scalar and vector polynomial multiplication ... 42

Chapter 6. Polynomial Modular Multiplier Architectures 44

6.1 GF(p^m) architectures ... 44

6.1.1 Related work ... 45

6.1.1.1 Element-serial architectures ... 47

6.1.1.2 Element-serial/parallel architectures ... 47

6.1.2 Contributions ... 47

6.1.2.1 Merged-arithmetic GF(p^m) multiplier architecture ... 48

6.2 GF(2^m) architectures ... 58

6.2.1 Related work ... 58

6.2.1.1 Bit-serial architectures ... 59

6.2.1.2 Bit-parallel architectures ... 59

6.2.2 Contributions ... 60

6.2.2.1 Scalar and vector polynomial multiplier architecture ... 60

Chapter 7. Conclusion **69**

7.1 GF(p) multiplier architectures ... 69

7.2 GF(2^m) and GF(p^m) polynomial multiplier architectures ... 69

Bibliography **71**

Vita **77**

List of Tables

4.1 Comparison and evaluation of parallel Montgomery multipliers 31

5.1 Examples of the special OEFs 39

6.1 Delay estimates for GF(p^m) multipliers.. 55

6.2 Area estimates for GF(p^m) multipliers 56

6.3 Comparison with Song/Parhi and Bertoni et al.. 57

6.4 Area estimates for scalar GF(2^m) multiply-accumulate units... 66

6.5 Delay estimates for scalar GF(2^m) multiply-accumulate units... 66

List of Figures

1.1 Integrated circuit complexity 3

1.2 Average transistor price by year 4

1.3 RSA and ECC key length by year 5

2.1 Carry-Save school-book multiplication algorithm 11

3.1 Classical modular multiplication algorithm 14

3.2 Montgomery modular multiplication algorithm 16

3.3 Barrett's modular reduction algorithm 17

3.4 Modular reduction algorithm for special moduli 18

4.1 Legend for the symbols 24

4.2 Reduction stages of the generalized parallel Montgomery multiplier 24

4.3 Implementation I 4x4-bit PMM 28

4.4 Implementation II 4x4-bit PMM 28

4.5 Implementation III 4x4-bit PMM 29

4.6 Implementation IV 4x4-bit PMM 29

4.7 Normalized area and delay estimates for the parallel Montgomery multipliers 32

5.1 Element-serial GF(p^m) multiplication algorithm 36

5.2 Element-serial/parallel GF(p^m) multiplication algorithm 37

5.3 Bit-serial GF(2^m) multiplication algorithm 41

6.1 The three classes of GF(p^m) multiplier architecture 45

6.2 Dot diagram for 8x8 Wallace tree 49

6.3 Dot diagram for modular reduction 50

6.4 Parallel multiplier scheme for GF(p^3), $p = 2^n - c$, $\log_2 c << n$, and $f(z) = z^m - 2$ 52

6.5 Transformation of the CDA output 54

6.6 Bit-serial multiplier for $GF(2^5)$ 59

6.7 GF(2^8) parallel multiply-accumulate architecture 62

6.8 High level view of the vector multiply-accumulate unit 63

6.9 Dot diagram for a GF(2^8)/GF(2^4) multiply-accumulate unit 64

6.10 Normalized delay estimates for the scalar and vector GF(2^m) MAC architectures 67

6.11 Normalized area estimates for the scalar and vector GF(2^m) MAC architectures 67

Chapter 1

Introduction

Modular multiplication is a core operation in almost all public-key cryptosystems in use today. In this research parallel, high-speed designs for modular multiplication are presented. This research takes advantage of the transistor bounty provided by Moore's law and the continuously diminishing average cost of a transistor. In addition, advances in automatic synthesis are leveraged to explore designs that are otherwise difficult to manually layout.

Given the large body of knowledge concentrating on cryptography and its implementation, the scope of this work is restricted to public key cryptosystems and parallel modular multiplier designs. Examples are provided to underscore the motivation for the research problem. The next section concludes with the statement of the research problem.

1.1 Motivation

In 1976, Whitfield Diffie and Martin Hellman [14] introduced the concept of public-key cryptography (PKC). PKC facilitates secure communication without the need for any prior agreement on a shared secret key. The field of public-key cryptography has blossomed into an array of algorithms, architectures and applications over the past three decades. The popularity of e-commerce is due in a large part to PKC.

Two widely adopted public-key cryptosystems are the Rivest-Shamir-Addleman (RSA) and Elliptic Curve Cryptography (ECC). Ronald Rivest, Adi Shamir and Leonard

Addleman [35] introduced the RSA scheme in 1978. ECC was proposed independently by Neal Koblitz [22] and Victor Miller [27] around 1985. These cryptographic schemes are based on the assumed hardness of some underlying mathematical problem. For the RSA, there are sub-exponential time algorithms for solving the underlying problem, whereas for the ECC, only fully-exponential time algorithms are yet known. Thus, ECC offers security equivalent to RSA for much smaller key sizes. The security of a 160-bit ECC is comparable to that of a 1024-bit RSA scheme [19]. There are no known polynomial-time algorithms for solving the underlying mathematical problem in either the RSA or the ECC scheme.

On one hand, there is a growing demand for high-speed hardware implementation of these cryptographic protocols, particularly in high-performance network routers and web servers. On the other hand, there is a demand for low-complexity, low-power hardware implementations as well. This is mainly in the resource-constrained smartcard environment. This research does not address the requirements of the resource-constrained segment. The focus of this research is the design of parallel multiplier schemes that can significantly improve the performance of cryptographic protocols of the future. Figure 1.1 [30] shows the trend for Moore's law on the number of transistors that can fit on a chip.

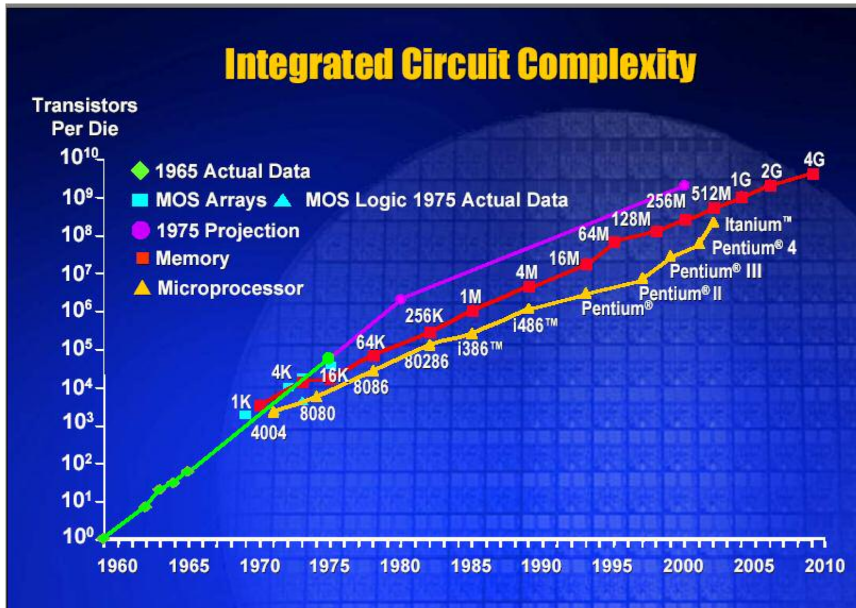


Figure 1.1: Integrated circuit complexity [30].

The actual number of transistors on microprocessors has closely tracked Gordon Moore's 1965 projection [29]. Figure 1.2 [30] shows the average price of a transistor from 1968 to 2002. The average price of a transistor has decreased markedly over the past four decades. Increasing transistor budgets and decreasing average price per transistor opens up the possibility for high-speed designs that were not technologically or economically feasible in the past. Lenstra and Verheul [23] predicted cryptographic key sizes up to the year 2050. Their projections are based on the assumptions that no new ground-breaking solutions will be discovered to the underlying mathematical problems of the RSA and ECC schemes. Recommended cryptographic key sizes change frequently since increased computing power makes it easier to solve the underlying mathematical problems of these cryptosystems. Thus,

the recommended key size must always be large enough such that the computational resources to solve the underlying mathematical problem are out of reach for practically anyone. Figure 1.3 shows the expected key sizes for equivalent security for both the RSA and ECC schemes up to the year 2010 [23]. Clearly, ECC seems more desirable than the RSA going forward. In the near future, parallel modular multiplier schemes will be feasible for operand sizes of up to 256 bits for ECC-based cryptosystems. Current operand sizes for conventional parallel multipliers used in microprocessors are only between 32 and 64 bits. Even with Moore's law, parallel modular multiplier schemes for the RSA's large operand sizes may not be feasible because of interconnect delay and large fan-outs.

Figure 1.2: Average transistor price by year [30].

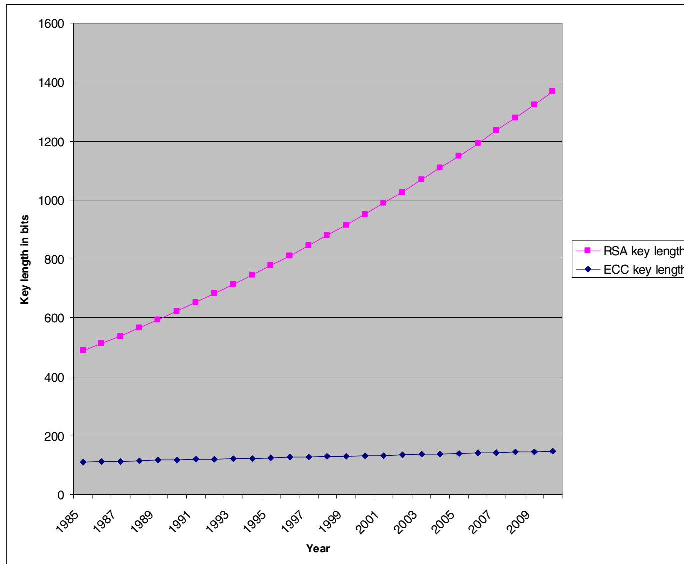


Figure 1.3: RSA and ECC key length by year [23].

Traditionally, research on high-speed modular multipliers has targeted the RSA with its very large operand size. Thus, most of the previous approaches have been iterative, bit-serial or digit-serial. Also, since ECC is traditionally used in resource-constrained environments, most of the multipliers targeted to ECC have also been bit-serial to meet the stringent area and power budgets of these devices. Recently, there has been a proposal to move ECC into high performance web servers [50]. This opens a vista of opportunity for the development of high-speed modular multipliers for ECC-sized operands which is the focus of this research work. These high-speed multipliers will be the building blocks of future hardware crypto-accelerators. In addition, ECC can utilize polynomial modular arithmetic. Like integer modular arithmetic, most of the architectures that have been proposed for polynomial modular arithmetic are not fully parallel. This research work also explores novel

schemes for parallel polynomial modular arithmetic. The following section gives a statement of the research problem.

Research Problem

The primary goal of this research is to explore novel algorithms and architectures for parallel integer and polynomial modular multipliers.

The major components of the research are:

- Exploration of novel parallel algorithms for modular multiplication
- Design of efficient hardware architectures based on the parallel algorithms
- Implementation and characterization of the hardware designs

1.2. Mathematical Background and Notation

Modular multiplication is simply the computation of the remainder of the product of two numbers with respect to a modulus. More formally, the modular multiplication problem is defined as the computation of $R = A \times B \bmod M$ given the integers A, B, M with $0 \leq A, B < M$. For most cryptographic applications such as the RSA or ECC scheme, only the case of an odd M is relevant. In the case where the modulus M is a prime, modular arithmetic is primarily arithmetic in a prime Galois field GF(p). A number of definitions relating to arithmetic in finite fields are presented below.

Definition 1.1: A ring (R, +, x) consists of a set R with two binary operations + (addition) and x (multiplication) on the set R, satisfying the following conditions:

1. (R, +) is an abelian group with identity denoted 0. An abelian group is a group for which the elements commute (i.e. $A \bullet B = B \bullet A$, for all elements A and B, where $\bullet$ is the group operation).

2. The operation x is associative. That is, a x (b x c) = (a x b) x c for all $a, b, c \in R$.

3. There is a multiplicative identity denoted 1, with $1 \neq 0$, such that 1 x a = a x 1 = a for all $a \in R$.

4. The operation x is distributive over +. That is, a x (b + c) = (a x b) + (a x c) and (b + c) x a = (b x a) + (c x a) for all $a, b, c \in R$.

The ring is a commutative ring if a x b = b x a for all $a, b \in R$.

Definition 1.2: A field is a commutative ring in which all non-zero elements have multiplicative inverses. This implies that there exists an element a^{-1} for all elements $a \in R$ apart from 0 for which a x $a^{-1} = 1$. The set of integers modulo a prime p with addition and multiplication performed modulo p is a field.

Definition 1.3: A field that has a finite number of elements is called a finite field or Galois field. The order of a finite field F is the number of elements in F ($\#F$). For any prime p, GF(p) is a prime Galois field with p elements. For every prime power p^m, there exists a unique finite field of order p^m. This Galois field is denoted by the prime extension field GF(p^m). The case where the prime $p = 2$ is the binary extension field GF(2^m).

Definition 1.4: Polynomial basis – the elements of the Galois field GF(p^m) can be represented as polynomials of the form: $a(z) = a_{m-1}z^{m-1} + ... + a_2z^2 + a_1z + a_0$. The element a_i is the

coefficient of z^i. The largest integer k for which $a_k \neq 0$ is called the degree of $a(z)$, denoted deg(a(z)). a_k is called the leading coefficient of $a(z)$.

1.3 Contribution and Dissertation Overview

In this dissertation novel algorithms and architectures for integer and polynomial basis parallel modular multipliers are presented. The major contributions of the research are:

- Novel parallel algorithms for integer and polynomial modular multiplication
- Design of efficient hardware architectures based on the parallel algorithms
- Implementation, quantitative analysis and characterization of the parallel hardware designs

The organization of this dissertation is as follows.

In chapter 2, a background on conventional (non-modular) multiplication algorithms and architectures are presented. This chapter discusses the algorithms for serial, serial-parallel and fully parallel multiplication. In addition, the hardware structures to implement these algorithms are also presented.

In chapter 3, a number of modular multiplication algorithms that have been proposed in the literature are reviewed. The chapter concludes with a presentation of a novel parallel modular algorithm introduced in this dissertation.

In chapter 4, a survey of existing architectures for modular multiplication is presented. In addition, a new parallel modular multiplier is also presented. Four variants of this architecture are discussed with emphasis on a trade-off between speed and area.

In chapter 5, GF(p^m) and GF(2^m) polynomial modular arithmetic is introduced in detail. Existing algorithms for performing polynomial modular arithmetic are presented along with new algorithms introduced in this dissertation.

In chapter 6, novel and previously proposed hardware architectures for polynomial modular arithmetic are discussed.

The dissertation is concluded in chapter 7.

Chapter 2

Conventional Multiplication Algorithms and Architectures

Let a and b be two n-digit numbers expressed in radix r as:

$$a = (a_{n-1}a_{n-2}...a_0) = \sum_{i=0}^{n-1} a_i r^i$$

$$b = (b_{n-1}b_{n-2}...b_0) = \sum_{i=0}^{n-1} b_i r^i$$

where the digits of a and $b$are in the range [0, r-1]. In general r can be any positive number. For computer implementations, r is often selected to be a power of 2. The school-book algorithm for multiplying a and b produces the partial products by multiplying each digit of the multiplier operand (b_i) by the entire number representation of the multiplicand (a). Let t_{ij} denote the (carry, sum) pair produced from the product of a_i and b_j. For instance, in radix 10, when $a_i = 6$ and $b_j = 4$, then $t_{ij} = (2, 4)$. The carry is $a_i \cdot b_j$ div r and the sum is $a_i \cdot b_j$ mod r. The partial products are typically arranged thus:

$$\begin{array}{rcccccccc}
 & & & & & a_3 & a_2 & a_1 & a_0 \\
\text{x} & & & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & t_{03} & t_{02} & t_{01} & t_{00} \\
 & & & & t_{13} & t_{12} & t_{11} & t_{10} & \\
 & & & t_{23} & t_{22} & t_{21} & t_{20} & & \\
 & & t_{33} & t_{32} & t_{31} & t_{30} & & & \\
\hline
 & t_7 & t_6 & t_5 & t_4 & t_3 & t_2 & t_1 & t_0
\end{array}$$

The last row is the summation of all the partial products – a $2n$-digit number. The sequence of operations involves a series of digit multiplies and adds. The school-book algorithm using carry-save representation is presented below:

Algorithm 2.1: Carry-Save school-book multiplication algorithm

Inputs: A, B, Output: R = (A·B)

Initialization: $t_{ij} = 0$ for $0 \leq i < 2n$

For i = 0 to n-1 do

  $C = 0$;

  For $j = 0$ to n-1 do

    $(C, S) = t_{ij} + a_i \cdot b_j. + C$

    $t_{i+j} = S$

  $t_{i+n} = S$

output: $(t_{2n-1}\ \ t_{2n-1} \ldots t_0)$

Figure 2.1: Carry-Save school-book multiplication algorithm

The standard multiplication algorithm can be implemented in hardware with varying degrees of parallelism - a serial, parallel or hybrid serial-parallel architecture.

2.1 Bit-serial multipliers

Bit-serial multipliers are useful when area overhead is of concern, and speed is not of primary importance. In bit-serial designs, both operands of the multiplication operation are processed serially. The basic building blocks of a bit-serial multiplier are a 1-bit multiplier which can be implemented with an AND gate, and a bit accumulator. The inputs are fed into the multiplier serially and the output is also transmitted in a serial fashion.

2.2 Serial-parallel multipliers

In this architecture, one of the operands is processed in parallel while the other is processed serially. The basic components are AND gates for generating the partial products

and carry-save adders. The architecture implements the inner loop of Algorithm 2.1 in parallel. In each iteration, a sum and carry vector pair is produced, which is added to the next set of partial products. In this architecture, the multiplier inputs are fed in serially and the final output is available in parallel after a number of cycles.

2.3 Parallel multipliers

Fully-parallel multiplier architectures process both operands in parallel. The components of this multiplier scheme are an array of AND gates and a tree of parallel counters. The partial products are compressed in parallel to a pair of sum and carry vectors in logarithmic time. A final fast carry-propagate adder is then used to produce the final result. Parallel multipliers require a large area for implementation but have a very low delay.

Chapter 3

Modular Multiplication Algorithms

Basically, the objective of an n-bit modular multiplication is to take two n-bit numbers A and B and derive a result $A \cdot B$ mod M that is at most n-bits as well. This is achieved by subtracting a multiple of the modulus M from $A \cdot B$ such that the result is n-bits wide and also less than M. This differs from conventional multiplication where the product of two n-bit numbers will be at most $2n$ bits wide. For modular multiplication, the product of two n-bit numbers modulo another n-bit number yields a result that is at most n-bits wide.

3.1. Related Work

A number of algorithms have also been proposed in the literature for modular multiplication. Some of these algorithms include the Brickell [9] and Montgomery [28] algorithms. For some of the algorithms, computation proceeds from the least significant digit to the most significant digit. One example of this approach is the Montgomery algorithm. The advantage of this approach in hardware implementations is the fact that modular correction bits and carry signals propagate in the same direction. There are also some algorithms in the literature where the computation proceeds from the most significant digit position to the least. Another approach is based on lookup tables [49]. The effectiveness of this approach depends on the ability to minimize the size of the lookup tables required.

Some of the algorithms that have been proposed only work for certain types of modulus while others are generally applicable to any modulus. The classical modular multiplication algorithm and Barrett's algorithm work for any modulus. However, the Montgomery algorithm works only for odd moduli. This does not pose a problem, as most of the application areas of modular arithmetic such as public-key cryptography require the use of odd moduli.

3.1.1 Classical modular multiplication algorithm

The classical approach for performing modular multiplication involves computing the product $A \cdot B$, and then subtracting a multiple of the modulus M that makes the result to be less than the modulus. An optimization of this approach interleaves the computation of the product and the subtraction of the modulus. The classical algorithm is generally inefficient and very slow. The interleaved classical modular multiplication is presented in Algorithm 3.1.

Algorithm 3.1: Classical modular multiplication

Inputs: A, B, M with $0 \leq A, B < M$, Output: $R = AB \bmod M$

$R = 0$;

For $i = n\text{-}1$ to 0

Begin

    $R = 2 \cdot R + a_i \cdot B$;

    $q_i = R \text{ div } M$;

    $R = R - q_i \cdot M$;

End

Figure 3.1: Classical modular multiplication algorithm.

3.1.2 Montgomery algorithm

Another approach to performing modular multiplication is the Montgomery algorithm. The basic idea behind Montgomery multiplication is the fact that one can add a multiple of the modulus M to the product $A \cdot B$ to yield a result that is at most $2n+1$ bits wide. Adding, instead of subtracting, a multiple of the modulus M does not affect the computation, since the result will be congruent to $A \cdot B$ modulo M. Two numbers are said to be congruent if their remainder when divided by the modulus is the same. Thus, $A \cdot B$, $A \cdot B + M$, $A \cdot B + 2M$, ... $A \cdot B + kM$ are all congruent modulo M. This implies: $A \cdot B \equiv A \cdot B + M \equiv A \cdot B + 2M \equiv \ldots (A \cdot B + kM) \bmod M$. In the Montgomery algorithm, the multiple of the modulus M that is added to $A \cdot B$ is chosen in such a way that the lower n-bits of the $2n+1$-bit result are all zeroes. The least significant half of the $2n+1$-bit result that are all zeroes are then discarded. This way, the result would have been reduced to at most $n+1$ bits in width. A single subtraction of the modulus M can then be performed to further reduce the result to at most n-bits and make it less than M if required. It has been shown by Walter [46] that the extra subtraction may not be necessary under certain conditions.

Montgomery's approach in essence achieves the objective of modular multiplication, which is to take two n-bit numbers, multiply them and derive a result that is at most n-bits wide. The resulting n-bit number is not exactly $A \cdot B \bmod M$. It is referred to in the literature as a Montgomery product $A \cdot B\ 2^{-n} \bmod M$. However, most cryptographic schemes make use of repeated modular multiplications such as modular exponentiation - $A^e \bmod M$. Montgomery multiplication can then be used in performing the repeated multiplications, and only the final result of the exponentiation is converted back from the Montgomery domain. Thus, the cost

of conversion to and from the Montgomery domain is amortized over the repeated modular multiplications. The conversion from the Montgomery domain is just another Montgomery multiplication by 2^{2n}. See [28] for the detailed proof of correctness of the Montgomery algorithm. Just like the classical modular algorithm, Montgomery's algorithm can also be performed in a fashion whereby the computation of the product and the addition of the modulus are interleaved. The interleaved Montgomery modular multiplication is presented in Algorithm 3.2.

Algorithm 3.2: Montgomery modular multiplication

Inputs: A, B, M with $0 \leq A, B < M$

Output: R = Montgomery Product $(A \cdot B2^{-n})$ mod M

R = 0;

For i = 0 to n-1 do

Begin

$R = R + a_i \cdot B$;

$R = R + r_0 \cdot M$;

$R = R$ div 2;

Figure 3.2: Montgomery modular multiplication algorithm.

In each iteration of the loop, the least significant bit of the intermediate result is inspected. If it is '1', i.e. the intermediate result is odd; we add the modulus M to make it even. This is possible since M is guaranteed to be odd in the cryptographic applications of interest. Thus, at each step the intermediate result is made to be even. This even number can be divided by 2 without any remainder. This division by 2 reduces the intermediate result to n+1 bits again. Dividing the intermediate result by 2 is equivalent to discarding the current

least significant bit of the intermediate result that is zero. After n steps these divisions add up to one division by 2^n, or discarding the least significant n-bits that are zeroes.

3.1.3 Barrett's algorithm

Barrett's algorithm computes $P = X \bmod M$ given X and M. Typically, the multiplication operation is first performed and then a modular reduction operation follows. The algorithm requires the pre-computation of the quantity r^{2k}/M where r is the radix in which the algorithm is implemented, usually a power of 2. The cost of the pre-computation can be amortized over repeated modular reductions.

Algorithm 3.3: Barrett's modular reduction

Inputs: X, M and $\mu = \lfloor r^{2k}/M \rfloor$ Output: $P = X \bmod M$

$Q1 = \lfloor X/r^{k-1} \rfloor$, $Q1 = Q2 \cdot \mu$, $Q3 = \lfloor Q2/r^{k+1} \rfloor$

$P1 = X \bmod r^{k+1}$, $P2 = Q3 \cdot M \bmod r^{k+1}$, $P = P_1 - P_2$

If $P < 0$ then $P = P + r^{k+1}$

While $P \geq M$ do

$P = P - M$

Output : P

Figure 3.3: Barrett's modular reduction algorithm.

3.1.4 Algorithms for special moduli

If the modulus for the modular multiplication operation has a special form, faster reduction techniques can be utilized. An efficient technique can be utilized for modular reduction if the modulus is of the form $r^n - c$ for a radix r implementation.

Algorithm 3.4: Modular reduction for $M = r^n - c$

Inputs: X and $M = r^n - c$ Output: $P = X \bmod M$

$Q_0 = \lfloor X/r^n \rfloor$, $P_0 = X - Q_0 \cdot r^n$, $P = P_0$, $i = 0$

While $Q_i \geq 0$ do

$Q_{i+1} = \lfloor Q_i \cdot c / r^n \rfloor$, $P_{i+1} = Q_i \cdot c - Q_{i+1} r^n$

$i = i + 1$, $P = P + P_i$

While $P \geq M$ do

$P = P - M$

Output: P

Figure 3.4: Modular reduction algorithm for special moduli.

3.2 Contributions

The major contribution [37] in this section is the generalization of the Montgomery modular algorithm to modular vector summation and the introduction of a fully parallel version of the Montgomery modular multiplication. Montgomery's algorithm is usually favored in hardware implementations over both the Barrett's algorithm and the classical algorithm since it handles the propagation of carries and the modular correction bits efficiently. The generalized parallel Montgomery algorithm is applicable to combined multi-operand addition and modular reduction.

3.2.1 Generalization of the Montgomery Algorithm

In this section, we make some extensions to and generalize the Montgomery algorithm to apply not only to multiplication but also to vector summation. The basic idea behind the extension is to transform an nxn-bit multiplication into a summation of k n-bit

numbers whose sum is congruent to the nxn-bit product with respect to the modulus. The modular reduction is then performed with the sum of the k n-bit numbers instead of the nxn-bit product. The advantage of this approach is that the vector summation of k n-bit numbers can be interleaved with the modular reduction in $\log_{1.5}k$ time using parallel counters - full adders and half adders. Thus, if k is less than 1.5^n, the new algorithm will be faster.

Given k n-bit integers $X^0, X^1, \ldots, X^{k-1}$ with $X^j = \sum_{i=0}^{n-1} x_i^j 2^i$ for $0 \le j < k$, the modular vector summation of the integers is given by: $\sum_{j=0}^{k-1} X^j \bmod M$. We define the Montgomery modular vector summation as:

$$\sum_{j=0}^{k-1} X^j 2^{-\log_{1.5} k} \bmod M \equiv \frac{\sum_{j=0}^{k-1} X^j + \sum_{j=0}^{\log_{1.5} k-1} 2^j e_j M}{2^{\log_{1.5} k}}, e_j \in \{0,1\}$$

3.2.2 Parallel Montgomery Algorithm

Three n-bit unsigned binary integers, $A = a_{n-1}\, a_{n-2}\, a_{n-3} \ldots a_2\, a_1\, a_0$, $B = b_{n-1}\, b_{n-2}\, b_{n-3} \ldots b_2\, b_1\, b_0$, and $M = m_{n-1}\, m_{n-2}\, m_{n-3} \ldots m_2\, m_1\, m_0$ have the values:

$$A = \sum_{i=0}^{n-1} a_i 2^i \qquad B = \sum_{i=0}^{n-1} b_i 2^i \qquad M = \sum_{i=0}^{n-1} m_i 2^i \tag{1}$$

The Montgomery Product $A \cdot B \cdot 2^{-n} \bmod M$ has the value

$$A \cdot B \cdot 2^{-n} \bmod M \equiv \frac{\sum_{i=0}^{n-1} 2^i a_i B + \sum_{i=0}^{n-1} 2^i e_i M}{2^n}, \; e_i \in \{0,1\} \tag{2}$$

Suppose we can find k n-bit numbers $\{X^0, X^1, X^2, X^3 \ldots X^{k-1}\} : 0 \le X^j < 2^n$ for $j = 0$ to k-1 that satisfy:

$$\sum_{j=0}^{k-1} X^j \equiv \left(\sum_{i=0}^{n-1} 2^i a_i B \right) \bmod M \tag{3}$$

Since each X^j is at most n-bits wide, $\sum_{j=0}^{k-1} X^j$ is at most $(n + \log_2 k)$ bits wide.

Furthermore, the expression $\sum_{j=0}^{\log_{1.5} k-1} 2^j e_j M$ will be at most $(n + \log_{1.5} k)$ bits wide. Therefore, the sum $\sum_{j=0}^{k-1} X^j + \sum_{j=0}^{\log_{1.5} k-1} 2^j e_j M$ will be at most $(n + \log_{1.5} k + 2)$ bits wide, since $\log_{1.5} k > \log_2 k$. Thus, the expression in (4) will be at most $(n + 2)$ bits wide.

$$\sum_{j=0}^{k-1} X^j 2^{-\log_{1.5} k} \bmod M \equiv \frac{\sum_{j=0}^{k-1} X^j + \sum_{j=0}^{\log_{1.5} k-1} 2^j e_j M}{2^{\log_{1.5} k}}, e_j \in \{0,1\} \tag{4}$$

This revised algorithm is contingent on being able to find k n-bit numbers $\{X^0, X^1, X^2, X^3, \ldots X^{k-1}\}$ such that $0 \le X^j < 2^n$ for $j = 0$ to k-1 that satisfy: $\sum_{j=0}^{k-1} X^j \equiv \sum_{i=0}^{n-1} 2^i a_i B \bmod M$.

Chapter 4

Modular Multiplier Architectures – GF(p)

This chapter focuses on hardware architectures for modular multiplication or arithmetic in prime Galois fields GF(p). The hardware architectures implement the algorithms presented in the previous chapters. Hardware architectures are usually more efficient than software implementations for modular arithmetic. However, hardware implementations may be less flexible than software-only approaches.

4.1 Related work

A number of modular multiplier architectures have been presented in the literature. Most of the architectures are based on Montgomery's algorithm because it is specially suited for hardware implementation. The hardware Montgomery multipliers can broadly be classified as either bit-serial or high-radix architectures.

4.1.1 Bit-serial architectures

Bit-serial Montgomery multipliers implement the inner loop of Algorithm 3.2. The basic components of such bit-serial architectures are AND gates and carry-save adders and an accumulator for accumulating the partial results. The division by 2 in the algorithm is readily implemented as a bit-shift in hardware.

4.1.2 High-radix architectures

High radix Montgomery multipliers are similar to the bit-serial architectures, but in each iteration, a digit of the multiplier operand is processed rather than just a bit. The multiplicand bits are processed in parallel. Processing multiple digits at a time makes the

quotient selection more complex and each iteration does not involve only a simple bit-shift as in the serial architecture. The basic components of the high-radix architecture are an array of digit-by-digit multipliers and adders. Each iteration takes more time to complete than in the bit-serial architecture, but the total number of iterations is reduced.

4.2 Contributions

The major contribution [37] in this section is the efficient implementation of the generalized Montgomery modular algorithm presented in the previous chapter. Four designs for implementing the parallel Montgomery algorithm are explored in this section. All the designs utilize small look-up tables and fast, massively parallel multipliers. Two of the designs trade off smaller look-up tables for a larger, slightly slower multiplier. The other two approaches use larger look-up tables but a smaller, faster multiplier.

4.2.1 Parallel Montgomery multipliers

In this section, the modified Montgomery algorithm is implemented in hardware. With conventional parallel multipliers, the partial product matrix bits are generated by an array of AND gates, and are reduced to sum and carry vectors using a Wallace [48] or Dadda [12] tree. The sum and carry vectors in the last stage can then be added using a high-speed carry-propagate adder. The Wallace reduction tree is usually composed of full-adders and half-adders. In the proposed scheme for the modified Montgomery multiplication algorithm, back-to-back full-adders and half-adders are utilized. Figure 4.1a shows in more detail the modifications to the conventional Wallace tree. The rows in the bit product array are banded together into groups of 3's and full adders are used to reduce 3 rows to 2 rows. So as to ensure that the least significant bit is zero at the end of each stage of the reduction tree, we

arrange the two full adders back-to-back. After the 3 rows have been reduced to 2 rows, if the least significant bit of the group is '1', then the bits of the modulus M are combined with the 2 rows to yield a new set of two rows with the least significant bit zero. The least significant bit is then discarded at the end of that reduction stage. For practical designs, additional buffers will be needed to handle the large fan-out of the least significant bit position.

It is possible that after the organization of the rows into groups of 3's, two rows are left. The two rows are combined with back-to-back half-adders as shown in Figure 4.1b. Still, after the groupings, it's possible that only one row is left. In this case, the single row is combined with the modulus M using half-adders to ensure the least significant bit is zero, as shown in Figure 4.1c. In this case, that single row will become two rows in the next stage after the modulus has been combined with it. This does not significantly affect the rate at which all the rows of the summation matrix are reduced to the final two rows. The number of rows in the stages of the Wallace reduction tree are n, ... 13, 9, 6, 4, 3, 2. With the modified Wallace tree, the width of each stage is n-bits except for the last two stages (with heights of $4 \rightarrow 3$ and $3 \rightarrow 2$) where the back-to-back adders are not used. It is not necessary that the least significant bits are zero in the last two stages. At this point, the widths of the rows will remain n+1-bits until the final stage. Thus, the final two rows are n+1-bits wide.

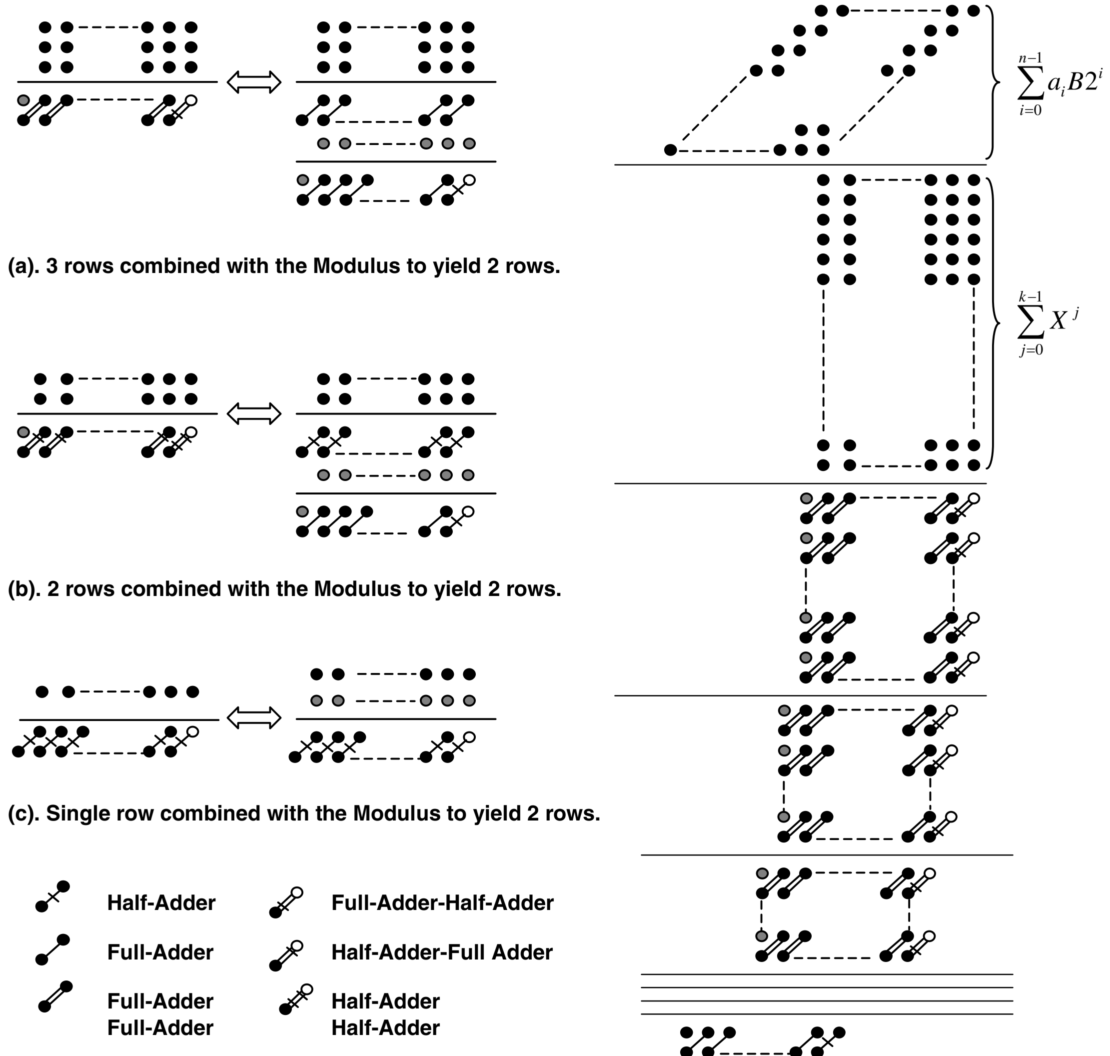


Figure 4.1: Legend for the symbols.

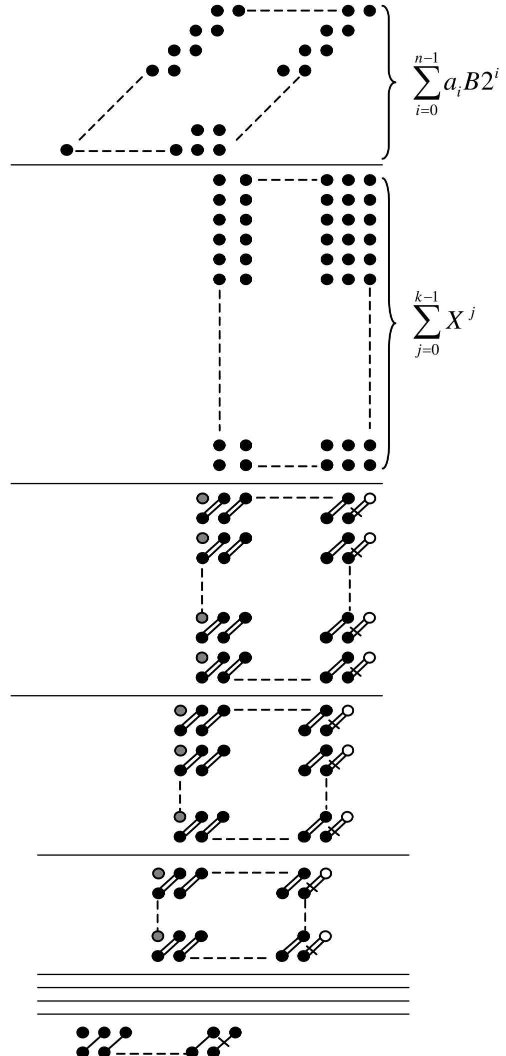


Figure 4.2: Reduction stages of the generalized parallel Montgomery multiplier.

The first stage in Figure 4.2 shows the partial product matrix for $A \cdot B$. This array is transformed to the summation matrix in the next stage. The remaining stages in Figure 4.2

are those of a modified Wallace tree that reduce the summation array to a pair of sum and carry vectors in the final stage. The primary difference between the conventional Wallace tree and the modified Wallace tree is the fixed width of each stage in the modified scheme. In the modified scheme, the least significant bits are discarded at each stage. This ensures that the widths of the successive stages are the same. Keeping the width constant is achieved by using back-to-back full adders and half adders. The modulus M is added to the output of a bank of full-adders if the least significant bit is '1'. Since the modulus M is odd, the resulting pair of carry and sum vectors at the output of the second set of full adders will have the least significant bit as '0'.

Design Alternatives

This section presents four design alternatives for the Parallel Montgomery Multipliers (PMMs). All the designs follow the generalized hardware implementation with different values of k for the summation array.

Implementation I

In this section, we focus on how to generate the summation array vectors X^js. The bit product array for multiplying two numbers can be divided into two halves. The least significant half has n rows ranging from 1 bit to n bits in width. Similarly, the most significant half has n-1 rows ranging in width from 1 to n-1 bits. Thus, the total number of bits in the most significant half is $n(n-1)/2$. We envision a lookup table (LUT) that holds pre-computed values of 2^i mod M for $n \le i < 2n-1$. This LUT will have $n-1$ entries of n-bits each. The right half of the bit product array is then expanded to $n(n+1)/2$ rows as shown in Figure

4.3 for a 4x4 bit multiplier. As shown in the figure, there are 6 dots in the left half of the bit product array, and each dot forms a new row in the summation matrix. These new 6 rows are then combined with the 4 rows from the right half of the bit product array to yield the summation matrix with 10 rows.

For every i^{th} bit in the left half of the bit product array that is '1', the output of the LUT forms a new row that represents $2^i \bmod M$ else the new row will be all zeroes. This guarantees that every row in the summation array represents a number that is less than 2^n. The rows of the summation array are then summed in logarithmic time using the modified Wallace tree. In this case, the parameter $k = n(n+1)/2$. Subsequent section explore ways to further reduce k and also the size of the LUTs.

Implementation II

In this section, an alternative scheme to reduce the size of the lookup table is proposed. In this approach, the size of the lookup table is reduced by half but the size of the summation array is increased by about a factor of 1.5. Since the reduction tree is of logarithmic time complexity, the total delay is minimally impacted. That is, the number of stages is increased from $\log_{1.5}(n(n+1)/2)$ to $\log_{1.5}(1.5 \cdot n(n+1)/2) \sim \log_{1.5}(n(n+1)/2) + 1 \sim 2\log_{1.5} n$. The basic idea behind this approach is to group adjacent bits in the left half of the bit product array into pairs and then convert each pair of bits into 3 unary bits. This is essentially performing the reverse operation of a full adder. This guarantees that every other column of bits in the left half of the bit product array will be zero. This approach is depicted in Figure 4.4 for a 4x4 bit multiplier. In the figure, the 6 dots in the left half of the bit product array are converted into 8 dots, so that the summation array has a total of 12 rows.

Thus, the height of summation array has increased by about a factor of 1.5 as compared to the example in the previous section. However, the number of non-zero columns will be reduced by half. Thus, the size of the lookup table will also be reduced half, since only the values 2^i mod M for $i = n$, n+2, n+4 ... 2n-1 need to be stored. As shown in Figure 4.4, there are only 2 non-zero columns in the left half of the bit product array. This approach trades off a smaller lookup table for a larger bit-product array without significantly impacting the total delay. However, the hardware complexity is significantly increased.

Implementation III

Another idea to simplify the hardware complexity is introduced in this section. Instead of increasing the height of the bit product array from n to $n(n+1)/2$, we leave the bit product array with a height of n. We then split the bit-product array into two halves, and the reduction of both halves proceeds in parallel as in the previous two approaches. Both halves will be simultaneously reduced to two rows in $\log_{1.5}n$ stages. Similarly, the reduction is interleaved with the "zeroing out" of the least significant bits at each stage. Thus, at the completion of both stages we will have two n-bit rows in the left half and two n+1-bit rows in the right half. The bits in the left half are scaled by a factor $2^{n + \log n}$. Thus, we can modify the lookup table so that it contains the residues 2^i mod M, $(n + \log n) \leq i \leq (2n + \log n)$. The first 2$n$ bits of the summation array are then generated from the two n-bit rows of the left half and the LUT. Each bit from the two rows on the left half becomes a new row in the resulting summation array.

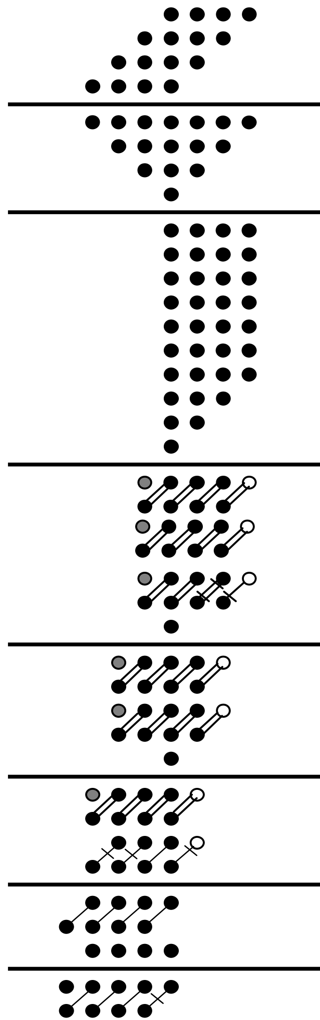


Figure 4.3: Implementation I 4x4-bit PMM.

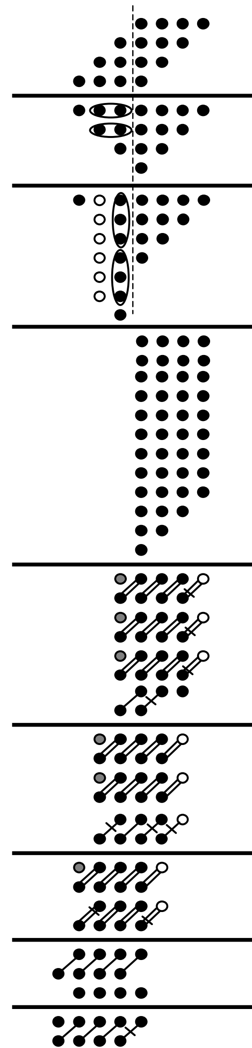


Figure 4.4: Implementation II 4x4-bit PMM.

The resulting summation array has $2n + 2$ rows and it can be reduced to two rows in $\log_{1.5}(2n + 2) \sim \log_{1.5}n$ stages. Thus, the total delay is $(\log_{1.5}n + \log_{1.5}(2n + 2)) \sim 2\log_{1.5}n$ stages. This is about the same delay as the two previous approaches. However, the hardware complexity

has been greatly reduced. The height of the bit product array has been reduced from $n(n+1)/2$ to two arrays of height n operating in parallel followed by one array of height $2n + 2$. Figure 4.5 shows an example for a 4x4 bit multiplier. It is split into the left and right halves with 3 and 4 rows respectively.

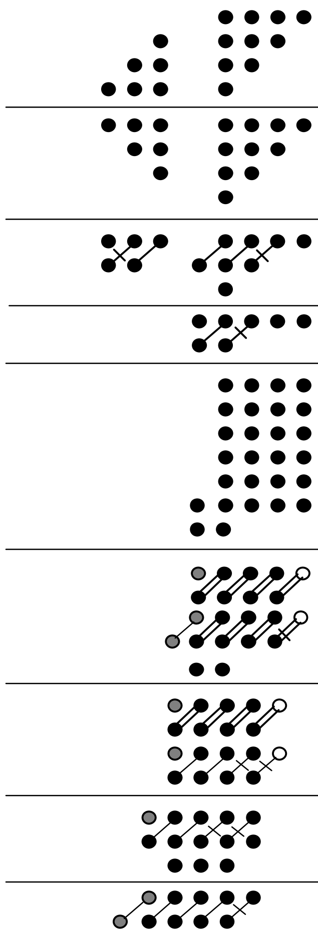


Figure 4.5: Implementation III 4x4 bit PMM.

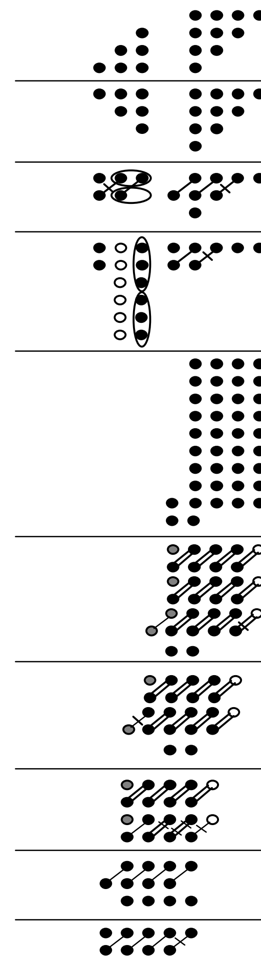


Figure 4.6: Implementation IV 4x4-bit PMM.

The left half requires only one stage to reduce it to two 3-bit wide rows. The right half requires two stages to reduce it to two 5-bit wide rows. As has been noted before, the back-to-back adders are only used in the reduction stages before the last two. Thus, the least significant bits are not discarded in this example because there are only two stages. However, for 5x5-bit, 6x6-bit and larger multipliers, each half will involve discarding the least significant bits at all but the last two stages. As shown in the figure, the 5 dots from the two rows on the left half are transformed into the first five rows of the summation matrix, and the two rows from the right half complete the summation matrix. The summation matrix of 7 rows is then reduced to the final sum and carry vectors in 4 stages.

Implementation IV

In this section, the size of the lookup table for Implementation III is further reduced. As in the approach in the last section, the bit product array is split into two halves that are reduced in parallel. However, in this case, after the left half has been reduced to two rows, we then group adjacent bits in pairs and split them into unary bits just like in the approach of Implementation II. This is depicted in Figure 4.6 for a 4x4 bit multiplier. The two rows in the left half are then transformed into 6 rows with every other bit guaranteed to be a zero. Thus, each row contains at most $n/2$ non-zero bits. The lookup table size can then be reduced by half. In this example, the left half has only two columns of non-zero bits. Each non-zero bit from the left half is then transformed into a new row in the resulting bit matrix. The resulting summation matrix will now have a height of $3n + 2$. This summation array will be reduced to two rows in $\log_{1.5}(3n + 2) \sim \log_{1.5}n$ stages. The total number of stages is then $(\log_{1.5}n + \log_{1.5}(3n + 2)) \sim 2\log_{1.5}n$ stages. As shown in Figure 4.6, the summation matrix is

reduced to the final two rows in 5 stages. Table 4.1 summarizes the four design approaches for the parallel Montgomery multipliers.

Table 4.1: Comparison and evaluation of parallel Montgomery multipliers.

	Height (k) of Summation Array(s)	Number of entries in LUT	Number of stages
I	$n(n+1)/2$	n	$\log_{1.5}(n(n+1)/2)$
II	$3n(n+1)/4$	$n/2$	$\log_{1.5}(3n(n+1)/4)$
III	n, $2n+2$	n	$\log_{1.5}(\mathrm{n}(2n+2))$
IV	n, $3\mathrm{n}+2$	$n/2$	$\log_{1.5}(n(3n+2))$

To obtain area and delay estimates, a specialized C++ program was developed that generates structural Verilog models for the Parallel Montgomery Multipliers (PMM), given an operand size, n. As a proof of concept, the C++ program was used to generate Verilog models of PMMs for various operand sizes. The designs were synthesized on a 0.18 micron CMOS standard cell library. The normalized delay and area estimates were extrapolated to show a trend of how each of the approaches scale as the operand size is increased. These results are presented in Figure 4.7. The area complexity is $O(k^2)$ and the delay complexity is $O(\log k)$ for all the four schemes. The trend lines in Figure 4.7a depict the area complexities of the four implementations. Note that k for each implementation type is derived from the operand size n using Table 4.1.

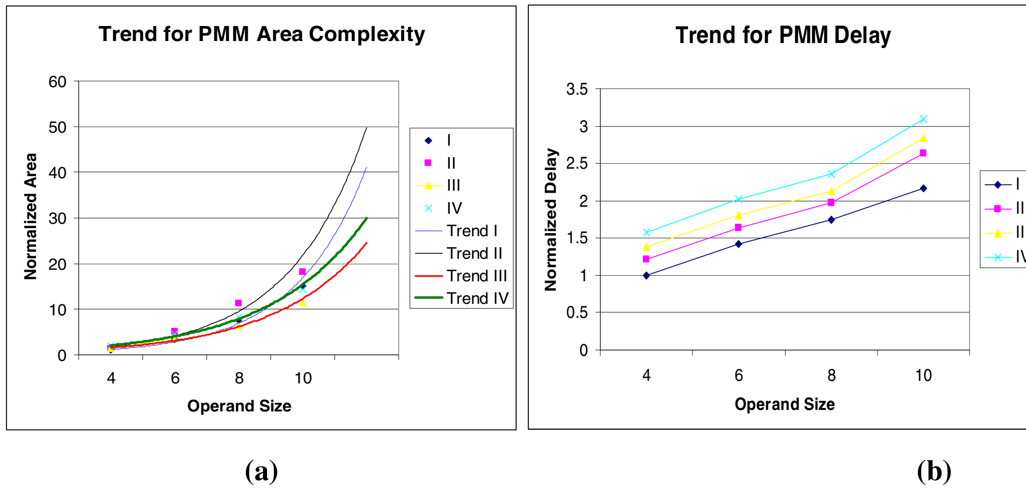


Figure 4.7: Normalized area and delay estimates for the parallel Montgomery multipliers.

Evaluation

The architectures presented in this section involve significant hardware complexity but exploit massive parallelism. All of the designs can be pipelined to further improve the throughput. Most of the implementations of Montgomery multiplication in the literature are based on bit-serial or systolic designs, pumping the global signals through flip-flops to reduce the delay and avoid large fan-out. The systolic-array method, however, needs a large number of registers and long latency. The effects of large fan-out and global wire delays in the parallel schemes presented are less severe for smaller operand sizes. Thus, the schemes presented may be targeted towards the Elliptic Curve Cryptosystem where the operand sizes are currently between 128 and 160 bits. For the RSA scheme where the operand sizes are about 1024 bits, large fan-out may pose a greater obstacle to practical implementations. The

other proposal in the literature for constructing logarithmic time modular multipliers [45] requires multiplication with real numbers and does not exclusively use fixed point arithmetic.

Chapter 5

Polynomial Modular Multiplication Algorithms

5.1 GF(p^m) polynomial arithmetic

Arithmetic in Galois fields is an integral part of elliptic curve cryptosystems [22] [27]. Typical choices of fields include the prime Galois field GF(p), realized as the integers modulo a prime p, and binary extension field GF(2^m), often realized as the set of binary polynomials of degree at most m-1. The elements of GF(2^m) can be represented in a number of ways such as the polynomial and the normal bases. In the polynomial basis, multiplication is performed modulo an irreducible polynomial. The polynomial basis representation in the binary extension field case can be generalized to all extension fields GF(p^m), with coefficient arithmetic performed in GF(p) (modulo p , p prime).

The prime Galois field GF(p) is the set of all positive integers less than a particular prime number. For instance, Galois Field GF(13) is the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}. With this finite field, addition is the same as conventional addition, but after addition, the result is reduced modulo the prime, in this case 13 – that is - divide the result by 13 and take the remainder. This ensures that the final result is always less than 13 and it is thus a member of the set {0, 1, 2, 3, ... , 12} which is GF(13). The same goes for field multiplication, the conventional multiplication followed by a modulo operation to reduce the result to be less than the prime p.

The finite field GF(p^m) can be represented in the polynomial basis representation. In this representation, the elements of the finite field are not integers but each member of the field is actually a polynomial of degree m-1. The degree of a polynomial is the power of the highest non-zero co-efficient. For instance, in GF(p^3) we can have a polynomial $a(z) = a_2*z^2 + a_1*z + a0$ – the degree of this polynomial is 2. Similarly we can have another polynomial $b(z) = b_2*z^2 + b_1*z + b_0$. Let $p = 13$, then the finite field GF(p^3) = GF(13^3) with two example elements $a(z) = 8z^2 + 7z + 3$ and $b(z) = 2z^2 + 9z + 1$. The coefficients of the polynomials are the integers in the range {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}, that is the coefficients are elements of GF(p) or GF(13) since p = 13 in this case. Addition involves just adding the corresponding coefficients of the polynomial, and performing the modulo operation if ever the result is larger than 13 or the prime p. So $a(z) + b(z) = (8 +2)*z^2 + (7+ 9)*z + (3+1) = 10z^2 + 3z + 4$.

Field multiplication of $a(z)$ and $b(z) = (8z^2 + 7z + 3)*(2z^2 + 9z + 1) = 16z^4 + 72z^3 + 8z^2 + 14z^3 + 63z^2 + 7z + 6z^2 + 27z + 3 = 16z^4 + 86z^3 + 77z^2 + 34z + 3$. The coefficients are then reduced to be in the range {0, 1, ... , 12}. This yields $3z^4 + 8z^3 + 12z^2 + 8z + 3$. However, the degree of the polynomial must be at most m-1 for elements of GF(p^m). A polynomial modulo operation is thus required. This involves dividing the intermediate product polynomial by another polynomial and taking the remainder polynomial as the result. The polynomial that is used to perform the modulo operation is a special polynomial called an irreducible polynomial for that finite field. The irreducible polynomial is a monic polynomial that cannot be factored into simpler terms over GF(p). For instance, let the

irreducible polynomial for the field GF(13^3) be z^3 -2. Then the polynomial modulo operation yields: $(3z^4 + 8z^3 + 12z^2 + 8z + 3) \bmod (z^3 - 2) = 12z^2 + z + 6$.

The integer modular arithmetic is referred to as subfield reduction and polynomial modulo operation as the extension field reduction.

5.1.1 Related work

There are three broad classes of algorithms for finite field multiplication: serial, serial-parallel, and parallel multiplication. Serial GF(p^m) multiplication involves processing all the coefficients of the multiplicand operand in parallel while processing the coefficients of the multiplier operand serially. Serial-parallel algorithms process more than one coefficient of the multiplier operand at a time. The fully parallel algorithms process both operands in parallel.

5.1.1.1 Element-serial algorithms

Algorithm 5.1: Serial GF(p^m) multiplication algorithm

Inputs: $A = \sum_{i=0}^{m-1} a_i z^i, B = \sum_{i=0}^{m-1} b_i z^i$ where a_i, $b_i \in$ GF(p) and an irreducible polynomial $f(z)$

Output: $C \equiv AB = \sum_{i=0}^{m-1} c_i z^i$ with $c_i \in$ GF(p)

$C = 0$

for $i = 0$ to m-1 do

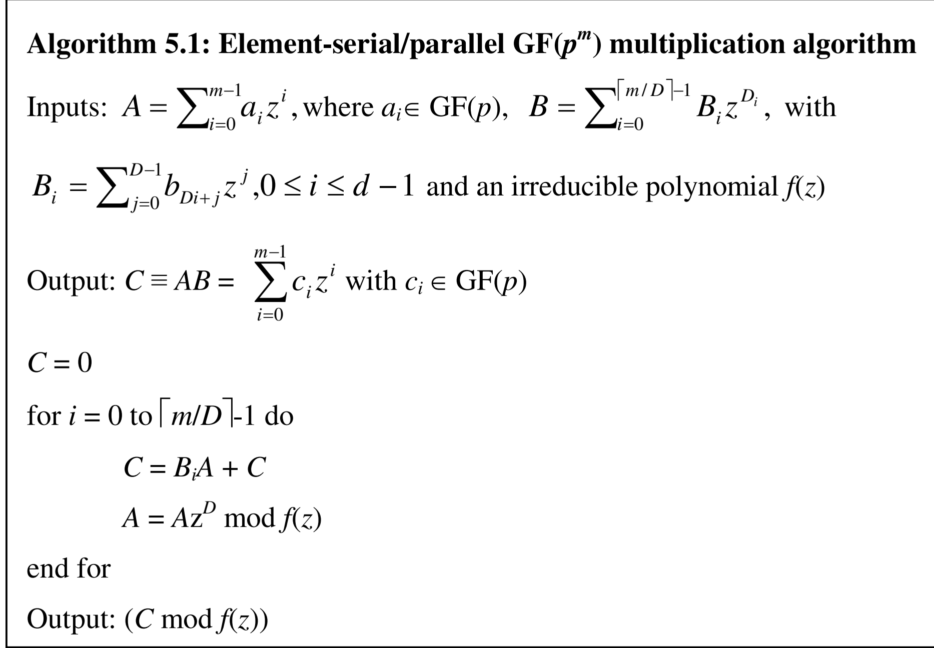
  $C = b_i A + C$

  $A = Az \bmod f(z)$

end for

Figure 5.1: Element-serial GF(p^m) multiplication algorithm.

5.1.1.2 Element-serial/parallel algorithms



Algorithm 5.1: Element-serial/parallel GF(p^m) multiplication algorithm

Inputs: $A = \sum_{i=0}^{m-1} a_i z^i$, where $a_i \in$ GF(p), $B = \sum_{i=0}^{\lceil m/D \rceil - 1} B_i z^{D_i}$, with

$B_i = \sum_{j=0}^{D-1} b_{Di+j} z^j, 0 \le i \le d-1$ and an irreducible polynomial $f(z)$

Output: $C \equiv AB = \sum_{i=0}^{m-1} c_i z^i$ with $c_i \in$ GF(p)

$C = 0$

for $i = 0$ to $\lceil m/D \rceil$-1 do

$C = B_i A + C$

$A = Az^D \bmod f(z)$

end for

Output: ($C \bmod f(z)$)

Figure 5.2: Element-serial/parallel GF(p^m) multiplication algorithm.

5.1.2 Contributions

The major contribution [38] in this section is a multiply-accumulate algorithm for a special class of GF(p^m) called Type II Optimal Extension Fields (OEFs) [3]. The proposed algorithm takes advantage of the special properties of this family of finite fields to perform efficient field operations. The concept of merged arithmetic is introduced to dissolve the boundaries between discrete modular multiplies and additions.

5.1.2.1 Merged-arithmetic GF(p^m) multiplication algorithm

Certain finite fields simplify the hardware implementation of finite field arithmetic. One such field is a prime Galois field where the prime is of the special form $2^n - c$,

where c is relatively "small" compared to the size of 2^n. The modulo operation is easily performed by successively substituting 2^n with the smaller integer c. For instance, say $p = 29 = 2^5 - 3$. Since $2^5 \equiv 3 \bmod 29$, we can replace the occurrence of 2^5 with 3. For example, $140 = 4 * 2^5 + 12 = 4*3 + 12 \equiv 24 \bmod 29$.

The same also holds for the polynomial modulo operation in GF(p^m). If the irreducible polynomial is in a simple form, then the polynomial modulo operation is simplified. For example, take the irreducible polynomial $f(z) = z^3 - 2$ for GF(13^3). To perform the polynomial modulo operation : $(3z^4 + 8z^3 + 12z^2 + 8z + 3) \bmod (z^3 - 2)$, we can use the congruence that $z^3 \equiv 2 \bmod f(z)$. Every occurrence of z^3 can be replaced with 2. This yields: $3z^4 + 8z^3 + 12z^2 + 8z + 3 = 3*2z + 8*2 + 12z^2 + 8z + 3 = 6z + 16 + 12z^2 + 8z + 3 = 12z^2 + z + 6$.

Optimal Extension field is another type of special finite field that facilitates simpler arithmetic introduced by Bailey and Paar [3] in 1998.

The specifications are for a Galois extension field GF(p^m) with the following restrictions:

- the prime p must be of the form $p = 2^n \pm c$ with $\log_2 c < n/2$
- the irreducible polynomial used for the polynomial modulo operation must be a binomial

This Optimal Extension Field (OEF) is further divided into two classes:

Type I is the case where the prime is of the form $2^n \pm 1$ – that is the value c is 1 which further simplifies the arithmetic for the integer modulo operation.

Type II is the case where the irreducible polynomial is of the form $f(z) = z^m - 2$ which further simplifies the arithmetic for the polynomial modulo operation.

In this dissertation, another class of Optimal Extension Fields that make for simpler and faster arithmetic is introduced.

The specifications are for a Galois extension field $GF(p^m)$ with the following restrictions:

- the prime p must be of the form $p = 2^n - c$ with $(2\log_2 c + \log_2 m + 1) \leq n$
- the irreducible polynomial used for the polynomial modulo operation must be $f(z) = z^m - 2$.

This special Optimal Extension Field (OEF) facilitates merged arithmetic which allows several multiply operations to be combined together. In addition, it allows subfield and extension field reductions to be combined together. Furthermore, the cost of one carry-propagate addition is distributed over several multiply operations when performing arithmetic in this finite field.

This special class of OEFs are a subset of the regular Type II OEFs and do not introduce any addition security considerations. Table 5.1 shows some examples of the special OEFs.

Table 5.1: Examples of the special OEFs.

p	f	parameters
$2^{13} - 1$	$z^{13} - 2$	$mn = 169$
$2^{16} - 15$	$z^{13} - 2$	$mn = 208$
$2^{17} - 31$	$z^{13} - 2$	$mn = 221$
$2^{18} - 11$	$z^{13} - 2$	$mn = 234$
$2^{55} - 55$	$z^{3} - 2$	$mn = 165$
$2^{56} - 57$	$z^{3} - 2$	$mn = 168$
$2^{57} - 13$	$z^{3} - 2$	$mn = 171$

5.2 GF(2^m) polynomial arithmetic

GF(2^m) arithmetic is the special case of GF(p^m) arithmetic where the prime p = 2. Arithmetic units for the Galois field GF(2^m) are readily implemented in hardware. The Galois field GF(2) has two elements: 0 and 1. Arithmetic operations in GF(2) are performed modulo 2. GF(2) addition is implemented by an XOR gate, while GF(2) multiplication is implemented by an AND gate. Each element of the extension field GF(2^m) is usually represented as a polynomial of degree at most m-1 with binary coefficients. For this representation, the addition operation is bit-independent and can be performed in parallel. However, multiplication is more complex. For polynomial multiplication, the product of the coefficients of the operands is first computed to yield a polynomial of degree at most 2m - 2. This polynomial is then reduced modulo an irreducible polynomial $p(z)$ to yield the final result.

5.2.1 Related work

A number of algorithms have been proposed for arithmetic in binary extension fields. Most of the algorithms that do not require special irreducible polynomials are bit-serial in nature. Some parallel algorithms are efficient for fields that have an irreducible polynomial of a special form e.g. a trinomial of the form $z^m + z + 1$.

5.2.1.1 Bit-serial algorithms

The shift-and-add method for GF(2^m) multiplication is based on the observation that:

$a(z)\cdot b(z) = a_{m-1}z^{m-1}b(z) + ... + a_2z^2b(z) + a_1zb(z) + a_0b(z)$.

Let the irreducible polynomial $f(z)$ be of the form $z^m + r(z)$. Then, $z^m \equiv r(z) \pmod{f(z)}$ since $-1 \equiv 1 \pmod 2$. Therefore :

$$\begin{aligned} b(z)\cdot z &= z\cdot(b_{m-1}z^{m-1} + b_{m-2}z^{m-2} + \ldots + b_2 z^2 + b_1 z + b_0) \\ &= b_{m-1}z^m \quad + (b_{m-2}z^{m-1} + \ldots + b_2 z^3 + b_1 z^2 + b_0 z) \\ &\equiv b_{m-1}r(z) + (b_{m-2}z^{m-1} + \ldots + b_2 z^3 + b_1 z^2 + b_0 z) \pmod{f(z)} \end{aligned}$$

Thus, $b(z)\cdot z \bmod f(z)$ can be computed by a left-shift of the vector representation of $b(z)$, followed by addition of $r(z)$ to $b(z)$ if the bit b_{m-1} is 1.

Algorithm 5.3: Bit-serial GF(2^m) multiplication algorithm

INPUT: $a = (a_{m-1}, \ldots, a_1, a_0)$, $b = (b_{m-1}, \ldots, b_1, b_0)$

and irreducible polynomial $f(z) = z^m + r(z)$, and $r = (r_{m-1}, \ldots, b_1, b_0)$

OUTPUT: $c = a\cdot b$

Initialization: $c = 0$

For i from m-1 to 1 do

c = leftshift(c) + $c_{m-1}r$.

$c = c + b_i a$.

Output: c

Figure 5.3: Bit-serial GF(2^m) multiplication algorithm.

5.2.1.2 Bit-parallel algorithms

The bit-parallel algorithm uses the property of the irreducible polynomial to combine the polynomial modulo operation and the multiplication together. Most of these bit-parallel

algorithms are for special trinomials and pentanomials. Thus, the algorithms are not flexible, and are constructed on a case-by-case basis for each irreducible polynomial.

5.2.2 Contributions

The major contribution [39] in this section is a multiply-accumulate algorithm for a GF(2^m) fields that can be adapted to perform both scalar and vector computations.

5.2.2.1 Parallel GF(2^m) scalar and vector polynomial multiplication

The field GF(2^m) is associated with a monic irreducible polynomial $p(z) = z^m + f(z)$, where $f(z) = f_{m-1}z^{m-1} + f_{m-2}z^{m-2} + ... + f_1 z + 1$, $f_{m-1}, f_{m-2}, ... f_1 \in \{0,1\}$. The multiply-accumulate result : $c(z) + a(z) \cdot b(z) \bmod p(z)$, $a(z)$, $b(z)$, $c(z) \in$ GF(2^m), is computed in two steps. The polynomial multiply-accumulate operation over GF(2), $c(z) + a(z) \cdot b(z)$, is first performed and an intermediate polynomial of degree $2m - 2$ is obtained. A polynomial modulo operation is then performed to reduce its degree to at most m-1. Since the irreducible polynomial $p(z) = z^m + f(z)$, then $z^m \equiv f(z) \pmod{p(z)}$, $z^{m+1} = z \cdot z^m \equiv z \cdot f(z) \pmod{p(z)}$, ... , $z^{2m-2} = z^{m-2} \cdot z^m \equiv z^{m-2} \cdot f(z) \pmod{p(z)}$. The polynomial modulo operation can thus be performed by successively replacing z^{m+i} with $z^i \cdot f(z)$ for $m \le i \le 2m - 2$. Two sub-arrays are required for the entire GF(2^m) MAC operation. One for the multiply-add operation over GF(2) and the other for the polynomial modulo operation. These sub-arrays complete the following operations:

INPUTS:
$c(z) = c_{m-1}z^{m-1} + c_{m-2}z^{m-2} + . . . + c_1 z + c_0$
$a(z) = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + . . . + a_1 z + a_0$
$b(z) = b_{m-1}z^{m-1} + b_{m-2}z^{m-2} + . . . + b_1 z + b_0$
$p(z) = z^m + f(z) = z^m + f_k z^k + f_{k-1}z^{k-1} + ... + f_1 z + 1$

MUL-ACC over GF(2) : $d(z) = c(z) + a(z)\cdot b(z) =$

$$(c_0 \oplus a_0b_0)$$
$$+ (c_1 \oplus a_0b_1 \oplus a_1b_0)z$$
$$+ (c_2 \oplus a_0b_2 \oplus a_1b_1 \oplus a_2b_0)z^2$$
$$+ \ldots + (c_{m-1} \oplus a_0b_{m-1} \oplus a_1b_{m-2} \oplus \ldots \oplus a_{m-1}b_0)z^{m-1}$$
$$+ \ldots + (a_{m-1}b_{m-2} \oplus a_{m-2}b_{m-1})z^{2m-3}$$
$$+ (a_{m-1}b_{m-1})z^{2m-2}$$

$$d(z) = d_{2m-2}z^{2m-2} + d_{2m-3}z^{2m-3} + \ldots + d_1z + d_0$$

POLY-REDC: $d(z)$ mod $p(z) =$

$$(d_0 \oplus d_m) +$$
$$(d_1 \oplus f_1d_m \oplus d_{m+1})z +$$
$$(d_2 \oplus f_2d_m \oplus f_1d_{m+1} \oplus d_{m+2})z^2 + \ldots +$$
$$(d_k \oplus f_kd_m \oplus \ldots \oplus f_2d_{m+k-2} \oplus \ldots \oplus d_{m+k})z^k +$$
$$(d_{k+1} \oplus f_kd_{m+1} \oplus \ldots \oplus f_2d_{m+k-1} \oplus \ldots \oplus d_{m+k+1})z^{k+1} + \ldots +$$
$$(d_{m-2} \oplus f_kd_{2m-k-2} \oplus \ldots \oplus f_2d_{2m-4} \oplus \ldots \oplus d_{2m-2})z^{m-2} +$$
$$(d_{m-1} \oplus f_kd_{2m-k-1} \oplus \ldots \oplus f_2d_{2m-3} \oplus f_1d_{2m-2})z^{m-1} +$$
$$(f_kd_{2m-k} \oplus \ldots \oplus f_3d_{2m-3} \oplus f_2d_{2m-2})z^m + \ldots +$$
$$(f_kd_{2m-2})z^{m+k-2}$$

Two rounds of the POLY-REDC operation suffice to reduce the degree of the intermediate polynomial $d(z)$ to at most m-1 if the generating polynomial is of the form $z^m + f(z)$, with the degree($f(z)$) = $k < m/2$.

Chapter 6

Polynomial Modular Multiplier Architectures

6.1 GF(p^m) architectures

GF(p^m) architectures can be broadly classified into element-serial, element-serial/parallel and parallel architectures. The distinction between the different classes is in the number of coefficient elements of the multiplier operand that are processed simultaneously. In all the three groups, the multiplicand operands are processed in parallel. There is an area-speed trade-off among the three architectures. The element-serial provides the smallest area but requires a large number of iterations to complete one GF(p^m) multiplication. On the other hand, the fully parallel architecture requires a large silicon area but completes the entire GF(p^m) multiplication in one step. Figure 6.1 depicts the three broad classes GF(p^m) multiplier architectures.

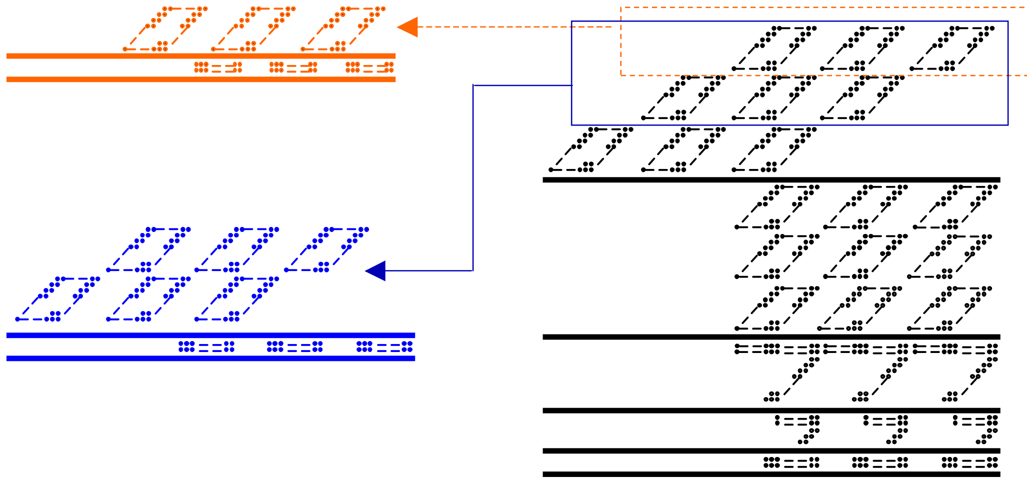


Figure 6.1: The three classes of GF(p^m) multiplier architecture.

6.1.1 Related work

Finite field multipliers for GF(p^m) have been extensively investigated for the case where p is relatively "small" (p = 2, 3 and $p < 2^8$) . Particularly, a number of parallel multiplier schemes for the binary extension field GF(2^m) where $p = 2$ have been proposed. A number of these multiplier architectures are based on the polynomial basis representation. The gate complexity and delay of these schemes is dependent on the choice of the irreducible polynomial for the extension field reduction as subfield operations involves modulo 2 or one's complement arithmetic and is readily implemented by XOR gates. Thus, most of the schemes focus on low-weight reduction polynomials with certain desirable properties. One of the first parallel polynomial basis multipliers for GF(2^m) was presented by Bartee and

Schnedier [5]. Another parallel multiplier design proposed by Mastrovito [25][26] formulates the entire GF(2^m) multiplication as a set of matrix operations. Mastrovito-type multipliers optimized for certain classes of irreducible polynomials have also been investigated by Reyhani-Masoleh and Hasan [34] and Rodriguez-Henriquez and Koc [36]. In addition to parallel multipliers, bit-serial multipliers have also been proposed for GF(2^m). Beth and Gollman [7] describe various msb- and lsb-first bit-serial multipliers. Song and Parhi [42] proposed a digit-serial/parallel multiplier for GF(2^m). A number of arithmetic units for GF(2^m) using the normal basis representation have also been presented in the literature [1][2][15][24].

Recently, there has been an interest in GF(p^m) multipliers for the case $p = 3$ because of the bandwidth advantages of finite fields of characteristic 3 in the implementation of identity-based cryptosystems [8]. Page and Smart [32] proposed a multiplier architecture specifically for GF(3^m) arithmetic. Bertoni et. al. [6] presented a digit/element architecture for GF(p^m) based on a generalization of the serial/parallel multiplier introduced by Song and Parhi [42], and also provided implementation details for GF(3^m). For GF(p^m) architectures where $p < 2^8$, some specialized GF(p) multipliers have been proposed as the basic building blocks. Some of these specialized GF(p) multipliers are table lookup-based [4][21] and utilize index calculus to transform the multiplication operation to an addition. Other specialized GF(p) multiplier schemes [18] [20] that have been proposed for $p < 2^5$, are based on combinatorial logic for the full truth-table of the modular multiplication operation.

A number of GF(p^m) multiplier architectures have been introduced for the case where p is a large prime and m is 1, the prime Galois field GF(p). Some of these scalable, bit-serial architectures [17] [40] are capable of performing both GF(p) and GF(2^m) operations.

6.1.1.1 Element-serial architectures

Element-serial architectures process the coefficients of the multiplier operand serially and the coefficients of the multiplicand operand in parallel. The basic components of this architecture are modular multipliers and modular adders. The modular multipliers can be implemented as bit-serial, digit-serial or parallel multipliers. Each modular multiplier representing the individual coefficient multiplication operates in parallel with all the others. Modular multipliers may also be required for the polynomial modulo operation performed at each step.

6.1.1.2 Element-serial/parallel architectures

Element-serial/parallel architectures process multiple coefficients of the multiplier operand at a time. The coefficients of the multiplicand operand are processed in parallel just like the element-serial architecture. The basic components of this architecture are also modular multipliers and modular adders. The outputs of the modular multipliers can be added iteratively or with a tree of modular adders. The utilization of a tree structure for summing provides greater benefits when the number of elements of the multiplier operand processed in parallel is large.

6.1.2 Contributions

The major contribution [38] in this section is a parallel multiply-accumulate architecture that utilizes merged arithmetic to combine subfield and extension field

reductions. The multiply-accumulate scheme delays the final carry-propagate additions until the partial products from the multiplication, the subfield reduction and the extension field reduction have been fully accumulated. This enables the cost of a single carry-propagate addition to be amortized over several multiply operations. The design requires a large area for practical implementation, but exploits massive parallelism at the subfield and extension field levels. In addition, the design can be pipelined to further improve the throughput.

As a trade-off between area and delay, a modified version of Bertoni et. al's [6] generalized digit/element-based GF(p^m) architecture that utilizes merged arithmetic is also implemented. This modified architecture provides an improvement over Bertoni et. al's scheme.

6.1.2.1 Merged-arithmetic GF(p^m) multiplier architecture

This section presents a multiply-accumulate architecture for multipliers over a special class of Type II Optimal Extension Fields (OEFs) discussed in the previous section. The Type II OEF multiplier presented uses merged arithmetic to combine multiple multiply and addition operations together. Unlike previous work, the multiplier also performs subfield and extension field reduction in parallel for this class of finite fields. Though the multiplier design requires large silicon area for practical implementation, it obviates the need for performing subfield and extension field reduction separately, thereby reducing the overall delay.

Hardware implementation of prime field reduction

The elements of a prime field GF(p), can be realized as the integers modulo the prime p. The advantages of some special classes of prime numbers for efficient modular arithmetic have been noted by Crandall [11]. Pseudo-Mersenne primes are one such class of primes. A pseudo-Mersenne prime of the form $p = 2^n - c$ for some integers n and c with $\log_2 c < n/2$ simplifies the modular reduction operation. This simplification is possible because of the congruence $2^n \equiv c \pmod{p}$, which allows for 2^n to be interchanged with the "smaller" c. The same concept can also be applied to high-speed parallel multipliers. Parallel multipliers using Dadda [12] or Wallace [48] trees perform multiplication in three stages. The first stage generates the bit product matrix using an array of AND gates, the next stage compresses the array down to sum and carry vectors using half-adders and full-adders. The half- and full-adders only propagate carries at most one bit position. The final stage uses a carry-propagate adder (CPA) to add the sum and carry vectors. Figure 6.2 shows a dot diagram for an 8x8-bit Wallace tree.

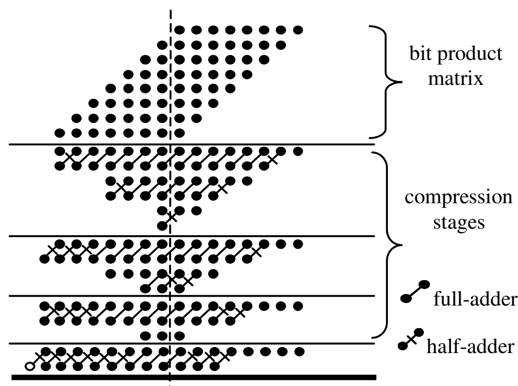


Figure 6.2: Dot diagram for 8x8 Wallace tree.

The output of the Wallace [48] tree can be reduced modulo a pseudo-Mersenne prime without first adding the sum and carry vectors with a CPA. Figure 6.3a shows the first step to perform this modular reduction. The bits in the most significant half of the output of the Wallace tree are ANDed with the bits of c to form a new bit product array. Each bit in the left half becomes a new row of gray dots. The new bits are represented by the diagonal gray dots in Figure 6.3a. This bit-product array is also reduced with a Dadda/Wallace tree. Note that the height of this bit product array is $2c + 2$, and is much smaller than the nxn multiplication array if $c << n$.

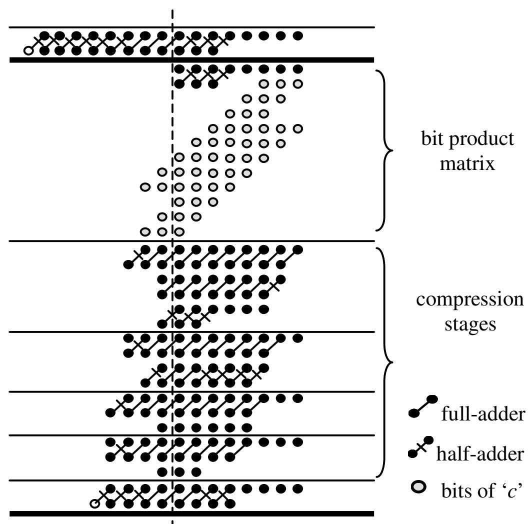


Figure 6.3a: Dot diagram for modular reduction.

The bits in the left half of the output of the second Wallace tree form a third array which is compressed to two rows of output bits as shown in Figure 6.3b. The final two rows are then summed with a CPA. Provided $\log_2 c << n/2$, the final result will be at most n+2 bits.

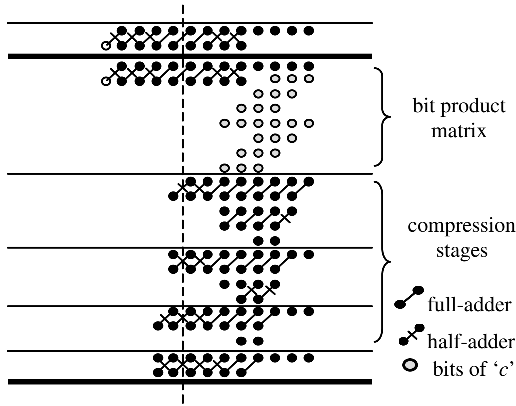


Figure 6.3b: Dot diagram for modular reduction.

Hardware implementation of extension field reduction

For extension field reduction, emphasis is placed on a special family of Type II OEFs in this work. This special class of Type II OEF is the Galois field GF(p^m) such that:

1. $p = 2^n - c$ for some integers n and $c > 0$, with $(2\log_2 c + \log_2 m + 1) \le n$; and
2. an irreducible polynomial $f(z) = z^m - 2$ exists in $F_p[z]$.

Most Type II OEFs already satisfy the extra requirement of this special class i.e. $\log_2 c \ll n/2$. For this special class of Type II OEF, a parallel multiplier scheme that uses the concept of merged arithmetic is presented. Merged arithmetic was proposed by Swartzlander [43] to implement a combination of multiplication and addition functions together. Figure 6.4 shows a parallel multiplier for GF(p^3) where $p = 2^n - c$, $(2\log_2 c + \log_2 3 + 1) \le n$, and $f(z) = z^m$-2. The polynomial product $c(z)$ and its coefficients c_t are computed as follows:

$$c(z) = a(z)b(z) = \left(\sum_{i=0}^{m-1} a_i z^i\right)\left(\sum_{j=0}^{m-1} b_j z^j\right)$$

$$\equiv \sum_{t=0}^{2m-2} c_t z^t \equiv c_{m-1} z^{m-1} + \sum_{t=0}^{m-2} (c_t + 2c_{t+m}) z^t \pmod{f(z)}$$

where $$c_t = \sum_{i+j=t} a_i b_j \bmod p.$$

Since the irreducible polynomial for Type II OEF is of the form $f(z) = z^m - 2$, then $z^m \equiv 2 \bmod f(z)$ and $z^{m+i} \equiv 2z^i \bmod f(z)$ for $m \leq i < 2m-1$. The columns of bit-product arrays of degree $m + i$ can be multiplied by 2, or shifted and included in the array column of degree i. This is depicted in the second stage in Figure 6.4. This yields an mxm matrix of nxn-bit product arrays. Using the concept of merged arithmetic, a column of m (nxn)-bit arrays can be compressed together in approximately $\log_{1.5}(mn)$ stages with a Wallace [48] or Dadda [12] tree. This is faster than if the column compression for each array was performed separately. The output of the compression tree for each column will be at most $2n + \log_2 m + 1$ bits wide.

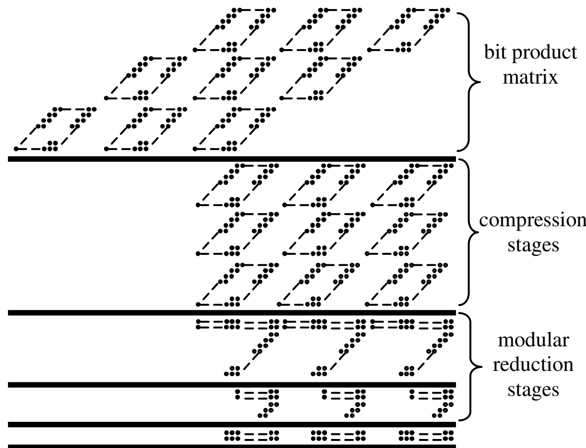


Figure 6.4: Parallel multiplier scheme for GF(p^3), $p = 2^n - c$, $\log_2 c \ll n$, and $f(z) = z^m - 2$.

Provided $(2\log_2 c + \log_2 m + 1) \leq n$, using a similar principle as presented for GF(p) in the previous section, at most two rounds of reduction with the bits of c will suffice to reduce each column to at most $n + 2$ bits.

The gate complexity of each column:

- mn^2 AND gates

- $mn^2 - 2(2n+\lg_2 m)$ full-adders
- $2(\lg_2 c)(n+\lg_2 m)$ AND gates
- $[2(\lg_2 c)(n+\lg_2 m)+2n] - [2(\lg_2 c+ \lg_2 m+n)]$ full-adders
- $2(\lg_2 c)(\lg_2 c+\lg_2 m)$ AND gates
- $[2(\lg_2 c)(\lg_2 c+\lg_2 m) +2n] - [2(n+2)]$ full-adders
- 1 carry-propagate adder

The total area for all the m columns is m times the gate complexity of one column. The total delay in terms of AND gate delay (T_A) and full-adder delay (T_{FA}):

- 1 T_A (bit array generation)
- $\lg_{1.5} mn$ T_{FA} (compression of the mn rows)
- 1 T_A (bit array generation)
- $\lg_{1.5}(2\lg_2 c +2)$ T_{FA} (subfield reduction)
- 1 T_A (bit array generation)
- $\lg_{1.5}(2\lg_2 c +2)$ T_{FA} (subfield reduction)
- 1 carry-propagate adder delay

The gate complexity and delay of the multiplier can be further reduced. A carry-delayed adder (CDA) [31] is utilized to minimize the height of the reduction trees in the modular reduction rounds. The CDA is a two-level carry save adder. The carry-delayed adder produces a pair of integers (D, T) called a carry-delayed number, using the following set of equations:

$$S_i = A_i \oplus B_i \oplus C_i, \quad C_{i+1} = (A_i \wedge B_i) \vee (A_i \wedge C_i) \vee (B_i \wedge C_i),$$
$$T_i = S_i \oplus C_i, \quad D_{i+1} = S_i \wedge C_i, D_0 = 0$$

The values D_{i+1} and T_i are the outputs of a half-adder cell with inputs S_i and C_i. An important property of the carry-delayed adder is that $D_{i+1} \wedge T_i = 0$ for all $i = 0,1,....n-1$. This is easily verified as:

$$D_{i+1} \wedge T_i = S_i \wedge C_i \wedge (S_i \oplus C_i)$$
$$= S_i \wedge C_i \wedge ((\overline{S_i} \wedge \mathrm{C_i}) \vee (\mathrm{S_i} \wedge \overline{\mathrm{C_i}})) = 0$$

In the first round, each column of *mn* rows is compressed to two rows. An extra half-adder stage is introduced in the most-significant half of th-e output of the Wallace/Dadda tree to guarantee the property of the carry-delayed adder that $D_{i+1} \wedge T_i = 0$. The bits of D and T now represent the most-significant half of the output of the first round. The bits of both T and D are ANDed with the bits of c. The appropriately shifted outputs of the AND gates are then "ORed" together. This approach reduces the height of the modular reduction rounds by half. Figure 6.5 shows the transformation of the output of the carry-delayed adder for a field GF(p^m) where $p = 2^8 - c$ and $\log_2 c \leq 3$.

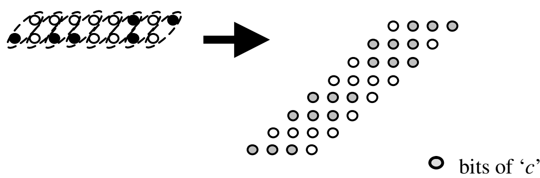


Figure 6.5: Transformation of the CDA output.

The gate complexity of the scheme using the CDA after the compression rounds involves extra half-adders and OR gates. However, the heights of the reduction trees for the modular

reduction rounds are reduced by half and this reduces the number of full-adders needed for the Wallace/Dadda trees in those stages.

Implementation

To obtain area and delay estimates, a specialized C++ program was developed that generates low-level structural Verilog models for GF(p^m) multipliers for any given value of p and m, where p is a pseudo-Mersenne prime. The GF(p^m) multipliers use a fast carry-lookahead adder for the final addition. The designs were synthesized using the Synopsys suite of tools on a 0.18µ standard cell library. The C++ program was used to generate Verilog models of multipliers for three of the special OEFs in Table 5.1. The fields are selected to have field orders large enough for current security needs and field parameters not easily susceptible to the generalized GHS attack [13]. The delay and area estimates are presented in Table 6.1 and 6.2. For easy comparison between the two schemes presented, the percentage decrease in delay and area provided by the carry-delayed adder (CDA) designs is also shown. The CDA versions have a smaller area because the reduction in the heights of the modular reduction trees more than compensates for the extra half-adder stage and OR gates. The number of gate delays in the critical path is a function of the logarithm of the parameters (n, m and c), and the extent of the benefit of the CDA version depends on the parameters of the GF(p^m) multiplier.

Table 6.1: Delay estimates for GF(p^m) multipliers.

	Unpipelined delay (nanoseconds)		
Field parameters	No CDA	CDA	% decrease
GF($(2^{13} - 1)^{13}$)	38.98	36.29	6.9%
GF($(2^{18} - 11)^{13}$)	56.55	55.34	2.1%
GF($(2^{57} - 13)^{3}$)	52.81	42.03	20.41%

Table 6.2: Area estimates for GF(p^m) multipliers.

	Area (equivalent gates)		
Field parameters	No CDA ($x10^3$)	CDA ($x10^3$)	% decrease
$GF(2^{13}-1)^{13}$	152.73	136.03	10.9%
$GF(2^{18}-11)^{13}$	306.75	271.33	11.6%
$GF(2^{57}-13)^{3}$	155.18	120.85	22.1%

Comparison to related work

Much of the prior work relating to OEFs has focused on software implementation [3] [16]. Großschädl et al. [16] explored the use of multiply-accumulate instructions to support OEF arithmetic.

Bertoni et. al. [6] presented a generalized digit/element GF(p^m) multiplier architecture. The basic idea of the scheme involves processing all the coefficients of the multiplicand $b(z)$ in parallel, while processing D elements of the coefficients of the multiplier operand $a(z)$ in each step. The extension field reduction is also performed in each step. This scheme is a generalization of the serial/parallel GF(2^m) architecture proposed by Song and Parhi [42]. The architecture requires approximately m/D cycles to complete one GF(p^m) multiplication. The C++ program was customized to generate structural Verilog models for the architectures presented in [6] and [42] based on the optimal irreducible polynomials. In addition, a modified version of the Bertoni et. al's [6] scheme that incorporates merged arithmetic was also developed. In this modified scheme, the boundaries between discrete modular multiplies and modular additions are dissolved. Table 6.3 compares the merged scheme with the architectures proposed in [6] and [42]. The percentage reduction in area and delay provided by the merged arithmetic version over the architecture in [6] is also presented. The GF(2^{173})

and GF(2^{239}) fields are chosen because they have about the same field orders with the parameters used for the GF(p^m) fields.

Table 6.3: Comparison with Song/Parhi and Bertoni et al.

	Delay (nanoseconds)			
	GF(2^{173})	GF(2^{13} -1)^{13}		
D	[42]	[6]	Merged	% decr.
1	1.35	30.89	22.88	25.93%
2	3.05	42.08	26.17	37.81%
4	4.14	55.16	30.69	44.36%

	Area (equivalent gates)			
	GF(2^{173})	GF(2^{13} -1)^{13}		
D	[42]	[6]	Merged	% decr.
1	311	20020	20381	-1.80%
2	2192	40538	35673	12.00%
4	3550	86283	66831	22.54%

	Delay (nanoseconds)			
	GF(2^{239})	GF(2^{18} -11)^{13}		
D	[42]	[6]	Merged	% decr.
1	1.35	49.51	37.62	24.02%
2	2.91	61.46	42.65	30.61%
4	3.63	80.41	46.43	42.26%

	Area (equivalent gates)			
	GF(2^{239})	GF(2^{18} -11)^{13}		
D	[42]	[6]	Merged	% decr.
1	429	45937	48699	-6.01%
2	1731	93254	73291	21.41%
4	5339	189423	129688	31.54%

Note that the case where the digit size $D = 1$ is equivalent to a bit-serial architecture. For $D >$ 1, the coefficient elements are summed with a tree of modular adders as in [6]. As the digit/element size is increased, the merged scheme provides greater improvement over

Bertoni's [6] scheme as several modular multipliers and modular adders are merged together, avoiding multiple carry-propagate adder delays. Compared to the parallel multiply-accumulate scheme, these digit/element schemes have smaller areas but require more than one cycle to complete a single multiplication. The multiplier GF(2^m) in [42] provides a very low cycle time since the subfield operations are basically XORs. Bit-serial multipliers for the prime field GF(p) are made up of carry-save adders. The bit-serial schemes require a lot more cycles to complete a modular multiplication. As noted in [6] fields of characteristic 2 are difficult to surpass if both area and time performance measures are considered.

6.2 GF(2^m) architectures

GF(2^m) architectures can be broadly classified into bit-serial and bit-parallel architectures. The bit-serial architectures require several clock cycles to complete one GF(2^m) multiplication but have a low area overhead. On the other hand, the bit-parallel architectures a low delay but high area overhead. Mastrovito multipliers [25][26] are an example of bit-parallel architectures.

6.2.1 Related work

A parallel polynomial basis GF(2^m) multiplier was first suggested by Bartee and Schneider [5]. Another multiplier scheme was proposed by Mastrovito [25][26] for polynomial basis multiplication. The Mastrovito design formulates the polynomial multiplication as a set of matrix operations. The gate complexity of the Mastrovito multiplier depends on the choice of the irreducible polynomial used and is suited to special classes of reduction polynomials. The multiplier schemes by Reyhani-Masoleh and Hasan [34] are very

similar to the Mastrovito scheme. The multiplier design presented by Rodriguez-Henriquez and Koc [36] are optimized for a special class of irreducible polynomial.

6.2.1.1 Bit-serial architectures

Bit-serial GF(2^m) multipliers implement Algorithm 5.3 in hardware. The figure below depicts a Most Significant Bit first (MSB) multiplier for $GF(2^5)$

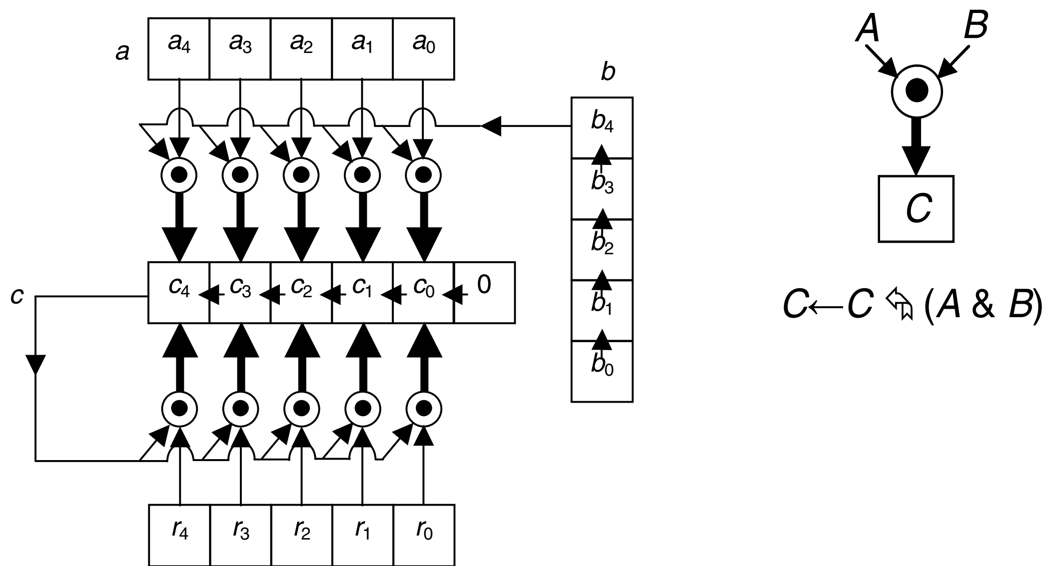


Figure 6.6: Bit-serial multiplier for $GF(2^5)$.

6.2.1.2 Bit-parallel architectures

Most bit-parallel GF(p^m) multiplier architectures are tailored to specific irreducible polynomials, particularly certain trinomials and pentanomials. Mastrovito [25] multipliers are an example of such multiplier architectures. Bit-parallel multipliers have a low delay but

a high area overhead and are inflexible since they can only be utilized for a specific irreducible polynomial.

6.2.2 Contributions

The main contribution [39] in this section is a parallel GF(2^m) multiply-accumulate architecture. In application areas where repeated finite field multiplications and additions are performed, the addition and multiplication operations can be combined together using a multiply-accumulate (MAC) unit. The vector MAC can be utilized in an environment where repeated GF(2^m) multiplications that have no dependencies need to be performed. Instead of serializing these individual operations, they can be performed in pairs.

6.2.2.1 Parallel scalar and vector polynomial multiplier architecture

This section presents a vector multiply-accumulate (MAC) architecture over the binary extension field GF(2^m) capable of supporting multiple precisions simultaneously. The vector MAC can perform one GF(2^m) or two GF($2^{(m/2)}$) multiply-accumulates using essentially the same hardware as a scalar GF(2^m) Mastrovito-type multiplier. The vector capability is enabled by inserting mode-dependent masks in the bit product and reduction arrays of the GF(2^m) MAC. This architecture leverages an existing scalar structure for performing multiple operations in vector mode. Essentially the same hardware is shared between scalar and vector modes. Although there is a slight delay and area penalty for the mode-dependent masking, this overhead is relatively insignificant. Both the stand-alone scalar GF(2^m) MAC and the vector GF(2^m) MAC were implemented in structural Verilog and synthesized on a 0.18 micron standard cell library to compare the area and delay for different values of *m*.

Figure 6.7 shows a parallel $GF(2^8)$ MAC scheme in Dadda[12] dot notation. The ellipses represent XOR gates. In Figure 6.7, the gray dots represent the bits of $f(z)$ and the black dots the partial product bits. Let the degree of $f(z)$ be k. In the example in Fig. 6.7, $m = 8$ and $k = 4$, so $f(z) = f_4z^4 + f_3z^3 + f_2z^2 + f_1z + 1$, $f_4, f_3, f_2, f_1 \in \{0, 1\}$. The irreducible polynomial is $p(z) = z^8 + f(z) = z^8 + f_4z^4 + f_3z^3 + f_2z^2 + f_1z + 1$. For example, $p(z) = z^8 + z^4 + z^3 + z + 1$ is a generating polynomial for $GF(2^8)$. The bit product matrix is generated by an array of AND gates. In the first compression round, bits in the same column are XORed together using a tree of XOR gates. In the next round, output bits representing a degree greater than or equal to m are ANDed with bits of $f(z)$, and shifted accordingly to form a new bit product array. Each output bit from the first round of degree greater than or equal to m becomes a new row of input bits in the next round. This is based on the congruence $z^{m+i} \equiv z^i \cdot f(z) \pmod{p(z)}$ for $m \leq i \leq 2m - 2$. Similarly, the next round involves column-wise XORs of the bits in the array.

The total delay in terms of AND gate delay (T_A) and XOR gate delay (T_X):

- T_A (to generate the bit product array for $a(z) \cdot b(z)$)
- $\log(m + 1)\ T_X$ (to reduce $m + 1$ rows to a single row)
- T_A (1st bit product array of $f(z)$ – gray dots)
- $\log(k + 2)\ T_X$ (to reduce the $(k+2)$ rows to a single row)
- T_A (2nd bit product array of $f(z)$ – gray dots)
- $\log k\ T_X$ (to reduce the new k rows to a single row)

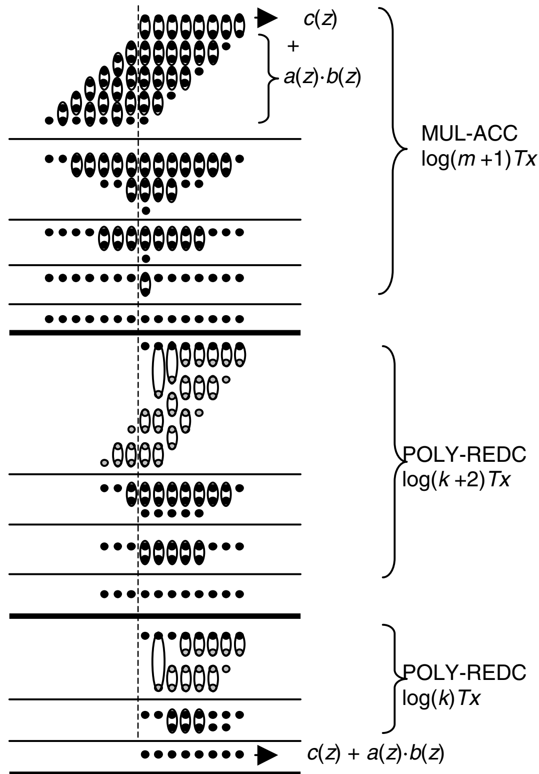


Figure 6.7: GF(2^8) parallel multiply-accumulate architecture.

The gate count:

- m^2 AND gates (bit product array for MUL-ACC)
- $(m^2 + m) - (2m - 1)$ XOR gates (MUL-ACC compression round)
- $(k + 1)\cdot(m - 1)$ AND gates (to generate the 1st bit product array for POLY-REDC)
- $m + (k + 1)\cdot(m - 1) - (k + m - 1) = k\cdot m - 2k + m$ XOR gates (POLY-REDC compression round)
- $(k + 1)\cdot(k - 1)$ AND gates (to generate the 2nd bit product array for POLY-REDC)
- $m + (k + 1)\cdot(k - 1) - m = (k + 1)\cdot(k - 1)$ XOR gates (POLY-REDC compression round)

Vector GF(2^m) Multiply-Accumulate Architecture

A MAC unit for GF(2^m) can be modified to support two GF($2^{m/2}$) MAC operations in vector mode. Figure 6.8 shows a shared segmentation [44] scheme to support two polynomial multiplications in parallel. The white regions in Figure 6.8 are masked (zero insert) depending upon the operating mode. The dark regions are not replaced with 'zeroes'. There is an extra area and delay overhead for masking and multiplexing the mode-dependent bits in the product array generation and in the XOR reduction trees. Figure 6.9 provides a detailed dot diagram for a GF(2^8)/GF(2^4) MAC scheme.

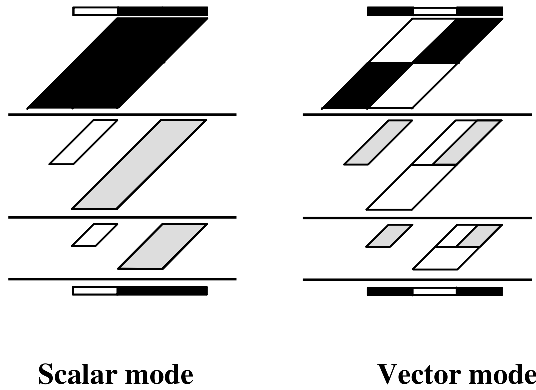


Figure 6.8: High-level view of the vector multiply-accumulate unit.

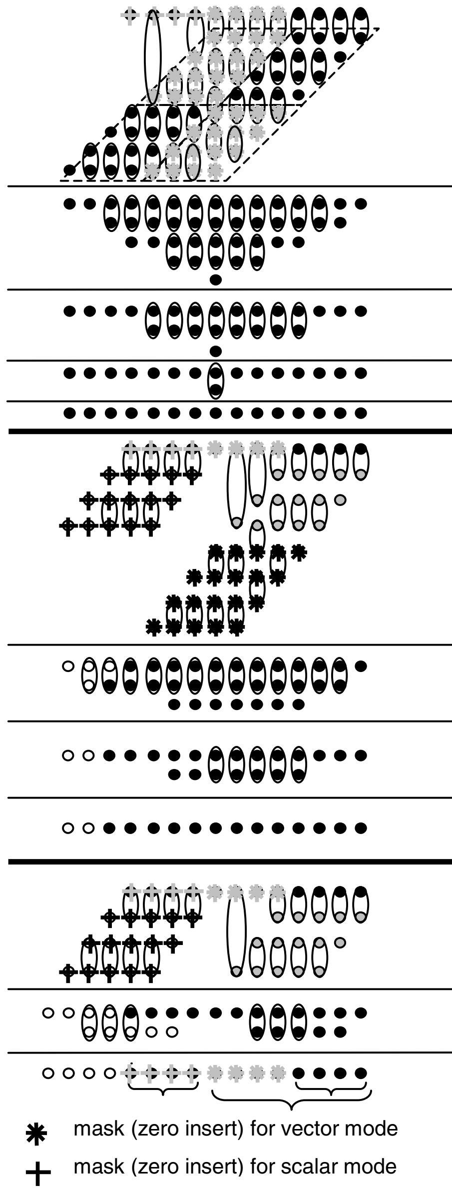


Figure 6.9: Dot diagram for a $GF(2^8)/GF(2^4)$ multiply-accumulate unit.

Implementation

In practice, for irreducible polynomials $z^m + f(z)$, the degree of $f(z)$ is usually much smaller than $m/2$. Let k be the degree of $f(z)$, empirical evidence suggests that the minimum possible value of $k << m/2$. Seroussi [41] tabulated low-weight binary irreducible polynomials over $GF(2^m)$ for $2 \leq m \leq 10{,}000$. All the irreducible polynomials were either trinomials $(z^m + z^k + 1)$, $m > k > 0$, or pentanomials $(z^m + z^{k3} + z^{k2} + z^{k1} + 1)$, $m > k_3 > k_2 > k_1 > 0$. The values were tabulated based on the smallest values of k and k_3 possible for the trinomials and pentanomials respectively. The highest tabulated value of k_3 for irreducible pentanomials was 56 for $m = 9760$. Thus, the MAC designs can be optimized for irreducible pentanomials where the value of k is significantly smaller than m. When higher-weight polynomials other than trinomials and pentanomials are considered, the value of k is even smaller. A Pari/GP[33] program was developed to find irreducible polynomials $p(z) = z^m + f(z)$ with the minimum possible degree of $f(z)$ for $2 \leq m \leq 256$. Unlike previous work, higher-weight polynomials other than trinomials and pentanomials were also considered. Table 6.4 gives the minimum values of k for irreducible polynomials $p(z) = z^i + f(z)$, degree($f(z)$) = k. In addition, Table 6.4 also shows the gate and transistor counts for the different values of m using the minimum value of k. The analysis assumes AND gates implemented with 6 transistors, and XOR gates implemented with 12 transistors. Note that this is a conservative estimate as an XOR gate can be implemented with only 4 transistors [10]. Table 6.5 shows the estimated gate delays for MACs with different bit-lengths. T_A denotes the delay of an AND gate and T_X denotes the delay of an XOR gate.

Table 6.4: Area estimates for scalar GF(2^m) multiply-accumulate units.

m	16	32	64	128	256
k	5	7	8	10	11
AND gates	370	1320	4648	17590	68440
XOR gates	351	1283	4579	17456	68176
AND transitors	2220	7920	27888	105540	410640
XOR transistors	4212	15396	54948	209472	818112
Total transistors	6432	23316	82836	315012	1228752

Table 6.5: Delay estimates for scalar GF(2^m) multiply-accumulate units.

m	16	32	64	128	256
k	5	7	8	10	11
(AND) T_A	3	3	3	3	3
(XOR) T_X	11	13	14	15	17

To obtain actual area and delay estimates, a specialized C++ program was developed to generate structural Verilog models for both the scalar and vector $GF(2^m)$ MAC designs for any given value of m and k. The C++ program was used to generate Verilog models of scalar and vector $GF(2^m)$ MAC designs for m = 16, 32 , 64, 128, and 256 bits . The values of k used for the implementation are the same as those of Table 6.4. The designs were synthesized on a 0.18 micron CMOS standard cell library. The normalized delay and area estimates are presented in Figure. 6.10 and 6.11 respectively. The "vectorized" MAC designs have a delay and area penalty because of the extra delay for the mode-dependent masking and

multiplexing. The extra overhead highly depends on the parameters m and k for the MAC designs. The overhead cost of the mode-dependent masking increases with higher values of k.

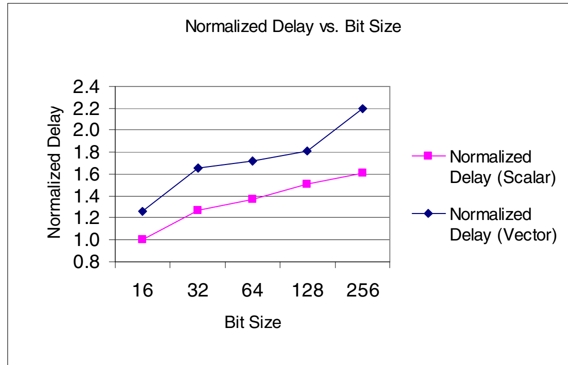


Figure 6.10: Normalized delay estimates for the scalar and vector GF(2^m) MAC architectures.

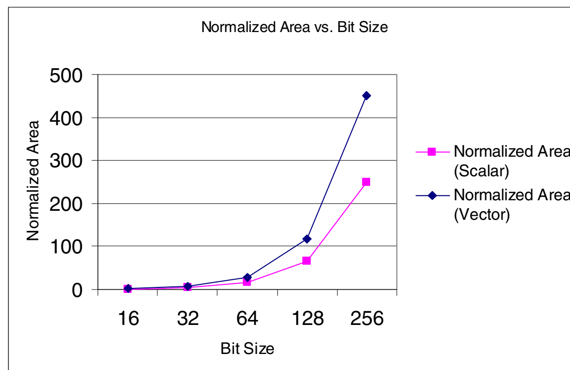


Figure 6.11: Normalized area estimates for the scalar and vector GF(2^m) MAC architectures.

Comparison to Related Work and Conclusion

The gate complexity and delay of the MAC designs presented in this work depend on the choice of the irreducible polynomial. In contrast to standard Mastrovito-type finite field multipliers, the MAC scheme presented here does not require the reduction polynomial to be

fixed. A $GF(2^m)$ MAC design with design parameters (m, k) can be used for arithmetic in any extension field $GF(2^i)$, $i \leq m$, provided that k is chosen such that there exists one or more reduction polynomials of the form $p(z) = z^i + f(z)$ where degree$(f(z)) \leq k$ for all $i \leq m$. The data from Seroussi's table [41] suggests that the minimum possible values of k are indeed very small for pentanomials. It should be noted that the flexibility of the $GF(2^m)$ MAC design comes at an extra cost to the gate complexity and delay compared to conventional Mastrovito-type $GF(2^m)$ multipliers.

Chapter 7

Conclusion

In this dissertation, innovations in algorithms and architectures for modular arithmetic have been presented:

- a parallel extension of Montgomery's modular multiplication algorithm
- an merged-arithmetic algorithm for a special class of Optimal Extension Fields
- an algorithm for scalar and vector mode GF(2^m) multiply-accumulate operation

7.1 GF(p) multiplier architectures

Chapter 4 presents the design space exploration for the implementation of the parallel Montgomery modular algorithm. In the chapter, four designs are introduced which trade-off speed and area to varying degrees. A proof of concept implementation and characterization is also presented.

7.2 GF(2^m) and GF(p^m) polynomial multiplier architectures

Chapter 5 presents a merged-arithmetic multiply-accumulate algorithm for a special family of Optimal Extension Fields (OEFs). The architecture for implementing the merged-arithmetic algorithm is presented in Chapter 6. In Chapter 5, an algorithm for adaptable for performing scalar and vector GF(2^m) multiply-accumulate operations is presented. The

hardware scheme for implementing the scalar and vector-mode GF(2^m) multiply-accumulate operations is presented in Chapter 6.

Bibliography

[1] G. Agnew, R. Mullin, I. Onyszchuk and S. Vanstone, "An implementation for a fast public-key cryptosystem," *Journal of Cryptology,* vol. 3, no. 2, 1991, pp. 63-79.

[2] G. Agnew, R. Mullin and S. Vanstone, "An implementation of elliptic curve cryptosystems over $F(2^{155})$," *IEEE Journal on Selected Areas in Communications,* vol. 11, 1993, pp. 804-813.

[3] D. Bailey and C. Paar, "Optimal extension fields for fast arithmetic in public-key algorithms," *Advances in Cryptology - CRYPTO '98*, (LNCS 1462), 1998, pp. 472-485.

[4] J. Bajard, L. Imbert, C. Negre and T. Plantard, "Efficient multiplication in $GF(p^k)$ for elliptic curve cryptography, " *Symposium on Computer Arithmetic*, 2003, pp. 181–187.

[5] T. Bartee and D. Schneider, "Computation with finite fields," *Information and Computer*s, vol. 5, 1963, pp. 79-98.

[6] G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar and T. Wollinger, "Efficient $GF(p^m)$ arithmetic architectures for cryptographic applications," *Topics in Cryptology-CT-RSA 2003* (LNCS 2612), 2003, pp. 158-175.

[7] T. Beth and D. Gollmann, "Algorithm engineering for public key algorithms," *IEEE Journal on Selected Areas in Communications,* vol. 7, 1989, pp. 458-465.

[8] D. Boneh and M. Franklin, "Identity-based encryption from the Weil pairing," *Advances in Cryptology –CRYPTO 2001* (LNCS 2139), 2001, pp. 213-229.

[9] E.F. Brickell, "A fast modular multiplication algorithm with application to two key cryptography," *Proceedings of Crypto 82*, pp. 51- 60, 1982.

[10] H. T. Bui, A. K. Al-Sheraidah and Y. Wang, "New 4-Transistor XOR and XNOR designs," *Proceedings of the 2nd IEEE Asia-Pacific Conference on ASIC*, Korea, 2000, pp. 25-28.

[11] R. E. Crandall, "*Method and apparatus for public key exchange in a cryptographic system*," U.S. Patent No. 5,159,632, 1992.

[12] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, 1965, pp. 349-356.

[13] C. Diem, "The GHS attack in odd characteristic," *Journal of Ramanunjan Mathematical Society,* vol. 18, no. 1, 2003, pp. 1-32.

[14] W. Diffie and M.E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, 1976, pp. 644-654.

[15] L. Gao, S. Shrivastava, and G. Sobelman, "Elliptic curve scalar multiplier design using FPGAs," *Cryptographic Hardware and Embedded Systems (*LNCS 2162), 2001, pp. 251-261.

[16] J. Großschädl, S. Kumar and C. Paar, "Architectural support for arithmetic in optimal extension fields," *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2004, pp. 111-124.

[17] J. Großschädl, "A bit-serial unified multiplier architecture for finite fields GF(p) and GF(2^m)," *Cryptographic Hardware and Embedded Systems* (LNCS 2162), 2001, pp. 202-219.

[18] J. Guajardo, T. Wollinger, and C. Paar, "Area efficient GF(p) architectures for GF(p^m) multipliers," *IEEE International Midwest Symposium on Circuits and Systems*, vol. 2, 2002, pp. 37-40.

[19] D. Hankerson, A. Menezes and S.Vanstone. "*Guide to elliptic curve cryptography*", Springer-Verlag, New York, 2004.

[20] A. Hiasat, "Semi-custom VLSI design for RNS multipliers using combinatorial logic approach," *Third IEEE International Conference on Electronics, Circuits, and Systems*, 1996, pp. 935-938.

[21] G. Jullien, W. Lue and N. Wigley, "High throughput VLSI DSP using replicated finite rings," *Journal of VLSI Signal Processing,* vol.14, no. 2, 1996, pp. 207-220.

[22] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, 1987, pp. 203-209.

[23] A. K. Lenstra and E. R. Verheul "Selecting Cryptographic Key Sizes", *Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography,* 2000, pp. 446 – 465.

[24] P. Leong and K. Leung, "A microcoded elliptic curve processor using FPGA technology," *IEEE Transactions on VLSI Systems,* vol. 10, 2002, pp. 550-559.

[25] E. D. Mastrovito, "*VLSI architectures for computation in Galois fields*," Ph.D. thesis, Linkoping University, 1991.

[26] E. D. Mastrovito, "VLSI designs for multiplication over finite fields GF(2^m)," *Sixth Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-6)*, 1988, pp. 297-309.

[27] V. Miller, "Uses of elliptic curves in cryptography," *Advances in Cryptology,* vol. 218, 1985, pp. 417 - 426.

[28] P. L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, 1985, pp. 519-521.

[29] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Electronic*s, vol. 38, no. 8, 1965, pp. 114 - 117.

[30] G. E. Moore, "No Exponential is Forever … but We Can Delay 'Forever'", *International Solid State Circuits Conference – Digest of Technical Papers*, 2003, pp.20-23.

[31] M. J. Norris and G. J. Simmons, "Algorithms for high-speed modular arithmetic," *Congressus Numeratium*, vol. 31, 1981, pp. 153-163.

[32] D. Page and N. Smart, "Hardware implementation of finite fields of characteristic three," *Cryptographic Hardware and Embedded Systems* (LNCS 2523), 2002, pp. 529-539.

[33] *Pari/GP (Computer Algebra System)* available from: http://pari.math.u-bordeaux.fr/.

[34] A. Reyhani-Masoleh and M. A. Hasan, "Low complexity bit parallel architectures for polynomial basis multiplication over GF(2^m)," *IEEE Transactions on Computers*, vol. 53, 2004, pp. 945-959.

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

[36] F. Rodriguez-Henriquez and C. Koc, "Parallel multipliers based on special irreducible pentanomials," *IEEE Transactions on Computers*, vol. 52, 2003, pp. 1535-1542.

[37] M. O. Sanu, E. E. Swartzlander Jr. and C. M. Chase, "Parallel Montgomery Multipliers", *IEEE 15th International Conference on Application-specific Systems, Architectures and Processors*, 2004, pp. 63-72.

[38] M. O. Sanu and E. E. Swartzlander, "Multiply-Accumulate Architecture for a Special Class of Optimal Extension Fields", *IEEE 16th International Conference on Application-specific Systems, Architectures and Processors*, 2005.

[39] M. O. Sanu and E. E. Swartzlander, "A Vector Multiply-Accumulate Architecture for GF(2^m) ," *IEEE* 48*th International Midwest Symposium on Circuits and Systems*, 2005.

[40] E. Savas, A. Tenca and C. Koc, "A scalable and unified multiplier architecture for finite fields GF(p) and GF(2^m)," *Cryptographic Hardware and Embedded Systems (*LNCS 1965), 2000, pp. 277-292.

[41] G. Seroussi, "*Table of low-weight binary irreducible polynomials*," Hewlet-Packard Labs, HPL-98-135 Tech Report, 1998.

[42] L. Song and K. Parhi, "Low-energy digit-serial/parallel finite field multipliers," Journal of VLSI Signal Processing, vol. 19, 1998, pp. 149-166.

[43] E. E. Swartzlander Jr., "Merged arithmetic," *IEEE Transactions on Computers*, vol. C-29, 1980, pp. 946-950.

[44] D. Tan, A. Danysh, and M. Liebelt "Multiple-precision fixed-point vector multiply-accumulator using shared segmentation," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic ARITH-16*, 2003, pp. 12-19.

[45] C. D. Walter, "Logarithmic Speed Modular Multiplication," *Electronics Letters*, vol. 30, no. 17, 1994, pp. 1397-1398.

[46] C. D. Walter, "Montgomery Exponentiation Needs No Final Subtractions," *Electronics Letters*, vol. 35, no. 21, 1999, pp. 1831-1832.

[47] C. D. Walter, "An Overview of Montgomery's Multiplication Technique: How to make it Smaller and Faster," *Proceedings of the First Workshop on Cryptographic Hardware and Embedded Systems*, Springer Lecture Notes in Computer Science, vol. 1717, 1999, pp. 80-93.

[48] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Computers*, vol. 13, 1964, pp. 14-17.

[49] C.-W. Wu and Y.-F. Chou, "General modular multiplication by block multiplication and table lookup," *Proceedings of the IEEE Int. Symp. Circuits and Systems (ISCAS), (London)*, 1994, pp. 295-298.

[50] http://research.sun.com/projects/crypto/draft-ietf-tls-ecc-05.txt, *ECC Cipher Suites for SSL, IETF Internet-draft specifying the use of Elliptic Curve Cryptography with SSL*, July 2004.

VITA

Moboluwaji Olusegun Sanu was born in Nigeria, the son of Dr. and Mrs. A. O. Sanu. He received the Bachelor of Science degree from Obafemi Awolowo University, Nigeria in 1999. He also received the Master of Science in Engineering degree from the University of Texas at Austin in 2002. He is a member of the Application Specific Processing Group, The University of Texas at Austin, where his research focuses on efficient computer arithmetic and VLSI design for Galois fields.

Permanent Address: 1 Child Evangelism Drive, Iwo Road, Ibadan, Nigeria.

This dissertation was typed by the author.