View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Scholarship at UWindsor

University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations

Winter 2014

Efficient finite field computations for elliptic curve cryptography

Wangchen Dai
University of Windsor

Follow this and additional works at: http://scholar.uwindsor.ca/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation
Dai, Wangchen, "Efficient finite field computations for elliptic curve cryptography" (2014). *Electronic Theses and Dissertations*. Paper 5017.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

EFFICIENT FINITE FIELD COMPUTATIONS FOR ELLIPTIC CURVE CRYPTOGRAPHY

by

WANGCHEN DAI

A Thesis
Submitted to the Faculty of Graduate Studies
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada
2013

© 2013 Wangchen DAI

EFFICIENT FINITE FIELD COMPUTATIONS FOR ELLIPTIC CURVE CRYPTOGRAPHY

by

WANGCHEN DAI

APPROVED BY:

____

Dr. D. Wu
School of Computer Science

____

Dr. C. Chen
Department of Electrical and Computer Engineering

____

Dr. H. Wu, Advisor
Department of Electrical and Computer Engineering

December 11, 2013

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyones copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Finite field multiplication and inversion are two basic operations involved in Elliptic Cure Cryptosystem (ECC), high performance of field operations can be applied to provide efficient computation of ECC. In this thesis, two classes of fields are proposed for multipliers with much reduced time delay. A most-significant-digit first and a least-significant-digit first digit-serial Montgomery multiplications are also proposed, using novel fixed elements $R(x)$ which are different from x^m and x^{m-1}. Architectures of the proposed Montgomery multipliers are studied and obtained for the fields generated by the irreducible pentanomials, which are selected based on the proposed special finite fields. Complexities of the Montgomery multipliers in term of critical path delay and gate count of the architectures are investigated; the critical path delay of the proposed multipliers are found to be as good as or better than the existing works for the same class of fields. Then, implementation of the proposed multipliers ($m = 233$) using Field Programmable Gate Array (FPGA) is provided. In addition, an FPGA implementation of an efficient normal basis inversion algorithm is also presented ($m = 163$). The normal basis multiplication unit is implemented using a digit-level structure, and a C-code is written to generate the first coordinate of the product of two normal basis elements for all field size m.

Key Words: Montgomery multiplication, digit-serial, Elliptic Curve Cryptography, normal basis inverse, FPGA.

Dedication

I dedicate this thesis to my parents for supporting me to accomplish my master's degree at University of Windsor in Canada.

Acknowledgments

I would like to express my sincere gratitude and appreciation to everyone who helped make this thesis possible. I am deeply indebted to my supervisor Prof. Huapeng Wu, Professor of Electrical and Computer Engineering at University of Windsor, for guiding me throughout the writing of this thesis. As one of best teachers I have ever had, Professor Wu impressed upon me that a good teacher instructs students in matters far beyond those in textbooks. His broad knowledge and logical way of thinking have been of great value; without his detailed and constructive comments on my research, none of this thesis would be possible.

I would also grateful to my colleagues and friends, Yiruo He, Ya Tan, Ran Xiao and Shoaleh Hashemi Namin for their time and support.

Finally, I with to extend my gratitude to everyone at UWindsor's Faculty of ECE for their efforts during my study in the M.A.Sc. Program. I also gratefully acknowledge the financial support form University of Windsor and Professor Huapeng Wu.

Contents

Author's Declaration of Originality iii

Abstract iv

Dedication v

Acknowledgments vi

List of Figures x

List of Tables xii

List of Appendices xiv

List of Abbreviations/Symbols xv

1 Introduction 1

2 Mathematical Preliminaries 4

2.1 Finite Field and Representations 4

2.2 Montgomery Multiplication over $GF(2^m)$ 6

2.3 Elliptic Curve Cryptosystem 7

2.3.1 Elliptic Curves 7

2.3.2 Finite Field Inversion Using Normal Basis 9

2.3.3 Elliptic Curve Cryptosystem 10

3 A Review of Existing Work 13

4 Proposed Digit-serial Montgomery Multipliers **19**
4.1 Proposed Digit-Serial MSD First Montgomery Multiplier 19
4.1.1 Algorithm 20
4.1.2 General Architecture 21
4.1.3 Advanced Architecture 26
4.2 Proposed Digit-Serial LSD First Montgomery Multiplier 31
4.2.1 Algorithm 31
4.2.2 General Architecture 32
4.2.3 LFSR-Based Architecture 36
4.3 Complexity Analysis 38
4.4 FPGA Implementation of the Proposed Multipliers 42
4.4.1 Summary of the MSD-First Multiplier Implementation 42
4.4.2 Summary of the LSD-First Multiplier Implementation 43

5 FPGA Implementation of Inverse Generator **45**
5.1 The Design of Inverse Generator 45
5.1.1 REG1 Module 46
5.1.2 REG2 Module 47
5.1.3 MUX Module 48
5.1.4 Digit-level Normal Basis Multiplier Module and Multiplication Algorithm 48
5.1.5 Top-Level 52
5.2 Simulation and Compilation 52
5.2.1 Simulation Results 52
5.2.2 Compilation Results 54

6 Conclusions **59**

A C-code of $F(s)$ and the First Coordinate c_0 Generation **61**

B Generated VerilogHDL-code of the First Coordinate c_0 **66**

Bibliography **72**

Vita Auctoris **76**

List of Figures

2.1 Operations in an elliptic curve 7
2.2 Elliptic curve over binary field $GF(2^m)$ 9
2.3 Encryption/decryption of elliptic curve cryptosystem 10
2.4 Computation structure of ECC over $GF(2^m)$ 11

3.1 (a)Tang's architecture of $GF(2^{233})$ multiplier [17] (b)Kumar's architecture of $GF(2^m)$ multiplier [19] 15
3.2 Tang's architecture of partial product multiplier, generates the product of $A_j \times B$ [17] 16
3.3 Meher's block diagram of proposed field multiplier over $GF(2^m)$ [24] . . . 17
3.4 Work reported in [28], (a)$R(x) = x^m$, (b)$R(x) = x^{m-1}$ 18

4.1 Block diagram of proposed digit-serial MSD-first Montgomery multiplier when $R(x) = x^l$ 22
4.2 General architecture of the proposed multiplier when $R(x) = x^u$ 23
4.3 Implementation of equation (4.11) 25
4.4 Model 1: multiply by x structure 27
4.5 Implementation of computation $A(x)x^2 \mod f(x)$ 28
4.6 Model 2: multiply by x^{-1} structure 29
4.7 Implementation of $A(x)B_{s-i-1}(x)x^{-l} \mod f(x)$ 30
4.8 Advanced architecture of proposed multiplier 30
4.9 General architecture of the proposed digit-serial LSD first multiplier 33
4.10 LFSR-based architecture of the proposed LSD Montgomery multiplier . . . 37

5.1 Architecture of the designed inverse generator 46
5.2 Block diagram of the inverse generator for FPGA implementation 47
5.3 REG1 module 48

5.4 REG2 module 48
5.5 MUX module 49
5.6 Digit-level normal basis multiplier module 49
5.7 Digit-level Normal Basis multiplier structure 50
5.8 Simulation result of the Inverse Generator 56
5.9 RTL of the design 57
5.10 Technology map viewer of the design 58

List of Tables

1.1 Key size comparison between RSA and ECC with same secure level 2

2.1 Algorithm of Binary Field Bit-Parallel Montgomery Multiplication 6

3.1 Algorithm of Bit-Serial Montgomery Multiplication 14

3.2 Algorithm of Digit-Serial Montgomery Multiplication, where d is the digit size, $f_0'(x)f_0(x) = 1 \mod x^d$, $C_0(x)$ and $f_0(x)$ are the least significant digits of $C(x)$ and $f(x)$, respectively 14

4.1 Digit-serial MSD-first Montgomery Multiplier ($R(x) = x^l$), where $0 \leqslant l \leqslant d-1$ 21

4.2 Complexity of each block of the proposed MSD-first Montgomery multiplier 26

4.3 Complexity of proposed digit-serial MSD-first Montgomery multiplication (Algorithm I, general architecture, when $k_{i+1} - k_i \geqslant d-1, k_0 = 0, k_4 = m$ and $0 \leqslant l \leqslant d-1$) 26

4.4 Complexity of proposed digit-serial MSD-first Montgomery multiplication (Algorithm I, advanced architecture, when $k_{i+1} - k_i \geqslant max\{l, d-l-1\},\ i = 0,1,2,3,\ k_0 = 0,\ k_4 = m$ and $0 \leqslant l \leqslant d-1$) 31

4.5 Digit-serial LSD-first Montgomery Multiplier ($R(x) = x^{sl}$), where $l \geqslant 0$ . . 32

4.6 33

4.7 Complexity of digit-serial LSD Montgomery multiplication (Algorithm II, when $1 \leqslant l \leqslant d-1$ and $k_{i+1} - k_i \geqslant d-1,\ k_0 = 0,\ k_4 = m$) 34

4.8 Complexity of digit-level Montgomery multiplication (Algorithm II, when $l = d$, and $k_{i+1} - k_i \geqslant d-1,\ k_0 = 0,\ k_4 = m$) 34

4.9 Degree range of each term of equation (4.22) 35

4.10 Value of l in terms of XOR gate usage of block S1 35

4.11 Complexity of digit-level Montgomery multiplication (Algorithm II, when $l > d$, and $k_{i+1} - k_i \geqslant l,\ k_0 = 0,\ k_4 = m$) 36
4.12 LFSR-Based Digit-serial LSD-first Montgomery Multiplier ($R(x) = x^{sl}$), where $0 \leqslant l \leqslant d-1$ 36
4.13 Complexity of digit-level Montgomery multiplication (Algorithm III, when $0 \leqslant l \leqslant d-1$, and $k_{i+1} - k_i \geqslant max\{l, d-l-1\},\ k_0 = 0,\ k_4 = m$) 38
4.14 Intrinsic delay of XOR2 and AND2 gate, we assume each gate could drive a maximum of two gates (25°C, $1.8V$, CMOSP18 Tech., $Y = A \cdot B$, or $Y = A \oplus B$) 38
4.15 Digit-serial Montgomery multipliers comparison ($f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, $s = m/d$) 39
4.16 Proposed multipliers compared with Polynomial Basis finite field multipliers (MSD cases, $f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, $s = \lceil m/d \rceil$, T_{DFF} represents the time delay of a D-flipflop) 39
4.17 Proposed multipliers compared with Polynomial Basis finite field multipliers (LSD cases, T_M represents the time delay of a 2×1 Multiplexer, T_{TFF} represents the time delay of a T-flipflop) 40
4.18 Efficiency of the proposed multipliers and existing Montgomery multipliers ($m = 233$, $d = 8$, if $l < d$, then $l = 4$) 41
4.19 Efficiency of the proposed multipliers and existing PB multipliers ($m = 233$, $d = 8$, if $l < d$, then $l = 4$) 41
4.20 Cells usage of compilation ($m = 233$, $d = 8$, $u = 4$) 42
4.21 Gate count of each module ($m = 233$, $d = 8$, $u = 4$) 42
4.22 Time complexity of the design ($m = 233$, $d = 8$, $u = 4$) 43
4.23 Cells usage of compilation ($m = 233$, $d = 8$, $l = 4$) 43
4.24 Gate count of each module ($m = 233$, $d = 8$, $l = 4$) 43
4.25 Time complexity of the design ($m = 233$, $d = 8$, $l = 4$) 44

5.1 Description of Each Clock cycle 52
5.2 Cells usage of compilation 55
5.3 Area cost of each module 55
5.4 Operation delay of the design Inverse Generator over $GF(2^{163})$ 55

List of Appendices

C-code of $F(s)$ and the First Coordinate c_0 Generation 61
Generated VerilogHDL-code of the First Coordinate c_0 66

List of Abbreviations/Symbols

GF	Finite Field or Galois Field
PB	Polynomial Basis
NB	Normal Basis
EC	Elliptic Curve
ECC	Elliptic Curve Cryptosystems
RSA	Rivest, Shamir, Adleman
FPGA	Field Programmable Gate Array
ALUT	Adaptive Look Up Tables
MSD	Most Significant Digit
LSD	Least Significant Digit
LE	Logic Element
MUX	Multiplexer
XOR	Exclusive OR
TFF	T-Flipflop
DFF	D-Flipflop
LFSR	Linear Feedback Shift Register
VLSI	Very Large Scale Integrated Circuits

Chapter 1

Introduction

The development of cryptography can be divided into the following two stages [1]: classical cryptography, and modern cryptography. Classical cryptography was the study of the confidentiality of a message through encryption and decryption. An encryption operation can be described as the conversion of a message or a piece of information from comprehensible text into some incomprehensible form. Transposition cipher [3] and substitution cipher [2] are two representative classical ciphers.

Due to the rapid development of computer and network technologies, and the worldwide application of on-line trading services, mobile phones, and credit cards, the increasing threat to personal privacy and information security is becoming a significant challenge to security engineers. Under this context, cryptography is no longer just a concern for governments, but for civilians as well. Therefore, this field has been expanded far beyond communication confidentiality to include identity authentication, digital signatures, message integrity verification, etc. This extension has led to modern cryptography. Cipher algorithms in modern cryptography are achieved by using a key to encrypt and decrypt information. The Data Encryption Standard (DES) and the Advanced Encryption Standard (AES) are two symmetrical cipher algorithms created in modern cryptography. The encryption and decryption of these algorithms share the same key. The problem is that over time, more users know the key, and the risk of security breaches increases: once the key is revealed by one of the uses, the entire cryptosystem will be on longer secure.

During the 1970s, the public-key cryptosystem, known as the most notable advance in the field of cryptography after World War II, was invented [1]. A public-key system is an

asymmetrical key system that uses a public key to encrypt but decrypts with a private key. The concept of public-key cryptography was first raised in 1976 by Diffie and Hellman [6]; they demonstrated the possibility of network communication when the public key could be widely distributed, while its paired private key remains secret. After that, RSA, which was first published in 1978 by three talented scientists [7], is considered to be the most widely used public-key cryptosystem. To break RSA, a large-number factorization problem must be solved first. Later, elliptic curve cryptosystem (ECC), another public-key system was proposed by Koblitz [8] and Miller [9] during 1985 to 1987. The breaking of an ECC is equivalent to solving discrete logarithm problems. A RSA algorithm with 768-bit key size was broken in 2010 [4], while the hardest ECC scheme broken at present had only a 112-bit key size [5], ECC seems to be superior to RSA. The following table lists the key size in terms of security level with regard to these two public-key cryptosystems.

Table 1.1: Key size comparison between RSA and ECC with same secure level

RSA(bit)	ECC(bit)	Key Size Comparison
1024	160	6 : 1
2048	224	9 : 1
3072	256	12 : 1
7680	384	20 : 1

ECC uses a binary field $GF(2^m)$ or a prime field $GF(p)$. The encryption and decryption speed is an important indicator for evaluating an ECC algorithm. Efficiency of finite field arithmetic operation has great impact on the performance of an ECC, since an ECC computation consists a set of point operations and field multiplication and field inversion are the basic operations involved in the point operation. Due to the fact that field inversion also requires field multiplication during the computation, as a consequence, a large number of studies are mainly aimed at high-speed and efficient implementations of field multiplication.

The binary field $GF(2^m)$ is widely used in field operations because it is very suitable for VLSI implementation. However, the multiplication is more complicated and time-consuming. Efficient computation of field multiplication is one of the critical issue of public-key based cipher algorithms. In 1985, Montgomery introduced a new method for integer modular multiplication [10], and proved that the time-consuming trial division operation can be avoided. Later, Koc [12] extended the method to binary field and showed

that binary field multiplication can be implemented dramatically faster than standard multiplication. A number of Montgomery multipliers has been designed, and in general, the existing Montgomery multipliers can be divided into two styles: general styles including bit-serial, bit-parallel, and digit-level sub-types; and systolic styles. Bit-serial multipliers have the least gate count but require the longest time to process one operation. In contrast, bit-parallel multipliers have the smallest time delay but require largest implementation area. Digit-level multipliers are available to combine the advantages of both of them and balance the relationship between gate count and critical path delay by processing constant bits each clock cycle.

The works reported in this thesis mainly focus on the efficient computation and hardware implementation of digit-serial Montgomery multiplication. A most-significant-digit first and a least-significant-digit first digit-serial Montgomery multiplier are proposed; two novel fixed elements $R(x)$, which are different from the general ones (x^{m-1} and x^m), are applied. Two classes of fields for the multipliers with much reduced critical path delay are also proposed. Architectures of the proposed Montgomery multipliers are studied and obtained for the fields generated by the irreducible pentanomials. The complexities of the proposed multipliers in terms of gate count and critical path delay of the architecture are investigated, and demonstrated that the critical path delay of the proposed multipliers can be further reduced by applying the special finite fields. The contributions of this research work also consist of an FPGA implementation of the proposed Montgomery multipliers in the case where $m = 233$. Furthermore, an FPGA implementation of a normal basis inversion algorithm in $GF(2^m)$ is also presented in this thesis.

The outline of this thesis is as follows. Chapter 2 presents the mathematical background of the finite field, digit-level Montgomery multiplication, elliptic curve cryptosystem, and some other related equations and concepts. After that, a review of the existing literatures will be presented in Chapter 3. Chapter 4 presents the details of the proposed digit-serial Montgomery multipliers, the comparison results of the proposed Montgomery multipliers in terms of cell usage and critical path delay, the FPGA simulation, and a compilation report of the proposed works. Chapter 5 describes the FPGA implementation of the normal basis inversion generator and provides the results. Finally, in the last chapter, there will be a profound discussion regarding the conclusions and further work.

Chapter 2

Mathematical Preliminaries

This chapter introduces the relevant mathematical background. The definition of finite field as well as its two general representation methods, the definition of Montgomery multiplication, and the algorithm of elliptic curve cryptosystem (ECC) will be introduced in turn.

2.1 Finite Field and Representations

A finite field (or Galois field) is a group of finitely many elements in which both the addition and the multiplication are defined, also the usual algebraic laws, commutative, associative, and distributive can be applied [14]. The number of elements contained in a finite field is called the order of the field. A finite field can be denoted as $GF(q)$, where q is an positive integer number greater than one. The order of a nonzero element $A \in GF(q)$ is defined as the smallest positive integer k to make $A^k = 1$, and k always divides $q-1$. In cryptography, there are two kinds of finite fields that are commonly used: prime field $GF(p)$, where p is prime, and binary extension field $GF(2^m)$ where m is a positive integer great then or equal to two. The representation of field $GF(p)$ is simply a set of integers modulo p, however, unlike prime field, the binary field has many frequently-used representations. Polynomial basis representation and normal basis representation are the two methods commonly used to represent a binary field element.

In a polynomial basis representation, every element in $GF(2^m)$ is represented by a unique polynomial of degree less than m. For example, element A of $GF(2^m)$ can be represented as $A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0 = (a_{m-1}a_{m-2} \ldots a_1 a_0)$, and

the coefficient a_i of each term equals either 0 or 1. The polynomial basis is the set:

$$PB = \{x^{m-1}, x^{m-2}, \ldots, x^2, x, 1\} \tag{2.1}$$

Using polynomial basis to represent elements in binary field $GF(2^m)$ has been proved to be well suited. By applying such a representation, an addition operation in binary field can be very efficiently implemented by a single XOR gate, and a multiplication operation are defined simply as the product of the corresponding polynomials reduced by modulo $f(x)$. $f(x)$ is an irreducible polynomial which generates the binary field $GF(2^m)$, see equation (2.2). If we let only three f_i equals to one, where $1 < i < m$, we could have an irreducible pentanomial $f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, where $1 < k_1 < k_2 < k_3 < m$. The works reported in this thesis are focusing on the binary field due to its efficient implementation in both hardware and software.

$$f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1 x + 1 = 0,\ \textit{where}\ f_i = 0\ \textit{or}\ 1 \tag{2.2}$$

In normal basis representation, we use the basis set

$$NB = \{\theta^{2^{m-1}}, \theta^{2^{m-2}}, \ldots, \theta^2, \theta\} \tag{2.3}$$

to represent elements in the binary field, and elements θ^{2^i}, where $i \in [0, m-1]$, in the basis set must be linearly independent. Using normal basis, a binary field element $A = (a_{m-1}a_{m-2}\ldots a_1 a_0)$ can be represented by equation (2.4):

$$A = a_{m-1}\theta^{2^{m-1}} + a_{m-2}\theta^{2^{m-2}} + \cdots + a_1\theta^2 + a_0\theta \tag{2.4}$$

Normal basis representation has the computational advantage that 2^x-power operations can be implemented very efficiently by a left-shift operation, see equation (2.5). But the multiplication operations are very complicated and time consuming (see [14] Section A.3.8 and Section A.6.4). In that case, a special class of normal bases called Gaussian normal bases are studied in order to minimize the complexity of multiplication.

$$
\begin{aligned}
A^{2^2} = (A^2)^2 &= (a_{m-1}\theta^{2^{(m \mod m)}} + a_{m-2}\theta^{2^{m-1}} + \cdots + a_1\theta^{2^2} + a_0\theta^{2^1})^2 \\
&= a_{m-3}\theta^{2^{m-1}} + a_{m-4}\theta^{2^{m-2}} + \cdots + a_0\theta^{2^2} + a_{m-1}\theta^2 + a_{m-2}\theta
\end{aligned}
\tag{2.5}
$$

2.2 Montgomery Multiplication over $GF(2^m)$

Montgomery multiplication was first proposed by Montgomery in 1985 [10] and was extended to binary field by Koc in 1998 [12]. Compared with the standard multiplication, the Montgomery multiplication can avoid trail division operations whereas standard modular multiplication cannot.

Montgomery multiplication in $GF(2^m)$ is defined by equation (2.6).

$$C(x) = A(x) \times B(x) \times R(x)^{-1} \mod f(x) \tag{2.6}$$

Table 2.1: Algorithm of Binary Field Bit-Parallel Montgomery Multiplication

Algorithm	Binary Field Bit-Parallel Montgomery Multiplication
Inputs:	$A(x), B(x) \in GF(2^m),\ f(x), f'(x)$
Outputs:	$C(x) = A(x) \times B(x) \times R^{-1}(x) \mod f(x)$
Step 1:	$T(x) = A(x)B(x)$
Step 2:	$U(x) = T(x)f'(x) \mod R(x)$
Step 3:	$C(x) = [T(x) + U(x)f(x)]/R(x)$

Instead of obtaining the product of $A(x)B(x) \mod f(x)$ directly, we multiply an extra polynomial $R(x)$, to compute $A(x)B(x)R(x)^{-1} \mod f(x)$. $f(x)$ is the irreducible polynomial which is used to generate the binary field $GF(2^m)$ and $R(x)$ is treated as a fixed element in $GF(2^m)$. The Montgomery multiplication requires that $R(x)$ and $f(x)$ are relatively prime. Under this condition, we have the property that $R(x) \cdot R(x)^{-1} + f(x)f'(x) = 1$, the two polynomials $R(x)^{-1}$ and $f'(x)$ can be computed using extended Euclidean algorithm [12], Table 2.1 presents an algorithm of bit-parallel binary field Montgomery multiplication. It has been proved that, by letting the value of $R(x)$ equals to x^m, efficient implementations of the Montgomery multiplier can be obtained [12]. For example, as the algorithm shown in Table 2.1, if $R(x) = x^m$, modular $R(x)$ in Step 2 can be accomplished simply by ignoring the terms which degree is larger than m, and in Step 3, the division operation can

be implemented by shifting the polynomial to the right side by m bits. Besides, in [25], the work shows $R(x) = x^{m-1}$ is also a suitable Montgomery factor for efficient implementation of Montgomery multiplication.

2.3 Elliptic Curve Cryptosystem

2.3.1 Elliptic Curves

Elliptic curves (EC) are a set of curves that satisfy equation (2.7), where $b \neq 0$, see Section A.9.1 in [14].

$$y^2 + xy = x^3 + ax^2 + b \tag{2.7}$$

On an elliptic curve, two point operations can be defined [14]: point addition and point doubling. One special point called point at infinity or zero point is also defined, see Fig 2.1.

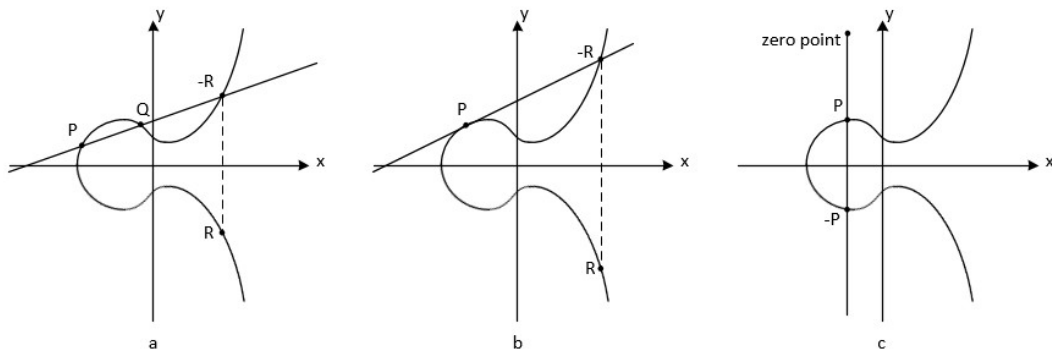

Figure 2.1: Operations in an elliptic curve

In Fig 2.1(a), the two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ do not overlap, line PQ intersects the curve at point $-R$, then we draw a vertical line via $-R$ to get its reflection point R on the curve, and $R = (x_3, y_3)$, thus the point addition operation can be defined as: $R = P + Q$, and x_3 and y_3 can be calculated by the equations presented in (2.8).

$$\begin{aligned} x_3 &= a + \lambda^2 + \lambda + x_1 + x_2 \\ y_3 &= (x_2 + x_3)\lambda + x_3 + y_2 \\ \lambda &= (y_1 + y_2)/(x_1 + x_2) \end{aligned} \tag{2.8}$$

In Fig 2.1(b), points P and Q are overlapped at point P, a tangent line is drawn via $P = (x_1, y_1)$ that intersects the curve at point $-R$, and the reflection of $-R$ is $R = (x_3, y_3)$. In this case, the point doubling operation is defined: $R = 2P$. Equations presented in (2.9) presents the coordinate computation of R.

$$\begin{aligned} x_3 &= a + \lambda^2 + \lambda \\ y_3 &= \lambda x_3 + x_3 + x_1^2 \\ \lambda &= x_1 + y_1/x_1 \end{aligned} \tag{2.9}$$

By combining point addition and point doubling operations, point scalar multiplication can be defined, for example: $5P = 4P + P = 2(2P) + P$, this indicates that one point addition and two point doubling operations are required to obtain $5P$.

In the third case of Fig 2.1, the line $P(-P)$ is perpendicular to the x-axis. Mathematically, we assume the line intersects the curve at a third point at infinite, and define this third point as the point at infinity or zero point, denoted as O. According to this definition, we have: $P + (-P) = O$, $P + O = P$, $O = -O$ and $P + Q + R = O$. The set of points on the elliptic curve is an Abelian group, which implies that the point operations satisfy the common algebraic laws: commutativity and associativity.

When we extend the elliptic curve to binary field $GF(2^m)$, then $a, b \in GF(2^m)$, and the equations in (2.8) and (2.9) should modular $f(x)$ at the end of each equation, $f(x)$ is the irreducible polynomial to generate $GF(2^m)$. In Fig 2.2, we see that the binary field elliptic curve presented in the coordinate graph is no longer a "curve" with a set of infinitely points in a real number field, instead, it consists of finite many points, the points being distributed separately on the first quadrant and the non-negative axises of the plane coordinate graph. The number of points involved in the elliptic curve E including O is called the order of E, denoted as $\#E(GF(2^m))$. The order of a single point P on curve E is defined as the

smallest positive integer n such that $nG = O$, every point on the curve as an order, and this order divides the order of the curve $\#E(GF(2^m))$. Commutativity and associativity are still satisfied for point operations in binary fields.

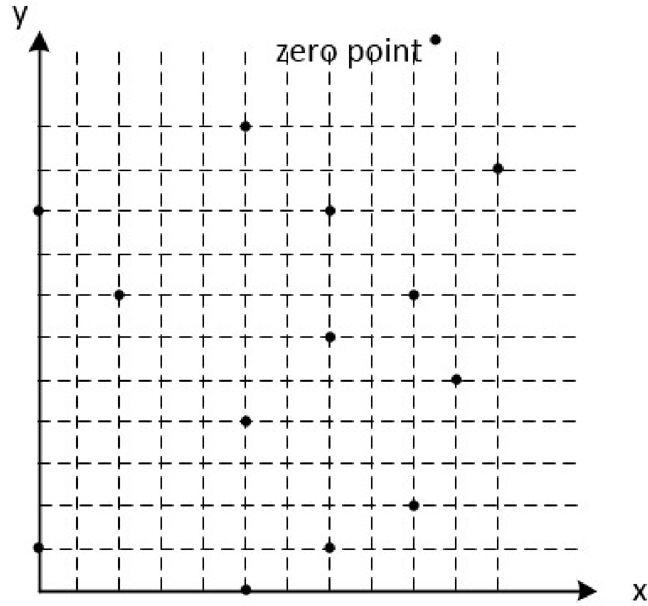

Figure 2.2: Elliptic curve over binary field $GF(2^m)$

2.3.2 Finite Field Inversion Using Normal Basis

Finite field inversion operation is one of the basic operations of ECC computation. Assume α belongs to the finite field $GF(2^m)$ and α is represented using normal basis:

$$\alpha = a_{m-1}\theta^{2^{m-1}} + a_{m-2}\theta^{2^{m-2}} + \cdots + a_1\theta^2 + a_0\theta \tag{2.10}$$

Since for $\forall\alpha \in GF(2^m)$ there exists an order, denoted as $ord(\alpha)$, and according to the definition of the order, we have:

$$\alpha^{ord(\alpha)} = 1 \tag{2.11}$$

Also, $ord(\alpha)$ divides $2^m - 1$. If we assume that $n \times ord(\alpha) = 2^m - 1$, by taking the

power of n from both sides of equation (2.11), we could have:

$$(\alpha^{ord(\alpha)})^n = \alpha^{2^m-1} = 1^n = 1 \tag{2.12}$$

By dividing α with both sides of equation (2.12), we could get the expression of inverse α:

$$\alpha^{-1} = \alpha^{2^m-2} \tag{2.13}$$

Since 2^x-power only needs a left shift, see equation (2.5) as a reference, we could take the advantages of this property and obtain an efficient algorithm to compute finite field inversion using normal basis representation. In Chapter 5, an efficient computation and implementation of finite field inversion in $GF(2^{163})$ is provided based on equation (2.13), using normal basis.

2.3.3 Elliptic Curve Cryptosystem

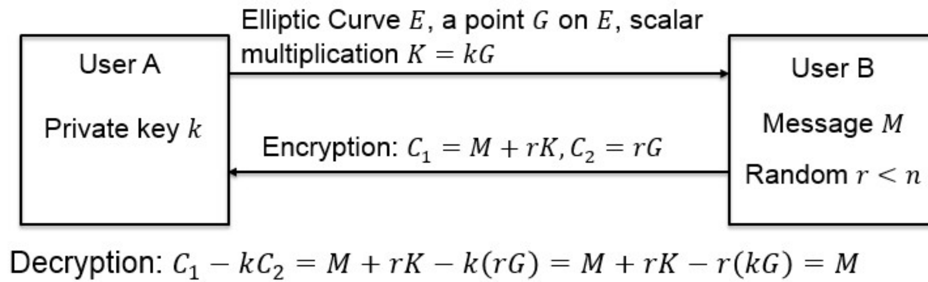

Figure 2.3: Encryption/decryption of elliptic curve cryptosystem

Elliptic curve cryptosystem (ECC) is a public-key cryptosystem that has a shorter key size compared with RSA in same secure level. Suppose a base point G on elliptic curve E has order n, then we could define the key pair as follows: the private key k is a positive integer smaller than n; the corresponding public key K is a point on the curve E, where $K = kG$ and K is computed by point scalar multiplication. The encryption/decryption operations can be described as follows:

(1) Alice (known as User A) selects an elliptic curve E, a base point G on E, and she determines the key pair of private key k and public key K. She then sends curve E, base point

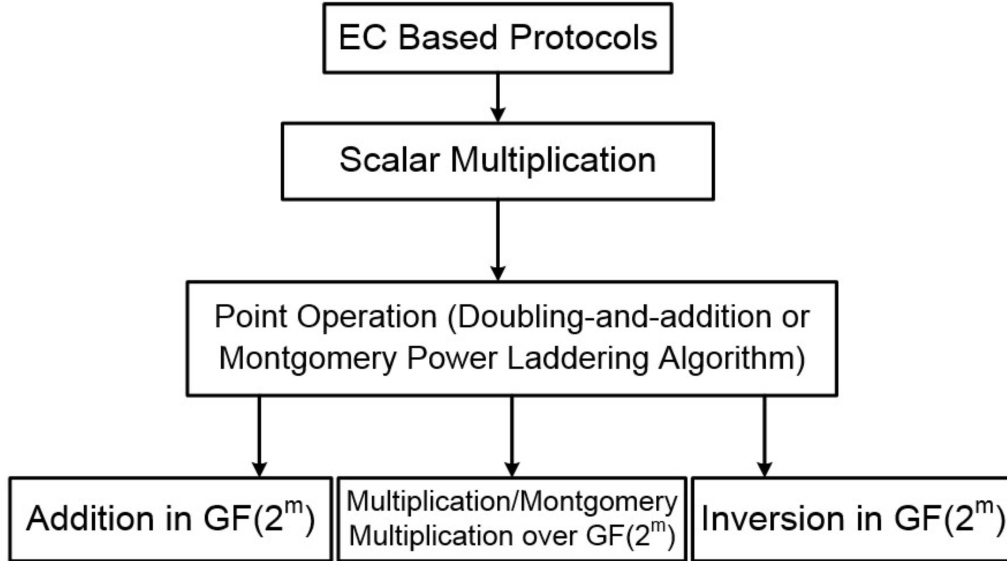

Figure 2.4: Computation structure of ECC over $GF(2^m)$

G together with the public key K to Bob (known as User B) for private communication;

(2) If Bob has a message (known as the plaintext) and he wants to send it to Alice privately. First, he maps or encodes the text to a point on E, denotes this point as M, different mapping methods can be found from [11], [23], [26] and [31]. Second, Bob chooses a random number $r < n$ and encrypts M with public key K and base point G, see equation (2.14). The two computed points (C_1, C_2) are knowns as the corresponding ciphertext of M. Third, Bob now sends the ciphertext (C_1, C_2) to Alice and this process can be described as encryption;

(3) Alice receives the ciphertext (C_1, C_2) sent from Bob and decrypt them with the private key k, see equation (2.15), and finally she can read the secret message Bob sends. This process can be described as decryption.

Fig 2.3 shows the process of ECC encryption and decryption.

$$C_1 = M + rK, \ \ C_2 = rG \tag{2.14}$$

$$C_1 - KC_2 = M + rK - k(rG) = M + rK - r(kG) = M \tag{2.15}$$

From the above brief introduction of the ECC encryption/decryption operations, we could see that the key issue to break this cryptosystem is to resolving the value of the private key k from the equation $K = kG$. The fact is that knowing k and G to compute K is simple by calculating a set of point addition and point doubling operations, however, knowing K and G to compute k is extremely hard when the size of the selected binary field is large. For this reason, this type of cryptosystem relies for its security level on the difficulty level of the elliptic curve discrete logarithm problem.

The computation of ECC contains four levels, see Fig 2.4. The top level is the ECC itself. The major operation involved in ECC is the point scalar multiplication, and it is the second level of ECC computation. A point scalar multiplication can be efficiently calculated by a set of point doubling and point addition operations, for example $9P = 2(2(2P)) + P$ can be decomposed into four point doubling operations plus one point addition operation. The computation of a point scalar multiplication is similar to the squaring-multiplying algorithm when calculating an exponentiation operation. Thus, the point operations are the third level of ECC computation. The coordinate computation of the point operations, see equations (2.8) and (2.9), indicates that the basic operations of ECC computation are finite field multiplication and finite field inversion.

Chapter 3

A Review of Existing Work

The existing Montgomery multipliers can be grouped into two types in terms of their architectures: general style including three sub-types: bit-serial [12], [22], [25], [28], bit-parallel [12], [15], [25] and digit-level [12], [28]; and systolic style [18], [20], [16], [27]. Bit-serial multipliers load one operand bit-by-bit and the other operand in parallel. They usually have the lowest gate complexity but require the longest time to process one operation; in contrast, bit-parallel multipliers could reach the fastest processing speed by loading and calculating both operands in parallel, but cost the most gate count when implementing. Digit-level multipliers allow us to combine the advantages of both and seek the balance between area and speed by processing one operand by constant bits each clock cycle and the other operand in parallel. Systolic style multipliers consist of matrix-like rows of data processing units (cells) known as a systolic array. These units are similar to central processing units, each unit shares the information with its neighbors. Systolic style architectures are well suited to VLSI design due to the scalability, short inter-connection and highly repetitive nature of the units. Our work will mainly focus on the architecture of digit-serial Montgomery multipliers, and we will review some digit-serial polynomial multipliers and digit-serial Montgomery multipliers first.

Montgomery multiplication was first applied to binary field multiplication in 1998 by Koc [12], who reported the general algorithms of bit-serial (Table 3.1), bit-parallel (Table 2.1) and digit-serial (Table 3.2) Montgomery multiplications. Koc [12] first showed that by selecting the Montgomery factor $R(x) = x^m$, the multiplication can be efficiently implemented in both bit-serial and digit-serial architectures. Besides, he proved that us-

ing digit-level Montgomery method for finite field multiplication could offer a much faster processing speed compared with the standard digit-level multiplication.

Table 3.1: Algorithm of Bit-Serial Montgomery Multiplication

Algorithm	Bit-Serial Montgomery Multiplication
Inputs:	$A(x), B(x) \in GF(2^m),\ f(x)$
Outputs:	$C(x) = A(x) \times B(x) \times x^{-m} \mod f(x)$
Step 1:	$C(x) = 0$
	for $i = 0$ to $m-1$ do
Step 2:	$C(x) = C(x) + a_i B(x)$
Step 3:	$C(x) = C(x) + c_0 f(x)$
Step 4:	$C(x) = C(x)/x$

Table 3.2: Algorithm of Digit-Serial Montgomery Multiplication, where d is the digit size, $f_0'(x) f_0(x) = 1 \mod x^d$, $C_0(x)$ and $f_0(x)$ are the least significant digits of $C(x)$ and $f(x)$, respectively

Algorithm	Digit-Serial Montgomery Multiplication
Inputs:	$A(x), B(x) \in GF(2^m),\ f(x), f_0'(x)$
Outputs:	$C(x) = A(x) \times B(x) \times x^{-m} \mod f(x)$
Step 1:	$C(x) = 0$
	for $i = 0$ to $s-1$ do
Step 2:	$C(x) = C(x) + A_i(x) B(x)$
Step 2:	$M(x) = C_0(x) f_0'(x) \mod x^d$
Step 3:	$C(x) = C(x) + M(x) f(x)$
Step 4:	$C(x) = C(x)/x^d$

In 1998, Song proposed two different polynomial multiplier architectures: least significant digit (LSD) first and most significant digit (MSD) first, respectively. In 2005, Tang reported a bit-parallel digit-serial multiplier in $GF(2^{233})$, the architecture of the proposed $GF(2^{233})$ multiplier is shown in Fig 3.1. Tang's architecture contains three main modules: a multiplier module to generate the partial product $A_j \times B$, a register to store the value of C_{30-j-1}, and a constant multiplier to calculate the product of $x^8 \times C_{30-j-1}$. The register module can be implemented by a D-flipflop array, and since Tang used an irreducible trinomial to generate $GF(2^{233})$, the constant multiplier can also be easily implemented. The most complicated module would be the partial product multiplier which computes $A_j \times B$. Fig 3.2(a) shows Tang's design of this module. In Fig 3.2(a), we could see that Tang's

structure of partial product multiplier includes an AND gate section to logic AND each bit of the digit A_j with operand B, a left-shift modular section to calculate the multiplied by x^i moduli operation, and finally an XOR tree section to add up all eight rows together. Tang's proposed digit-serial architecture can be treated as a landmark work since subsequent works on digit-serial finite field multipliers are more or less optimizations or modifications of his work.

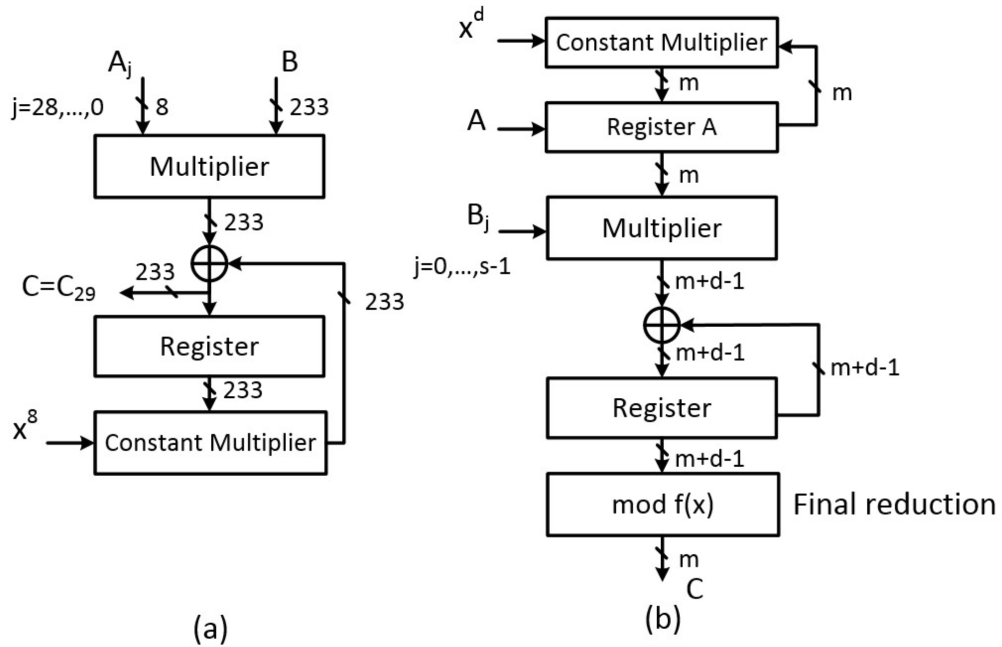

Figure 3.1: (a)Tang's architecture of $GF(2^{233})$ multiplier [17] (b)Kumar's architecture of $GF(2^m)$ multiplier [19]

In 2006, Kumar proposed another polynomial multiplier in $GF(2^m)$ [19]. There are two major differences between Kumar's work and Tang's: one is that in the partial product generator unit, after logic AND each bit of the digit A_j and left shift the bit-string by corresponding i bits ($0 \leq i \leq d-1$), Kumar directly added up all rows together with no reduction operations, thus, the data-flow during processing has $m+d-1$-bit bandwidth. As a consequence, Kumar added an extra module called the final reduction unit to process modular $f(x)$ operation when the whole computation operation is over. The other difference is that Tang begins the multiplication from the most-significant-digit while Kumar begins from the least-significant-digit. In that case, Kumar saves the cost of modular reduction

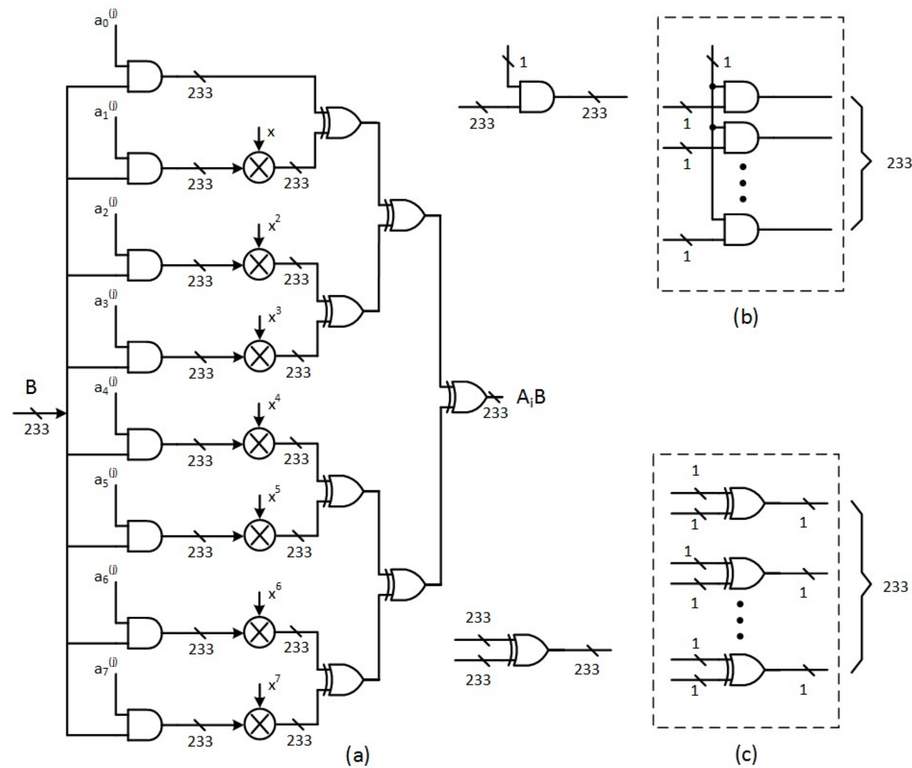

Figure 3.2: Tang's architecture of partial product multiplier, generates the product of $A_j \times B$ [17]

operation for all d rows in the partial product module, but as a trade off, one extra clock cycle would be needed to complete the multiplication, another register for storing the value of $Ax^d \mod f(x)$ is required, and in addition, the bandwidth of the data-flow was enlarged by d bits, see Fig 3.1(b).

In 2009, Meher [24] proposed a polynomial multiplier with a new structure of finite field accumulator unit, which is the major difference between his work and the former works reviewed. The block diagram of Meher's work is presented in Fig 3.3. In the finite field accumulator block, he used a T-flipflop array to implement the accumulate operation instead of the structure using XOR gates and D-flipflop array. Besides, Meher also combined the constant multiplier and partial product multiplier to generate $Ax^d \mod f(x)$ and $A \times B_j \mod f(x)$ in parallel in order to further reduce the number of blocks. However, this modification has not resulted in the reduction of gate count or critical path delay. Also in

the same year, Hariri [25] published his work proving that, besides $R(x) = x^m$, $R(x) = x^{m-1}$ could also be an efficient Montgomery factor in bit-serial structure, and later it was proved by [28] that it can be applied to the digit-serial structure of Montgomery multiplications.

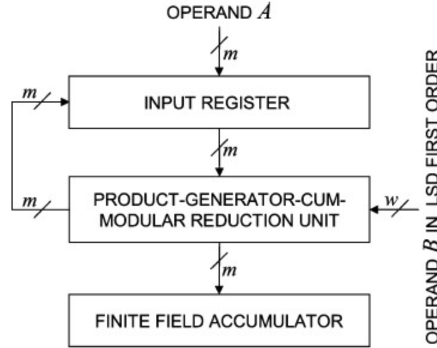

Figure 3.3: Meher's block diagram of proposed field multiplier over $GF(2^m)$ [24]

The most recent digit-serial Montgomery multiplication architecture was that proposed by [28] in 2011. A Linear Feedback Shift Register (LFSR) was used as the main building block to implement the Montgomery multiplication. In this work, the cases when $R(x) = x^m$ and $R(x) = x^{m-1}$ are discussed. As reported, the proposed multipliers could adapt to different classes of irreducible polynomials such as general cases, all one polynomials, trinomials and pentanomials, by changing the value of digit size d, the reported multipliers could also work as bit-serial multipliers or bit-parallel multipliers. The high flexibility of their work is the critical contribution to the study of field multiplication. See Fig 3.4 of [28]'s work.

In this thesis, a constraint condition is proposed to select the irreducible pentanomials for the generation of finite field $GF(2^m)$. A most-significant-digit first and a least-significant-digit first digit-serial Montgomery multiplications are also proposed. The architectures proposed in this work have some similarities with the works reported in [17] and [24]. However, these two architectures have two major differences compared with the proposed works in this thesis. First, the algorithms in [17] and [24] are about polynomial multiplication: they consider the product of $A(x) \times B(x) \mod f(x)$ rather than the product of $A(x) \times B(x) \times x^{-m} \mod f(x)$ (or $A(x) \times B(x) \times x^{-(m-1)} \mod f(x)$). Second, we proposed novel fixed elements $R(x)$ which are different from x^m and x^{m-1}. By applying the proposed constraint condition, the critical path delay of the architectures are found to

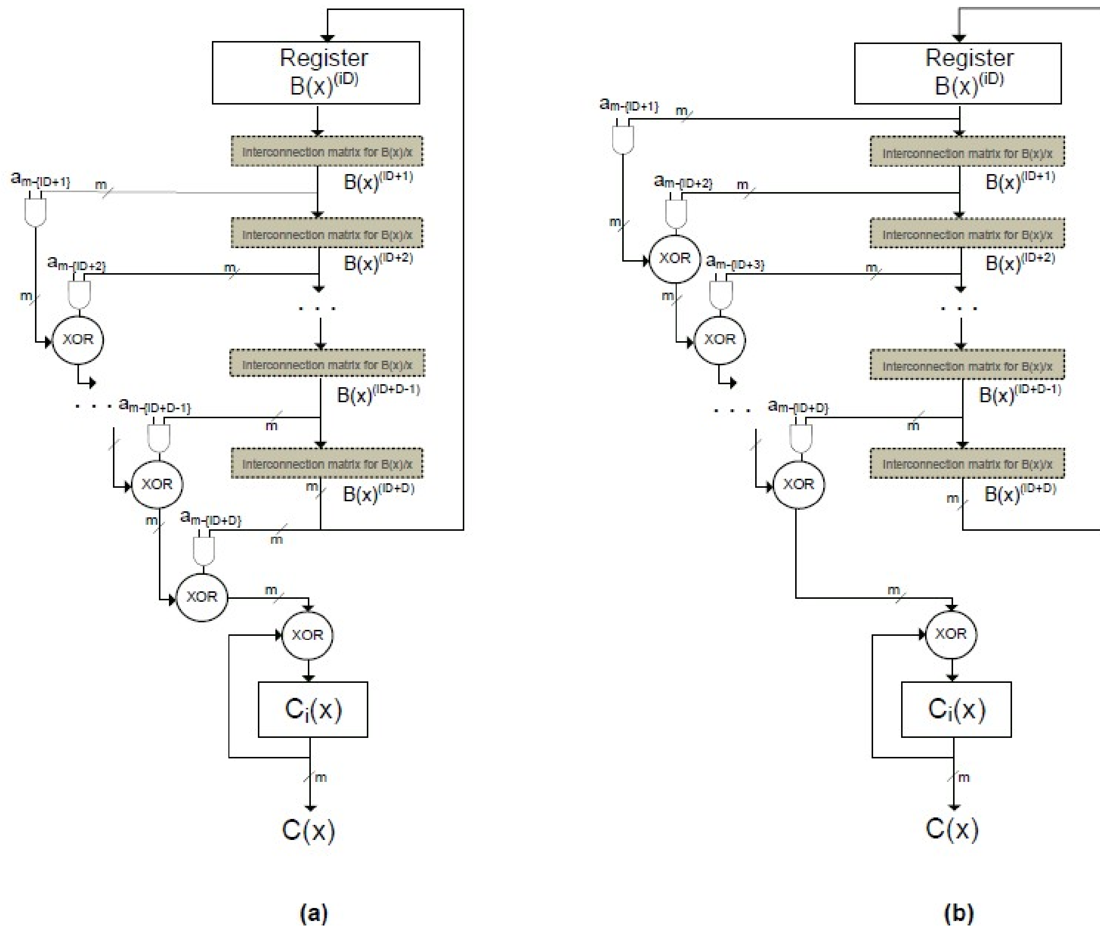

Figure 3.4: Work reported in [28], (a)$R(x) = x^m$, (b)$R(x) = x^{m-1}$

be as good as or better than the existing works for the fields generated by the irreducible pentanomials.

Chapter 4

Proposed Digit-serial Montgomery Multipliers

In this chapter, the detailed algorithm and architecture of the proposed digit-serial most-significant-digit first and least-significant-digit first Montgomery multiplier will be introduced. The finite field is generated by irreducible pentanomial polynomials. The parameter selection of the irreducible pentanomials is discussed, and a general condition to further reduce the time delay of the multiplier is proposed. The gate count and time delay of the multiplier will be considered and analyzed when $R(x) = x^u$, where the value of u is different from m or $m-1$. Further discussions are is included. After this, comparisons with other types of digit-serial multipliers are provided. Finally, the FPGA implementation of the proposed digit-serial Montgomery multipliers will be given, as well as its simulation and compilation results.

4.1 Proposed Digit-Serial MSD First Montgomery Multiplier

In this section, a digit-serial MSD first Montgomery multiplier will be proposed. Two different architectures of the proposed multiplier are presented. In the first architecture, the multiplication and reduction operations are processed in separate units; in the latter architecture, the multiplication and reduction operations are combined and implemented in

one circuit block, and the performance is proved to be more efficient than architecture 1.

4.1.1 Algorithm

Consider the field elements A, B, R, and their product C over $GF(2^m)$. Using the polynomial representation, we have:

$$A(x) = \sum_{i=0}^{m-1} a_i x^i, \quad B(x) = \sum_{i=0}^{m-1} b_i x^i, \quad R(x) = x^l, \quad C(x) = \sum_{i=0}^{m-1} c_i x^i. \tag{4.1}$$

Here we use irreducible pentanomial $f(x)$ to generate $GF(2^m)$, and $f(x)$ is represented as:

$$f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1 \tag{4.2}$$

Since the idea in a digit-level multiplier is to compute a set of constant d bits from $B(x)$, where d usually equals to a power of two (2, 4, 8, etc.) in practice, neither one bit at each clock cycle, nor all bits in parallel at the same time, we divide $B(x)$ into blocks with equal length d, such that $B(x)$ has s blocks, $s = \lceil m/d \rceil$. Thus the digit-level polynomial representation of $B(x)$ can be written as:

$$B(x) = \sum_{i=0}^{s-1} B_i(x) x^{id} = \sum_{i=0}^{s-1} \sum_{j=0}^{d-1} b_{id+j} x^{id+j} \tag{4.3}$$

Note that, due to the fact that m may not always divisible by d, terms that generated by equation (4.3) with degree larger than $m-1$ or smaller than 0 should be set to 0. For example when $m = 233$, $d = 8$ and $s = 30$, when $i = 29$, $B_{29}(x)x^{232} = (b_{232} + b_{233}x^1 + \cdots + b_{239}x^7)x^{232} = b_{232}x^{232}$.

Using the digit-level representation of $B(x)$, we can write $C(x)$ as:

$$C(x) = A(x) \times \sum_{i=0}^{s-1} B_i(x) x^{id} \times R^{-1}(x) \mod f(x) \tag{4.4}$$

Defining integer l as always satisfying $0 \leqslant l \leqslant d-1$, where d is the digit size, and $R(x) = x^l$,

then we could use equation (4.5) to compute $C(x)$:

$$\begin{aligned} C(x) =&((A(x)B_{s-1}(x)x^{-l}x^{d} + A(x)B_{s-2}(x)x^{-l})x^{d} + \cdots + A(x)B_{1}(x)x^{-l})x^{d} \\ &+ A(x)B_{0}(x)x^{-l} \end{aligned} \tag{4.5}$$

Thus an algorithm of MSD-first Montgomery multiplication can be presented by Table 4.1.

Table 4.1: Digit-serial MSD-first Montgomery Multiplier ($R(x) = x^{l}$), where $0 \leqslant l \leqslant d-1$

Algorithm I	Digit-serial MSD-first Montgomery Multiplier
Inputs:	$A(x), B_0(x), B_1(x), \ldots, B_{s-1}(x), f(x)$
Outputs:	$C(x) = A(x)B(x)x^{-l} \mod f(x),\ 0 \leqslant l \leqslant d-1$
Step 1:	$C^{(0)}(x) = 0$
	For $i = 0$ to $s-1$
Step 2:	$T(x) = A(x)B_{s-1-i}(x)x^{-l} \mod f(x)$
Step 3:	$C^{(i+1)}(x) = C^{(i)}(x)x^{d} + T(x) \mod f(x)$
Step 4:	$C(x) = C^{(s)}(x)$

Step 1 is the initialization step, register C is set to zero, $C(x)^{(0)} = 0$. In Step 2, the product of $A(x)$, B_{s-i-1} and x^{-l} is computed, and the reduction operation is also processed in the same step. Then, in Step 3, the value generated in Step 2 is added with $C^{(i)}x^{d}$ mod $f(x)$ and the result is stored back to the register. When $i = s-1$, $C^{(s)}$ will be obtained, the multiplier will provide the final result. Step 2 and 3 are processed in the same cycle, also note that the calculation of Step 2 and $C^{(i)}(x)x^{d}$ in Step 3 can be done in parallel.

4.1.2 General Architecture

Fig 4.1 presents the block diagram of the proposed multiplier. The Multiplier Core unit implements Step 2, also an XOR array is included in Multiply Core to implement the addition operation in Step 3. Modular-Shift unit corresponds to $C^{(i)}x^{d} \mod f(x)$ in Step 3. The final result is provided in the register unit, REG C. By computing Step 2 in different orders, different architectures of the proposed multiplier can be obtained. This subsection will present a general architecture.

In Step 2, if the product of $A(x)$ and $B_{s-i-1}(x)$ is computed first, then the result times x^{-l} followed by the reduction operation $\mod f(x)$. The degree range of $A(x)B_{s-i-1}(x)x^{-l}$ would be from $-l$ to $m+d-l-2$, and the reduction operation needs to reduct the product

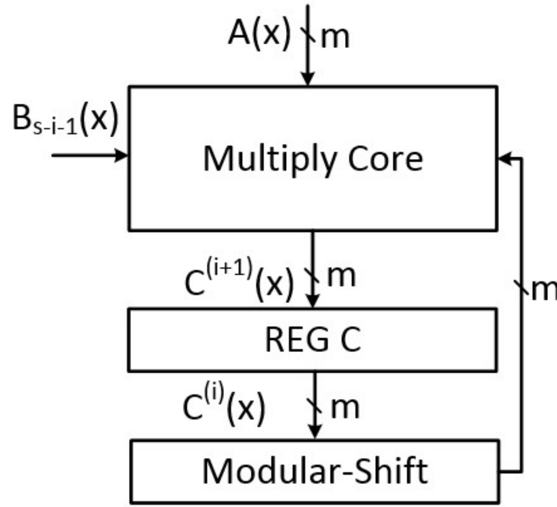

Figure 4.1: Block diagram of proposed digit-serial MSD-first Montgomery multiplier when $R(x) = x^l$

from $[-l, m+d-l-2]$ to $[0, m-1]$. It is clearly a two-side reduction operation: both side of polynomial $A(x)B_{s-i-1}(x)x^{-l}$ are beyond the bandwidth of $GF(2^m)$. To further analyze the computation of Step 2, we let $A(x)B_{s-i-1}(x)x^{-l} = T_H(x) + T_M(x) + T_L(x)$, the degree range of $T_H(x), T_M(x), T_L(x)$ are $[m, m+d-2-l], [0, m-1], [-l, -1]$, respectively. The reduction operation can be calculated as following equations:

$$
\begin{aligned}
\textit{Terms in } T_H(x): \quad x^{m+d-2-l} \mod f(x) =& x^{k_3+d-2-l} + x^{k_2+d-2-l} + x^{k_1+d-2-l} \\
&+ x^{d-2-l} \\
&\vdots \\
x^{m+1} \mod f(x) =& x^{k_3+1} + x^{k_2+1} + x^{k_1+1} + x \\
x^{m} \mod f(x) =& x^{k_3} + x^{k_2} + x^{k_1} + 1 \\
\textit{Terms in } T_L(x): \quad x^{-1} \mod f(x) =& x^{m-1} + x^{k_3-1} + x^{k_2-1} + x^{k_1-1} \\
x^{-2} \mod f(x) =& x^{m-2} + x^{k_3-2} + x^{k_2-2} + x^{k_1-2} \\
&\vdots \\
x^{-l} \mod f(x) =& x^{m-l} + x^{k_3-l} + x^{k_2-l} + x^{k_1-l}
\end{aligned}
\tag{4.6}
$$

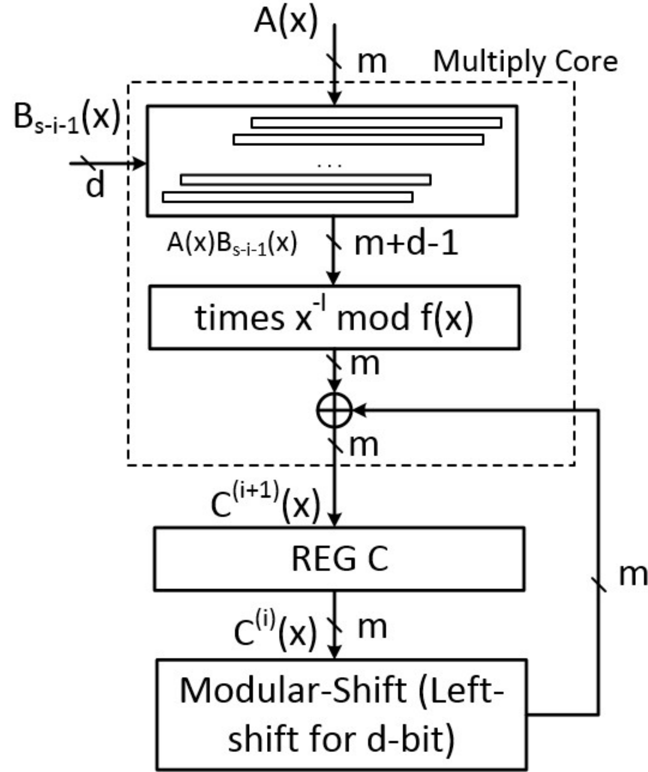

Figure 4.2: General architecture of the proposed multiplier when $R(x) = x^u$

According to equation (4.6), from $T_H(x)$ reduction, four extra bit-strings are generated, the degree range of these are: $[k_3, k_3+d-2-l]$, $[k_2, k_2+d-2-l]$, $[k_1, k_1+d-2-l]$, $[0, d-2-l]$. Similarly, another four bit-strings are generated by $T_L(x)$ reduction operation: $[m-l, m-1]$, $[k_3-l, k_3-1]$, $[k_2-l, k_2-1]$, $[k_1-l, k_1-1]$. Especially, bit-string $[k_3, k_3+d-2-l]$ and $[k_3-l, k_3-1]$ can be combined into one bit-string with range $[k_3-l, k_3+d-2-l]$, in this way, all eight bit-strings can be transformed into five bit-string with degree range equal to: $[m-l, m-1]$, $[k_3-l, k_3+d-2-l]$, $[k_2-l, k_2+d-2-l]$, $[k_1-l, k_1+d-2-l]$, $[0, d-2-l]$, respectively. In order to avoid further reduction operation, the equations 4.6) must satisfy such conditions:

$$\begin{aligned} k_3+d-2-l &\leqslant m-1 \\ k_1-l &\geqslant 0 \end{aligned} \tag{4.7}$$

After simplifying equation (4.7):

$$\begin{aligned} k_3 &\leqslant m+1+l-d \\ k_1 &\geqslant l \end{aligned} \tag{4.8}$$

From equation (4.6), we notice that we need to add up five bit-strings to $T_M(x)$, the gate usage is a constant number which is equal to $4(d-1)$. This fact indicates that the computation of $A(x)B_{s-1-i}(x)x^{-l} \mod f(x)$ will not generate extra gate delay when compared with the computation of $A(x)B_{s-1-i}(x) \mod f(x)$. However, the time delay varies with different value of k_1, k_2, k_3. To have the minimum time delay T_X, the five bit-strings should have no overlapped parts, so the following conditions must be satisfied:

$$\begin{aligned} m-l &> k_3+d-2-l \\ k_3-l &> k_2+d-2-l \\ k_2-l &> k_1+d-2-l \\ k_1-l &> d-2-l \end{aligned} \tag{4.9}$$

To sum up, k_i $(i=0,1,2,3)$ must satisfy:

$$k_{i+1}-k_i \geqslant d-1 \tag{4.10}$$

Where $k_0=0$ and $k_4=m$, this condition is denoted as Constraint Condition 1. Besides, from equation (4.10), $k_1 \geqslant d-1 \geqslant l$, and $m-k_3 \geqslant d-1 > d-1-l$, this fact implies that if equation (4.10) is applied when selecting the irreducible pentanomials of $GF(2^m)$, equation (4.8) will also be satisfied. The general architecture of the proposed multiplier is presented in Fig 4.2.

In Multiply Core unit, the implementation of $A(x)B_{s-i-1}(x)$ is simple: operand $A(x)$ is multiplied by each bit of $B_{s-1-i}(x)$, and add up the terms with same degree. This block costs totally md AND gates for the multiplication operation, and $(m-1)(d-1)$ XOR gates for the field addition operations. The critical path delay of this unit is $log_2 dT_X+T_A$. The reduction operation costs $4(d-1)$ XOR gates and if Constraint Condition 1 is applied, the time delay is T_X. Also, the XOR array needs m XOR gates and time delay is T_X. Where T_X and T_A donate a two-input XOR gate and a two-input AND gate respectively. Thus, the

Multiply Core unit costs md AND gates, $(md+3d-3)$ XOR gates, and critical path delay is $T_A+(2+log_2 d)T_X$

REG C unit updates the value of $C^{(i)}(x)$ every clock cycle. This unit is implemented by a D-flipflop array, with m D-flipflops connected in parallel.

Modular-Shift unit computes the modular multiplication $C^{(i)}(x)x^d \mod f(x)$. If $C_d^{(i)}(x)$ represents the most significant d bits of $C^{(i)}$, equation (4.11) can be used to present the computation of $C^{(i)}(x)x^d \mod f(x)$.

$$C^{(i)}(x)x^d \mod f(x) = C_d^{(i)}(x)(x^{k_3}+x^{k_2}+x^{k_1}+1)+\sum_{i=d}^{m-1} c_{i-d}^{(i)}x^i \tag{4.11}$$

To add up the five bit-strings together, in total $3d$ XOR gates will be needed, see Fig 4.3 for the implementation of equation (4.11) operation. By applying the condition obtained by equation (4.10), when $k_{i+1}-k_i \geqslant d$, the four bit-strings, $C_d^{(i)}(x)x^{k_3}$, $C_d^{(i)}(x)x^{k_2}$, $C_d^{(i)}(x)x^{k_1}$, and $C_d^{(i)}(x)$ will share no terms with same degree, thus the time delay of this circuit would be T_X. For example, we let $k_2 = k_3 - d$, so the degree range of $C_d^{(i)}(x)x^{k_3}$ is $[k_3, k_3+d-1]$ while the degree range of $C_d^{(i)}(x)x^{k_2}$ is $[k_2, k_2+d-1]$ which equals $[k_2, k_3-1]$. When $k_{i+1}-k_i = d-1$, the degree range of $C_d^{(i)}(x)x^{k_3}$, $C_d^{(i)}(x)x^{k_2}$, $C_d^{(i)}(x)x^{k_1}$, and $C_d^{(i)}(x)$ are $[k_3, m]$, $[k_2, k_3]$, $[k_1, k_2]$ and $[0, k_1]$ respectively, it can be seen that each two of them having one term with the same degree, thus the maximum depth of XORing these four bit strings is 2. In addition, since the range of $C_d^{(i)}(x)x^{k_3}$ is $[k_3, m]$, the term with degree m needs another reduction operation and this will generate three more bits for XORing. As a consequence, the maximum depth of the XOR tree involved in this unit is 4, the time delay of this unit is maximumly $log_2 4T_X = 2T_X$, gate count is $\leqslant 3d+3$.

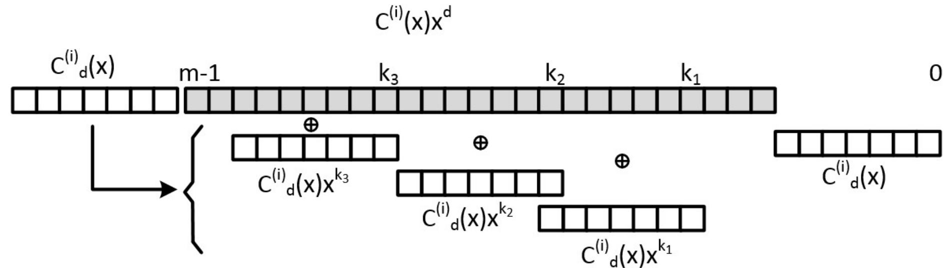

Figure 4.3: Implementation of equation (4.11)

The critical path of this architecture is: $Multiply\ Core1 \rightarrow REG\ C$. The complexity of each block is presented in Table 4.2, and the complexity of the proposed multiplier is presented in Table 4.3. In the tables, T_{DFF} represents the delay of a D-flipflop.

Table 4.2: Complexity of each block of the proposed MSD-first Montgomery multiplier

Block	Gate Count	Time Delay
Multiply Core	md AND, $md+3d-3$ XOR	$T_A+(2+log_2 d)T_X$
REG C	m D-flipflop	T_{DFF}
Modular-Shift	$\leqslant 3d+3$ XOR	$\leqslant 2T_X$

Table 4.3: Complexity of proposed digit-serial MSD-first Montgomery multiplication (Algorithm I, general architecture, when $k_{i+1}-k_i \geqslant d-1, k_0=0, k_4=m$ and $0 \leqslant l \leqslant d-1$)

Work	#AND	#XOR	#FF/Reg	#CLK	Critical path delay
MSD(Arch.1)	md	$\leqslant md+6d$	m	s	$T_A+(2+log_2 d)T_X+T_{DFF}$

4.1.3 Advanced Architecture

In this subsection, another architecture of the proposed multiplier one is introduced, the structure of Multiply Core unit is different from the previous one.

If $A(x)B_{s-i-1}(x)x^{-l} \mod f(x)$ is computed in a different order, see equation 4.12. This indicates that $A(x)x^{-l} \mod f(x)$, $A(x)x^{-l+1} \mod f(x)$, $\cdots$, $A(x)x^{d-l-1} \mod f(x)$ are computed first, then multiply each term with the corresponding bit of $B_{s-1-i}(x)$.

$$
\begin{aligned}
&A(x)B_{s-i-1}(x)x^{-l} \mod f(x)\\
=&A(x)\sum_{j=0}^{d-1} b_{(s-i-1)d+j}x^j x^{-l} \mod f(x)\\
=&\sum_{j=0}^{d-1}(A(x)x^{j-l} \mod f(x))b_{(s-i-1)d+j}
\end{aligned}
\tag{4.12}
$$

The reduction operation is provided in equation 4.6 Based on that equation, $A(x)x$

mod $f(x)$ can be computed in this way:

$$A(x)x \mod f(x) = a_{m-1}(x^{k_3} + x^{k_2} + x^{k_1} + 1) + \sum_{i=1}^{m-1} a_{i-1}x^i \tag{4.13}$$

Fig 4.4 is a circuit diagram which implements equation (4.13): when input $A(x)$, it will output $A(x)x \mod f(x)$. If we connect two of such models in serial, the final output would

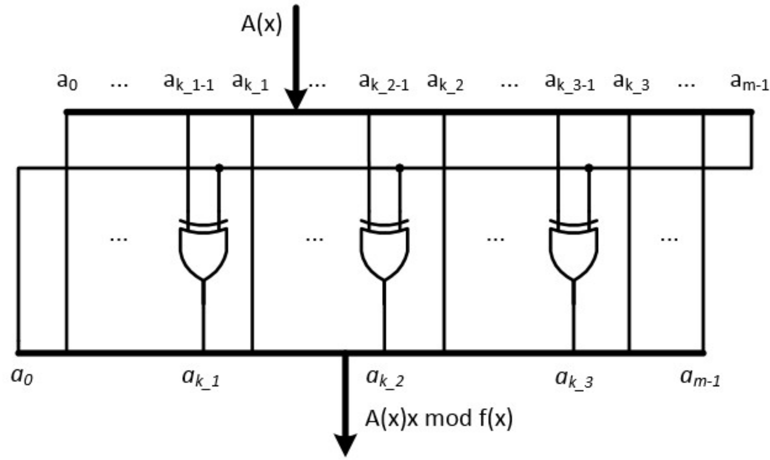

Figure 4.4: Model 1: multiply by x structure

be $A(x)x^2 \mod f(x)$, see Fig 4.5. In the same way, we could obtain each value of $A(x)x^{j-l} \mod f(x)$, when $j = l+1, l+2, \ldots, d-1$.

Similarly, the computation of $A(x)/x \mod f(x)$ is shown in equation (4.14)

$$A(x)/x \mod f(x) = a_0(x^m + x^{k_3} + x^{k_2} + x^{k_1}) \sum_{i=0}^{m-2} a_{i+1}x^i \tag{4.14}$$

The implementation of equation (4.14) is shown in Fig 4.6. As a consequence, by combining multiples of the same circuit unit shown in Fig 4.6, each value of $A(x)x^{j-l} \mod f(x)$ can be obtained, where $j = 0, 1, 2, \ldots, l-1$.

If we apply both Model 1 shown in Fig. 4.4 and Model 2 shown in Fig 4.6 to the implementation of equation (4.12), then the reduction operation is divided into two separate branches: one for the reduction of degrees larger than $m-1$; another for the reduction of degrees smaller than 0. The architecture is presented in Fig 4.7. Note that blocks marked

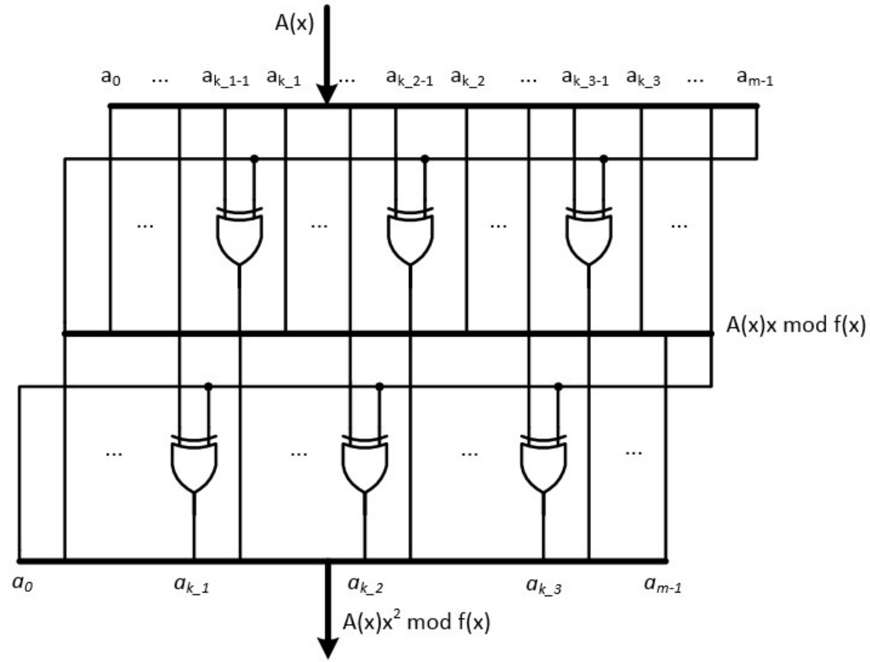

Figure 4.5: Implementation of computation $A(x)x^2 \mod f(x)$

with x represent the circuit structure of Model 1, and blocks marked with x^{-1} represent Model 2. The architecture of the proposed multiplier is given in Fig 4.8. The depth of the XOR tree is $d+1$, the XOR tree adds up all d products of $A(x)x^{j-l}b_{(s-1-i)d+j} \mod f(x)$ plus the value of REG C.

In order to have the least critical path delay of the Multiply Core unit, two groups of conditions must be satisfied at the same time:

Condition 1:

$$\begin{aligned} m-l &> k_3-1 \\ k_3-l &> k_2-1 \\ k_2-l &> k_1-1 \\ k_1-l &> -1 \end{aligned} \tag{4.15}$$

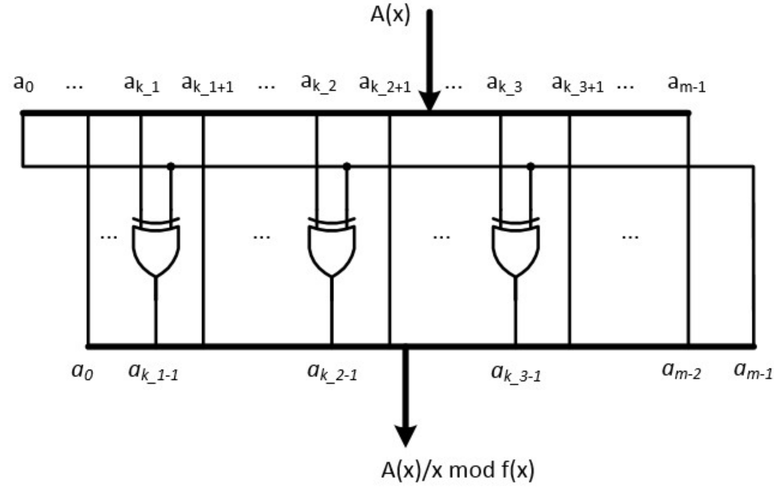

Figure 4.6: Model 2: multiply by x^{-1} structure

Condition 2:

$$\begin{aligned} m &> k_3 + d - 2 - l \\ k_3 &> k_2 + d - 2 - l \\ k_2 &> k_1 + d - 2 - l \\ k_1 &> d - 2 - l \end{aligned} \tag{4.16}$$

To sum up, Condition 1 is $k_{i+1} - k_i > l - 1$, and Condition 2 is $k_{i+1} - k_i > d - l - 2$, where $i = 0, 1, 2, 3$, $k_0 = 0$, $k_4 = m$. Thus, to have the least time delay, k_3, k_2, k_1 must satisfy:

$$k_{i+1} - k_i \geqslant max\{l, d - l - 1\}, \;\; i = 0, 1, 2, 3 \; and \; k_0 = 0, \; k_4 = m \tag{4.17}$$

This condition is denoted as Constraint Condition 2. More specifically, when $l > (d-1)/2$, $k_{i+1} - k_i \geqslant l$; when $l < (d-1)/2$, $k_{i+1} - k_i \geqslant d - l - 1$; when $l = (d-1)/2$, $k_{i+1} - k_i \geqslant d/2 - 1/2$, since d usually an even number, $k_{i+1} - k_i \geqslant d/2$.

The remaining two units are completely the same with the architecture shown in Fig 4.2. REG C is implemented by a D-flipflop array, and Modular Shift unit costs a maximum of $5d$ XOR gates, and $2T_X$ time delay. The complexity of this architecture is shown in Table 4.4.

Comparing the general architecture with the advanced architecture, the gate count of

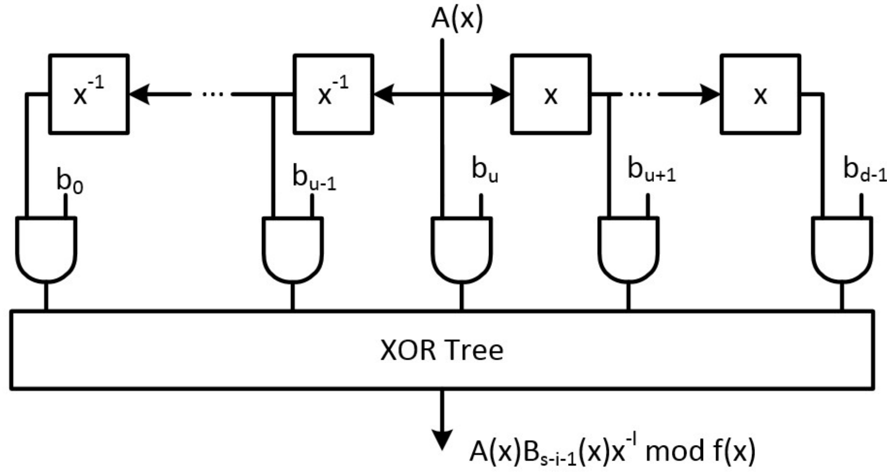

Figure 4.7: Implementation of $A(x)B_{s-i-1}(x)x^{-l} \mod f(x)$

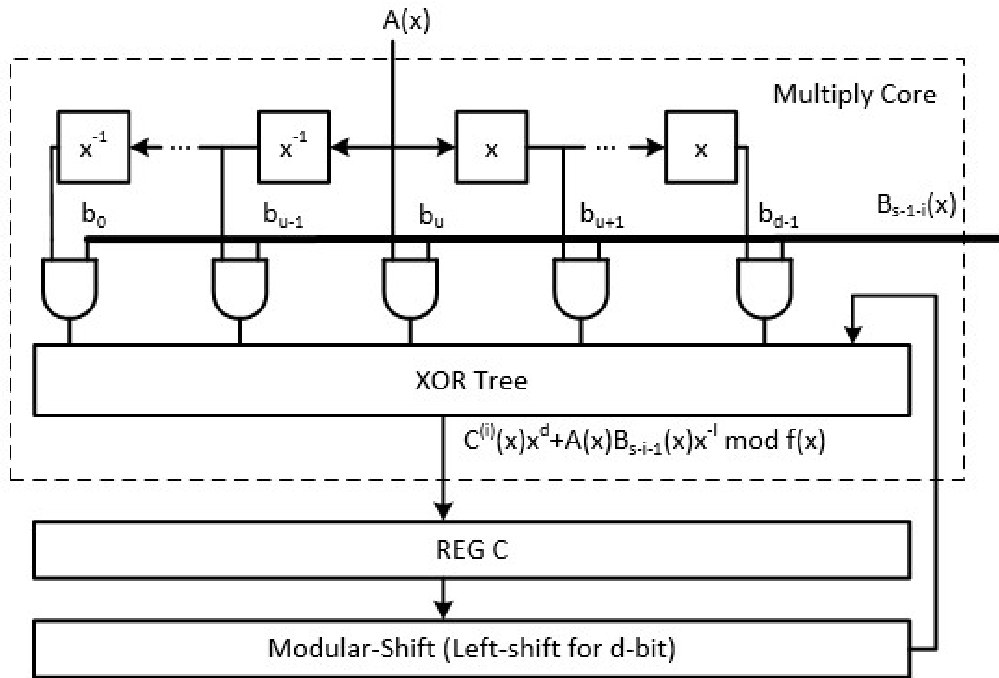

Figure 4.8: Advanced architecture of proposed multiplier

both architectures are completely the same, and they all need $s = \lceil m/d \rceil$ clock cycles to complete computation. However, the time delay of the latter one is shorter. Besides, the

Table 4.4: Complexity of proposed digit-serial MSD-first Montgomery multiplication (Algorithm I, advanced architecture, when $k_{i+1} - k_i \geqslant max\{l, d-l-1\},\ i = 0,1,2,3,\ k_0 = 0,\ k_4 = m$ and $0 \leqslant l \leqslant d-1$)

Work	#AND	#XOR	#FF/Reg	#CLK	Critical path delay
MSD(Arch.2)	md	$\leqslant md + 8d - 3$	m	s	$T_A + (1 + log_2(d+1))T_X + T_{DFF}$

latter architecture further extends the constraint condition of k_i from $k_{i+1} - k_i \geqslant d-1$ to $k_{i+1} - k_i \geqslant max\{l, d-l-1\}$, which indicates that more irreducible pentanomials can be applied to such Montgomery multiplication.

4.2 Proposed Digit-Serial LSD First Montgomery Multiplier

In this section, a digit-serial LSD first Montgomery Multiplier is proposed, and two different architectures are discussed when implementing the proposed multiplier. One of the architectures uses separate multiplication and reduction units, while the other one uses a linear-feedback-shift-register (LFSR) based structure.

4.2.1 Algorithm

Suppose $A(x), B(x) \in GF(2^m)$, in polynomial representation, $B(x)$ is divided into digits of the same size:

$$B(x) = \sum_{i=0}^{s-1} x^{id} B_i(x),\ where\ s = \lceil m/d \rceil \tag{4.18}$$

Let $C(x)$ be the product of $A(x)$, $B(x)$, and a fixed element $R^{-1}(x) = x^{-u} = x^{-sl}$, where $l \geqslant 0$, and l is an integer, the Montgomery multiplication can be computed by the way shown in equation (4.19). Based on equation (4.19), an algorithm of digit-serial LSD-first Montgomery multiplier can be proposed, see Table 4.5.

$$\begin{aligned}
C(x) =& A(x)\sum_{i=0}^{s-1} x^{id}B_i(x)x^{-sl} \\
=& A(x)B_0(x)x^{-sl} + A(x)x^{d}B_1(x)x^{-sl} + A(x)x^{2d}B_2(x)x^{-sl} + \\
& \cdots + A(x)x^{(s-1)d}B_{s-1}(x)x^{-sl} \\
=& A(x)B_0(x)x^{-sl} + A(x)x^{d-l}B_1(x)x^{-(s-1)l} + A(x)x^{2(d-l)}B_2(x)x^{-(s-2)l} + \\
& \cdots + A(x)x^{(s-1)(d-l)}B_{s-1}(x)x^{-l} \\
=& (((A(x)B_0(x)x^{-l} + A(x)x^{d-l}B_1(x))x^{-l} + A(x)x^{2(d-l)}B_2(x))x^{-l} + \\
& \cdots + A(x)x^{(s-1)(d-l)}B_{s-1}(x))x^{-l}
\end{aligned} \tag{4.19}$$

Table 4.5: Digit-serial LSD-first Montgomery Multiplier ($R(x) = x^{sl}$), where $l \geqslant 0$

Algorithm II	Digit-serial LSD-first Montgomery Multiplier
Input:	$A(x)$, $B_i(x)$, $f(x)$, $i = 0, 1, \ldots, s-1$
Outputs:	$C(x) = A(x)B(x)x^{-sl} \mod f(x)$, where $s = \lceil m/d \rceil$
Step 1:	$A^{(0)}(x) = A(x)$, $C^{(0)}(x) = 0$
	For $i = 0$ to $s-1$
Step 2:	$T_i(x) = A^{(i)}(x)B_i(x)$
Step 3:	$C^{(i+1)}(x) = (C^{(i)}(x) + T_i(x))/x^l \mod f(x)$
Step 4:	$A^{(i+1)}(x) = A^{(i)}(x)x^{d-l} \mod f(x)$
Step 5:	$C(x) = C^{(s)}(x)$

In Table 4.5, Step 1 is an initialization step, registers A and C are both set to zero; Step 2 computes the product of $A^{(i)}(x)$ and $B_i(x)$; the result of Step 2 is forwarded to Step 3, after adding the value of register C, a shift-to-right modulo operation is processed; Step 4 generates $A^{i+1}(x)$ as the operand of next clock cycle; when $i = s-1$, register C will output the final result at the end of the clock cycle.

4.2.2 General Architecture

The structure of the multiplier is shown in Fig 4.9. From top to bottom, block S1 computes Step 4; REG A updates every clock cycle; the Multiply Core computes the product of $A^{(i)}(x)$ and $B_i(x)$, note that the output bandwidth of the core is $m+d-1$; the XOR symbol

represents the operation $T_i(x) + C^{(i)}(x)$; block S2 computes the operation multiply by x^{-l} modulo $f(x)$; and finally, REG C stores the result of each clock cycle, and obtains the final product.

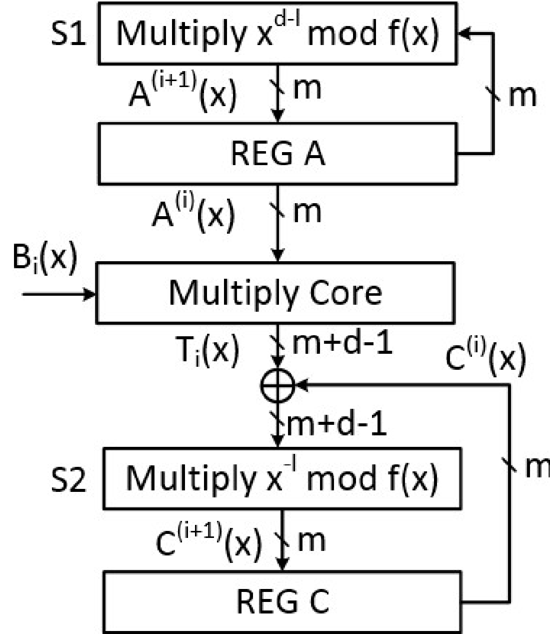

Figure 4.9: General architecture of the proposed digit-serial LSD first multiplier

When $0 \leqslant l \leqslant d-1$, with the change of l, the complexity of block S1 and S2 will also change, while the rest of the blocks remain the same. The implementation of Multiply Core is simply logic AND each of the two operands, then add up the terms which have the same degree. The two register unit includes only D-flipflops. Table 4.6 shows the complexity of Multiply Core, REG A, REG C, and the XOR array: The same as the proposed MSD first

Table 4.6:

Block	Gate Count	Time Delay
Multiply Core & XOR array	md AND, $md-d+1$ XOR	$T_A + log_2(d+1)T_X$
REG A	m D-flipflop	T_{DFF}
REG C	m D-flipflop	T_{DFF}

multiplier, the reduction operation in Step 3 is also a two-side reduction. The computation

equation is referred to equation (4.6):

$$\begin{aligned}
x^{m+d-l-2} &\mod f(x) = x^{k_3+d-l-2} + x^{k_2+d-l-2} + x^{k_1+d-l-2} + x^{d-l-2} \\
&\vdots \\
x^{m} &\mod f(x) = x^{k_3} + x^{k_2} + x^{k_1} + 1 \\
x^{-1} &\mod f(x) = x^{m-1} + x^{k_3-1} + x^{k_2-1} + x^{k_1-1} \\
x^{-2} &\mod f(x) = x^{m-2} + x^{k_3-2} + x^{k_2-2} + x^{k_1-2} \\
&\vdots \\
x^{-l} &\mod f(x) = x^{m-l} + x^{k_3-l} + x^{k_2-l} + x^{k_1-l}
\end{aligned} \tag{4.20}$$

In order to further optimize the time delay, the condition of $k_i,\ i = 1,2,3$, must be satisfied, see equation (4.10) Specifically, when $l = 0$, the proposed multiplier would be a

Table 4.7: Complexity of digit-serial LSD Montgomery multiplication (Algorithm II, when $1 \leqslant l \leqslant d-1$ and $k_{i+1} - k_i \geqslant d-1,\ k_0 = 0,\ k_4 = m$)

Work	#AND	#XOR	#FF/Reg	#CLK	Critical path delay
LSD($1 \leqslant l \leqslant d-1$)	md	$md + 3(2d-l-1)$	$2m$	s	$T_A + (1 + log_2(d+1))T_X + T_{DFF}$

standard polynomial multiplier; when $l = d-1$, the XOR gate cost is the lowest, which is equal to $md + 3d$.

When $l = d$, the architecture of the multiplier can be further optimized: since $d - l = 0$, REG A and S1 can be saved, S2 computes multiply by $x^{-d} \mod f(x)$, the reduction operation is only one-side. Table 4.8 gives the complexity summary when $l = d$.

Table 4.8: Complexity of digit-level Montgomery multiplication (Algorithm II, when $l = d$, and $k_{i+1} - k_i \geqslant d-1,\ k_0 = 0,\ k_4 = m$)

Work	#AND	#XOR	#FF/Reg	#CLK	Critical path delay
LSD($l = d$)	md	$md + 3d$	m	s	$T_A + (1 + log_2(d+1))T_X + T_{DFF}$

When $l \geqslant d+1$, since $l > l-d$, we could predict that if we avoid multiple reduction operations in block S2, we could also avoid the multiple reduction in block S1. Besides, since $l \geqslant d+1$, the modulo $f(x)$ operation is only one-side reduction. l must satisfy $l \leqslant k_1$ to avoid multiple reduction.

Assume we divide $(C^{(i)}(x)+T_i(x))/x^l$ into two parts:

$$(C^{(i)}(x)+T_i(x))/x^l = T(x)+T_L(x) \tag{4.21}$$

Then the reduction operation would be:

$$(C^{(i)}(x)+T_i(x))/x^l \mod f(x) = T_L(x)x^m + T_L(x)x^{k_3} + T_L(x)x^{k_2} + T_L(x)x^{k_1} + T(x) \tag{4.22}$$

The degree range of each product in equation (4.22) is presented in Table 4.9.

Table 4.9: Degree range of each term of equation (4.22)

Terms	Degree Range
$T_L(x)x^m$	$[m-l, m-1]$
$T_L(x)x^{k_3}$	$[k_3-l, k_3-1]$
$T_L(x)x^{k_2}$	$[k_2-l, k_2-1]$
$T_L(x)x^{k_1}$	$[k_1-l, k_1-1]$
$T(x)$	$[0, m+d-l-2]$

From Table 4.9, obviously, $m+d-l-2 < m-1$, thus, instead of using l XOR gates to add term $T_L(x)x^{m-l}$ to $T(x)$, we only need $d-1$ XOR gates. Similarly, if $m+d-l-2 < k_3-1$, the XOR gate count of block S2 can be further reduced. Here we use a table to present this result, see Table 4.10

Table 4.10: Value of l in terms of XOR gate usage of block S1

Conditions	XOR Gate Count of Block S1
$d+1 \leqslant l \leqslant min\{m+d-k_3-1, k_1\}$	$3l+d-1$
$m+d-k_3 \leqslant l \leqslant min\{m+d-k_2-1, k_1\}$	$2l+2(d-1)+(m-k_3)$
$m+d-k_2 \leqslant l \leqslant min\{m+d-k_1-1, k_1\}$	$l+3(d-1)+(m-k_3)+(m-k_2)$
$m+d-k_1 \leqslant l \leqslant k_1$	$4(d-1)+(m-k_3)+(m-k_2)+(m-k_1)$

Considering the time delay of block S2, when conditions $k_3 < m-l+1$, $k_2 < k_3-l+1$, and $k_1 < k_2-l+1$ are all satisfied, the time delay is T_X. To sum up, k_i must satisfy:

$$k_{i+1}-k_i \geqslant l \tag{4.23}$$

where $i = 0,1,2,3$ and $k_0 = 0$, $k_4 = m$. However, comparing with equation (4.10), equation

(4.23) has narrowed the condition. The complexities of the multiplier when $l > d$ is referred to Table 4.11.

Table 4.11: Complexity of digit-level Montgomery multiplication (Algorithm II, when $l > d$, and $k_{i+1} - k_i \geqslant l,\ k_0 = 0,\ k_4 = m$)

Work	#AND	#XOR	#FF/Reg	#CLK	Critical path delay
LSD($l > d$)	md	$\leqslant md + 6l - 3d$	$2m$	s	$T_A + (1 + log_2(d+1))T_X + T_{DFF}$

4.2.3 LFSR-Based Architecture

When $0 \leqslant l \leqslant d-1$, a LFSR-based architecture can be provided.

Table 4.12: LFSR-Based Digit-serial LSD-first Montgomery Multiplier ($R(x) = x^{sl}$), where $0 \leqslant l \leqslant d-1$

Algorithm III	LFSR-Based Digit-serial LSD-first Montgomery Multiplier
Input:	$A(x)$, $B_i(x)$, $f(x)$, $i = 0, 1, \ldots, s-1$
Outputs:	$C(x) = A(x)B(x)x^{-sl} \mod f(x)$, where $s = \lceil m/d \rceil$
Step 1:	$A^{(0)}(x) = A(x)$, $C^{(0)}(x) = 0$
	For $i = 0$ to $s-1$
Step 2:	$T_i(x) = A^{(i)}(x)B_i(x)/x^l \mod f(x)$
Step 3:	$C^{(i+1)}(x) = C^{(i)}(x)/x^l \mod f(x) + T_i(x)$
Step 4:	$A^{(i+1)}(x) = A^{(i)}(x)x^{d-l} \mod f(x)$
Step 5:	$C(x) = C^{(s)}(x)$

A minor change in Step 2 and 3 of Algorithm II is applied, and Table 4.12 presents the new algorithm. In Algorithm III, Step 2 is computed as follows:

$$T_i(x) = \sum_{j=0}^{d-1} A^{(i)} x^{j-l} \mod f(x) \cdot b_{id+j} \tag{4.24}$$

In equation (4.24), $A^{(i)}x^{j-l} \mod f(x)$ is computed first, then logic AND each bit of $B_i(x)$. Since $j = 0, 1, 2, \ldots, d-1$, when $j = d-1$, the corresponding term of $A^{(i)}x^{j-l}$ equals $A^{(i)}x^{d-l-1}$, also note that in Step 4 of Algorithm III, $A^{(i)}x^{d-l} = A^{(i)}x^{d-l-1} \cdot x$, thus, by applying the circuit structure provided by Fig 4.4 and Fig 4.6, a LFSR based architecture can be obtained, see Fig 4.10.

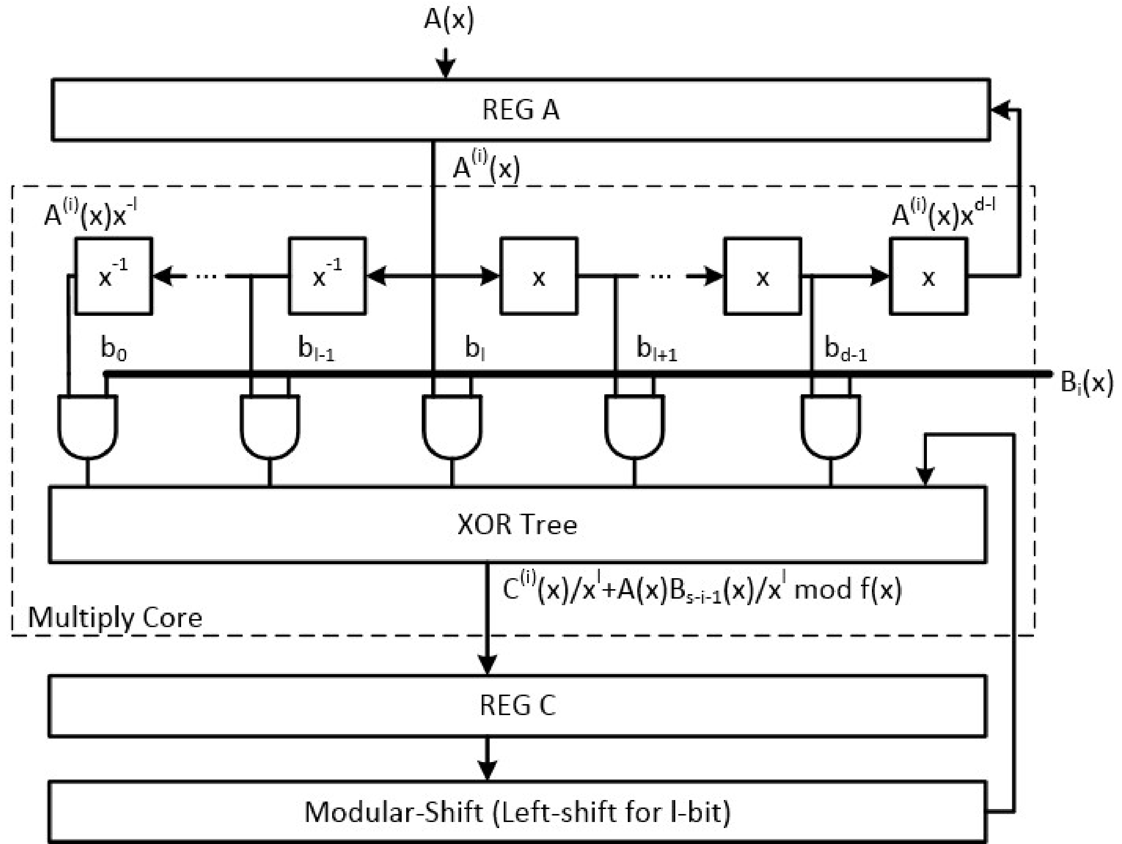

Figure 4.10: LFSR-based architecture of the proposed LSD Montgomery multiplier

In the architecture, register A and $d-l$ Model 1 units consist of a linear feedback shift circuit, in addition, multiplying x modular operation and multiplying x^{-1} modular operation are divided into two separate parts. Each unit of Model 1 and Model 2 cost 3 XOR gates, and in total, $3(d)d$XOR gates. Thus, to have the minimum time delay of the architecture, k_i must satisfy:

$$k_{i+1} - k_i \geqslant max\{l, d-l-1\} \tag{4.25}$$

where $i = 0, 1, 2, 3$, $k_0 = 0$, and $k_0 = m$. By applying the condition described in equation (4.25), the time delay of Multiply $x{-}l \mod f(x)$ will be T_X, and costs $3l$ XOR gates. The remaining blocks, REG A and REC C, have the same structure with as the general architecture reported in subsection 4.2.2. Table 4.13 gives the complexity of such architecture. Compared with the general architecture, the proposed architecture in this subsection has

the same critical path delay. However, the LFSR-based architecture broadens the condition of irreducible pentanomial selection, the condition is extended from equation (4.10) to equation (4.25), also note that if we change parameter l to u, equation (4.25) is completely the same with equation (4.17).

Table 4.13: Complexity of digit-level Montgomery multiplication (Algorithm III, when $0 \leqslant l \leqslant d-1$, and $k_{i+1} - k_i \geqslant max\{l, d-l-1\},\ k_0 = 0,\ k_4 = m$)

Work	#AND	#XOR	#FF/Reg	#CLK	Critical path delay
LSD(LSFR)	md	$md+3d+3l$	$2m$	s	$T_A + (1+log_2(d+1))T_X + T_{DFF}$

4.3 Complexity Analysis

In this section, complexities of the proposed work in terms of gate count and time delay will be investigated and compared with several types of digit-level multipliers. Table 4.14 gives the practical time delay of 2-input AND gate and 2-input XOR gate at 25°C, $1.8V$ based on CMOSP18 technology as a reference.

Table 4.14: Intrinsic delay of XOR2 and AND2 gate, we assume each gate could drive a maximum of two gates (25°C, $1.8V$, CMOSP18 Tech., $Y = A \cdot B$, or $Y = A \oplus B$)

Description	Delay of AND2(ns)	Delay of XOR2(ns)
$A \rightarrow Y \uparrow$	0.0720	0.1351
$A \rightarrow Y \downarrow$	0.0970	0.1294
$B \rightarrow Y \uparrow$	0.0763	0.1209
$B \rightarrow Y \downarrow$	0.1091	0.1475

Table 4.15 gives the comparison result of the work reported in [28] and our proposed work, both are digit-serial Montgomery multipliers. From the table, it can be seen that the critical path delay of the proposed works are better then [28]. As a trade off, the XOR gate count is greater than [28], except when the case $l = d$, our proposed works have even better gate count than the works reported in [28].

The proposed works are Montgomery multiplier, the definition of which is different from general polynomial basis multipliers. Since both multiplications can be done using polynomial basis, and they are similar in architecture level, thus we consider they are comparable.

Table 4.15: Digit-serial Montgomery multipliers comparison ($f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, $s = m/d$)

Type	Work	#AND	#XOR	#DFF	#CLK	Critical path delay
MSD	$f(x)$ for general irreducible pentanomials					
	[28](x^m)	md	$md+3d$	$2m$	s	$T_A + (3+\lceil log_2 d\rceil)T_X + T_{DFF}$
	[28](x^{m-1})	md	$md+3d$	$2m$	s	$T_A + (3+\lceil log_2 d\rceil)T_X + T_{DFF}$
	$f(x)$ satisfying $k_{i+1} - k_i \geqslant d-1$, $i = 0,1,2,3$, $k_0 = 0$, $k_4 = m$					
	Proposed($Arch.1$,x^l)	md	$\leqslant md+6d$	m	s	$T_A + (2+\lceil log_2 d\rceil)T_X + T_{DFF}$
	$f(x)$ satisfying $k_{i+1} - k_i \geqslant max\{l, d-l-1\}$					
	Proposed($Arch.2$,x^l)	md	$\leqslant md+8d-3$	m	s	$T_A + (1+\lceil log_2 (d+1)\rceil)T_X + T_{DFF}$
LSD	$f(x)$ for general irreducible pentanomials					
	[12](x^m)	$2m^2+md$	$2m^2+md+4s^2$	$4ms+2m$	s	$4T_A + 7T_X + 4T_{DFF}$
	$f(x)$ satisfying $k_{i+1} - k_i \geqslant d-1$					
	Proposed($l<d$,x^{sl})	md	$md+3(2d-l-1)$	$2m$	s	$T_A + (1+\lceil log_2 (d+1)\rceil)T_X + T_{DFF}$
	Proposed($l=d$,x^m)	md	$md+3d$	m	s	$T_A + (1+\lceil log_2 (d+1)\rceil)T_X + T_{DFF}$
	$f(x)$ satisfying $k_{i+1} - k_i \geqslant max\{l, d-l-1\}$					
	Proposed(LSFR, x^{sl})	md	$md+3d+3l$	$2m$	s	$T_A + (1+\lceil log_2 (d+1)\rceil)T_X + T_{DFF}$

Table 4.16 shows the comparison between the proposed MSD first multipliers and a group of MSD first Polynomial Basis finite field multipliers, and the field is generated by irreducible pentanomials. The table implies that the proposed MSD first multipliers have the smallest gate count. The time delay of the proposed multipliers are smaller than [17], but larger than that of [13], however, [13]'s work need one extra clock cycle to obtain the final result since a final reduction unit is applied in [13]'s proposed multiplier.

Table 4.16: Proposed multipliers compared with Polynomial Basis finite field multipliers (MSD cases, $f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, $s = \lceil m/d \rceil$, T_{DFF} represents the time delay of a D-flipflop)

Work	#AND	#XOR	#DFF	#CLK	Critical path delay
$f(x)$ for general irreducible pentanomials					
[13](MSD)	md	$2md-d+m$	$m+d$	$s+1$	$T_A + \lceil log_2 (2d+1)\rceil T_X + T_{DFF}$
[17]	md	$3(d^2+d)/2+md$	m	s	$T_A + (3+\lceil log_2 d\rceil)T_X + T_{DFF}$
$f(x)$ satisfying $k_{i+1} - k_i \geqslant d-1$, $i = 0,1,2,3$, $k_0 = 0$, $k_4 = m$					
Proposed($Arch.1$)	md	$\leqslant md+6d$	m	s	$T_A + (2+\lceil log_2 d\rceil)T_X + T_{DFF}$
$f(x)$ satisfying $k_{i+1} - k_i \geqslant max\{l, d-l-1\}$					
Proposed($Arch.2$)	md	$\leqslant md+8d-3$	m	s	$T_A + (1+\lceil log_2 (d+1)\rceil)T_X + T_{DFF}$

Table 4.17 presents the comparison results between proposed LSD first multiplier and LSD first PB multipliers. When $l = d$ and $l = d-1$, the XOR gate usage of the proposed LSD first multipliers is minimum. It can be seen the proposed works have the least usage of AND gate, register and MUX cell, but the XOR gate usage is more than the architecture

reported in [24], that is because a T-flipflop is applied to implement the accumulator instead of a D-flipflop and a XOR gate. Also, [19]'s work uses a final reduction unit to compute the reduction operations instead of computing the reduction operation in each clock cycle, the critical path delay of [19] is shorter than the proposed multipliers, but a one more clock cycle is required before obtaining the final result. Therefore, by taking the both gate count and time delay into consideration, the proposed LSD-first multipliers are still remarkable.

Table 4.17: Proposed multipliers compared with Polynomial Basis finite field multipliers (LSD cases, T_M represents the time delay of a 2×1 Multiplexer, T_{TFF} represents the time delay of a T-flipflop)

Work	#AND	#XOR	#DFF/TFF	#MUX	#CLK	Critical path delay
$f(x)$ for general irreducible pentanomials						
[13](LSD)	md	$2md-d+m$	$2m+d-1$	m	$s+1$	$T_A+\lceil log_2(d+1)\rceil T_X+T_{DFF}+T_M$
[19]	$md+8d-4$	$md+7d-4$	$2m+d-1$	0	$s+1$	$T_A+\lceil log_2(d+1)\rceil T_X+T_{DFF}$
[24]	md	$m(d-1)+3(d^2+d)/2$	$2m+d$	0	s	$T_A+(2+\lceil log_2 d\rceil)T_X+T_{TFF}$
$f(x)$ satisfying $k_{i+1}-k_i \geqslant d-1$, $i=0,1,2,3$, $k_0=0$, $k_4=m$						
Proposed($l<d$)	md	$md+3(2d-l-1)$	$2m$	0	s	$T_A+(1+\lceil log_2(d+1)\rceil)T_X+T_{DFF}$
Proposed($l=d$)	md	$md+3d$	m	0	s	$T_A+(1+\lceil log_2(d+1)\rceil)T_X+T_{DFF}$
$f(x)$ satisfying $k_{i+1}-k_i \geqslant max\{l,d-l-1\}$						
Proposed(LSFR)	md	$md+3d+3l$	$2m$	0	s	$T_A+(1+\lceil log_2(d+1)\rceil)T_X+T_{DFF}$

If we let $m=233$, $d=8$ and $l=4$, thus, $s=m/d=30$, and consider the area and latency by making the following assumptions: (1) The VLSI areas of an XOR gate is approximately two times of the area of an AND gate 2AND=XOR, as well as the gate delay $2T_A=T_X$; (2) The VLSI areas and time delay of an DFF is approximately three times of an AND gate 3AND=DFF, $3T_A=T_X$; (3) The VLSI areas and time delay of an TFF is approximately 3.5 times of an AND gate 3.5AND=TFF, $3.5T_A=T_X$; (4) The VLSI areas and time delay of an 2X1 Multiplexer is approximately two times of an AND gate 2AND=MUX, $2T_A=T_M$. Based on these assumptions, we could use the gate count and delay of AND gate to estimate the efficiency of the proposed and existing works. See Table 4.18 and Table 4.19.

In Table 4.18, use area and delay of AND gate to estimate the Montgomery multipliers, and assume the area and time efficiency of the proposed MSD multiplier is 100%, then calculate the efficiency of other proposed architecture and existing Montgomery multipliers. Note that when value of efficiency less than 100% implies a improvement is applied. The result shows when $m=233$ and $d=8$, the architecture of the proposed MSD and LSD-first Montgomery multipliers could reduce the time delay compared with [28] and [12], also with the reduced area cost.

Table 4.18: Efficiency of the proposed multipliers and existing Montgomery multipliers ($m = 233$, $d = 8$, if $l < d$, then $l = 4$)

Work	Area	Area Efficiency	Time Delay	Time Efficiency
[12]	423804	6635.42%	30	214.29%
[28](x^m)	7038	110.19%	16	114.29%
[28](x^{m-1})	7038	110.19%	16	114.29%
Proposed(MSD, $Arch.1$, x^l)	6387	100%	14	100%
Proposed(MSD, $Arch.1$, x^l)	6413	100.41%	14	100%
Proposed(LSD, $l < d$, x^{sl})	7056	110.47%	14	100%
Proposed(LSD, $l = d$, x^{sl})	6339	99.25%	14	100%
Proposed(LSD, LSFR, x^{sl})	7062	110.57%	14	100%

Table 4.19: Efficiency of the proposed multipliers and existing PB multipliers ($m = 233$, $d = 8$, if $l < d$, then $l = 4$)

Work	Area	Area Efficiency	Time Delay	Time Efficiency
[13](MSD)*	10493	164.29%	14	100%
[13](LSD)*	11655	182.48%	14	100%
[17]	6483	101.50%	16	114.29%
[19]*	7175	112.34%	12	85.71%
[24]	6470	101.30%	14.5	103.57%
Proposed(MSD, $Arch.1$, x^l)	6387	100%	14	100%
Proposed(MSD, $Arch.1$, x^l)	6413	100.41%	14	100%
Proposed(LSD, $l < d$, x^{sl})	7056	110.47%	14	100%
Proposed(LSD, $l = d$, x^{sl})	6339	99.25%	14	100%
Proposed(LSD, LSFR, x^{sl})	7062	110.57%	14	100%

In Table 4.19, works mark with "*" need one extra clock cycle to obtain the final result. We assume the proposed MSD-first architecture has 100% efficiency and compare it with other proposed Montgomery multipliers and existing PB multipliers. In general, the proposed architectures further reduce the time delay, the area cost is within comparable size.

According to these comparisons, by applying the proposed two classes of fields, the proposed MSD-first digit-serial Montgomery multiplier and LSD-first digit-serial Montgomery multiplier have less time delay than the existing digit-level Montgomery multipliers, and less than most of the existing Polynomial Basis multipliers. The gate count of the proposed multipliers is also comparable with the most existing works. The proposed work

is remarkable in terms of the further reduction of the critical path delay.

4.4 FPGA Implementation of the Proposed Multipliers

In this section, the proposed MSD-first and LSD-first Montgomery multipliers are implemented using FPGA. The advanced architecture of the MSD-first multiplier is selected to be implemented while the general architecture of the LSD-first multiplier is selected. The finite field size is set to be $m = 233$, and polynomial $f(x) = x^{233} + x^{185} + x^{121} + x^{105} + 1$ is chosen to generate $GF(2^{233})$. Digit size $d = 8$; integer u of the MSD-first multiplier and l of the LSD-first multiplier equal 4 respectively. FPGA development tool: Quartus II v9.1 and ModelSim v6.5b. FPGA model: Stratix II, EP2S60F1020C3.

4.4.1 Summary of the MSD-First Multiplier Implementation

Table 4.20: Cells usage of compilation ($m = 233$, $d = 8$, $u = 4$)

Cells	Usage
Total registers	233
Total pins	476
$\leq$ 3-input combinational ALUT	0
4-input combinational ALUT	227
5-input combinational ALUT	436
6-input combinational ALUT	275
Total combinational functions	938

Table 4.21: Gate count of each module ($m = 233$, $d = 8$, $u = 4$)

Module	#Logic combinational functions	#Register
REG C	0	233
Multiply Core	959	0
Unit of Multiplied by x^d	24	0
Top-level	983	233

Table 4.20 provides the usage of logic cells, including gates, pins as well as registers, after compiling. The term logic unit in the table represents logic gates and other types of

Table 4.22: Time complexity of the design ($m = 233$, $d = 8$, $u = 4$)

Clock setup	Restricted to 500.00MHz
Clock period	2.000ns
Number of clock cycles for one multiplication	30
Total time cost for one multiplication	60.0ns

logic cells which are involved in a FPGA device. Table 4.21 shows the gate count of each module, since the compiler may optimize the structure when compiling, thus some modules may contains fewer logic elements than the designed architecture. Table 4.22 is a summary of time complexity of the design. Clock setup is the maximum operation speed the design can reach. In this implementation, the maximum clock frequency of the selected FPGA is 500MHz. Also the number of cycles and processing time for one multiplication are also included.

4.4.2 Summary of the LSD-First Multiplier Implementation

Table 4.23: Cells usage of compilation ($m = 233$, $d = 8$, $l = 4$)

Cells	Usage
Total registers	466
Total pins	476
$\leq$ 3-input combinational ALUT	12
4-input combinational ALUT	224
5-input combinational ALUT	451
6-input combinational ALUT	257
Total combinational functions	944

Table 4.24: Gate count of each module ($m = 233$, $d = 8$, $l = 4$)

Module	#Logic combinational functions	#Register
REG A	0	233
REG C	0	233
Multiply Core	911	0
S1	12	0
S2	21	0
Top-level	944	233

Table 4.25: Time complexity of the design ($m = 233$, $d = 8$, $l = 4$)

Clock setup	$326.16MHz$
Clock period	$3.066ns$
Number of clock cycles for one multiplication	30
Total time cost for one multiplication	$91.98ns$

Note that the fixed element $R(x)$ of the proposed digit-serial LSD-first Montgomery multiplier is $R(x) = x^{sl}$. The compilation results shows that the proposed LSD-first multiplier doubles the usage of the register compared with the proposed MSD-first multiplier, also the clock frequency is lower, however, the usage of logic element is less than the proposed MSD-first one.

Chapter 5

FPGA Implementation of Inverse Generator

In this chapter, we will introduce the FPGA implementation of a normal basis inverse generator. We first give the architecture of the inverse generator. Then, schemes for each module of the generator are provided and explained respectively, as well as an algorithm of the normal basis multiplication over $GF(2^m)$. We also obtain the simulation and compilation result of our designed inverse generator, the gate usage and clock setup are included in our implementation result. FPGA development tool: Quartus II v9.1 and ModelSim v6.5b. FPGA model: Stratix II, EP2S60F1020C3.

5.1 The Design of Inverse Generator

Fig 5.1 presents the architecture of the designed normal basis inverse generator [30]. And Fig 5.2 shows the block diagram for FPGA implementation. Comparing the previous two figures, the REG2 and 2^x-power blocks are replaced by a shift register block, since the 2^x exponentiation operation in normal basis is simply shift operation, see equation (5.1) as an example. Besides, the normal basis multiplier module is implemented using a digit-level structure in order to reduce the gate count, and the number of clock cycle will be increased as a trade off. Also we add some control signals in order to control the operation of the design: input ”clk” signal to provide system clock; ”clk1” signal to enable or disable REG1 and REG2 module; input ”rst” signal to restart or reset the inverse generator; output ”rdy”

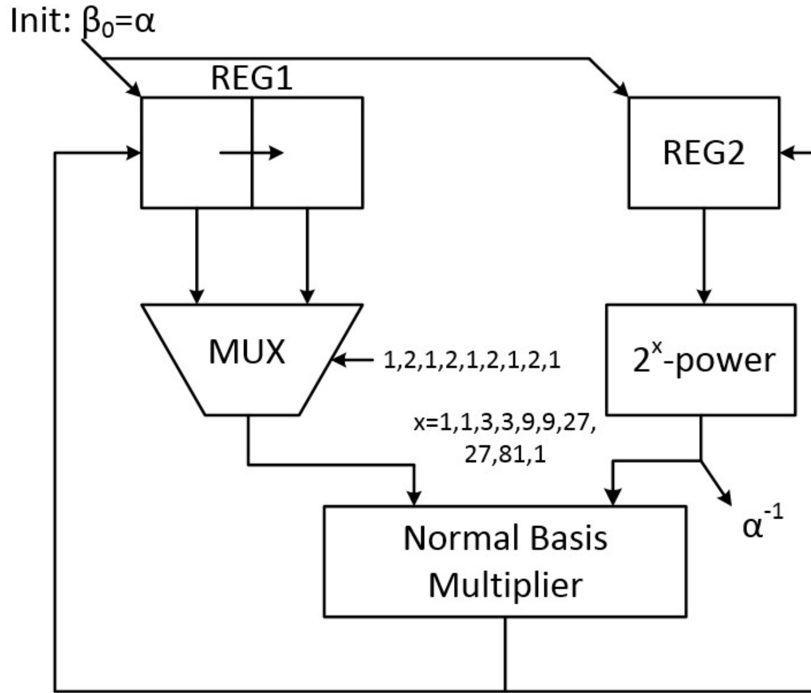

Figure 5.1: Architecture of the designed inverse generator

signal to indicate the final result is generated; and finally, the output ”ctrl” takes the place of select signal of MUX. In the following subsections, the design of each module will be introduced.

$$(\theta^{2^i})^{2^x} = \theta^{2^i \cdot 2^x} = \theta^{2^{i+x}} \tag{5.1}$$

5.1.1 REG1 Module

See Fig 5.3 for the REG1 module. The module contains two 163-bit registers, say R0 and R1, ”reg_out0” is the output of R0 and ”reg_out1” is the output of R1. For each positive edge of ”clk1” signal, when ”rst” is logic one, REG1 will load the data from port ”reg_init” into R0; when ”rst” is logic zero, REG1 will load the data from ”reg_in” into R0, then at the same clock cycle, R0 passes its data to R1. ”rdy” acts as an enable signal, when ”rdy” equals logic one, REG1 remains no change. ”ctrl” signal controls the data selection of

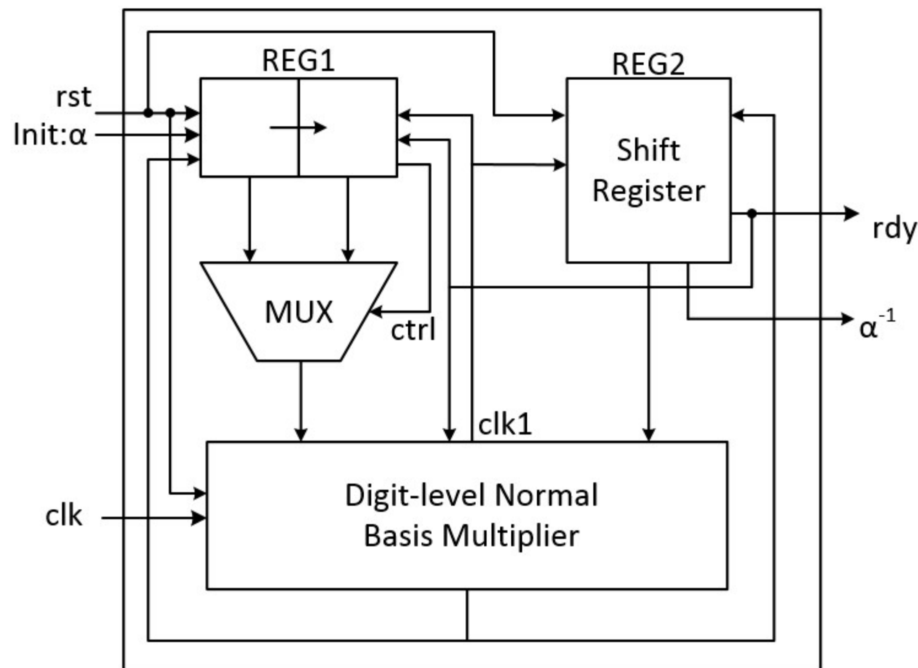

Figure 5.2: Block diagram of the inverse generator for FPGA implementation

MUX module: when ”rst” is one, ”ctrl” signal is set low-level voltage, which is logic zero, and when ”rst” is zero, ”ctrl” jumps to the opposite voltage level for each positive clock edge, for example, when ”rst” is zero, the output value of ”ctrl” will be: $1,0,1,0,\ldots$.

5.1.2 REG2 Module

See Fig 5.4 for the REG2 module. A counter is included in this module. ”clk1” inputs the clock signal. When ”rst” is one, the register will load the data from ”reg_init” and circular left shift for one bit, then forward the result to output port ”reg_out”, also ”rdy”, ”inverse_out” and the counter are all set to zero. When ”rst” equals to zero, the register will load the data from ”reg_in”, circular left shift the data, then forward the data to the output port ”reg_out”, at the same time, the counter is increased by one. For the different value of the counter increase from 0 to 8, the register shifts the input data by $1,3,3,9,9,27,27,81,1$-bit, respectively. When counter is equal to 9, ”rdy” signal is set to be logic one, the ”inverse_out” output the final result α^{-1} of the normal basis inverse generator.

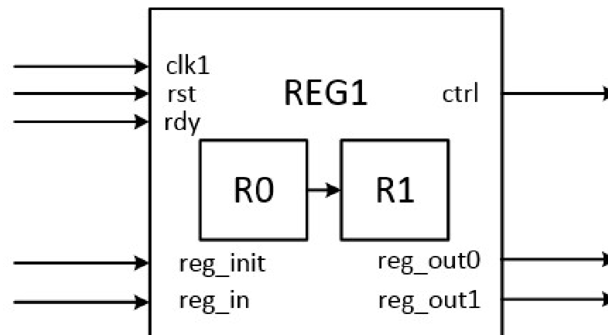

Figure 5.3: REG1 module

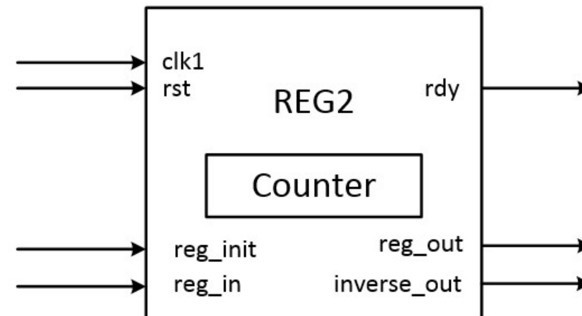

Figure 5.4: REG2 module

5.1.3 MUX Module

Fig 5.5 presents the multiplexer module. ”d_0” and ”d_1” are two data inputs. When ”ctrl” equals to zero, ”q” select and output the data of ”d_0”; and when ”ctrl” is one, ”q” is equal to the value of ”d_1”.

5.1.4 Digit-level Normal Basis Multiplier Module and Multiplication Algorithm

Fig 5.6 is block diagram of the digit-level Normal Basis multiplier module, ”a_in” and ”b_in” are two operands of the multiplication operation, and ”out” port outputs the product of finite field $GF(2^m)$ multiplication; ”rdy” could enable/disable the module, and ”rst” is a

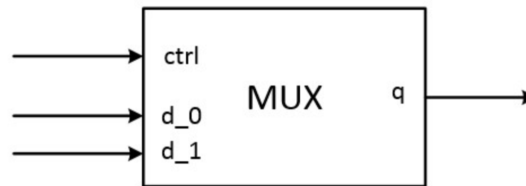

Figure 5.5: MUX module

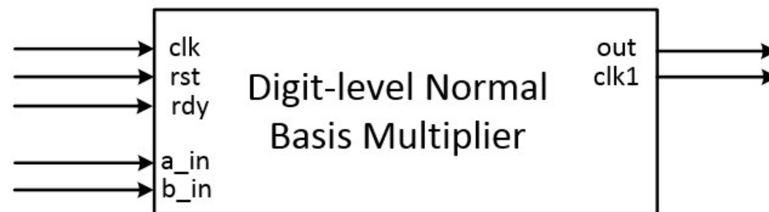

Figure 5.6: Digit-level normal basis multiplier module

reset signal, when ”rst” is equal to logic one, the module is reset to its initial state; ”clk” is the clock signal and output signal ”clk1” is the drive signal of REG1 and REG2.

This module contains three sub-modules: Input Reg module, NB Multiplier module, and Output Reg module, see Fig 5.7 for details. For each clock cycle, if ”rst” and ”rdy” signals both are logic zero, the input_reg module will do two alternative jobs: first, if the inner counter equals to zero, the module sets ”clk1” to logic one, and reads the data from ”a_in” and ”b_in” and stores them in the register after circular left shift both two bit strings by 5 bits (since 163 is not dividable by 8, the sixth bit of the least significant digit must be the LSB of a_in×b_in, in that case, after right shift the bit string of the product a_in×b_in for 21 times and 8 bits each time, we could finally get the right answer); second, for other cases, the module will right shift the data by 8 bits, and ”clk1” is set to be logic zero.

The NB Multiplier module contains only combinational circuits, no drive signals, nor control signals. The two inputs are both 163-bit and output is 8-bit. Which indicates the multiplier needs 21 clock cycles to calculate all 163-bit consequences. The Output Reg module stores the results of the NB multiplier module for each positive clock edge and right shift by 8 bits.

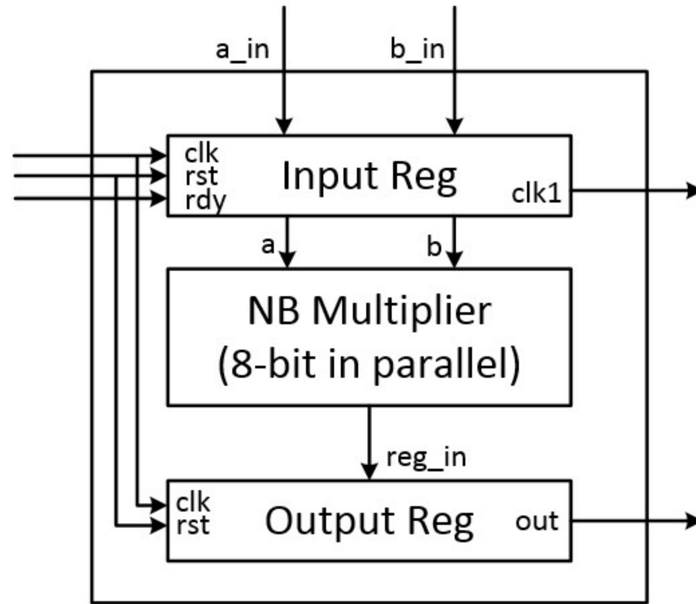

Figure 5.7: Digit-level Normal Basis multiplier structure

Following by the structure of the multiplier module, an algorithm of normal basis multiplication is provided. Since the design is a normal basis multiplier over $GF(2^{163})$, according to [14], there should exist a type 4 ($T = 4$) Gaussian normal basis for $GF(2^{163})$. Here we first check the existence of this Gaussian normal basis for $GF(2^{163})$ of given type $T = 4$. The algorithm is given below:

Input: an integer $m > 1$ not divisible by 8; a positive integer T.
Output: if a type T Gaussian normal basis for $GF(2^m)$ exits, the message "True"; otherwise "False".

1. Set $p \leftarrow Tm + 1$.
2. If p is not prime then output "False" and stop.
3. Compute the order k of 2 module p.
4. Set $h \leftarrow Tm/k$.
5. Compute $d := GCD(h, m)$.
6. If $d = 1$ then output "True"; else output "False".

In this case, $m = 163$, $T = 4$, then get $p = Tm + 1 = 653$ is a prime number. After computing, we know that 2 has the order of 652 module 653, thus $h = 1$, and $d = GCD(1, 163) = 1$, so there does exist a Gaussian normal basis of type 4 for $GF(2^{163})$. Therefore, we could generate the first coordinate of the product of two elements which belong to the $GF(2^{163})$ type 4 Gaussian normal basis. The algorithm is given below [14]:

Input: integers $m > 1$ and T for which there exit a type T Gaussian normal basis G for $GF(2^m)$, $A, B \in GF(2^m)$, $A = (a_{m-1}a_{m-2}\dots a_1a_0)$, $B = (b_{m-1}b_{m-2}\dots b_1b_0)$.
Output: an explicit formula for the first coordinate of the product of two elements with respect to G.

1. Set $p \leftarrow Tm + 1$.
2. Generate an integer u having order T modulo p.
3. Compute the sequence $F(1),\ F(2),\ \dots,\ F(p-1)$ as follows:
   - 3.1. Set $w \leftarrow 1$.
   - 3.2. For j from 0 to $T-1$ do
     - Set $n \leftarrow w$
     - For i from 0 to $m-1$ do
       - Set $F(n) \leftarrow i$
       - Set $n \leftarrow 2n \bmod p$
     - Set $w \leftarrow uw \bmod p$
4. If T is even, then set $J \leftarrow 0$, else set

$$J := \sum_{k=1}^{m/2} (a_{k-1}b_{m/2+k-1} + a_{m/2+k-1}b_{k-1})$$

5. Output the formula

$$c_0 = J + \sum_{k=1}^{p-2} a_{F(k+1)}b_{F(p-k)}$$

For $T = 4$ normal basis of $GF(2^{163})$, we could calculate the value of p is equal to 653, and $u = 149$ have the order 4 modulo 653. Since $T = 4$ is even, we have $J = 0$. Then we

use C-program to generate all the value of $F(s)$, where $s = 1, 2, \ldots, p-1$. Furthermore, C-program is used to generate the VerilogHDL code of first coordinate c_0, which can be applied to FPGA implementation, see Appendix A and Appendix B. The other coordinates of the product are obtained from the formula c_0 by cycling the subscripts modulo m.

5.1.5 Top-Level

See Fig 5.2 for reference. The top-level module has three input ports: ”clk” is the designed system clock signal, ”rst” is the reset signal, ”alpha” is the value of normal basis element which is going to be inversed; two output ports: ”rdy” is a indicator signal to imply the final result is ready, ”inverse_alpha” is the value of the consequence. The top-level module is also known as the design entity of the normal basis inverse generator.

5.2 Simulation and Compilation

5.2.1 Simulation Results

Table. 5.1 presents an operation description of each clock, also the change of all the signals and registers involved in the generator is included.

Table 5.1: Description of Each Clock cycle

clock cycle#	description
Initialization	reg1 and reg2 read the data from port ”alpha”; reg1 stores the data into register R0, and register R1 is set to zero; MUX forwards the selected data into the multiplier as operand A; reg2 module circular left shifts the input data and forwards it to the multiplier as operand B. all counters are set to zero (counter of reg2 and counter of nb_multiplier) $rst <= 1, rdy <= 0, ctrl <= 0, clk1 <= 1$ $reg2/count <= 0, nb_multiplier/count <= 0$ $alpha <= 163'h00000000000000000000000000000000000000001$
Continued on next page	

Table 5.1 – continued from previous page

clock cycle#	description
	$a_in <= 163'h00000000000000000000000000000000000000001$ $b_in <= 163'h00000000000000000000000000000000000000002$ $inverse_alpha <= 163'h000000\ldots000000$
# 1	"rst" signal is set to zero; input_reg1 circular right shifts the two operands by 5 bits, then forwards them to then to the nb_multiplier; nb_multiplier stop loading data from reg1 and reg2; nb_multiplier compute the product and forwards it to output_reg; output_reg stores the result at most significant 8 bits and left shift 8 bits; counter of nb_multiplier increased by 1; all the another registers remain no change. $rst <= 0$, $rdy <= 0$, $ctrl <= 0$, $clk1 <= 0$ $reg2/count <= 0$, $nb_multiplier/count <= 1$ $a_out <= 163'h00000000000000000000000000000000000000020$ $b_out <= 163'h00000000000000000000000000000000000000040$ $output_reg/reg_in <= 8'h20$ $inverse_alpha <= 163'h000000\ldots00000$
# 2	input_reg1 circular left shifts the two operands by 8 bits, then forwards them to then to the nb_multiplier; output_reg stores the result at most significant 8 bits and left shift 8 bits; counter of nb_multiplier increased by 1; all the another registers remain no change. $reg2/count <= 0$, $nb_multiplier/count <= 2$ $a_out <= 163'h10000000000000000000000000000000000000000$ $b_out <= 163'h20000000000000000000000000000000000000000$ $output_reg/reg_in <= 8'h00$
# 3 - # 21	same operation as clock cycle # 2.
# 22	"clk1" signal is set to high voltage; nb_multiplier counter is set to zero;
	Continued on next page

Table 5.1 – continued from previous page

clock cycle#	description
	reg2 counter increased by 1; reg1 and reg2 load the data from output_reg; in reg1, R0 pass its previous value to R1; "ctrl" signal jump to its opposite value; input_reg load two new operands from MUX and reg2. $rst <= 0$, $rdy <= 0$, $ctrl <= 1$, $clk1 <= 1$ $reg2/count <= 1$, $nb_multiplier/count <= 0$ $reg_out0 <= 163'h0000000100020000000000000000000000002001$ $reg_out1 <= 163'h0000000000000000000000000000000000000001$ $a_in <= 163'h0000000000000000000000000000000000000001$ $b_in <= 163'h0000000200040000000000000000000000004002$ $inverse_alpha <= 163'h000\ldots000000$
# 23 - # 197	the system will repeat the operation from clock cycle # 1 to # 22.
# 198	"rdy" signal jump to logic one; "inverse_alpha" output the final result; all the modules are disabled until the next reset signal comes; $rst <= 0$, $rdy <= 1$ $reg2/count <= 9$, $nb_multiplier/count <= 0$ $inverse_alpha <=$ $163'h247b6d09c86737e79f68bb3a908196b768a7b1203$

See Fig 5.8 for the simulation results. Vector signal "alpha" is the input 163-bit data, and "inverse_alpha" is the output data of the generator. In this simulation, we use $alpha <= 163'h000000\ldots000000001$.

5.2.2 Compilation Results

Fig 5.9 is the RTL of the design, and Fig 5.10 technology map view of the design. Note that we have combine the REG2 and the 2^x-power module into one shift register. From

Fig 5.9, we can see that the input normal basis element ”alpha” is loaded into REG1 and REG2, respectively. REG1 send the data into the multiplexer and at the same moment REG2 does a cyclic shifting operation. Then the NB-multipleir get the two operands from both multiplexer and REG2 and calculate the product digit-by-digit.

Table 5.2: Cells usage of compilation

Total logic elements	3944
Total registers	1154
Total pins	329
≤2-input logic unit	184
3-input logic unit	507
4-input logic unit	2609

Table 5.3: Area cost of each module

module	logic combinational functions	register
reg1	164	327
reg2	663	331
input_reg	342	333
multiplier	2131	0
output_reg	0	163
top-level	3300	1154

Table 5.4: Operation delay of the design Inverse Generator over $GF(2^{163})$

Clock setup	130.28 MHz $(period = 7.676\ ns)$
Clock period	7.676 ns
Number of cycles for one inversion	198
Total time for one inversion	1519.848 ns

Table 5.2, Table 5.3, and Table 5.4 present the cell usage, gate count of the each module and time delay of the design normal basis inverse generator, respectively. Note that the data of Clock Setup is the maximum clock frequency the system could reach.

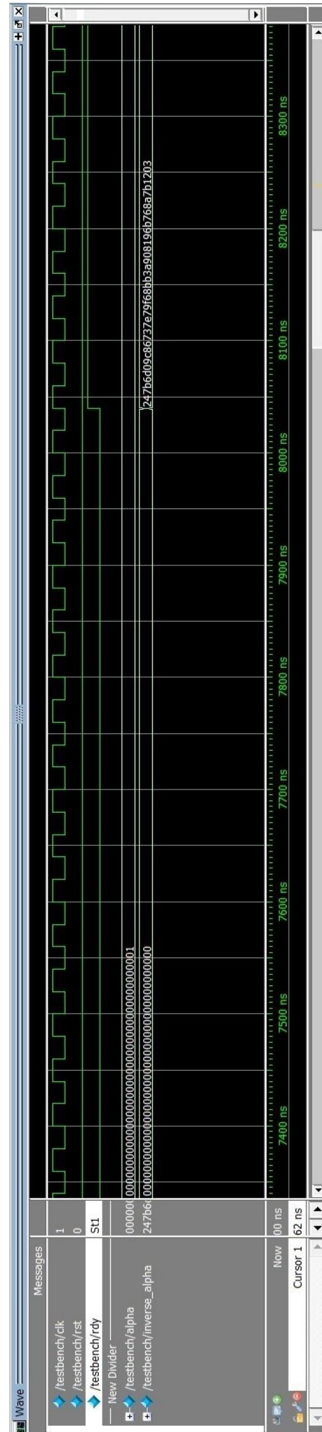

Figure 5.8: Simulation result of the Inverse Generator

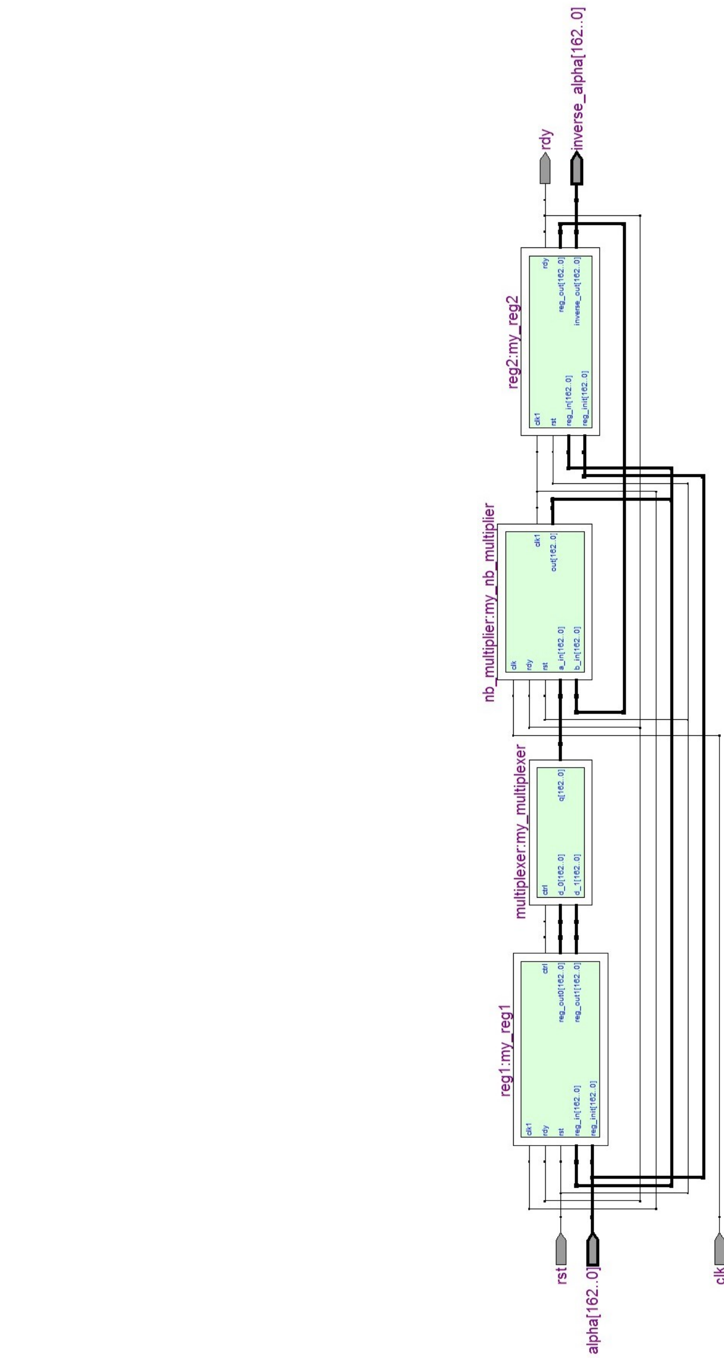

Figure 5.9: RTL of the design

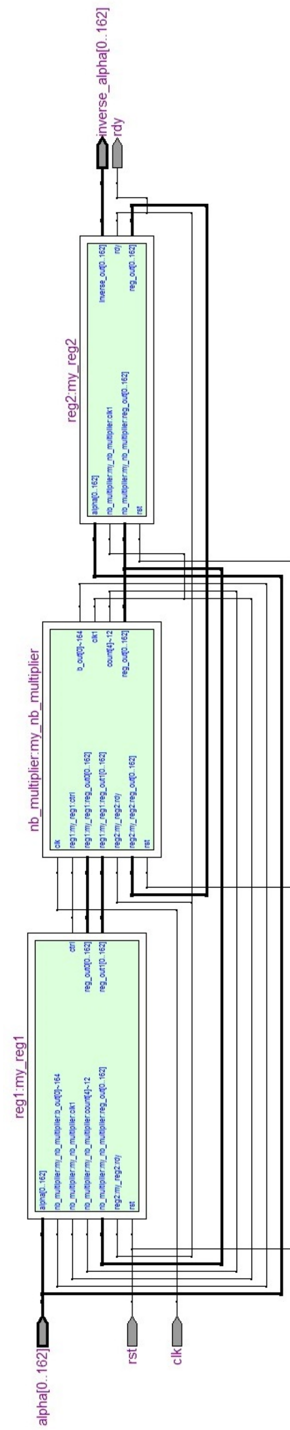

Figure 5.10: Technology map viewer of the design

Chapter 6

Conclusions

Cipher algorithms, especially for public key systems, are demanded for short key size as well as fast processing speed with a high secure level due to the widely application on small portable electronic devices, such as mobile phones, pads, and embedded systems, also the increasing secure threat to the personal privacy plays a not negligible role. In this case, Elliptic Curve Cryptosystems is studied extensively, since it seems the only suitable public key cryptosystem by now. The study shows that the processing speed of elliptic curve based cryptosystem is one of the bottleneck to implement fast ECC encryption/decryption, in addition, field multiplication and field inversion are the two basic operations involved in ECC. As the result of this situation, to speedup finite field computations could efficiently speed up ECC algorithms.

In this thesis, a brief introduction of cryptography is provided in the first chapter. Then, the mathematical backgrounds of finite field, Montgomery multiplications, field inversion and the concept of elliptic curve encryption/decryption are included in Chapter 2. After these, in Chapter 3 we have a brief review of the existing field multipliers, including bit-serial, bit-parallel, digit-level and systolic style architectures. In Chapter 4 we reported a digit-serial MSD-first and a LSD-first Montgomery multiplications, as well as their architectures and FPGA implementations. In Chapter 5, we reported a FPGA implementation of finite field inverse generator using normal basis.

For the proposed Montgomery multiplication, we have provided the architectures for different value of the Montgomery factor: $R(x) = x^u$ and $R(x) = x^{sl}$. The main contribution to the Montgomery multiplication is that we proposed two classes of finite fields $GF(2^m)$

for the multipliers with much reduced critical path delay. By applying the special fields, the time delay of reduction operation can be reduced to one T_X. The FPGA implementations of the proposed architectures are presented for the field $GF(2^{233})$ with digit size $d = 8$ to further verify the correctness of it.

In Chapter 5 of this thesis, we provide an FPGA implementation of a novel finite field $GF(2^{163})$ inversion algorithm using normal basis. This architecture involves two registers, one multiplexer and one normal basis multiplier core and we used a digit-serial architecture to implement this multiplier core.

In the future study, how to apply these fast finite field operation architectures to the higher level computation of ECC in point scalar multiplication is still a critical problem for fast ECC algorithms processing. And how to take the advantages of Montgomery reduction or Montgomery multiplication method on efficient implementation of point addition and point doubling operations will be the next goal of our work.

Appendix A

C-code of $F(s)$ and the First Coordinate c_0 Generation

```
#include <stdio.h>
#define N 1000
#define M 300
FILE *fp;
FILE *fpt;
FILE *fptfp;
int count=0;

main(){
  void write_result(int a, int b);
  void write_result1(int array_FN[N], int p_1);
  int finite_field_exp(int p_0, int Type);

  int w=1;
  int i, j, o, n, Fn, T, p, m, u, k;
  int array_Fn[N];
  int c_0[M][M]={0};

  fptfp=fopen("c_0.txt", "w");
```

```
  fpt=fopen("F(n)value.txt", "w");
  fp=fopen("F(n).txt", "w");
  printf("input the field size m=");
  scanf("%d", &m);
  printf("input the ONB type T=");
  scanf("%d", &T);

  p=T*m+1;
  u=finite_field_exp(p,T);

  printf("%d", u);
  for(j=0;j<T;j++){
    n=w;
    for(i=0;i<m;i++){
      Fn=i;
      write_result(n,Fn);
      array_Fn[n]=Fn;
      n=2*n%p;
    }
    w=u*w%p;
  }

  write_result1(array_Fn,p);

/* J generator */
  if(T%2!=0){
    for(k=1;k<=m/2;k++){
      if(c_0[k-1][m/2+k-1]==0)
        c_0[k-1][m/2+k-1]+=1;
      else
        c_0[k-1][m/2+k-1]-=1;
      if(c_0[m/2+k-1][k-1]==0)
```

```
            c_0[m/2+k-1][k-1]+=1;
         else
            c_0[m/2+k-1][k-1]-=1;
      }
   }

/* generate c_0 */
   for(k=1;k<=(p-2);k++){
      if(c_0[array_Fn[k+1]][array_Fn[p-k]]==0)
         c_0[array_Fn[k+1]][array_Fn[p-k]]+=1;
      else
         c_0[array_Fn[k+1]][array_Fn[p-k]]-=1;
      array_Fn[k+1], array_Fn[p-k]);
   }
   for(o=0;o<m;o++){
      fprintf(fptfp, "assign c[%d] =", o);
      for(i=0;i<m;i++){
         k=0;
         fprintf(fptfp, "(a[%d] & (", (i+o)%m);
         for(j=0;j<m;j++){
            if(c_0[i][j]!=0){
               if(k==0)
                  fprintf(fptfp, "b[%d]", (j+o)%m);
               else
                  fprintf(fptfp, " ^ b[%d]", (j+o)%m);
               k++;
            }
         }
         fprintf(fptfp, "))");
         if(i!=m-1)
            fprintf(fptfp, " ^ ");
         else fprintf(fptfp, ";");
```

```
    }
    fprintf(fptfp, "\n\n");
  }
  fclose(fptfp);
  fclose(fpt);
  fclose(fp);
}

void write_result(int a, int b){
  fprintf(fpt, "F(%3d)=%3d,␣␣␣␣", a, b);
  if(count%10==9)
    fprintf(fpt, "\n");
  count++;
}

void write_result1(int array_FN[N], int p_1){
  int i;
  int temp=0;
  for(i=1;i<p_1;i++){
    fprintf(fp, "F(%3d)=%3d,", i, array_FN[i]);
    if(temp%10==9)
      fprintf(fp, "\n");
    temp++;
  }
}

int finite_field_exp(int p_0, int Type){
  int i, n;
  int g=2;
  int k=1;

  while(k%Type!=0 || g<p_0){
```

```
        k=1;
        n=g;
        while(n>1){
            n=n*g%p_0;
            k++;
        }
        g++;
    }
    g--;
    n=g;
    for(i=1;i<(k/Type);i++){
        n=n*g%p_0;
    }
    return(n);
}
```

Appendix B

Generated VerilogHDL-code of the First Coordinate c_0

```
assign c[0] =(a[0] & (b[1]))
 ^ (a[1] & (b[0] ^ b[13] ^ b[117] ^ b[132]))
 ^ (a[2] & (b[92] ^ b[111] ^ b[117] ^ b[145]))
 ^ (a[3] & (b[9] ^ b[71] ^ b[89] ^ b[125]))
 ^ (a[4] & (b[40] ^ b[87] ^ b[99] ^ b[137]))
 ^ (a[5] & (b[17] ^ b[60] ^ b[105] ^ b[121]))
 ^ (a[6] & (b[22] ^ b[134] ^ b[136] ^ b[160]))
 ^ (a[7] & (b[21] ^ b[43] ^ b[58] ^ b[90]))
 ^ (a[8] & (b[33] ^ b[61] ^ b[124] ^ b[139]))
 ^ (a[9] & (b[3] ^ b[20] ^ b[73] ^ b[93]))
 ^ (a[10] & (b[35] ^ b[63] ^ b[77] ^ b[137]))
 ^ (a[11] & (b[54] ^ b[101] ^ b[130] ^ b[154]))
 ^ (a[12] & (b[110] ^ b[131] ^ b[158] ^ b[162]))
 ^ (a[13] & (b[1] ^ b[51] ^ b[82] ^ b[83]))
 ^ (a[14] & (b[54] ^ b[101] ^ b[111] ^ b[156]))
 ^ (a[15] & (b[60] ^ b[121] ^ b[128] ^ b[129]))
 ^ (a[16] & (b[37] ^ b[59] ^ b[64] ^ b[157]))
 ^ (a[17] & (b[5] ^ b[55] ^ b[79] ^ b[88]))
 ^ (a[18] & (b[20] ^ b[93] ^ b[96] ^ b[136]))
```

```
^ (a[19] & (b[72] ^ b[74] ^ b[107] ^ b[135]))
^ (a[20] & (b[9] ^ b[18] ^ b[89] ^ b[140]))
^ (a[21] & (b[7] ^ b[63] ^ b[135] ^ b[147]))
^ (a[22] & (b[6] ^ b[45] ^ b[61] ^ b[68]))
^ (a[23] & (b[43] ^ b[98] ^ b[119] ^ b[141]))
^ (a[24] & (b[32] ^ b[109] ^ b[115] ^ b[126]))
^ (a[25] & (b[85] ^ b[91] ^ b[153] ^ b[155]))
^ (a[26] & (b[30] ^ b[36] ^ b[84] ^ b[133]))
^ (a[27] & (b[33] ^ b[45] ^ b[61] ^ b[113]))
^ (a[28] & (b[47] ^ b[49] ^ b[57] ^ b[76]))
^ (a[29] & (b[35] ^ b[63] ^ b[74] ^ b[135]))
^ (a[30] & (b[26] ^ b[56] ^ b[86] ^ b[122]))
^ (a[31] & (b[32] ^ b[113] ^ b[114] ^ b[126]))
^ (a[32] & (b[24] ^ b[31] ^ b[44] ^ b[123]))
^ (a[33] & (b[8] ^ b[27] ^ b[44] ^ b[85]))
^ (a[34] & (b[49] ^ b[93] ^ b[134] ^ b[136]))
^ (a[35] & (b[10] ^ b[29] ^ b[50] ^ b[94]))
^ (a[36] & (b[26] ^ b[101] ^ b[156] ^ b[159]))
^ (a[37] & (b[16] ^ b[61] ^ b[68] ^ b[124]))
^ (a[38] & (b[41] ^ b[79] ^ b[146] ^ b[150]))
^ (a[39] & (b[47] ^ b[76] ^ b[98] ^ b[141]))
^ (a[40] & (b[4] ^ b[72] ^ b[107] ^ b[149]))
^ (a[41] & (b[38] ^ b[71] ^ b[105] ^ b[125]))
^ (a[42] & (b[47] ^ b[57] ^ b[97] ^ b[142]))
^ (a[43] & (b[7] ^ b[23] ^ b[147] ^ b[152]))
^ (a[44] & (b[32] ^ b[33] ^ b[67] ^ b[113]))
^ (a[45] & (b[22] ^ b[27] ^ b[134] ^ b[148]))
^ (a[46] & (b[47] ^ b[48] ^ b[97] ^ b[141]))
^ (a[47] & (b[28] ^ b[39] ^ b[42] ^ b[46]))
^ (a[48] & (b[46] ^ b[72] ^ b[135] ^ b[147]))
^ (a[49] & (b[28] ^ b[34] ^ b[80] ^ b[106]))
^ (a[50] & (b[35] ^ b[77] ^ b[81] ^ b[94]))
```

```
^ (a[51] & (b[13] ^ b[111] ^ b[117] ^ b[156]))
^ (a[52] & (b[54] ^ b[66] ^ b[103] ^ b[130]))
^ (a[53] & (b[65] ^ b[144] ^ b[153] ^ b[155]))
^ (a[54] & (b[11] ^ b[14] ^ b[52] ^ b[78]))
^ (a[55] & (b[17] ^ b[121] ^ b[144] ^ b[158]))
^ (a[56] & (b[30] ^ b[75] ^ b[96] ^ b[133]))
^ (a[57] & (b[28] ^ b[42] ^ b[106] ^ b[114]))
^ (a[58] & (b[7] ^ b[63] ^ b[99] ^ b[137]))
^ (a[59] & (b[16] ^ b[124] ^ b[128] ^ b[129]))
^ (a[60] & (b[5] ^ b[15] ^ b[112] ^ b[138]))
^ (a[61] & (b[8] ^ b[22] ^ b[27] ^ b[37]))
^ (a[62] & (b[73] ^ b[76] ^ b[98] ^ b[146]))
^ (a[63] & (b[10] ^ b[21] ^ b[29] ^ b[58]))
^ (a[64] & (b[16] ^ b[68] ^ b[122] ^ b[154]))
^ (a[65] & (b[53] ^ b[88] ^ b[104] ^ b[127]))
^ (a[66] & (b[52] ^ b[108] ^ b[112] ^ b[138]))
^ (a[67] & (b[44] ^ b[85] ^ b[123] ^ b[153]))
^ (a[68] & (b[22] ^ b[37] ^ b[64] ^ b[160]))
^ (a[69] & (b[104] ^ b[119] ^ b[143] ^ b[150]))
^ (a[70] & (b[79] ^ b[88] ^ b[104] ^ b[150]))
^ (a[71] & (b[3] ^ b[41] ^ b[73] ^ b[146]))
^ (a[72] & (b[19] ^ b[40] ^ b[48] ^ b[97]))
^ (a[73] & (b[9] ^ b[62] ^ b[71] ^ b[80]))
^ (a[74] & (b[19] ^ b[29] ^ b[77] ^ b[94]))
^ (a[75] & (b[56] ^ b[92] ^ b[140] ^ b[145]))
^ (a[76] & (b[28] ^ b[39] ^ b[62] ^ b[80]))
^ (a[77] & (b[10] ^ b[50] ^ b[74] ^ b[107]))
^ (a[78] & (b[54] ^ b[103] ^ b[111] ^ b[145]))
^ (a[79] & (b[17] ^ b[38] ^ b[70] ^ b[105]))
^ (a[80] & (b[49] ^ b[73] ^ b[76] ^ b[93]))
^ (a[81] & (b[50] ^ b[94]))
^ (a[82] & (b[13] ^ b[132]))
```

```
^ (a[83] & (b[13] ^ b[132] ^ b[156] ^ b[159]))
^ (a[84] & (b[26] ^ b[101] ^ b[122] ^ b[154]))
^ (a[85] & (b[25] ^ b[33] ^ b[67] ^ b[139]))
^ (a[86] & (b[30] ^ b[96] ^ b[136] ^ b[160]))
^ (a[87] & (b[4] ^ b[115] ^ b[126] ^ b[149]))
^ (a[88] & (b[17] ^ b[65] ^ b[70] ^ b[144]))
^ (a[89] & (b[3] ^ b[20] ^ b[108] ^ b[118]))
^ (a[90] & (b[7] ^ b[99] ^ b[152] ^ b[161]))
^ (a[91] & (b[25] ^ b[110] ^ b[131] ^ b[139]))
^ (a[92] & (b[2] ^ b[75] ^ b[95] ^ b[133]))
^ (a[93] & (b[9] ^ b[18] ^ b[34] ^ b[80]))
^ (a[94] & (b[35] ^ b[50] ^ b[74] ^ b[81]))
^ (a[95] & (b[92] ^ b[117] ^ b[132] ^ b[159]))
^ (a[96] & (b[18] ^ b[56] ^ b[86] ^ b[140]))
^ (a[97] & (b[42] ^ b[46] ^ b[72] ^ b[149]))
^ (a[98] & (b[23] ^ b[39] ^ b[62] ^ b[151]))
^ (a[99] & (b[4] ^ b[58] ^ b[90] ^ b[115]))
^ (a[100] & (b[110] ^ b[121] ^ b[129] ^ b[158]))
^ (a[101] & (b[11] ^ b[14] ^ b[36] ^ b[84]))
^ (a[102] & (b[110] ^ b[124] ^ b[129] ^ b[139]))
^ (a[103] & (b[52] ^ b[78] ^ b[108] ^ b[118]))
^ (a[104] & (b[65] ^ b[69] ^ b[70] ^ b[120]))
^ (a[105] & (b[5] ^ b[41] ^ b[79] ^ b[112]))
^ (a[106] & (b[49] ^ b[57] ^ b[134] ^ b[148]))
^ (a[107] & (b[19] ^ b[40] ^ b[77] ^ b[137]))
^ (a[108] & (b[66] ^ b[89] ^ b[103] ^ b[125]))
^ (a[109] & (b[24] ^ b[120] ^ b[123] ^ b[161]))
^ (a[110] & (b[12] ^ b[91] ^ b[100] ^ b[102]))
^ (a[111] & (b[2] ^ b[14] ^ b[51] ^ b[78]))
^ (a[112] & (b[60] ^ b[66] ^ b[105] ^ b[125]))
^ (a[113] & (b[27] ^ b[31] ^ b[44] ^ b[148]))
^ (a[114] & (b[31] ^ b[57] ^ b[142] ^ b[148]))
```

```
^ (a[115] & (b[24] ^ b[87] ^ b[99] ^ b[161]))
^ (a[116] & (b[144] ^ b[155] ^ b[158] ^ b[162]))
^ (a[117] & (b[1] ^ b[2] ^ b[51] ^ b[95]))
^ (a[118] & (b[89] ^ b[103] ^ b[140] ^ b[145]))
^ (a[119] & (b[23] ^ b[69] ^ b[151] ^ b[152]))
^ (a[120] & (b[104] ^ b[109] ^ b[127] ^ b[143]))
^ (a[121] & (b[5] ^ b[15] ^ b[55] ^ b[100]))
^ (a[122] & (b[30] ^ b[64] ^ b[84] ^ b[160]))
^ (a[123] & (b[32] ^ b[67] ^ b[109] ^ b[127]))
^ (a[124] & (b[8] ^ b[37] ^ b[59] ^ b[102]))
^ (a[125] & (b[3] ^ b[41] ^ b[108] ^ b[112]))
^ (a[126] & (b[24] ^ b[31] ^ b[87] ^ b[142]))
^ (a[127] & (b[65] ^ b[120] ^ b[123] ^ b[153]))
^ (a[128] & (b[15] ^ b[59] ^ b[138] ^ b[157]))
^ (a[129] & (b[15] ^ b[59] ^ b[100] ^ b[102]))
^ (a[130] & (b[11] ^ b[52] ^ b[138] ^ b[157]))
^ (a[131] & (b[12] ^ b[91] ^ b[155] ^ b[162]))
^ (a[132] & (b[1] ^ b[82] ^ b[83] ^ b[95]))
^ (a[133] & (b[26] ^ b[56] ^ b[92] ^ b[159]))
^ (a[134] & (b[6] ^ b[34] ^ b[45] ^ b[106]))
^ (a[135] & (b[19] ^ b[21] ^ b[29] ^ b[48]))
^ (a[136] & (b[6] ^ b[18] ^ b[34] ^ b[86]))
^ (a[137] & (b[4] ^ b[10] ^ b[58] ^ b[107]))
^ (a[138] & (b[60] ^ b[66] ^ b[128] ^ b[130]))
^ (a[139] & (b[8] ^ b[85] ^ b[91] ^ b[102]))
^ (a[140] & (b[20] ^ b[75] ^ b[96] ^ b[118]))
^ (a[141] & (b[23] ^ b[39] ^ b[46] ^ b[147]))
^ (a[142] & (b[42] ^ b[114] ^ b[126] ^ b[149]))
^ (a[143] & (b[69] ^ b[120] ^ b[152] ^ b[161]))
^ (a[144] & (b[53] ^ b[55] ^ b[88] ^ b[116]))
^ (a[145] & (b[2] ^ b[75] ^ b[78] ^ b[118]))
^ (a[146] & (b[38] ^ b[62] ^ b[71] ^ b[151]))
```

```
^ (a[147] & (b[21] ^ b[43] ^ b[48] ^ b[141]))
^ (a[148] & (b[45] ^ b[106] ^ b[113] ^ b[114]))
^ (a[149] & (b[40] ^ b[87] ^ b[97] ^ b[142]))
^ (a[150] & (b[38] ^ b[69] ^ b[70] ^ b[151]))
^ (a[151] & (b[98] ^ b[119] ^ b[146] ^ b[150]))
^ (a[152] & (b[43] ^ b[90] ^ b[119] ^ b[143]))
^ (a[153] & (b[25] ^ b[53] ^ b[67] ^ b[127]))
^ (a[154] & (b[11] ^ b[64] ^ b[84] ^ b[157]))
^ (a[155] & (b[25] ^ b[53] ^ b[116] ^ b[131]))
^ (a[156] & (b[14] ^ b[36] ^ b[51] ^ b[83]))
^ (a[157] & (b[16] ^ b[128] ^ b[130] ^ b[154]))
^ (a[158] & (b[12] ^ b[55] ^ b[100] ^ b[116]))
^ (a[159] & (b[36] ^ b[83] ^ b[95] ^ b[133]))
^ (a[160] & (b[6] ^ b[68] ^ b[86] ^ b[122]))
^ (a[161] & (b[90] ^ b[109] ^ b[115] ^ b[143]))
^ (a[162] & (b[12] ^ b[116] ^ b[131] ^ b[162]));
```

Bibliography

[1] D. Froomkin, *Deciphering Encryption*, Washington Post, May 1998, Retrieved 18 September 2013.

[2] D. Crawford and M. Esterl, “At Siemens, witnesses cite pattern of bribery”, *The Wall Street Journal*, January 31, 2007.

[3] D. Kahn, *The Codebreakers: The Story of Secret Writing*, Rev Sub. Scribner, 1996.

[4] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thom, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev and P. Zimmermann, “Factorization of a 768-bit RSA modulus”, *Cryptology ePrint Archive: Report 2010/006*, http://eprint.iacr.org/2010/006, February 2010.

[5] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra and P. L. Montgomery, “PlayStation 3 Computing Breaks 2^{60} Barrier 112-bit Prime ECDLP Solved”, http://lacal.epfl.ch/112bit_prime, 2009.

[6] W. Diffie and M. E. Hellman, “New Directions in Cryptography”, *IEEE Transactions on Information Theory*, Vol. 22, pp. 644-654, 1976.

[7] R. Rivest, A. Shamir and L. Adleman, “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”, *Comm. ACM*, Vol. 21, No. 2, pp. 120-126, February 1978.

[8] N. Koblitz, “Elliptic Curve Cryptosystem”, *Math. Comp.*, Vol. 48, pp. 203-209, 1987.

[9] V. Miller, “Uses of Elliptic Curves in Cryptography”, *Advances in Cryptology: Proceedings of CRYPTO’85*, H. C. Williams, Ed. 1985, No. 218 in Lecture Notes in Computer Science, pp. 417-426, Springer-Verlag.

[10] P. L. Montgomery, “Modular Multiplication Without Trial Division”, Mathematics of Computation, Vol. 44, No. 170, pp. 519-521, 1985.

[11] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, 2nd Edition, New York, 1994.

[12] C. K. Koc and T. Acar, “Montgomery Multiplication in $GF(2^k)$”, *Designs, Codes and Cryptography*, Vol. 14, pp. 57-69, 1998.

[13] L. Song and K. Parhi, “Low-energy digit-serial/parallel finite field multipliers”, *The Journal of VLSI Signal Processing*, Vol. 19, No. 2, pp. 149-166, 1998.

[14] ”Standard Specifications for Public Key Cryptography, Annex A (informative) Number-Theoretic Background,” *IEEE P1363 / D9 (Draft Version 9),* pp. 76-172, February 1999.

[15] H. Wu, “Montgomery Multiplier and Squarer for a Class of Finite Fields”, *IEEE Transactions on Computers*, Vol. 51, No. 5, pp. 521-529, May 2002.

[16] N. Mentens, S. B. Ors, B. Preneel and J. Vandewalle, “AN FPGA IMPLEMENTATION OF A MONTGOMERY MULTIPLIER OVER $GF(2^M)$”, *Computing and Informatics*, Vol. 23, pp. 487-499, 2004.

[17] W. Tang, H. Wu and M. Ahmadi, “VLSI implementation of bit-parallel word-serial multiplier in $GF(2^{233})$”, *IEEE-NEWCAS Conference, 2005. The 3rd International*, pp. 399-402, June 2005.

[18] C. Lee, J. Horng, I, Jou and E. Lu, “Low-complexity Bit Parallel Systolic Montgomery Multipliers for Special Classes of $GF(2^m)$”, *Transactions on Computers*, Vol. 54, No. 9, pp. 1061-1070, 2005.

[19] S. Kumar, T. Wollinger and C. Paar, “Optimum Digit Serial $GF(2^m)$ Multipliers for Curve-based Cryptography”, *IEEE TRANSACTIONS ON COMPUTERS*, Vol. 55, No. 10, pp. 1306-1311, October 2006.

[20] C. Y. Lee, C. W. Chiou, J. M. Lin and C. C. Chang, “Scalable and Systolic Montgomery Multiplier over $GF(2^m)$ Generated by Trinomials”, *IET Circuits Devices Syst*, Vol. 1, No. 6, pp. 477-484, 2007.

[21] B. A. Forouzan, *Introduction to Cryptography and Network Security*, McGraw-Hill, 1st edition, 2008.

[22] A. P. Fournaris and O. Koufopavlou, "Versatile Multiplier Architecture in $GF(2^k)$ Fields Using the Montgomery Multiplication Algorithm", *INTEGRATION, the VLSI journal*, Vol. 41, pp. 371-384, 2008.

[23] B. King, "Mapping an Arbitrary Message to an Elliptic Curve when Defined over $GF(2^n)$", *International Journal of Network Security*, Vol. 8, No. 2, pp. 169-176, March 2009.

[24] P. K. Meher, "On Efficient Implementation of Accumulation in Finite Field Over $GF(2^m)$ and its Applications", *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, Vol. 17, No. 4, pp. 541-550, April 2009.

[25] A. Hariri and A. R. Masoleh, "Bit-Serial and Bit-Parallel Montgomery Multiplication and Squaring over $GF(2^m)$", *IEEE Transactions on Computers*, Vol. 58, No. 10, pp, 1332-1345, October 2009.

[26] P. Bh, D. Chandravathi and P. P. Roja, "Encoding and Decoding of a Message in the Implementation of Elliptic Curve Cryptography Using Koblitz's Method", *International Journal on Computer Science and Engineering*, Vol. 2, No. 5, pp. 1904-1907, November 2010.

[27] S. Talapatra, H. Rahaman and J. mathew, "Low Complexity Digit Serial Systolic Montgomery Multiplier for Special Class of $GF(2^m)$", *Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 18, No. 5, pp. 847-852, May 2010.

[28] M. Morales-Sandoval, C. Feregrino-Uribe and P. Kitsos, "Bit-serial and Digit-serial $GF(2^m)$ Montgomery Multipliers Using Linear Feedback Shift Registers", *Computers & Digital Techniques, IET*, Vol. 5, No. 2, pp. 86-94, 2011.

[29] *0.18μm TSMC CMOS Technology*, Standard Cell Library, September 1999, available through Canadian Microelectronics Corporation.

[30] H. Wu, *Personal communication*, Dept. of E&CE, U. of Windsor, 2012.

[31] R. Amiri and O. Elkeelany, “Concurrent Reconfigurable Architecture for Mapping and Encrypting a Message in Elliptic Curve Cryptography”, *IEEE SoutheastCon 2013 conference*, pp. 1-6, April 2013.

Vita Auctoris

NAME:	Wangchen DAI
PLACE OF BIRTH:	Handan, Heibei, P.R.China
YEAR OF BIRTH:	1988
EDUCATION:	Beijing Institute of Technology, B.Sc., Beijing, P.R.China, 2010
	University of Windsor, M.A.Sc., Windsor, ON, CANADA, 2013