國立交通大學

電子工程學系 電子研究所碩士班

碩士論文

抵抗簡單能量攻擊法的橢圓曲線運算單元之設計與實現

Design and Implementation of an SPA-Resistant Dual-Field Elliptic Curve Arithmetic Unit

研究生：曾知業

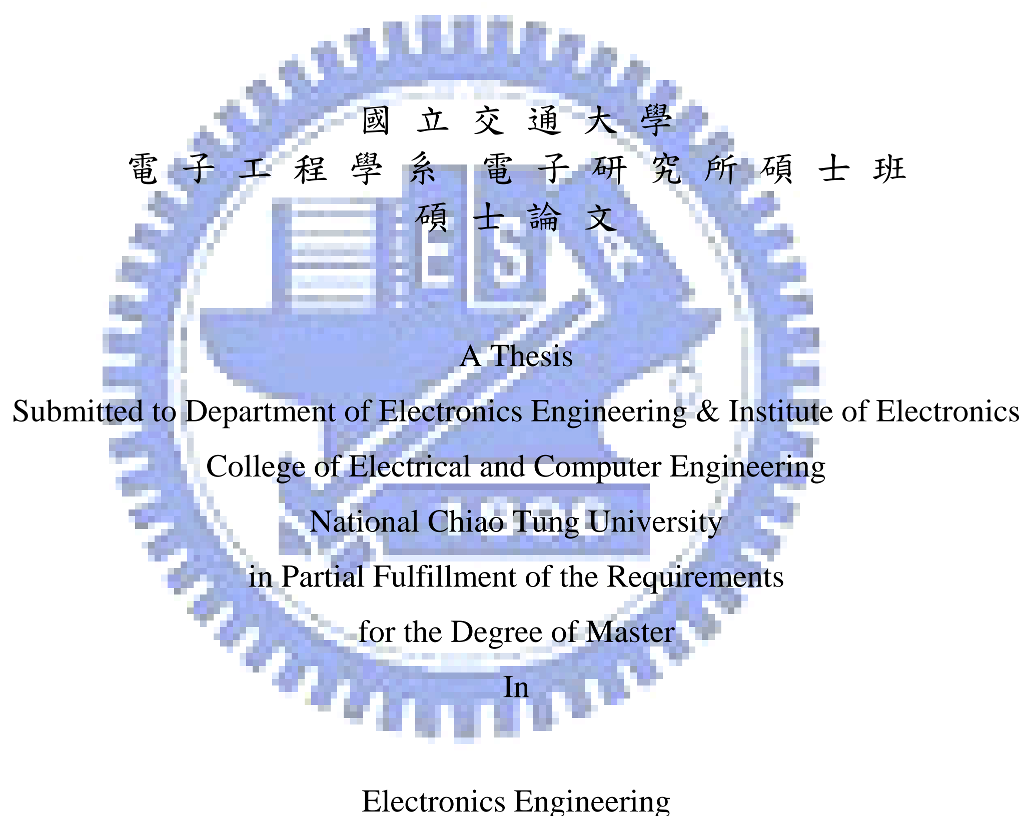
指導教授：張錫嘉 教授

中華民國九十七年十一月

抵抗簡單能量攻擊法的橢圓曲線運算單元之設計與實現
Design and Implementation of an SPA-Resistant Dual-Field Elliptic Curve Arithmetic Unit

研 究 生：曾知業　　Student：Chih-Yeh Tseng

指導教授：張錫嘉　　Advisor：Hsie-Chia Chang



國 立 交 通 大 學

電 子 工 程 學 系　電 子 研 究 所 碩 士 班

碩 士 論 文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of Master

In

Electronics Engineering

November 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年十一月

抵抗簡單能量攻擊法的橢圓曲線運算單元之設計與實現

學生：曾知業　　　　　　　　　　　　　　　　　　　　　指導教授：張錫嘉

國立交通大學電子工程學系 電子研究所碩士班

1896

摘　　要

這篇論文中介紹了一個同時適用在 *GF(p)*和 $GF(2^m)$的抗簡單能量攻擊法之橢圓曲線運算單元(ECAU)的通用型硬體架構，這個架構能支援最多 512 位元任意長度的有限場。在這個運算單元中提出一種隨機交錯計算 $k_1P_1+k_2P_2$ 的演算法，藉此抵抗簡單能量攻擊法。其中的橢圓曲線運算建構在仿射座標系，並使用高速的蒙哥馬利除法演算法。為了減少硬體複雜度，我們提出了有限場運算單元(GFAU)來計算同餘加法、減法和蒙哥馬利乘法、除法。

使用 ASIC 設計流程實現這個架構後，GFAU 所需的合成邏輯閘個數比先人所提出的少 25%。在所提出的抵抗簡單能量攻擊法的 ECAU 中，我們只運用一套 GFAU，因此合成結果只需 277.K 個邏輯閘。在 133MHz 的時脈下進行，計算一筆 512 位元的橢圓曲線純量乘法平均需要 13.76ms，而計算一筆抵抗簡單能量攻擊法的 $k_1P_1+k_2P_2$ 運算只需要 27.53ms。

Design and Implementation of an SPA-Resistant Dual-Field Elliptic Curve Arithmetic Unit

student：Chih-Yeh Tseng Advisors：Hsie-Chia Chang

Department of Electronics Engineering & Institute of Electronics
National Chiao Tung University

ABSTRACT

A universal hardware architecture of SPA-resistant elliptic curve arithmetic unit (ECAU) suitable for both *GF(p)* and $GF(2^m)$ is introduced to work in arbitrary field lengths within a maximum 512-bit length. The proposed algorithm used in ECAU can randomly interleave $k_1P_1+k_2P_2$ operations to cope with SPA. The elliptic curve operations are calculated over affine coordinate using high speed Montgomery division algorithm. To reduce hardware complexity, the sharing architecture called Galois field arithmetic unit (GFAU) is proposed to perform modular addition, modular subtraction, Montgomery multiplication and Montgomery division.

After implemented by ASIC design flow, the GFAU occupies 25% less synthesized gatecount than previous work. With only one set of GFAU, the proposed SPA-resistant ECAU occupies 277.5K gatecount. It averagely takes 13.76ms to perform one 512-bit scalar multiplication and 27.53ms to perform a SPA-resistant 512-bit $k_1P_1+k_2P_2$ operation both at 133MHz clock rate.

誌　謝

說長不長，說短也不短，大約兩年半的研究所生活即將要結束。總結起來，可以說是既充實又快樂的一段時光。首先，我要感謝指導老師：張錫嘉博士，不但在研究領域上很有耐心的指引我們，也鼓勵我們嘗試新的思維，給予學生們相當大的空間發揮。生活上也像是朋友般的關心和照顧，讓實驗室一直保持著和樂融融的氣氛。此外，建青學長帶領 STAR 團隊漸漸的茁壯，研究上遇到許多問題，都是靠學長豐富的經驗才獲得解決，真的非常感謝。感謝 SECURITY 團隊之前的學長們，尤其是乀羅學長，你的研究論文帶領我進入橢圓曲線密碼學的世界，給予我非常多的想法，祝你現在工作順利，賺大錢回來請客。一起努力的 STAR 團隊的成員們，柏均學長統籌研究計畫的大小事，還要幫忙我們解決各種工具軟體的問題，真的很辛苦；人偉幫忙研究 ECC 的 projective coordinate 給了我不少非常有用的資訊，最後還得收拾我在計畫的爛攤子，萬分感謝；一起努力的 Q 毛，都靠你的 word-base 乘法器，計畫才沒有開天窗，也多虧了你帶起的大胃王活動，讓我的肚子越來越大；廷聿和我們一起討論在能量攻擊法，才能夠領悟到一些原本沒有想到的事情；其他的學弟們也不斷的替我加油打氣，很感謝你們。此外也感謝 OASIS 實驗室的所有成員，大頭學長從我剛進實驗室就給我不少指導；國光不厭其煩的替我們解決工作站的問題，真的很辛苦；修齊也幫我們 review 論文和投影片，當然也幫我搶了不少籃板；永裕、企鵝、鑫偉都是一起努力拼畢業的夥伴，不但一起寫論文，也一起吃了很多垃圾食物；學弟妹們也為實驗室帶來許多歡笑，尤其是高手和圓淳，不但在研究上給我不少想法，也是休閒活動上的好伙伴。還要感謝桑老師的同學和學弟們，尤其是北科第一控衛哲聖，總是跟我、永裕、鑫偉和圓淳一起奮戰到深夜。

當然，最需要感謝的還是我的家人。很感謝父母給予我的支持和鼓勵，因為你們一路走來不變的支持和照顧，讓我沒有後顧之憂的完成我的學業，我只有滿心的感謝。遠在美國的哥哥，你在 MSN 上給我的鼓勵，也讓我感到很窩心，相信你也可以在美國順利的完成學業。最後還要特別感謝雯怡，研究所兩年多的生活，有妳的陪伴，讓我度過了研究和生活上的許多挫折。沒有你們就沒有這篇論文，在此再次表達我由衷的感謝。

Design and Implementation of an SPA-Resistant Dual-Field Elliptic Curve Arithmetic Unit

Student: Chih-Yeh Tseng

Advisor: Dr. Hsie-Chia Chang

Department of Electronics Engineering
National Chiao Tung University

Abstract

A universal hardware architecture of SPA-resistant elliptic curve arithmetic unit (ECAU) suitable for both $GF(p)$ and $GF(2^m)$ is introduced to work in arbitrary field lengths within a maximum 512-bit length. The proposed algorithm used in ECAU can randomly interleave $k_1P_1+k_2P_2$ operations to cope with SPA. The elliptic curve operations are calculated over affine coordinate using high speed Montgomery division algorithm. To reduce hardware complexity, the sharing architecture called Galois field arithmetic unit (GFAU) is proposed to perform modular addition, modular subtraction, Montgomery multiplication and Montgomery division.

After implemented by ASIC design flow, the GFAU occupies 25% less synthesized gatecount than previous work. With only one set of GFAU, the proposed SPA-resistant ECAU occupies $277.5K$ gatecount. It averagely takes 13.76ms to perform one 512-bit scalar multiplication and 27.53ms to perform a SPA-resistant 512-bit $k_1P_1+k_2P_2$ operation both at 133MHz clock rate.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Motivation 4

1.3 Thesis Organization 5

2 Elliptic Curves **6**

2.1 Basic Facts 7

2.2 Elliptic Curves Arithmetics over Affine Coordinates 9

2.2.1 Elliptic Curves over the Reals 9

2.2.2 Elliptic Curves over Prime Fields 14

2.2.3 Elliptic Curves over Extension of Binary Fields 14

2.3 Elliptic Curve Arithmetics over Projective Coordinates 16

2.3.1 Homogeneous Projective Coordinates 16

2.3.2 Jacobian Coordinates 17

2.3.3 Chudnovsky-Jacobian Coordinates 18

2.3.4 Modified Jacobian Coordinates 19

2.4 Elliptic Curves Scalar Multiplication 20

2.4.1 Double-and-Add Algorithm 20

2.4.2 Addition-Subtraction Method 21

2.4.3 Binary NAF Method 21

3 Galois Field Arithmetics **23**

3.1 Modular Multiplication 23

3.1.1 Traditional Modular Multiplication Algorithm 23

3.1.2 Montgomery Multiplication Algorithm 24

3.1.3 Modified Montgomery Multiplication Algorithm 26
3.1.4 Integer Domain and Montgomery Domain 27
3.2 Modular Inversion 28
3.2.1 Fermat's Little Theory 28
3.2.2 Extended Euclidean Algorithm 29
3.2.3 Montgomery Modular Inversion Algorithm 33
3.3 Modular Division 35
3.3.1 Multiplication after Inversion 35
3.3.2 Modular Division Algorithm 36
3.3.3 Montgomery Modular Division Algorithm 38
3.4 Domain Transformation 41
3.5 Summary 42

4 Power Analysis **45**
4.1 Simple Power Analysis 45
4.2 Differential Power Analysis 47
4.3 Proposed Countermeasures against SPA 49

5 Proposed Architectures **52**
5.1 Galois Field Arithmetic Unit 52
5.2 Elliptic Curve Scalar Multiplier 58
5.3 SPA-Resistant Elliptic Curve Arithmetic Unit 61

6 Implementation Results **66**
6.1 Design and Test Consideration 66
6.2 Implementation Results and Comparison 67
6.2.1 ASIC Implementation 67
6.2.2 FPGA Implementation 69

7 Conclusion and Disscusion **71**

List of Figures

1.1 Symmetric-key cryptography block diagram 1

1.2 Asymmetric-key cryptography block diagram 2

2.1 Hierarchal organization of elliptic curve cryptography. 6

2.2 Elliptic curves over the reals 10

2.3 Adding two distinct points $P + Q = -R$ 11

2.4 Doubling a point $2P = -R$ 12

4.1 Interleaved scalar multiplication. 51

5.1 Flow chart of the Montgomery division algorithm. 53

5.2 Flow chart of the Montgomery multiplication algorithm. 55

5.3 Finite states transfer chart of the GFAU. 56

5.4 Architecture of the GFAU. 57

5.5 State transition chart of the ECSM. 59

5.6 Pie graph of the area consumption of the ECSM. 60

5.7 Flow chart of the scalar determination scheme. 62

5.8 Pie graph of the area consumption of the ECAU. 63

5.9 Architecture of the proposed ECSM. 64

5.10 Architecture of the proposed ECAU. 65

List of Tables

1.1 Comparable security strength for given cryptography 3

2.1 Point addition formula over reals. 14

2.2 Point addition formula over $GF(p)$ 14

2.3 Point addition formula over $GF(2^m)$ 15

3.1 Latency of the Montgomery modular inverse from and to both domains. . . 35

3.2 Latency of modular division using traditional method from and to both domain 36

3.3 Latency of Montgomery modular division from and to both domain 41

3.4 Latency of Montgomery division algorithms comparison. 41

3.5 Scalar multiplication comparison between different coordinates. 43

3.6 Comparison between different coordinates. 44

5.1 Synthesized results for proposed universal dual-field Galois field arithmetic unit on ASIC and FPGA design. 58

5.2 Synthesize results for proposed ECSM. 61

5.3 Synthesize results for proposed ECAU. 63

6.1 ASIC synthesis results comparison 67

6.2 Elliptic Curve Scalar Multiplication ASIC Performance Comparison 68

6.3 512-bit FPGA synthesis results. 69

6.4 Elliptic Curve Scalar Multiplication FPGA Performance Comparison . . . 70

Chapter 1

Introduction

1.1 Background

The modern cryptography can be roughly split into two kind: symmetric-key cryptography (secret-key cryptography) and asymmetric-key cryptography (public-key cryptography).

Symmetric-key cryptography means that both the sender and receiver share the same key or the decryption key can be easily derived from the encryption key. It can be illustrated in Figure 1.1. This was the only kind of encryption publicly known until June 1976. Now the most popular symmetric-key cryptography algorithm is the Advanced Encryption Standard (AES) [1].

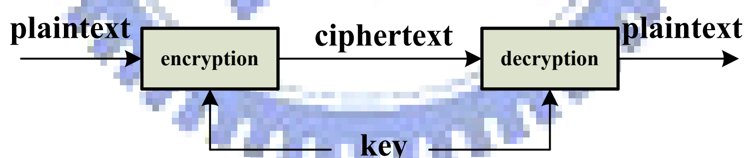


Figure 1.1: Symmetric-key cryptography block diagram

A significant disadvantage of symmetric cryptography is the key management. The sender and the receiver should exchange the same key through a trusted channel. Besides, each distinct pair of communication parties must, ideally, share a different key. The number of keys required increases as the square of the number network members, therefore, complex key management schemes are demanded to keep them all straight and secret.

In 1976, Whitfield Diffe and Martin Hellman [2] proposed a novel cryptography called public-key cryptography (also called as asymmetric-key cryptography), which used discrete logarithm problem to prevent the secret-key from being acquired with known public-key. It can be illustrated in Figure 1.2. This method of exponential key exchange came to be known as Diffie-Hellman key exchange. RSA and El-Gamal are two of the popular public-key cyrptosystems widely used nowadays. The RSA algorithm based on the difficult of factoring large numbers was published by Rivest, Shamir and Adleman [3] at MIT[1] in 1978. Further, the El-Gamal algorithm based on Diffie-Hellman key agreement describes the public-key system and digital signature schemes, and it was proposed by Taher ElGamal [4] in 1985.

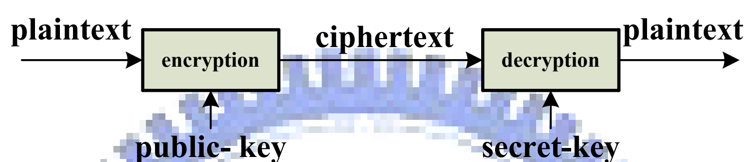


Figure 1.2: Asymmetric-key cryptography block diagram

The public-key cryptosystem such as RSA is still widely used in electronic commerce protocols. However, there are many efficient attacks known for both RSA and modular p discrete log based cryptosystems such as the Number Field Sieve [5] attacks for RSA and the index calculus attacks for the modular p systems. These methods decrease the computational time in attacking, therefore the length of the public-key should be much longer than the secret-key in symmetric cryptography. It is believed to be secure enough as long as it has sufficiently long keys.

The elliptic curve cryptography (ECC) is kind of public-key cryptography based on the algebraic structure of elliptic curves over finite fields. It was independently proposed by Victor S. Miller of IBM[2] in 1986 [6] and Neal Koblitz of the University of Washington in 1987 [7]. There are no subexponential algorithms known for the elliptic curve discrete logarithm problem (ECDLP) and denotes that there are no efficient mathematical attacks known on it. Consequently, the parameters for ECC can be chosen to be much smaller than the parameters for RSA with the same level of resistance against the best known

[1] Massachusetts Institute of Technology, located in Cambridge, MA, USA. http://web.mit.edu/

[2] International Business Machines Corporation. http://www.ibm.com/

attacks. Table 1.1 shows each different parameter size with the same level of security strengths compared with given cryptography [8].

Table 1.1: Comparable security strength for given cryptography

ECC (e.g., ECDSA)	IFC (e.g., RSA)	Symmetric key algorithms
$f = 160 - 223$	$k = 1024$	-
$f = 224 - 255$	$k = 2048$	-
$f = 256 - 383$	$k = 3072$	AES-128
$f = 384 - 511$	$k = 7680$	AES-192
$f = 512 \uparrow$	$k = 15360$	AES-256

[1] ECDSA [9].

[2] IFC denotes integer factorization cryptography.

[3] f is the size of n, where n is the order of the base point G.

[4] k is the size of the modulus p.

Note that in Table 1.1, the difference of the size between ECC and RSA becomes more enormous as the security level increases. It is attractive that the ECC has much smaller parameters leads to more significant performance advantages contrast to RSA. Therefore, the ECC takes advantages for wireless applications where the computing power, memory and battery life are limited such as smart cards and wireless devices.

Another type of attack called side-channel attack is much more efficient over attacks based on mathematical theorem. Among them, power analysis, proposed by Paul Kocher, Joshua Jaffe and Benjamin Jun in 1998 [10], is the most discussed one. This kind of attack can extract cryptographic keys and other secret information from the device without invasion. It works over all kinds of public-key and secret-key cyrptosystems. Most of all, it's the only efficient attack on elliptic curve cryptosystem. Lots of countermeasures were proposed to resist power analysis. To learn more information about the countermeasures, [11] can be referred. A detailed introduction of power analysis attack will be given in chapter 4.

Furthermore, the performance of ECC mainly depends on the efficiency of its modular arithmetics, namely, scalar multiplication. Given a positive integer, and a point P on an

elliptic curve. The scalar multiplication kP can easily be computed by iterative additions and doublings. There are some algorithms to compute the multiple of points on elliptic curves. More details will be discussed in chapter 2.4 later.

1.2 Motivation

The scalar multiplication is the most important operation in an elliptic curve cryptosystem due to the ECDLP. In the traditional affine coordinate elliptic curve point representation, the result of addition and doubling can be derived through several modular multiplication and one modular division. Modular multiplication has been improved by the Montgomery's technique [12] which will be discussed in chapter 3.1.2. Traditionally, modular division can be achieved by modular multiplication after modular inversion. Then modular inversion can be done by iterative modular multiplication introduced in the Fermat's little theorem or by iterative modular addition, subtraction and shifting introduced by extended Euclidean algorithm. But the first one is extremely time-consuming and the second one is area-consuming.

Elliptic curve is not only presented in general affine coordinates, it has some different projective coordinates representations. Equations in elliptic curve scalar multiplication only consists of modular addition and modular multiplication. Though more multiplications are utilized, it only requires one modular multiplier. Therefore, most research in recent years focus on the optimization on the Montgomery multiplication algorithm.

The modular division is further improved and modified with the Montgomery technique by Yao-Jen Liu in 2007 [13]. In his design, existing inversion after multiplication algorithm is replaced by just one Montgomery division. Hence the area and the computational time is reduced, which will be discussed in chapter 3.3.

Therefore, in this thesis, an approach is provided to compute the scalar multiplication on elliptic curves in both $GF(p)$ and $GF(2^m)$, and a unified Montgomery multiplication and division design is proposed to deal with various finite field degrees and different primitive polynomials in $GF(2^m)$. In this way, performance in terms of the area by computational time is near to that of projective coordinates algorithms. Therefore affine coordinates algorithms can be bring back to compete with projective coordinates ones.

1.3 Thesis Organization

In this thesis, a universal dual-field elliptic curve arithmetic unit is proposed. In Chapter 2, the preliminary mathematical background of elliptic curves is introduced. In Chapter 3, the Galois field arithmetics is introduced. The Montgomery technique is also involved to improve the multiplication and the division. In Chapter 4, the power analysis attack and its countermeasures are introduced. In Chapter 5, all the proposed universal dual-field architectures are described. In Chapter 6, it shows the hardware implementation results and test consideration. The conclusion is given in Chapter 7.

Chapter 2

Elliptic Curves

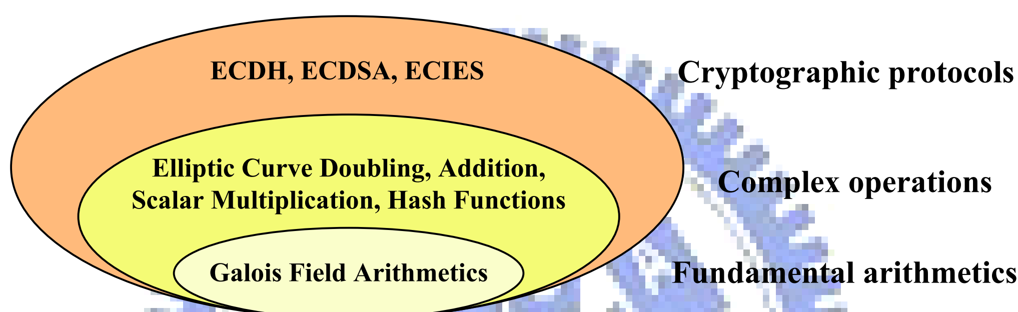


Figure 2.1: Hierarchal organization of elliptic curve cryptography.

The hierarchal organization of elliptic curve cryptography is in Figure 2.1. Galois field arithmetics construct the elliptic curve arithmetics and the elliptic curve arithmetics construct complicated protocols.

Elliptic curves [14] [15] are not ellipses as shown in literal. In mathematics, an elliptic curve is an algebraic curve defined by a cubic equation such as $y^2 = x^3 + ax + b$, which is non-singular, i.e. its graph has no cusps or self-intersections. Elliptic curves received their name from their relation to elliptic integrals such as

$$\int_{z_1}^{z_2} \frac{dx}{\sqrt{x^3 + ax + b}} \quad \text{and} \quad \int_{z_1}^{z_2} \frac{x\, dx}{\sqrt{x^3 + ax + b}} \tag{2.1}$$

that arose in connection with the computation of the circumference of ellipses.

2.1 Basic Facts

Let $\mathbb{F}$ be an algebraically closed field and $\mathbb{F}^2$ denote the affine plane $\mathbb{A}^2$, the usual plane, $\mathbb{A}^2(\mathbb{F}) = \{(x, y)|x, y \in \mathbb{F}\}$. Let $C(x, y)$ be an irreducible polynomial over $\mathbb{F}$, and the curve C means the set of zeros of C in the affine plane $\mathbb{F}^2$, i.e. $\{(x, y) \in \mathbb{F}^2 | C(x, y) = 0\}$. Assume that P is a point (x_p, y_p) on the curve C. If both of the partial derivatives vanish at P, that is $\frac{\partial C(x_p,y_p)}{\partial x} = \frac{\partial C(x_p,y_p)}{\partial y} = 0$, then the point P is called a singular point on the curve C. A curve is called a singular curve if and only if it has at least one singular point on it, otherwise it is called a non-singular curve. An elliptic curve commonly used in cryptography is a non-singular curve because of its better security level relative to a singular curve. A singular elliptic curve is thought of insecure in general. Definition 2.1 shows the algebraic equation of the elliptic curve in a more general form.

Definition 2.1. *An elliptic curve E over the field $\mathbb{F}$ defined by an affine Weierstrass equation is an equation of the form*

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6, \quad \forall a_i \in \mathbb{F} \tag{2.2}$$

let $E(\mathbb{F})$ denote the elliptic curve E over $\mathbb{F}$, i.e. the set of points $(x, y) \in \mathbb{F}^2$ that satisfy this equation, along with the point at infinity denoted by $\mathcal{O}$.

Definition 2.2. *The point at infinity called $\mathcal{O}$ is the intersection of the y-axis and the line at infinity. The line at infinity is the set of points on the projective plane for which $Z = 0$. Therefore, the point at infinity $\mathcal{O}$ is $(0, 1, 0)$ in the projective plane, i.e. the equivalence class with $X = Z = 0$.*

No further details about projective plane are shown in this thesis since only affine coordinates are discussed in the remaining chapters.

In order to describe a singular or non-singular curve clearly, an important quantity Δ related to the elliptic curve called the discriminant of E is defined.

Definition 2.3. *Δ is the discriminant of E and is given by*

$$\Delta = -b_2^2 b_8 - 8b_4^3 - 27b_6^2 + 9b_2 b_4 b_6, \quad where \begin{cases} b_2 &= a_1^2 + 4a_2 \\ b_4 &= 2a_4 + a_1 a_3 \\ b_6 &= a_3^2 + 4a_6 \\ b_8 &= a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 \\ & \quad +a_2 a_3^2 - a_4^2 \end{cases} \tag{2.3}$$

and the symbols above correspond to (2.2).

Theorem 2.1. *A cubic curve defined by a Weierstrass equation (2.2) is singular if and only if its discriminant Δ is zero.*

The Definition 2.1 is feasible for any field $\mathbb{F}$. However, the elliptic curves commonly used in cryptography are over the finite field $GF(q)$, where q is either a large prime p or a power of p. If q is a large prime p, the prime field $GF(p)$, also labeled as $\mathbb{F}_p$ or $\mathbb{Z}_p$, is a field of characteristic p where $p \neq 2, 3$, that is, $\sum_{i=1}^{k} 1 = k \neq 0$ for $1 \leq k < p$ and $\sum_{i=1}^{p} 1 = 0$. If q is a power of p, denoted by p^m, the Galois field $GF(p^m)$ is an extension field of $GF(p)$, where p is typically chosen as 2 for the sake of binary property in hardware. The finite fields are also called Galois fields, in honor of their discoverer.

On the basis of various characteristics, the Weierstrass equation (2.2) can be simplified into different forms by a linear change of variables. The following paragraphs shows the equation for a field of characteristic $\neq 2, 3$ and a field of characteristic 2.

Let $\mathbb{F}$ be a field of characteristic $\neq 2, 3$ and $char(\mathbb{F})$ denote the characteristic of $\mathbb{F}$. Since the $char(\mathbb{F}) \neq 2$, substitute (X, Y) by $(X, Y - \frac{a_1 X + a_3}{2})$ on the left hand side in (2.2).

$$\begin{aligned} & Y^2 + a_1 XY + a_3 Y \quad \text{substitute} \quad (X, Y) \rightarrow (X, Y - \frac{a_1 X + a_3}{2}) \\ \Rightarrow \quad & (Y - \tfrac{a_1 X + a_3}{2})^2 + a_1 X (Y - \tfrac{a_1 X + a_3}{2}) + a_3 (Y - \tfrac{a_1 X + a_3}{2}) \\ = \quad & Y^2 - \tfrac{a_1^2}{4} X^2 - \tfrac{a_1 a_3}{2} X - \tfrac{a_3^2}{4} \end{aligned} \tag{2.4}$$

Notice that both XY and Y term are eliminated so the coefficients a_1 and a_3 should be zero. Thus the equation (2.4) results in Y^2 by substitution for $a_1 = a_3 = 0$. Further, the $char(\mathbb{F}) \neq 3$ so substitute (X, Y) by $(X - \frac{a_2}{3}, Y)$ on the right hand side in equation (2.2).

$$\begin{aligned} & X^3 + a_2 X^2 + a_4 X + a_6 \quad \text{substitute} \quad (X, Y) \rightarrow (X - \frac{a_2}{3}, Y) \\ \Rightarrow \quad & (X - \tfrac{a_2}{3})^3 + a_2 (X - \tfrac{a_2}{3})^2 + a_4 (X - \tfrac{a_2}{3}) + a_6 \\ = \quad & X^3 + (\tfrac{-1}{3} a_2^2 + a_4) X + (\tfrac{2}{27} a_2^3 - \tfrac{1}{3} a_2 a_4 + a_6) \end{aligned} \tag{2.5}$$

Then again, the X^2 term is eliminated so that the coefficient a_2 should be zero and the equation (2.5) results in $X^3 + a_4X + a_6$ by setting $a_2 = 0$. According to (2.4) and (2.5), let $a_1 = a_2 = a_3 = 0, a_4 = a, a_6 = b$ and the equation (2.2) is modified as follows

$$Y^2 = X^3 + aX + b, \quad a, b \in \mathbb{F} \tag{2.6}$$

where $char(\mathbb{F}) \neq 2, 3$. Note that the elliptic curve is a smooth curve, i.e. the curve is non-singular. Review in Theorem (2.1), an elliptic curve should have its discriminant nonzero. Therefore, the discriminant of the cubic curve (2.6) can be derived through (2.3) by substitution for $a_1 = a_2 = a_3 = 0, a_4 = a, a_6 = b$. Thus $\Delta = -16(4a^3 + 27b^2) \neq 0$.

For a field of characteristic 2, only the non-supersingular case is considered. In brief, non-supersingular has the result of the coefficient $a_1 \neq 0$. Since $a_1 \neq 0$, substitute (X, Y) by $(a_1^2X + \frac{a_3}{a_1}, a_1^3Y + \frac{a_1^2a_4+a_3^2}{a_1^3})$ in (2.2) likewise. A simplified form is obtained as follows

$$Y^2 + XY = X^3 + aX^2 + b, \quad a, b \in \mathbb{F} \tag{2.7}$$

where $char(\mathbb{F}) = 2$. There is no need to care whether or not the cubic polynomial on the right hand side in (2.7) has multiple roots.

2.2 Elliptic Curves Arithmetics over Affine Coordinates

Elliptic curve cryptography makes use of elliptic curves where the variables and coefficients are belong to a finite field. Two kinds of elliptic curves are commonly used in cryptographic applications. They are prime curves over $GF(p)$ and binary curves over $GF(2^m)$ respectively. Before discussion on the above curves, the elliptic curves over the reals are first introduced because some of the basic concepts are easier to visualize.

2.2.1 Elliptic Curves over the Reals

According to equation (2.6), a definition for elliptic curves over the reals is given below.

Definition 2.4. *A non-singular elliptic curve E over the reals is an equation of the form*

$$y^2 = x^3 + ax + b \tag{2.8}$$

where $a, b \in \mathbb{R}$ are constants such that $4a^3 + 27b^2 \neq 0$.

It can be shown that the condition $4a^3 + 27b^2 \neq 0$ is necessary and sufficient to ensure that the equation (2.8) has three distinct roots which may be real or complex numbers. Figure 2.2 shows two non-singular elliptic curves and one singular elliptic curve whose equation are $y^2 = x^3 - 4x$, $y^2 = x^3 + 73$, and $y^2 = x^3 - 3x - 2$ respectively.

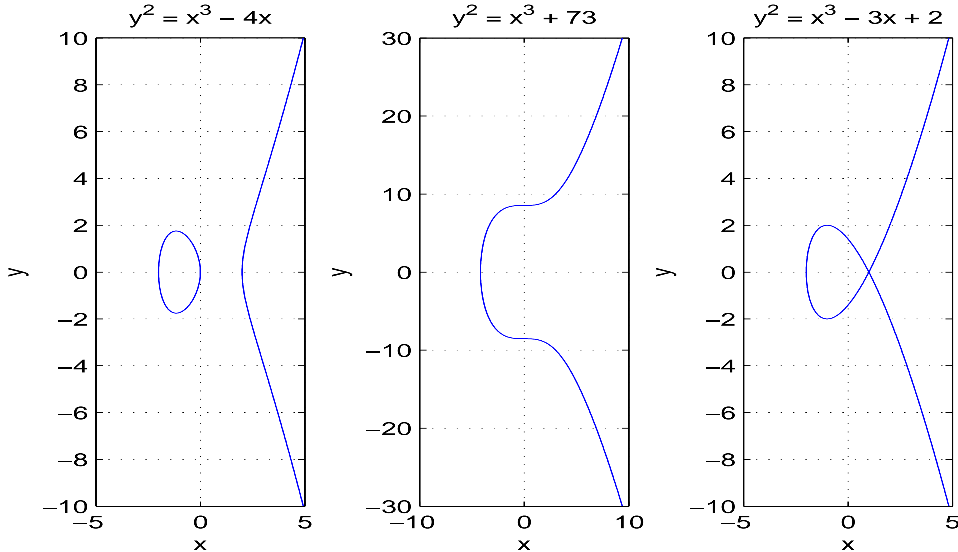


Figure 2.2: Elliptic curves over the reals

Let E be a non-singular elliptic curve over the reals. Given two points P and Q on E, the negative of P, denoted by $-P$, and the sum $P + Q$ is defined as follows:

1. If P is the point at infinity $\mathcal{O}$, then $-P$ is $\mathcal{O}$ and $P + Q$ is Q; that is, $\mathcal{O}$ is the additive identity which is also called zero element of the group of points.

2. If P is not the point at infinity $\mathcal{O}$, then $-P$ is the symmetry point of P on the curve E; that is, $-P$ is the point with the same x-coordinate and negative the y-coordinate of P, i.e. $-(x, y) = (x, -y)$. According to equation (2.8), if (x, y) is a point on the curve E, then the point $(x, -y)$ is consequently on the curve E.

3. If P and Q are different points on E with different x-coordinates, then let l be the line through P and Q, and the line l intersects the curve E in exactly one more point R. Then the sum $P + Q = -R$ is defined and is illustrated in Figure 2.3.

4. If P and Q are different points on E with the same x-coordinates, that is, Q is a symmetry point of P equal to $-P$, then the sum $P + Q = P + (-P) = \mathcal{O}$ is defined.

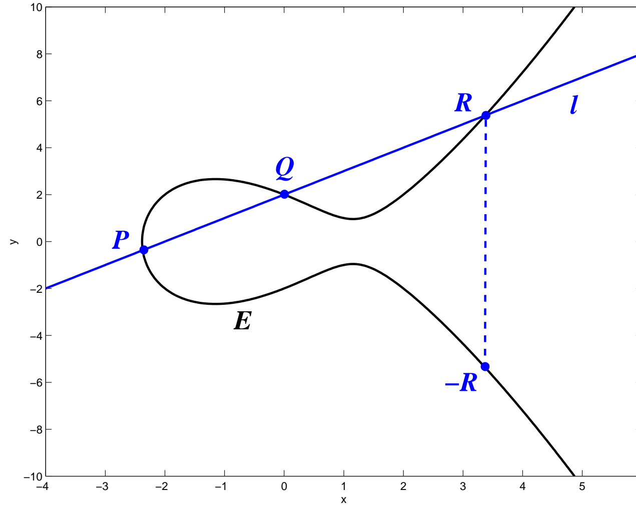


Figure 2.3: Adding two distinct points $P + Q = -R$

5. If P and Q are the same points on E, then let the line l be the tangent line to the curve at P and the point R be the only other point of intersection of l with the curve E. Thus the sum $P + Q = P + P = 2P = -R$ is defined and is illustrated in Figure 2.4. Furthermore, if the tangent line has a double tangency at P, that is, P is a point of inflection, then the sum $P + Q = P + P = 2P = -P$ is defined.

In figure 2.3, let (x_1, y_1), (x_2, y_2), $(x_3, -y_3)$ and (x_3, y_3) denote the coordinates of P, Q, R and $P+Q$ respectively. Let $l : y = \lambda x + \beta$ be the equation of the line through P and Q then $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ is the slope of the line l and $\beta = y_1 - \lambda x_1 = y_2 - \lambda x_2$ is the consequence of the point P lying on the line l. Assume that t is a variable and $(t, \lambda t + \beta)$ denotes the coordinates of arbitrary points on the line l. The point on l simultaneously lies on the elliptic curve E if and only if $(t, \lambda t + \beta)$ satisfies equation (2.8) so that $(\lambda t + \beta)^2 = t^3 + at + b$ and rearrange it below by order of t.

$$t^3 + (-\lambda^2)t^2 + (a - 2\lambda\beta)t + (b - \beta^2) = 0 \tag{2.9}$$

Note that the equation has exactly three distinct roots and two of them are known as x_1 and x_2. Remember the relation between roots and coefficient mentioned in Viéte formula first proposed by François Viéte (1540–1603), a French mathematician.

Theorem 2.2. *(**Viéte's Formula**) Assume $P(x)$ is a polynomial of degree n with roots*

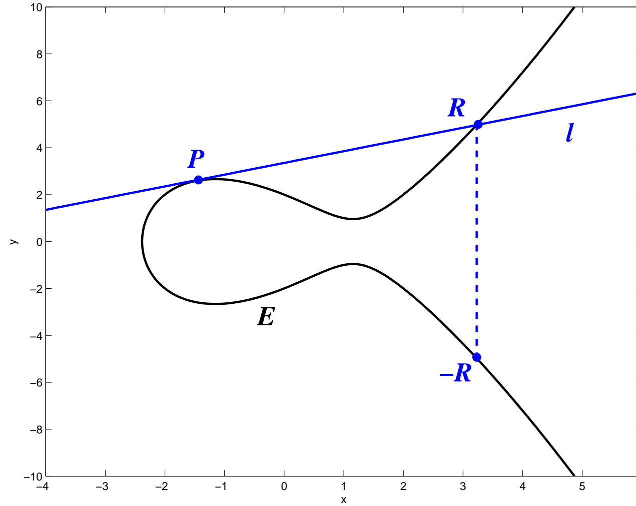


Figure 2.4: Doubling a point $2P = -R$

$x_1, x_2, \ldots, x_n$. For $1 \le i \le n$, let S_i be the sum of the products of distinct polynomial roots x_j of the polynomial

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0 \tag{2.10}$$

where the roots are taken i at a time, i.e. S_i is defined as the symmetric polynomial $\Pi_i(x_1, \ldots, x_n)$ for $i = 1, \ldots, n$, where

$$S_i = \Pi_i(x_1, \ldots, x_n) = \sum_{1 \le \alpha_1 < \alpha_2 < \ldots < \alpha_k \le n} x_{\alpha_1} x_{\alpha_2} \cdots x_{\alpha_k} \tag{2.11}$$

For example, the first few values of S_i are

$$\begin{aligned}
S_1 &= \Pi_1(x_1, \ldots, x_n) &= \sum_{1 \le i \le n} x_i &= x_1 + x_2 + x_3 + x_4 + \cdots \\
S_2 &= \Pi_2(x_1, \ldots, x_n) &= \sum_{1 \le i < j \le n} x_i x_j &= x_1 x_2 + x_1 x_3 + x_1 x_4 + x_2 x_3 + \cdots \\
S_3 &= \Pi_3(x_1, \ldots, x_n) &= \sum_{1 \le i < j < k \le n} x_i x_j x_k &= x_1 x_2 x_3 + x_1 x_2 x_4 + x_2 x_3 x_4 + \cdots
\end{aligned}$$

and so on. Then Viéte's formula states that

$$S_i = (-1)^i \frac{a_{n-i}}{a_n} \tag{2.12}$$

Proof. The polynomial $P(x)$ can also be written as

$$\begin{aligned}
P(x) &= a_n (x - x_1)(x - x_2) \cdots (x - x_n) \\
&= a_n (x^n - S_1 x^{n-1} + S_2 x^{n-2} - \cdots + (-1)^n S_n)
\end{aligned} \tag{2.13}$$

According to equation (2.10), setting the coefficients equal yields

$$a_n(-1)^i S_i = a_{n-i}$$

which is what the Viéte's formula states for. **Q.E.D.**

The Viéte formula was proved by Viéte (1579) for positive roots only, and the general theorem was proved by Gérard Desargues (1591–1661). Therefore the sum of the roots s_1 of a monic polynomial shown in (2.9) is equal to minus the coefficient of the second-to-highest order. A monic polynomial or normed polynomial is a polynomial whose leading coefficient is equal to 1. It concludes that the third root x_3 in (2.9) is equal to $\lambda^2 - x_1 - x_2$ since the sum of the three distinct roots s_1 is λ^2. Then the y-coordinate of R is $\lambda x_3 + \beta$ and y_3 is minus the y-coordinate of R. Therefore the coordinate of $P + Q$ in terms of x_1, x_2, y_1, y_2 is shown below.

$$\begin{aligned} x_3 &= (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2 \\ y_3 &= (\frac{y_2 - y_1}{x_2 - x_1})(x_1 - x_3) - y_1 \end{aligned} \tag{2.14}$$

In figure 2.4, let (x_1, y_1), (x_2, y_2), $(x_3, -y_3)$ and (x_3, y_3) denote the coordinates of P, Q, R and $P + Q$ respectively. Since P and Q are the same point, $x_2 = x_1$ and $y_2 = y_1$. Let $l : y = \lambda x + \beta$ be the equation of the tangent line to the curve E at P. The slope of the tangent line at P can be derived by differentiation of the equation (2.8) as follows.

$$\begin{aligned} \frac{d}{dx}(y^2) &= \frac{d}{dx}(x^3 + ax + b) \\ (\frac{dy}{dx}) &= \frac{3x^2 + a}{2y} \end{aligned} \tag{2.15}$$

So the slope of the tangent line $\lambda = \frac{3x_1^2+a}{2y_1}$. According to (2.14), substitute $(\frac{y_2-y_1}{x_2-x_1})$ for $(\frac{3x_1^2+a}{2y_1})$ and $x_2 = x_1$, $y_2 = y_1$. A formula for doubling a point is obtained.

$$\begin{aligned} x_3 &= (\frac{3x_1^2 + a}{2y_1})^2 - 2x_1 \\ y_3 &= (\frac{3x_1^2 + a}{2y_1})(x_1 - x_3) - y_1 \end{aligned} \tag{2.16}$$

Table 2.1 shows the addition formula mentioned above.

	Point Addition ($P \neq Q$)	Point Doubling ($P = Q$)
$P+Q$	$x_3 = \lambda^2 - x_1 - x_2$	$x_3 = \lambda^2 - 2x_1$
	$y_3 = \lambda(x_1 - x_3) - y_1$	$y_3 = \lambda(x_1 - x_3) - y_1$
	$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$	$\lambda = \frac{3x_1^2 + a}{2y_1}$

Table 2.1: Point addition formula over reals.

2.2.2 Elliptic Curves over Prime Fields

Let $p > 3$ be a prime. Elliptic curves over $GF(p)$ are defined almost the same as they are over the reals and the operations over the reals are replaced by modulus operations.

Definition 2.5. *Let $p > 3$ be a prime. A non-singular elliptic curve E over the finite field $GF(p)$ is an equation of the form*

$$y^2 \equiv x^3 + ax + b \pmod p \tag{2.17}$$

where $a, b \in GF(p)$ are constants such that $4a^3 + 27b^2 \not\equiv 0 \pmod p$.

Assume that (x_1, y_1), (x_2, y_2) and (x_3, y_3) denote the coordinates of P, Q and $P+Q$ respectively. Then the coordinate of $-P$ is defined as $(x_1, -y_1)$ and $P + (-P) = \mathcal{O}$. According to equation (2.14) and (2.16), the point addition formula of the elliptic curves over $GF(p)$ is shown in Table 2.2.

	Point Addition ($P \neq Q$)	Point Doubling ($P = Q$)
$P+Q$	$x_3 \equiv \lambda^2 - x_1 - x_2 \pmod p$	$x_3 \equiv \lambda^2 - 2x_1 \pmod p$
	$y_3 \equiv \lambda(x_1 - x_3) - y_1 \pmod p$	$y_3 \equiv \lambda(x_1 - x_3) - y_1 \pmod p$
	$\lambda \equiv \frac{y_2 - y_1}{x_2 - x_1} \pmod p$	$\lambda \equiv \frac{3x_1^2 + a}{2y_1} \pmod p$

Table 2.2: Point addition formula over $GF(p)$

2.2.3 Elliptic Curves over Extension of Binary Fields

Definition 2.6. *Let $p(x)$ be a primitive polynomial of degree m. A non-supersingular elliptic curve E over the extension of binary field $GF(2^m)$ is an equation of the form*

$$y^2 + xy = x^3 + ax^2 + b \tag{2.18}$$

where $a, b \in GF(2^m)$ are constants.

Note that in this subsection, all of the arithmetic operations are defined over $GF(2^m)$ and all of the parameters are belong to $GF(2^m)$, too. Assume that (x_1, y_1), (x_2, y_2) and (x_3, y_3) denote the coordinates of P, Q and $P + Q$ respectively. Then the coordinate of $-P$ is defined as $(x_1, x_1 + y_1)$ and $P + (-P) = \mathcal{O}$.

If $P \neq Q$, let $l : y = \lambda x+\beta$ be the equation of the line through P and Q then $\lambda = \frac{y_2+y_1}{x_2+x_1}$ is the slope of the line l and $\beta = y_1 + \lambda x_1 = y_2 + \lambda x_2$ is the consequence. The following equation shows all of the points $(t, \lambda x + \beta)$ on l simultaneously lies on the curve E.

$$t^3 + (\lambda^2 + \lambda + a)t + (\beta^2 + b) = 0 \tag{2.19}$$

Thus the third root $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ and the corresponding y-coordinate is $\lambda x_3 + \beta$. So the negative of the y-coordinate $y_3 = (\lambda x_3 + \beta) + x_3 = \lambda(x_1 + x_3) + x_3 + y_1$.

If $P = Q$, let $l : \lambda x + \beta$ be the equation of the tangent line to the curve E at P. The slope of the tangent line at P can be derived by differentiation of the equation (2.18).

$$\begin{aligned} \frac{d}{dx}(y^2 + xy) &= \frac{d}{dx}(x^3 + ax^2 + b) \\ (\frac{dy}{dx}) &= x + \frac{y}{x} \end{aligned} \tag{2.20}$$

So the slope of the tangent line $\lambda = x_1 + \frac{y_1}{x_1}$. Since $P = Q$, $x_3 = \lambda^2 + \lambda + a$ and $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$; moreover, there is another formula commonly used for y_3 by changing varibale. Given $\lambda = x_1 + \frac{y_1}{x_1} = \frac{x_1^2+y_1}{x_1}$ that leads to $\lambda x_1 + y_1 = x_1^2$. Thus rearrange $y_3 = (\lambda+1)x_3 + \lambda x_1 + y_1$ and adapt it for $y_3 = (\lambda+1)x_3 + x_1^2$. Table 2.3 lists all obtained formulas of the above together for $GF(2^m)$.

	Point Addition ($P \neq Q$)	Point Doubling ($P = Q$)
$P + Q$	$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$ $\lambda = \frac{y_2+y_1}{x_2+x_1}$	$x_3 = \lambda^2 + \lambda + a$ $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$ $= (\lambda + 1)x_3 + x_1^2$ $\lambda = x_1 + \frac{y_1}{x_1}$

Table 2.3: Point addition formula over $GF(2^m)$

2.3 Elliptic Curve Arithmetics over Projective Coordinates

Traditionally, the Galois field inversion is accomplished using the Fermat's little theorem described in chapter 3.2.1 which is estimated to take 9 to 30 Galois field multiplication's computational time in case of $GF(p)$ with p larger than 100 bits [16]. Therefore, transferring the point coordinates into another coordinates that can eliminate the inversion operation can greatly improve the performance.

Except *Affine coordinate*, there are four different coordinate systems mentioned in this section: *Homogeneous Projective*, *Jacobian*, *Chudnovsky-Jacobian* and *Modified Jacobian*. The computational time is represented in terms of number of multiplications (M) and squaring (S). For simplicity, the addition, subtraction and multiplication by a small constant will not be taken into consideration because they are very fast compared to multiplication, squaring and inversion operations. A comparison table including *Affine* coordinate is given at chapter 3.5 in Table 3.6. Note that the equations mentioned in the following sections are all used over prime field since the prime field operation is the main concern in this thesis.

2.3.1 Homogeneous Projective Coordinates

In the homogeneous projective coordinates the following transformation functions are used to substitute x and y in the affine Weierstrass equation in equation 2.2 to get the projected coordinates:

$$\begin{cases} x \rightarrow \frac{X}{Z} \\ y \rightarrow \frac{Y}{Z} \end{cases} \tag{2.21}$$

The elliptic curve equation becomes:

$$Y^2Z = X^3 + aXZ^2 + bZ^3 \tag{2.22}$$

In this coordinate system, the points P, Q and R are represented as follows:
$P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ and $R = P + Q = (X_3, Y_3, Z_3)$.

- The addition formulas are given by:
  $X_3 = vA$, $Y_3 = u(v^2X_1Z_2 - A) - v^3Y_1Z_2$, $Z_3 = v^3Z_1Z_2$

where:

$u = Y_2Z_1 - Y_1Z_2$, $v = X_2Z_1 - X_1Z_2$ and $A = u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2$

- The doubling formulas are given by ($R = 2P$):

  $X_3 = 2hs$, $Y_3 = w(4B - h) - 8Y_1^2s^2$, $Z_3 = 8s^3$

  where:

  $w = aZ_1^2 + 3X_1^2$, $s = Y_1Z_1$, $B = X_1Y_1s$ and $h = w^2 - 8B$

The computational time for addition and doubling operations using homogeneous coordinates is (12M+2S) and (7M+5S) respectively.

2.3.2 Jacobian Coordinates

In the Jacobian coordinates the following transformation functions are used:

$$\begin{cases} x \longrightarrow \frac{X}{Z^2} \\ y \longrightarrow \frac{Y}{Z^3} \end{cases} \tag{2.23}$$

The elliptic curve equation becomes:

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{2.24}$$

In this coordinate system, the points P, Q and R are represented as follows:
$P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ and $R = P + Q = (X_3, Y_3, Z_3)$.

- The addition formulas are given by:

  $X_3 = -H^3 - 2U_1H^2 + r^2$, $Y_3 = -S_1H^3 + r(U_1H^2 - X_3)$, $Z_3 = Z_1Z_2H$

  where:

  $U_1 = X_1Z_2^2$, $U_2 = X_2Z_1^2$, $S_1 = Y_1Z_2^3$, $S_2 = Y_2Z_1^3$, $H = U_2 - U_1$ and $r = S_2 - S_1$

- The doubling formulas are given by ($R = 2P$):

  $X_3 = T$, $Y_3 = -8Y_1^4 + M(S - T)$, $Z_3 = 2Y_1Z_1$

  where:

  $S = 4X_1Y_1^2$, $M = 3X_1^2 + aZ_1^4$, and $T = -2S + M^2$

The computational time for addition and doubling operations using Jacobian coordinates is (12M+4S) and (4M+6S) respectively.

2.3.3 Chudnovsky-Jacobian Coordinates

D. V. Chudnovsky [17] concluded that Jacobian coordinate system provide faster doubling and slower addition compared to projective coordinates. In order to speedup addition, he proposed the Chudnovsky-Jacobian coordinate system.In this coordinates a Jacobian coordinates point is represented internally as 5-tupel point (X, Y, Z, Z^2, Z^3). The transformation and elliptic curve equations are the same as in Jacobian coordinates, while the points P, Q, and R represented as follows: In this coordinate system, the points P, Q and R are represented as follows:

$P = (X_1, Y_1, Z_1, Z_1^2 Z_1^3)$, $Q = (X_2, Y_2, Z_2, Z_2^2, Z_2^3)$ and $R = P+Q = (X_3, Y_3, Z_3, Z_3^2, Z_3^3)$.

The main idea in Chudnovsky-Jacobian coordinate is that the Z_2, Z_3 are already calculated in the previous iteration and are not necessary to be calculated again in the current iteration. In other words, Z_1^2, Z_1^3, Z_2^2, Z_2^3 are computed during the previous iteration and fed into the current iteration as inputs, while Z_3^2, Z_3^3 need to be calculated.

- The addition formulas are given by:
  $X_3 = -H^3 - 2U_1H^2 + r^2$, $Y_3 = -S_1H^3 + r(U_1H^2 - X_3)$, $Z_3 = Z_1Z_2H$, $Z_3^2 = Z_3^2$, $Z_3^3 = Z_3^3$
  where:
  $U_1 = X_1Z_2^2$, $U_2 = X_2Z_1^2$, $S_1 = Y_1Z_2^3$, $S_2 = Y_2Z_1^3$, $H = U_2 - U_1$ and $r = S_2 - S_1$

- The doubling formulas are given by $(R = 2P)$:
  $X_3 = T$, $Y_3 = -8Y_1^4 + M(S - T)$, $Z_3 = 2Y_1Z_1$, $Z_3^2 = Z_3^2$, $Z_3^3 = Z_3^3$
  where:
  $S = 4X_1Y_1^2$, $M = 3X_1^2 + aZ_1^4$, and $T = -2S + M^2$

The computational time for addition and doubling operations using Chudnovsky-Jaconian coordinates is (11M+3S) and (5M+6S) respectively.

2.3.4 Modified Jacobian Coordinates

Henri Cohen et.al. modified the Jacobian coordinates [16] and claimed that they got the fastest possible point doubling. The term (aZ^4) is needed in doubling rather than in addition. Taking this into consideration, they employed the idea of internally representing this term and provide it as input to the doubling formula. The point is represented in 4-tuple representation (X, Y, Z, aZ^4). It uses the same transformation equations used in Jacobian coordinates.

In this coordinate system, the points P, Q and R are represented as follows: $P = (X_1, Y_1, Z_1, aZ_1^4)$, $Q = (X_2, Y_2, Z_2, aZ_2^4)$ and $R = P + Q = (X_3, Y_3, Z_3, aZ_3^4)$.

- The addition formulas are given by:
  $X_3 = -H^3 - 2U_1H^2 + r^2$, $Y_3 = -S_1H^3 + r(U_1H^2 - X_3)$, $Z_3 = Z_1Z_2H$, $aZ_3^4 = aZ_3^4$
  where:
  $U_1 = X_1Z_2^2$, $U_2 = X_2Z_1^2$, $S_1 = Y_1Z_2^3$, $S_2 = Y_2Z_1^3$, $H = U_2 - U_1$ and $r = S_2 - S_1$

- The doubling formulas are given by $(R = 2P)$:
  $X_3 = T$, $Y_3 = M(S - T) - U$, $Z_3 = 2Y_1Z_1$, $aZ_3^4 = 2U(aZ_1^4)$
  where:
  $S = 4X_1Y_1^2$, $U = 8Y_1^4$, $M = 3X_1^2 + aZ_1^4$, and $T = -2S + M^2$

The computational time for addition and doubling operations using Modified Jacobian coordinates is (13M+6S) and (4M+4S) respectively.

2.4 Elliptic Curves Scalar Multiplication

Scalar multiplication is used to compute a multiple of an Elliptic curve point kP, where P is an elliptic curve point and k is a positive integer except the condition that k equals to the order of P, then kP is the point obtained by adding together k copies of P and this operation dominates the execution time of elliptic curve cryptographic schemes.

2.4.1 Double-and-Add Algorithm

Algorithm 2.1. *(**Left-Right Double-and-Add Algorithm**)*

Input: *A positive integer $k<n$, where n is the order of P; and an elliptic curve point P.*

Output: *The elliptic curve point kP.*

1. *Let $k_n k_{n-1} \ldots k_1 k_0$ be the binary representation of k, where the leftmost bit k_n is 1.*
2. *Set $R = P$.*
3. *For i from $n-1$ down to 0 do*
   - *3.1 Set $R = 2R$.*
   - *3.2 If $k_i = 1$, then set $R = R + P$.*
4. *Output R.*

Algorithm 2.2. *(**Right-Left Double-and-Add Algorithm**)*

Input: *A positive integer $k<n$, where n is the order of P; and an elliptic curve point P.*

Output: *The elliptic curve point kP.*

1. *Let $k_n k_{n-1} \ldots k_1 k_0$ be the binary representation of k, where the leftmost bit k_n is 1.*
2. *Set $R = infinity$, $Q = P$.*
3. *For i from 0 down to $n-1$ do*
   - *3.1 If $k_i = 1$, then set $R = R + Q$.*
   - *3.2 Set $Q = 2Q$.*
4. *Output R.*

The double-and-add algorithm is a basic method for calculating scalar multiplication. It achieves by repeated point double and add operations. The expected number of ones in the binary representation of k is $\frac{m}{2}$, where m is the length of the integer k. The number of ones in k indicates the number of times that point addition performs and the number of times that point doubling operation performs is approximately equal to m.

Thus Algorithm 2.1 averagely takes $\frac{m}{2}$ times point addition and m times point doubling to perform m-bit elliptic curve scalar multiplication once.

Algorithm 2.2 calculates the scalar multiplication in an opposite way. It also achieves by repeated point double and add operations. But if there exist one doubling hardware and one addition hardware, it can achieve the double and add operations simultaneously. It is useful in DPA resistant mentioned later in chapter 4.

2.4.2 Addition-Subtraction Method

If $P(x, y) \in E(\mathbb{F}_p)$ then $-P = (x, -y)$; else if $P(x, y) \in E(\mathbb{F}_{2^m})$ then $-P = (x, x+y)$. Thus the point subtraction is as efficient as point addition. Then Algorithm 2.1 is replaced by using addition-subtraction method and shown in Algorithm 2.3.

Algorithm 2.3. ***(Addition-Subtraction Method)***

Input: *A positive integer $k < n$, where n is the order of P; and an elliptic curve point P.*

Output: *The elliptic curve point kP.*

1. *Let $e_n e_{n-1} \ldots e_1 e_0$ be the binary representation of $3k$, where the leftmost bit e_n is 1.*
2. *Let $k_n k_{n-1} \ldots k_1 k_0$ be the binary representation of k.*
3. *Set $R = P$.*
4. *For i from $n-1$ down to 1 do*
   - *4.1 Set $R = 2R$.*
   - *4.2 If $e_i = 1$ and $k_i = 0$, then set $R = R + P$.*
   - *4.3 If $e_i = 0$ and $k_i = 1$, then set $R = R - P$.*
5. *Output R.*

2.4.3 Binary NAF Method

Owing to point subtraction is as efficient as point addition, the signed digit representation $k = \sum k_i 2^i$ is used, where $k_i \in \{0, \pm 1\}$. A non-adjacent form (NAF) is a useful signed representation which has the property that no two consecutive bits in k are nonzero and has the fewest nonzero bits of any signed digit representation of k. Each positive integer k has its unique NAF, denoted by NAF(k). The NAF of an integer k can be computed efficiently by using Algorithm 2.4 [18].

Algorithm 2.4. ***(NAF of a Positive Integer)***

Input: *A positive integer k.*

Output: *NAF(k).*

1. *Set $i = 0$.*
2. *While $k \geq 1$ do*
   - *2.1 If k is odd, then set $k_i = 2 - (k \bmod 4)$ and then set $k = k - k_i$; else set $k_i = 0$.*
   - *2.2 Set $k = \frac{k}{2}$ and $i = i + 1$.*
3. *Output k, whose binary representation is $(k_{i-1}k_{i-2}\ldots k_1k_0)$.*

Note that the length of NAF(k) is at most one bit longer than the binary representation of k and the average density of nonzero bits in NAF(k) is approximately $\frac{m}{3}$ [19], where m is the length of the integer k.

Algorithm 2.5. ***(Binary NAF Method)***

Input: *NAF(k) and an elliptic curve point P.*

Output: *The elliptic curve point kP.*

1. *Let $k_nk_{n-1}\ldots k_1k_0$ be signed digit representation of k, where the leftmost bit k_n is 1.*
2. *Set $R = P$.*
3. *For i from $n-1$ down to 0 do*
   - *3.1 Set $R = 2R$.*
   - *3.2 If $k_i = 1$, then set $R = R + P$.*
   - *3.3 If $k_i = -1$, then set $R = R - P$.*
4. *Output R.*

Then the Algorithm 2.5 modifies Algorithm 2.1 by using NAF(k) instead of the binary representation of k and averagely takes approximately $\frac{m}{3}$ times point addition and m times point doubling to perform m-bit elliptic curve scalar multiplication once. Furthermore, it follows that the expected running time of Algorithm 2.3 and Algorithm 2.5 are the same.

Chapter 3

Galois Field Arithmetics

In abstract algebra, Galois field, named in honor of Évariste Galois(1811–1832), which has the same meaning with finite fields is a field that contains only finitely many elements. It plays an important role in number theory, algebraic theory, Galois theory, coding theory, and cryptography. In an elliptic curve cryptosystem, Galois field arithmetics occupy almost all computation time. Therefore, the most efficient method to improve the performance of the entire cryptosystem is to optimize the algorithms for the Galois field arithmetics.

A thorough introduction to $GF(p)$ and $GF(2^m)$ can be found in publications about abstract algebra. To understand the basic mathematical background required in cryptography, Chapter II in [14] written by Neal Kobitz can be referred. In this chapter, algorithms for modular multiplication, inversion, and division are respectively introduced in section 3.1, 3.2, and 3.3. Domain transformation illustrated in section 3.4, and performance comparison is shown in section 3.5.

3.1 Modular Multiplication

3.1.1 Traditional Modular Multiplication Algorithm

In human basic cognition, $A \times B \pmod{p} \equiv C \pmod{p}$, means to multiply A by B, then to find the product modulo p. But this computation flow does not work with computers, because trial division is involved. A traditional modular multiplication algorithm for computers is described below.

Algorithm 3.1. ***(Traditional Modular Multiplication over*** $GF(p)$***)***

Input: *A, B and p, where $A, B \in GF(p)$ and p is the modulus p.*

Output: *C, where $C \equiv A \times B \pmod{P}$.*

1. *Let $a_{m-1}a_{m-2}\ldots a_1a_0$ be the binary representation of A.*
2. *Set $C = 0$.*
3. *For i from $m-1$ to 0 do*
   - 3.1. *Set $C = C \times 2$.*
   - 3.2. *Set $C = (C + a_iB)$.*
   - 3.3. *While $C \geq p$, set $C = C - p$.*
4. *Output C.*

The suffix i of the variable indicates the ith bit in the binary representation of the variable A.

In Algorithm 3.1, the while loop bound C smaller than p. But more than one iteration are required in the loop, that's quite annoying. If variable C can be bounded, then the while loop can be eliminated. From this point of view, the radix-2 Montgomery multiplication algorithm in sub-section 3.1.3 can be intuitively comprehended.

3.1.2 Montgomery Multiplication Algorithm

Montgomery multiplication algorithm, proposed by P. L. Montgomery in 1985 [12], computes the modular multiplication without trial division. It turns the modular multiplication into iterations of addition and shift operations. Thus the Montgomery multiplication is quite appropriate for hardware implementation. Let the modulus p be an m-bit integer with $2^{m-1} \leq p < 2^m$, let A, B, C be positive integers smaller than p, and let r be an integer where p and r are relatively prime, i.e. $\gcd(p, r) = 1$. There exists an integer r^{-1} that indicates the multiplicative inverse of $r \pmod{p}$, i.e. $r \times r^{-1} \equiv 1 \pmod{p}$. Another integer p' is involved, which satisfies $r \times r^{-1} - p \times p' = 1$. Here, both r^{-1} and p' can be derived by the extended Euclidean algorithm described later in sub-section 3.2.2. The Montgomery multiplication algorithm is described below.

Algorithm 3.2. *(Montgomery Multiplication Algorithm)*

Input: *A, B and p, where $A, B \in GF(p)$ and p is the modulus p.*

Output: *C, where $C \equiv A \times B \times r^{-1} \pmod{p}$.*

1. *Set $U = A \times B$*
2. *Set $V \equiv U \times p' \pmod{r}$*
3. *Set $C = (U + V \times p)/r$*
4. *If $C \geq p$, then set $C = C - p$*

Proof. Given $r \times r^{-1} - p \times p' = 1$, so that

$$p \times p' = r \times r^{-1} - 1 \tag{3.1}$$

and in step 2,

$$V = U \times p' - r \times r' \tag{3.2}$$

where r' is a positive integer which satisfies $V \equiv U \times p' \pmod{r}$. Substitute U in equation 3.2 and multiply r on both sides, following equations can be derived.

$$C \times r = U + U \times p' \times p - r \times r' \times p \tag{3.3}$$

Substitute equation 3.1 into equation 3.2 and divide r on both sides:

$$C = U \times r^{-1} - r' \times p \tag{3.4}$$

Modulo p on both sides of equation 3.4:

$$C \equiv U \times r^{-1} \pmod{p} \tag{3.5}$$

$$\equiv A \times B \times r^{-1} \pmod{p} \tag{3.6}$$

In step 3, C is the sum of $\frac{U}{r}$ and $\frac{V \times p}{r}$. Both of them are smaller then p, since r is always set to be an integer larger than p. As a result, C will not be larger than $2 \times p$. Just one subtraction in step 4 can bound C in $GF(p)$.

In this way, Algorithm 3.2 is proven. **Q.E.D.**

In Montgomery multiplication algorithm, the operations of modulo r and divide by r are both trivial operations since r is always given as 2^m. Thus Montgomery multiplication has the advantage of hardware implementation, and it's simpler and faster than traditional modular multiplication.

3.1.3 Modified Montgomery Multiplication Algorithm

Various methods are proposed to realize the Montgomery multiplication algorithm [20]. The radix-2 Montgomery multiplication algorithm [21] over $GF(p)$ is shown in Algorithm 3.3 and it can be easily adapted to do multiplication $GF(2^m)$. Algorithm 3.4 shows the binary version of the radix-2 Montgomery multiplication algorithm and it has been proven by [22].

Algorithm 3.3. *(**Montgomery Multiplication over** $GF(p)$)*

Input: *A, B and p, where $A, B \in GF(p)$ and p is the modulus p.*

Output: *C, where $C \equiv A \times B \times 2^{-m} \pmod{p}$.*

1. *Let $a_{m-1}a_{m-2} \dots a_1a_0$ be the binary representation of A.*
2. *Set $C = 0$.*
3. *For i from 0 to $m-1$ do*
   - 3.1. *Set $T = (C + a_iB)$.*
   - 3.2. *Set $C = (T + t_0p)/2$.*
4. *If $C \geq p$, then set $C = C - p$.*
5. *Output C.*

Algorithm 3.4. *(**Montgomery Multiplication over** $GF(2^m)$)*

Input: *$A(x)$, $B(x)$ and $P(x)$, where $A(x)$, $B(x)$ and $P(x) \in GF(2^m)$, and $GF(2^m)$ is generated by $P(x)$.*

Output: *$C(x)$, where $C(x) \equiv A(x) \times B(x) \times x^{-m} \pmod{P(x)}$.*

1. *Let $A(x) = \sum_{i=0}^{m-1} a_ix^i$, $\forall a_i \in GF(2)$, be the polynomial representation of $A(x)$.*
2. *Set $C(x) = 0$.*
3. *For i from 0 to $m-1$ do*
   - 3.1. *Set $T(x) = (C(x) + a_iB(x))$.*
   - 3.2. *Set $C(x) = (T(x) + t_0P(x))/x$.*
4. *Output $C(x)$.*

The suffix i of the variable indicates the i-th bit in the binary or polynomial representation of the variable, i.e. t_0 denotes the least significant bit of T. Note that the coefficients of the polynomial representation of $A(x)$, i.e. $a_{m-1}a_{m-2} \dots a_1a_0$, are also the

binary representation of the integer $A(2)$ since $A(2) = \sum_{i=0}^{m-1} a_i 2^i$. The addition and subtraction are the same as those in $GF(p)$ and $GF(2^m)$. Furthermore, division by 2 in $GF(p)$ and division by x in $GF(2^m)$ are both shift operations.

Here a more intuitively illustration is given. Algorithm 3.3 can be easily adapted from Algorithm 3.1. To bound the variable C in Algorithm 3.1, C is set to be $(T + t_0 p)/2$ in Algorithm 3.3. Adding $t_0 p$ makes $(T + t_0 p)$ even, thus the right shift operation won't lose the least significant bit. In this way, C is always smaller than $2 \times p$. After the for loop, only one subtraction is involved to bound C in $GF(p)$. To sum up, C suffers m times right shift operation, so C is equal to $A \times B \times 2^{-m} \pmod{p}$ at last.

3.1.4 Integer Domain and Montgomery Domain

In above section, Montgomery modular multiplication is introduced to speed up the traditional modular multiplication. However, if integer a is Montgomery modular multiplied by integer b, the result is:

$$\text{MontMul}(a, b) \equiv a \cdot b \cdot r^{-1} \pmod{p} \tag{3.7}$$

Hence the Montgomery domain representation of a denoted as A is defined as $A = a \cdot r \pmod{p}$. And the integer domain representation of a denotes an ordinary modular operation which is defined as $a \pmod{p}$. If each input of Montgomery modular multiplier is in the Montgomery domain,

$$\begin{aligned} \text{MontMul}(A, B) &\equiv C \\ &\equiv A \cdot B \cdot r^{-1} \pmod{p} &(3.8) \\ &\equiv (a \cdot r) \cdot (b \cdot r) \cdot r^{-1} \pmod{p} &(3.9) \\ &\equiv a \cdot b \cdot r \pmod{p} &(3.10) \\ &\equiv c \cdot r \pmod{p} &(3.11) \end{aligned}$$

the output will also be in the Montgomery domain. Which implies that if inputs are in the Montgomery domain, no matter how many Montgomery modular multiplications are utilized, the final output is still in the Montgomery domain, and only one domain transform is required to transform the result back to integer domain representation.

In elliptic curve cryptosystems, for speed and implementation consideration, the Montgomery modular multiplication is chosen. Thus modular addition, subtraction, and division algorithms must also keep the output results in the Montgomery domain representation. Modular addition and subtraction in Montgomery domain are the same as addition and subtraction in integer domain. And the modular inversion and division are illustrated in later sections.Detailed transformations from and to both domains are described in section 3.4.

3.2 Modular Inversion

Modular inversion is used in cryptographic applications such as the Diffie-Hellman key exchange [23], the public and private key pair generations in RSA and point operations in ECC. Given an m-bit modulus p, modular inversion computes the inversion of a non-zero field element $a \in GF(p)$. The multiplicative inverse of a is denoted as $a^{-1} \pmod{p}$, where $a \times a^{-1} \equiv 1 \pmod{p}$. Furthermore, the multiplicative inverse of a exists if and only if a and p are relatively prime and its proof can be found in publications about abstract algebra.

3.2.1 Fermat's Little Theory

Fermat's little theorem was first stated in 1640 by Pierre de Fermat(1601–1665). It states that if p is a prime number, then for any integer a, $a^p - a$ will be evenly divisible by p. This can be expressed in the notation as follows:

$$a^p \equiv a \pmod{p} \tag{3.12}$$

A variant of this theorem is stated in the following form: if p is a prime and a is an integer co-prime to p, then $a^{p-1} - 1$ will be evenly divisible by p. Expressed as follows:

$$a^{p-1} \equiv 1 \pmod{p} \tag{3.13}$$

The proof of Fermat's little theorem is given as follows.

Proof. $GF(p)$, can be recognized as:

$$GF(p) = 1, 2, \ldots, p-1 \tag{3.14}$$

Given $1 \leq a < p$, that is a is an element of $GF(p)$. Let k be the order of a, so that

$$a^k \equiv 1 \pmod p \tag{3.15}$$

By Lagrange's theorem, k divides the order of $GF(p)$, which is $p-1$, so $p-1 = k \times m$:

$$a^{p-1} \equiv a^{k \times m} \equiv (a^k)^m \equiv 1 \pmod p \tag{3.16}$$

Q.E.D.

Divide both sides of equation 3.13 by a,

$$a^{p-2} \equiv a^{-1} \pmod p \tag{3.17}$$

the multiplication inverse of a over $GF(p)$ is derived. The multiplicative inverse of a over $GF(2^m)$ can also be derived from Fermat's little theorem.

$$a^{-1} = a^{2^m-2} \tag{3.18}$$

a^{p-2} and a^{2^m-2} can be easily derived with iterative multiplications and squares. But when the modulus p or m is extremely large, too much time will be consumed. This method is not adopted in this work.

3.2.2 Extended Euclidean Algorithm

The Euclidean algorithm determines the greatest common divisor (GCD) of two integers. The GCD of a and b, written as $\gcd(a, b)$, is the largest positive integer that evenly divides both a and b. Two integers are called co-prime if and only if their GCD equals 1.

The extended Euclidean algorithm (EEA) [24] is an extension of the Euclidean algorithm and it can be used to find the x and y in Bézout's identity which is a linear diophantine equation. Bézout's identity, named after Étienne Bézout (1730–1783), states that if a and b are non-negative integers, there exist integers x and y (typically either x or y is negative) such that

$$a \cdot x + b \cdot y = \gcd(a, b) \tag{3.19}$$

where x and y can be obtained by the EEA, but they are not uniquely determined. Set $x' = x - k \cdot b$ and $y' = y + k \cdot a$, then (x', y') is another solution to (3.19) since $a \cdot x' + b \cdot y' = a(x - k \cdot b) + b(y + k \cdot a) = a \cdot x + b \cdot y = \gcd(a, b)$. Bézout's identity is proved as follows.

Proof. Let G be the set of all positive integers of $a \cdot x + b \cdot y$, where x and y are integers. Since G is not empty, it has a smallest element by the well-ordering principle. Let $s = a \cdot x_t + b \cdot y_t$ be the smallest element of the set S. According to the division algorithm, there are unique integers q and r that satisfy $a = s \cdot q + r$ with $0 \leq r < s$. Then

$$r = a - s \cdot q = a - (a \cdot x_t + b \cdot y_t)q = a(1 - q \cdot x_t) + b(-q \cdot y_t) \tag{3.20}$$

Note that $(1 - q \cdot x_t)$ and $(-q \cdot y_t)$ are both integers so that r should be in the set S. But the condition $0 \leq r < s$ contradicts the premise that s is the smallest element of S. Thus r must be equal to 0, that is, $a = s \cdot q$, which indicates that a is divisible by s. Similarly, b is also divisible by s. Therefore s is one of the common divisor of a and b. Assume that c is another common divisor of a and b. Let $a = c \cdot q_1$ and $b = c \cdot q_2$, then

$$s = a \cdot x + b \cdot y = c(q_1 \cdot x + q_2 \cdot y) \tag{3.21}$$

which implies c can evenly divide s. Therefore s is the GCD of a and b, i.e. $s = \gcd(a, b)$. **Q.E.D.**

The extended Euclidean algorithm can be written as following algorithm:

Algorithm 3.5. ***(Extended Euclidean Algorithm)***
Input: *Integer A and B, where $A < B$.*
Output: *Integer C, where $C = gcd(A, B)$.*

1. *Set $A' = 1$, $q = 0$.*
2. *While $A' \neq 0$ do*
   - 2.1. *Set $q = \lfloor \frac{B}{A} \rfloor$.*
   - 2.2. *Set $A' = B - q \cdot A$.*
   - 2.3. *If $A' = 0$, then set $C = A$.*
   - 2.4. *Set $B = A$, $A = A'$.*
3. *Output C.*

When a and b are relatively prime, i.e. $a \cdot x + b \cdot y = 1$, x is the multiplicative inverse of $a \pmod b$. To find the multiplicative inverse of A, following simultaneous equations are introduced.

$$\begin{cases} R \cdot A + e \cdot p = U \\ S \cdot A + d \cdot p = V \end{cases} \tag{3.22}$$

P is set as modulus p, U, V, R,and S are variables, d and e are two variant integer, but they are not substantially calculated or presented. Initially, U, V, R and S are respectively set as p, A, 0 and 1:

$$\begin{cases} 0 \cdot A & + & e \cdot p & = & p \\ 1 \cdot A & + & d \cdot p & = & A \end{cases} \tag{3.23}$$

Here, d and e are 0 and 1 respectively. With EEA, which means elimination method in simultaneous equations, introduced, finally equation 3.23 turns into equation 3.24:

$$\begin{cases} R \cdot A & + & e \cdot p & = & 1 \\ 0 \cdot A & + & 0 \cdot p & = & 0 \end{cases} \tag{3.24}$$

The algorithm terminates when $V = 0$, in which case $U = 1$ and then $R \cdot A + e \cdot P = 1$ leads to $R \cdot A \equiv 1 \pmod{P}$, hence $R \equiv A^{-1} \pmod{p}$.

The EEA can solve the equation $a \cdot x + b \cdot y = \gcd(x, y)$ efficiently. But there exists a quotient q in EEA, which means a division is required. As a result, the EEA should be modified to suit binary representation systems.

A binary GCD algorithm was first devised by Josef Stein in 1961 and published in 1967 [25]. Before describing this algorithm, three simple facts are introduced as follows:

1. If a and b are both even, then $\gcd(a, b) = 2 \times \gcd(a/2, b/2)$;
2. If a is even and b are odd, then $\gcd(a, b) = \gcd(a/2, b)$;
3. If a and b are both odd, then $\gcd(a, b) = \gcd(a - b, b)$;

The modular multiplicative inverse algorithm adapted from the binary GCD algorithm is described below:

Algorithm 3.6. *(**Modular Inverse Algorithm over** $GF(p)$)*
Input: *A and P, where $A \in GF(p)$ and P is the modulus p.*
Output: *R, where $R \equiv A^{-1} \pmod{P}$.*

1. *Set $U = P$, $V = A$, $R = 0$ and $S = 1$.*
2. *While $V \neq 0$ do*
   - 2.1. *While U is even do*
     - 2.1.1. *Set $U = U/2$.*
     - 2.1.2. *If R is even, then set $R = R/2$.*
     - 2.1.3. *Else set $R = (R + P)/2$.*
   - 2.2. *While V is even do*
     - 2.2.1. *Set $V = V/2$.*
     - 2.2.2. *If S is even, then set $S = S/2$.*
     - 2.2.3. *Else set $S = (S + P)/2$.*
   - 2.3. *If $U > V$, then set $U = U - V$, $R = R - S$.*
   - 2.4. *Else if $V \geq U$, then set $V = V - U$, $S = S - R$.*
3. *Output $R \pmod{P}$.*

Only by iterative subtractions, parity testings, and right shifts, the multiplicative inverse can be found. In each iteration, the least significant bit of either a or b is reduced, thus the entire routine takes no more than $2(m - 1)$ iterations.

3.2.3 Montgomery Modular Inversion Algorithm

The Montgomery modular inverse [26], based on the EEA, was proposed by Kaliski to match the Montgomery domain operations. Given an m-bit modulus p, the Montgomery modular inverse of a non-zero integer $a \in GF(p)$ is defined as the integer x,

$$x \equiv a^{-1} \cdot 2^m \pmod{p} \tag{3.25}$$

The Kaliski's Montgomery inverse algorithm was re-written as follows with combination of the two phases in one algorithm. It provides two alternative outputs which is in Montgomery domain's result and in the integer domain respectively.

Algorithm 3.7. *(**Montgomery Modular Inverse Algorithm over** $GF(p)$)*

Input: *A and P, where $A \in GF(p)$ and P is the modulus p.*

Output: *R, where* $\begin{cases} R \equiv A^{-1} \cdot 2^m \pmod{P}. \\ R \equiv A^{-1} \pmod{P}. \end{cases}$

1. *Set $U = P$, $V = A$, $R = 0$, $S = 1$ and $k = 0$.*
2. *While $V > 0$ do*
   - *2.1. If U is even, then set $U = U/2$, $S = 2S$.*
   - *2.2. Else if V is even, then set $V = V/2$, $R = 2R$.*
   - *2.3. Else if $U > V$, then set $U = (U - V)/2$, $R = R + S$ and $S = 2S$.*
   - *2.4. Else if $V \geq U$, then set $V = (V - U)/2$, $S = S + R$ and $R = 2R$.*
   - *2.5. Set $k = k + 1$.*
3. *While* $\begin{cases} k \neq m \\ k \neq 1 \end{cases}$ *do*
   - *3.1. If R is even, then set $R = R/2$.*
   - *3.2. Else set $R = (R + P)/2$.*
   - *3.3. Set $k = k - 1$.*
4. *If $R \geq P$, then set $R = 2P - R$, else set $R = P - R$.*
5. *Output R.*

Note that the upper in the braces derives Montgomery modular inverse result and the lower derives modular inverse result, that is to say, output U is equivalent to $A^{-1} \cdot 2^m \pmod{P}$ or $A^{-1} \pmod{P}$ depends on the termination condition in Step 3.

The Montgomery modular inverse algorithm can also be represented as solving the simultaneous equation 3.26.

$$\begin{cases} R \cdot A & + & e \cdot p & = & U \\ S \cdot A & + & d \cdot p & = & V \end{cases} \tag{3.26}$$

The difference between modular inverse and Montgomery inverse is described now. In step 2.1, S is multiplied by 2 while U is even. Which implies that A in the simultaneous equation is divided by 2. Simultaneously, A in lower equation is also divided by 2. To ensure the equivalence between the left-side and the right-side of the lower equation, S is multiplied by 2. Illustrated in equation 3.27:

$$\begin{cases} R \cdot A/2 & + & e/2 \cdot p & = & U/2 \\ 2S \cdot A/2 & + & d \cdot p & = & V \end{cases} \tag{3.27}$$

Step 2.2 is similar to step 2.1. Note that in step 1, R is set as 0. In fact, this R is $-R$. Therefore, in step 3.3, $R = -(-R - S) = R + S$ and in step 3.4, $S = S - (-R) = S + R$. With the two identical operations, the hardware cost is reduced. Hence R and S are set as 0 and 1, no overflow condition on R and S occur. After k iterations, the simultaneous equation is showed below:

$$\begin{cases} R \cdot A/2^k & + & e \cdot p & = & 1 \\ S \cdot A/2^k & + & d \cdot p & = & 0 \end{cases} \tag{3.28}$$

As a result, $R \equiv A^{-1} \cdot 2^k$. Step 3 iteratively divide R by 2 until $R \equiv A^{-1} \cdot 2^m$. Step 4 negates R and bound it in $GF(p)$ at the end of this algorithm.

The Step 2 is iterative and it reduces U or V by one bit in each iteration at least. Step 3 iteratively right shift the result of step 2. When k is subtracted to the field m or 0, variable R equals to $-A^{-1} \cdot 2^m$ or $-A^{-1}$. In step 4, R is negated back to $A^{-1} \cdot 2^m$ or A^{-1}. Obviously, U and V initially have at most $2m$ bits in total since $2^{m-1} \leq U < 2^m$ and $0 < V < U$. But U equals 1 and V equals 0 in the last iteration, therefore, the iteration count in Step 2 takes no more than $(2m-1)$ iterations. Similarly, U and V initially have at least $m+1$ bits in total while V is equal to 1, thus, the iteration count in Step 2 takes no less than m iterations. So the boundary of k is $m \leq k < 2m$ and the total iteration count i of this algorithm is $m \leq i < 3m$. The average iteration count of i is $2m$ by simulation of about two millions of different input V.

If the input is originally in the Montgomery domain, i.e. $A \equiv a \cdot 2^m \pmod{P}$, then the output of the Montgomery inverse is given below.

$$\begin{aligned} X &\equiv (A)^{-1} \cdot 2^m \pmod{P} \\ &\equiv (a \cdot 2^m)^{-1} \cdot 2^m \pmod{P} \\ &\equiv a^{-1} \pmod{P} \end{aligned} \tag{3.29}$$

In order to convert the output to the Montgomery domain, an additional Montgomery multiplication operation is added afterward. If the input is originally in the integer domain, i.e. $A \equiv a \pmod{P}$, then the output in the integer domain needs m iterations more than output in the Montgomery domain. The latencies of the Montgomery modular inverse from and to both domains are listed in the Table3.1:

Table 3.1: Latency of the Montgomery modular inverse from and to both domains.

Domain		Latency (cycles)		
From →	To	MontInv	MontMul	Total
Int →	Int	$4m+1$	-	$4m+1$
Int →	Mont	$3m+1$	-	$3m+1$
Mont →	Int	$3m+1$	-	$3m+1$
Mont →	Mont	$3m+1$	$m+1$	$4m+2$

[1] Int means integer and Mont means Montgomery.

[2] MontInv denotes Montgomery inversion.

[3] MontMul denotes Montgomery multiplication.

[4] m is the bit length of the modulus p.

3.3 Modular Division

3.3.1 Multiplication after Inversion

The modular division operation is traditionally accomplished by modular inversion followed by modular multiplication since the modular division is believed to be slow. It

can be applied to the computation of the parameter λ in ECC. Given two integers a and b in integer domain, divide a by b is done as follows:

$$c \equiv \frac{a}{b} \pmod{p} \quad (3.30)$$

$$\equiv a \cdot b^{-1} \pmod{p} \quad (3.31)$$

which can be performed by modular inverse followed by modular multiplication. But in ECC, operands are represented in the Montgomery domain. Given two integers A and B in the Montgomery domain, the Montgomery modular division is defined as follows:

$$C \equiv \text{MontMul}(A, \text{MontMul}(\text{MontInv}(B), r^2)) \quad (3.32)$$

$$\equiv a \cdot b^{-1} \cdot r \pmod{p} \quad (3.33)$$

$$\equiv c \cdot r \pmod{p} \quad (3.34)$$

$$\equiv \text{MontDiv}(A, B) \quad (3.35)$$

One extra Montgomery multiplication is required to keep the result in the Montgomery domain. Table 3.2 shows the latency of traditional method using a Montgomery inversion followed by a Montgomery multiplication.

Table 3.2: Latency of modular division using traditional method from and to both domain

Domain	Latency (cycles)			
From → To	MonInv	MonMul	MonMul	Total
Int → Int	$3m+1$	$m+1$	-	$4m+2$
Int → Mont	$3m+1$	$m+1$	$m+1$	$5m+3$
Mont → Int	$3m+1$	$m+1$	-	$4m+2$
Mont → Mont	$3m+1$	$m+1$	$m+1$	$5m+3$

3.3.2 Modular Division Algorithm

Given an m-bit modulus p, the modular division of two integers $a, b \in GF(p)$, where $b \neq 0$, is defined as the integer x,

$$x \equiv a \cdot b^{-1} \pmod{p} \quad (3.36)$$

The following algorithm shows a binary add-and-shift algorithm proposed by Sheueling Chang Shantz [27] for modular divison in a residue class.

Algorithm 3.8. ***(Modular Division Algorithm over*** $GF(p)$***)***

Input: *A, B and P, where $A, B \in GF(p)$ and P is the modulus p.*

Output: *R, where $R \equiv A \cdot B^{-1} \pmod{P}$.*

1. *Set $U = P$, $V = B$, $R = 0$ and $S = A$.*
2. *While $U \neq V$ do*
   - *2.1. If U is even, then set $U = U/2$.*
     - *2.1.1. If R is even, then set $R = R/2$.*
     - *2.1.1. Else set $R = (R + P)/2$.*
   - *2.2. Else if V is even, then set $V = V/2$.*
     - *2.2.2. If S is even, then set $S = S/2$.*
     - *2.2.2. Else set $S = (S + P)/2$.*
   - *2.3. Else if $U - V > 0$, then set $U = (U - V)/2$ and $R = R - S$.*
     - *2.3.3. If $R < 0$, then set $R = R + P$.*
     - *2.3.3. If R is even, then set $R = R/2$. Else set $R = (R + P)/2$.*
   - *2.4. Else if $V - U \geq 0$, then set $V = (V - U)/2$ and $S = S - R$.*
     - *2.4.4. If $S < 0$, then set $S = S + P$.*
     - *2.4.4. If S is even, then set $S = S/2$. Else set $S = (S + P)/2$.*
3. *Output R.*

The modular division algorithm above is an iterative process of additions, parity-testings, and shifts. Like Montgomery modular inverse algorithm, it reduces U or V by one bit. But it is different that in the last iteration, U and V are both equal to 1. Thus the entire division routine takes no more than $2(m - 1)$ iterations.

Like the modular inversion algorithm, the modular division algorithm also works by using the elimination method for solving the simultaneous equations below, where d and e are not really computed, too.

$$\begin{cases} R \cdot (A^{-1}B) & + & e \cdot P & = & U \\ S \cdot (A^{-1}B) & + & d \cdot P & = & V \end{cases} \tag{3.37}$$

Note that this algorithm terminates when $U = V = 1$, in which case $R \cdot (A^{-1}B) + eP = 1$ and $S \cdot (A^{-1}B) + dP = 1$. Since P is a prime, i.e. $\gcd(A^{-1}B, P) = 1$, the equation above

definitely exists integer solutions that satisfy $R \cdot (A^{-1}B) + e \cdot P = 1$, where $R \cdot (A^{-1}B) \equiv 1 \pmod{P}$, that is, $R \equiv AB^{-1} \pmod{P}$. Similarly, $S \equiv AB^{-1} \pmod{P}$, too. And it can also be easily obtained that an identical equation that fits equation (3.37) is written below,

$$\begin{cases} 0 \cdot (A^{-1}B) & + & 1 \cdot P & = & P \\ A \cdot (A^{-1}B) & + & d \cdot P & = & B \end{cases} \tag{3.38}$$

Thus the two algorithms of modular inverse and division only differ from the initial value of the variable S with $S = A$ instead. Although d is not really computed, in this case, $d = (-kB)$, where k is an integer that $AA^{-1} = 1 + kP$ since $AA^{-1} \equiv 1 \pmod{P}$.

$$A \cdot (A^{-1}B) + d \cdot P = (1 + kP)B + (-kB) \cdot P = B \tag{3.39}$$

Thus, there exists an integer d satisfy the equation (3.38).

3.3.3 Montgomery Modular Division Algorithm

Given an m-bit modulus p, the Montgomery modular division of the two integers $a, b \in GF(p)$, where $b \neq 0$, is defined as the integer q, where

$$q \equiv a \cdot b^{-1} \cdot 2^m \pmod{p} \tag{3.40}$$

And given a primitive polynomial $p(x)$ with degree m which generates the $GF(2^m)$, the Montgomery modular division of the two element $a(x), b(x) \in GF(2^m)$, where $b(x) \neq 0$, is defined as the polynomial $q(x)$, where

$$q(x) \equiv a(x) \cdot b^{-1}(x) \cdot x^m \pmod{p(x)} \tag{3.41}$$

An alternative algorithm for calculating the Montgomery modular division or real modular division suitable for both $GF(p)$ and $GF(2^m)$ is proposed by Yao-Jen Liu [13]. The Montgomery modular division algorithm over $GF(p)$ is re-written below:

Algorithm 3.9. *(**Montgomery Modular Division Algorithm over** $GF(p)$)*

Input: *A, B and P, where $A, B \in GF(p)$ and P is the modulus p.*

Output: *R, where* $\begin{cases} R \equiv A \cdot B^{-1} \cdot 2^m \pmod{P}. \\ R \equiv A \cdot B^{-1} \pmod{P}. \end{cases}$

1. *Set $U = P$, $V = B$, $R = 0$, $S = A$ and $k = 0$.*
2. *While $V > 0$ do*
   - 2.1. *If U is even, then set $U = U/2$, $S = 2S$.*
   - 2.2. *Else if V is even, then set $V = V/2$, $R = 2R$.*
   - 2.3. *Else if $U - V > 0$, then set $U = (U - V)/2$, $R = R + S$ and $S = 2S$.*
   - 2.4. *Else if $V - U \geq 0$, then set $V = (V - U)/2$, $S = S + R$ and $R = 2R$.*
   - 2.5. *If $R \geq P$, then set $R = R - P$.*
   - 2.6. *If $S \geq P$, then set $S = S - P$.*
   - 2.7. *Set $k = k + 1$.*
3. *While* $\begin{cases} k \neq m \\ k \neq 1 \end{cases}$ *do*
   - 3.1. *If R is even, then set $R = R/2$.*
   - 3.2. *Else set $R = (R + P)/2$.*
   - 3.3. *Set $k = k - 1$.*
4. *Set $R = P - R$.*
5. *Output R.*

The algorithm above is based on EEA and the binary GCD algorithm [28]. It mainly modifies the Montgomery modular inverse algorithm by setting the dividend to the initial value of S. Since $S = A$ at the beginning, there would be overflow conditions. Step 2.5 and step 2.6 bound R and S in $GF(p)$. The Montgomery modular division over $GF(2^m)$ can be simply derived by changing the addition and subtraction to exclusive-or and changing the overflow condition to degree comparison between R, S,and p. The Montgomery modular division algorithm over $GF(2^m)$ is re-written as follows:

Algorithm 3.10. *(Montgomery Modular Division Algorithm over $GF(2^m)$)*

Input: *$A(x)$, $B(x)$ and $P(x)$, where $A(x)$, $B(x)$ and $P(x) \in GF(2^m)$ and $GF(2^m)$ is generated by $P(x)$.*

Output: *$R(x)$, where* $\begin{cases} R \equiv A(x) \cdot B^{-1} \cdot (x)x^m \pmod{P(x)}. \\ R \equiv A(x) \cdot B^{-1}(x) \pmod{P(x)}. \end{cases}$

1. *Set $U(x) = P(x)$, $V(x) = B(x)$, $R(x) = 0$, $S(x) = A(x)$ and $k = 0$.*
2. *While $V(x) \neq 0$ do*
   - 2.1. *If $U(2)$ is even, then set $U(x) = U(x)/x$, $S(x) = xS(x)$.*
   - 2.2. *Else if $V(2)$ is even, then set $V(x) = V(x)/x$, $R(x) = xR(x)$.*
   - 2.3. *Else if $U(2) - V(2) > 0$,*
     then set $U(x) = (U(x) + V(x))/x$, $R(x) = R(x) + S(x)$ and $S(x) = xS(x)$.
   - 2.4. *Else if $V(2) - U(2) \geq 0$,*
     then set $V(x) = (V(x) + U(x))/x$, $S(x) = S(x) + R(x)$ and $R(x) = xR(x)$.
   - 2.5. *If $deg(R) = deg(P)$, then set $R(x) = R(x) + P(x)$.*
   - 2.6. *If $deg(S) = deg(P)$, then set $S(x) = S(x) + P(x)$.*
   - 2.7. *Set $k = k + 1$.*
3. *While* $\begin{cases} k \neq m \\ k \neq 1 \end{cases}$ *do*
   - 3.1. *If $R(2)$ is even, then set $R(x) = R(x)/x$.*
   - 3.2. *Else set $R(x) = (R(x) + P(x))/x$.*
   - 3.3. *Set $k = k - 1$.*
4. *Output $R(x)$.*

The total latency of the traditional method above seems worse in Montgomery domain, so the Montgomery modular division algorithm combines the inversion with multiplication to improve this shortcoming. The maximum number of iterations in Montgomery modular division algorithm is $3m$ or $4m$ depends on the output is in Montgomery or integer domain. The average number of iterations consumed in the Montgomery modular division algorithm is $2m$ or $3m$ depends on the output is in Montgomery or integer domain. This average number is obtained by over millions of simulation. Table 3.3 shows each latency in the worst case and the average case of the Montgomery division from and to both domain. No additional Montgomery multiplication operations is required.

The following table shows the performance comparison between the Montgomery division and previous works. B. S. Kaliski Jr. employed two Montgomery multiplication after

Table 3.3: Latency of Montgomery modular division from and to both domain

Domain From → To	Worst Latency (cycles) $GF(p)$	Worst Latency (cycles) $GF(2^m)$	Avg. Latency (cycles) $GF(p)$	Avg. Latency (cycles) $GF(2^m)$
Int → Int	$4m$	$4m-1$	$3m$	$3m-1$
Int → Mont	$3m$	$3m-1$	$2m$	$2m-1$
Mont → Int	$4m$	$4m-1$	$3m$	$3m-1$
Mont → Mont	$3m$	$3m-1$	$2m$	$2m-1$

one Montgomery inversion [26] and A. Daly et al. used one Montgomery multiplication after one modular division [29].

Table 3.4: Latency of Montgomery division algorithms comparison.

Operation	Best Case	Worst Case	Ave. Case
MontInv + 2MontMul [26]	$3m+4$	$5m+3$	$4m+0.5$
ModDiv + MontMul [29]	$2m+2$	$4m-1$	$2.5m$
MontDiv [13]	$m+2$	$3m$	$2m$

[1] m denotes the degree of the Galois field.

Table 3.4 shows that the Montgomery division algorithm in [13] has better performance than other works.

3.4 Domain Transformation

From above sections, Montgomery domain operations are used in ECC. That is to say, the integer domain inputs should be transformed into Montgomery domain inputs. And the Montgomery domain output of the ECC should also be transformed back to integer domain representation. To do integer domain to Montgomery domain transformation traditionally, let the integer a multiplied by $r^2 \pmod p$, i.e. $2^{2m} \pmod p$, with

the Montgomery multiplication operation.

$$
\begin{aligned}
A &= \text{MontMul}(a, r^2) \\
&\equiv a \cdot r^2 \cdot r^{-1} \pmod p \\
&\equiv a \cdot r \pmod p
\end{aligned}
$$

Note that the constant $r^2 \pmod p$ needs to be precomputed externally. But in a universal design, the modulus p and r are not fixed. Thus additional input port or additional computational time is demanded. In this thesis, the transformation from integer domain to Montgomery domain is done by Montgomery dividing a by a constant 1:

$$
\begin{aligned}
A &= \text{MontDiv}(a, 1) \\
&\equiv a \cdot 1^{-1} \cdot r \pmod p \\
&\equiv a \cdot r \pmod p
\end{aligned}
$$

In this way, m more iterations are consumed than traditional method. But this m iterations are trivial in the entire system.

Similarly, in order to convert the result A back to the integer domain, it can be achieved by Montgomery multiplying A by a constant 1.

$$
\begin{aligned}
a &= \text{MonMul}(A, 1) \\
&\equiv A \cdot 1 \cdot r^{-1} \pmod p \\
&\equiv (a \cdot r) \cdot 1 \cdot r^{-1} \pmod p \\
&\equiv a \pmod p
\end{aligned}
$$

3.5 Summary

In section 2.3, the elliptic curve operations on projective coordinates are introduced. Obviously, the advantage of projective coordinates representations is that no modular division exists. Hence the hardware cost is lower then the design with a modular division. However, with the Montgomery division algorithm, one division only costs 2 times of computational time of one multiplication which is significantly faster than traditional methods. A comparison table is given below.

Table 3.5: Scalar multiplication comparison between different coordinates.

Coordinate	Affine	Projective	Jacobian	C.J.	M.J.
Latency ($m \times n$)	$5 + 4/2 = 7$	$12 + 14/2 = 19$	$10 + 16/2 = 18$	$11 + 14/2 = 18$	$8 + 19/2 = 17.5$

[1] m denotes the degree of the Galois field.

[2] n denotes the length of the scalar.

[3] C.J. denotes the Chudnovsky Jacobian coordinate.

[4] M.J. denotes the modified Jacobian coordinate.

Table 3.5 shows the comparison between affine coordinate representation and projective coordinates representations. In this comparison, double-and-add scalar multiplication is used and the number of addition is estimated as half of the number of doublings. And in affine coordinate, the division is done by the Montgomery division algorithm proposed in [13]. The latency of the Montgomery division algorithm is taken as the average one which is $2m$. And the calculation of total latency of one scalar multiplication excludes the computational time of the domain transformations, the additions, and the subtractions because the computational time of these operations is trivial comparing to the whole system. The latency of one scalar multiplication on affine coordinate is below 40% of scalar multiplications on other coordinates.

From Table 3.6, we can see that the affine coordinate elliptic curve scalar multiplier with the proposed GFAU requires about 13.7% cycles of the latency to complete one elliptic curve scalar multiplication and requires 75% registers of the design in [30].

Table 3.6: Comparison between different coordinates.

Author	Proposed	S. B. Ors [30]
Coordinate	affine	Modified Jacobian
GF operations (avg. cycles)	Add(1) Sub.(1) Mul.(m) Div.($2m$)	Add($2m+1$) Sub.($2m+1$) Mul.($3m+4$) Inv.($\frac{9m^2}{2}+6n$)
E.C. Doubling(avg. cycles)	$5m+8$	$40m+38$
E.C. Addition(avg. cycles)	$4m+6$	$42m+56$
E.C. Multiplication(avg. cycles)	$n(7m+11)$	$n(51m+66)$
Register	$15m$, $1n$	$20m$, $1n$
Multiplier Type	Radix-2	Systolic
Hardware	GFAU	MMM,MAS

[1] m denotes the latency of one Montgomery multiplication.

[2] n denotes the length of the scalar.

[3] GFAU denotes the proposed Galois field arithmetic unit.

[4] MMM denotes the Montgomery modular multiplier.

[5] MAS denotes Modular Adder/Subtracter.

Chapter 4

Power Analysis

Power analysis is a kind of side-channel attack in which the adversary collects the power consumption of a cryptographic hardware device such as a smart card or an integrated circuit. The adversary can extract cryptographic keys and other secret information from the device without invasion.

Simple power analysis (SPA) directly interprets the power traces or graphs of electrical activity over time into useful information. Differential power analysis is a more complex method. It allows an adversary to compute the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations. Both SPA and DPA are proposed in 1999 by Paul Kocher, Joshua Jaffe and Benjamin Jun [10]. In chapter 4.1, simple power analysis and its countermeasures are introduced. In chapter 4.2, differential power analysis and its countermeasure are introduced. In the chapter 4.3, a double-and-add algorithm that is resistant to SPA and DPA is proposed.

4.1 Simple Power Analysis

Simple power analysis observes the measurement of the power consumption of a cryptographic device. A *trace* refers to a set of power consumption measurements taken across a cryptographic operation. To measure a circuit's power consumption, a small (e.g., 50 ohm) resistor is inserted in series with the power or ground input. Then, dividing the voltage difference across the resistor by the resistance yields the current. Equipped with

extraordinary high sample rate (1GHz) instruments, it's able to extract the information at 99% accuracy. Measuring instruments determine the precision of the power analysis. Obviously, to perform such an attack the side-channel information needs to be strong enough to be directly visible in the trace. Further, the secret information needs to have some simple relationship with the operations that the difference in the power trace is visible.

To exercise a SPA on a specific cryptographic device, the adversary is supposed to have a detailed comprehension about the implementation. Besides, the parts of the trace corresponding to the operations under attack needs to be clearly distinguishable from the whole trace.Using SPA in an unprotected elliptic curve cryptographic device, the power trace of the doubling operations can be easily distinguished from the addition operations since the different numbers of Galois field multiplications exist in them.

J. Coron proposed a countermeasure against SPA in 1999 [11] which is showed below in algorithm 4.1.

Algorithm 4.1. *(**Double-and-Add Algorithm Resistant against SPA**)*

Input: *A positive integer $k < n$, where n is the order of P; and an elliptic curve point P.*

Output: *The elliptic curve point kP.*

1. *Let $k_n k_{n-1} \ldots k_1 k_0$ be the binary representation of k, where the leftmost bit k_n is 1.*
2. *Set $R[0] = P$.*
3. *For i from $n-1$ down to 0 do*
   - *3.1 Set $R[0] = 2R[0]$.*
   - *3.2 Set $R[1] = R[0] + P$.*
   - *3.3 Set $R[0] = R[k_i]$.*
4. *Output $R[0]$.*

This algorithm can be called as double-and-add always algorithm. Unlike traditional double-and-add algorithm, both double and add are executed regardless of the scanned bit. Therefore, no conditional branch can be found and one cannot distinguish the scanned bit is 0 or 1 from SPA.

Another method is to change the double-and-add chain. With an additional basis, the scalar k can be represented as a series of $0, 1, -1$ and the binary NAF method introduced

in chapter 2.4.3 can be used to resist the SPA. Since the point addition and the point subtraction contain almost the same operations, one cannot identify them simply from the observing the power trace. The NAF accelerates the whole scalar multiplication due to lower Hamming weight, but it requires additional hardware resource. For more randomness, the width-w NAF method was proposed by K. Okeya and T. Takagi in 2003 [31]. Of course, the more the width w is, the more additional memory space is required. It requires a table of 2^{w-2} pre-computed points.

4.2 Differential Power Analysis

Differential power analysis involves statistically analyzing power consumption measurements from a cryptographic device. The attack exploits biases in power consumption of hardware devices while performing operations. DPA attacks have signal processing and error correction properties which can extract secrets from measurements which contain too much noise to be analyzed using simple power analysis. Using DPA, an adversary can obtain secret keys by statistically analyzing power consumption of multiple cryptographic operations performed by a vulnerable smart card or other devices.

Assume that the scalar multiplication algorithm is immune against SPA by using double-and-add always method (Algorithm 4.1)and the adversary can get the intermediate results in the operation. The scalar is represented by binary representation as $k_{n-1}, k_{n-2}, \cdots, k_0$ where k_i is the i-th bit of the scalar and n is the total length of the scalar. At step i, the processed point depends only on the first $(n-i)$bits $\{k_{n-1}, \cdots, k_i\}$ of the secret scalar. When a point is processed, power trace is correlated to the bits of it. No correlation will be observed if the point is not computed. For example, the second most significant bit k_{n-2} can be learned by calculating the correlation between the power trace and any specific bit of the binary representation of $4P$. If $k_{n-2} = 0$, $4P$ is computed during the binary algorithm. Otherwise, $4P$ is never computed and no correlation will be observed. This correlation method is used to classify power traces of several input points chosen by the attacker.

The following introduces the general form of differential power analysis on ECC. It is a variant form of the zero-exponent, multiple-data (ZEMD) attack algorithm proposed

by T.S. Messerges, E. A. Dabbish and R. H. Sloan in 1999 [32]. Assume that the highest bits,$k_{n-1}, k_{n-2}, \cdots, k_{j+1}$ are known by the attacker (j denotes the current position). A scenario of DPA which finds k_j is given below:

1. The bad boy makes a guess: $k_j = 1$.
2. He chooses several input points $P_1, \cdots, P_t$ and computes $Q_i = 2(\sum_{d=j}^{n-1} k_d 2^{d-j})P_i$.
3. Select a certain bit in the binary representation of $Q_1, \cdots, Q_t$ (fixed for all points) as a selection function g to construct the following two sets:

   $S_t = \{i : g(Q_i) = true\}$ and $S_f = \{i : g(Q_i) = false\}$.
4. Let $C_i = C_i(\tau)$ = power trace obtained from the computation of a full scalar multiplication kP_i. This is a function of the time τ.
5. Let $\langle C_i \rangle_{i \in S}$ denote the average of the functions C_i for the $i \in S$, $S = S_t \cup S_f$. If $\langle C_i \rangle_{i \in S_t} - \langle C_i \rangle_{i \in S_f} \approx 0$, which means the two sets are uncorrelated, it indicates that the guess of k_j is incorrect. On the other hand, if there are spikes in the difference $\langle C_i \rangle_{i \in S_t} - \langle C_i \rangle_{i \in S_f}$, it indicates that the guess of k_j is correct.

To sum up, differential power analysis means that the adversary makes one guess on one specified bit of the secret scalar, then multiplies lots of points by it, classifies these power trace by any fixed bit of the results, averages these two power trace groups respectively, gets the difference of these two average traces and then if the guess is right, there exist spikes in the difference trace, if no spike comes out, then the guess is wrong. In this way, the secret scalar k can be found out bit by bit.

Since DPA can extract the scalar through statistical analysis, some system parameters or computation procedures must be randomized. J. Coron proposed three countermeasures:

1. Randomization of the private scalar:

   $\#\epsilon$ denotes the order of the elliptic curve E. The scalar multiplication $Q = kP$ can be computed as $Q = (k+n\#\epsilon)P$ where n is a random number. This countermeasure makes the DPA infeasible since the scalar changes at each new execution of the algorithm. But more computational time is its disadvantage.

2. Blinding the point P:

   Scalar multiplication $Q = kP$ is randomized by adding a secret random point R for which known as $S = kR$. The computation is accomplished by $Q = k(R + P) - S$. In a reconfigurable design, different elliptic curve require different S. It means additional

3. Randomized projective coordinates:

   Randomized projective coordinates can use the Homogeneous or Jacobian coordinate to randomize a point $P = (x, y)$. For homogeneous projective coordinate, P can be randomized to (rx, ry, r) for a random number $r \in GF(p)$. Similarly, P can be randomized to (r^2x, r^3y, r) in case of using Jacobian coordinates where r is a random number in $GF(p)$.

Countermeasures above try to randomize the power traces to make it harder for an adversary to exploit the differences between these traces. Another countermeasure called randomized exponentiation algorithm which is used to protect RSA cryptosystems proposed in [32] can be modified to resist DPA in elliptic curve cryptosystems. It is randomly chose one bit in the scalar as a starting bit. Compute the scalar multiplication from the chosen bit to the most significant bit of the scalar using Algorithm 2.2, and then compute the remaining scalar to the least significant bit using Algorithm 2.1. The authors of [32] claim that all power analysis attacks proposed in [32] would be significantly diminished by this kind of randomized exponentiation.

4.3 Proposed Countermeasures against SPA

In this section, a countermeasure against SPA is proposed. The main idea of it is to interleave two scalar multiplication's double-and-add chain. Since $k_1P_1 + k_2P_2$ is widely used in elliptic curve cryptographic protocols, for example ECDSA, using a random number to determine which point should be taken into computation. In this way, if an adversary tries to extract the key using SPA, he will get a rearranged key composed by interleaved k_1 and k_2. The proposed randomized interleaving double-and-add algorithm 4.2 is introduced below:

Algorithm 4.2. ***(SPA Resistant Double-and-Add Algorithm)***

Input: *Two positive integer k_1 and k_2; and two elliptic curve point P_1 and P_2.*

Output: *The elliptic curve point $Q = k_1P_1 + k_2P_2$.*

1. *Let $k_{1(n)}k_{1(n-1)} \dots k_{1(1)}k_{1(0)}$ be the binary representation of k_1; $k_{2(m)}k_{2(m-1)} \dots k_{2(1)}k_{2(0)}$ be the binary representation of k_2, where the leftmost bits $k_{1(n)}$ and $k_{2(m)}$ are 1; and a random number r.*
2. *Set $Q_1 = P_1$, $Q_2 = P_2$, $i = n$, $j = m$.*
3. *While $i \neq 0$ and $j \neq 0$ do*

   3.1 If $r = 0$, then do

   - *i. Set $Q_1 = 2Q_1$.*
   - *ii. If $k_{1(i)} = 1$, then set $Q_1 = Q_1 + P_1$*
   - *iii. Set $i = i - 1$.*

   3.2 Else, do

   - *i. Set $Q_2 = 2Q_2$.*
   - *ii. If $k_{2(j)} = 1$, then set $Q_2 = Q_2 + P_2$*
   - *iii. Set $j = j - 1$.*

   3.3 If $i = 0$, then set $r = 1$.

   3.4 Else if $j = 0$, then set $r = 0$.
4. *Set $Q = Q_1 + Q_2$.*
5. *Output Q.*

Comparing to the countermeasures described in section 4.1, the proposed algorithm computes one more scalar multiplication with additional registers and control signal. But if the linear combination is required in the protocol, additional memory space is required to save k_1P_1 and k_2P_2, it means that no extra resource of hardware and computational time is employed. The interleaved scalar multiplication can be illustrated as Figure 4.1.

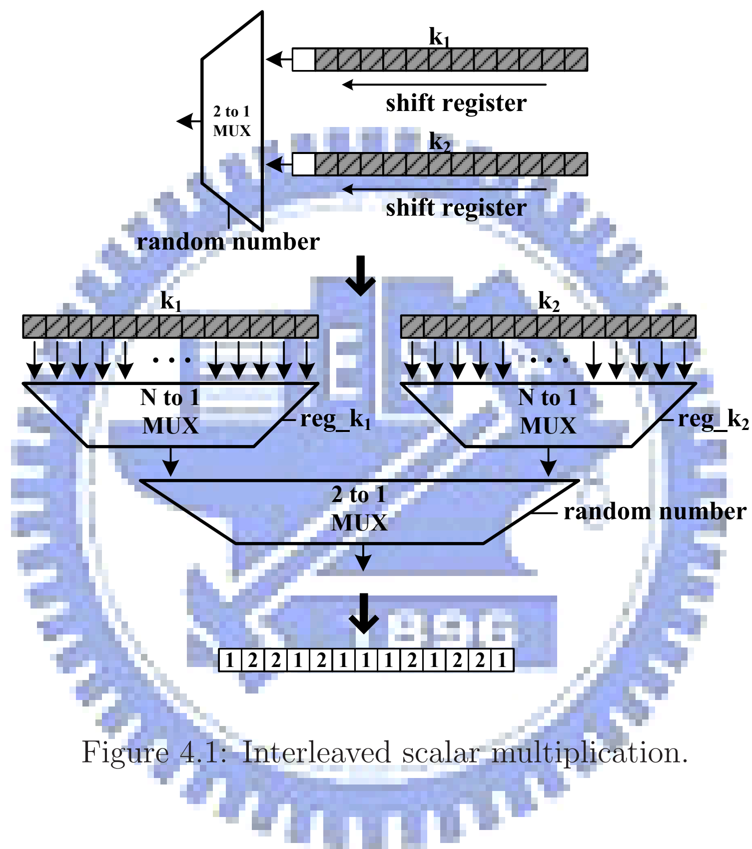


Figure 4.1: Interleaved scalar multiplication.

Chapter 5

Proposed Architectures

In this chapter, a bottom-up illustration of the proposed architectures is presented. The architecture of universal dual-field Galois field arithmetic unit is illustrated in section 5.1. The architecture of universal dual-field elliptic curve scalar multiplier is illustrated in section 5.2. And the architecture of universal dual-field elliptic curve arithmetic unit is illustrated in section 5.3. These designs are suitable for any field length which is shorter than the given ones. Both prime field, $GF(p)$, and binary extension field, $GF(2^m)$, applications are included. All of the background knowledge and mathematical theorems are mentioned in earlier chapters.

In this thesis, all of hardware implementation are coded in Verilog HDL (hardware description language) and synthesized on both application-specific integrated circuit (ASIC) and field-programmable gate arrays (FPGAs). The designs are implemented with UMC[1] 0.18-μm CMOS process and the Synopsys[2] Design Compiler, and the FPGA platform is Xilinx[3] Virtex-4 XC4VLX160.

5.1 Galois Field Arithmetic Unit

In an elliptic curve cryptosystem, four operations : modular addition, modular subtraction, Montgomery multiplication, and Montgomery division, are used. Therefore an area-efficient universal Galois field arithmetic unit (GFAU) is proposed to meet this re-

[1]United Microelectronics Corporation. The SoC solution foundry. http://www.umc.com

[2]Synopsys, Inc. The developer of EDA tools. http://www.synopsys.com

[3]Xilinx, Inc. The developer and fabless manufacturer of FPGAs. http://www.xilinx.com

quirement. The "universal" used here indicates that the field length is designed as an input, length smaller than 512 is permitted. Among these operations, the Montgomery division is the most complicated, and consumes most iterations. Thus, how to integrate the other operations into the hardware of Montgomery division algorithm is the most important topic in this section. Back to the Algorithm 3.9, the Montgomery division flow is showed below:

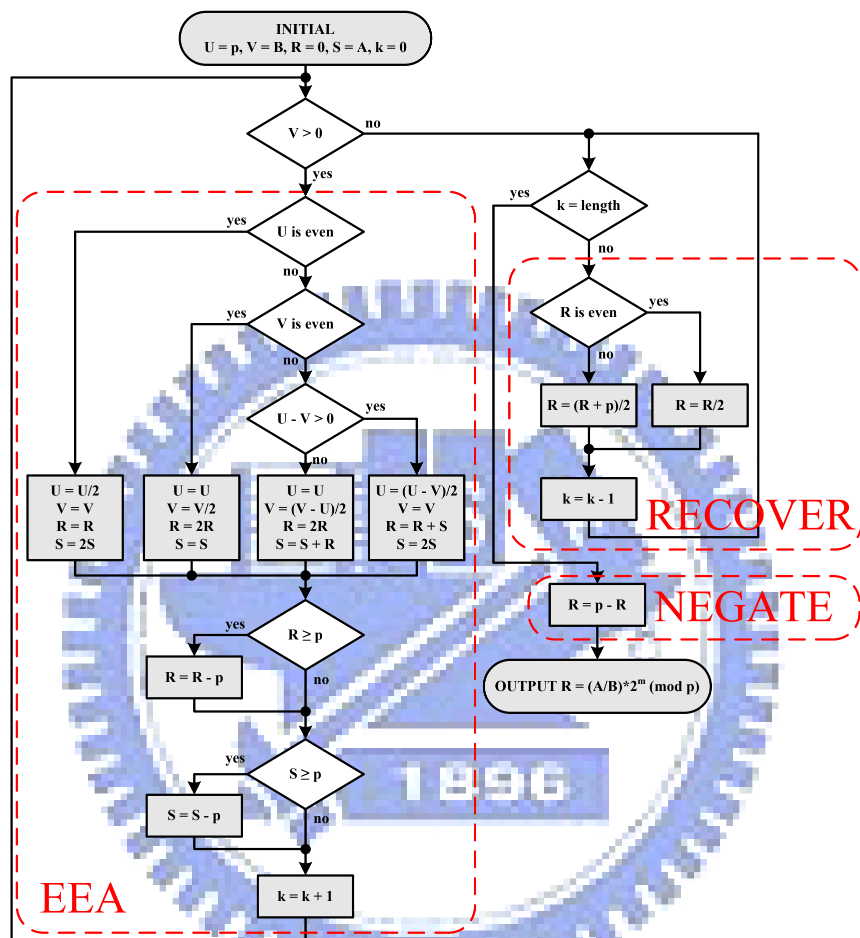


Figure 5.1: Flow chart of the Montgomery division algorithm.

The Montgomery division algorithm can be separated into three parts:

1. *EEA*: Bit-wise reduce U and V until $V = 0$.

2. *RECOVER*: Divide R by 2 until $k = m$.

3. *NEGATE*: Negate R to get the final result.

These three main parts are main states in the finite state machine of the the Montgomery divider. In part *EEA*, one subtracter is used to handle $(U - V)/2$ and $(V - U)/2$. The most significant bit (MSB) of $U - V$ determines if the result of $U - V$ should be negated. $R + S$, $2R$, and $2S$ can be combined with the conditional subtraction of R and S in step 2.5 and step 2.6 by one carry-save adder (CSA), three adder, and two multiplexer. Which is controlled by the MSB of $2R - p$ or $2S - p$ and $R + S - p$ respectively. With these elements, each iteration of part *EEA* can be accomplished by one cycle. In part *RECOVER*, $R = (R + p)/2$ simply reuse one adder. And in part *NEGATE*, $R = p - R$ reuses the only one subtracter. Note that in 2's complement number system, $-p$ can be derived by adding 1's complement of p with 1, that is:

$$-p = 2^m - p = (2^m - 1 - p) + 1 = \bar{p} + 1 \tag{5.1}$$

Since p is odd, $\bar{p}$ is always even. Adding $\bar{p}$ by 1 is simply turning the MSB of $\bar{p}$ from 0 to 1. Therefore negating p only requires bit-wise inversing p except the MSB. An incrementer is spared here. From above analysis, a 514-bit Montgomery divider totally takes one 514-bit CSA, four 514-bit CPA (including one 514-bit subtracter), one 514-bit negater, one 10-bit incrementer, and one 10-bit decrementer.

In Montgomery multiplication, looking back to algorithm 3.3, involves two main parts:

1. *MM*: Adding partial products and modular right shift.

2. *RECOVER*: Bound C in $GF(p)$.

Part *MM* executes step 3 in algorithm 3.3 and algorithm 3.4. Step 3 in algorithm 3.3 is implemented by a CSA and a carry propagation adder (CPA). But step 3 in algorithm 3.4 only requires the CSA. Part *RECOVER* take charge of step 4 in algorithm 3.3 with only changing the input of the CSA after part *MM*. Thus for the dual-field design, the hardware implementation of the Montgomery multiplication only contains one 514-bit

CSA and one 514-bit CPA. Besides, modular addition and modular subtraction simply utilize the existing elements of the Montgomery divider. The graphical illustration of the flow of the Montgomery multiplication is showed below: Merge the flow of the Montgomery

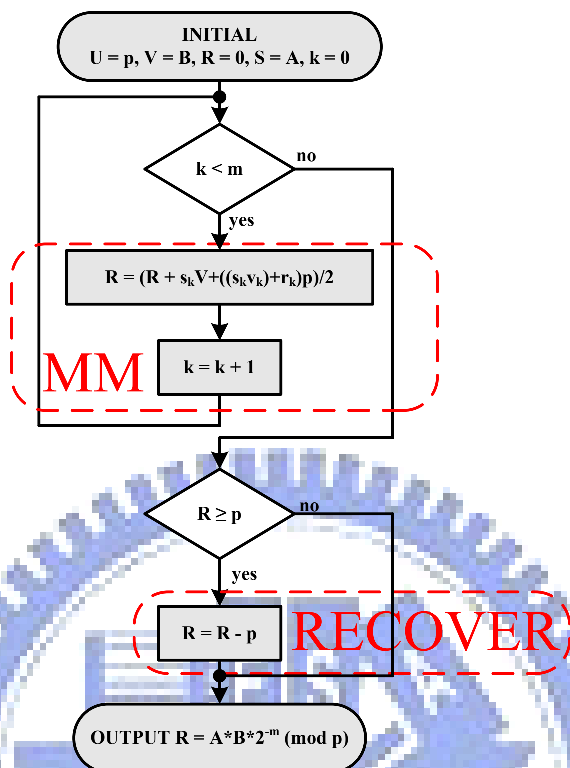


Figure 5.2: Flow chart of the Montgomery multiplication algorithm.

multiplication with the flow of the Montgomery division, part *MM* in the Montgomery multiplication and part *EEA* in the Montgomery division can be replaced by a new part named as *EEA_MM*. Part RECOVER and part NEGATE are retained in the GFAU. As a result, the GFAU consists of three main parts:

1. *EEA_MM*: Operation of EEA for the Montgomery division, MM for the Montgomery multiplication, modular addition, and modular subtraction.

2. *RECOVER*: Divide R by 2 until $k = m$ in the Montgomery division and bound C in $GF(p)$ in the Montgomery multiplication.

3. *NEGATE*: Negate R to get the final result in the Montgomery division and bound C in $GF(p)$ in the Montgomery multiplication.

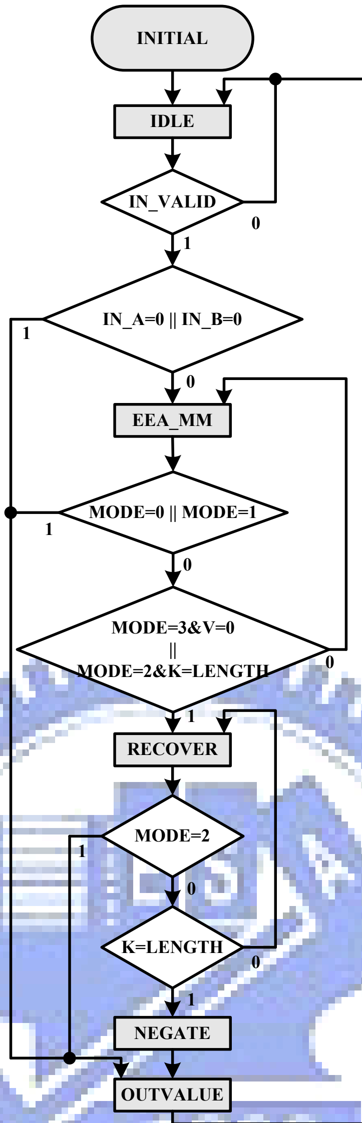


Figure 5.3: Finite states transfer chart of the GFAU.

The finite states transfer chart is described in Figure 5.3. In state *IDLE*, if IN_A or IN_B is zero, state machine directly transfers to state *OUTVALUE*. In state *EEA_MM*, if addition (MODE 0) and subtraction (MODE 1) are demanded, state machine transfers to state *OUTVALUE* and output the result. While multiplication or division is demanded, state machine transfers to state *RECOVER* when register V equals to zero or counter K equals to LENGTH respectively. Division (MODE 3) is the only one operation that requires state *NEGATE*. Therefore, if MODE is 2 which means multiplication, the state machine should directly transfer from *RECOVER* to *NEGATE* with doing anything. State *NEGATE* transfers to state *OUTVALUE* when K equals to LENGTH given from

input which denotes termination of the division.

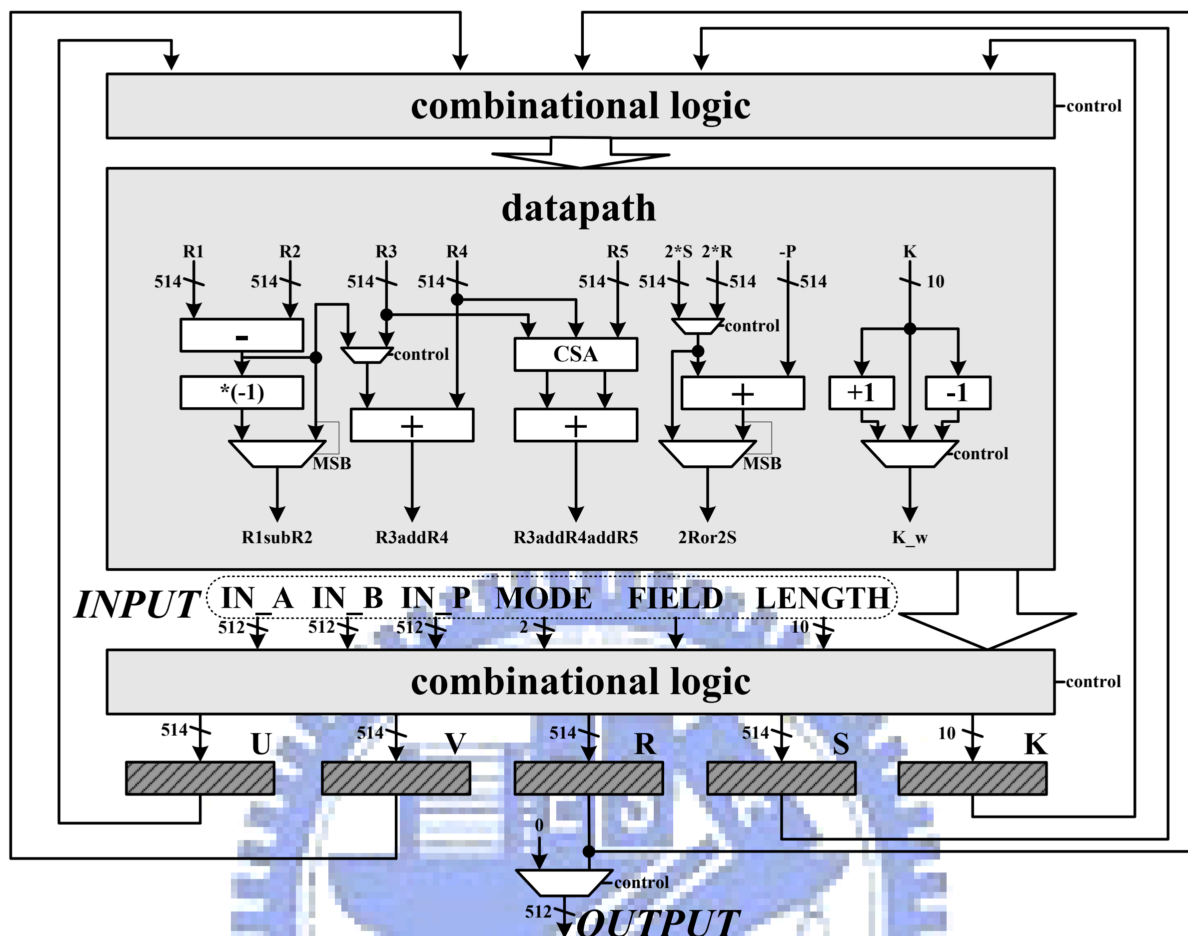


Figure 5.4: Architecture of the GFAU.

The complete architecture of the GFAU is showed above in Figure 5.4. The control signal is generated by the finite state machine in Figure 5.3. All inputs are stored through some combinational logic controlled by the finite state machine in four main registers: U(514-bit), V(514-bit), S(514-bit), K(10-bit) and two 1-bit register. Values are pulled out to one level of combinational logic , then temporary wires named as R1, R2, R3, R4, R5, $2 \times R$, and $2 \times S$ are produced. These values get through the datapath and another level of combinational logic and update the value of the registers at each rising edge of the clock signal. Recall the hardware requirement of the Montgomery division mentioned before, the chief advantage of the GFAU is revealed: with almost the same hardware requirement, just changing the control signal, the GFAU can do the four fundamental operations of arithmetic over the Galois field.

The implementation results of the proposed universal Galois field arithmetic unit is given in Table 5.1. It shows the synthesized gate count at 133MHz and shows the area and speed results on FPGAs. Each number of gates in the field Gatecount consists of two parts which are non-combinational logic and combinational logic respectively. It doubles when the bit length doubles. The area is approximately in proportion to the bit length m of the field.

Table 5.1: Synthesized results for proposed universal dual-field Galois field arithmetic unit on ASIC and FPGA design.

Length	ASIC		FPGA	
	Area (Gatecount)	Frequency (MHz)	Slice (Slice+Slice FF)	Frequency (MHz)
128-bit	23.6k (18.9k+4.7k)	133	3764 + 687	52.5
256-bit	47.4k (38.2k+9.2k)		7363 + 1360	35.2
512-bit	97.3k (79.2k+18.1k)		17131 + 2744	20.8

5.2 Elliptic Curve Scalar Multiplier

A universal elliptic curve scalar multiplier (ECSM) simply computes a point P multiplied by a scalar k using iterative point doubling and point addition which is illustrated in section 2.2. There are four main part in the flow of the ECSM which results in four state in the controlling finite state machine showed below:

1. *ItoM*: Convert the values from integer domain to Montgomery domain except the scalar k.

2. *DOUBLE*: Point doubling calculation.

3. *ADD*: Point addition calculation.

4. *MtoI*: Convert the output x and y back to integer domain representation.

The following figure is a flow chart of the ECSM:

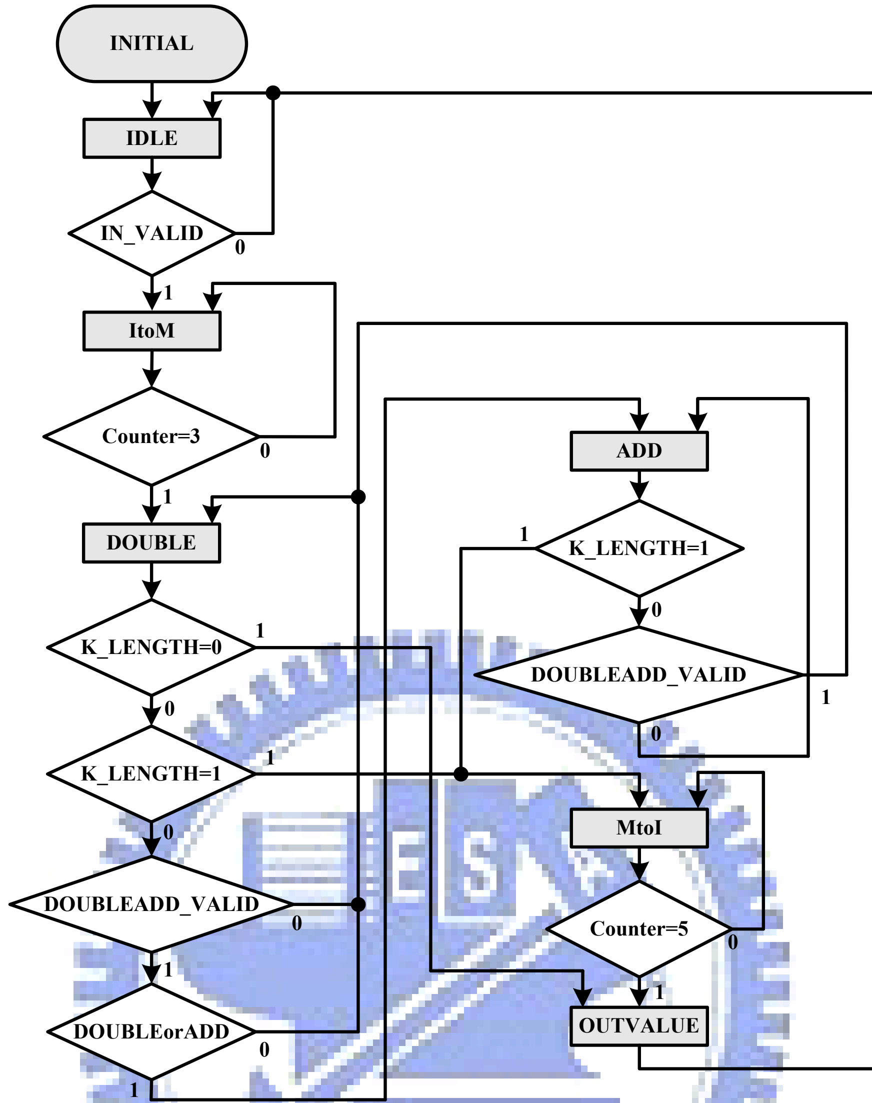


Figure 5.5: State transition chart of the ECSM.

According to section 3.4, the integer domain to Montgomery domain conversion can be done by a Montgomery division and the Montgomery domain to integer domain conversion can be done by a Montgomery multiplication. An additional counter determines which one of the values, including input IN_X1, input IN_Y1, coefficient IN_A, output OUT_X, and output OUT_Y, is involved in the Montgomery multiplication or the Montgomery division. Afterward, Algorithm 2.1 is used to construct the scalar multiplication. The state transits between state *DOUBLE* and state *ADD* according to each bit of the scalar. If the length of the scalar is 0, which indicates that the output is the point at the infinity, the state will directly transit to state *OUTVALUE* and return to state *IDLE*. If the length

of the scalar is 1, which indicates the end of the double and add sequence, the finite state machine will transit to state *MtoI*.

The architecture consists of five main blocks: registers, combinational logics, GFAU, FSM, and DA_FSM. Four 512-bits inputs are stored in register reg_IN_X, reg_IN_Y, reg_IN_A, and reg_IN_P. Coordinates of the intermediate point P3 are stored in reg_P3_X and reg_P3_Y. λ is stored in reg_LAMBDA. reg_TEMP and reg_GFAU_OUT make it feasible to use only one GFAU without any additional addition or subtraction. Besides, the value of x_3 in the next double or add iteration is changed before y_3, however the original value of x_3 is demanded during the calculation of y_3, therefore a register is added to store the original value of x_3.

Looking back to section 2.2.2 and 2.2.3, different addition, subtraction, multiplication, and division exist in the mathematical representation of x_3 and y_3. All these operation are executed by only one GFAU since they are time-dependent operations. Every addition and every subtraction occupies 1 state in a finite state machine called DA_FSM that controls the GFAU. Because the state transition flow chart is quite complicated, it occupies almost 38% area of the whole ECSM. That's really a huge percentage. The area consumption percentage is demostated in the following pie graph: Table 5.2 shows the synthesized

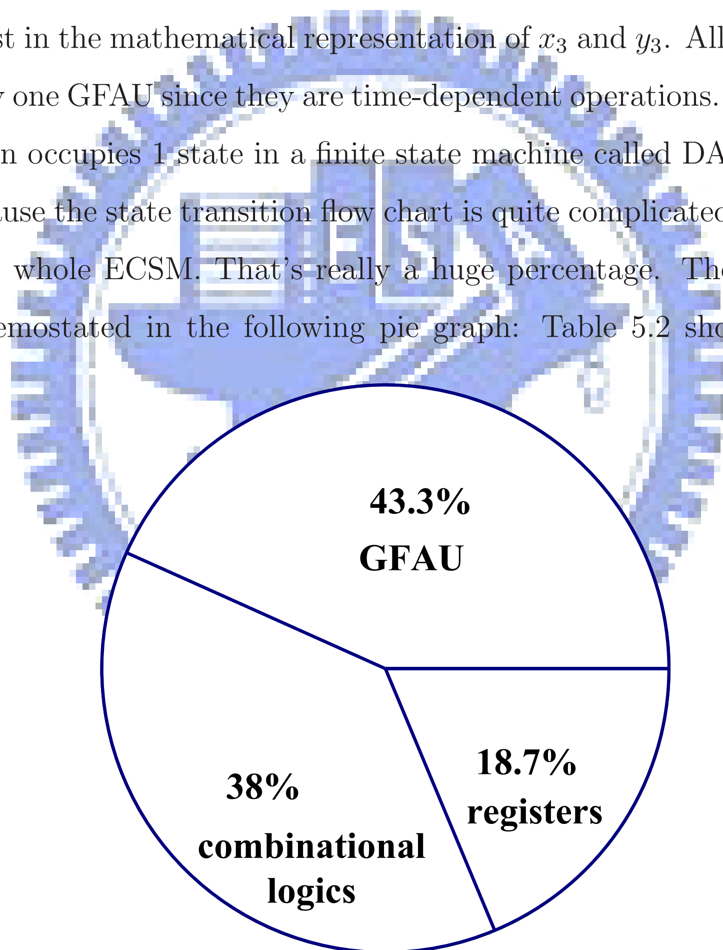


Figure 5.6: Pie graph of the area consumption of the ECSM.

result of the proposed ECSM. Figure 5.9 shows the architecture of the proposed ECSM.

Table 5.2: Synthesize results for proposed ECSM.

	ASIC		FPGA	
Bit-length	Gatecount (Gates)	Frequency (MHz)	Slice (Slice+Slice FF)	Frequency (MHz)
512-bit	225k (171k+54k)	133	34384 + 8505	20.48

It is arranged at the second last page of this chapter.

5.3 SPA-Resistant Elliptic Curve Arithmetic Unit

There is not only scalar multiplication in elliptic curve cryptographic protocols. Point addition and point addition after scalar multiplication play important roles in elliptic curve cryptography. For example, $k_1P_1 + k_2P_2$ is the most important part of the elliptic curve digital signature algorithm (ECDSA). If there is only the elliptic curve scalar multiplier available, performing the operations above takes extra memory space and extra memory access time to load and store the scalar multiplication results. Besides, power analysis have been the most important threat in recent years. A universal SPA&DPA-resistant elliptic curve arithmetic unit is proposed in this section.

With three more input ports: IN_X2 (512-bit), IN_Y2 (512-bit) and IN_K2 (513-bit); three more input buffer: reg_IN_X2 (512-bit), reg_IN_Y2 (512-bit), and reg_IN_K2 (513-bit); two more intermediate point register : reg_P4_X and reg_P4_Y and modification of the main finite state machine, the ECAU is designed without additional GFAU.

If one design is meant to act as an elliptic curve cryptographic co-processor, it should be able to handle the Galois operations and make it more easier for a software engineer to work with. In the proposed ECAU, to make the most use of the existing control signal, the operand of a modular operation should be converted to Montgomery domain before the operation and converted back to integer domain after the operation. Compared with the design specific for modular operations, the proposed ECAU requires more time to do modular operations. But when the ECAU is used to accelerate the process of an elliptic cryptographic protocol, the slight degradation in computational time of the modular operations can be ignored. With the proposed ECAU, all the software engineer

has to do is to arrange the order of the input and output of the ECAU.

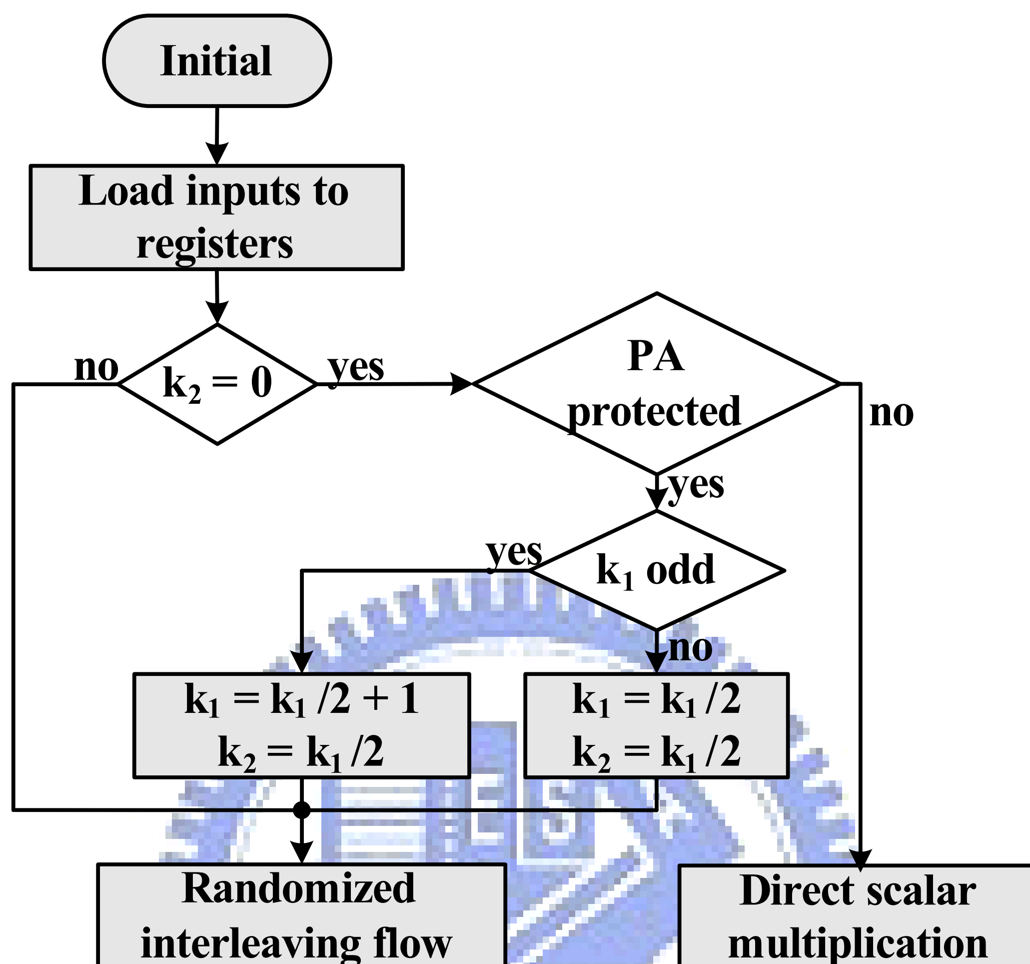
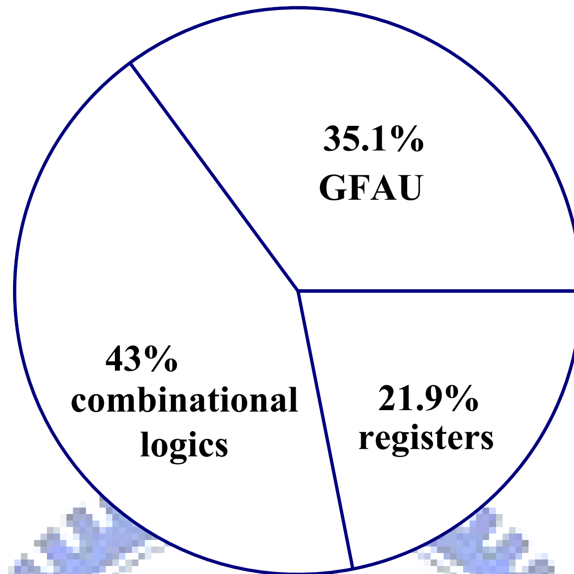


Figure 5.7: Flow chart of the scalar determination scheme.

Figure 5.7 shows the decision flow of the scalar k_1 and k_2. If the operation contains two scalar multiplication and one point addition, the ECAU will randomly interleave these two scalar multiplication. If there is only one scalar multiplication demanded, a option is given to decide if these scalar multiplication is power analysis protected. If "yes", the scalar k_1 will be divided into k_1 and k_2 by half and randomly interleaved. If "no", the ECAU will execute direct scalar multiplication. Figure 5.10 demonstrates the architecture of the proposed ECAU. It is arranged at the last page of this chapter.The area consumption ratio is demonstrated in Figure 5.8.

From the pie graph, the additional registers mentioned above contributes to the main part of the increase of area. The modified control signal also increase the area.

Table 5.3 shows the synthesized result of the proposed ECAU.



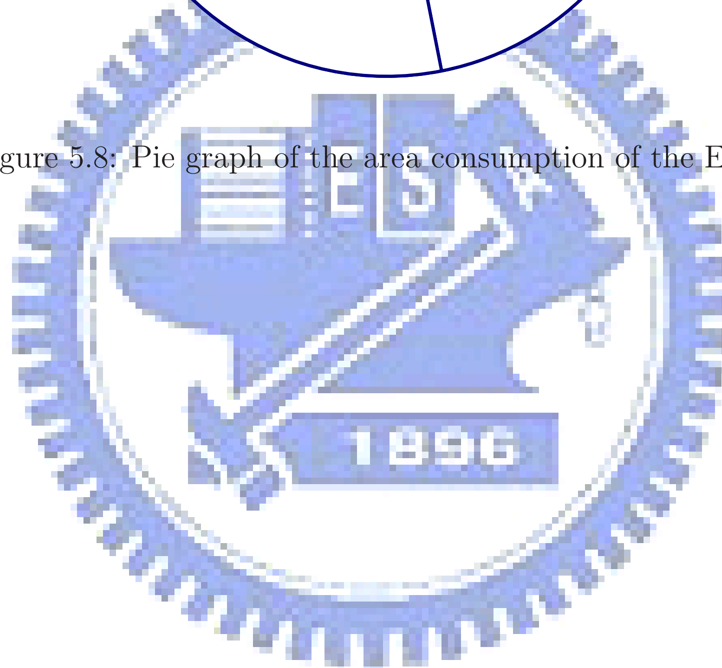


Figure 5.8: Pie graph of the area consumption of the ECAU.

Table 5.3: Synthesize results for proposed ECAU.

	ASIC		FPGA	
Bit-length	Gatecount (Gates)	Frequency (MHz)	Slice (Slice+Slice FF)	Frequency (MHz)
512-bit	277.5k (198.5k+79k)	133	54376 + 16319	17.76

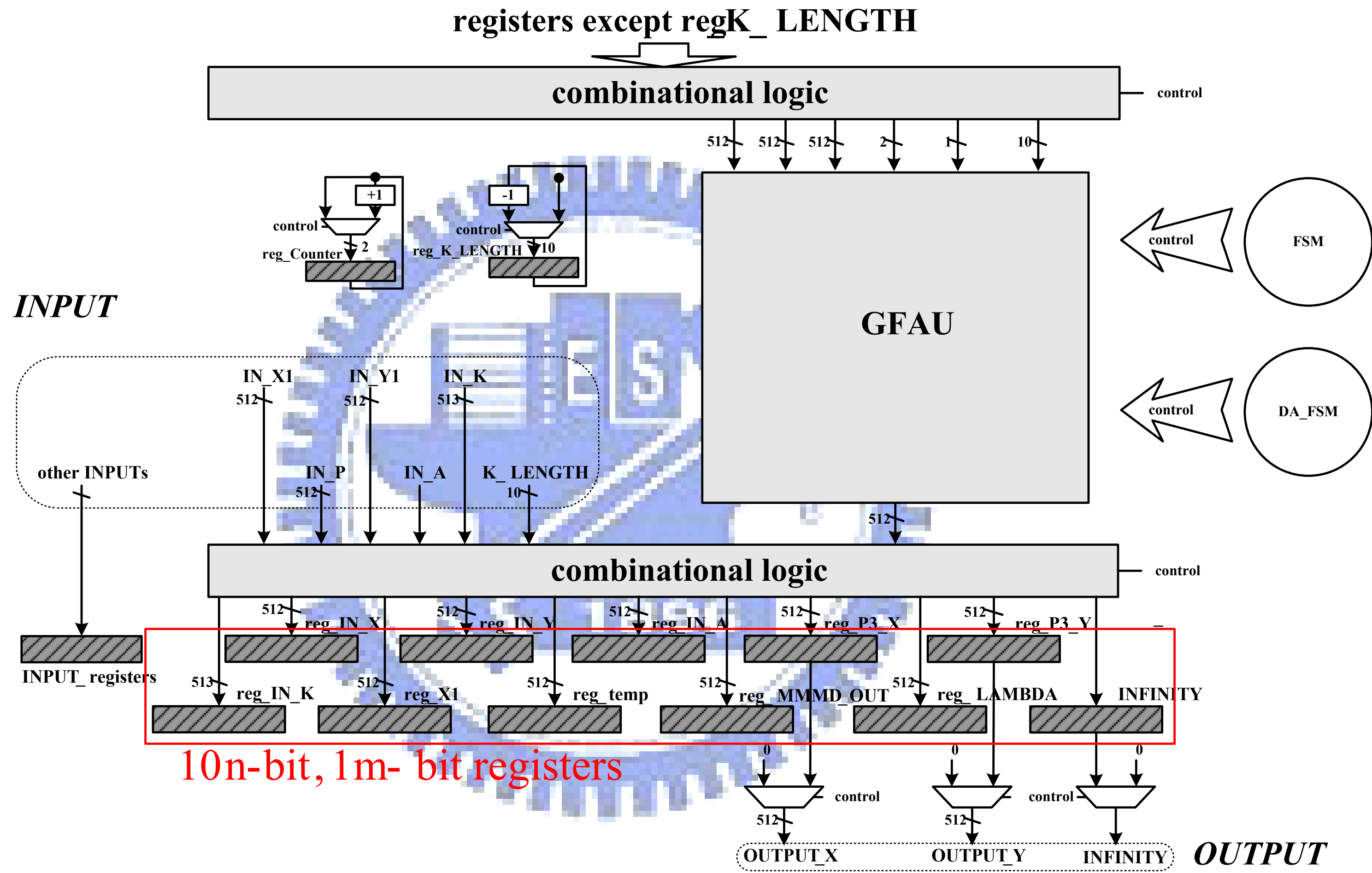


Figure 5.9: Architecture of the proposed ECSM.

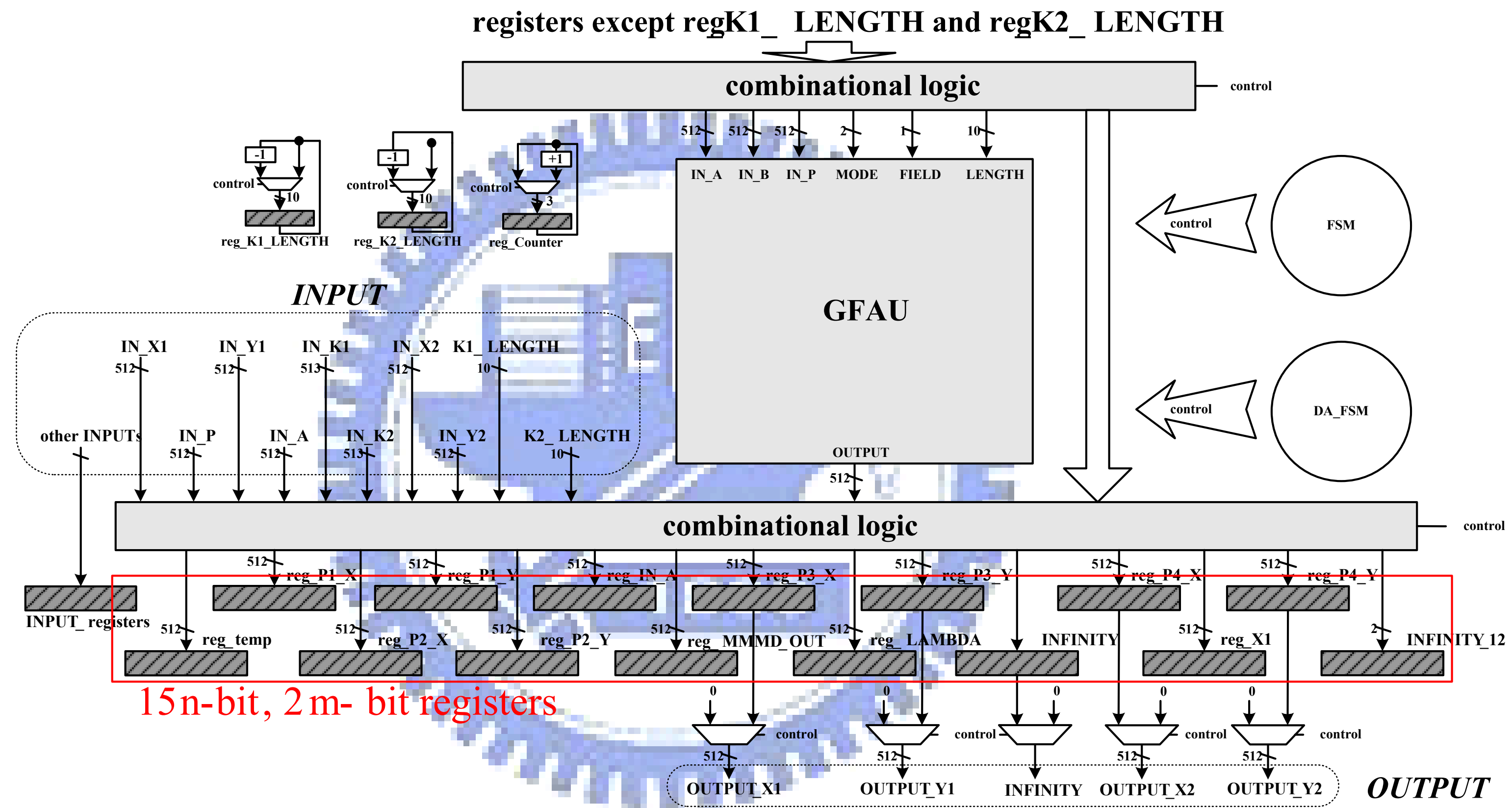


Figure 5.10: Architecture of the proposed ECAU.

Chapter 6

Implementation Results

Solutions for elliptic curve arithmetics in both software and hardware are given in this work. The software simulation environment is constructed in C programing languages. The design and test consideration are discussed in Chapter 6.1. The hardware implementation results and design flow are described in Chapter 6.2. The RTL synthesizer uses Synopsys[1] Design Compiler for ASIC and Xilinx XST or Synplicity[2] Synplify Pro for FPGA. The Cadence[3] Encounter is used for backend Auto Place & Route implementation.

6.1 Design and Test Consideration

The hardware is designed to accelerate the operations on elliptic curves and it deals with different field parameters using Montgomery technique. The main part in hardware is the point operation on elliptic curves and the implementation of scalar multiplication on hardware uses only Double-and-Add algorithm.

The Verilog code for this design was generated using the parameterized module for different values of m. The test patterns are generated randomly by software. The verification for the design uses not only hardware-software co-simulation but also confirms with the examples of NIST[4] publications for more confidence. No special technique is introduced in the FPGA implementation.

[1]Synopsys, Inc. http://www.synopsys.com/

[2]Synplicity, Inc. http://www.synplicity.com/

[3]Cadence Design Systems, Inc. http://www.cadence.com/

[4]National Institute of Standards and Technology. http://www.nist.gov/

6.2 Implementation Results and Comparison

6.2.1 ASIC Implementation

Table 6.1 shows the ASIC synthesized result comparison between the proposed GFAU and the others. The proposed universal dual-field GFAU consumes about 75% of the total gatecount of the universal dual-field Montgomery multiplier and the universal dual-field Mongomery divider proposed in [13]. In [33], a dual-field modular divider is proposed. But it's modular divider requires one more Montgomery multiplier to convert the result back into the Montgomery domain.

Table 6.1: ASIC synthesis results comparison

Length	Freq.(MHz)	Area(Gatecount)			
		ModDiv [33]	MontDiv [13]	MontMul [13]	GFAU
128-bit	100	22.8k	20.8k	8.3k	23.65k
256-bit	100	45.6k	42.1k	16.3k	47.4k
512-bit	100	N/A	N/A	32.1k	97.3k

[1] GFAU can be synthesized at clock frequency 133MHz.

[2] GFAU is synthesized with UMC 0.18-μm CMOS process.

[3] Modular divider in [33] is synthesized with 0.5-μm CMOS process.

[4] Montgomery divider and Montgomery multiplier in [13] are synthesized with UMC 0.18-μm CMOS process.

In this work, a universal dual-field elliptic curve scalar multiplier and a universal dual-field elliptic curve arithmetic unit are proposed. The most important part of them is the proposed area-efficient GFAU. The ASIC synthesized gatecount are $226K$ and $277.5K$ respectively at 133MHz clock frequency using TSMC 0.18μm CMOS process. It takes 1.93ms to complete a 192-bit prime field elliptic curve multiplication using the proposed ECSM. To make a fair comparison, we multiply the GF(P_{192}) equivalent gatecount by elliptic curve multiplication computational time. The value of ECSM and ECAU are 163.54(gates×ms) and 401.68(gates×ms). It's better then previous works.

Table 6.2 shows a comparison for the ASIC performance of scalar multiplication.

Table 6.2: Elliptic Curve Scalar Multiplication ASIC Performance Comparison

Author	A. Satoh [34]	G. Z. Lu [35]	Y. J. Liu [13]	ECSM	ECAU
Field	$P_{192}/2^{160}$	$P_{192}/2^{192}$	$P_{256}/2^{256}$	$P_{512}/2^{512}$	$P_{512}/2^{512}$
Process	.13μm	.25μm	.18μm	.18μm	.18μm
Area(Gatecount)	118k	26.7k	292.5k	225k	277k
Freq.(Mhz)	137.7	285.7	75	133	133
EC mult.(ms)	1.44/0.19	9.75/6.75	3.3	1.93	3.86
P_{192} Equivalent Area×EC mult. (gatecount×ms)	172.8	260.3	965.25	163.54	401.68
Coordinate	projective	modified Jacobian	affine	affine	affine
Multiplication	multiplier based	systolic radix-2	radix-2	radix-2	radix-2
Division	Fermat's little theorem	Fermat's little theorem	Mont. division	Mont. division	Mont. division
Note	64-bit multiplier	8PEs with w = 8bits	universal architecture	universal architecture	SPA resistant

In [13], a novel Montgomery division algorithm is proposed and utilized in the implementation of a universal dual-field elliptic curve scalar multiplier. The Montgomery multiplier and the Montgomery divider occupy most of the area and no area reuse technique is introduced in his work. Therefore, the gatecount is 292.5k when the field length is 256. The execution time for computing kP in $GF(P_{192})$ is average 3.3 ms.

In work [35], the design uses Fermat's Little Theorem for the modular inversion operation. However, it is not considered efficient in a large field design since the computation complexity increases significantly.

Besides, the work [34] shows a great performance using a elliptic curve cryptographic processor. It has a optimized multiplier-based Montgomery multiplier and uses projective coordinates to avoid inversion operations. In scalar multiplication, it uses software NAF method to reduce the number of 1 terms in k.

In software simulation on C on Intel Core 2 Duo E7200 and 2G RAM, it takes around 17 seconds averagely to do scalar multiplication once. The simulation results below show

significant improvement on the computation time for scalar multiplication.

In the auto place and route stage, we face a big problem. The data path in the proposed design is 512 bit, there are too many wires in it. Therefore, the CAD tool cannot place them without negative timing slacks and design rule violations. We have tried it on UMC 0.18μm 1P5M, TSMC 0.18μm 1P5M and UMC 90nm 1P9M CMOS processes and enlarge the timing margin. But all these effort are ineffective. We have also tried 256-bit design, but it doesn't work either. It indicates that the parallel architecture is not feasible with currently available APR tools. We suggest to use word-based architecture like [34] to solve this question.

6.2.2 FPGA Implementation

The FPGA synthesis result is showed in Table 6.3:

Table 6.3: 512-bit FPGA synthesis results.

	GFAU	UESM	UEAU
Slice	17131	34384	54376
Slice Flip Flop	2744	8505	16319
4 input LUT	33074	65904	94596
Clock rate(MHz)	20.8	20.48	17.75

C. J. McIvor proposed a multiplier-based architecture in [36]. With cascaded 16×16-bit multipliers, it only requires 32 clock cycles to accomplish one 256×256-bit Montgomery modular multiplication. It performs fast operation with relatively high area consumption.

The proposed architectures don't have good area and timing performance in FPGA simulation. In our judgement, the highly reused hardware improve the gatecount synthesized by Synopsys design compiler, but in Xilinx ISE, the larger MUXs consume much more slices than the datapath does. So the result of FPGA synthesis shows more slices and longer critical path.

Table 6.4: Elliptic Curve Scalar Multiplication FPGA Performance Comparison

Author	C. J. McIvor [36]	S. B. Ors [30]	Y. J. Liu [13]	ECSM	ECAU
Field	2^{256}	P_{160}	$P_{256}/2^{256}$	$P_{512}/2^{512}$	$P_{512}/2^{512}$
Platform	XC2VP125	XV1000E	XC2V8000	XC4VLX160	XC4VLX160
Slices	15755	N/A	18146	34384	54376
Freq.(Mhz)	39.46	91.3	18.768	20.48	17.75
EC mult. (ms)	3.86 (256-bit)	14 (160-bit)	18.77 (192-bit)	1.93 (192-bit)	3.86 (SPA)
Coordinate	projective	modified Jacobian	affine	affine	affine
Multiplication	multiplier based	systolic radix-2	radix-2	radix-2	radix-2
Division	ModDiv	Fermat's little theorem	Mont. division	Mont. division	Mont. division
Note	16-bit multipliers		Not optimized	scalar multiplier	SPA resistant

Chapter 7

Conclusion and Disscusion

A SPA-resistant solution in hardware to the operations on elliptic curves in both $GF(p)$ and $GF(2^m)$ is given in this thesis. The proposed architecture is implemented over affine coordinate using a highly integrated Galois field arithmetic. Not only elliptic curve operations, but also modular arithmetics are provided, thus a software engineer can easily use it to accelerate all kinds of elliptic curve protocols. According to comparisons above, the proposed GFAU gives an advantage in area-latency combined comparison, which gives an opportunity to put affine coordinate computations back into implementing consideration.

Besides, the proposed SPA-resistant algorithm randomly interleaves two scalar multiplication $k_1P_1 + k_2P_2$. In this way, least hardware overhead is added than other know countermeasures. This method may also be resistant to DPA, but it has to be confirmed by further simulation. For more protection, this algorithm can also be combined with other countermeasures like windows-NAF method.

However, a big problem occurs in the auto place and route stage. The data path in the proposed design is 512 bit, there are too many wires in it. Therefore, the CAD tool cannot place them without negative timing slacks and design rule violations. We have tried it on UMC 0.18μm 1P5M, TSMC 0.18μm 1P5M and UMC 90nm 1P9M CMOS processes and enlarge the timing margin. We have also tried a 256-bit version. But all these effort are ineffective.

We suggest to use word-based architecture to solve this problem. The Montgomery multiplier can be easily modified into a pipelined word-based architecture, but its hard

to pipeline the Montgomery divider since there exist data dependency between two consecutive iterations. So it may be worth-researching to develop a pipelined Montgomery divider architecture.

Bibliography

[1] *Advanced Encryption Standard (AES)*, FIPS PUBS Std. 197, 2001.

[2] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, 1976.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[4] T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of CRYPTO 84 on Advances in cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 10–18.

[5] J. Cowie, B. Dodson, R. M. Elkenbracht-Huizing, A. K. Lenstra, P. L. Montgomery, and J. Zayer, "A world wide number field sieve factoring record: On to 512 bits." in *ASIACRYPT '96: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security*, 1996, pp. 382–394.

[6] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology - CRYPTO '85*, ser. Lecture Notes in Computer Science, H. C. Williams, Ed., vol. 218. Springer-Verlag, 1986, pp. 417–426.

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

[8] *Recommendation on Key Management*, NIST Special Publications Std. 800-57, 2005.

[9] *Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, ANSI Std. X9.62, 2005.

[10] B. J. Paul Kocher, Joshua Jaffe, "Differential power analysis," in *Advances in Cryptology - Crypto 99 Proceedings*, ser. Lecture Notes in Computer Science, M. Wiener, Ed., vol. 1666. Springer-Verlag, 1999, pp. 388–397.

[11] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptography," in *CHES'99*, ser. Lecture Notes in Computer Science, Ç. K. Koç and C. Paar, Eds., vol. 1717. Springer-Verlag, 1999, pp. 292–302.

[12] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, April 1985.

[13] Y. J. Liu, "An implementation of universal dual-field scalar multiplication on elliptic curve cryptosystems." Master's thesis, National Chiao Tung University, 2007.

[14] N. Koblitz, *A course in number theory and cryptography*. New York, NY, USA: Springer-Verlag New York, Inc., 1987.

[15] J. H. Silverman, *The Arithmetic of Elliptic Curves*. New York, NY, USA: Springer-Verlag New York, Inc., 1986.

[16] A. M. H. Cohen and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," in *Advances in Cryptology-Asiacrypt'98*, ser. Lecture Notes in Computer Science, K. Ohta and D. Pei, Eds., vol. 1514. Springer-Verlag, 1998, pp. 51–65.

[17] D. V. Chudnovsky and G. V. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests," in *Advances in Applied Math.*, vol. 7. Academic Press, Inc. Orlando, FL, USA, 1986, pp. 385–434.

[18] J. A. Solinas, "Efficient arithmetic on koblitz curves," *Des. Codes Cryptography*, vol. 19, no. 2-3, pp. 195–249, 2000.

[19] F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains," *Informatique théorique et Applications*, vol. 24, pp. 531–544, 1990.

[20] Çetin Kaya Koç, T. Acar, and B. S. Kaliski, Jr., "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, June 1996.

[21] C. D. Walter, "Montgomery exponentiation needs no final subtractions," *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, October 1999.

[22] Ç. K. Koç and T. Acar, "Fast software exponentiation in $GF(2^k)$," in *ARITH '97: Proceedings of the 13th Symposium on Computer Arithmetic (ARITH '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 225.

[23] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Info. Theory*, vol. IT-22, pp. 644–654, November 1976.

[24] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1998, vol. 2, ch. Seminumerical Algorithms.

[25] J. Stein, "Computational problems associated with racah algebra," *Joural of Computational Physics*, pp. 397–405, January 1967.

[26] B. S. K. Jr., "The montgomery inverse and its applications," *IEEE Transactions on Computers*, vol. 44, no. 8, pp. 1064–1065, Auguest 1995.

[27] S. C. Shantz, "From euclid's gcd to montgomery multiplication to the great divide," Sun Microsystems laboratories, Tech. Rep. TR-2001-95, June 2001.

[28] N. Takagi, "A vlsi algorithm for modular division based on the binary GCD algorithm," *IEICE Trans. Fundamentals*, vol. E81-A, no. 5, pp. 724–728, May 1998.

[29] A. Daly, W. P. Marnane, T. Kerins, and E. M. Popovici, "Fast modular division for application in ecc on reconfigurable logic." in *FPL*, 2003, pp. 786–795.

[30] L. B. S. B. Ors and J. Vandewalle, "Hardware implementation of an elliptic curve processor over gf(p)," in *14th IEEE International Conference on the Application-Specific Systems, Architectures, and Processors (ASAP03)*, 2003, pp. 433–443.

[31] K. Okeya and T. Takagi, "The width-w naf method provides small memory and fast elliptic scalar multiplications secure against side channel attacks," in *CT-RSA 2003*, ser. Lecture Notes in Computer Science, M. Joye, Ed., vol. 2612. Springer-Verlag, 2003, pp. 328–343.

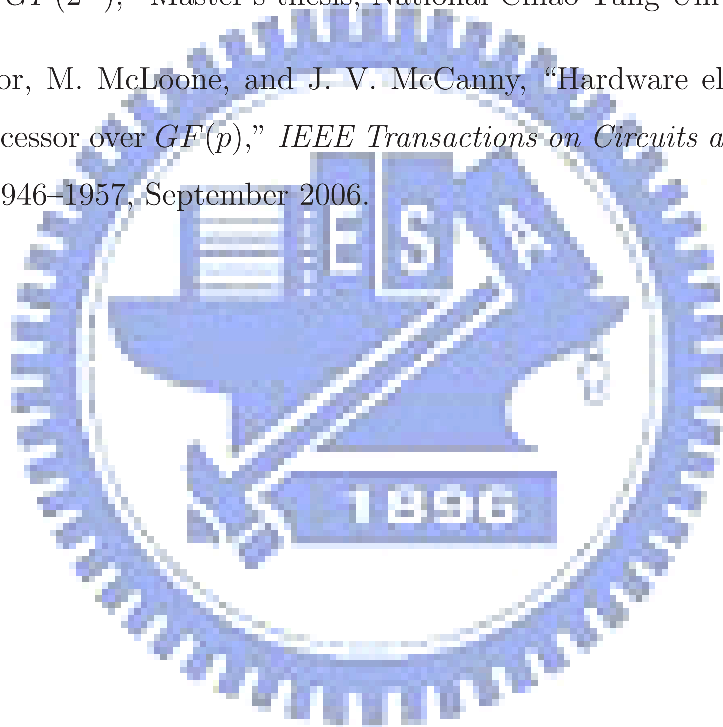
[32] E. A. D. T. S. Messerges and R. H. Sloan, "Power analysis attacks of modular exponentiation in smartcards," in *CHES'99*, ser. Lecture Notes in Computer Science, Ç. K. Koç and C. Paar, Eds., vol. 1717. Springer-Verlag, 1999, pp. 144–157.

[33] L. A. Tawalbeh, A. F. Tenca, S. Park, and C. K. Koç, "Use of elliptic curves in cryptography," in *Thirty-Eighth Asilomar Conference on Signals, Systems, and Computers*, vol. 1, November 2004, pp. 483–487.

[34] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Trans. Comput.*, vol. 52, no. 4, pp. 449–460, 2003.

[35] G. Z. Lu, "Hardware implementation of elliptic curve cryptosystem over finite fields $GF(p)$ and $GF(2^m)$," Master's thesis, National Chiao Tung University, 2004.

[36] C. J. McIvor, M. McLoone, and J. V. McCanny, "Hardware elliptic curve cryptographic processor over $GF(p)$," *IEEE Transactions on Circuits and Systems*, vol. 53, no. 9, pp. 1946–1957, September 2006.



作者簡介

姓名：曾知業

出生：台北市

學歷：民權國小、介壽國中、延平中學

90.9 ~ 95.6 國立交通大學電子工程學系

95.9 ~ 97.11 國立交通大學電子研究所系統組