2005

Efficient implementation of elliptic curve cryptography.

Bijan Ansari
University of Windsor

Efficient Implementation of Elliptic Curve Cryptography

by

Bijan Ansari

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

Windsor, Ontario, Canada
2004

Your file Votre référence
ISBN: 0-494-09774-4
Our file Notre référence
ISBN: 0-494-09774-4

Efficient Implementation of Elliptic Curve Cryptography

by

Bijan Ansari

APPROVED BY:

B. Zhou
Department of Mechanical and Materials Engineering

M. Ahmadi
Department of Electrical and Computer Engineering

H. Wu, Advisor
Department of Electrical and Computer Engineering

M. A. Sid-Ahmed, Chair of Defense
Chair, Department of Electrical and Computer Engineering

University of Windsor

May 13, 2004

Abstract

Elliptic Curve Cryptosystems (ECC) were introduced in 1985 by Neal Koblitz and Victor Miller. Small key size made elliptic curve attractive for public key cryptosystem implementation. This thesis introduces solutions of efficient implementation of ECC in algorithmic level and in computation level.

In algorithmic level, a fast parallel elliptic curve scalar multiplication algorithm based on a dual-processor hardware system is developed. The method has an average computation time of $\frac{n}{3}$ *Elliptic Curve Point Addition* on an n-bit scalar. The improvement is n *Elliptic Curve Point Doubling* compared to conventional methods. When a proper coordinate system and binary representation for the scalar k is used the average execution time will be as low as n *Elliptic Curve Point Doubling*, which makes this method about two times faster than conventional single processor multipliers using the same coordinate system.

In computation level, a high performance elliptic curve processor (ECP) architecture is presented. The processor uses parallelism in finite field calculation to achieve high speed execution of scalar multiplication algorithm. The architecture relies on compile-time detection rather than of run-time detection of parallelism which results in less hardware. Implemented on FPGA, the proposed processor operates at $66MHz$ in $GF(2^{167})$ and performs scalar multiplication in $100\mu Sec$, which is considerably faster than recent implementations.

To the young man who was me, and perished under fanaticism

Acknowledgments

I would like to thank my advisor Dr. Huapeng Wu for introducing me to elliptic curves and giving me all that advice, help, and support throughout this research work. I also want to thank professor Majid Ahmadi for his expert guidance and support, and Dr. B. Zhou for reviewing this work.

Special thanks is due to professor Maher Sid-Ahmed who supported me by all means through my study.

I am also grateful to my colleagues and friends, Katy Modaressi, Muqeeth Seyed Ali and Kevin Banovic, for their time and friendship.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
BPWS	Bit Parallel Word Serial
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DH	Diffie-Hellman
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
EC	Elliptic Curve
EUA	Extended Euclidean Algorithm
ECADD	Elliptic Curve Addition operation
ECC	Elliptic Curve Cryptography, Elliptic Curve Cryptosystem
ECDBL	Elliptic Curve Doubling operation
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme

ECMQV	Elliptic Curve Menezes-Qu-Vanstone Protocol
ECP	Elliptic Curve Processor
FF	Finite Field
F_2^m	Galois Field of 2^m
FIPS	Federal Information Processing Standards
F_p	Galois Field of prime p
FPGA	Field Programmable Gate Array
GF	Galois Field
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IOB	Input/Output Block
ISO	International Standard Organization
IT	Information Technology
LB	Lower Bound
LSB	Least Significant Bit
NAF	Non-Adjacent form
NIST	National Institute of Standards in Technology
ONB	Optimal Normal Basis
PB	Polynomial Basis
RISC	Reduced Instruction Set Computer
RSA	Rivest,Shamir, Adleman
RTL	Register Transfer Level
SCA	Side Channel Attack
SD	Signed-Digit
SIMD	Single Instruction Multiple Data
SoC	System on Chip
SSL	Secure Socket Layer
UB	Uppen Bound

Chapter 1

Introduction

1.1 Motivation

With the rapid and expansive growth of Internet, the need for communication security is increasing. Financial institutions, manufacturing plants and general public use Internet to exchange private information. Further expansion of information technology (IT) is tied to the confidence of Internet users to the security of data transaction on Internet. Secure information exchange is vital for E-commerce, and public key cryptography is the most efficient way to achieve data exchange security between two unfamiliar parties on the Internet.

Public key cryptography was introduced in 1976 by Diffie and Hellman [28]. RSA, the first popular public key cryptosystem, which is based on the difficulty of integer factorization was introduced shortly after. RSA is widely accepted and is used for many cryptographic applications. In 1985, Koblitz [3] and Miller [4] independently introduced elliptic curve cryptography, which is basically based on the group of points on an elliptic curve (EC) over a finite field.

Providing the same security level, elliptic curve cryptosystem (ECC) uses smaller key size compared to RSA. ECC implementations require less power, less memory and less computation power compared to RSA implementations. These features makes ECC very attractive for implementation on constrained devices such as wireless devices, handheld computers and smart cards.

Efficient implementation of elliptic curves cryptosystems can be classified into two basic levels. In the higher level efficiency is tied to the efficiency of the scalar multiplication algorithms(Chapter 3 and 4). On lower level, efficiency goes down to finite field arithmetic, and mostly to finite field multiplication(Chapter 5). This thesis proposes an efficient scalar multiplication algorithm as well as a new architecture for efficient elliptic curve arithmetic implementation.

Although implementing security algorithms in software is easier, it is relatively slow, and has the effect of slowing down and consuming the valuable time of the main processor of the host system. Hardware solutions are attractive specially when there is a large volume of secure transactions. Considering the current growth trends it is expected that the demand for fast security processors will be high in the future.

1.2 Thesis Outline

Chapter 2, gives an elementary introduction to Finite Fields and Elliptic Curves. It covers some of the mathematical theory behind the construction of finite fields and elliptic curve group and the basic equations that govern the point addition and point doubling on an elliptic curve. Finally, it describes the idea of creating a security system based on elliptic curve and gives estimation of the strength of elliptic curve cryptosystem.

Chapter 3, provides a comprehensive survey on currently used elliptic curve scalar multiplication algorithms. Different coordinate systems are explained and EC point addition and doubling formula in each coordinate is expressed and compared to each other. Scalar multiplication algorithms are categorized . Algorithms based on scalar recording explained and evaluated. Special scalar multiplication techniques such as point halving method, Montgomery algorithm and ,ECC based on Koblitz curve discussed at the end of the chapter.

Chapter 4, introduces a new fast algorithm for scalar multiplication. The new technique is explained and simulation results are compared to conventional double and add methods [10].

Chapter 5, describes the proposed architecture for a high speed elliptic curve processor. A thorough survey on the elliptic curve processors hardware implementations is carried out, and the proposed processor is compared to them. The RTL simulation result is provided and is compared to few similar design. The results of the survey in chapter 2 is used here to implement an efficient scalar multiplication algorithm.

Chapter 2

Preliminaries on Elliptic Curve Cryptography

2.1 Basic Concepts

Groups

Definition 1. A group consists of a set G together with an operation $\star$ defined on G which satisfies the following axioms.

1. Closure: for all $a, b \in G$ we have $a \star b \in G$
2. Associativity: for all $a, b, c \in G$ we have $(a \star b) \star c = a \star (b \star c)$
3. Identity: for all $a \in G$ there exists $e \in G$ so that $a \star e = e \star a = a$. The unique element e is called the *neutral element* in G.
4. Inverse: for all $a \in G$ there exists $i \in G$ so that $a \star i = i \star a = e$. i is unique and is called *inverse* of a

We use the notation $\langle G, \star \rangle$ to represent group G with group operation $\star$. $\langle G, \times \rangle$ and $\langle G, + \rangle$ are called multiplicative and additive group respectively. In an additive group, the neutral element is represented by the symbol 0 and the inverse of a is denoted as $-a$. In a multiplicative group, the neutral element is represented by the symbol 1 and the inverse of a is denoted as a^{-1}.

$\langle G, \star \rangle$ is called an Abelian or commutative group if for any a and $b \in G$ we have $a \star b = b \star a$.

if set G is finite, the group $\langle G, \star \rangle$ is called a finite group. The number of elements in G is called the order of the group and is denoted by $\mid G \mid$

Rings

Definition 2. A ring is a set R and two operations $+$ and $\times$ (called addition and multiplication, respectively) defined over R which satisfies the following axioms:

1. $\langle R, + \rangle$ is a commutative group.
2. Associativity of $\times$: For all $a, b, c \in R$ we have $(a \times b) \times c = a \times (b \times c)$
3. Distributivity of $\times$ over $+$: For all $a, b, c \in R$, $a \times (b + c) = a \times b + a \times c$ and $(a + b) \times c = a \times c + b \times c$

A ring in which the multiplication $\times$ is commutative is called a commutative ring.

Fields

Definition 3. A field is a ring in which multiplication is commutative and every element except 0 has a multiplicative inverse.

So, we can define the field F with respect to the operations $\times$ and $+$ if:

1. $\langle R, + \rangle$ is a commutative group.
2. $\langle R - \{0\}, \times \rangle$ is a commutative group
3. $\times$ is distributive over $+$

If set F has finite number of elements then F is a finite field or a *Galois Field* . For example the set $\mathbb{Z}_p = \{0, 1, \ldots, p-2, p-1\}$ where p is a prime, with modular addition and modular multiplication is a finite field.

Definition 4. One way function is a function that provides for a computationally inexpensive mapping from set X to set Y for all $x \in X$ but becomes computationally infeasible when mapping an element from set Y to set X for most $y \in Y$.

Discrete logarithm (DL) problem: A particular one-way function with $x, y \in G$ such that the discrete logarithm of x to base y, denoted by log y (x), has a unique integer solution z where $x = y^z$.

2.2 Elliptic Curves

Elliptic curves have been studied by mathematicians for more than a century. They have been playing an important role in number theory and cryptography. Elliptic Curves have been used in integer factorization and have played an important role in solving the famous problem known as Fermat's last theorem. Elliptic curve cryptography was proposed independently by Victor Miller [4] and Neil Koblitz [3] in the 1985. Elliptic curve cryptosystems are standardized and are commercially available.

2.2.1 Definition of Elliptic Curves

Definition 5. Elliptic curve E over field $\mathcal{K}$ is a set of points (x, y) with $x, y \in \mathcal{K}$ which satisfy the equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

where $a_1, a_2, a_4, a_6 \in \mathcal{K}$, together with a single element denoted $\mathcal{O}$ are called point of infinity [10].

The elliptic curve over $\mathcal{K}$ is denoted by $E(\mathcal{K})$. The number of points on E(the cardinality) is denoted $\#E(\mathcal{K})$ or just $\#E$.

An elliptic curve can be defined over various fields. For example, field of complex numbers $\mathbb{C}$, field of real numbers $\mathbb{R}$, field of rational numbers $\mathbb{Q}$, finite field over prime $\mathbb{F}_p$ or an extension field $\mathbb{F}_{p^n}$. If $\mathcal{K}$ is a field, and $a_1, a_2, a_4, a_6 \in \mathcal{K}$, we say *$E$ is defined over $\mathcal{K}$*. In this case the elliptic curve will be the set of points (x, y) where $x, y \in \mathcal{K}$ and (x, y) satisfy equation 2.1. In cryptography, elliptic curves over finite field $\mathbb{F}_p$ or $\mathbb{F}_{p^n}$ are used. Specifically $\mathbb{F}_{2^n}$ is used more often since it leads to a more efficient design.

For fields of various characteristics, the equation 2.1 can be changed into simpler forms by a linear change of variables. For fields of characteristics two equation 2.1 is simplified to

$$E : y^2 + xy = x^3 + a_2x^2 + a_6 \tag{2.2}$$

where $a_2, a_6 \in \mathbb{F}_{2^n}$.

We consider the equations for field of characteristic 2 which is used in this work. Equation for a field other than characteristic 2 was omitted since they are not central to the discussions.

The Graph of Elliptic Curves

Figure 2.1 shows graphs of two typical elliptic curves defined over the field of real numbers. The graph of elliptic curve over a finite field is a finite of set of points as is depicted in figure 2.2. Each point in graph 2.2 is called *a point on the elliptic curve* and is denoted by a single letter such as P. The number of points on a elliptic curve over a finite field is an important cryptographic aspect of the curve and will be discussed later.

2.2.2 Point Addition Formula

Suppose $P1$ and $P2$ are two points on elliptic curve $E(\mathcal{K})$. Choose P1 and P2 and construct a line through these 2 points. In the general case, this line will always have a point of intersection with the curve. Now take this third point and construct a vertical line through it. The other point of intersection of this vertical line with the curve is defined as the sum of $P1$ and $P2$, i.e. $P3 = P1 + P2$. If $P1$ and $P2$ are equal, then the line constructed

Figure 2.1: Typical Graph of Elliptic Curve defined over the Field of Real Numbers

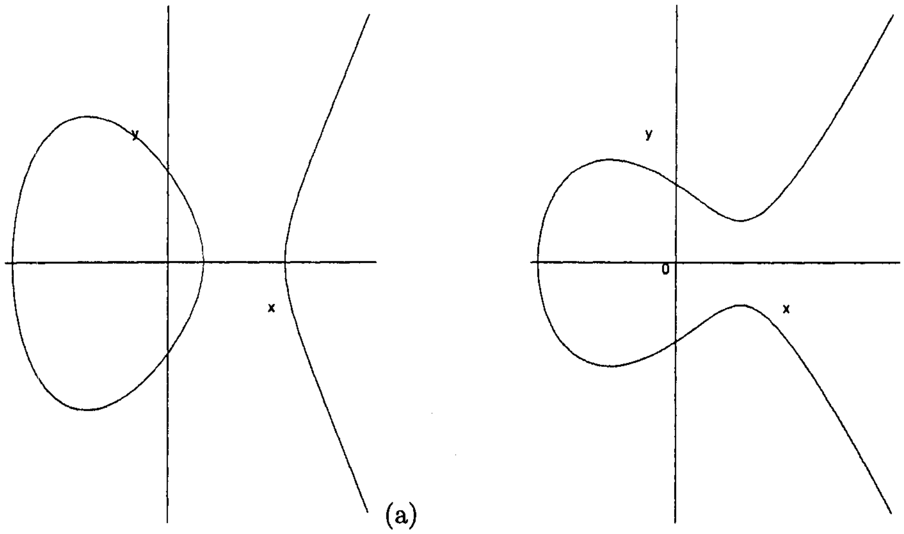


Figure 2.2: Graph of Elliptic Curve defined over $GF(2^{23})$

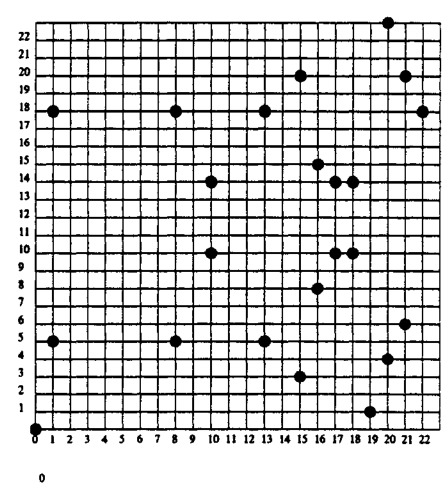
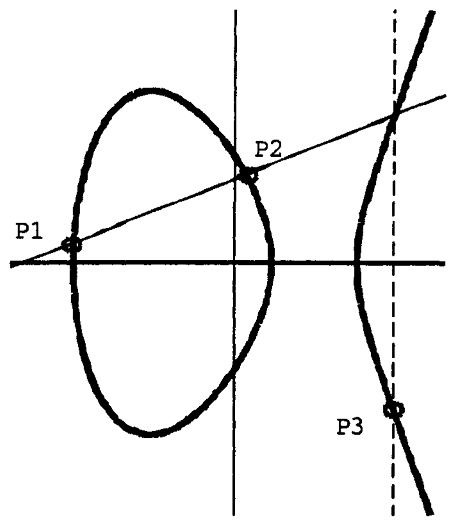


Figure 2.3: Elliptic Curve Point Addition Operation $P3 = P1 + P2$.



in the first step is the tangent to the curve, which again, has exactly one other point of intersection with the curve. This operation is illustrated graphically in figure 2.3.

For each of the two elliptic curves equation 2.2 and 2.1 Analytical formulas representing $P3$ can easily be derived from the explained geometric procedures.

Addition formula for equation 2.1: The inverse of $P1 = (x_1, y_1) \in E$ is $-P = (x_1, -y_1)$. If $P2 \neq -P1$, then $P3 = P1 + P2 = (x3, y3)$ where

$$\text{If } P1 \neq P2 \quad \begin{cases} \lambda = \frac{y_2+y_1}{x_2+x_1} \\ x_3 = \lambda^2 - \lambda + x_1 - x_2 \\ y_3 = (x_1 - x_3)\lambda - y_1 \end{cases} \tag{2.3}$$

$$\text{if } P1 = P2 \qquad \begin{cases} \lambda = \frac{y_1}{x_1} + x_1 \\ x_3 = \lambda^2 + \lambda + a_2 \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1 \end{cases} \tag{2.4}$$

Addition formula for equation 2.2: The inverse of $P1 = (x_1, y_1) \in E$ is $-P = (x_1, x_1 + y_1)$. If $P2 \neq -P1$, then $P3 = P1 + P2 = (x3, y3)$ where

$$\text{if } P1 \neq P2 \qquad \begin{cases} \lambda = \frac{y_1+y_2}{x_1+x_2} \\ x_3 = \lambda^2 + \lambda + x_1 + x_2 + a_2 \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1 \end{cases} \tag{2.5}$$

$$\text{if } P1 = P2 \qquad \begin{cases} \lambda = \frac{y_1}{x_1} + x_1 \\ x_3 = \lambda^2 + \lambda + a_2 \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1 \end{cases} \tag{2.6}$$

In summary we define the following rules for elliptic curve point addition:

- If $P = \mathcal{O}$ we define $-P = \mathcal{O}$
- Equation 2.1: If $P = (x, y) \Rightarrow -P = (x, -y)$
  Equation 2.2: If $P = (x, y) \Rightarrow -P = (x, x + y)$
- If $P1 \neq P2 \Rightarrow P3 = P1 + P2$ equation 2.1 and 2.2
- If $P1 = -P2 \Rightarrow P1 + P2 = \mathcal{O}$

Elliptic Curve Group Law

The Elliptic Curve addition operation satisfies the following properties:

1. Closer: $(P + Q) \in E$
2. Commutativity: $P + Q = Q + P$
3. Existence of identity: $P + \mathcal{O} = \mathcal{O} + P$
4. Existence of inverse: $\forall P \in E \exists Q \in E$ so that $P + Q = Q + P = \mathcal{O}$

5. Associativity: $(P+Q)+R = (P+Q)+R$

All properties except 2 are easy to prove. For a proof on property 2 see [20].

Therefore *Points on E form an finite additive Abelian group with $\mathcal{O}$ as the identity element.* If the elliptic curve is defined over a finite field, the elliptic curve additive group forms a finite Abelian group.

2.2.3 Elliptic Curve Discrete Logarithm Problem

For some group $\langle G, \times \rangle$, suppose $\alpha, \beta \in G$. Given α and β find for an integer x such that $\alpha^x = \beta$ is called the discrete logarithm problem (DLP). The DLP in $\mathbb{Z}_p$ is considered difficult if p has at least 150 digits and $p-1$ has at least one large prime factor (as close to p as possible). These criteria for p are safeguards against the known attacks on DLP. Although the discrete logarithm problem exists in any group, when used for cryptographic purposes the group is usually $\mathbb{Z}_p$. In fact discrete logarithm problem can be used to build cryptosystems with any finite Abelian group. Multiplicative groups in a finite field were originally proposed.

Definition 6. elliptic curve discrete logarithm problem (ECDLP) is defined as follows: we define, $kP = \underbrace{P+P+P+\cdots+P}_{k \quad times}$

- ECDLP: Suppose $P, Q \in E(\mathbb{F}_q)$ and $Q = kP$ for some k . Given P and Q find k

No efficient algorithm is known to date to solve the ECDLP. Numerous cryptosystems based their security on the difficulty of solving the DLP. For example El-Gamal Cryptosystem in $\mathbb{Z}_p$ and Diffie-Hellman key exchange [20].

There are also a number of cryptosystems whose security is based on the difficulty of factoring large integers. One well-known example is the public-key system called the *RSA* cryptosystem, which is by far the most popular public key algorithm.

2.3 Elliptic Curve Cryptosystem

Cryptosystems using elliptic curves are based on ECDLP. The basic operation in ECC is $kP = \underbrace{P + P + P + \cdots + P}_{k\ \ times}$. The following list shows some encryption system based on ECC

- Diffie-Hellman key exchange
- Messy-Omura Encryption
- El-Gamal Public Key Encryption
- El-Gamal Digital Signature
- Elliptic Curve Digital Signature Algorithm (ECDSA).

Detail explanation of these encryption systems can be found in [20] and [21]

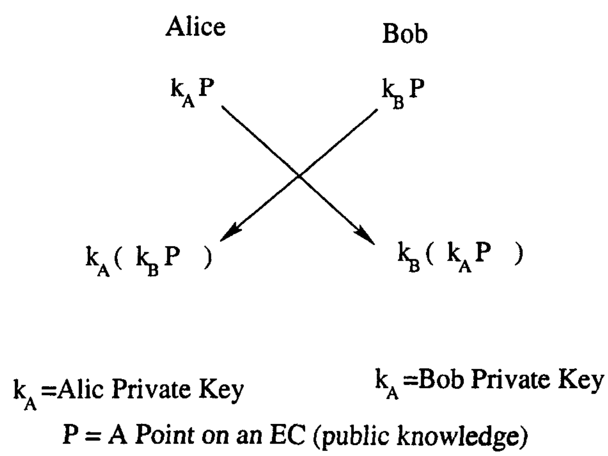
Example of an Elliptic Curve Cryptosystems: Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange protocol was proposed in 1976 [28]. This protocol allows two or more participants to agree on a secret key without ever requiring access to a private channel. Even if Eve (The Eavesdropper) is able to see every message passed between the principles, it is mathematically infeasible for her to deduce the secret key. The protocol is as follows:

Suppose Alice and Bob want to agree on a shared secret key . First of all, there are public parameters $P \in E$. Then they start the following communication.

1. Alice secretly chooses a random number n and sends Bob $k_A P$.
2. Bob secretly chooses a random number m and sends Alice $k_B P$.
3. The secret key is $k_A k_B P = k_B k_A P$. Both Alice and Bob can easily compute, but Eve can't, because of the difficulty of the *discrete logarithm problem.*

Figure 2.4: Diffie-Hellman key exchange



4. Now Alice and Bob have the same key, $k_B(k_A P)$ and can use this key to send encrypted messages to each other

The most time consuming calculation in this system is kP (Scalar Multiplication). Diffie-Hellman key exchange works for DLP as well as ECDLP.

Security of an Elliptic Curve Cryptosystem

In this section we try to provide an overview of the security strength elliptic curve cryptosystems. A typical system is based on Galois fields between 150-160, which are small enough for efficiency and are large enough for security.

There are two basic type of algorithms to solve discrete logarithm problem. General attacks which do not depend on the underlying group and specific attacks which depend on the representation[32].

Elliptic curve discrete logarithm problem is defined as follows: Let $E(\mathbb{F}_q)$ be an elliptic curve over $\mathbb{F}_q$ and let P be a point in $E(\mathbb{F}_q)$. For any point $R \in E(\mathbb{F}_q)$ find the integer $k, 0 \leq k \leq \#P - 1$, ($\#P$ is the order of P) such that $kP = R$.

The most powerful general algorithm known at present is baby-step giant-step technique [20]. Algorithms in this group have running time no better that $O(\sqrt{p})$, where p is the largest prime dividing n. Shank's baby-step giant-step method [20] requires $O(\sqrt{p})$ in both time and space. The storage requirement can be reduced significantly by using the Pollard method [20]. Pollard method requires $\sqrt{p}$ iterations on elliptic curve where each iteration requires 3 elliptic curve additions. Each addition take 10 field multiplications where each field multiplication takes 4 clock cycles to complete (using the proposed processor described in the last chapter). Then we need $40\sqrt{p}$ clock cycles or $0.4\sqrt{q}\mu Sec$ to solve ECDLP. If the order of the curve E contains a prime factor of at least 36 decimal digits, then we need $\approx 0.4 \times 10^{18}\mu Sec$ which is about 12000 years to complete the operation. See [32] for more explanation.

All methods for solving the discrete logarithm problem, except index-calculus method, can be adapted to solve EC discrete logarithm problem (ECDLP). This means that there exists no method for solving m with a sub-exponential running time. m should be prime, in order to be safeguarded against *Weil decent* attacks [63].

Certicom (www.certicom.com), a Canadian company, has announced challenges to break a typical ECC. Table 2.1 shows the challenge and the estimated time to break the ECC.

2.4 Elliptic Curve Cryptography Standardization

The development of standards is a very important point for the use of a cryptosystem. Standards help ensure security and interpret-ability of different implementations of one cryptosystem. There are several major organizations that develop standards. The most important for security in information technology are:

- International Standards Organization (ISO)
- American National Standards Institute (ANSI)
- Institute of Electrical and Electronics Engineers (IEEE)
- Federal Information Processing Standards (FIPS)
- National Institute of Standards and Technology (NIST)

Table 2.1: Elliptic Curve Cryptography Challenge(www.certicom.com)

Curve Curve	Field size (in bits)	Estimated number of machine days	Prize (US$)	Status Status
ECC2-79	79	352	HAC, Maple	SOLVED Dec. 1997
ECC2-89	89	11278	HAC, Maple	SOLVED Feb. 1998
ECC2K-95	97	8637	$ 5,000	SOLVED May 1998
ECC2-97	97	180448	$ 5,000	
ECC2K-108	109	1.3×10^6	$ 10,000	SOLVED Apr. 2000
ECC2-109	109	2.1×10^7	$ 10,000	
ECC2K-130	131	2.7×10^9	$ 20,000	
ECC2-131	131	6.6×10^{10}	$ 20,000	
ECC2-163	163	2.9×10^{15}	$ 30,000	
ECC2K-163	163	4.6×10^{14}	$ 30,000	
ECC2-191	191	1.4×10^{20}	$ 40,000	
ECC2-238	239	3.0×10^{27}	$ 50,000	
ECC2K-238	239	1.3×10^{26}	$ 50,000	
ECC2-353	359	1.4×10^{45}	$ 100,000	
ECC2K-358	359	2.8×10^{44}	$ 100,000	

Table 2.2: Elliptic Curve Standards and Algorithms

Standard	Schemes
ANSI X9.62	ECDSA
ANSI X9.63	ECIES, ECDH, ECMQV
FIPS 186-2	ECDSA
IEEE P1363	ECDSA, ECDH, ECMQV
IEEE P1363A	ECIES
ISO 14888-3	ECDSA
ISO 15946	ECDSA, ECDH, ECMQV

Elliptic Curve Digital Signature Algorithm (ECDSA)
Elliptic Curve Integrated Encryption Scheme (ECIES)
Elliptic Curve Menezes-Qu-Vanstone Protocol (ECMQV)
Elliptic Curve Diffie-Hellman (ECDH)

The most prominent ECC algorithm, the ECDSA was accepted in 1998 as ISO standard (ISO14888-3), 1999 as ANSI standard (ANSI X9.62), and 2000 as IEEE (P1363) and Fips (186-2) standard. Several other standardization efforts are in progress. Table 2.2 shows the Elliptic Curve standards

2.5 Intellectual Property Issues

Contrary to RSA, the basic idea of Elliptic Curve Cryptosystems has not been patented, and in the beginning this seemed to be an important advantage. However, a number of patents have been applied for, on techniques that mostly aim at improving efficiency. In principle, it should still be possible to construct a secure, albeit not extremely efficient elliptic curve cryptosystems without licensing patents. The patents are mostly held by Certicom, a Canadian company which is marketing elliptic curve cryptosystem.

A number of these techniques are being considered for inclusion in standards and this will potentially make it hard to implement interpretable elliptic curve systems without licensing patents. On the other hand, some standardization organizations require the holders of patents on standardized techniques to guarantee 'reasonable' licensing conditions. In summary, elliptic curves have lost many of their advantages as far as patents are concerned.

Chapter 3

Introduction to ECC Computations

3.1 Introduction

In order to implement and elliptic curve cryptosystem one has to decide on the following options:

1. Defining Equation for Elliptic curve
   - Weierstrass form [6]
   - Koblitz Curves [2]
2. Representation of points [10]
   - Affine Coordinates
   - Projective
   - Mixed Coordinates
3. Scalar Multiplication technique kP ie. $kP = \underbrace{P + P + P + \cdots + P}_{k \quad times}$
   - Comb method [16]

- Window method [10]
- Montgomery method [61]
- Scalar Recording [7]

4. Field Representation

- Polynomial Basis
- Normal Basis
- Dual Basis

5. Finite Field operation Algorithm

- Multiplication
- Squaring
- Inversion

In this chapter items 1, 2 and 3 are explained. Algorithms for finite field operation are explained in the last chapter. Item 4 is not discusses here.

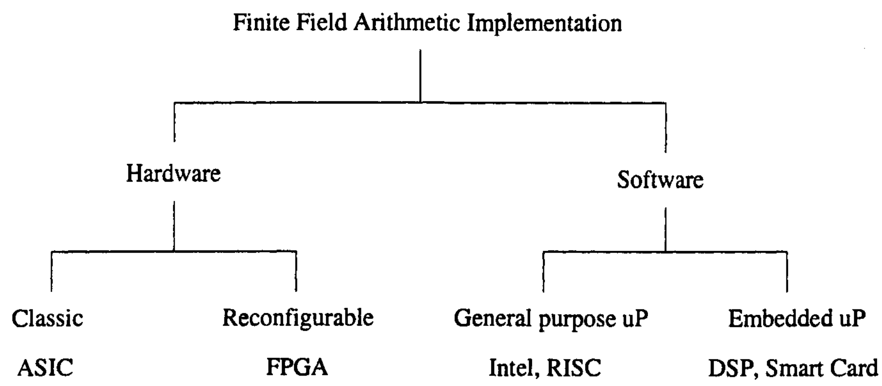
Speed of a ECC system is determined by the above factors as well as implementation platform (Fig. 3.1). Using a dedicated hardware to speedup the underlying finite field arithmetic will increase the speed of elliptic curve operations as it is explained in the last chapter.

3.2 Elliptic Curve Definition

Definition 7. Let K be a field of characteristics $\neq 2, 3$, lets $x^3 + ax + b$ (where $a, b \in K$) be a cubic polynomial with no multiple roots. An elliptic curve over K is the set of points (x, y) with $x, y \in K$ which satisfy the equation

$$y^2 = x^3 + ax + b \tag{3.1}$$

Figure 3.1: Platform option for ECC implementation



together with a single element denoted $\mathcal{O}$ and is called *point at infinity.* If K is of characteristics 2, then an elliptic curve over K is the set of points satisfying the equation

$$y^2 + y = x^3 + ax + b \tag{3.2}$$

[1].

3.2.1 Different Forms of Elliptic Curve Equation

Weierstrass Form [6]

An affine Weierstrass equation over field K is an equation of the form

$$E(K) : Y^2 = a_1XY + a_3y = Xx^3 + a_2X^3 + a_4X + a_6 \tag{3.3}$$

with $a_1, a_2, a_4, a_6 \in K$.

Koblitz Form [2]

Two extremely convenient families of curves are the anamolaus binary curves (or ABC's or Koblitz curves). These are the curves E_0 and E_l defined over $\mathbb{F}_{2^m}$ by $E_a : x^2 + xy = x^3 + ax^2 + 1$. We denote by $E_a(\mathbb{F}_{2^m})$ the group of $\mathbb{F}_{2^m}$-rational points on E_a This is the group on which the public-key protocols are performed. As we will see, this group of curves speeds up the scalar multiplication [7].

3.3 Elliptic Curve Point Representation

An elliptic curve can be represented using several coordinate systems. For each such system, the speed of point additions ($ECADD$)and doubling ($ECDBL$) are different. Therefore a good choice of coordinate system is an important factor for elliptic curve exponentiations. We give here the addition and doubling formulas for affine, projective, Jacobian, Chudnovsky and Lopez-Dahab coordinates. These coordinates are defined in section 3.4.1.

3.3.1 The Addition Formulas in Affine Coordinate

Let

$$E_a : y^2 + xy = x^3 + ax^2 + b \qquad a, b \in \mathbb{F}_{2^m}$$

be an elliptic curve E over $\mathbb{F}_{2^m}$. The addition formula for affine coordinates are the followings. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on E_a . Then the coordinates of $P_3 = P_1 + P_2 = (x_3, y_3)$ can be computed as shown in table 3.1.

Table 3.1: Addition Formula in Affine Coordinate

$P1 \neq P2$	$P1 = P2$
$\lambda = \frac{y_1 - y_2}{x_1 - x_2}$	$\lambda = \frac{y_1}{x_1} + x_1$
$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$	$x_3 =$ same
$y_3 = (x_1 + x_3)\lambda + x_3 + y_1$	$x_3 =$ same
Cost: $I + 2M + S$	Cost: $I + 2M + S$

For simplicity, we neglect addition and subtraction in $\mathbb{F}_{2^m}$ because they are much faster than multiplication and inversion in $\mathbb{F}_{2^m}$. Let us denote the computation time of an addition (resp. a doubling) by $t(P + P)$ or $t(ECADD)$ (resp. $t(2P)$ or $t(ECDBL)$) and represent multiplication (resp. inverse, resp. squaring) in $\mathbb{F}_{2^m}$ by M (resp. I, resp. S). Then we see that $t(P + Q) = I + 2M + S$ and $t(2A) = I + 2M + 2S$ [8] .

3.3.2 Projective Space and the Point at Infinity

Definition 8. n-Dimensional projective space P_K^n over field K is the set of equivalence classes of n-tuple $(x_0, x_1, x_2, \ldots, x_n)$ with $x_0, x_1, x_2, \ldots, x_n \in K$. Two n−tuple $(x_0, x_1, x_2, \ldots, x_n)$ and $(y_0, y_1, y_2, \ldots, y_n)$ are said to be equivalent iff there exists non-zero element $\lambda \in K$ such that

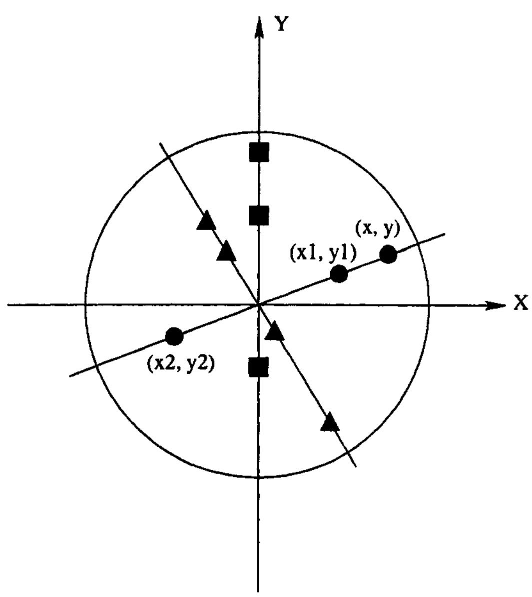
$$(x_0, x_1, x_2, \ldots, x_n) = (\lambda y_1, \lambda y_2, \lambda y_3, \ldots, \lambda y_n)$$

We write

$$(x_0, x_1, x_2, \ldots, x_n) \sim (y_0, y_1, y_2, \ldots, y_n)$$

Example: Projective line $P_{\mathbb{R}}^1$. It is the set of points (x, y) excluding $(0, 0)$ with the points $(\lambda x, \lambda y)$ identified with (x, y). If we select $P = (x, y)$, then all the points $(\lambda x, \lambda y)$ are on the line joining P to the origin. This is visualized in figure 3.2. Points with the same shape are equivalent. For every equivalence class we can choose a point lying on the unit circle as a representative. The projective line $P_{\mathbb{R}}^1$ is then represented by the unit circle with diagonally opposite points identified together.

Figure 3.2: Projective Line



The equivalence class of (x, y, z) is denoted by $(x : y : z)$. If $(x : y : z)$ is a point with $z \neq 0$, then $(x : y : z) = (x/z : y/z : 1)$. These are the finite points in P^3_K.However, If $z = 0$, then dividing by z should be thought of as giving ∞ in either the x or y coordinate, and therefore the points $(x : y : 0)$ are called points at infinity in P^n_K. The point at infinity on an elliptic curve is identified with one of these points at infinity in P^3_K.

The two-dimensional **affine plane** over K is defined by

$$A^2_k = \{(x, y) \mid x, y \in k\}$$

We have an inclusion

$$A^2_k \hookrightarrow P^2_K$$

given by

$$(x, y) \hookrightarrow (x : y : 1)$$

In this way affine plane is defined with the **finite points** in P^3_K.

A polynomial is homogeneous of degree n if it is a sum of terms of the form $ax^iy^jz^k$ with $a \in K$ and $i + j + k = n$. If $f(x, y)$ is a polynomial in x and y, then we can make it homogeneous by inserting appropriate powers of z. For example, if $f(x, y) = y^2 - x^3 - Ax - B$ then we obtain the homogeneous polynomial $F(x, y) = y^2z - x^3 - Axz^2 - Bz^3$. If F is homogeneous of degree n then

$$F(x, y, z) = z^3 f(x/z, y/z)$$

and

$$f(x, y) = F(x, y, 1)$$

The elliptic curve E is given by $y^2 = x^3 + Ax + B$. The homogeneous from is $y^2z = x^3 + Axz^2 + Bz^3$. The point (x, y) on the original curve, corresponds to points $(x : y : 1)$ in the projective version. To see what points on E lie at infinity, set $z = 0$ and obtain $x = 0$. Therefore $x = 0$, and y can be any nonzero number. Rescale by y to find that $(0 : y : 0) = (0 : 1 : 0)$ is the only point at infinity on E. Using projective coordinate speeds up computation on elliptic curve.

3.4 Choosing a Coordinate System

Using different projections, points on an elliptic curve can be represented in many different ways, as it is shown in the following list.

- Affine Plane: (x, y) $\quad E_a : y^2 + xy = x^3 + ax^2 + b \quad a, b \in \mathbb{F}_{2^m}$
- Projective Plane: $(x = X/Z, y = Y/Z)$ $\quad E_p : Y^2Z + XYZ = X^3 + aX^2Z + bZ^3 \quad a, b \in \mathbb{F}_{2^m}$
- Jacobian: $(x = X/Z^2, y = Y/Z^3)$ $\quad E_J : Y^2 = X^3 + aXZ^4 + bZ^6 \quad a, b \in \mathbb{F}_p$
- Chudnovsky: (X, Y, Z, Z^2, Z^3) $\quad P_3 = P_1 + P_2 = P2 = (X_3, Y_3, Z_3, Z_32, Z_33)$.
- Lopez-Dahab: $(x = X/Z, y = Y/Z^2)$ $\quad E_d : Y^2 + XYZ = X^3 + aXZ^2 + bZ^4 \quad a, b \in \mathbb{F}_{2^m}$

3.4.1 Different Coordinate Systems

The Addition Formulas in Projective Coordinates

For projective coordinates, we set $x = X/Z$ and $y = Y/Z$, giving the equation:

$$E_p : Y^2Z = X^3 + aXZ^2 + bZ^3 \qquad a, b \in \mathbb{F}_p$$

$$E_p : Y^2Z + XYZ = X^3 + aX^2Z + bZ^3 \qquad a, b \in \mathbb{F}_{2^m}$$

The addition formulas in projective coordinates for $\mathbb{F}_p$ are the following. Let $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$ and $P_3 = P_1 + P_2 = (X3, Y3, Z3)$, table 3.2 summarized the addition formula [8].

The Addition Formulas in Jacobian Coordinates

For Jacobian coordinates, we set $x = X/Z^2$ and $y = Y/Z^3$, giving the equation:

$$E_J : Y^2 = X^3 + aXZ^4 + bZ^6 \qquad a, b \in \mathbb{F}_p$$

Table 3.2: Addition Formula in Projective Coordinates for $\mathbb{F}_p$

$P1 \neq P2$	$P1 = P2$
$u = Y_2Z_1 - Y_1Z_2$	$u = aZ_1{}^2 + 3X_1{}^2$
$v = X_2Z_1 - X_1Z_2$	$v = Y_1Z_1$
$w = u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2$	$w = X_1Y_1v$
	$t = u^8w$
$X_3 = vw$	$X_3 = 2vt$
$Y_3 = u(v^2X_1Z_2 - w) - v^3Y_1Z_2$	$Y_3 = u(4w - t) - 8Y^2v^2$
$Z_3 = v^3Z_1Z_2$	$Z_3 = 8v^3$
Cost: $12M + 2S$	Cost: $7M + 5S$

The addition formulas in the Jacobian coordinates are presented in table 3.3. Table 3.4 represents the point addition and point doubling formulae adapted from IEEE P1363 standard for comparison[21]A10-5, A10-7.

The Addition Formulas in Chudnovsky Jacobian Coordinates

We see that Jacobian coordinates offer a faster doubling and a slower addition than projective coordinates. In order to make an addition faster, we should represent internally a Jacobian point as the quintuple (X, Y, Z, Z^2, Z^3). This is called the Chudnovsky Jacobian coordinate and denoted by J_c. The addition formulas in the Chudnovsky Jacobian coordinates are the following. Let $P1 = (X_1, Y_1, Z_1, Z_12, Z_13)$, $P2 = (X_2, Y_2, Z_2, Z_22, Z_23)$ and $P_3 = P_1 + P_2 = P2 = (X_3, Y_3, Z_3, Z_32, Z_33)$. Table 3.5 shows the addition procedure in $mathbbF_p$.

The Addition Formulas in Lopez-Dahab Coordinates

We set $x = X/Z$ and $y = Y/Z^2$, giving the equation:

$$E_d : Y^2 + XYZ = X^3 + aXZ^2 + bZ^4 \qquad a, b \in \mathbb{F}_{2^m}$$

Table 3.3: Addition Formula in Jacobian coordinates for $\mathbb{F}_p$

$P1 \neq P2$	$P1 = P2$
$U_1 = X_1 {Z_2}^2$	$S = 4X_1Y^2$
$U_2 = X_2 {Z_1}^2$	$M = 3{X_1}^2 + a{Z_1}^4$
$S_1 = X_1 {Z_2}^3$	$T = -2S + M^2$
$S_1 = Y_2 {Z_1}^3$	
$H1 = U_2 - U1$	
$R = S_2 - S - 1$	
$X_3 = -H^3 - 2U_1H^2 + R^2$	$X_3 = T$
$Y_3 = -S_1H^3 + R(U_1H^2 - X_3)$	$Y_3 = -8Y_14 + M(S - T)$
$Z_3 = Z_1Z_2H$	$Z_3 = 2Y_1Z_1$
Cost: $12M + 4S$	Cost: $4M + 6S$

like other projective coordinates this coordinate we don't need inversion for $ECADD$ and $ECDBL$ (Table 3.6)[9].

The key observation is that, point addition in projective coordinates can be done using field multiplication only, with *no inversion* required. Thus the inversion are deferred, and only one need to be performed at the end of a point calculation, if it is required that the final result be given in affine coordinates. The cost of eliminating inversion is an increased number of multiplication. So the appropriateness of using coordinated is strongly determined by the ratio I/M. for an $I/M \geq 10$ projective coordinates is recommended[9] [10].

Mixed Coordinate

It is evidently possible to mix different coordinates, i.e. to add two points where one is given in some coordinate system, and the other point is in some other coordinate system. We can also choose the coordinate system of the result. Proper use of mixed coordinates can lead to a faster point calculation. For a table of mix coordinate system refer to [8].

Table 3.4: Addition Formula in IEEE Standard for $\mathbb{F}_{2^m}$

$P1 \neq P2$	$P1 = P2$
$U_0 = X_0 Z_1^2$	$(b = c^4)$
$S_0 = Y_0 Z_1^3$	$Z_2 = X_1 Z_1^2$
$U_1 = X_1 Z_0^2$	$X_2 = (X_1 + cZ_1^2)^4$
$S_1 = Y_1 Z_0^3$	$U = Z_2 + X_1^2 + Y_1 Z_1$
$W = U_0 + U_1$	$Y_2 = X_1^4 Z_2 + UX^2$
$R = S_0 + S_1$	
$T = R + Z_2$	
$L = Z_0 W$	
$Z_2 = LZ_1$	
$X_2 = aZ_2^2 + TR + W^3$	
$V = RX_1 + LY_1$	
$Y_2 = TX_2 + VL^2$	
Cost: $15M + 7A + 5S$	Cost: $5M + 4A + 5S$
Cost ($Z_1 = 1$): $11M + 7A + 4S$	

3.4.2 Coordinates Summary

Table 3.7 [1]summarizes the cost of elliptic curve point calculation in different coordinates. Selection of the coordinate system depends on the implementation platform. As a rule of thumb, projective coordinates are preferred, unless there exists an efficient division implementation.

[1]In some cases number of additions is calculated to be used in the performance calculation of the developed processor(chapter 5)

Table 3.5: Addition Formula in Chudnovsky Jacobian Coordinates for $\mathbb{F}_p$

$P1 \neq P2$	$P1 = P2$
$U_1 = X_1 {Z_2}^2$	$S = 4X_1 Y^2$
$U_2 = X_2 {Z_1}^2$	$M = 3{X_1}^2 + a{Z_1}^4$
$S_1 = Y_1 {Z_2}^3$	$T = -2S + M^2$
$S_2 = Y_2 {Z_1}^3$	
$H1 = U_2 - U1$	
$R = S_2 - S - 1$	
$X_3 = -H^3 - 2U_1 H^2 + R^2$	$X_3 = T$
$Y_3 = -S_1 H^3 + R(U_1 H^2 - X_3)$	$Y_3 = -8{Y_1}^4 + M(S - T)$
$Z_3 = Z_1 Z_2 H$	$Z_3 = 2Y_1 Z_1$
Cost: $11M + 4S$	Cost: $5M + 6S$

3.5 Scalar Multiplication

Scalar multiplication (or point multiplication) is the heart of Elliptic Curve Cryptography(ECC), which computes kP for a given point P and a scalar k. In public-key cryptographic systems, elements of some group are raised to large powers. In case of RSA it is a^k and in case of Elliptic curve it is kP .

The scalar multiplication in ECC is the most dominant computation part of ECC. There are many algorithms for computing the scalar multiplication. The IEEE standard one is the binary non-adjacent form (NAF) which is not the most efficient one. Table 3.8 summarizes scalar multiplication techniques.

Scalar multiplication in elliptic curves is a special case of the general problem of modular exponentiation in Abelian group. Therefore it benefits from all the techniques available for the general problem and the related short addition chain problem for integers. However there are also efficiency improvements available elliptic curve case that have no analogue in modular exponentiation. There are three kinds of these [10]:

Table 3.6: Addition Formula in Lopez-Dahab Projective Coordinates for $\mathbb{F}_{2^m}$

$P1 \neq P2$	$P1 = P2$
$A = Y_2Z_1{}^2 + Y_1$	$A = bZ_1{}^4$
$B = X_2Z_1 + X_1$	
$C = Z_1B$	
$D = B^2(C + aZ_1{}^2)$	
$E = AC$	
$F = X_3 + X_2Z_3$	
$G = X_3 + Y_2Z_3$	
$X_3 = A^2 + D + E$	$X_3 = X_1{}^4 + A$
$Y_3 = EF + Z_3G$	$Y_3 = AZ_3 + X_3(aZ_3 + Y_1{}^2 + A^4)$
$Z_3 = C^2$	$Z_3 = X_1{}^2Z_1{}^2$
Cost: $14M$	Cost: $5M$

1. Choose the curve, and the base field over which it is defined, so as to optimize the efficiency of elliptic scalar multiplication.

2. Use the fact that subtraction of points on an elliptic curve is just as efficient as addition.If we allow subtractions of points as well, we can replace the binary expansion of the coefficient n by a more efficient signed binary expansion.

3. Use complex multiplication. Every elliptic curve over a finite field comes equipped with a set of operations which can be viewed as multiplication by complex algebraic integers (as opposed to ordinary integers).

In general the following methods try to optimize kP. Generally the optimization is based on[11]:

1. Recording of multiplier k

2. Precomputation

Table 3.7: Cost of Point Addition and Doubling in Different Coordinate System

Coordinate	Transform	$P+Q$	$2P$	Field
Affine	(X,Y)	$I+2M+S$	$I+2M+S$	$\mathbb{F}_p$
Standard projective	$(X/Z,Y/Z)$	$12M+2S$	$7M+5S$	$\mathbb{F}_p$
Jacobian projective (IEEE)	$(X/Z^2,Y/Z^3)$	$12M+4S$	$4M+5S$	$\mathbb{F}_p$
Jacobian projective (IEEE) Using mixed coordinate	$(X/Z^2,Y/Z^3)$	$15M+5S+7A$ $11M+4S+7A$	$5M+5S+4A$	$\mathbb{F}_{2^m}$
Chudnovsky projective	(X,Y,Z,Z^2,Z^3)	$11M+4S$	$5M+6S$	$\mathbb{F}_p$
Lopez-Dahab projective	$(X/Z,Y/Z^2)$	$14M+5S$	$5M+9S$	$\mathbb{F}_{2^m}$

3.5.1 Speeding up Scalar Multiplication (kP)

Binary Method

This method which is also known as the double-and-add (square and multiply for RSA) method, is over 2000 years old [12]. The basic idea is to compute g^k or kP using the binary expansion of k. Let

$$k = \sum_{i=0}^{n-1} b_i 2^i \tag{3.4}$$

Then the following algorithm will compute kP using binary method, it takes $n \times ECDBL$ and $\frac{n}{2} \times ECADD$ on average [10].

m-ary Method

The binary method has an obvious generalization: Let

$$k = \sum_{i=0}^{d-1} c_i m^i \tag{3.5}$$

The algorithm in table 3.10 computes kP using this representation.

Table 3.8: Classification of scalar multiplication techniques

Name of Method	Basic Idea	Application	Example
Comb [16]	Precompute tables of $\sum_{i=0}^{n-1} 2^{wi}Q$	Q fix	DH key exchange
addition chains [7]	$sum_{i=0}^{n-1} k_i$	k fix	DSA
Windowing (Fix, Variable) $m-$ary [10]	Precompute tables memory $k = \sum_{i=0}^{d-1} c_i m^i$	Q is not known	Security Server
Scalar recoding [7]	fewer zero in binary representation of k (NAF)		

Table 3.9: kP using Double and Add Method

Algorithm: Scalar Multiplication: *Binary Method* [10]
`Input:` `A point` P, `an integer` $k = \sum_{i=0}^{n-1} b_i 2^i, b_i = 0, 1$
`Output` $Q = kP$
$Q \leftarrow \mathcal{O}$
`For i = n-1 to 0 by -1`
$Q \leftarrow 2Q$
`if` $b_i = 1$ `then` $Q \leftarrow Q + P$
`EndFor`
`Return` Q

This method is particulary attractive if $m = 2^r$. For $r = 3$ it will be similar to octal representation of k, and for $r = 4$ it will be similar to hexadecimal representation of k. If $m = 2^r$ this algorithm takes $(n - r) \times ECDBL$ (since $d = n/r, (d - 1)r = n - r$) and $d \times ECADD$ and $(m - 1) \times ECADD$ for precomputation [7][10].

Modified m-ary Method

In case of $m = 2^r$, It is possible to save some $ECADD$ at precalculation phase, by dropping the trailing zeros at each m_i . ie. we calculate $m_i P$ when m_i is odd.

Using this method number of $ECADD$ is $n/r + (m - 2)/2$. The number of $ECDBL$ remain the same. It is worth mentioning that we need to select the *optimized* r for a specific length of k. There is always a specific r for a k which minimizes the number of elliptic computations [7].

Window Method

The m-ary or 2^r-ary method may be thought of as taking k-bit windows in the binary representation of r, calculating the powers in the windows one by one, squaring them r times to shift them over, and then multiplying by the power in the next window [7]. In

Table 3.10: kP using m-ary Method

Algorithm: Scalar Multiplication: *m-ary Method* [10]

Input: A point P, an integer $k = \sum_{i=0}^{d-1} k_i m^i, k_i \in \{0, 1, \ldots, m-1\}$

Output $Q = kP$

$P1 \leftarrow P$

For i =2 to (m-1) by -1

  $P1_i \leftarrow P_{i-1} + P$ (pre calculate, $P_i = iP$)

$Q \leftarrow \mathcal{O}$

For i = d-1 to 0 by -1

  $Q \leftarrow mQ$ (if $m = 2^r$, this requires r doubling)

  $Q \leftarrow Q + k_i P$ (pre calculations is required to calculate all $c_i P$)

EndFor

Return Q

other words it can be regarded as a specific case of window method, where bits of the multiplier k are processed in blocks of r bits. Window method processes windows up to length r disregarding fixed digit boundaries, and skips runs of zeros between them. These runs are taken care of by point doubling, which need to be computed in any case. We assume $r \geq 1$.

Using sliding windows has an effect equivalent to using fixed windows one bit larger, but without increasing the precomputation cost. The computation cost of sliding window method is estimated as $n \times ECDBL$ and $n/(r+1) \times ECADD$ [10].

Redundant Number System: Binary NAF

Subtraction has virtually the same cost as addition in the elliptic curve group. The group negative of (x, y) is $(x, x+y)$ in characteristics two and $(x, -y)$ in odd characteristics. This naturally leads us to scalar multiplication methods based on addition-subtraction chains, which may reduce the number of point operation. The signed-digit (SD) representation can

Table 3.11: kP using Modified m-ary Method

Algorithm: Scalar Multiplication: *Modified m-ary [10] Method*

```
Input:  A point P, an integer k = Σ_{i=0}^{d-1} k_i m^i, k_i ∈ {0,1,...,m-1}
Output Q = kP
P_1 ← P, P_2 ← 2P
For i =1 to (m-2)/2 by -1
    P_{2i+1} ← P_{2i-1} + P_2 (pre calculate, odd multiplies of P)
Q ← O
For i = d-1 to 0 by -1
    If k_j ≠ 0 then
        Let s_j and h_j be such that k_j = 2^{s_j} h_j, h_j odd
        Q ← (2^{r-s_j})Q
        Q ← Q + P_{h_j}
    Else s_j ← r
    Q ← 2^{s_j} Q
EndFor
Return Q
```

be applied to all methods discussed so far, but this technique cannot be used for modular exponentiation in RSA.

This begins with the non-adjacent form (NAF) of the coefficient k: a signed binary expansion with the property that no two consecutive coefficients are nonzero. For example, $NAF(29) = (1, 0, 0, -1, 0, 1)$ since $29 = 32 - 4 + 1$.

Just as every positive integer has a unique binary expansion, it also has a unique NAF. Moreover, $NAF(k)$ has the fewest nonzero coefficients of any signed binary expansion of k [7]. There are several ways to construct the NAF of k from its binary expansion.

Table 3.12: kP using Window Method

Algorithm: Scalar Multiplication: *Sliding Window Method* [10]

Input: A point P, an integer $k = \sum_{i=0}^{d-1} b_i 2^i, k_i \in \{0,1\}$

Output $Q = kP$

$P_1 \leftarrow P, P_2 \leftarrow 2P$

For i =1 to (2^{r-1}-1)

  $P_{2i+1} \leftarrow P_{2i-1} + P_2$ (pre calculate, odd multiples of P)

$j \leftarrow n - 1Q \leftarrow \mathcal{O}$

For i = d-1 to 0 by -1

  If $k_j \neq 0$ then

    Let t be the least integer such that $j - t + 1 \leq r$ and $k_t = 1$

    $h_j \leftarrow (k_j, k_{j-1}, \ldots, k_t)_2$

    $Q \leftarrow (2^{(j-t+1)})Q + P_{h_j}$

    $j \leftarrow t - 1$

  Else $Q \leftarrow 2Q, j \leftarrow j - 1$

EndFor

Return Q

Consider representations

$$n = \sum_{i=0}^{n-1} c_i 2^i \qquad \text{where} \quad c_i \in \{-1, 0, 1\} \quad \text{for all} \quad i \tag{3.6}$$

Let the weight of a representation be the number of nonzero c_i, and let $w(x)$ be the minimum weight of any such representation of x. A non-adjacent form NAF is a representation with $c_i c_{i+1} = 0$ for all $i \geq 0$.

Theorem: *Every integer x has exactly one NAF. The number of nonzero in the NAF is $w(x)$* The advantage of using the NAF is that, in general it has fewer nonzero than the binary representation, reducing the number of multiplications. The expected number of nonzero in a length n NAF is $n/3$. $NAF(k)$ can be efficiently computed using the following in table 3.13. Table 3.14 shows the algorithm for scalar multiplication using Binary NAF method.

Table 3.13: Converting a number to NAF

Algorithm: Computing the NAF of a positive [10] integer

```
Input:  A positive integer k
Output NAF(k)
i ← 0
While k >= 1
    If k is odd then:  k_i ← 2 − (k  mod  4), k ← k − k_i
    Else k ← 0
    k ← k/2, i = i + 1
EndWhile
Return (k_{i−1}, k_{i−2}, ..., k_1, k_0)
```

The $m-ary$ method may of course also be generalized to allow negative digits. However, the savings quickly go down, since the average number of nonzero in an n-digit generalized NAF is $n(m-1)/(m+1)$, which is not much better than the $n(m-1)/(m)$ in the base-m representation for large m. Using Binary NAF the algorithm in table 3.14 will compute kP.

The cost of the algorithm is n doubles and $n/3$ additions. For a total of $4n/3$ elliptic operation. This is about one-eighth faster than the binary method, which uses the ordinary binary expansion in place of the NAF and therefore requires an average of $n/2$ elliptic additions rather than $n/3$.

Width-w NAF Method [10]

The so called width-w NAF method is the special case of signed modified m-ary method, or NAF representation of modified m-ary method, where $m = 2^w$. A width-w NAF of an

Table 3.14: kP using NAF representation for k

Algorithm: Scalar Multiplication: *NAF Binary Method* [10]
`Input:` A point P, an integer $k = \sum_{i=0}^{n-1} c_i 2^i, c_i = -1, 0, 1$
`Output` $Q = kP$
$Q \leftarrow \mathcal{O}$
`For i = n-1 to 0 by -1`
$Q \leftarrow 2Q$
`if` $b_i = 1$ `then` $Q \leftarrow Q + P$
`if` $b_i = -1$ `then` $Q \leftarrow Q - P$
`EndFor`
`Return` Q

integer k is an expression

$$k = \sum_{i=0}^{d-1} k_i m^i, \quad k_i \in \{-2^{w-1}+1, \ldots, 0, 1, 3, \ldots, 2^w - 1\}$$

In other words each non-zero coefficient k_i is odd, $|k_j| < 2^{w-1}$, and at most one of any w consecutive coefficients is nonzero. Every positive integer has a unique width-w NAF, denoted $NAF_w(k)$. Note that $NAF_2(k) = NAF(k)$. $NAF_w(k)$ can be efficiently computed using NAF algorithm in table 3.13 modified as follows: in the first statement of the `While` loop replace $k_i \leftarrow 2 - (k \mod 4)$ by $k_i \leftarrow 2 - (k \mod 2^w)$, where $k \mod 2^w$ denotes the integer u satisfying $u = k(mod 2^w)$ and $-2^{w-1} \leq u < 2^{w-1}$.

It is known that the length of $NAF_w(k)$ is at most one bit longer than the binary representation of k. Also, the average density of non-zero coefficients among all width-w NAFs of length n is approximately $n/(w+1)$ [11]. It follows that the expected running time of scalar multiplication using Width-w is approximately $ECDBL + (2^{w-2} ECADD)$ for precalculation and $(w+1)ECADD + n.ECDBL)$ for the scalar multiplication itself[9]. Note that the number of $ECDBL$ is not changed. When using projective coordinates, the running time in the case $n = 163$ is minimized when $w = 4$. For the cases $n = 233$ and

$n = 283$, the minimum is attained when $w = 5$; however, the running times are only slightly greater when $w = 4$.

3.5.2 Scalar Multiplication Summary

Table 3.15 summarizes number of point addition and point doubling in each of the discussed scalar multiplication methods. As it is clear form the table, recording methods decrease number of additions, but number of point doubling remains almost the same. Although window methods are faster but they need extra memory to save $2P, 3P, \ldots, (w-1)P$.

Table 3.15: Number of Point operation in different scalar multiplication Method

Method	$\#P+Q$ (Average)	$\#2P$
Binary (double-add)	$n/2$	n
m-ary, $m = 2^r$	$n/r + (2^r - 1)$	$n - r$
modified m-ary, $m = 2^r$	$n/r + (2^{r-1} - 1)$	$n - r$
Binary NAF (double-add,sub)	$n/3$	n
width-w NAF Method	$n/(r+1) + 2^{r-2}$	$\approx n$
τ-adic NAF (Koblitz curves only)	$n/3$	0

3.6 Special Methods for Scalar Multiplication

3.6.1 Anomalous Binary Curves (Koblitz Curves)

Two extremely convenient families of curves are the anomalous binary curves (or ABC's). These are the curves E_0 and E_1 defined over $\mathbb{F}_2$ by

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \qquad a \in \{0, 1\}$$

Using Koblitz curves speeds up the scalar multiplication calculation as indicated in table 3.15. However, there are concerns about the security of ECC using Koblitz curves. A complete discussion on Koblitz curves can be found in [2].

3.6.2 Point Halving

In [13], Knudsen introduces a new method for scalar multiplication on a non-supersingular elliptic curve over $GF(2^m)$. The idea is to replace all point doubling with a faster operation, called point halving. Moreover, Knudsen shows that the halving algorithm is superior to previous algorithms when it is implemented using affine coordinates and normal basis. However, the halving algorithm has a storage limitation if a polynomial basis is used, where the required storage is in the order of magnitude $O(n^2)$ bits. The halving algorithm and the Montgomery method cannot take advantage of Koblitz curves properties.

3.7 Montgomery Scalar Multiplication Algorithm

A different approach for computing kP was introduced by Montgomery [17] in 1987. This approach is based on the binary method and the observation that the x-coordinates of the sum of two points whose difference is known can be computed in terms of x-coordinates of the involved points. This method uses the following variant of binary method.

Table 3.16: Montgomery Scalar Multiplication Algorithm

Algorithm: Montgomery Scalar Multiplication, in Projective Coordinate

```
Input:  A point P = (x,y) ∈ E,   an integer k > 0,   k = Σ_{i=0}^{n-1} b_i 2^i,   b_i ∈ {0,1}
Output:  Q = kP
 P1 ← P,   P2 ← 2P
For i = n-2 to 0
  if b_i = 1 then
    P1 ← P1 + P2,   P2 ← 2P2
  else
    P2 ← P1 + P2,   P1 ← 2P1
EndFor
Q ← P1
Return Q
```

Table 3.17: Montgomery Scalar Multiplication Algorithm in Projective Coordinate

Algorithm: Montgomery Scalar Multiplication, in Projective Coordinate

```
Input: A point $P = (x, y) \in E$, an integer $k = \sum_{i=0}^{n-1} b_i 2^i, b_i = 0, 1$
Output: $Q = kP$
 $X1 \leftarrow x, Z1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$
 If ($k = 0$ or $x = 0$)
  $R \leftarrow \mathcal{O}$
  Stop
For $i = n - 2$ to 0
  if $k_i = 1$ then
    Madd($X1, Z1, X2, Z2$), Mdouble($X2, Z2$)
  else
    Madd($X2, Z2, X1, Z1$), Mdouble($X1, Z1$)
EndFor
$Q$ =Mxy($X1, Z1, X2, Z2$)
Return $Q$
```

This method maintains the invariant relationship $P_2 - P_1 = P$, and performs an addition and a doubling in each iteration. In [61] this algorithm is converted to projective space and after simplification the following algorithm is derived.

3.7.1 Calculation

Doubling algorithm

Input: the finite field $GF(2^m)$; the field elements a and $c = b^{2^{m-1}} (c^2 = b)$ defining a curve E over $GF(2^m)$, the x-coordinate X/Z for a point P. Output: the x-coordinate X/Z for the point $2P$.

$$x(2P) = X^4 + b \times Z^4 \tag{3.7}$$

Table 3.18: Steps in Point Doubling, Mdouble()

1	$T_1 = c$
2	$X = X^2$
3	$Z = Z^2$
4	$T_1 = Z \times T_1$
5	$Z = Z \times X$
6	$T_1 = T_1^2$
7	$X = X^2$
8	$X = X + T_1$

$$z(2P) = X^2 \times Z^2 \tag{3.8}$$

This algorithm requires one general field multiplication, one field multiplication by the constant c, four field squaring and one temporary variable (Table 3.18).

Addition algorithm

Input: the finite field $GF(2^m)$; the field elements a and b defining a curve E over $GF(2^m)$; the x-coordinate of the point P; the x-coordinates $X1/Z1$ and $X2/Z2$ for the points $P1$ and $P2$ on E. Output: The x-coordinate $X1/Z1$ for the point $P1 + P2$.

$$Z_3 = (x_1 \times Z_2 + X_2 \times Z_1)^2 \tag{3.9}$$

$$X_3 = x \times Z_3 + (X_1 \times Z_2) \times (X_2 \times Z_1) \tag{3.10}$$

This algorithm requires three general field multiplications, one field multiplication by x, one field squaring and two temporary variables(Table 3.19).

Affine coordinates algorithm Mxy()

Input: the finite field $GF(2^m)$; the affine coordinates of the point $P = (x, y)$; the x-coordinates $X1/Z1$ and $X2/Z2$ for the points $P1$ and $P2$. Output: The affine coordinates

Table 3.19: Steps in Points Addition, Madd()

1	$T_1 = x$
2	$X_1 = X_1 \times Z_2$
3	$Z_1 = Z_1 \times X_2$
4	$T_2 = X_1 \times Z_1$
5	$Z_1 = Z_1 + X_1$
6	$Z_1 = Z_1^2$
7	$X_1 = Z_1 \times T_1$

$(xk, yk) = (X2, Z2)$ for the point $P1$.

$$x_k = \frac{X_1}{Z_1} \tag{3.11}$$

$$y_k = (x + x_k)[(y + x^2) + (\frac{X_2}{Z_2} + x)(\frac{X_1}{Z_1} + x)] \times \frac{1}{x} + y \tag{3.12}$$

This algorithm requires one field inversion, ten general field multiplications, one field squaring and four temporary variables(Table 3.20).

3.7.2 Performance

The performance of Montgomery scalar multiplication algorithm is shown in Table 3.21. Note that in Montgomery algorithm one point addition and one point multiplication is needed for each bit in the scalar, while, whereas using NAF, on an average $n/3$ number of point addition are needed for scalar multiplication. Even if the number of operation is divided by 3 the number of operation in Montgomery algorithm is less that the other methods.

3.7.3 Side channel Attack

Side channel attack (SCA) on cryptosystems uses leakage of a certain side-channel information such as timing, electromagnetic radiation and power consumption to obtain information about the private key.In elliptic curve cryptosystems scalar multiplication algorithms are target for SCA. In scalar multiplication kP is calculated were k is a secret key and P is usually not a secret and even can be chosen by the attacker. If the sequence of executed instructions in the algorithm is directly related to the bits of the private key a successful power-analysis attack can be carried out on the cryptosystem. As in can be seen in table 3.9 it is possible to distinguish a point addition by measuring the power of the device which is executing the algorithm. This makes the insecure against SCA. The algorithm presented in 3.16 is secure against power attack since the operation performed in each step of the scalar multiplication algorithm is not dependent to the bits of k.

The execution time of the algorithm in table 3.9 depends on the number of bits in the binary representation of k. This makes the algorithm vulnerable to time analysis attack.

Table 3.20: Steps in Converting the Coordinates Mxy() (Table 3.17)

1	if $Z_1 = 0$ then output $(0,0)$ and stop
2	if $Z_2 = 0$ then output $(x, x+y)$ and stop
3	$T_1 = x$
4	$T_2 = y$
5	$T_3 = Z_1 \times Z_2$
6	$Z_1 = Z_1 \times T_1$
7	$Z_1 = Z_1 + X_1$
8	$Z_2 = Z_2 \times T_1$
9	$X_1 = Z_2 \times X_1$
10	$Z_2 = Z_2 + X_2$
11	$Z_2 = Z_2 \times Z_1$
12	$T_4 = T_1^2$
13	$T_4 = T_4 + T_2$
14	$T_4 = T_4 \times T_3$
15	$T_4 = T_4 + Z_2$
16	$T_3 = T_3 \times T_1$
17	$T_3 = inverse(T_3)$
18	$T_4 = T_3 \times T_4$
19	$X_2 = X_1 \times T_3$
20	$Z_2 = X_2 + T_1$
21	$Z_2 = Z_2 \times T_4$
22	$Z_2 = Z_2 + T_2$

Table 3.21: Cost of scalar multiplication for projective version of Montgomery algorithm

Representation	Point Addition	Point Doubling
Montgomery, Projective version	4M+1S+2A	2M + 4S + 1A

Chapter 4

Fast Parallel Elliptic Curve Scalar Multiplication

4.1 Introduction

This chapter presents a fast parallel elliptic curve scalar multiplication algorithm based on a dual-processor hardware system. The method has an average computation time of $\frac{n}{3}$ECADD on an n-bit scalar. The improvement is nECDBL compared to conventional methods. When a proper coordinate system and binary representation for the scalar k is used, the average execution time will be as low as nECDBL, which proves this method to be about two times faster than conventional single processor multipliers using the same coordinate system.

4.2 Previous Work

Scalar multiplication is the basic operation for Elliptic Curve public key cryptography. The operation is defined as

$$Q = kP = P + P + \ldots + P \tag{4.1}$$

where P and Q are points on elliptic curve E defined over $GF(2^n)$ and k is a scalar in the range of $1 < k < Ord(E)$.

4.2.1 Conventional Scalar Multiplication Methods [10]

Double-and-add is probably the simplest (and oldest) method of scalar multiplication. The basic idea is to compute kP using the binary expansion of k. Let

$$k = \sum_{i=0}^{n-1} b_i 2^i, \tag{4.2}$$

then algorithm 4.1 computes kP using Double-and-add method. The bit examination can be done from the most significant bit (MSB first method) or the least significant bit (LSB first method).

Table 4.1: Scalar Multiplication using standard binary method (LSB first)

Algorithm: Point Multiplication, Binary Method

Input: A point P, an integer $k = \sum_{i=0}^{n-1} b_i 2^i, b_i \in 0, 1$

Output: $Q = kP$

$Q \leftarrow P$

$R \leftarrow \mathcal{O}$ For i = 0 to n-1 by 1

  If $b_i = 1$ Then

    $R \leftarrow R + Q$

  $Q \leftarrow 2Q$

EndFor

Return R

The execution time for the algorithm is proportional to n Elliptic Curve point doubling operation (ECDBL), and on average $\frac{n}{2}$ Elliptic Curve point addition operation (ECADD). Therefore the total average execution time will be nECDBL + $\frac{n}{2}$ECADD. If redundant representation (ie., binary NAF) is used to represent the scalar k, the average number of one or minus one in the representation of k will be reduced to $\frac{1}{3}$. In this case the average execution time will be proportional to nECDBL + $\frac{n}{3}$ECADD [10] [7]. Table 4.2 summarizes the execution time of different conventional scalar multiplication methods.

Table 4.2: Execution time of kP using different conventional methods

Method	Average Execution Time
Binary [10]	$(n-1)$ECDBL + $\frac{n-1}{2}$ECADD
Binary NAF [10]	$(n-1)$ECDBL + $\frac{n-1}{3}$ECADD
Window [10]	nECDBL + $\frac{n}{w+1}$ECADD

It can be seen from the algorithm that in least significant bit-first (LSB first) method ECDBL and ECADD operations are independent, and they can be performed in parallel.

4.2.2 Speeding up Scalar Multiplication

Many methods have been proposed in the literature to speed up scalar multiplication. These methods are classified in table 3.8. Constraints in scalar multiplications are speed, memory usage and security against side channel attack (SCA). Methods with precomputations, like Window method and Comb method are faster but they need extra memory to store pre-calculated values. Addition Chain methods and Comb methods are very effective when k and P are known in advance, respectively. In comparison Window methods are efficient for most cases.

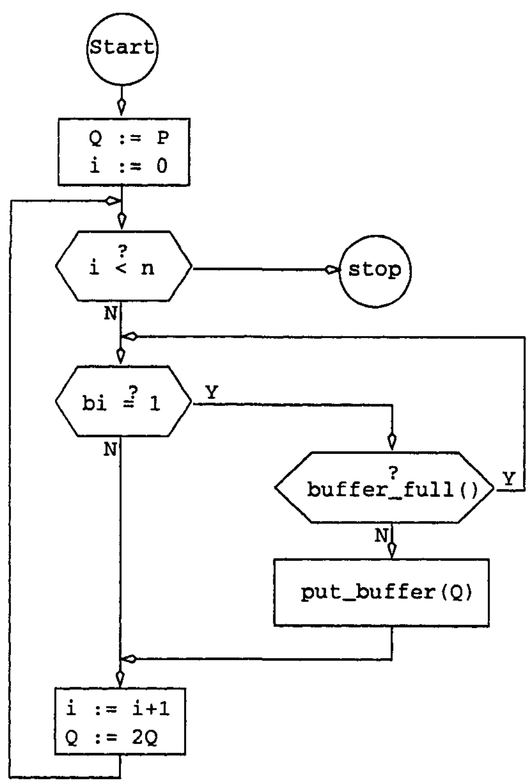
4.2.3 Parallel Architectures

Parallel architectures for scalar multiplication can be done in the scalar multiplication algorithm level or in the calculation of ECDBL or ECADD itself. In [19] Moller proposes a parallel algorithm for scalar multiplication which is fast and secure against side channel attack. This paper proposes a method which uses two processors and a circular buffer, which acts as a communication channel between the two processors to reduce the average time of the scalar multiplication to nECDBL. This way the total time for ECADD is saved and the system can be as fast as a system using τadic NAF for Koblitz curves.

4.3 Improved Parallel Scalar Multiplication

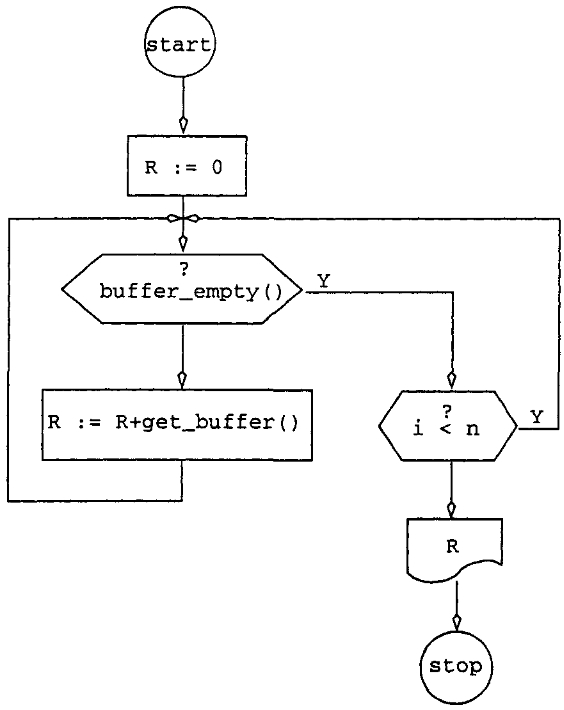
The proposed method for calculating kP uses two processors, one for execution of ECDBL and one for ECADD. The two processors may operate asynchronously. The ECDBL processor calculates 2^iP and stores them to a circular buffer. The ECADD processor reads from the circular and performs the addition. Figures 4.1 and 4.3 depicts the operation flowchart of the ECDBL processor and ECADD processor respectively.

Figure 4.1: Point doubling Flowchart, Runs on ECDBL processor



The two processors share the circular buffer and a counter. The buffer can be a standard circular buffer and should provide *empty* and *full* flags.

Figure 4.2: Point Doubling Flowchart, Runs on ECADD processor



The ECDBL processor fills up the buffer with 2^iP, and ECDBL processor takes the points from the buffer. If the data in the buffer are not consumed by the ECADD processor the buffer becomes full and the ECDBL processor needs to wait until there is free room in the buffer. On the other hand if there is not enough ones in the binary representation of k, the buffer becomes empty after a while and ECADD processor needs to wait until data is put into the buffer by ECADD processor. In the hardware implementation the buffer should be implemented using dual port RAM/register so that both processors can have simultaneous access to it. In software implementation locking mechanism is needed for accessing the counter and the buffer, since they are accessed from the two processes.

Table 4.3: Point Doubling Algorithm, Runs on ECADD processor

Algorithm: Point Doubling

```
Input: A point P, an integer k = \sum_{i=0}^{n-1} b_i 2^i, b_i ∈ 0,1
Output: 2^i P, Stored in the buffer
Global: i, buffer
Q ← P
i := 0
While i<n
  If b_i = 1 then
    If buffer_full()
      Continue
    put_buffer(Q)
  EndIf
  Q ← 2Q
  i := i +1
EndWhile
```

4.3.1 Performance of the Parallel Algorithm

The performance of the algorithm depends on the ratio of ECADD/ECDBL and the probability of occurrence of nonzero ($1 - P(0)$) in the binary representation of the multiplier k. The ECADD/ECDBL ratio depends on the coordinate system in which the elliptic curve calculation is performed. And $P(nonzero)$ depends on the binary representation form of k. For example in NAF representation $P(nonzero) = \frac{1}{3}$. Table 4.5 summarizes the cost of elliptic curve point calculation in different coordinate systems.

Simulation results of the algorithm are summarized in table 4.6. The results show that when NAF representation for k is used, the algorithm keeps the average number of ECADD operations at about $n/3$, regardless of n and ECADD/ECDBL ratio. The number of extra ECDBLs that we need in addition to $\frac{n}{3}$ECADD depends on ADD/DBL ratio.Therfore for

Table 4.4: Point Adding Algorithm, Runs on ECADD processor

```
Algorithm: Point Addition
Input: 2^i P, Read from the buffer
Output: R = kP
Global: counter i, buffer
R ← O
While i<n Or Not buffer_empty()
  If Not buffer_empty()
    R ← R+ get_buffer()
EndWhile
Return R
```

equal ECADD the faster the ECDBL, the faster the multiplication will be. It can be seen from the results that if ECADD/ECDBL $> P(1)$ then essentially the number of ECDBL remains constant, which means ECDBL is being executed almost always in the background. Running the simulation for $n = 160$ leads to table 4.7 which predicts the execution of the algorithm using different coordinate system for elliptic curve and NAF for representation of k. It can bee seen from table 4.7 that the algorithm is 2 times faster than single processor scalar multiplication method.

4.3.2 Security Against Side Channel Attack (SCA)

The execution time of the algorithm depends on the scalar integer k. For example if $k = 100\ldots1001$ the execution time will be close to nECDBL. In case of $k = 10101\ldots101010$ the execution time will be $\frac{n}{2}$ECADD. Therefore the algorithm cannot be immune to SCA. But, since the execution time depends on the total number of ones and on the distribution of ones, many values of k will have the same execution time. Therefore the algorithm offers better security against SCA when compared to the standard double-and-add methods.

Table 4.5: Execution time of ECADD and ECDBL in different coordinate systems

Coordinate	Transform	ECADD/ECDBL		Field
Affine	(X, Y)	$I + 2M/I + 2M$	$= 1$	$\mathbb{F}_p$
Standard projective	$(X/Z, Y/Z)$	$12M/7M$	=1.7	$\mathbb{F}_p$
Jacobian projective	$(X/Z^2, Y/Z^3)$	$12M/4M$	=3	$\mathbb{F}_p$
Jacobian projective	$(X/Z^2, Y/Z^3)$	$14M/5M$	=2.8	$\mathbb{F}_{2^m}$
Chudnovsky projective	(X, Y, Z, Z^2, Z^3)	$11M/5M$	=2.2	$\mathbb{F}_p$
Lopez-Dahab projective	$(X/Z, Y/Z^2)$	$14M/4M$	=3.5	$\mathbb{F}_{2^m}$

4.4 Conclusion

A parallel method for scalar multiplication is introduced which uses two processors to perform the kP operation. Using proper implementation this method is 200% faster than single processor methods. The method can be implemented both in hardware and software.

Table 4.6: Simulation result of the parallel algorithm

#bits	ADD/DBL	#ECADD	#ECDBL	#Op	#Op Standard DBL-ADD Method	Speed up	Ave. #Data in buff	Max #Data in buff
150	1	50	100	150	200	1.3	0	1
200	1	67	133	200	266	1.3	0	1
250	1	83	167	250	333	1.3	0	1
300	1	100	200	300	400	1.3	0	1
150	2	50	50	150	250	1.7	0	1
200	2	67	66	200	333	1.7	0	1
250	2	83	83	250	416	1.7	0	1
300	2	100	100	300	500	1.7	0	1
150	3	50	10	160	300	1.9	2	4
200	3	67	13	213	400	1.9	2	4
250	3	83	16	266	500	1.9	2	4
300	3	100	19	320	600	1.9	2	4
150	4	50	41	242	350	1.5	3	4
200	4	67	58	325	466	1.4	3	4
250	4	83	75	409	583	1.4	3	4
300	4	100	92	492	700	1.4	3	4
150	5	50	87	337	400	1.2	3	4
200	5	67	120	454	533	1.2	3	4
250	5	83	153	571	666	1.2	3	4
300	5	100	187	688	800	1.2	3	4

Table 4.7: Simulation result for 160-bit scalar, for different coordinate system.

Coordinate	#Proc.	ECADD/ECDBL	#ECADD	#ECDBL	#Op
Affine	2	1	53	106	1440M
Chudnovsky projective	2	$2.2 \approx 2$	53	54	800M
Jacobian projective	2	3	53	8	**672M**
Lopez-Dahab projective	2	$3.5 \approx 4$	53	2	860M
Jacobian projective	1 (Table 3.14)		53	160	1276M

Chapter 5

Architecture for a Fast Elliptic Curve Processor (ECP)

5.1 Introduction

A high performance elliptic curve processor is presented. The processor uses parallelism in instruction level to achieve high speed execution of scalar multiplication algorithm. The architecture relies on compile-time detection rather than run-time detection of parallelism which results in less hardware. Implemented on Xilinx Virtex 2000 FPGA, the proposed processor operates at $66MHz$ in $GF(2^{167})$ and performs scalar multiplication in $100\mu Sec$, which is considerably faster than recent implementations. The $0.18\mu m$ ate level simulation, shows that the processor can at $300MHz$, performing kP in $22\mu Sec$.

Efficient utilization of hardware resources is a key element in a fast processor design. Most fast elliptic curve processors (ECP) use a bit-parallel word-serial (BPWS) finite field multiplier, either in direct form [57] [53] or in Karatsuba form [46] [49] [53] . In all the processors multipliers occupy the bulk of hardware. The proposed architecture maximizes the utilization of the multiplier.

In the field of elliptic curve cryptography, when calculating the speed of a scalar multiplication algorithm, finite field multiplication is considered to be the most time consuming operation. Finite field addition (and squaring in ONB designs) is considered to be free[10] [21] (pp 127-130). It goes to such an extent that in the analysis of scalar multiplication algorithms, the cost of addition is ignored . In some software implementation reports, the cost of addition and squaring is ignored [9] as well. This can be true in software implementations or in hardware designs using serial finite field multipliers (see section 5.2). Considering some high speed hardware designs, we conclude that, the execution time of addition and squaring becomes comparable to execution time of multiplication(table 5.1).

Table 5.1: Typical number of execution cycle of basic FF operations

Design	Multiplication	Addition	Squaring
[46]	9	≥ 2	2 (est.)
[57]	7	3	3
[53]	$12 \leq M \geq 7$	2	2
[62]	7	3	2
presented	8	3	2

Deducing from the above, overlapped execution hardware can be used to increase performance. This approach, which is closer to complex instruction set computer (CISC) design, is successfully employed in [53] to pair multiplication with addition, and multiplication with squaring to increase the performance. However this approach increases the size and complexity of hardware. Using parallelism in instruction level , the compiler analyzes the program and detects operations to be executed in parallel. Such operations are packed into one large instruction. Therefore no hardware in needed for run-time detection of parallelism. The reduced instruction set computer (RISC) type instruction set helps to prepare a more efficient instruction pack(fig. 5.10). The presented processor implements the following features to achieve high execution speed.

- Parallelism in instruction level

- RISC type instruction set
- One cycle instruction execution
- Pipeline finite field multiplier

5.2 Previous Work

The hardware implementation of ECC has come a long way from a modest beginning of ASIC implementation on a 2 micron technology [32] running at $40MHz$ to the 0.13 micron technology running at $500MHz$ [52]. The FPGA implementation started off on Xilinx XC4000 with 2304 slices and 13000 gates [33] and presently is on Xilinx XC2V6000 having 6,000,000 gates running at $100MHz$. [46]

Advances in ASIC and FPGA technologies have led to new architectures and faster designs. Most changes are in the design of the finite field multiplier and in the architecture itself. New designs take advantage of this to introduce more parallelism in finite field calculation.

Elliptic curve cryptosystems can be implemented on $GF(p)$ and $GF(2^m)$. Usually $GF(2^m)$ lead to a smaller and faster design. However, due to pending patents there are some restrictions on $GF(2^m)$ implementations. This thesis mainly discusses $GF(2^m)$ implementations. Based on the design constraints ECPs are implemented using ASIC or FPGA. Elliptic curve hardware implementations can be categorized as follows:

1. Implementations utilizing a general purpose CPU and a finite field accelerator: The early hardware implementations fall into this category [32] and recently [41]. However, because of the evolution of system on chip (SoC) these implementations are becoming attractive [49].

2. Elliptic curve processors (ECP) based on serial finite field multiplier on $GF(2^m)$: These processors are compact but slower than other implementations [45][34].

3. ECPs based on bit-parallel word-serial (BPWS) finite field multiplier on $GF(2^m)$: This architecture results in a fast design and relatively larger hardware. With the

dramatic increase of hardware accommodation , most recent fast designs fall in to this category [46] [57] [53].

4. Processors on $GF(p)$: These processor use modular operations for finite field arithmetic, therefore they utilize more hardware resources and are relatively slower than $GF(2^m)$ implementations [29].

5. Dual field, general purpose crypto-processors: These processors are also available commercially. They work in $GF(p)$ as well as $GF(2^m)$. Since the design in not optimized for $GF(2^m)$ they are usually slower than the third category [38] [52].

Table 5.2 [1] summarizes most published designs. In table 5.3 speed of these implementations are listed. Comparing these designs is not easy, since they have been optimized for different purposes, having different architectures and are implemented on different platforms. Since this work is optimized towards operating speed, in the following sections we compare our results to the faster designs. Wherever possible, we estimate the speed of the design we are comparing to, as if it would be implemented on a hardware similar to ours.

5.3 Elliptic Curve Calculation, Arithmetic Hierarchy

The hierarchy of arithmetics for EC point multiplication is depicted in figure 5.1. The scalar multiplication (kP) algorithm is performed by repeated point addition and doubling operations. The point operations in turn are composed of basic operations in the underlying finite field(FF). The proposed processor performs finite field addition and squaring in one

[1] est.: estimated

FF: Flip Flop LUT: Look Up Table

M.O.: Massey Omura multiplier

ONB: Optimal Normal Basis

Poly.: Polynomial multiplier

Pr.: Presented

Sc.: Scalable, Being able to change both field size and the elliptic curve parameters without reprogramming the hardware

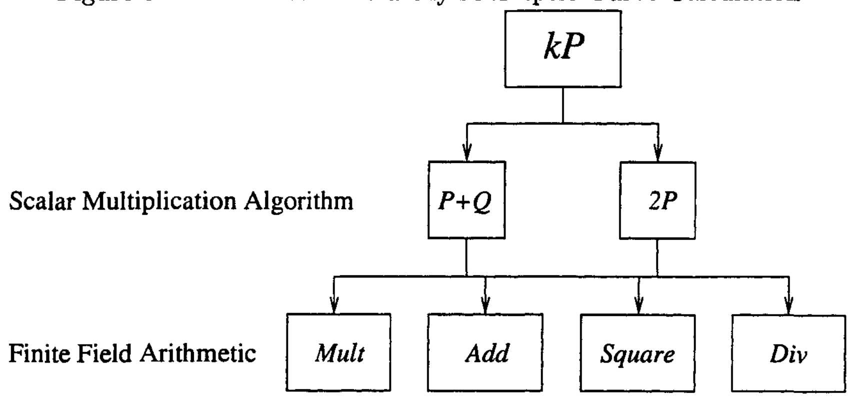
Table 5.2: List of EC hardware implementations

	Platform	Year		HW Res.		Sc.	
[32]	ASIC	1993	ONB	11000	Gates		
[33]	XC4062XL	1998	Poly.	1810	CLB		Only GF((2^4)^9) could be placed and routed
[34]	XCV300-4	2000	ONB	1290	Slice		Only 64bits of k are set to one
[57]	XCV400E	2000	Poly	3002, 1769	LUT, FF		D=16, Montgomery kP
[36]	ASIC 0.25	2000	Poly	165000	Gates	√	Simulation result
[37]	XC4085XLA	2001	M.O.	1450	CLB		Rapid Prototyping, Core Generator
[38]	ASIC 0.25	2001	Poly	880000	Gates	√	Dual Field, Power consideration
[39]	XCV1000	2002	M.O.	48300	LUT		
[41]	XCV2000E	2002	Poly	2790	Slice (est.)		Koblitz Curve
[42]	ASIC 0.35	2002	Poly	14298	Gates		Compact
[43]	XCV1000-6	2002	ONB	2614	Slice		
[44]	XC2S200	2002	Poly			√	Montgomery kP
[45]	ASIC 0.35	2002	ONB	20000	Gates		
[46]	XC2V6000	2003	Poly	19440, 16970	LUT, FF		Clock is Predicted,
[47]	ASIC 0.35	2003	Poly	56000	Gates	√	Montgomery affine, EUA for inverse
[48]	ASIC 0.35	2003	ONB				ALU, Asynchronous
[52]	Asic 0.13	2003	Poly	117500	Gates	√	Dual Field, 500MHz (max) for this particular field
[54]	XC2V2000E-7	2003	Poly	20068, 6321	LUT, FF	√	Montgomery kP, 0.302mSec for unnamed curves
[62]	XC2V2000	2003	Poly	10017, 1930	LUT, FF		
Pr.	XC2V2000	2004	Poly	13900, 3200	LUT, FF		Montgomery kP

Table 5.3: Speed of kp of different ECPs, at the specified finite field, and maximum frequency

	Platform		**GF(2^m)**	**Clk (Mhz)**	**kP(ms)**		**Scalable**
[32]	ASIC	ONB	155	40	27.000	est.	
[33]	XC4062XL	Poly.	8x21	16	4.500	est.	
[34]	XCV300-4	ONB	113	45	3.700		
[57]	XCV400E	Poly	167	76.7	0.210		
[36]	ASIC 0.25	Poly	163	66	1.100		√
	EPF10K250		163	3	80.000		
[37]	XC4085XLA	M.O.	155	37	1.290		
[38]	ASIC 0.25	Poly	160bits	50	5.200	est.	√
[39]	XCV1000	M.O.	191	36	0.270		
[41]	XCV2000E	Poly	176	40	6.900		
[42]	ASIC 0.35	Poly	160	10	20.602	est.	
[43]	XCV1000-6	ONB	113	31	0.810		
[44]	XC2S200	Poly	163	55	3.770		√
[45]	ASIC 0.35	ONB	209	20	30.000	est.	
[46]	XC2V6000	Poly	233	100	0.123	est.	
[47]	ASIC 0.35	Poly	167	100	2.300	est.	√
[48]	ASIC 0.35	ONB	173	Asynch.	1.200	est.	
[52]	Asic 0.13	Poly	160 bits	500	0.190		√
[54]	XC2V2000E-7	Poly	163	66.4	0.143		√
[62]	XC2V2000	Poly	163	66	0.233		
Pr.	XC2V2000	Poly	167	66	0.100		

Figure 5.1: Arithmetic Hierarchy in Elliptic Curve Calculation



clock cycle (excluding register load and unload time). The finite field multiplication is more costly. The number of clock cycles for its computation depends on size of the finite field. Compared to FF-addition and FF-squaring and FF-Multiplication, the FF inversion is a very expensive operation. It is performed by software using basic finite field operations (Sect. 5.3.2).

5.3.1 Finite Field Arithmetic

Elliptic curve calculation over finite fields is based on finite field addition, subtraction, multiplication, squaring and division(Fig. 5.1). Here, we will focus on binary polynomial fields $GF(2^m)$. Using polynomial basis for finite field representation a field element $a \in GF(2^m)$ can be represented as $a = a_{m-1}x^{m-1} + a_{m-1}x^{m-1} + \ldots + a_1x^1 + a_0x_0$ where $a_i \in GF(2)$. Addition of two polynomials a and b is performed by adding coefficient a_i and b_i in *modulo* 2, which is a bitwise XOR operation of a and b. For example, adding two polynomials $a = x^3 + x^2 + 1$ and $b = x^2 + x^1$ can be computed as $(1101 + 0110) = (1101 \quad XOR \quad 0110) = 1011$ or $c = a + b = x^3 + x^1 + 1$. In $GF(2^m)$ calculation addition and subtraction are the same, since $1 + 1 = 0 \quad mod \quad 2$, i.e. 1 is the inverse of 1. It is clear that representing elements of $GF(2^m)$ in a digital computer is easy, since it contains only zeroes and ones(Fig. 5.2).

Multiplication of two elements $a, b \in GF(2^m)$ is carried out by multiplying two poly-

Figure 5.2: Representing an element in Galois field $GF(2^m)$

a_0	a_1	a_2	a_3	a_4	a_5	a_6

A member of GF(2^7)

nomials using the distributive law and then reducing the resultant polynomial in modulo 2 and then modulo $f(x)$. $f(x)$ is of degree of m and defines $GF(2^m)$ for a chosen field of degree m. For example, given polynomials $a = x^3 + x + 1$ and $b = x^3 + 1$ of $GF(24)$, represented as $a = 1011$ and $b = 1001, c_0 = a \times b = x^6 + x^4 + x + 1$ can be computed as:

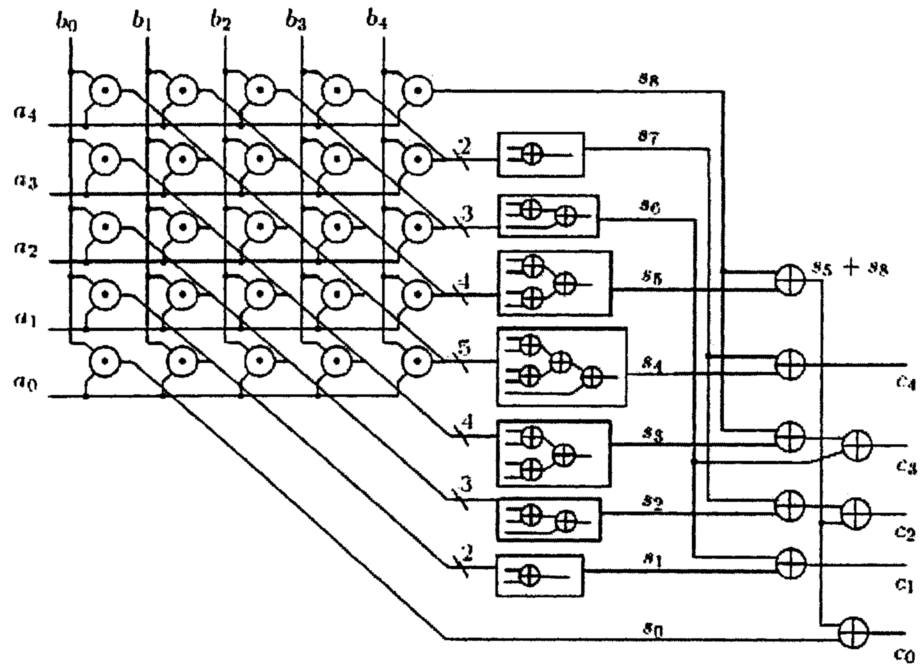
```
1011 x  1001
------------
        1001
+      1001
+     0000
+    1001
------------
=    1010011
```

Assuming $f(x) = x^4+x^3+1$, represented as $f = 11001$, the reduction $c = c_0 \quad mod \quad f = x^2 + 1$ can be performed as:

```
 1010011
+11001
 -------
=0110111
+ 11001
 -------
=0000101
```

An illustrative way to look at reduction is that f is aligned with the most significant bit of the operand and added until the degree of the result is smaller than m. A parallel

Figure 5.3: Parallel Finite Field Multiplier in $GF(2^5)$ [58]



architecture for finite field multiplication is depicted in figure 5.3. An AND gate matrix and an XOR tree performs the multiplication. Squaring can be performed easily using XOR gates, specially if the finite field is defined over a trinomial [58].

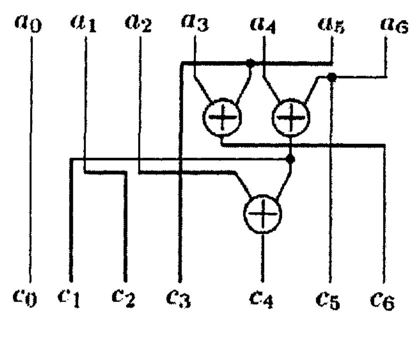
5.3.2 Finite Field Inverse

The multiplicative inverse of any element $a \in \mathbb{F}_{2^m}$ is the element $a^{-1} \in \mathbb{F}_{2^m}$ such that $aa^{-1} = 1 \quad mod \quad f(x)$, where $f(x)$ is the irreducible polynomial of the finite field.

Inversion is the most costly operation in finite field arithmetic. Basically there are two methods for calculating inverse, using Fermat's little theorem and using extended Euclidean algorithm [64].

The Itoh-Tsuji algorithm [59] is the most efficient technique to compute an inverse based

Figure 5.4: Finite Field Squarer in $GF(2^7)$ [58]

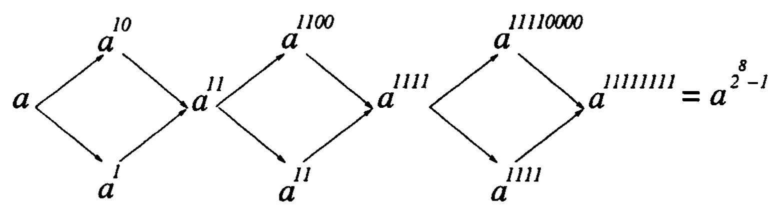


on Fermat's little theorem. Fermat theorem in finite field states that,

$$a^{2^m-1} = 1 \quad mod \quad f(x) \text{ , therefore } \Rightarrow a^{-1} = a^{2^m-2} = (a^{2^{m-1}-1})^2.$$

Figure 5.5 depicts the basic idea in Itoh-Tsuji inverse algorithm, where a^{2^8-1} is calculated in 3 steps ($\log_2 8$). In step n one field multiplication and 2^{n-1} field squaring is needed.

Figure 5.5: Simplified Inverse Calculation



In general a^{2^n-1} can be calculated iteratively using equation 5.1. The complete algorithm for inverse is shown in table 5.4.

$$a^{2^n-1} = \begin{cases} (a^{2^{\frac{n}{2}}})^{2^{n/2}}(a^{2^{\frac{n}{2}}-1}) & n \text{ even} \\ a(a^{2^{\frac{n}{2}}-1})^2 & n \text{ odd} \end{cases} \tag{5.1}$$

Calculating a^{-1} in $GF(2^m)$ needs $M(m) = \lfloor \log_2(m-1) \rfloor + h(m-1) - 1$ multiplication and $m-1$ squaring. where $h(x)$ is hamming weight of x (the number of non-zero bits in the binary representation of x).

Table 5.4: Itoh-Tsuji Inverse Algorithm

Algorithm: Itoh-Tsuji Inverse Algorithm

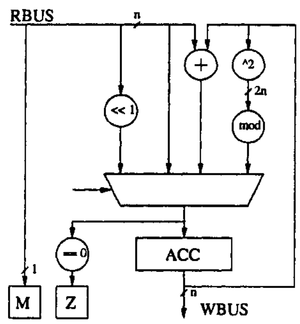
```
Input: a ∈ GF(2^n),   m = Σ_{i=0}^{n-1} m_i 2^i, m_i ∈ {0,1}
Output: b = a^{-1}
b = a^{m_{n-1}}
e = 1
For i = n-2 to 0
  b = b^{2^e} × b
  e = 2e
  if m_i == 1 then
    b = b^2 × a
    e = e + 1
  EndIf
EndFor
b = b^2
Return b
```

If the processor is meant to be used on a single finite field so the squaring can be efficiently optimized [58]. For irreducible polynomial $f(x) = x^m + x^t + 1$ the maximum squarer complexity is $(m + t + 1)/2$ and $4m$ gates for $f(x) = x^m + x^{t1} + x^{t2} + x^{t3} + 1$. For trinomial the critical path delay is at most two gate delays [58].

Since the Itoh-Tsuji inverse algorithm is based on squaring and multiplication, only a small hardware structure is needed for inverse. In fact, in the presented processor inverse is performed by software. In order to perform efficient squaring, `REP SQR A` instruction is defined, which performs squaring in one clock cycle. A data path from accumulator to the squarer makes this instruction possible (Fig. 5.6).

The simulation waveforms which shows the squaring is shown in figure 5.13. For scalable processors using Itoh-Tsuji algorithm is not efficient since squaring hardware cannot be

Figure 5.6: ALU Architecture for calculating Inverse Calculation



optimized for a specific field and therefore cannot be done in a single cycle.

Effect of inverse calculation in performance

For $GF(2^m)$ where $m < 256$ the inversion takes approximately $10M + (m-1)S$. A scalar multiplication using Montgomery method takes $6(m-1)M + 5(m-1)S + 3(m-1)A$. Implemented on an architecture similar to those in table 5.1 for $GF(2^{167})$, inversion time will be about 5% of scalar multiplication time. It can be concluded that fine tuning on the inversion algorithm will not result in a high boost on the overall performance.

5.3.3 Scalar Multiplication Algorithm

Scalar multiplication is the fundamental operation in any elliptic curve cryptosystem. Points on an elliptic curve E over finite field $GF(2^m)$ with a binary operation, called point addition, form an finite additive Abelian group. If P is a point on elliptic curve E and k is a large scalar, computation of the form $Q = kP = \underbrace{P + P + P + \cdots + P}_{k \ \ times}$ is defined as scalar multiplication. The result of scalar multiplication is another point Q on the elliptic curve. The main question in any elliptic curve cryptosystems is: How fast can this operation can be done? Table 5.5 categorizes commonly used methods for fast scalar multiplication [7][10][9]. Selecting a proper method for kP depends on the cryptography protocol being used as well as the implementation platform.

Table 5.5: Classification of scalar multiplication techniques

Name of Method	Basic Idea
Comb [16]	Precompute tables of $\sum_{i=0}^{n-1} 2^{wi}P$
Addition chains [7]	$k = \sum_{i=0}^{n-1} k_i$
Windowing (Fix, Variable) $m-$ary [10]	Precompute tables of $k_iP \quad k_i \in \{0, 1, \ldots, m-1\}$
Scalar recoding [7]	Fewer zero in binary representation of k (NAF)
Point Halving [13] [13]	All point doubling replaced with point halving operation
Montgomery kP method [61]	The x-coordinates of the sum of two points whose difference is known can be computed in terms of x-coordinates of the involved points.
Koblitz curves [2]	Using anomalous binary curves (or ABC's)

In 1987 a new approach to scalar multiplication was proposed by Montgomery[17]. In [61] Montgomery method is converted to projective space and a very efficient scalar multiplication algorithm is derived. Table 5.6 compares the calculation cost of Montgomery method with IEEE standard method. As it is shown implementations based on the Montgomery algorithm are faster. Most high speed ECC implementation in table 5.3, including the proposed processor, have used this algorithm for scalar multiplication[57][53][44][47][52]. The interesting fact about this algorithm is that it is inherently secure against side channel attack. In the proposed architecture, the algorithm is tuned for the pipeline multiplier and the processor's parallel architecture. The complete explanation of Montgomery scalar multiplication is given in chapter 3.

Table 5.6: Cost of scalar multiplication on $GF(2^m)$ for different algorithms

Scalar Multiplication Algorithm	# Operations
Montgomery, Projective version [61]	(m-1)(6M+3A+5S) + (10M+7A+4S+I)
IEEE 1362, NAF representation (Average) [21]	(m-1)(8.7M+6.3A+6.3S) +(3M+S+I)

5.3.4 Performance Estimation for ECPs Based on BPWS Multipliers

Minimum number of clock cycle for kP calculation

The lower and upper bound of performance for the architectures which use Bit Parallel Word Serial (BPWS) multipliers can be estimated as follows. The multiplication takes $M = \lceil m/D \rceil + 3$ cycles, assuming 2 clock cycles for loading the input registers of the multiplier and one cycle for storing the result. Although addition and squaring are performed in one cycle, extra cycles are needed to load and unload the registers, therefore $A = 3$ cycles for addition and $S = 2$ cycle for squaring is assumed. Using Montgomery scalar multiplication [61], the upper bound (UB) is derived in table 5.6. At the best case, where all additions and squaring operations can be performed in parallel with multiplication (we assume $M > A$, $M > S$) the lower bound (LB)can be calculated by omitting all additions and squaring operations. Therefore we will have,

$$\begin{aligned} UB &= (m-1)(6M+3A+5S) + (10M+7A+4S+I) \\ LB &= (m-1)(6M) + (10M+I) \\ &\text{where } M = \lceil m/D \rceil + 3, A = 3, S = 2, I \approx 10M + (m-1)S \end{aligned} \tag{5.2}$$

Experimenting with the processor architecture shows that the $\lceil m/D \rceil = 4$ ratio minimizes the number of multiplication cycles but is long enough to let additions and/or squaring to be done in parallel with multiplication. Therefore the lower bound for kP can be approximated as

$$LB \approx 43(m-1). \tag{5.3}$$

Unless a proper pipeline mechanism is used, faster operation cannot be achieved using this class of architecture.

Critical Path length

If the finite field $GF(2^m)$ is generated by and irreducible polynomial $f(x)$ then the maximum critical path is equal to $C_T = T_A + (\lceil \log_2 m \rceil + (r-1))T_X$ where r is the number of terms in the irreducible polynomial $f(x)$. In BPWS multipliers where,

$$A(x) = \sum_{i=0}^{m-1} a_i x^i \text{ , and } B(x) = \sum_{i=0}^{D-1} b_i x^i, \text{ , where } a_i, b_i \in \{0,1\}$$

the critical path will be

$$C_T = T_A + (\lceil \log_2 D \rceil + (r-1))T_X \tag{5.4}$$

, where T_X and T_A are the delays of AND gate and XOR gate. Using irreducible trinomial this can be further reduced to $C_T = T_A + (\lceil \log_2 (m-1) \rceil + 2)T_X$ [58]. C_T determines the upper bound for the clock frequency of the ECP.

5.4 Design Flow

The presented crypto-processor requires components that operate on large bit vectors (167 bits on $GF(2^{167})$). This makes validation of synthesis results difficult and time consuming due to large amount of simulation elements. The complexity often can be reduced by scaling the signal vectors down. Adding such flexibility is excess work, but it pays off. The processor is designed to work with any finite field which is based on a trinomial or a pentanomial. Therefore most validations were performed on small fields like $GF(2^{15})$.

The design flow is depicted in Fig. 5.4. A bit-exact C program was developed, which allows us to check the HDL thoroughly. Test vectors for Galois field of different sizes were applied to both the HDL and the bit-exact program, and the results were checked against each other using another program to ensure the proper operation of the hardware. An assembler program for the crypto-processor is also developed which lets us to assemble programs written for the processor. The processor was synthesized and optimized using Synopsys Design Analyzer ® for CMOS 0.18 and Xilinx ISE® for FPGA.

Figure 5.7: Elliptic Curve Processor Design Flow

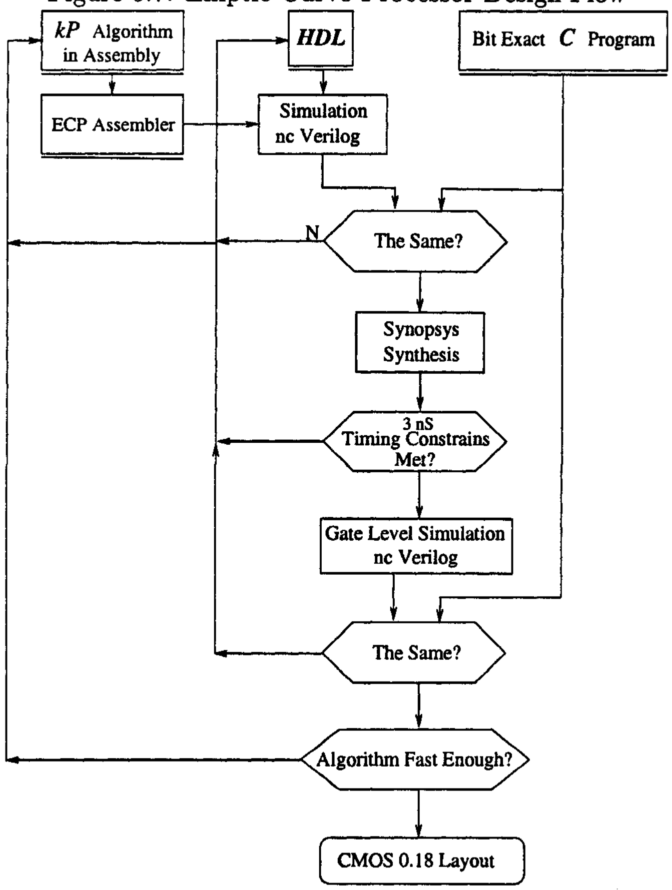
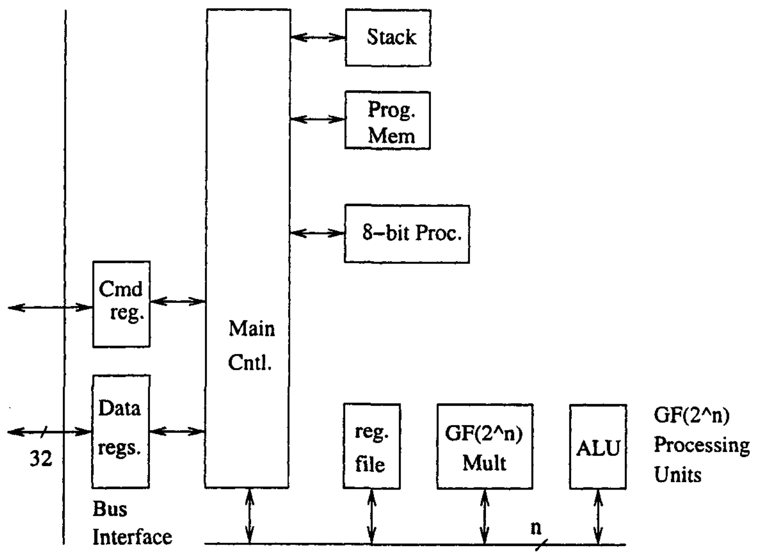


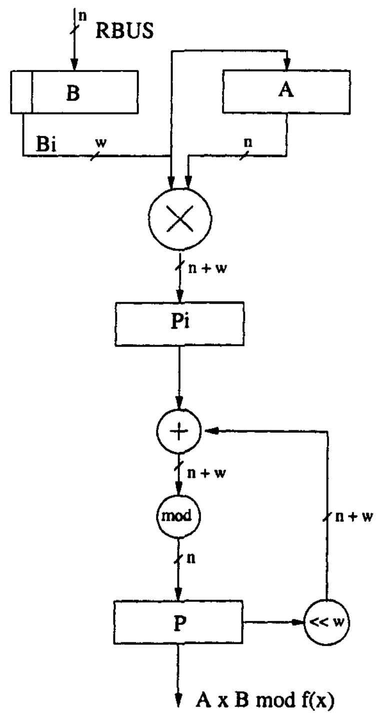
Figure 5.8: Architecture of the Processor



5.5 Architecture

The architecture is highly optimized toward the execution of scalar multiplication algorithm. It supports finite field arithmetic, some 8 bits integer calculation and control transfer instructions. The finite field arithmetic unit utilizes parallelism in instruction level , which permits parallel execution of addition, squaring and multiplication. The finite field processing unit consists of an ALU, a multiplier and a register file. These units are controlled by the main control unit. In addition, a very small 8-bit processor is provided which performs integer calculations like counting and shifting. The communication with the host processor is implemented through utilization of a command register and a data register. Initially, the host processor uploads elliptic curve domain parameters and the code using these two registers (Fig. 5.8). From then on, communication is limited to the exchange of raw and processed data. Utilization of communication registers allows the two processors to operate independently, and have different clock signals. The processor is implemented in $GF(2^{167})$ but neither the scalar multiplication code nor the architecture is hardwired to the size of the Galois Field.

Figure 5.9: Architecture of the Finite Field Multiplier



Multiplier

The number of finite field multiplication in a scalar multiplication is approximately $6(m-1)$ for $GF(2^m)$(Table 5.6). Therefore a high performance multiplier is very crucial. ALU uses a bit parallel word serial (BPWS) multiplier based on the algorithm in [60]. In order to achieve a performance better than $LB \approx 43(m-1)$, the input registers A and B, intermediate register P_i and output register P are configured as a pipeline (Fig.5.9). This arrangement permits a finite field multiplication to be performed in $M = \lceil m/D \rceil + 1$ cycles, which would otherwise take $M = \lceil m/D \rceil + 3$ in similar designs [53] [57].

Squarer

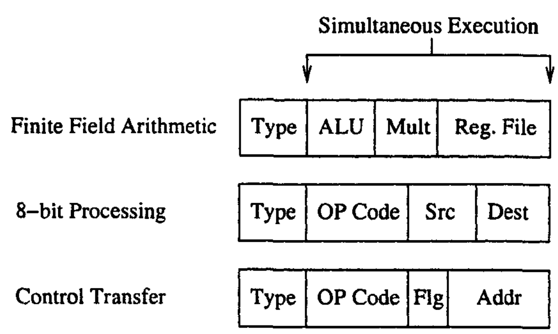
The ALU employs a bit-parallel squarer [58]. Synthesized for a specific Galois field, this squarer leads to a very efficient hardware which performs the squaring in one clock cycle.

Scalable [2] ECP implementations cannot use this architecture, since the size of finite field is not known at the time of hardware synthesis. Therefore they have relatively longer kP execution time [36][38][44][47].

Instruction Set

The instruction set is sub divided into three categories: Finite field arithmetic, integer processing and control transfer (Fig. 5.10, table 5.7). Finite field arithmetic instructions are further split into three threads. The compiler analyzes the scalar multiplication program and detects finite field operations to be executed in parallel. Such operations are packed into one *finite field arithmetic* type instruction.

Figure 5.10: Instruction set categories



5.6 Implementation

5.6.1 HDL Simulation

HDL simulation is carried out using Cadence NCVerilog®. Figure 5.11 and 5.12 shows the waveforms at the beginning and end of the simulation on $GF(2^{167})$. The hardware was simulated and tested for $GF(2^{16})$, $GF(2^{167})$ and $GF(2^{233})$ using 1000, 100 and 10 random test vectors respectively. The simulation takes 6660 clock cycles on $GF(2^{167})$ which is

[2] Being able to change both field size and the elliptic curve parameters without reprogramming the hardware

$0.1mSec$ at $66MHz$. In terms of execution speed, this result is faster than similar FPGA implementations [53][46][62][57].

5.6.2 Synthesis Result

FPGA

The HDL is synthesized for Xilinx XC2V2000 FPGA using Xilinx tools. Table 5.10 summarizes the hardware resource usage of the processor in terms of lookup tables (LUT) and flip-flops (FF) in FPGA implementation. The processor operates at $66MHz$ and performs the scalar multiplication in $GF(2^{167})$ in $100\mu Sec$. The synthesis result shows that the maximum operation frequency for the processor is $90MHz$.

ASIC Simulation

The processor is synthesized and simulated for TSMC CMOS 0.18 technology using Synopsys® and Cadence NCVerilog®. Using synthesis information obtained from Synopsys®, the performance and the hardware size of the processor on TSMC 0.18 μm technology is obtained. The hardware size is about 36000 gates and the clock frequency can be as high as $300MHz$. For the proposed architecture we have $r = 3, D = 42, T_A \approx T_X \approx 0.3nSec$ (from Synopsys report). Putting into equation 5.4 results to $C_T \approx 9T_X = 2.7nSec$. Synopsys report shows that the critical path equals to $3.2nSec$. This confirms that the proposed architecture satisfies the critical path bound Implemented on ASIC. It takes $22\mu Sec$ to complete one scalar multiplication operation in $GF(2^{167})$, which is faster than reported ASIC implementations. Table 5.8 summarizes the synthesis results in CMOS 0.18.

ASIC Implementation

The ASIC design flow in fig. 5.4 is carried out to the very end. ie. The CMOS 0.18 layout is implemented using Cadence SoC Encounter. This layout is ready for fabrication. Refer to appendix for a snap shop of the layout.

5.6.3 Performance and comparison

Table 5.9 shows the number of clock cycle needed to execute kP, for several processors. These processors have the following specifications in common:

- They are among the fastest implementations of ECP (see Table 5.3).
- They are implemented on an advanced FPGA architecture.
- All use parallel polynomial based finite field multipliers.
- Number of clock cycles needed to perform kP is linearly dependent on field size m (If we keep the size of m/D in finite field multiplier constant, where D is the sized of digit or word in the bit-parallel word- serial multiplier).
- They Perform inverse using Itoh-Tsuji algorithm (except [53]).
- All Use Projective coordinates for kP calculation (most use BPWS).

It can be concluded that, for non scalable ECP processors, these specifications lead to an efficient design. Among them, the proposed architecture needs less clock cycles to perform scalar multiplication. Another important factor in the architecture is the maximum critical path in the processor. However it is not easy to estimate what the maximum clock rate for [57] [46] would be if they would have been implemented on the a platform like ours. Simulation shows that the proposed processor can run at $300MHz$ when implemented on CMOS 0.18 technology, which is the minimum possible critical path for this type of architecture. This is also a good number compared to the designs in tables 5.3 and 5.2.

5.7 Conclusion

An architecture for an Elliptic curve processor is proposed. The processor can perform $10,000$ scalar multiplications per second on $GF(2^{167})$, which is considerably faster that the recent FPGA implementations. The processor has a very short critical path which is on the parallel multiplier. Synthesis results in CMOS 0.18 micron show that the proeessor can run at $300MHz$ clock frequency which results in $22\mu Sec$ for a scalar multiplication on $GF(2^{167})$. The synthesis result confirms that the design satisfies the critical bound.

Table 5.7: Elliptic Curve Processor Instruction Set

8-bit processor	
`MOV rx, d8`	move immediate data to rx register
`DJNZ rx, addr`	decrement rx jump to addr if not zero
`DEC rx`	decrement rx
`INC rx`	increment rx
`SHL {c,rx}`	shift left Carry and rx
`SHL {rx,c}`	shift left rx and Carry
`MOV ry, rx`	move rx to ry
FF Arithmetic Unit	
`SQR A`	
`ADD A, Rx`	
`SHL A`	
FF Multiplier	
`START Mul`	
`STOP Mul`	
Register File	
`MOV Rx, P`	move product to Rx
`MOV Rx, A`	
`MOV A, Rx`	
`MOV S, Rx`	load multiplier register with Rx
Control Transfer	
`JMP flg,set, addr`	flg is Z (Zero flag), C (Carry flag), M (User flag)
`CALL flg,set, addr`	
`SET M`	
`CLR M`	
`RET`	
`HALT`	

Table 5.8: Area report in CMOS 0.18

Unit	Area (micron)
Multiplier	1272102
ALU	28585
Squarer	4976
Register File	202799
Proc8	5617
Total	≈ 1555271

Table 5.9: Number of clock cycles for kP

Design	Number of Clk for kP	Point Representation
Presented	$39(m-1)$+inv.	Montgomery Projective
[46]	$44(m-1)$+inv. (est.)	Projective with NAF [a]
[57]	$47(m-1)$+inv. (est.)	Montgomery Projective, D=42 [b]
[53]	$57(m-1)$ (est.)	Montgomery Projective
[62]	$93(m-1)$+inv. (est.)	Projective with NAF

inv. $= (m-1) + M(\lfloor \log_2(m-1) \rfloor + h(m-1) - 1), M \approx 7$

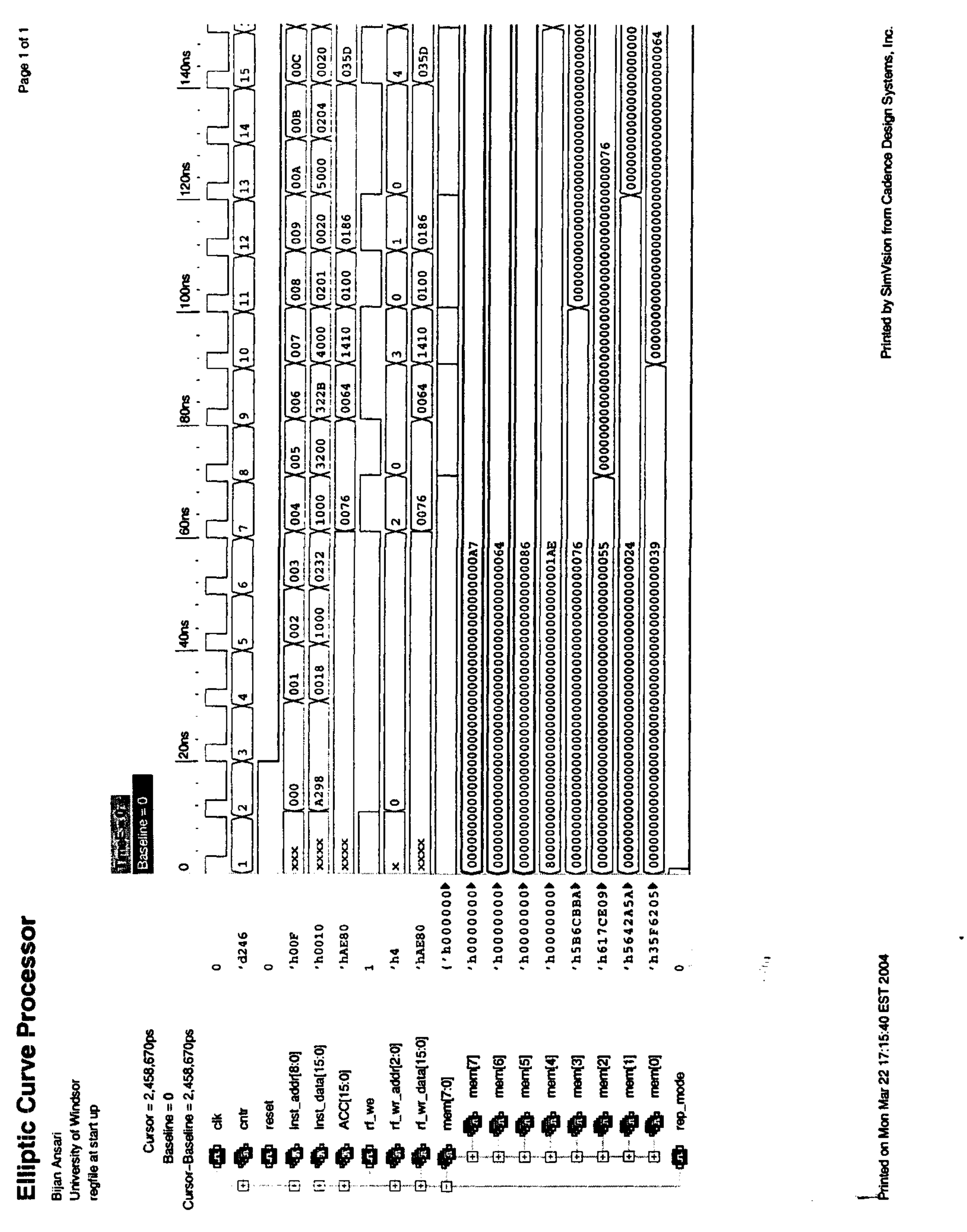
[a] In [46] authors didn't assume NAF representation for scalar k.

[b] In [57], maximum D is 16. Probably they were not able to use D=42 due to limited resource in their FPGA. We assume D=42 here.

Table 5.10: Performance of the Elliptic Curve Processor

Design	kP mSec	Inversion Cycle	$GF(2^m)$	FPGA LUT, FF	Clk MHz	FPGA	Year
Proposed	0.100	285	167	7562, 2378	66	XCV2000	2004
[57]	0.210		167	3000, 1769	76.6	XCV400E	2000
[53]	0.143	326= $2m$	163	20068, 6321	66.4	XCV2000	2002
[62]	0.233	250	163	10017, 1930	66	XCV2000	2003
Proposed	0.140	451	233	13900, 3200	66	XCV2000	2004
[46]	0.123 est.	-	233	19440, 16970	100	XCV6000	2003

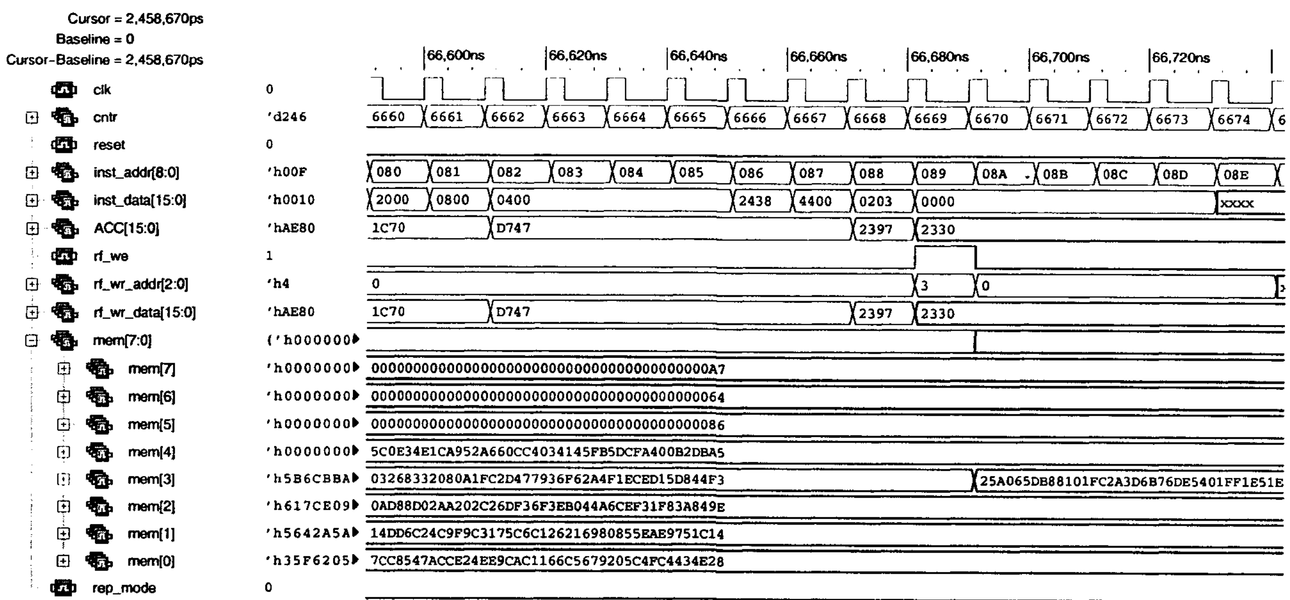
Figure 5.11: Simulation Waveforms at startup



Elliptic Curve Processor

Page 1 of 1

Bijan Ansari
University of Windsor
regfile after kP

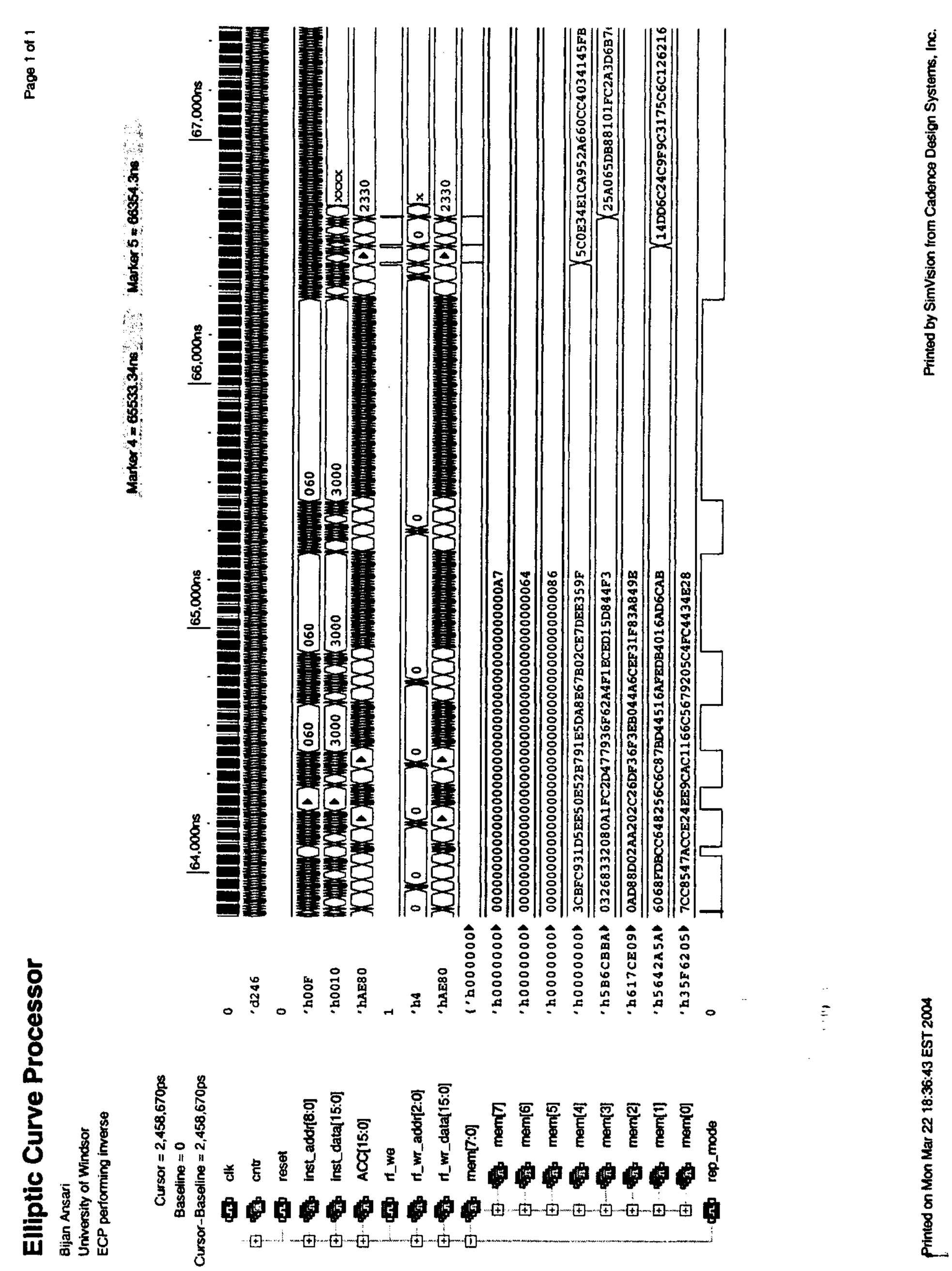


Printed on Mon Mar 22 17:18:17 EST 2004

Printed by SimVision from Cadence Design Systems, Inc.

Figure 5.12: Simulation Waveforms at the end of calculation

Figure 5.13: Simulation Waveforms while calculating Inverse



Chapter 6

Discussions

6.1 Summary of Contribution

This work proposes efficient methods for ECC both in algorithm level and in arithmetic level. In algorithm level a parallel method for scalar multiplication is introduced which uses two processors to perform the kP operation. Using proper implementation this method is 200% faster than conventional single processor methods. The method can be implemented both in hardware and software.

At the arithmetic level, a high performance elliptic curve processor architecture on $GF(2^m)$ is proposed. The architecture employs parallel execution of finite field arithmetic, to achieve high execution speed. Implemented on Xilinx Virtex 2000 FPGA, the processor can perform 10,000 scalar multiplications per second on $GF(2^{167})$, which is considerably faster that the recent FPGA implementations. The processor has a very short critical path which is on the parallel multiplier. Synthesis results on CMOS 0.18 micron show that the processor can run at $300MHz$ clock frequency which results in $22\mu Sec$ for a scalar multiplication on $GF(2^{167})$. The processor is compared to various ECC hardware implementations. The comparison is limited to the processors on $GF(2^m)$. The processor speed presented is higher than any other reported ECC hardware implementation.

6.2 Future Work

FPGAs are a suitable platform for the hardware implementation of the proposed parallel algorithm. The information in chapter 3 can be used for the selection of proper point representation system. For the proposed processor ASIC implementation is very desirable since the simulation results shows the it will be the fastest kP calculation ever reported.

References

[1] Neal Koblitz "A course in Number Theory and Cryptography" *Spreiger Verlag* 1987

[2] Neal Koblitz "CM-Curves With Good Cryptographic Properties" *CRYPTO '91, Springer-Verlag LNCS 756* pp 279-287, 1992.

[3] Neal Koblitz "Elliptic curve cryptosystems." "Mathematics of Computation, 48:203-209, 1987.

[4] V.S. Miller "Use of elliptic curves in cryptography" Advances in Cryptology Proc. Crypto'85 *LNCS 218, H.C. Williams, Ed., Springer-Verlag, 1985, pp. 417-426*

[5] V.S. Miller "Elliptic curve cryptosystems" Advances in Cryptology Proc. Crypto'85 *Mathematics of Computation, Vol. 48, no. 177 (1987), pp. 203–209*

[6] Andreas Enge "Elliptic Curves and their applications to cryptography: An introduction" *Kluwer academic press* 1999

[7] Daniel M. Gordon "A Survey of Fast Exponentiation Methods" *journal of algorithms* 27, 129-146 (1998),

[8] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono "Efficient Elliptic Curve Exponentiation Using Mixed Coordinates" *ASIACRYPT'98, Springer-Verlag LNCS 1514,* pp. 51-65, 1998.

[9] Darrel Hankerson, Julio Lopez Hernandez, and Alfred Menezes "Software Implementation of Elliptic Curve Cryptography over Binary Fields" *CHES 2000, Springer-Verlag LNCS 1965,* pp. 1-24, 2000

[10] Ian Blake, Gadiel Seroussi and Nigel Smart "Elliptic Curves in Cryptography" *Cambridge University Press* 2002

[11] Jerome A. Solinas "An Improved Algorithm for Arithmetic on a Family of Elliptic Curves" *CRYPTO '97, Springer-Verlag LNCS 1294* pp. 357-371, 1997.

[12] Knuth D. E. Knuth "Seminumerical Algorithms"

[13] E. Knudsen "Elliptic Scalar Multiplication Using Point Halving" *Proc. Advances in Cryptology Asiacrypt 99* pp. 135-149, 1999.

[14] FIPS 186-2 "Digital Signature Standard (DSS)" Federal Information Processing Standards Publication 186-2, National Institute of Standards and Technology, 2000.

[15] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms" *CRYPTO 2001, LNCS 2139,* pp. 190-200, 2001.

[16] C. Lim and P. Lee, "More flexible exponentiation with precomputation", *Advances in Cryptology,* Crypto '94, 1994, 95-107.

[17] P. Montgomery "Speeding the Pollard and elliptic curve methods of factorization", *Mathematics of Computation,* vol 48, pp.243-264.

[18] J. Lopez and R. Dahab "Improved algorithms for an elliptic curve arithmetic in $GF(2^n)$" *Selected areas in Cryptography - SAC 98, Springer-Verlag, LNCS 1556,* 1996

[19] B. Moller "Parallelizable Elliptic Curve Point Multiplication method with Resistance against Side-Channel Attacks" *Information Security ISC-2002, Springer-Verlag LNCS 2433*

[20] L.C. Washington "Elliptic Curves, Number Theory and Cryptography" *Chapman & Hall/CRC,* 2003

[21] IEEE "IEEE 1363-2000 standard" *Standard specification for public key cryptography.*

[22] D. Boneh and N.Daswani "Experimenting with electronic commerce on The Palm Pilot" *Financial Cryptography '99, Springer-Verlag LNCS 1423* pp. 1-16, 1999

[23] J. Solinas, "Mersenne numbers" *Technical Report CORR 99-39, University of Waterloo, 1999*

[24] D. Bailey and C. Paar, "Extension fields for fast arithmetic in public-key algorithms" *Advances in Cryptology, Crypto '98, 1998,* pp 472-485

[25] D. Gollmann, Y. Han and C. Mitchell, "integer representations and fast exponentiation" *Designs, Codes and Cryptography, 7 (1996)* pp.135-151

[26] A. Menezes, P. van Oorschot and S. Vanstone, "Handbook of Applied Cryptography" *CRC Press, 1996*

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

[28] W. Diffie, and M.E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory,* , vol. 22, no. 6, November 1976, pp. 644-654.

[29] S.B. Ors, L. Batina, B. Preneel and J. Vandewalle, "Hardware implementation of an elliptic curve processor over GF(p)", *IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2003. Proceedings,* pp. 433 - 44324-26 June 2003.

[30] Gutub A.A.-A., Ibrahim M.K., "High radix parallel architecture for GF(p) elliptic curve processor", *International Conference on Acoustics, Speech, and Signal Processing (ICASSP '03)* Volume: 2 , 6-10 April 2003 Pages:II - 625-8 vol.2

[31] Gutub A.A.-A., Ibrahim M.K., "High performance elliptic curve $GF(2^k)$ crypto-processor architecture for multimedia" *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference on* , Volume: 3 , 6-9 July 2003 Pages:81 - 84

[32] Agnew, G.B., Mullin, R.C.,Vanstone, S.A. "An implementation of elliptic curve cryptosystems over $GF(2^{155})$" *IEEE Journal on Selected Areas in Communications,* Volume: 11 , Issue: 5 , pp. 804 - 813, June 1993.

[33] M. Rosner. "Elliptic curve cryptosystems on reconfigurable hardware", *Master's thesis, ECE Dept., Worcester Polytechnic Institute,* Worcester, USA, May 1998.

[34] "K.H. Leung , K.W. Ma , W.K. Wong , P.H.W. Leong ", FPGA implementation of a microcoded elliptic curve cryptographic processor, *IEEE Symposium on Field-Programmable Custom Computing Machines, 2000* , pp. 68 - 76, 17-19 April 2000.

[35] Gerardo Orlando and Christof Paar "A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$" *CHES 2000, LNCS 1965, pp. 41-56* 2000

[36] S. Okada, N. Torii, K. Itoh and M. Takenaka "Implementation of Elliptic Curve Cryptographic Co-processor over $GF(2^m)$ on an FPGA", *CHES 2000,* LNCS 1965, pp. 25-40, 2000.

[37] Ernst M., Klupsch S., Hauck O., Huss S.A., "Rapid prototyping for hardware accelerated elliptic curve public-key cryptosystems", *International Workshop on Rapid System Prototyping,* pp.24 - 29, 25-27 June 2001.

[38] Goodman J., Chandrakasan A.P., "An energy-efficient reconfigurable public-key cryptography processor", *IEEE Journal of Solid-State Circuits,* Volume: 36 , Issue: 11 , Nov. 2001 Pages:1808 - 1820,

[39] M. Bednara ,M. Daldrup, J. von zur Gathen , J. Shokrollahi , J. Teich , "Reconfigurable implementation of elliptic curve crypto algorithms", *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM* , Pages:157 - 164, 15-19 April 2002.

[40] M. Bednara, M. Daldrup , J. Teich , J. von zur Gathen, J. Shokrollahi, "Tradeoff analysis of FPGA based elliptic curve cryptography", *IEEE International Symposium on Circuits and Systems, 2002. ISCAS 2002,* Volume: 5 pp.V-797 - V-800, 26-29 May 2002.

[41] T. Kerins, E. Popovici, W. Marnane , and P. Fitzpatrick "Fully Parameterizable Elliptic Curve Cryptography Processor over $GF(2^m)$", *Springer-Verlag, LNCS 2438,* pp. 750-759, 2002.

[42] K. Ju-Hyun, L. Dong-Ho, "A compact finite field processor over $GF(2^m)$ for elliptic curve cryptography", *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on , Volume: 2 , 26-29 May 2002* Pages:II-340 - II-343 vol.2.

[43] P.H.W. Leong ,I.K.H. Leung , "A microcoded elliptic curve processor using FPGA technology", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Volume: 10, Issue: 5, Pages:550 - 559, Oct. 2002.

[44] M.J. Potgieter , B.J. van Dyk, "Two hardware implementations of the group operations necessary for implementing an elliptic curve cryptosystem over a characteristic two finite field", *African Conference in Africa, 2002. IEEE AFRICON. 6th ,* Volume: 1 , 2-4 Oct. 2002, Pages:187 - 192.

[45] X. Zeng, X. Zhou, Q. Zhang, "Hardware/software co-design of elliptic curves public-key cryptosystems", *IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions,* Volume: 2 , pp. 1496 - 1499 vol.2, 29 June-1 July 2002.

[46] Grabbe C., Bednara M., von zur Gathen J., Shokrollahi J., Teich J., "A high performance VLIW processor for finite field arithmetic", *Proceedings of the International Parallel and Distributed Processing Symposium,* 22-26 April 2003.

[47] H. Chi, L. Jimnei, Junyan Ren, Qianling Zhang, "Scalable elliptic curve encryption processor for portable application", 5th International Conference on ASIC, 2003. Proceedings, Volume: 2 , Pages:1312 - 1316, Oct. 21-24, 2003.

[48] L. Pak-Keung ,C. Chiu-Sing ,C. Cheong-Fat ,P. Kong-Pang, "A low power asynchronous $GF(2^{173})$ ALU for elliptic curve crypto-processor", *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03.,* Volume: 5, Pages:V-337 - V-340, 25-28 May 2003.

[49] M. Jung, F. Madlener, M. Ernst and S. A. Huss "A Reconfigurable Coprocessor for Finite Field Multiplication in $GF(2^n)$", *IEEE Workshop on Heterogeneous reconfigurable Systems on Chip,* Hamburg, April 2002.

[50] L. Gao , L. Hanbo, G.E. Sobelman , "A compact fast variable key size elliptic curve cryptosystem co-processor", *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings.* Pages:304 - 305, 21-23 April 1999.

[51] G. Lijun, S. Shrivastava, G. E. Sobelman A1, "Elliptic Curve Scalar Multiplier Design Using FPGAs", *Springer-Verlag,* LNCS, Volume 1717 / 1999.

[52] A. Satoh, K. Takano, "A scalable dual-field elliptic curve cryptographic processor", *IEEE Transactions on Computers,* Volume: 52 , Issue: 4 , pp. 449 - 460, April 2003.

[53] Nils Gura, Sheueling Chang Shantz, Hans Eberle, Sumit Gupta, Vipul Gupta, Daniel Finchelstein, Edouard Goupy, Douglas Stebila, "An End-to-End Systems Approach to Elliptic Curve Cryptography" *Sun Microsystems Laboratories* 2002-2003

[54] N. Gura, S.C.Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchelstein, E. Goupy, D. Stebila, "An End-to-End Systems Approach to Elliptic Curve Cryptography", *Sun Microsystems Laboratories* 2002-2003.

[55] N. Nguyen, K. Gaj , D. Caliga , T. El-Ghazawi, "Implementation of elliptic curve cryptosystems on a reconfigurable computer", *IEEE International Conference on Field-Programmable Technology (FPT), 2003. Proceedings. 2003,* pp. 60 - 67, 15-17 Dec. 2003.

[56] Hauck O., Katoch A., Huss S.A., "VLSI system design using asynchronous wave pipelines: a 0.35μm CMOS 1.5 GHz elliptic curve public key cryptosystem chip", *Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems, (ASYNC 2000) Proceedings,* pp. 188 - 197, 2-6 April 2000.

[57] Gerardo Orlando and Christof Paar "A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$" *CHES 2000, LNCS 1965, pp. 41-56* 2000

[58] H. Wu, "Bit-Parallel Finite Field Multiplier and Squarer using polynomial basis", *IEEE transaction on Computers,* Vol. 51, No.7, July 2002.

[59] T. Itoh and S.Tsuji "A Fast algorithm for computing multiplicative inverse in $GF(2^m)$ using normal bases", Info. and Comput., col. 78(3), pp.171-177, 1998.

[60] L. Song and K. K. Parhi. "Low-energy digit-serial/parallel finite field multipliers", *Journal of VLSI Signal Processing Systems,* pp. 1-17, 1997.

[61] J. Lopez and R. Dahab "Fast Multiplication on Elliptic Curves over $GF(2^n)$ without Precomputation" *CHESS'99, Springer-Verlag, LNCS 1717, pp. 316-327* 1999

[62] Jonathan Lutz "High performance elliptic curve cryptographic co-processor" *Masters thesis, University of Waterloo* 2003

[63] S.Galbraith and N. Smart "A cryptographic application of Weil descent" *Codes and Cryptography, LNCS 1746, Springer-Verlag, pp. 191-200,* 1999

[64] D. W. Hardy "Applied algebra, Codes. Cipher and Discrete Algorithms" *Prentice Hall,* 2003

Appendix A

Test Code

```
#include <stdlib.h>
#include <stdio.h>
#include "gmp.h"
#include <time.h>
#include <math.h>

//typedef unsigned long long scalar_t;
#define scalar_t mpz_t

int   get_bit        (scalar_t k, int i);
void  set_bit        (scalar_t k, int i);
void  clr_bit        (scalar_t k, int i);
int   kP_time_s      (char * ks, int t_add, int *na, int *nd);
int   kP_time_s2     (char * ks, int ADD_DBL_ratio);
char *str_reverse    (char *d, char *s);
void  to_NAF         (scalar_t k);
void  to_NAF2        (scalar_t k);
char *itos           (scalar_t k);
void  test_recording(void);
int   kP_time        (scalar_t k, int n_bits, int t_add, int *na, int *nd,
                      int *cnt_in_add_ave, int *cnt_in_add_max, int buf_len
                      );
int   ave_kP_time    (int n_samples, int n_bits, int ADD_DBL_ratio, int
                      buf_len);
//#define ADD_DBL_RATIO 3
```

```

int main(void)
{
    //performance_table();
    performance_vs_buflen_graph();

    return 0;
}

int performance_table(void)
{
  int n_bits, t;
  char *s, d[50];
  time_t rawtime;

  //- algorithm parameters
  int ADD_over_DBL_ratio = 3;
  int buf_len = 4;
  int nsamples = 10000;

  time ( &rawtime ); printf("\n%s\n\n",ctime(&rawtime));
  printf("#Samples = %i", nsamples);
  printf("\n\\#bits &  ADD/DBL &  \\#ECADD & \\#ECDBL & \\#Op & \\#Op Std
         DBL-ADD Method & Ave \\#Data in buf & Max \\#Data in buf & Speed
         up \\hline\\hline");
  printf("\n==================================================");

  for(ADD_over_DBL_ratio=1; ADD_over_DBL_ratio<6; ADD_over_DBL_ratio++)
  {
      printf("\n-------------------------------------------\\hline");

      //nsamples = 0x7FFFFFFF; //result of the scount takes 2 days and is
                   wrong!
      for(n_bits=150; n_bits<=300; n_bits+=50)
          ave_kP_time(nsamples, n_bits, ADD_over_DBL_ratio,buf_len );
  }

   /*
  test_recording(); printf("\n\n");

  s = "1010101000000111111111110000011000000001";
  t = kP_time_s2(s, ADD_over_DBL_ratio);

  str_reverse(d, s);
```

```
  t = kP_time_s2(d, ADD_over_DBL_ratio);
   */
  time ( &rawtime ); printf("\n%s\n\n",ctime(&rawtime));

  return 0;
}

int performance_vs_buflen_graph(void)
{

  int ADD_over_DBL_ratio = 3;
  int buf_len = 4;
  int nsamples = 100;
  int n_bits;

  printf("#Samples = %i\n\n", nsamples);

  //for(n_bits=150; n_bits<=300; n_bits+=50)
      n_bits = 160;


     for(buf_len=1; buf_len<=10; buf_len++)

      {
           printf("\n %i  ", buf_len);
          for(ADD_over_DBL_ratio=1; ADD_over_DBL_ratio<6;
              ADD_over_DBL_ratio++)
          {
              ave_kP_time(nsamples, n_bits, ADD_over_DBL_ratio, buf_len );
          }
           printf("    ", buf_len);
      }

      /*
      for(ADD_over_DBL_ratio=1; ADD_over_DBL_ratio<6; ADD_over_DBL_ratio++)
      {
          printf("\n\n #ADD/DBL = %i   ", ADD_over_DBL_ratio);
          for(buf_len=1; buf_len<=10; buf_len++)
          {
              printf("\n %i  ", buf_len);
              ave_kP_time(nsamples, n_bits, ADD_over_DBL_ratio, buf_len );
          }
      }*/

```

```
  return 0;
}

#define MAX_INT ~((unsigned long int) 1)
#define ave(i) ((int)((double)(i)/(n_samples)+0.5))
int ave_kP_time(int n_samples, int n_bits, int ADD_DBL_ratio, int buf_len)
{
  mpz_t              k;
  gmp_randstate_t    r_state;
  int                na, nd, cnt_in_add_ave, cnt_in_add_max, t;
  unsigned long int  i;
  int                t_sum, na_sum, nd_sum;
  int                cnt_in_add_ave_ave, cnt_in_add_max_ave;

  gmp_randinit_default (r_state);
  mpz_init(k);


  cnt_in_add_max_ave = cnt_in_add_ave_ave = t_sum =  na_sum = nd_sum = 0;
  for(i=0; i<n_samples; i++)
  {
    //mpz_rrandomb generates long strings of zeros or ones, might be
    better for testing
    mpz_urandomb (k, r_state, n_bits); //200 bits  random number
    to_NAF2(k);
    t = kP_time(k, n_bits, ADD_DBL_ratio, &na, &nd, &cnt_in_add_ave,
        &cnt_in_add_max, buf_len );
    t_sum  += t;
    na_sum += na;
    nd_sum += nd;
    cnt_in_add_ave_ave += cnt_in_add_ave;
    cnt_in_add_max_ave += cnt_in_add_max;
    //printf("%i-", cnt_in_add_max_ave);
    // if((i& 0x0000FFFF) == 0) printf(" %lu", i);
    //printf("\n-- k=%s   nADD =%d   nDBL =%d    T =%d", itos(k), na, nd,
           t);
    // gmp_printf("\n-- k=%#04Zx   nADD =%d   nDBL =%d    T =%d", k, na,
                nd, t);
  }

  printf("\nn_bits=%i, n_samples=%lu ADD/DBL=%i t_ave=%i  n Add_ave=%i, n
        DBL_ave=%i", n_bits, nsamples,  ADD_DBL_ratio, t_sum/nsamples,
        na_sum/nsamples, nd_sum/nsamples);

```

```
  //- printf for performance_table
  printf( "\n%i   &    %i & %i  &  %i &    %i  &    %i & %3.1f  &  %i  &
          %i \\hline", n_bits,  ADD_DBL_ratio, ave(na_sum), ave(nd_sum),
          ave(t_sum), n_bits+ n_bits*ADD_DBL_ratio/3, ((double)n_bits +
          n_bits/3.*ADD_DBL_ratio)/(t_sum/n_samples), ave((double)
          cnt_in_add_ave_ave/ADD_DBL_ratio), ave((double)
          cnt_in_add_max_ave/ADD_DBL_ratio));

  //- printf for performance_vs_buflen_graph
  printf( "%3.1f ",((double)n_bits + n_bits/3.*ADD_DBL_ratio)/(
          t_sum/n_samples));

  mpz_clear (k);
  return 1;
}

int kP_time(scalar_t k, int n_bits, int t_add, int *na, int *nd,  int
            *cnt_in_add_ave, int *cnt_in_add_max, int buf_len)
{
  int i, b, in_add, n_add, n_dbl, cnt_in_add, dbl_wait;
  long long int ciaa; //count in add average!
  int max_cnt_in_add = buf_len*t_add;

  dbl_wait = n_add =  n_dbl =0;
  in_add =  get_bit(k, 0)==1; //put initial conditions
  cnt_in_add = in_add ? t_add : 0;

  //cannot find n_bit by mpz functions  because it omits leading zeros
  //and so decreases n_dbl!
  //n_bits  = mpz_sizeinbase(k, 2);


  *cnt_in_add_ave = *cnt_in_add_max = ciaa =0;
  for(i=0; i<n_bits; i++)
  {

    while (cnt_in_add > max_cnt_in_add)
    {
        cnt_in_add --;
        dbl_wait   ++;
    }

    if(cnt_in_add > *cnt_in_add_max) //this gives the maximum buffer size
        *cnt_in_add_max = cnt_in_add;
```

```

    ciaa += cnt_in_add; //average of cnt_in_add essentially it is proportional to the number of data in the circular buffer
    b = get_bit(k, i);

    //if in addition state
    if(in_add)
    {
      cnt_in_add--;
      if(b==1)
      {
        n_add ++;
        cnt_in_add  += t_add; //accumulate the time that you need to stay in add mode
      }
      else //b is 0
      {
        if(cnt_in_add ==0 ) //if u have been enuf in add state and there is no more one
        {
            in_add = 0; //change state
            n_dbl ++;
        }
      }
    }
    else //in dbl state
    {
      if(b==0)
        n_dbl ++;
      else //b is 1
      {
        in_add = 1; //change state
        n_add ++;
        cnt_in_add  = t_add;
      }
    }
  }

  //should it be added to n_add? I think it should but the result is wrond.
  FATAL chk bjn
  //  n_add += (cnt_in_add/t_add) +((cnt_in_add%t_add)!=0 ? 1 :0) ; //ceil(cnt_in_add/t_add)
  *na = n_add;
  *nd = n_dbl + dbl_wait;
```

```
  *cnt_in_add_ave = ciaa / n_bits;

  return *na * t_add + *nd;
}

int get_bit(scalar_t k, int i)
{
  return mpz_tstbit (k, i);
}

void set_bit(scalar_t k, int i)
{
   mpz_setbit (k, i);
}

void clr_bit(scalar_t k, int i)
{
   mpz_clrbit (k, i);
}

int kP_time_s(char * ks, int t_add, int *na, int *nd)
{
  mpz_t k;
  int rc;

  mpz_init(k);
  mpz_set_str(k, ks, 2);
  //rc = kP_time(k, strlen(ks), t_add, na, nd);
  mpz_clear (k);

  return rc;

}

int kP_time_s2(char * ks, int ADD_DBL_ratio)
{
  int na, nd, t;


  t = kP_time_s(ks, ADD_DBL_ratio, &na, &nd);
  printf("\ns=%s, n_bits=%i, t=%i  na=%i, nd=%i", ks, strlen(ks), t, na,
         nd);

  return 0;
```

```
}

char *str_reverse(char *d, char *s)
{
  int len, i;

  len = strlen(s);
  d[len] = 0;
  for(i=0; i<len; i++)
    d[len-i-1] = s[i];

  return d;
}

/*this pice of software is from
~/ansari4/Tutorials/Cryptography/C_Libraries/ECC/elliptic/ec_curve.c
 an elliptic curve library writen by Paulo S.L.M. Barreto <pbarreto@uninet.
 com.b r>  http://planeta.terra.com.br/informatica/paulobarreto/
 it shows a parrallel way of converting and integer to NAF
 */
void to_NAF2(scalar_t k)
{
  mpz_t h;
  int nb;

  mpz_init(h);
  mpz_mul_ui (h, k, 3);
  mpz_xor(k, h, k); //we treat -1 and 1 the same! because we only want to
          count
  mpz_div_2exp (k, k, 1);
  nb  = mpz_sizeinbase(k, 2);
  // if( nb > *n_bits)
  // *n_bits  = nb;


  mpz_clear(h);
}


void to_NAF(scalar_t x)
{
    int s, i, n_bits;
    mpz_t y;
    int xi, xi_1, ci;
```

```

    //See Coren, Computer Arithmetic book, Page 146, Table 6.4 for the
    Algorithm
    int state_table[] = { 0, 2, 2, 1, 0, 3, 3, 1};

    mpz_init(y);
    n_bits  = mpz_sizeinbase(x, 2);
    ci = 0;
    //- it checks one extra bit, but that extra bit is zero and I need it
    to make NAF
    for(i=0; i<=n_bits; i++)
    {
        xi   = get_bit(x, i  );
        xi_1 = get_bit(x, i+1);
        s = state_table[(xi_1<<2) | (xi << 1) | ci];
        if(s & 2)
            set_bit(y, i);
        else
            clr_bit(y, i);
        ci = s&1;
    }

    mpz_set(x, y);
    mpz_clear(y);
}

void test_recording(void)
{
  mpz_t k;
  char *s;

  s =  "1010101111010111111010111111000111010010100101010111111100110101";
  //s =  "111101";
  mpz_init(k);

  mpz_set_str(k, s, 2);
  to_NAF(k);
  printf(" \ns=%s \nk=%s", s, itos(k));

  mpz_set_str(k, s, 2);
  to_NAF2(k);
  printf(" \ns=%s \nk=%s", s, itos(k));
  printf(" \n");
  mpz_clear(k);
```

```
}

#define MAX_N_BITS 1024
char *itos( scalar_t k)
{
  int i, nb;
  static char buf[ MAX_N_BITS+1];
  char *s = buf;

  nb =  mpz_sizeinbase(k, 2)-1;
  if(nb> MAX_N_BITS) nb =  MAX_N_BITS;
  for(i=nb; i>=0; i--)
      *s++ = get_bit(k, i)? '1' : '0';
  *s = 0;

  return buf;
}


```

```
#include <borzoi.h>
#include <fstream>
#include <unistd.h>
#include "nist_curves.h"

/*
(c) Bijan Ansari Tue Dec 16 14:59:49 EST 2003
all parts of Monti algorithm works Mon Dec 29 21:28:08 EST 2003

This program uses borZoi Elliptic Curve library to Implement Projective
coordinate version of Montgomery scalar multiplication.
This is done to check the result of the Elliptic Curve Processor

*/


// the register file, and an indexed way to access it!
F2M X1;
F2M X2;
F2M Z1;
F2M Z2;
F2M R4;
F2M b;
F2M x;
```

```
F2M y;

F2M *R = &X1;

BigInt k;   //the scalar
EC_Domain_Parameters dp = NIST_B_233;
//int m, k1, k2;          //f(x) = x^m + x^k1 + x^k2 + 1
//const int m = 15, k1 = 4;
//const int m = 167, k1 = 6;
//const int m = 233, k1 = 74;
//longinteger k           //the scalar

typedef unsigned char byte;
int  scalar_mult(void);
void projective_montgomery_scalar_multiplication1(void);
void projective_montgomery_scalar_multiplication2(void);
void original_montgomery_scalar_multiplication(void);
void affine_to_projective(void);
void Montgomery_P_plus_Q__P_plus_P1(void);
void Montgomery_P_plus_Q__P_plus_P2(void);
void Itoh_Tsuji_inverse(int m, int in, int out);
void calc_xy1(void);
void calc_xy2(void);
void Mdouble(int src);
void Madd(int dest);
int  scalar_mult(void);
void swap (void);
void print(char *s, F2M x, F2M y);
void init_regfile(void);
void dump_regfile(int n);
#define dump(A){ std::cout << "\n" << #A << "= " << A; }
//void dump(F2M A);
void use_Trionomial(int m, int k1);
inline F2M operator^ (const F2M& a, int n);


/*-
   scalar_mult() tested at Tue Feb 17 19:58:53 EST 2004 again
   it produces correct result using all 4 scalar multiplication
   functions
*/
```

```
int main(void)
{

    use_Trionomial(167, 6);
    //scalar_mult();

    init_regfile();
    //WARNING msb of k MUST be one, otherwise the result is not the same
    as the
    //asm program in the ECP. because ECP assumes MSB of k is one.

    //for GF(2^233)
    k = 1;
    k <<= 232;

   //for GF(2^15)
    k = 1;//in the asm program R4 is k!
    k <<= 14;


    //for GF(2^167)
    k = 1;
    k <<= 166;


    k |= hexto_BigInt("D7");

    projective_montgomery_scalar_multiplication2();

    std::cout << "\n--" ;
}

void print(char *s, F2M x, F2M y)
{
    std::cout << "\n--" << s ;
    std::cout <<"\nx=" << x << "\ny=" << y;
}

int scalar_mult(void)
{
    /**Warning**
    original_montgomery_scalar_multiplication(), curve.mul(k, dp.G) use
                                                the
    global "dp" variable and the finite field which is defined there while
```

```
    projective_montgomery_scalar_multiplication1() and
    projective_montgomery_scalar_multiplication2() use the finite filed
    which is defined at the start of the main() program ie use_Trionomial(
                                                   233, 74)
    */
    //k = hexto_BigInt("A9993E364706816ABA3E25717850C26C9CD0D89D");
     k = hexto_BigInt("D7"); //in the asm program R4 is k!

     use_Trionomial(15, 4);
     b = dp.b;
     x = dp.G.x;
     y = dp.G.y;
     R[3] = 1; //R[3] must be zero otherwise affine_to_projective()
            doesn't work fine

     //in the hardware R4 is k, but here k is in another variable
     init_regfile();
     print("original points", x, y);

     projective_montgomery_scalar_multiplication2();
     print("projective_montgomery_scalar_multiplication2()", X2, Z2);

     projective_montgomery_scalar_multiplication1();
     print("projective_montgomery_scalar_multiplication1()", X2, Z2);


     original_montgomery_scalar_multiplication();
     print("original_scalar_multiplication()", X2, Z2);

     Curve curve (dp.a, dp.b);
     Point P = curve.mul(k, dp.G);
     print("Borzoi library", P.x, P.y);

     std::cout << "\n--" ;
}

void init_regfile()
{
    //values are interpreted as hex

    str_to_F2M("39",R[0]);
    str_to_F2M("24",R[1]);
    str_to_F2M("55",R[2]);
    str_to_F2M("76",R[3]);
```

```
    str_to_F2M("D7",R[4]); //MSB of k MUST be one, ECP asm programs
                assumes so!
    str_to_F2M("86",R[5]);   //R[4] is k and MSB of k must be one, that's
                why it is 16 bits and the others are 8 bits just to make
                things simple
    str_to_F2M("64",R[6]);
    str_to_F2M("A7",R[7]);

/*
    str_to_F2M("1",R[0]);
    str_to_F2M("2",R[1]);
    str_to_F2M("3",R[2]);
    str_to_F2M("4",R[3]);
    str_to_F2M("5",R[4]);
    str_to_F2M("6",R[5]);
    str_to_F2M("7",R[6]);
    str_to_F2M("8",R[7]);
*/
}



void affine_to_projective(void)
{
    X1 = x;
    Z1 = R[3]; //R[3]; in the ECP assembly file here we have R[3]

    Z2 = x^2;
    X2 = (Z2^2) + b;

}
void projective_montgomery_scalar_multiplication1()
{
    int l;
    int i;

    l = k.numBits ();
    affine_to_projective();

    for(i=l-2; i>=0; i--)
    {
        std::cout << "\n==1==\nbit " << i << "= "<< k.getBit(i) ;
        if(k.getBit(i) == 1)
        {
```

```
            Madd(1); Mdouble(2);
        }
        else
        {
            Madd(2); Mdouble(1);
        }
        dump_regfile(i);
        std::cout << "\n==1==";
    }
    //calc_xy1(); //answer is in X2, Z2
}

//this is the implemented algorithm
void projective_montgomery_scalar_multiplication2()
{
    int l;
    int i;

    dump_regfile(0);
    l = k.numBits ();
    affine_to_projective();
    dump_regfile(1);

    std::cout << "\nnum bits= " << l ;
    std::cout << "\nk= " << k ;
    for(i=l-2; i>=0; i--)
    {
        std::cout << "\n==2==\nbit " << i << "= "<< k.getBit(i) ;
        if(k.getBit(i) == 1)
            swap();

        Montgomery_P_plus_Q__P_plus_P2();

        if(k.getBit(i) == 1)
            swap();
        dump_regfile(i);
        std::cout << "\n==2==";
    }
    calc_xy2(); //answer is in X2, Z2     */
    dump_regfile(-1);
}

void original_montgomery_scalar_multiplication(void)
{
```

```
    Curve curve (dp.a, dp.b);
    Point P1, P2;
    int i, l;
    Point P(x, y);

    l = k.numBits ();
    P1 = P;
    P2 = curve.dbl(P);
    for(i=l-2; i>=0; i--)
    {
        if(k.getBit(i) ==1)
        {
            P1 = curve.add(P1, P2);
            P2 = curve.dbl(P2);
        }
        else
        {
            P2 = curve.add(P1, P2);
            P1 = curve.dbl(P1);
        }
    }

    X2 = P1.x;
    Z2 = P1.y;
}

void Mdouble(int src)
{
    F2M X, Z;

    if(src==1)
    {
        X = X1;
        Z = Z1;
    }
    else
    {
        X = X2;
        Z = Z2;
    }

    F2M x3 = (X^4) + b * (Z^4);
    F2M z3 = (Z^2) * (X^2);

```

```
    if(src==1)
    {
        X1 = x3;
        Z1 = z3;
    }
    else
    {
        X2 = x3;
        Z2 = z3;
    }

}

void Madd(int dest)
{
    F2M z3 = (X1 * Z2 + X2* Z1)^2;
    F2M x3 = (x * z3) + (X1 * Z2) * (X2 * Z1);

    if(dest==1)
    {
        X1 = x3;
        Z1 = z3;
    }
    else
    {
        X2 = x3;
        Z2 = z3;
    }
}

void swap (void)
{
    F2M T;

    T = X1; X1 = X2; X2 = T;
    T = Z1; Z1 = Z2; Z2 = T;
}\
void Montgomery_P_plus_Q__P_plus_P1(void)
{
    /* equivalent to
        (X1, Z1) = Mdouble(X1, Z1)
        (X2, Z2) = Madd(X1, Z1, X2, Z2)
    */
   //this is implemented in monti4.s
```

```
    X2 = Z1 * X2; //1
    Z1 = Z1 ^ 2;

    Z2 = X1 * Z2; //2
    X1 = X1 ^ 2;

    R4 = Z1 ^ 2;

    Z1 = X1 * Z1; //3


    X1 = X1 ^ 2;

    F2M t = X2 + Z2;
    X2 = X2 * Z2; //4
    Z2 = t ^ 2;


    R4 = R4 * b;  //5
    X1 = X1 + R4;


    R4 = x * Z2; //6
    X2 = X2 + R4;
}


void Montgomery_P_plus_Q__P_plus_P2(void)
{
   //this is implemented in monti4.s
    R[1] = R[2] * R[1]; dump(R[1]);//1
    R[2] = R[2] ^ 2;

    R[3] = R[0] * R[3]; dump(R[3]);//2
    R[0] = R[0] ^ 2;

    R[4] = R[2] ^ 2;

    R[2] = R[0] * R[2]; dump(R[2]);//3

    R[0] = R[0] ^ 2;

    F2M t = R[1] + R[3];
    R[1] = R[1] * R[3]; dump(R[1]);//4
```

```
    R[3] = t ^ 2;


    R[4] = R[4] * R[5]; dump(R[4]);//5  R5 = b
    R[0] = R[0] + R[4];


    R[4] = R[6] * R[3]; dump(R[4]);//6 R6 = x
    R[1] = R[1] + R[4];

    std::cout<<"\n-----";
    dump(R[3]);
    dump(R[2]);
    dump(R[1]);
    dump(R[0]);
}

//this routine is written in a way to be the same as the
//hardware implementation, and it doesn't mean it is a good
//software implementation
void Itoh_Tsuji_inverse(int m, int in, int out)
{
  //A is the accumulator, S is the input register of the multiplier
  //this is implemented in inv.rom
  F2M A,  S;
  byte m0, e, sq_cnt, i, c;

  A = R[in];
  e = 1;
  //m0 = dp.m & (~1);
  m0 = m & (!1);
  i = 8;

  while( (i!=0) && ((m0 & 0x80) == 0))
  {
      m0 <<= 1;
      i--;
  }


  if(i!=0) //skip the first '1' too
  {
      m0 <<=1;
      i--;
```

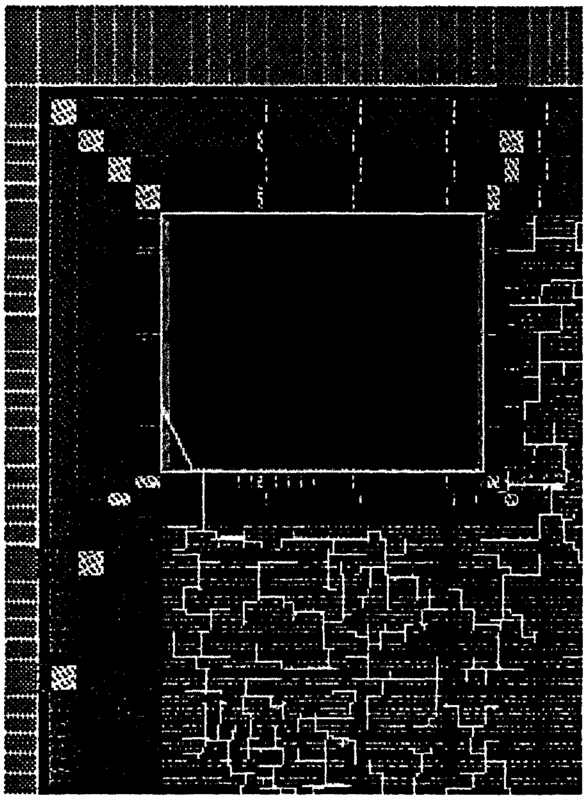
```
    }

    while(i!=0)
    {
        S = A;
        sq_cnt = e;
        while(sq_cnt--)
            A = A^2;

        A = S * A;

        c = (m0 & 0x80) != 0;
        m0 <<= 1;
        e = (e<<1) | c;

        if(c)
        {
            S = R[in];
            A = A^2;
            A = S * A;

        }
        i--;
    }

    A = A^2;
    R[out] = A;


    //borzoi is stupid!, sometimes doesn't reduce the result of
    multiplication!!
}


void calc_xy1()
{
// find this from Lopez paper and orlando paper (all are in the white
                                                              folder)
    F2M xk, yk;

//F2M F2M::inverse ()
//F2M F2M::sqr ()

    R4 = x*Z1*Z2;
```

```


    R4 = R4.inverse();
    /*
    R4 = T3;
    Itoh_Tsuji_inverse(dp.m, 4, 4); use this one becaue dp is not always
                        correctly set!
    T3 = R4;
    */

    F2M T = x * Z2 * X1;
    dump(T);

    xk = x * Z2 * X1 * R4;
    yk = (xk + x) * R4 * (Z1*Z2 * (y+ (x^2)) + (X2 + x*Z2) * (X1 + x*Z1)) +
         y;

    X2 = xk;
    Z2 = yk;

}


//--------------------------------------------------------------------------
---
void dump_regfile(int n)
{
    std::cout << "\n-"<<n<<" Reg file -";
    for(int i=7; i>=0; i--)
        std::cout << "\nR" << i << "= " << R[i];
}

void use_Trionomial(int m, int k)
{
    F2X pt=Trinomial (m, k, 0);
    setModulus (pt);
}

inline F2M operator^ (const F2M& a, int n)
{
    F2M c=a;

    while(--n>0)  //>0 for n equal 0
```

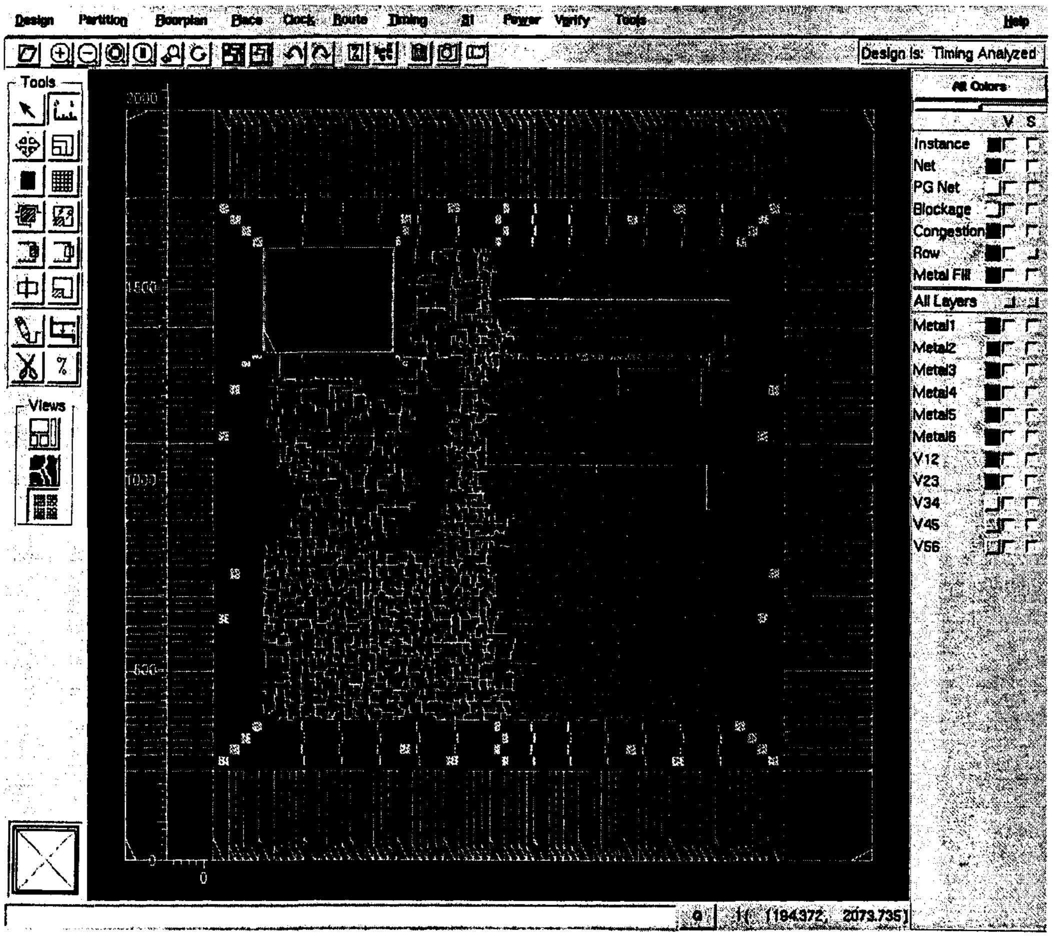
```
        c*=a;
    return c;
}
```

Appendix B

Chip Layout



Chip layout: program memory, power rings, power strips, clock tree



Chip layout: The whole chip in SoCE

VITA AUCTORIS

Name:	Bijan Ansari
Year of Birth:	1964
Education:	
1979 - 1982	High school diploma, Isfahan University High School, Isfahan, Iran
1983 - 1988	B.Sc. Isfahan University of Technology, Isfahan, Iran
2002 - 2004	M.A.Sc. University of Windsor, Windsor, Ontario
Email:	bijan486@yahoo.com